

# Python Plotting With Matplotlib (Guide)

by Brad Solomon 30 Comments basics data-science

Mark as Completed

Tweet

Share

Email

## Table of Contents

- [Why Can Matplotlib Be Confusing?](#)
- [Pylab: What Is It, and Should I Use It?](#)
- [The Matplotlib Object Hierarchy](#)
- [Stateful Versus Stateless Approaches](#)
- [Understanding plt.subplots\(\) Notation](#)
- [The “Figures” Behind The Scenes](#)
- [A Burst of Color: imshow\(\) and matshow\(\)](#)
- [Plotting in Pandas](#)
- [Wrapping Up](#)
- [More Resources](#)
- [Appendix A: Configuration and Styling](#)
- [Appendix B: Interactive Mode](#)

Master **Real-World Python Skills**  
With a Community of Experts

Level Up With Unlimited Access to Our Vast Library  
of Python Tutorials and Video Lessons

**Watch Now »**

Remove ads

Watch Now

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Plotting With Matplotlib](#)

A picture is worth a thousand words, and with Python’s **matplotlib** library, it fortunately takes far less than a thousand words of code to create a production-quality graphic.

However, matplotlib is also a massive library, and getting a plot to look just right is often achieved through trial and error. Using one-liners to generate basic plots in matplotlib is fairly simple, but skillfully commanding the remaining 98% of the library can be daunting.

Improve Your Python

This article is a beginner-to-intermediate-level walkthrough on matplotlib that mixes theory with examples. While learning by example can be tremendously insightful, it helps to have even just a surface-level understanding of the library’s inner workings and layout as well.

Here’s what we’ll cover:

- Pylab and pyplot: which is which?
- Key concepts of matplotlib’s design
- Understanding `plt.subplots()`
- Visualizing arrays with matplotlib
- Plotting with the pandas + matplotlib

**Free Bonus:** [Click here to download!](#)  
as a basis for making your own plots a

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

This article assumes the user knows a tin data, drawing samples from different [stat](#)

If you don’t already have matplotlib insta

## Why Can Matplotlib Be

Learning matplotlib can be a frustrating r  
lacking: the documentation is actually extensive. But the following issues can cause some challenges:


- The library itself is huge, at something like 70,000 total lines of code.
- Matplotlib is home to several different interfaces (ways of constructing a figure) and capable of interacting with a handful of different backends. (Backends deal with the process of how charts are actually rendered, not just structured internally.)
- While it is comprehensive, some of matplotlib’s own public documentation is seriously [out-of-date](#). The library is still evolving, and many older examples floating around online may take 70% fewer lines of code in their modern version.

So, before we get to any glitzy examples, it’s useful to grasp the core concepts of matplotlib’s design.

Real Python  
MERCH STORE



SHOP NOW >>

 [Remove ads](#)

## Pylab: What Is It, and Should I Use It?

Let’s start with a bit of history: John D. Hunter, a neurobiologist, began developing matplotlib around 2003, originally inspired to emulate commands from Mathworks’ [MATLAB](#) software. John passed away tragically young at age 44, in 2012, and matplotlib is now a full-fledged community effort, developed and maintained by a host of others. (John gave a [talk](#) about the evolution of matplotlib at the 2012 SciPy conference, which is worth a watch.)



One relevant feature of MATLAB is its global style. The Python concept of importing is not heavily used in MATLAB, and most of MATLAB’s [functions](#) are readily available to the user at the top level.

Knowing that matplotlib has its roots in MATLAB helps to explain why pylab exists. pylab is a module within the matplotlib library that was built to mimic MATLAB’s global style. It exists only to bring a number of functions and classes from both NumPy and matplotlib into the [namespace](#), making for an easy transition for former MATLAB users who were not used to needing `import` statements.

Ex-MATLAB converts (who are all fine people, I promise!) liked this functionality, because with `from pylab import *`, they could simply call `plot()` or `array()` directly, as they would in MATLAB.

The issue here may be apparent to some Python users: using `from pylab import *` in a session or script is generally bad practice. Matplotlib now directly advises against this in [Improve Your Python](#)

### Improve Your Python

...with a fresh  **Python Trick**   
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

“[pylab] still exists for historical reasons, but it is highly advised not to use. It pollutes namespaces with functions that will shadow Python built-ins and can lead to hard-to-track bugs. To get IPython integration without imports the use of the `%matplotlib` magic is preferred.” [\[Source\]](#)

Internally, there are a ton of potentially conflicting imports being masked within the short `pylab` [source](#). In fact, using `ipython --pylab` (from the terminal/command line) or `%pylab` (from IPython/Jupyter tools) simply calls `from pylab import *` under the hood.

The bottom line is that **matplotlib has abandoned this convenience module and now explicitly recommends against using pylab**, bringing things mo

Without the need for `pylab`, we can usual

```
Python
>>> import matplotlib.pyplot as plt
```

While we’re at it, let’s also import NumPy make examples with (pseudo)random da

```
Python
>>> import numpy as np
>>> np.random.seed(444)
```

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh Python Trick

code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

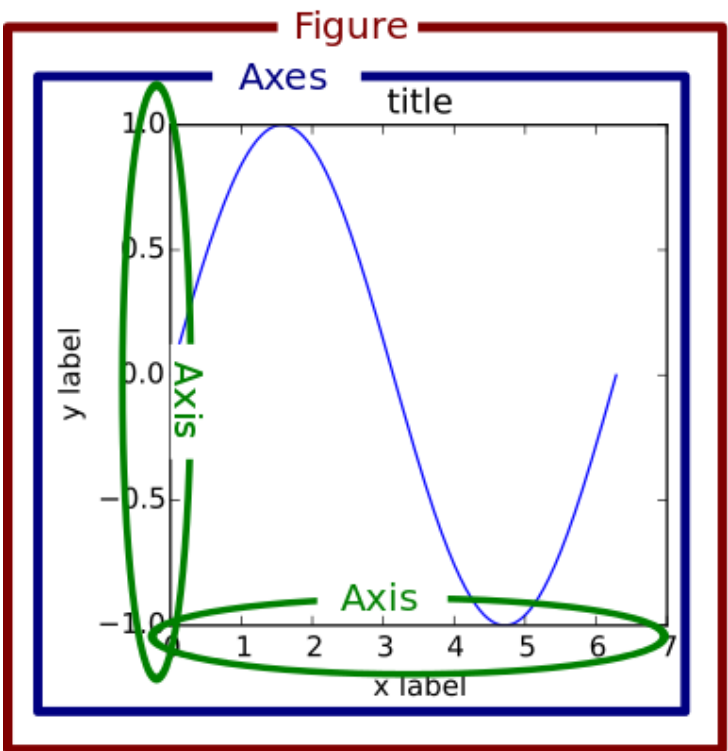
# The Matplotlib Object Hierarchy

One important big-picture matplotlib concept is its object hierarchy.

If you’ve worked through any introductory matplotlib tutorial, you’ve probably called something like `plt.plot([1, 2, 3])`. This one-liner hides the fact that a plot is really a hierarchy of nested Python objects. A “hierarchy” here means that there is a tree-like structure of matplotlib objects underlying each plot.

A `Figure` object is the outermost container for a matplotlib graphic, which can contain multiple `Axes` objects. One source of confusion is the name: an `Axes` actually translates into what we think of as an individual plot or graph (rather than the plural of “axis,” as we might expect).

You can think of the `Figure` object as a box-like container holding one or more `Axes` (actual plots). Below the `Axes` in the hierarchy are smaller objects such as tick marks, individual lines, legends, and text boxes. Almost every “element” of a chart is its own manipulable Python object, all the way down to the ticks and labels:



Here’s an illustration of this hierarchy in action. Don’t worry if you’re not completely familiar with this notation, which we’ll cover later on:

Python

>>>

```
>>> fig, _ = plt.subplots()
>>> type(fig)
<class 'matplotlib.figure.Figure'>
```

Above, we created two [variables](#) with `plt.subplots()`. The first is a top-level Figure object. The second is a “throwaway” variable that we don’t need just yet, denoted with an underscore. Using attribute notation, it is easy to traverse down the figure hierarchy and see the first tick of the y axis of the first Axes object:

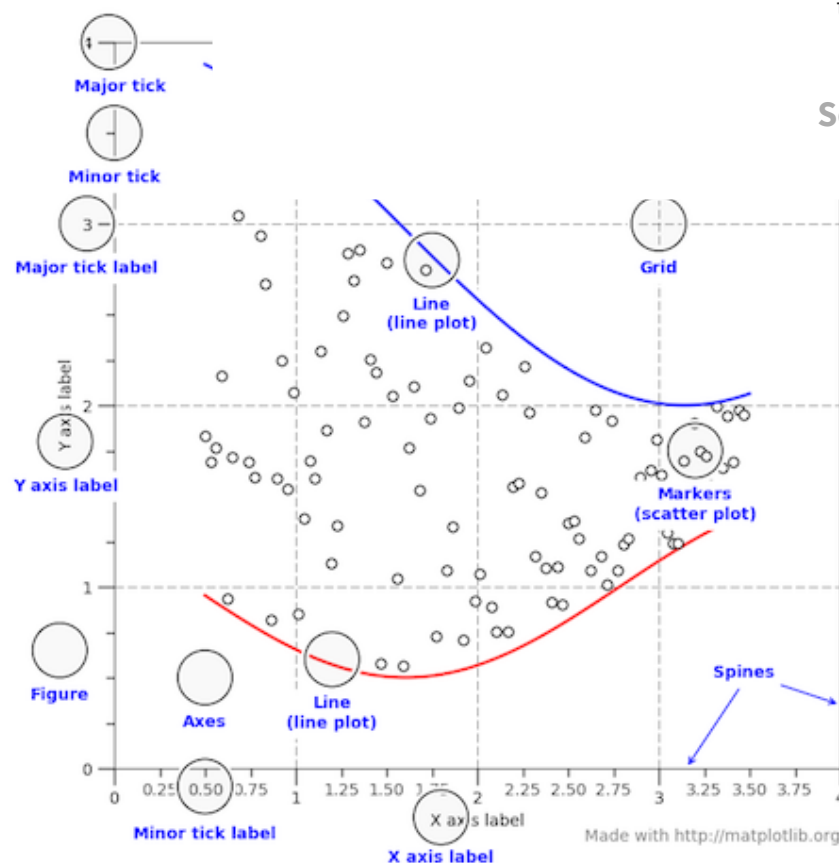
Python

>>>

```
>>> one_tick = fig.axes[0].yaxis.get_major_ticks()[0].label
>>> type(one_tick)
<class 'matplotlib.axis.YTick'>
```

Above, `fig` (a Figure class instance) has a `axes` attribute, which is a list of Axes objects. Each Axes object has a `yaxis` and `xaxis`, each of which have a `get_major_ticks()` method.

Matplotlib presents this as a figure anatomy diagram:



## Improve Your Python

...with a fresh 🐍 **Python Trick** ❤️  
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

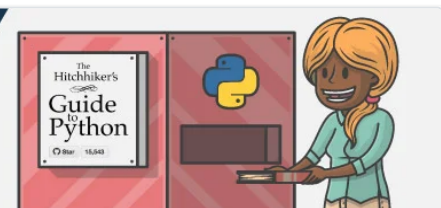
Send Python Tricks »

(In true matplotlib style, the figure above is created in the matplotlib docs [here](#).)

## A Python Best Practices Handbook

[python-guide.org](http://python-guide.org)

[Remove ads](#)



## Stateful Versus Stateless Approaches

Alright, we need one more chunk of theory before we can get around to the shiny visualizations: the difference between the stateful (state-based, state-machine) and stateless ([object-oriented](#), OO) interfaces.

Above, we used `import matplotlib.pyplot as plt` to import the pyplot module from matplotlib and name it `plt`.

Almost all functions from pyplot, such as `plt.plot()`, are implicitly either referring to an existing current Figure and current Axes, or creating them anew if none exist. Hidden in the matplotlib docs is this helpful snippet:

“[With pyplot], simple functions are used to add plot elements (lines, images, text, etc.) **to the current axes in the current figure.**” [emphasis added]

Improve Your Python



Hardcore ex-MATLAB users may choose to word this by saying something like, “`plt.plot()` is a state-machine interface that implicitly tracks the current figure!” In English, this means that:

- The stateful interface makes its calls with `plt.plot()` and other top-level pyplot functions. There is only ever one Figure or Axes that you’re manipulating at a given time, and you don’t need to explicitly refer to it.
- Modifying the underlying objects directly is the object-oriented approach. We usually do this by calling methods of an Axes object, which is the object that represents a plot itself.



The flow of this process, at a high level, looks like this:

plt.plot()

• Top-level function that adheres to the stateful interface.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  Python Trick  code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

Tying these together, most of the function

This is easier to see by peeking under the

Python

```
# matplotlib/pyplot.py
>>> def plot(*args, **kwargs):
...     """An abridged version of
...     ax = plt.gca()
...     return ax.plot(*args, **kw

>>> def gca(**kwargs):
...     """Get the current Axes of the current Figure."""
...     return plt.gcf().gca(**kwargs)
```

Calling `plt.plot()` is just a convenient way to get the current Axes of the current Figure and then call its `plot()` method. This is what is meant by the assertion that the stateful interface always “implicitly tracks” the plot that it wants to reference.

pyplot is home to a [batch of functions](#) that are really just wrappers around matplotlib’s object-oriented interface. For example, with `plt.title()`, there are corresponding setter and getter methods within the OO approach, `ax.set_title()` and `ax.get_title()`. (Use of getters and setters tends to be more popular in languages such as [Java](#) but is a key feature of matplotlib’s OO approach.)

Calling `plt.title()` gets translated into this one line: `gca().set_title(s, *args, **kwargs)`. Here’s what that is doing:

- `gca()` grabs the current axis and returns it.
- `set_title()` is a setter method that sets the title for that Axes object. The “convenience” here is that we didn’t need to specify any Axes object explicitly with `plt.title()`.

Similarly, if you take a few moments to look at the source for top-level functions like [plt.grid\(\)](#), [plt.legend\(\)](#), and [plt.ylabels\(\)](#), you’ll notice that all of them follow the same structure of delegating to the current Axes with `gca()` and then calling some method of the current Axes. (This is the underlying object-oriented approach!)

## Understanding `plt.subplots()` Notation

Alright, enough theory. Now, we’re ready to tie everything together and do some plotting. From here on out, we’ll mostly rely on the stateless (object-oriented) approach, which is more customizable and comes in handy as graphs become more complex.

The prescribed way to create a Figure with a single Axes under the OO approach is (not too intuitively) with `plt.subplots()`. This is really the only time that the OO approach uses `pyplot`, to create a Figure and Axes:

Python

```
>>> fig, ax = plt.subplots()
```

Above, we took advantage of iterable unpacking to assign a `plt.subplots()`. Notice that we didn’t pass arguments to `si`

ncols=1). Consequently, ax is a single AxesSubplot object:

Python>>>

>>> type(ax)  
<class 'matplotlib.axes.\_subplots.AxesSubplot'>



We can call its instance methods to manipulate the plot similarly to how we call pyplots functions. Let’s illustrate with a stacked area graph of three time series:

Python>>>

>>> rng = np.arange(50)  
>>> rnd = np.random.randint(0, 10,  
>>> yrs = 1950 + rng  
  
>>> fig, ax = plt.subplots(figsize=  
>>> ax.stackplot(yrs, rng + rnd, 1  
>>> ax.set\_title('Combined debt gr  
>>> ax.legend(loc='upper left')  
>>> ax.set\_ylabel('Total debt')  
>>> ax.set\_xlim(xmin=yrs[0], xmax=  
>>> fig.tight\_layout()

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  Python Trick 

code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

Here’s what’s going on above:

- After creating three random time se
- We call methods of ax directly to create a stacked area chart and to add a legend, title, and y-axis label. Under the object-oriented approach, it’s clear that all of these are attributes of ax.
- tight\_layout() applies to the Figure object as a whole to clean up whitespace padding.

Let’s look at an example with multiple subplots (Axes) within one Figure, plotting two correlated arrays that are drawn from the [discrete uniform distribution](#):

Python>>>

>>> x = np.random.randint(low=1, high=11, size=50)  
>>> y = x + np.random.randint(1, 5, size=x.size)  
>>> data = np.column\_stack((x, y))  
  
>>> fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,  
... figsize=(8, 4))  
  
>>> ax1.scatter(x=x, y=y, marker='o', c='r', edgecolor='b')  
>>> ax1.set\_title('Scatter: \$\$ versus \$\$')  
>>> ax1.set\_xlabel('\$x\$')  
>>> ax1.set\_ylabel('\$y\$')  
  
>>> ax2.hist(data, bins=np.arange(data.min(), data.max()),  
... label=('\$x', 'y'))  
>>> ax2.legend(loc=(0.65, 0.8))

Improve Your Python

```
>>> ax2.set_title('Frequencies of $x$ and $y$')
>>> ax2.yaxis.tick_right()
```

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick** 

code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

There’s a little bit more going on in this e:

- Because we’re creating a “1x2” Figure, we need to create a NumPy array of Axes objects. (You can see this in the code above.)
- We deal with ax1 and ax2 individually, which is a good illustration of the object hierarchy, where we are modifying the yaxis belonging to the second Axes, placing its ticks and ticklabels to the right.
- Text inside dollar signs utilizes [TeX markup](#) to put variables in italics.

Remember that multiple Axes can be enclosed in or “belong to” a given figure. In the case above, fig.axes gets us a list of all the Axes objects:

Python

>>> (fig.axes[0] is ax1, fig.axes[1] is ax2)
(True, True)

(fig.axes is lowercase, not uppercase. There’s no denying the terminology is a bit confusing.)

Taking this one step further, we could alternatively create a figure that holds a 2x2 grid of Axes objects:

Python

>>> fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(7, 7))

Now, what is ax? It’s no longer a single Axes, but a two-dimensional NumPy array of them:

Python

>>> type(ax)
numpy.ndarray

>>> ax
array([[<matplotlib.axes.\_subplots.AxesSubplot object at 0x1106daf98>,
 <matplotlib.axes.\_subplots.AxesSubplot object at 0x113045c88>],
 [<matplotlib.axes.\_subplots.AxesSubplot object at 0x11d573cf8>,
 <matplotlib.axes.\_subplots.AxesSubplot object at 0x1130117f0>]],
 dtype=object)

>>> ax.shape
(2, 2)

This is reaffirmed by the docstring:

“ax can be either a single matplotlib.axes.Axes object or a 2D array of them, as created by plt.subplots()”

Improve Your Python

was created.

We now need to call plotting methods on each of these Axes (but not the NumPy array, which is just a container in this case). A common way to address this is to use iterable unpacking after flattening the array to be one-dimensional:

Python>>>

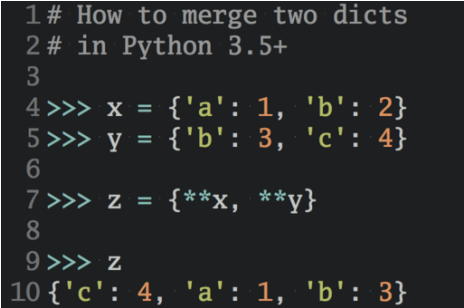
>>> fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(7, 7))  
>>> ax1, ax2, ax3, ax4 = ax.flatten() # flatten a 2d NumPy array to 1d

We could’ve also done this with `((ax1, ax2), (ax3, ax4)) = ax`, but the first approach tends to be more flexible.



To illustrate some more advanced subplo  
from a compressed tar archive, using `io`,

Python

>>> from io import BytesIO  
>>> import tarfile  
>>> from urllib.request import urlopen  
  
>>> url = 'http://www.dcc.fc.up.pt  
>>> b = BytesIO(urlopen(url).read()  
>>> fpath = 'CaliforniaHousing/cal  
  
>>> with tarfile.open(mode='r', fi  
... housing = np.loadtxt(archi



Improve Your Python

...with a fresh  Python Trick   
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

The “response” variable `y` below, to use the statistical term, is an area’s average home value. `pop` and `age` are the area’s population and average house age, respectively:

Python>>>

>>> y = housing[:, -1]  
>>> pop, age = housing[:, [4, 7]].T

Next let’s define a “helper function” that places a text box inside of a plot and acts as an “in-plot title”:

Python>>>

>>> def add\_titlebox(ax, text):  
... ax.text(.55, .8, text,  
... horizontalalignment='center',  
... transform=ax.transAxes,  
... bbox=dict(facecolor='white', alpha=0.6),  
... fontsize=12.5)  
... return ax

We’re ready to do some plotting. Matplotlib’s `gridspec` module allows for more subplot customization. pyplot’s `subplot2grid()` interacts with this module nicely. Let’s say we want to create a layout like this:



Above, what we actually have is a 3x2 grid. ax1 is twice the height and width of ax2/ax3, meaning that it takes up two columns and two rows.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh 🐍 Python Trick 📧  
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

The second argument to subplot2grid(

Python

```
>>> gridsize = (3, 2)
>>> fig = plt.figure(figsize=(12, 8))
>>> ax1 = plt.subplot2grid(gridsize, (0, 0), colspan=2, rowspan=2)
>>> ax2 = plt.subplot2grid(gridsize, (2, 0))
>>> ax3 = plt.subplot2grid(gridsize, (2, 1))
```

Now, we can proceed as normal, modifying each Axes individually:

Python

```
>>> ax1.set_title('Home value as a function of home age & area population',
...               fontsize=14)
>>> sctr = ax1.scatter(x=age, y=pop, c=y, cmap='RdYlGn')
>>> plt.colorbar(sctr, ax=ax1, format='$%d')
>>> ax1.set_yscale('log')
>>> ax2.hist(age, bins='auto')
>>> ax3.hist(pop, bins='auto', log=True)

>>> add_titlebox(ax2, 'Histogram: home age')
>>> add_titlebox(ax3, 'Histogram: area population (log scl.)')
```

Above, `colorbar()` (different from `ColorMap` earlier) gets called on the Figure directly, rather than the Axes. Its first argument uses Matplotlib's `.scatter()` and is the result of `ax1.scatter()`, which functions as a mapping of y-values to a `ColorMap`.

Visually, there isn't much differentiation in color (the y-variable) as we move up and down the y-axis, indicating that home age seems to be a stronger determinant of house value.

Your Guide to the Python Language and a Best Practicepython-guide.org

Remove ads

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh Python Trick code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

## The “Figures” Behind

Each time you call `plt.subplots()` or the you are creating a new Figure object that concept of a current Figure and current A we can show with the built-in function `id`

Python

```
>>> fig1, ax1 = plt.subplots()

>>> id(fig1)
4525567840

>>> id(plt.gcf()) # `fig1` is the current figure.
4525567840

>>> fig2, ax2 = plt.subplots()
>>> id(fig2) == id(plt.gcf()) # The current figure has changed to `fig2`.
True
```

(We could also use the built-in [is operator](#) here.)

After the above routine, the current figure is `fig2`, the most recently created figure. However, both figures are still hanging around in memory, each with a corresponding ID number (1-indexed, in MATLAB style):

Python

```
>>> plt.get_fignums()
[1, 2]
```

A useful way to get all of the Figures themselves is with a mapping of `plt.figure()` to each of these integers:

Python

```
>>> def get_all_figures():
...     return [plt.figure(i) for i in plt.get_fignums()]

>>> get_all_figures()
[<matplotlib.figure.Figure at 0x10dbeaf60>,
 <matplotlib.figure.Figure at 0x1234cb6d8>]
```

Be cognizant of this if running a script where you're creating a group of figures. You'll want to explicitly close each of them after use to avoid a `MemoryError`. By itself, `plt.close()` closes the current figure, `plt.close(num)` closes the figure number `num`, and `plt.close('all')` closes all the figure windows:

Python

>>>

```
>>> plt.close('all')
>>> get_all_figures()
[]
```

## A Burst of Color: imshow() and matshow()

While `ax.plot()` is one of the most common plotting methods on an Axes, there are a whole host of others, as well. (We used `ax.stackplot()` above. You can find more information about `ax.plot()` [here](#).)

Methods that get heavy use are `imshow()` and `matshow()`. These methods are useful anytime that a raw numerical array needs to be visualized.

First, let’s create two distinct grids with some data.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

### Improve Your Python

...with a fresh 🐍 Python Trick 📧  
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

Python

```
>>> x = np.diag(np.arange(2, 12))
>>> x[np.diag_indices_from(x[:-1])] = 0
>>> x2 = np.arange(x.size).reshape(x.shape)
```

Next, we can map these to their image representations. In this specific case, we toggle “off” all axis labels and ticks by using a dictionary comprehension and passing the result to `ax.tick_params()`:

Python

>>>

```
>>> sides = ('left', 'right', 'top', 'bottom')
>>> nolabels = {s: False for s in sides}
>>> nolabels.update({'label%s' % s: False for s in sides})
>>> print(nolabels)
{'left': False, 'right': False, 'top': False, 'bottom': False, 'labelleft': False,
 'labelright': False, 'labeltop': False, 'labelbottom': False}
```

Then, we can use a [context manager](#) to disable the grid, and call `matshow()` on each Axes. Lastly, we need to put the colorbar in what is technically a new Axes within `fig`. For this, we can use a bit of an esoteric function from deep within matplotlib:

Python

>>>

```
>>> from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable

>>> with plt.rc_context(rc={'axes.grid': False}):
...     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
...     ax1.matshow(x)
...     img2 = ax2.matshow(x2, cmap='RdYlGn_r')
...     for ax in (ax1, ax2):
...         ax.tick_params(axis='both', which='both', **nolabels)
...     for i, j in zip(*x.nonzero()):
...         ax1.text(j, i, x[i, j], color='white', ha='center', va='center')
...
...     divider = make_axes_locatable(ax2)
...     cax = divider.append_axes("right", size='5%', pad=0)
...     plt.colorbar(img2, cax=cax, ax=[ax1, ax2])
...     fig.suptitle('Heatmaps with `Axes.matshow`', fontsize=16)
```



[Remove ads](#)

# Plotting in Pandas

The pandas library has become popular for its canned plotting methods. Interestingly though, it doesn't replace existing matplotlib calls.

That is, the `plot()` method on pandas' `Series` objects, provided, for example, is that if the `DataFrame` is plotted by pandas to get the current `Figure` and `Axis`.

In turn, remember that `plt.plot()` (the state-based approach) is implicitly aware of the current `Figure` and current `Axes`, so pandas is following the state-based approach by extension.

We can prove this “chain” of function calls with a bit of introspection. First, let's construct a plain-vanilla pandas `Series`, assuming we're starting out in a fresh interpreter session:

Python>>>

```
>>> import pandas as pd

>>> s = pd.Series(np.arange(5), index=list('abcde'))
>>> ax = s.plot()

>>> type(ax)
<matplotlib.axes._subplots.AxesSubplot at 0x121083eb8>

>>> id(plt.gca()) == id(ax)
True
```

This internal architecture is helpful to know when you are mixing pandas plotting methods with traditional matplotlib calls, which is done below in plotting the moving average of a widely watched financial time series. `ma` is a pandas `Series` for which we can call `ma.plot()` (the pandas method), and then customize by retrieving the `Axes` that is created by this call (`plt.gca()`), for matplotlib to reference:

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

## Improve Your Python

...with a fresh **Python Trick** code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

[Send Python Tricks »](#)



Python

&gt;&gt;&gt;

```
>>> import pandas as pd
>>> import matplotlib.transforms as mtransforms

>>> url = 'https://fred.stlouisfed.org/graph/fredgraph.csv?id=VIXCLS'
>>> vix = pd.read_csv(url, index_col=0, parse_dates=True, na_values='.',
...                   infer_datetime_format=True,
...                   squeeze=True).dropna()
>>> ma = vix.rolling('90d').mean()
>>> state = pd.cut(ma, bins=[-np.inf, 14, 18, 24, np.inf],
...                labels=range(4))

>>> cmap = plt.get_cmap('RdYlGn_r')
>>> ma.plot(color='black', linewidth=1,
...         label='VIX 90d MA')
>>> ax = plt.gca() # Get the current Axes
>>> ax.set_xlabel('')
>>> ax.set_ylabel('90d moving average')
>>> ax.set_title('Volatility Regime')
>>> ax.grid(False)
>>> ax.legend(loc='upper center')
>>> ax.set_xlim(xmin=ma.index[0],

>>> trans = mtransforms.blended_transform_factory(ax.transData, ax.transFigure)
>>> for i, color in enumerate(cmap.colors[0:4]):
...     ax.fill_between(ma.index[0], ma.index[0] + 1000,
...                     facecolor=color,
...                     label=f'state {i}')
>>> ax.axhline(vix.mean(), linestyle='dashed',
...            alpha=0.6, label='VIX Mean')
```

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

## Improve Your Python



...with a fresh **Python Trick** every couple of days:

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

[Send Python Tricks »](#)

There's a lot happening above:

- `ma` is a 90-day moving average of the VIX Index, a measure of market expectations of near-term stock volatility. `state` is a binning of the moving average into different regime states. A high VIX is seen as signaling a heightened level of fear in the marketplace.
- `cmap` is a `ColorMap`—a matplotlib object that is essentially a mapping of floats to RGBA colors. Any colormap can be reversed by appending `'_r'`, so `'RdYlGn_r'` is the reversed Red-Yellow-Green colormap. Matplotlib maintains a handy [visual reference guide](#) to ColorMaps in its docs.
- The only real pandas call we're making here is `ma.plot()`. This calls `plt.plot()` internally, so to integrate the object-oriented approach, we need to get an explicit reference to the current Axes with `ax = plt.gca()`.
- The second chunk of code creates color-filled blocks that correspond to each bin of `state`. `cmap([0.2, 0.4, 0.6, 0.8])` says, "Get us an RGBA sequence for the colors at the 20th, 40th, 60th, and 80th 'percentile' along the ColorMaps' spectrum." `enumerate()` is used because we want to map each RGBA color back to a state.

Pandas also comes built-out with a smattering of [more advanced plots](#) (which could take up an entire tutorial all on their own). However, all of these, like their simpler counterparts, rely on matplotlib machinery internally.

Improve Your Python

## Wrapping Up

As shown by some of the examples above, there’s no getting around the fact that matplotlib can be a technical, syntax-heavy library. Creating a production-ready chart sometimes requires a half hour of Googling and combining a hodgepodge of lines in order to fine-tune a plot.

However, understanding how matplotlib’s interfaces interact is an investment that can pay off down the road. As Real Python’s own Dan Bader has advised, taking the time to dissect code rather than resorting to the Stack Overflow “copy pasta” solution tends to be a smarter long-term solution. Sticking to the object-oriented approach can save hours of frustration when you want to take a plot from plain to a work of art.

## More Resources

From the matplotlib documentation:

- An index of matplotlib [examples](#)
- The usage [FAQ](#)
- The [tutorials](#) page, which is broken
- [Lifecycle of a plot](#), which touches o

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

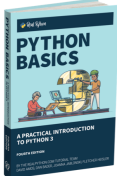
**Free Bonus:** [Click here to download!](#) as a basis for making your own plots a

Third-party resources:

- DataCamp’s matplotlib [cheat sheet](#)
- PLOS Computational Biology: [Ten Simple Rules for Better Figures](#)
- Chapter 9 (Plotting & Visualization) of Wes McKinney’s [Python for Data Analysis, 2nd ed.](#)
- Chaper 11 (Visualization with Matplotlib, Pandas, and Seaborn) of Ted Petrou’s [Pandas Cookbook](#)
- Section 1.4 (Matplotlib: Plotting) of the [SciPy Lecture Notes](#)
- The [xkcd](#) color palette
- The matplotlib [external resources](#) page
- [Matplotlib, Pylab, Pyplot, etc: What’s the difference between these and when to use each?](#) from queirozf.com
- The [visualization](#) page in the pandas documentation

Other plotting libraries:

- The [seaborn](#) library, built on top of matplotlib and designed for advanced statistical graphics, which could take up an entire tutorial all on its own
- [Datashader](#), a graphics library geared specifically towards large datasets
- A list of [other third-party packages](#) from the matplotlib documentation



### Your **Practical Introduction to Python 3** »

 [Remove ads](#)

## Appendix A: Configuration and Styling

If you’ve been following along with this tutorial, it’s likely that the plots popping up on your screen look different stylistically than the ones shown here.

Matplotlib offers two ways to configure style in a uniform way across different plots:

1. By customizing a [matplotlibrc](#) file
2. By changing your configuration parameters interactively, or from a `.py` script.

A matplotlibrc file (Option #1 above) is basically a text file sp between Python sessions. On Mac OS X, this normally reside

## Improve Your Python

...with a fresh 🐍 **Python Trick** ❤️  
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

[Send Python Tricks »](#)

**Quick Tip:** GitHub is a great place to keep configuration files. I keep mine [here](#). Just make sure that they don’t contain personally identifiable or private information, such as passwords or SSH private keys!

Alternatively, you can change your configuration parameters interactively (Option #2 above). When you import `matplotlib.pyplot` as `plt`, you get access to an `rcParams` object that resembles a Python dictionary of settings. All of the module objects starting with “rc” are a means to interact with your plot styles and settings:

Python>>>

>>> [attr for attr in dir(plt) if  
['rc', 'rcParams', 'rcParamsDefaul]

Of these:

- `plt.rcParamsDefault` restores the rc p  
`plt.rcParamsDefault`. This will rev
- `plt.rc()` is used for setting parame
- `plt.rcParams` is a (mutable) dictior  
customized settings in a matplotlibli

```
1# How to merge two dicts  
2# in Python 3.5+  
3  
4>>> x = {'a': 1, 'b': 2}  
5>>> y = {'b': 3, 'c': 4}  
6  
7>>> z = {**x, **y}  
8  
9>>> z  
10{'c': 4, 'a': 1, 'b': 3}
```

## Improve Your Python

...with a fresh  **Python Trick**   
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

With `plt.rc()` and `plt.rcParams`, these

Python

>>> plt.rc('lines', linewidth=2, color='r') # Syntax 1  
  
>>> plt.rcParams['lines.linewidth'] = 2 # Syntax 2  
>>> plt.rcParams['lines.color'] = 'r'

Notably, the `Figure` class then [uses some of these](#) as its default arguments.

Relatedly, a style is just a predefined cluster of custom settings. To view available styles, use:

Python>>>

>>> plt.style.available  
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',  
'seaborn-whitegrid', 'classic', '\_classic\_test', 'fast', 'seaborn-talk',  
'seaborn-dark-palette', 'seaborn-bright', 'seaborn-pastel', 'grayscale',  
'seaborn-notebook', 'ggplot', 'seaborn-colorblind', 'seaborn-muted',  
'seaborn', 'Solarize\_Light2', 'seaborn-paper', 'bmh', 'seaborn-white',  
'dark\_background', 'seaborn-poster', 'seaborn-deep']

To set a style, make this call:

Python>>>

>>> plt.style.use('fivethirtyeight')

Your plots will now take on a new look:

This full example is available [here](#).

For inspiration, matplotlib keeps some [style sheet displays](#) for reference as well.

# Your Weekly Dose of All Things Python!

pycoders.com



 [Remove ads](#)

## Appendix B: Interactiv

Behind the scenes, matplotlib also intera rendering a chart. (On the popular Anaco backends are interactive, meaning they a

While interactive mode is off by default, y `plt.isinteractive()`, and toggle it on a

Python

```
>>> plt.rcParams['interactive'] # True
```

Python

```
>>> plt.ioff()
>>> plt.rcParams['interactive']
False
```

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

## Improve Your Python ×

...with a fresh  **Python Trick**   
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

[Send Python Tricks »](#)

In some code examples, you may notice the presence of `plt.show()` at the end of a chunk of code. The main purpose of `plt.show()`, as the name implies, is to actually “show” (open) the figure when you’re running with interactive mode turned off. In other words:

- If interactive mode is on, you don’t need `plt.show()`, and images will automatically pop-up and be updated as you reference them.
- If interactive mode is off, you’ll need `plt.show()` to display a figure and `plt.draw()` to update a plot.


Below, we make sure that interactive mode is off, which requires that we call `plt.show()` after building the plot itself:


Python >>>

```
>>> plt.ioff()
>>> x = np.arange(-4, 5)
>>> y1 = x ** 2
>>> y2 = 10 / (x ** 2 + 1)
>>> fig, ax = plt.subplots()
>>> ax.plot(x, y1, 'rx', x, y2, 'b+', linestyle='solid')
>>> ax.fill_between(x, y1, y2, where=y2>y1, interpolate=True,
...               color='green', alpha=0.3)
>>> lgnd = ax.legend(['y1', 'y2'], loc='upper center', shadow=True)
>>> lgnd.get_frame().set_facecolor('#ffb19a')
>>> plt.show()
```

Notably, interactive mode has nothing to do with what IDE you’re using, or whether you’ve enable inline plotting with something like `jupyter notebook --matplotlib inline` or `%matplotlib`.

Mark as Completed



 Watch Now

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Plotting With Matplotlib](#)



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

Email Address

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh 🐍 Python Trick ❤️

code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

About Brad Solomon

Brad

[» More about Brad](#)

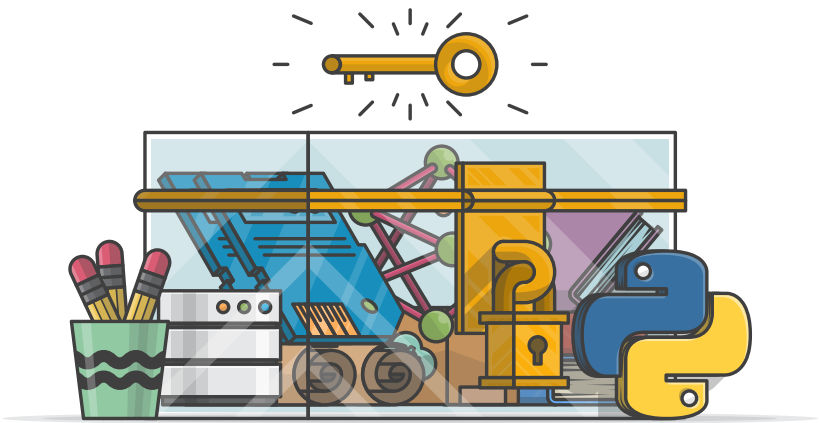
Send Python Tricks »

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Adriana](#)

[Dan](#)

Master Real-World Python Skills  
With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Improve Your Python

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

**Real Python Comment Policy:** The from or helping out other readers— Complaints and insults generally w

What’s your #1 takeaway or favorite th  
Leave a comment below and let us kn

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh **Python Trick**   
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

Send Python Tricks »

Keep Learning

Related Tutorial Categories: [basics](#) [da](#)

Recommended Video Course: [Python Plotting with Matplotlib](#)

— FREE Email Series —

Python Tricks

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

- [advanced](#)[api](#)[basics](#)[best-practices](#)[community](#)[databases](#)[data-science](#)[devops](#)[django](#)[docker](#)[flask](#)[front-end](#)[gamedev](#)[gui](#)[intermediate](#)[machine-learning](#)[projects](#)[python](#)[testing](#)[tools](#)[web-dev](#)[web-scraping](#)

Improve Your Python



# Master Python With a Level Up Our Value

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

## Improve Your Python

...with a fresh 🐍 **Python Trick** ❤️  
code snippet every couple of days:

Email Address

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

[Send Python Tricks »](#)

Table of Contents

- [Why Can Matplotlib Be So Slow?](#)
- [Pylab: What It Is and How to Use It](#)
- [The Matplotlib Architecture](#)
- [Stateful Versus Stateless Plotting](#)
- [Understanding the Figure and Axes Objects](#)
- [The “Figures” Behind The Scenes](#)
- [A Burst of Color: imshow\(\) and matshow\(\)](#)
- [Plotting in Pandas](#)
- [Wrapping Up](#)
- [More Resources](#)
- [Appendix A: Configuration and Styling](#)
- [Appendix B: Interactive Mode](#)

Mark as Completed

Tweet

Share

Email


Recommended Video Course

[Python Plotting With Matplotlib](#)



### High Quality Python Video Courses

**Watch Now »**



[Become a Python Expert »](#)

[Remove ads](#)