
Matplotlib

Release 2.0.0

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

January 16, 2017

CONTENTS

I User’s Guide	1
1 Introduction	3
2 Installing	5
3 Tutorials	11
4 Working with text	97
5 Colors	145
6 Customizing matplotlib	173
7 Interactive plots	189
8 Selected Examples	205
9 What’s new in matplotlib	239
10 GitHub Stats	339
11 License	399
12 Credits	403
II The Matplotlib API	405
13 Plotting commands summary	407
14 API Changes	415
15 The top level <code>matplotlib</code> module	465
16 afm (Adobe Font Metrics interface)	469
17 animation module	473

18	artist Module	517
19	Axes class	537
20	axis and tick API	773
21	backends	893
22	cbook	933
23	cm (colormap)	951
24	collections	955
25	colorbar	1125
26	colors	1131
27	dates	1145
28	dviread	1159
29	figure	1163
30	finance	1185
31	font_manager	1197
32	gridspec	1205
33	image	1209
34	legend and legend_handler	1215
35	lines	1223
36	markers	1233
37	mathtext	1237
38	mlab	1257
39	offsetbox	1291
40	patches	1303
41	path	1343
42	patheffects	1351
43	projections	1355
44	pyplot	1363

45	resetup	1565
46	sankey	1569
47	scale	1577
48	spines	1587
49	style	1591
50	text	1593
51	ticker	1607
52	tight_layout	1619
53	Working with transformations	1621
54	triangular grids	1643
55	type1font	1655
56	units	1657
57	widgets	1659
III The Matplotlib FAQ		1675
58	Installation	1677
59	Usage	1685
60	How-To	1697
61	Troubleshooting	1713
62	Environment Variables	1717
63	Working with Matplotlib in Virtual environments	1719
64	Working with Matplotlib on OSX	1721
IV Matplotlib AxesGrid Toolkit		1725
65	Overview of AxesGrid toolkit	1729
66	The Matplotlib AxesGrid Toolkit User's Guide	1751
67	The Matplotlib AxesGrid Toolkit API	1767

68	The Matplotlib axes_grid1 Toolkit API	1777
V	mplot3d	1795
69	Matplotlib mplot3d toolkit	1797
VI	Toolkits	1851
70	Mapping Toolkits	1855
71	General Toolkits	1857
72	High-Level Plotting	1861
VII	External Resources	1865
73	Books, Chapters and Articles	1867
74	Videos	1869
75	Tutorials	1871
VIII	The Matplotlib Developers' Guide	1873
76	Contributing	1875
77	Developer's tips for testing	1883
78	Developer's tips for documenting matplotlib	1889
79	Developer's guide for creating scales and transformations	1901
80	Developer's tips for writing code for Python 2 and 3	1905
81	Working with <i>matplotlib</i> source code	1909
82	Reviewers guideline	1927
83	Release Guide	1929
84	Matplotlib Enhancement Proposals	1935
85	Licenses	1987
86	Default Color changes	1989

IX Matplotlib Examples	1993
87 animation Examples	1995
88 api Examples	1999
89 axes_grid Examples	2093
90 color Examples	2145
91 event_handling Examples	2157
92 frontpage Examples	2191
93 images_contours_and_fields Examples	2195
94 lines_bars_and_markers Examples	2211
95 misc Examples	2225
96 mplot3d Examples	2249
97 pie_and_polar_charts Examples	2297
98 pylab_examples Examples	2303
99 pyplots Examples	2747
100scales Examples	2769
101shapes_and_collections Examples	2771
102showcase Examples	2777
103specialty_plots Examples	2795
104statistics Examples	2803
105style_sheets Examples	2835
106subplots_axes_and_figures Examples	2849
107tests Examples	2853
108text_labels_and_annotations Examples	2865
109ticks_and_spines Examples	2871
110units Examples	2885
111user_interfaces Examples	2909
112widgets Examples	2975

X	Glossary	2989
	Bibliography	2993
	Python Module Index	2995

Part I

User's Guide

**CHAPTER
ONE**

INTRODUCTION

Matplotlib is a library for making 2D plots of arrays in [Python](#). Although it has its origins in emulating the MATLAB®¹ graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although Matplotlib is written primarily in pure Python, it makes heavy use of [NumPy](#) and other extension code to provide good performance even for large arrays.

Matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so on; it should just work.

For years, I used to use MATLAB exclusively for data analysis and visualization. MATLAB excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed an EEG analysis application in MATLAB. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB as a programming language, and decided to start over in Python. Python more than makes up for all of MATLAB's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D [VTK](#) more than exceeds all of my needs).

When I went searching for a Python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

Finding no package that suited me just right, I did what any self-respecting Python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate MATLAB's plotting capabilities because that is something MATLAB does very well. This had the added advantage that many people have a lot of MATLAB experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the `pylab` interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

The Matplotlib code is conceptually divided into three parts: the *pylab interface* is the set of functions provided by `matplotlib.pyplot` which allow the user to create plots with code quite similar to MATLAB

¹ MATLAB is a registered trademark of The MathWorks, Inc.

figure generating code ([Pyplot tutorial](#)). The *Matplotlib frontend* or *Matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on ([Artist tutorial](#)). This is an abstract interface that knows nothing about output. The *backends* are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device ([What is a backend?](#)). Example backends: PS creates PostScript® hardcopy, SVG creates Scalable Vector Graphics hardcopy, Agg creates PNG output using the high quality [Anti-Grain Geometry](#) library that ships with Matplotlib, GTK embeds Matplotlib in a [Gtk+](#) application, GTKAgg uses the Anti-Grain renderer to create a figure and embed it in a Gtk+ application, and so on for [PDF](#), [WxWidgets](#), [Tkinter](#), etc.

Matplotlib is used by many people in many different contexts. Some people want to automatically generate PostScript files to send to a printer or publishers. Others deploy Matplotlib on a web application server to generate PNG output for inclusion in dynamically-generated web pages. Some use Matplotlib interactively from the Python shell in Tkinter on Windows™. My primary use is to embed Matplotlib in a Gtk+ EEG application that runs on Windows, Linux and Macintosh OS X.

INSTALLING

There are many different ways to install matplotlib, and the best way depends on what operating system you are using, what you already have installed, and how you want to use it. To avoid wading through all the details (and potential complications) on this page, there are several convenient options.

2.1 Installing pre-built packages

2.1.1 Most platforms : scientific Python distributions

The first option is to use one of the pre-packaged python distributions that already provide matplotlib built-in. The Continuum.io Python distribution ([Anaconda](#) or [miniconda](#)) and the Enthought distribution ([Canopy](#)) are both excellent choices that “just work” out of the box for Windows, OSX and common Linux platforms. Both of these distributions include matplotlib and *lots* of other useful tools.

2.1.2 Linux : using your package manager

If you are on Linux, you might prefer to use your package manager. matplotlib is packaged for almost every major Linux distribution.

- Debian / Ubuntu : `sudo apt-get install python-matplotlib`
- Fedora / Redhat : `sudo yum install python-matplotlib`

2.1.3 Mac OSX : using pip

If you are on Mac OSX you can probably install matplotlib binaries using the standard Python installation program `pip`. See [Installing OSX binary wheels](#).

2.1.4 Windows

If you don’t already have Python installed, we recommend using one of the [scipy-stack compatible Python distributions](#) such as WinPython, Python(x,y), Enthought Canopy, or Continuum Anaconda, which have matplotlib and many of its dependencies, plus other useful packages, preinstalled.

For standard Python installations, install matplotlib using `pip`:

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

In case Python 2.7 or 3.4 are not installed for all users, the Microsoft Visual C++ 2008 ([64 bit](#) or [32 bit](#) for Python 2.7) or Microsoft Visual C++ 2010 ([64 bit](#) or [32 bit](#) for Python 3.4) redistributable packages need to be installed.

Matplotlib depends on [Pillow](#) for reading and saving JPEG, BMP, and TIFF image files. Matplotlib requires [MiKTeX](#) and [GhostScript](#) for rendering text with LaTeX. [FFmpeg](#), [avconv](#), [mencoder](#), or [ImageMagick](#) are required for the animation module.

The following backends should work out of the box: `agg`, `tkagg`, `ps`, `pdf` and `svg`. For other backends you may need to install [pycairo](#), [PyQt4](#), [PyQt5](#), [PySide](#), [wxPython](#), [PyGTK](#), [Tornado](#), or [GhostScript](#).

`TkAgg` is probably the best backend for interactive use from the standard Python shell or IPython. It is enabled as the default backend for the official binaries. `GTK3` is not supported on Windows.

The Windows wheels (`*.whl`) on the [PyPI download page](#) do not contain test data or example code. If you want to try the many demos that come in the matplotlib source distribution, download the `*.tar.gz` file and look in the `examples` subdirectory. To run the test suite, copy the `lib\matplotlib\tests` and `lib\mpl_toolkits\tests` directories from the source distribution to `sys.prefix\Lib\site-packages\matplotlib` and `sys.prefix\Lib\site-packages\mpl_toolkits` respectively, and install `nose`, `mock`, [Pillow](#), [MiKTeX](#), [GhostScript](#), [ffmpeg](#), [avconv](#), [mencoder](#), [ImageMagick](#), and [Inkscape](#).

2.2 Installing from source

If you are interested in contributing to matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build matplotlib from source. Grab the latest `tar.gz` release file from [the PyPI files page](#), or if you want to develop matplotlib or just need the latest bugfixed version, grab the latest git version [Source install from git](#).

The standard environment variables `CC`, `CXX`, `PKG_CONFIG` are respected. This means you can set them if your toolchain is prefixed. This may be used for cross compiling.

```
export CC=x86_64-pc-linux-gnu-gcc export CXX=x86_64-pc-linux-gnu-g++ export
PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```

Once you have satisfied the requirements detailed below (mainly `python`, `numpy`, `libpng` and `freetype`), you can build matplotlib:

```
cd matplotlib
python setup.py build
python setup.py install
```

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

If you have installed prerequisites to nonstandard places and need to inform matplotlib where they are, edit `setupext.py` and add the base dirs to the `basedir` dictionary entry for your `sys.platform`. e.g., if

the header to some required library is in `/some/path/include/someheader.h`, put `/some/path` in the `basedir` list for your platform.

2.2.1 Build requirements

These are external packages which you will need to install before installing matplotlib. If you are building on OSX, see [Building on OSX](#). If you are building on Windows, see [Building on Windows](#). If you are installing dependencies with a package manager on Linux, you may need to install the development packages (look for a “-dev” postfix) in addition to the libraries themselves.

Required Dependencies

python 2.7, 3.4, 3.5 or 3.6 Download [python](#).

numpy 1.7.1 (or later) array support for python ([download numpy](#))

setuptools Setuptools provides extensions for python package installation.

dateutil 1.1 or later Provides extensions to python datetime handling. If using pip, easy_install or installing from source, the installer will attempt to download and install `python_dateutil` from PyPI.

pyparsing Required for matplotlib’s mathtext math rendering support. If using pip, easy_install or installing from source, the installer will attempt to download and install `pyparsing` from PyPI.

libpng 1.2 (or later) library for loading and saving *PNG* files ([download](#)). libpng requires zlib.

pytz Used to manipulate time-zone aware datetimes. <https://pypi.python.org/pypi/pytz>

FreeType 2.3 or later Library for reading true type font files. If using pip, easy_install or installing from source, the installer will attempt to locate FreeType in expected locations. If it cannot, try installing `pkg-config`, a tool used to find required non-python libraries.

cycler 0.10.0 or later Composable cycle class used for constructing style-cycles

six Required for compatibility between python 2 and python 3

Dependencies for python 2

functools32 Required for compatibility if running on Python 2.7.

subprocess32 Optional, unix only. Backport of the subprocess standard library from 3.2+ for Python 2.7. It provides better error messages and timeout support.

Optional GUI framework

These are optional packages which you may want to install to use matplotlib with a user interface toolkit. See [What is a backend?](#) for more details on the optional matplotlib backends and the capabilities they provide.

`tk` 8.3 or later, not 8.6.0 or 8.6.1 The TCL/Tk widgets library used by the TkAgg backend.

Versions 8.6.0 and 8.6.1 are known to have issues that may result in segfaults when closing multiple windows in the wrong order.

`pyqt` 4.4 or later The Qt4 widgets library python wrappers for the Qt4Agg backend

`pygtk` 2.4 or later The python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend

`wxpython` 2.8 or later The python wrappers for the wx widgets library for use with the WX or WXAgg backend

Optional external programs

ffmpeg/avconv or mencoder Required for the animation module to be save out put to movie formats.

ImageMagick Required for the animation module to be able to save to animated gif.

Optional dependencies

Pillow If Pillow is installed, matplotlib can read and write a larger selection of image file formats.

pkg-config A tool used to find required non-python libraries. This is not strictly required, but can make installation go more smoothly if the libraries and headers are not in the expected locations.

Required libraries that ship with matplotlib

`agg` 2.4 The antigrain C++ rendering engine. matplotlib links against the agg template source statically, so it will not affect anything on your system outside of matplotlib.

`qhull` 2012.1 A library for computing Delaunay triangulations.

`ttcconv` truetype font utility

2.2.2 Building on Linux

It is easiest to use your system package manager to install the dependencies.

If you are on Debian/Ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get the install the build dependencies, which will make building from source easier.

2.2.3 Building on OSX

The build situation on OSX is complicated by the various places one can get the libpng and freetype requirements (darwinports, fink, /usr/X11R6) and the different architectures (e.g., x86, ppc, universal) and the different OSX version (e.g., 10.4 and 10.5). We recommend that you build the way we do for the OSX release: get the source from the tarball or the git repository and follow the instruction in `README.osx`.

2.2.4 Building on Windows

The Python shipped from <https://www.python.org> is compiled with Visual Studio 2008 for versions before 3.3, Visual Studio 2010 for 3.3 and 3.4, and Visual Studio 2015 for 3.5 and 3.6. Python extensions are recommended to be compiled with the same compiler.

Since there is no canonical Windows package manager, the methods for building freetype, zlib, and libpng from source code are documented as a build script at [matplotlib-winbuild](#).

**CHAPTER
THREE**

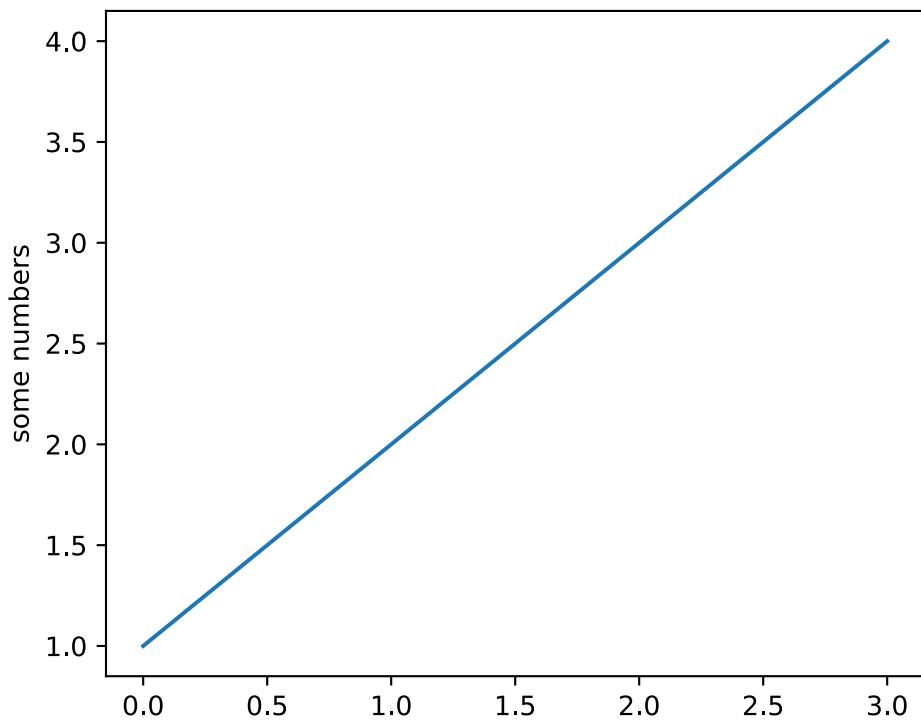
TUTORIALS

3.1 Introductory

3.1.1 Pyplot tutorial

`matplotlib.pyplot` is a collection of command style functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In `matplotlib.pyplot` various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that “axes” here and in most places in the documentation refers to the *axes part of a figure* and not the strict mathematical term for more than one axis).

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



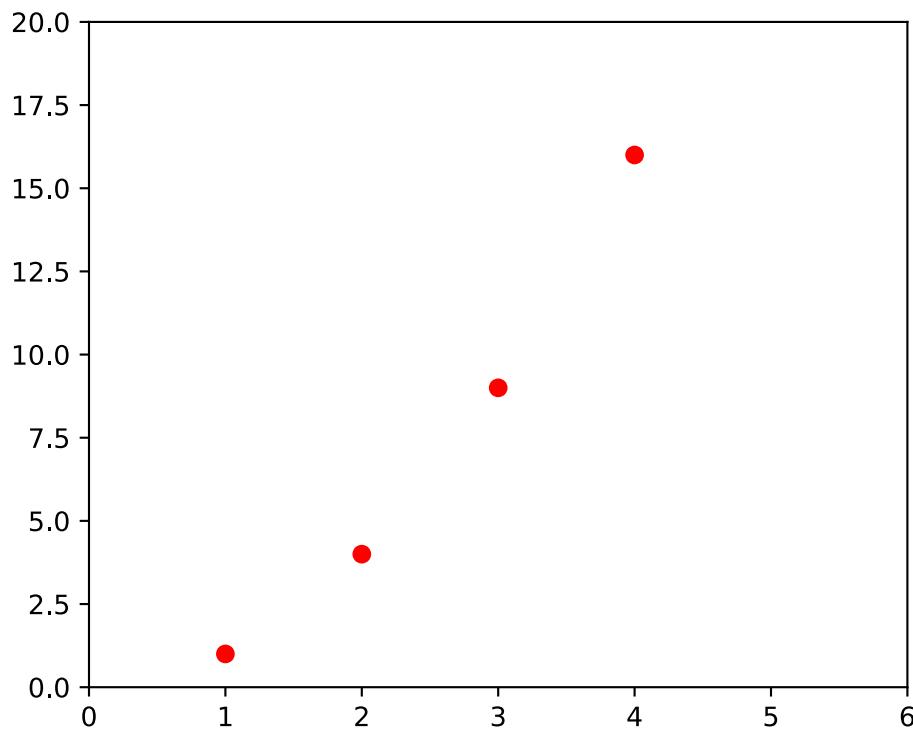
You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the `plot()` command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0, 1, 2, 3].

`plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



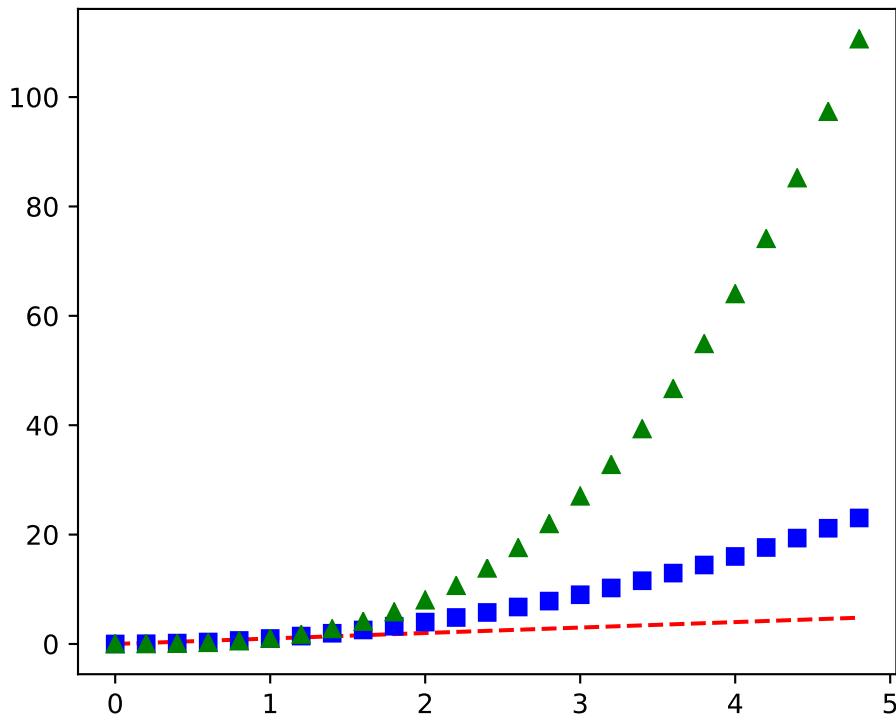
See the [plot\(\)](#) documentation for a complete list of line styles and format strings. The [axis\(\)](#) command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use [numpy](#) arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates a plotting several lines with different format styles in one command using arrays.

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

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see [matplotlib.lines.Line2D](#). There are several ways to set line properties

- Use keyword args:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a Line2D instance. `plot` returns a list of Line2D objects; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with `line`, to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialiasing
```

- Use the `setp()` command. The example below uses a MATLAB-style command to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

```
lines = plt.plot(x1, y1, x2, y2)
# use keyword args
plt.setp(lines, color='r', linewidth=2.0)
```

```
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Here are the available `Line2D` properties.

Property	Value Type
alpha	float
animated	[True False]
antialiased or aa	[True False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	['butt' 'round' 'projecting']
dash_joinstyle	['miter' 'round' 'bevel']
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	['-' '--' '-.' ':' 'steps' ...]
linewidth or lw	float value in points
lod	[True False]
marker	['+' ' ' '.' '1' '2' '3' '4']
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	[None integer (startind, stride)]
picker	used in interactive line selection
pickradius	the line pick selection radius
solid_capstyle	['butt' 'round' 'projecting']
solid_joinstyle	['miter' 'round' 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True False]
xdata	np.array
ydata	np.array
zorder	any number

To get a list of settable line properties, call the `setp()` function with a line or lines as argument

```
In [69]: lines = plt.plot([1, 2, 3])
```

```
In [70]: plt.setp(lines)
alpha: float
animated: [True | False]
antialiased or aa: [True | False]
```

```
...snip
```

Working with multiple figures and axes

MATLAB, and *pyplot*, have the concept of the current figure and the current axes. All plotting commands apply to the current axes. The function *gca()* returns the current axes (a *matplotlib.axes.Axes* instance), and *gcf()* returns the current figure (*matplotlib.figure.Figure* instance). Normally, you don't have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

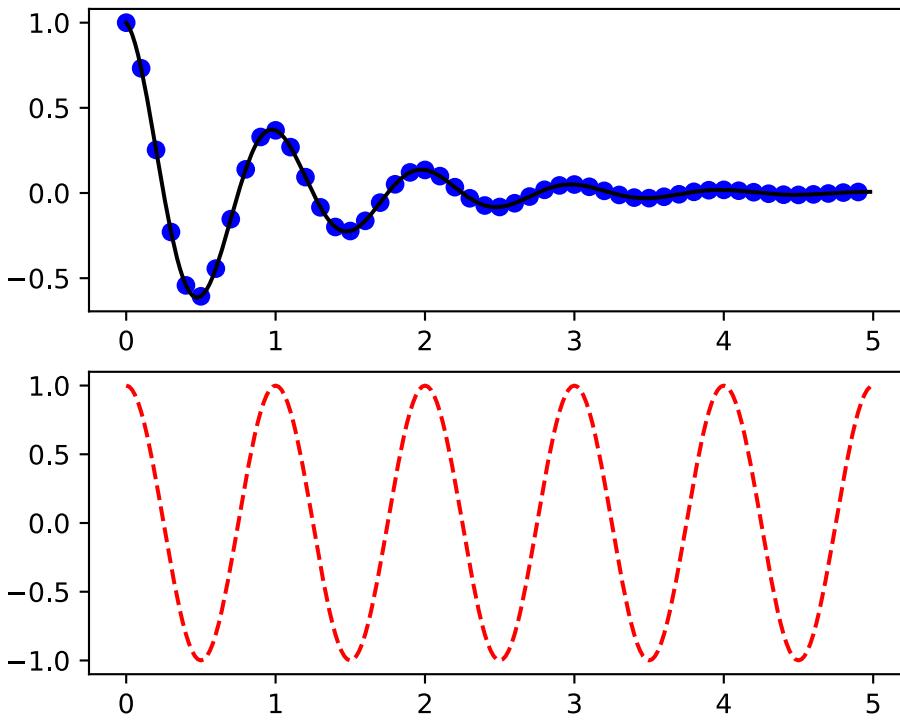
```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



The `figure()` command here is optional because `figure(1)` will be created by default, just as a `subplot(111)` will be created by default if you don't manually specify any axes. The `subplot()` command specifies `numrows`, `numcols`, `fignum` where `fignum` ranges from 1 to `numrows*numcols`. The commas in the `subplot` command are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`. You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use the `axes()` command, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See [pylab_examples example code: axes_demo.py](#) for an example of placing axes manually and [pylab_examples example code: subplots_demo.py](#) for an example with lots of subplots.

You can create multiple figures by using multiple `figure()` calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
import matplotlib.pyplot as plt
plt.figure(1)                  # the first figure
plt.subplot(211)                # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)                # the second subplot in the first figure
plt.plot([4, 5, 6])

plt.figure(2)                  # a second figure
plt.plot([4, 5, 6])             # creates a subplot(111) by default

plt.figure(1)                  # figure 1 current; subplot(212) still current
```

```
plt.subplot(211)          # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

You can clear the current figure with `clf()` and the current axes with `cla()`. If you find it annoying that states (specifically the current image, figure and axes) are being maintained for you behind the scenes, don't despair: this is just a thin stateful wrapper around an object oriented API, which you can use instead (see [Artist tutorial](#))

If you are making lots of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with `close()`. Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until `close()` is called.

Working with text

The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations (see [Text introduction](#) for a more detailed example)

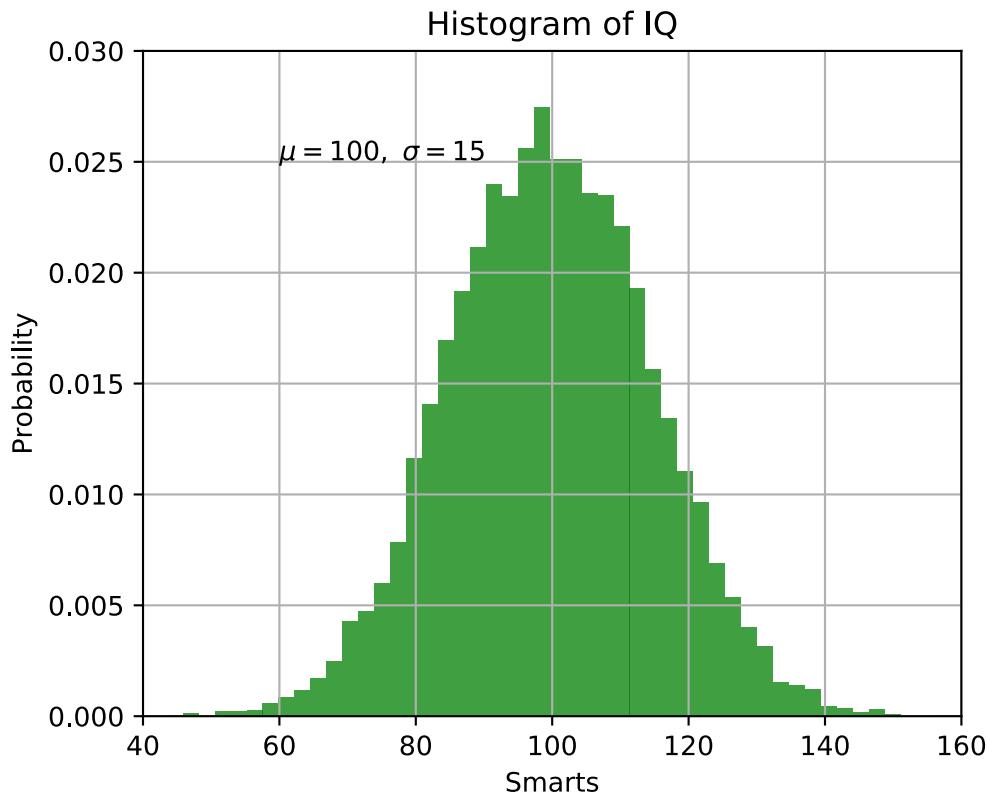
```
import numpy as np
import matplotlib.pyplot as plt

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

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100, \ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



All of the `text()` commands return an `matplotlib.text.Text` instance. Just as with with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in [Text properties and layout](#).

Using mathematical expressions in text

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

```
plt.title(r'$\sigma_i=15$')
```

The `r` preceding the title string is important – it signifies that the string is a *raw* string and not to treat backslashes as python escapes. matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts – for details see [Writing mathematical expressions](#). Thus you can use mathematical text across platforms without requiring a TeX installation. For those who have LaTeX and dvipng installed, you can also use LaTeX to format your text and incorporate the output directly into your display figures or saved postscript – see [Text rendering With LaTeX](#).

Annotating text

The uses of the basic `text()` command above place text at an arbitrary position on the Axes. A common use for text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x,y)` tuples.

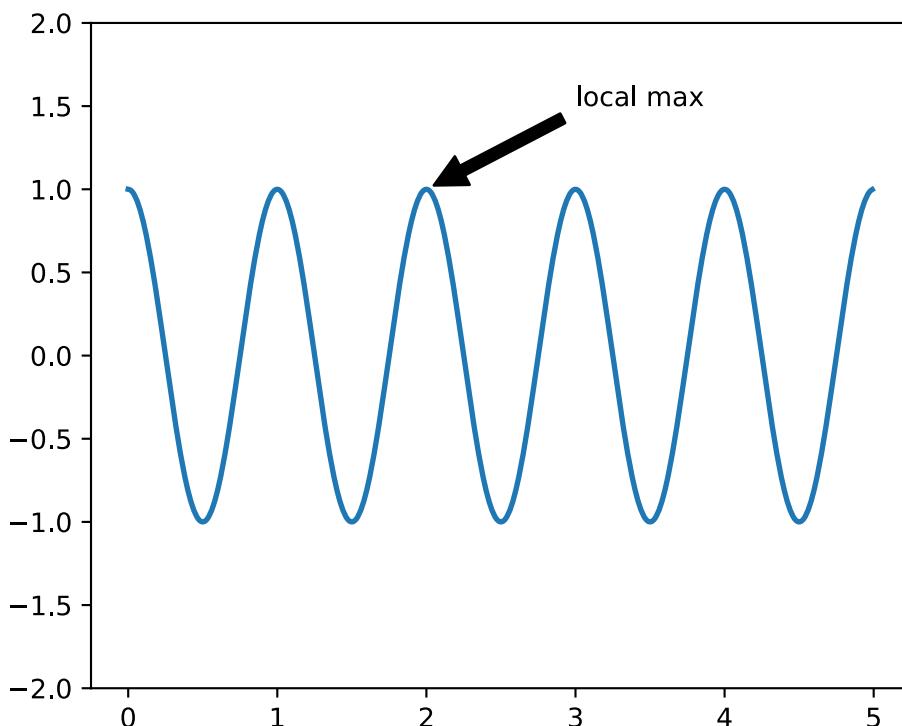
```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)

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

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

plt.ylim(-2,2)
plt.show()
```



In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates.

There are a variety of other coordinate systems one can choose – see [Basic annotation](#) and [Advanced Annotation](#) for details. More examples can be found in [pylab_examples example code: annotation_demo.py](#).

Logarithmic and other nonlinear axis

`matplotlib.pyplot` supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

```
plt.xscale('log')
```

An example of four plots with the same data and different scales for the y axis is shown below.

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

from matplotlib.ticker import NullFormatter # useful for `logit` scale

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

# make up some data in the interval ]0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure(1)

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

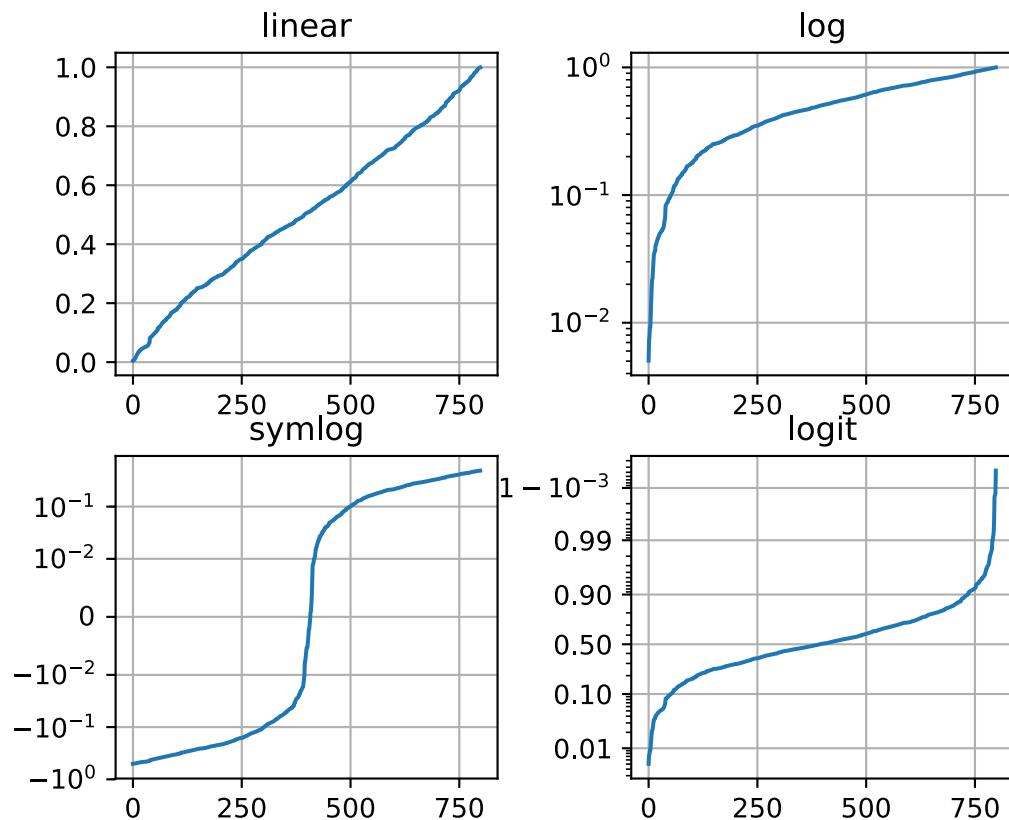
# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthreshy=0.01)
plt.title('symlog')
plt.grid(True)

# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
```

```
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
# Format the minor tick labels of the y-axis into empty strings with
# `NullFormatter`, to avoid cumbering the axis with too many labels.
plt.gca().yaxis.set_minor_formatter(NullFormatter())
# Adjust the subplot layout, because the logit one may take more space
# than usual, due to y-tick labels like "1 - 10^{-3}"
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25,
                    wspace=0.35)

plt.show()
```



It is also possible to add your own scale, see [Developer's guide for creating scales and transformations](#) for details.

3.1.2 Image tutorial

Startup commands

First, let's start IPython. It is a most excellent enhancement to the standard Python prompt, and it ties in especially well with Matplotlib. Start IPython either at a shell, or the IPython Notebook now.

With IPython started, we now need to connect to a GUI event loop. This tells IPython where (and how) to display plots. To connect to a GUI loop, execute the `%matplotlib` magic at your IPython prompt. There's more detail on exactly what this does at [IPython's documentation on GUI event loops](#).

If you're using IPython Notebook, the same commands are available, but people commonly use a specific argument to the `%matplotlib` magic:

```
In [1]: %matplotlib inline
```

This turns on inline plotting, where plot graphics will appear in your notebook. This has important implications for interactivity. For inline plotting, commands in cells below the cell that outputs a plot will not affect the plot. For example, changing the color map is not possible from cells below the cell that creates a plot. However, for other backends, such as qt4, that open a separate window, cells below those that create the plot will change the plot - it is a live object in memory.

This tutorial will use matplotlib's imperative-style plotting interface, pyplot. This interface maintains global state, and is very useful for quickly and easily experimenting with various plot settings. The alternative is the object-oriented interface, which is also very powerful, and generally more suitable for large application development. If you'd like to learn about the object-oriented interface, a great place to start is our [FAQ on usage](#). For now, let's get on with the imperative-style approach:

```
In [2]: import matplotlib.pyplot as plt  
In [3]: import matplotlib.image as mpimg  
In [4]: import numpy as np
```

Importing image data into Numpy arrays

Loading image data is supported by the `Pillow` library. Natively, matplotlib only supports PNG images. The commands shown below fall back on Pillow if the native read fails.

The image used in this example is a PNG file, but keep that Pillow requirement in mind for your own data.

Here's the image we're going to play with:



It's a 24-bit RGB PNG image (8 bits for each of R, G, B). Depending on where you get your data, the other kinds of image that you'll most likely encounter are RGBA images, which allow for transparency, or single-channel grayscale (luminosity) images. You can right click on it and choose "Save image as" to download it to your computer for the rest of this tutorial.

And here we go...

```
In [5]: img=mpimg.imread('stinkbug.png')
Out[5]:
array([[[ 0.40784314,  0.40784314,  0.40784314],
       [ 0.40784314,  0.40784314,  0.40784314],
       [ 0.40784314,  0.40784314,  0.40784314],
       ...,
       [ 0.42745098,  0.42745098,  0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098],
       [ 0.42745098,  0.42745098,  0.42745098]],

      ...,
      [[ 0.44313726,  0.44313726,  0.44313726],
       [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
       [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
       ...,
       [ 0.44705883,  0.44705883,  0.44705883],
```

```
[ 0.44705883,  0.44705883,  0.44705883],  
[ 0.44313726,  0.44313726,  0.44313726]]], dtype=float32)
```

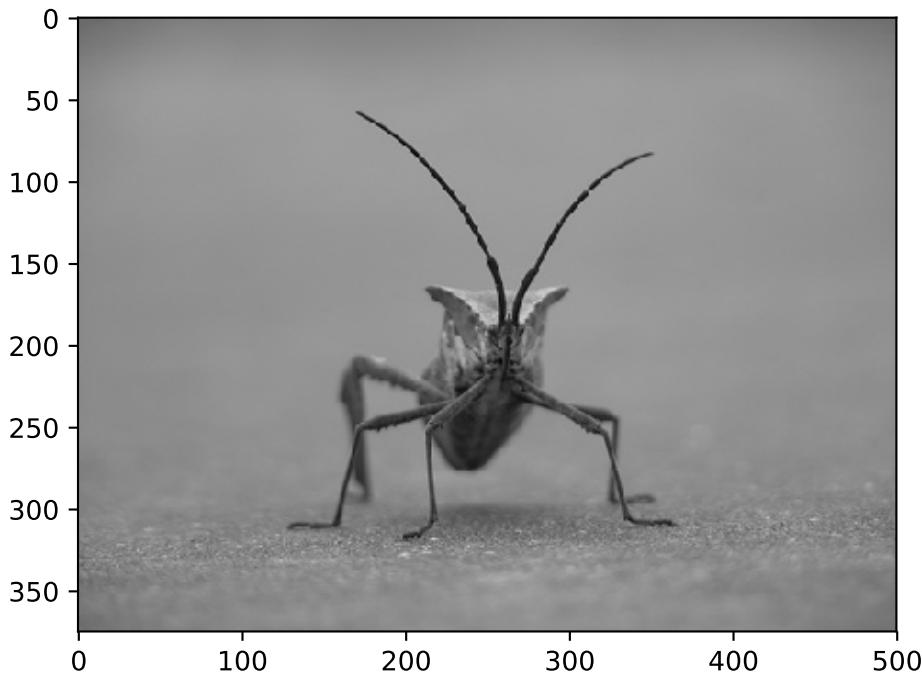
Note the `dtype` there - `float32`. Matplotlib has rescaled the 8 bit data from each channel to floating point data between 0.0 and 1.0. As a side note, the only datatype that Pillow can work with is `uint8`. Matplotlib plotting can handle `float32` and `uint8`, but image reading/writing for any format other than PNG is limited to `uint8` data. Why 8 bits? Most displays can only render 8 bits per channel worth of color gradation. Why can they only render 8 bits/channel? Because that's about all the human eye can see. More here (from a photography standpoint): [Luminous Landscape bit depth tutorial](#).

Each inner list represents a pixel. Here, with an RGB image, there are 3 values. Since it's a black and white image, R, G, and B are all similar. An RGBA (where A is alpha, or transparency), has 4 values per inner list, and a simple luminance image just has one value (and is thus only a 2-D array, not a 3-D array). For RGB and RGBA images, matplotlib supports `float32` and `uint8` data types. For grayscale, matplotlib supports only `float32`. If your array data does not meet one of these descriptions, you need to rescale it.

Plotting numpy arrays as images

So, you have your data in a numpy array (either by importing it, or by generating it). Let's render it. In Matplotlib, this is performed using the `imshow()` function. Here we'll grab the plot object. This object gives you an easy way to manipulate the plot from the prompt.

```
In [6]: imgplot = plt.imshow(img)
```



You can also plot any numpy array.

Applying pseudocolor schemes to image plots

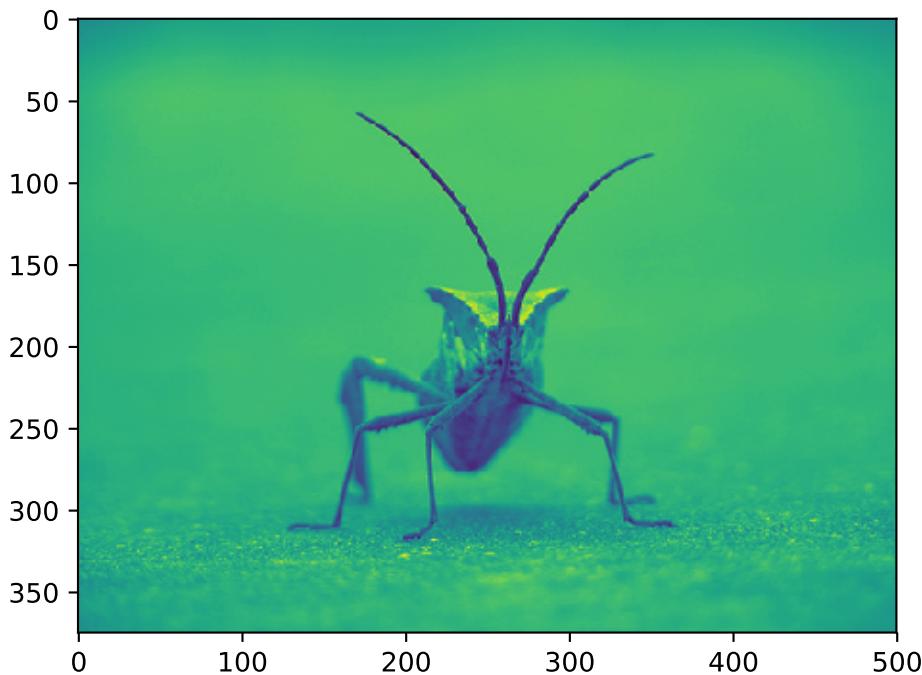
Pseudocolor can be a useful tool for enhancing contrast and visualizing your data more easily. This is especially useful when making presentations of your data using projectors - their contrast is typically quite poor.

Pseudocolor is only relevant to single-channel, grayscale, luminosity images. We currently have an RGB image. Since R, G, and B are all similar (see for yourself above or in your data), we can just pick one channel of our data:

```
In [7]: lum_img = img[:, :, 0]
```

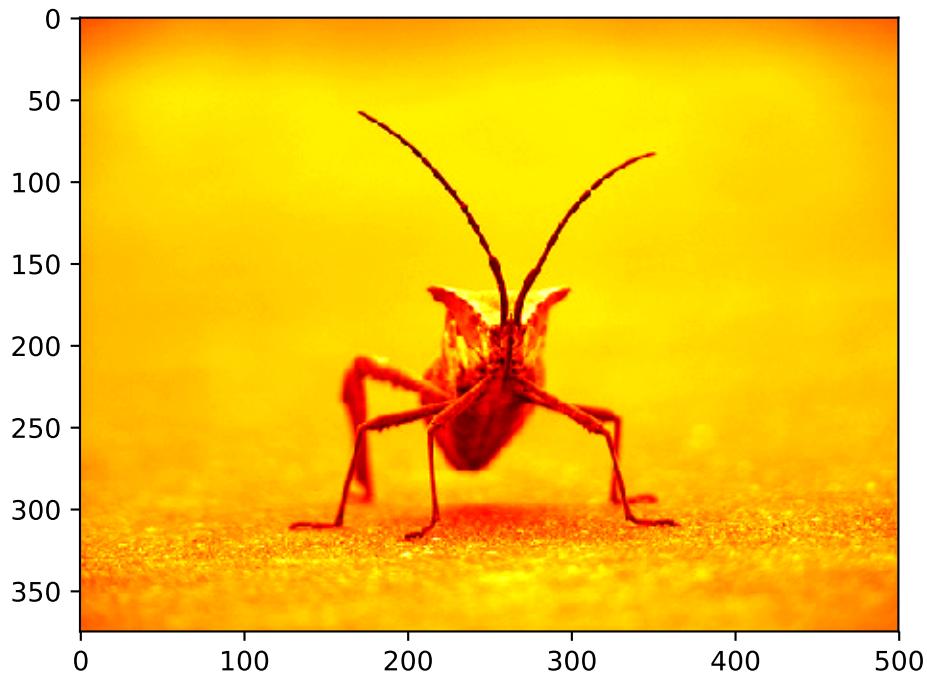
This is array slicing. You can read more in the [Numpy tutorial](#).

```
In [8]: plt.imshow(lum_img)
```



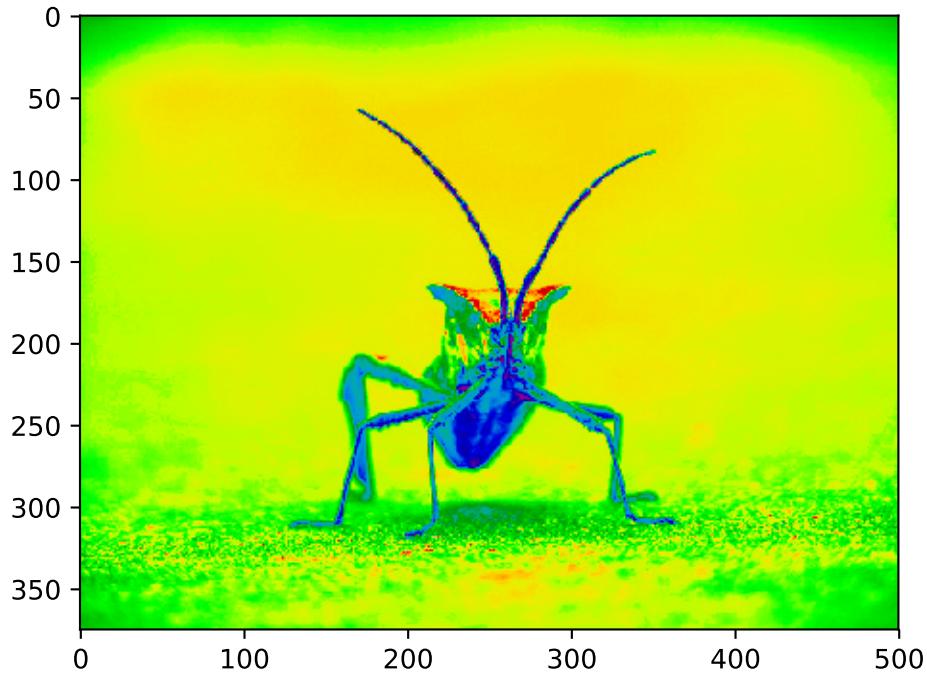
Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table, LUT), is applied. The default is called jet. There are plenty of others to choose from.

```
In [9]: plt.imshow(lum_img, cmap="hot")
```



Note that you can also change colormaps on existing plot objects using the `set_cmap()` method:

```
In [10]: imgplot = plt.imshow(lum_img)
In [11]: imgplot.set_cmap('nipy_spectral')
```



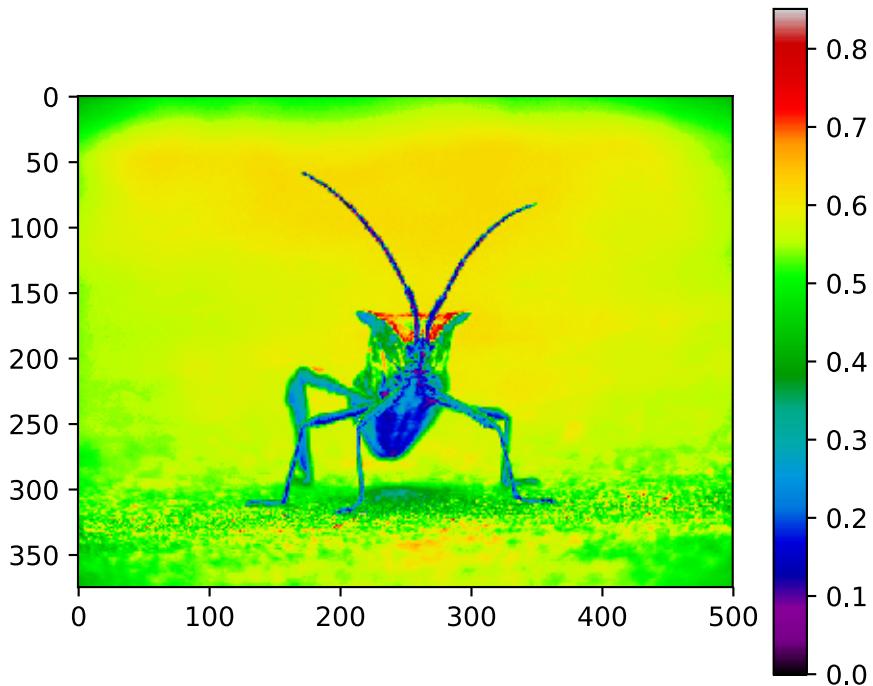
Note: However, remember that in the IPython notebook with the inline backend, you can't make changes to plots that have already been rendered. If you create imgplot here in one cell, you cannot call set_cmap() on it in a later cell and expect the earlier plot to change. Make sure that you enter these commands together in one cell. plt commands will not change plots from earlier cells.

There are many other colormap schemes available. See the list and images of the colormaps.

Color scale reference

It's helpful to have an idea of what value a color represents. We can do that by adding color bars.

```
In [12]: imgplot = plt.imshow(lum_img)
In [13]: plt.colorbar()
```

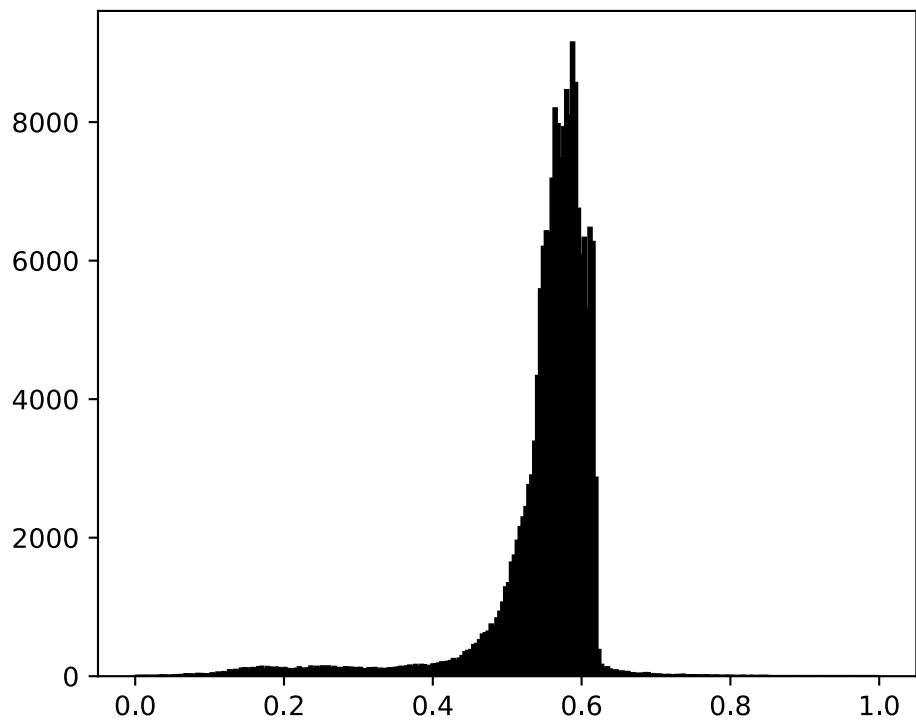


This adds a colorbar to your existing figure. This won't automatically change if you switch to a different colormap - you have to re-create your plot, and add in the colorbar again.

Examining a specific data range

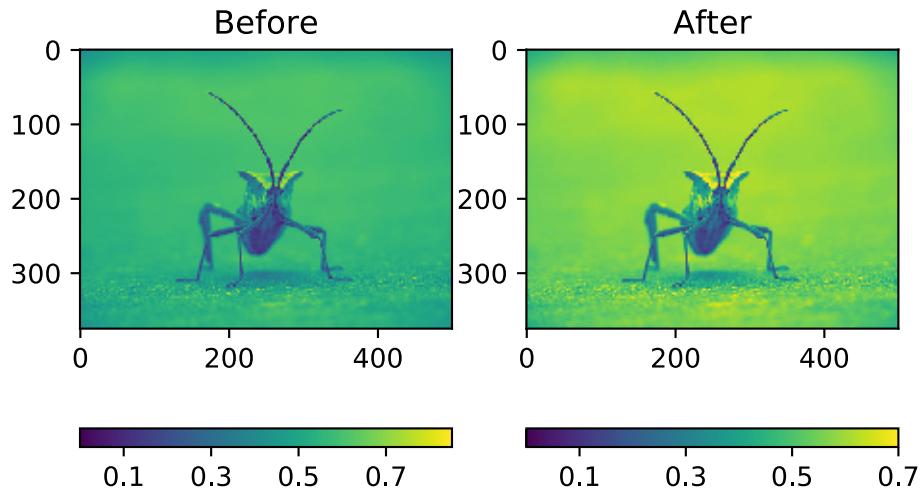
Sometimes you want to enhance the contrast in your image, or expand the contrast in a particular region while sacrificing the detail in colors that don't vary much, or don't matter. A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the `hist()` function.

```
In [14]: plt.hist(lum_img.ravel(), bins=256, range=(0.0, 1.0), fc='k', ec='k')
```



Most often, the “interesting” part of the image is around the peak, and you can get extra contrast by clipping the regions above and/or below the peak. In our histogram, it looks like there’s not much useful information in the high end (not many white things in the image). Let’s adjust the upper limit, so that we effectively “zoom in on” part of the histogram. We do this by passing the `clim` argument to `imshow`. You could also do this by calling the `set_clim()` method of the image plot object, but make sure that you do so in the same cell as your plot command when working with the IPython Notebook - it will not change plots from earlier cells.

```
In [15]: imgplot = plt.imshow(lum_img, clim=(0.0, 0.7))
```

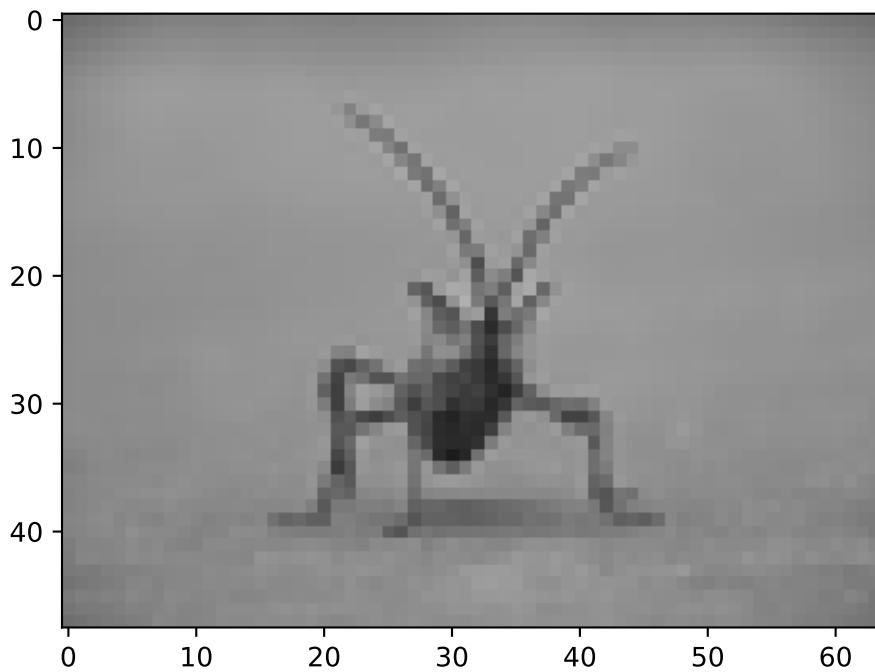


Array Interpolation schemes

Interpolation calculates what the color or value of a pixel “should” be, according to different mathematical schemes. One common place that this happens is when you resize an image. The number of pixels change, but you want the same information. Since pixels are discrete, there’s missing space. Interpolation is how you fill that space. This is why your images sometimes come out looking pixelated when you blow them up. The effect is more pronounced when the difference between the original image and the expanded image is greater. Let’s take our image and shrink it. We’re effectively discarding pixels, only keeping a select few. Now when we plot it, that data gets blown up to the size on your screen. The old pixels aren’t there anymore, and the computer has to draw in pixels to fill that space.

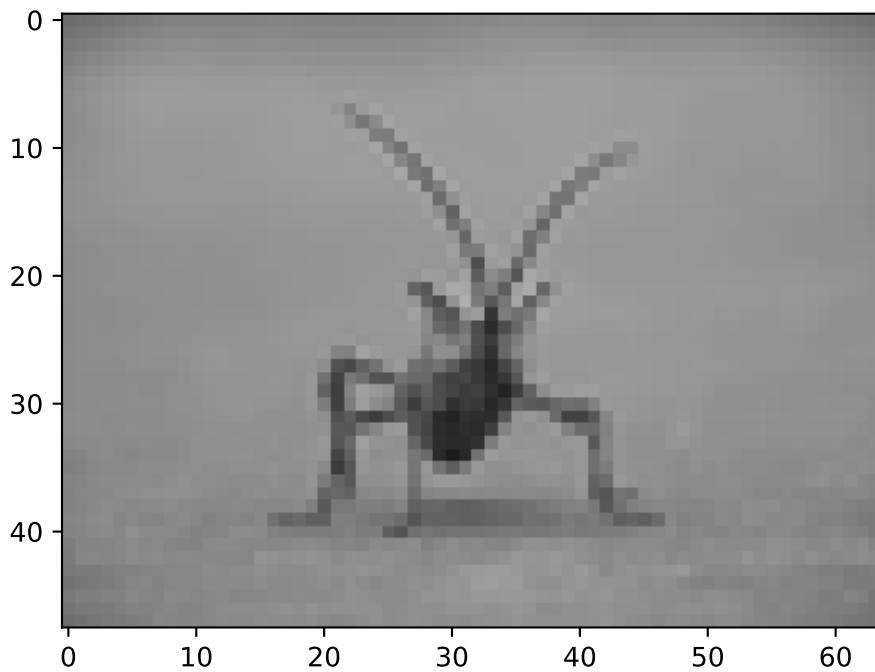
We’ll use the Pillow library that we used to load the image also to resize the image.

```
In [16]: from PIL import Image
In [17]: img = Image.open('../_static/stinkbug.png')
In [18]: img.thumbnail((64, 64), Image.ANTIALIAS) # resizes image in-place
In [19]: imgplot = plt.imshow(img)
```

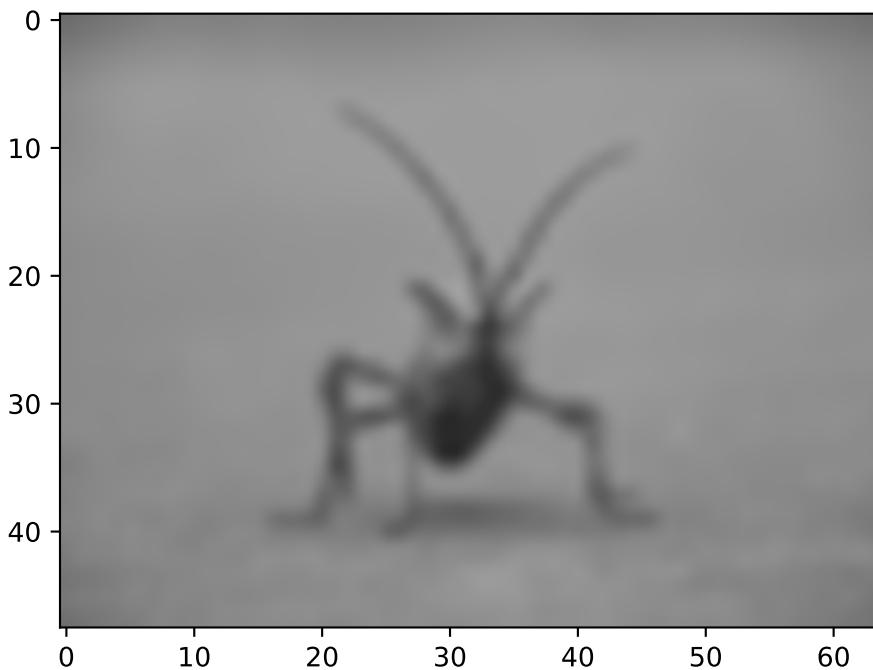


Here we have the default interpolation, bilinear, since we did not give `imshow()` any interpolation argument.
Let's try some others:

```
In [20]: imgplot = plt.imshow(img, interpolation="nearest")
```



```
In [21]: imgplot = plt.imshow(img, interpolation="bicubic")
```



Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

3.1.3 Customizing Location of Subplot Using GridSpec

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

SubplotSpec specifies the location of the subplot in the given *GridSpec*.

subplot2grid a helper function that is similar to “`pyplot.subplot`” but uses 0-based indexing and let subplot to occupy multiple cells.

Basic Example of using subplot2grid

To use `subplot2grid`, you provide geometry of the grid and the location of the subplot in the grid. For a simple single-cell subplot:

```
ax = plt.subplot2grid((2,2),(0, 0))
```

is identical to

```
ax = plt.subplot(2,2,1)
```

Note that, unlike matplotlib's subplot, the index starts from 0 in gridspec.

To create a subplot that spans multiple cells,

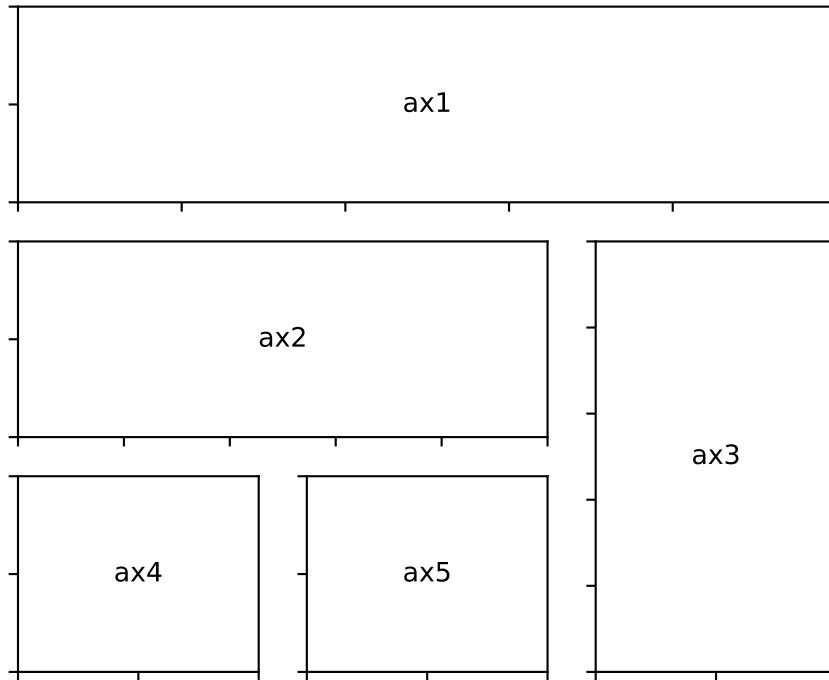
```
ax2 = plt.subplot2grid((3,3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
```

For example, the following commands

```
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2, 0))
ax5 = plt.subplot2grid((3,3), (2, 1))
```

creates

subplot2grid



GridSpec and SubplotSpec

You can create GridSpec explicitly and use them to create a Subplot.

For example,

```
ax = plt.subplot2grid((2,2),(0, 0))
```

is equal to

```
import matplotlib.gridspec as gridspec
gs = gridspec.GridSpec(2, 2)
ax = plt.subplot(gs[0, 0])
```

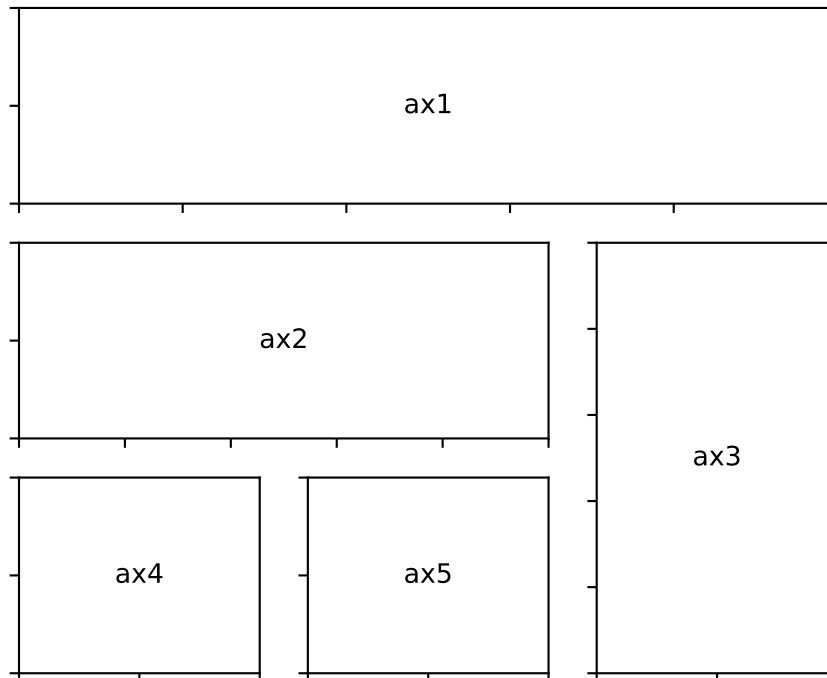
A gridspec instance provides array-like (2d or 1d) indexing that returns the SubplotSpec instance. For, SubplotSpec that spans multiple cells, use slice.

```
ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])
```

The above example becomes

```
gs = gridspec.GridSpec(3, 3)
ax1 = plt.subplot(gs[0, :])
ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])
ax4 = plt.subplot(gs[-1,0])
ax5 = plt.subplot(gs[-1,-2])
```

GridSpec



Adjust GridSpec layout

When a GridSpec is explicitly used, you can adjust the layout parameters of subplots that are created from the gridspec.

```
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
```

This is similar to `subplots_adjust`, but it only affects the subplots that are created from the given GridSpec.

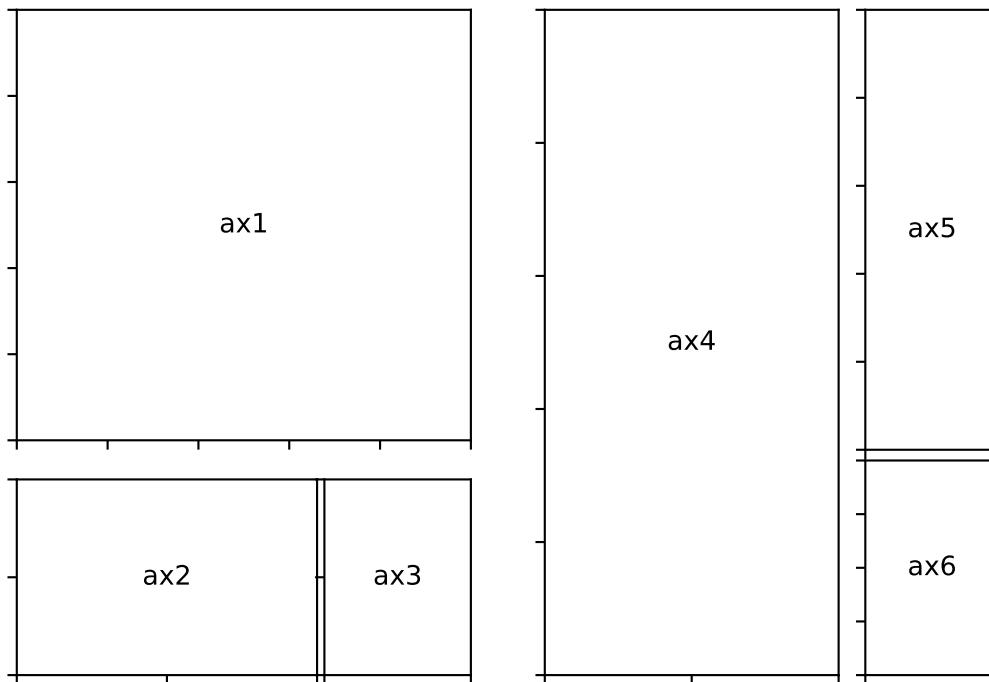
The code below

```
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
ax1 = plt.subplot(gs1[:-1, :])
ax2 = plt.subplot(gs1[-1, :-1])
ax3 = plt.subplot(gs1[-1, -1])

gs2 = gridspec.GridSpec(3, 3)
gs2.update(left=0.55, right=0.98, hspace=0.05)
ax4 = plt.subplot(gs2[:, :-1])
ax5 = plt.subplot(gs2[:-1, -1])
ax6 = plt.subplot(gs2[-1, -1])
```

creates

GridSpec w/ different subplotpars



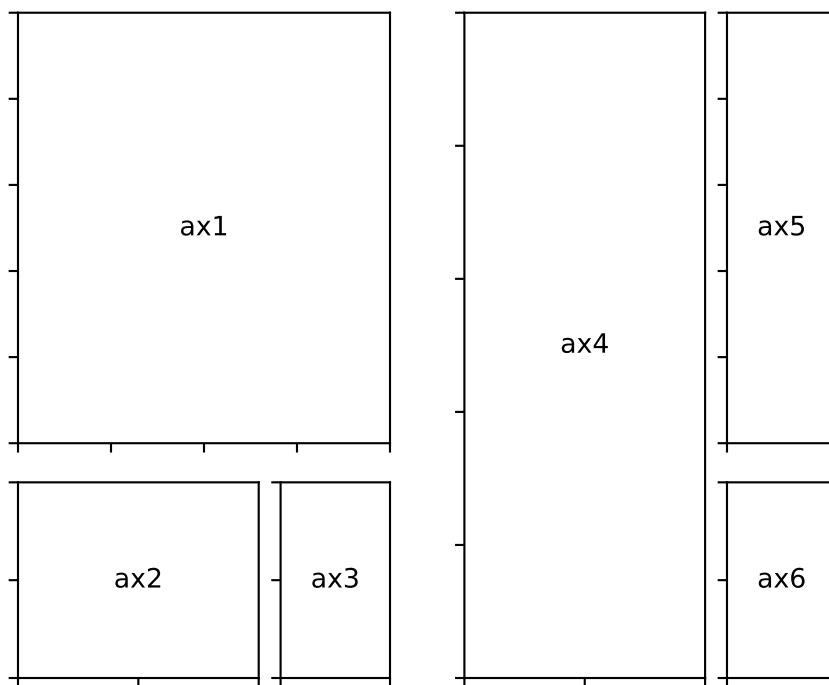
GridSpec using SubplotSpec

You can create GridSpec from the SubplotSpec, in which case its layout parameters are set to that of the location of the given SubplotSpec.

```
gs0 = gridspec.GridSpec(1, 2)

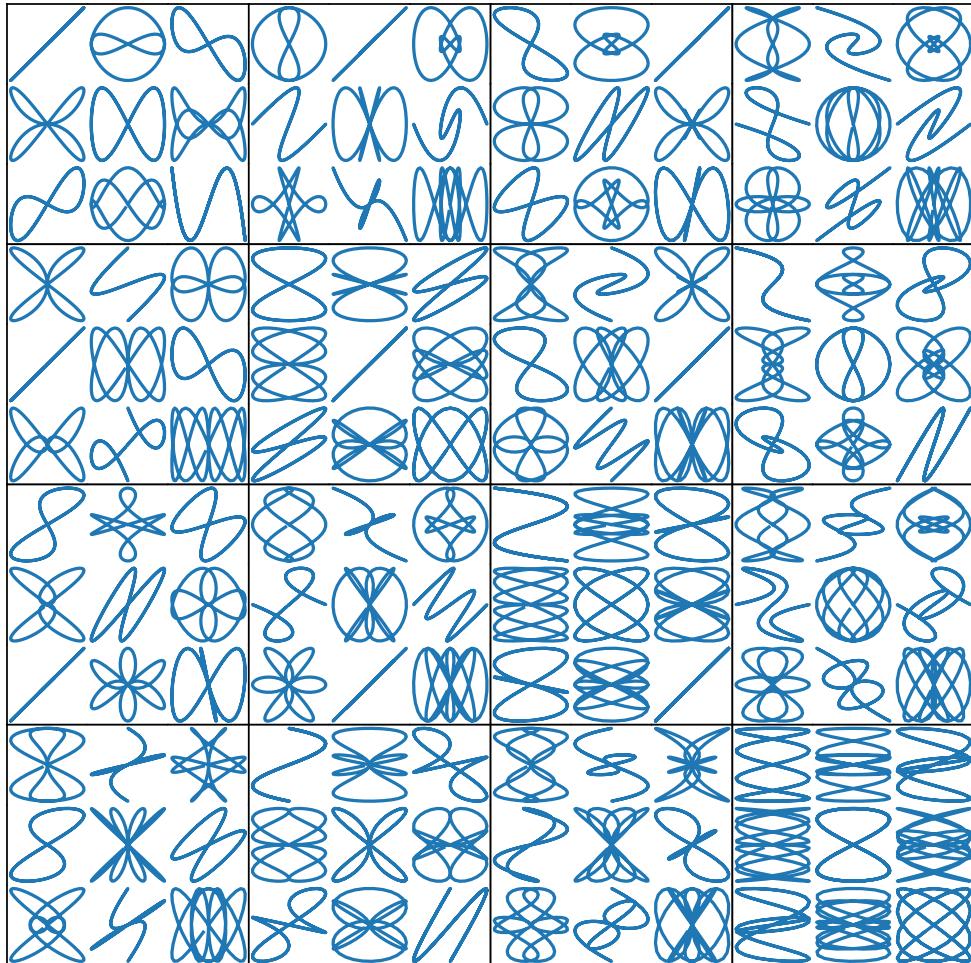
gs00 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[0])
gs01 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[1])
```

GridSpec Inside GridSpec



A Complex Nested GridSpec using SubplotSpec

Here's a more sophisticated example of nested gridspec where we put a box around each cell of the outer 4x4 grid, by hiding appropriate spines in each of the inner 3x3 grids.



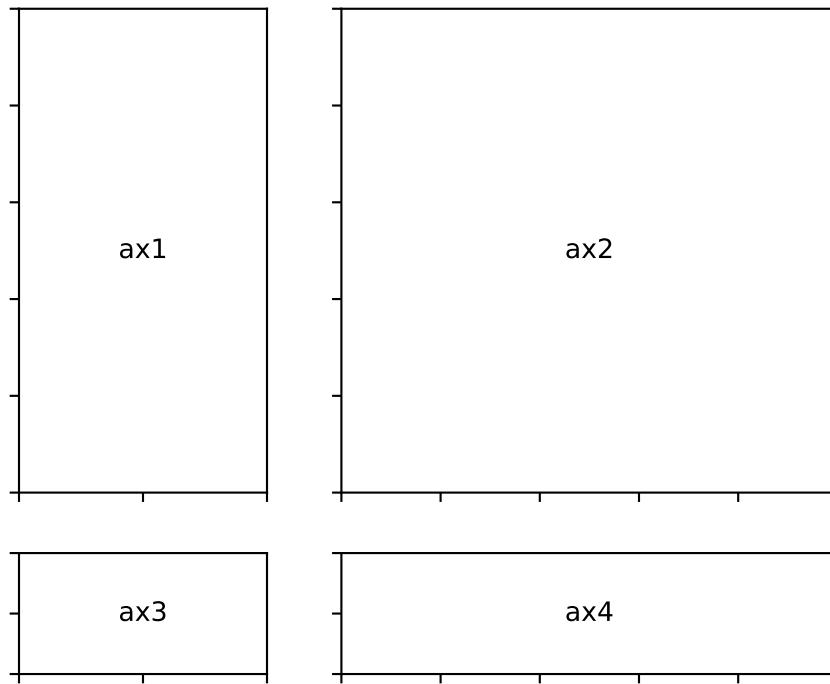
GridSpec with Varying Cell Sizes

By default, GridSpec creates cells of equal sizes. You can adjust relative heights and widths of rows and columns. Note that absolute values are meaningless, only their relative ratios matter.

```
gs = gridspec.GridSpec(2, 2,
                      width_ratios=[1, 2],
                      height_ratios=[4, 1]
                     )

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
```

```
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])
```



3.1.4 Tight Layout guide

tight_layout automatically adjusts subplot params so that the subplot(s) fits in to the figure area. This is an experimental feature and may not work for some cases. It only checks the extents of ticklabels, axis labels, and titles.

Simple Example

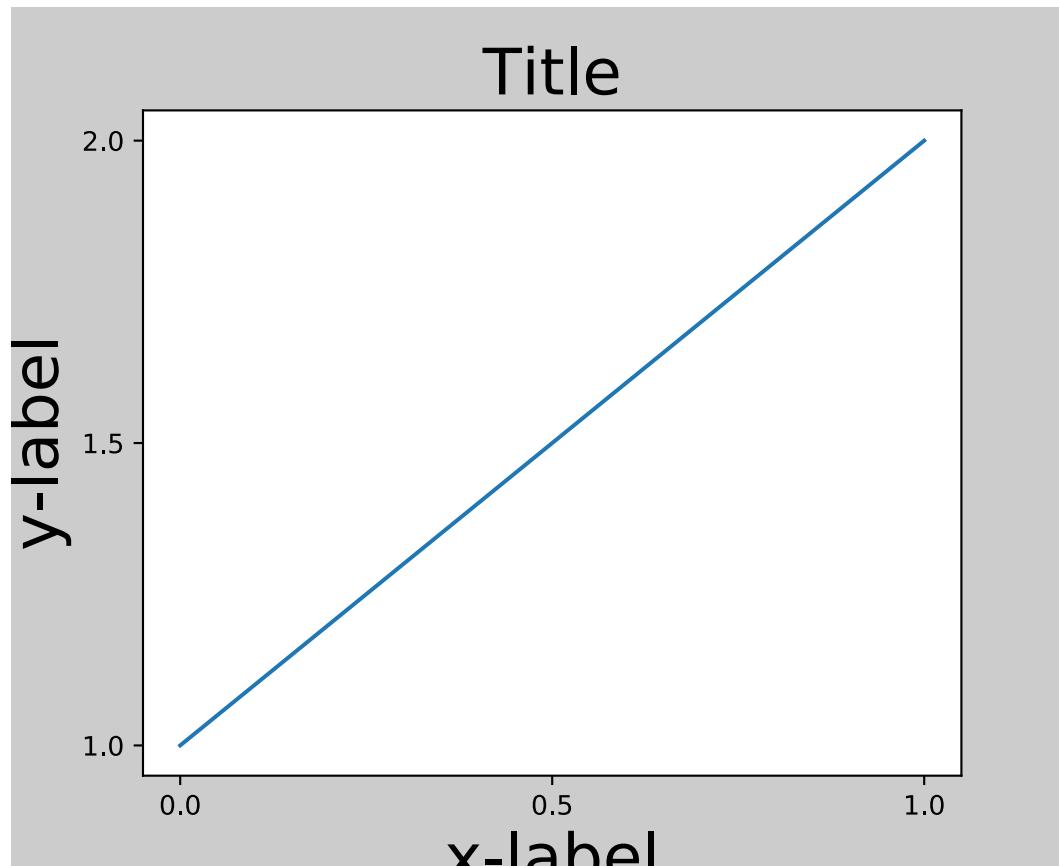
In matplotlib, the location of axes (including subplots) are specified in normalized figure coordinates. It can happen that your axis labels or titles (or sometimes even ticklabels) go outside the figure area, and are thus clipped.

```
plt.rcParams['savefig.facecolor'] = "0.8"

def example_plot(ax, fontsize=12):
    ax.plot([1, 2])
    ax.locator_params(nbins=3)
    ax.set_xlabel('x-label', fontsize=fontsize)
    ax.set_ylabel('y-label', fontsize=fontsize)
```

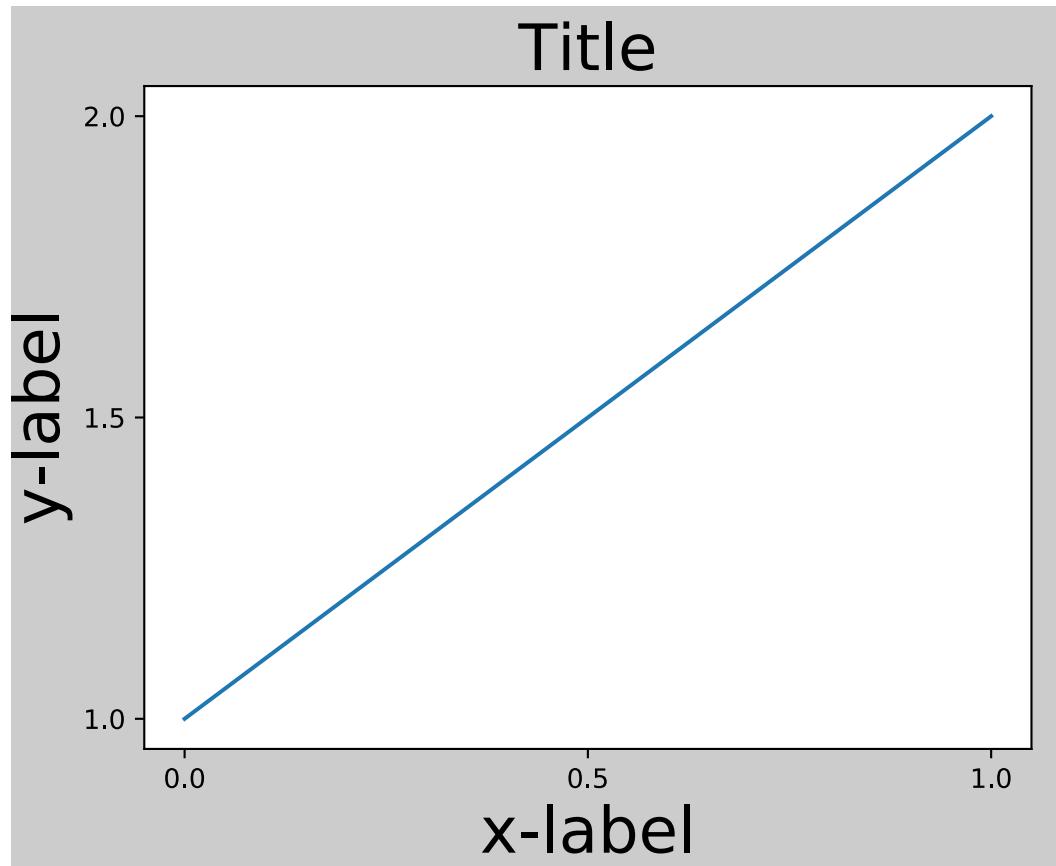
```
ax.set_title('Title', fontsize=fontsize)

plt.close('all')
fig, ax = plt.subplots()
example_plot(ax, fontsize=24)
```



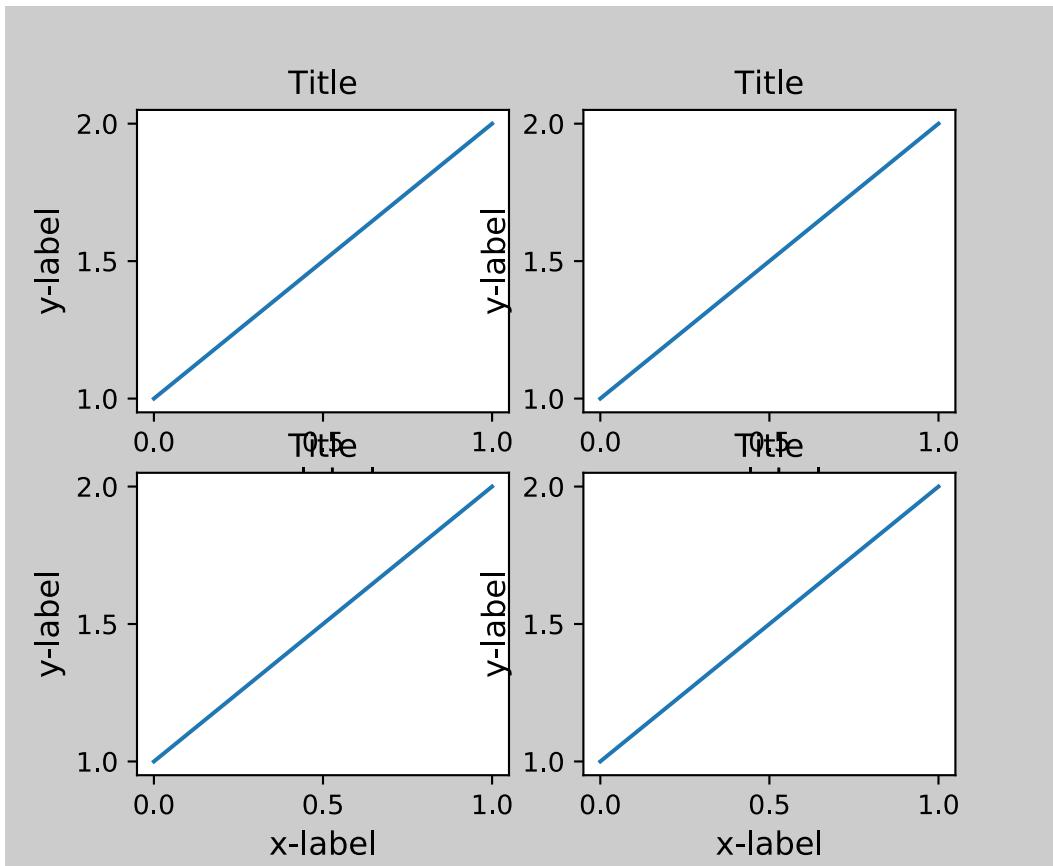
To prevent this, the location of axes needs to be adjusted. For subplots, this can be done by adjusting the subplot params ([Move the edge of an axes to make room for tick labels](#)). Matplotlib v1.1 introduces a new command `tight_layout()` that does this automatically for you.

```
plt.tight_layout()
```



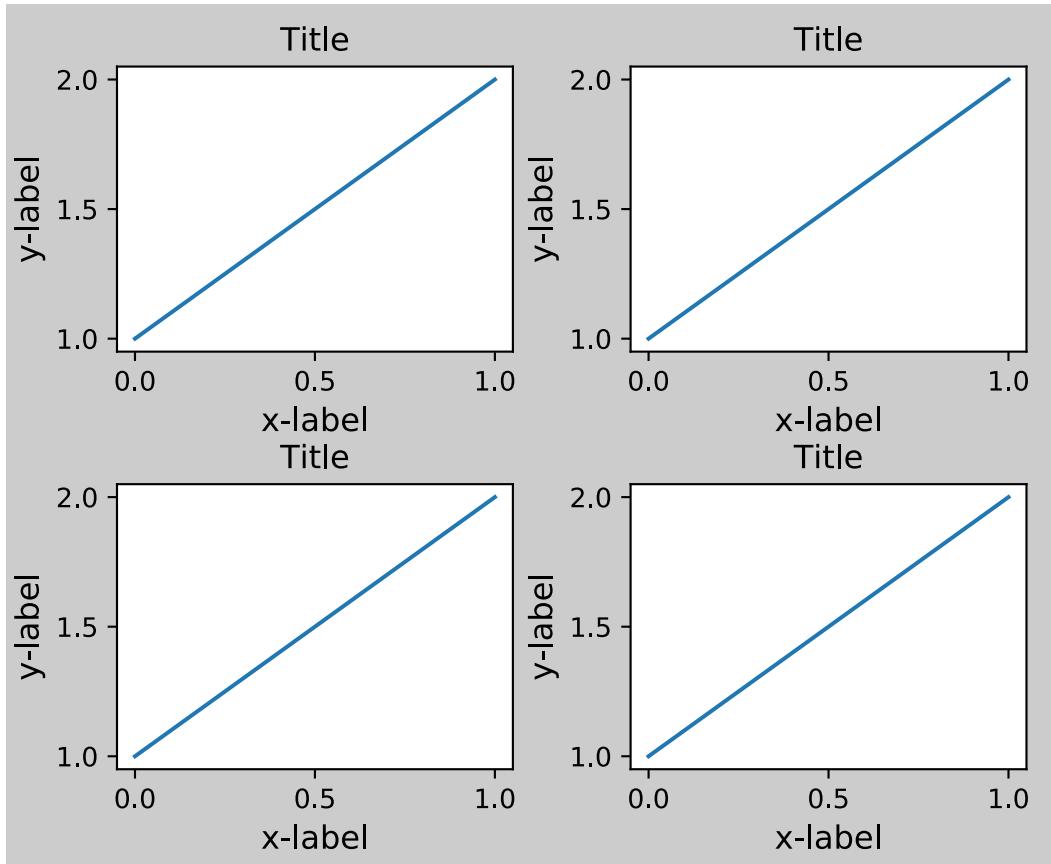
When you have multiple subplots, often you see labels of different axes overlapping each other.

```
plt.close('all')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```



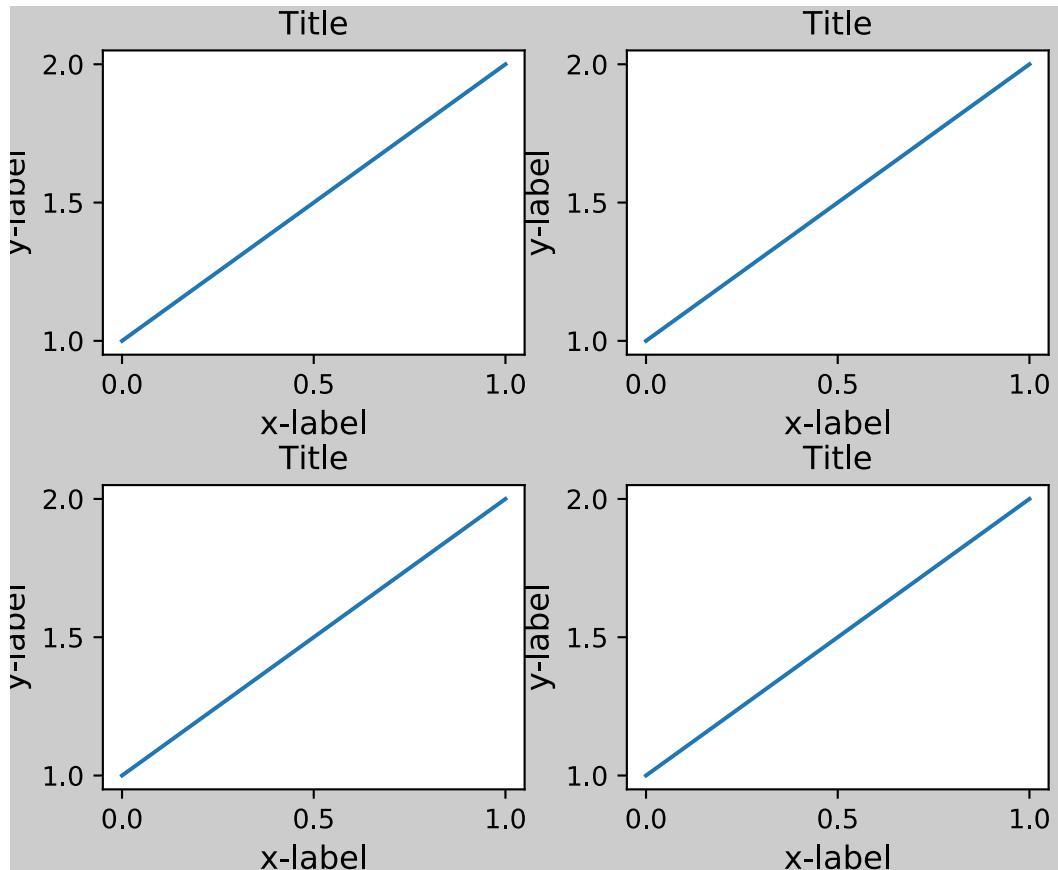
`tight_layout()` will also adjust spacing between subplots to minimize the overlaps.

```
plt.tight_layout()
```



`tight_layout()` can take keyword arguments of `pad`, `w_pad` and `h_pad`. These control the extra padding around the figure border and between subplots. The pads are specified in fraction of fontsize.

```
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



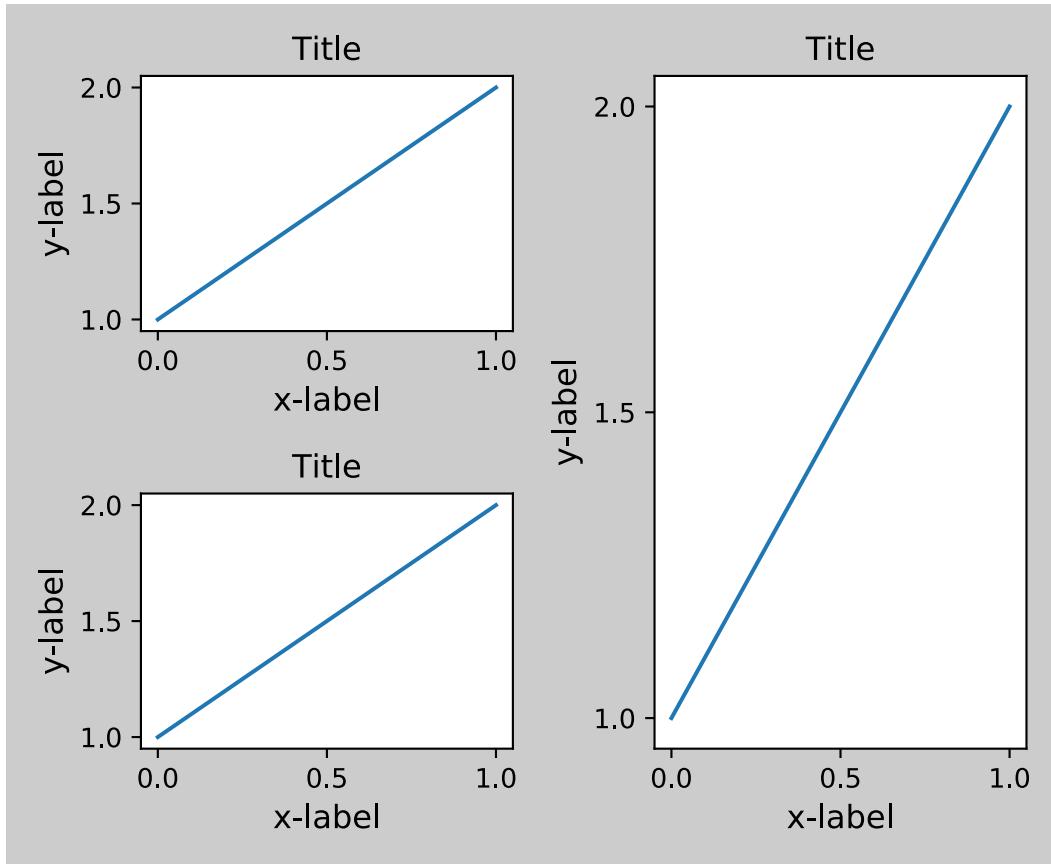
`tight_layout()` will work even if the sizes of subplots are different as far as their grid specification is compatible. In the example below, `ax1` and `ax2` are subplots of a 2×2 grid, while `ax3` is of a 1×2 grid.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

plt.tight_layout()
```



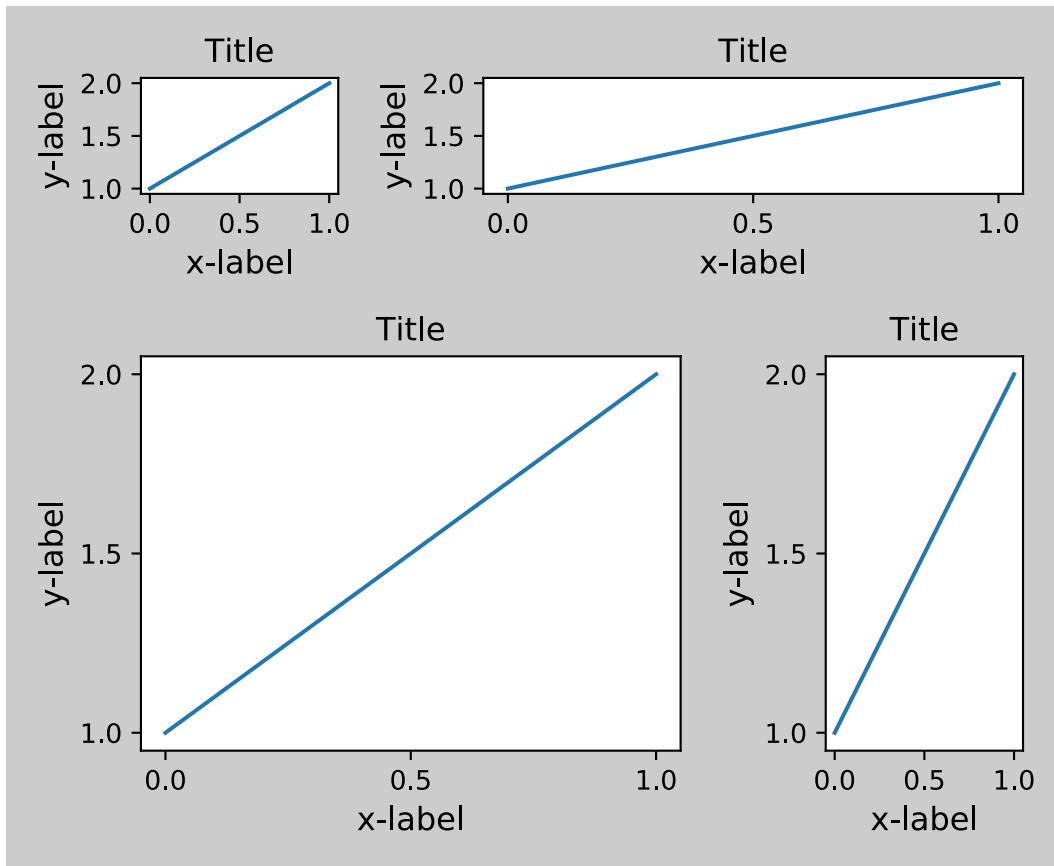
It works with subplots created with `subplot2grid()`. In general, subplots created from the gridspec ([Customizing Location of Subplot Using GridSpec](#)) will work.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()
```



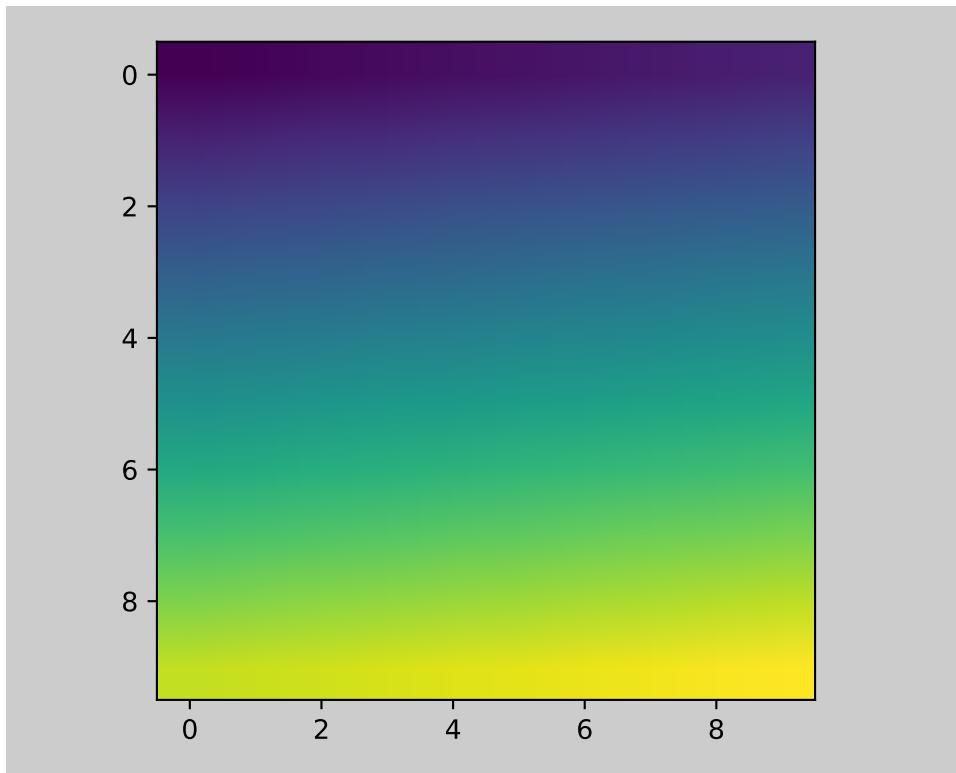
Although not thoroughly tested, it seems to work for subplots with aspect != "auto" (e.g., axes with images).

```
arr = np.arange(100).reshape((10,10))

plt.close('all')
fig = plt.figure(figsize=(5,4))

ax = plt.subplot(111)
im = ax.imshow(arr, interpolation="none")

plt.tight_layout()
```



Caveats

- `tight_layout()` only considers ticklabels, axis labels, and titles. Thus, other artists may be clipped and also may overlap.
- It assumes that the extra space needed for ticklabels, axis labels, and titles is independent of original location of axes. This is often true, but there are rare cases where it is not.
- pad=0 clips some of the texts by a few pixels. This may be a bug or a limitation of the current algorithm and it is not clear why it happens. Meanwhile, use of pad at least larger than 0.3 is recommended.

Use with GridSpec

GridSpec has its own `tight_layout()` method (the pyplot api `tight_layout()` also works).

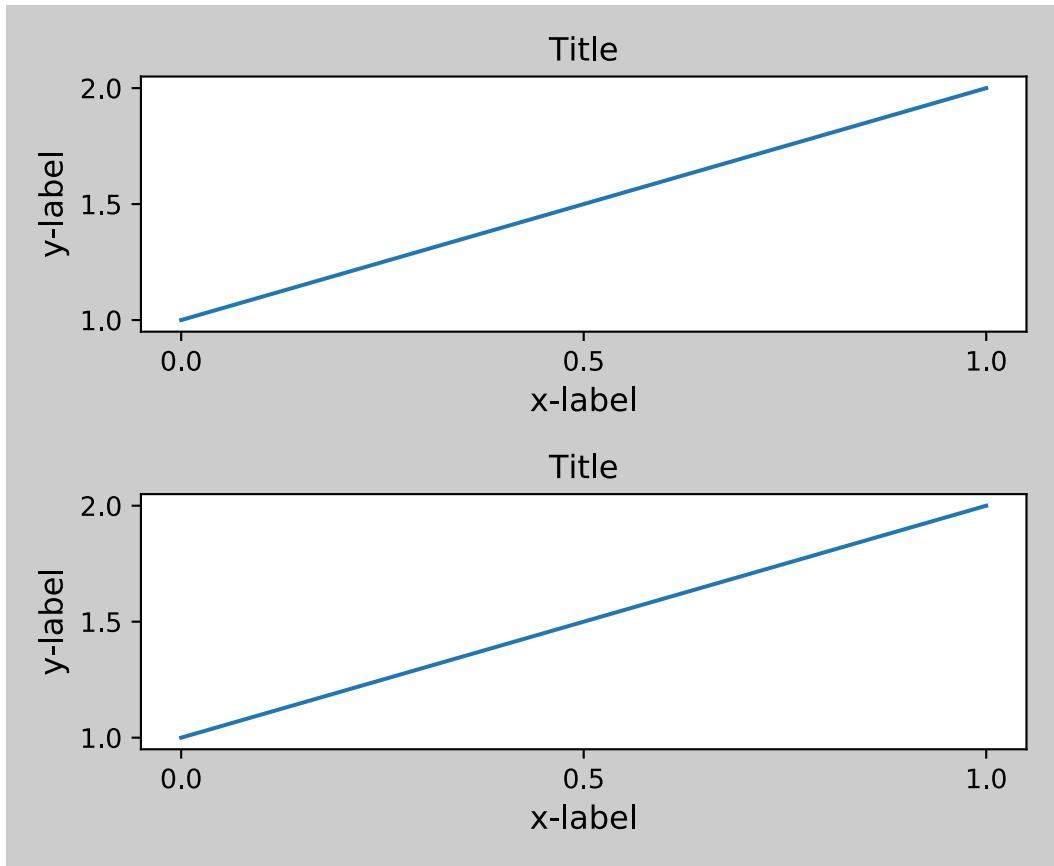
```
plt.close('all')
fig = plt.figure()

import matplotlib.gridspec as gridspec

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])
```

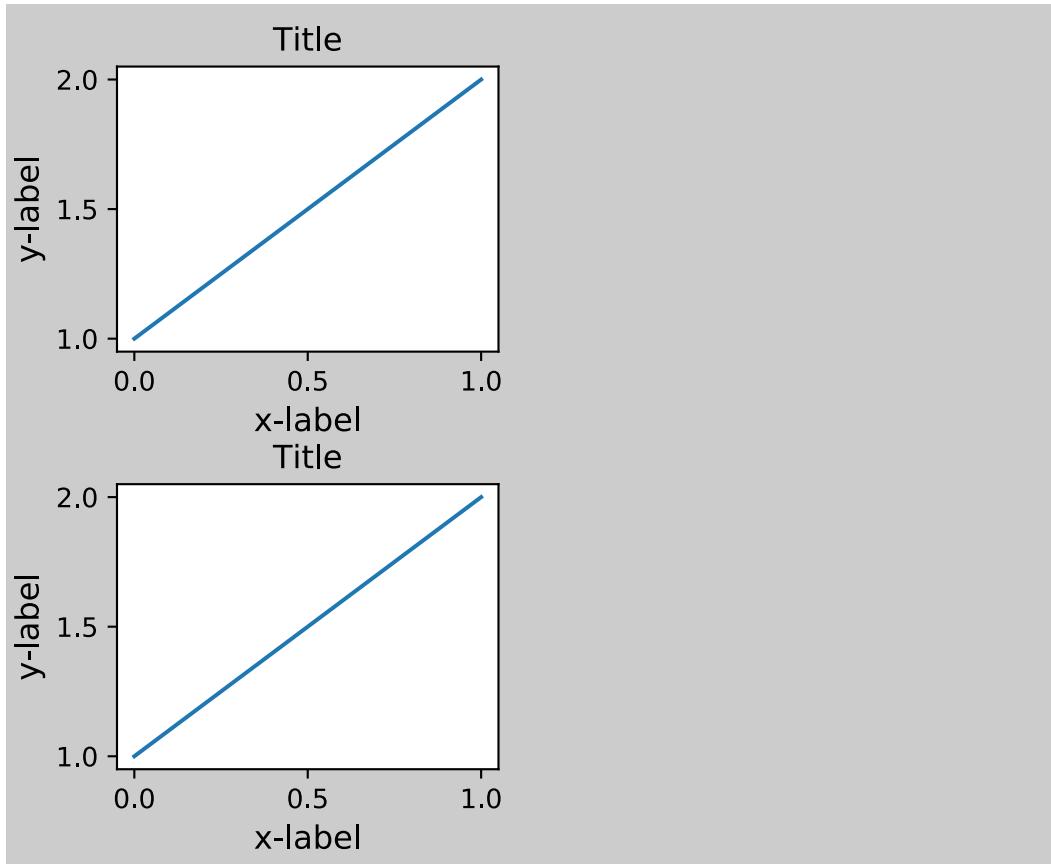
```
example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig)
```



You may provide an optional `rect` parameter, which specifies the bounding box that the subplots will be fit inside. The coordinates must be in normalized figure coordinates and the default is (0, 0, 1, 1).

```
gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
```



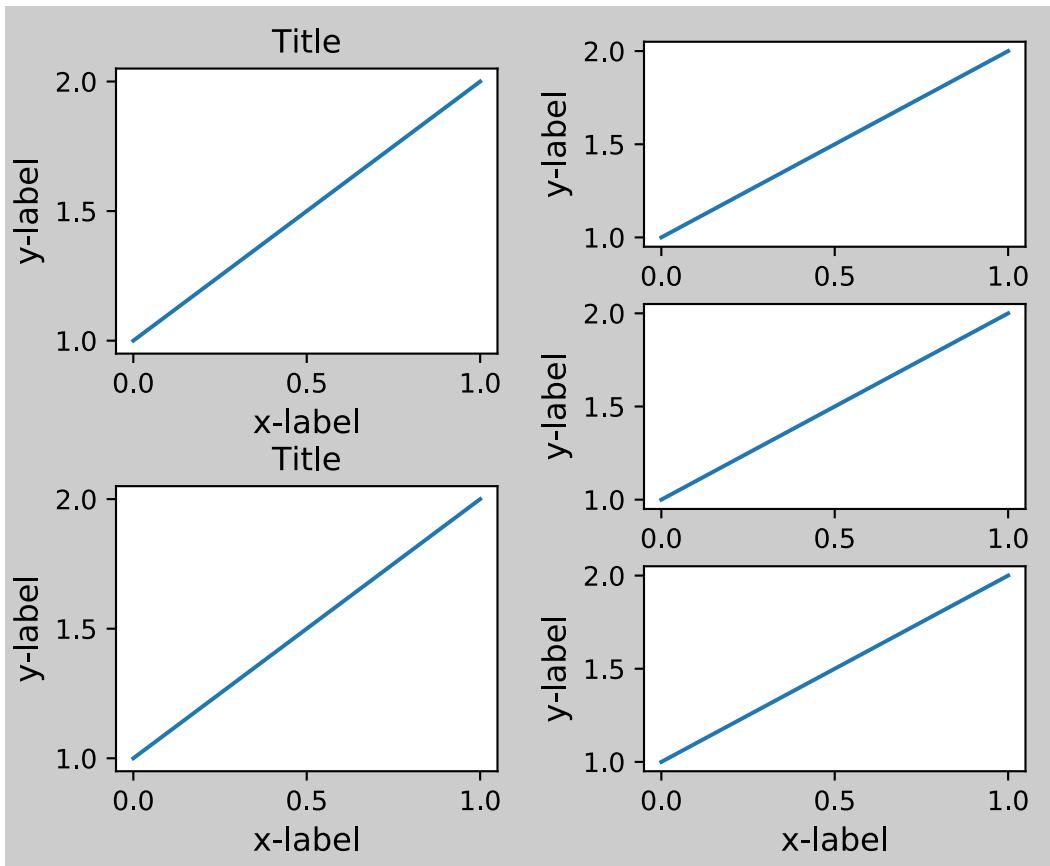
For example, this can be used for a figure with multiple gridspecs.

```
gs2 = gridspec.GridSpec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title("Title")
    ax.set_xlabel("x-label")

    ax.set_xlabel("x-label", fontsize=12)

gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)
```



We may try to match the top and bottom of two grids

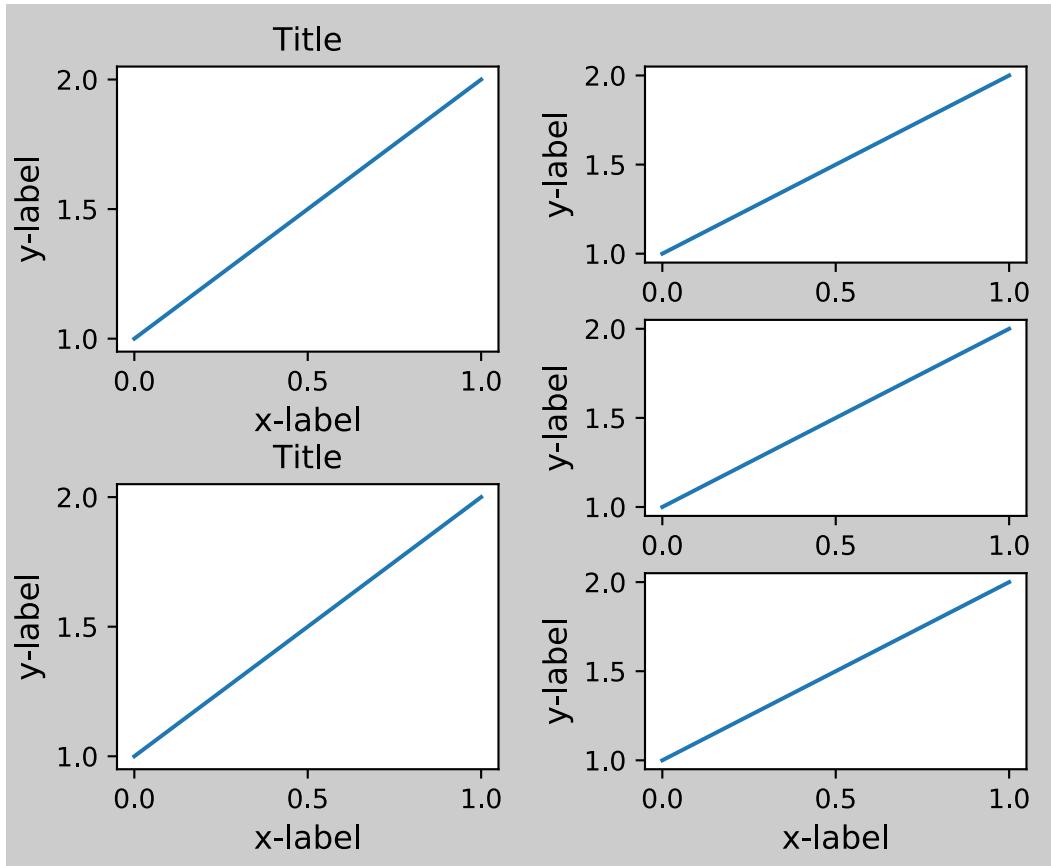
```
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)
```

While this should be mostly good enough, adjusting top and bottom may require adjustment of hspace also. To update hspace & vspace, we call `tight_layout()` again with updated rect argument. Note that the rect argument specifies the area including the ticklabels, etc. Thus, we will increase the bottom (which is 0 for the normal case) by the difference between the *bottom* from above and the bottom of each gridspec. Same thing for the top.

```
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom),
                           0.5, 1 - (gs1.top-top)])
gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom),
                           None, 1 - (gs2.top-top)],
                 h_pad=0.5)
```



Use with AxesGrid1

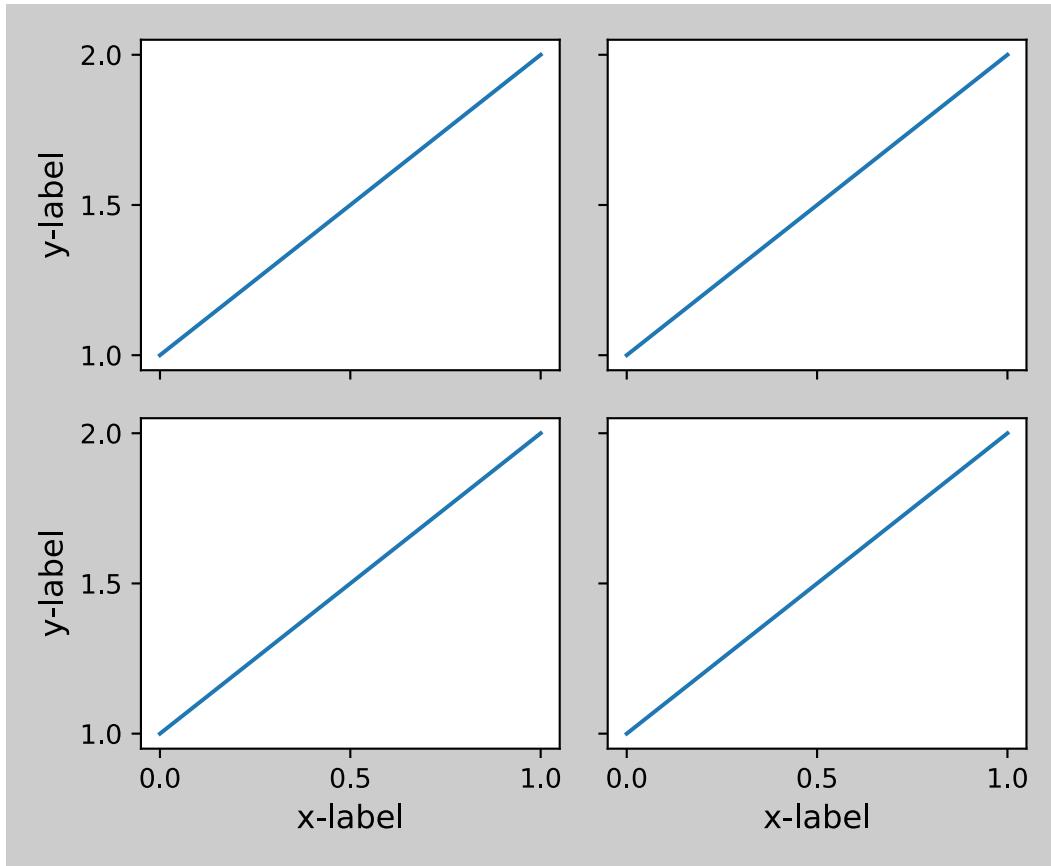
While limited, the axes_grid1 toolkit is also supported.

```
plt.close('all')
fig = plt.figure()

from mpl_toolkits.axes_grid1 import Grid
grid = Grid(fig, rect=111, nrows_ncols=(2,2),
            axes_pad=0.25, label_mode='L',
            )

for ax in grid:
    example_plot(ax)
    ax.title.set_visible(False)

plt.tight_layout()
```



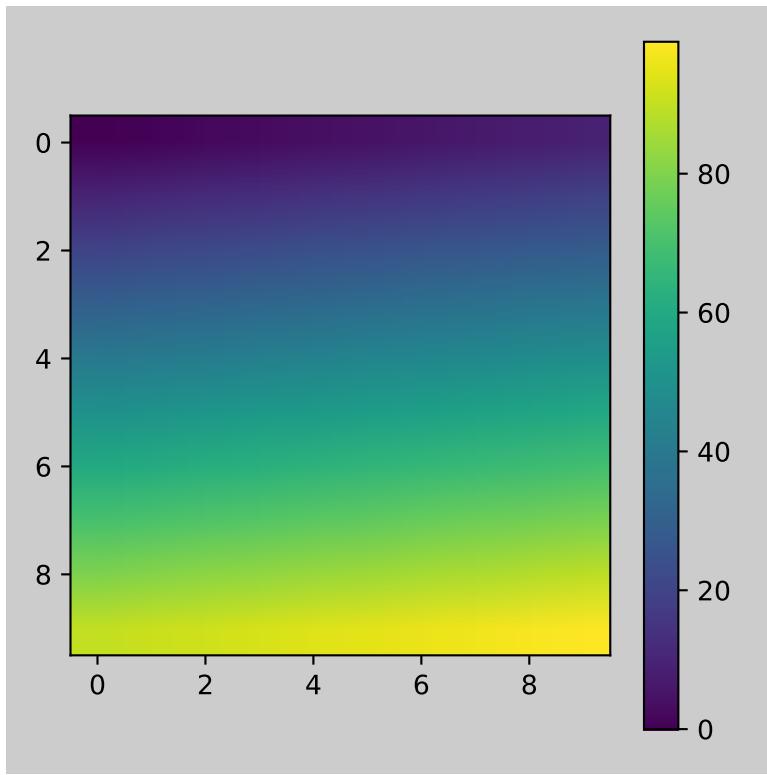
Colorbar

If you create a colorbar with the `colorbar()` command, the created colorbar is an instance of `Axes`, *not* `Subplot`, so `tight_layout` does not work. With Matplotlib v1.1, you may create a colorbar as a subplot using the `gridspec`.

```
plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

plt.colorbar(im, use_gridspec=True)

plt.tight_layout()
```

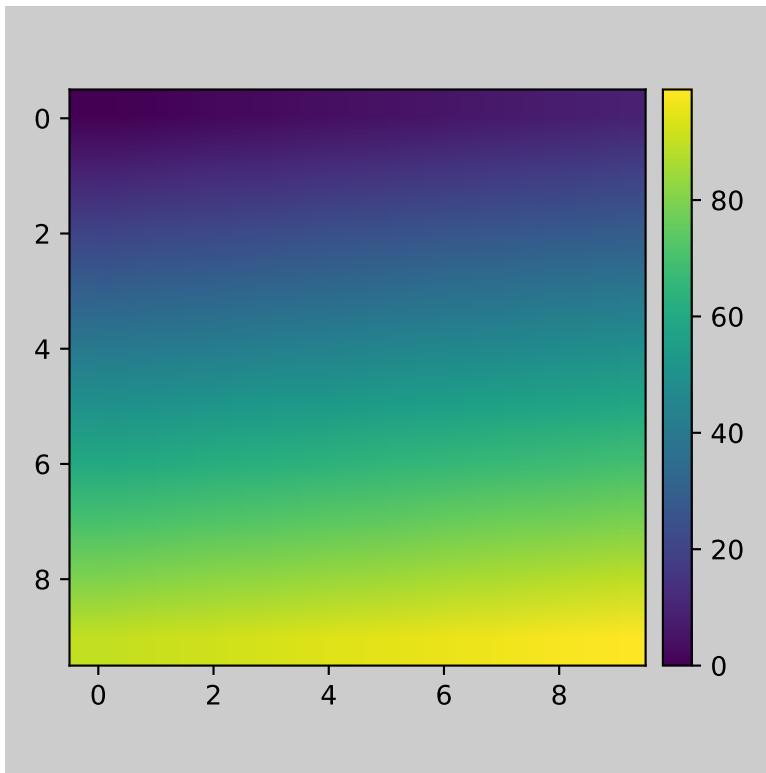


Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.

```
plt.close('all')
arr = np.arange(100).reshape((10,10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

from mpl_toolkits.axes_grid1 import make_axes_locatable
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%", pad="3%")
plt.colorbar(im, cax=cax)

plt.tight_layout()
```



3.2 Intermediate

3.2.1 Artist tutorial

There are three layers to the matplotlib API. The `matplotlib.backend_bases.FigureCanvas` is the area onto which the figure is drawn, the `matplotlib.backend_bases.Renderer` is the object which knows how to draw on the `FigureCanvas`, and the `matplotlib.artist.Artist` is the object that knows how to use a renderer to paint onto the canvas. The `FigureCanvas` and `Renderer` handle all the details of talking to user interface toolkits like `wxPython` or drawing languages like PostScript®, and the `Artist` handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of his time working with the `Artists`.

There are two types of `Artists`: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: `Line2D`, `Rectangle`, `Text`, `AxesImage`, etc., and the containers are places to put them (`Axis`, `Axes` and `Figure`). The standard use is to create a `Figure` instance, use the `Figure` to create one or more `Axes` or Subplot instances, and use the `Axes` instance helper methods to create the primitives. In the example below, we create a `Figure` instance using `matplotlib.pyplot.figure()`, which is a convenience method for instantiating `Figure` instances and connecting them with your user interface or drawing toolkit `FigureCanvas`. As we will discuss below, this is not necessary – you can work directly with PostScript, PDF Gtk+, or `wxPython` `FigureCanvas` instances, instantiate your `Figures` directly and connect them yourselves – but since we are focusing here on the `Artist` API we'll let `pyplot` handle some of those details for us:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2,1,1) # two rows, one column, first plot
```

The `Axes` is probably the most important class in the matplotlib API, and the one you will be working with most of the time. This is because the `Axes` is the plotting area into which most of the objects go, and the `Axes` has many special helper methods (`plot()`, `text()`, `hist()`, `imshow()`) to create the most common graphics primitives (`Line2D`, `Text`, `Rectangle`, `Image`, respectively). These helper methods will take your data (e.g., numpy arrays and strings) and create primitive `Artist` instances as needed (e.g., `Line2D`), add them to the relevant containers, and draw them when requested. Most of you are probably familiar with the `Subplot`, which is just a special case of an `Axes` that lives on a regular rows by columns grid of `Subplot` instances. If you want to create an `Axes` at an arbitrary location, simply use the `add_axes()` method which takes a list of [`left`, `bottom`, `width`, `height`] values in 0-1 relative figure coordinates:

```
fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])
```

Continuing with our example:

```
import numpy as np
t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)
```

In this example, `ax` is the `Axes` instance created by the `fig.add_subplot` call above (remember `Subplot` is just a subclass of `Axes`) and when you call `ax.plot`, it creates a `Line2D` instance and adds it to the `Axes.lines` list. In the interactive `ipython` session below, you can see that the `Axes.lines` list is length one and contains the same line that was returned by the `line, = ax.plot...` call:

```
In [101]: ax.lines[0]
Out[101]: <matplotlib.lines.Line2D instance at 0x19a95710>

In [102]: line
Out[102]: <matplotlib.lines.Line2D instance at 0x19a95710>
```

If you make subsequent calls to `ax.plot` (and the hold state is “on” which is the default) then additional lines will be added to the list. You can remove lines later simply by calling the list methods; either of these will work:

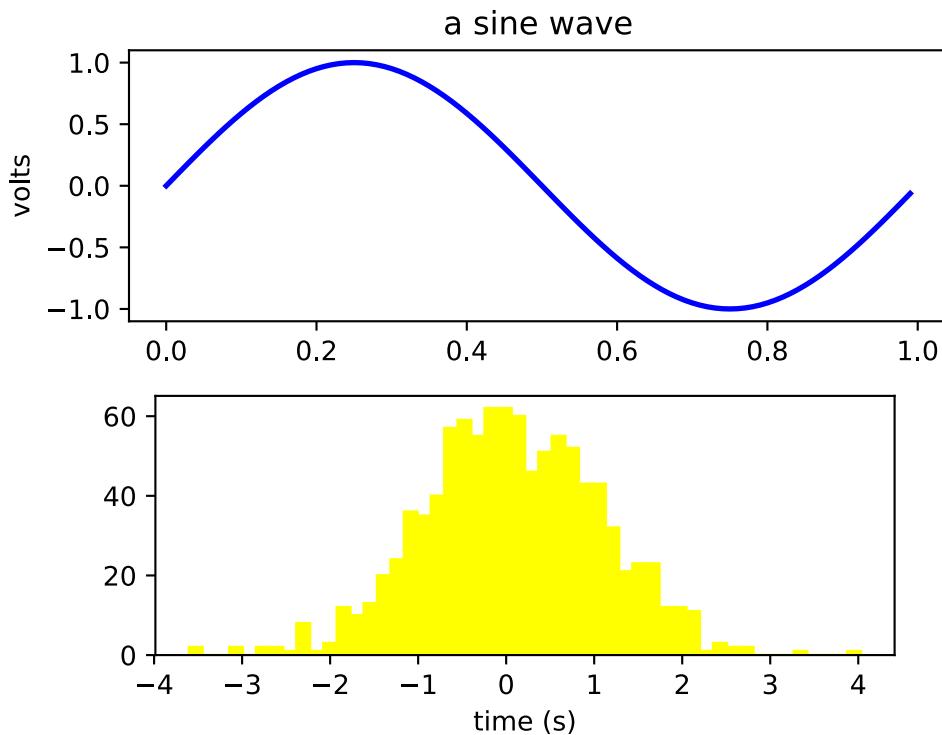
```
del ax.lines[0]
ax.lines.remove(line) # one or the other, not both!
```

The `Axes` also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

```
xtext = ax.set_xlabel('my xdata') # returns a Text instance
ytext = ax.set_ylabel('my ydata')
```

When you call `ax.set_xlabel`, it passes the information on the `Text` instance of the `XAxis`. Each `Axes` instance contains an `XAxis` and a `YAxis` instance, which handle the layout and drawing of the ticks, tick labels and axis labels.

Try creating the figure below.



Customizing your objects

Every element in the figure is represented by a matplotlib `Artist`, and each has an extensive list of properties to configure its appearance. The figure itself contains a `Rectangle` exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each `Axes` bounding box (the standard white box with black edges in the typical matplotlib plot, has a `Rectangle` instance that determines the color, transparency, and other properties of the Axes. These instances are stored as member variables `Figure.patch` and `Axes.patch` (“Patch” is a name inherited from MATLAB, and is a 2D “patch” of color on the figure, e.g., rectangles, circles and polygons). Every matplotlib `Artist` has the following properties

Property	Description
alpha	The transparency - a scalar from 0-1
animated	A boolean that is used to facilitate animated drawing
axes	The axes that the Artist lives in, possibly None
clip_box	The bounding box that clips the Artist
clip_on	Whether clipping is enabled
clip_path	The path the artist is clipped to
contains	A picking function to test whether the artist contains the pick point
figure	The figure instance the artist lives in, possibly None
label	A text label (e.g., for auto-labeling)
picker	A python object that controls object picking
transform	The transformation
visible	A boolean whether the artist should be drawn
zorder	A number which determines the drawing order
rasterized	Boolean; Turns vectors into rastergraphics: (for compression & eps transparency)

Each of the properties is accessed with an old-fashioned setter or getter (yes we know this irritates Pythonistas and we plan to support direct access via properties or traits but it hasn't been done yet). For example, to multiply the current alpha by a half:

```
a = o.get_alpha()  
o.set_alpha(0.5*a)
```

If you want to set a number of properties at once, you can also use the `set` method with keyword arguments. For example:

```
o.set(alpha=0.5, zorder=2)
```

If you are working interactively at the python shell, a handy way to inspect the `Artist` properties is to use the `matplotlib.artist.getp()` function (simply `getp()` in pylab), which lists the properties and their values. This works for classes derived from `Artist` as well, e.g., `Figure` and `Rectangle`. Here are the `Figure` rectangle properties mentioned above:

```
In [149]: matplotlib.artist.getp(fig.patch)  
alpha = 1.0  
animated = False  
antialiased or aa = True  
axes = None  
clip_box = None  
clip_on = False  
clip_path = None  
contains = None  
edgecolor or ec = w  
facecolor or fc = 0.75  
figure = Figure(8.125x6.125)  
fill = 1  
hatch = None  
height = 1  
label =  
linewidth or lw = 1.0  
picker = None
```

```

transform = <Affine object at 0x134cca84>
verts = ((0, 0), (0, 1), (1, 1), (1, 0))
visible = True
width = 1
window_extent = <Bbox object at 0x134acbcc>
x = 0
y = 0
zorder = 1

```

The docstrings for all of the classes also contain the `Artist` properties, so you can consult the interactive “help” or the [artist Module](#) for a listing of properties for a given object.

Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to now how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a `Text` instance, the width of a `Line2D`) although the containers also have some properties as well – for example the `Axes` `Artist` is a container that contains many of the primitives in your plot, but it also has properties like the `xscale` to control whether the xaxis is ‘linear’ or ‘log’. In this section we’ll review where the various container objects store the `Artists` that you want to get at.

Figure container

The top level container `Artist` is the `matplotlib.figure.Figure`, and it contains everything in the figure. The background of the figure is a `Rectangle` which is stored in `Figure.patch`. As you add subplots (`add_subplot()`) and axes (`add_axes()`) to the figure these will be appended to the `Figure.axes`. These are also returned by the methods that create them:

```

In [156]: fig = plt.figure()

In [157]: ax1 = fig.add_subplot(211)

In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])

In [159]: ax1
Out[159]: <matplotlib.axes.Subplot instance at 0xd54b26c>

In [160]: print fig.axes
[<matplotlib.axes.Subplot instance at 0xd54b26c>, <matplotlib.axes.Axes instance at
           0xd3f0b2c>]

```

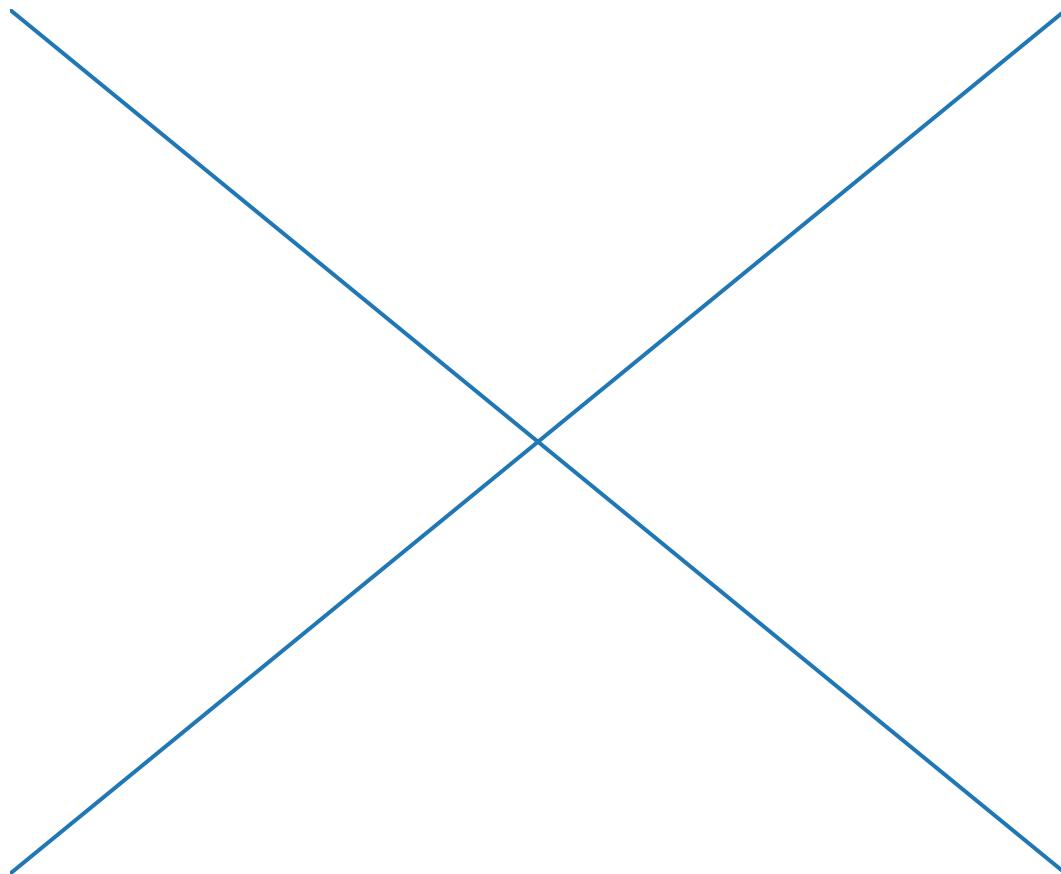
Because the figure maintains the concept of the “current axes” (see `Figure.gca` and `Figure.sca`) to support the pylab/pyplot state machine, you should not insert or remove axes directly from the axes list, but rather use the `add_subplot()` and `add_axes()` methods to insert, and the `delaxes()` method to delete. You are free however, to iterate over the list of axes or index into it to get access to `Axes` instances you want to customize. Here is an example which turns all the axes grids on:

```
for ax in fig.axes:  
    ax.grid(True)
```

The figure also has its own text, lines, patches and images, which you can use to add primitives directly. The default coordinate system for the `Figure` will simply be in pixels (which is not usually what you want) but you can control this by setting the `transform` property of the `Artist` you are adding to the figure.

More useful is “figure coordinates” where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of the figure which you can obtain by setting the `Artist` transform to `fig.transFigure`:

```
In [191]: fig = plt.figure()  
  
In [192]: l1 = matplotlib.lines.Line2D([0, 1], [0, 1],  
           transform=fig.transFigure, figure=fig)  
  
In [193]: l2 = matplotlib.lines.Line2D([0, 1], [1, 0],  
           transform=fig.transFigure, figure=fig)  
  
In [194]: fig.lines.extend([l1, l2])  
  
In [195]: fig.canvas.draw()
```



Here is a summary of the Artists the figure contains

Figure attribute	Description
axes	A list of Axes instances (includes Subplot)
patch	The Rectangle background
images	A list of FigureImages patches - useful for raw pixel display
legends	A list of Figure Legend instances (different from Axes.legends)
lines	A list of Figure Line2D instances (rarely used, see Axes.lines)
patches	A list of Figure patches (rarely used, see Axes.patches)
texts	A list Figure Text instances

Axes container

The `matplotlib.axes.Axes` is the center of the matplotlib universe – it contains the vast majority of all the `Artists` used in a figure with many helper methods to create and add these `Artists` to itself, as well as helper methods to access and customize the `Artists` it contains. Like the `Figure`, it contains a `Patch` patch which is a `Rectangle` for Cartesian coordinates and a `Circle` for polar coordinates; this patch determines the shape, background and border of the plotting region:

```
ax = fig.add_subplot(111)
rect = ax.patch # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical `plot()` and pass in arrays or lists of values, the method will create a `matplotlib.lines.Line2D()` instance, update the line with all the `Line2D` properties passed as keyword arguments, add the line to the `Axes.lines` container, and returns it to you:

```
In [213]: x, y = np.random.rand(2, 100)
```

```
In [214]: line, = ax.plot(x, y, '- ', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the `line` variable. The line has been added to the `Axes.lines` list:

```
In [229]: print ax.lines
[<matplotlib.lines.Line2D instance at 0xd378b0c>]
```

Similarly, methods that create patches, like `bar()` creates a list of rectangles, will add the patches to the `Axes.patches` list:

```
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50, facecolor='yellow')
```

```
In [234]: rectangles
Out[234]: <a list of 50 Patch objects>
```

```
In [235]: print len(ax.patches)
```

You should not add objects directly to the `Axes.lines` or `Axes.patches` lists unless you know exactly what you are doing, because the `Axes` needs to do a few things when it creates and adds an object. It sets the figure and axes property of the `Artist`, as well as the default `Axes` transformation (unless a transformation is set). It also inspects the data contained in the `Artist` to update the data structures controlling auto-scaling,

so that the view limits can be adjusted to contain the plotted data. You can, nonetheless, create objects yourself and add them directly to the `Axes` using helper methods like `add_line()` and `add_patch()`. Here is an annotated interactive session illustrating what is going on:

```
In [261]: fig = plt.figure()

In [262]: ax = fig.add_subplot(111)

# create a rectangle instance
In [263]: rect = matplotlib.patches.Rectangle( (1,1), width=5, height=12)

# by default the axes instance is None
In [264]: print rect.get_axes()
None

# and the transformation instance is set to the "identity transform"
In [265]: print rect.get_transform()
<Affine object at 0x13695544>

# now we add the Rectangle to the Axes
In [266]: ax.add_patch(rect)

# and notice that the ax.add_patch method has set the axes
# instance
In [267]: print rect.get_axes()
Axes(0.125,0.1;0.775x0.8)

# and the transformation has been set too
In [268]: print rect.get_transform()
<Affine object at 0x15009ca4>

# the default axes transformation is ax.transData
In [269]: print ax.transData
<Affine object at 0x15009ca4>

# notice that the xlims of the Axes have not been changed
In [270]: print ax.get_xlim()
(0.0, 1.0)

# but the data limits have been updated to encompass the rectangle
In [271]: print ax.dataLim.bounds
(1.0, 1.0, 5.0, 12.0)

# we can manually invoke the auto-scaling machinery
In [272]: ax.autoscale_view()

# and now the xlim are updated to encompass the rectangle
In [273]: print ax.get_xlim()
(1.0, 6.0)

# we have to manually force a figure draw
In [274]: ax.figure.canvas.draw()
```

There are many, many `Axes` helper methods for creating primitive `Artists` and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of `Artist` they create, and where they store them

Helper method	Artist	Container
<code>ax.annotate</code> - text annotations	<code>Annotate</code>	<code>ax.texts</code>
<code>ax.bar</code> - bar charts	<code>Rectangle</code>	<code>ax.patches</code>
<code>ax.errorbar</code> - error bar plots	<code>Line2D</code> and <code>Rectangle</code>	<code>ax.lines</code> and <code>ax.patches</code>
<code>ax.fill</code> - shared area	<code>Polygon</code>	<code>ax.patches</code>
<code>ax.hist</code> - histograms	<code>Rectangle</code>	<code>ax.patches</code>
<code>ax.imshow</code> - image data	<code>AxesImage</code>	<code>ax.images</code>
<code>ax.legend</code> - axes legends	<code>Legend</code>	<code>ax.legends</code>
<code>ax.plot</code> - xy plots	<code>Line2D</code>	<code>ax.lines</code>
<code>ax.scatter</code> - scatter charts	<code>PolygonCollection</code>	<code>ax.collections</code>
<code>ax.text</code> - text	<code>Text</code>	<code>ax.texts</code>

In addition to all of these `Artists`, the `Axes` contains two important `Artist` containers: the `XAxis` and `YAxis`, which handle the drawing of the ticks and labels. These are stored as instance variables `xaxis` and `yaxis`. The `XAxis` and `YAxis` containers will be detailed below, but note that the `Axes` contains many helper methods which forward calls on to the `Axis` instances so you often do not need to work with them directly unless you want to. For example, you can set the font size of the `XAxis` ticklabels using the `Axes` helper method:

```
for label in ax.get_xticklabels():
    label.set_color('orange')
```

Below is a summary of the `Artists` that the `Axes` contains

Axes attribute	Description
<code>artists</code>	A list of <code>Artist</code> instances
<code>patch</code>	<code>Rectangle</code> instance for <code>Axes</code> background
<code>collections</code>	A list of <code>Collection</code> instances
<code>images</code>	A list of <code>AxesImage</code>
<code>legends</code>	A list of <code>Legend</code> instances
<code>lines</code>	A list of <code>Line2D</code> instances
<code>patches</code>	A list of <code>Patch</code> instances
<code>texts</code>	A list of <code>Text</code> instances
<code>xaxis</code>	<code>matplotlib.axis.XAxis</code> instance
<code>yaxis</code>	<code>matplotlib.axis.YAxis</code> instance

Axis containers

The `matplotlib.axis.Axis` instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The `Axis` also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the `Locator` and `Formatter` instances which control where the ticks are placed and how they are represented as strings.

Each `Axis` object contains a `label` attribute (this is what `pylab` modifies in calls to `xlabel()` and

`ylabel()` as well as a list of major and minor ticks. The ticks are `XTick` and `YTick` instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor ticks through their accessor methods `get_major_ticks()` and `get_minor_ticks()`. Although the ticks contain all the primitives and will be covered below, the `Axis` methods contain accessor methods to return the tick lines, tick labels, tick locations etc.:

```
In [285]: axis = ax.xaxis

In [286]: axis.get_ticklocs()
Out[286]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

In [287]: axis.get_ticklabels()
Out[287]: <a list of 10 Text major ticklabel objects>

# note there are twice as many ticklines as labels because by
# default there are tick lines at the top and bottom but only tick
# labels below the xaxis; this can be customized
In [288]: axis.get_ticklines()
Out[288]: <a list of 20 Line2D ticklines objects>

# by default you get the major ticks back
In [291]: axis.get_ticklines()
Out[291]: <a list of 20 Line2D ticklines objects>

# but you can also ask for the minor ticks
In [292]: axis.get_ticklines(minor=True)
Out[292]: <a list of 0 Line2D ticklines objects>
```

Here is a summary of some of the useful accessor methods of the `Axis` (these have corresponding setters where useful, such as `set_major_formatter`)

Accessor method	Description
<code>get_scale</code>	The scale of the axis, e.g., ‘log’ or ‘linear’
<code>get_view_interval</code>	The interval instance of the axis view limits
<code>get_data_interval</code>	The interval instance of the axis data limits
<code>get_gridlines</code>	A list of grid lines for the Axis
<code>get_label</code>	The axis label - a <code>Text</code> instance
<code>get_ticklabels</code>	A list of <code>Text</code> instances - keyword <code>minor=True False</code>
<code>get_ticklines</code>	A list of <code>Line2D</code> instances - keyword <code>minor=True False</code>
<code>get_ticklocs</code>	A list of Tick locations - keyword <code>minor=True False</code>
<code>get_major_locator</code>	The <code>matplotlib.ticker.Locator</code> instance for major ticks
<code>get_major_formatter</code>	The <code>matplotlib.ticker.Formatter</code> instance for major ticks
<code>get_minor_locator</code>	The <code>matplotlib.ticker.Locator</code> instance for minor ticks
<code>get_minor_formatter</code>	The <code>matplotlib.ticker.Formatter</code> instance for minor ticks
<code>get_major_ticks</code>	A list of <code>Tick</code> instances for major ticks
<code>get_minor_ticks</code>	A list of <code>Tick</code> instances for minor ticks
<code>grid</code>	Turn the grid on or off for the major or minor ticks

Here is an example, not recommended for its beauty, which customizes the axes and tick properties

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

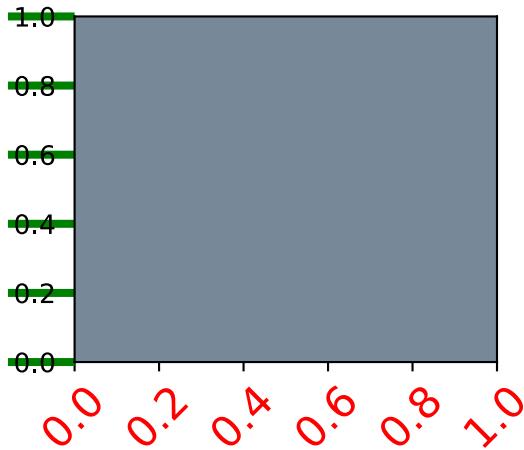
# plt.figure creates a matplotlib.figure.Figure instance
fig = plt.figure()
rect = fig.patch # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategray')

for label in ax1.xaxis.get_ticklabels():
    # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():
    # line is a Line2D instance
    line.set_color('green')
    line.set_markersize(25)
    line.set_markeredgewidth(3)

plt.show()
```



Tick containers

The `matplotlib.axis.Tick` is the final container object in our descent from the `Figure` to the `Axes` to the `Axis` to the `Tick`. The Tick contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the Tick. In addition, there are boolean variables that determine whether the upper labels and ticks are on for the x-axis and whether the right labels and ticks are on for the y-axis.

Tick attribute	Description
<code>tick1line</code>	Line2D instance
<code>tick2line</code>	Line2D instance
<code>gridline</code>	Line2D instance
<code>label1</code>	Text instance
<code>label2</code>	Text instance
<code>gridOn</code>	boolean which determines whether to draw the tickline
<code>tick1On</code>	boolean which determines whether to draw the 1st tickline
<code>tick2On</code>	boolean which determines whether to draw the 2nd tickline
<code>label1On</code>	boolean which determines whether to draw tick label
<code>label2On</code>	boolean which determines whether to draw tick label

Here is an example which sets the formatter for the right side ticks with dollar signs and colors them green on the right side of the yaxis

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

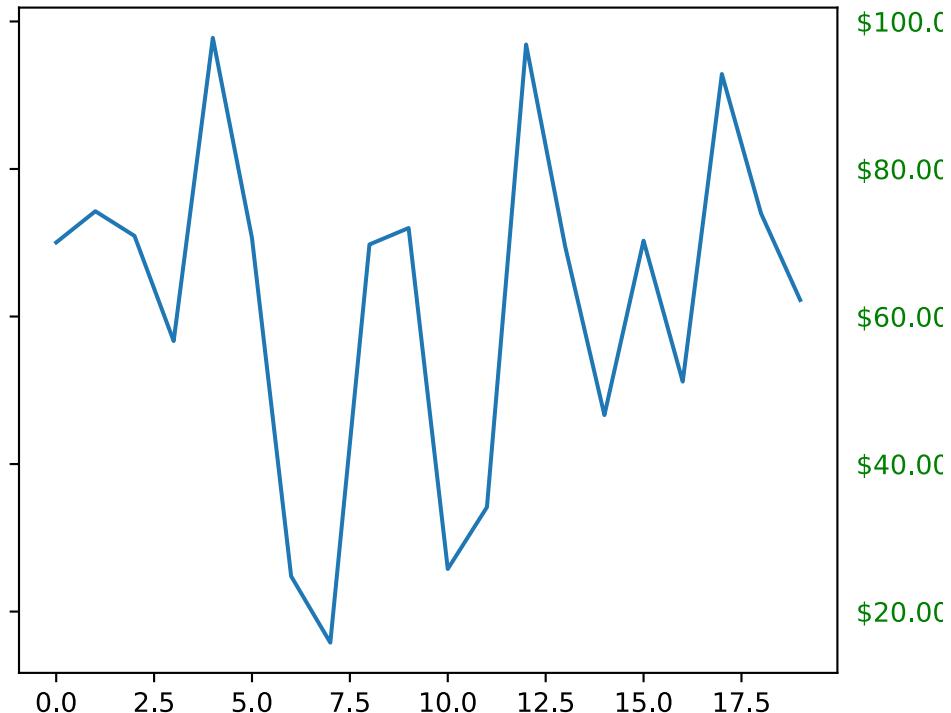
# Fixing random state for reproducibility
np.random.seed(19680801)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(100*np.random.rand(20))

formatter = ticker.FormatStrFormatter('$%1.2f')
ax.yaxis.set_major_formatter(formatter)

for tick in ax.yaxis.get_major_ticks():
    tick.label1On = False
    tick.label2On = True
    tick.label2.set_color('green')

plt.show()
```



3.2.2 Legend guide

This legend guide is an extension of the documentation available at [`legend\(\)`](#) - please ensure you are familiar with contents of that documentation before proceeding with this guide.

This guide makes use of some common terms, which are documented here for clarity:

legend entry A legend is made up of one or more legend entries. An entry is made up of exactly one key and one label.

legend key The colored/patterned marker to the left of each legend label.

legend label The text which describes the handle represented by the key.

legend handle The original object which is used to generate an appropriate entry in the legend.

Controlling the legend entries

Calling [`legend\(\)`](#) with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:

```
handles, labels = ax.get_legend_handles_labels()  
ax.legend(handles, labels)
```

The [`get_legend_handles_labels\(\)`](#) function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend, at which point a “proxy” will have to be created (see [`Creating artists specifically for adding to the legend \(aka. Proxy artists\)`](#) for further details).

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to [`legend\(\)`](#):

```
line_up, = plt.plot([1,2,3], label='Line 2')  
line_down, = plt.plot([3,2,1], label='Line 1')  
plt.legend(handles=[line_up, line_down])
```

In some cases, it is not possible to set the label of the handle, so it is possible to pass through the list of labels to [`legend\(\)`](#):

```
line_up, = plt.plot([1,2,3], label='Line 2')  
line_down, = plt.plot([3,2,1], label='Line 1')  
plt.legend([line_up, line_down], ['Line Up', 'Line Down'])
```

Creating artists specifically for adding to the legend (aka. Proxy artists)

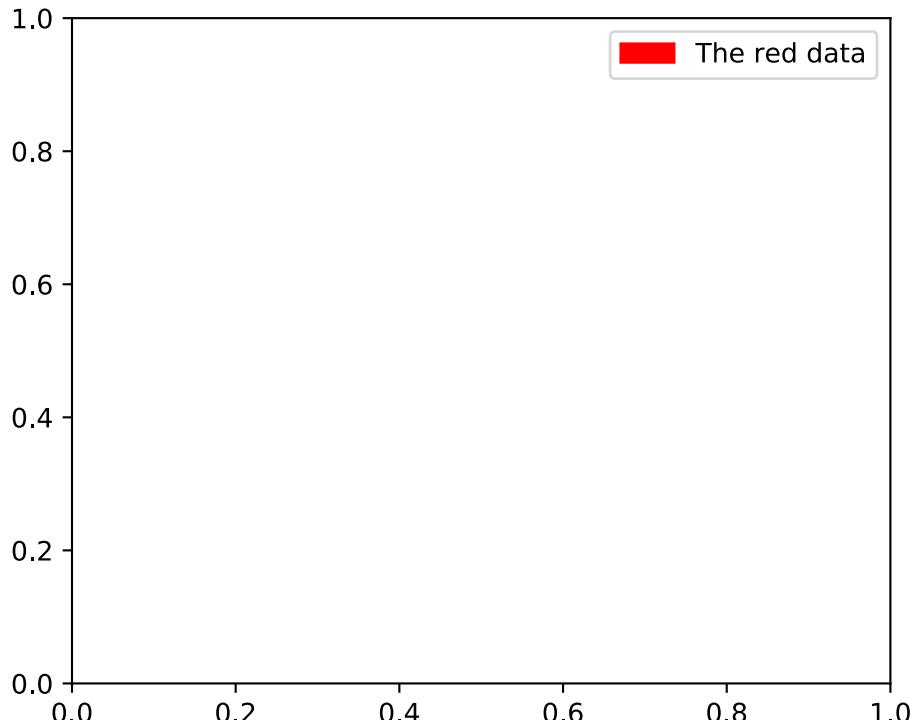
Not all handles can be turned into legend entries automatically, so it is often necessary to create an artist which *can*. Legend handles don't have to exist on the Figure or Axes in order to be used.

Suppose we wanted to create a legend which has an entry for some data which is represented by a red color:

```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

red_patch = mpatches.Patch(color='red', label='The red data')
plt.legend(handles=[red_patch])

plt.show()
```

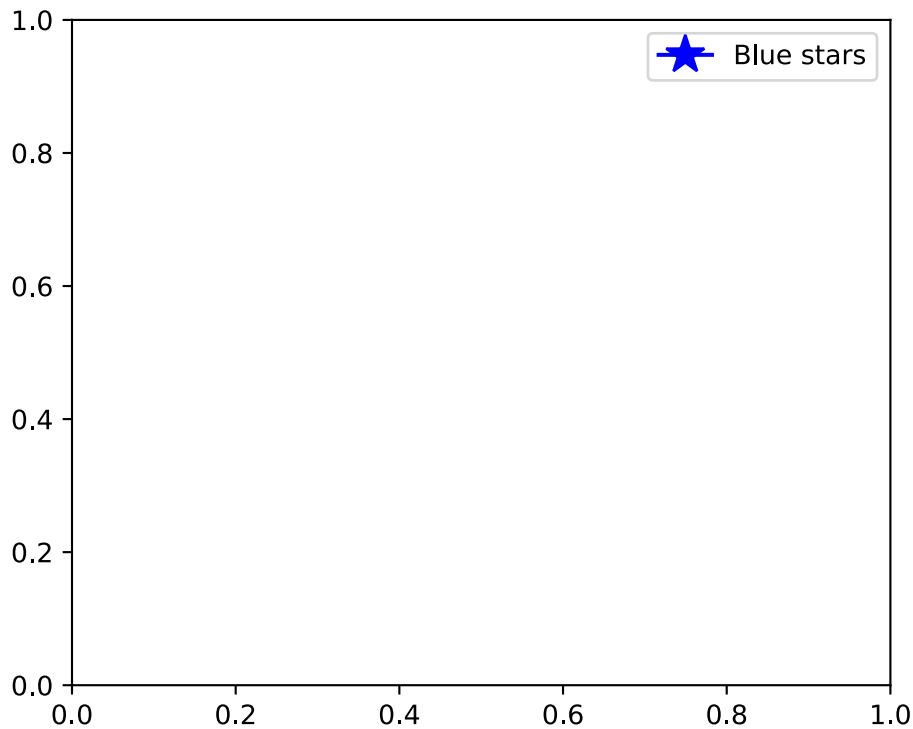


There are many supported legend handles, instead of creating a patch of color we could have created a line with a marker:

```
import matplotlib.lines as mlines
import matplotlib.pyplot as plt

blue_line = mlines.Line2D([], [], color='blue', marker='*',
                        markersize=15, label='Blue stars')
plt.legend(handles=[blue_line])

plt.show()
```



Legend location

The location of the legend can be specified by the keyword argument *loc*. Please see the documentation at [legend\(\)](#) for more details.

The `bbox_to_anchor` keyword gives a great degree of control for manual legend placement. For example, if you want your axes legend located at the figure's top right-hand corner instead of the axes' corner, simply specify the corner's location, and the coordinate system of that location:

```
plt.legend(bbox_to_anchor=(1, 1),  
          bbox_transform=plt.gcf().transFigure)
```

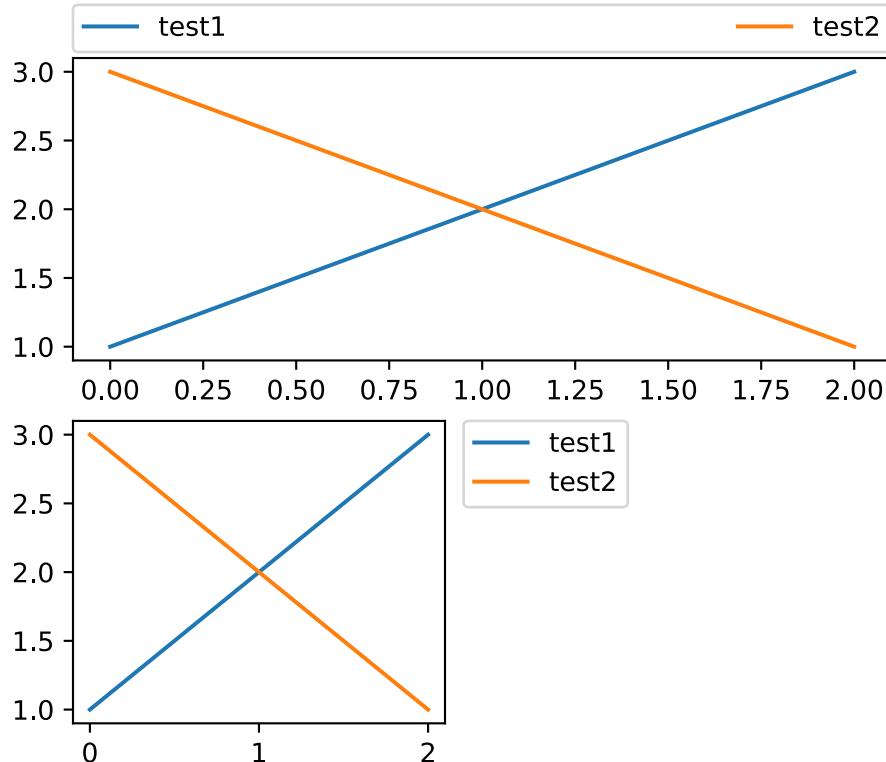
More examples of custom legend placement:

```
import matplotlib.pyplot as plt  
  
plt.subplot(211)  
plt.plot([1,2,3], label="test1")  
plt.plot([3,2,1], label="test2")  
# Place a legend above this subplot, expanding itself to  
# fully use the given bounding box.  
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,  
          ncol=2, mode="expand", borderaxespad=0.)
```

```

plt.subplot(223)
plt.plot([1,2,3], label="test1")
plt.plot([3,2,1], label="test2")
# Place a legend to the right of this smaller subplot.
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()

```



Multiple legends on the same Axes

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach to doing this might be to call the `legend()` function multiple times, you will find that only one legend ever exists on the Axes. This has been done so that it is possible to call `legend()` repeatedly to update the legend to the latest handles on the Axes, so to persist old legend instances, we must add them manually to the Axes:

```

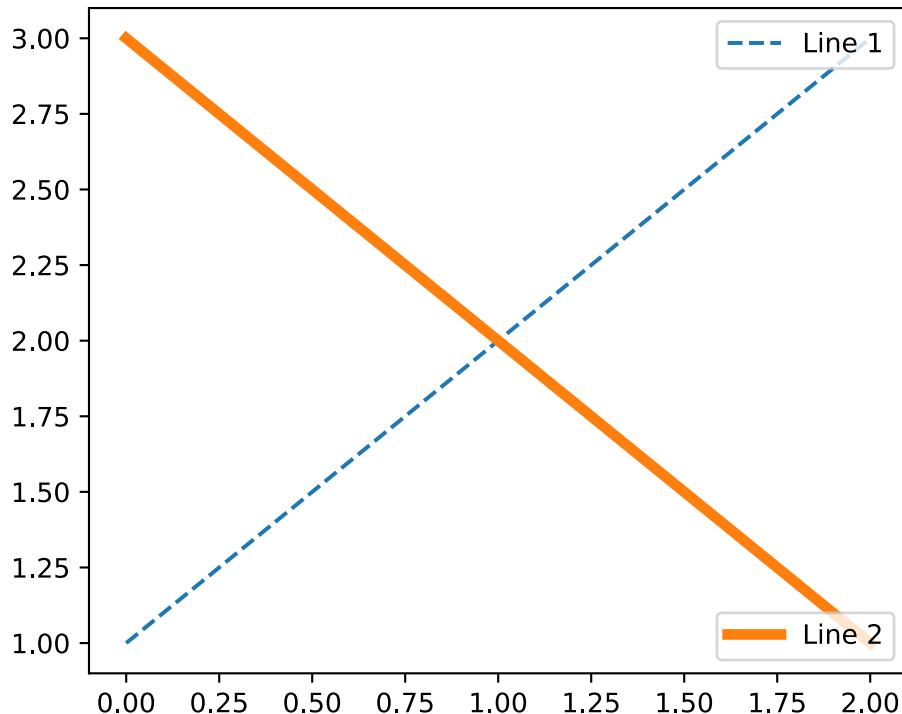
import matplotlib.pyplot as plt

line1, = plt.plot([1,2,3], label="Line 1", linestyle='--')
line2, = plt.plot([3,2,1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = plt.legend(handles=[line1], loc=1)

```

```
# Add the legend manually to the current Axes.  
ax = plt.gca().add_artist(first_legend)  
  
# Create another legend for the second line.  
plt.legend(handles=[line2], loc=4)  
  
plt.show()
```



Legend Handlers

In order to create legend entries, handles are given as an argument to an appropriate `HandlerBase` subclass. The choice of handler subclass is determined by the following rules:

1. Update `get_legend_handler_map()` with the value in the `handler_map` keyword.
2. Check if the handle is in the newly created `handler_map`.
3. Check if the type of handle is in the newly created `handler_map`.
4. Check if any of the types in the handle's mro is in the newly created `handler_map`.

For completeness, this logic is mostly implemented in `get_legend_handler()`.

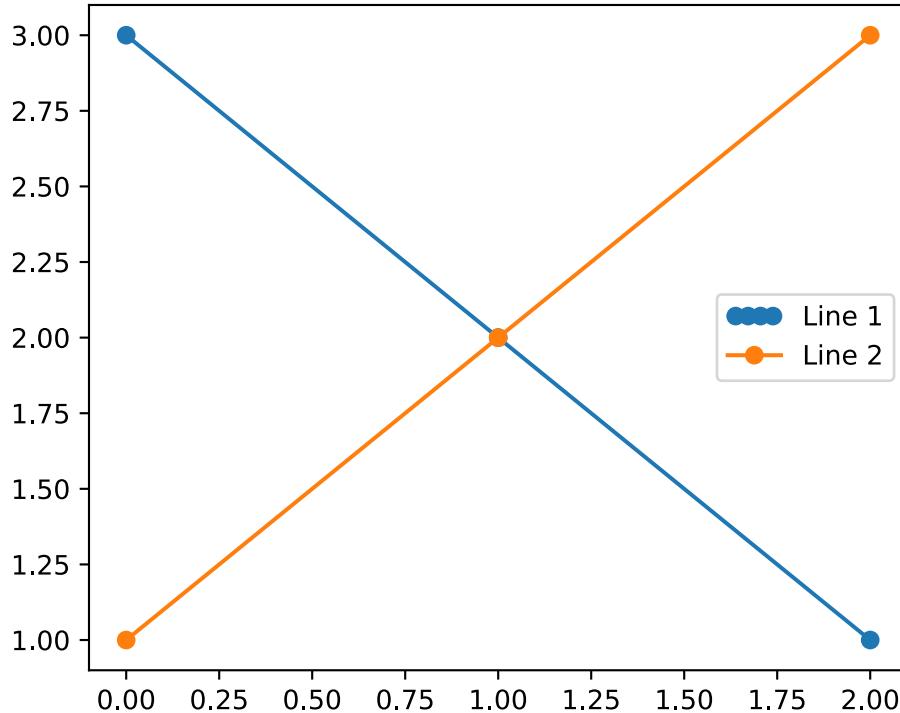
All of this flexibility means that we have the necessary hooks to implement custom handlers for our own type of legend key.

The simplest example of using custom handlers is to instantiate one of the existing `HandlerBase` subclasses. For the sake of simplicity, let's choose `matplotlib.legend_handler.HandlerLine2D` which accepts a `numpoints` argument (note `numpoints` is a keyword on the `legend()` function for convenience). We can then pass the mapping of instance to Handler as a keyword to `legend`.

```
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D

line1, = plt.plot([3,2,1], marker='o', label='Line 1')
line2, = plt.plot([1,2,3], marker='o', label='Line 2')

plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
```



As you can see, “Line 1” now has 4 marker points, where “Line 2” has 2 (the default). Try the above code, only change the map’s key from `line1` to `type(line1)`. Notice how now both `Line2D` instances get 4 markers.

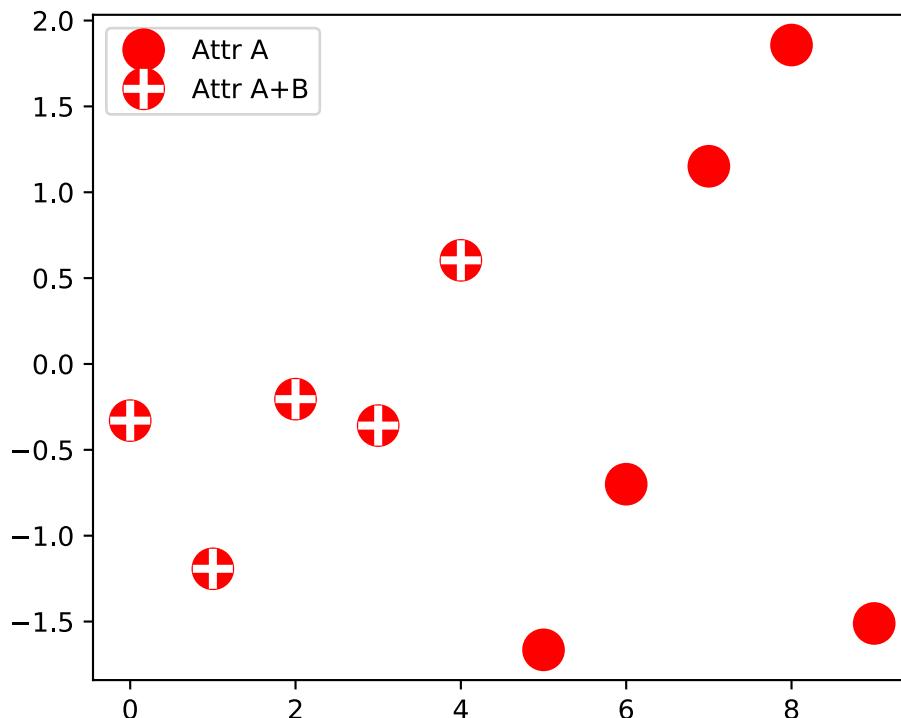
Along with handlers for complex plot types such as errorbars, stem plots and histograms, the default `handler_map` has a special tuple handler (`HandlerTuple`) which simply plots the handles on top of one another for each item in the given tuple. The following example demonstrates combining two legend keys on top of one another:

```
import matplotlib.pyplot as plt
from numpy.random import randn

z = randn(10)

red_dot, = plt.plot(z, "ro", markersize=15)
# Put a white cross over some of the data.
white_cross, = plt.plot(z[:5], "w+", markeredgewidth=3, markersize=15)

plt.legend([red_dot, (red_dot, white_cross)], ["Attr A", "Attr A+B"])
```



Implementing a custom legend handler

A custom handler can be implemented to turn any handle into a legend key (handles don't necessarily need to be matplotlib artists). The handler must implement a “legend_artist” method which returns a single artist for the legend to use. Signature details about the “legend_artist” are documented at [legend_artist\(\)](#).

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

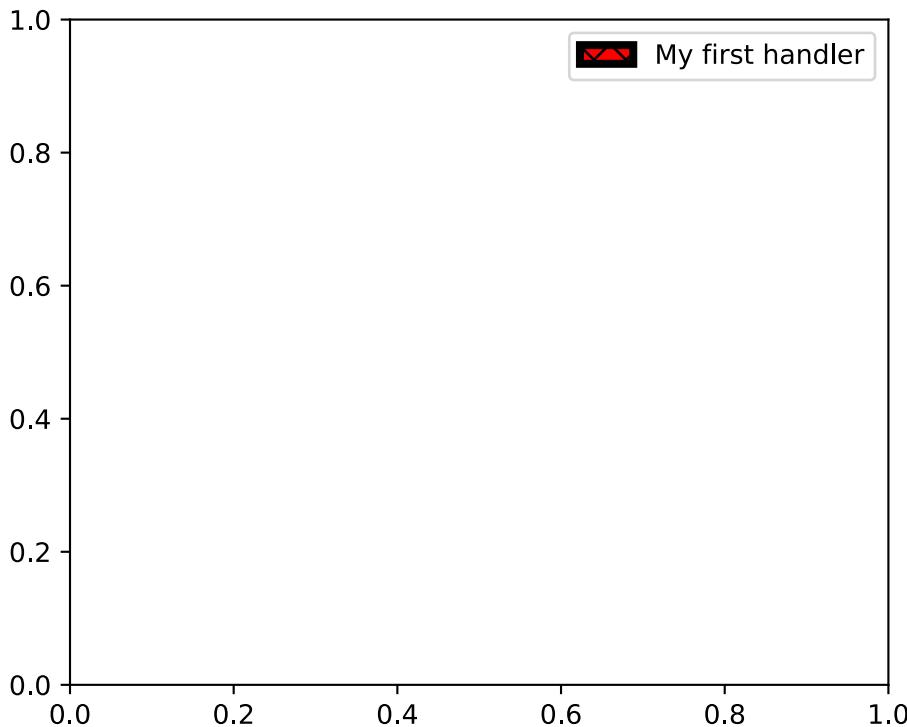
class AnyObject(object):
    pass
```

```

class AnyObjectHandler(object):
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                                   edgecolor='black', hatch='xx', lw=3,
                                   transform=handlebox.get_transform())
        handlebox.add_artist(patch)
        return patch

plt.legend([AnyObject()], ['My first handler'],
          handler_map={AnyObject: AnyObjectHandler()})

```



Alternatively, had we wanted to globally accept `AnyObject` instances without needing to manually set the `handler_map` keyword all the time, we could have registered the new handler with:

```

from matplotlib.legend import Legend
Legend.update_default_handler_map({AnyObject: AnyObjectHandler()})

```

Whilst the power here is clear, remember that there are already many handlers implemented and what you want to achieve may already be easily possible with existing classes. For example, to produce elliptical legend keys, rather than rectangular ones:

```

from matplotlib.legend_handler import HandlerPatch
import matplotlib.pyplot as plt

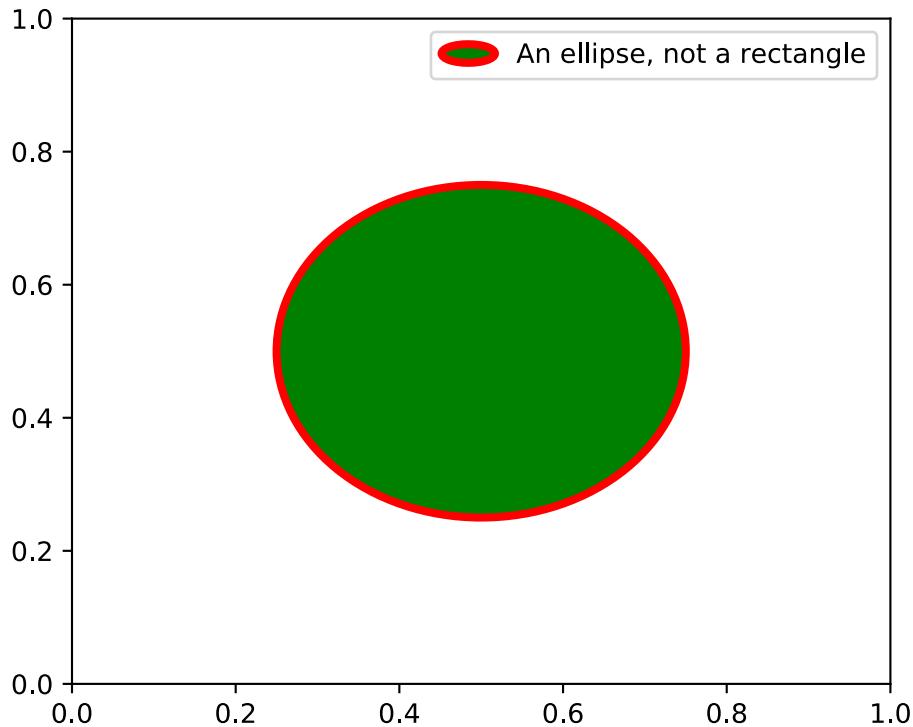
```

```
import matplotlib.patches as mpatches

class HandlerEllipse(HandlerPatch):
    def create_artists(self, legend, orig_handle,
                       xdescent, ydescent, width, height, fontsize, trans):
        center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
        p = mpatches.Ellipse(xy=center, width=width + xdescent,
                             height=height + ydescent)
        self.update_prop(p, orig_handle, legend)
        p.set_transform(trans)
        return [p]

c = mpatches.Circle((0.5, 0.5), 0.25, facecolor="green",
                     edgecolor="red", linewidth=3)
plt.gca().add_patch(c)

plt.legend([c], ["An ellipse, not a rectangle"],
           handler_map={mpatches.Circle: HandlerEllipse()})
```



Known examples of using legend

Here is a non-exhaustive list of the examples available involving legend being used in various ways:

- *lines_bars_and_markers example code: scatter_with_legend.py*
- *api example code: legend_demo.py*
- *pylab_examples example code: contourf_hatching.py*
- *pylab_examples example code: figlegend_demo.py*
- *pylab_examples example code: scatter_symbol.py*

3.3 Advanced

3.3.1 Transformations Tutorial

Like any graphics packages, matplotlib is built on top of a transformation framework to easily move between coordinate systems, the userland data coordinate system, the `axes` coordinate system, the `figure` coordinate system, and the `display` coordinate system. In 95% of your plotting, you won't need to think about this, as it happens under the hood, but as you push the limits of custom figure generation, it helps to have an understanding of these objects so you can reuse the existing transformations matplotlib makes available to you, or create your own (see [matplotlib.transforms](#)). The table below summarizes the existing coordinate systems, the transformation object you should use to work in that coordinate system, and the description of that system. In the `Transformation Object` column, `ax` is a `Axes` instance, and `fig` is a `Figure` instance.

Co- or- di- nate	Transfor- mation Object	Description
<code>data</code>	<code>ax. transData</code>	The userland data coordinate system, controlled by the <code>xlim</code> and <code>ylim</code>
<code>axes</code>	<code>ax. transAxes</code>	The coordinate system of the <code>Axes</code> ; (0,0) is bottom left of the axes, and (1,1) is top right of the axes.
<code>figure</code>	<code>fig. transFigure</code>	The coordinate system of the <code>Figure</code> ; (0,0) is bottom left of the figure, and (1,1) is top right of the figure.
<code>display</code>	<code>None</code>	This is the pixel coordinate system of the display; (0,0) is the bottom left of the display, and (width, height) is the top right of the display in pixels. Alternatively, the identity transform (matplotlib.transforms.IdentityTransform()) may be used instead of None.

All of the transformation objects in the table above take inputs in their coordinate system, and transform the input to the `display` coordinate system. That is why the `display` coordinate system has `None` for the `Transformation Object` column – it already is in display coordinates. The transformations also know how to invert themselves, to go from `display` back to the native coordinate system. This is particularly useful when processing events from the user interface, which typically occur in display space, and you want to know where the mouse click or key-press occurred in your data coordinate system.

Data coordinates

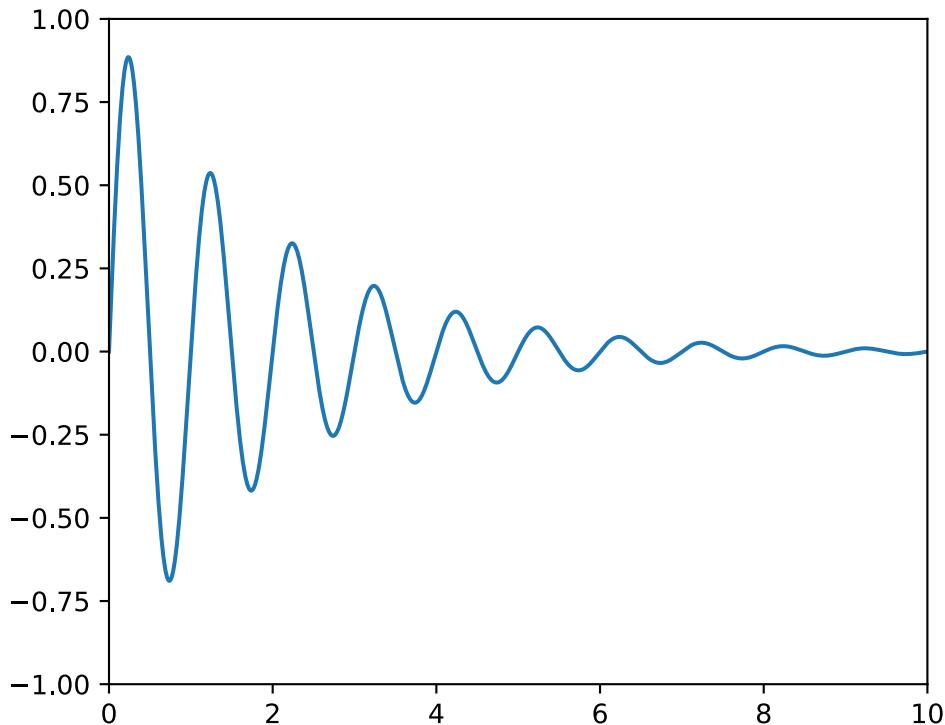
Let's start with the most commonly used coordinate, the `data` coordinate system. Whenever you add data to the axes, matplotlib updates the datalimits, most commonly updated with the `set_xlim()` and `set_ylim()` methods. For example, in the figure below, the data limits stretch from 0 to 10 on the x-axis, and -1 to 1 on the y-axis.

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

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

plt.show()
```



You can use the `ax.transData` instance to transform from your data to your display coordinate system, either a single point or a sequence of points as shown below:

```
In [14]: type(ax.transData)
Out[14]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [15]: ax.transData.transform((5, 0))
Out[15]: array([ 335.175,  247.    ])

In [16]: ax.transData.transform([(5, 0), (1, 2)])
Out[16]:
array([[ 335.175,  247.    ],
       [ 132.435,  642.2 ]])
```

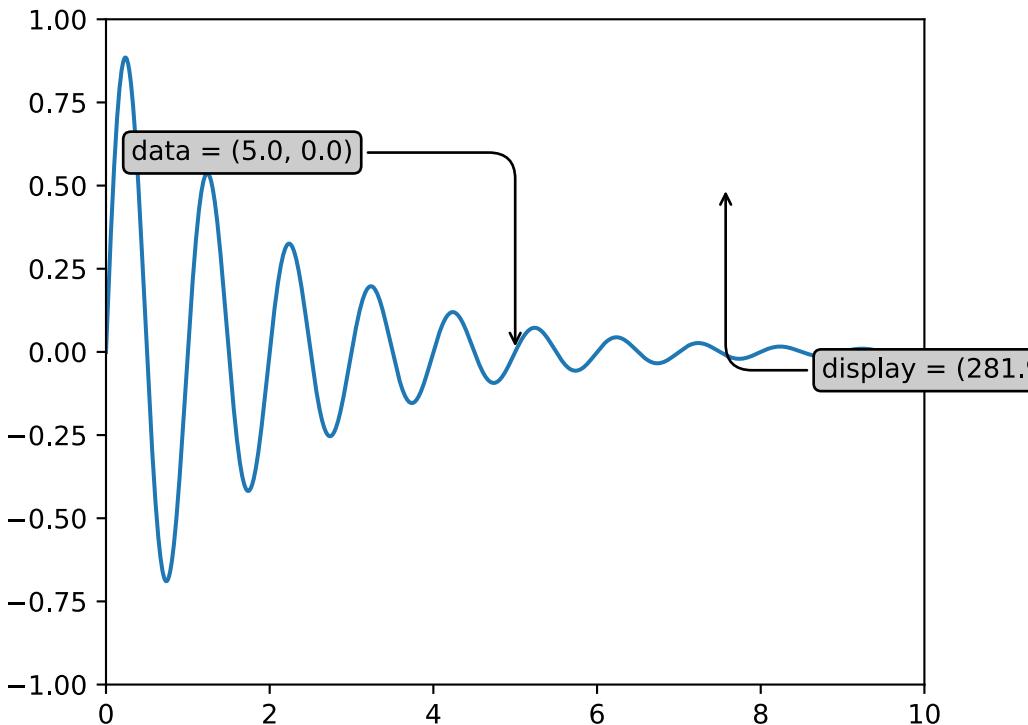
You can use the `inverted()` method to create a transform which will take you from display to data coordinates:

```
In [41]: inv = ax.transData.inverted()

In [42]: type(inv)
Out[42]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [43]: inv.transform((335.175, 247.))
Out[43]: array([ 5.,  0.])
```

If you are typing along with this tutorial, the exact values of the display coordinates may differ if you have a different window size or dpi setting. Likewise, in the figure below, the display labeled points are probably not the same as in the ipython session because the documentation figure size defaults are different.



Note: If you run the source code in the example above in a GUI backend, you may also find that the two arrows for the data and display annotations do not point to exactly the same point. This is because the display point was computed before the figure was displayed, and the GUI backend may slightly resize the figure when it is created. The effect is more pronounced if you resize the figure yourself. This is one good reason why you rarely want to work in display space, but you can connect to the 'on_draw' [Event](#) to update figure coordinates on figure draws; see [Event handling and picking](#).

When you change the x or y limits of your axes, the data limits are updated so the transformation yields a new display point. Note that when we just change the ylim, only the y-display coordinate is altered, and when we change the xlim too, both are altered. More on this later when we talk about the [Bbox](#).

```
In [54]: ax.transData.transform((5, 0))
Out[54]: array([ 335.175,   247.    ])

In [55]: ax.set_ylim(-1,2)
Out[55]: (-1, 2)

In [56]: ax.transData.transform((5, 0))
Out[56]: array([ 335.175      ,  181.13333333])

In [57]: ax.set_xlim(10,20)
Out[57]: (10, 20)

In [58]: ax.transData.transform((5, 0))
Out[58]: array([-171.675      ,  181.13333333])
```

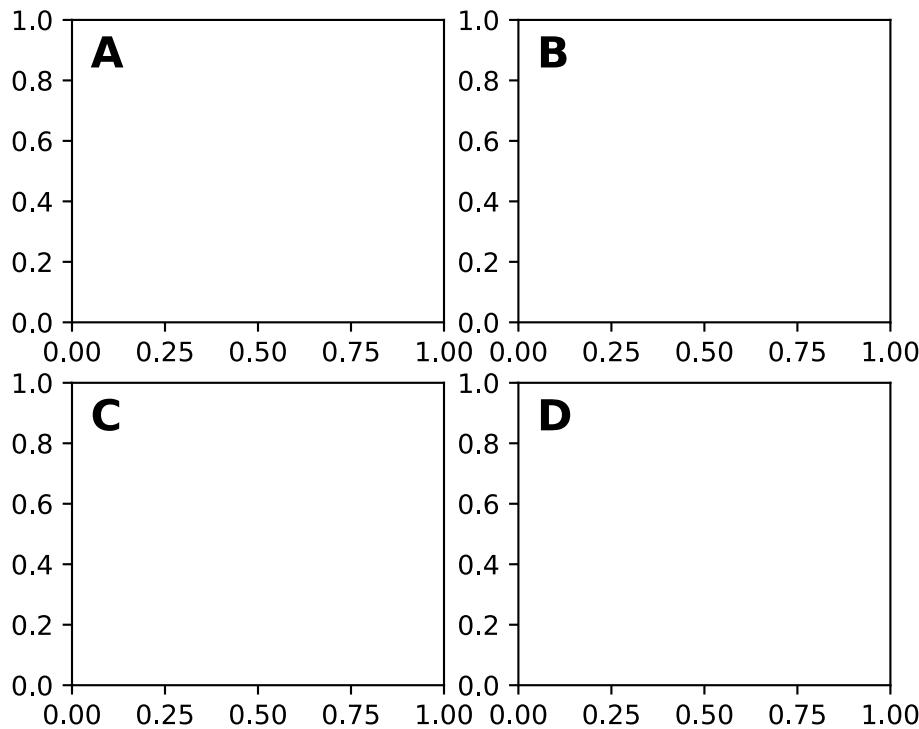
Axes coordinates

After the data coordinate system, axes is probably the second most useful coordinate system. Here the point (0,0) is the bottom left of your axes or subplot, (0.5, 0.5) is the center, and (1.0, 1.0) is the top right. You can also refer to points outside the range, so (-0.1, 1.1) is to the left and above your axes. This coordinate system is extremely useful when placing text in your axes, because you often want a text bubble in a fixed, location, e.g., the upper left of the axes pane, and have that location remain fixed when you pan or zoom. Here is a simple example that creates four panels and labels them 'A', 'B', 'C', 'D' as you often see in journals.

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

fig = plt.figure()
for i, label in enumerate(['A', 'B', 'C', 'D']):
    ax = fig.add_subplot(2,2,i+1)
    ax.text(0.05, 0.95, label, transform=ax.transAxes,
            fontsize=16, fontweight='bold', va='top')

plt.show()
```

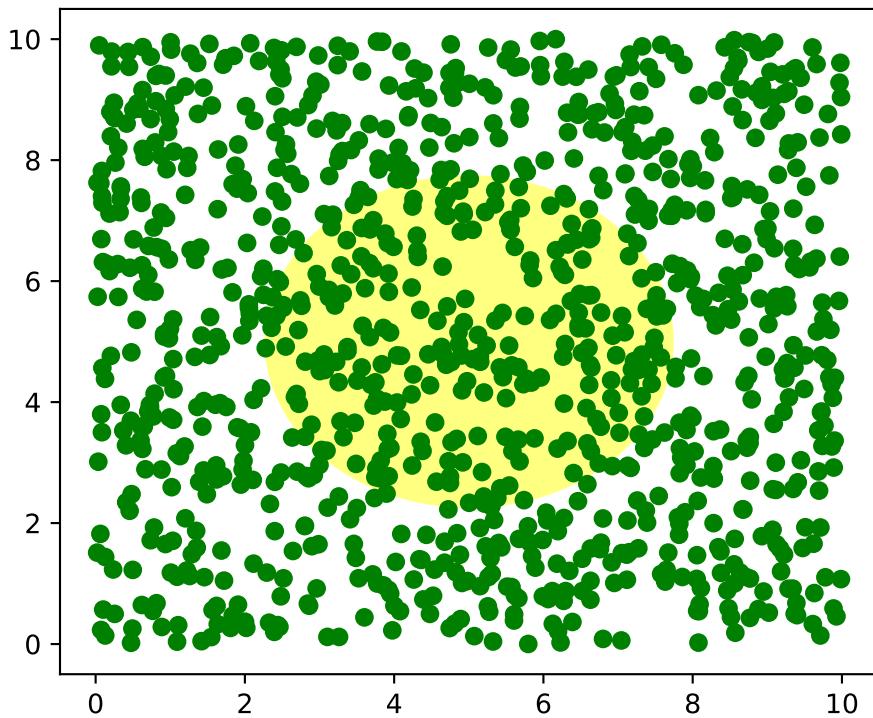


You can also make lines or patches in the axes coordinate system, but this is less useful in my experience than using `ax.transAxes` for placing text. Nonetheless, here is a silly example which plots some random dots in data space, and overlays a semi-transparent `Circle` centered in the middle of the axes with a radius one quarter of the axes – if your axes does not preserve aspect ratio (see `set_aspect()`), this will look like an ellipse. Use the pan/zoom tool to move around, or manually change the data `xlim` and `ylim`, and you will see the data move, but the circle will remain fixed because it is not in data coordinates and will always remain at the center of the axes.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
fig = plt.figure()
ax = fig.add_subplot(111)
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y, 'go') # plot some data in data coordinates

circ = patches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                      facecolor='yellow', alpha=0.5)
ax.add_patch(circ)

plt.show()
```



Blended transformations

Drawing in blended coordinate spaces which mix axes with data coordinates is extremely useful, for example to create a horizontal span which highlights some region of the y-data but spans across the x-axis regardless of the data limits, pan or zoom level, etc. In fact these blended lines and spans are so useful, we have built in functions to make them easy to plot (see `axhline()`, `axvline()`, `axhspan()`, `axvspan()`) but for didactic purposes we will implement the horizontal span here using a blended transformation. This trick only works for separable transformations, like you see in normal Cartesian coordinate systems, but not on inseparable transformations like the `PolarTransform`.

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

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.random.randn(1000)

ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \vee \dots \vee \sigma=2$', fontsize=16)

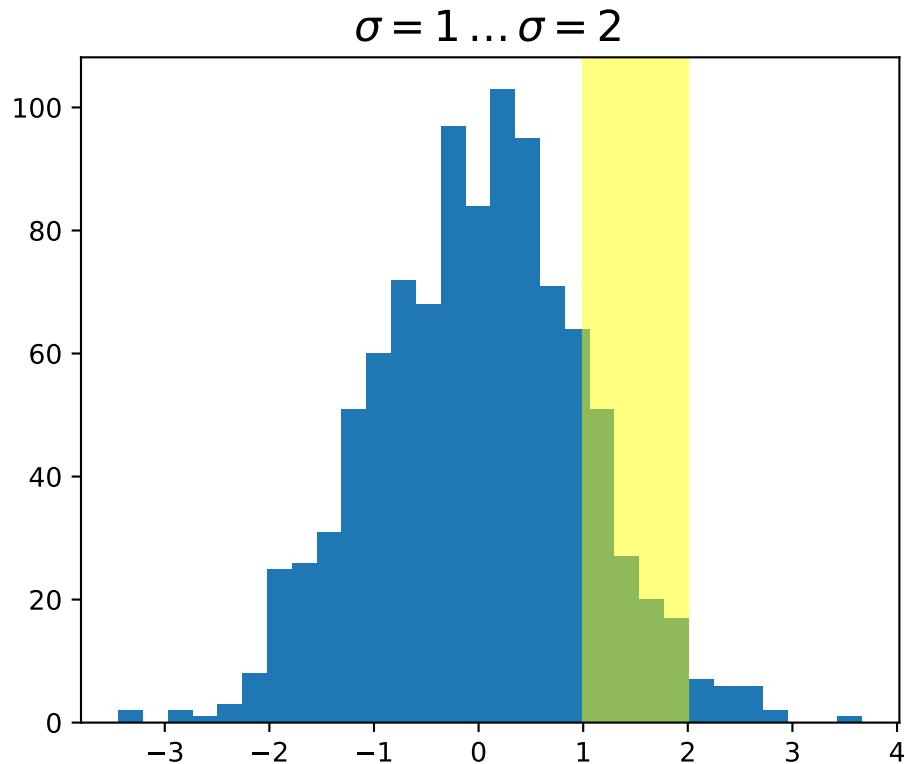
# the x coords of this transformation are data, and the
```

```
# y coord are axes
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)

# highlight the 1..2 stddev region with a span.
# We want x to be in data coordinates and y to
# span from 0..1 in axes coords
rect = patches.Rectangle((1, 0), width=1, height=1,
                        transform=trans, color='yellow',
                        alpha=0.5)

ax.add_patch(rect)

plt.show()
```



Note: The blended transformations where x is in data coords and y in axes coordinates is so useful that we have helper methods to return the versions mpl uses internally for drawing ticks, ticklabels, etc. The methods are `matplotlib.axes.Axes.get_xaxis_transform()` and `matplotlib.axes.Axes.get_yaxis_transform()`. So in the example above, the call to `blended_transform_factory()` can be replaced by `get_xaxis_transform`:

```
trans = ax.get_xaxis_transform()
```

Using offset transforms to create a shadow effect

One use of transformations is to create a new transformation that is offset from another transformation, e.g., to place one object shifted a bit relative to another object. Typically you want the shift to be in some physical dimension, like points or inches rather than in data coordinates, so that the shift effect is constant at different zoom levels and dpi settings.

One use for an offset is to create a shadow effect, where you draw one object identical to the first just to the right of it, and just below it, adjusting the zorder to make sure the shadow is drawn first and then the object it is shadowing above it. The transforms module has a helper transformation `ScaledTranslation`. It is instantiated with:

```
trans = ScaledTranslation(xt, yt, scale_trans)
```

where `xt` and `yt` are the translation offsets, and `scale_trans` is a transformation which scales `xt` and `yt` at transformation time before applying the offsets. A typical use case is to use the figure `fig.dpi_scale_trans` transformation for the `scale_trans` argument, to first scale `xt` and `yt` specified in points to display space before doing the final offset. The dpi and inches offset is a common-enough use case that we have a special helper function to create it in `matplotlib.transforms.offset_copy()`, which returns a new transform with an added offset. But in the example below, we'll create the offset transform ourselves. Note the use of the plus operator in:

```
offset = transforms.ScaledTranslation(dx, dy,
    fig.dpi_scale_trans)
shadow_transform = ax.transData + offset
```

showing that can chain transformations using the addition operator. This code says: first apply the data transformation `ax.transData` and then translate the data by `dx` and `dy` points. In typography, a point <https://en.wikipedia.org/wiki/Point_%28typography%29> is 1/72 inches, and by specifying your offsets in points, your figure will look the same regardless of the dpi resolution it is saved in.

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

fig = plt.figure()
ax = fig.add_subplot(111)

# make a simple sine wave
x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, lw=3, color='blue')

# shift the object over 2 points, and down 2 points
dx, dy = 2/72., -2/72.
offset = transforms.ScaledTranslation(dx, dy,
    fig.dpi_scale_trans)
shadow_transform = ax.transData + offset

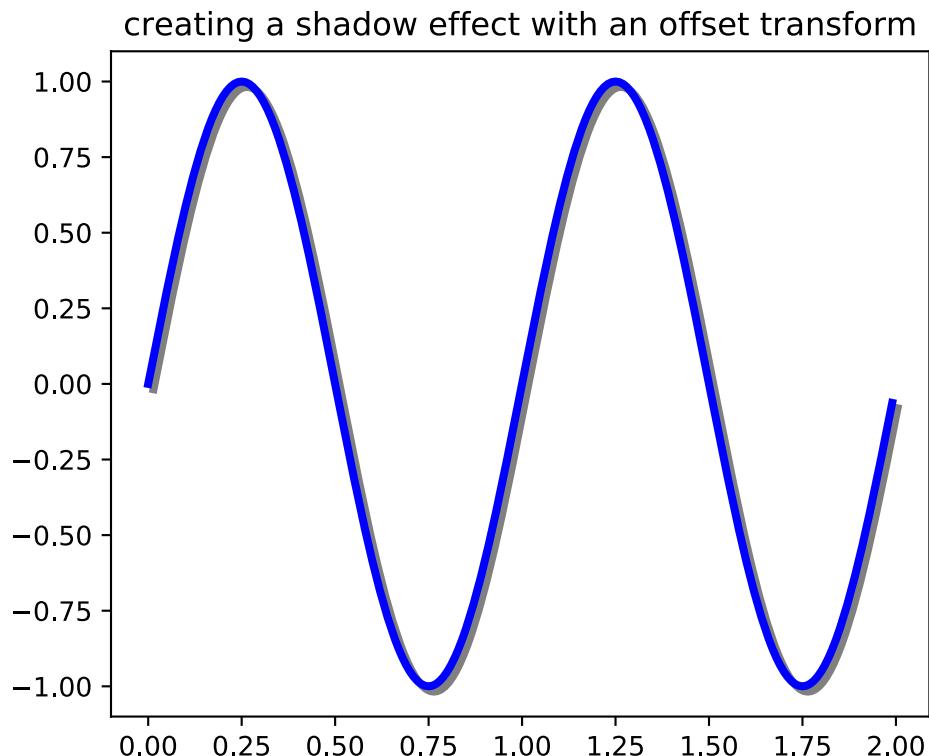
# now plot the same data with our offset transform;
# use the zorder to make sure we are below the line
```

```

ax.plot(x, y, lw=3, color='gray',
        transform=shadow_transform,
        zorder=0.5*line.get_zorder())

ax.set_title('creating a shadow effect with an offset transform')
plt.show()

```



The transformation pipeline

The `ax.transData` transform we have been working with in this tutorial is a composite of three different transformations that comprise the transformation pipeline from data \rightarrow display coordinates. Michael Droettboom implemented the transformations framework, taking care to provide a clean API that segregated the nonlinear projections and scales that happen in polar and logarithmic plots, from the linear affine transformations that happen when you pan and zoom. There is an efficiency here, because you can pan and zoom in your axes which affects the affine transformation, but you may not need to compute the potentially expensive nonlinear scales or projections on simple navigation events. It is also possible to multiply affine transformation matrices together, and then apply them to coordinates in one step. This is not true of all possible transformations.

Here is how the `ax.transData` instance is defined in the basic separable axis `Axes` class:

```

self.transData = self.transScale + (self.transLimits + self.transAxes)

```

We've been introduced to the `transAxes` instance above in *Axes coordinates*, which maps the (0,0), (1,1) corners of the axes or subplot bounding box to `display` space, so let's look at these other two pieces.

`self.transLimits` is the transformation that takes you from `data` to `axes` coordinates; i.e., it maps your view `xlim` and `ylim` to the unit space of the axes (and `transAxes` then takes that unit space to `display` space). We can see this in action here

```
In [80]: ax = subplot(111)

In [81]: ax.set_xlim(0, 10)
Out[81]: (0, 10)

In [82]: ax.set_ylim(-1, 1)
Out[82]: (-1, 1)

In [84]: ax.transLimits.transform((0,-1))
Out[84]: array([ 0.,  0.])

In [85]: ax.transLimits.transform((10,-1))
Out[85]: array([ 1.,  0.])

In [86]: ax.transLimits.transform((10,1))
Out[86]: array([ 1.,  1.])

In [87]: ax.transLimits.transform((5,0))
Out[87]: array([ 0.5,  0.5])
```

and we can use this same inverted transformation to go from the unit `axes` coordinates back to `data` coordinates.

```
In [90]: inv.transform((0.25, 0.25))
Out[90]: array([ 2.5, -0.5])
```

The final piece is the `self.transScale` attribute, which is responsible for the optional non-linear scaling of the data, e.g., for logarithmic axes. When an `Axes` is initially setup, this is just set to the identity transform, since the basic matplotlib axes has linear scale, but when you call a logarithmic scaling function like `semilogx()` or explicitly set the scale to logarithmic with `set_xscale()`, then the `ax.transScale` attribute is set to handle the nonlinear projection. The scales transforms are properties of the respective `xaxis` and `yaxis` `Axis` instances. For example, when you call `ax.set_xscale('log')`, the `xaxis` updates its scale to a `matplotlib.scale.LogScale` instance.

For non-separable axes the `PolarAxes`, there is one more piece to consider, the projection transformation. The `transData` `matplotlib.projections.polar.PolarAxes` is similar to that for the typical separable matplotlib Axes, with one additional piece `transProjection`:

```
self.transData = self.transScale + self.transProjection + \
    (self.transProjectionAffine + self.transAxes)
```

`transProjection` handles the projection from the space, e.g., latitude and longitude for map data, or radius and theta for polar data, to a separable Cartesian coordinate system. There are several projection examples in the `matplotlib.projections` package, and the best way to learn more is to open the source for those packages and see how to make your own, since matplotlib supports extensible axes and projections. Michael

Droettboom has provided a nice tutorial example of creating a hammer projection axes; see [api example code: custom_projection_example.py](#).

3.3.2 Path Tutorial

The object underlying all of the `matplotlib.patch` objects is the `Path`, which supports the standard set of `moveto`, `lineto`, `curveto` commands to draw simple and compound outlines consisting of line segments and splines. The `Path` is instantiated with a $(N,2)$ array of (x,y) vertices, and a N -length array of path codes. For example to draw the unit rectangle from $(0,0)$ to $(1,1)$, we could use this code

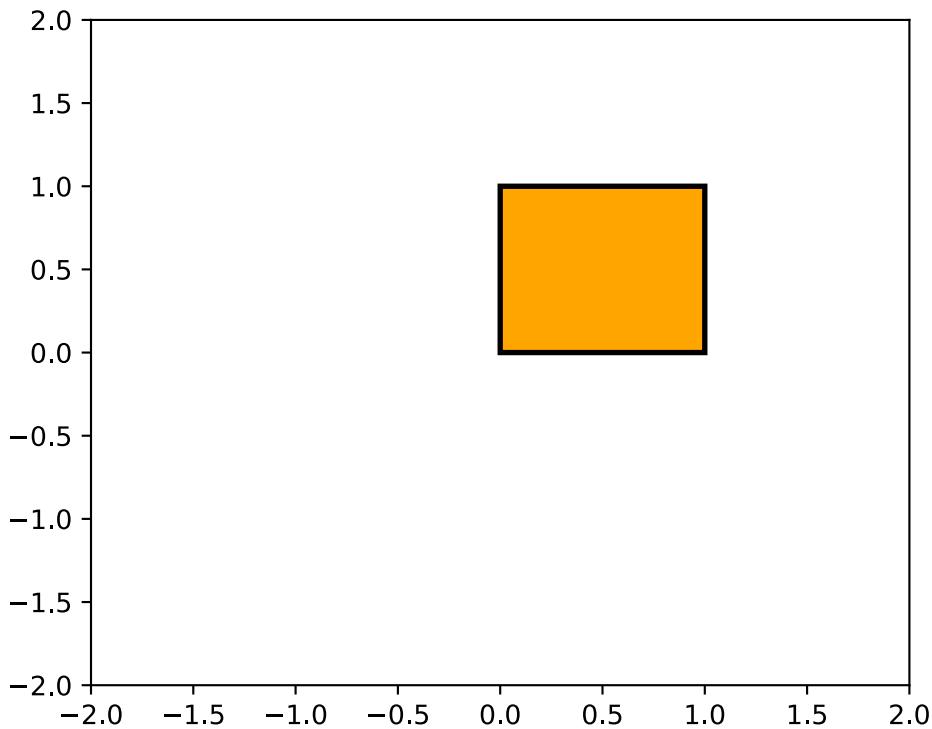
```
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [
    (0., 0.), # left, bottom
    (0., 1.), # left, top
    (1., 1.), # right, top
    (1., 0.), # right, bottom
    (0., 0.), # ignored
]

codes = [Path.MOVETO,
         Path.LINETO,
         Path.LINETO,
         Path.LINETO,
         Path.CLOSEPOLY,
         ]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='orange', lw=2)
ax.add_patch(patch)
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)
plt.show()
```



The following path codes are recognized

Code	Vertices	Description
STOP	1 (ignored)	A marker for the end of the entire path (currently not required and ignored)
MOVETO	1	Pick up the pen and move to the given vertex.
LINETO	1	Draw a line from the current position to the given vertex.
CURVE3	2 (1 control point, 1 endpoint)	Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.
CURVE4	3 (2 control points, 1 endpoint)	Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.
CLOSEPOLY (point itself is ignored)		Draw a line segment to the start point of the current polyline.

Bézier example

Some of the path components require multiple vertices to specify them: for example CURVE 3 is a Bézier curve with one control point and one end point, and CURVE4 has three vertices for the two control points and the end point. The example below shows a CURVE4 Bézier spline – the Bézier curve will be contained in the convex hull of the start point, the two control points, and the end point

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
```

```
import matplotlib.patches as patches

verts = [
    (0., 0.), # P0
    (0.2, 1.), # P1
    (1., 0.8), # P2
    (0.8, 0.), # P3
]

codes = [Path.MOVETO,
         Path.CURVE4,
         Path.CURVE4,
         Path.CURVE4,
         ]

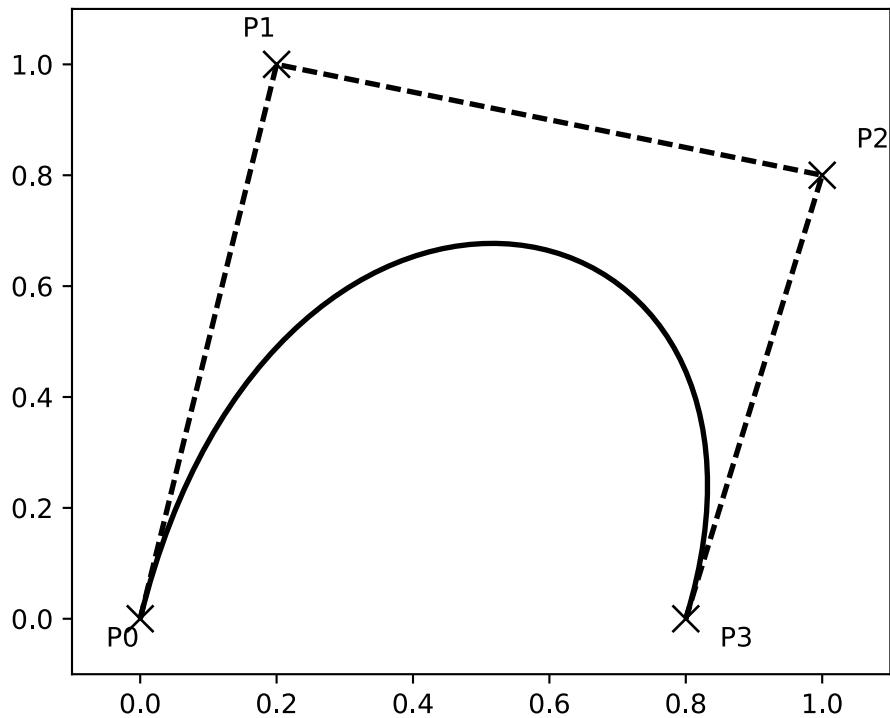
path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='none', lw=2)
ax.add_patch(patch)

xs, ys = zip(*verts)
ax.plot(xs, ys, 'x--', lw=2, color='black', ms=10)

ax.text(-0.05, -0.05, 'P0')
ax.text(0.15, 1.05, 'P1')
ax.text(1.05, 0.85, 'P2')
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
plt.show()
```



Compound paths

All of the simple patch primitives in matplotlib, Rectangle, Circle, Polygon, etc, are implemented with simple path. Plotting functions like `hist()` and `bar()`, which create a number of primitives, e.g., a bunch of Rectangles, can usually be implemented more efficiently using a compound path. The reason `bar` creates a list of rectangles and not a compound path is largely historical: the `Path` code is comparatively new and `bar` predates it. While we could change it now, it would break old code, so here we will cover how to create compound paths, replacing the functionality in `bar`, in case you need to do so in your own code for efficiency reasons, e.g., you are creating an animated bar plot.

We will make the histogram chart by creating a series of rectangles for each histogram bar: the rectangle width is the bin width and the rectangle height is the number of datapoints in that bin. First we'll create some random normally distributed data and compute the histogram. Because numpy returns the bin edges and not centers, the length of `bins` is 1 greater than the length of `n` in the example below:

```
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
```

We'll now extract the corners of the rectangles. Each of the `left`, `bottom`, etc, arrays below is `len(n)`, where `n` is the array of counts for each histogram bar:

```
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
```

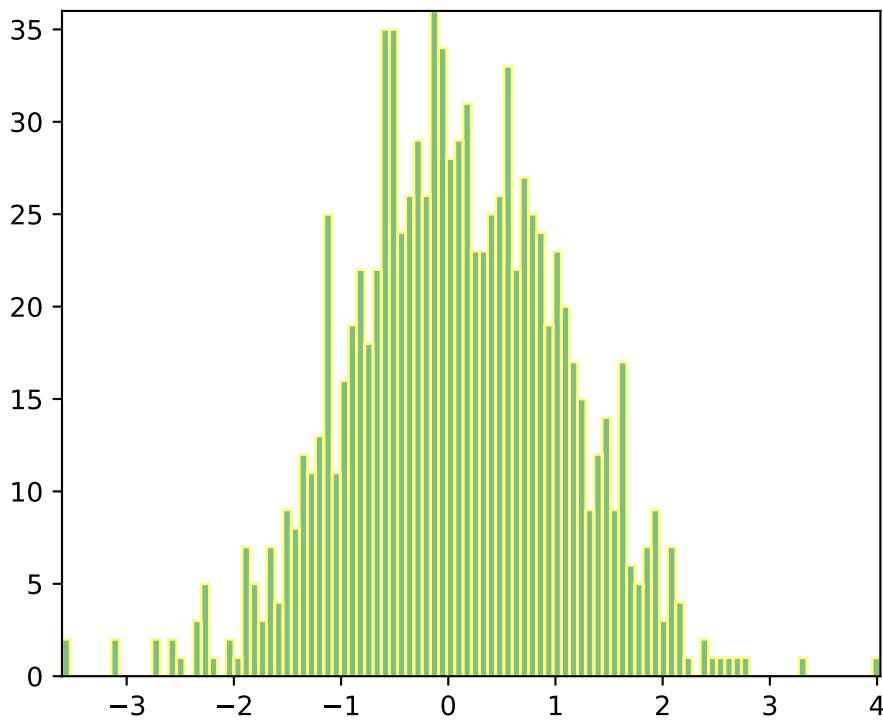
Now we have to construct our compound path, which will consist of a series of MOVETO, LINETO and CLOSEPOLY for each rectangle. For each rectangle, we need 5 vertices: 1 for the MOVETO, 3 for the LINETO, and 1 for the CLOSEPOLY. As indicated in the table above, the vertex for the closepoly is ignored but we still need it to keep the codes aligned with the vertices:

```
nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5,0] = left
verts[0::5,1] = bottom
verts[1::5,0] = left
verts[1::5,1] = top
verts[2::5,0] = right
verts[2::5,1] = top
verts[3::5,0] = right
verts[3::5,1] = bottom
```

All that remains is to create the path, attach it to a PathPatch, and add it to our axes:

```
barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
    edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)
```

Here is the result



3.3.3 Path effects guide

Matplotlib's `patheffects` module provides functionality to apply a multiple draw stage to any Artist which can be rendered via a `Path`.

Artists which can have a path effect applied to them include `Patch`, `Line2D`, `Collection` and even `Text`. Each artist's path effects can be controlled via the `set_path_effects` method (`set_path_effects`), which takes an iterable of `AbstractPathEffect` instances.

The simplest path effect is the `Normal` effect, which simply draws the artist without any effect:

```
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(5, 1.5))
text = fig.text(0.5, 0.5, 'Hello path effects world!\nThis is the normal '
                     'path effect.\nPretty dull, huh?',
               ha='center', va='center', size=20)
text.set_path_effects([path_effects.Normal()])
plt.show()
```

Hello path effects world!

This is the normal path effect.

Pretty dull, huh?

Whilst the plot doesn't look any different to what you would expect without any path effects, the drawing of the text now been changed to use the path effects framework, opening up the possibilities for more interesting examples.

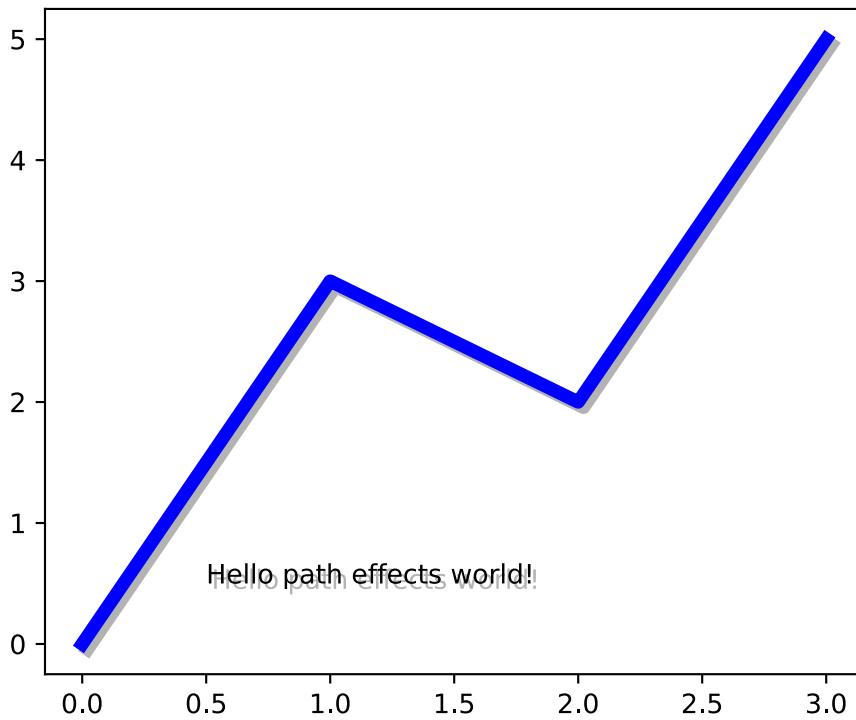
Adding a shadow

A far more interesting path effect than `Normal` is the drop-shadow, which we can apply to any of our path based artists. The classes `SimplePatchShadow` and `SimpleLineShadow` do precisely this by drawing either a filled patch or a line patch below the original artist:

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

text = plt.text(0.5, 0.5, 'Hello path effects world!',
                path_effects=[path_effects.withSimplePatchShadow()])

plt.plot([0, 3, 2, 5], linewidth=5, color='blue',
         path_effects=[path_effects.SimpleLineShadow(),
                     path_effects.Normal()])
plt.show()
```



Notice the two approaches to setting the path effects in this example. The first uses the `with*` classes to include the desired functionality automatically followed with the “normal” effect, whereas the latter explicitly defines the two path effects to draw.

Making an artist stand out

One nice way of making artists visually stand out is to draw an outline in a bold color below the actual artist. The `Stroke` path effect makes this a relatively simple task:

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

fig = plt.figure(figsize=(7, 1))
text = fig.text(0.5, 0.5, 'This text stands out because of\n' +
               'its black border.', color='white',
               ha='center', va='center', size=30)
text.set_path_effects([path_effects.Stroke(linewidth=3, foreground='black'),
                      path_effects.Normal()])
plt.show()
```

This text stands out because of its black border.

It is important to note that this effect only works because we have drawn the text path twice; once with a thick black line, and then once with the original text path on top.

You may have noticed that the keywords to `Stroke` and `SimplePatchShadow` and `SimpleLineShadow` are not the usual Artist keywords (such as `facecolor` and `edgecolor` etc.). This is because with these path effects we are operating at lower level of matplotlib. In fact, the keywords which are accepted are those for a `matplotlib.backend_bases.GraphicsContextBase` instance, which have been designed for making it easy to create new backends - and not for its user interface.

Greater control of the path effect artist

As already mentioned, some of the path effects operate at a lower level than most users will be used to, meaning that setting keywords such as `facecolor` and `edgecolor` raise an `AttributeError`. Luckily there is a generic `PathPatchEffect` path effect which creates a `PathPatch` class with the original path. The keywords to this effect are identical to those of `PathPatch`:

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

fig = plt.figure(figsize=(8, 1))
t = fig.text(0.02, 0.5, 'Hatch shadow', fontsize=75, weight=1000, va='center')
t.set_path_effects([path_effects.PathPatchEffect(offset=(4, -4), hatch='xxxx',
                                                facecolor='gray'),
                    path_effects.PathPatchEffect(edgecolor='white', linewidth=1.1,
                                                facecolor='black')])
plt.show()
```


WORKING WITH TEXT

4.1 Text introduction

matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Because it embeds fonts directly in output documents, e.g., for postscript or PDF, what you see on the screen is what you get in the hardcopy. [FreeType](#) support produces very nice, antialiased fonts, that look good even at small raster sizes. matplotlib includes its own [`matplotlib.font_manager`](#) (thanks to Paul Barrett), which implements a cross platform, W3C compliant font finding algorithm.

The user has a great deal of control over text properties (font size, font weight, text location and color, etc.) with sensible defaults set in the `rc` file. And significantly, for those interested in mathematical or scientific figures, matplotlib implements a large number of TeX math symbols and commands, supporting [*mathematical expressions*](#) anywhere in your figure.

4.2 Basic text commands

The following commands are used to create text in the pyplot interface

- `text()` - add text at an arbitrary location to the `Axes`; [`matplotlib.axes.Axes.text\(\)`](#) in the API.
- `xlabel()` - add a label to the x-axis; [`matplotlib.axes.Axes.set_xlabel\(\)`](#) in the API.
- `ylabel()` - add a label to the y-axis; [`matplotlib.axes.Axes.set_ylabel\(\)`](#) in the API.
- `title()` - add a title to the `Axes`; [`matplotlib.axes.Axes.set_title\(\)`](#) in the API.
- `figtext()` - add text at an arbitrary location to the `Figure`; [`matplotlib.figure.Figure.text\(\)`](#) in the API.
- `suptitle()` - add a title to the `Figure`; [`matplotlib.figure.Figure.suptitle\(\)`](#) in the API.
- **`annotate()` - add an annotation, with optional arrow, to the `Axes` ; `matplotlib.axes.Axes.annotate()`** in the API.

All of these functions create and return a [`matplotlib.text.Text`](#) instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action.

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')

ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, u'unicode: Institut für Festkörperphysik')

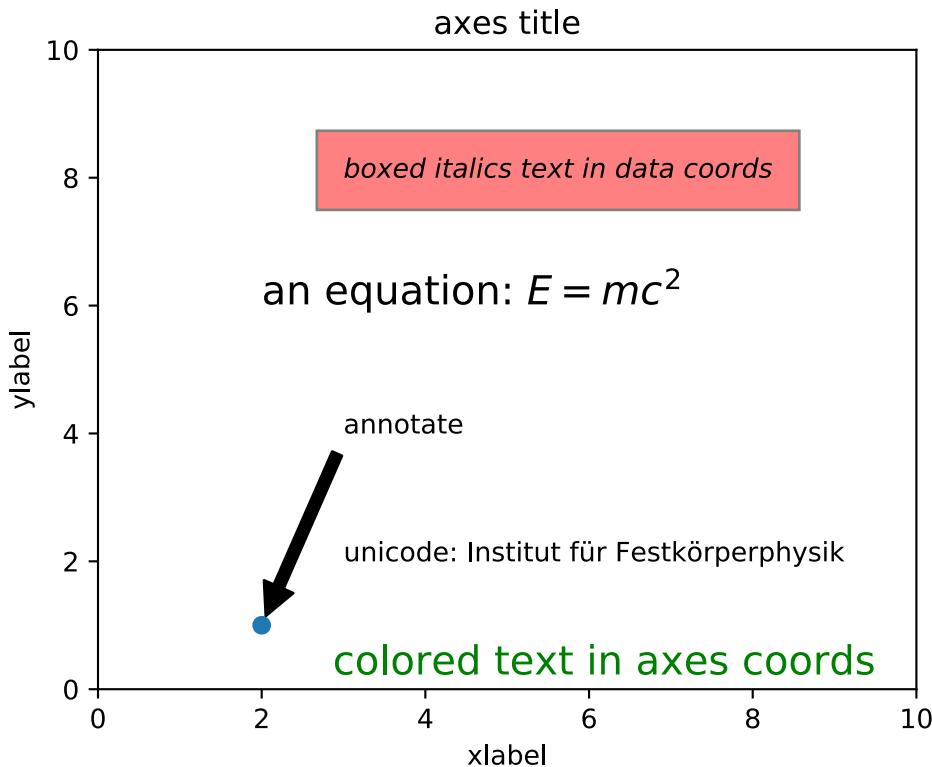
ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)

ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10])

plt.show()
```

bold figure suptitle



4.3 Text properties and layout

The `matplotlib.text.Text` instances have a variety of properties which can be configured via keyword arguments to the text commands (e.g., `title()`, `xlabel()` and `text()`).

Property	Value Type
alpha	float
backgroundcolor	any matplotlib color
bbox	Rectangle prop dict plus key 'pad' which is a pad in points
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True False]
clip_path	a Path instance and a Transform instance, a Patch
color	any matplotlib color
family	['serif' 'sans-serif' 'cursive' 'fantasy' 'monospace']
fontproperties	a FontProperties instance
horizontalalignment or ha	['center' 'right' 'left']
label	any string
linespacing	float
multialignment	['left' 'right' 'center']
name or fontname	string e.g., ['Sans' 'Courier' 'Helvetica' ...]
picker	[None float boolean callable]
position	(x, y)
rotation	[angle in degrees 'vertical' 'horizontal']
size or fontsize	[size in points relative size, e.g., 'smaller', 'x-large']
style or fontstyle	['normal' 'italic' 'oblique']
text	string or anything printable with '%s' conversion
transform	a Transform instance
variant	['normal' 'small-caps']
verticalalignment or va	['center' 'top' 'bottom' 'baseline']
visible	[True False]
weight or fontweight	['normal' 'bold' 'heavy' 'light' 'ultrabold' 'ultralight']
x	float
y	float
zorder	any number

You can lay out text with the alignment arguments `horizontalalignment`, `verticalalignment`, and `multialignment`. `horizontalalignment` controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. `verticalalignment` controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. `multialignment`, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the `text()` command to show the various alignment possibilities. The use of `transform=ax.transAxes` throughout the code indicates that the coordinates are given relative to the axes bounding box, with 0,0 being the lower left of the axes and 1,1 the upper right.

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

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
```

```
top = bottom + height

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

# axes coordinates are 0,0 is bottom left and 1,1 is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False
)

ax.add_patch(p)

ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

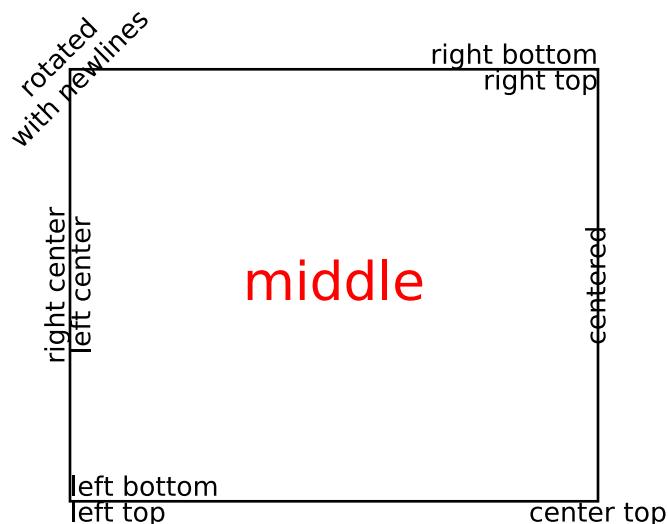
ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
```

```
    fontsize=20, color='red',
    transform=ax.transAxes)

ax.text(right, 0.5*(bottom+top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

ax.set_axis_off()
plt.show()
```



4.4 Default Font

The base default font is controlled by a set of rcParams:

rcParam	usage
'font.family'	List of either names of font or {'cursive', 'fantasy', 'monospace', 'sans', 'sans serif', 'sans-serif', 'serif'}.
'font.style'	The default style, ex 'normal', 'italic'.
'font.variant'	Default variant, ex 'normal', 'small-caps' (untested)
'font.stretch'	Default stretch, ex 'normal', 'condensed' (incomplete)
'font.weight'	Default weight. Either string or integer
'font.size'	Default font size in points. Relative font sizes ('large', 'x-small') are computed against this size.

The mapping between the family aliases ({'cursive', 'fantasy', 'monospace', 'sans', 'sans serif', 'sans-serif', 'serif'}) and actual font names is controlled by the following rcParams:

family alias	rcParam with mappings
'serif'	'font.serif'
'monospace'	'font.monospace'
'fantasy'	'font.fantasy'
'cursive'	'font.cursive'
{'sans', 'sans serif', 'sans-serif'}	'font.sans-serif'

which are lists of font names.

4.4.1 Text with non-latin glyphs

As of v2.0 the *default font* contains glyphs for many western alphabets, but still does not cover all of the glyphs that may be required by mpl users. For example, DejaVu has no coverage of Chinese, Korean, or Japanese.

To set the default font to be one that supports the code points you need, prepend the font name to 'font.family' or the desired alias lists

```
matplotlib.rcParams['font.sans-serif'] = ['Source Han Sans TW', 'sans-serif']
```

or set it in your `.matplotlibrc` file:

```
font.sans-serif: Source Han Sans TW, Arial, sans-serif
```

To control the font used on per-artist basis use the 'name', 'fontname' or 'fontproperties' kwargs documented [above](#).

On linux, `fc-list` can be a useful tool to discover the font name; for example

```
$ fc-list :lang=zh family
Noto to Sans Mono CJK TC,Noto Sans Mono CJK TC Bold
Noto Sans CJK TC,Noto Sans CJK TC Medium
Noto Sans CJK TC,Noto Sans CJK TC DemiLight
Noto Sans CJK KR,Noto Sans CJK KR Black
```

```
Noto Sans CJK TC,Noto Sans CJK TC Black  
Noto Sans Mono CJK TC,Noto Sans Mono CJK TC Regular  
Noto Sans CJK SC,Noto Sans CJK SC Light
```

lists all of the fonts that support Chinese.

4.5 Annotation

Table of Contents

- *Annotation*
 - *Basic annotation*
 - *Advanced Annotation*
 - * *Annotating with Text with Box*
 - * *Annotating with Arrow*
 - * *Placing Artist at the anchored location of the Axes*
 - * *Using Complex Coordinates with Annotations*
 - * *Using ConnectorPatch*
 - * *Zoom effect between Axes*
 - * *Define Custom BoxStyle*

4.5.1 Basic annotation

The uses of the basic `text()` will place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x, y)` tuples.

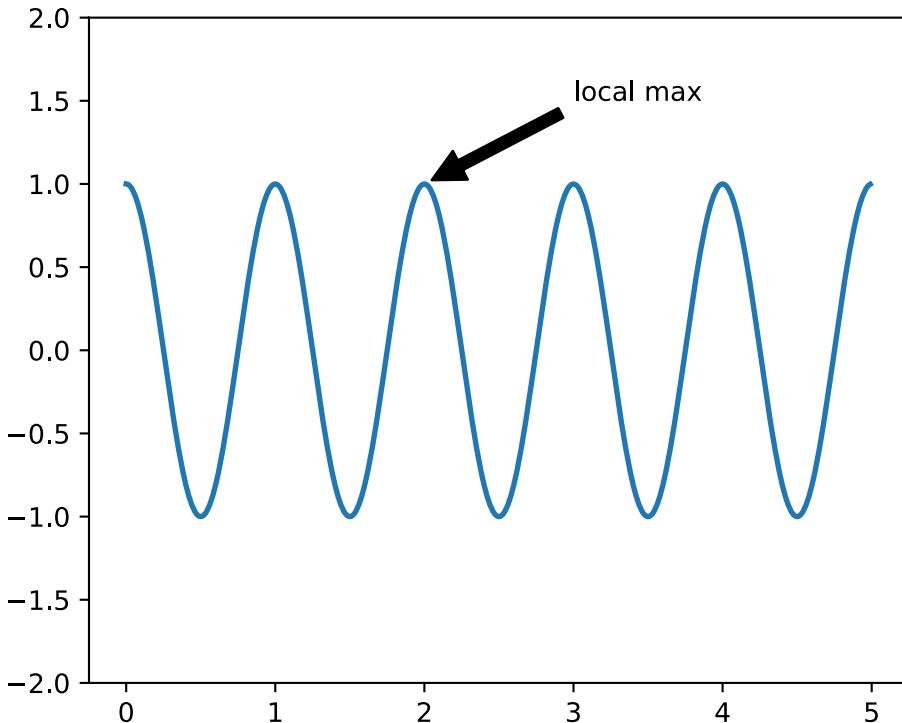
```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)

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

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

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



In this example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose – you can specify the coordinate system of `xy` and `xytext` with one of the following strings for `xycoords` and `textcoords` (default is ‘data’)

argument	coordinate system
‘figure points’	points from the lower left corner of the figure
‘figure pixels’	pixels from the lower left corner of the figure
‘figure fraction’	0,0 is lower left of figure and 1,1 is upper right
‘axes points’	points from lower left corner of axes
‘axes pixels’	pixels from lower left corner of axes
‘axes fraction’	0,0 is lower left of axes and 1,1 is upper right
‘data’	use the axes data coordinate system

For example to place the text coordinates in fractional axes coordinates, one could do:

```
ax.annotate('local max', xy=(3, 1), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
            )
```

For physical coordinate systems (points or pixels) the origin is the bottom-left of the figure or axes.

Optionally, you can enable drawing of an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

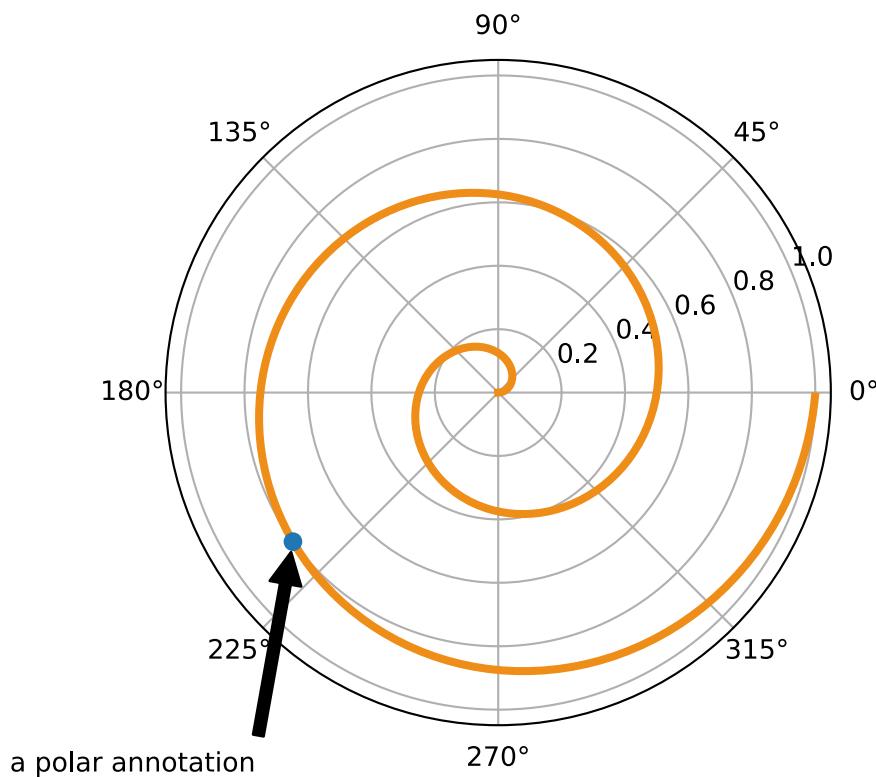
arrowprops key	description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
headwidth	the width of the base of the arrow head in points
shrink	move the tip and base some percent away from the annotated point and text
**kwargs	any key for <code>matplotlib.patches.Polygon</code> , e.g., <code>facecolor</code>

In the example below, the `xy` point is in native coordinates (`xycoords` defaults to ‘data’). For a polar axes, this is in (theta, radius) space. The text in this example is placed in the fractional figure coordinate system. `matplotlib.text.Text` keyword args like `horizontalalignment`, `verticalalignment` and `fontsize` are passed from `annotate` to the `Text` instance.

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

fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
r = np.arange(0,1,0.001)
theta = 2*2*np.pi*r
line, = ax.plot(theta, r, color="#ee8d18", lw=3)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr), # theta, radius
            xytext=(0.05, 0.05), # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
            )
plt.show()
```



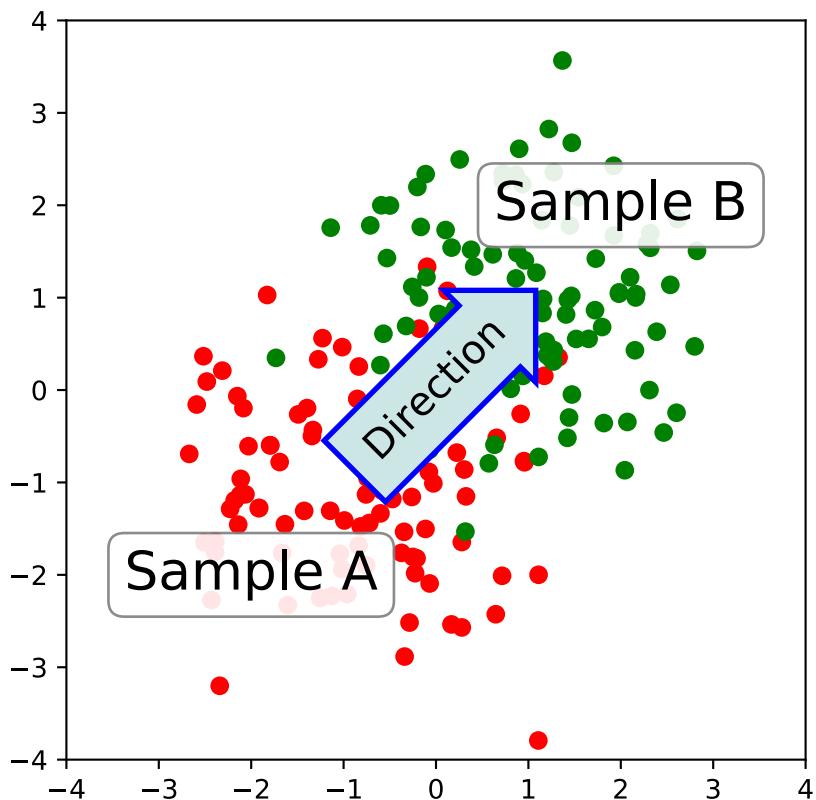
For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see [Advanced Annotation](#) and [pylab_examples example code: annotation_demo.py](#).

Do not proceed unless you have already read [Basic annotation](#), `text()` and `annotate()`!

4.5.2 Advanced Annotation

Annotating with Text with Box

Let's start with a simple example.



The `text()` function in the pyplot module (or `text` method of the Axes class) takes `bbox` keyword argument, and when given, a box around the text is drawn.

```
bbox_props = dict(boxstyle="rarrow,pad=0.3", fc="cyan", ec="b", lw=2)
t = ax.text(0, 0, "Direction", ha="center", va="center", rotation=45,
            size=15,
            bbox=bbox_props)
```

The patch object associated with the text can be accessed by:

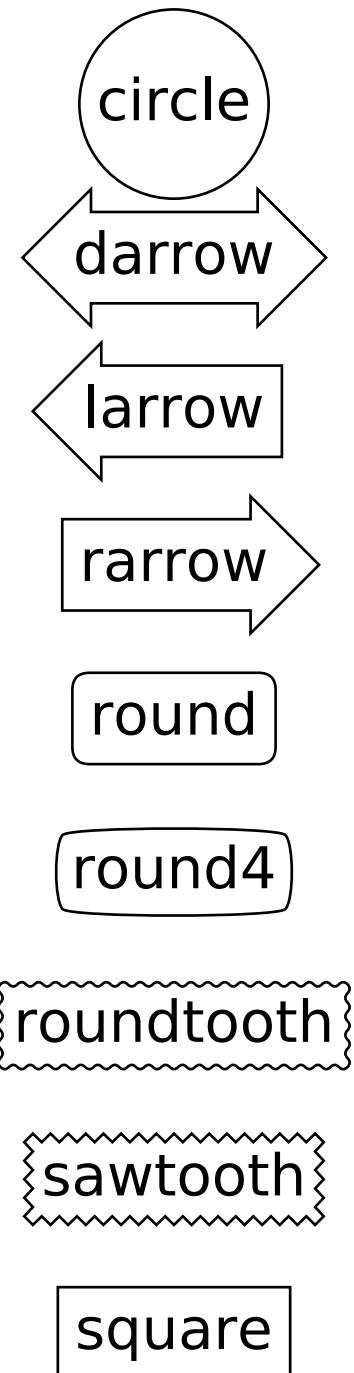
```
bb = t.get_bbox_patch()
```

The return value is an instance of `FancyBboxPatch` and the patch properties like `facecolor`, `edgecolor`, etc. can be accessed and modified as usual. To change the shape of the box, use the `set_boxstyle` method.

```
bb.set_boxstyle("rarrow", pad=0.6)
```

The arguments are the name of the box style with its attributes as keyword arguments. Currently, following box styles are implemented.

Class	Name	Attrs
Circle	circle	pad=0.3
DArrow	darrow	pad=0.3
LArrow	larrow	pad=0.3
RArrow	rarrow	pad=0.3
Round	round	pad=0.3,rounding_size=None
Round4	round4	pad=0.3,rounding_size=None
Roundtooth	roundtooth	pad=0.3,tooth_size=None
Sawtooth	sawtooth	pad=0.3,tooth_size=None
Square	square	pad=0.3



Note that the attribute arguments can be specified within the style name with separating comma (this form can be used as “boxstyle” value of bbox argument when initializing the text instance)

```
bb.set_boxstyle("rarrow, pad=0.6")
```

Annotating with Arrow

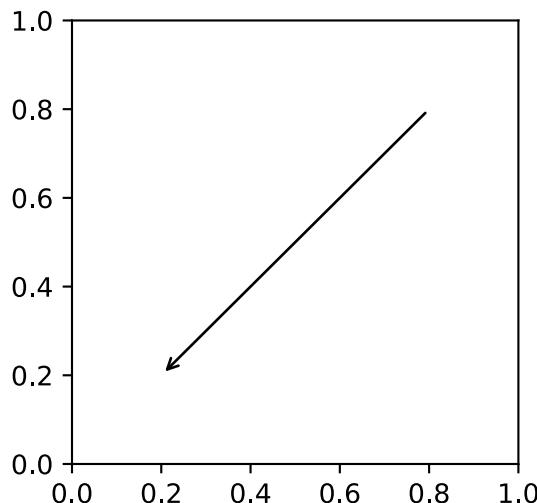
The `annotate()` function in the pyplot module (or `annotate` method of the `Axes` class) is used to draw an arrow connecting two points on the plot.

```
ax.annotate("Annotation",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='offset points',
            )
```

This annotates a point at `xy` in the given coordinate (`xycoords`) with the text at `xytext` given in `textcoords`. Often, the annotated point is specified in the `data` coordinate and the annotating text in `offset points`. See `annotate()` for available coordinate systems.

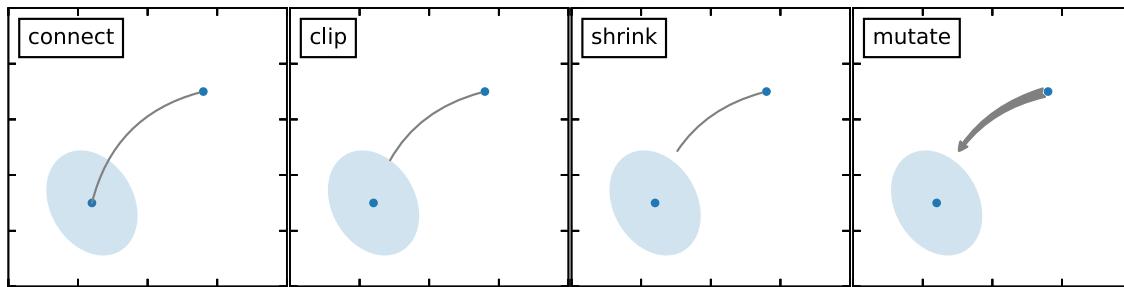
An arrow connecting two points (`xy` & `xytext`) can be optionally drawn by specifying the `arrowprops` argument. To draw only an arrow, use empty string as the first argument.

```
ax.annotate("", xy=(0.2, 0.2), xycoords='data',
            xytext=(0.8, 0.8), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                           connectionstyle="arc3"),
            )
```



The arrow drawing takes a few steps.

1. a connecting path between two points are created. This is controlled by `connectionstyle` key value.
2. If patch object is given (`patchA` & `patchB`), the path is clipped to avoid the patch.
3. The path is further shrunk by given amount of pixels (`shrinkA` & `shrinkB`)
4. The path is transmuted to arrow patch, which is controlled by the `arrowstyle` key value.

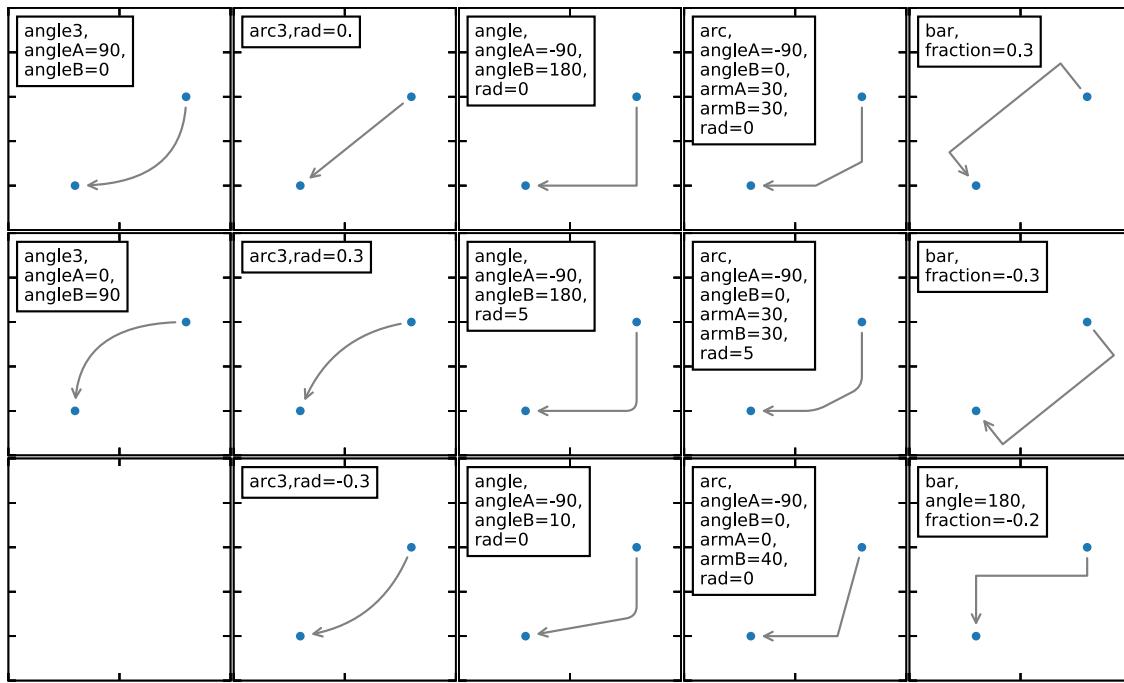


The creation of the connecting path between two points is controlled by `connectionstyle` key and the following styles are available.

Name	Attrs
<code>angle</code>	<code>angleA=90,angleB=0,rad=0.0</code>
<code>angle3</code>	<code>angleA=90,angleB=0</code>
<code>arc</code>	<code>angleA=0,angleB=0,armA=None,armB=None,rad=0.0</code>
<code>arc3</code>	<code>rad=0.0</code>
<code>bar</code>	<code>armA=0.0,armB=0.0,fraction=0.3,angle=None</code>

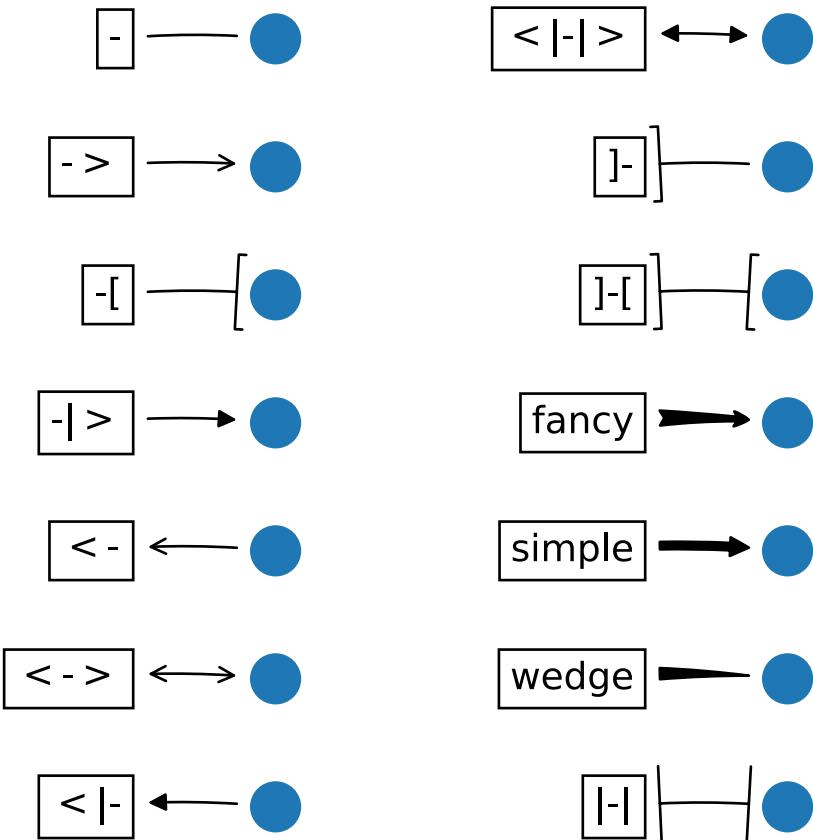
Note that “3” in `angle3` and `arc3` is meant to indicate that the resulting path is a quadratic spline segment (three control points). As will be discussed below, some arrow style options can only be used when the connecting path is a quadratic spline.

The behavior of each connection style is (limitedly) demonstrated in the example below. (Warning : The behavior of the `bar` style is currently not well defined, it may be changed in the future).



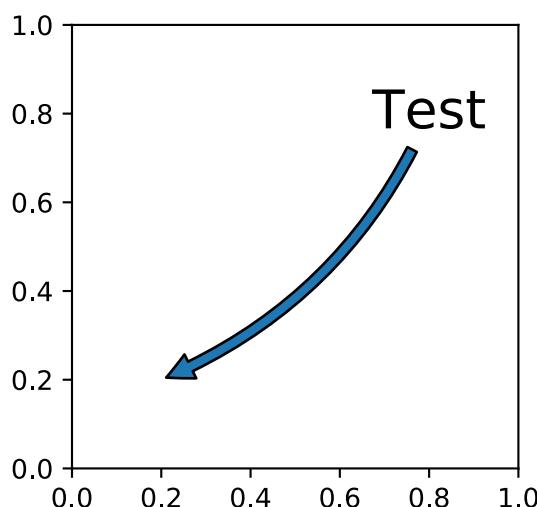
The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given `arrowstyle`.

Name	Attrs
-	None
->	head_length=0.4,head_width=0.2
-[widthB=1.0,lengthB=0.2,angleB=None
-	widthA=1.0,widthB=1.0
- >	head_length=0.4,head_width=0.2
<-	head_length=0.4,head_width=0.2
<->	head_length=0.4,head_width=0.2
< -	head_length=0.4,head_width=0.2
< - >	head_length=0.4,head_width=0.2
fancy	head_length=0.4,head_width=0.4,tail_width=0.4
simple	head_length=0.5,head_width=0.5,tail_width=0.2
wedge	tail_width=0.3,shrink_factor=0.5

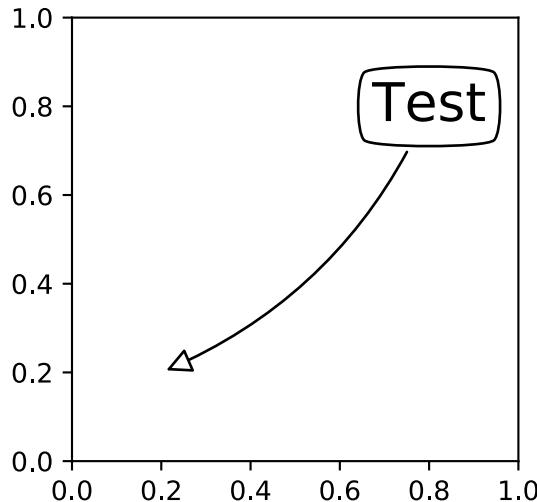


Some arrowstyles only work with connection styles that generate a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use the “angle3” or “arc3” connection style.

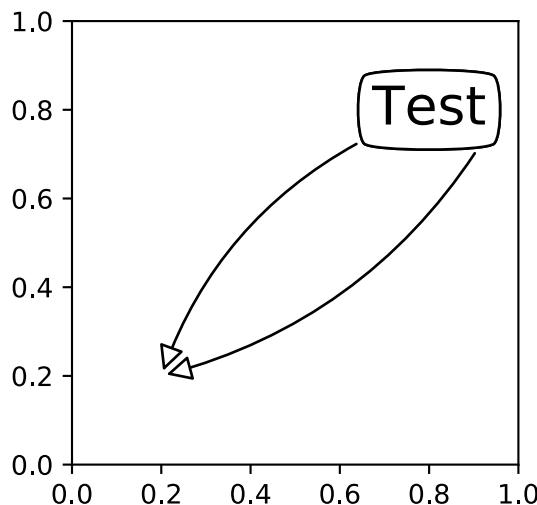
If the annotation string is given, the `patchA` is set to the `bbox` patch of the text by default.



As in the text command, a box around the text can be drawn using the `bbox` argument.



By default, the starting point is set to the center of the text extent. This can be adjusted with `relpos` key value. The values are normalized to the extent of the text. For example, (0,0) means lower-left corner and (1,1) means top-right.

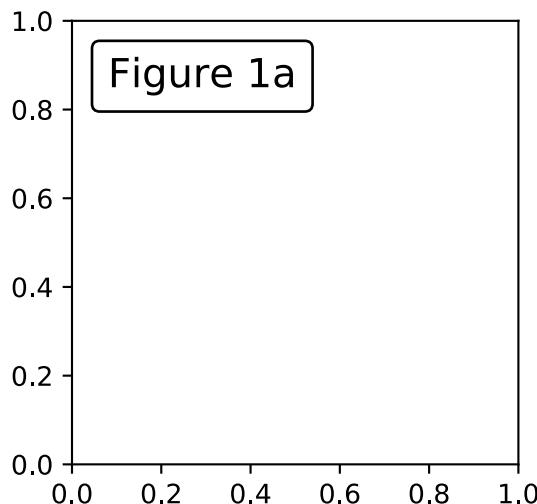


Placing Artist at the anchored location of the Axes

There are classes of artists that can be placed at an anchored location in the Axes. A common example is the legend. This type of artist can be created by using the `OffsetBox` class. A few predefined classes are available in `mpl_toolkits.axes_grid.anchored_artists`.

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredText
at = AnchoredText("Figure 1a",
```

```
prop=dict(size=8), frameon=True,
          loc=2,
        )
at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```



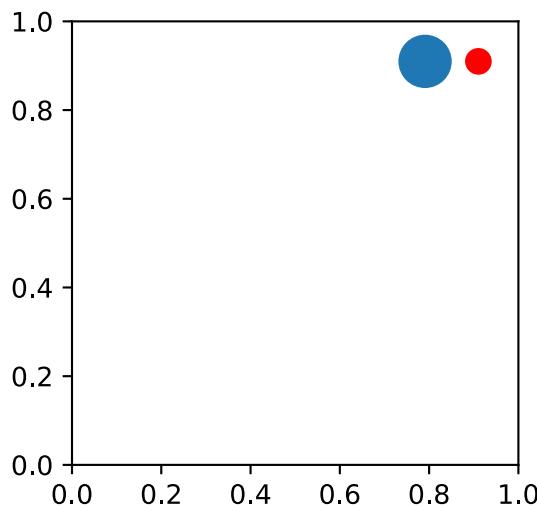
The *loc* keyword has same meaning as in the legend command.

A simple application is when the size of the artist (or collection of artists) is known in pixel size during the time of creation. For example, If you want to draw a circle with fixed size of 20 pixel x 20 pixel (radius = 10 pixel), you can utilize AnchoredDrawingArea. The instance is created with a size of the drawing area (in pixels), and arbitrary artists can added to the drawing area. Note that the extents of the artists that are added to the drawing area are not related to the placement of the drawing area itself. Only the initial size matters.

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredDrawingArea

ada = AnchoredDrawingArea(20, 20, 0, 0,
                         loc=1, pad=0., frameon=False)
p1 = Circle((10, 10), 10)
ada.drawing_area.add_artist(p1)
p2 = Circle((30, 10), 5, fc="r")
ada.drawing_area.add_artist(p2)
```

The artists that are added to the drawing area should not have a transform set (it will be overridden) and the dimensions of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixels and 5 pixels, respectively.

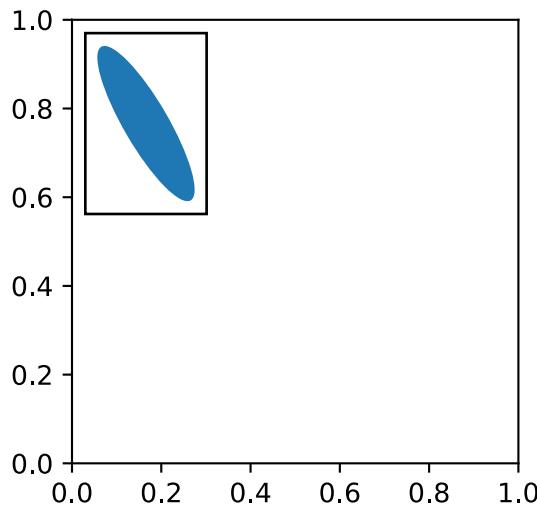


Sometimes, you want your artists to scale with the data coordinate (or coordinates other than canvas pixels). You can use `AnchoredAuxTransformBox` class. This is similar to `AnchoredDrawingArea` except that the extent of the artist is determined during the drawing time respecting the specified transform.

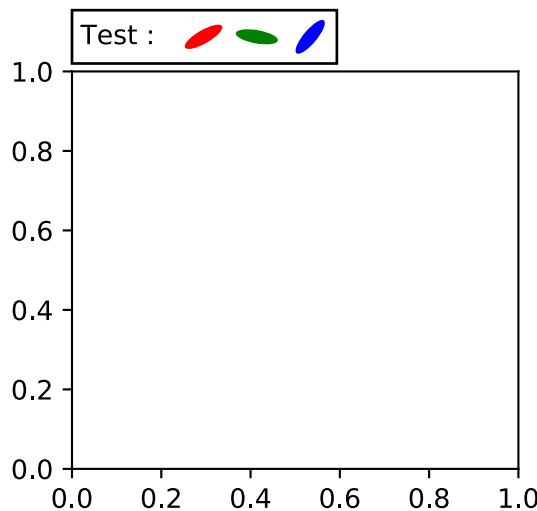
```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredAuxTransformBox

box = AnchoredAuxTransformBox(ax.transData, loc=2)
el = Ellipse((0,0), width=0.1, height=0.4, angle=30) # in data coordinates!
box.drawing_area.add_artist(el)
```

The ellipse in the above example will have width and height corresponding to 0.1 and 0.4 in data coordinates and will be automatically scaled when the view limits of the axes change.



As in the legend, the `bbox_to_anchor` argument can be set. Using the HPacker and VPacker, you can have an arrangement(?) of artist as in the legend (as a matter of fact, this is how the legend is created).



Note that unlike the legend, the `bbox_transform` is set to `IdentityTransform` by default.

Using Complex Coordinates with Annotations

The Annotation in matplotlib supports several types of coordinates as described in [Basic annotation](#). For an advanced user who wants more control, it supports a few other options.

1. `Transform` instance. For example,

```
ax.annotate("Test", xy=(0.5, 0.5), xycoords=ax.transAxes)
```

is identical to

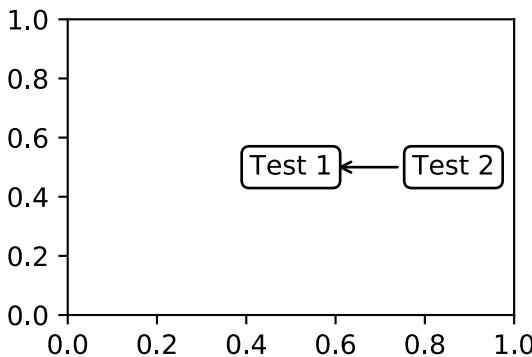
```
ax.annotate("Test", xy=(0.5, 0.5), xycoords="axes fraction")
```

With this, you can annotate a point in other axes.

```
ax1, ax2 = subplot(121), subplot(122)
ax2.annotate("Test", xy=(0.5, 0.5), xycoords=ax1.transData,
             xytext=(0.5, 0.5), textcoords=ax2.transData,
             arrowprops=dict(arrowstyle="->"))
```

2. `Artist` instance. The `xy` value (or `xytext`) is interpreted as a fractional coordinate of the bbox (return value of `get_window_extent`) of the artist.

```
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1, # (1,0.5) of the an1's bbox
                  xytext=(30,0), textcoords="offset points",
                  va="center", ha="left",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
```



Note that it is your responsibility that the extent of the coordinate artist (`an1` in above example) is determined before `an2` gets drawn. In most cases, it means that `an2` needs to be drawn later than `an1`.

3. A callable object that returns an instance of either `BboxBase` or `Transform`. If a transform is returned, it is the same as 1 and if a bbox is returned, it is the same as 2. The callable object should take a single argument of the renderer instance. For example, the following two commands give identical results

```
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,
                  xytext=(30, 0), textcoords="offset points")
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1.get_window_extent,
                  xytext=(30, 0), textcoords="offset points")
```

4. A tuple of two coordinate specifications. The first item is for the x-coordinate and the second is for the y-coordinate. For example,

```
annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))
```

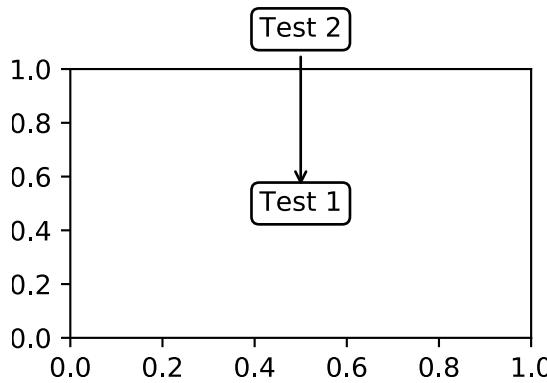
0.5 is in data coordinates, and 1 is in normalized axes coordinates. You may use an artist or transform as with a tuple. For example,

```
import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2", xy=(0.5, 1.), xycoords=an1,
                  xytext=(0.5,1.1), textcoords=(an1, "axes fraction"),
                  va="bottom", ha="center",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))

plt.show()
```



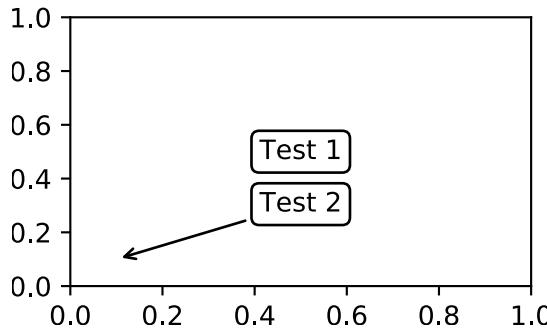
- Sometimes, you want your annotation with some “offset points”, not from the annotated point but from some other point. `OffsetFrom` is a helper class for such cases.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

from matplotlib.text import OffsetFrom
offset_from = OffsetFrom(an1, (0.5, 0))
an2 = ax.annotate("Test 2", xy=(0.1, 0.1), xycoords="data",
                  xytext=(0, -10), textcoords=offset_from,
                  # xytext is offset points from "xy=(0.5, 0)", xycoords=an1"
                  va="top", ha="center",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))

plt.show()
```



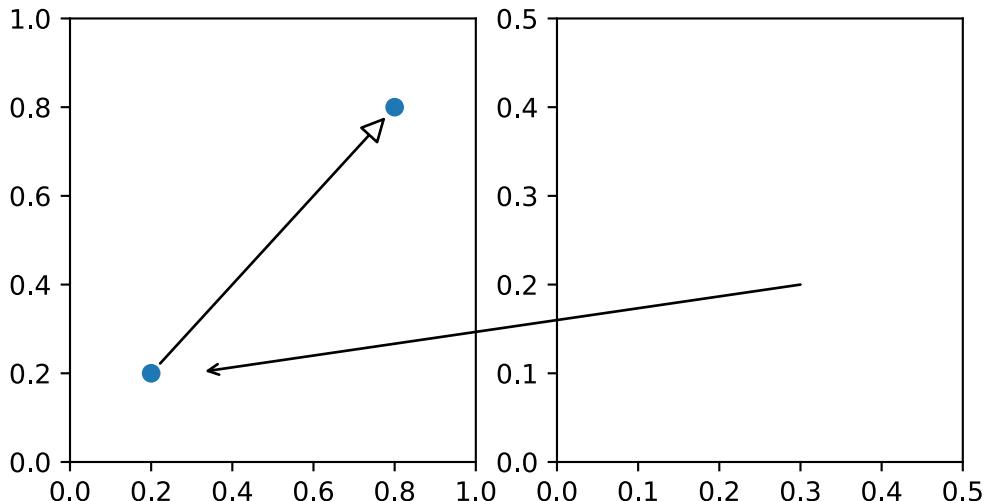
You may take a look at this example [pylab_examples example code: annotation_demo3.py](#).

Using ConnectorPatch

The ConnectorPatch is like an annotation without text. While the `annotate` function is recommended in most situations, the `ConnectorPatch` is useful when you want to connect points in different axes.

```
from matplotlib.patches import ConnectionPatch
xy = (0.2, 0.2)
con = ConnectionPatch(xyA=xy, xyB=xy, coordsA="data", coordsB="data",
                      axesA=ax1, axesB=ax2)
ax2.add_artist(con)
```

The above code connects point `xy` in the data coordinates of `ax1` to point `xy` in the data coordinates of `ax2`. Here is a simple example.

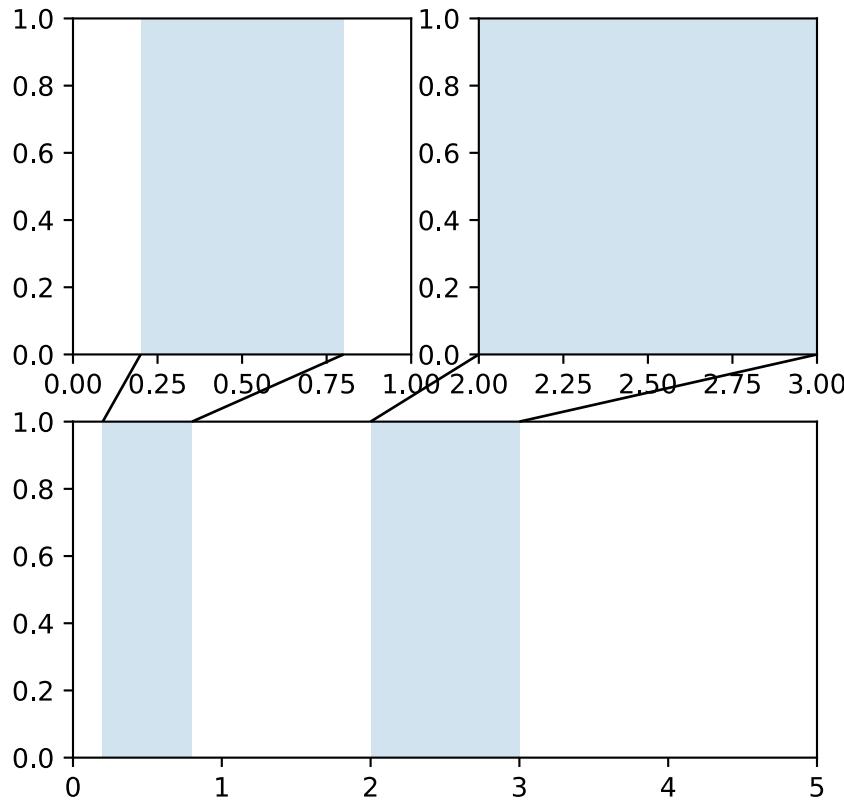


While the `ConnectorPatch` instance can be added to any axes, you may want to add it to the axes that is latest in drawing order to prevent overlap by other axes.

Advanced Topics

Zoom effect between Axes

`mpl_toolkits.axes_grid.inset_locator` defines some patch classes useful for interconnecting two axes. Understanding the code requires some knowledge of how `mpl`'s transform works. But, utilizing it will be straight forward.



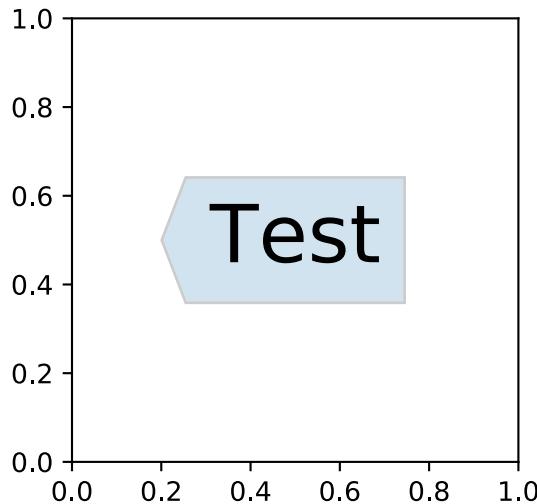
Define Custom BoxStyle

You can use a custom box style. The value for the `boxstyle` can be a callable object in the following forms.:

```
def __call__(self, x0, y0, width, height, mutation_size,
            aspect_ratio=1.):
    """
    Given the location and size of the box, return the path of
    the box around it.

    - *x0*, *y0*, *width*, *height* : location and size of the box
    - *mutation_size* : a reference scale for the mutation.
    - *aspect_ratio* : aspect-ratio for the mutation.
    """
    path = ...
    return path
```

Here is a complete example.



However, it is recommended that you derive from the `matplotlib.patches.BoxStyle._Base` as demonstrated below.

```
from matplotlib.path import Path
from matplotlib.patches import BoxStyle
import matplotlib.pyplot as plt

# we may derive from matplotlib.patches.BoxStyle._Base class.
# You need to override transmute method in this case.

class MyStyle(BoxStyle._Base):
    """
    A simple box.
    """

    def __init__(self, pad=0.3):
        """
        The arguments need to be floating numbers and need to have
        default values.

        *pad*
            amount of padding
        """

        self.pad = pad
        super(MyStyle, self).__init__()

    def transmute(self, x0, y0, width, height, mutation_size):
        """
        Given the location and size of the box, return the path of
        the box around it.

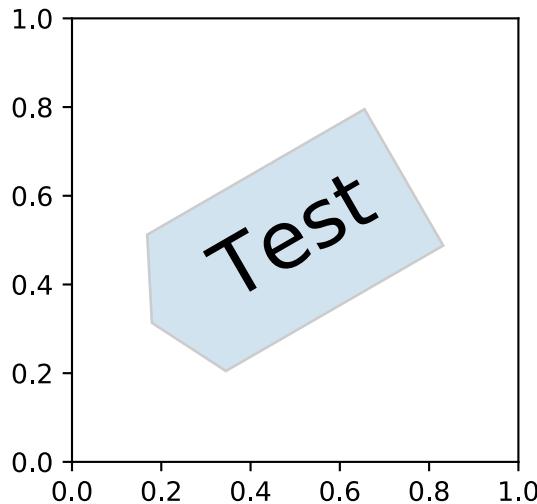
        - *x0*, *y0*, *width*, *height* : location and size of the box
        - *mutation_size* : a reference scale for the mutation.
        
```

```
Often, the *mutation_size* is the font size of the text.  
You don't need to worry about the rotation as it is  
automatically taken care of.  
"""
```

```
# padding  
pad = mutation_size * self.pad  
  
# width and height with padding added.  
width, height = width + 2.*pad, \  
                height + 2.*pad,  
  
# boundary of the padded box  
x0, y0 = x0-pad, y0-pad,  
x1, y1 = x0+width, y0 + height  
  
cp = [(x0, y0),  
       (x1, y0), (x1, y1), (x0, y1),  
       (x0-pad, (y0+y1)/2.), (x0, y0),  
       (x0, y0)]  
  
com = [Path.MOVETO,  
       Path.LINETO, Path.LINETO, Path.LINETO,  
       Path.LINETO, Path.LINETO,  
       Path.CLOSEPOLY]  
  
path = Path(cp, com)  
  
return path
```



```
# register the custom style  
BoxStyle._style_list["angled"] = MyStyle  
  
plt.figure(1, figsize=(3,3))  
ax = plt.subplot(111)  
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,  
       bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))  
  
del BoxStyle._style_list["angled"]  
  
plt.show()
```



Similarly, you can define a custom `ConnectionStyle` and a custom `ArrowStyle`. See the source code of `lib/matplotlib/patches.py` and check how each style class is defined.

4.6 Writing mathematical expressions

You can use a subset TeX markup in any matplotlib text string by placing it inside a pair of dollar signs (\$).

Note that you do not need to have TeX installed, since matplotlib ships its own TeX expression parser, layout engine and fonts. The layout engine is a fairly direct adaptation of the layout algorithms in Donald Knuth's TeX, so the quality is quite good (matplotlib also provides a `usetex` option for those who do want to call out to TeX to generate their text (see [Text rendering With LaTeX](#)).

Any text element can use math text. You should use raw strings (precede the quotes with an 'r'), and surround the math text with dollar signs (\$), as in TeX. Regular text and mathtext can be interleaved within the same string. Mathtext can use DejaVu Sans (default), DejaVu Serif, the Computer Modern fonts (from (La)TeX), STIX fonts (which are designed to blend well with Times), or a Unicode font that you provide. The mathtext font can be selected with the customization variable `mathtext.fontset` (see [Customizing matplotlib](#))

Note: On “narrow” builds of Python, if you use the STIX fonts you should also set `ps.fonttype` and `pdf.fonttype` to 3 (the default), not 42. Otherwise some characters will not be visible.

Here is a simple example:

```
# plain text
plt.title('alpha > beta')
```

produces “alpha > beta”.

Whereas this:

```
# math text
plt.title(r'$\alpha > \beta$')
```

produces “ $\alpha > \beta$ ”.

Note: Mathtext should be placed between a pair of dollar signs (\$). To make it easy to display monetary values, e.g., “\$100.00”, if a single dollar sign is present in the entire string, it will be displayed verbatim as a dollar sign. This is a small change from regular TeX, where the dollar sign in non-math text would have to be escaped ('\\$').

Note: While the syntax inside the pair of dollar signs (\$) aims to be TeX-like, the text outside does not. In particular, characters such as:

```
# $ % & ~ _ ^ \ { } \( \) \[ \]
```

have special meaning outside of math mode in TeX. Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag. See the [usetex tutorial](#) for more information.

4.6.1 Subscripts and superscripts

To make subscripts and superscripts, use the '`_`' and '`^`' symbols:

```
r'$\alpha_i > \beta_i$'
```

$$\alpha_i > \beta_i \tag{4.1}$$

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of x_i from 0 to ∞ , you could do:

```
r'$\sum_{i=0}^\infty x_i$'
```

$$\sum_{i=0}^{\infty} x_i \tag{4.2}$$

4.6.2 Fractions, binomials and stacked numbers

Fractions, binomials and stacked numbers can be created with the `\frac{...}{...}`, `\binom{...}{...}` and `\stackrel{...}{...}` commands, respectively:

```
r'$\frac{3}{4} \binom{3}{4} \stackrel{3}{4}$'
```

produces

$$\frac{3}{4} \binom{3}{4} \stackrel{3}{4} \tag{4.3}$$

Fractions can be arbitrarily nested:

```
r'$\frac{5 - \frac{1}{x}}{4}$'
```

produces

$$\frac{5 - \frac{1}{x}}{4} \quad (4.4)$$

Note that special care needs to be taken to place parentheses and brackets around fractions. Doing things the obvious way produces brackets that are too small:

```
r'$(\frac{5 - \frac{1}{x}}{4})$'
```

$$\left(\frac{5 - \frac{1}{x}}{4}\right) \quad (4.5)$$

The solution is to precede the bracket with `\left` and `\right` to inform the parser that those brackets encompass the entire object:

```
r'$\left(\frac{5 - \frac{1}{x}}{4}\right)$'
```

$$\left(\frac{5 - \frac{1}{x}}{4}\right) \quad (4.6)$$

4.6.3 Radicals

Radicals can be produced with the `\sqrt[]{} command. For example:`

```
r'$\sqrt{2}$'
```

$$\sqrt{2} \quad (4.7)$$

Any base can (optionally) be provided inside square brackets. Note that the base must be a simple expression, and can not contain layout commands such as fractions or sub/superscripts:

```
r'$\sqrt[3]{x}$'
```

$$\sqrt[3]{x} \quad (4.8)$$

4.6.4 Fonts

The default font is *italics* for mathematical symbols.

Note: This default can be changed using the `mathtext.default` rcParam. This is useful, for example, to use the same font as regular non-math text for math text, by setting it to `regular`.

To change fonts, e.g., to write “sin” in a Roman font, enclose the text in a font command:

```
r'$s(t) = \mathcal{A}\mathrm{sin}(2 \omega t)$'
```

$$s(t) = \mathcal{A} \sin(2\omega t) \quad (4.9)$$

More conveniently, many commonly used function names that are typeset in a Roman font have shortcuts. So the expression above could be written as follows:

```
r'$s(t) = \mathcal{A}\sin(2 \omega t)$'
```

$$s(t) = \mathcal{A} \sin(2\omega t) \quad (4.10)$$

Here “s” and “t” are variable in italics font (default), “sin” is in Roman font, and the amplitude “A” is in calligraphy font. Note in the example above the calligraphy A is squished into the sin. You can use a spacing command to add a little whitespace between them:

```
s(t) = \mathcal{A} \sin(2 \omega t)
```

$$s(t) = \mathcal{A} \sin(2\omega t) \quad (4.11)$$

The choices available with all fonts are:

Command	Result
<code>\mathrm{Roman}</code>	Roman
<code>\mathit{Italic}</code>	<i>Italic</i>
<code>\mathtt{Typewriter}</code>	Typewriter
<code>\mathcal{CALLIGRAPHY}</code>	<i>CALLIGRAPHY</i>

When using the `STIX` fonts, you also have the choice of:

Command	Result
<code>\mathbb{blackboard}</code>	\mathbb{C}
<code>\mathrm{\mathbb{blackboard}}</code>	\mathbb{C}
<code>\mathfrak{Fraktur}</code>	\mathfrak{F}
<code>\mathsf{sansserif}</code>	S
<code>\mathrm{\mathsf{sansserif}}</code>	S

There are also three global “font sets” to choose from, which are selected using the `mathtext.fontset` parameter in `matplotlibrc`.

`cm`: Computer Modern (TeX)

$$\mathcal{R} \prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)$$

`stix`: STIX (designed to blend well with Times)

$$\mathcal{R} \prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)$$

`stixsans`: STIX sans-serif

$$\mathcal{R} \prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)$$

Additionally, you can use `\mathdefault{...}` or its alias `\mathregular{...}` to use the font used for regular text outside of mathtext. There are a number of limitations to this approach, most notably that far fewer symbols will be available, but it can be useful to make math expressions blend well with other text in the plot.

Custom fonts

mathtext also provides a way to use custom fonts for math. This method is fairly tricky to use, and should be considered an experimental feature for patient users only. By setting the rcParam `mathtext.fontset` to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

Parameter	Corresponds to
<code>mathtext.it</code>	<code>\mathit{}</code> or default italic
<code>mathtext.rm</code>	<code>\mathrm{}</code> Roman (upright)
<code>mathtext.tt</code>	<code>\mathtt{}</code> Typewriter (monospace)
<code>mathtext.bf</code>	<code>\mathbf{}</code> bold italic
<code>mathtext.cal</code>	<code>\mathcal{}</code> calligraphic
<code>mathtext.sf</code>	<code>\mathsf{}</code> sans-serif

Each parameter should be set to a fontconfig font descriptor (as defined in the yet-to-be-written font chapter).

The fonts used should have a Unicode mapping in order to find any non-Latin characters, such as Greek. If you want to use a math symbol that is not contained in your custom fonts, you can set the rcParam `mathtext.fallback_to_cm` to `True` which will cause the mathtext system to use characters from the default Computer Modern fonts whenever a particular character can not be found in the custom font.

Note that the math glyphs specified in Unicode have evolved over time, and many fonts may not have glyphs in the correct place for mathtext.

4.6.5 Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

Command	Result
\acute{a} or \'a	á
\bar{a}	ā
\breve{a}	ă
\ddot{a} or \"a	ä
\dot{a} or \.{a}	à
\grave{a} or \`{a}	à
\hat{a} or \^a	â
\tilde{a} or \~{a}	ã
\vec{a}	ã
\overline{abc}	abc

In addition, there are two special accents that automatically adjust to the width of the symbols below:

Command	Result
\widehat{xyz}	xyz
\widetilde{xyz}	xyz

Care should be taken when putting accents on lower-case i's and j's. Note that in the following \imath is used to avoid the extra dot over the i:

```
r"\hat{i} \ \ \hat{\imath} \imath"
```

$$\hat{i} \ \hat{\imath} \imath \quad (4.12)$$

4.6.6 Symbols

You can also use a large number of the TeX symbols, as in \infty, \leftarrow, \sum, \int.

Lower-case Greek

α \alpha	β \beta	χ \chi	δ \delta	F \digamma
ϵ \epsilon	η \eta	γ \gamma	ι \iota	κ \kappa
λ \lambda	μ \mu	ν \nu	ω \omega	ϕ \phi
π \pi	ψ \psi	ρ \rho	σ \sigma	τ \tau
θ \theta	υ \upsilon	ε \varepsilon	\varkappa \varkappa	φ \varphi
ϖ \varpi	ϱ \varrho	ς \varsigma	ϑ \vartheta	ξ \xi
ζ \zeta				

Upper-case Greek

Δ \Delta	Γ \Gamma	Λ \Lambda	Ω \Omega	Φ \Phi	Π \Pi
Ψ \Psi	Σ \Sigma	Θ \Theta	Υ \Upsilon	Ξ \Xi	Υ \Upsilon
∇ \nabla					

Hebrew

```
aleph \aleph beth \beth daleth \daleth gimmel \gimmel
```

Delimiters

//	[[↓\Downarrow	↑\Uparrow	\Vert	\backslash
↓\downarrow	<\langle	⌈\lceil	⌊\lfloor	✉\llcorner	✉\lrcorner
)\rangle] \rceil	⌋\rfloor	✉\ulcorner	↑\uparrow	↑\urcorner
\vert	{ \{	\	} \}]]	

Big symbols

∩ \bigcap	∪ \bigcup	○ \bigodot	⊕ \bigoplus	⊗ \bigotimes
⊕ \biguplus	∨ \bigvee	∧ \bigwedge	⊔ \coprod	∫ \int
∮ \oint	∏ \prod	Σ \sum		

Standard function names

Pr \Pr	arccos \arccos	arcsin \arcsin	arctan \arctan
arg \arg	cos \cos	cosh \cosh	cot \cot
coth \coth	csc \csc	deg \deg	det \det
dim \dim	exp \exp	gcd \gcd	hom \hom
inf \inf	ker \ker	lg \lg	lim \lim
lim inf \liminf	lim sup \limsup	ln \ln	log \log
max \max	min \min	sec \sec	sin \sin
sinh \sinh	sup \sup	tan \tan	tanh \tanh

Binary operation and relation symbols

≈ \Bumpeq	⊩ \Cap	⊪ \Cup
÷ \Doteq	⊲ \Join	⊴ \Subset
⊸ \Supset	⊵ \Vdash	⊶ \Vdash
≈ \approx	≈ \approxeq	* \ast
× \asymp	⊸ \backepsilon	⊷ \backsim
⊜ \backsimeq	⊸ \barwedge	∴ \because
∅ \between	○ \bigcirc	▽ \bigtriangledown
△ \bigtriangleup	◀ \blacktriangleleft	▶ \blacktriangleright
⊥ \bot	⊲ \bowtie	⊻ \boxdot
⊤ \boxminus	⊦ \boxplus	⊸ \boxtimes
• \bullet	⊜ \bumpeq	⊓ \cap
· \cdot	○ \circ	⊝ \circeq
:- \coloneq	⊗ \cong	⊔ \cup
≤ \curlyeqprec	≥ \curlyeqsucc	∨ \curlyvee
∧ \curlywedge	† \dag	+ \dashv
‡ \ddag	◊ \diamond	÷ \div
* \divideontimes	÷ \doteq	÷ \doteqdot
+ \dotplus	⊸ \doublebarwedge	= \eqcirc
-: \eqcolon	≈ \eqsim	≥ \eqslantgtr
≤ \eqslantless	≡ \equiv	⊝ \fallingdotseq

\frown	\geq	\geqq
\geqslant	\gg	\ggg
\gnapprox	\gneqq	\gnsim
\gtrapprox	\gtrdot	\gtreqless
\gtreqless	\gtrless	\gtrsim
\in	\intercal	\leftthreetimes
\leq	\leqq	\leqslant
\lessapprox	\lessdot	\lesseqgtr
\lesseqgtr	\lessgtr	\lessim
\ll	\lll	\lnapprox
\lneqq	\lnsim	\ltimes
\mid	\models	\mp
\nDash	\nDash	\napprox
\ncong	\neq	\neq
\neq	\nequiv	\ngeq
\ngtr	\ni	\nleq
\nless	\nmid	\notin
\nparallel	\nprec	\nsim
\nssubset	\nsseteq	\nsucc
\nsupset	\nsupseteq	\ntriangleleft

\trianglelefteq	\triangleright	\trianglerighteq
\nvDash	\nvDash	\odot
\ominus	\oplus	\oslash
\otimes	\parallel	\perp
\pitchfork	\pm	\prec
\precapprox	\preccurlyeq	\preceq
\precnapprox	\precnsim	\precsim
\propto	\rightthreetimes	\risingdotseq
\rtimes	\sim	\simeq
\slash	\smile	\sqcap
\sqcup	\sqsubset	\sqsubset
\sqsubseteq	\sqsupset	\sqsupset
\sqsupseteq	\star	\subset
\subseteq	\subseteqqq	\subsetneq
\subsetneqq	\succ	\succapprox
\succcurlyeq	\succeq	\succnapprox
\succnsim	\succsim	\supset
\supseteq	\supseteqqq	\supsetneq
\supsetneqq	\therefore	\times
\top	\triangleleft	\trianglelefteq
\triangleq	\triangleright	\trianglerighteq
\uplus	\vDash	\varpropto
\vartriangleleft	\vartriangleright	\vdash
\vee	\veebar	\wedge
\wr		

Arrow symbols

$\Downarrow \Downarrow$	$\Leftarrow \Leftarrow$
$\Leftrightarrow \Leftrightarrow$	$\Lleftarrow \Lleftarrow$
$\Longleftarrow \Longleftarrow$	$\Longleftarrow \Longleftarrow$
$\Longrightarrow \Longrightarrow$	$\Lsh \Lsh$
$\nearrow \nearrow$	$\nwarrow \nwarrow$
$\Rightarrow \Rightarrow$	$\Rrightarrow \Rrightarrow$
$\Rsh \Rsh$	$\Searrow \Searrow$
$\swarrow \swarrow$	$\Uparrow \Uparrow$
$\Updownarrow \Updownarrow$	$\circlearrowleft \circlearrowleft$
$\circlearrowright \circlearrowright$	$\curvearrowleft \curvearrowleft$
$\curvearrowright \curvearrowright$	$\dashleftarrow \dashleftarrow$
$\dashrightarrow \dashrightarrow$	$\downarrow \downarrow$
$\downdownarrows \downdownarrows$	$\downarrow \downharpoonleft \downarrow \downharpoonleft$
$\downharpoonright \downharpoonright$	$\hookleftarrow \hookleftarrow$
$\hookrightarrow \hookrightarrow$	$\leadsto \leadsto$
$\leftarrow \leftarrow$	$\leftarrowtail \leftarrowtail$
$\leftharpoondown \leftharpoondown$	$\leftharpoonup \leftharpoonup$
$\leftleftarrows \leftleftarrows$	$\leftrightarrows \leftrightarrows$
$\leftrightsquigarrow \leftrightsquigarrow$	$\leftrightharpoons \leftrightharpoons$
$\rightsquigarrow \rightsquigarrow$	$\leftsquigarrow \leftsquigarrow$

$\longleftarrow \longleftarrow$	$\longrightarrow \longrightarrow$
$\longmapsto \longmapsto$	$\longrightarrow \longrightarrow$
$\looparrowleft \looparrowleft$	$\looparrowright \looparrowright$
$\mapsto \mapsto$	$\multimap \multimap$
$\nLeftarrow \nLeftarrow$	$\nLeftrightarrow \nLeftrightarrow$
$\nRightarrow \nRightarrow$	$\nearrow \nearrow$
$\nleftarrow \nleftarrow$	$\nleftrightarrow \nleftrightarrow$
$\nrightarrow \nrightarrow$	$\nwarrow \nwarrow$
$\rightarrow \rightarrow$	$\rightarrowtail \rightarrowtail$
$\rightharpoondown \rightharpoondown$	$\rightharpoonup \rightharpoonup$
$\rightleftarrows \rightleftarrows$	$\rightleftarrows \rightleftarrows$
$\rightleftharpoons \rightleftharpoons$	$\rightleftharpoons \rightleftharpoons$
$\rightrightarrows \rightrightarrows$	$\rightrightarrows \rightrightarrows$
$\rightsquigarrow \rightsquigarrow$	$\searrow \searrow$
$\swarrow \swarrow$	$\rightarrow \rightarrow$
$\twoheadleftarrow \twoheadleftarrow$	$\twoheadrightarrow \twoheadrightarrow$
$\uparrow \uparrow$	$\downarrow \downarrow$
$\updownarrow \updownarrow$	$\upharpoonleft \upharpoonleft$
$\upharpoonright \upharpoonright$	$\upuparrows \upuparrows$



Miscellaneous symbols

$\$ \backslash \$$	$\AA \backslash \text{AA}$	$\exists \backslash \text{Finv}$
$\eth \backslash \text{Game}$	$\Im \backslash \text{Im}$	$\P \backslash \text{P}$
$\Re \backslash \text{Re}$	$\S \backslash \text{S}$	$\angle \backslash \text{angle}$
$\flat \backslash \text{backprime}$	$\star \backslash \text{bigstar}$	$\blacksquare \backslash \text{blacksquare}$
$\blacktriangle \backslash \text{blacktriangle}$	$\blacktriangledown \backslash \text{blacktriangledown}$	$\cdots \backslash \text{cdots}$
$\checkmark \backslash \text{checkmark}$	$\circledR \backslash \text{circledR}$	$\circledS \backslash \text{circledS}$
$\clubsuit \backslash \text{clubsuit}$	$\complement \backslash \text{complement}$	$\circledC \backslash \text{copyright}$
$\ddots \backslash \text{ddots}$	$\diamondsuit \backslash \text{diamondsuit}$	$\ell \backslash \text{ell}$
$\emptyset \backslash \text{emptyset}$	$\eth \backslash \text{eth}$	$\exists \backslash \text{exists}$
$\flat \backslash \text{flat}$	$\forall \backslash \text{forall}$	$\hbar \backslash \text{hbar}$
$\heartsuit \backslash \text{heartsuit}$	$\hslash \backslash \text{hslash}$	$\iiint \backslash \text{iiint}$
$\iint \backslash \text{iint}$	$\iint \backslash \text{iint}$	$\imath \backslash \text{imath}$
$\infty \backslash \text{infty}$	$\jmath \backslash \text{jmath}$	$\ldots \backslash \text{ldots}$
$\measuredangle \backslash \text{measuredangle}$	$\natural \backslash \text{natural}$	$\neg \backslash \text{neg}$
$\nexists \backslash \text{nexists}$	$\oiint \backslash \text{oiint}$	$\partial \backslash \text{partial}$
$\prime \backslash \text{prime}$	$\sharp \backslash \text{sharp}$	$\spadesuit \backslash \text{spadesuit}$
$\sphericalangle \backslash \text{sphericalangle}$	$\ss \backslash \text{ss}$	$\triangledown \backslash \text{triangledown}$
$\varnothing \backslash \text{varnothing}$	$\vartriangle \backslash \text{vartriangle}$	$\vdots \backslash \text{vdots}$
$\wp \backslash \text{wp}$	$\yen \backslash \text{yen}$	

If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

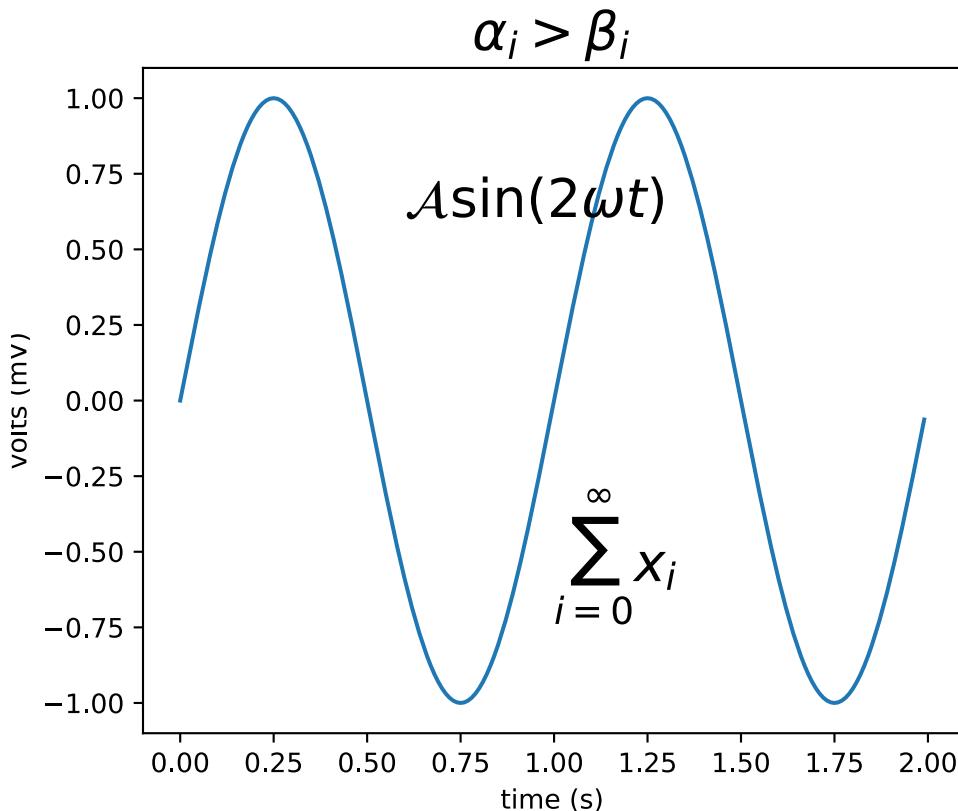
```
ur' $\u23ce$'
```

4.6.7 Example

Here is an example illustrating many of these features in context.

```
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)

plt.plot(t,s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^{\infty} x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\sin(2 \omega t)$',
         fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
plt.show()
```



4.7 Text rendering With LaTeX

Matplotlib has the option to use LaTeX to manage all text layout. This option is available with the following backends:

- Agg
- PS
- PDF

The LaTeX option is activated by setting `text.usetex : True` in your rc settings. Text handling with matplotlib's LaTeX support is slower than matplotlib's very capable `mathtext`, but is more flexible, since different LaTeX packages (font packages, math packages, etc.) can be used. The results can be striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib's LaTeX support requires a working `LaTeX` installation, `dvipng` (which may be included with your LaTeX installation), and `Ghostscript` (GPL Ghostscript 8.60 or later is recommended). The executables for these external dependencies must all be located on your `PATH`.

There are a couple of options to mention, which can be changed using `rc settings`. Here is an example `matplotlibrc` file:

```

font.family      : serif
font.serif       : Times, Palatino, New Century Schoolbook, Bookman, Computer Modern
  ↵Roman
font.sans-serif : Helvetica, Avant Garde, Computer Modern Sans serif
font.cursive     : Zapf Chancery
font.monospace   : Courier, Computer Modern Typewriter

text.usetex      : true

```

The first valid font in each family is the one that will be loaded. If the fonts are not specified, the Computer Modern fonts are used by default. All of the other fonts are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts. See the [PSNFSS](#) documentation for more details.

To use LaTeX and select Helvetica as the default font, without editing matplotlibrc use:

```

from matplotlib import rc
rc('font', **{'family':'sans-serif','sans-serif':['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font', **{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

```

Here is the standard example, `tex_demo.py`:

```

"""
Demo of TeX rendering.

You can use TeX to render all of your matplotlib text if the rc
parameter text.usetex is set. This works currently on the agg and ps
backends, and requires that you have tex and the other dependencies
described at http://matplotlib.org/users/usetex.html
properly installed on your system. The first time you run a script
you will see a lot of output from tex and associated tools. The next
time, the run may be silent, as a lot of the information is cached in
~/.tex.cache

"""

import numpy as np
import matplotlib.pyplot as plt

# Example data
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * t) + 2

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.plot(t, s)

plt.xlabel(r'\textbf{time} (s)')
plt.ylabel(r'\textit{voltage} (mV)', fontsize=16)
plt.title(r"\TeX\ is Number "
          r"\$\\displaystyle\\sum_{n=1}^{\\infty}\\frac{(-e^{i\\pi})^n}{2^n}\$!")

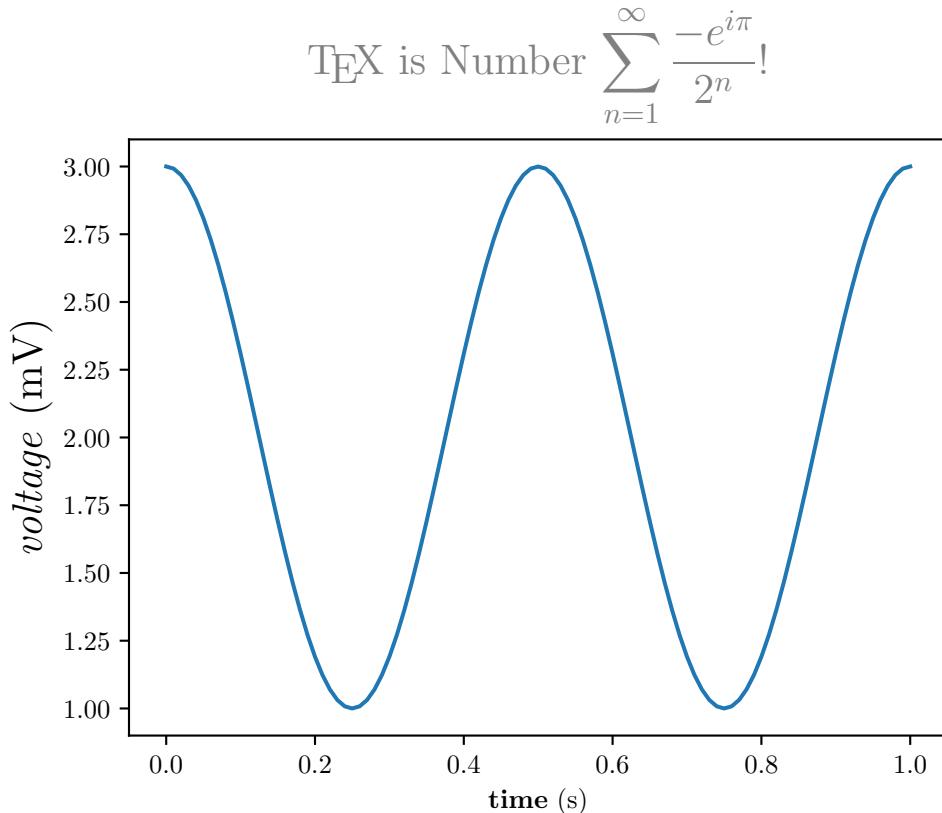
```

```

    fontsize=16, color='gray')
# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)

plt.savefig('tex_demo')
plt.show()

```



Note that display math mode (\$\$ e=mc^2 \$\$) is not supported, but adding the command `\displaystyle`, as in `tex_demo.py`, will produce the same results.

Note: Certain characters require special escaping in TeX, such as:

```
# $ % & ~ _ ^ { } \{ \} \{ \} \[ \]
```

Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag.

4.7.1 usetex with unicode

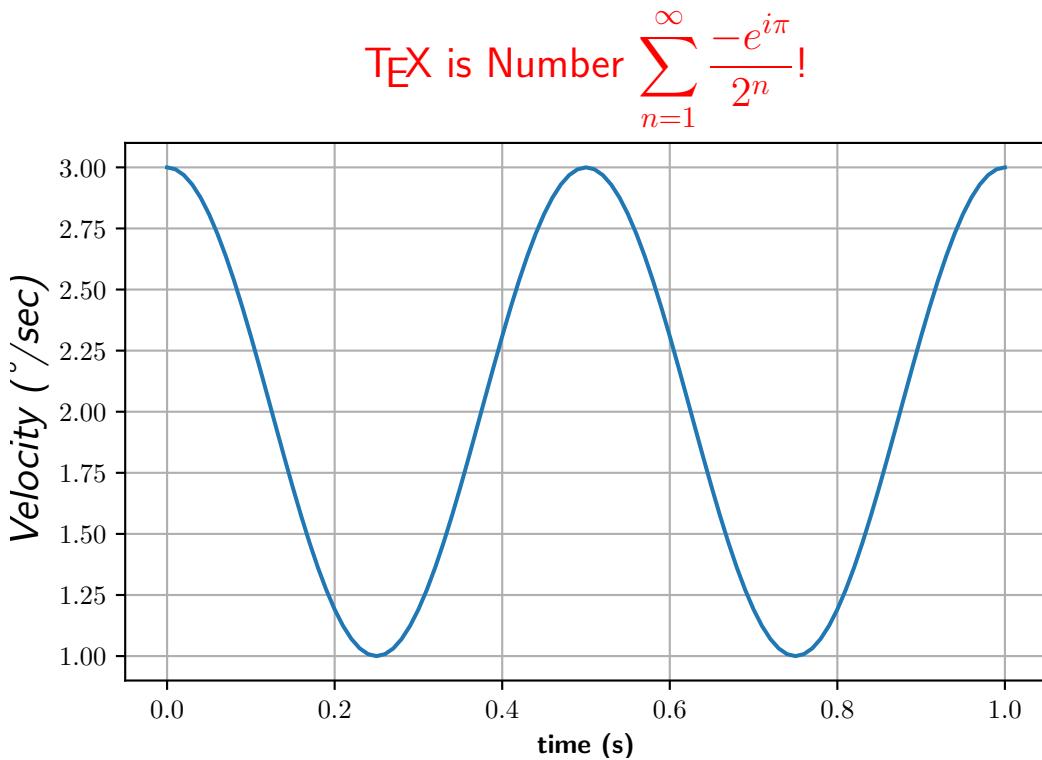
It is also possible to use unicode strings with the LaTeX text manager, here is an example taken from `tex_unicode_demo.py`:

```
# -*- coding: utf-8 -*-
"""
This demo is tex_demo.py modified to have unicode. See that file for
more information.
"""

from __future__ import unicode_literals
import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
matplotlib.rcParams['text.latex.unicode'] = True
import matplotlib.pyplot as plt

plt.figure(1, figsize=(6, 4))
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*np.pi*t) + 2
plt.plot(t, s)

plt.xlabel(r'\textbf{time (s)}')
plt.ylabel('Velocity (\u00b0/sec)', fontsize=16)
plt.title(r'\TeX\ is Number $\displaystyle\sum_{n=1}^{\infty}\frac{-e^{i\pi}}{2^n}$',
          fontsize=16, color='r')
plt.grid(True)
plt.show()
```



4.7.2 Postscript options

In order to produce encapsulated postscript files that can be embedded in a new LaTeX document, the default behavior of matplotlib is to distill the output, which removes some postscript operators used by LaTeX that are illegal in an eps file. This step produces results which may be unacceptable to some users, because the text is coarsely rasterized and converted to bitmaps, which are not scalable like standard postscript, and the text is not searchable. One workaround is to set `ps.distiller.res` to a higher value (perhaps 6000) in your rc settings, which will produce larger files but may look better and scale reasonably. A better workaround, which requires [Poppler](#) or [Xpdf](#), can be activated by changing the `ps.usedistiller` rc setting to `xpdf`. This alternative produces postscript without rasterizing text, so it scales properly, can be edited in Adobe Illustrator, and searched text in pdf documents.

4.7.3 Possible hangups

- On Windows, the `PATH` environment variable may need to be modified to include the directories containing the `latex`, `dvipng` and `ghostscript` executables. See [Environment Variables](#) and [Setting environment variables in windows](#) for details.
- Using MiKTeX with Computer Modern fonts, if you get odd *Agg and PNG results, go to MiKTeX/Options and update your format files
- The fonts look terrible on screen. You are probably running Mac OS, and there is some funny business with older versions of `dvipng` on the mac. Set `text.dvipnghack : True` in your `matplotlibrc` file.
- On Ubuntu and Gentoo, the base texlive install does not ship with the `type1cm` package. You may need to install some of the extra packages to get all the goodies that come bundled with other latex distributions.
- Some progress has been made so matplotlib uses the `dvi` files directly for text layout. This allows `latex` to be used for text layout with the `pdf` and `svg` backends, as well as the *Agg and PS backends. In the future, a `latex` installation may be the only external dependency.

4.7.4 Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see [matplotlib configuration and cache directory locations](#).
- Make sure LaTeX, dvipng and ghostscript are each working and on your `PATH`.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- Most problems reported on the mailing list have been cleared up by upgrading [Ghostscript](#). If possible, please try upgrading to the latest release before reporting problems to the list.
- The `text.latex.preamble` rc setting is not officially supported. This option provides lots of flexibility, and lots of ways to cause problems. Please disable this option before reporting problems to the mailing list.
- If you still need help, please see [Getting help](#)

4.8 Typesetting With XeLaTeX/LuaLaTeX

Using the pgf backend, matplotlib can export figures as pgf drawing commands that can be processed with pdflatex, xelatex or lualatex. XeLaTeX and LuaLaTeX have full unicode support and can use any font that is installed in the operating system, making use of advanced typographic features of OpenType, AAT and Graphite. Pgf pictures created by plt.savefig('figure.pgf') can be embedded as raw commands in LaTeX documents. Figures can also be directly compiled and saved to PDF with plt.savefig('figure.pdf') by either switching to the backend

```
matplotlib.use('pgf')
```

or registering it for handling pdf output

```
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
```

The second method allows you to keep using regular interactive backends and to save xelatex, lualatex or pdflatex compiled PDF files from the graphical user interface.

Matplotlib's pgf support requires a recent [TeX](#) installation that includes the TikZ/PGF packages (such as [TeXLive](#)), preferably with XeLaTeX or LuaLaTeX installed. If either pdftocairo or ghostscript is present on your system, figures can optionally be saved to PNG images as well. The executables for all applications must be located on your [PATH](#).

Rc parameters that control the behavior of the pgf backend:

Parameter	Documentation
pgf.preamble	Lines to be included in the LaTeX preamble
pgf.rcfonts	Setup fonts from rc params using the fontspec package
pgf.texsystem	Either “xelatex” (default), “lualatex” or “pdflatex”

Note: TeX defines a set of special characters, such as:

```
# $ % & ~ _ ^ \ { }
```

Generally, these characters must be escaped correctly. For convenience, some characters (_,%^) are automatically escaped outside of math environments.

4.8.1 Font specification

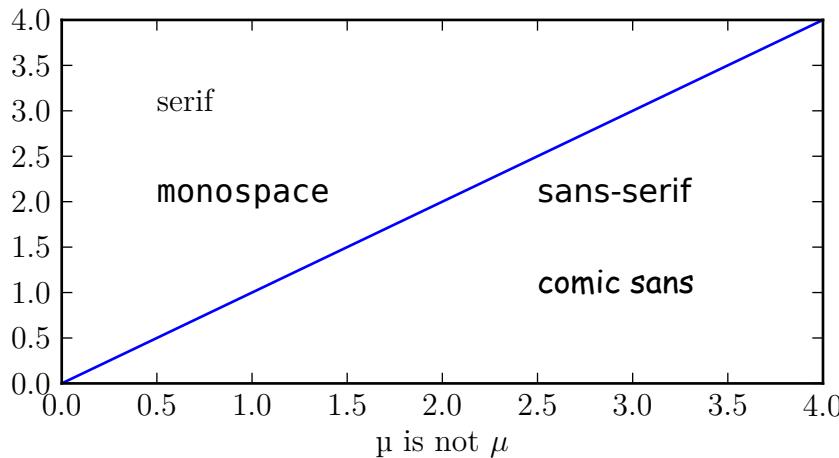
The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the matplotlib rc parameters. You can also use the LaTeX default Computer Modern fonts by clearing the lists for font.serif, font.sans-serif or font.monospace. Please note that the glyph coverage of these fonts is very limited. If you want to keep the Computer Modern font face but require extended unicode support, consider installing the [Computer Modern Unicode](#) fonts *CMU Serif*, *CMU Sans Serif*, etc.

When saving to .pgf, the font configuration matplotlib used for the layout of the figure is included in the header of the text file.

```
# -*- coding: utf-8 -*-

import matplotlib as mpl
mpl.use("pgf")
pgf_with_rc_fonts = {
    "font.family": "serif",
    "font.serif": [],                      # use latex default serif font
    "font.sans-serif": ["DejaVu Sans"],      # use a specific sans-serif font
}
mpl.rcParams.update(pgf_with_rc_fonts)

import matplotlib.pyplot as plt
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.text(2.5, 1., "comic sans", family="Comic Sans MS")
plt.xlabel(u"\u03bc is not \$\\mu$")
plt.tight_layout(.5)
```



4.8.2 Custom preamble

Full customization is possible by adding your own commands to the preamble. Use the `pgf.preamble` parameter if you want to configure the math fonts, using `unicode-math` for example, or for loading additional packages. Also, if you want to do the font configuration yourself instead of using the fonts specified in the rc parameters, make sure to disable `pgf.rcfonts`.

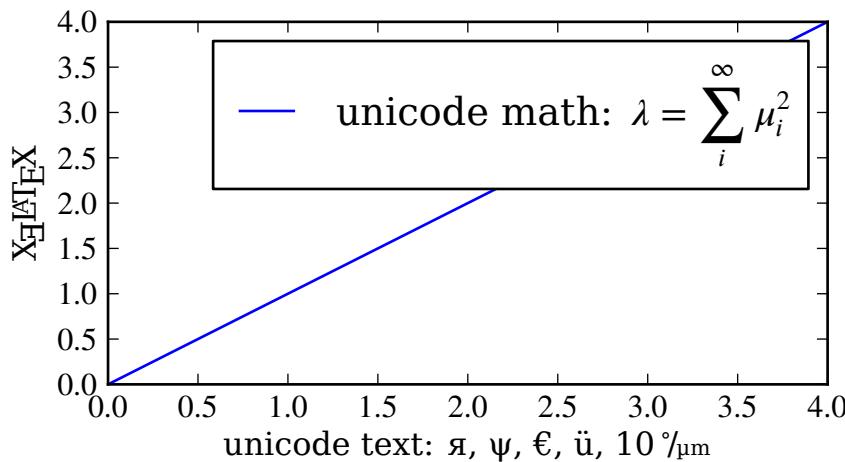
```
# -*- coding: utf-8 -*-
from __future__ import (absolute_import, division, print_function,
                       unicode_literals)

import six
```

```

import matplotlib as mpl
mpl.use("pgf")
pgf_with_custom_preamble = {
    "font.family": "serif", # use serif/main font for text elements
    "text.usetex": True,     # use inline math for ticks
    "pgf.rcfonts": False,   # don't setup fonts from rc parameters
    "pgf.preamble": [
        r"\usepackage{units}",           # load additional packages
        r"\usepackage{metabook}",
        r"\usepackage{unicode-math}",    # unicode math setup
        r"\setmathfont{xits-math.otf}",
        r"\setmainfont{DejaVu Serif}",   # serif font via preamble
    ]
}
mpl.rcParams.update(pgf_with_custom_preamble)

```



4.8.3 Choosing the TeX system

The TeX system to be used by matplotlib is chosen by the `pgf.texsystem` parameter. Possible values are '`xelatex`' (default), '`lualatex`' and '`pdflatex`'. Please note that when selecting `pdflatex` the fonts and unicode handling must be configured in the preamble.

```

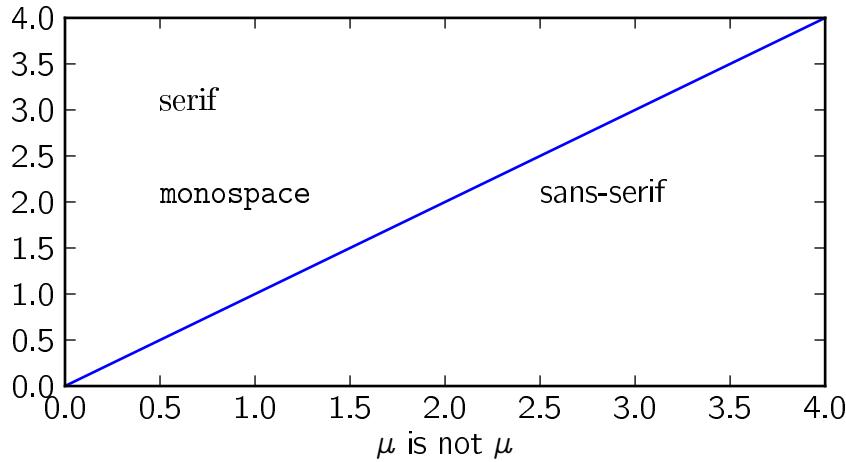
# -*- coding: utf-8 -*-

import matplotlib as mpl
mpl.use("pgf")
pgf_with_pdflatex = {
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": [
        r"\usepackage[utf8x]{inputenc}",
        r"\usepackage[T1]{fontenc}",
        r"\usepackage{cmbright}",
    ]
}

```

```
mpl.rcParams.update(pgf_with_pdflatex)

import matplotlib.pyplot as plt
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif", family="serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.xlabel(u"\u03bc is not \$\\mu$")
plt.tight_layout(.5)
```



4.8.4 Troubleshooting

- Please note that the TeX packages found in some Linux distributions and MiKTeX installations are dramatically outdated. Make sure to update your package catalog and upgrade or install a recent TeX distribution.
- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment Variables* and *Setting environment variables in windows* for details.
- A limitation on Windows causes the backend to keep file handles that have been opened by your application open. As a result, it may not be possible to delete the corresponding files until the application closes (see #1324).
- Sometimes the font rendering in figures that are saved to png images is very bad. This happens when the pdftocairo tool is not available and ghostscript is used for the pdf to png conversion.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- The pgf.preamble rc setting provides lots of flexibility, and lots of ways to cause problems. When experiencing problems, try to minimize or disable the custom preamble.

- Configuring an `unicode-math` environment can be a bit tricky. The TeXLive distribution for example provides a set of math fonts which are usually not installed system-wide. XeTeX, unlike LuaLatex, cannot find these fonts by their name, which is why you might have to specify `\setmathfont{xits-math.otf}` instead of `\setmathfont{XITS Math}` or alternatively make the fonts available to your OS. See this [tex.stackexchange.com question](#) for more details.
- If the font configuration used by matplotlib differs from the font setting in your LaTeX document, the alignment of text elements in imported figures may be off. Check the header of your `.pgf` file if you are unsure about the fonts matplotlib used for the layout.
- Vector images and hence `.pgf` files can become bloated if there are a lot of objects in the graph. This can be the case for image processing or very big scatter graphs. In an extreme case this can cause TeX to run out of memory: “TeX capacity exceeded, sorry” You can configure latex to increase the amount of memory available to generate the `.pdf` image as discussed on [tex.stackexchange.com](#). Another way would be to “rasterize” parts of the graph causing problems using either the `rasterized=True` keyword, or `.set_rasterized(True)` as per [this example](#).
- If you still need help, please see [Getting help](#)

COLORS

5.1 Specifying Colors

In almost all places in matplotlib where a color can be specified by the user it can be provided as:

- an RGB or RGBA tuple of float values in [0, 1] (e.g., (0.1, 0.2, 0.5) or (0.1, 0.2, 0.5, 0.3))
- a hex RGB or RGBA string (e.g., '#0F0F0F' or '#0F0F0F0F')
- a string representation of a float value in [0, 1] inclusive for gray level (e.g., '0.5')
- one of {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}
- a X11/CSS4 color name
- a name from the xkcd color survey prefixed with 'xkcd:' (e.g., 'xkcd:sky blue')
- one of {'C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'}
- one of {'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'} which are the Tableau Colors from the 'T10' categorical palette (which is the default color cycle).

All string specifications of color are case-insensitive.

5.1.1 'CN' color selection

Color can be specified by a string matching the regex C[0-9]. This can be passed any place that a color is currently accepted and can be used as a ‘single character color’ in format-string to `matplotlib.Axes.plot`.

The single digit is the index into the default property cycle (`matplotlib.rcParams['axes.prop_cycle']`). If the property cycle does not include 'color' then black is returned. The color is evaluated when the artist is created. For example,

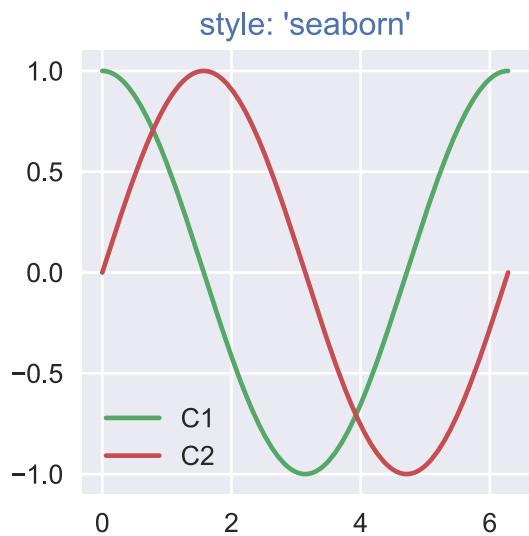
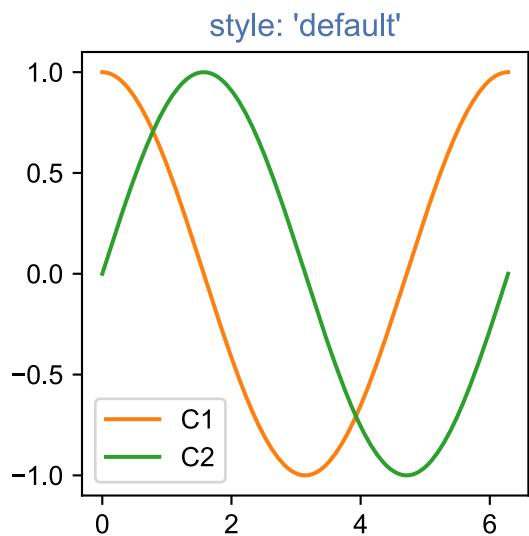
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
th = np.linspace(0, 2*np.pi, 128)
```

```
def demo(sty):
    mpl.style.use(sty)
    fig, ax = plt.subplots(figsize=(3, 3))

    ax.set_title('style: {!r}'.format(sty), color='C0')

    ax.plot(th, np.cos(th), 'C1', label='C1')
    ax.plot(th, np.sin(th), 'C2', label='C2')
    ax.legend()

demo('default')
demo('seaborn')
```



will use the first color for the title and then plot using the second and third colors of each style's `mpl.rcParams['axes.prop_cycle']`.

5.1.2 xkcd v X11/CSS4

The xkcd colors are derived from a user survey conducted by the webcomic xkcd. [Details of the survey are available on the xkcd blog.](#)

Out of 148 colors in the CSS color list, there are 95 name collisions between the X11/CSS4 names and the xkcd names, all but 3 of which have different hex values. For example 'blue' maps to '#0000FF' where as 'xkcd:blue' maps to '#0343DF'. Due to these name collisions all of the xkcd colors have 'xkcd:' prefixed. As noted in the blog post, while it might be interesting to re-define the X11/CSS4 names based on such a survey, we do not do so unilaterally.

The name collisions are shown in the table below; the color names where the hex values agree are shown in bold.



5.2 Choosing Colormaps

5.2.1 Overview

The idea behind choosing a good colormap is to find a good representation in 3D colorspace for your data set. The best colormap for any given data set depends on many things including:

- Whether representing form or metric data ([\[Ware\]](#))
- Your knowledge of the data set (*e.g.*, is there a critical value from which the other values deviate?)
- If there is an intuitive color scheme for the parameter you are plotting
- If there is a standard in the field the audience may be expecting

For many applications, a perceptually uniform colormap is the best choice — one in which equal steps in data are perceived as equal steps in the color space. Researchers have found that the human brain perceives changes in the lightness parameter as changes in the data much better than, for example, changes in hue. Therefore, colormaps which have monotonically increasing lightness through the colormap will be better interpreted by the viewer. A wonderful example of perceptually uniform colormaps is [\[colorcet\]](#).

Color can be represented in 3D space in various ways. One way to represent color is using CIELAB. In CIELAB, color space is represented by lightness, L^* ; red-green, a^* ; and yellow-blue, b^* . The lightness parameter L^* can then be used to learn more about how the matplotlib colormaps will be perceived by viewers.

An excellent starting resource for learning about human perception of colormaps is from [\[IBM\]](#).

5.2.2 Classes of colormaps

Colormaps are often split into several categories based on their function (see, *e.g.*, [\[Moreland\]](#)):

1. Sequential: change in lightness and often saturation of color incrementally, often using a single hue; should be used for representing information that has ordering.
2. Diverging: change in lightness and possibly saturation of two different colors that meet in the middle at an unsaturated color; should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.
3. Qualitative: often are miscellaneous colors; should be used to represent information which does not have ordering or relationships.

5.2.3 Lightness of matplotlib colormaps

Here we examine the lightness values of the matplotlib colormaps. Note that some documentation on the colormaps is available ([\[list-colormaps\]](#)).

Sequential

For the Sequential plots, the lightness value increases monotonically through the colormaps. This is good. Some of the L^* values in the colormaps span from 0 to 100 (binary and the other grayscale), and others start around $L^* = 20$. Those that have a smaller range of L^* will accordingly have a smaller perceptual range. Note also that the L^* function varies amongst the colormaps: some are approximately linear in L^* and others are more curved.

Sequential2

Many of the L^* values from the Sequential2 plots are monotonically increasing, but some (autumn, cool, spring, and winter) plateau or even go both up and down in L^* space. Others (afmhot, copper, gist_heat, and hot) have kinks in the L^* functions. Data that is being represented in a region of the colormap that is at a plateau or kink will lead to a perception of banding of the data in those values in the colormap (see [\[mycarta-banding\]](#) for an excellent example of this).

Diverging

For the Diverging maps, we want to have monotonically increasing L^* values up to a maximum, which should be close to $L^* = 100$, followed by monotonically decreasing L^* values. We are looking for approximately equal minimum L^* values at opposite ends of the colormap. By these measures, BrBG and RdBu are good options. coolwarm is a good option, but it doesn't span a wide range of L^* values (see grayscale section below).

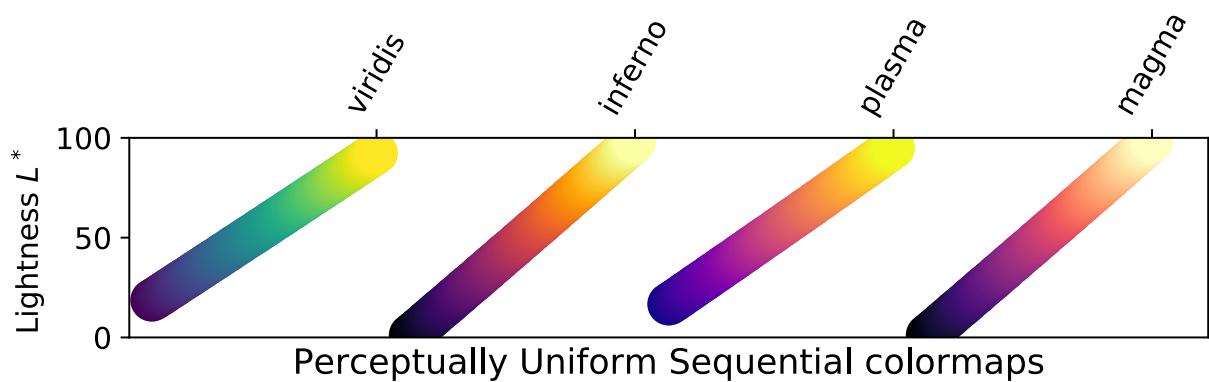
Qualitative

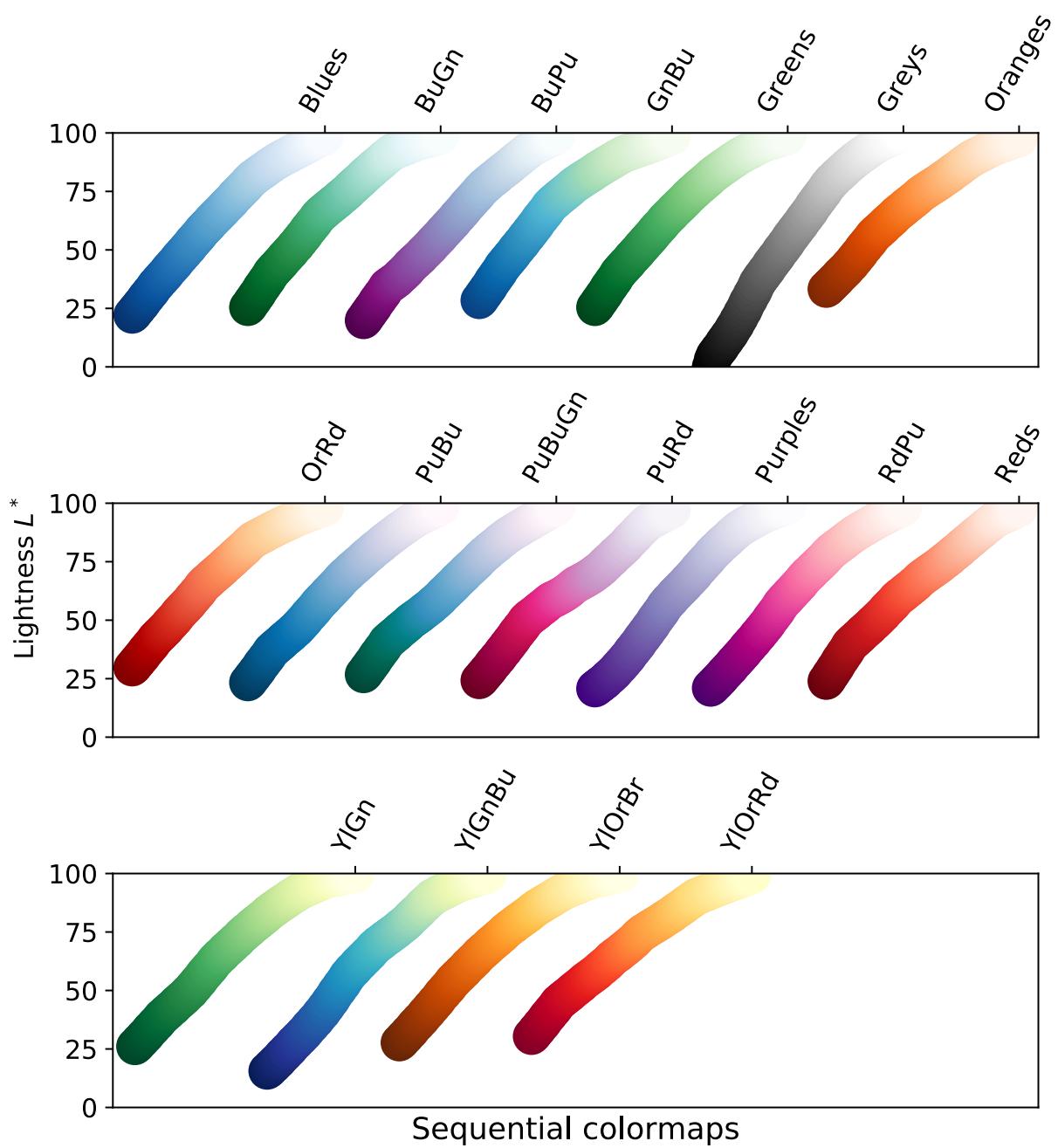
Qualitative colormaps are not aimed at being perceptual maps, but looking at the lightness parameter can verify that for us. The L^* values move all over the place throughout the colormap, and are clearly not monotonically increasing. These would not be good options for use as perceptual colormaps.

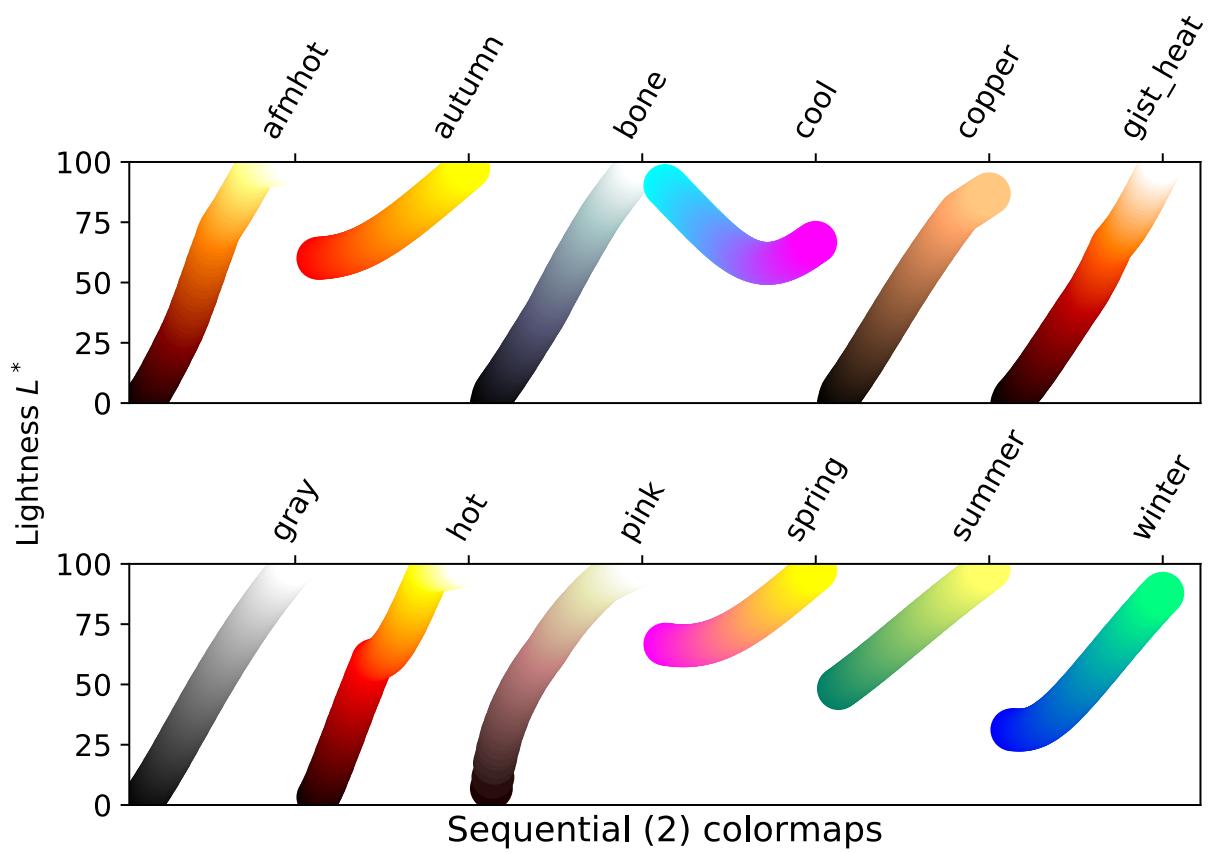
Miscellaneous

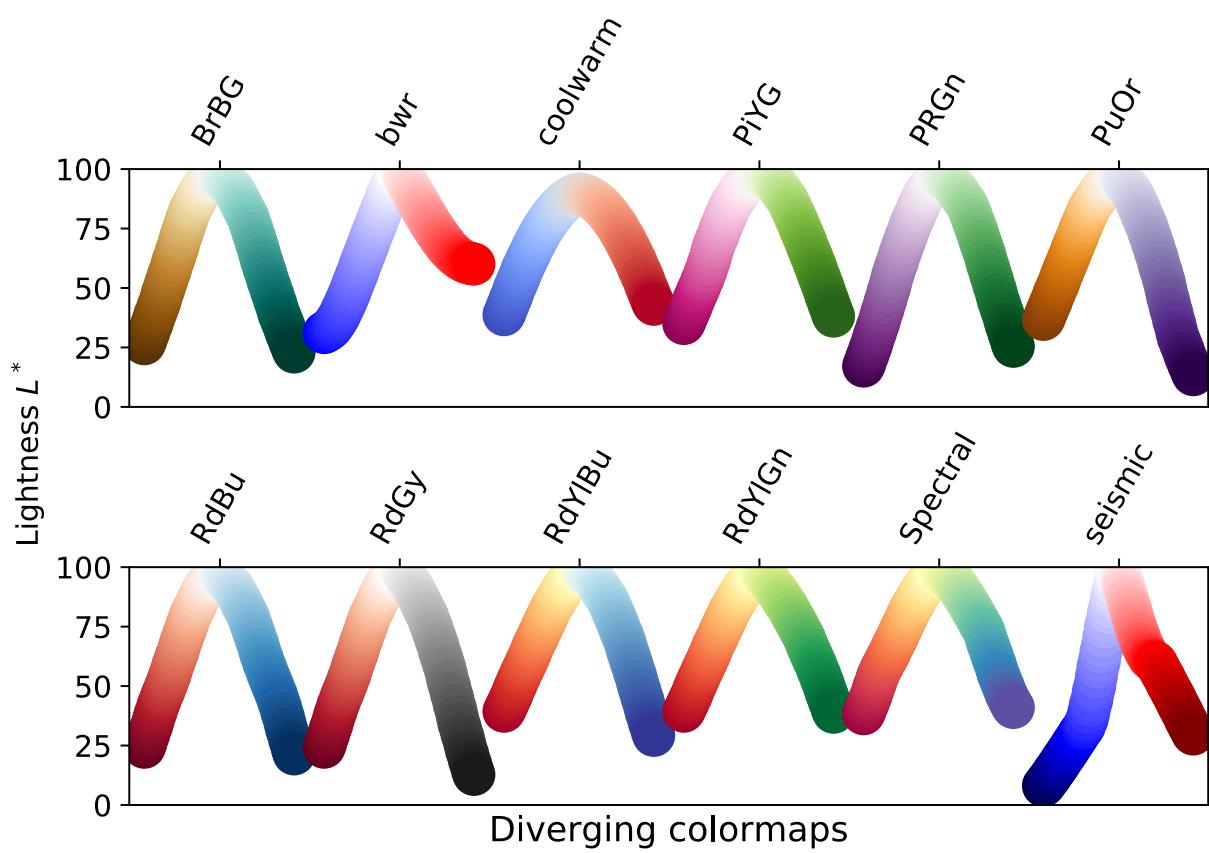
Some of the miscellaneous colormaps have particular uses for which they have been created. For example, gist_earth, ocean, and terrain all seem to be created for plotting topography (green/brown) and water depths (blue) together. We would expect to see a divergence in these colormaps, then, but multiple kinks may not be ideal, such as in gist_earth and terrain. CMRmap was created to convert well to grayscale, though it does appear to have some small kinks in L^* . cubehelix was created to vary smoothly in both lightness and hue, but appears to have a small hump in the green hue area.

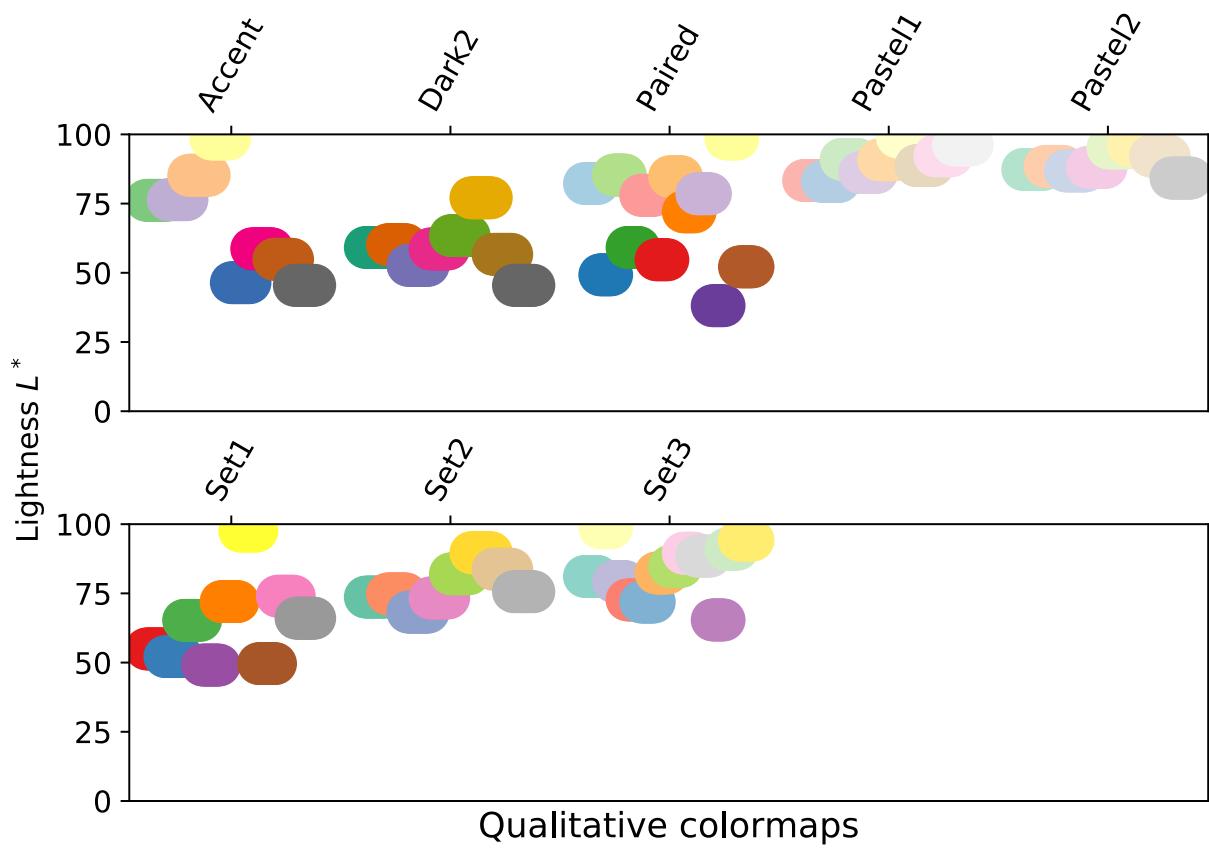
The often-used jet colormap is included in this set of colormaps. We can see that the L^* values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [\[mycarta-jet\]](#).

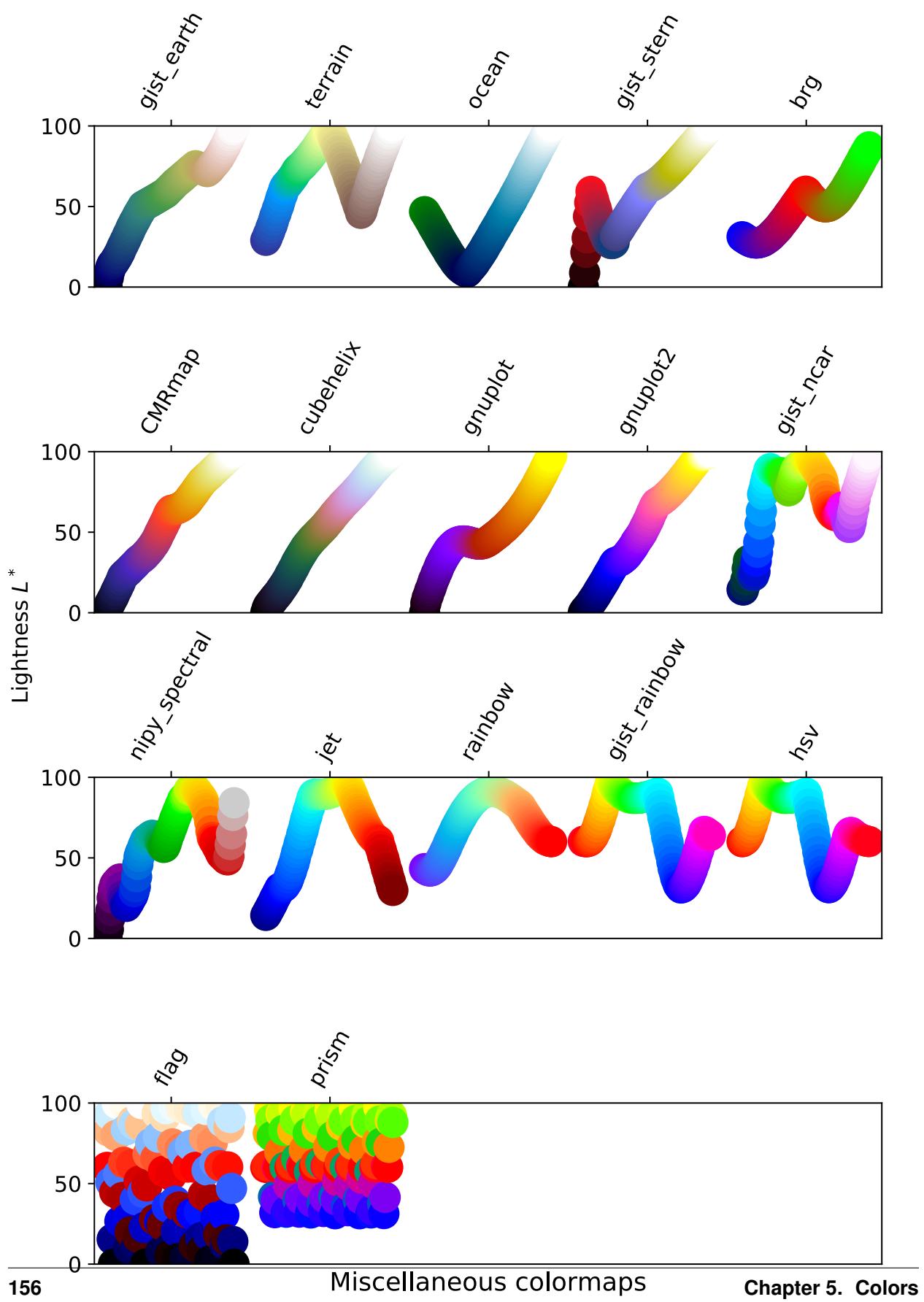










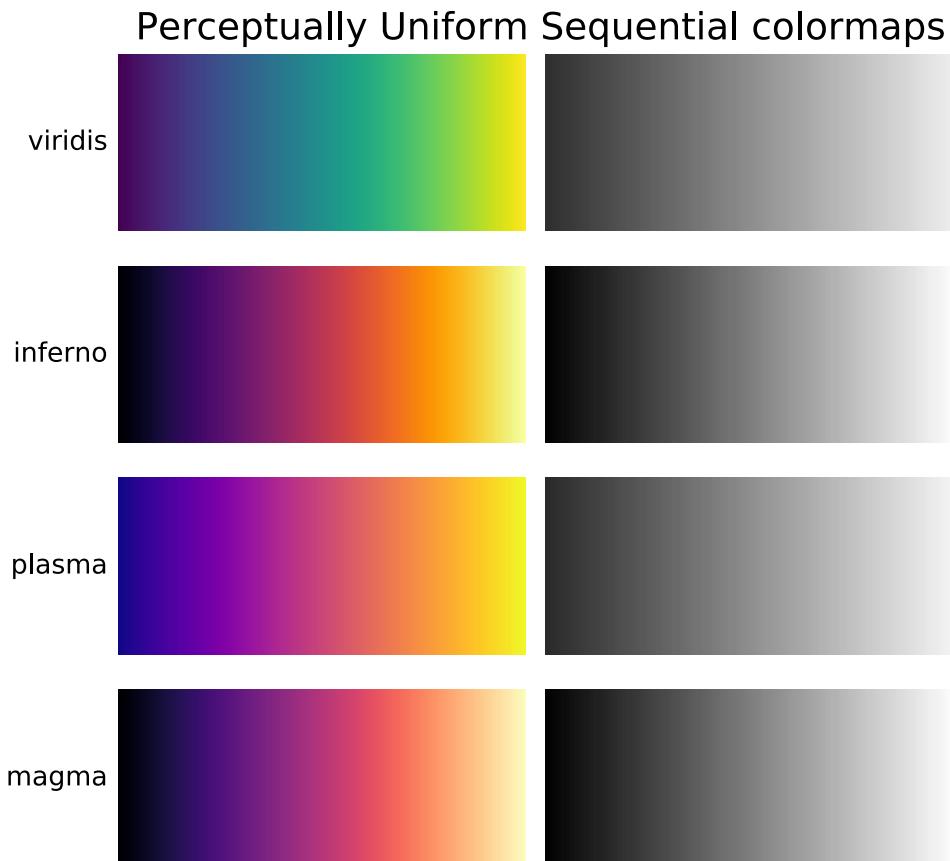


5.2.4 Grayscale conversion

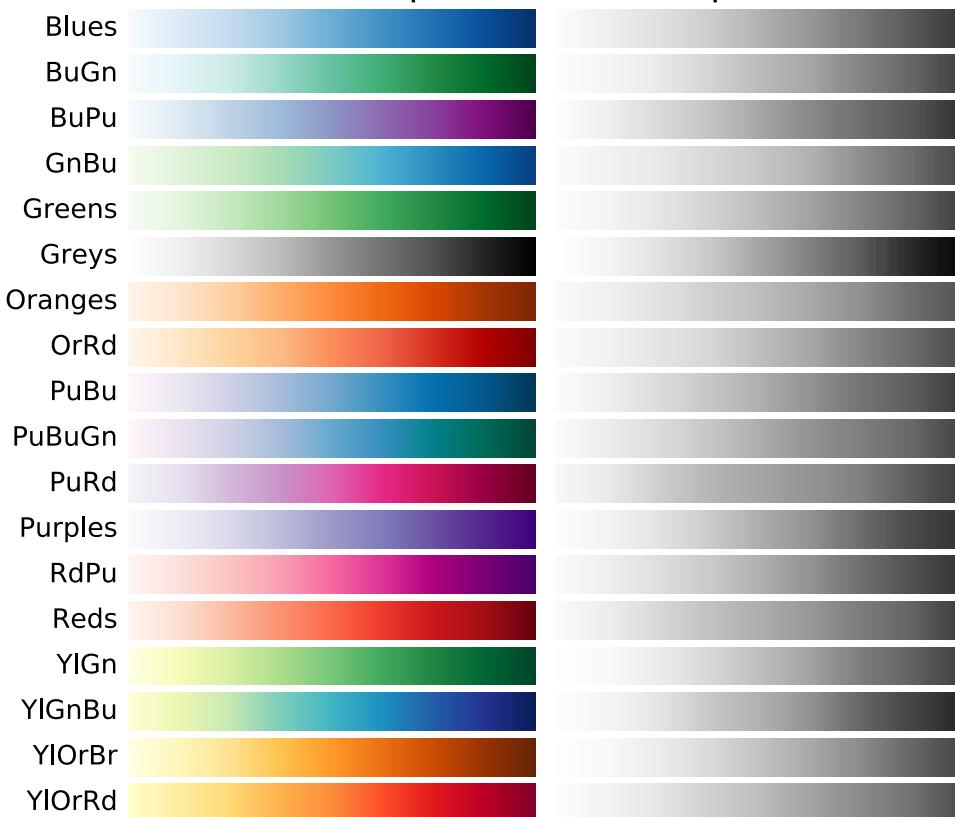
It is important to pay attention to conversion to grayscale for color plots, since they may be printed on black and white printers. If not carefully considered, your readers may end up with indecipherable plots because the grayscale changes unpredictably through the colormap.

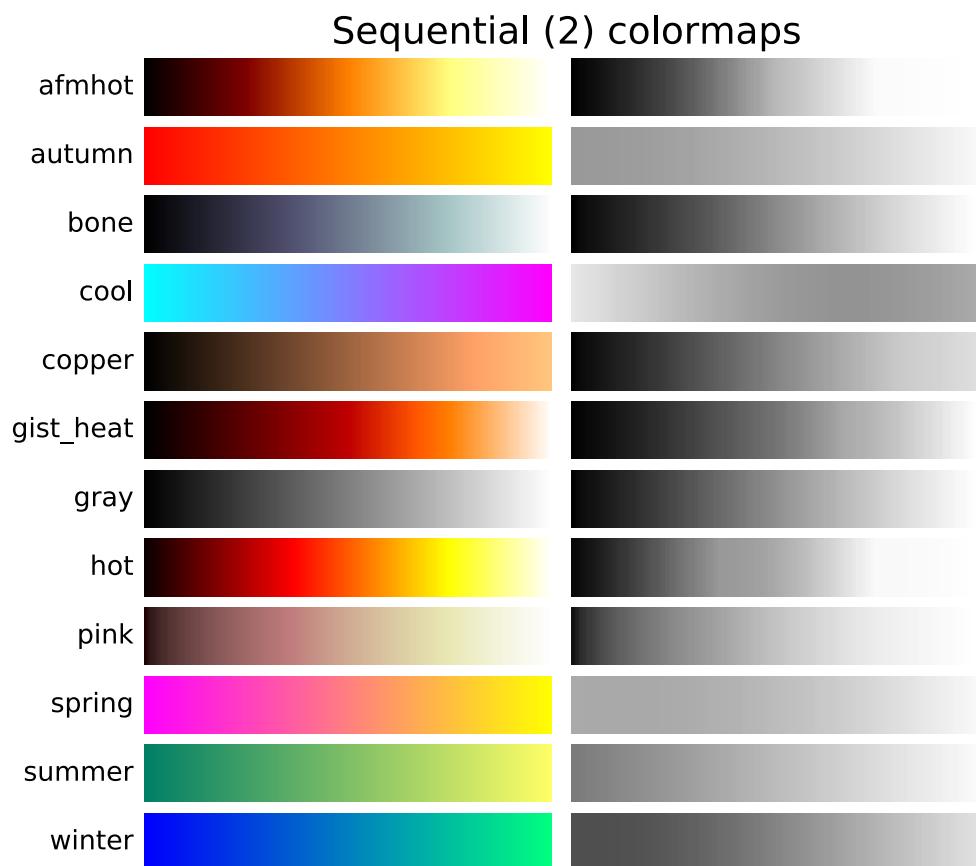
Conversion to grayscale is done in many different ways [\[bw\]](#). Some of the better ones use a linear combination of the `rgb` values of a pixel, but weighted according to how we perceive color intensity. A nonlinear method of conversion to grayscale is to use the L^* values of the pixels. In general, similar principles apply for this question as they do for presenting one's information perceptually; that is, if a colormap is chosen that is monotonically increasing in L^* values, it will print in a reasonable manner to grayscale.

With this in mind, we see that the Sequential colormaps have reasonable representations in grayscale. Some of the Sequential2 colormaps have decent enough grayscale representations, though some (autumn, spring, summer, winter) have very little grayscale change. If a colormap like this was used in a plot and then the plot was printed to grayscale, a lot of the information may map to the same gray values. The Diverging colormaps mostly vary from darker gray on the outer edges to white in the middle. Some (PuOr and seismic) have noticeably darker gray on one side than the other and therefore are not very symmetric. coolwarm has little range of gray scale and would print to a more uniform plot, losing a lot of detail. Note that overlaid, labeled contours could help differentiate between one side of the colormap vs. the other since color cannot be used once a plot is printed to grayscale. Many of the Qualitative and Miscellaneous colormaps, such as Accent, hsv, and jet, change from darker to lighter and back to darker gray throughout the colormap. This would make it impossible for a viewer to interpret the information in a plot once it is printed in grayscale.

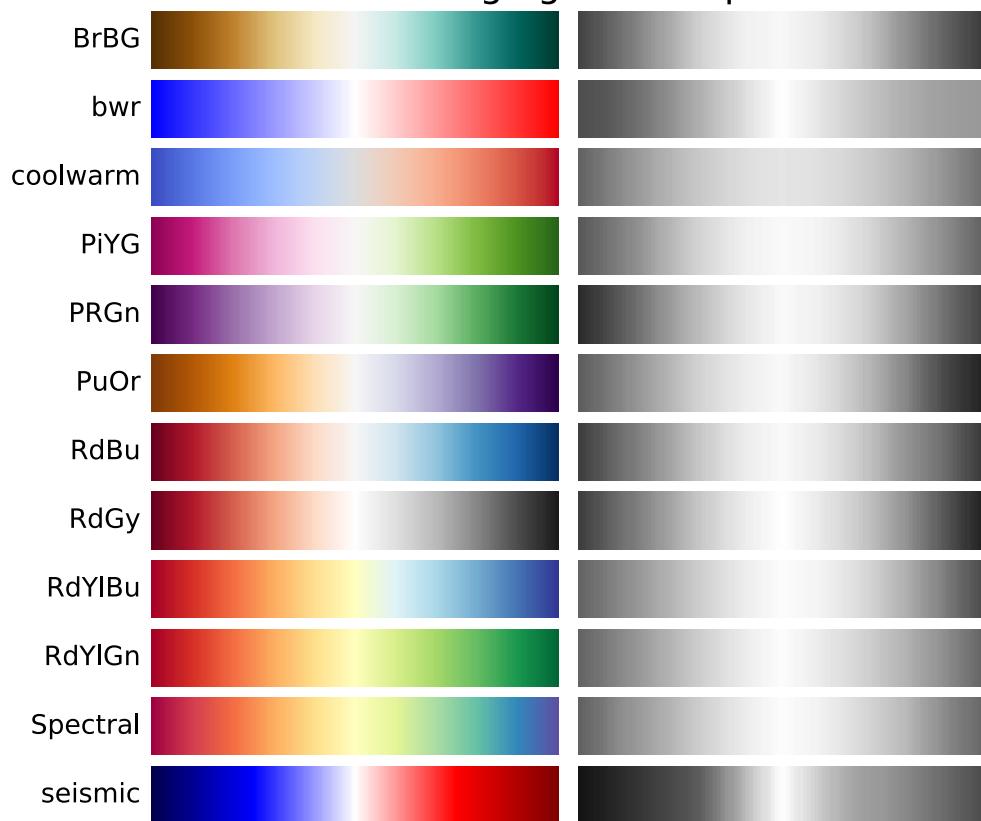


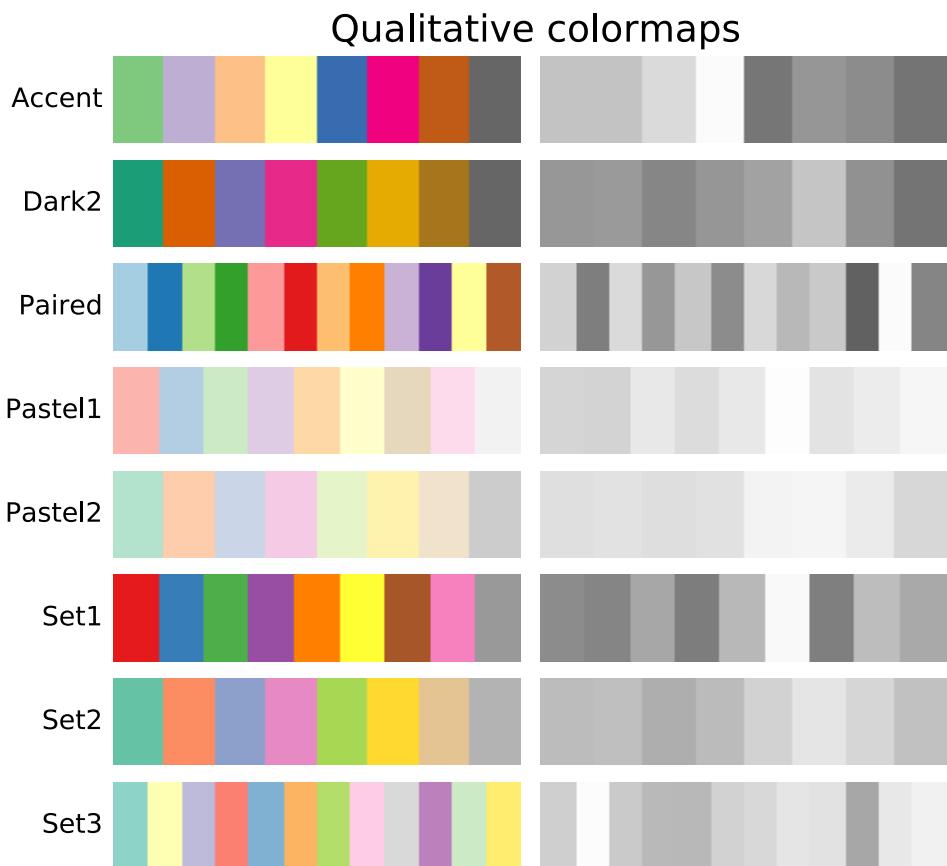
Sequential colormaps



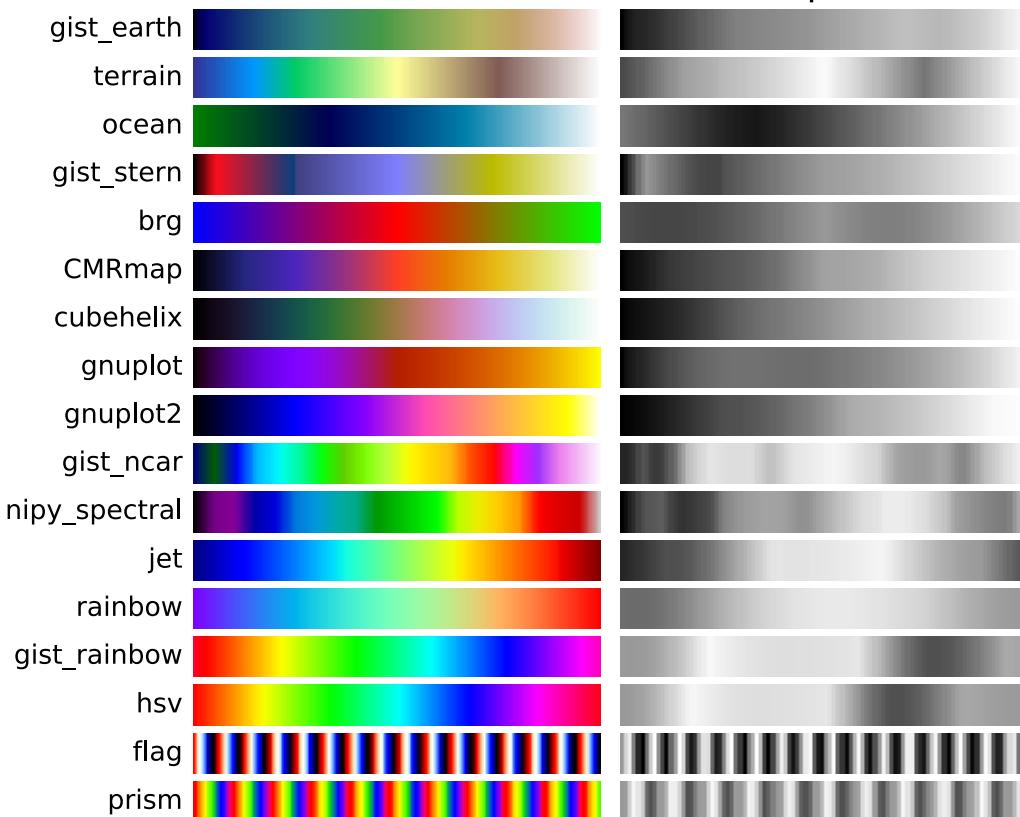


Diverging colormaps





Miscellaneous colormaps



5.2.5 Color vision deficiencies

There is a lot of information available about color blindness (*e.g.*, [\[colorblindness\]](#)). Additionally, there are tools available to convert images to how they look for different types of color vision deficiencies (*e.g.*, [\[vischeck\]](#)).

The most common form of color vision deficiency involves differentiating between red and green. Thus, avoiding colormaps with both red and green will avoid many problems in general.

5.2.6 References

5.3 Colormap Normalization

Objects that use colormaps by default linearly map the colors in the colormap from data values v_{min} to v_{max} . For example:

```
pcm = ax.pcolormesh(x, y, Z, vmin=-1., vmax=1., cmap='RdBu_r')
```

will map the data in Z linearly from -1 to +1, so $Z=0$ will give a color at the center of the colormap *RdBu_r* (white in this case).

Matplotlib does this mapping in two steps, with a normalization from [0,1] occurring first, and then mapping onto the indices in the colormap. Normalizations are classes defined in the `matplotlib.colors()` module. The default, linear normalization is `matplotlib.colors.Normalize()`.

Artists that map data to color pass the arguments `vmin` and `vmax` to construct a `matplotlib.colors.Normalize()` instance, then call it:

```
In [1]: import matplotlib as mpl
In [2]: norm = mpl.colors.Normalize(vmin=-1., vmax=1.)
In [3]: norm(0.)
Out[3]: 0.5
```

However, there are sometimes cases where it is useful to map data to colormaps in a non-linear fashion.

5.3.1 Logarithmic

One of the most common transformations is to plot data by taking its logarithm (to the base-10). This transformation is useful to display changes across disparate scales. Using `colors.LogNorm()` normalizes the data via \log_{10} . In the example below, there are two bumps, one much smaller than the other. Using `colors.LogNorm()`, the shape and location of each bump can clearly be seen:

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

"""
Lognorm: Instead of pcolor log10(Z1) you can have colorbars that have
the exponential labels using a norm.
"""

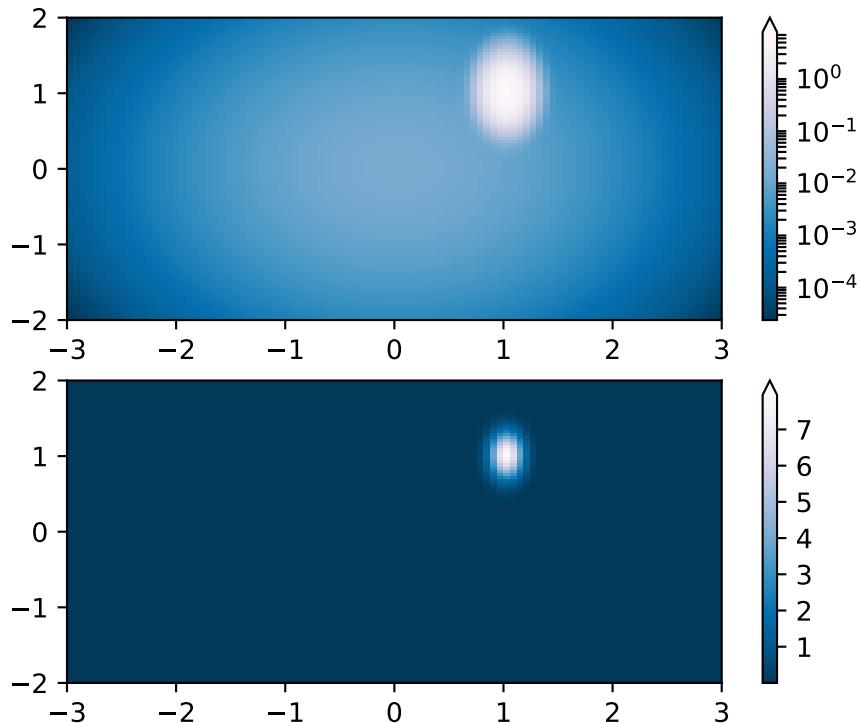
N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right. Needs to have
# z/colour axis on a log scale so we see both hump and spike. linear
# scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + \
    0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolor(X, Y, Z1,
                    norm=colors.LogNorm(vmin=Z1.min(), vmax=Z1.max()),
                    cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[0], extend='max')
```

```
pcm = ax[1].pcolor(X, Y, Z1, cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[1], extend='max')
fig.show()
```



5.3.2 Symmetric logarithmic

Similarly, it sometimes happens that there is data that is positive and negative, but we would still like a logarithmic scaling applied to both. In this case, the negative numbers are also scaled logarithmically, and mapped to smaller numbers; e.g., if `vmin=-vmax`, then the negative numbers are mapped from 0 to 0.5 and the positive from 0.5 to 1.

Since the logarithm of values close to zero tends toward infinity, a small range around zero needs to be mapped linearly. The parameter `linthresh` allows the user to specify the size of this range (`-linthresh`, `linthresh`). The size of this range in the colormap is set by `linscale`. When `linscale == 1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

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

```
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

"""
SymLogNorm: two humps, one negative and one positive, The positive
with 5-times the amplitude. Linearly, you cannot see detail in the
negative hump. Here we logarithmically scale the positive and
negative data separately.

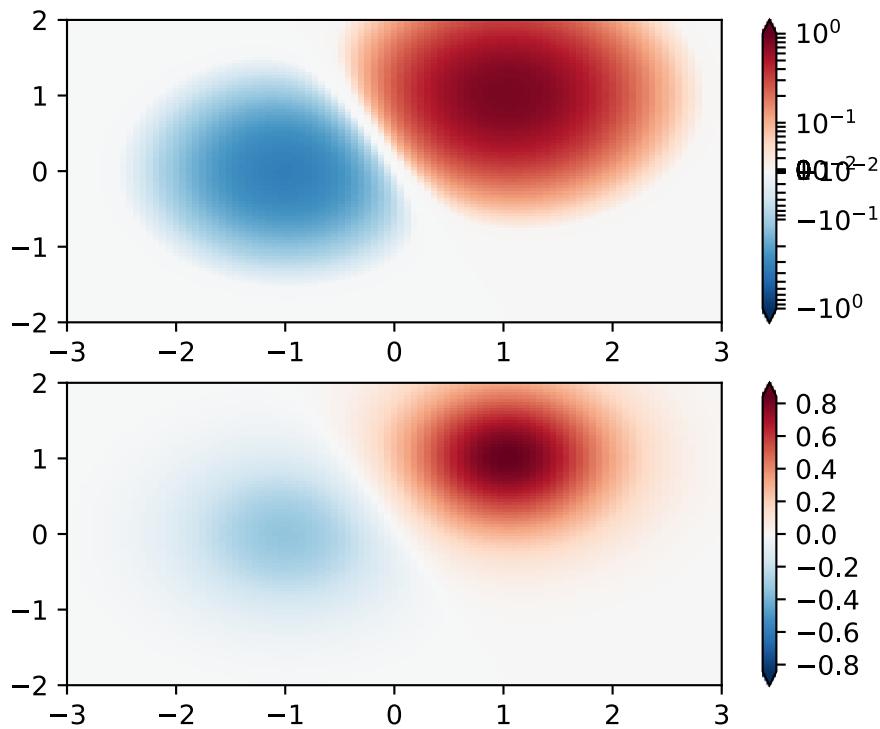
Note that colorbar labels do not come out looking very good.
"""

N=100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2 \
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1,
                       norm=colors.SymLogNorm(linthresh=0.03, linscale=0.03,
                                              vmin=-1.0, vmax=1.0),
                       cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[1], extend='both')
fig.show()
```



5.3.3 Power-law

Sometimes it is useful to remap the colors onto a power-law relationship (i.e. $y = x^\gamma$, where γ is the power). For this we use the `colors.PowerNorm()`. It takes as an argument `gamma` (`gamma == 1.0` will just yield the default linear normalization):

Note: There should probably be a good reason for plotting the data using this type of transformation. Technical viewers are used to linear and logarithmic axes and data transformations. Power laws are less common, and viewers should explicitly be made aware that they have been used.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
```

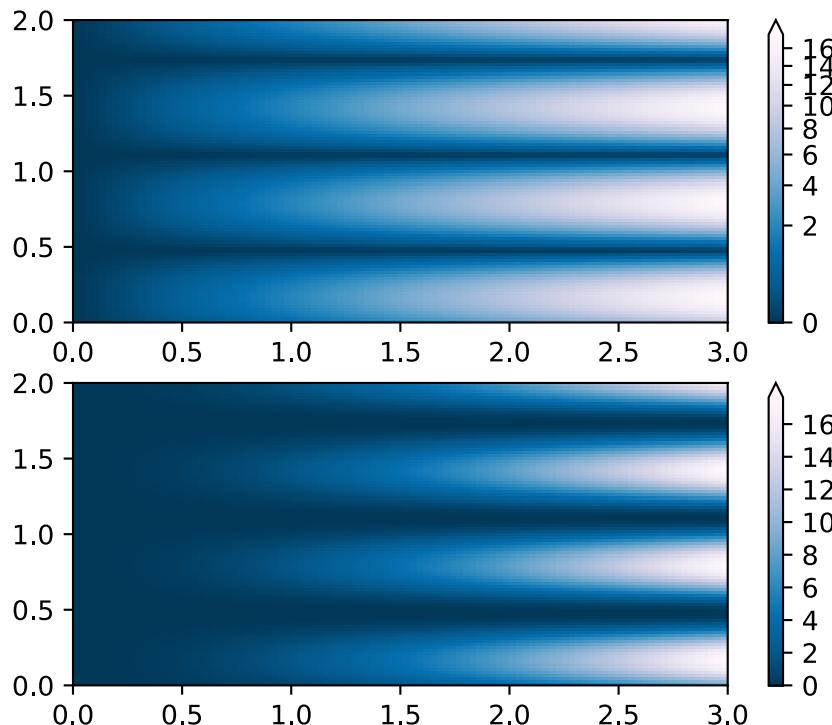
```
""
PowerNorm: Here a power-law trend in X partially obscures a rectified
sine wave in Y. We can remove the power law using a PowerNorm.
"""

X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z1 = (1 + np.sin(Y * 10.)) * X**2.

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1, norm=colors.PowerNorm(gamma=1./2.),
                       cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[0], extend='max')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[1], extend='max')
fig.show()
```



5.3.4 Discrete bounds

Another normalization that comes with matplotlib is `colors.BoundaryNorm()`. In addition to `vmin` and `vmax`, this takes as arguments boundaries between which data is to be mapped. The colors are then linearly distributed between these “bounds”. For instance:

```
In [4]: import matplotlib.colors as colors

In [5]: bounds = np.array([-0.25, -0.125, 0, 0.5, 1])

In [6]: norm = colors.BoundaryNorm(boundaries=bounds, ncolors=4)

In [7]: print(norm([-0.2,-0.15,-0.02, 0.3, 0.8, 0.99]))
[0 0 1 2 3 3]
```

Note unlike the other norms, this norm returns values from 0 to *ncolors*-1.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

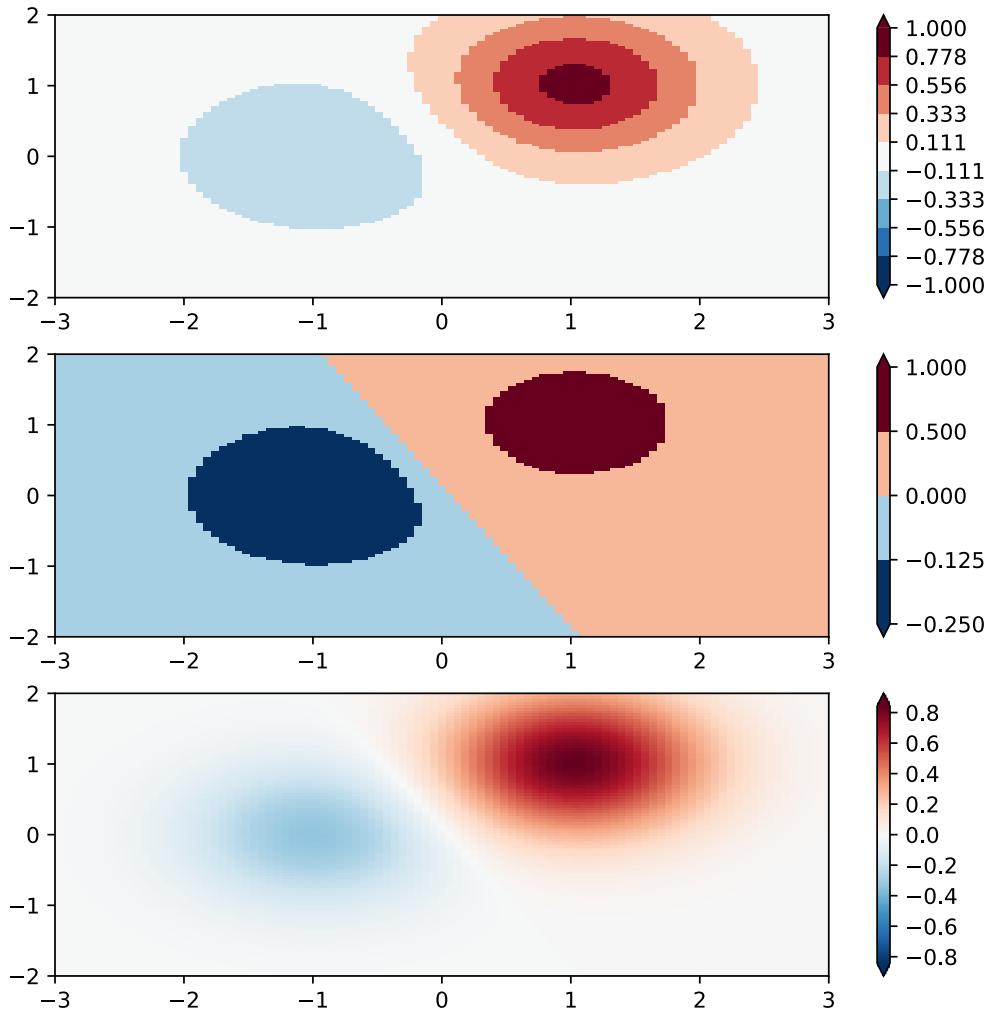
N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2 \
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

"""
BoundaryNorm: For this one you provide the boundaries for your colors,
and the Norm puts the first color in between the first pair, the
second color between the second pair, etc.
"""

fig, ax = plt.subplots(3, 1, figsize=(8, 8))
ax = ax.flatten()
# even bounds gives a contour-like effect
bounds = np.linspace(-1, 1, 10)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[0].pcolormesh(X, Y, Z1,
                       norm=norm,
                       cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both', orientation='vertical')

# uneven bounds changes the colormapping:
bounds = np.array([-0.25, -0.125, 0, 0.5, 1])
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[1].pcolormesh(X, Y, Z1, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[1], extend='both', orientation='vertical')

pcm = ax[2].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[2], extend='both', orientation='vertical')
fig.show()
```



5.3.5 Custom normalization: Two linear ranges

It is possible to define your own normalization. In the following example, we modify `colors.SymLogNorm()` to use different linear maps for the negative data values and the positive. (Note that this example is simple, and does not validate inputs or account for complex cases such as masked data)

Note: This may appear soon as `colors.OffsetNorm()`.

As above, non-symmetric mapping of data to color is non-standard practice for quantitative data, and should only be used advisedly. A practical example is having an ocean/land colormap where the land and ocean

data span different ranges.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
"""

Custom Norm: An example with a customized normalization. This one
uses the example above, and normalizes the negative data differently
from the positive.
"""

X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2 \
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

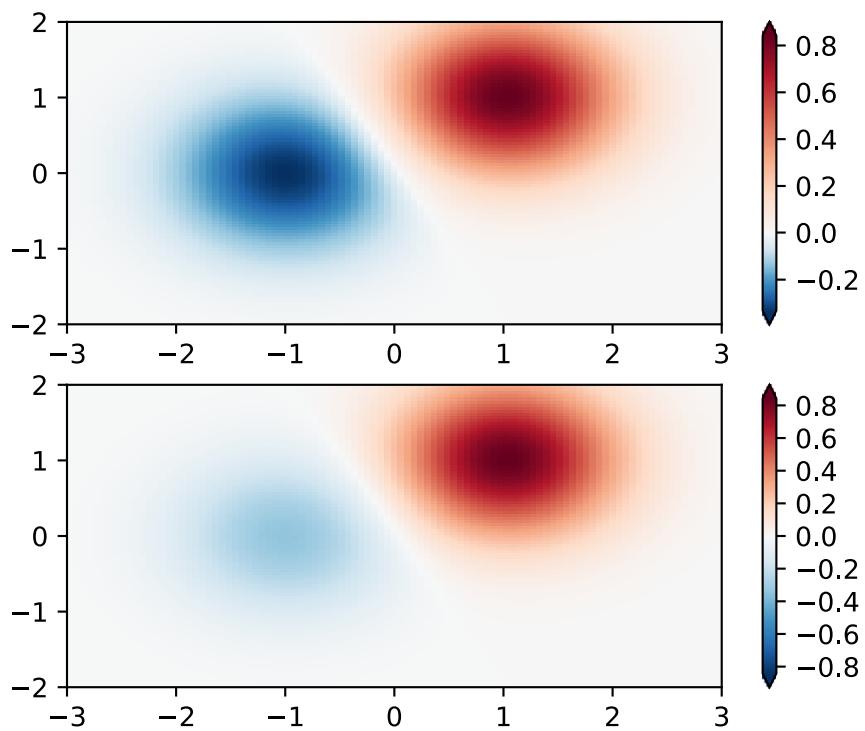
# Example of making your own norm. Also see matplotlib.colors.
# From Joe Kington: This one gives two different linear ramps:

class MidpointNormalize(colors.Normalize):
    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        colors.Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))
#####
fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1,
                       norm=MidpointNormalize(midpoint=0.),
                       cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[1], extend='both')
fig.show()
```



CUSTOMIZING MATPLOTLIB

6.1 Using style sheets

The `style` package adds support for easy-to-switch plotting “styles” with the same parameters as a `matplotlibrc` file (which is read at startup to configure matplotlib).

There are a number of pre-defined styles provided by matplotlib. For example, there’s a pre-defined style called “`ggplot`”, which emulates the aesthetics of `ggplot` (a popular plotting package for R). To use this style, just add:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
```

To list all available styles, use:

```
>>> print(plt.style.available)
```

6.1.1 Defining your own style

You can create custom styles and use them by calling `style.use` with the path or URL to the style sheet. Additionally, if you add your `<style-name>.mplstyle` file to `mpl_configdir/stylelib`, you can reuse your custom style sheet with a call to `style.use(<style-name>)`. By default `mpl_configdir` should be `~/.config/matplotlib`, but you can check where yours is with `matplotlib.get_configdir()`; you may need to create this directory. You also can change the directory where matplotlib looks for the `stylelib/` folder by setting the `MPLCONFIGDIR` environment variable, see *matplotlib configuration and cache directory locations*.

Note that a custom style sheet in `mpl_configdir/stylelib` will override a style sheet defined by matplotlib if the styles have the same name.

For example, you might want to create `mpl_configdir/stylelib/presentation.mplstyle` with the following:

```
axes.titlesize : 24
axes.labelsize : 20
lines.linewidth : 3
lines.markersize : 10
```

```
xtick.labelsize : 16  
ytick.labelsize : 16
```

Then, when you want to adapt a plot designed for a paper to one that looks good in a presentation, you can just add:

```
>>> import matplotlib.pyplot as plt  
>>> plt.style.use('presentation')
```

6.1.2 Composing styles

Style sheets are designed to be composed together. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations. These styles can easily be combined by passing a list of styles:

```
>>> import matplotlib.pyplot as plt  
>>> plt.style.use(['dark_background', 'presentation'])
```

Note that styles further to the right will overwrite values that are already defined by styles on the left.

6.1.3 Temporary styling

If you only want to use a style for a specific block of code but don't want to change the global styling, the style package provides a context manager for limiting your changes to a specific scope. To isolate your styling changes, you can write something like the following:

```
>>> import numpy as np  
>>> import matplotlib.pyplot as plt  
>>>  
>>> with plt.style.context('dark_background'):  
>>>     plt.plot(np.sin(np.linspace(0, 2 * np.pi)), 'r-o')  
>>>  
>>> # Some plotting code with the default style  
>>>  
>>> plt.show()
```

6.2 matplotlib rcParams

6.2.1 Dynamic rc settings

You can also dynamically change the default rc settings in a python script or interactively from the python shell. All of the rc settings are stored in a dictionary-like variable called `matplotlib.rcParams`, which is global to the matplotlib package. `rcParams` can be modified directly, for example:

```
import matplotlib as mpl
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r'
```

Matplotlib also provides a couple of convenience functions for modifying rc settings. The `matplotlib.rc()` command can be used to modify multiple settings in a single group at once, using keyword arguments:

```
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')
```

The `matplotlib.rcdefaults()` command will restore the standard matplotlib default settings.

There is some degree of validation when setting the values of `rcParams`, see `matplotlib.rcsetup` for details.

6.2.2 The `matplotlibrc` file

matplotlib uses `matplotlibrc` configuration files to customize all kinds of properties, which we call `rc` settings or `rc` parameters. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on. matplotlib looks for `matplotlibrc` in four locations, in the following order:

1. `matplotlibrc` in the current working directory, usually used for specific customizations that you do not want to apply elsewhere.
2. `$MATPLOTLIBRC/matplotlibrc`.
3. It next looks in a user-specific place, depending on your platform:
 - On Linux, it looks in `.config/matplotlib/matplotlibrc` (or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc`) if you've customized your environment.
 - On other platforms, it looks in `.matplotlib/matplotlibrc`.

See [matplotlib configuration and cache directory locations](#).

4. `INSTALL/matplotlib/mpl-data/matplotlibrc`, where `INSTALL` is something like `/usr/lib/python3.5/site-packages` on Linux, and maybe `C:\Python35\Lib\site-packages` on Windows. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be saved, please move this file to your user-specific matplotlib directory.

To display where the currently active `matplotlibrc` file was loaded from, one can do the following:

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'/home/foo/.config/matplotlib/matplotlibrc'
```

See below for a sample *matplotlibrc file*.

A sample matplotlibrc file

```
### MATPLOTLIBRC FORMAT

# This is a sample matplotlib configuration file - you can find a copy
# of it on your system in
# site-packages/matplotlib/mpl-data/matplotlibrc. If you edit it
# there, please note that it will be overwritten in your next install.
# If you want to keep a permanent local copy that will not be
# overwritten, place it in the following location:
# unix/linux:
#     $HOME/.config/matplotlib/matplotlibrc or
#     $XDG_CONFIG_HOME/matplotlib/matplotlibrc (if $XDG_CONFIG_HOME is set)
# other platforms:
#     $HOME/.matplotlib/matplotlibrc
#
# See http://matplotlib.org/users/customizing.html#the-matplotlibrc-file for
# more details on the paths which are checked for the configuration file.
#
# This file is best viewed in a editor which supports python mode
# syntax highlighting. Blank lines, or lines starting with a comment
# symbol, are ignored, as are trailing comments. Other lines must
# have the format
#     key : val # optional comment
#
# Colors: for the color values below, you can either use - a
# matplotlib color string, such as r, k, or b - an rgb tuple, such as
# (1.0, 0.5, 0.0) - a hex string, such as ff00ff - a scalar
# grayscale intensity such as 0.75 - a legal html color name, e.g., red,
# blue, darkslategray

#### CONFIGURATION BEGINS HERE

# The default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# MacOSX Qt4Agg Qt5Agg TkAgg WX WXAgg Agg Cairo GDK PS PDF SVG
# Template.
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH) as
# 'module://my_backend'.
backend      : qt5agg

# If you are using the Qt4Agg backend, you can choose here
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
#backend.qt4 : PyQt4      # PyQt4 | PySide

# Note that this can be overridden by the environment variable
# QT_API used by Enthought Tool Suite (ETS); valid values are
# "pyqt" and "pyside". The "pyqt" setting has the side effect of
# forcing the use of Version 2 API for QString and QVariant.

# The port to use for the web server in the WebAgg backend.
# webagg.port : 8888
```

```

# If webagg.port is unavailable, a number of other random ports will
# be tried until one that is available is found.
# webagg.port_retries : 50

# When True, open the webbrowser to the plot that is shown
# webagg.open_in_browser : True

# When True, the figures rendered in the nbagg backend are created with
# a transparent background.
# nbagg.transparent : False

# if you are running pyplot inside a GUI and your backend choice
# conflicts, we will automatically try to find a compatible one for
# you if backend_fallback is True
#backendFallback: True

#interactive : False
#toolbar      : toolbar2 # None | toolbar2 ("classic" is deprecated)
#timezone     : UTC       # a pytz timezone string, e.g., US/Central or Europe/Paris

# Where your matplotlib data lives if you installed to a non-default
# location. This is where the matplotlib fonts, bitmaps, etc reside
#datapath : /home/jdhunter/mpldata

### LINES
# See http://matplotlib.org/api/artist_api.html#module-matplotlib.lines for more
# information on line properties.
#lines.linewidth   : 1.5      # line width in points
#lines.linestyle   : -        # solid line
#lines.color       : C0       # has no affect on plot(); see axes.prop_cycle
#lines.marker      : None     # the default marker
#lines.markeredgewidth : 1.0    # the line width around the marker symbol
#lines.markersize  : 6        # markersize, in points
#lines.dash_joinstyle : miter      # miter/round/bevel
#lines.dash_capstyle : butt       # butt/round/projecting
#lines.solid_joinstyle : miter      # miter/round/bevel
#lines.solid_capstyle : projecting # butt/round/projecting
#lines.antialiased : True        # render lines in antialiased (no jaggies)

# The three standard dash patterns. These are scaled by the linewidth.
#lines.dashed_pattern : 2.8, 1.2
#lines.dashdot_pattern : 4.8, 1.2, 0.8, 1.2
#lines.dotted_pattern : 1.1, 1.1
#lines.scale_dashes : True

#markers.fillstyle: full # full/left/right/bottom/top/none

### PATCHES
# Patches are graphical objects that fill 2D space, like polygons or
# circles. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.patches

```

```
# information on patch properties
#patch.linewidth      : 1          # edge width in points.
#patch.facecolor       : C0
#patch.edgecolor        : black     # if forced, or patch is not filled
#patch.force_edgecolor : False     # True to always use edgecolor
#patch.antialiased     : True      # render patches in antialiased (no jaggies)

### HATCHES
#hatch.color      : k
#hatch.linewidth : 1.0

### Boxplot
#boxplot.notch      : False
#boxplot.vertical    : True
#boxplot.whiskers   : 1.5
#boxplot.bootstrap   : None
#boxplot.patchartist : False
#boxplot.showmeans   : False
#boxplot.showcaps    : True
#boxplot.showbox     : True
#boxplot.showfliers  : True
#boxplot.meanline    : False

#boxplot.flierprops.color      : 'k'
#boxplot.flierprops.marker     : 'o'
#boxplot.flierprops.markerfacecolor : 'none'
#boxplot.flierprops.markeredgecolor : 'k'
#boxplot.flierprops.markersize  : 6
#boxplot.flierprops.linestyle  : 'none'
#boxplot.flierprops.linewidth  : 1.0

#boxplot.boxprops.color      : 'k'
#boxplot.boxprops.linewidth : 1.0
#boxplot.boxprops.linestyle : '-'

#boxplot.whiskerprops.color      : 'k'
#boxplot.whiskerprops.linewidth : 1.0
#boxplot.whiskerprops.linestyle : '-'

#boxplot.capprops.color      : 'k'
#boxplot.capprops.linewidth : 1.0
#boxplot.capprops.linestyle : '-'

#boxplot.medianprops.color    : 'C1'
#boxplot.medianprops.linewidth : 1.0
#boxplot.medianprops.linestyle : '-'

#boxplot.meanprops.color      : 'C2'
#boxplot.meanprops.marker     : '^'
#boxplot.meanprops.markerfacecolor : 'C2'
#boxplot.meanprops.markeredgecolor : 'C2'
#boxplot.meanprops.markersize  : 6
#boxplot.meanprops.linestyle  : 'none'
```

```
#boxplot.meanprops.linewidth      : 1.0

### FONT
#
# font properties used by text.Text. See
# http://matplotlib.org/api/font_manager_api.html for more
# information on font properties. The 6 font properties used for font
# matching are given below with their default values.
#
# The font.family property has five values: 'serif' (e.g., Times),
# 'sans-serif' (e.g., Helvetica), 'cursive' (e.g., Zapf-Chancery),
# 'fantasy' (e.g., Western), and 'monospace' (e.g., Courier). Each of
# these font families has a default list of font names in decreasing
# order of priority associated with them. When text.usetex is False,
# font.family may also be one or more concrete font names.
#
# The font.style property has three values: normal (or roman), italic
# or oblique. The oblique style will be used for italic, if it is not
# present.
#
# The font.variant property has two values: normal or small-caps. For
# TrueType fonts, which are scalable fonts, small-caps is equivalent
# to using a font size of 'smaller', or about 83% of the current font
# size.
#
# The font.weight property has effectively 13 values: normal, bold,
# bolder, lighter, 100, 200, 300, ..., 900. Normal is the same as
# 400, and bold is 700. bolder and lighter are relative values with
# respect to the current weight.
#
# The font.stretch property has 11 values: ultra-condensed,
# extra-condensed, condensed, semi-condensed, normal, semi-expanded,
# expanded, extra-expanded, ultra-expanded, wider, and narrower. This
# property is not currently implemented.
#
# The font.size property is the default font size for text, given in pts.
# 10 pt is the standard value.
#
#font.family      : sans-serif
#font.style       : normal
#font.variant    : normal
#font.weight     : medium
#font.stretch    : normal
# note that font.size controls default text sizes. To configure
# special text sizes tick labels, axes, labels, title, etc, see the rc
# settings for axes and ticks. Special text sizes can be defined
# relative to font.size, using the following values: xx-small, x-small,
# small, medium, large, x-large, xx-large, larger, or smaller
#font.size        : 10.0
#font.serif       : DejaVu Serif, Bitstream Vera Serif, New Century Schoolbook,_
#                   ↵Century Schoolbook L, Utopia, ITC Bookman, Bookman, Nimbus Roman No9 L, Times New_
#                   ↵Roman, Times, Palatino, Charter, serif
#font.sans-serif   : DejaVu Sans, Bitstream Vera Sans, Lucida Grande, Verdana, Geneva,_
#                   ↵Lucid, Arial, Helvetica, Avant Garde, sans-serif
```

```
#font.cursive      : Apple Chancery, Textile, Zapf Chancery, Sand, Script MT, Felipa, ↵
˓→cursive
#font.fantasy     : Comic Sans MS, Chicago, Charcoal, Impact, Western, Humor Sans, ↵
˓→xkcd, fantasy
#font.monospace   : DejaVu Sans Mono, Bitstream Vera Sans Mono, Andale Mono, Nimbus ↵
˓→Mono L, Courier New, Courier, Fixed, Terminal, monospace

### TEXT
# text properties used by text.Text. See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.text for more
# information on text properties

{text.color}       : black

### LaTeX customizations. See http://wiki.scipy.org/Cookbook/Matplotlib/UsingTex
{text.usetex}      : False # use latex for all text handling. The following fonts
# are supported through the usual rc parameter settings:
# new century schoolbook, bookman, times, palatino,
# zapf chancery, charter, serif, sans-serif, helvetica,
# avant garde, courier, monospace, computer modern roman,
# computer modern sans serif, computer modern typewriter
# If another font is desired which can loaded using the
# LaTeX \usepackage command, please inquire at the
# matplotlib mailing list
{text.latex.unicode} : False # use "ucs" and "inputenc" LaTeX packages for handling
# unicode strings.
{text.latex.preamble} : # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
# AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
# IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
# preamble is a comma separated list of LaTeX statements
# that are included in the LaTeX document preamble.
# An example:
# text.latex.preamble : \usepackage{bm},\usepackage{euler}
# The following packages are always loaded with usetex, so
# beware of package collisions: color, geometry, graphicx,
# type1cm, textcomp. Adobe Postscript (PSSNFS) font packages
# may also be loaded, depending on your font settings

{text.dvipnghack} : None      # some versions of dvipng don't handle alpha
# channel properly. Use True to correct
# and flush ~/.matplotlib/tex.cache
# before testing and False to force
# correction off. None will try and
# guess based on your dvipng version

{text.hinting} : auto # May be one of the following:
# 'none': Perform no hinting
# 'auto': Use FreeType's autohinter
# 'native': Use the hinting information in the
#           font file, if available, and if your
#           FreeType library supports it
# 'either': Use the native hinting information,
#           or the autohinter if none is available.
```

```

        # For backward compatibility, this value may also be
        # True === 'auto' or False === 'none'.
#text.hinting_factor : 8 # Specifies the amount of softness for hinting in the
                        # horizontal direction. A value of 1 will hint to full
                        # pixels. A value of 2 will hint to half pixels etc.

#text.antialiased : True # If True (default), the text will be antialiased.
                        # This only affects the Agg backend.

# The following settings allow you to select the fonts in math mode.
# They map from a TeX font name to a fontconfig font pattern.
# These settings are only used if mathtext.fontset is 'custom'.
# Note that this "custom" mode is unsupported and may go away in the
# future.
#mathtext.cal : cursive
#mathtext.rm : serif
#mathtext.tt : monospace
#mathtext.it : serif:italic
#mathtext.bf : serif:bold
#mathtext.sf : sans
#mathtext.fontset : dejavusans # Should be 'dejavusans' (default),
                            # 'dejavuserif', 'cm' (Computer Modern), 'stix',
                            # 'stixsans' or 'custom'
#mathtext.fallback_to_cm : True # When True, use symbols from the Computer Modern
                                # fonts when a symbol can not be found in one of
                                # the custom math fonts.

#mathtext.default : it # The default font to use for math.
                        # Can be any of the LaTeX font names, including
                        # the special name "regular" for the same font
                        # used in regular text.

### AXES
# default face and edge color, default tick sizes,
# default fontsizes for ticklabels, and so on. See
# http://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.facecolor      : white    # axes background color
#axes.edgecolor      : black     # axes edge color
#axes.linewidth     : 0.8      # edge linewidth
#axes.grid          : False     # display grid or not
#axes.titlesize     : large     # fontsize of the axes title
#axes.titlepad       : 6.0       # pad between axes and title in points
#axes.labelsize      : medium    # fontsize of the x any y labels
#axes.labelpad       : 4.0       # space between label and axis
#axes.labelweight    : normal    # weight of the x and y labels
#axes.labelcolor     : black
#axes.axisbelow      : 'line'   # draw axis gridlines and ticks below
                            # patches (True); above patches but below
                            # lines ('line'); or above all (False)

#axes.formatter.limits : -7, 7 # use scientific notation if log10
                            # of the axis range is smaller than the
                            # first or larger than the second

```

```
#axes.formatter.use_locale : False # When True, format tick labels
# according to the user's locale.
# For example, use ',' as a decimal
# separator in the fr_FR locale.
#axes.formatter.use_mathtext : False # When True, use mathtext for scientific
# notation.
#axes.formatter.useoffset      : True   # If True, the tick label formatter
# will default to labeling ticks relative
# to an offset when the data range is
# small compared to the minimum absolute
# value of the data.
#axes.formatter.offset_threshold : 4    # When useoffset is True, the offset
# will be used when it can remove
# at least this number of significant
# digits from tick labels.

# axes.spines.left   : True   # display axis spines
# axes.spines.bottom : True
# axes.spines.top    : True
# axes.spines.right  : True

#axes.unicode_minus  : True    # use unicode for the minus symbol
# rather than hyphen. See
# http://en.wikipedia.org/wiki/Plus_and_minus_signs
#Character_codes
#axes.prop_cycle     : cycler('color',
#                               ['1f77b4', 'ff7f0e', '2ca02c', 'd62728',
#                                '9467bd', '8c564b', 'e377c2', '7f7f7f',
#                                'bcbd22', '17becf'])
#                                         # color cycle for plot lines
#                                         # as list of string colorspecs:
#                                         # single letter, long name, or
#                                         # web-style hex
#axes.autolimit_mode : data # How to scale axes limits to the data.
#                         # Use "data" to use data limits, plus some margin
#                         # Use "round_number" move to the nearest "round" number
#axes.xmargin        : .05 # x margin. See `axes.Axes.margins`
#axes.ymargin        : .05 # y margin See `axes.Axes.margins`

#polaraxes.grid      : True    # display grid on polar axes
#axes3d.grid         : True    # display grid on 3d axes

### DATES
# These control the default format strings used in AutoDateFormatter.
# Any valid format datetime format string can be used (see the python
# `datetime` for details). For example using '%x' will use the locale date representation
# '%X' will use the locale time representation and '%c' will use the full locale datetime
# representation.
# These values map to the scales:
#     {'year': 365, 'month': 30, 'day': 1, 'hour': 1/24, 'minute': 1 / (24 * 60)}

# date.autoformatter.year      : %Y
```

```

# date.autoformatter.month      : %Y-%m
# date.autoformatter.day       : %Y-%m-%d
# date.autoformatter.hour      : %m-%d %H
# date.autoformatter.minute    : %d %H:%M
# date.autoformatter.second    : %H:%M:%S
# date.autoformatter.microsecond : %M:%S.%f

### TICKS
# see http://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.top          : False   # draw ticks on the top side
#xtick.bottom        : True    # draw ticks on the bottom side
#xtick.major.size   : 3.5     # major tick size in points
#xtick.minor.size   : 2        # minor tick size in points
#xtick.major.width  : 0.8     # major tick width in points
#xtick.minor.width  : 0.6     # minor tick width in points
#xtick.major.pad    : 3.5     # distance to major tick label in points
#xtick.minor.pad    : 3.4     # distance to the minor tick label in points
#xtick.color         : k       # color of the tick labels
#xtick.labelsize    : medium  # fontsize of the tick labels
#xtick.direction    : out     # direction: in, out, or inout
#xtick.minor.visible: False   # visibility of minor ticks on x-axis
#xtick.major.top    : True    # draw x axis top major ticks
#xtick.major.bottom : True    # draw x axis bottom major ticks
#xtick.minor.top    : True    # draw x axis top minor ticks
#xtick.minor.bottom : True    # draw x axis bottom minor ticks

#ytick.left          : True   # draw ticks on the left side
#ytick.right         : False  # draw ticks on the right side
#ytick.major.size   : 3.5     # major tick size in points
#ytick.minor.size   : 2        # minor tick size in points
#ytick.major.width  : 0.8     # major tick width in points
#ytick.minor.width  : 0.6     # minor tick width in points
#ytick.major.pad    : 3.5     # distance to major tick label in points
#ytick.minor.pad    : 3.4     # distance to the minor tick label in points
#ytick.color         : k       # color of the tick labels
#ytick.labelsize    : medium  # fontsize of the tick labels
#ytick.direction    : out     # direction: in, out, or inout
#ytick.minor.visible: False   # visibility of minor ticks on y-axis
#xtick.major.left   : True    # draw y axis left major ticks
#xtick.major.right  : True    # draw y axis right major ticks
#xtick.minor.left   : True    # draw y axis left minor ticks
#xtick.minor.right  : True    # draw y axis right minor ticks

### GRIDS
#grid.color         : b0b0b0  # grid color
#grid.linestyle     : -       # solid
#grid.linewidth    : 0.8     # in points
#grid.alpha         : 1.0     # transparency, between 0.0 and 1.0

### Legend
#legend.loc         : best
#legend.frameon    : True    # if True, draw the legend on a background patch

```

```
#legend.framealpha      : 0.8      # legend patch transparency
#legend.facecolor       : inherit   # inherit from axes.facecolor; or color spec
#legend.edgecolor        : 0.8      # background patch boundary color
#legend.fancybox        : True     # if True, use a rounded box for the
#                                 # legend background, else a rectangle
#legend.shadow          : False    # if True, give background a shadow effect
#legend.numpoints        : 1        # the number of marker points in the legend line
#legend.scatterpoints    : 1        # number of scatter points
#legend.markerscale      : 1.0     # the relative size of legend markers vs. original
#legend.fontsize         : medium
# Dimensions as fraction of fontsize:
#legend.borderpad        : 0.4      # border whitespace
#legend.labelspacing      : 0.5      # the vertical space between the legend entries
#legend.handlelength      : 2.0      # the length of the legend lines
#legend.handleheight      : 0.7      # the height of the legend handle
#legend.handletextpad     : 0.8      # the space between the legend line and legend text
#legend.borderaxespad    : 0.5      # the border between the axes and legend edge
#legend.columnspacing     : 2.0      # column separation

### FIGURE
# See http://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure
#figure.titlesize : large      # size of the figure title (Figure.suptitle())
#figure.titleweight : normal    # weight of the figure title
#figure.figsize   : 6.4, 4.8    # figure size in inches
#figure.dpi       : 100        # figure dots per inch
#figure.facecolor : white      # figure facecolor; 0.75 is scalar gray
#figure.edgecolor : white      # figure edgecolor
#figure.autolayout : False     # When True, automatically adjust subplot
#                               # parameters to make the plot fit the figure
#figure.max_open_warning : 20    # The maximum number of figures to open through
#                               # the pyplot interface before emitting a warning.
#                               # If less than one this feature is disabled.

# The figure subplot parameters. All dimensions are a fraction of the
#figure.subplot.left      : 0.125   # the left side of the subplots of the figure
#figure.subplot.right     : 0.9      # the right side of the subplots of the figure
#figure.subplot.bottom    : 0.11     # the bottom of the subplots of the figure
#figure.subplot.top       : 0.88     # the top of the subplots of the figure
#figure.subplot.wspace   : 0.2      # the amount of width reserved for blank space between
#                                 # subplots,
#                               # expressed as a fraction of the average axis width
#figure.subplot.hspace   : 0.2      # the amount of height reserved for white space between
#                                 # subplots,
#                               # expressed as a fraction of the average axis height

### IMAGES
#image.aspect : equal           # equal / auto / a number
#image.interpolation : nearest  # see help(imshow) for options
#image.cmap   : viridis         # A colormap name, gray etc...
#image.lut    : 256              # the size of the colormap lookup table
#image.origin : upper            # lower / upper
#image.resample : True
```

```

#image.composite_image : True      # When True, all the images on a set of axes are
# combined into a single composite image before
# saving a figure as a vector graphics file,
# such as a PDF.

### CONTOUR PLOTS
#contour.negative_linestyle : dashed # dashed / solid
#contour.corner_mask       : True   # True / False / legacy

### ERRORBAR PLOTS
#errorbar.capsize : 0             # length of end cap on error bars in pixels

### HISTOGRAM PLOTS
#hist.bins : 10                  # The default number of histogram bins.
#                                # If Numpy 1.11 or later is
#                                # installed, may also be `auto`

### SCATTER PLOTS
#scatter.marker : o              # The default marker type for scatter plots.

### Agg rendering
### Warning: experimental, 2008/10/10
#agg.path.chunksize : 0           # 0 to disable; values in the range
#                                # 10000 to 100000 can improve speed slightly
#                                # and prevent an Agg rendering failure
#                                # when plotting very large data sets,
#                                # especially if they are very gappy.
#                                # It may cause minor artifacts, though.
#                                # A value of 20000 is probably a good
#                                # starting point.

### SAVING FIGURES
#path.simplify : True            # When True, simplify paths by removing "invisible"
#                                # points to reduce file size and increase rendering
#                                # speed
#path.simplify_threshold : 0.1    # The threshold of similarity below which
#                                # vertices will be removed in the simplification
#                                # process
#path.snap : True                # When True, rectilinear axis-aligned paths will be snapped to
#                                # the nearest pixel when certain criteria are met. When False,
#                                # paths will never be snapped.
#path.sketch : None              # May be none, or a 3-tuple of the form (scale, length,
#                                # randomness).
#                                # *scale* is the amplitude of the wiggle
#                                # perpendicular to the line (in pixels). *length*
#                                # is the length of the wiggle along the line (in
#                                # pixels). *randomness* is the factor by which
#                                # the length is randomly scaled.

# the default savefig params can be different from the display params
# e.g., you may want a higher resolution, or to make the figure
# background white
#savefig.dpi       : figure    # figure dots per inch or 'figure'
#savefig.facecolor : white     # figure facecolor when saving

```

```
#savefig.edgecolor : white    # figure edgecolor when saving
#savefig.format   : png      # png, ps, pdf, svg
#savefig.bbox     : standard # 'tight' or 'standard'.
                           # 'tight' is incompatible with pipe-based animation
                           # backends but will workd with temporary file based ones:
                           # e.g. setting animation.writer to ffmpeg will not work,
                           # use ffmpeg_file instead
#savefig.pad_inches : 0.1    # Padding to be used when bbox is set to 'tight'
#savefig.jpeg_quality: 95   # when a jpeg is saved, the default quality parameter.
#savefig.directory  : ~      # default directory in savefig dialog box,
                           # leave empty to always use current working directory
#savefig.transparent : False # setting that controls whether figures are saved with a
                           # transparent background by default

# tk backend params
#tk.window_focus : False    # Maintain shell focus for TkAgg

# ps backend params
#ps.papersize     : letter   # auto, letter, legal, ledger, A0-A10, B0-B10
#ps.useafm        : False    # use of afm fonts, results in small files
#ps.usedistiller  : False    # can be: None, ghostscript or xpdf
                           # Experimental: may produce smaller files.
                           # xpdf intended for production of publication
                           # but requires ghostscript, xpdf and ps2eps
#ps.distiller.res : 6000    # dpi
#ps.fonttype       : 3       # Output Type 3 (Type3) or Type 42 (TrueType)

# pdf backend params
#pdf.compression  : 6 # integer from 0 to 9
                      # 0 disables compression (good for debugging)
#pdf.fonttype     : 3       # Output Type 3 (Type3) or Type 42 (TrueType)

# svg backend params
#svg.image_inline : True    # write raster image data directly into the svg file
#svg.fonttype     : 'path'   # How to handle SVG fonts:
#   'none': Assume fonts are installed on the machine where the SVG will be viewed.
#   'path': Embed characters as paths -- supported by most SVG renderers
#   'svgfont': Embed characters as SVG fonts -- supported only by Chrome,
#             Opera and Safari
#svg.hashsalt : None       # if not None, use this string as hash salt
                           # instead of uuid4

# docstring params
#docstring.hardcopy = False # set this when you want to generate hardcopy docstring

# Set the verbose flags. This controls how much information
# matplotlib gives you at runtime and where it goes. The verbosity
# levels are: silent, helpful, debug, debug-annoying. Any level is
# inclusive of all the levels below it. If your setting is "debug",
# you'll get all the debug and helpful messages. When submitting
# problems to the mailing-list, please set verbose to "helpful" or "debug"
# and paste the output into your report.
```

```

#
# The "fileo" gives the destination for any calls to verbose.report.
# These objects can a filename, or a filehandle like sys.stdout.
#
# You can override the rc default verbosity from the command line by
# giving the flags --verbose-LEVEL where LEVEL is one of the legal
# levels, e.g., --verbose-helpful.
#
# You can access the verbose instance in your code
#   from matplotlib import verbose.
#verbose.level : silent      # one of silent, helpful, debug, debug-annoying
#verbose.fileo : sys.stdout # a log filename, sys.stdout or sys.stderr

# Event keys to interact with figures/plots via keyboard.
# Customize these settings according to your needs.
# Leave the field(s) empty if you don't need a key-map. (i.e., fullscreen : "")

#keymap.fullscreen : f, ctrl+f      # toggling
#keymap.home : h, r, home          # home or reset mnemonic
#keymap.back : left, c, backspace # forward / backward keys to enable
#keymap.forward : right, v        # left handed quick navigation
#keymap.pan : p                  # pan mnemonic
#keymap.zoom : o                # zoom mnemonic
#keymap.save : s                # saving current figure
#keymap.quit : ctrl+w, cmd+w     # close the current figure
#keymap.grid : g                # switching on/off a grid in current axes
#keymap.yscale : l              # toggle scaling of y-axes ('log'/linear')
#keymap.xscale : L, k            # toggle scaling of x-axes ('log'/linear')
#keymap.all_axes : a             # enable all axes

# Control location of examples data files
#examples.directory : " # directory to look in for custom installation

###ANIMATION settings
#animation.html : 'none'          # How to display the animation as HTML in
#                                 # the IPython notebook. 'html5' uses
#                                 # HTML5 video tag.
#animation.writer : ffmpeg         # MovieWriter 'backend' to use
#animation.codec : h264            # Codec to use for writing movie
#animation.bitrate: -1            # Controls size/quality tradeoff for movie.
#                                 # -1 implies let utility auto-determine
#animation.frame_format: 'png'    # Controls frame format used by temp files
#animation.ffmpeg_path: 'ffmpeg'  # Path to ffmpeg binary. Without full path
#                                 # $PATH is searched
#animation.ffmpeg_args: ""        # Additional arguments to pass to ffmpeg
#animation.avconv_path: 'avconv'  # Path to avconv binary. Without full path
#                                 # $PATH is searched
#animation.avconv_args: ""        # Additional arguments to pass to avconv
#animation.mencoder_path: 'mencoder'
#                                 # Path to mencoder binary. Without full path
#                                 # $PATH is searched
#animation.mencoder_args: ""      # Additional arguments to pass to mencoder
#animation.convert_path: 'convert' # Path to ImageMagick's convert binary.

```

```
# On Windows use the full path since convert  
# is also the name of a system tool.
```

INTERACTIVE PLOTS

7.1 Interactive navigation



All figure windows come with a navigation toolbar, which can be used to navigate through the data set. Here is a description of each of the buttons at the bottom of the toolbar



The Home, Forward and Back buttons These are akin to a web browser's home, forward and back controls. Forward and Back are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click Back on your web browser before visiting a new page or Forward before you have gone back to a page – nothing happens. Home always takes you to the first, default view of your data. Again, all of these buttons should feel very familiar to any user of a web browser.



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

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



The Zoom-to-rectangle button Click this toolbar button to activate this mode. Put your mouse somewhere over an axes and press the left mouse button. Drag the mouse while holding the button to a new location and release. The axes view limits will be zoomed to the rectangle you have defined. There is also an experimental ‘zoom out to rectangle’ in this mode with the right button, which will place your entire axes in the region defined by the zoom out rectangle.



The Subplot-configuration button Use this tool to configure the appearance of the subplot: you can stretch or compress the left, right, top, or bottom side of the subplot, or the space between the rows or space between the columns.



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

7.1.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your matplotlibrc (#keymap.*).

Command	Keyboard Shortcut(s)
Home/Reset	h or r or home
Back	c or left arrow or backspace
Forward	v or right arrow
Pan/Zoom	p
Zoom-to-rect	o
Save	ctrl + s
Toggle fullscreen	f or ctrl + f
Close plot	ctrl + w
Constrain pan/zoom to x axis	hold x when panning/zooming with mouse
Constrain pan/zoom to y axis	hold y when panning/zooming with mouse
Preserve aspect ratio	hold CONTROL when panning/zooming with mouse
Toggle grid	g when mouse is over an axes
Toggle x axis scale (log/linear)	L or k when mouse is over an axes
Toggle y axis scale (log/linear)	I when mouse is over an axes

If you are using `matplotlib.pyplot` the toolbar will be created automatically for every figure. If you are writing your own user interface code, you can add the toolbar as a widget. The exact syntax depends on your UI, but we have examples for every supported UI in the `matplotlib/examples/user_interfaces` directory. Here is some example code for GTK:

```
import gtk

from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as_
NavigationToolbar

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(111)
ax.plot([1,2,3])

canvas = FigureCanvas(fig) # a gtk.DrawingArea
vbox.pack_start(canvas)
```

```
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)

win.show_all()
gtk.main()
```

7.2 Using matplotlib in a python shell

Warning: This page is significantly out of date

By default, matplotlib defers drawing until the end of the script because drawing can be an expensive operation, and you may not want to update the plot every time a single property is changed, only once after all the properties have changed.

But when working from the python shell, you usually do want to update the plot with every command, e.g., after changing the `xlabel()`, or the marker style of a line. While this is simple in concept, in practice it can be tricky, because matplotlib is a graphical user interface application under the hood, and there are some tricks to make the applications work right in a python shell.

7.2.1 IPython to the rescue

Note: The mode described here still exists for historical reasons, but it is highly advised not to use. It pollutes namespaces with functions that will shadow python built-in and can lead to hard to track bugs. To get IPython integration without imports the use of the `%matplotlib` magic is preferred. See [ipython documentation](#).

Fortunately, `ipython`, an enhanced interactive python shell, has figured out all of these tricks, and is matplotlib aware, so when you start `ipython` in the `pylab` mode.

```
johnh@flag:~> ipython
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.
```

In [1]: `%pylab`

```
Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.
```

In [2]: `x = randn(10000)`

In [3]: `hist(x, 100)`

it sets everything up for you so interactive plotting works as you would expect it to. Call `figure()` and a figure window pops up, call `plot()` and your data appears in the figure window.

Note in the example above that we did not import any matplotlib names because in pylab mode, ipython will import them automatically. ipython also turns on *interactive* mode for you, which causes every pyplot command to trigger a figure update, and also provides a matplotlib aware run command to run matplotlib scripts efficiently. ipython will turn off interactive mode during a run command, and then restore the interactive state at the end of the run so you can continue tweaking the figure manually.

There has been a lot of recent work to embed ipython, with pylab support, into various GUI applications, so check on the ipython mailing [list](#) for the latest status.

7.2.2 Other python interpreters

If you can't use ipython, and still want to use matplotlib/pylab from an interactive python shell, e.g., the plain-ole standard python interactive interpreter, you are going to need to understand what a matplotlib backend is [What is a backend?](#).

With the TkAgg backend, which uses the Tkinter user interface toolkit, you can use matplotlib from an arbitrary non-gui python shell. Just set your backend : TkAgg and interactive : True in your `matplotlibrc` file (see [Customizing matplotlib](#)) and fire up python. Then:

```
>>> from pylab import *
>>> plot([1,2,3])
>>> xlabel('hi mom')
```

should work out of the box. This is also likely to work with recent versions of the qt4agg and gtkagg backends, and with the macosx backend on the Macintosh. Note, in batch mode, i.e. when making figures from scripts, interactive mode can be slow since it redraws the figure with each command. So you may want to think carefully before making this the default behavior via the `matplotlibrc` file instead of using the functions listed in the next section.

Gui shells are at best problematic, because they have to run a mainloop, but interactive plotting also involves a mainloop. Ipython has sorted all this out for the primary matplotlib backends. There may be other shells and IDEs that also work with matplotlib in interactive mode, but one obvious candidate does not: the python IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend.

7.2.3 Controlling interactive updating

The *interactive* property of the pyplot interface controls whether a figure canvas is drawn on every pyplot command. If *interactive* is *False*, then the figure state is updated on every plot command, but will only be drawn on explicit calls to `draw()`. When *interactive* is *True*, then every pyplot command triggers a draw.

The pyplot interface provides 4 commands that are useful for interactive control.

`isinteractive()` returns the interactive setting *True|False*

`ion()` turns interactive mode on

`ioff()` turns interactive mode off

`draw()` forces a figure redraw

When working with a big figure in which drawing is expensive, you may want to turn matplotlib's interactive setting off temporarily to avoid the performance hit:

```
>>> #create big-expensive-figure
>>> ioff()      # turn updates off
>>> title('now how much would you pay?')
>>> xticklabels(fontsize=20, color='green')
>>> draw()      # force a draw
>>> savefig('alldone', dpi=300)
>>> close()
>>> ion()       # turn updating back on
>>> plot(rand(20), mfc='g', mec='r', ms=40, mew=4, ls='--', lw=3)
```

7.3 Event handling and picking

matplotlib works with a number of user interface toolkits (wxpython, tkinter, qt4, gtk, and macosx) and in order to support features like interactive panning and zooming of figures, it is helpful to the developers to have an API for interacting with the figure via key presses and mouse movements that is “GUI neutral” so we don't have to repeat a lot of code across the different user interfaces. Although the event handling API is GUI neutral, it is based on the GTK model, which was the first user interface matplotlib supported. The events that are triggered are also a bit richer vis-a-vis matplotlib than standard GUI events, including information like which `matplotlib.axes.Axes` the event occurred in. The events also understand the matplotlib coordinate system, and report event locations in both pixel and data coordinates.

7.3.1 Event connections

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

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.random.rand(10))

def onclick(event):
    print('button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          (event.button, event.x, event.y, event.xdata, event.ydata))

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

The `FigureCanvas` method `mpl_connect()` returns a connection id which is simply an integer. When you want to disconnect the callback, just call:

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

Note: The canvas retains only weak references to the callbacks. Therefore if a callback is a method of

a class instance, you need to retain a reference to that instance. Otherwise the instance will be garbage-collected and the callback will vanish.

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

Event name	Class and description
'button_press_event'	<i>MouseEvent</i> - mouse button is pressed
'button_release_event'	<i>MouseEvent</i> - mouse button is released
'draw_event'	<i>DrawEvent</i> - canvas draw
'key_press_event'	<i>KeyEvent</i> - key is pressed
'key_release_event'	<i>KeyEvent</i> - key is released
'motion_notify_event'	<i>MouseEvent</i> - mouse motion
'pick_event'	<i>PickEvent</i> - an object in the canvas is selected
'resize_event'	<i>ResizeEvent</i> - figure canvas is resized
'scroll_event'	<i>MouseEvent</i> - mouse scroll wheel is rolled
'figure_enter_event'	<i>LocationEvent</i> - mouse enters a new figure
'figure_leave_event'	<i>LocationEvent</i> - mouse leaves a figure
'axes_enter_event'	<i>LocationEvent</i> - mouse enters a new axes
'axes_leave_event'	<i>LocationEvent</i> - mouse leaves an axes

7.3.2 Event attributes

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

name the event name

canvas the FigureCanvas instance generating the event

guiEvent the GUI event that triggered the matplotlib event

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

x x position - pixels from left of canvas

y y position - pixels from bottom of canvas

inaxes the `Axes` instance if mouse is over axes

xdata x coord of mouse in data coords

ydata y coord of mouse in data coords

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

```
from matplotlib import pyplot as plt

class LineBuilder:
```

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

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

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click to build line segments')
line, = ax.plot([0], [0]) # empty line
linebuilder = LineBuilder(line)

plt.show()
```

The `MouseEvent` that we just used is a `LocationEvent`, so we have access to the data and pixel coordinates in `event.x` and `event.xdata`. In addition to the `LocationEvent` attributes, it has

`button` button pressed None, 1, 2, 3, ‘up’, ‘down’ (up and down are used for scroll events)
`key` the key pressed: None, any character, ‘shift’, ‘win’, or ‘control’

Draggable rectangle exercise

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

Here is the solution:

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

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

    def connect(self):
        'connect to all the events we need'
```

```

self.cidpress = self.rect.figure.canvas.mpl_connect(
    'button_press_event', self.on_press)
self.cidrelease = self.rect.figure.canvas.mpl_connect(
    'button_release_event', self.on_release)
self.cidmotion = self.rect.figure.canvas.mpl_connect(
    'motion_notify_event', self.on_motion)

def on_press(self, event):
    'on button press we will see if the mouse is over us and store some data'
    if event.inaxes != self.rect.axes: return

    contains, attrd = self.rect.contains(event)
    if not contains: return
    print('event contains', self.rect.xy)
    x0, y0 = self.rect.xy
    self.press = x0, y0, event.xdata, event.ydata

def on_motion(self, event):
    'on motion we will move the rect if the mouse is over us'
    if self.press is None: return
    if event.inaxes != self.rect.axes: return
    x0, y0, xpress, ypress = self.press
    dx = event.xdata - xpress
    dy = event.ydata - ypress
    #print('x0=%f, xpress=%f, event.xdata=%f, dx=%f, x0+dx=%f' %
    #      (x0, xpress, event.xdata, dx, x0+dx))
    self.rect.set_x(x0+dx)
    self.rect.set_y(y0+dy)

    self.rect.figure.canvas.draw()

def on_release(self, event):
    'on release we reset the press data'
    self.press = None
    self.rect.figure.canvas.draw()

def disconnect(self):
    'disconnect all the stored connection ids'
    self.rect.figure.canvas.mpl_disconnect(self.cidpress)
    self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
    self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()

```

Extra credit: use the animation blit techniques discussed in the [animations recipe](#) to make the animated drawing faster and smoother.

Extra credit solution:

```
# draggable rectangle with the animation blit techniques; see
# http://www.scipy.org/Cookbook/Matplotlib/Animations
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    lock = None # only one can be animated at a time
    def __init__(self, rect):
        self.rect = rect
        self.press = None
        self.background = None

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

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
        if DraggableRectangle.lock is not None: return
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print('event contains', self.rect.xy)
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata
        DraggableRectangle.lock = self

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

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

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

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if DraggableRectangle.lock is not self:
            return


```

```

if event.inaxes != self.rect.axes: return
x0, y0, xpress, ypress = self.press
dx = event.xdata - xpress
dy = event.ydata - ypress
self.rect.set_x(x0+dx)
self.rect.set_y(y0+dy)

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

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

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

def on_release(self, event):
    'on release we reset the press data'
    if DraggableRectangle.lock is not self:
        return

    self.press = None
    DraggableRectangle.lock = None

    # turn off the rect animation property and reset the background
    self.rect.set_animated(False)
    self.background = None

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

def disconnect(self):
    'disconnect all the stored connection ids'
    self.rect.figure.canvas.mpl_disconnect(self.cidpress)
    self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
    self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()

```

7.3.3 Mouse enter and leave

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

```
"""
Illustrate the figure and axes enter and leave events by changing the
frame colors on enter and leave
"""

import matplotlib.pyplot as plt

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

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

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

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

fig1 = plt.figure()
fig1.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig1.add_subplot(211)
ax2 = fig1.add_subplot(212)

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

fig2 = plt.figure()
fig2.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig2.add_subplot(211)
ax2 = fig2.add_subplot(212)

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

plt.show()
```

7.3.4 Object picking

You can enable picking by setting the `picker` property of an `Artist` (e.g., a matplotlib `Line2D`, `Text`, `Patch`, `Polygon`, `AxesImage`, etc...)

There are a variety of meanings of the `picker` property:

None picking is disabled for this artist (default)

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

float if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event.

function if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event. The signature is `hit, props = picker(artist, mouseevent)` to determine the hit test. If the mouse event is over the artist, return `hit=True` and `props` is a dictionary of properties you want added to the `PickEvent` attributes

After you have enabled an artist for picking by setting the `picker` property, you need to connect to the figure canvas `pick_event` to get pick callbacks on mouse press events. e.g.:

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

The `PickEvent` which is passed to your callback is always fired with two attributes:

mouseevent the mouse event that generate the pick event. The mouse event in turn has attributes like `x` and `y` (the coords in display space, e.g., pixels from left, bottom) and `xdata`, `ydata` (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which `Axes` the mouse is over, etc. See `matplotlib.backend_bases.MouseEvent` for details.

artist the `Artist` that generated the pick event.

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

Simple picking example

In the example below, we set the line picker property to a scalar, so it represents a tolerance in points (72 points per inch). The `onpick` callback function will be called when the pick event is within the tolerance distance from the line, and has the indices of the data vertices that are within the pick distance tolerance. Our `onpick` callback function simply prints the data that are under the pick location. Different matplotlib Artists can attach different data to the `PickEvent`. For example, `Line2D` attaches the `ind` property, which are

the indices into the line data under the pick point. See `pick()` for details on the `PickEvent` properties of the line. Here is the code:

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

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on points')

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

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

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

plt.show()
```

Picking exercise

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

Exercise solution:

```
"""
compute the mean and stddev of 100 data sets and plot mean vs stddev.
When you click on one of the mu, sigma points, plot the raw data from
the dataset that generated the mean and stddev
"""

import numpy as np
import matplotlib.pyplot as plt

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

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5) # 5 points tolerance
```

```
def onpick(event):

    if event.artist!=line: return True

    N = len(event.ind)
    if not N: return True

    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N,1,subplotnum+1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f\nsigma=%1.3f'%(xs[dataind], ys[dataind]),
               transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

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

plt.show()
```

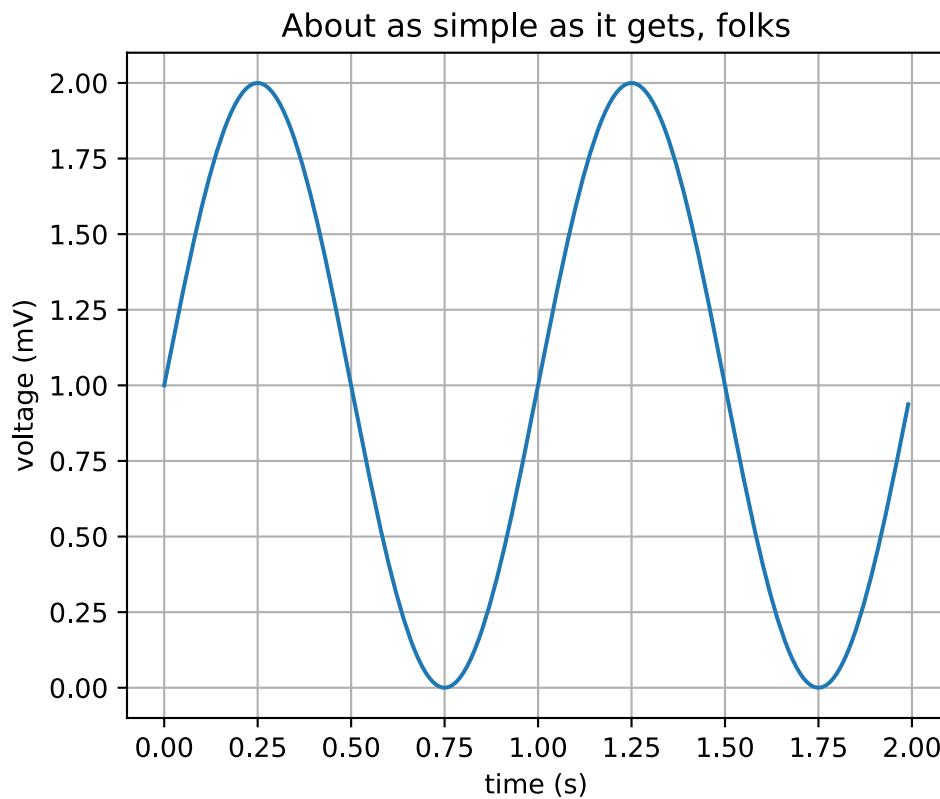

SELECTED EXAMPLES

8.1 Screenshots

Here you'll find a host of example plots with the code that generated them.

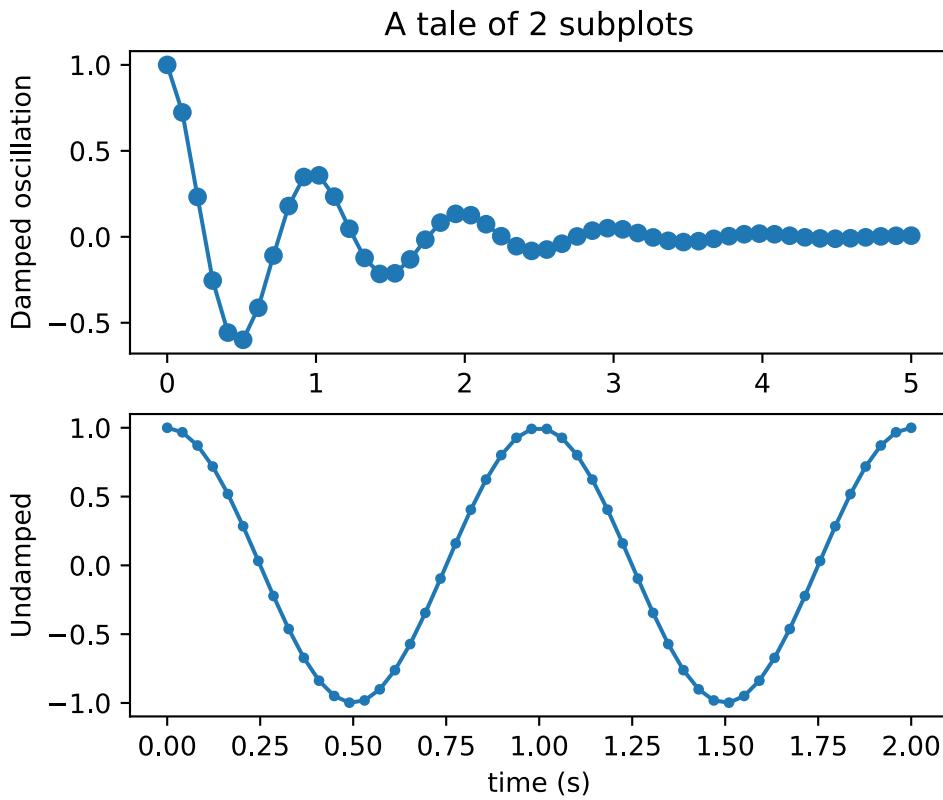
8.1.1 Simple Plot

Here's a very basic `plot()` with text labels:



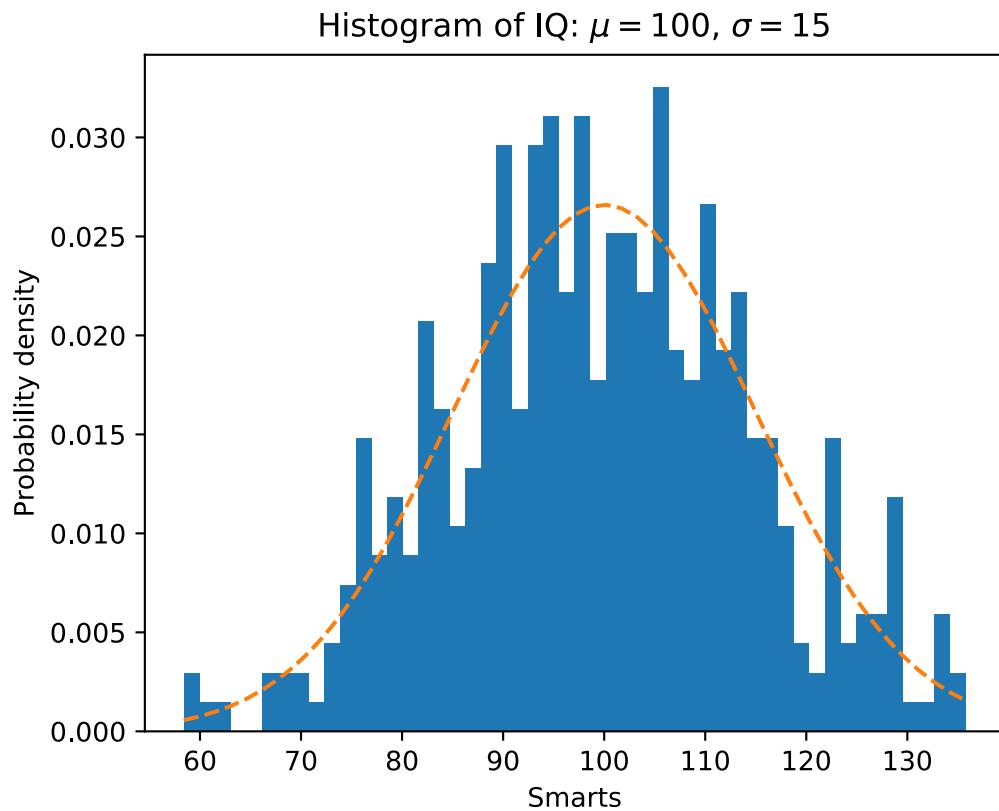
8.1.2 Subplot demo

Multiple axes (i.e. subplots) are created with the `subplot()` command:



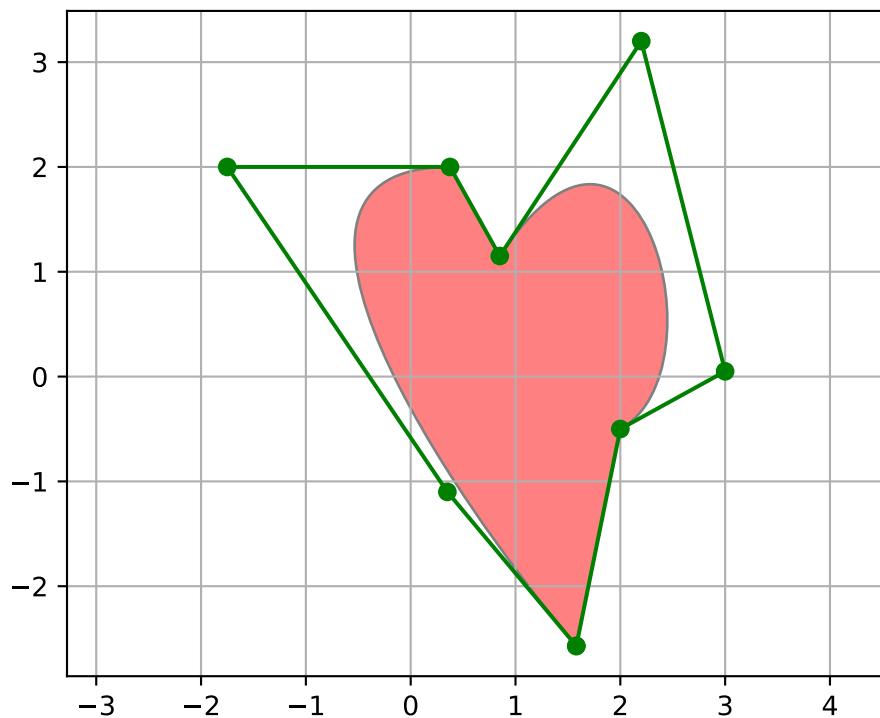
8.1.3 Histograms

The `hist()` command automatically generates histograms and returns the bin counts or probabilities:



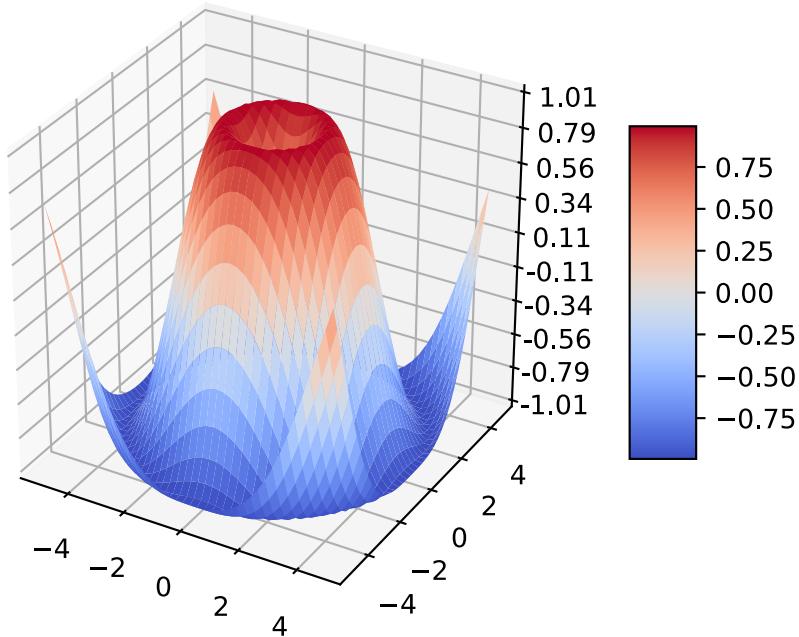
8.1.4 Path demo

You can add arbitrary paths in matplotlib using the `matplotlib.path` module:



8.1.5 mplot3d

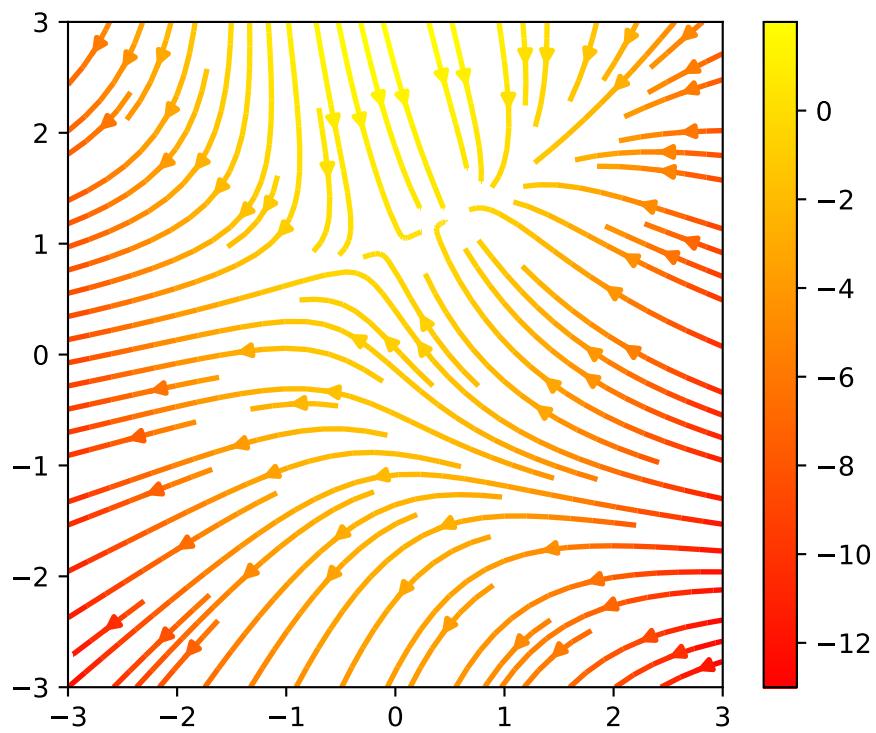
The mplot3d toolkit (see [mplot3d tutorial](#) and [mplot3d Examples](#)) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts.

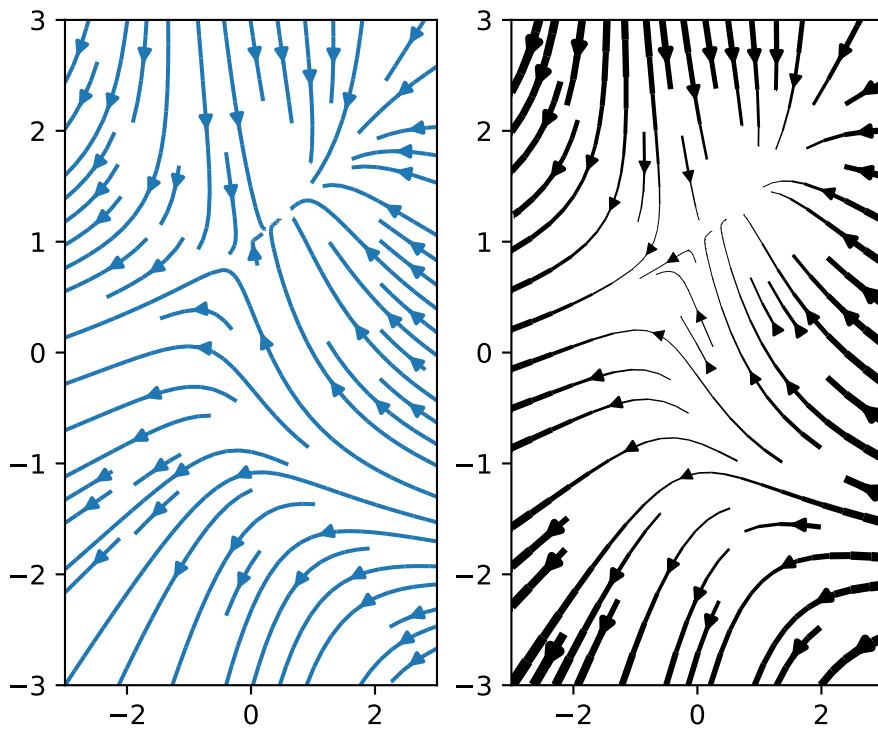


Thanks to John Porter, Jonathon Taylor, Reinier Heeres, and Ben Root for the `mplot3d` toolkit. This toolkit is included with all standard matplotlib installs.

8.1.6 Streamplot

The `streamplot()` function plots the streamlines of a vector field. In addition to simply plotting the streamlines, it allows you to map the colors and/or line widths of streamlines to a separate parameter, such as the speed or local intensity of the vector field.

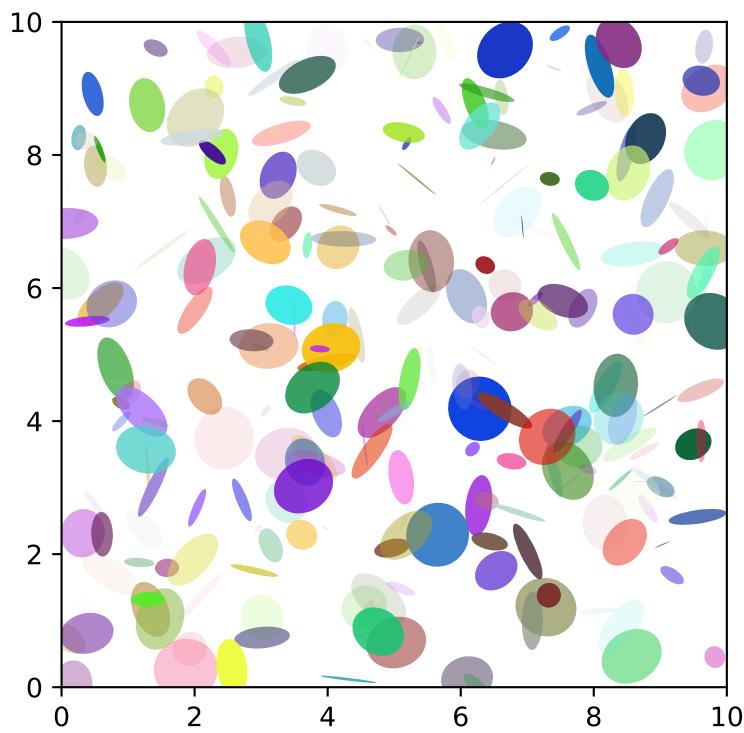




This feature complements the `quiver()` function for plotting vector fields. Thanks to Tom Flannaghan and Tony Yu for adding the streamplot function.

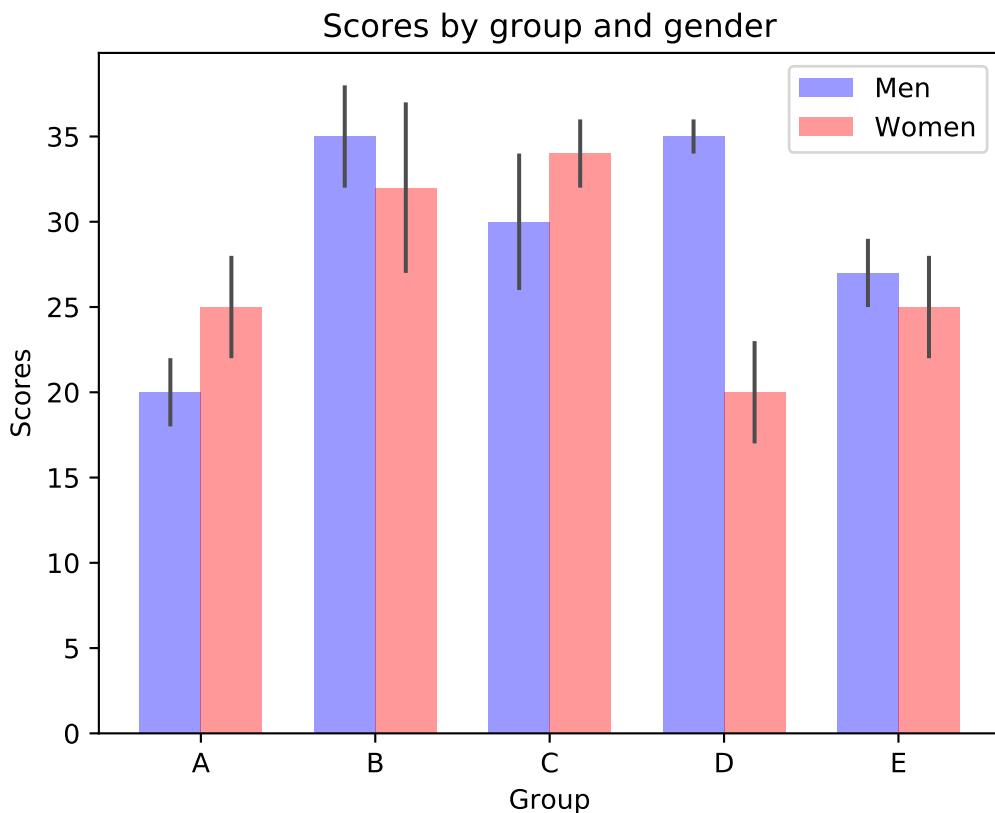
8.1.7 Ellipses

In support of the [Phoenix](#) mission to Mars (which used matplotlib to display ground tracking of spacecraft), Michael Droettboom built on work by Charlie Moad to provide an extremely accurate 8-spline approximation to elliptical arcs (see [Arc](#)), which are insensitive to zoom level.



8.1.8 Bar charts

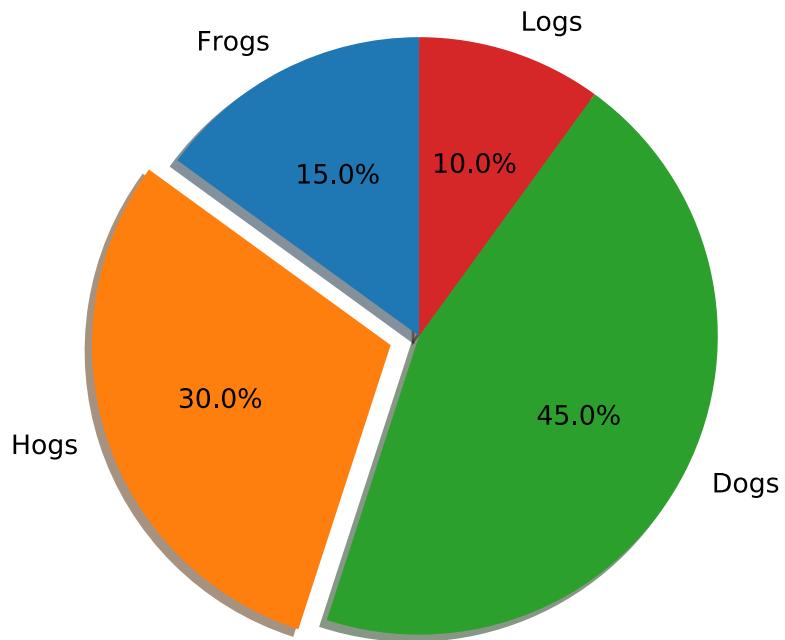
Bar charts are simple to create using the `bar()` command, which includes customizations such as error bars:



It's also simple to create stacked bars (`bar_stacked.py`), or horizontal bar charts (`barh_demo.py`).

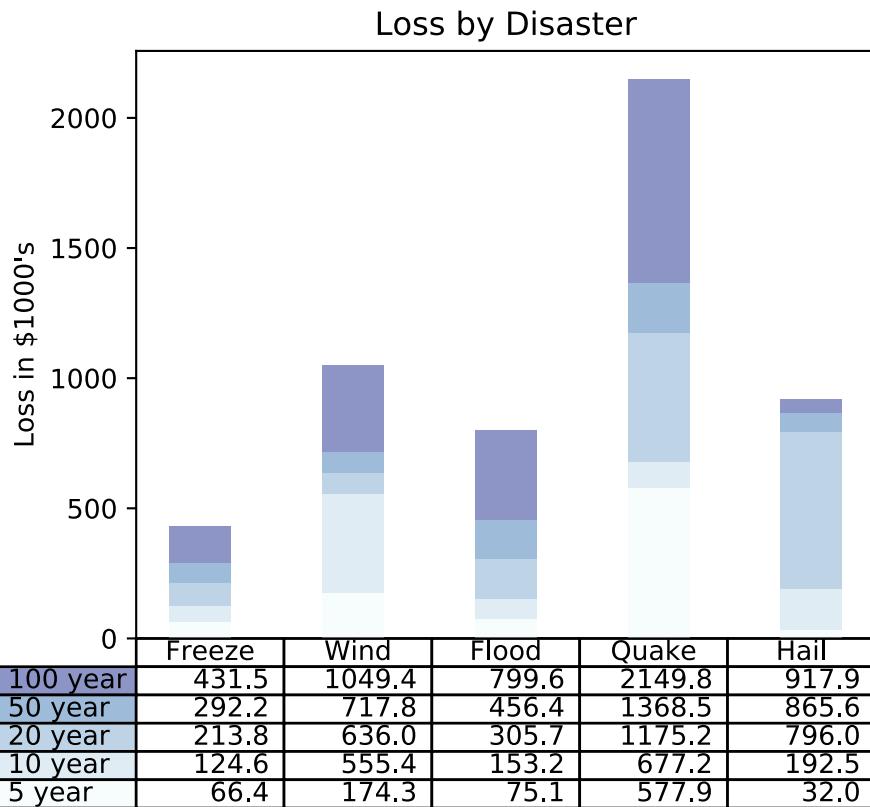
8.1.9 Pie charts

The `pie()` command allows you to easily create pie charts. Optional features include auto-labeling the percentage of area, exploding one or more wedges from the center of the pie, and a shadow effect. Take a close look at the attached code, which generates this figure in just a few lines of code.



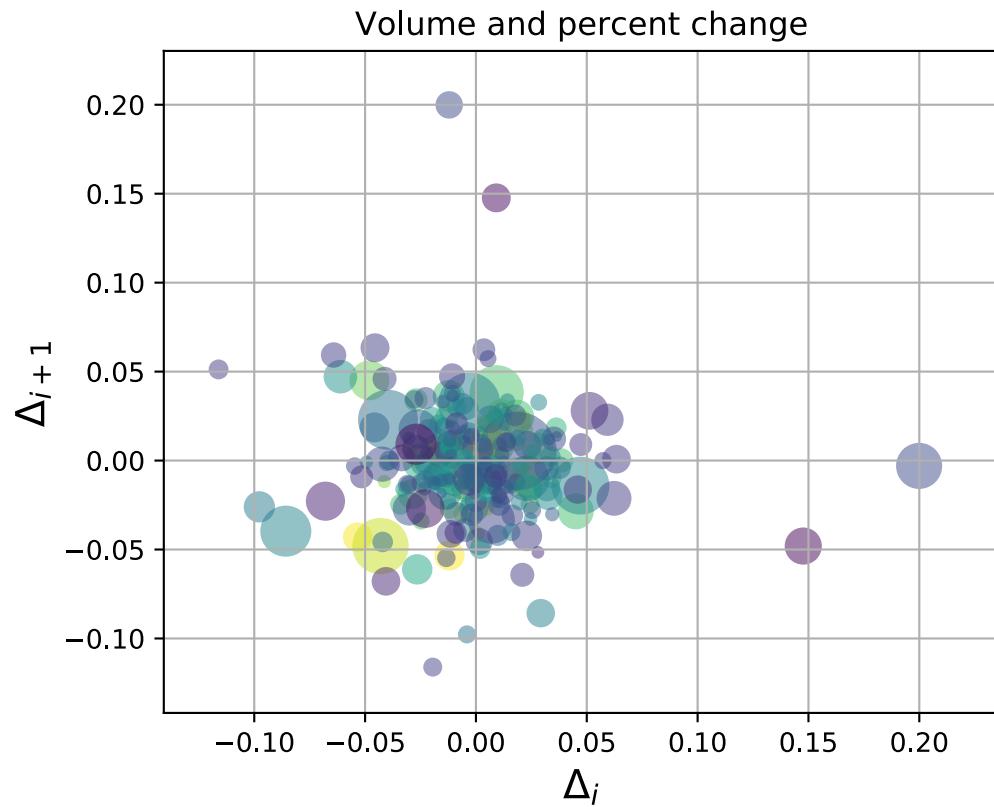
8.1.10 Table demo

The `table()` command adds a text table to an axes.



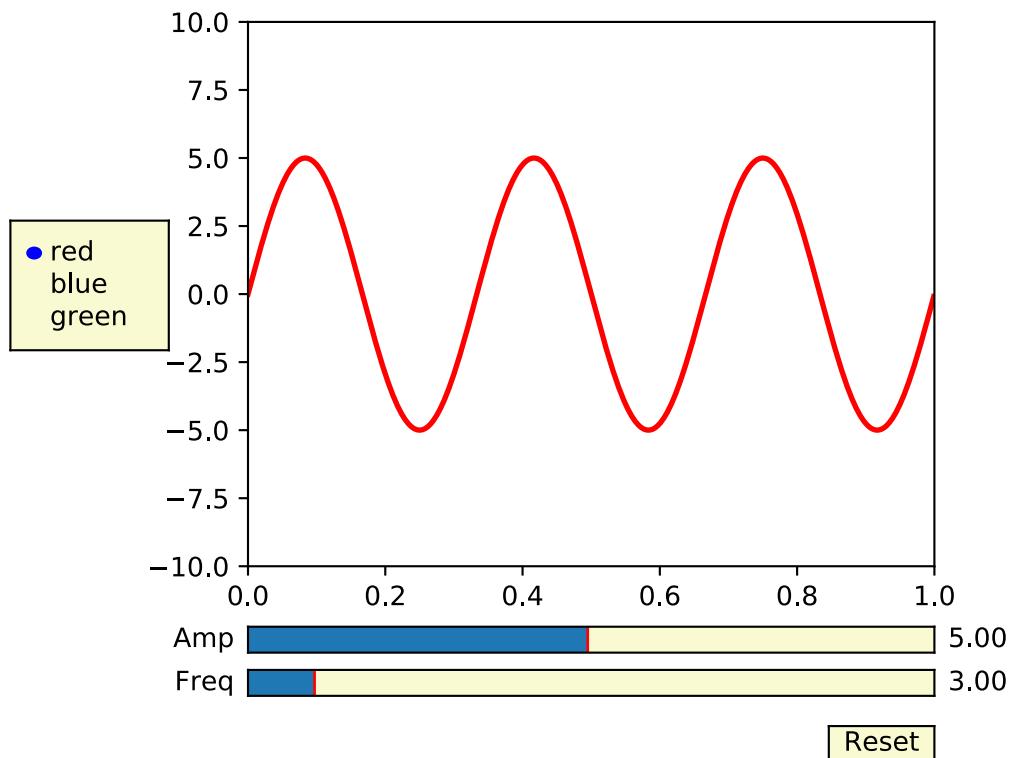
8.1.11 Scatter demo

The `scatter()` command makes a scatter plot with (optional) size and color arguments. This example plots changes in Google's stock price, with marker sizes reflecting the trading volume and colors varying with time. Here, the alpha attribute is used to make semitransparent circle markers.



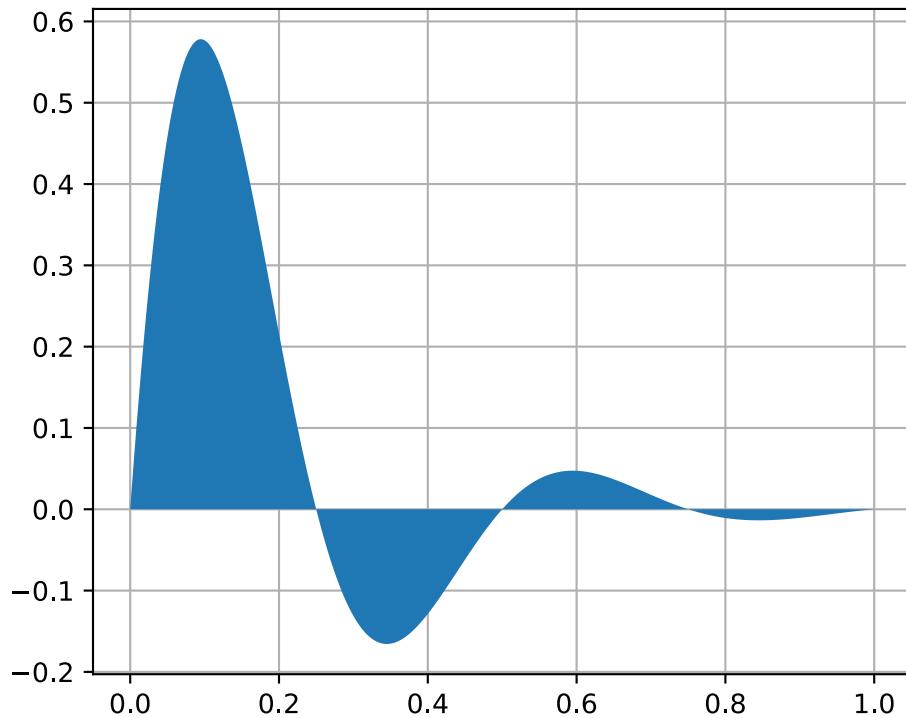
8.1.12 Slider demo

Matplotlib has basic GUI widgets that are independent of the graphical user interface you are using, allowing you to write cross GUI figures and widgets. See [matplotlib.widgets](#) and the widget examples.



8.1.13 Fill demo

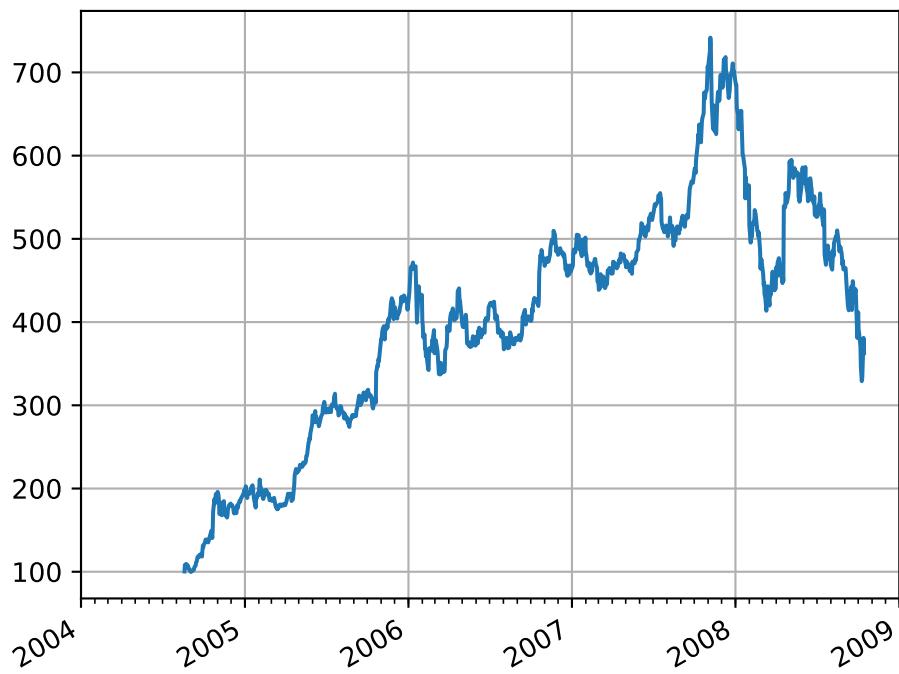
The `fill()` command lets you plot filled curves and polygons:



Thanks to Andrew Straw for adding this function.

8.1.14 Date demo

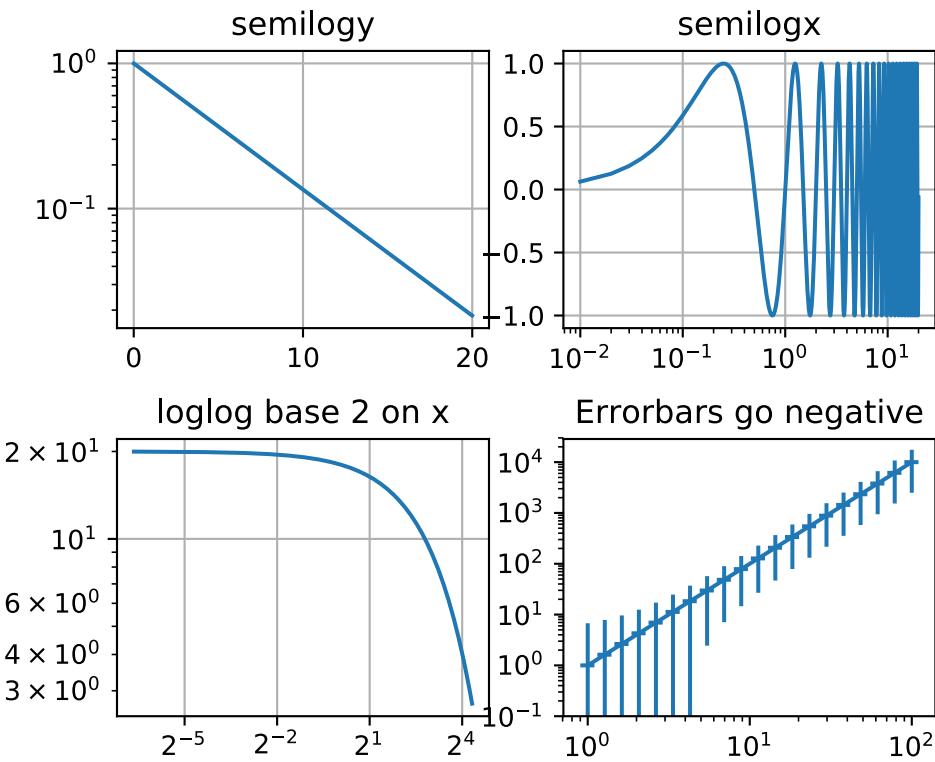
You can plot date data with major and minor ticks and custom tick formatters for both.



See [`matplotlib.ticker`](#) and [`matplotlib.dates`](#) for details and usage.

8.1.15 Log plots

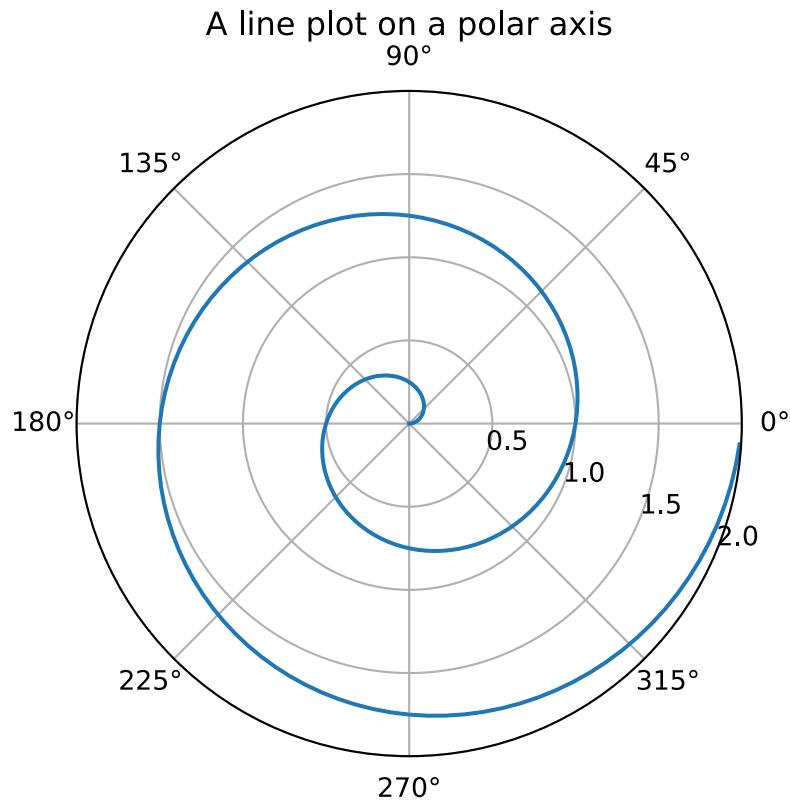
The [`semilogx\(\)`](#), [`semilogy\(\)`](#) and [`loglog\(\)`](#) functions simplify the creation of logarithmic plots.



Thanks to Andrew Straw, Darren Dale and Gregory Lielens for contributions log-scaling infrastructure.

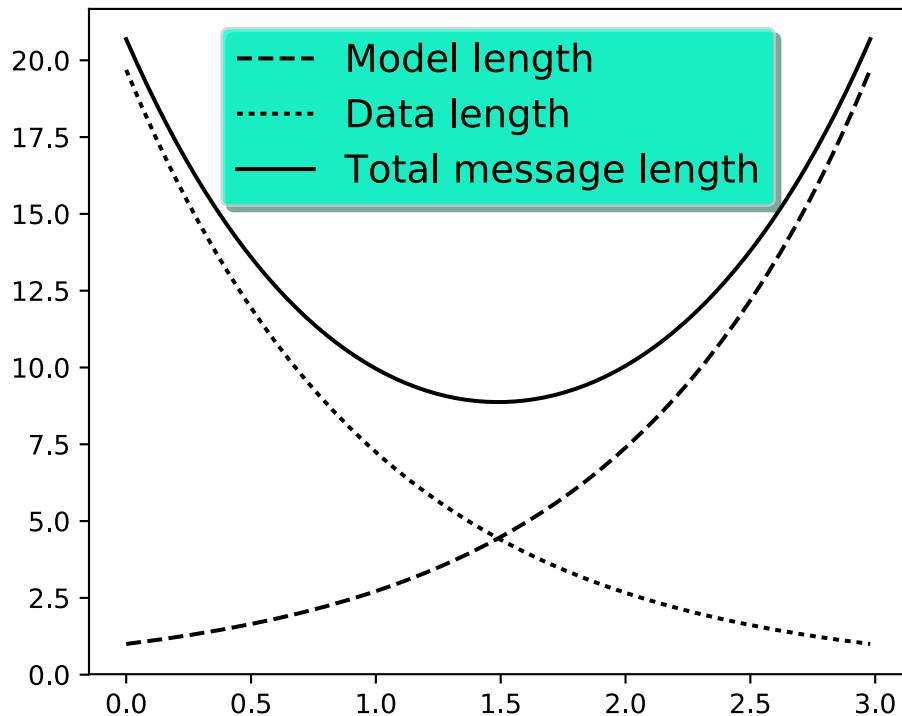
8.1.16 Polar plots

The `polar()` command generates polar plots.



8.1.17 Legends

The `legend()` command automatically generates figure legends, with MATLAB-compatible legend placement commands.



Thanks to Charles Twardy for input on the legend command.

8.1.18 Mathtext_examples

Below is a sampling of the many TeX expressions now supported by matplotlib's internal mathtext engine. The mathtext module provides TeX style mathematical expressions using FreeType and the DejaVu, BaKoMa computer modern, or STIX fonts. See the [`matplotlib.mathtext`](#) module for additional details.

Matplotlib's math rendering engine

$$W_{\delta_1\rho_1\sigma_2}^{3\beta} = U_{\delta_1\rho_1}^{3\beta} + \frac{1}{8\pi^2} \int_{\alpha_2}^{\alpha_2} d\alpha'_2 \left[\frac{U_{\delta_1\rho_1}^{2\beta} - \alpha'_2 U_{\rho_1\sigma_2}^{1\beta}}{U_{\rho_1\sigma_2}^{0\beta}} \right]$$

Subscripts and superscripts:

$$\alpha_i > \beta_i, \quad \alpha_{i+1}^j = \sin(2\pi f_j t_i) e^{-5t_i/\tau}, \quad \dots$$

Fractions, binomials and stacked numbers:

$$\frac{3}{4}, \quad \binom{3}{4}, \quad \frac{3}{4}, \quad \left(\frac{5-\frac{1}{x}}{4}\right), \quad \dots$$

Radicals:

$$\sqrt{2}, \quad \sqrt[3]{X}, \quad \dots$$

Fonts:

Roman , Italic , Typewriter or *CALLIGRAPHY*

Accents:

$$\acute{a}, \bar{a}, \check{a}, \grave{a}, \ddot{a}, \grave{\grave{a}}, \hat{a}, \tilde{a}, \vec{a}, \widehat{xyz}, \widetilde{xyz}, \dots$$

Greek, Hebrew:

$$\alpha, \beta, \chi, \delta, \lambda, \mu, \Delta, \Gamma, \Omega, \Phi, \Pi, \Upsilon, \nabla, \aleph, \beth, \daleth, \beth_1, \beth_2, \beth_3, \beth_4,$$

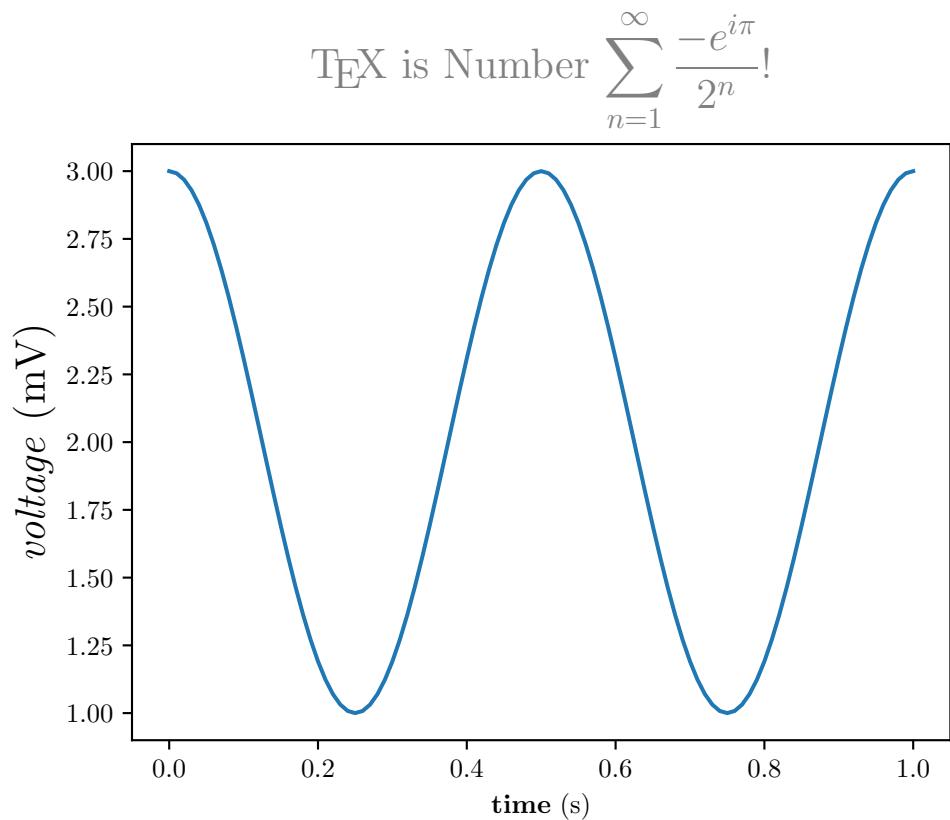
Delimiters, functions and Symbols:

$$\lfloor, \lceil, \oint, \prod, \sum, \log, \sin, \approx, \oplus, \star, \propto, \infty, \partial, \Re,$$

Matplotlib's mathtext infrastructure is an independent implementation and does not require TeX or any external packages installed on your computer. See the tutorial at [Writing mathematical expressions](#).

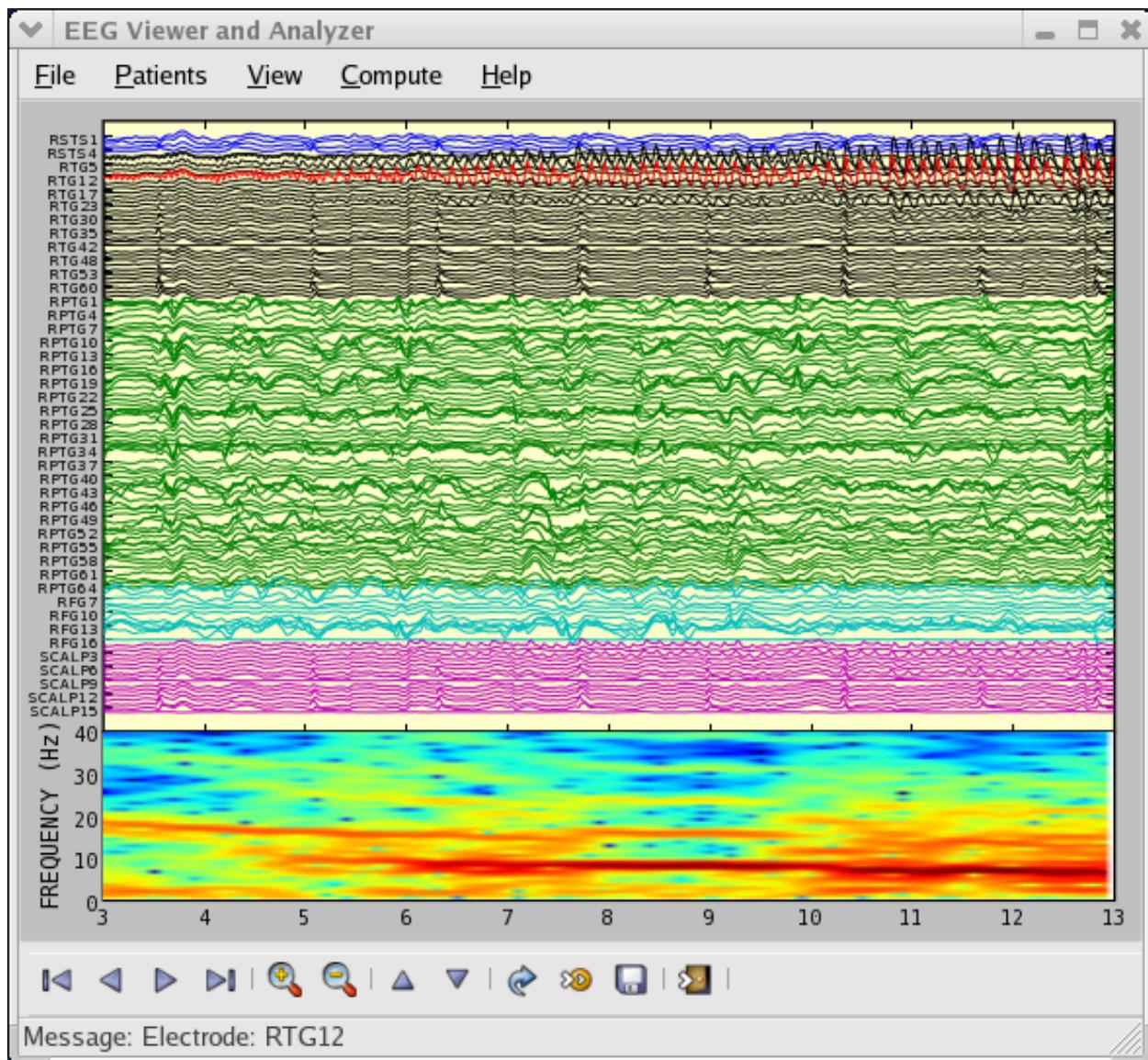
8.1.19 Native TeX rendering

Although matplotlib's internal math rendering engine is quite powerful, sometimes you need TeX. Matplotlib supports external TeX rendering of strings with the `usetex` option.



8.1.20 EEG demo

You can embed matplotlib into pygtk, wx, Tk, or Qt applications. Here is a screenshot of an EEG viewer called [pbrain](#).



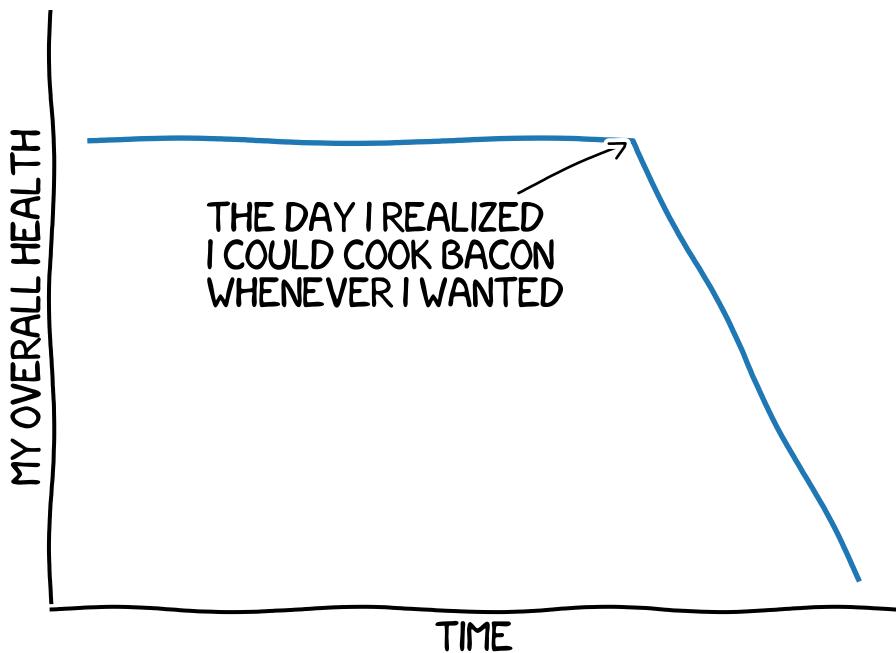
The lower axes uses `specgram()` to plot the spectrogram of one of the EEG channels.

For examples of how to embed matplotlib in different toolkits, see:

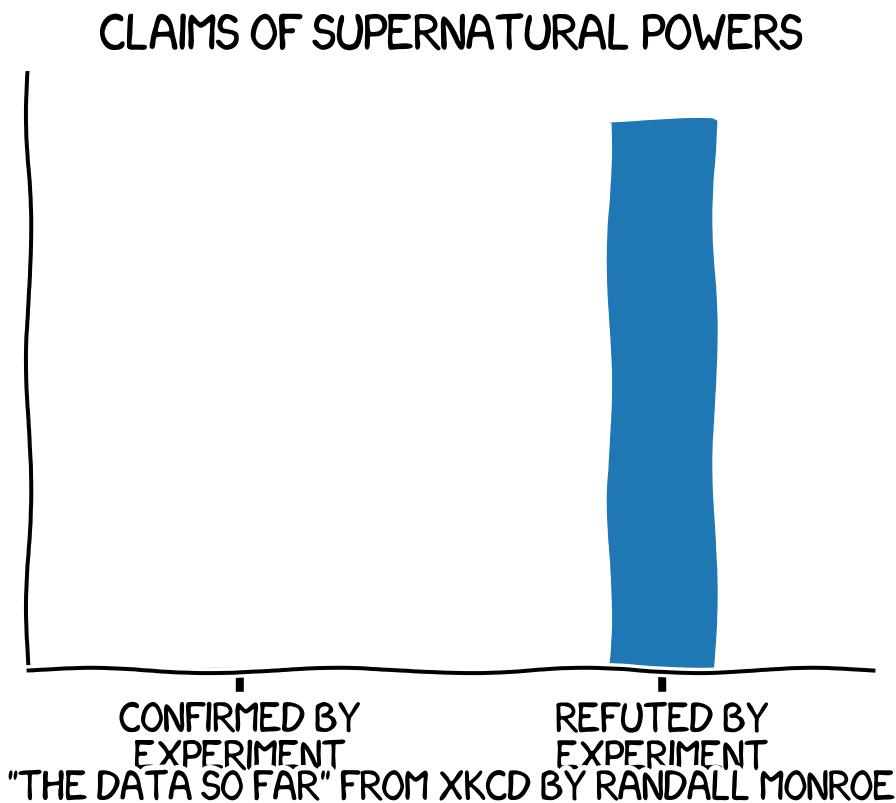
- *user_interfaces example code: embedding_in_gtk2.py*
- *user_interfaces example code: embedding_in_wx2.py*
- *user_interfaces example code: mpl_with_glade.py*
- *user_interfaces example code: embedding_in_qt4.py*
- *user_interfaces example code: embedding_in_tk.py*

8.1.21 XKCD-style sketch plots

matplotlib supports plotting in the style of xkcd.



"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE



8.2 Our Favorite Recipes

Here is a collection of short tutorials, examples and code snippets that illustrate some of the useful idioms and tricks to make snazzier figures and overcome some matplotlib warts.

8.2.1 Sharing axis limits and views

It's common to make two or more plots which share an axis, e.g., two subplots with time as a common axis. When you pan and zoom around on one, you want the other to move around with you. To facilitate this, matplotlib Axes support a `sharex` and `sharey` attribute. When you create a `subplot()` or `axes()` instance, you can pass in a keyword indicating what axes you want to share with

```
In [96]: t = np.arange(0, 10, 0.01)
In [97]: ax1 = plt.subplot(211)
In [98]: ax1.plot(t, np.sin(2*np.pi*t))
Out[98]: [

```

```
In [100]: ax2.plot(t, np.sin(4*np.pi*t))
Out[100]: [
```

8.2.2 Easily creating subplots

In early versions of matplotlib, if you wanted to use the pythonic API and create a figure instance and from that create a grid of subplots, possibly with shared axes, it involved a fair amount of boilerplate code. e.g.

```
# old style
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(223, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(224, sharex=ax1, sharey=ax1)
```

Fernando Perez has provided a nice top level method to create in `subplots()` (note the “s” at the end) everything at once, and turn on x and y sharing for the whole bunch. You can either unpack the axes individually:

```
# new style method 1; unpack the axes
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, sharey=True)
ax1.plot(x)
```

or get them back as a numrows x numcolumns object array which supports numpy indexing:

```
# new style method 2; use an axes array
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
axs[0,0].plot(x)
```

8.2.3 Fixing common date annoyances

matplotlib allows you to natively plots python datetime instances, and for the most part does a good job picking tick locations and string formats. There are a couple of things it does not handle so gracefully, and here are some tricks to help you work around them. We’ll load up some sample date data which contains `datetime.date` objects in a numpy record array:

```
In [63]: datafile = cbook.get_sample_data('goog.npy')

In [64]: r = np.load(datafile).view(np.recarray)

In [65]: r.dtype
Out[65]: dtype([('date', '|O4'), ('', '|V4'), ('open', '<f8'),
   high', '<f8'), ('low', '<f8'), ('close', '<f8'),
   ('volume', '<i8'), ('adj_close', '<f8')])

In [66]: r.date
Out[66]:
array([2004-08-19, 2004-08-20, 2004-08-23, ..., 2008-10-10, 2008-10-13,
   2008-10-14], dtype=object)
```

The dtype of the numpy record array for the field `date` is `|O` which means it is a 4-byte python object pointer; in this case the objects are `datetime.date` instances, which we can see when we print some samples in the ipython terminal window.

If you plot the data,

```
In [67]: plot(r.date, r.close)
Out[67]: [<matplotlib.lines.Line2D object at 0x92a6b6c>]
```

you will see that the x tick labels are all squashed together.

Default date handling can cause overlapping labels

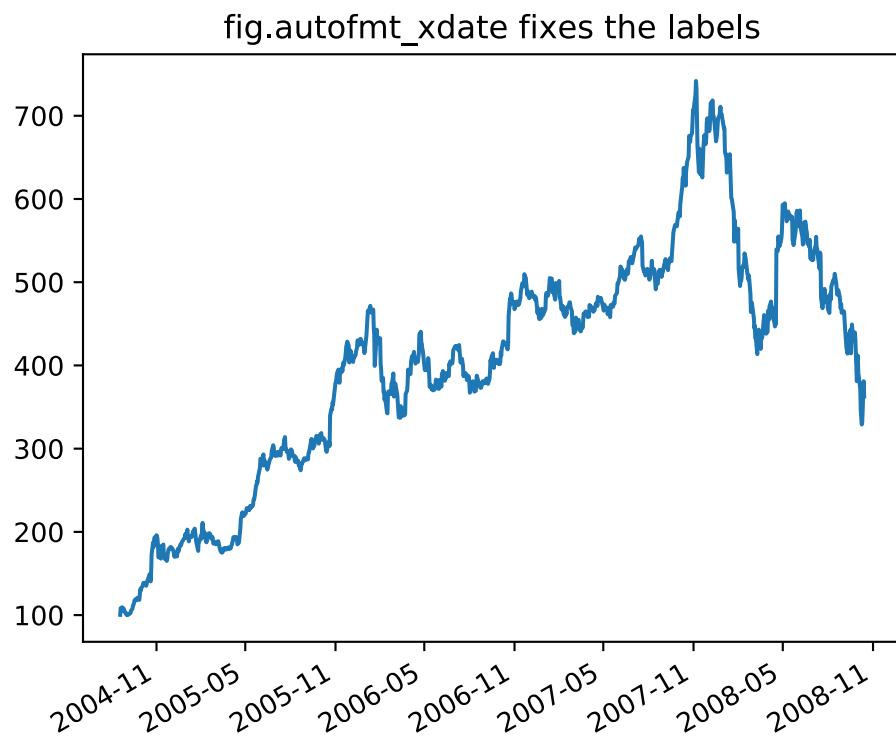


Another annoyance is that if you hover the mouse over the window and look in the lower right corner of the matplotlib toolbar ([Interactive navigation](#)) at the x and y coordinates, you see that the x locations are formatted the same way the tick labels are, e.g., “Dec 2004”. What we’d like is for the location in the toolbar to have a higher degree of precision, e.g., giving us the exact date out mouse is hovering over. To fix the first problem, we can use `matplotlib.figure.Figure.autofmt_xdate()` and to fix the second problem we can use the `ax(fmt_xdata` attribute which can be set to any function that takes a scalar and returns a string. matplotlib has a number of date formatters built in, so we’ll use one of those.

```
plt.close('all')
fig, ax = plt.subplots(1)
ax.plot(r.date, r.close)

# rotate and align the tick labels so they look better
fig.autofmt_xdate()
```

```
# use a more precise date string for the x axis locations in the
# toolbar
import matplotlib.dates as mdates
ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')
plt.title('fig.autofmt_xdate fixes the labels')
```



Now when you hover your mouse over the plotted data, you'll see date format strings like 2004-12-01 in the toolbar.

8.2.4 Fill Between and Alpha

The `fill_between()` function generates a shaded region between a min and max boundary that is useful for illustrating ranges. It has a very handy `where` argument to combine filling with logical ranges, e.g., to just fill in a curve over some threshold value.

At its most basic level, `fill_between` can be used to enhance a graph's visual appearance. Let's compare two graphs of a financial times with a simple line plot on the left and a filled line on the right.

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

import matplotlib.cbook as cbook
```

```
# load up some sample financial data
datafile = cbook.get_sample_data('goog.npy')
try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. However this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later.
    r = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    r = np.load(datafile).view(np.recarray)
# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1,2, sharex=True, sharey=True)

pricemin = r.close.min()

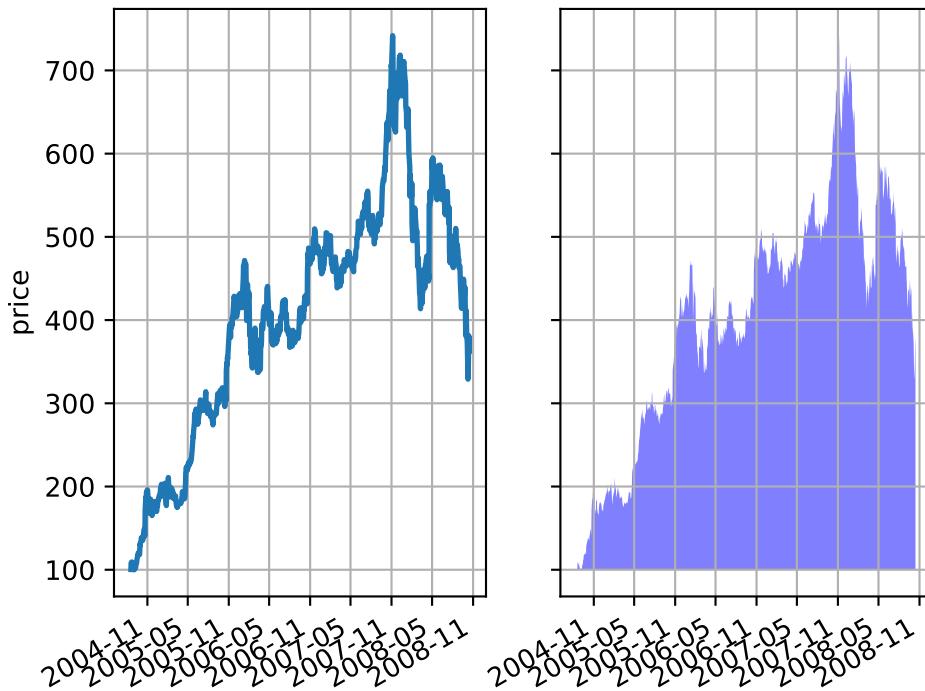
ax1.plot(r.date, r.close, lw=2)
ax2.fill_between(r.date, pricemin, r.close, facecolor='blue', alpha=0.5)

for ax in ax1, ax2:
    ax.grid(True)

ax1.set_ylabel('price')
for label in ax2.get_yticklabels():
    label.set_visible(False)

fig.suptitle('Google (GOOG) daily closing price')
fig.autofmt_xdate()
```

Google (GOOG) daily closing price



The alpha channel is not necessary here, but it can be used to soften colors for more visually appealing plots. In other examples, as we'll see below, the alpha channel is functionally useful as the shaded regions can overlap and alpha allows you to see both. Note that the postscript format does not support alpha (this is a postscript limitation, not a matplotlib limitation), so when using alpha save your figures in PNG, PDF or SVG.

Our next example computes two populations of random walkers with a different mean and standard deviation of the normal distributions from which the steps are drawn. We use shared regions to plot +/- one standard deviation of the mean position of the population. Here the alpha channel is useful, not just aesthetic.

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

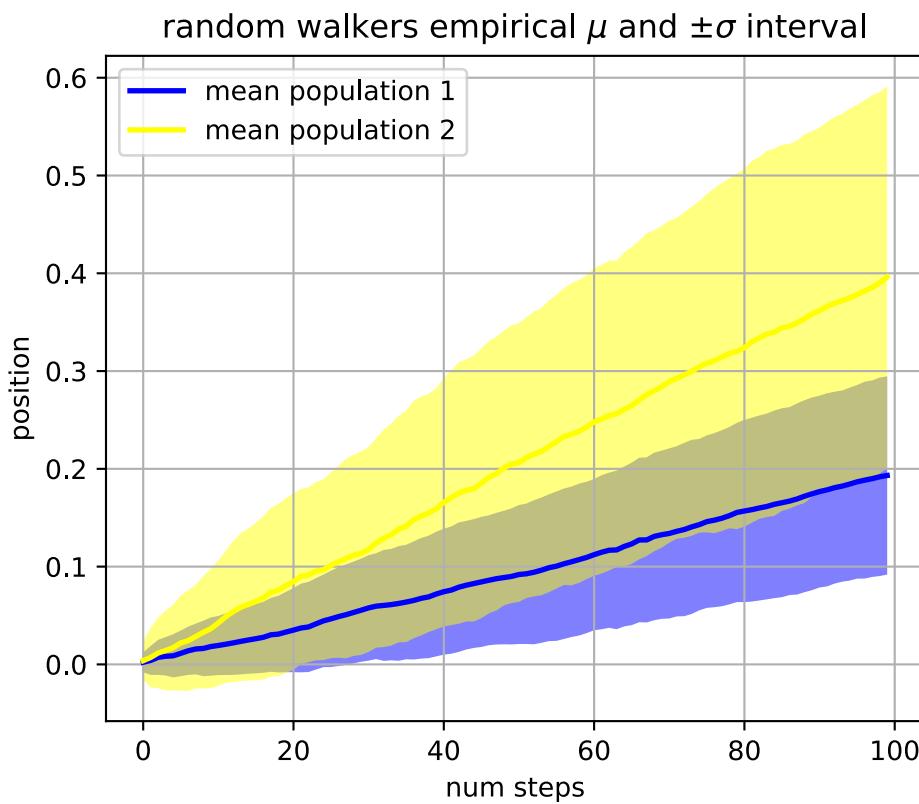
Nsteps, Nwalkers = 100, 250
t = np.arange(Nsteps)

# an (Nsteps x Nwalkers) array of random walk steps
S1 = 0.002 + 0.01*np.random.randn(Nsteps, Nwalkers)
S2 = 0.004 + 0.02*np.random.randn(Nsteps, Nwalkers)

# an (Nsteps x Nwalkers) array of random walker positions
X1 = S1.cumsum(axis=0)
X2 = S2.cumsum(axis=0)
```

```
# Nsteps length arrays empirical means and standard deviations of both
# populations over time
mu1 = X1.mean(axis=1)
sigma1 = X1.std(axis=1)
mu2 = X2.mean(axis=1)
sigma2 = X2.std(axis=1)

# plot it!
fig, ax = plt.subplots(1)
ax.plot(t, mu1, lw=2, label='mean population 1', color='blue')
ax.plot(t, mu2, lw=2, label='mean population 2', color='yellow')
ax.fill_between(t, mu1+sigma1, mu1-sigma1, facecolor='blue', alpha=0.5)
ax.fill_between(t, mu2+sigma2, mu2-sigma2, facecolor='yellow', alpha=0.5)
ax.set_title('random walkers empirical $\mu$ and $\pm\sigma$ interval')
ax.legend(loc='upper left')
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```



The `where` keyword argument is very handy for highlighting certain regions of the graph. `where` takes a boolean mask the same length as the `x`, `ymin` and `ymax` arguments, and only fills in the region where the boolean mask is True. In the example below, we simulate a single random walker and compute the analytic mean and standard deviation of the population positions. The population mean is shown as the black dashed line, and the plus/minus one sigma deviation from the mean is shown as the yellow filled region. We use the where mask `X>upper_bound` to find the region where the walker is above the one sigma boundary, and

shade that region blue.

```
np.random.seed(1234)

Nsteps = 500
t = np.arange(Nsteps)

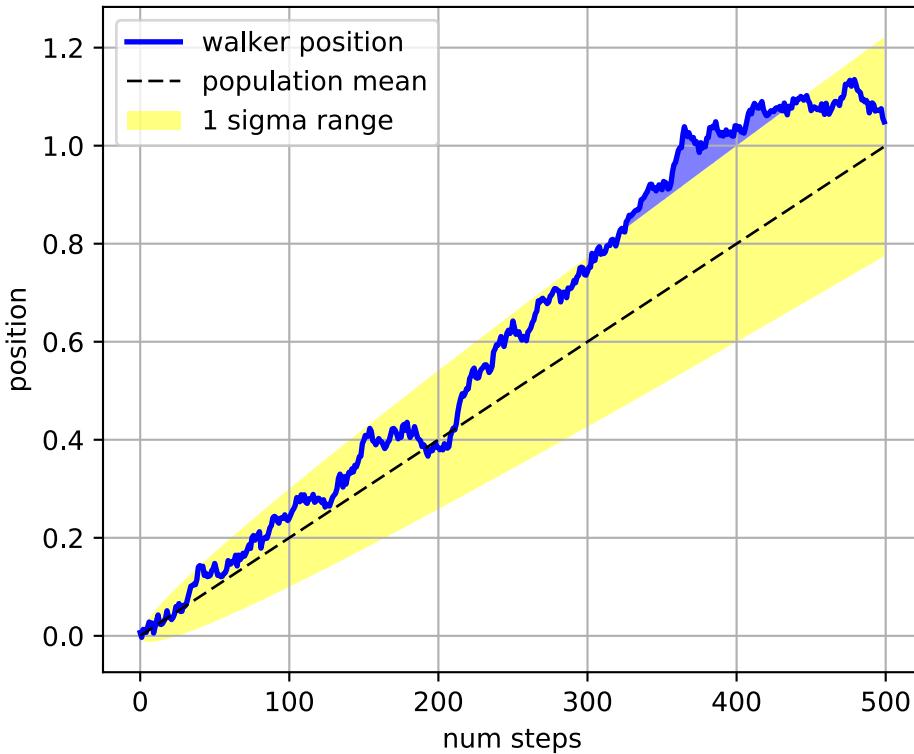
mu = 0.002
sigma = 0.01

# the steps and position
S = mu + sigma*np.random.randn(Nsteps)
X = S.cumsum()

# the 1 sigma upper and lower analytic population bounds
lower_bound = mu*t - sigma*np.sqrt(t)
upper_bound = mu*t + sigma*np.sqrt(t)

fig, ax = plt.subplots(1)
ax.plot(t, X, lw=2, label='walker position', color='blue')
ax.plot(t, mu*t, lw=1, label='population mean', color='black', ls='--')
ax.fill_between(t, lower_bound, upper_bound, facecolor='yellow', alpha=0.5,
                 label='1 sigma range')
ax.legend(loc='upper left')

# here we use the where argument to only fill the region where the
# walker is above the population 1 sigma boundary
ax.fill_between(t, upper_bound, X, where=X>upper_bound, facecolor='blue', alpha=0.5)
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```



Another handy use of filled regions is to highlight horizontal or vertical spans of an axes – for that matplotlib has some helper functions `axhspan()` and `axvspan()` and example [pylab_examples example code: axhspan_demo.py](#).

8.2.5 Transparent, fancy legends

Sometimes you know what your data looks like before you plot it, and may know for instance that there won't be much data in the upper right hand corner. Then you can safely create a legend that doesn't overlap your data:

```
ax.legend(loc='upper right')
```

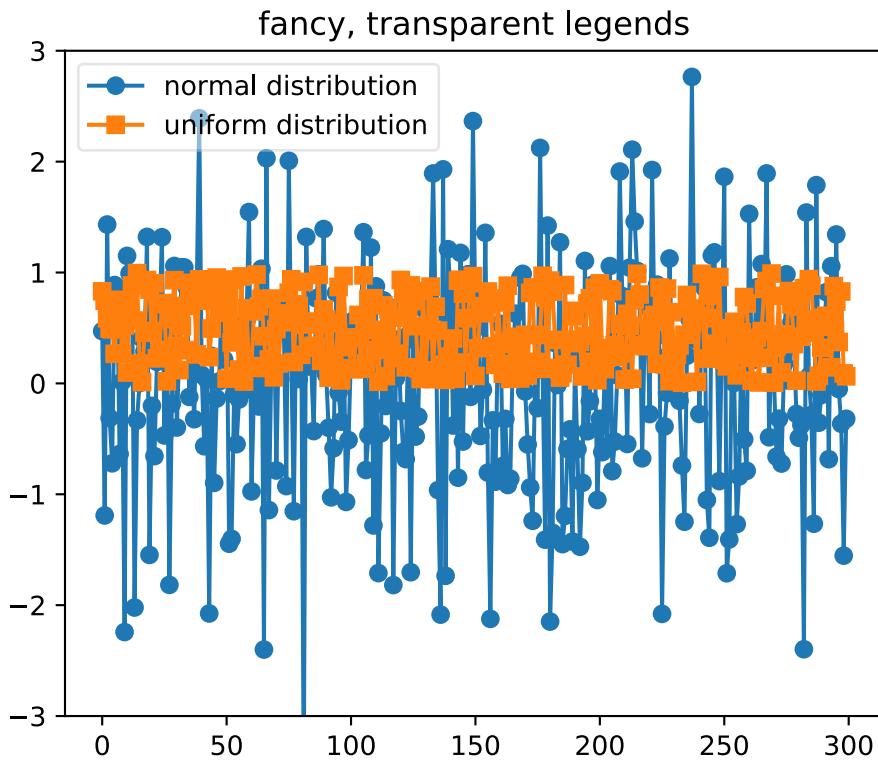
Other times you don't know where your data is, and `loc='best'` will try and place the legend:

```
ax.legend(loc='best')
```

but still, your legend may overlap your data, and in these cases it's nice to make the legend frame transparent.

```
np.random.seed(1234)
fig, ax = plt.subplots(1)
ax.plot(np.random.randn(300), 'o-', label='normal distribution')
ax.plot(np.random.rand(300), 's-', label='uniform distribution')
ax.set_ylim(-3, 3)
ax.legend(loc='best', fancybox=True, framealpha=0.5)
```

```
ax.set_title('fancy, transparent legends')
```



8.2.6 Placing text boxes

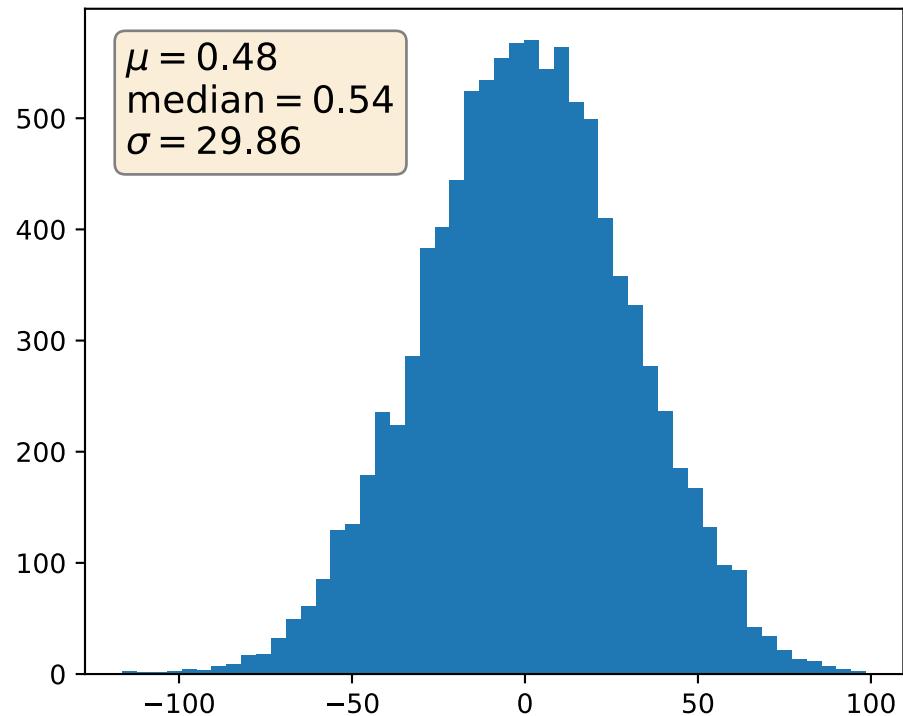
When decorating axes with text boxes, two useful tricks are to place the text in axes coordinates (see [Transformations Tutorial](#)), so the text doesn't move around with changes in x or y limits. You can also use the `bbox` property of text to surround the text with a `Patch` instance – the `bbox` keyword argument takes a dictionary with keys that are `Patch` properties.

```
np.random.seed(1234)
fig, ax = plt.subplots(1)
x = 30*np.random.randn(10000)
mu = x.mean()
median = np.median(x)
sigma = x.std()
textstr = '$\\mu=%2f$\\n$\\mathrm{median}=%2f$\\n$\\sigma=%2f$'%(mu, median, sigma)

ax.hist(x, 50)
# these are matplotlib.patch.Patch properties
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

# place a text box in upper left in axes coords
```

```
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)
```



WHAT'S NEW IN MATPLOTLIB

For a list of all of the issues and pull requests since the last revision, see the [GitHub Stats](#).

Table of Contents

- *What's new in matplotlib*
 - *New in matplotlib 2.0*
 - * *Default style changes*
 - * *Improved color conversion API and RGBA support*
 - * *New Configuration (rcParams)*
 - * *Qualitative colormaps*
 - * *Axis offset label now responds to labelcolor*
 - * *Improved offset text choice*
 - * *Style parameter blacklist*
 - * *Change in default font*
 - * *Faster text rendering*
 - * *Improvements for the Qt figure options editor*
 - * *Improved image support*
 - * *Support for HiDPI (Retina) displays in the NbAgg and WebAgg backends*
 - * *Change in the default animation codec*
 - * *Deprecated support for mencoder in animation*
 - * *Boxplot Zorder Keyword Argument*
 - * *Filled + and x markers*
 - * *rcount and ccount for plot_surface()*
 - * *Streamplot Zorder Keyword Argument Changes*
 - *Previous Whats New*

9.1 New in matplotlib 2.0

Note: matplotlib 2.0 supports Python 2.7, and 3.4+

9.1.1 Default style changes

The major changes in v2.0 are related to overhauling the default styles.

Changes to the default style

The most important changes in matplotlib 2.0 are the changes to the default style.

While it is impossible to select the best default for all cases, these are designed to work well in the most common cases.

A ‘classic’ style sheet is provided so reverting to the 1.x default values is a single line of python

```
mpl.style.use('classic')
```

See *The matplotlibrc file* for details about how to persistently and selectively revert many of these changes.

Table of Contents

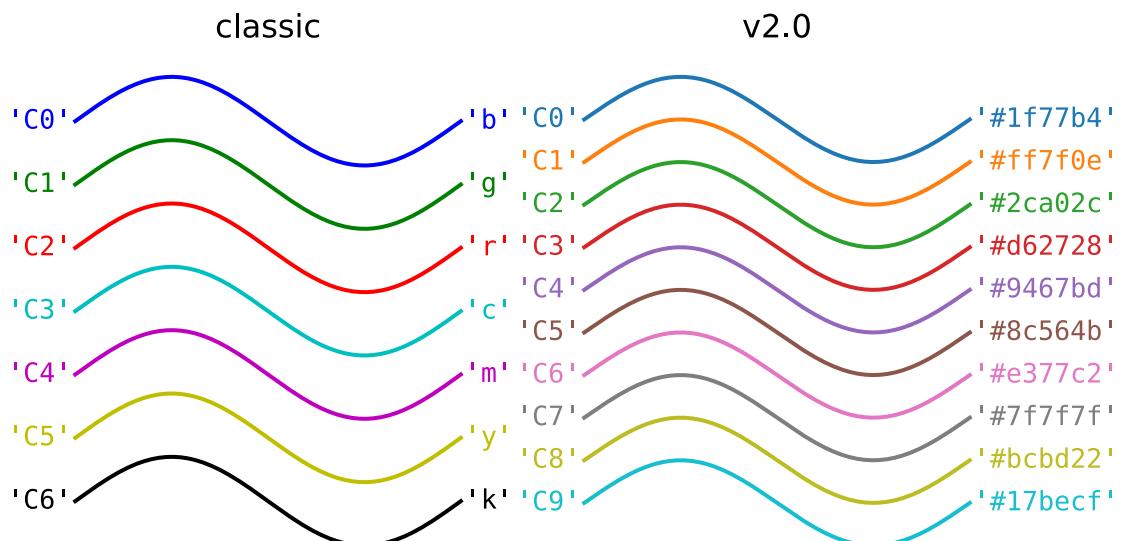
- *Colors, color cycles, and color maps*
 - *Colors in default property cycle*
 - *Colormap*
 - *Interactive figures*
 - *Grid lines*
- *Figure size, font size, and screen dpi*
- *Plotting functions*
 - *scatter*
 - *plot*
 - *errorbar*
 - *boxplot*
 - *fill_between and fill_betweenx*
 - *Patch edges and color*
 - *bar and barh*

- *Hatching*
- *Fonts*
 - *Normal text*
 - *Math text*
- *Legends*
- *Image*
 - *Interpolation*
 - *Colormapping pipeline*
 - *Shading*
- *Plot layout*
 - *Auto limits*
 - *Z-order*
 - *Ticks*
 - *Tick label formatting*
- *mplot3d*

Colors, color cycles, and color maps

Colors in default property cycle

The colors in the default property cycle have been changed from `['b', 'g', 'r', 'c', 'm', 'y', 'k']` to the category10 color palette used by [Vega](#) and [d3](#) originally developed at Tableau.



In addition to changing the colors, an additional method to specify colors was added. Previously, the default colors were the single character short-hand notations for red, green, blue, cyan, magenta, yellow, and black. This made them easy to type and usable in the abbreviated style string in `plot`, however the new default colors are only specified via hex values. To access these colors outside of the property cycling the notation for colors '`CN`', where `N` takes values 0-9, was added to denote the first 10 colors in `mpl.rcParams['axes.prop_cycle']`. See [Specifying Colors](#) for more details.

To restore the old color cycle use

```
from cycler import cycler
mpl.rcParams['axes.prop_cycle'] = cycler(color='bgrcmyk')
```

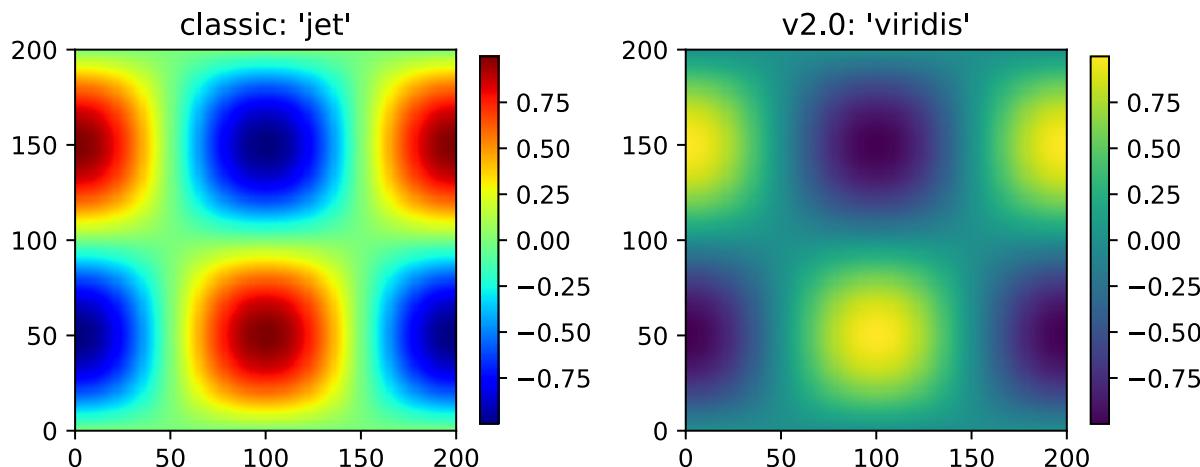
or set

```
axes.prop_cycle : cycler('color', 'bgrcmyk')
```

in your `matplotlibrc` file.

Colormap

The new default color map used by `matplotlib.cm.ScalarMappable` instances is '`viridis`' (aka option D).



For an introduction to color theory and how '`viridis`' was generated watch Nathaniel Smith and Stéfan van der Walt's talk from SciPy2015. See [here](#) for many more details about the other alternatives and the tools used to create the color map. For details on all of the color maps available in matplotlib see [Choosing Colormaps](#).

The previous default can be restored using

```
mpl.rcParams['image.cmap'] = 'jet'
```

or setting

```
image.cmap      : 'jet'
```

in your `matplotlibrc` file; however this is strongly discouraged.

Interactive figures

The default interactive figure background color has changed from grey to white, which matches the default background color used when saving.

The previous defaults can be restored by

```
mpl.rcParams['figure.facecolor'] = '0.75'
```

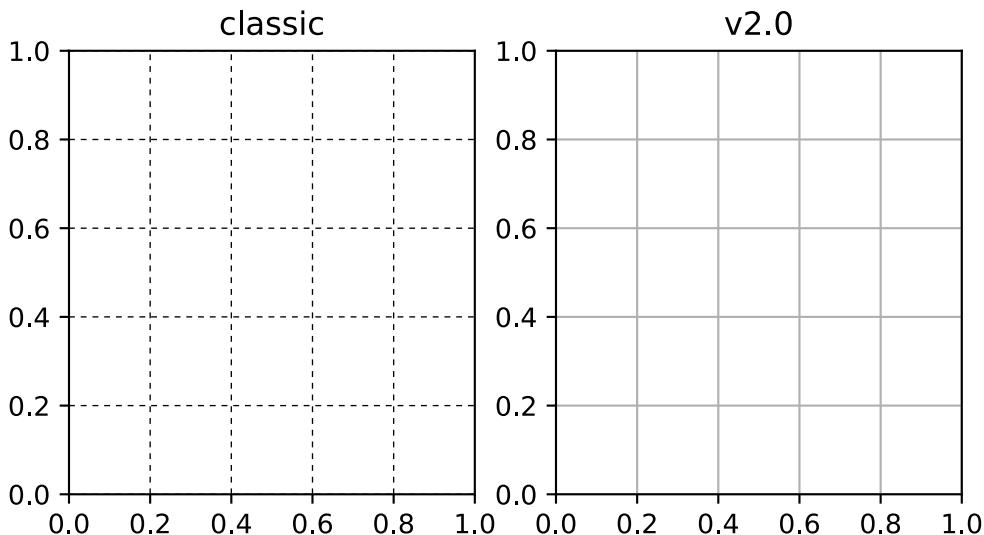
or by setting

```
figure.facecolor : '0.75'
```

in your `matplotlibrc` file.

Grid lines

The default style of grid lines was changed from black dashed lines to thicker solid light grey lines.



The previous default can be restored by using:

```
mpl.rcParams['grid.color'] = 'k'
mpl.rcParams['grid.linestyle'] = ':'
mpl.rcParams['grid.linewidth'] = 0.5
```

or by setting:

```
grid.color      : k          # grid color
grid.linestyle  : :          # dotted
grid.linewidth  : 0.5        # in points
```

in your `matplotlibrc` file.

Figure size, font size, and screen dpi

The default dpi used for on-screen display was changed from 80 dpi to 100 dpi, the same as the default dpi for saving files. Due to this change, the on-screen display is now more what-you-see-is-what-you-get for saved files. To keep the figure the same size in terms of pixels, in order to maintain approximately the same size on the screen, the default figure size was reduced from 8x6 inches to 6.4x4.8 inches. As a consequence of this the default font sizes used for the title, tick labels, and axes labels were reduced to maintain their size relative to the overall size of the figure. By default the dpi of the saved image is now the dpi of the `Figure` instance being saved.

This will have consequences if you are trying to match text in a figure directly with external text.

The previous defaults can be restored by

```
mpl.rcParams['figure.figsize'] = [8.0, 6.0]
mpl.rcParams['figure.dpi'] = 80
mpl.rcParams['savefig.dpi'] = 100

mpl.rcParams['font.size'] = 12
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['figure.titlesize'] = 'medium'
```

or by setting:

```
figure.figsize   : [8.0, 6.0]
figure.dpi       : 80
savefig.dpi     : 100

font.size        : 12.0
legend.fontsize  : 'large'
figure.titlesize : 'medium'
```

In your `matplotlibrc` file.

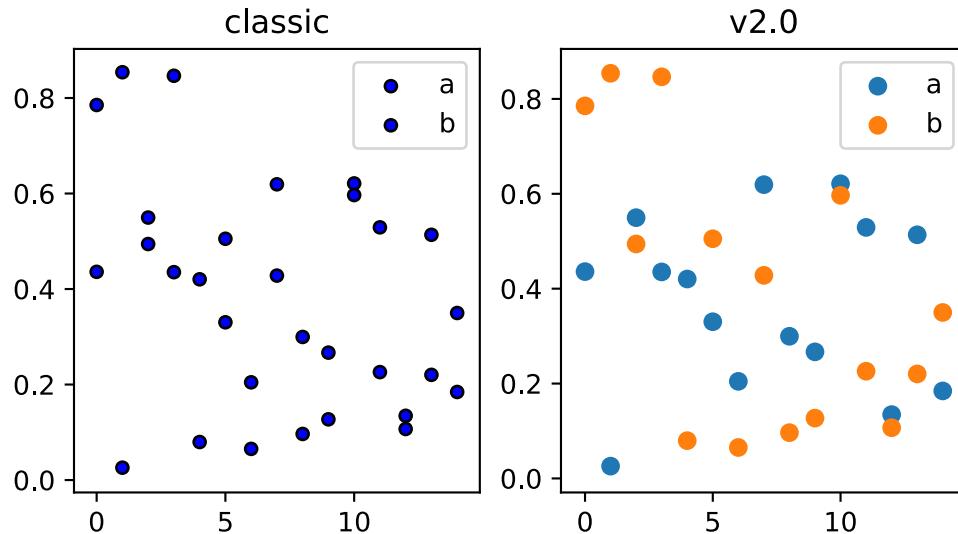
Plotting functions

`scatter`

The following changes were made to the default behavior of `scatter`

- The default size of the elements in a scatter plot is now based on the rcParam `lines.markersize` so it is consistent with `plot(X, Y, 'o')`. The old value was 20, and the new value is 36 (6^2).

- scatter markers no longer have a black edge.
- if the color of the markers is not specified it will follow the property cycle, pulling from the ‘patches’ cycle on the Axes.



The classic default behavior of `scatter` can only be recovered through `mpl.style.use('classic')`. The marker size can be recovered via

```
mpl.rcParams['lines.markersize'] = np.sqrt(20)
```

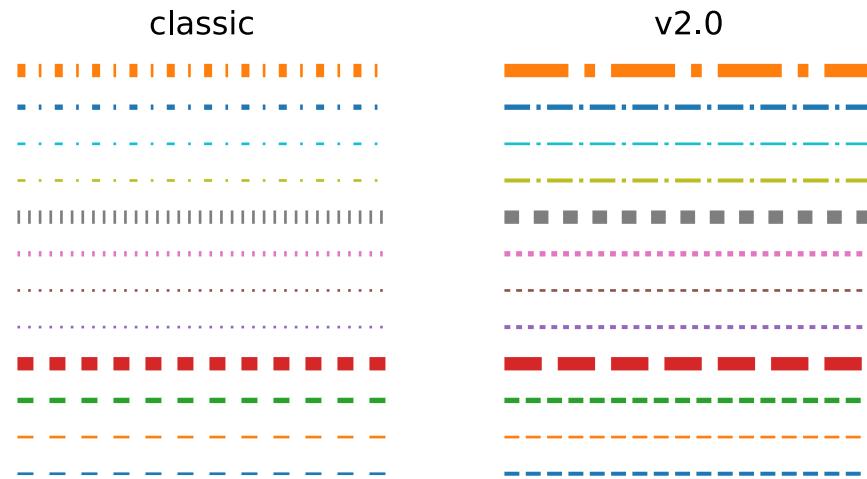
however, this will also affect the default marker size of `plot`. To recover the classic behavior on a per-call basis pass the following kwargs:

```
classic_kwargs = {'s': 20, 'edgecolors': 'k', 'c': 'b'}
```

plot

The following changes were made to the default behavior of `plot`

- the default linewidth increased from 1 to 1.5
- the dash patterns associated with '--', ':', and '-.' have changed
- the dash patterns now scale with line width



The previous defaults can be restored by setting:

```
mpl.rcParams['lines.linewidth'] = 1.0
mpl.rcParams['lines.dashed_pattern'] = [6, 6]
mpl.rcParams['lines.dashdot_pattern'] = [3, 5, 1, 5]
mpl.rcParams['lines.dotted_pattern'] = [1, 3]
mpl.rcParams['lines.scale_dashes'] = False
```

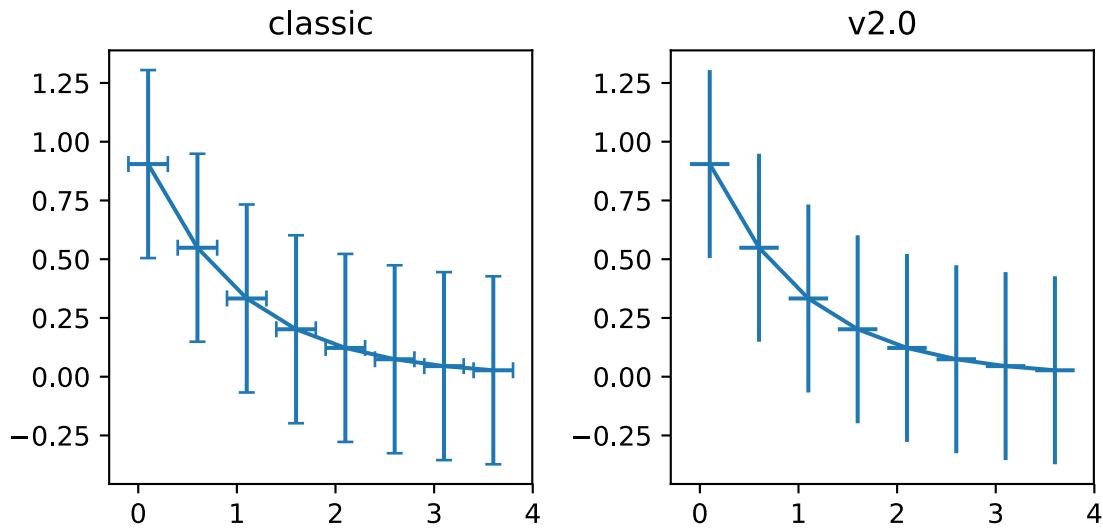
or by setting:

```
lines.linewidth    : 1.0
lines.dashed_pattern : 6, 6
lines.dashdot_pattern : 3, 5, 1, 5
lines.dotted_pattern : 1, 3
lines.scale_dashes: False
```

in your `matplotlibrc` file.

errorbar

By default, caps on the ends of errorbars are not present.



The previous defaults can be restored by setting:

```
mpl.rcParams['errorbar.capsize'] = 3
```

or by setting

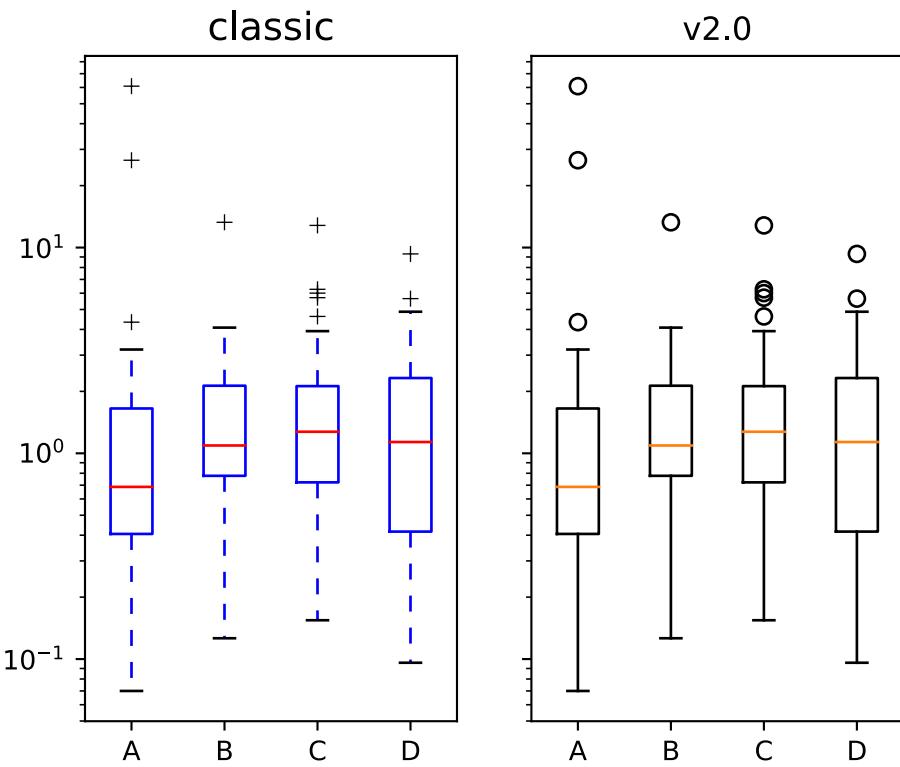
```
errorbar.capsize : 3
```

in your `matplotlibrc` file.

boxplot

Previously, boxplots were composed of a mish-mash of styles that were, for better or worse, inherited from Matlab. Most of the elements were blue, but the medians were red. The fliers (outliers) were black plus-sign symbols (+) and the whiskers were dashed lines, which created ambiguity if the (solid and black) caps were not drawn.

For the new defaults, everything is black except for the median and mean lines (if drawn), which are set to the first two elements of the current color cycle. Also, the default flier markers are now hollow circles, which maintain the ability of the plus-signs to overlap without obscuring data too much.



The previous defaults can be restored by setting:

```
mpl.rcParams['boxplot.flierprops.color'] = 'k'
mpl.rcParams['boxplot.flierprops.marker'] = '+'
mpl.rcParams['boxplot.flierprops.markerfacecolor'] = 'none'
mpl.rcParams['boxplot.flierprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.boxprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.linestyle'] = '--'
mpl.rcParams['boxplot.medianprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.marker'] = '^'
mpl.rcParams['boxplot.meanprops.markerfacecolor'] = 'r'
mpl.rcParams['boxplot.meanprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.meanprops.markersize'] = 6
mpl.rcParams['boxplot.meanprops.linestyle'] = '--'
mpl.rcParams['boxplot.meanprops.linewidth'] = 1.0
```

or by setting:

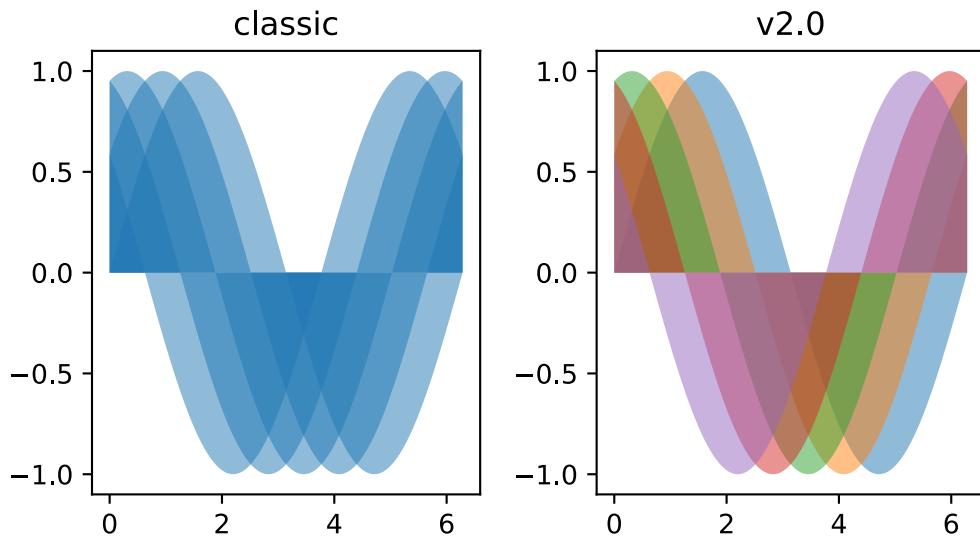
```
boxplot.flierprops.color:          'k'
boxplot.flierprops.marker:        '+'
boxplot.flierprops.markerfacecolor: 'none'
boxplot.flierprops.markeredgecolor: 'k'
boxplot.boxprops.color:           'b'
boxplot.whiskerprops.color:       'b'
```

```
boxplot.whiskerprops.linestyle:      '--'
boxplot.medianprops.color:          'r'
boxplot.meanprops.color:            'r'
boxplot.meanprops.marker:           '^'
boxplot.meanprops.markerfacecolor:  'r'
boxplot.meanprops.markeredgecolor:  'k'
boxplot.meanprops.markersize:       6
boxplot.meanprops.linestyle:        '--'
boxplot.meanprops.linewidth:        1.0
```

in your `matplotlibrc` file.

`fill_between` and `fill_betweenx`

`fill_between` and `fill_betweenx` both follow the patch color cycle.

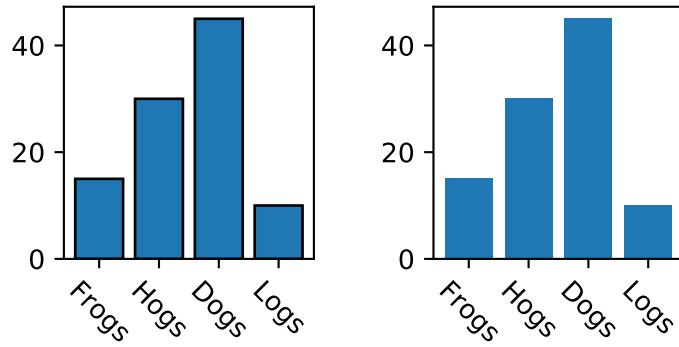
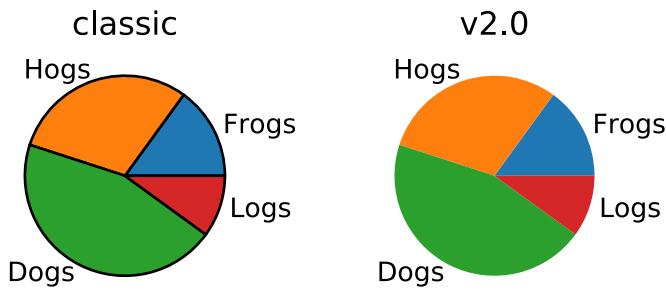


If the `facecolor` is set via the `facecolors` or `color` keyword argument, then the color is not cycled.

To restore the previous behavior, explicitly pass the keyword argument `facecolors='C0'` to the method call.

Patch edges and color

Most artists drawn with a patch (`~matplotlib.axes.Axes.bar`, `~matplotlib.axes.Axes.pie`, etc) no longer have a black edge by default. The default face color is now '`C0`' instead of '`b`'.



The previous defaults can be restored by setting:

```
mpl.rcParams['patch.force_edgecolor'] = True
mpl.rcParams['patch.facecolor'] = 'b'
```

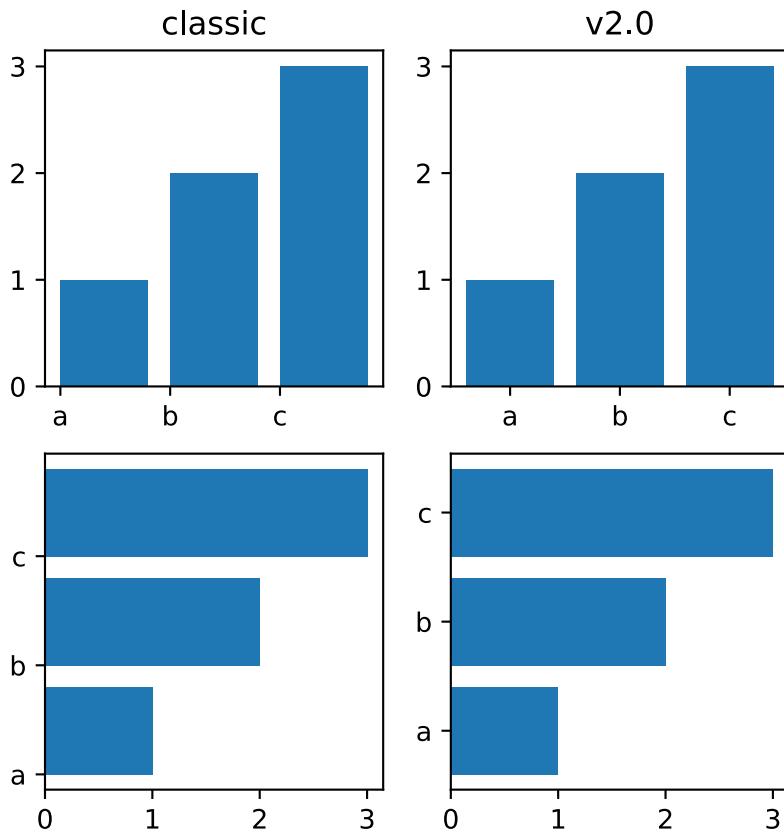
or by setting:

```
patch.facecolor      : b
patch.force_edgecolor : True
```

in your `matplotlibrc` file.

bar and barh

The default value of the `align` kwarg for both `bar` and `barh` is changed from '`edge`' to '`center`'.



To restore the previous behavior explicitly pass the keyword argument `align='edge'` to the method call.

Hatching

The color and width of the lines in a hatch pattern are now configurable by the rcParams `hatch.color` and `hatch.linewidth`, with defaults of black and 1 point, respectively. The old behaviour for the color was to apply the edge color or use black, depending on the artist; the old behavior for the line width was different depending on backend:

- PDF: 0.1 pt
- SVG: 1.0 pt
- PS: 1 px
- Agg: 1 px

The old color behavior can not be restored. The old line width behavior can not be restored across all backends simultaneously, but can be restored for a single backend by setting:

```
mpl.rcParams['hatch.linewidth'] = 0.1 # previous pdf hatch linewidth
mpl.rcParams['hatch.linewidth'] = 1.0 # previous svg hatch linewidth
```

The behavior of the PS and Agg backends was DPI dependent, thus:

```
mpl.rcParams['figure.dpi'] = dpi
mpl.rcParams['savefig.dpi'] = dpi # or leave as default 'figure'
mpl.rcParams['hatch.linewidth'] = 1.0 / dpi # previous ps and Agg hatch linewidth
```

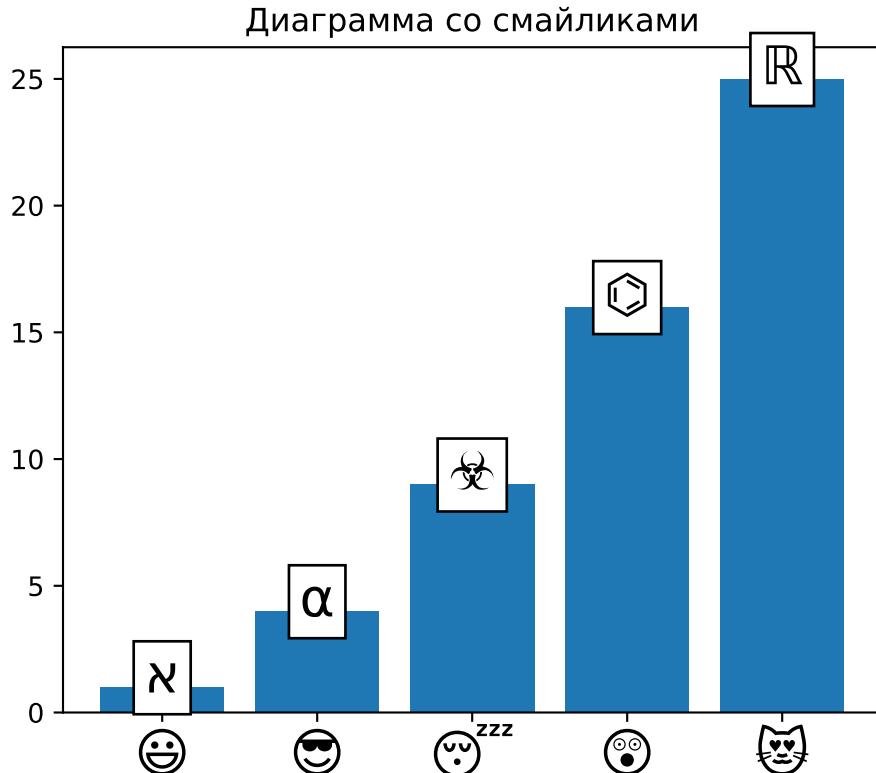
There is no API level control of the hatch color or linewidth.

Hatching patterns are now rendered at a consistent density, regardless of DPI. Formerly, high DPI figures would be more dense than the default, and low DPI figures would be less dense. This old behavior cannot be directly restored, but the density may be increased by repeating the hatch specifier.

Fonts

Normal text

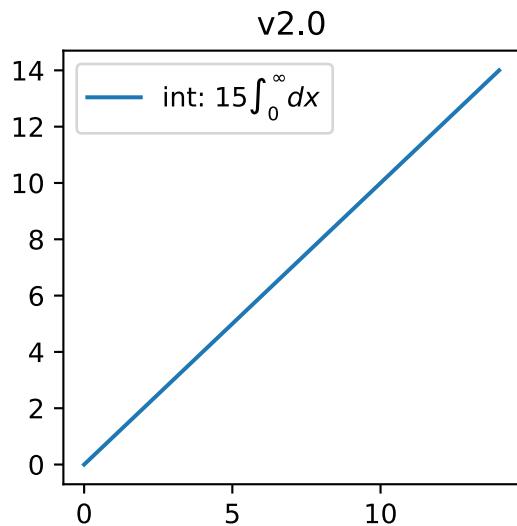
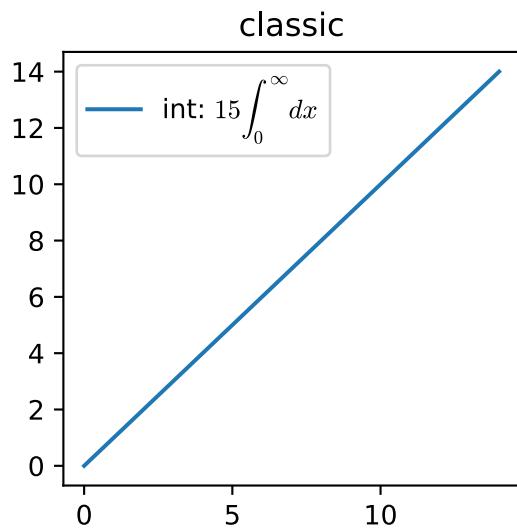
The default font has changed from “Bitstream Vera Sans” to “DejaVu Sans”. DejaVu Sans has additional international and math characters, but otherwise has the same appearance as Bitstream Vera Sans. Latin, Greek, Cyrillic, Armenian, Georgian, Hebrew, and Arabic are [all supported](#) (but right-to-left rendering is still not handled by matplotlib). In addition, DejaVu contains a sub-set of emoji symbols.



See the [DejaVu Sans PDF sample](#) for full coverage.

Math text

The default math font when using the built-in math rendering engine (mathtext) has changed from “Computer Modern” (i.e. LaTeX-like) to “DejaVu Sans”. This change has no effect if the TeX backend is used (i.e. `text.usetex` is True).



To revert to the old behavior set the:

```
mpl.rcParams['mathtext.fontset'] = 'cm'
mpl.rcParams['mathtext.rm'] = 'serif'
```

or set:

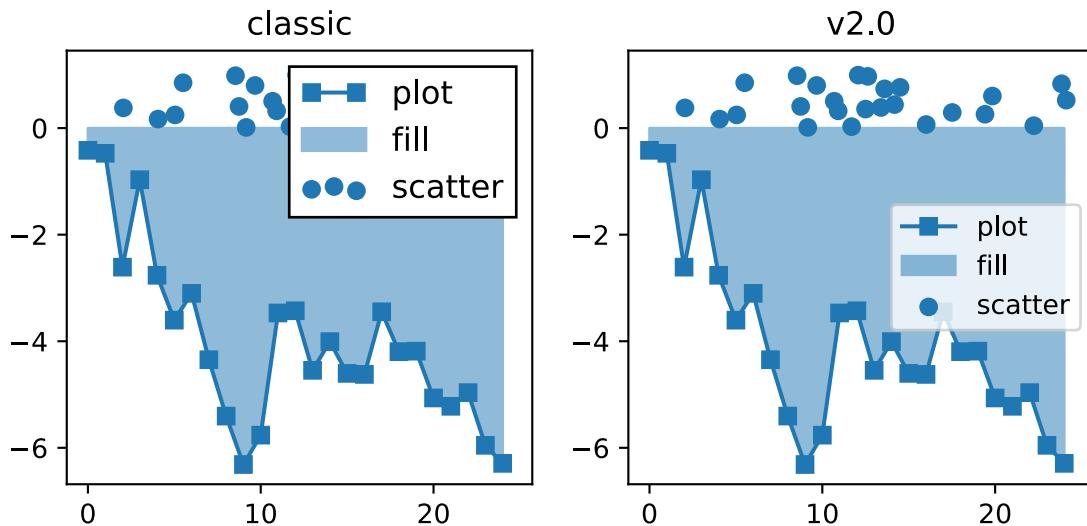
```
mathtext.fontset: cm
mathtext.rm : serif
```

in your `matplotlibrc` file.

This `rcParam` is consulted when the text is drawn, not when the artist is created. Thus all mathtext on a given canvas will use the same fontset.

Legends

- By default, the number of points displayed in a legend is now 1.
- The default legend location is '`'best'`', so the legend will be automatically placed in a location to minimize overlap with data.
- The legend defaults now include rounded corners, a lighter boundary, and partially transparent boundary and background.



The previous defaults can be restored by setting:

```
mpl.rcParams['legend.fancybox'] = False
mpl.rcParams['legend.loc'] = 'upper right'
mpl.rcParams['legend.numpoints'] = 2
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['legend.framealpha'] = None
mpl.rcParams['legend.scatterpoints'] = 3
mpl.rcParams['legend.edgecolor'] = 'inherit'
```

or by setting:

<code>legend.fancybox</code>	<code>: False</code>
<code>legend.loc</code>	<code>: upper right</code>
<code>legend.numpoints</code>	<code>: 2 # the number of points in the legend line</code>
<code>legend.fontsize</code>	<code>: large</code>
<code>legend.framealpha</code>	<code>: None # opacity of legend frame</code>
<code>legend.scatterpoints</code>	<code>: 3 # number of scatter points</code>

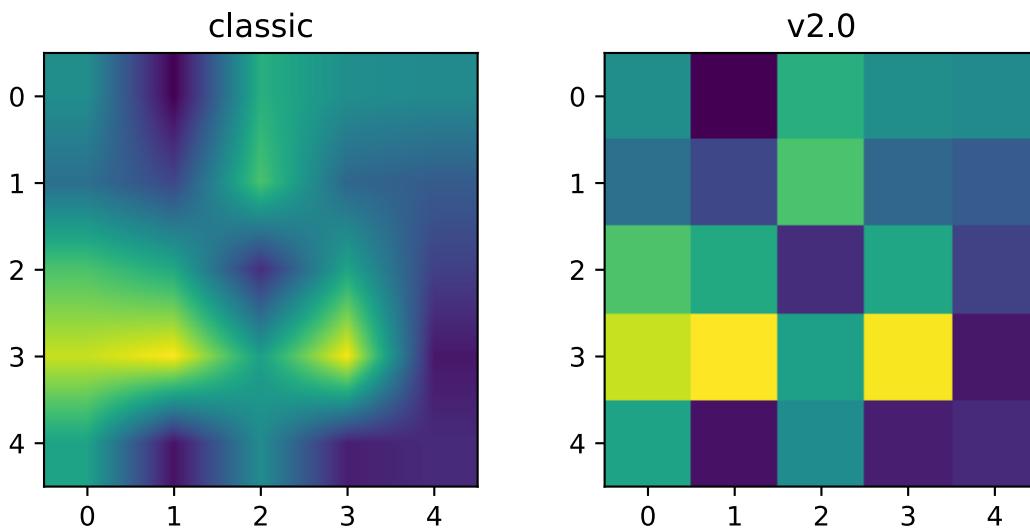
```
legend.edgecolor      : inherit  # legend edge color ('inherit'
                                # means it uses axes.edgecolor)
```

in your `matplotlibrc` file.

Image

Interpolation

The default interpolation method for `imshow` is now 'nearest' and by default it resamples the data (both up and down sampling) before color mapping.



To restore the previous behavior set:

```
mpl.rcParams['image.interpolation'] = 'bilinear'
mpl.rcParams['image.resample'] = False
```

or set:

```
image.interpolation : bilinear  # see help(imshow) for options
image.resample   : False
```

in your `matplotlibrc` file.

Colormapping pipeline

Previously, the input data was normalized, then color mapped, and then resampled to the resolution required for the screen. This meant that the final resampling was being done in color space. Because the color maps are not generally linear in RGB space, colors not in the color map may appear in the final image. This bug was addressed by an almost complete overhaul of the image handling code.

The input data is now normalized, then resampled to the correct resolution (in normalized dataspace), and then color mapped to RGB space. This ensures that only colors from the color map appear in the final image. (If your viewer subsequently resamples the image, the artifact may reappear.)

The previous behavior cannot be restored.

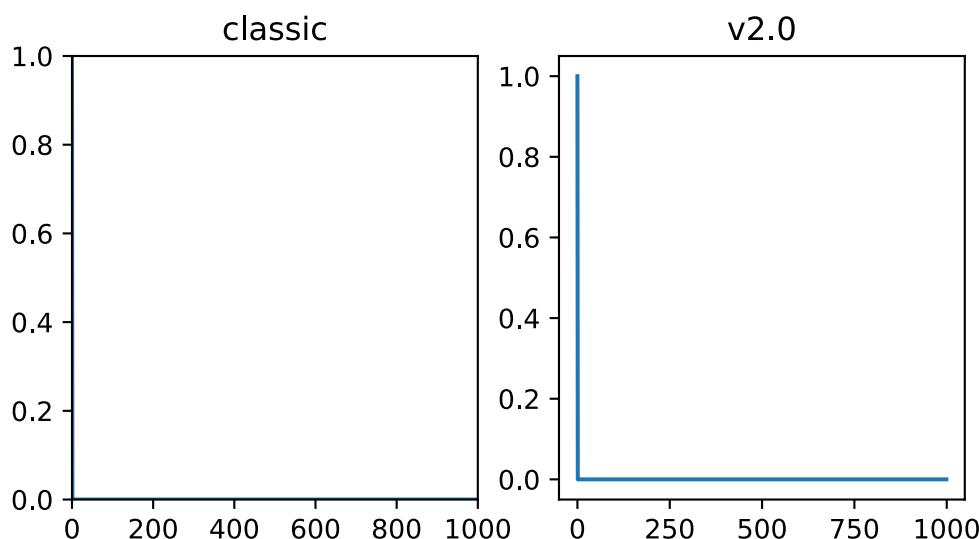
Shading

- The default shading mode for light source shading, in `matplotlib.colors.LightSource.shade`, is now `overlay`. Formerly, it was `hsv`.

Plot layout

Auto limits

The previous auto-scaling behavior was to find ‘nice’ round numbers as view limits that enclosed the data limits, but this could produce bad plots if the data happened to fall on a vertical or horizontal line near the chosen ‘round number’ limit. The new default sets the view limits to 5% wider than the data range.



The size of the padding in the x and y directions is controlled by the `'axes.xmargin'` and `'axes.ymargin'` rcParams respectively. Whether the view limits should be ‘round numbers’ is controlled by the `'axes.autolimit_mode'` rcParam. In the original `'round_number'` mode, the view limits coincide with ticks.

The previous default can be restored by using:

```
mpl.rcParams['axes.autolimit_mode'] = 'round_numbers'  
mpl.rcParams['axes.xmargin'] = 0  
mpl.rcParams['axes.ymargin'] = 0
```

or setting:

```
axes.autolimit_mode: round_numbers
axes.xmargin: 0
axes.ymargin: 0
```

in your `matplotlibrc` file.

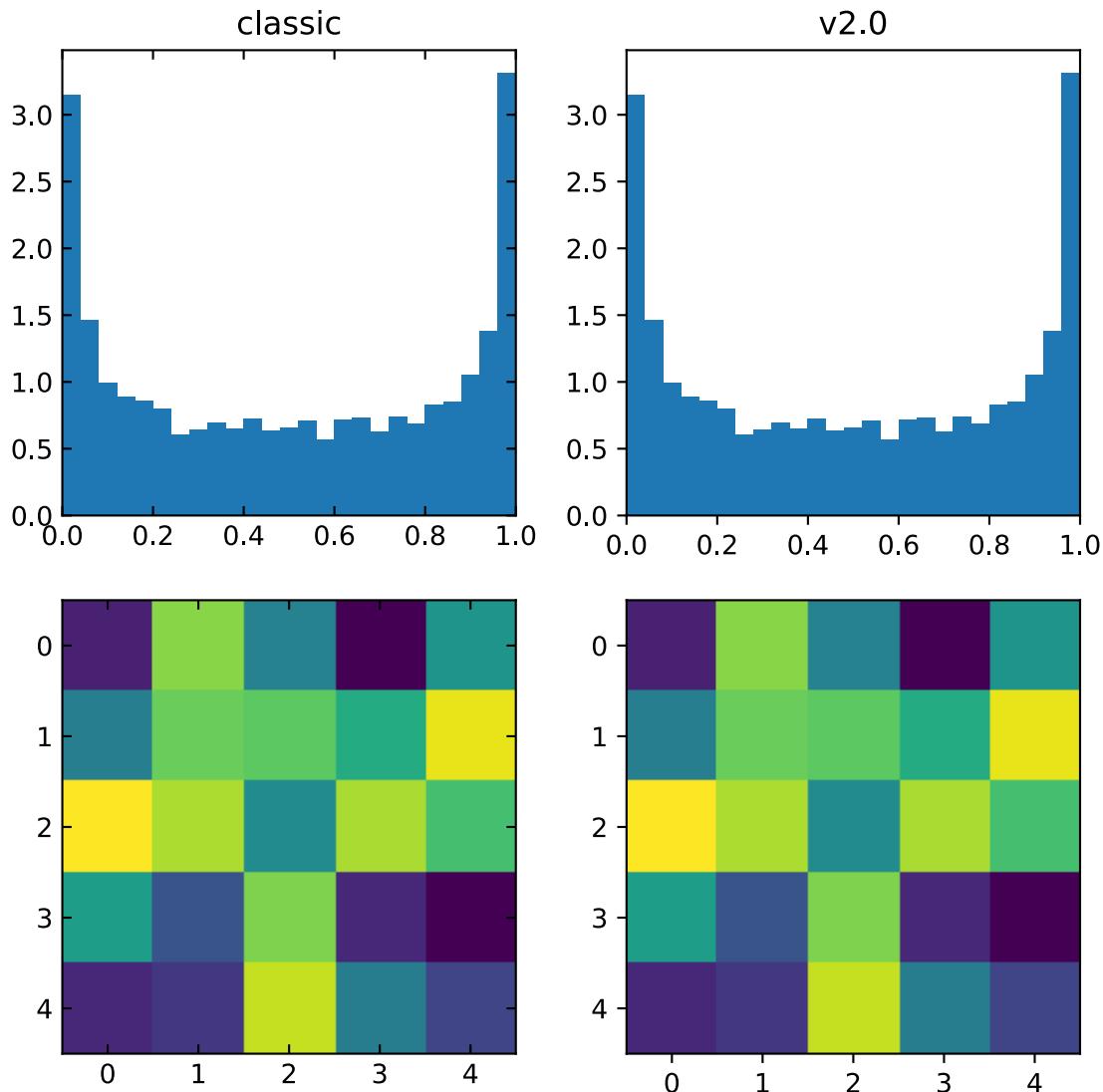
Z-order

- Ticks and grids are now plotted above solid elements such as filled contours, but below lines. To return to the previous behavior of plotting ticks and grids above lines, set `rcParams['axes.axisbelow'] = False`.

Ticks

Direction

To reduce the collision of tick marks with data, the default ticks now point outward by default. In addition, ticks are now drawn only on the bottom and left spines to prevent a porcupine appearance, and for a cleaner separation between subplots.



To restore the previous behavior set:

```
mpl.rcParams['xtick.direction'] = 'in'
mpl.rcParams['ytick.direction'] = 'in'
mpl.rcParams['xtick.top'] = True
mpl.rcParams['ytick.right'] = True
```

or set:

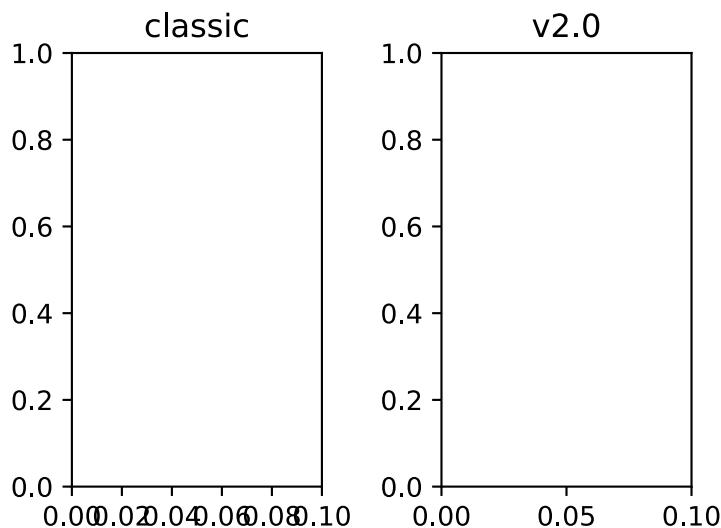
```
xtick.top: True
xtick.direction: in

ytick.right: True
ytick.direction: in
```

in your `matplotlibrc` file.

Number of ticks

The default *Locator* used for the x and y axis is *AutoLocator* which tries to find, up to some maximum number, ‘nicely’ spaced ticks. The locator now includes an algorithm to estimate the maximum number of ticks that will leave room for the tick labels. By default it also ensures that there are at least two ticks visible.



There is no way, other than using `mpl.style.use('classic')`, to restore the previous behavior as the default. On an axis-by-axis basis you may either control the existing locator via:

```
ax.xaxis.get_major_locator().set_params(nbins=9, steps=[1, 2, 5, 10])
```

or create a new *MaxNLocator*:

```
import matplotlib.ticker as mticker
ax.set_major_locator(mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10]))
```

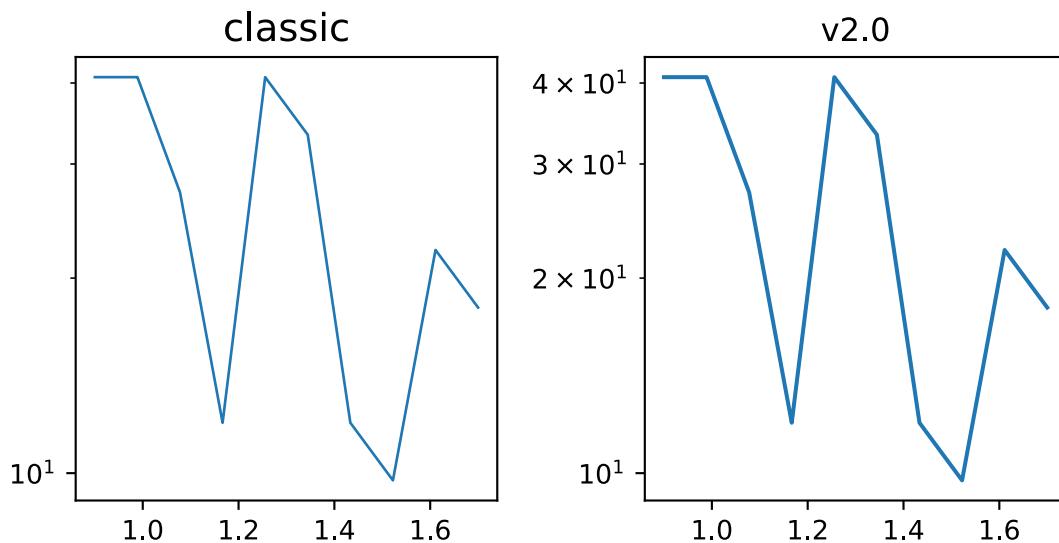
The algorithm used by *MaxNLocator* has been improved, and this may change the choice of tick locations in some cases. This also affects *AutoLocator*, which uses *MaxNLocator* internally.

For a log-scaled axis the default locator is the *LogLocator*. Previously the maximum number of ticks was set to 15, and could not be changed. Now there is a `numticks` kwarg for setting the maximum to any integer value, to the string ‘auto’, or to its default value of `None` which is equivalent to ‘auto’. With the ‘auto’ setting the maximum number will be no larger than 9, and will be reduced depending on the length of the axis in units of the tick font size. As in the case of the AutoLocator, the heuristic algorithm reduces the incidence of overlapping tick labels but does not prevent it.

Tick label formatting

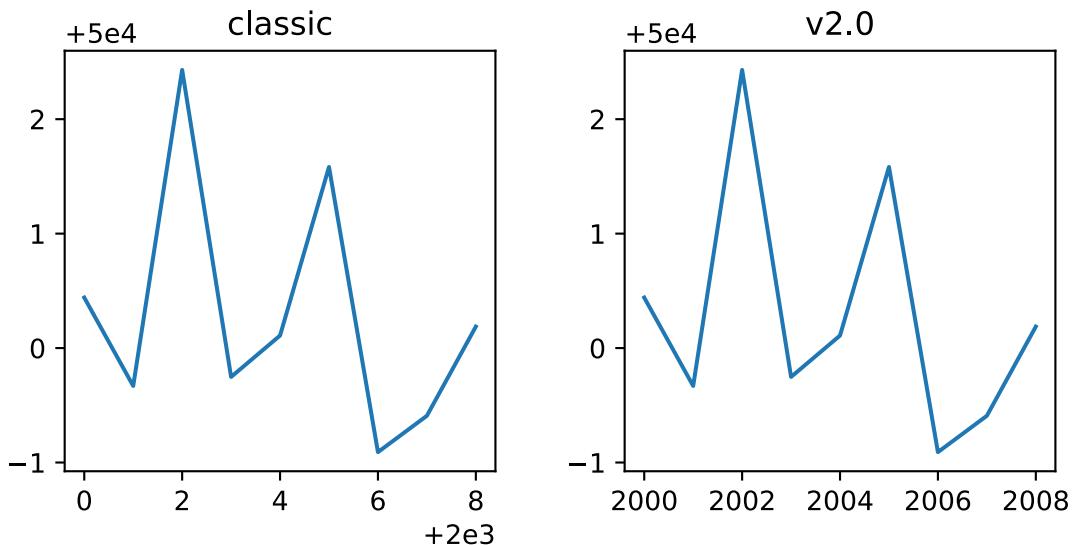
LogFormatter labeling of minor ticks

Minor ticks on a log axis are now labeled when the axis view limits span a range less than or equal to the interval between two major ticks. See [LogFormatter](#) for details. The minor tick labeling is turned off when using `mpl.style.use('classic')`, but cannot be controlled independently via `rcParams`.



ScalarFormatter tick label formatting with offsets

With the default of `rcParams['axes.formatter.useoffset'] = True`, an offset will be used when it will save 4 or more digits. This can be controlled with the new `rcParam, axes.formatter.offset_threshold`. To restore the previous behavior of using an offset to save 2 or more digits, use `rcParams['axes.formatter.offset_threshold'] = 2`.



AutoDateFormatter format strings

The default date formats are now all based on ISO format, i.e., with the slowest-moving value first. The date formatters are configurable through the `date.autoformatter.*` rcParams.

Threshold (tick interval >= than)	rcParam	classic	v2.0
365 days	'date.autoformatter.year'	'%Y'	'%Y'
30 days	'date.autoformatter.month'	'%b %Y'	'%Y-%m'
1 day	'date.autoformatter.day'	'%b %d %Y'	'%Y-%m-%d'
1 hour	'date.autoformatter.hour'	'%H:%M:%S'	'%H:%M'
1 minute	'date.autoformatter.minute'	'%H:%M:%S.%f'	'%H:%M:%S'
1 second	'date.autoformatter.second'	'%H:%M:%S.%f'	'%H:%M:%S'
1 microsecond	'date.autoformatter.microsecond'	'%H:%M:%S.%f'	'%H:%M:%S.%f'

Python's `%x` and `%X` date formats may be of particular interest to format dates based on the current locale.

The previous default can be restored by:

```
mpl.rcParams['date.autoformatter.year'] = '%Y'
mpl.rcParams['date.autoformatter.month'] = '%b %Y'
mpl.rcParams['date.autoformatter.day'] = '%b %d %Y'
mpl.rcParams['date.autoformatter.hour'] = '%H:%M:%S'
mpl.rcParams['date.autoformatter.minute'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.second'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.microsecond'] = '%H:%M:%S.%f'
```

or setting

```
date.autoformatter.year    : %Y
date.autoformatter.month   : %b %Y
date.autoformatter.day     : %b %d %Y
date.autoformatter.hour    : %H:%M:%S
date.autoformatter.minute  : %H:%M:%S.%f
date.autoformatter.second  : %H:%M:%S.%f
date.autoformatter.microsecond : %H:%M:%S.%f
```

in your `matplotlibrc` file.

mplot3d

- mplot3d now obeys some style-related rcParams, rather than using hard-coded defaults. These include:
 - `xtick.major.width`
 - `ytick.major.width`

- xtick.color
- ytick.color
- axes.linewidth
- axes.edgecolor
- grid.color
- grid.linewidth
- grid.linestyle

9.1.2 Improved color conversion API and RGBA support

The `colors` gained a new color conversion API with full support for the alpha channel. The main public functions are `is_color_like()`, `matplotlib.colors.to_rgba()`, `matplotlib.colors.to_rgba_array()` and `to_hex()`. RGBA quadruplets are encoded in hex format as #rrggbbaa.

A side benefit is that the Qt options editor now allows setting the alpha channel of the artists as well.

9.1.3 New Configuration (rcParams)

New rcparams added

Parameter	Description
<code>date.autoformatter.year</code>	format string for ‘year’ scale dates
<code>date.autoformatter.month</code>	format string for ‘month’ scale dates
<code>date.autoformatter.day</code>	format string for ‘day’ scale dates
<code>date.autoformatter.hour</code>	format string for ‘hour’ scale times
<code>date.autoformatter.minute</code>	format string for ‘minute’ scale times
<code>date.autoformatter.second</code>	format string for ‘second’ scale times
<code>date.autoformatter.microsecond</code>	format string for ‘microsecond’ scale times
<code>scatter.marker</code>	default marker for scatter plot
<code>svg.hashsalt</code>	see note
<code>xtick.top, xtick.minor.top, xtick.major.top</code> <code>xtick.bottom, xtick.minor.bottom,</code> <code>xtick.major.bottom</code> <code>y tick.left, y tick.minor.left,</code> <code>y tick.major.left</code> <code>y tick.right, y tick.minor.right,</code> <code>y tick.major.right</code>	Control where major and minor ticks are drawn. The global values are added with the corresponding major/minor values.
<code>hist.bins</code>	The default number of bins to use in <code>hist</code> . This can be an <code>int</code> , a list of floats, or ‘auto’ if numpy >= 1.11 is installed.
<code>lines.scale_dashes</code>	Whether the line dash patterns should scale with linewidth.
<code>axes.formatter.offset_threshold</code>	Minimum number of digits saved in tick labels that triggers using an offset.

Added `svg.hashsalt` key to rcParams

If `svg.hashsalt` is `None` (which it is by default), the `svg` backend uses `uuid4` to generate the hash salt. If it is not `None`, it must be a string that is used as the hash salt instead of `uuid4`. This allows for deterministic SVG output.

Removed the `svg.image_noscale` rcParam

As a result of the extensive changes to image handling, the `svg.image_noscale` rcParam has been removed. The same functionality may be achieved by setting `interpolation='none'` on individual images or globally using the `image.interpolation` rcParam.

9.1.4 Qualitative colormaps

ColorBrewer’s “qualitative” colormaps (“Accent”, “Dark2”, “Paired”, “Pastel1”, “Pastel2”, “Set1”, “Set2”, “Set3”) were intended for discrete categorical data, with no implication of value, and therefore have been

converted to `ListedColormap` instead of `LinearSegmentedColormap`, so the colors will no longer be interpolated and they can be used for choropleths, labeled image features, etc.

9.1.5 Axis offset label now responds to `labelcolor`

Axis offset labels are now colored the same as axis tick markers when `labelcolor` is altered.

9.1.6 Improved offset text choice

The default offset-text choice was changed to only use significant digits that are common to all ticks (e.g. 1231..1239 -> 1230, instead of 1231), except when they straddle a relatively large multiple of a power of ten, in which case that multiple is chosen (e.g. 1999..2001->2000).

9.1.7 Style parameter blacklist

In order to prevent unexpected consequences from using a style, style files are no longer able to set parameters that affect things unrelated to style. These parameters include:

```
'interactive', 'backend', 'backend.qt4', 'webagg.port',
'webagg.port_retries', 'webagg.open_in_browser', 'backend_fallback',
'toolbar', 'timezone', 'datapath', 'figure.max_open_warning',
'savefig.directory', 'tk.window_focus', 'docstring.hardcopy'
```

9.1.8 Change in default font

The default font used by matplotlib in text has been changed to DejaVu Sans and DejaVu Serif for the sans-serif and serif families, respectively. The DejaVu font family is based on the previous matplotlib default –Bitstream Vera– but includes a much wider range of characters.

The default mathtext font has been changed from Computer Modern to the DejaVu family to maintain consistency with regular text. Two new options for the `mathtext.fontset` configuration parameter have been added: `dejavusans` (default) and `dejavuserif`. Both of these options use DejaVu glyphs whenever possible and fall back to STIX symbols when a glyph is not found in DejaVu. To return to the previous behavior, set the rcParam `mathtext.fontset` to `cm`.

9.1.9 Faster text rendering

Rendering text in the Agg backend is now less fuzzy and about 20% faster to draw.

9.1.10 Improvements for the Qt figure options editor

Various usability improvements were implemented for the Qt figure options editor, among which:

- Line style entries are now sorted without duplicates.

- The colormap and normalization limits can now be set for images.
- Line edits for floating values now display only as many digits as necessary to avoid precision loss. An important bug was also fixed regarding input validation using Qt5 and a locale where the decimal separator is “,”.
- The axes selector now uses shorter, more user-friendly names for axes, and does not crash if there are no axes.
- Line and image entries using the default labels (“_lineX”, “_imageX”) are now sorted numerically even when there are more than 10 entries.

9.1.11 Improved image support

Prior to version 2.0, matplotlib resampled images by first applying the color map and then resizing the result. Since the resampling was performed on the colored image, this introduced colors in the output image that didn’t actually exist in the color map. Now, images are resampled first (and entirely in floating-point, if the input image is floating-point), and then the color map is applied.

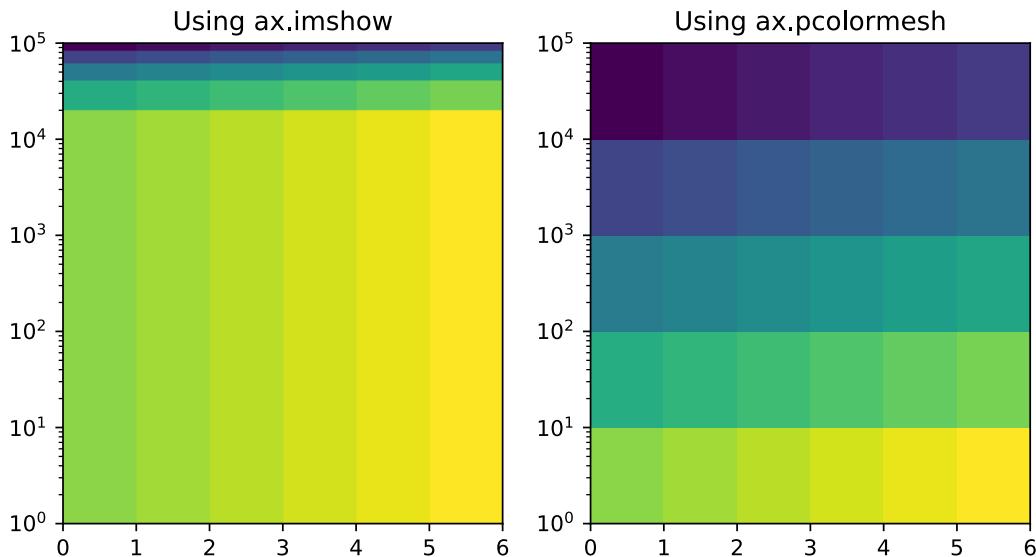
In order to make this important change, the image handling code was almost entirely rewritten. As a side effect, image resampling uses less memory and fewer datatype conversions than before.

The experimental private feature where one could “skew” an image by setting the private member `_image_skew_coordinate` has been removed. Instead, images will obey the transform of the axes on which they are drawn.

Non-linear scales on image plots

`imshow()` now draws data at the requested points in data space after the application of non-linear scales.

The image on the left demonstrates the new, correct behavior. The old behavior can be recreated using `pcolormesh()` as demonstrated on the right.



This can be understood by analogy to plotting a histogram with linearly spaced bins with a logarithmic x-axis. Equal sized bins will be displayed as wider for small x and narrower for large x .

9.1.12 Support for HiDPI (Retina) displays in the NbAgg and WebAgg backends

The NbAgg and WebAgg backends will now use the full resolution of your high-pixel-density display.

9.1.13 Change in the default animation codec

The default animation codec has been changed from `mpeg4` to `h264`, which is more efficient. It can be set via the `animation.codec` rcParam.

9.1.14 Deprecated support for mencoder in animation

The use of mencoder for writing video files with mpl is problematic; switching to ffmpeg is strongly advised. All support for mencoder will be removed in version 2.2.

9.1.15 Boxplot Zorder Keyword Argument

The `zorder` parameter now exists for `boxplot()`. This allows the zorder of a boxplot to be set in the plotting function call.

```
boxplot(np.arange(10), zorder=10)
```

9.1.16 Filled + and x markers

New fillable *plus* and *x* markers have been added. See the [markers](#) module and [marker reference](#) examples.

9.1.17 rcount and ccount for plot_surface()

As of v2.0, mplot3d's `plot_surface()` now accepts `rcount` and `ccount` arguments for controlling the sampling of the input data for plotting. These arguments specify the maximum number of evenly spaced samples to take from the input data. These arguments are also the new default sampling method for the function, and is considered a style change.

The old `rstride` and `cstride` arguments, which specified the size of the evenly spaced samples, become the default when ‘classic’ mode is invoked, and are still available for use. There are no plans for deprecating these arguments.

9.1.18 Streamplot Zorder Keyword Argument Changes

The `zorder` parameter for `streamplot()` now has default value of `None` instead of `2`. If `None` is given as `zorder`, `streamplot()` has a default `zorder` of `matplotlib.lines.Line2D.zorder`.

9.2 Previous Whats New

9.2.1 New in matplotlib 0.98.4

Table of Contents

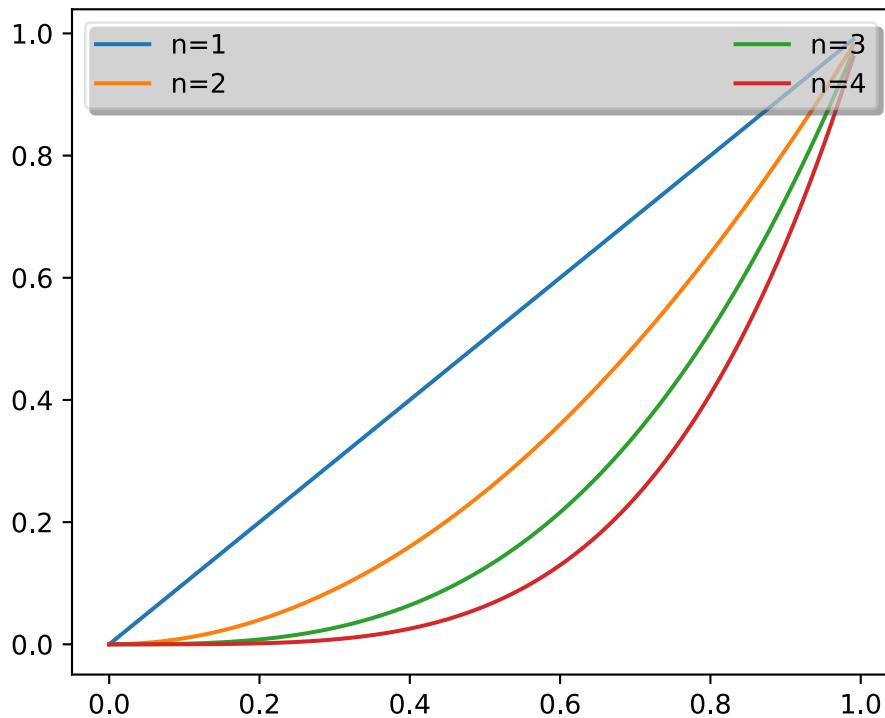
- *New in matplotlib 0.98.4*
 - *Legend enhancements*
 - *Fancy annotations and arrows*
 - *Native OS X backend*
 - *psd amplitude scaling*
 - *Fill between*
 - *Lots more*

It's been four months since the last matplotlib release, and there are a lot of new features and bug-fixes.

Thanks to Charlie Moad for testing and preparing the source release, including binaries for OS X and Windows for python 2.4 and 2.5 (2.6 and 3.0 will not be available until numpy is available on those releases). Thanks to the many developers who contributed to this release, with contributions from Jae-Joon Lee, Michael Droettboom, Ryan May, Eric Firing, Manuel Metz, Jouni K. Seppänen, Jeff Whitaker, Darren Dale, David Kaplan, Michiel de Hoon and many others who submitted patches

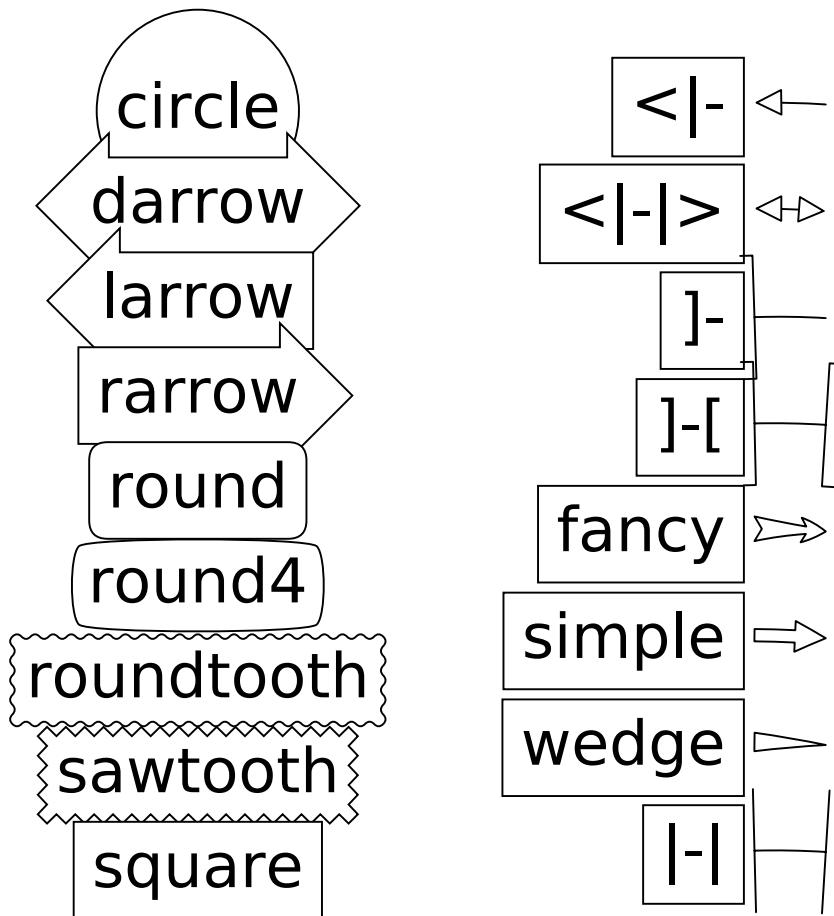
Legend enhancements

Jae-Joon has rewritten the legend class, and added support for multiple columns and rows, as well as fancy box drawing. See [legend\(\)](#) and [matplotlib.legend.Legend](#).



Fancy annotations and arrows

Jae-Joon has added lots of support to annotations for drawing fancy boxes and connectors in annotations. See [annotate\(\)](#) and [BoxStyle](#), [ArrowStyle](#), and [ConnectionStyle](#).



Native OS X backend

Michiel de Hoon has provided a native Mac OSX backend that is almost completely implemented in C. The backend can therefore use Quartz directly and, depending on the application, can be orders of magnitude faster than the existing backends. In addition, no third-party libraries are needed other than Python and NumPy. The backend is interactive from the usual terminal application on Mac using regular Python. It hasn't been tested with ipython yet, but in principle it should work there as well. Set 'backend : macosx' in your matplotlibrc file, or run your script with:

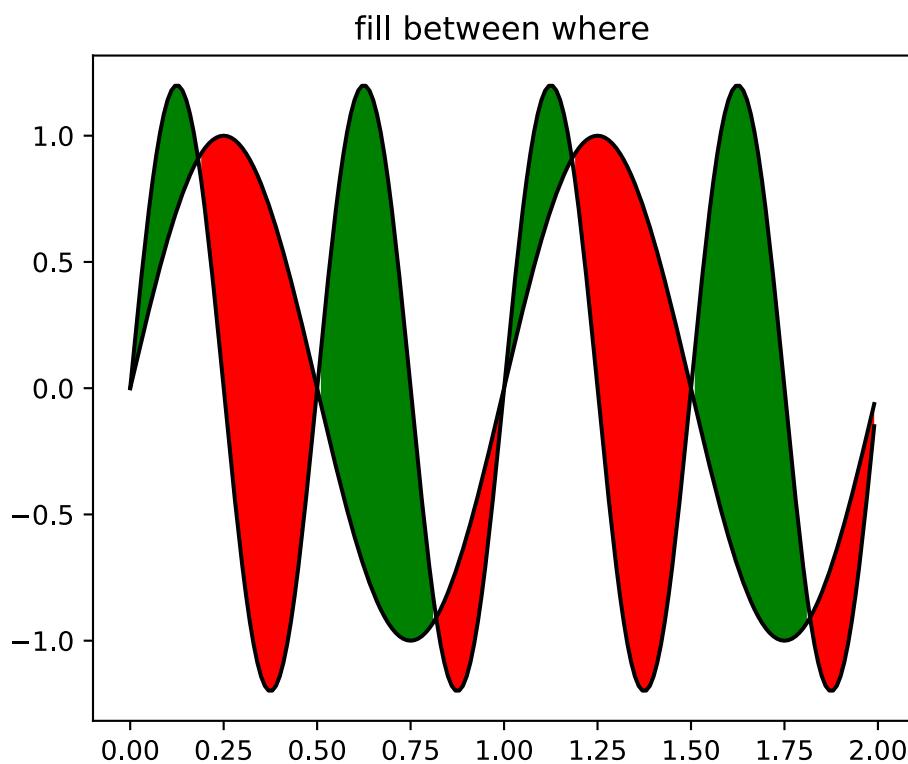
```
> python myfile.py -dmacosx
```

psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of `psd()` and friends. See [pylab_examples example code: psd_demo2.py](#). and [pylab_examples example code: psd_demo3.py](#). The changes should increase MATLAB compatibility and increase scaling options.

Fill between

Added a `fill_between()` function to make it easier to do shaded region plots in the presence of masked data. You can pass an `x` array and a `ylower` and `yupper` array to fill between, and an optional `where` argument which is a logical mask where you want to do the filling.



Lots more

Here are the 0.98.4 notes from the CHANGELOG:

```
Added mdehoon's native macosx backend from sf patch 2179017 - JDH
```

```
Removed the prints in the set_*style commands. Return the list of
pretty-printed strings instead - JDH
```

```
Some of the changes Michael made to improve the output of the
```

`property` tables `in` the rest docs broke of made difficult to use some of the interactive doc helpers, e.g., `setp` **and** `getp`. Having `all` the rest markup `in` the ipython shell also confused the docstrings. I added a new rc param `docstring.harcopy`, to `format` the docstrings differently `for` hardcopy **and** other use. The `ArtistInspector` could use a little refactoring now since there **is** duplication of effort between the rest out put **and** the non-rest output - JDH

Updated spectral methods (`psd`, `csd`, etc.) to scale one-sided densities by a factor of `2` **and**, optionally, scale `all` densities by the sampling frequency. This gives better MATLAB compatibility. -RM

Fixed alignment of ticks `in` colorbars. -MGD

drop the deprecated "new" keyword of `np.histogram()` **for** numpy `1.2` **or** later. -JJL

Fixed a bug `in` `svg` backend that `new_figure_manager()` ignores keywords arguments such `as` `figsize`, etc. -JJL

Fixed a bug that the handlelength of the new legend `class set` too short when `numpoints=1` -JJL

Added support `for` data `with` units (e.g., dates) to `Axes.fill_between`. -RM

Added fancybox keyword to legend. Also applied some changes `for` better look, including baseline adjustment of the multiline texts so that it **is** center aligned. -JJL

The transmuter classes `in` the `patches.py` are reorganized `as` subclasses of the Style classes. A few more box **and** arrow styles are added. -JJL

Fixed a bug `in` the new legend `class that` didn't allowed a tuple of coordinate values `as` `loc`. -JJL

Improve checks `for` external dependencies, using subprocess (instead of deprecated `popen*`) **and** distutils (`for` version checking) - DSD

Reimplementation of the legend which supports baseline alignment, multi-column, **and** expand mode. - JJL

Fixed histogram autoscaling bug when bins `or` range are given explicitly (fixes Debian bug [503148](#)) - MM

Added `rcParam axes.unicode_minus` which allows plain hyphen `for` minus when `False` - JDH

Added scatterpoints support `in` Legend. patch by Erik Tollerud - JJL

Fix crash `in` log ticking. - MGD

Added static helper method `BrokenHBarCollection.span_where` and `Axes/pyplot` method `fill_between`. See `examples/pylab/fill_between.py` - JDH

Add `x_isdata` and `y_isdata` attributes to `Artist` instances, and use them to determine whether either or both coordinates are used when updating `dataLim`. This is used to fix autoscaling problems that had been triggered by `axhline`, `axhspan`, `axvline`, `axvspan`. - EF

Update the `psd()`, `csd()`, `cohere()`, and `specgram()` methods of `Axes` and the `csd()`, `cohere()`, and `specgram()` functions in `mlab` to be in sync with the changes to `psd()`. In fact, under the hood, these all call the same core to do computations. - RM

Add '`pad_to`' and '`sides`' parameters to `mlab.psd()` to allow controlling of zero padding and returning of negative frequency components, respectively. These are added in a way that does not change the API. - RM

Fix handling of c kwarg by `scatter`; generalize `is_string_like` to accept `numpy` and `numpy.ma` string array scalars. - RM and EF

Fix a possible EINTR problem in `dviread`, which might help when saving pdf files from the qt backend. - JKS

Fix bug with zoom to rectangle and twin axes - MGD

Added Jae Joon's fancy arrow, box and annotation enhancements -- see `examples/pylab_examples/annotation_demo2.py`

Autoscaling is now supported with shared axes - EF

Fixed exception in `dviread` that happened with Minion - JKS

`set_xlim`, `ylim` now return a copy of the `viewlim` array to avoid modify inplace surprises

Added image thumbnail generating function `matplotlib.image.thumbnail`. See `examples/misc/image_thumbnail.py` - JDH

Applied scatleg patch based on ideas and work by Erik Tollerud and Jae-Joon Lee. - MM

Fixed bug in pdf backend: if you pass a file object for output instead of a filename, e.g., in a wep app, we now flush the object at the end. - JKS

Add path simplification support to paths with gaps. - EF

Fix problem **with** AFM files that don't specify the font's full name **or** family name. - JKS

Added 'scilimits' kwarg to Axes.ticklabel_format() method, **for** easy access to the set_powerlimits method of the major ScalarFormatter. - EF

Experimental new kwarg borderpad to replace pad **in** legend, based on suggestion by Jae-Joon Lee. - EF

Allow spy to ignore zero values **in** sparse arrays, based on patch by Tony Yu. Also fixed plot to handle empty data arrays, **and** fixed handling of markers **in** figlegend. - EF

Introduce drawstyles **for** lines. Transparently split linestyles like 'steps--' into drawstyle 'steps' **and** linestyle '--'. Legends always use drawstyle 'default'. - MM

Fixed quiver **and** quiverkey bugs (failure to scale properly when resizing) **and** added additional methods **for** determining the arrow angles - EF

Fix polar interpolation to handle negative values of theta - MGD

Reorganized cbook **and** mlab methods related to numerical calculations that have little to do **with** the goals of those two modules into a separate module numerical_methods.py Also, added ability to select points **and** stop point selection **with** keyboard **in** ginput **and** manual contour labeling code. Finally, fixed contour labeling bug. - DMK

Fix backtick **in** Postscript output. - MGD

[2089958] Path simplification **for** vector output backends Leverage the simplification code exposed through path_to_polygons to simplify certain well-behaved paths **in** the vector backends (PDF, PS **and** SVG). "path.simplify" must be set to True **in** matplotlibrc **for** this to work. - MGD

Add "filled" kwarg to Path.intersects_path **and** Path.intersects_bbox. - MGD

Changed full arrows slightly to avoid an xpdf rendering problem reported by Friedrich Hagedorn. - JKS

Fix conversion of quadratic to cubic Bezier curves **in** PDF **and** PS backends. Patch by Jae-Joon Lee. - JKS

Added 5-point star marker to plot command q- EF

Fix hatching **in** PS backend - MGD

Fix log **with** base 2 - MGD

```
Added support for bilinear interpolation in
NonUniformImage; patch by Gregory Lielens. - EF

Added support for multiple histograms with data of
different length - MM

Fix step plots with log scale - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

fix quiver so masked values are not plotted - JSW

improve interactive pan/zoom in qt4 backend on windows - DSD

Fix more bugs in NaN/inf handling. In particular, path
simplification (which does not handle NaNs or infs) will be turned
off automatically when infs or NaNs are present. Also masked
arrays are now converted to arrays with NaNs for consistent
handling of masks and NaNs - MGD and EF

Added support for arbitrary rasterization resolutions to the SVG
backend. - MW
```

9.2.2 New in matplotlib 0.99

Table of Contents

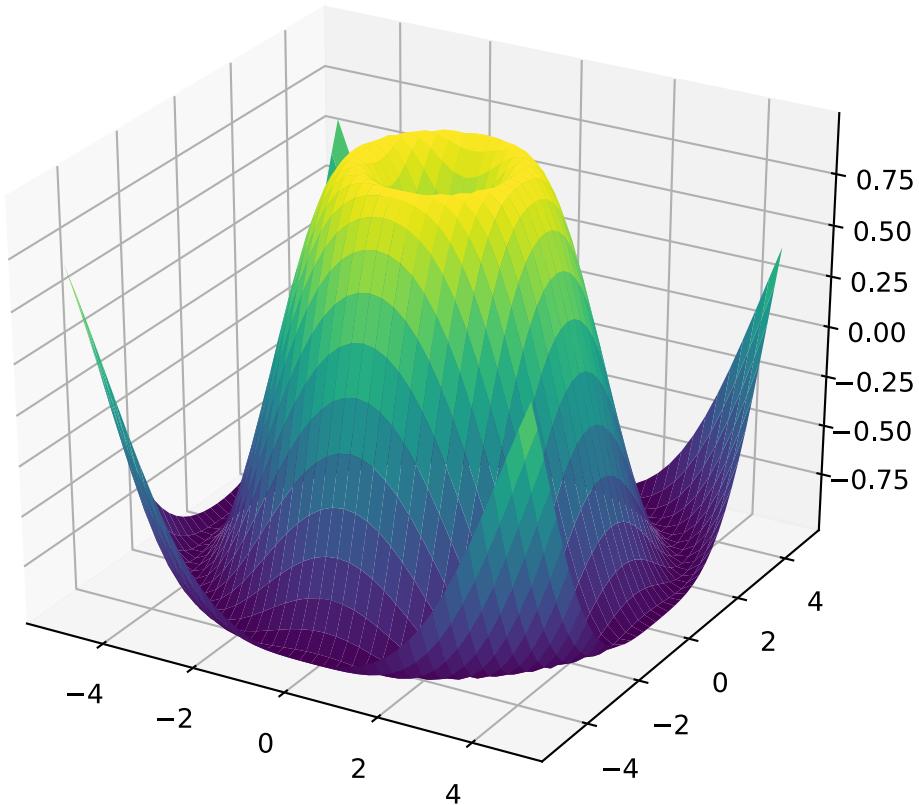
- *New in matplotlib 0.99*
 - *New documentation*
 - *mplot3d*
 - *axes grid toolkit*
 - *Axis spine placement*

New documentation

Jae-Joon Lee has written two new guides *Legend guide* and *Advanced Annotation*. Michael Sarahan has written *Image tutorial*. John Hunter has written two new tutorials on working with paths and transformations: *Path Tutorial* and *Transformations Tutorial*.

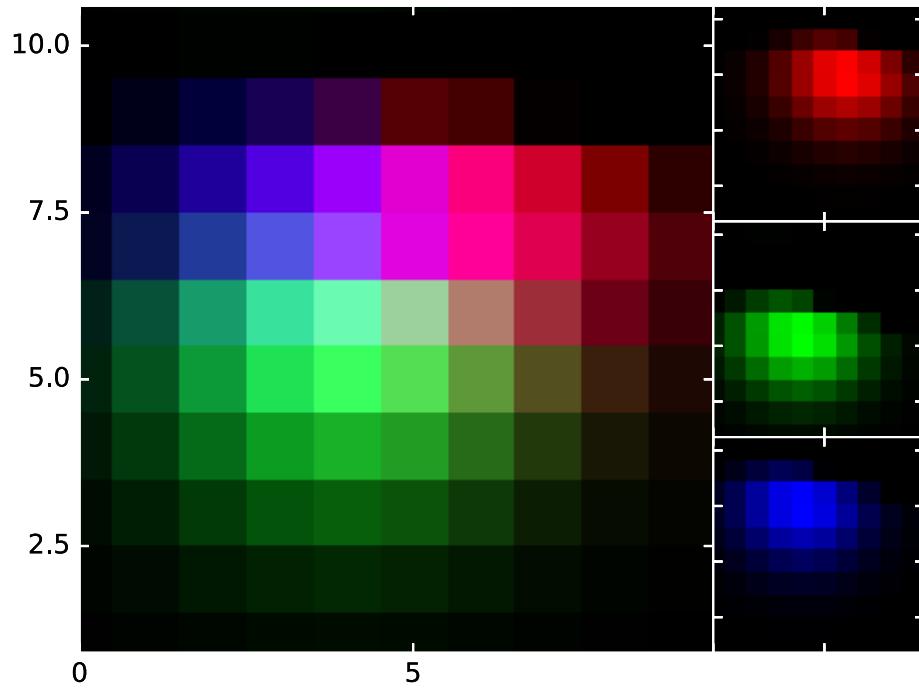
mplot3d

Reinier Heeres has ported John Porter's mplot3d over to the new matplotlib transformations framework, and it is now available as a toolkit `mpl_toolkits.mplot3d` (which now comes standard with all mpl installs). See [mplot3d Examples](#) and [mplot3d tutorial](#)



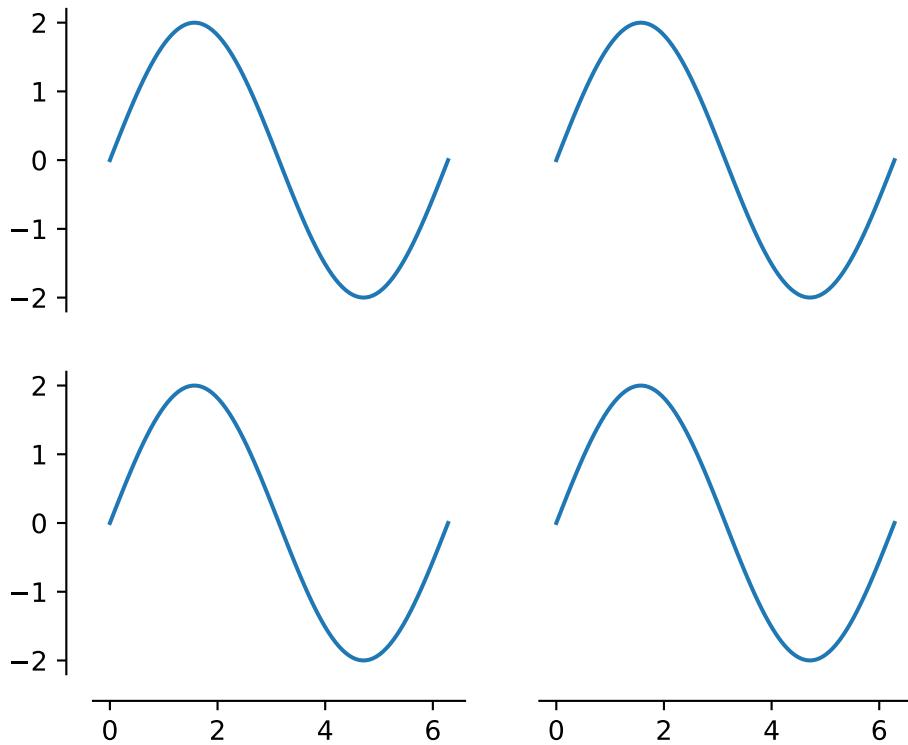
axes grid toolkit

Jae-Joon Lee has added a new toolkit to ease displaying multiple images in matplotlib, as well as some support for curvilinear grids to support the world coordinate system. The toolkit is included standard with all new mpl installs. See [axes_grid Examples](#) and [The Matplotlib AxesGrid Toolkit User's Guide](#).



Axis spine placement

Andrew Straw has added the ability to place “axis spines” – the lines that denote the data limits – in various arbitrary locations. No longer are your axis lines constrained to be a simple rectangle around the figure – you can turn on or off left, bottom, right and top, as well as “detach” the spine to offset it away from the data. See [pylab_examples example code: spine_placement_demo.py](#) and [matplotlib.spines.Spine](#).



9.2.3 New in matplotlib 1.0

Table of Contents

- *New in matplotlib 1.0*
 - *HTML5/Canvas backend*
 - *Sophisticated subplot grid layout*
 - *Easy pythonic subplots*
 - *Contour fixes and and triplot*
 - *multiple calls to show supported*
 - *mplot3d graphs can be embedded in arbitrary axes*
 - *tick_params*
 - *Lots of performance and feature enhancements*
 - *Much improved software carpentry*
 - *Bugfix marathon*

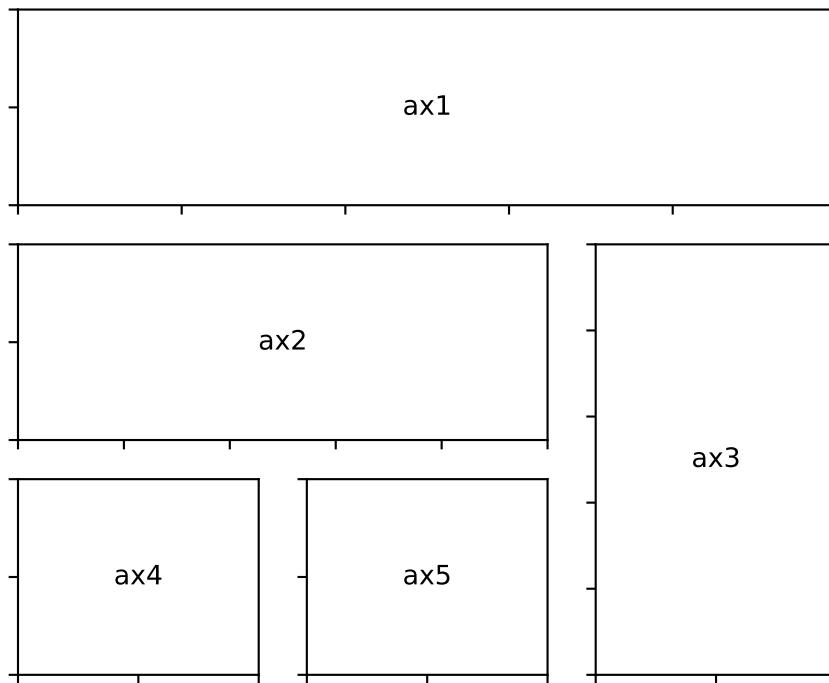
HTML5/Canvas backend

Simon Ratcliffe and Ludwig Schwardt have released an [HTML5/Canvas](#) backend for matplotlib. The backend is almost feature complete, and they have done a lot of work comparing their html5 rendered images with our core renderer Agg. The backend features client/server interactive navigation of matplotlib figures in an html5 compliant browser.

Sophisticated subplot grid layout

Jae-Joon Lee has written [*gridspec*](#), a new module for doing complex subplot layouts, featuring row and column spans and more. See [Customizing Location of Subplot Using GridSpec](#) for a tutorial overview.

subplot2grid



Easy pythonic subplots

Fernando Perez got tired of all the boilerplate code needed to create a figure and multiple subplots when using the matplotlib API, and wrote a [`subplots\(\)`](#) helper function. Basic usage allows you to create the figure and an array of subplots with numpy indexing (starts with 0). e.g.:

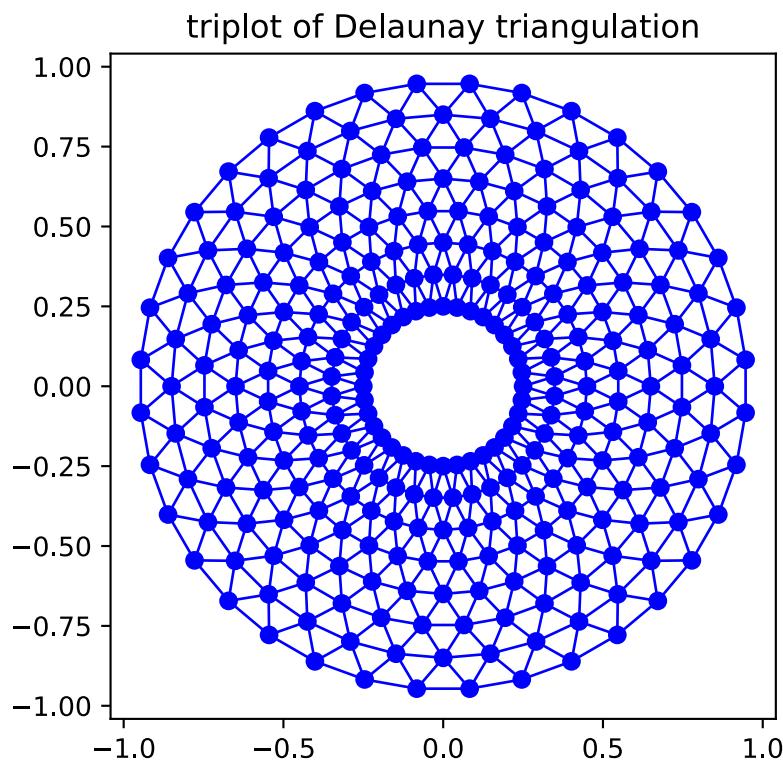
```
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3]) # upper, left
```

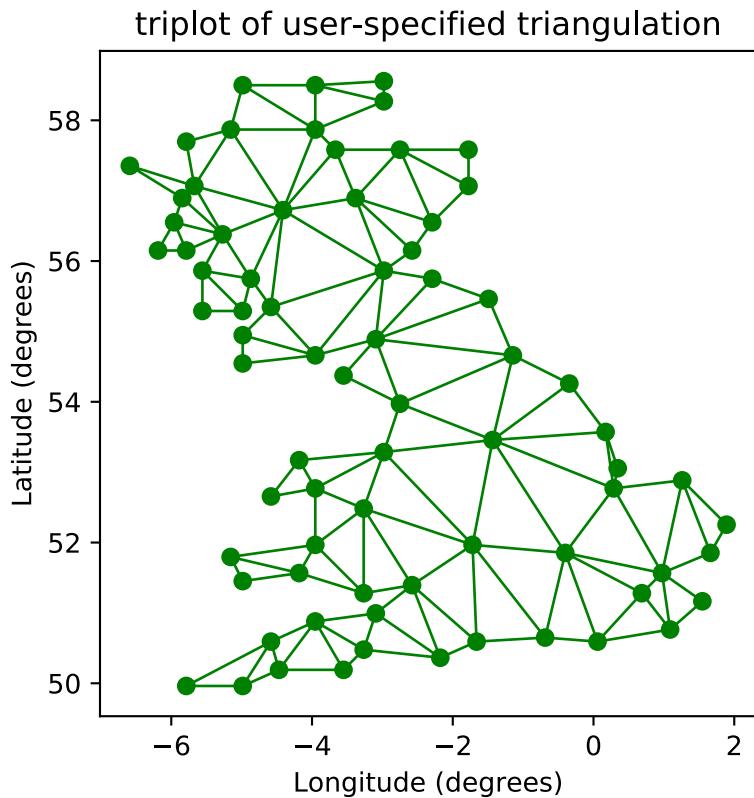
See [pylab_examples example code: subplots_demo.py](#) for several code examples.

Contour fixes and and triplot

Ian Thomas has fixed a long-standing bug that has vexed our most talented developers for years. `contourf()` now handles interior masked regions, and the boundaries of line and filled contours coincide.

Additionally, he has contributed a new module `tri` and helper function `triplot()` for creating and plotting unstructured triangular grids.



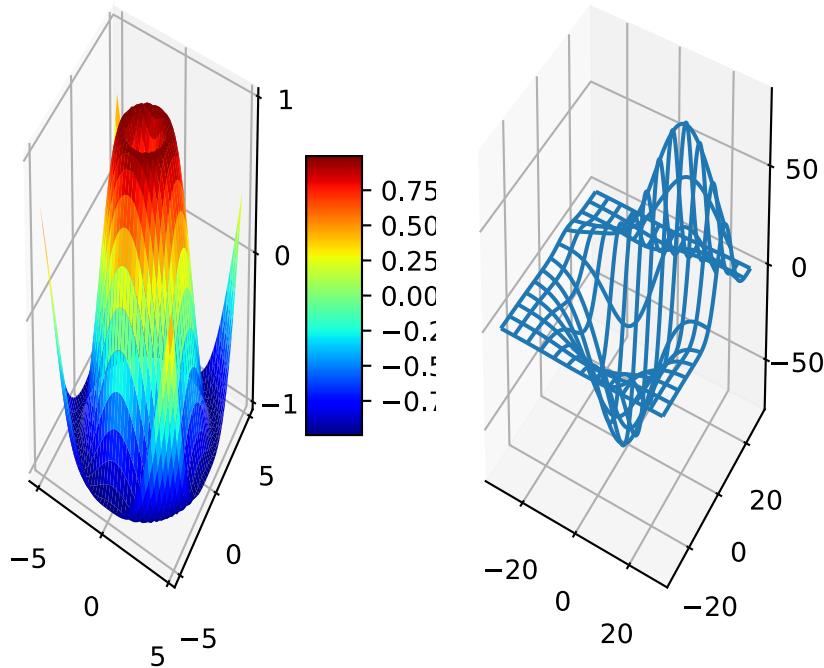


multiple calls to show supported

A long standing request is to support multiple calls to `show()`. This has been difficult because it is hard to get consistent behavior across operating systems, user interface toolkits and versions. Eric Firing has done a lot of work on rationalizing show across backends, with the desired behavior to make show raise all newly created figures and block execution until they are closed. Repeated calls to show should raise newly created figures since the last call. Eric has done a lot of testing on the user interface toolkits and versions and platforms he has access to, but it is not possible to test them all, so please report problems to the mailing list and bug tracker.

mplot3d graphs can be embedded in arbitrary axes

You can now place an mplot3d graph into an arbitrary axes location, supporting mixing of 2D and 3D graphs in the same figure, and/or multiple 3D graphs in a single figure, using the “projection” keyword argument to add_axes or add_subplot. Thanks Ben Root.



tick_params

Eric Firing wrote `tick_params`, a convenience method for changing the appearance of ticks and tick labels. See pyplot function `tick_params\(\)` and associated Axes method `tick_params\(\)`.

Lots of performance and feature enhancements

- Faster magnification of large images, and the ability to zoom in to a single pixel
- Local installs of documentation work better
- Improved “widgets” – mouse grabbing is supported
- More accurate snapping of lines to pixel boundaries
- More consistent handling of color, particularly the alpha channel, throughout the API

Much improved software carpentry

The matplotlib trunk is probably in as good a shape as it has ever been, thanks to improved `software carpentry`. We now have a `buildbot` which runs a suite of `nose` regression tests on every svn commit, auto-generating a set of images and comparing them against a set of known-goods, sending emails to developers on failures

with a pixel-by-pixel image comparison. Releases and release bugfixes happen in branches, allowing active new feature development to happen in the trunk while keeping the release branches stable. Thanks to Andrew Straw, Michael Droettboom and other matplotlib developers for the heavy lifting.

Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the [bug tracker](#) with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

9.2.4 New in matplotlib 1.1

Table of Contents

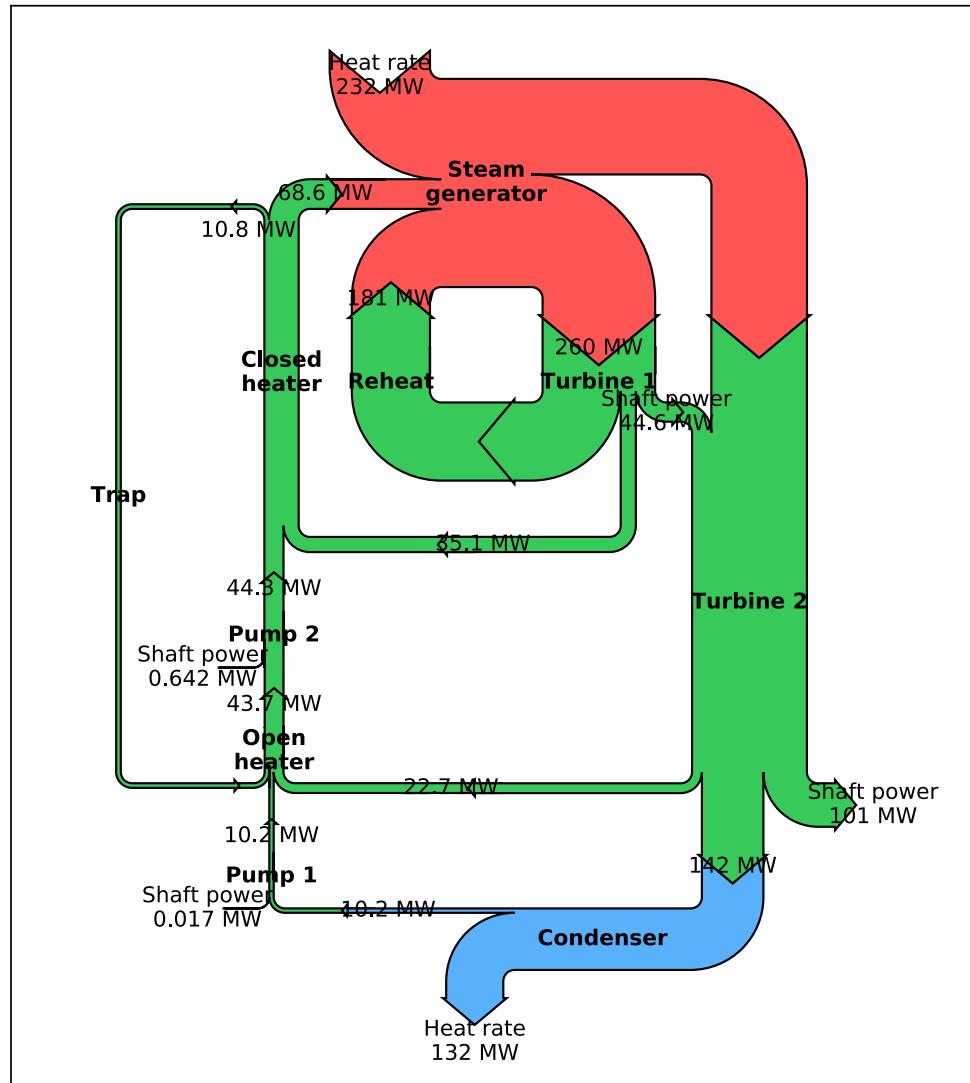
- *New in matplotlib 1.1*
 - *Sankey Diagrams*
 - *Animation*
 - *Tight Layout*
 - *PyQT4, PySide, and IPython*
 - *Legend*
 - *mplot3d*
 - *Numerix support removed*
 - *Markers*
 - *Other improvements*

Note: matplotlib 1.1 supports Python 2.4 to 2.7

Sankey Diagrams

Kevin Davies has extended Yannick Copin's original Sankey example into a module ([sankey](#)) and provided new examples ([api example code: sankey_demo_basics.py](#), [api example code: sankey_demo_links.py](#), [api example code: sankey_demo_rankine.py](#)).

Rankine Power Cycle: Example 8.6 from Moran and Shapiro
 "Fundamentals of Engineering Thermodynamics ", 6th ed., 2008



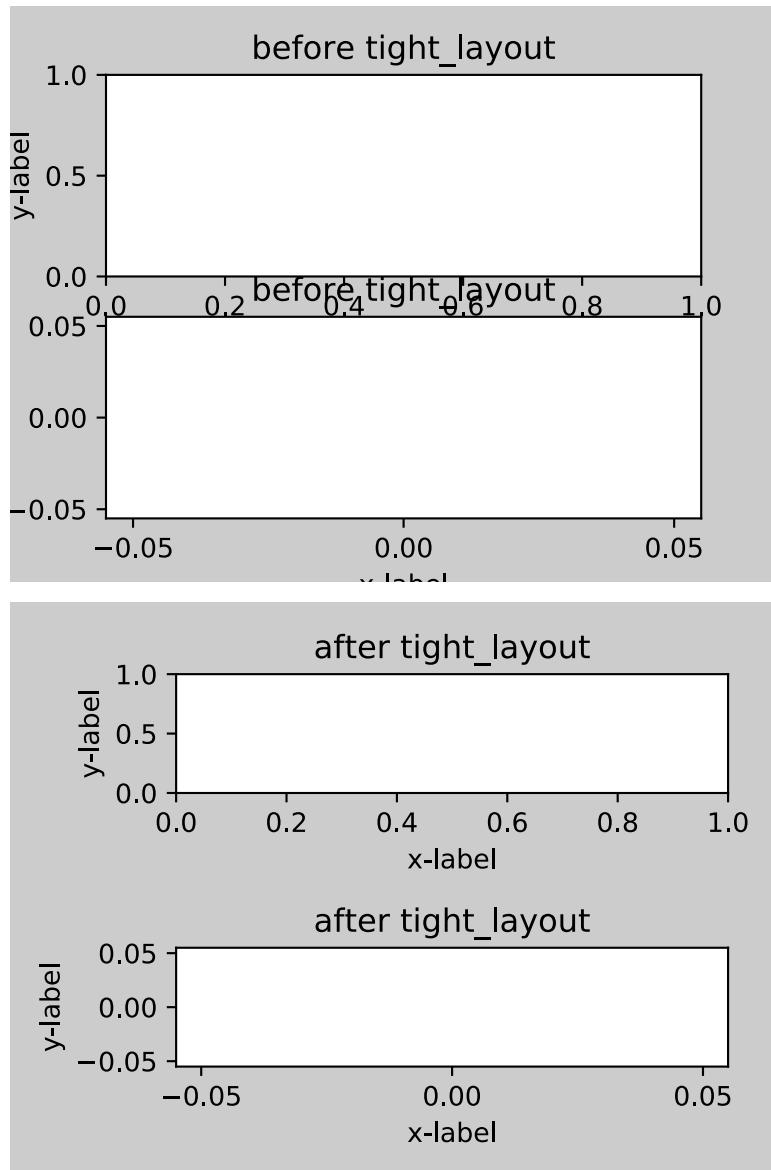
Animation

Ryan May has written a backend-independent framework for creating animated figures. The [animation](#) module is intended to replace the backend-specific examples formerly in the [Matplotlib Examples](#) listings. Examples using the new framework are in [animation Examples](#); see the entrancing [double pendulum](#) which uses `matplotlib.animation.Animation.save()` to create the movie below.

This should be considered as a beta release of the framework; please try it and provide feedback.

Tight Layout

A frequent issue raised by users of matplotlib is the lack of a layout engine to nicely space out elements of the plots. While matplotlib still adheres to the philosophy of giving users complete control over the placement of plot elements, Jae-Joon Lee created the [tight_layout](#) module and introduced a new command `tight_layout()` to address the most common layout issues.



The usage of this functionality can be as simple as

```
plt.tight_layout()
```

and it will adjust the spacing between subplots so that the axis labels do not overlap with neighboring subplots. A [Tight Layout guide](#) has been created to show how to use this new tool.

PyQT4, PySide, and IPython

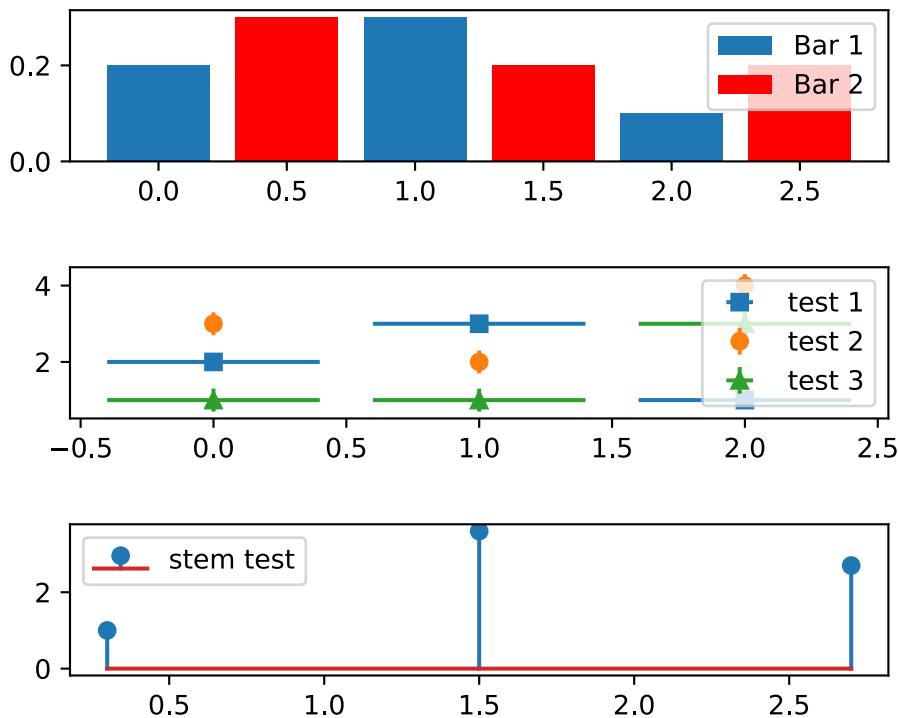
Gerald Storer made the Qt4 backend compatible with PySide as well as PyQt4. At present, however, PySide does not support the PyOS_InputHook mechanism for handling gui events while waiting for text input, so it cannot be used with the new version 0.11 of [IPython](#). Until this feature appears in PySide, IPython users should use the PyQt4 wrapper for QT4, which remains the matplotlib default.

An rcParam entry, “backend.qt4”, has been added to allow users to select PyQt4, PyQt4v2, or PySide. The latter two use the Version 2 Qt API. In most cases, users can ignore this rcParam variable; it is available to aid in testing, and to provide control for users who are embedding matplotlib in a PyQt4 or PySide app.

Legend

Jae-Joon Lee has improved plot legends. First, legends for complex plots such as [stem](#) plots will now display correctly. Second, the ‘best’ placement of a legend has been improved in the presence of NaNs.

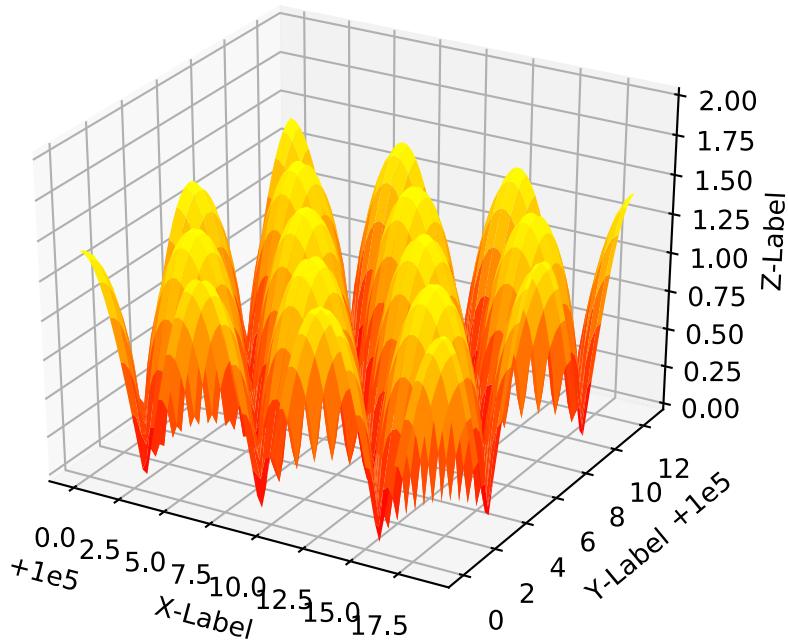
See the [Legend guide](#) for more detailed explanation and examples.



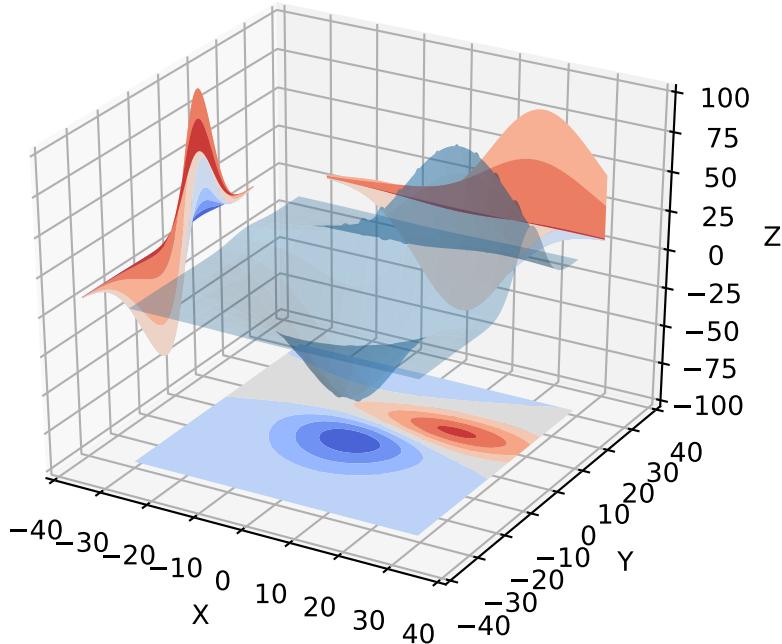
mplot3d

In continuing the efforts to make 3D plotting in matplotlib just as easy as 2D plotting, Ben Root has made several improvements to the `mplot3d` module.

- `Axes3D` has been improved to bring the class towards feature-parity with regular Axes objects
- Documentation for `mplot3d` was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:



- `contourf()` gains `zdir` and `offset` kwargs. You can now do this:



Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

Markers

The list of available markers for `plot()` and `scatter()` has now been merged. While they were mostly similar, some markers existed for one function, but not the other. This merge did result in a conflict for the ‘d’ diamond marker. Now, ‘d’ will be interpreted to always mean “thin” diamond while ‘D’ will mean “regular” diamond.

Thanks to Michael Droettboom for this effort.

Other improvements

- Unit support for polar axes and `arrow()`
- `PolarAxes` gains getters and setters for “theta_direction”, and “theta_offset” to allow for theta to go in either the clock-wise or counter-clockwise direction and to specify where zero degrees should be placed. `set_theta_zero_location()` is an added convenience function.

- Fixed error in argument handling for tri-functions such as `tripcolor()`
- `axes.labelweight` parameter added to rcParams.
- For `imshow()`, `interpolation='nearest'` will now always perform an interpolation. A “none” option has been added to indicate no interpolation at all.
- An error in the Hammer projection has been fixed.
- `clabel` for `contour()` now accepts a callable. Thanks to Daniel Hyams for the original patch.
- Jae-Joon Lee added the HBox and VBox classes.
- Christoph Gohlke reduced memory usage in `imshow()`.
- `scatter()` now accepts empty inputs.
- The behavior for ‘symlog’ scale has been fixed, but this may result in some minor changes to existing plots. This work was refined by ssyr.
- Peter Butterworth added named figure support to `figure()`.
- Michiel de Hoon has modified the MacOSX backend to make its interactive behavior consistent with the other backends.
- Pim Schellart added a new colormap called “cubehelix”. Sameer Grover also added a colormap called “coolwarm”. See it and all other colormaps [here](#).
- Many bug fixes and documentation improvements.

9.2.5 New in matplotlib 1.2

Table of Contents

- *New in matplotlib 1.2*
 - *Python 3.x support*
 - *PGF/TikZ backend*
 - *Locator interface*
 - *Tri-Surface Plots*
 - *Control the lengths of colorbar extensions*
 - *Figures are picklable*
 - *Set default bounding box in matplotlibrc*
 - *New Boxplot Functionality*
 - *New RC parameter functionality*
 - *Streamplot*
 - *New hist functionality*

- Updated shipped dependencies
- Face-centred colors in tripcolor plots
- Hatching patterns in filled contour plots, with legends
- Known issues in the matplotlib 1.2 release

Note: matplotlib 1.2 supports Python 2.6, 2.7, and 3.1

Python 3.x support

Matplotlib 1.2 is the first version to support Python 3.x, specifically Python 3.1 and 3.2. To make this happen in a reasonable way, we also had to drop support for Python versions earlier than 2.6.

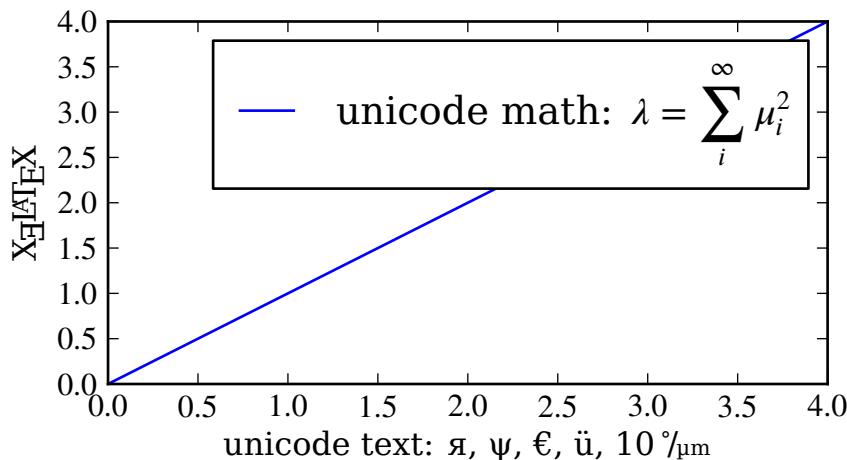
This work was done by Michael Droettboom, the Cape Town Python Users' Group, many others and supported financially in part by the SAGE project.

The following GUI backends work under Python 3.x: Gtk3Cairo, Qt4Agg, TkAgg and MacOSX. The other GUI backends do not yet have adequate bindings for Python 3.x, but continue to work on Python 2.6 and 2.7, particularly the Qt and QtAgg backends (which have been deprecated). The non-GUI backends, such as PDF, PS and SVG, work on both Python 2.x and 3.x.

Features that depend on the Python Imaging Library, such as JPEG handling, do not work, since the version of PIL for Python 3.x is not sufficiently mature.

PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX. These can be processed by PdfLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in [Typesetting With XeLaTeX/LuaLaTeX](#).



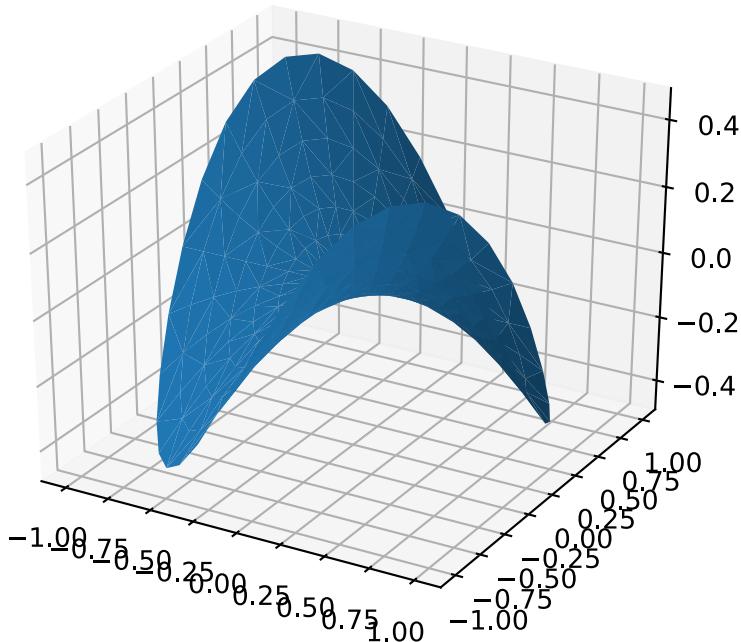
Locator interface

Philip Elson exposed the intelligence behind the tick Locator classes with a simple interface. For instance, to get no more than 5 sensible steps which span the values 10 and 19.5:

```
>>> import matplotlib.ticker as mticker  
>>> locator = mticker.MaxNLocator(nbins=5)  
>>> print(locator.tick_values(10, 19.5))  
[ 10.  12.  14.  16.  18.  20.]
```

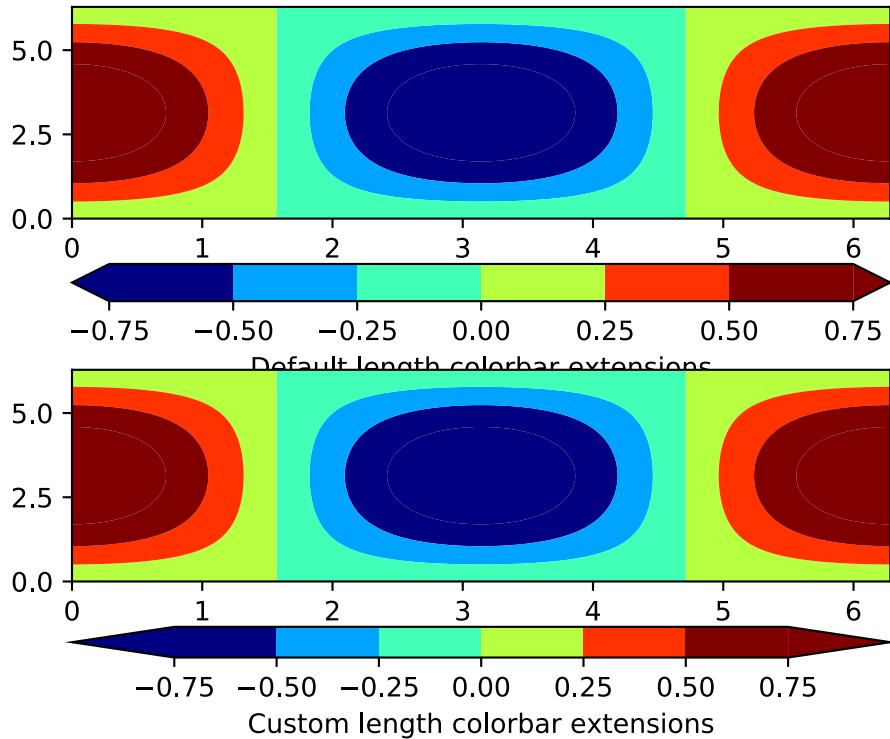
Tri-Surface Plots

Damon McDougall added a new plotting method for the `mpl_toolkits.mplot3d` toolkit called `plot_trisurf()`.



Control the lengths of colorbar extensions

Andrew Dawson added a new keyword argument `extendfrac` to `colorbar()` to control the length of minimum and maximum colorbar extensions.



Figures are picklable

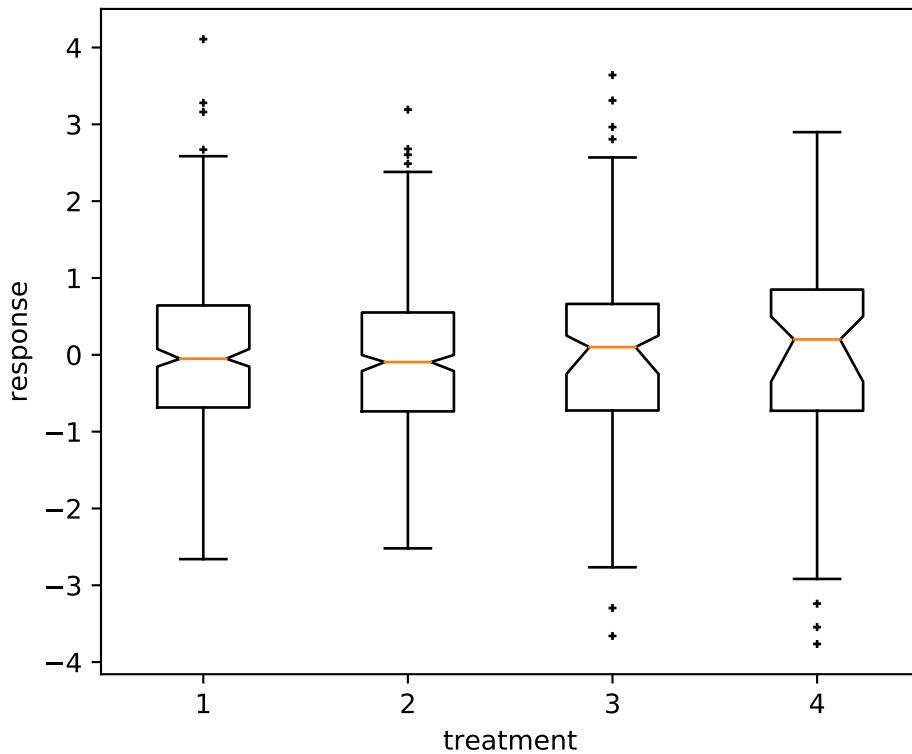
Philip Elson added an experimental feature to make figures picklable for quick and easy short-term storage of plots. Pickle files are not designed for long term storage, are unsupported when restoring a pickle saved in another matplotlib version and are insecure when restoring a pickle from an untrusted source. Having said this, they are useful for short term storage for later modification inside matplotlib.

Set default bounding box in matplotlibrc

Two new defaults are available in the matplotlibrc configuration file: `savefig.bbox`, which can be set to ‘standard’ or ‘tight’, and `savefig.pad_inches`, which controls the bounding box padding.

New Boxplot Functionality

Users can now incorporate their own methods for computing the median and its confidence intervals into the `boxplot()` method. For every column of data passed to `boxplot`, the user can specify an accompanying median and confidence interval.



New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: `rc_file()` and `rc_context`. To load RC parameters from a file:

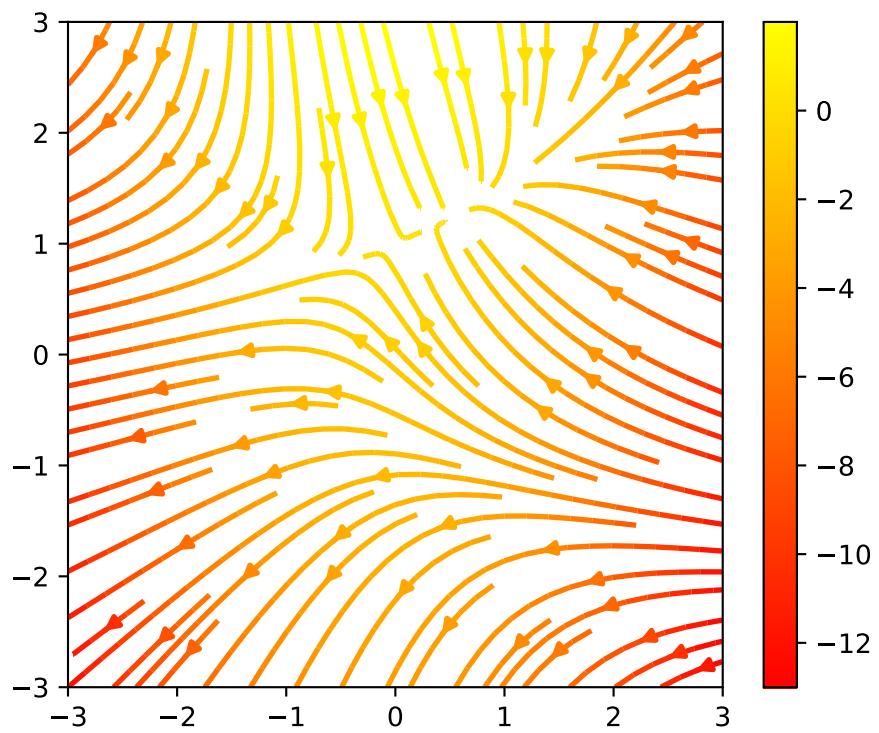
```
>>> mpl.rc_file('mpl.rc')
```

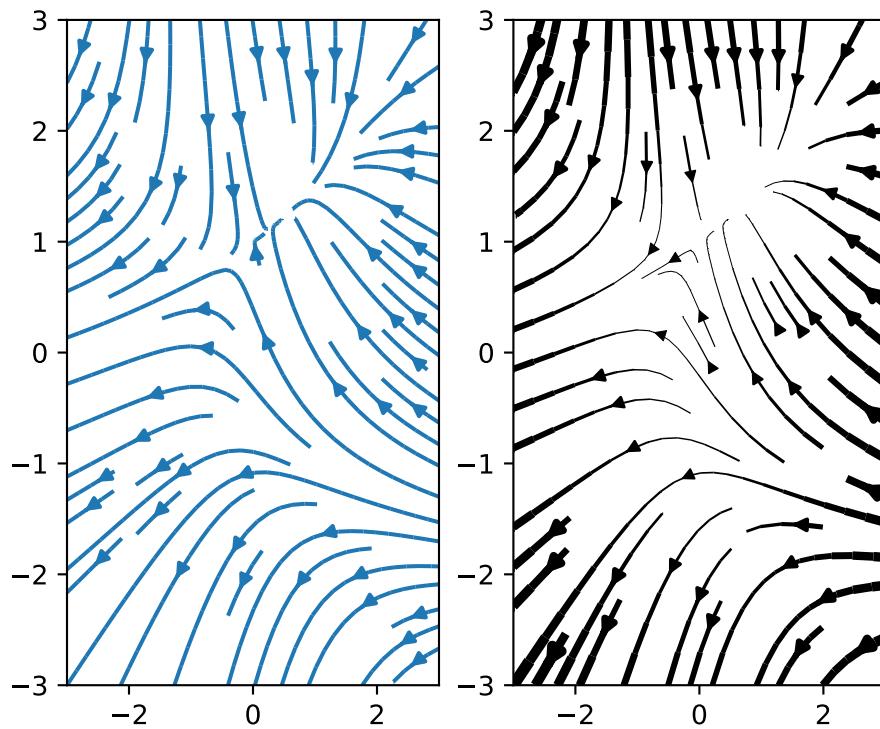
To temporarily use RC parameters:

```
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
>>>     ...
```

Streamplot

Tom Flannaghan and Tony Yu have added a new `streamplot()` function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing `quiver()` function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, `streamplot()` allows users to map the colors and/or line widths of the streamlines to a separate parameter, such as the speed or local intensity of the vector field.





New hist functionality

Nic Eggert added a new `stacked` kwarg to `hist()` that allows creation of stacked histograms using any of the histogram types. Previously, this functionality was only available by using the `barstacked` histogram type. Now, when `stacked=True` is passed to the function, any of the histogram types can be stacked. The `barstacked` histogram type retains its previous functionality for backwards compatibility.

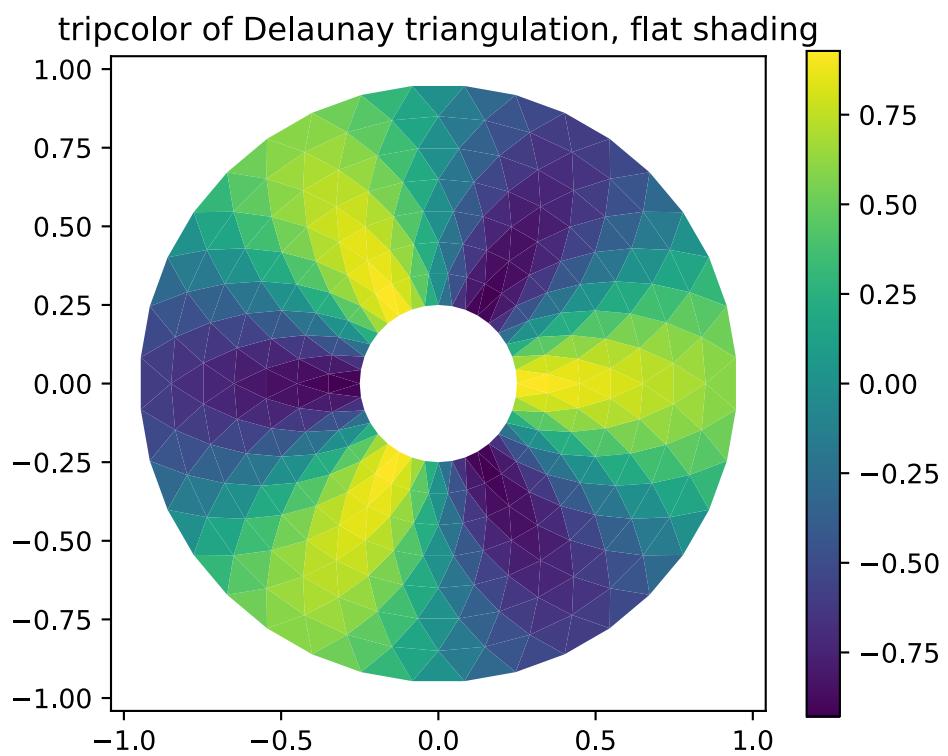
Updated shipped dependencies

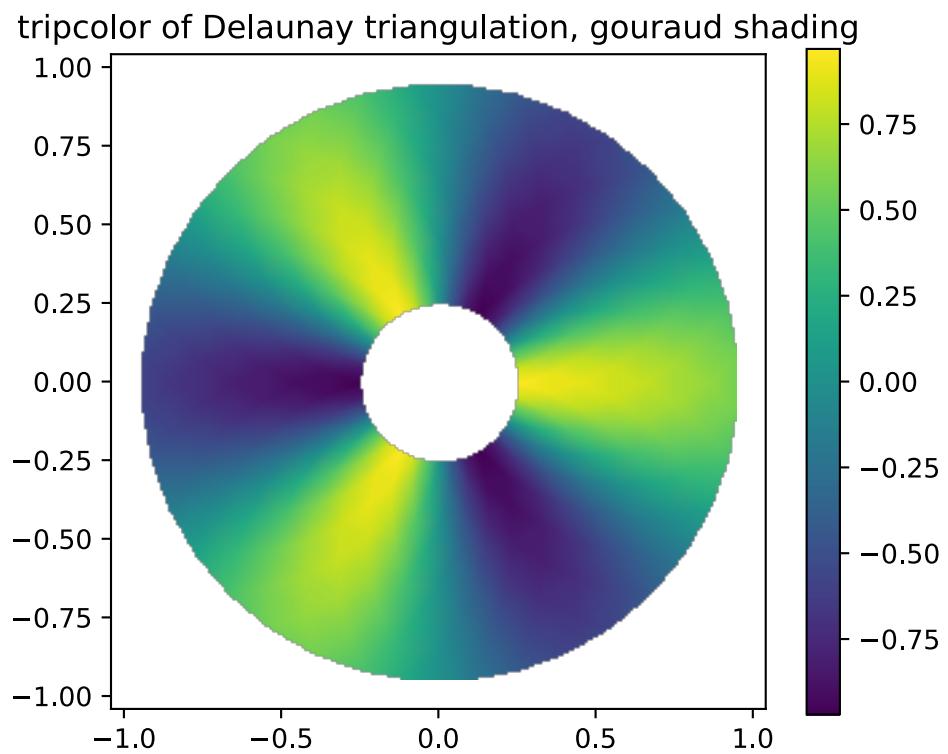
The following dependencies that ship with matplotlib and are optionally installed alongside it have been updated:

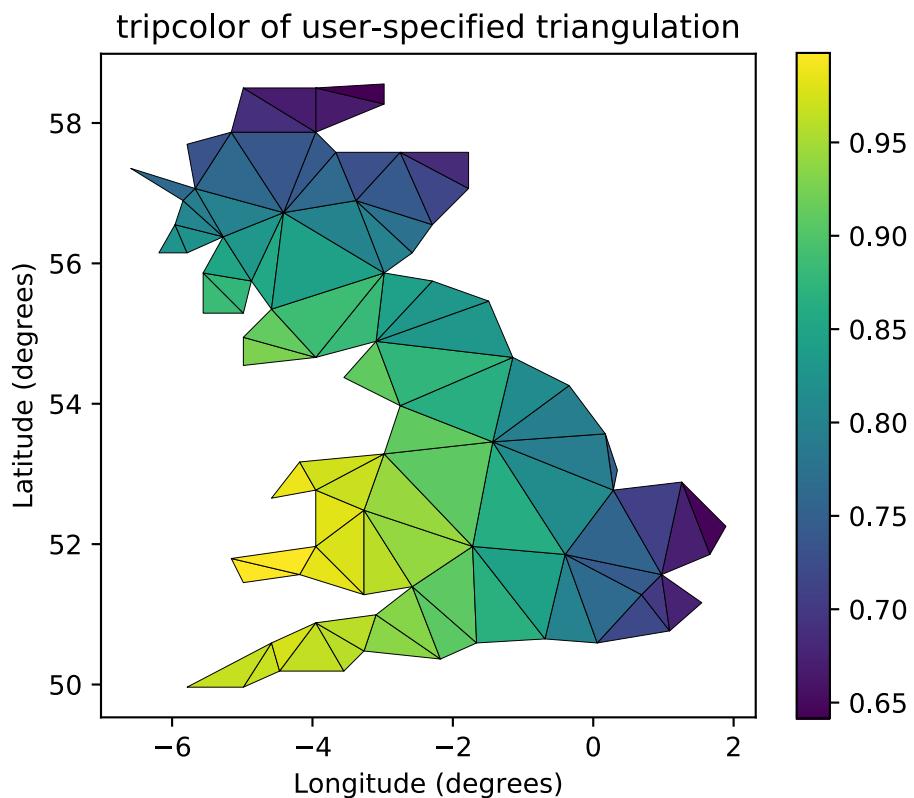
- `pytz` 2012d
- `dateutil` 1.5 on Python 2.x, and 2.1 on Python 3.x

Face-centred colors in tripcolor plots

Ian Thomas extended `tripcolor()` to allow one color value to be specified for each triangular face rather than for each point in a triangulation.

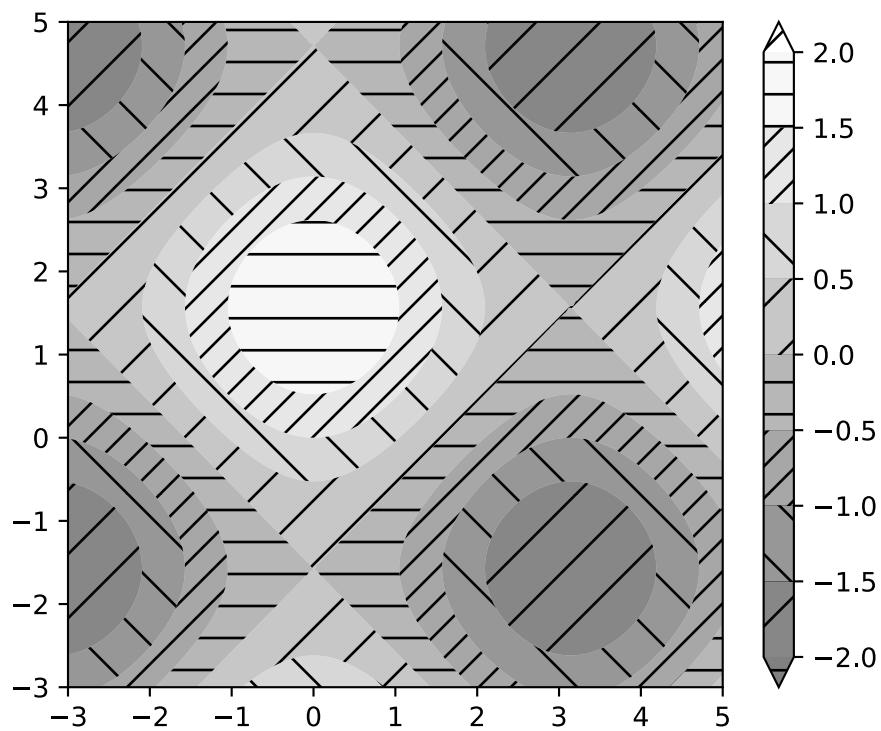


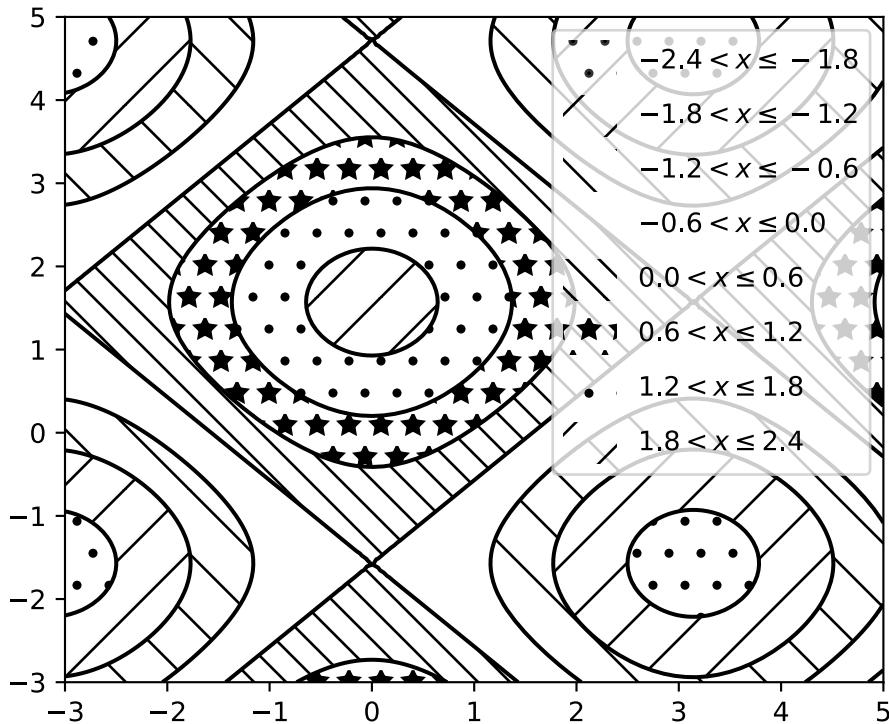




Hatching patterns in filled contour plots, with legends

Phil Elson added support for hatching to `contourf()`, together with the ability to use a legend to identify contoured ranges.





Known issues in the matplotlib 1.2 release

- When using the Qt4Agg backend with IPython 0.11 or later, the save dialog will not display. This should be fixed in a future version of IPython.

9.2.6 New in matplotlib 1.2.2

Table of Contents

- *New in matplotlib 1.2.2*
 - *Improved collections*
 - *Multiple images on same axes are correctly transparent*

Improved collections

The individual items of a collection may now have different alpha values and be rendered correctly. This also fixes a bug where collections were always filled in the PDF backend.

Multiple images on same axes are correctly transparent

When putting multiple images onto the same axes, the background color of the axes will now show through correctly.

9.2.7 New in matplotlib 1.3

Table of Contents

- *New in matplotlib 1.3*
 - *New in 1.3.1*
 - *New plotting features*
 - *Updated Axes3D.contour methods*
 - *Drawing*
 - *Text*
 - *Configuration (rcParams)*
 - *Backends*
 - *Documentation and examples*
 - *Infrastructure*

Note: matplotlib 1.3 supports Python 2.6, 2.7, 3.2, and 3.3

New in 1.3.1

1.3.1 is a bugfix release, primarily dealing with improved setup and handling of dependencies, and correcting and enhancing the documentation.

The following changes were made in 1.3.1 since 1.3.0.

Enhancements

- Added a context manager for creating multi-page pdfs (see `matplotlib.backends.backend_pdf.PdfPages`).
- The WebAgg backend should now have lower latency over heterogeneous Internet connections.

Bug fixes

- Histogram plots now contain the endline.
- Fixes to the Molleweide projection.
- Handling recent fonts from Microsoft and Macintosh-style fonts with non-ascii metadata is improved.
- Hatching of fill between plots now works correctly in the PDF backend.
- Tight bounding box support now works in the PGF backend.
- Transparent figures now display correctly in the Qt4Agg backend.
- Drawing lines from one subplot to another now works.
- Unit handling on masked arrays has been improved.

Setup and dependencies

- Now works with any version of pyparsing 1.5.6 or later, without displaying hundreds of warnings.
- Now works with 64-bit versions of Ghostscript on MS-Windows.
- When installing from source into an environment without Numpy, Numpy will first be downloaded and built and then used to build matplotlib.
- Externally installed backends are now always imported using a fully-qualified path to the module.
- Works with newer version of wxPython.
- Can now build with a PyCXX installed globally on the system from source.
- Better detection of Gtk3 dependencies.

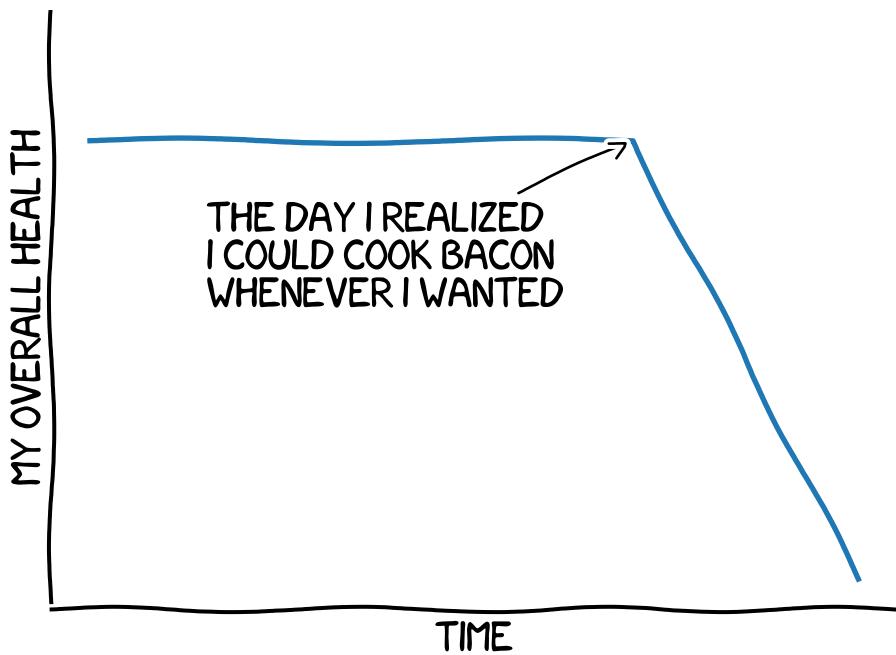
Testing

- Tests should now work in non-English locales.
- PEP8 conformance tests now report on locations of issues.

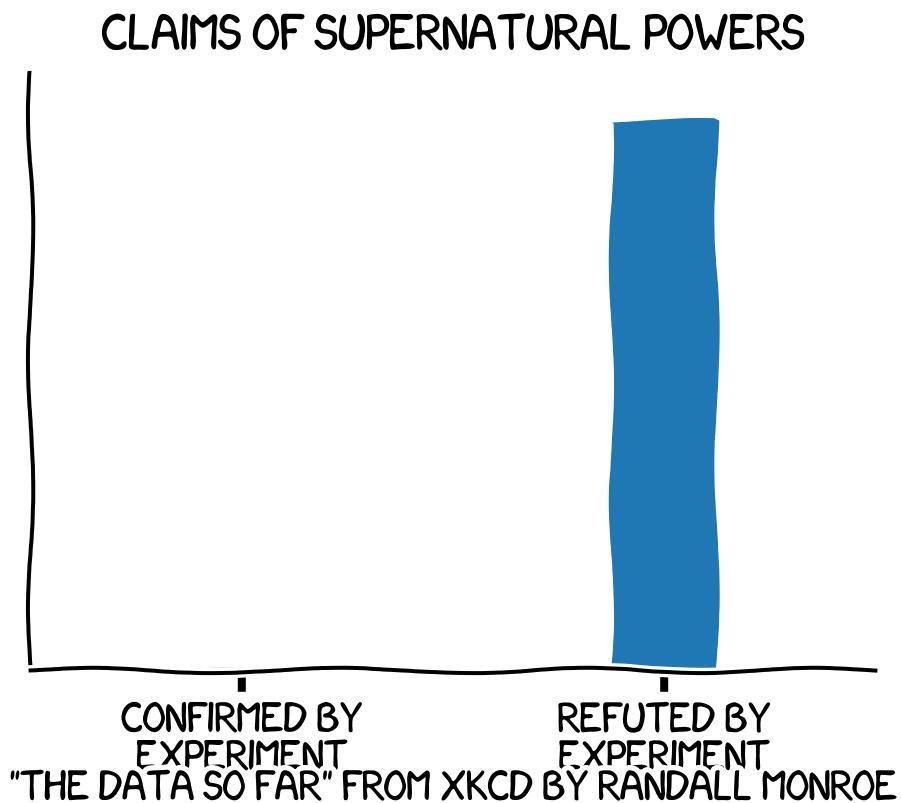
New plotting features

xkcd-style sketch plotting

To give your plots a sense of authority that they may be missing, Michael Droettboom (inspired by the work of many others in [PR #1329](#)) has added an `xkcd-style` sketch plotting mode. To use it, simply call `matplotlib.pyplot.xkcd()` before creating your plot. For really fine control, it is also possible to modify each artist's sketch parameters individually with `matplotlib.artist.Artist.set_sketch_params()`.

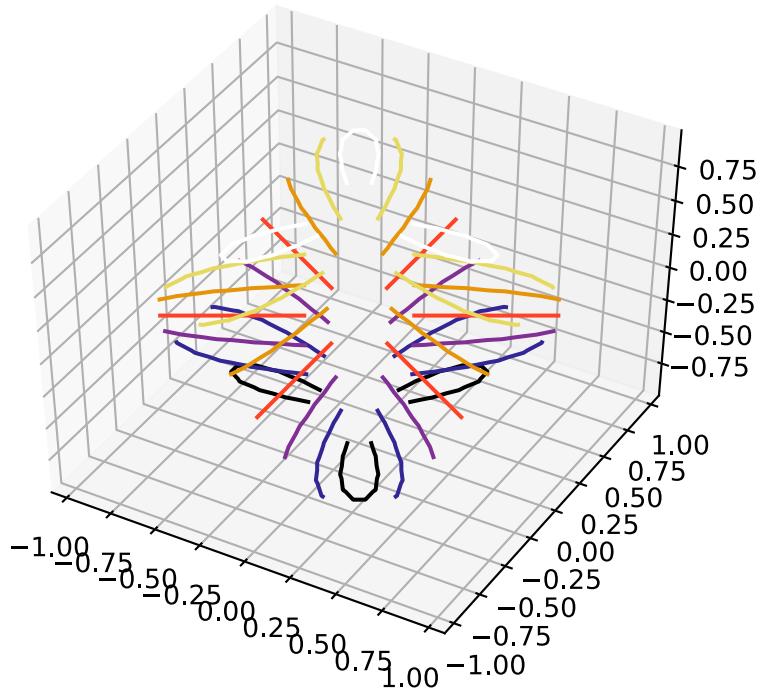


"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE



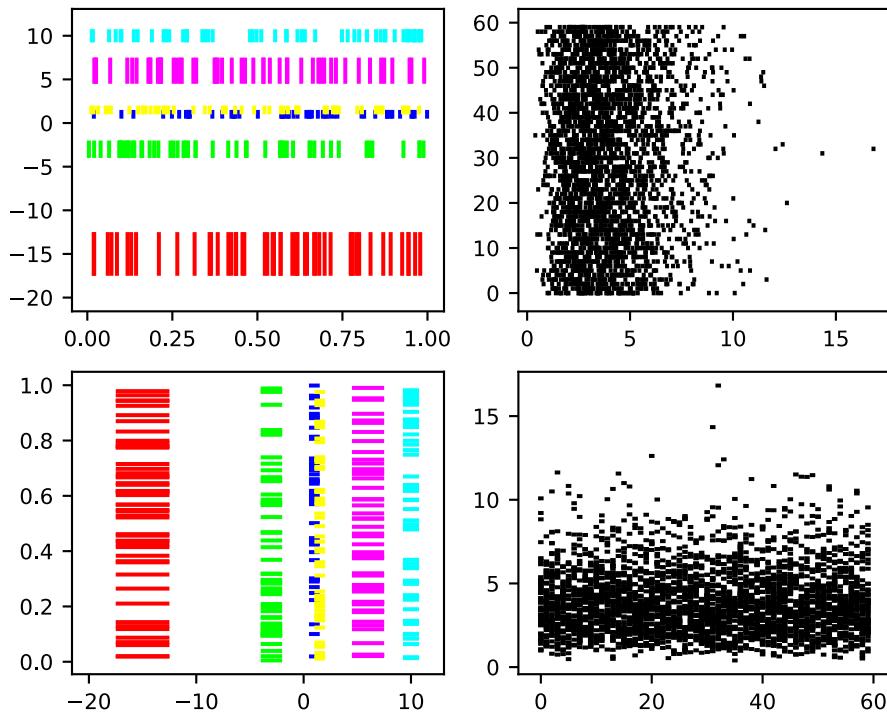
Updated Axes3D.contour methods

Damon McDougall updated the `tricontour()` and `tricontourf()` methods to allow 3D contour plots on arbitrary unstructured user-specified triangulations.



New eventplot plot type

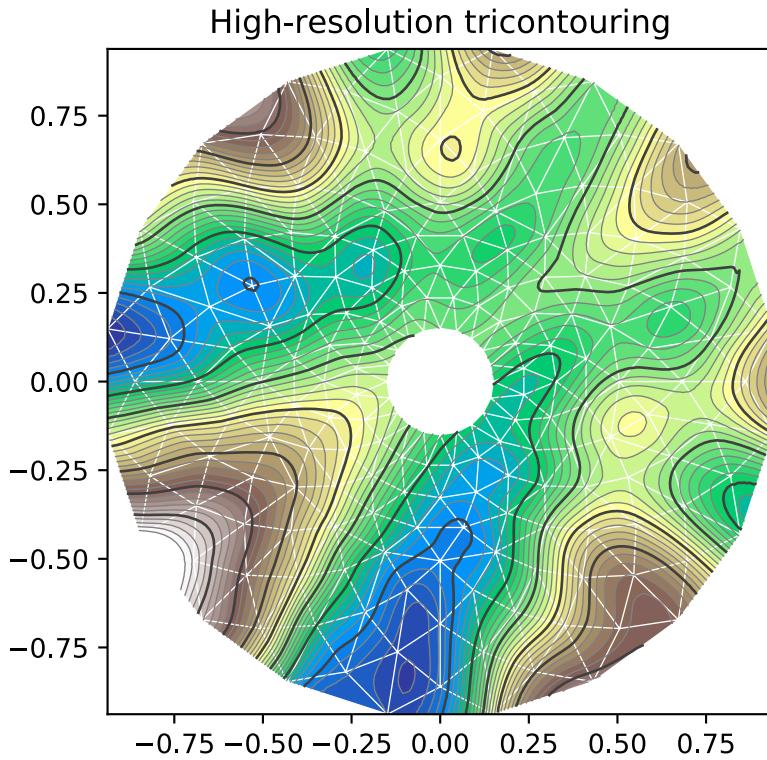
Todd Jennings added a `eventplot()` function to create multiple rows or columns of identical line segments



As part of this feature, there is a new [EventCollection](#) class that allows for plotting and manipulating rows or columns of identical line segments.

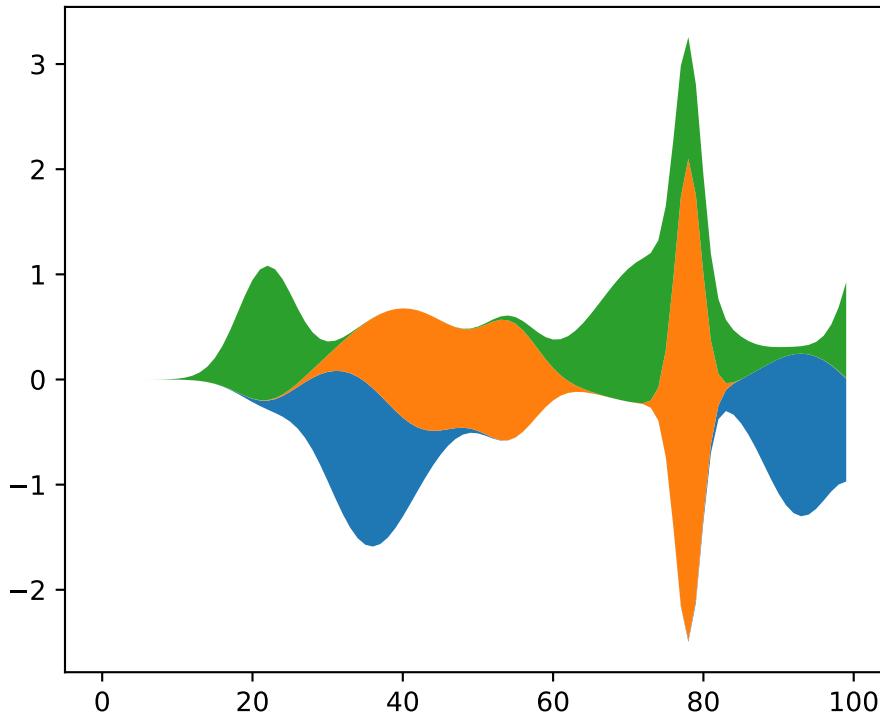
Triangular grid interpolation

Geoffroy Billotey and Ian Thomas added classes to perform interpolation within triangular grids: ([LinearTriInterpolator](#) and [CubicTriInterpolator](#)) and a utility class to find the triangles in which points lie ([TrapezoidMapTriFinder](#)). A helper class to perform mesh refinement and smooth contouring was also added ([UniformTriRefiner](#)). Finally, a class implementing some basic tools for triangular mesh improvement was added ([TriAnalyzer](#)).



Baselines for stackplot

Till Stensitzki added non-zero baselines to `stackplot()`. They may be symmetric or weighted.



Rectangular colorbar extensions

Andrew Dawson added a new keyword argument `extendrect` to `colorbar()` to optionally make colorbar extensions rectangular instead of triangular.

More robust boxplots

Paul Hobson provided a fix to the `boxplot()` method that prevent whiskers from being drawn inside the box for oddly distributed data sets.

Calling subplot() without arguments

A call to `subplot()` without any arguments now acts the same as `subplot(111)` or `subplot(1,1,1)` – it creates one axes for the whole figure. This was already the behavior for both `axes()` and `subplots()`, and now this consistency is shared with `subplot()`.

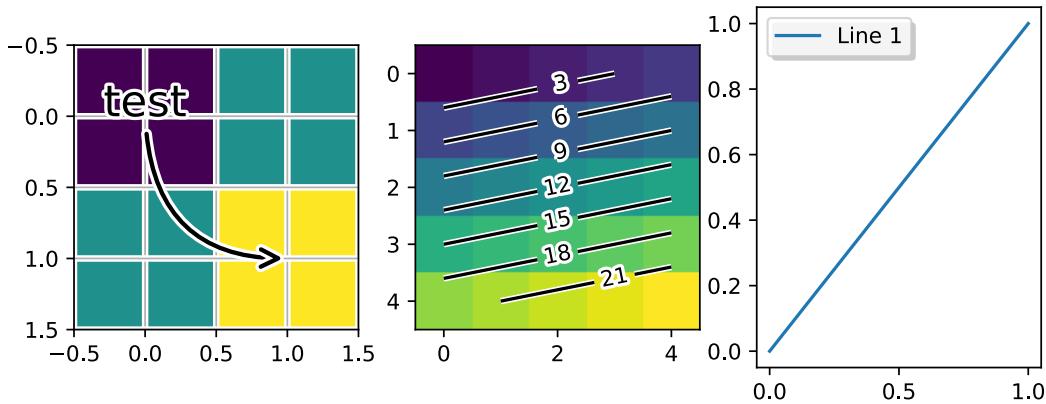
Drawing

Independent alpha values for face and edge colors

Wes Campaigne modified how `Patch` objects are drawn such that (for backends supporting transparency) you can set different alpha values for faces and edges, by specifying their colors in RGBA format. Note that if you set the `alpha` attribute for the patch object (e.g. using `set_alpha()` or the `alpha` keyword argument), that value will override the alpha components set in both the face and edge colors.

Path effects on lines

Thanks to Jae-Joon Lee, path effects now also work on plot lines.



Easier creation of colormap and normalizer for levels with colors

Phil Elson added the `matplotlib.colors.from_levels_and_colors()` function to easily create a colormap and normalizer for representation of discrete colors for plot types such as `matplotlib.pyplot.pcolor()`, with a similar interface to that of `contourf()`.

Full control of the background color

Wes Campaigne and Phil Elson fixed the Agg backend such that PNGs are now saved with the correct background color when `fig.patch.get_alpha()` is not 1.

Improved bbox_inches="tight" functionality

Passing `bbox_inches="tight"` through to `plt.savefig()` now takes into account *all* artists on a figure - this was previously not the case and led to several corner cases which did not function as expected.

Initialize a rotated rectangle

Damon McDougall extended the `Rectangle` constructor to accept an `angle` kwarg, specifying the rotation of a rectangle in degrees.

Text

Anchored text support

The `svg` and `pgf` backends are now able to save text alignment information to their output formats. This allows to edit text elements in saved figures, using Inkscape for example, while preserving their intended position. For `svg` please note that you'll have to disable the default text-to-path conversion (`mpl.rc('svg', fonttype='none')`).

Better vertical text alignment and multi-line text

The vertical alignment of text is now consistent across backends. You may see small differences in text placement, particularly with rotated text.

If you are using a custom backend, note that the `draw_text` renderer method is now passed the location of the baseline, not the location of the bottom of the text bounding box.

Multi-line text will now leave enough room for the height of very tall or very low text, such as superscripts and subscripts.

Left and right side axes titles

Andrew Dawson added the ability to add axes titles flush with the left and right sides of the top of the axes using a new keyword argument `loc` to `title()`.

Improved manual contour plot label positioning

Brian Mattern modified the manual contour plot label positioning code to interpolate along line segments and find the actual closest point on a contour to the requested position. Previously, the closest path vertex was used, which, in the case of straight contours was sometimes quite distant from the requested location. Much more precise label positioning is now possible.

Configuration (rcParams)

Quickly find rcParams

Phil Elson made it easier to search for rcParameters by passing a valid regular expression to `matplotlib.RcParams.find_all()`. `matplotlib.RcParams` now also has a pretty `repr` and `str` representation so that search results are printed prettily:

```
>>> import matplotlib
>>> print(matplotlib.rcParams.find_all('.size'))
RcParams({'font.size': 12,
          'xtick.major.size': 4,
          'xtick.minor.size': 2,
          'ytick.major.size': 4,
          'ytick.minor.size': 2})
```

axes.xmargin and axes.ymargin added to rcParams

rcParam values (axes.xmargin and axes.ymargin) were added to configure the default margins used. Previously they were hard-coded to default to 0, default value of both rcParam values is 0.

Changes to font rcParams

The font.* rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

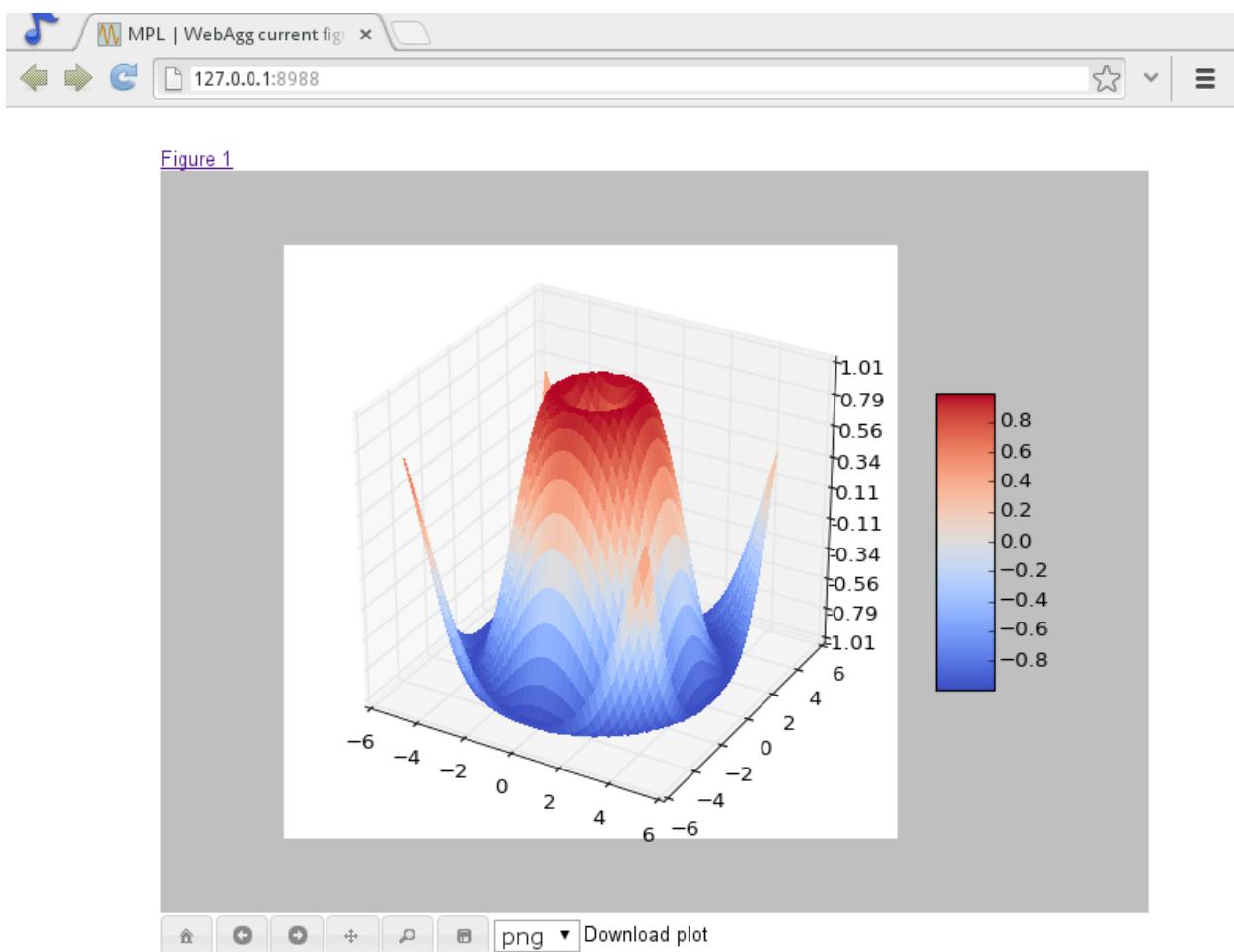
savefig.jpeg_quality added to rcParams

rcParam value savefig.jpeg_quality was added so that the user can configure the default quality used when a figure is written as a JPEG. The default quality is 95; previously, the default quality was 75. This change minimizes the artifacting inherent in JPEG images, particularly with images that have sharp changes in color as plots often do.

Backends

WebAgg backend

Michael Droettboom, Phil Elson and others have developed a new backend, WebAgg, to display figures in a web browser. It works with animations as well as being fully interactive.



Future versions of matplotlib will integrate this backend with the IPython notebook for a fully web browser based plotting frontend.

Remember save directory

Martin Spacek made the save figure dialog remember the last directory saved to. The default is configurable with the new `savefig.directory` rcParam in `matplotlibrc`.

Documentation and examples

Numpydoc docstrings

Nelle Varoquaux has started an ongoing project to convert matplotlib's docstrings to numpydoc format. See [MEP10](#) for more information.

Example reorganization

Tony Yu has begun work reorganizing the examples into more meaningful categories. The new gallery page is the fruit of this ongoing work. See [MEP12](#) for more information.

Examples now use subplots()

For the sake of brevity and clarity, most of the [examples](#) now use the newer `subplots()`, which creates a figure and one (or multiple) axes object(s) in one call. The old way involved a call to `figure()`, followed by one (or multiple) `subplot()` calls.

Infrastructure

Housecleaning

A number of features that were deprecated in 1.2 or earlier, or have not been in a working state for a long time have been removed. Highlights include removing the Qt version 3 backends, and the FltkAgg and Emf backends. See [Changes in 1.3.x](#) for a complete list.

New setup script

matplotlib 1.3 includes an entirely rewritten setup script. We now ship fewer dependencies with the tarballs and installers themselves. Notably, `pytz`, `dateutil`, `pyparsing` and `six` are no longer included with matplotlib. You can either install them manually first, or let pip install them as dependencies along with matplotlib. It is now possible to not include certain subcomponents, such as the unit test data, in the install. See `setup.cfg.template` for more information.

XDG base directory support

On Linux, matplotlib now uses the XDG base directory specification to find the `matplotlibrc` configuration file. `matplotlibrc` should now be kept in `config/matplotlib`, rather than `matplotlib`. If your configuration is found in the old location, it will still be used, but a warning will be displayed.

Catch opening too many figures using pyplot

Figures created through `pyplot.figure` are retained until they are explicitly closed. It is therefore common for new users of matplotlib to run out of memory when creating a large series of figures in a loop without closing them.

matplotlib will now display a `RuntimeWarning` when too many figures have been opened at once. By default, this is displayed for 20 or more figures, but the exact number may be controlled using the `figure.max_open_warning` rcParam.

9.2.8 New in matplotlib 1.4

Thomas A. Caswell served as the release manager for the 1.4 release.

Table of Contents

- *New in matplotlib 1.4*
 - *New colormap*
 - *The nbagg backend*
 - *New plotting features*
 - *Date handling*
 - *Configuration (rcParams)*
 - *style package added*
 - *Backends*
 - *Text*
 - *Sphinx extensions*
 - *Legend and PathEffects documentation*
 - *Widgets*
 - *GAE integration*

Note: matplotlib 1.4 supports Python 2.6, 2.7, 3.3, and 3.4

New colormap

In heatmaps, a green-to-red spectrum is often used to indicate intensity of activity, but this can be problematic for the red/green colorblind. A new, colorblind-friendly colormap is now available at `matplotlib.cm.Wistia`. This colormap maintains the red/green symbolism while achieving deuteranopic legibility through brightness variations. See [here](#) for more information.

The nbagg backend

Phil Elson added a new backend, named “nbagg”, which enables interactive figures in a live IPython notebook session. The backend makes use of the infrastructure developed for the webagg backend, which itself gives standalone server backed interactive figures in the browser, however nbagg does not require a dedicated matplotlib server as all communications are handled through the IPython Comm machinery.

As with other backends nbagg can be enabled inside the IPython notebook with:

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

Once figures are created and then subsequently shown, they will placed in an interactive widget inside the notebook allowing panning and zooming in the same way as any other matplotlib backend. Because figures require a connection to the IPython notebook server for their interactivity, once the notebook is saved, each figure will be rendered as a static image - thus allowing non-interactive viewing of figures on services such as nbviewer.

New plotting features

Power-law normalization

Ben Gamari added a power-law normalization method, [*PowerNorm*](#). This class maps a range of values to the interval [0,1] with power-law scaling with the exponent provided by the constructor's `gamma` argument. Power law normalization can be useful for, e.g., emphasizing small populations in a histogram.

Fully customizable boxplots

Paul Hobson overhauled the `boxplot()` method such that it is now completely customizable in terms of the styles and positions of the individual artists. Under the hood, `boxplot()` relies on a new function (`boxplot_stats()`), which accepts any data structure currently compatible with `boxplot()`, and returns a list of dictionaries containing the positions for each element of the boxplots. Then a second method, `bxp()` is called to draw the boxplots based on the stats.

The `boxplot()` function can be used as before to generate boxplots from data in one step. But now the user has the flexibility to generate the statistics independently, or to modify the output of `boxplot_stats()` prior to plotting with `bxp()`.

Lastly, each artist (e.g., the box, outliers, cap, notches) can now be toggled on or off and their styles can be passed in through individual kwargs. See the examples: [*statistics example code: boxplot_demo.py*](#) and [*statistics example code: bxp_demo.py*](#)

Added a bool kwarg, `manage_xticks`, which if False disables the management of the ticks and limits on the x-axis by `bxp()`.

Support for datetime axes in 2d plots

Andrew Dawson added support for datetime axes to `contour()`, `contourf()`, `pcolormesh()` and `pcolor()`.

Support for additional spectrum types

Todd Jennings added support for new types of frequency spectrum plots: `magnitude_spectrum()`, `phase_spectrum()`, and `angle_spectrum()`, as well as corresponding functions in mlab.

He also added these spectrum types to `specgram()`, as well as adding support for linear scaling there (in addition to the existing dB scaling). Support for additional spectrum types was also added to `specgram()`.

He also increased the performance for all of these functions and plot types.

Support for detrending and windowing 2D arrays in mlab

Todd Jennings added support for 2D arrays in the `detrend_mean()`, `detrend_none()`, and `detrend()`, as well as adding `apply_window()` which support windowing 2D arrays.

Support for strides in mlab

Todd Jennings added some functions to mlab to make it easier to use numpy strides to create memory-efficient 2D arrays. This includes `stride_repeat()`, which repeats an array to create a 2D array, and `stride_windows()`, which uses a moving window to create a 2D array from a 1D array.

Formatter for new-style formatting strings

Added `FormatStrFormatterNewStyle` which does the same job as `FormatStrFormatter`, but accepts new-style formatting strings instead of printf-style formatting strings

Consistent grid sizes in streamplots

`streamplot()` uses a base grid size of 30x30 for both `density=1` and `density=(1, 1)`. Previously a grid size of 30x30 was used for `density=1`, but a grid size of 25x25 was used for `density=(1, 1)`.

Get a list of all tick labels (major and minor)

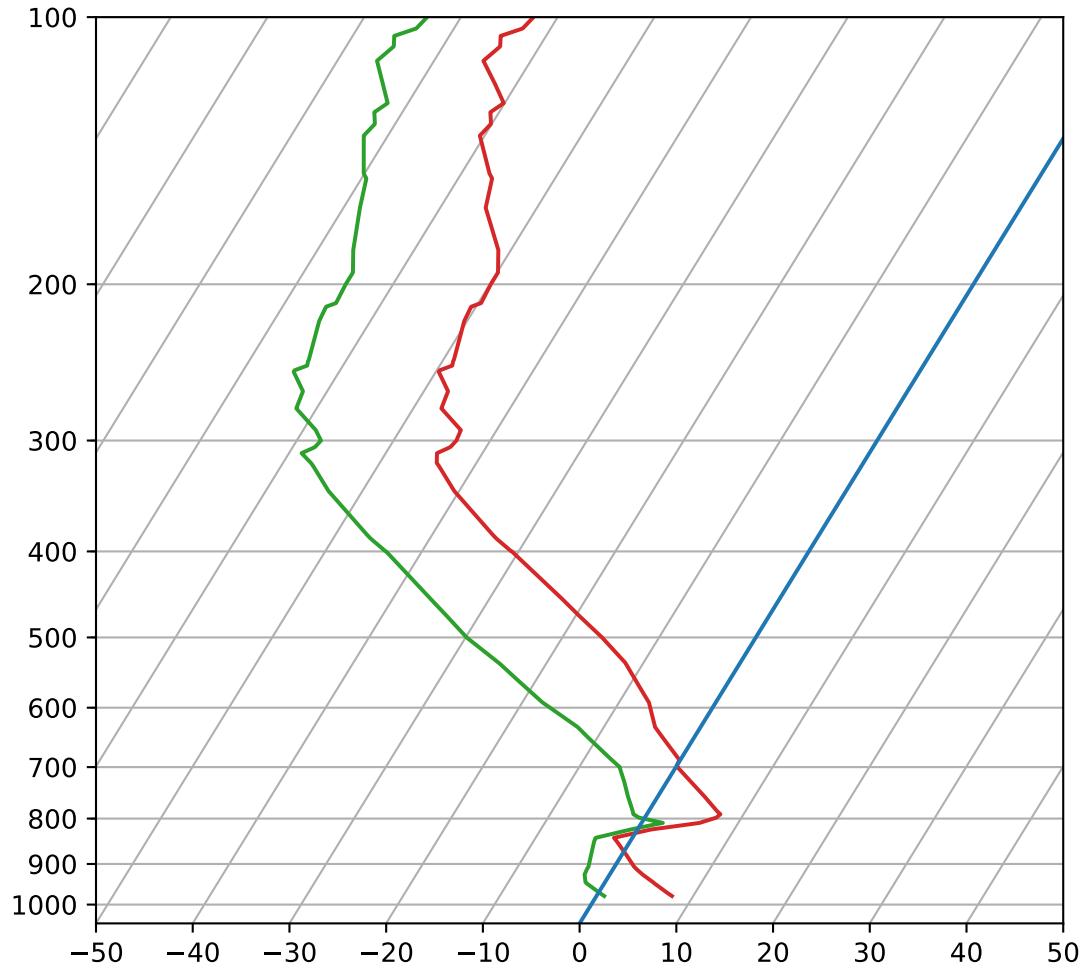
Added the kwarg ‘which’ to `get_xticklabels()`, `get_yticklabels()` and `get_ticklabels()`. ‘which’ can be ‘major’, ‘minor’, or ‘both’ select which ticks to return, like `set_ticks_position()`. If ‘which’ is None then the old behaviour (controlled by the bool `minor`).

Separate horizontal/vertical axes padding support in ImageGrid

The kwarg ‘axes_pad’ to `mpl_toolkits.axes_grid1.ImageGrid` can now be a tuple if separate horizontal/vertical padding is needed. This is supposed to be very helpful when you have a labelled legend next to every subplot and you need to make some space for legend’s labels.

Support for skewed transformations

The [Affine2D](#) gained additional methods `skew` and `skew_deg` to create skewed transformations. Additionally, matplotlib internals were cleaned up to support using such transforms in `Axes`. This transform is important for some plot types, specifically the Skew-T used in meteorology.



Support for specifying properties of wedge and text in pie charts.

Added the kwargs ‘`wedgeprops`’ and ‘`textprops`’ to `pie()` to accept properties for wedge and text objects in a pie. For example, one can specify `wedgeprops = {‘linewidth’:3}` to specify the width of the borders of the wedges in the pie. For more properties that the user can specify, look at the docs for the wedge and text objects.

Fixed the direction of errorbar upper/lower limits

Larry Bradley fixed the `errorbar()` method such that the upper and lower limits (`lolims`, `uplims`, `xlolims`, `xuplims`) now point in the correct direction.

More consistent add-object API for Axes

Added the Axes method `add_image` to put image handling on a par with artists, collections, containers, lines, patches, and tables.

Violin Plots

Per Parker, Gregory Kelsie, Adam Ortiz, Kevin Chan, Geoffrey Lee, Deokjae Donald Seo, and Taesu Terry Lim added a basic implementation for violin plots. Violin plots can be used to represent the distribution of sample data. They are similar to box plots, but use a kernel density estimation function to present a smooth approximation of the data sample used. The added features are:

`violin()` - Renders a violin plot from a collection of statistics. `violin_stats()` - Produces a collection of statistics suitable for rendering a violin plot. `violinplot()` - Creates a violin plot from a set of sample data. This method makes use of `violin_stats()` to process the input data, and `violin_stats()` to do the actual rendering. Users are also free to modify or replace the output of `violin_stats()` in order to customize the violin plots to their liking.

This feature was implemented for a software engineering course at the University of Toronto, Scarborough, run in Winter 2014 by Anya Tafliovich.

More markevery options to show only a subset of markers

Rohan Walker extended the `markevery` property in `Line2D`. You can now specify a subset of markers to show with an int, slice object, numpy fancy indexing, or float. Using a float shows markers at approximately equal display-coordinate-distances along the line.

Added size related functions to specialized Collections

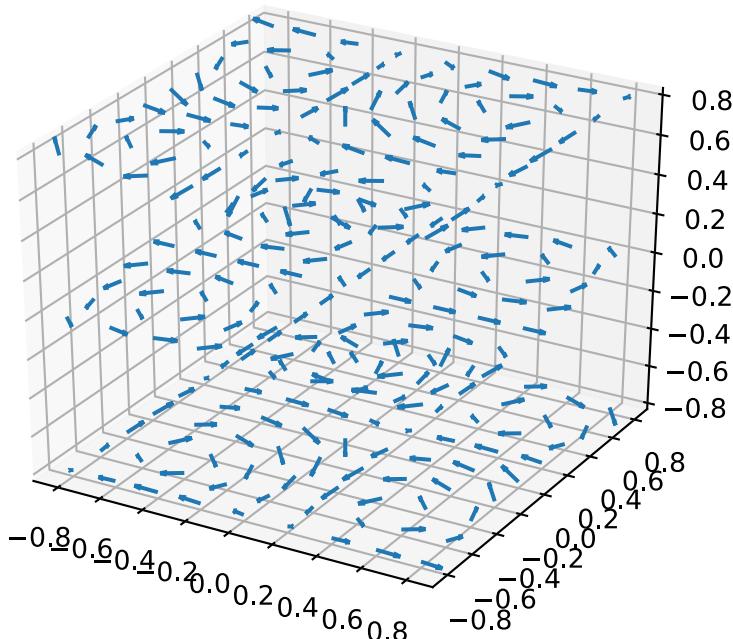
Added the `get_size` and `set_size` functions to control the size of elements of specialized collections (`AsteriskPolygonCollection` `BrokenBarHCollection` `CircleCollection` `PathCollection` `PolyCollection` `RegularPolyCollection` `StarPolygonCollection`).

Fixed the mouse coordinates giving the wrong theta value in Polar graph

Added code to `transform_non_affine()` to ensure that the calculated theta value was between the range of 0 and $2 * \pi$ since the problem was that the value can become negative after applying the direction and rotation to the theta calculation.

Simple quiver plot for mplot3d toolkit

A team of students in an *Engineering Large Software Systems* course, taught by Prof. Anya Tafliovich at the University of Toronto, implemented a simple version of a quiver plot in 3D space for the mplot3d toolkit as one of their term project. This feature is documented in [`quiver\(\)`](#). The team members are: Ryan Steve D'Souza, Victor B, xbtsw, Yang Wang, David, Caradec Bisesar and Vlad Vassilovski.



polar-plot r-tick locations

Added the ability to control the angular position of the r-tick labels on a polar plot via `set_rlabel_position()`.

Date handling

n-d array support for date conversion

Andrew Dawson added support for n-d array handling to `matplotlib.dates.num2date()`, `matplotlib.dates.date2num()` and `matplotlib.dates.datestr2num()`. Support is also added to the unit conversion interfaces `matplotlib.dates.DateConverter` and `matplotlib.units.Registry`.

Configuration (rcParams)

`savefig.transparent` added

Controls whether figures are saved with a transparent background by default. Previously `savefig` always defaulted to a non-transparent background.

`axes.titleweight`

Added rcParam to control the weight of the title

`axes.formatter.useoffset` added

Controls the default value of `useOffset` in `ScalarFormatter`. If True and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. If False then the full number will be formatted in all conditions.

`nbagg.transparent` added

Controls whether nbagg figures have a transparent background. `nbagg.transparent` is True by default.

XDG compliance

Matplotlib now looks for configuration files (both rcparsms and style) in XDG compliant locations.

`style package` added

You can now easily switch between different styles using the new `style` package:

```
>>> from matplotlib import style  
>>> style.use('dark_background')
```

Subsequent plots will use updated colors, sizes, etc. To list all available styles, use:

```
>>> print style.available
```

You can add your own custom `<style name>.mplstyle` files to `~/ .matplotlib/stylelib` or call `use` with a URL pointing to a file with `matplotlibrc` settings.

Note that this is an experimental feature, and the interface may change as users test out this new feature.

Backends

Qt5 backend

Martin Fitzpatrick and Tom Badran implemented a Qt5 backend. The differences in namespace locations between Qt4 and Qt5 was dealt with by shimming Qt4 to look like Qt5, thus the Qt5 implementation is the primary implementation. Backwards compatibility for Qt4 is maintained by wrapping the Qt5 implementation.

The Qt5Agg backend currently does not work with IPython's %matplotlib magic.

The 1.4.0 release has a known bug where the toolbar is broken. This can be fixed by:

```
cd path/to/installed/matplotlib
wget https://github.com/matplotlib/matplotlib/pull/3322.diff
# unix2dos 3322.diff (if on windows to fix line endings)
patch -p2 < 3322.diff
```

Qt4 backend

Rudolf Höfler changed the appearance of the subplottool. All sliders are vertically arranged now, buttons for tight layout and reset were added. Furthermore, the subplottool is now implemented as a modal dialog. It was previously a QMainWindow, leaving the SPT open if one closed the plot window.

In the figure options dialog one can now choose to (re-)generate a simple automatic legend. Any explicitly set legend entries will be lost, but changes to the curves' label, linestyle, et cetera will now be updated in the legend.

Interactive performance of the Qt4 backend has been dramatically improved under windows.

The mapping of key-signals from Qt to values matplotlib understands was greatly improved (For both Qt4 and Qt5).

Cairo backends

The Cairo backends are now able to use the `cairocffi` bindings which are more actively maintained than the `pycairo` bindings.

Gtk3Agg backend

The Gtk3Agg backend now works on Python 3.x, if the `cairocffi` bindings are installed.

PDF backend

Added context manager for saving to multi-page PDFs.

Text

Text URLs supported by SVG backend

The `svg` backend will now render `Text` objects' url as a link in output SVGs. This allows one to make clickable text in saved figures using the `url` kwarg of the `Text` class.

Anchored sidebar font

Added the `fontproperties` kwarg to `AnchoredSizeBar` to control the font properties.

Sphinx extensions

The `:context:` directive in the `plot_directive` Sphinx extension can now accept an optional `reset` setting, which will cause the context to be reset. This allows more than one distinct context to be present in documentation. To enable this option, use `:context: reset` instead of `:context:` any time you want to reset the context.

Legend and PathEffects documentation

The `Legend guide` and `Path effects guide` have both been updated to better reflect the full potential of each of these powerful features.

Widgets

Span Selector

Added an option `span_stays` to the `SpanSelector` which makes the selector rectangle stay on the axes after you release the mouse.

GAE integration

Matplotlib will now run on google app engine.

9.2.9 New in matplotlib 1.5

Table of Contents

- *New in matplotlib 1.5*
 - *Interactive OO usage*

- Working with labeled data like pandas DataFrames
- Added axes.prop_cycle key to rcParams
- New Colormaps
- Styles
- Backends
- Configuration (rcParams)
- Widgets
- New plotting features
- ToolManager
- cbook.is_sequence_of_strings recognizes string objects
- New close-figs argument for plot directive
- Support for URL string arguments to imread
- Display hook for animations in the IPython notebook
- Prefixed pkg-config for building

Note: matplotlib 1.5 supports Python 2.7, 3.4, and 3.5

Interactive OO usage

All Artists now keep track of if their internal state has been changed but not reflected in the display ('stale') by a call to draw. It is thus possible to pragmatically determine if a given Figure needs to be re-drawn in an interactive session.

To facilitate interactive usage a draw_all method has been added to pyplot which will redraw all of the figures which are 'stale'.

To make this convenient for interactive use matplotlib now registers a function either with IPython's 'post_execute' event or with the displayhook in the standard python REPL to automatically call plt.draw_all just before control is returned to the REPL. This ensures that the draw command is deferred and only called once.

The upshot of this is that for interactive backends (including %matplotlib notebook) in interactive mode (with plt.ion())

```
In [1]: import matplotlib.pyplot as plt  
In [2]: fig, ax = plt.subplots()  
In [3]: ln, = ax.plot([0, 1, 4, 9, 16])
```

```
In [4]: plt.show()
```

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

will automatically update the plot to be green. Any subsequent modifications to the `Artist` objects will do likewise.

This is the first step of a larger consolidation and simplification of the pyplot internals.

Working with labeled data like pandas DataFrames

Plot methods which take arrays as inputs can now also work with labeled data and unpack such data.

This means that the following two examples produce the same plot:

Example

```
df = pandas.DataFrame({ "var1": [1,2,3,4,5,6], "var2": [1,2,3,4,5,6] })
plt.plot(df["var1"], df["var2"])
```

Example

```
plt.plot("var1", "var2", data=df)
```

This works for most plotting methods, which expect arrays/sequences as inputs. `data` can be anything which supports `__getitem__` (`dict`, `pandas.DataFrame`, `h5py`, ...) to access array like values with string keys.

In addition to this, some other changes were made, which makes working with labeled data (ex `pandas.Series`) easier:

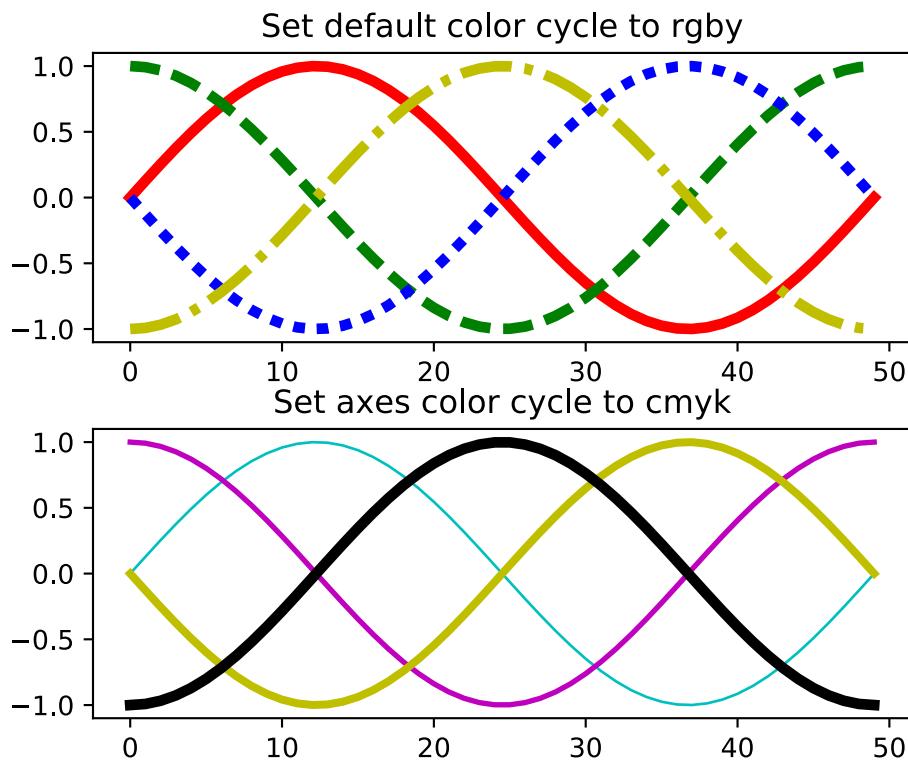
- For plotting methods with `label` keyword argument, one of the data inputs is designated as the label source. If the user does not supply a `label` that value object will be introspected for a label, currently by looking for a `name` attribute. If the value object does not have a `name` attribute but was specified by as a key into the `data` kwarg, then the key is used. In the above examples, this results in an implicit `label="var2"` for both cases.
- `plot()` now uses the index of a `Series` instead of `np.arange(len(y))`, if no `x` argument is supplied.

Added `axes.prop_cycle` key to `rcParams`

This is a more generic form of the now-deprecated `axes.color_cycle` param. Now, we can cycle more than just colors, but also linestyles, hatches, and just about any other artist property. Cycler notation is used for defining property cycles. Adding cyclers together will be like you are `zip()`-ing together two or more property cycles together:

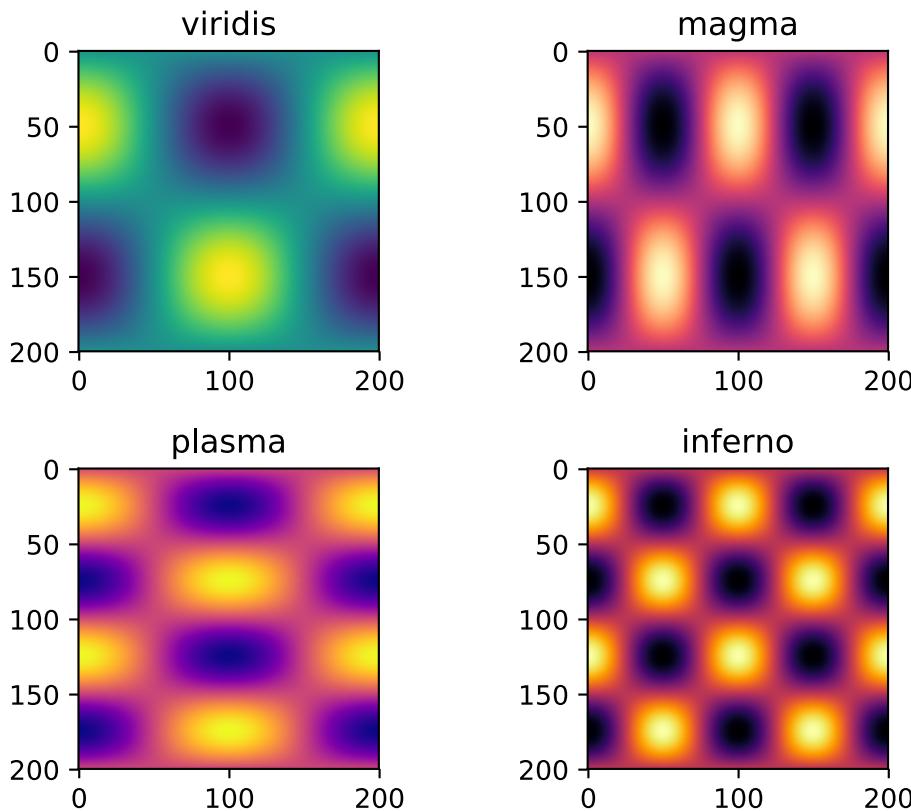
```
axes.prop_cycle: cycler('color', 'rgb') + cycler('lw', [1, 2, 3])
```

You can even multiply cyclers, which is like using `itertools.product()` on two or more property cycles. Remember to use parentheses if writing a multi-line `prop_cycle` parameter.



New Colormaps

All four of the colormaps proposed as the new default are available as 'viridis' (the new default in 2.0), 'magma', 'plasma', and 'inferno'



Styles

Several new styles have been added, including many styles from the Seaborn project. Additionally, in order to prep for the upcoming 2.0 style-change release, a ‘classic’ and ‘default’ style has been added. For this release, the ‘default’ and ‘classic’ styles are identical. By using them now in your scripts, you can help ensure a smooth transition during future upgrades of matplotlib, so that you can upgrade to the snazzy new defaults when you are ready!

```
import matplotlib.style  
matplotlib.style.use('classic')
```

The ‘default’ style will give you matplotlib’s latest plotting styles:

```
matplotlib.style.use('default')
```

Backends

New backend selection

The environment variable `MPLBACKEND` can now be used to set the matplotlib backend.

wx backend has been updated

The wx backend can now be used with both wxPython classic and [Phoenix](#).

wxPython classic has to be at least version 2.8.12 and works on Python 2.x. As of May 2015 no official release of wxPython Phoenix is available but a current snapshot will work on Python 2.7+ and 3.4+.

If you have multiple versions of wxPython installed, then the user code is responsible setting the wxPython version. How to do this is explained in the comment at the beginning of the example `examples/user_interfacesembedding_in_wx2.py`.

Configuration (rcParams)

Some parameters have been added, others have been improved.

Parameter	Description
{x,y}axis. labelpad	mplot3d now respects these parameters
axes. labelpad	Default space between the axis and the label
errorbar. capsize	Default length of end caps on error bars
{x,y}tick. minor. visible	Default visibility of minor x/y ticks
legend. framealpha	Default transparency of the legend frame box
legend. facecolor	Default facecolor of legend frame box (or 'inherit' from axes.facecolor)
legend. edgecolor	Default edgecolor of legend frame box (or 'inherit' from axes.edgecolor)
figure. titlesize	Default font size for figure subtitles
figure. titleweight	Default font weight for figure subtitles
image. composite_image	Whether a vector graphics backend should composite several images into a single image or not when saving. Useful when needing to edit the files further in Inkscape or other programs.
markers. fillstyle	Default fillstyle of markers. Possible values are 'full' (the default), 'left', 'right', 'bottom', 'top' and 'none'
toolbar	Added 'toolmanager' as a valid value, enabling the experimental ToolManager feature.

Widgets

Active state of Selectors

All selectors now implement `set_active` and `get_active` methods (also called when accessing the `active` property) to properly update and query whether they are active.

Moved `ignore`, `set_active`, and `get_active` methods to base class `Widget`

Pushes up duplicate methods in child class to parent class to avoid duplication of code.

Adds enable/disable feature to MultiCursor

A `MultiCursor` object can be disabled (and enabled) after it has been created without destroying the object. Example:

```
multi_cursor.active = False
```

Improved RectangleSelector and new EllipseSelector Widget

Adds an `interactive` keyword which enables visible handles for manipulating the shape after it has been drawn.

Adds keyboard modifiers for:

- Moving the existing shape (default key = ‘space’)
- Making the shape square (default ‘shift’)
- Make the initial point the center of the shape (default ‘control’)
- Square and center can be combined

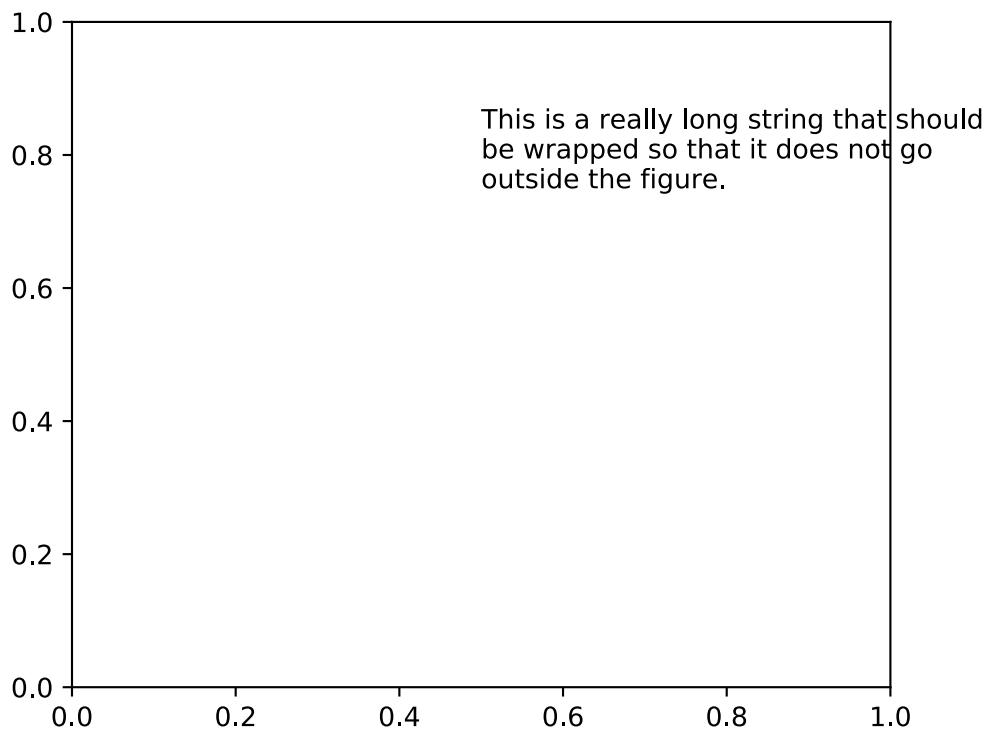
Allow Artists to Display Pixel Data in Cursor

Adds `get_pixel_data` and `format_pixel_data` methods to artists which can be used to add zdata to the cursor display in the status bar. Also adds an implementation for Images.

New plotting features

Auto-wrapping Text

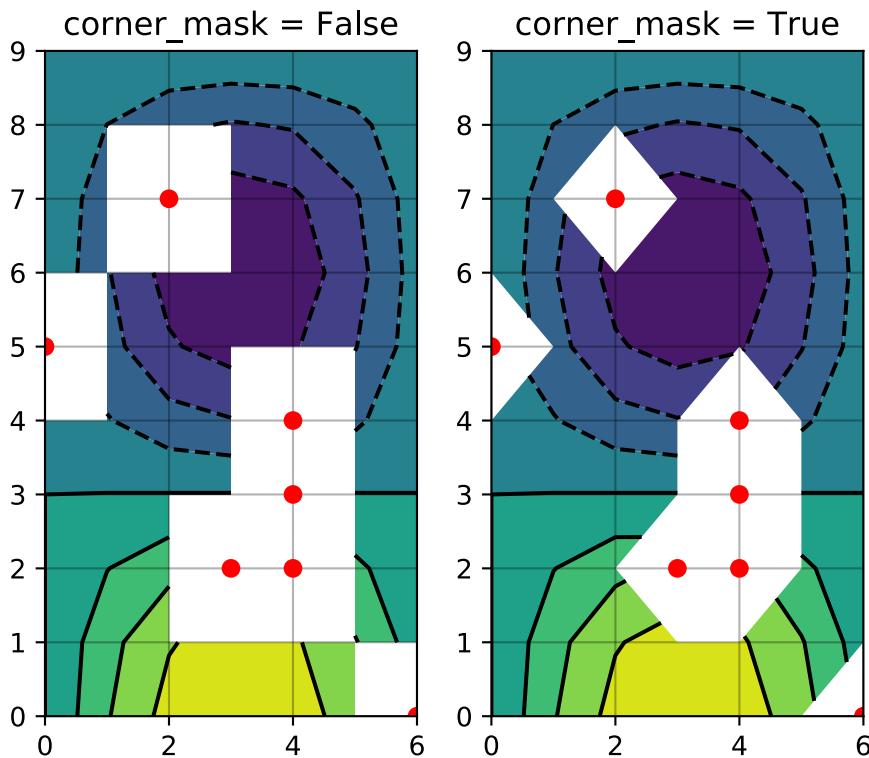
Added the keyword argument “wrap” to `Text`, which automatically breaks long lines of text when being drawn. Works for any rotated text, different modes of alignment, and for text that are either labels or titles. This breaks at the `Figure`, not `Axes` edge.



This is a really long string that should be wrapped so that it does not go outside the figure.

Contour plot corner masking

Ian Thomas rewrote the C++ code that calculates contours to add support for corner masking. This is controlled by a new keyword argument `corner_mask` in the functions `contour()` and `contourf()`. The previous behaviour, which is now obtained using `corner_mask=False`, was for a single masked point to completely mask out all four quads touching that point. The new behaviour, obtained using `corner_mask=True`, only masks the corners of those quads touching the point; any triangular corners comprising three unmasked points are contoured as usual. If the `corner_mask` keyword argument is not specified, the default value is taken from rcParams.



Mostly unified linestyles for Line2D, Patch and Collection

The handling of linestyles for Lines, Patches and Collections has been unified. Now they all support defining linestyles with short symbols, like `--`, as well as with full names, like `"dashed"`. Also the definition using a dash pattern (`(0., [3., 3.])`) is supported for all methods using Line2D, Patch or Collection.

Legend marker order

Added ability to place the label before the marker in a legend box with `markerfirst` keyword

Support for legend for PolyCollection and stackplot

Added a `legend_handler` for `PolyCollection` as well as a `labels` argument to `stackplot()`.

Support for alternate pivots in mplot3d quiver plot

Added a `pivot` kwarg to `quiver()` that controls the pivot point around which the quiver line rotates. This also determines the placement of the arrow head along the quiver line.

Logit Scale

Added support for the ‘logit’ axis scale, a nonlinear transformation

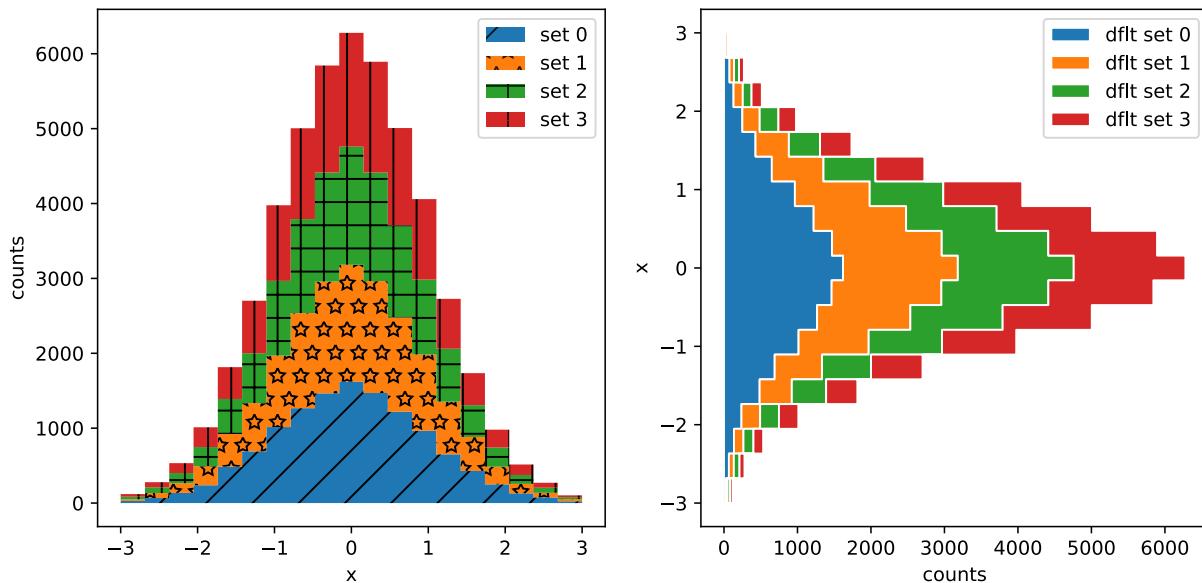
$$x \rightarrow \log 10(x/(1 - x)) \quad (9.1)$$

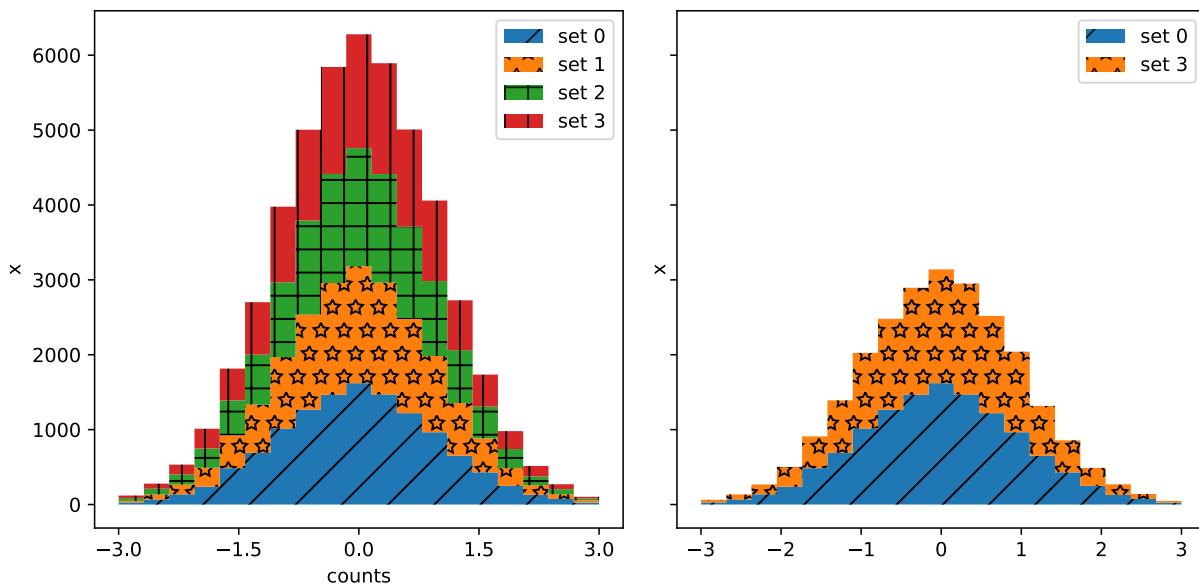
for data between 0 and 1 excluded.

Add step kwargs to fill_between

Added `step` kwarg to `Axes.fill_between` to allow to fill between lines drawn using the ‘step’ draw style. The values of `step` match those of the `where` kwarg of `Axes.step`. The asymmetry of the kwargs names is not ideal, but `Axes.fill_between` already has a `where` kwarg.

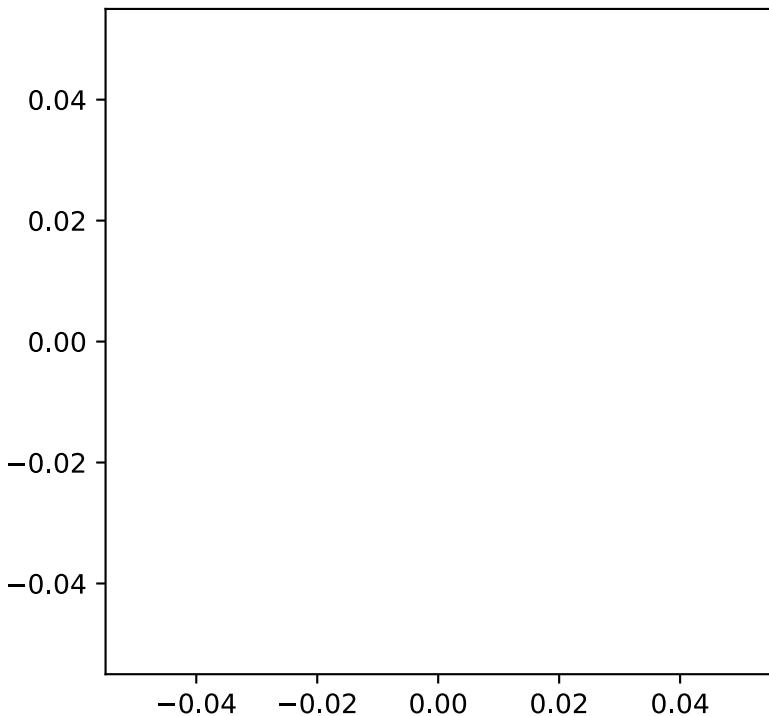
This is particularly useful for plotting pre-binned histograms.





Square Plot

Implemented square plots feature as a new parameter in the axis function. When argument ‘square’ is specified, equal scaling is set, and the limits are set such that `xmax-xmin == ymax-ymin`.



Updated `figimage` to take optional `resize` parameter

Added the ability to plot simple 2D-Array using `plt.figimage(X, resize=True)`. This is useful for plotting simple 2D-Array without the Axes or whitespacing around the image.



Updated `Figure.savefig()` can now use figure's dpi

Added support to save the figure with the same dpi as the figure on the screen using `dpi='figure'`.

Example:

```
f = plt.figure(dpi=25)          # dpi set to 25
S = plt.scatter([1,2,3],[4,5,6])
f.savefig('output.png', dpi='figure')    # output savefig dpi set to 25 (same as figure)
```

Updated Table to control edge visibility

Added the ability to toggle the visibility of lines in Tables. Functionality added to the `pyplot.table()` factory function under the keyword argument “edges”. Values can be the strings “open”, “closed”, “horizon-

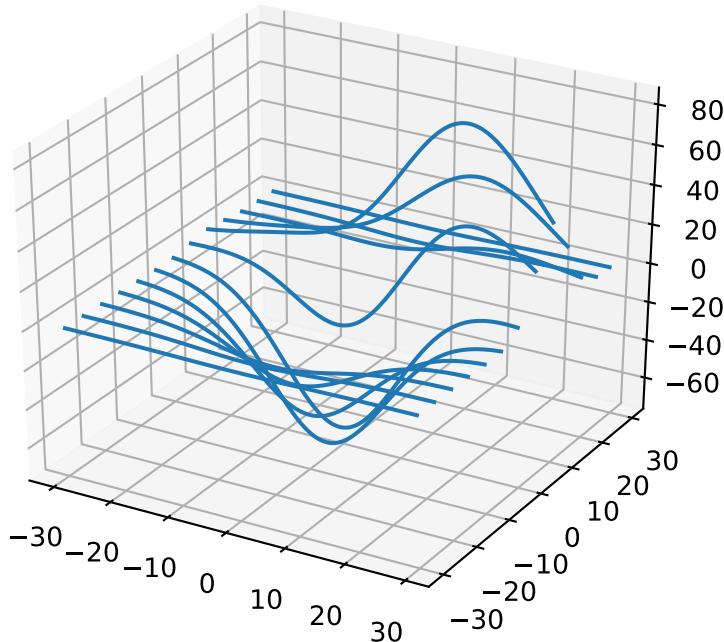
tal”, “vertical” or combinations of the letters “L”, “R”, “T”, “B” which represent left, right, top, and bottom respectively.

Example:

```
table(..., edges="open") # No line visible
table(..., edges="closed") # All lines visible
table(..., edges="horizontal") # Only top and bottom lines visible
table(..., edges="LT") # Only left and top lines visible.
```

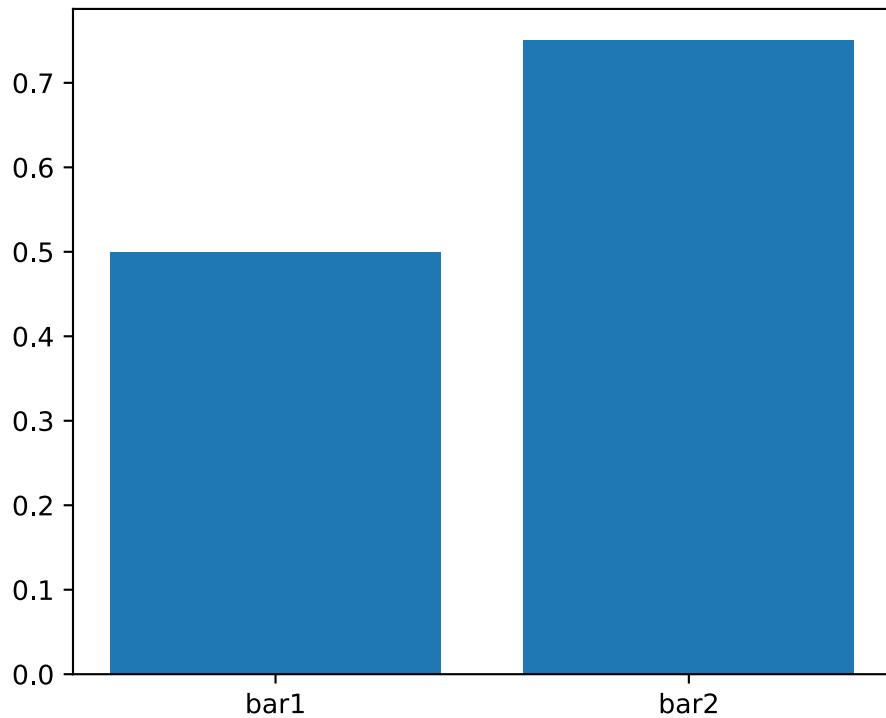
Zero r/cstride support in plot_wireframe

Adam Hughes added support to mplot3d’s plot_wireframe to draw only row or column line plots.



Plot bar and barh with labels

Added kwarg "tick_label" to bar and barh to support plotting bar graphs with a text label for each bar.



Added center and frame kwargs to pie

These control where the center of the pie graph are and if the Axes frame is shown.

Fixed 3D filled contour plot polygon rendering

Certain cases of 3D filled contour plots that produce polygons with multiple holes produced improper rendering due to a loss of path information between `PolyCollection` and `Poly3DCollection`. A function `set_verts_and_codes()` was added to allow path information to be retained for proper rendering.

Dense colorbars are rasterized

Vector file formats (pdf, ps, svg) are efficient for many types of plot element, but for some they can yield excessive file size and even rendering artifacts, depending on the renderer used for screen display. This is a problem for colorbars that show a large number of shades, as is most commonly the case. Now, if a colorbar is showing 50 or more colors, it will be rasterized in vector backends.

DateFormatter strftime

`strftime` method will format a `datetime.datetime` object with the format string passed to the formatter's constructor. This method accepts datetimes with years before 1900, unlike `datetime.datetime.strftime()`.

Artist-level {get,set}_usetex for text

Add `{get, set}_usetex` methods to `Text` objects which allow artist-level control of LaTeX rendering vs the internal mathtex rendering.

ax.remove() works as expected

As with artists added to an `Axes`, `Axes` objects can be removed from their figure via `remove()`.

API Consistency fix within Locators set_params() function

`set_params()` function, which sets parameters within a `Locator` type instance, is now available to all Locator types. The implementation also prevents unsafe usage by strictly defining the parameters that a user can set.

To use, call `set_params()` on a Locator instance with desired arguments:

```
loc = matplotlib.ticker.LogLocator()
# Set given attributes for loc.
loc.set_params(numticks=8, numdecs=8, subs=[2.0], base=8)
# The below will error, as there is no such parameter for LogLocator
# named foo
# loc.set_params(foo='bar')
```

Date Locators

Date Locators (derived from `DateLocator`) now implement the `tick_values()` method. This is expected of all Locators derived from Locator.

The Date Locators can now be used easily without creating axes

```
from datetime import datetime
from matplotlib.dates import YearLocator
t0 = datetime(2002, 10, 9, 12, 10)
tf = datetime(2005, 10, 9, 12, 15)
loc = YearLocator()
values = loc.tick_values(t0, tf)
```

OffsetBoxes now support clipping

Artists draw onto objects of type `OffsetBox` through `DrawingArea` and `TextArea`. The `TextArea` calculates the required space for the text and so the text is always within the bounds, for this nothing has changed.

However, `DrawingArea` acts as a parent for zero or more Artists that draw on it and may do so beyond the bounds. Now child Artists can be clipped to the bounds of the `DrawingArea`.

OffsetBoxes now considered by tight_layout

When `tight_layout()` or `Figure.tight_layout()` or `GridSpec.tight_layout()` is called, `OffsetBoxes` that are anchored outside the axes will not get chopped out. The `OffsetBoxes` will also not get overlapped by other axes in case of multiple subplots.

Per-page pdf notes in multi-page pdfs (PdfPages)

Add a new method `attach_note()` to the `PdfPages` class, allowing the attachment of simple text notes to pages in a multi-page pdf of figures. The new note is visible in the list of pdf annotations in a viewer that has this facility (Adobe Reader, OSX Preview, Skim, etc.). Per default the note itself is kept off-page to prevent it to appear in print-outs.

`PdfPages.attach_note` needs to be called before `savefig()` in order to be added to the correct figure.

Updated `fignum_exists` to take figure name

Added the ability to check the existence of a figure using its name instead of just the figure number. Example:

```
figure('figure')
fignum_exists('figure') #true
```

ToolManager

Federico Ariza wrote the new `ToolManager` that comes as replacement for `NavigationToolbar2`

`ToolManager` offers a new way of looking at the user interactions with the figures. Before we had the `NavigationToolbar2` with its own tools like `zoom/pan/home/save/...` and also we had the shortcuts like `yscale/grid/quit/....` `Toolmanager` relocate all those actions as Tools (located in `backend_tools`), and defines a way to access/trigger/reconfigure them.

The Toolbars are replaced for ToolContainers that are just GUI interfaces to trigger the tools. But don't worry the default backends include a ToolContainer called `toolbar`

Note: At the moment, we release this primarily for feedback purposes and should be treated as experimental until further notice as API changes will occur. For the moment the `ToolManager` works only with the GTK3 and Tk backends. Make sure you use one of those. Port for the rest of the backends is comming soon.

To activate the ToolManager include the following at the top of your file

```
>>> matplotlib.rcParams['toolbar'] = 'toolmanager'
```

Interact with the ToolContainer

The most important feature is the ability to easily reconfigure the ToolContainer (aka toolbar). For example, if we want to remove the “forward” button we would just do.

```
>>> fig.canvas.manager.toolmanager.remove_tool('forward')
```

Now if you want to programmatically trigger the “home” button

```
>>> fig.canvas.manager.toolmanager.trigger_tool('home')
```

New Tools for ToolManager

It is possible to add new tools to the ToolManager

A very simple tool that prints “You’re awesome” would be:

```
from matplotlib.backend_tools import ToolBase
class AwesomeTool(ToolBase):
    def trigger(self, *args, **kwargs):
        print("You're awesome")
```

To add this tool to ToolManager

```
>>> fig.canvas.manager.toolmanager.add_tool('Awesome', AwesomeTool)
```

If we want to add a shortcut (“d”) for the tool

```
>>> fig.canvas.manager.toolmanager.update_keymap('Awesome', 'd')
```

To add it to the toolbar inside the group ‘foo’

```
>>> fig.canvas.manager.toolbar.add_tool('Awesome', 'foo')
```

There is a second class of tools, “Toggleable Tools”, this are almost the same as our basic tools, just that belong to a group, and are mutually exclusive inside that group. For tools derived from ToolToggleBase there are two basic methods `enable` and `disable` that are called automatically whenever it is toggled.

A full example is located in [user_interfaces example code: toolmanager.py](#)

cbook.is_sequence_of_strings recognizes string objects

This is primarily how pandas stores a sequence of strings

```
import pandas as pd
import matplotlib.cbook as cbook

a = np.array(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(a)) # True

a = np.array(['a', 'b', 'c'], dtype=object)
print(cbook.is_sequence_of_strings(a)) # True

s = pd.Series(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(s)) # True
```

Previously, the last two prints returned false.

New `close-figs` argument for plot directive

Matplotlib has a sphinx extension `plot_directive` that creates plots for inclusion in sphinx documents. Matplotlib 1.5 adds a new option to the plot directive - `close-figs` - that closes any previous figure windows before creating the plots. This can help avoid some surprising duplicates of plots when using `plot_directive`.

Support for URL string arguments to `imread`

The `imread()` function now accepts URL strings that point to remote PNG files. This circumvents the generation of a `HTTPResponse` object directly.

Display hook for animations in the IPython notebook

`Animation` instances gained a `_repr_html_` method to support inline display of animations in the notebook. The method used to display is controlled by the `animation.html` rc parameter, which currently supports values of `none` and `html5`. `none` is the default, performing no display. `html5` converts the animation to an h264 encoded video, which is embedded directly in the notebook.

Users not wishing to use the `_repr_html_` display hook can also manually call the `to_html5_video` method to get the HTML and display using IPython's HTML display class:

```
from IPython.display import HTML
HTML(anim.to_html5_video())
```

Prefixed pkg-config for building

Handling of `pkg-config` has been fixed in so far as it is now possible to set it using the environment variable `PKG_CONFIG`. This is important if your toolchain is prefixed. This is done in a similar way as setting `CC` or `CXX` before building. An example follows.

```
export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```

GITHUB STATS

GitHub stats for 2015/10/29 - 2017/01/16 (tag: v1.5.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 889 issues and merged 732 pull requests.

The following 233 authors contributed 3210 commits.

- 4over7
- Abdealijk
- Acanthostega
- Adam Williamson
- Adrien Chardon
- Adrien F. Vincent
- Alan Bernstein
- Alberto
- alcinos
- Alex Rothberg
- Alexis Bienvenüe
- Ali Uneri
- Alvaro Sanchez
- alvarosg
- AndersonDaniel
- Andreas Hilboll
- Andreas Mayer
- aneda
- Anton Akhmerov
- Antony Lee

- Arvind
- Ben Root
- Benedikt Daurer
- Benedikt J. Daurer
- Benjamin Berg
- Benjamin Congdon
- BHT
- Bruno Zohreh
- Cameron Davidson-Pilon
- Cameron Fackler
- Chen Karako
- Chris Holdgraf
- Christian Stade-Schuldt
- Christoph Deil
- Christoph Gohlke
- Cimarron Mittelsteadt
- CJ Carey
- Conner R. Phillips
- DaCoEx
- Dan Hickstein
- Daniel C. Marcu
- danielballan
- Danny Hermes
- David A
- David Kent
- David Stansby
- deeenes
- Devashish Deshpande
- Diego Mora Cespedes
- Dietrich Brunn
- dlmccaffrey
- Dora Fraeman

- DoriekeMG
- Drew J. Sonne
- Dylan Evans
- 5. (a) Patrick Bos
- Egor Panfilov
- Elliott Sales de Andrade
- Elvis Stansvik
- endolith
- Eric Dill
- Eric Firing
- Eric Larson
- Eugene Yurtsev
- Fabian-Robert Stöter
- Federico Ariza
- fibersnet
- Florencia Noriega
- Florian Le Bourdais
- Francoise Provencher
- Frank Yu
- Gaute Hope
- gcallah
- Geoffrey Spear
- gepcel
- greg-roper
- Grillard
- Guillermo Breto
- Hakan Kucukdereli
- hannah
- Hans Moritz Günther
- Hassan Kibirige
- Hastings Greer
- Heiko Oberdiek

- Henning Pohl
- Herbert Kruitbosch
- Ian Thomas
- Ilia Kurenkov
- ImSoErgodic
- Isaac Schwabacher
- Isaac Slavitt
- 10. Goutin
- Jaap Versteegh
- Jacob McDonald
- Jae-Joon Lee
- James A. Bednar
- Jan Schlüter
- Jan Schulz
- Jarrod Millman
- Jason King
- Jason Zheng
- Jeffrey Hokanson @ Loki
- Jens Hedegaard Nielsen
- John Vandenberg
- JojoBoulix
- jonchar
- Joseph Fox-Rabinovitz
- Joseph Jon Booker
- Jouni K. Seppänen
- Juan Nunez-Iglesias
- juan.gonzalez
- Julia Sprenger
- Julian Mehne
- Julian V. Modesto
- Julien Lhermitte
- Julien Schueller

- Jun Tan
- Kacper Kowalik (Xarthisius)
- Kanwar245
- Kevin Keating
- khyox
- Kjartan Myrdal
- Klara Gerlei
- klaus
- klonuo
- Kristen M. Thyng
- Kyle Bridgemohansingh
- Kyler Brown
- Laptop11_ASPP2016
- lboogaard
- Leo Singer
- lspvic
- Luis Pedro Coelho
- lzkelley
- Magnus Nord
- mamrehn
- Manuel Jung
- Matt Hancock
- Matthew Brett
- Matthias Bussonnier
- Matthias Lüthi
- Maximilian Albert
- Maximilian Maahn
- Mher Kazandjian
- Michael Droettboom
- Michiel de Hoon
- Mike Henninger
- Mike Jarvis

- MinRK
- mlub
- mobando
- muahah
- myyc
- Naoya Kanai
- Nathan Goldbaum
- Nathan Musoke
- nbrunett
- Nelle Varoquaux
- nepix32
- Nicolas P. Rougier
- Nicolas Tessore
- Nikita Kniazev
- Nils Werner
- OceanWolf
- Orso Meneghini
- Pankaj Pandey
- Paul Ganssle
- Paul Hobson
- Paul Ivanov
- Paul Kirow
- Paul Romano
- Pete Huang
- Pete Peterson
- Peter Iannucci
- Peter Mortensen
- Peter Würtz
- Petr Danecek
- Phil Elson
- Phil Ruffwind
- Pierre de Buyl

- productivememberofsociety666
- Przemysław Dąbek
- Qingpeng “Q.P.” Zhang
- Ramiro Gómez
- Randy Olson
- Rishikesh
- Robin Dunn
- Robin Wilson
- Rui Lopes
- Ryan May
- RyanPan
- Salganos
- Salil Vanvari
- Samson
- Samuel St-Jean
- Sander
- scls19fr
- Scott Howard
- scott-vsi
- Sebastian Raschka
- Sebastián Vanrell
- Seraphim Alvanides
- Simon Gibbons
- Stefan Pfenninger
- Stephan Erb
- Sterling Smith
- Steven Sylvester
- Steven Tilley
- Tadeo Corradi
- Terrence J. Katzenbaer
- The Gitter Badger
- Thomas A Caswell

- Thomas Hisch
- Thomas Robitaille
- Thorsten Liebig
- Till Stensitzki
- tmdavison
- tomoemon
- Trish Gillett-Kawamoto
- Truong Pham
- u55
- ultra-andy
- Valentin Schmidt
- Victor Zabalza
- vraelvrangr
- Víctor Zabalza
- Warren Weckesser
- Wieland Hoffmann
- Will Silva
- William Granados
- Xufeng Wang
- Zbigniew Jędrzejewski-Szmeek
- Zohreh

GitHub issues and pull requests:

Pull Requests (732):

- PR #7845: Fixed bug with default parameters NFFT and nooverlap in specgram()
- PR #7800: DOC: explain non-linear scales and imshow (closes #7661)
- PR #7639: Enh color names
- PR #7829: MAINT tests should not use relative imports
- PR #7828: MAINT added early checks for dependencies for doc building
- PR #7424: Numpy Doc Format
- PR #7821: DOC: Changes to screenshots plots.
- PR #7644: Allow scalar height for plt.bar
- PR #7838: Merge v2.x

- PR #7823: MAINT matplotlib -> Matplotlib
- PR #7833: Deprecate unused verification code.
- PR #7827: [MRG+1] Cast stackplot input to float when required.
- PR #7834: Remove deprecated get_example_data.
- PR #7826: Remove invalid dimension checking in axes_rgb.
- PR #7831: Function wrapper for examples/api/two_scales.py
- PR #7801: [MRG+1] Add short-circuit return to matplotlib.artist.setp if input is length 0
- PR #7740: Beautified frontpage plots and two pylab examples
- PR #7730: Fixed GraphicsContextBase linestyle getter
- PR #7747: Update qhull to 2015.2
- PR #7645: Clean up stock sample data.
- PR #7753: Clarify the uses of whiskers float parameter.
- PR #7765: TST: Clean up figure tests
- PR #7729: For make raw_input compatible with python3
- PR #7783: Raise exception if negative height or width is passed to axes()
- PR #7727: [MRG+1] DOC: Fix invalid nose argument in testing.rst
- PR #7731: Check origin when saving image to PNG
- PR #7782: Fix some more integer type inconsistencies in Freetype code
- PR #7781: Fix integer types for font metrics in PyGlyph class
- PR #7791: Use reliable int type for mesh size in draw_quad_mesh (#7788)
- PR #7796: Only byte-swap 16-bit PNGs on little-endian (#7792)
- PR #7794: Ignore images that doc build produces
- PR #7790: Adjust markdown and text in ISSUE_TEMPLATE.md
- PR #7773: Fix more invalid escapes sequences.
- PR #7769: [MRG+1] Remove redundant pep8 entry in .travis.yml.
- PR #7760: DOC: Correct subplot() doc
- PR #7768: Convert unicode index to long, not int, in get_char_index
- PR #7770: BUG: improve integer step selection in MaxNLocator
- PR #7766: Invalid escape sequences are deprecated in Py3.6.
- PR #7758: fix axes barh default align option document
- PR #7749: DOC: Sync keyboard shortcuts for fullscreen toggle
- PR #7757: By default, don't include tests in binary distributions.

- PR #7762: DOC: Fix finance depr docs to point to mpl_finance
- PR #7737: Ensure that pyenv command is in a literal block
- PR #7732: Add rcsetup_api.rst, fix typo for rcsetup.cycler
- PR #7726: FIX: Clean up in the new quiverkey test; make new figs in scale tests
- PR #7620: Add warning context
- PR #7719: Add angle kwarg to quiverkey
- PR #7701: DOC: Add bug report reqs and template to contributing guide
- PR #7723: Use mplDeprecation class for all deprecations.
- PR #7676: Makes eventplot legend work
- PR #7714: TST: switch from 3.6-dev to 3.6
- PR #7713: Declare Python 3.6 support via classifier in setup.py
- PR #7693: Change majority of redirected links in docs
- PR #7705: Fixes tzname return type
- PR #7703: BF: Convert namespace path to list
- PR #7702: DOC: Add link to bokeh/colorcet in colormaps.rst
- PR #7700: DOC: Add gitter to home page
- PR #7692: Corrected default values of xextent in specgram(). Fixes Bug #7666.
- PR #7698: Update INSTALL for Python 3.6
- PR #7694: Fix a few broken links in docs
- PR #7349: Add support for png_text metadata, allow to customize metadata for other backends.
- PR #7670: Decode error messages from image converters.
- PR #7677: Make new default style examples consistent
- PR #7674: Serialize comparison of multiple baseline images.
- PR #7665: FIX: Fix super call for Python 2.7
- PR #7668: Save SVG test directly to file instead of its name.
- PR #7549: Cleanup: sorted, dict iteration, array.{ndim,size}, ...
- PR #7667: FIX: Fix missing package
- PR #7651: BUG,ENH: make deprecated decorator work (and more flexibly)
- PR #7658: Avoid comparing numpy array to strings in two places
- PR #7657: Fix warning when setting markeredgecolor to a numpy array
- PR #7659: DOC: Original documentation was misleading
- PR #6780: Call _transform_vmin_vmax during SymLogNorm.__init__

- [PR #7646](#): Improve deprecation documentation. Closes #7643
- [PR #7604](#): Warn if different axis projection requested
- [PR #7568](#): Deprecate unused functions in cbook.
- [PR #6428](#): Give a better error message on missing PostScript fonts
- [PR #7585](#): Fix a bug in TextBox where shortcut keys were not being reenabled
- [PR #7628](#): picker may not be callable.
- [PR #7464](#): ENH: _StringFuncParser to get numerical functions callables from strings
- [PR #7622](#): Mrg animation merge
- [PR #7618](#): DOC: fixed typo in mlab.py
- [PR #7596](#): Delay fc-list warning by 5s.
- [PR #7607](#): TST: regenerate patheffect2
- [PR #7608](#): Don't call np.min on generator.
- [PR #7570](#): Correctly skip colors for nan points given to scatter
- [PR #7605](#): Make bars stick to explicitly-specified edges.
- [PR #6597](#): Reproducible PS/PDF output (master)
- [PR #7546](#): Deprecate update_datalim_numerix&update_from_data.
- [PR #7574](#): Docs edits
- [PR #7538](#): Don't work out packages to install if user requests information from setup.p
- [PR #7577](#): Spelling fix: corosponding -> corresponding
- [PR #7536](#): Rectangle patch angle attribute and patch __str__ improvements
- [PR #7547](#): Additional cleanups
- [PR #7544](#): Cleanups
- [PR #7548](#): Clarify to_rgba docstring.
- [PR #7476](#): Sticky margins
- [PR #7552](#): Correctly extend a lognormed colorbar
- [PR #7499](#): Improve the the marker table in markers_api documentation
- [PR #7468](#): TST: Enable pytest-xdist
- [PR #7530](#): MAINT: TkAgg default backend depends on tkinter
- [PR #7531](#): double tolerance for test_png.py/pngsuite on Windows
- [PR #7533](#): FIX chinese character are hard to deal with in latex
- [PR #7525](#): Avoid division by zero if headlength=0 for quiver
- [PR #7522](#): Check at least one argument is provided for plt.table

- PR #7520: Fix table.py bug
- PR #7397: Numpydoc for backends
- PR #7513: Doc: Typo in gridspec example subtitle
- PR #7494: Remove some numpy 1.6 workarounds
- PR #7500: Set hexbin default linecolor to ‘face’
- PR #7498: Fix double running of explicitly chosen tests.
- PR #7475: Remove deprecated “shading” option to pcolor.
- PR #7436: DOC: Fixed Unicode error in gallery template cache
- PR #7496: Commit to fix a broken link
- PR #6062: Add maximum streamline length property.
- PR #7470: Clarify cross correlation documentation #1835
- PR #7481: Minor cleanup of hist().
- PR #7474: FIX/API: regenerate test figure due to hatch changes
- PR #7469: TST: Added codecov
- PR #7467: TST: Fixed part of a test that got displaced in all the changes somehow
- PR #7447: Showcase example: (kind of mandatory) Mandelbrot set
- PR #7463: Added additional coverage excludes
- PR #7449: Clarify documentation of pyplot.draw
- PR #7454: Avoid temporaries when preparing step plots.
- PR #7455: Update two_scales.py example.
- PR #7456: Add pytest’s .cache to .gitignore.
- PR #7453: TST: Fixed `test_log_margins` test
- PR #7144: Cleanup scales
- PR #7442: Added spacer to Tk toolbar
- PR #7444: Enhance `annotation_demoX` examples
- PR #7439: MEP12 API examples
- PR #7416: MAINT deprecated ‘spectral’ in favor of ‘nipy_spectral’
- PR #7435: restore test that was inadvertently removed by 5901b38
- PR #7363: Add appropriate error on color size mismatch in `scatter`
- PR #7433: FIX: search for tkinter first in builtins
- PR #7362: Added -j shortcut for --processes=
- PR #7408: Handle nan/masked values Axes.vlines and hlines

- PR #7409: FIX: MPL should not use new tool manager unless explicitly asked for. Closes #7404
- PR #7389: DOC Convert axes docstrings to numpydoc: #7205
- PR #7417: Merge from v2.x
- PR #7398: Moved python files from doc/pyplots to examples folder
- PR #7291: MEP 29: Markup text
- PR #6560: Fillbetween
- PR #7399: Clarify wspace/hspace in documentation/comments
- PR #7400: fix ReST tag
- PR #7381: Updating the readme
- PR #7384: change hardcopy.docstring to docstring.hardcopy
- PR #7386: ENH examples are now reproducible
- PR #7395: Drop code that supports numpy pre-1.6.
- PR #7385: remove unused random import
- PR #7236: ENH Improving the contribution guidelines
- PR #7370: Add example use of axes.spines.SIDE prop in matplotlibrc
- PR #7367: Warn on invalid log axis limits, per issue #7299
- PR #7360: Updated violin plot example as per suggestions in issue #7251
- PR #7357: Added notes on how to use matplotlib in pyenv
- PR #7329: DOC MEP12 - converted animation to SG/MEP12 compatible
- PR #7337: FIX symlog scale now shows negative labels.
- PR #7354: fix small error in poly_editor example
- PR #7310: TST: Make proj3d tests into real tests
- PR #7331: MEP12 improvements for statistics plots
- PR #7340: DOC: Normalize symlink target
- PR #7328: TST: Fixed rcparams test_Issue_1713 test
- PR #7303: Traceback to help fixing double-calls to mpl.use.
- PR #7346: DOC: Fix annotation position (issue #7345)
- PR #5392: BUG: arrowhead drawing code
- PR #7318: Convert a few test files to Pytest
- PR #7323: Fix #6448: set xmin/ymin even without non-zero bins in ‘step’ hist
- PR #7326: Enable coverage sending on pytest build
- PR #7321: Remove bundled virtualenv module

- PR #7290: Remove deprecated stuff schedule for removal.
- PR #7324: DOC: Boxplot color demo update
- PR #6476: Add a common example to compare style sheets
- PR #7309: MEP28: fix rst syntax for code blocks
- PR #7250: Adds docstrings to demo_curvelinear_grid.py and demo_curvelinear_grid...
- PR #4128: Code removal for post 1.5/2.1
- PR #7308: Fix travis nightly build
- PR #7282: Draft version of MEP28: Simplification of boxplots
- PR #7304: DOC: Remove duplicate documentation from last merge.
- PR #7249: add docstring to example: axisartist/demo_floating_axes.py
- PR #7296: MAINT removing docstring dedent_interp when possible
- PR #7298: Changed Examples for Pep8 Compliance
- PR #7295: MAINT finance module is deprecated
- PR #7214: FIX: Only render single patch for scatter
- PR #7297: MAINT docstring appending doesn't mess with rendering anymore.
- PR #6907: Filled + and x markers
- PR #7288: Style typos fixes
- PR #7277: MEP12 - added sphinx-gallery docstrings
- PR #7286: DOC: Fix for #7283 by adding a trailing underscore to misrendered URLs
- PR #7285: added some fixes to the documentation of the functions
- PR #6690: Tidying up and tweaking mplot3d examples [MEP12]
- PR #7273: Fix image watermark example where image was hidden by axes (#7265)
- PR #7276: FIX: don't compute flier positions if not showing
- PR #7267: DOC: changed documentation for axvspan to numpydoc format
- PR #7268: DOC Numpydoc documentation for def fill()
- PR #7272: Don't use __builtins__ (an impl. detail) in pylab.
- PR #7241: Categorical support for NumPy string arrays.
- PR #7232: DOC improved subplots' docstring
- PR #7256: CI: skip failing test on appveyor
- PR #7255: CI: pin to qt4
- PR #7229: DOC: instructions on installing matplotlib for dev
- PR #7252: ENH: improve PySide2 loading

- PR #7245: TST: Always produce image comparison test result images
- PR #6677: Remove a copy in pcolormesh.
- PR #6814: Customize violin plot demo, see #6723
- PR #7067: DOC: OO interface in api and other examples
- PR #6790: BUG: fix C90 warning -> error in new tkagg code
- PR #7242: Add mplcursors to third-party packages.
- PR #7222: Catch IO errors when building font cache
- PR #7220: Fix innocent typo in comments
- PR #7192: DOC: switch pylab example `mri_with_eeg.py` to OO interface + cosmetick fixes
- PR #6583: Fix default parameters of FancyArrow
- PR #7195: remove check under linux for `~/.matplotlib`
- PR #6753: Don't warn when legend() finds no labels.
- PR #7178: Boxplot zorder kwarg
- PR #6327: Fix captions for plot directive in latex target
- PR #7188: Remove hard-coded streamplot zorder kwarg
- PR #7170: DOC updated hexbin documentation to numpydoc format.
- PR #7031: DOC Replaced documentation with numpydoc for semilogx
- PR #7029: [WIP] DOC Updated documentation of arrow function to numpy docs format.
- PR #7167: Less stringent normalization test for float128.
- PR #7169: Remove unused variable.
- PR #7066: DOC: switch to O-O interface in basic examples
- PR #7084: [DOC] Tick locators & formatters examples
- PR #7152: Showcase example: Bézier curves & SVG
- PR #7019: Check for fontproperties in figure.suptitle.
- PR #7145: Add `style` to api doc; fix capitalization.
- PR #7097: `image_comparison` decorator refactor
- PR #7096: DOC refer to plot in the scatter plot doc
- PR #7140: FIX added matplotlib.testing.nose.plugins to setupext.py
- PR #5112: OffsetImage: use `dpi_cor` in `get_extent`
- PR #7136: DOC: minor fix in development_workflow.rst
- PR #7137: DOC: improve engineering formatter example
- PR #7131: Fix branch name in “Deleting a branch on GitHub_” section

- [PR #6521](#): Issue #6429 fix
- [PR #7111](#): [DOC] Fix example following comments in issue #6865
- [PR #7118](#): PR # 7038 rebased (DOC specgram() documentation now in numpy style)
- [PR #7117](#): PR #7030 rebased
- [PR #6618](#): Small improvements to legend's docstring.
- [PR #7102](#): Adding the artist data on mouse move event message
- [PR #7110](#): [DOC] Apply comments from issue #7017
- [PR #7087](#): [DOC] Example of user-defined linestyle (TikZ linestyle)
- [PR #7108](#): Typos in ticker.py
- [PR #7035](#): DOC Update semilogy docstring to numpy doc format
- [PR #7033](#): DOC Updated plot_date to NumPy/SciPy style
- [PR #7032](#): DOC: Updating docstring to numpy doc format for errorbar
- [PR #7094](#): TST: Restore broken `test_use14corefonts`
- [PR #6995](#): Turn off minor grids when interactively turning off major grids.
- [PR #7072](#): [DOC] New figure for the gallery (showcase section)
- [PR #7077](#): label_outer() should remove inner minor ticks too.
- [PR #7037](#): DOC change axhspan to numpydoc format
- [PR #7047](#): DOC - SpanSelector widget documentation
- [PR #7049](#): Documented dependencies to the doc and remove unnecessary dependencies.
- [PR #7063](#): Tweek tol for test_hist_steplog to fix tests on appveyor
- [PR #7055](#): FIX: testings.nose was not installed
- [PR #7058](#): Minor animation fixes
- [PR #7057](#): FIX: Removed financial demos that stalled because of yahoo requests
- [PR #7052](#): Uncaught exns are fatal for PyQt5, so catch them.
- [PR #7048](#): FIX: remove unused variable
- [PR #7042](#): FIX: ticks filtered by Axis, not in Tick.draw
- [PR #7026](#): Merge 2.x to master
- [PR #6988](#): Text box widget, take over of PR5375
- [PR #6957](#): DOC: clearing out some instances of using pylab in the docs
- [PR #7012](#): Don't blacklist test_usetex using pytest
- [PR #7011](#): TST: Fixed `skip_if_command_unavailable` decorator problem
- [PR #6918](#): enable previously leftout test_usetex

- [PR #7006](#): FIX: sphinx 1.4.0 details
- [PR #6900](#): Enh: break website screenshot banner into 4 pieces and introduce a responsive layout
- [PR #6997](#): FIX: slow plots of pandas objects (Second try)
- [PR #6792](#): PGF Backend: Support interpolation='none'
- [PR #6983](#): Catch invalid interactive switch to log scale.
- [PR #6491](#): Don't warn in Collections.contains if picker is not numlike.
- [PR #6978](#): Add link to O'Reilly video course covering matplotlib
- [PR #6930](#): BUG: PcolorImage handles non-contiguous arrays, provides data readout
- [PR #6889](#): support for updating axis ticks for categorical data
- [PR #6974](#): Fixed wrong expression
- [PR #6730](#): Add Py.test testing framework support
- [PR #6904](#): Use edgecolor rather than linewidth to control edge display.
- [PR #6919](#): Rework MaxNLocator, eliminating infinite loop; closes #6849
- [PR #6955](#): Add parameter checks to DayLocator initiator
- [PR #5161](#): [WIP] Proposed change to default log scale tick formatting
- [PR #6875](#): Add keymap (default: G) to toggle minor grid.
- [PR #6920](#): Prepare for cross-framework test suite
- [PR #6944](#): Restore cbook.report_memory, which was deleted in d063dee.
- [PR #6961](#): remove extra "a"
- [PR #6947](#): Changed error message. Issue #6933
- [PR #6923](#): Make sure nose is only imported when needed
- [PR #6851](#): Do not restrict coverage to `matplotlib` module only
- [PR #6938](#): Image interpolation selector in Qt figure options.
- [PR #6787](#): Python3.5 dictview support
- [PR #6407](#): adding default toggled state for toggle tools
- [PR #6898](#): Fix read mode when loading cached AFM fonts
- [PR #6892](#): Don't force anncoords to fig coords upon dragging.
- [PR #6895](#): Prevent forced alpha in figureoptions.
- [PR #6877](#): Fix Path deepcopy signature
- [PR #6822](#): Use travis native cache
- [PR #6821](#): Break reference cycle Line2D <-> Line2D._lineFunc.
- [PR #6879](#): Delete font cache in one of the configurations

- PR #6832: Fix for ylabel title in example `tex_unicode_demo.py`
- PR #6848: `test_tinypages`: pytest compatible module level setup
- PR #6881: add doi to bibtex entry for Hunter (2007)
- PR #6842: Clarify Axes.hexbin *extent* docstring
- PR #6861: Update ggplot URLs
- PR #6878: DOC: use venv instead of virtualenv on python 3
- PR #6837: Fix Normalize(<signed integer array>).
- PR #6874: Update bachelors_degree_by_gender example.
- PR #6867: Mark `make_all_2d_testfuncs` as not a test
- PR #6854: Fix for PyQt5.7 support.
- PR #6862: Change default doc image format to png and pdf
- PR #6819: Add `mpl_toolkits` to coveragerc.
- PR #6840: Fixed broken `test_pickle.test_complete` test
- PR #6841: DOC: Switch to OO code style & ensure fixed y-range in `psd_demo3`
- PR #6843: DOC: Fix `psd_demo_complex` similarly to `psd_demo3`
- PR #6829: Tick label rotation via `set_tick_params`
- PR #6799: Allow creating annotation arrows w/ default props.
- PR #6262: Properly handle UTC conversion in `date2num`.
- PR #6777: Raise lock timeout as actual exception
- PR #6817: DOC: Fix a few typos and formulations
- PR #6826: Clarify doc for “norm” kwarg to `imshow`.
- PR #6807: Deprecate `{get, set}_cursorprops`.
- PR #6811: Add xkcd font as one of the options
- PR #6815: Rename tests in `test_mlab.py`
- PR #6808: Don’t forget to disconnect callbacks for dragging.
- PR #6803: better freetype version checking
- PR #6778: Added contribute information to readme
- PR #6786: 2.0 Examples fixes. See #6762
- PR #6774: Appveyor: use newer conda packages and only run all tests on one platform
- PR #6779: Fix tutorial pyplot scales (issue #6775)
- PR #6768: Takeover #6535
- PR #6763: Invalidate test cache on gs/inkscape version

- [PR #6765](#): Get more rcParams for 3d
- [PR #6764](#): Support returning polylines from to_polygons
- [PR #6760](#): DOC: clean up of demo_annotation_box.py
- [PR #6735](#): Added missing side tick rcParams
- [PR #6761](#): Fixed warnings catching and counting with `warnings.catch_warnings`
- [PR #5349](#): Add a Gitter chat badge to README.rst
- [PR #6755](#): PEP: fix minor formatting issues
- [PR #6699](#): Warn if MPLBACKEND is invalid.
- [PR #6754](#): Fixed error handling in `ImageComparisonTest.setup_class`
- [PR #6734](#): register IPython's eventloop integration in `plt.install_repl_displayhook`
- [PR #6745](#): DOC: typo in broken_axis pylab example
- [PR #6747](#): Also output the actual error on svg backend tests using subprocess
- [PR #6744](#): Add workaround for failures due to newer miktex
- [PR #6741](#): Missing `cleanup` decorator in `test_subplots.test_exceptions`
- [PR #6736](#): doc: fix unescaped backslash
- [PR #6733](#): Merge v2.x to master
- [PR #6729](#): Fix crash if byte-compiled level 2
- [PR #6575](#): setup.py: Recommend installation command for pkgs
- [PR #6645](#): Fix containment and subslice optim. for steps.
- [PR #6619](#): Hide “inner” {x,y}labels in `label_outer` too.
- [PR #6639](#): Simplify `get_legend_handler` method
- [PR #6694](#): Improve Line2D and MarkerStyle instantiation
- [PR #6692](#): Remove explicit children invalidation in `update_position` method
- [PR #6703](#): DOC: explain behavior of notches beyond quartiles
- [PR #6707](#): Call `gc.collect` after each test only if the user asks for it
- [PR #6711](#): Added support for mgs to Ghostscript dependency checker
- [PR #6700](#): Don't convert vmin, vmax to floats.
- [PR #6714](#): fixed `font_manager.is_opentype_cff_font()`
- [PR #6701](#): Colours like ‘XeYYYY’ don't get recognised properly if X, Y's are numbers
- [PR #6512](#): Add computer modern font family
- [PR #6383](#): Qt editor alpha
- [PR #6381](#): Fix canonical name for “None” linestyle.

- PR #6689: Str Categorical Axis Support
- PR #6686: Merged _bool from axis into cbook._string_to_bool
- PR #6683: New entry in .mailmap
- PR #6520: Appveyor overhaul
- PR #6697: Fixed path caching bug in Path.unit_regular_star
- PR #6688: DOC: fix radial increase of size & OO style in polar_scatter_demo
- PR #6681: Fix #6680 (minor typo in IdentityTransform docstring)
- PR #6676: Fixed AppVeyor building script
- PR #6672: Fix example of streamplot start_points option
- PR #6601: BF: protect against locale in sphinxext text
- PR #6662: adding from_list to custom cmap tutorial
- PR #6666: Guard against too-large figures
- PR #6659: Fix image alpha
- PR #6642: Fix rectangle selector release bug
- PR #6652: Minor doc updates.
- PR #6653: DOC: Incorrect rendering of dashes
- PR #6648: adding a new color and editing an existing color in fivethirtyeight.mplstyle
- PR #6548: Fix typo.
- PR #6628: fix the swab bug to compile on solaris system
- PR #6622: colors: ensure masked array data is an ndarray
- PR #6625: DOC: Found a typo.
- PR #6614: Fix docstring for PickEvent.
- PR #6554: Update mpl_toolkits gtktools.
- PR #6564: Cleanup for drawstyles.
- PR #6577: Fix mlab.rec_join.
- PR #6596: Added a new example to create error boxes using a PatchCollection
- PR #2370: Implement draw_markers in the cairo backend.
- PR #6599: Drop conditional import of figureoptions.
- PR #6573: Some general cleanups
- PR #6568: Add OSX to travis tests
- PR #6600: Typo: markeredgewith -> markeredgewidth
- PR #6526: ttconv: Also replace carriage return with spaces.

- PR #6530: Update make.py
- PR #6405: ToolManager/Tools adding methods to set figure after initialization
- PR #6553: Drop prettyplotlib from the list of toolkits.
- PR #6557: Merge 2.x to master
- PR #5626: New toolbar icons
- PR #6555: Fix docstrings for `warn_deprecated`.
- PR #6544: Fix typo in margins handling.
- PR #6014: Patch for issue #6009
- PR #6517: Fix conversion of string grays with alpha.
- PR #6522: DOC: made sure boxplot demos share y-axes
- PR #6529: TST Remove plt.show() from test_axes.test_dash_offset
- PR #6519: Fix FigureCanvasAgg.print_raw(...)
- PR #6481: Default boxplot style rebase
- PR #6504: Patch issue 6035 rebase
- PR #5593: ENH: errorbar color cycle clean up
- PR #6497: Line2D._path obeys drawstyle.
- PR #6487: Added docstring to scatter_with_legend.py [MEP12]
- PR #6485: Barchart demo example clean up [MEP 12]
- PR #6472: Install all dependencies from pypi
- PR #6482: Skip test broken with numpy 1.11
- PR #6475: Do not turn on interactive mode on in example script
- PR #6442: loading TCL / Tk symbols dynamically
- PR #6467: ENH: add unified seaborn style sheet
- PR #6465: updated boxplot figure
- PR #6462: CI: Use Miniconda already installed on AppVeyor.
- PR #6456: FIX: unbreak master after 2.x merge
- PR #6445: Offset text colored by labelcolor param
- PR #6417: Showraise gtk gtk3
- PR #6423: TST: splitlines in rec2txt test
- PR #6427: Output pdf dicts in deterministic order
- PR #6431: Merge from v2.x
- PR #6433: Make the frameworkpython script compatible with Python 3

- [PR #6358](#): Stackplot weighted_wiggle zero-area fix
- [PR #6382](#): New color conversion machinery.
- [PR #6372](#): DOC: add whats_new for qt configuration editor.
- [PR #6415](#): removing unused DialogLineprops from gtk3
- [PR #6390](#): Use xkcd: prefix to avoid color name clashes.
- [PR #6397](#): key events handler return value to True to stop propagation
- [PR #6402](#): more explicit message for missing image
- [PR #5785](#): Better choice of offset-text.
- [PR #6302](#): FigureCanvasQT key auto repeat
- [PR #6334](#): ENH: webagg: Handle ioloop shutdown correctly
- [PR #5267](#): AutoMinorLocator and and logarithmic axis
- [PR #6386](#): Minor improvements concerning #6353 and #6357
- [PR #6388](#): Remove wrongly committed test.txt
- [PR #6379](#): Install basemap from git trying to fix build issue with docs
- [PR #6369](#): Update demo_floating_axes.py with comments
- [PR #6377](#): Remove unused variable in GeoAxes class
- [PR #6373](#): Remove misspelled and unused variable in GeoAxes class
- [PR #6376](#): Update index.rst - add Windrose as third party tool
- [PR #6371](#): Set size of static figure to match widget on hidp displays
- [PR #6370](#): Restore webagg backend following the merge of widget nbagg backend
- [PR #6366](#): Sort default labels numerically in Qt editor.
- [PR #6367](#): Remove stray nonascii char from nbagg
- [PR #5754](#): IPython Widget
- [PR #6146](#): ticker.LinearLocator view_limits algorithm improvement closes #6142
- [PR #6287](#): ENH: add axisbelow option ‘line’, make it the default
- [PR #6339](#): Fix #6335: Queue boxes to update
- [PR #6347](#): Allow setting image clim in Qt options editor.
- [PR #6354](#): Update events handling documentation to work with Python 3.
- [PR #6356](#): Merge 2.x to master
- [PR #6304](#): Updating animation file writer to allow keyword arguments when using with construct
- [PR #6328](#): Add default scatter marker option to rcParams
- [PR #6342](#): Remove shebang lines from all examples. [MEP12]

- PR #6337: Add a ‘useMathText’ param to method ‘ticklabel_format’
- PR #6346: Avoid duplicate cmap in image options.
- PR #6253: MAINT: Updates to formatters in `matplotlib.ticker`
- PR #6291: Color cycle handling
- PR #6340: BLD: make minimum cycler version 0.10.0
- PR #6322: Typo fixes and wording modifications (minor)
- PR #6319: Add PyUpSet as extension
- PR #6314: Only render markers on a line when markersize > 0
- PR #6303: DOC Clean up on about half the Mplot3d examples
- PR #6311: Seaborn sheets
- PR #6300: Remake of #6286
- PR #6297: removed duplicate word in Choosing Colormaps documentation
- PR #6200: Tick vertical alignment
- PR #6203: Fix #5998: Support fallback font correctly
- PR #6198: Make hatch linewidth an rcParam
- PR #6275: Fix cycler validation
- PR #6283: Use `figure.stale` instead of internal member in macosx
- PR #6247: DOC: Clarify fillbetween_x example.
- PR #6251: ENH: Added a PercentFormatter class to `matplotlib.ticker`
- PR #6267: MNT: trap inappropriate use of color kwarg in scatter; closes #6266
- PR #6249: Adjust test tolerance to pass for me on OSX
- PR #6263: TST: skip broken test
- PR #6260: Bug fix and general touch ups for hist3d_demo example (#1702)
- PR #6239: Clean warnings in examples
- PR #6170: getter for ticks for colorbar
- PR #6246: Merge v2.x into master
- PR #6238: Fix sphinx 1.4.0 issues
- PR #6241: Force Qt validator to use C locale.
- PR #6234: Limit Sphinx to 1.3.6 for the time being
- PR #6178: Use Agg for rendering in the Mac OSX backend
- PR #6232: MNT: use stdlib tools in allow_rasterization
- PR #6211: A method added to Colormap classes to reverse the colormap

- PR #6205: Use io.BytesIO instead of io.StringIO in examples
- PR #6229: Add a locator to AutoDateFormatters example code
- PR #6222: ENH: Added `file` keyword to `setp` to redirect output
- PR #6217: BUG: Made `setp` accept arbitrary iterables
- PR #6154: Some small cleanups based on Quantified code
- PR #4446: Label outer offset text
- PR #6218: DOC: fix typo
- PR #6202: Fix #6136: Don't hardcode default scatter size
- PR #6195: Documentation bug #6180
- PR #6194: Documentation bug fix: #5517
- PR #6011: Fix issue #6003
- PR #6179: Issue #6105: Adds targetfig parameter to the subplot2grid function
- PR #6185: Fix to csv2rec bug for review
- PR #6192: More precise choice of axes limits.
- PR #6176: DOC: Updated docs for rc_context
- PR #5617: Legend tuple handler improve
- PR #6188: Merge 2x into master
- PR #6158: Fix: pandas series of strings
- PR #6156: Bug: Fixed regression of `drawstyle=None`
- PR #5343: Boxplot stats w/ equal quartiles
- PR #6132: Don't check if in range if the caller passed norm
- PR #6091: Fix for issue 5575 along with testing
- PR #6123: docstring added
- PR #6145: BUG: Allowing unknown drawstyles
- PR #6148: Fix: Pandas indexing Error in collections
- PR #6140: clarified color argument in scatter
- PR #6137: Fixed outdated link to thirdpartypackages, and simplified the page
- PR #6095: Bring back the module level 'backend'
- PR #6124: Fix about dialog on Qt 5
- PR #6110: Fixes matplotlib/matplotlib#1235
- PR #6122: MNT: improve image array argument checking in `to_rgba`. Closes #2499.
- PR #6047: bug fix related #5479

- PR #6119: added comment on “usetex=False” to aide debugging when latex not ava...
- PR #6073: fixed bug 6028
- PR #6116: CI: try explicitly including msvc_runtime
- PR #6100: Update INSTALL
- PR #6099: Fix #6069. Handle image masks correctly
- PR #6079: Fixed Issue 4346
- PR #6102: Update installing_faq.rst
- PR #6101: Update INSTALL
- PR #6074: Fixes an error in the documentation, linestyle is dash_dot and should be dashdot
- PR #6068: Text class: changed __str__ method and added __repr__ method
- PR #6018: Added get_status() function to the CheckButtons widget
- PR #6013: Mnt cleanup pylab setup
- PR #5984: Suggestion for Rasterization to docs pgf-backend
- PR #5911: Fix #5895: Properly clip MOVETO commands
- PR #6039: DOC: added missing import to navigation_toolbar.rst
- PR #6036: BUG: fix ListedColormap._resample, hence plt.get_cmap; closes #6025
- PR #6029: TST: Always use / in URLs for visual results.
- PR #6022: Make @cleanup *really* support generative tests.
- PR #6024: Add Issue template with some guidelines
- PR #5718: Rewrite of image infrastructure
- PR #3973: WIP: BUG: Convert qualitative colormaps to ListedColormap
- PR #6005: FIX: do not short-cut all white-space strings
- PR #5727: Refresh pgf baseline images.
- PR #5975: ENH: add kwarg normalization function to cbook
- PR #5931: use locale.getpreferredencoding() to prevent OS X locale issues
- PR #5972: add support for PySide2, #5971
- PR #5625: DOC: add FAQ about np.datetime64
- PR #5131: fix #4854: set default numpoints of legend entries to 1
- PR #5926: Fix #5917. New dash patterns. Scale dashes by lw
- PR #5976: Lock calls to latex in texmanager
- PR #5628: Reset the available animation movie writer on rcParam change
- PR #5951: tkagg: raise each new window; partially addresses #596

- [PR #5958](#): TST: add a test for tilde in tempfile for the PS backend
- [PR #5957](#): Win: add mgs as a name for ghostscript executable
- [PR #5928](#): fix for latex call on PS backend (Issue #5895)
- [PR #5954](#): Fix issues with getting tempdir when unknown uid
- [PR #5922](#): Fixes for Windows test failures on appveyor
- [PR #5953](#): Fix typos in Axes.boxplot and Axes.bxp docstrings
- [PR #5947](#): Fix #5944: Fix PNG writing from notebook backend
- [PR #5936](#): Merge 2x to master
- [PR #5629](#): WIP: more windows build and CI changes
- [PR #5914](#): Make barbs draw correctly (Fixes #5803)
- [PR #5906](#): Merge v2x to master
- [PR #5809](#): Support generative tests in @cleanup.
- [PR #5910](#): Fix reading/writing from urllib.request objects
- [PR #5882](#): mathtext: Fix comma behaviour at start of string
- [PR #5880](#): mathtext: Fix bugs in conversion of apostrophes to primes
- [PR #5872](#): Fix issue with Sphinx 1.3.4
- [PR #5894](#): Boxplot concept figure update
- [PR #5870](#): Docs / examples fixes.
- [PR #5892](#): Fix gridspec.Gridspec: check ratios for consistency with rows and columns
- [PR #5901](#): Fixes incorrect ipython sourcecode
- [PR #5893](#): Show significant digits by default in QLineEdit.
- [PR #5881](#): Allow build children to run
- [PR #5886](#): Revert “Build the docs with python 3.4 which should fix the Traitlets...”
- [PR #5877](#): DOC: added blurb about external mpl-proscale package
- [PR #5879](#): Build the docs with python 3.4 which should fix the Traitlets/IPython...
- [PR #5871](#): Fix sized delimiters for regular-sized mathtext (#5863)
- [PR #5852](#): FIX: create _dashSeq and _dashOffset before use
- [PR #5832](#): Rewordings for normalizations docs.
- [PR #5849](#): Update setupext.py to solve issue #5846
- [PR #5853](#): Typo: fix some typos in patches.FancyArrowPatch
- [PR #5842](#): Allow image comparison outside tests module
- [PR #5845](#): V2.x merge to master

- PR #5813: mathtext: no space after comma in brackets
- PR #5828: FIX: overzealous clean up of imports
- PR #5826: Strip spaces in properties doc after newline.
- PR #5815: Properly minimize the rasterized layers
- PR #5752: Reorganise mpl_toolkits documentation
- PR #5788: Fix ImportError: No module named ‘StringIO’ on Python 3
- PR #5797: Build docs on python3.5 with linkcheck running on python 2.7
- PR #5778: Fix #5777. Don’t warn when applying default style
- PR #4857: Toolbars keep history if axes change (navtoolbar2 + toolmanager)
- PR #5790: Fix ImportError: No module named ‘Tkinter’ on Python 3
- PR #5789: Index.html template. Only insert snippet if found
- PR #5783: MNT: remove reference to deleted example
- PR #5780: Choose offset text from ticks, not axes limits.
- PR #5776: Add .noseids to .gitignore.
- PR #5466: Fixed issue with rasterized not working for errorbar
- PR #5773: Fix eb rasterize
- PR #5440: Fix #4855: Blacklist rcParams that aren’t style
- PR #5764: BUG: make clabel obey fontsize kwarg
- PR #5771: Remove no longer used Scikit image code
- PR #5766: Deterministic LaTeX text in SVG images
- PR #5762: Don’t fallback to old ipython_console_highlighting
- PR #5728: Use custom RNG for sketch path
- PR #5454: ENH: Create an abstract base class for movie writers.
- PR #5600: Fix #5572: Allow passing empty range to broken_barh
- PR #4874: Document mpl_toolkits.axes_grid1.anchored_artists
- PR #5746: Clarify that easy_install may be used to install all dependencies
- PR #5739: Silence labeled data warning in tests
- PR #5732: RF: fix annoying parens bug
- PR #5735: Correct regex in filterwarnings
- PR #5640: Warning message prior to fc-list command
- PR #5686: Remove banner about updating styles in 2.0
- PR #5676: Fix #5646: bump the font manager version

- PR #5719: Fix #5693: Implemented is_sorted in C
- PR #5721: Remove unused broken doc example axes_zoom_effect
- PR #5664: Low-hanging performance improvements
- PR #5709: Addresses issue #5704. Makes usage of parameters clearer
- PR #5716: Fix #5715.
- PR #5690: Fix #5687: Don't pass unicode to QApplication()
- PR #5707: Fix string format substitution key missing error
- PR #5706: Fix SyntaxError on Python 3
- PR #5700: BUG: handle colorbar ticks with boundaries and NoNorm; closes #5673
- PR #5702: Add missing substitution value
- PR #5701: str.formatter invalid
- PR #5697: TST: add missing decorator
- PR #5683: Include outward ticks in bounding box
- PR #5688: Improved documentation for FuncFormatter formatter class
- PR #5469: Image options
- PR #5677: Fix #5573: Use SVG in docs
- PR #4864: Add documentation for mpl_toolkits.axes_grid1.inset_locator
- PR #5434: Remove setup.py tests and adapt docs to use tests.py
- PR #5586: Fix errorbar extension arrows
- PR #5653: Update banner logo on main website
- PR #5667: Nicer axes names in selector for figure options.
- PR #5672: Fix #5670. No double endpoints in Path.to_polygon
- PR #5553: qt: raise each new window
- PR #5594: FIX: formatting in LogFormatterExponent
- PR #5588: Adjust number of ticks based on length of axis
- PR #5671: Deterministic svg
- PR #5659: Change savefig.dpi and figure.dpi defaults
- PR #5662: Bugfix for test_triage tool on Python 2
- PR #5661: Fix #5660. No FileNotFoundError on Py2
- PR #4921: Add a quit_all key to the default keymap
- PR #5651: Shorter svg files
- PR #5656: Fix #5495. Combine two tests to prevent race cond

- PR #5383: Handle HiDPI displays in WebAgg/NbAgg backends
- PR #5307: Lower test tolerance
- PR #5631: WX/WXagg backend add code that zooms properly on a Mac with a Retina display
- PR #5644: Fix typo in pyplot_scales.py
- PR #5639: Test if a frame is not already being deleted before trying to Destroy.
- PR #5583: Use data limits plus a little padding by default
- PR #4702: sphinxext/plot_directive does not accept a caption
- PR #5612: mathtext: Use DejaVu display symbols when available
- PR #5374: MNT: Mailmap fixes and simplification
- PR #5516: OSX virtualenv fixing by creating a simple alias
- PR #5546: Fix #5524: Use large, but finite, values for contour extensions
- PR #5621: Tst up coverage
- PR #5620: FIX: quiver key pivot location
- PR #5607: Clarify error when plot() args have bad shapes.
- PR #5604: WIP: testing on windows and conda packages/ wheels for master
- PR #5611: Update colormap user page
- PR #5587: No explicit mathdefault in log formatter
- PR #5591: fixed ordering of lightness plots and changed from getting lightness ...
- PR #5605: Fix DeprecationWarning in stackplot.py
- PR #5603: Draw markers around center of pixels
- PR #5596: No edges on filled things by default
- PR #5249: Keep references to modules required in pgf LatexManager destructor
- PR #5589: return extension metadata
- PR #5566: DOC: Fix typo in Axes.bxp.__doc__
- PR #5570: use base64.encodestring on python2.7
- PR #5578: Fix #5576: Handle CPLUS_INCLUDE_PATH
- PR #5555: Use shorter float repr in figure options dialog.
- PR #5552: Dep contourset vminmax
- PR #5433: ENH: pass dash_offset through to gc for Line2D
- PR #5342: Sort and uniquify style entries in figure options.
- PR #5484: fix small typo in documentation about CheckButtons.
- PR #5547: Fix #5545: Fix collection scale in data space

- PR #5500: Fix #5475: Support tolerance when picking patches
- PR #5501: Use facecolor instead of axisbg/axis_bgcolor
- PR #5544: Revert “Fix #5524. Use finfo.max instead of np.inf”
- PR #5146: Move impl. of plt.subplots to Figure.add_subplots.
- PR #5534: Fix #5524. Use finfo.max instead of np.inf
- PR #5521: Add test triage tool
- PR #5537: Fix for broken matplotlib.test function
- PR #5539: Fix docstring of violin{,plot} for return value.
- PR #5515: Fix some theoretical problems with png reading
- PR #5526: Add boxplot params to rctemplate
- PR #5533: Fixes #5522, bug in custom scale example
- PR #5514: adding str to force string in format
- PR #5512: V2.0.x
- PR #5465: Better test for isarray in figaspect(). Closes #5464.
- PR #5503: Fix #4487: Take hist bins from rcParam
- PR #5485: Contour levels must be increasing
- PR #4678: TST: Enable coveralls/codecov code coverage
- PR #5437: Make “classic” style have effect
- PR #5458: Removed normalization of arrows in 3D quiver
- PR #5480: make sure an autoreleasepool is in place
- PR #5451: [Bug] masking of NaN Z values in pcolormesh
- PR #5453: Force frame rate of FFmpegFileWriter input
- PR #5452: Fix axes.set_prop_cycle to handle any generic iterable sequence.
- PR #5448: Fix #5444: do not access subsuper nucleus _metrics if not available
- PR #5439: Use DejaVu Sans as default fallback font
- PR #5204: Minor cleanup work on navigation, text, and customization files.
- PR #5432: Don’t draw text when it’s completely clipped away
- PR #5426: MNT: examples: Set the aspect ratio to “equal” in the double pendulum animation.
- PR #5214: Use DejaVu fonts as default for text and mathtext
- PR #5306: Use a specific version of Freetype for testing
- PR #5410: Remove uses of font.get_charmap
- PR #5407: DOC: correct indentation

- PR #4863: [mpl_toolkits] Allow “figure” kwarg for host functions in parasite_axes
- PR #5166: [BUG] Don’t allow 1d-arrays in plot_surface.
- PR #5360: Add a new memleak script that does everything
- PR #5361: Fix #347: Faster text rendering in Agg
- PR #5373: Remove various Python 2.6 related workarounds
- PR #5398: Updating 2.0 schedule
- PR #5389: Faster image generation in WebAgg/NbAgg backends
- PR #4970: Fixed ZoomPanBase to work with log plots
- PR #5387: Fix #3314 assert mods.pop(0) fails
- PR #5385: Faster event delegation in WebAgg/NbAgg backends
- PR #5384: BUG: Make webagg work without IPython installed
- PR #5358: Fix #5337. Turn off –no-capture (-s) on nose
- PR #5379: DOC: Fix typo, broken link in references
- PR #5371: DOC: Add what’s new entry for TransformedPatchPath.
- PR #5299: Faster character mapping
- PR #5356: Replace numpy funcs for scalars.
- PR #5359: Fix memory leaks found by memleak_hawaii3.py
- PR #5357: Fixed typo
- PR #4920: ENH: Add TransformedPatchPath for clipping.

Issues (889):

- #7810: Dimensions sanity check in axes_rgb swaps x and y of shape when checking, prevents use with non-square images.
- #7704: screenshots in the front page of devdocs are ugly
- #7746: imshow should silence warnings on invalid values, at least when masked
- #7661: document that imshow now respects scale
- #6820: nonsensical error message for invalid input to plt.bar
- #7814: Legend for lines created using LineCollection show different handle line scale.
- #7816: re-enable or delete xmllint tests
- #7802: plt.stackplot not working for integer input with non-default ‘baseline’ parameters
- #6149: travis dedup
- #7822: Weird stroke join with patheffects
- #7784: Simple IndexError in artist.setp if empty list is passed

- #3354: Unnecessary argument in GraphicsContextBase get_linestyle
- #7820: Figure.savefig() not respecting bbox_inches='tight' when dpi specified
- #7715: Can't import pyplot in matplotlib 2.0 rc2
- #7745: Wheel distributables include unnecessary files
- #7812: On MacOS Sierra with IPython, inconsistent results with %gui, %matplotlib magic commands and –gui, and –matplotlib command-line options for ipython and qtconsole; complete failure of qtconsole inline figures
- #7808: Basemap uses deprecated methods
- #7487: Funny things happen when a rectangle with negative width/height is passed to axes()
- #7649: –nose-verbose isn't a correct option for nose
- #7656: imsave ignores origin option
- #7792: test_png.test_pngsuite.test fails on ppc64 (big-endian)
- #7788: Colorbars contain no colors when created on ppc64 (big-endian)
- #4285: plt.yscale("log") gives FloatingPointError: underflow encountered in multiply
- #7724: Can't import Matplotlib.widgets.TextBox
- #7798: test_psd_csd_equal fails for 12 (but not all) of the test_mlab.spectral_TestCase s on ppc64 (big-endian)
- #7778: Different font size between math mode and regular text
- #7777: mpl_toolkits.basemap: ValueError: level must be >= 0
- #4353: different behaviour of zoom while using ginput with MouseEvent vs KeyEvent
- #4380: horizontalalignment 'left' and 'right' do not handle spacing consistently
- #7393: subplot(): incorrect description of deletion of overlapping axes in the docs
- #7759: matplotlib dynamic plotting
- #2025: TkAgg build seems to favor Framework Tcl/Tk on OS-X
- #3991: SIGINT is ignored by MacOSX backend
- #2722: limited number of grid lines in matplotlib?
- #3983: Issue when trying to plot points with transform that requires more/fewer coordinates than it returns
- #7734: inconsistent doc regarding keymap.fullscreen default value
- #7761: Deprecation warning for finance is very unclear
- #7223: matplotlib.rcsetup docs
- #3917: OS X Cursor Not working on command line
- #4038: Hist Plot Normalization should allow a 'Per Bin' Normalization

- #3486: Update Selection Widgets
- #7457: Improvements to pylab_examples/stock_demo.py
- #7755: Can't open figure
- #7299: Raise an error or a warning when ylim's min == 0 and yscale == "log"
- #4977: Improve resolution of canvas on HiDPI with PyQt5 backend
- #7495: Missing facecolor2d attribute
- #3727: plot_date() does not work with x values of type pandas.Timestamp (pandas version 0.15.0)?
- #3368: Variable number of ticks with LogLocator for a fixed number of tick labels displayed
- #1835: docstrings of cross-correlation functions (acorr and xcorr) need clarification
- #6972: quiverkey problem when angles=array
- #6617: Problem of fonts with LaTeX rendering due to fonts-lyx package
- #7717: make all deprecation warnings be `mplDeprecation` instances
- #7662: eventplot legend fails (linewidth)
- #7673: Baseline image reuse breaks parallel testing
- #7666: Default scaling of x-axis in specgram() is incorrect (i.e. the default value for the `xextent` parameter)
- #7709: Running into problems in seaborn after upgrading matplotlib
- #7684: 3-D scatter plot disappears when overlaid over a 3-D surface plot.
- #7630: Unicode issue in matplotlib.dates
- #7678: add link to bokeh/colorcet
- #2078: linespacing of multiline texts.
- #6727: scipy 2016 sprint ideas
- #3212: Why are numpoints and scatterpoints two different keywords?
- #7697: Update INSTALL file to include py3.6
- #4428: Hyphen as a subscript doesn't appear at certain font sizes
- #2886: The wrong Game symbol is used
- #7603: scatter `color` vs `c`
- #7660: 2.0rc2: title too close to frame?
- #7672: standardize classic/v2.x order is docs
- #7680: OverflowError: Python int too large to convert to C long during plotting simple numbers on debian testing
- #7664: BUG: `super` requires at least 1 argument
- #7669: rc on conda-forge

- #5363: Warnings from test_contour.test_corner_mask
- #7663: BUG: Can't import `matplotlib._backports`
- #7647: Decorator for deprecation ignores arguments other than ‘message’
- #5806: FutureWarning with Numpy 1.10
- #6480: Setting `markeredgecolor` raises a warning
- #7653: legend doesn’t show all markers
- #7643: Matplotlib 2.0 deprecations
- #7642: imshow seems to “shift” grid.
- #7633: All attempts to plot fail with “OverflowError: Python int too large to convert to C long”
- #7637: Stacked 2D plots with interconnections in Matplotlib
- #7353: auto legend position changes upon saving the figure
- #7626: Saturation mask for `imshow()`
- #7623: potential bug with `plt.arrow` and `plt.annotate` when setting `linestyle` via tuples
- #7005: `rcParams['font.size']` is consulted at render time
- #7587: BUG: shared log axes lose `_minpos` and revert to default
- #7493: Plotting zero values with logarithmic axes triggers OverflowError, Matplotlib hangs permanently
- #7595: math domain error using `symlog` norm
- #7588: 2.0.0rc1 cannot import name ‘_macosx’
- #2051: Consider making default verticalalignment `baseline`
- #4867: Add additional minor labels in log axis with a span less than two decades
- #7489: Too small axis arrow when savefig to png
- #7611: UnicodeDecodeError when using matplotlib save SVG file and open it again
- #7592: font cache: a possibility to disable building it
- #5836: Repeated warning about `fc-list`
- #7609: The best channel to ask questions related to using matplotlib
- #7141: Feature request: auto locate minor ticks on log scaled color bar
- #3489: matplotlib scatter shifts color codes when NaN is present
- #4414: Specifying `histtype='stepfilled'` and `normed=True` when using `plt.hist` causes `ymax` to be set incorrectly
- #7597: python complain about “This application failed to start because it could not find or load the Qt platform plugin ‘xcb’ ” after an update of matplotlib
- #7578: Validate steps input to `MaxNLocator`

- #7590: Subtick labels are not disabled in classic style
- #6317: PDF file generation is not deterministic - results in different outputs on the same input
- #6543: Why does fill_betweenx not have interpolate?
- #7437: Broken path to example with strptime2num
- #7593: Issue: Applying Axis Limits
- #7591: Number of subplots in mpl.axes.Subplot object
- #7056: setup.py –name and friends broken
- #7044: location of convert in rcparams on windows
- #6813: avoid hiding edge pixels of images
- #7579: OS X libpng incompatibility
- #7576: v2.0.0rc1 conda-forge dependency issue
- #7558: Colorbar becomes 0 to 1 after colorbar ax.yaxis.set_major_formatter
- #7526: Cannot Disable TkAgg Backend
- #6565: Questionable margin-cancellation logic
- #7175: new margin system doesn't handle negative values in bars
- #5201: issue with colorbar using LogNorm and extend='min'
- #6580: Ensure install requirements in documentation are up to date before release
- #5654: Update static images in docs to reflect new style
- #7553: frange returns a last value greater than limit
- #5961: track bdist_wheel release and remove the workaround when 0.27 is released
- #7554: TeX formula rendering broken
- #6885: Check if ~/.matplotlib/ is a symlink to ~/.config/matplotlib/
- #7202: Colorbar with SymmetricalLogLocator : issue when handling only negative values
- #7542: Plotting masked array lose data points
- #6678: dead links in docs
- #7534: nbagg doesn't change figure's facecolor
- #7535: Set return of type Axes in Numpydoc docstring return type hint for Figure.add_subplot and Figure.add_axes to help jedi introspection
- #7443: pdf doc build is sort of broken
- #7521: Figure.show() fails with Qt5Agg on Windows (plt.show() works)
- #7423: Latex cache error when building docs
- #7519: plt.table() without any kwargs throws exception

- #3070: remove hold logic from library
- #1910: Pylab contourf plot using Mollweide projection create artefacts
- #5350: Minor Bug on table.py
- #7518: Incorrect transData in a simple plot
- #6985: Animation of contourf becomes extremely slow
- #7508: Legend not displayed
- #7484: Remove numpy 1.6 specific work-arounds
- #6746: Matplotlib.pyplot 2.0.0b2 fails to import with Conda Python 3.5 on OS X
- #7505: Default color cycler for plots should have more than 8 colors
- #7185: Hexbin default edgecolors kwarg is misnamed
- #7478: ‘alpha’ kwarg overrides facecolor=’none’ when plotting circle
- #7375: Patch edgecolor of a legend item does not follow look of figure
- #6873: examples/api/skewt.py is not displaying the right part of the grid by default
- #6773: Shifted image extents in 2.0.0.b3
- #7350: Colors drawn outside axis for hist2d
- #7485: Is there a way to subclass the zoom() function from the NavigationToolbar backends and modify its mouse button definition?
- #7396: Bump numpy minimal version to 1.7.0?
- #7466: missing trigger for autoscale
- #7477: v2.0.0b4 fails to build with python-3.5: Requires pygtk
- #7113: Problems with anatomy figure on v2.x
- #6722: Text: rotation inconsistency
- #7244:Codecov instead of coveralls?
- #5076: RuntimeError: LaTeX was not able to process the following string:
‘z=\$\\mathregular{{}^{}_{}}\$’ in matplotlib
- #7450: Using Matplotlib in Abaqus
- #7314: Better error message in scatter plot when len(x) != len(c)
- #7432: Failure to re-render after Line2D.set_color
- #6695: support markdown or similar
- #6228: Rasterizing patch changes filling of hatches in pdf backend
- #3023: contourf hatching and saving to pdf
- #4108: Hatch pattern changes with dpi
- #6968: autoscale differences between 1.5.1 and 2.0.0b3

- #7452: `test_log_margins` test failure
- #7143: spurious warning with nans in log-scale plot
- #7448: Relative lengths in 3d quiver plots
- #7426: prop_cycler validation over-zealous
- #6899: `savefig` has sideeffects
- #7440: Confusing examples in `annotation_demo2`
- #7441: Loading a matplotlib figure pickle within a tkinter GUI
- #6643: Incorrect margins in log scale
- #7356: `plt.hist` does not obey the `hist.bins` rcparams
- #6845: SVG backend: anomaly in gallery scatter legend
- #6527: Documentation issues
- #7315: Spectral vs spectral Deprecation warning
- #7428: from `matplotlib.backends import _tkagg` raises `AttributeError`: ‘module’ object has no attribute ‘`__file__`’
- #7431: %matplotlib notebook offsetting `sns.palplot`
- #7361: add multi-process flag as `-j` to `test.py`
- #7406: NaN causes `plt.vlines` to not scale y limits
- #7104: set offset threshold to 4
- #7404: obnoxious double warning at each script startup
- #7373: Regression in `imshow`
- #7166: Hatching in legends is broken
- #6939: `wspace` is not “The amount of width reserved for blank space between subplots” as documented
- #4026: control hatch linewidth and fill border linewidth separately
- #7390: MAINT move the examples from `doc/pyplots` to `examples` and make them reproducible
- #7198: style blacklist includes `hardcopy.docstring` but it should be `docstring.hardcopy`
- #7391: How to apply `ax.margins` to current axes limits?
- #7234: Improving documentation: Tests failing on a osx setup
- #7379: Mp4’s generated by movie writer do not appear work in browser
- #6870: Figure is unpickleable after `savefig`
- #6181: When using Agg driver, pickling fails with `TypeError` after writing figure to PDF
- #6926: SVG backend closes BytesIO on print if were `usetex=True` and `cleanup` decorator used
- #3899: Pickle not working in interactive ipython session

- #7251: Improve violin plot demo
- #7146: symlog scale no longer shows labels on the negative side
- #3420: simple plotting of numpy 2d-arrays
- #7287: Make matplotlib.use() report where the backend was set first, in case of conflict
- #7305: RuntimeError In FT2Font with NISC18030.ttf
- #7351: Interactive mode seems to be broken on MacOSX
- #7313: Axes3D.plot_surface with meshgrid args stopped working
- #7281: rcparam encoding test is broken
- #7345: Annotation minor issue in the example linestyles.py
- #7210: variable frame size support in animation is a misfeature
- #5222: legend–plot handle association
- #7312: get_facecolors() reports incorrect colors
- #7332: plot range
- #1719: Can't pickle bar plots: Failed to pickle attribute "gridline"
- #6348: When I run a file that uses matplotlib animation, I keep getting this error. Using OS X, Python 2.7, installed Python from python.org then used homebrew. Matplotlib install from pip.
- #5386: Error loading fonts on OSX 10.11
- #6448: hist step UnboundLocalError
- #6958: Document the verts kwarg to scatter
- #7204: Integrate sphinx-gallery to our user documentation
- #7325: Anaconda broken after trying to install matplotlib 2.0 beta (ubuntu)
- #7218: v1.5.3: marker=None no longer works in plot()
- #7271: BUG: symmetric kwarg in locator is not honored by contourf
- #7095: _preprocess_data interferes in the docstrings Notes section
- #7283: DOC: Misrendered URLs in the development_worflow section of devdocs.
- #7109: backport #7108 to v2.x
- #7265: Image watermark hidden by Axes in example
- #7263: axes.bxp fails without fliers
- #7274: Latex greek letters in axis labels
- #7186: matplotlib 1.5.3 raise TypeError: 'module' object is not subscriptable on pylab.py
- #6865: custom_projection_example.py is completely out of date
- #7224: FancyArrowPatch linestyle always solid

- #7215: BUG: bar deals with bytes and string x data in different manners, both that are unexpected
- #7270: Pylab import
- #7230: subplots docstring: no example of NxN grid
- #7269: documentation: texinfo markup error in matplotlib 1.4.3 and matplotlib 1.5.3
- #7264: matplotlib dependency cycle matplotlib <- ipython <- matplotlib - how to resolve?
- #7261: Legend not displayed in Plot-Matplot lib
- #7260: Unknown exception in resize
- #7259: autoscaling of yaxis fails
- #7257: How can plot a figure with matplotlib like this?
- #3959: setting up matplotlib for development
- #7240: New tests without baseline images never produce a result
- #7156: Inverted imshow using Cairo backend
- #6723: How to customize violinplots?
- #5423: fill_between wrong edge line color
- #5999: Math accents are not correctly aligned
- #1039: Cairo backend marker/line style
- #7174: default value of `lines.dash_capstyle`
- #7246: Inconsistent behaviour of `subplots` for one and more-than-one axes
- #7228: axes tick_params label color not respected when showing scientific notation for axes scale
- #7225: `get_geometry()` wrong if subplots are nested (e.g., subplots with colorbars)
- #7221: Why does pyplot display wrong grayscale image?
- #7191: BUG: Animation bugs fixed in master should be backported to 2.x
- #7017: Doc typos in “Our favorite recipes”
- #3343: Issues with `imshow` and RGBA values
- #7157: should fill_between Cycle?
- #7159: `test_colors.test_Normalize` fails in 2.0.0b4 on Fedora rawhide/aarch64 (ARMv8)
- #7201: RGBA values produce different result for `imshow` and for markers
- #3232: Navigation API Needed
- #7001: Default log ticker can make too many ticks
- #806: Provide an option for the `Animation` class to retain the previously rendered frames
- #6135: `matplotlib.animate` writes png frames in cwd instead of temp files
- #7189: graph not showing when I set format to line

- #7080: Difference in symbol sizes using Mathtext with stixsans
- #7162: _axes.py linestyle_map unused
- #7163: pyplot.subplots() is slow
- #7161: matplotlib.ticker.FormatStrFormatter clashes with ax2.set_yticklabels when dual y-axis is used
- #6549: Log scale tick labels are overlapping
- #7154: bar graph with nan values leads to “No current point in closepath” in evince
- #7149: unable to save .eps plot
- #7090: fix building pdf docs
- #6996: FontProperties size and weight ignored by figure.suptitle
- #7139: float128s everywhere for dates?
- #7083: DOC: Clarify the relationship between `plot` and `scatter`
- #7125: Import Error on matplotlib.pyplot: PyQt4
- #7124: Updated matplotlib 1.5.3 broken in default Anaconda channel
- #6429: Segfault when calling `show()` after using `Popen` (test code inside)
- #7114: BUG: `ax.tick_params` change in tick length does not adjust tick labels
- #7120: Polar plot $\cos(2x)$
- #7081: enh: additional colorblind-friendly colormaps
- #7103: Problem with discrete `ListedColormaps` when more than 4 colors are present
- #7115: Using matplotlib without Tkinter
- #7106: Wrong reader in mathtext.py
- #7078: `imshow()` does not interpret aspect/extent when interpolation='none' in svg output
- #6616: Keyboard shortcuts for toggling minor ticks grid and opening figureoptions window
- #7105: Can't pickle <type ‘instancemethod’>
- #7086: DOC (released) style is badly broken on the user doc.
- #7065: backport #7049
- #7091: v2.0.0b4 breaks viscm
- #7043: BUG: LogLocator.set_params is broken
- #7070: autoscale does not work for axes added by `fig.add_axes()`
- #3645: Proposal: Add rc setting to control dash spacing
- #7009: No good way to disable SpanSelector
- #7040: It is getting increasingly difficult to build the matplotlib documentation

- #6964: Docstring for ArtistAnimation is incorrect
- #6965: ArtistAnimation cannot animate Figure-only artists
- #7062: remove the contour on a Basemap object
- #7061: remove the contour on Basemap
- #7054: Whether the new version 2.0 will support high-definition screen?
- #7053: When will release 2.0 official version?
- #6797: Undefined Symbol Error On Ubuntu
- #6523: matplotlib-2.0.0b1 test errors on Windows
- #4753: rubber band in qt5agg slow
- #6959: extra box on histogram plot with a single value
- #6816: Segmentation fault on Qt5Agg when using the wrong linestyle
- #4212: Hist showing wrong first bin
- #4602: bar / hist : gap between first bar and other bars with lw=0.0
- #6641: Edge ticks sometimes disappear
- #7041: Python 3.5.2 crashes when launching matplotlib 1.5.1
- #7028: Latex Greek fonts not working in legend
- #6998: dash pattern scaling with linewidth should get it's own rcParam
- #7021: How to prevent matplotlib from importing qt4 libraries when only
- #7020: Using tick_right() removes any styling applied to tick labels.
- #7018: Website Down
- #6785: Callbacks of draggable artists should check that they have not been removed
- #6783: Draggable annotations specified in offset coordinates switch to figure coordinates after dragging
- #7015: pcolor() not using “data” keyword argument
- #7014: matplotlib works well in ipython note book but can't display in a terminal running
- #6999: cycler 0.10 is required due to change_key() usage
- #6794: Incorrect text clipping in presence of multiple subplots
- #7004: Zooming with a large range in y-values while using the linestyle “–” is very slow
- #6828: Spikes in small wedges of a pie chart
- #6940: large memory leak in new contour routine
- #6894: bar(..., linewidth=None) doesn't display bar edges with mpl2.0b3
- #6989: bar3d no longer allows default colors

- #6980: problem accessing canvas on MacOS 10.11.6 with matplotlib 2.0.0b3
- #6804: Histogram of xarray.DataArray can be extremely slow
- #6859: Update URL for links to ggplot
- #6852: Switching to log scale when there is no positive data crashes the Qt5 backend, causes inconsistent internal state in others
- #6740: PGF Backend: Support interpolation='none'?
- #6665: regression: builtin latex rendering doesn't find the right mathematical fonts
- #6984: plt.annotate(): segmentation fault when coordinates are too high
- #6979: plot won't show with plt.show(block=False)
- #6981: link to ggplot is broken...
- #6975: [Feature request] Simple ticks generator for given range
- #6905: pcolorfast results in invalid cursor data
- #6970: quiver problems when angles is an array of values rather than 'uv' or 'xy'
- #6966: No Windows wheel available on PyPI for new version of matplotlib (1.5.2)
- #6721: Font cache building of matplotlib blocks requests made to HTTPd
- #6844: scatter edgecolor is broken in Matplotlib 2.0.0b3
- #6849: BUG: endless loop with MaxNLocator integer kwarg and short axis
- #6935: matplotlib.dates.DayLocator cannot handle invalid input
- #6951: Ring over A in AA is too high in Matplotlib 1.5.1
- #6960: axvline is sometimes not shown
- #6473: Matplotlib manylinux wheel - ready to ship?
- #5013: Add Hershey Fonts a la IDL
- #6953: ax.vlines adds unwanted padding, changes ticks
- #6946: No Coveralls reports on GitHub
- #6933: Misleading error message for matplotlib.pyplot.errorbar()
- #6945: Matplotlib 2.0.0b3 wheel can't load libpng in OS X 10.6
- #3865: Improvement suggestions for matplotlib.Animation.save('video.mp4')
- #6932: Investigate issue with pyparsing 2.1.6
- #6941: Interfering with yahoo_finance
- #6913: Cant get currency from yahoo finance with matplotlib
- #6901: Add API function for removing legend label from graph
- #6510: 2.0 beta: Boxplot patches zorder differs from lines

- #6911: freetype build won't become local
- #6866: examples/misc/longshort.py is outdated
- #6912: Matplotlib fail to compile matplotlib._png
- #1711: Autoscale to automatically include a tiny margin with `Axes.errorbar()`
- #6903: RuntimeError('Invalid DISPLAY variable') - With docker and django
- #6888: Can not maintain zoom level when left key is pressed
- #6855: imsave-generated PNG files missing edges for certain resolutions
- #6479: Hexbin with log scale takes extent range as logarithm of the data along the log axis
- #6795: suggestion: set_xticklabels and set_yticklabels default to current labels
- #6825: I broke imshow(<signed integer array>) :-(
- #6858: PyQt5 pyplot error
- #6853: PyQt5 (v5.7) backend - TypeError upon calling figure()
- #6835: Which image formats to build in docs.
- #6856: Incorrect plotting for versions > 1.3.1 and GTK.
- #6838: Figures not showing in interactive mode with macosx backend
- #6846: GTK Warning
- #6839: Test `test_pickle.test_complete` is broken
- #6691: rcParam missing tick side parameters
- #6833: plot contour with levels from discrete data
- #6636: DOC: gallery supplies 2 pngs, neither of which is default
- #3896: dates.date2num bug with daylight switching hour
- #6685: 2.0 dev legend breaks on scatterplot
- #3655: ensure removal of font cache on version upgrade
- #6818: Failure to build docs: unknown property
- #6798: clean and regenerate travis cache
- #6782: 2.x: Contour level count is not respected
- #6796: plot/lines not working for datetime objects that span old dates
- #6660: cell focus/cursor issue when plotting to nbagg
- #6775: Last figure in http://matplotlib.org/users/pyplot_tutorial.html is not displayed correctly
- #5981: Increased tick width in 3D plots looks odd
- #6771: ImportError: No module named artist
- #6289: Grids are not rendered in backend implementation

- #6621: Change in the result of test_markevery_linear_scales_zoomed
- #6515: Dotted grid lines in v2.0.0b1
- #6511: Dependencies in installation of 2.0.0b1
- #6668: “Bachelor’s degrees...” picture in the gallery is cropped
- #6751: Tableau style
- #6742: import matplotlib.pyplot as plt throws an error
- #6097: anaconda package missing nose dependency
- #6299: savefig() to eps/pdf does not work
- #6387: import matplotlib causes UnicodeDecodeError
- #6471: Colorbar label position different when executing a block of code
- #6732: Adding pairplot functionality?
- #6749: Step diagram does not support xlim() and ylim()
- #6748: Step diagram does not support
- #6615: Bad event index for step plots
- #6588: Different line styles between PNG and PDF exports.
- #6693: linestyle=”None” argument for fill_between() doesn’t work
- #6592: Linestyle pattern depends on current style, not style set at creation
- #5430: Linestyle: dash tuple with offset
- #6728: Can’t install matplotlib with specific python version
- #6546: Recommendation to install packages for various OS
- #6536: get_sample_data() in cbook.py duplicates code from _get_data_path() __init__.py
- #3631: Better document meaning of notches in boxplots
- #6705: The test suite spends 20% of its time in gc.collect()
- #6698: Axes3D scatter crashes without alpha keyword
- #5860: Computer Modern Roman should be the default serif when using TeX backend
- #6702: Bad fonts crashes matplotlib on startup
- #6671: Issue plotting big endian images
- #6196: Qt properties editor discards color alpha
- #6509: pylab image_masked is broken
- #6657: appveyor is failing on pre-install
- #6610: Icons for Tk are not antialiased.
- #6687: Small issues with the example polar_scatter_demo.py

- #6541: Time to deprecate the GTK backend
- #6680: Minor typo in the docstring of `IdentityTransform`?
- #6670: `plt.text` object updating incorrectly with `blit=False`
- #6646: Incorrect `fill_between` chart when use `set_xscale('log')`
- #6540: `imshow(..., alpha=0.5)` produces different results in 2.x
- #6650: `fill_between()` not working properly
- #6566: Regression: `Path.contains_points` now returns `uint` instead of `bool`
- #6624: bus error: fc-list
- #6655: Malware found on matplotlib components
- #6623: RectangleSelector disappears after resizing
- #6629: matplotlib version error
- #6638: `get_ticklabels` returns “” in ipython/python interpreter
- #6631: can’t build matplotlib on smartos system(open solaris)
- #6562: 2.x: Cairo backends cannot render images
- #6507: custom scatter marker demo broken
- #6591: DOC: update static image for `interpolation_none_vs_nearest.py` example
- #6607: BUG: saving image to png changes colors
- #6587: please copy <http://matplotlib.org/devdocs/users/colormaps.html> to <http://matplotlib.org/users>
- #6594: Documentation Typo
- #5784: dynamic ticking (#5588) should avoid (if possible) single ticks
- #6492: `mpl_toolkits.mplot3d` has a null byte somewhere
- #5862: Some Microsoft fonts produce unreadable EPS
- #6537: bundled six 1.9.0 causes `ImportError: No module named ‘winreg’` in Pympler
- #6563: `pyplot.errorbar` attempts to plot 0 on a log axis in SVGs
- #6571: Unexpected behavior with `ttk.Notebook` - graph not loaded unless tab preselected
- #6570: Unexpected behavior with `ttk.Notebook` - graph not loaded unless tab preselected
- #6539: network tests are not skipped when running `tests.py` with `-no-network`
- #6567: `qt_compat` fails to identify PyQt5
- #6559: `mpl 1.5.1` requires `pyqt` even with a `wx` backend
- #6009: No space before unit symbol when there is no SI prefix in `ticker.EngFormatter`
- #6528: Fail to install matplotlib by “`pip install`” on SmartOS(like open solaris system)
- #6531: Segmentation fault with any backend (matplotlib 1.4.3 and 1.5.1) when calling `pyplot.show()`

- #6513: Using gray shade from string ignores alpha parameters
- #6477: Savefig() to pdf renders markers differently than show()
- #6525: PS export issue with custom font
- #6514: LaTeX axis labels can no longer have custom fonts
- #2663: Multi Cursor disable broken
- #6083: Figure linewidth default in rcparams
- #1069: Add a donation information page
- #6035: Issue(?): head size of FancyArrowPatch changes between interactive figure and picture export
- #6495: new figsize is bad for subplots with fontsize 12
- #6493: Stepfilled color cycle for background and edge different
- #6380: Implicit addition of “color” to property_cycle breaks semantics
- #6447: Line2D.contains does not take drawstyle into account.
- #6257: option for default space between title and axes
- #5868: tight_layout doesn’t leave enough space between outwards ticks and axes title
- #5987: Outward ticks cause labels to be clipped by default
- #5269: Default changes: legend
- #6489: Test errors with numpy 1.11.1rc1
- #5960: Misplaced shadows when using FilteredArtistList
- #6452: Please add a generic “seaborn” style
- #6469: Test failures testing matplotlib 1.5.1 manylinux wheels
- #5854: New cycler does not work with bar plots
- #5977: legend needs logic to deal with new linestyle scaling by linewidth
- #6365: Default format time series xtick labels changed
- #6104: docs: latex required for PDF plotting?
- #6451: Inequality error on web page http://matplotlib.org/faq/howto_faq.html
- #6459: use conda already installed on appveyor
- #6043: Advanced hillshading example looks strange with new defaults.
- #6440: BUG: set_tick_params labelcolor should apply to offset
- #6458: Wrong package name in INSTALL file
- #2842: matplotlib.tests.test_basic.test_override_builtins() fails with Python >=3.4
- #2375: matplotlib 1.3.0 doesn’t compile with Solaris Studio 12.1 CC
- #2667: matplotlib.tests.test_mathtext.test_mathtext_{cm,stix,stixsans}_{37,53}.test are failing

- #2243: axes limits with aspect='equal'
- #1758: y limit with dashed or dotted lines hangs with somewhat big data
- #5994: Points annotation coords not working in 2.x
- #6444: matplotlib.path.contains_points is a LOT slower in 1.51
- #5461: Feature request: allow a default line alpha to be set in mpl.rcParams
- #5132: ENH: Set the alpha value for plots in rcParams
- #6449: axhline and axvline linestyle as on-off seq doesn't work if set directly in function call
- #6416: animation with 'ffmpeg' backend and 'savefig.bbox = tight' garbles video
- #6437: Improperly spaced time axis
- #5974: scatter is not changing color in Axes3D
- #6436: clabels plotting outside of projection limb
- #6438: Cant get emoji working in Pie chart legend with google app engine. Need help.
- #6362: greyscale scatter points appearing blue
- #6301: tricky bug in ticker due to special behaviour of numpy
- #6276: Ticklabel format not preserved after editing plot limits
- #6173: linestyle parameter does not support default cycler through None, crashes instead.
- #6109: colorbar _ticker +_locate bug
- #6231: Segfault when figures are deleted in random order
- #6432: micro sign doesn't show in EngFormatter
- #6057: Infinite Loop: LogLocator Colorbar & update_ticks
- #6270: pyplot.contour() not working with matplotlib.ticker.LinearLocator()
- #6058: "Configure subplots" tool is initialized very inefficiently in the Qt backends
- #6363: Change legend to accept alpha instead of (only) framealpha.
- #6394: Severe bug in ``imshow`` when plotting images with small values
- #6368: Bug: matplotlib.pyplot.spy: does not work correctly for sparse matrices with many entries ($\geq 2^{**}32$)
- #6419: Imshow does not copy data array but determines colormap values upon call
- #3615: mouse scroll event in Gtk3 backend
- #3373: add link to gtk embedding cookbook to website
- #6121: opening the configure subplots menu moves the axes by a tiny amount
- #2511: NavigationToolbar breaks if axes are added during use.
- #6349: Down arrow on GTK3 backends selects toolbar, which eats further keypress events

- #6408: minor ticks don't respect rcParam xtick.top / ytick.right
- #6398: sudden install error with pip (pyparsing 2.1.2 related)
- #5819: 1.5.1rc1: dont use absolute links in the "new updates" on the homepage
- #5969: urgent bug after 1.5.0: offset of LineCollection when apply agg_filter
- #5767: axes limits (in old "round_numbers" mode) affected by floating point issues
- #5755: Better choice of axes offset value
- #5938: possible bug with ax.set_yscale('log') when all values in array are zero
- #6399: pyparsing version 2.1.2 not supported (2.1.1 works though)
- #5884: numpy as no Attribute `string0`
- #6395: Deprecation warning for axes.color_cycle
- #6385: Possible division by zero in new `get_tick_space()` methods; is rotation ignored?
- #6344: Installation issue
- #6315: Qt properties editor could sort lines labels using natsort
- #5219: Notebook backend: possible to remove javascript/html when figure is closed?
- #5111: nbagg backend captures exceptions raised by callbacks
- #4940: NBAgg figure management issues
- #4582: Matplotlib IPython Widget
- #6142: matplotlib.ticker.LinearLocator view_limits algorithm improvement?
- #6326: Unicode invisible after image saved
- #5980: Gridlines on top of plot by default in 2.0?
- #6272: Ability to set default scatter marker in matplotlibrc
- #6335: subplots animation example is broken on OS X with qt4agg
- #6357: pyplot.hist: normalization fails
- #6352: clim doesn't update after draw
- #6353: hist won't norm for small numbers
- #6343: prop_cycle breaks keyword aliases
- #6226: Issue saving figure as eps when using gouraud shaded triangulation
- #6330: ticklabel_format reset to default by ScalarFormatter
- #4975: Non-default `color_cycle` not working in Pie plot
- #5990: Scatter markers do not follow new colour cycle
- #5577: Handling of "next color in cycle" should be handled differently
- #5489: Special color names to pull colors from the currently active color cycle

- #6325: Master requires cycler 0.10.0
- #6278: imshow with pgf backend does not render transparency
- #5945: Figures in the notebook backend are too large following DPI changes
- #6332: Animation with blit broken
- #6331: matplotlib pcolormesh seems to slide some data around on the plot
- #6307: Seaborn style sheets don't edit `patch.facecolor`
- #6294: Zero size ticks show up as single pixels in rendered pdf
- #6318: Cannot import mpl_toolkits in Python3
- #6316: Viridis exists but not in plt.cm.datad.keys()
- #6082: Cannot interactively edit axes limits using Qt5 backend
- #6309: Make CheckButtons based on subplots automatically
- #6306: Can't show images when plt.show() was executed
- #2527: Vertical alignment of text is too high
- #4827: Pickled Figure Loses sharedx Properties
- #5998: math??{} font styles are ignored in 2.x
- #6293: matplotlib notebook magic cells with output plots - skips next cell for computation
- #235: hatch linewidth patch
- #5875: Manual linestyle specification ignored if 'prop_cycle' contains 'ls'
- #5959: imshow rendering issue
- #6237: Mac OSX agg version: doesn't redraw after keymap.grid keypress
- #6266: Better fallback when color is a float
- #6002: Potential bug with 'start_points' argument of 'pyplot.streamplot'
- #6265: Document how to set viridis as default colormap in mpl 1.x
- #6258: Rendering vector graphics: parsing polygons?
- #1702: Bug in 3D histogram documentation
- #5937: xticks/yticks default behaviour
- #4706: Documentation - Basemap
- #6255: Can't build matplotlib.ft2font in cygwin
- #5792: Not easy to get colorbar tick mark locations
- #6233: ImportError from Sphinx plot_directive from Cython
- #6235: Issue with building docs with Sphinx 1.4.0
- #4383: xkcd color names

- #6219: Example embedding_in_tk.py freezes in Python3.5.1
- #5067: improve whats_new entry for prop cycler
- #4614: Followup items from the matplotlib 2.0 BoF
- #5986: mac osx backend does not scale dashes by linewidth
- #4680: Set forward=True by default when setting the figure size
- #4597: use mkdtemp in _create_tmp_config_dir
- #3437: Interactive save should respect ‘savefig.facecolor’ rcParam.
- #2467: Improve default colors and layouts
- #4194: matplotlib crashes on OS X when saving to JPEG and then displaying the plot
- #4320: Pyplot.imshow() “None” interpolation is not supported on Mac OSX
- #1266: Draggable legend results RuntimeError and AttributeError on Mac OS 10.8.1
- #5442: xkcd plots rendered as regular plots on Mac OS X
- #2697: Path snapping does not respect quantization scale appropriate for Retina displays
- #6049: Incorrect TextPath display under interactive mode
- #1319: macosx backend lacks support for cursor-type widgets
- #531: macosx backend does not work with blitting
- #5964: slow rendering with backend_macosx on El Capitan
- #5847: macosx backend color rendering
- #6224: References to non-existing class FancyBoxPatch
- #781: macosx backend doesn’t find fonts the same way as other backends
- #4271: general colormap reverser
- #6201: examples svg_histogram.html failes with UnicodeEncodeError
- #6212: ENH? BUG? pyplot.setup/Artist.setup does not accept non-indexable iterables of handles.
- #4445: Two issues with the axes offset indicator
- #6209: Qt4 backend uses Qt5 backend
- #6136: Feature request: configure default scatter plot marker size
- #6180: Minor typos in the style sheets users’ guide
- #5517: “interactive example” not working with PySide
- #4607: bug in font_manager.FontManager.score_family()
- #4400: Setting annotation background covers arrow
- #596: Add “bring window to front” functionality

- [#4674](#): Default marker edge width in plot vs. scatter
- [#5988](#): rainbow_text example is missing some text
- [#6165](#): MacOSX backend hangs drawing lines with many dashes/dots
- [#6155](#): Deprecation warnings with Dateutil 2.5
- [#6003](#): In ‘pyplot.streamplot’, starting points near the same streamline raise ‘InvalidIndexError’
- [#6105](#): Accepting figure argument in subplot2grid
- [#6184](#): csv2rec handles dates differently to datetimes when datefirst is specified.
- [#6164](#): Unable to use PySide with gui=qt
- [#6166](#): legends do not refresh
- [#3897](#): bug: inconsistent types accepted in DateLocator subclasses
- [#6160](#): EPS issues with rc parameters used in seaborn library on Win 8.1
- [#6163](#): Can’t make matplotlib run in my computer
- [#5331](#): Boxplot with zero IQR sets whiskers to max and min and leaves no outliers
- [#5575](#): plot_date() ignores timezone
- [#6143](#): drawstyle accepts anything as default rather than raising
- [#6151](#): Matplotlib 1.5.1 ignores annotation_clip parameter
- [#6147](#): colormaps issue
- [#5916](#): Headless get_window_extent or equivalent
- [#6141](#): Matplotlib subplots and datetime x-axis functionality not working as intended?
- [#6138](#): No figure shows, no error
- [#6134](#): Cannot plot a line of width=1 without antialiased
- [#6120](#): v2.x failures on travis
- [#6092](#): %matplotlib notebook broken with current matplotlib master
- [#1235](#): Legend placement bug
- [#2499](#): Showing np.uint16 images of the form (h,w,3) is broken
- [#5479](#): Table: auto_set_column_width not working
- [#6028](#): Appearance of non-math hyphen changes with math in text
- [#6113](#): ValueError after moving legend and rcParams.update
- [#6111](#): patches fails when data are array, not list
- [#6108](#): Plot update issue within event callback for multiple updates
- [#6069](#): imshow no longer correctly handles ‘bad’ (nan) values
- [#6103](#): ticklabels empty when not interactive

- #6084: Despined figure is cropped
- #6067: pyplot.savefig doesn't expand ~ (tilde) in path
- #4754: Change default color cycle
- #6063: Axes.relim() seems not to work when copying Line2D objects
- #6065: Proposal to change color – ‘indianred’
- #6056: quiver plot in polar projection - how to make the quiver density latitude-dependent ?
- #6051: Matplotlib v1.5.1 apparently not compatible with python-dateutil 2.4.2
- #5513: Call get_backend in pylab_setup
- #5983: Option to Compress Graphs for pgf-backend
- #5895: Polar Projection PDF Issue
- #5948: tilted line visible in generated pdf file
- #5737: matplotlib 1.5 compatibility with wxPython
- #5645: Missing line in a self-sufficient example in navigation_toolbar.rst :: a minor bug in docs
- #6037: Matplotlib xtick appends .%f after %H:%M%:%S on chart
- #6025: Exception in Tkinter/to_rgb with new colormaps
- #6034: colormap name is broken for ListedColormap?
- #5982: Styles need update after default style changes
- #6017: Include tests.py in archive of release
- #5520: ‘nearest’ interpolation not working with low dpi
- #4280: imsave reduces 1row from the image
- #3057: DPI-connected bug of imshow when using multiple masked arrays
- #5490: Don't interpolate images in RGB space
- #5996: 2.x: Figure.add_axes(..., facecolor='color') does not set axis background colour
- #4760: Default linewidth thicker than axes linewidth
- #2698: ax.text() fails to draw a box if the text content is full of blank spaces and linefeeds.
- #3948: a weird thing in the source code comments
- #5921: test_backend.pgf.check_for(texsystem) does not do what it says...
- #4295: Draggable annotation position wrong with negative x/y
- #1986: Importing pyplot messes with command line argument parsing
- #5885: matplotlib stepfilled histogram breaks at the value 10^-1 on xubuntu
- #5050: pandas v0.17.0rc1
- #3658: axes.locator_params fails with LogLocator (and most Locator subclasses)

- #3742: Square plots
- #3900: debugging Segmentation fault with Qt5 backend
- #4192: Error when color value is None
- #4210: segfault: fill_between with Python3
- #4325: FancyBboxPatch wrong size
- #4340: Histogram gap artifacts
- #5096: Add xtick.top.visible, xtick.bottom.visible, ytick.left.visible, ytick.right.visible to rcParams
- #5120: custom axis scale doesn't work in 1.4.3
- #5212: shifted(?) bin positions when plotting multiple histograms at the same time
- #5293: Qt4Agg: RuntimeError: call __init__ twice
- #5971: Add support for PySide2 (Qt5)
- #5993: Basemap readshapefile should read shapefile for the long/lat specified in the Basemap instance.
- #5991: basemap crashes with no error message when passed numpy nan's
- #5883: New colormaps : Inferno, Viridis, ...
- #5841: extra label for non-existent tick
- #4502: Default style proposal: outward tick marks
- #875: Replace "jet" as the default colormap
- #5047: Don't draw end caps on error bars by default
- #4700: Overlay blend mode
- #4671: Change default legend location to 'best'.
- #5419: Default setting of figure transparency in NbAgg is a performance problem
- #4815: Set default axis limits in 2D-plots to the limits of the data
- #4854: set numpoints to 1
- #5917: improved dash styles
- #5900: Incorrect Image Tutorial Inline Sample Code
- #5965: xkcd example in gallery
- #5616: Better error message if no animation writer is available
- #5920: How to rotate secondary y axis label so it doesn't overlap with y-ticks, matplotlib
- #5966: SEGFAULT if pyplot is imported
- #5967: savefig SVG and PDF output for scatter plots is excessively complex, crashes Inkscape
- #1943: legend doesn't work with stackplot

- #5923: Windows usetex=True error in long usernames
- #5940: KeyError: ‘getpwuid(): uid not found: 5001’
- #5748: Windows test failures on appveyor
- #5944: Notebook backend broken on Master
- #5946: Calling subplots_adjust breaks savefig output
- #5929: Fallback font doesn’t work on windows?
- #5925: Data points beyond axes range plotted when saved to SVG
- #5918: Pyplot.savefig is very slow with some combinations of data/ylim scales
- #5919: Error when trying to import matplotlib into IPython notebook
- #5803: Barbs broken
- #5846: setupext.py: problems parsing setup.cfg (not updated to changes in configparser)
- #5309: Differences between function and keywords for savefig.bbox and axes.facecolor
- #5889: Factual errors in HowTo FAQ Box Plot Image
- #5618: New rcParams requests
- #5810: Regression in test_remove_shared_axes
- #5281: plt.tight_layout(pad=0) cuts away outer ticks
- #5909: The documentation for LinearLocator’s presets keyword is unclear
- #5864: mathtext mishandling of certain exponents
- #5869: doc build fails with mpl-1.5.1 and sphinx-1.3.4 (sphinx-1.3.3 is fine)
- #5835: gridspec.Gridspec doesn’t check for consistency in arguments
- #5867: No transparency in *.pgf file when using pgf Backend.
- #5863: left(... right) are too small
- #5850: prop_cycler for custom dashes – linestyle such as (<offset>, (<on>, <off>)) throws error
- #5861: Marker style request
- #5851: Bar and box plots use the ‘default’ matplotlib colormap, even if the style is changed
- #5857: FAIL: matplotlib.tests.test_coding_standards.test_pep8_conformance_examples
- #5831: tests.py is missing from pypi tarball
- #5829: test_rasterize_dpi fails with 1.5.1
- #5843: what is the source code of ax.pcolormesh(T, R, Z,vmin=0,vmax=255,cmap=’jet’) ?
- #5799: mathtext kerning around comma
- #2841: There is no set_linestyle_cycle in the matplotlib axes API
- #5821: Consider using an offline copy of Raleway font

- #5822: FuncAnimation.save() only saving 1 frame
- #5449: Incomplete dependency list for installation from source
- #5793: GTK backends
- #5814: Adding colorbars to row subplots doesn't render the main plots when saving to .eps in 1.5.0
- #5816: matplotlib.pyplot.boxplot ignored showmeans keyword
- #5086: Default date format for axis formatting
- #4808: AutoDateFormatter shows too much precision
- #5812: Widget event issue
- #5794: --no-network not recognized as valid option for tests.py
- #5801: No such file or directory: '/usr/share/matplotlib/stylelib'
- #5777: Using default style raises warnings about non style parameters
- #5738: Offset text should be computed based on lowest and highest ticks, not actual axes limits
- #5403: Document minimal MovieWriter sub-class
- #5558: The link to the John Hunter Memorial fund is a 404
- #5757: Several axes_grid1 and axisartist examples broken on master
- #5557: plt.hist throws KeyError when passed a pandas.Series without 0 in index
- #5550: Plotting datetime values from Pandas dataframe
- #4855: Limit what style.use can affect?
- #5765: import matplotlib._png as _png ImportError: libpng16.so.16: cannot open shared object
- #5753: Handling of zero in log shared axes depends on whether axes are shared
- #5756: 3D rendering, scatterpoints disappear near edges of surfaces
- #5747: Figure.suptitle does not respect size argument
- #5641: plt.errorbar error with empty list
- #5476: annotate doesn't trigger redraw
- #5572: Matplotlib 1.5 broken_barh fails on empty data.
- #5089: axes.properties calls get_axes internally
- #5745: Using internal qhull despite the presence of pyqhull installed in the system
- #5744: cycler is required, is missing, yet build succeeds.
- #5592: Problem with __init_func in ArtistAnimation
- #5729: Test matplotlib.tests.test_backend_svg.test_determinism fails on OSX in virtual envs.
- #4756: font_manager.py takes multiple seconds to import
- #5435: Unable to upgrade matplotlib 1.5.0 through pip

- #5636: Generating legend from figure options panel of qt backend raise exception for large number of plots
- #5365: Warning in test_lines.test_nan_is_sorted
- #5646: Version the font cache
- #5692: Can't remove StemContainer
- #5635: RectangleSelector creates not wanted lines in axes
- #5427: BUG? Normalize modifies pandas Series inplace
- #5693: Invalid caching of long lines with nans
- #5705: doc/users/plotting/examples/axes_zoom_effect.py is not a Python file
- #4359: savefig crashes with malloc error on os x
- #5715: Minor error in set up fork
- #5687: Segfault on plotting with PySide as backend.qt4
- #5708: Segfault with Qt4Agg backend in 1.5.0
- #5704: Issue with xy and xytext
- #5673: colorbar labelling bug (1.5 regression)
- #4491: Document how to get a framework build in a virtual env
- #5468: axes selection in axes editor
- #5684: AxesGrid demo exception with LogNorm: 'XAxis' object has no attribute 'set_scale'
- #5663: AttributeError: 'NoneType' object has no attribute 'canvas'
- #5573: Support HiDPI (retina) displays in docs
- #5680: SpanSelector span_stays fails with use_blit=True
- #5679: Y-axis switches to log scale when an X-axis is shared multiple times.
- #5655: Problems installing basemap behind a proxy
- #5670: Doubling of coordinates in polygon clipping
- #4725: change set_adjustable for share axes with aspect ratio of 1
- #5488: The default number of ticks should be based on the length of the axis
- #5543: num2date ignoring tz in v1.5.0
- #305: Change canvas.print_figure default resolution
- #5660: Cannot raise FileNotFoundError in python2
- #5658: A way to remove the image of plt.figimage()?
- #5495: Something fishy in png reading
- #5549: test_streamplot:test_colormap test broke unintentionally

- #5381: HiDPI support in Notebook backend
- #5531: test_mplot3d:test_quiver3d broke unintentionally
- #5530: test_axes:test_polar_unit broke unintentionally
- #5525: Comparison failure in text_axes:test_phase_spectrum_freqs
- #5650: Wrong backend selection with PyQt4
- #5649: Documentation metadata (release version) does not correspond with some of the ‘younger’ documentation content
- #5648: Some tests require non-zero tolerance
- #3980: zoom in wx with retina behaves badly
- #5642: Mistype in pyplot_scales.py of pyplot_tutorial.rst :: a minor bug in docs
- #3316: wx crashes on exit if figure not shown and not explicitly closed
- #5624: Cannot manually close matplotlib plot window in Mac OS X Yosemite
- #4891: Better auto-selection of axis limits
- #5633: No module named externals
- #5634: No module named ‘matplotlib.tests’
- #5473: Strange OS warning when import pyplot after upgrading to 1.5.0
- #5524: Change in colorbar extensions
- #5627: Followup for Windows CI stuff
- #5613: Quiverkey() positions arrow incorrectly with labelpos ‘N’ or ‘S’
- #5615: tornado now a requirement?
- #5582: FuncAnimation crashes the interpreter (win7, 64bit)
- #5610: Testfailures on windows
- #5595: automatically build windows conda packages and wheels in master
- #5535: test_axes:test_rc_grid image comparison test has always been broken
- #4396: Qt5 is not mentioned in backends list in doc
- #5205: pcolor does not handle non-array C data
- #4839: float repr in axes parameter editing window (aka the green tick button)
- #5542: Bad superscript positioning for some fonts
- #3791: Update colormap examples.
- #4679: Relationship between line-art markers and the markeredgewidth parameter
- #5601: Scipy/matplotlib recipe with plt.connect() has trouble in python 3 (AnnoteFinder)
- #4211: Axes3D quiver: variable length arrows

- #773: mplot3d enhancement
- #395: need 3D examples for tricontour and tricontourf
- #186: Axes3D with PolyCollection broken
- #178: Incorrect mplot3d contourf rendering
- #5508: Animation.to_html5_video requires python3 base64 module
- #5576: Improper reliance upon pkg-config when C_INCLUDE_PATH is set
- #5369: Change in zorder of streamplot between 1.3.1 and 1.4.0
- #5569: Stackplot does not handle NaNs
- #5565: label keyword is not interpreted properly in errorbar() for pandas.DataFrame-like objects
- #5561: interactive mode doesn't display images with standard python interpreter
- #5559: Setting window titles when in interactive mode
- #5554: Cropping text to axes
- #5545: EllipseCollection renders incorrectly when passed a sequence of widths
- #5475: artist picker tolerance has no effect
- #5529: Wrong image/code for legend_demo (pylab)
- #5139: plt.subplots for already existing Figure
- #5497: violin{,plot} return value
- #5441: boxplot rcParams are not in matplotlibrc.template
- #5522: axhline fails on custom scale example
- #5528: \$rho\$ in text for plots erroring
- #4799: Probability axes scales
- #5487: Trouble importing image_comparison decorator in v1.5
- #5464: figaspect not working with numpy floats
- #4487: Should default hist() bins be changed in 2.0?
- #5499: UnicodeDecodeError in IPython Notebook caused by negative numbers in plt.legend()
- #5498: Labels' collisions while plotting named DataFrame iterrows
- #5491: clippedline.py example should be removed
- #5482: RuntimeError: could not open display
- #5481: value error : unknown locale: UTF-8
- #4780: Non-interactive backend calls draw more than 100 times
- #5470: colorbar values could take advantage of offsetting and/or scientific notation
- #5471: FuncAnimation video saving results in one frame file

- #5457: Example of new colormaps is misleading
- #3920: Please fix pip install, so that plt.show() etc works correctly
- #5418: install backend gtk in Cygwin
- #5368: New axes.set_prop_cycle method cannot handle any generic iterable
- #5446: Tests fail to run (killed manually after 7000 sec)
- #5225: Rare race condition in makedirs with parallel processes
- #5444: overline and subscripts/superscripts in mathtext
- #4859: Call `tight_layout()` by default
- #5429: Segfault in `matplotlib.tests.test_image:test_get_window_extent_for_AxisImage` on python3.5
- #5431: Matplotlib 1.4.3 broken on Windows
- #5409: Match zdata cursor display scalling with colorbar ?
- #5128: ENH: Better default font
- #5420: [Mac OS X 10.10.5] Macports install error :unknown locale: UTF-8
- #3867: OSX compile broken since CXX removal (conda only?)
- #5411: XKCD style fails except for inline mode
- #5406: Hangs on OS X 10.11.1: No such file or directory: ‘~/.matplotlib/fontList.json’
- #3116: `mplplot3d`: argument checking in `plot_surface` should be improved.
- #347: Faster Text drawing needed
- #5399: FuncAnimation w/o `init_func` breaks when saving
- #5395: Style changes doc has optimistic release date
- #5393: wrong legend in errorbar plot for pandas series
- #5396: `fill_between()` with gradient
- #5221: infinite range for `hist(histtype="step")`
- #4901: Error running double pendulum animation example
- #3314: assert `mods.pop(0) == 'tests'` errors for multiprocessing tests on OSX
- #5337: Remove `-nocapture` from nosetests on `.travis.yml`?
- #5378: errorbar fails with pandas data frame
- #5367: histogram and digitize do not agree on the definition of a bin
- #5314: `ValueError: insecure string pickle`
- #5347: Problem with importing `matplotlib.animation`
- #4788: Modified axes patch will not re-clip artists
- #4968: Lasso-ing in WxAgg causes flickering of the entire figure

LICENSE

Matplotlib only uses BSD compatible code, and its license is based on the [PSF](#) license. See the Open Source Initiative [licenses page](#) for details on individual licenses. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see [Licenses](#).

11.1 Copyright Policy

John Hunter began matplotlib around 2003. Since shortly before his passing in 2012, Michael Droettboom has been the lead maintainer of matplotlib, but, as has always been the case, matplotlib is the work of many.

Prior to July of 2013, and the 1.3.0 release, the copyright of the source code was held by John Hunter. As of July 2013, and the 1.3.0 release, matplotlib has moved to a shared copyright model.

matplotlib uses a shared copyright model. Each contributor maintains copyright over their contributions to matplotlib. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the matplotlib source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire matplotlib Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the matplotlib repositories.

The Matplotlib Development Team is the set of all contributors to the matplotlib project. A full list can be obtained from the git version control logs.

11.2 License agreement for matplotlib 2.0.0

1. This LICENSE AGREEMENT is between the Matplotlib Development Team (“MDT”), and the Individual or Organization (“Licensee”) accessing and otherwise using matplotlib software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, MDT hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 2.0.0 alone or in any derivative version, provided, however, that MDT’s License Agreement and MDT’s notice of copyright, i.e., “Copyright (c) 2012-2013

Matplotlib Development Team; All Rights Reserved” are retained in matplotlib 2.0.0 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 2.0.0 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 2.0.0.

4. MDT is making matplotlib 2.0.0 available to Licensee on an “AS IS” basis. MDT MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, MDT MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 2.0.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 2.0.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 2.0.0, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between MDT and Licensee. This License Agreement does not grant permission to use MDT trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 2.0.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

11.3 License agreement for matplotlib versions prior to 1.3.0

1. This LICENSE AGREEMENT is between John D. Hunter (“JDH”), and the Individual or Organization (“Licensee”) accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, JDH hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 2.0.0 alone or in any derivative version, provided, however, that JDH’s License Agreement and JDH’s notice of copyright, i.e., “Copyright (c) 2002-2009 John D. Hunter; All Rights Reserved” are retained in matplotlib 2.0.0 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 2.0.0 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 2.0.0.

4. JDH is making matplotlib 2.0.0 available to Licensee on an “AS IS” basis. JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 2.0.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 2.0.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 2.0.0, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between JDH and Licensee. This License Agreement does not grant permission to use JDH trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using matplotlib 2.0.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

**CHAPTER
TWELVE**

CREDITS

Matplotlib was written by John D. Hunter, with contributions from an ever-increasing number of users and developers. The current co-lead developers are Michael Droettboom and Thomas A. Caswell; they are assisted by many [active](#) developers.

The following is a list of contributors extracted from the git revision control history of the project:

4over7, Aaron Boushley, Acanthostega, Adam Ginsburg, Adam Heck, Adam Ortiz, Adrian Price-Whelan, Adrien F. Vincent, Ahmet Bakan, Alan Du, Alejandro Dubrovsky, Alex C. Szatmary, Alex Loew, Alexander Taylor, Alexei Colin, Ali Mehdi, Alistair Muldal, Allan Haldane, Amit Aronovitch, Amy, AmyTeegarden, Andrea Bedini, Andreas Hilboll, Andreas Wallner, Andrew Dawson, Andrew Merrill, Andrew Straw, Andy Zhu, Anton Akhmerov, Antony Lee, Arie, Ariel Hernán Curiale, Arnaud Gardelein, Arpad Horvath, Aseem Bansal, Behram Mistree, Ben Cohen, Ben Gamari, Ben Keller, Ben Root, Benjamin Reedlunn, Binglin Chang, Bradley M. Froehle, Brandon Liu, Brett Cannon, Brett Graham, Brian Mattern, Brian McLaughlin, Bruno Beltran, CJ Carey, Cameron Bates, Cameron Davidson-Pilon, Carissa Brittain, Carl Michal, Carwyn Pelley, Casey Webster, Casper van der Wel, Charles Moad, Chris Beaumont, Chris G, Christian Brueffer, Christian Stade-Schuldt, Christoph Dann, Christoph Gohlke, Christoph Hoffmann, Cimarron Mittelsteadt, Corey Farwell, Craig M, Craig Tenney, Damon McDougall, Dan Hickstein, Daniel Hyams, Daniel O'Connor, Dara Adib, Darren Dale, David Anderson, David Haberthür, David Huard, David Kaplan, David Kua, David Trémouilles, Dean Malmgren, Dmitry Lupyan, DonaldSeo, Dora Fraeman, Duncan Macleod, Edin Salkovic, Elena Glassman, Elias Pipping, Elliott Sales de Andrade, Emil Mikulic, Eric Dill, Eric Firing, Eric Ma, Eric O. LEBIGOT (EOL), Erik Bray, Eugen Beck, Eugene Yurtsev, Evan Davey, Ezra Peisach, Fabien Maussion, Fabio Zanini, Federico Ariza, Felipe, Fernando Perez, Filipe Fernandes, Florian Rhiem, Francesco Montesano, Francis Colas, François Magimel, Gaute Hope, Gellule Xg, Geoffroy Billotey, Gerald Storer, Giovanni, Graham Poulter, Gregory Ashton, Gregory R. Lee, Grégory Lielens, Guillaume Gay, Gustavo Braganca, Hans Dembinski, Hans Meine, Hans Moritz Günther, Hassan Kibirige, Holger Peters, Hubert Holin, Ian Thomas, Ignas Anikevicius (gns_ank), Ilia Kurenkov, Ioannis Filippidis, Ismo Toijala, J. Goutin, Jack Kelly, Jae-Joon Lee, Jaime Fernandez, Jake Vanderplas, James A. Bednar, James Pallister, James R. Evans, JamesMakela, Jan Schulz, Jan-Philip Gehrcke, Jan-willem De Bleser, Jarrod Millman, Jascha Ulrich, Jason Grout, Jason Liw Yan Chong, Jason Miller, JayP16, Jeff Lutgen, Jeff Whitaker, Jeffrey Bingham, Jens Hedegaard Nielsen, Jeremy Fix, Jeremy O'Donoghue, Jeremy Thurgood, Jessica B. Hamrick, Jim Radford, Jochen Voss, Jody Klymak, Joe Kington, Joel B. Mohler, John Hunter, Jonathan Waltman, Jorrit Wronski, Josef Heinen, Joseph Jon Booker, José Ricardo, Jouni K. Seppänen, Julian Mehne, Julian Taylor, JulianCienfuegos, Julien Lhermitte, Julien Schueller, Julien Woillez, Julien-Charles Lévesque, Kanwar245, Katy Huff, Ken McIvor, Kevin Chan, Kevin Davies, Kevin Keating, Kimmo Palin, Konrad Förstner, Konstantin Tretyakov, Kristen M. Thyng, Lance Hepler, Larry Bradley, Leonadoh, Lennart Fricke, Leo Singer, Levi Kilcher, Lion Krischer, Lodato Luciano, Lori J, Loïc Estève, Loïc Séguin-C, Magnus Nord, Majid alDosari, Maksym P, Manuel GOACOLOU, Manuel Metz, Marc Abramowitz, Marcos Duarte, Marek Rud-

nicki, Marianne Corvellec, Marin Gilles, Markus Roth, Markus Rothe, Martin Dengler, Martin Fitzpatrick, Martin Spacek, Martin Teichmann, Martin Thoma, Martin Ueding, Masud Rahman, Mathieu Duponchelle, Matt Giuca, Matt Klein, Matt Li, Matt Newville, Matt Shen, Matt Terry, Matthew Brett, Matthew Emmett, Matthias Bussonnier, Matthieu Caneill, Matěj Týč, Maximilian Albert, Maximilian Trescher, Mellissa Cross, Michael, Michael Droettboom, Michael Sarahan, Michael Welter, Michiel de Hoon, Michka Popoff, Mike Kaufman, Mikhail Korobov, MinRK, Minty Zhang, MirandaXM, Miriam Sierig, Muhammad Mehdi, Neil, Neil Crighton, Nelle Varoquaux, Niall Robinson, Nic Eggert, Nicholas Devenish, Nick Semenkovich, Nicolas P. Rougier, Nicolas Pinto, Nikita Kniazev, Niklas Koep, Nikolay Vyahhi, Norbert Nemec, Ocean-Wolf, Oleg Selivanov, Olga Botvinnik, Oliver Willekens, Parfenov Sergey, Pascal Bugnion, Patrick Chen, Patrick Marsh, Paul, Paul Barret, Paul G, Paul Hobson, Paul Ivanov, Pauli Virtanen, Per Parker, Perry Greenfield, Pete Bachant, Peter Iannucci, Peter St. John, Peter Würtz, Phil Elson, Pierre Haessig, Pim Schellart, Piti Ongmongkolkul, Puneeth Chaganti, Ramiro Gómez, Randy Olson, Reinier Heeres, Remi Rampin, Richard Hattersley, Richard Trieu, Ricky, Robert Johansson, Robin Dunn, Rohan Walker, Roland Wirth, Russell Owen, RutgerK, Ryan Blomberg, Ryan D'Souza, Ryan Dale, Ryan May, Ryan Nelson, RyanPan, Salil Vanvari, Sameer D'Costa, Sandro Tosi, Scott Lasley, Scott Lawrence, Scott Stevenson, Sebastian Pinnau, Sebastian Raschka, Sergey Kholodilov, Sergey Koposov, Silviu Tantos, Simon Cross, Simon Gibbons, Skelpdar, Skipper Seabold, Slav Basharov, Spencer McIntyre, Stanley, Simon, Stefan Lehmann, Stefan van der Walt, Stefano Rivera, Stephen Horst, Sterling Smith, Steve Chaplin, Steven Sylvester, Stuart Mumford, Takafumi Arakaki, Takeshi Kanmae, Tamas Gal, Thomas A Caswell, Thomas Hisch, Thomas Kluyver, Thomas Lake, Thomas Robitaille, Thomas Spura, Till Stensitzki, Timo Vanwynsberghe, Tobias Hoppe, Tobias Megies, Todd Jennings, Todd Miller, Tomas Kazmar, Tony S Yu, Tor Colvin, Travis Oliphant, Trevor Bekolay, Ulrich Dobramysl, Umair Idris, Vadim Markovtsev, Valentin Haenel, Victor Zabalza, Viktor Kerkez, Vlad Seghete, Víctor Terrón, Víctor Zabalza, Wen Li, Wendell Smith, Werner F Bruhin, Wes Campagne, Wieland Hoffmann, William Manley, Wouter Overmeire, Xiaowen Tang, Yann Tambouret, Yaron de Leeuw, Yu Feng, Yunfei Yang, Yuri D'Elia, Yuval Langer, Zach Pincus, Zair Mubashar, alex, anykraus, arokem, aseagram, aszilagy, bblay, bev-a-tron, blackw1ng, blah blah, burrbull, butterw, cammil, captainwhippet, chadawagner, chebee7i, danielballan, davidovitch, daydreamt, domspad, donald, drevicko, e-q, elpres, endolith, fardal, ffteja, fgb, fibersnet, frenchwr, fvgoto, gitj, gluap, goir, hugadams, insertroar, itziakos, jbbrokaw, juan.gonzalez, kcrisman, kelsiegr, khyox, kikocorreoso, kramer65, kshramt, lichri12, limtaesu, marky, masamson, mbyt, mcelrath, mdipierro, mrkrd, nickystringer, nwin, pkienzle, profholzer, pupssman, rahiels, rhoef, rsnape, s9w, sdementen, sfroid, sohero, spiessbuerger, stahlous, switham, syngron, torfbolt, u55, ugurthemaster, vbr, xbtsw, and xuanyansen.

Some earlier contributors not included above are (with apologies to any we have missed):

Charles Twardy, Gary Ruben, John Gill, David Moore, Paul Barrett, Jared Wahlstrand, Jim Benson, Paul McGuire, Andrew Dalke, Nadia Dencheva, Baptiste Carvello, Sigve Tjoraand, Ted Drain, James Amundson, Daishi Harada, Nicolas Young, Paul Kienzle, John Porter, and Jonathon Taylor.

We also thank all who have reported bugs, commented on proposed changes, or otherwise contributed to Matplotlib's development and usefulness.