Matplotlib

matplotlib is the most popular Python library for producing plots and other two dimensional data visualizations. It was originally created by John D. Hunter and is now maintained by a large team of developers. It is designed for creating plots suit- able for publication. While there are other visualization libraries available to Python programmers, matplotlib is the most widely used and as such has generally good integration with the rest of the ecosystem. I think it is a safe choice as a default visualization tool.

The simplest way to follow the code examples in the chapter is to use interactive plot- ting in the Jupyter notebook. To set this up, execute the following statement in a Jupyter notebook: %matplotlib notebook

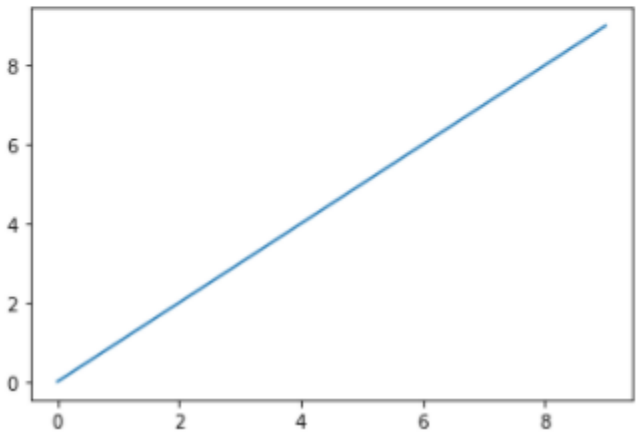
```
In [1]: import matplotlib.pyplot as plt

In [2]: import numpy as np
        data = np.arange(10)
        data

Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [3]: plt.plot(data)

Out[3]: [<matplotlib.lines.Line2D at 0x1e399adb970>]
```



Figures and Subplots

Plots in matplotlib reside within a Figure object. You can create a new figure with plt.figure:

```
In [4]: fig = plt.figure()
```

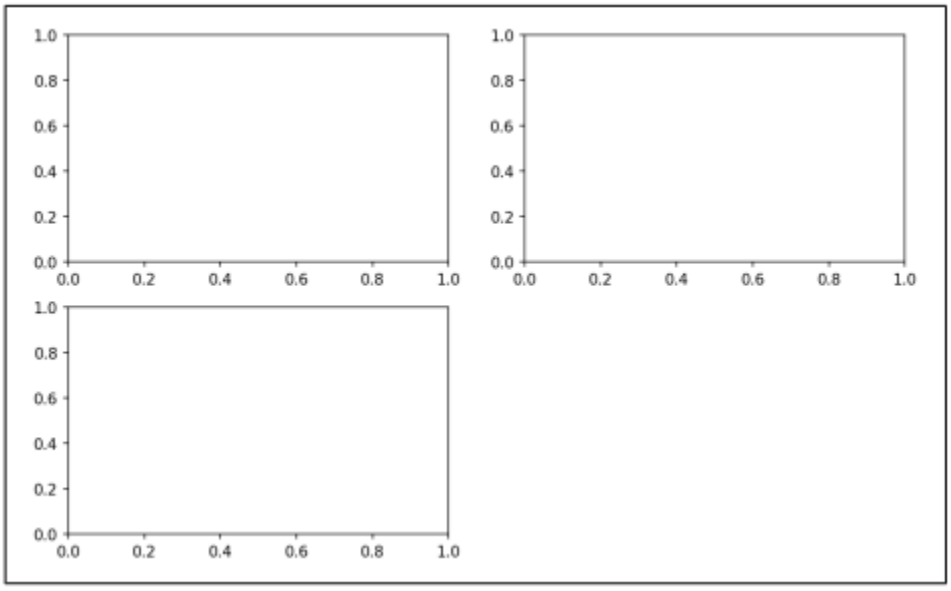
```
<Figure size 432x288 with 0 Axes>
```

In IPython, an empty plot window will appear, but in Jupyter nothing will be shown until we use a few more commands. plt.figure has a number of options; notably, figsize will guarantee the figure has a certain size and aspect ratio if saved to disk.
You can't make a plot with a blank figure. You have to create one or more subplots using add_subplot:

```
In [5]: ax1 = fig.add_subplot(2, 2, 1)
```

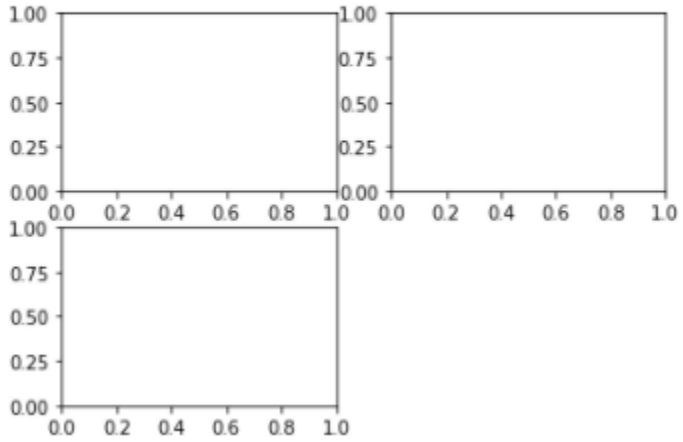
This means that the figure should be 2 × 2 (so up to four plots in total), and we're selecting the first of four subplots (numbered from 1). If you create the next two sub- plots, you'll end up with a visualization that looks like the figure below.

```
In [6]: ax2 = fig.add_subplot(2, 2, 2)
        ax3 = fig.add_subplot(2, 2, 3)
```



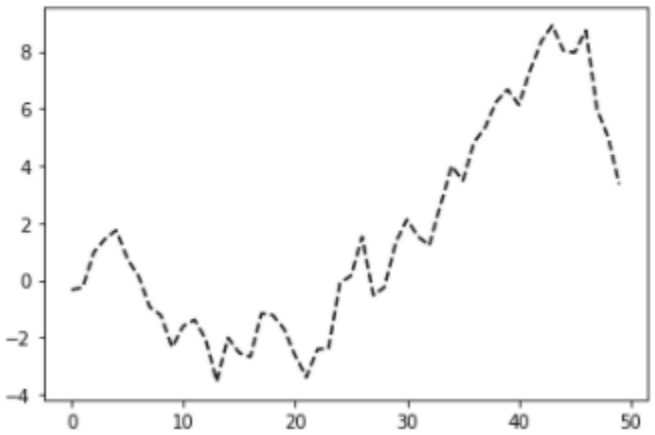
Here we run all of these commands in the same cell

```
In [8]: fig = plt.figure()
        ax1 = fig.add_subplot(2, 2, 1)
        ax2 = fig.add_subplot(2, 2, 2)
        ax3 = fig.add_subplot(2, 2, 3)
```



When you issue a plotting command like plt.plot([1.5, 3.5, -2, 1.6]), matplotlib draws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation. So if we add the following command, you'll get something like the figure below.

```
In [9]: plt.plot(np.random.randn(50).cumsum(), 'k--')
Out[9]: [<matplotlib.lines.Line2D at 0x1e3a4e09040>]
```

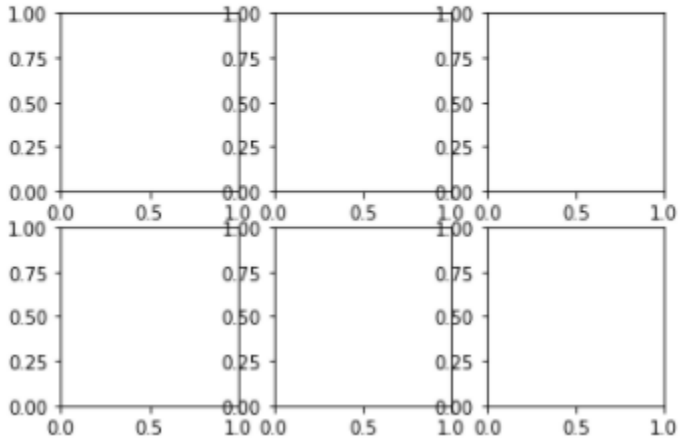


The 'k--' is a style option instructing matplotlib to plot a black dashed line. The objects returned by fig.add_subplot here are AxesSubplot objects, on which you can directly plot on the other empty subplots by calling each one's instance method:

Creating a figure with a grid of subplots is a very common task, so matplotlib includes a convenience method, plt.subplots, that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [10]: fig, axes = plt.subplots(2, 3)
         axes

Out[10]: array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
                [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```



This is very useful, as the axes array can be easily indexed like a two-dimensional array; for example, axes[0, 1]. You can also indicate that subplots should have the same x- or y-axis using sharex and sharey, respectively. This is especially useful when you're comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently

Argument	Description
nrows	Number of rows of subplots
ncols	Number of columns of subplots
sharex	All subplots should use the same x-axis ticks (adjusting the xlim will affect all subplots)
sharey	All subplots should use the same y-axis ticks (adjusting the ylim will affect all subplots)
subplot_kw	Dict of keywords passed to add_subplot call used to create each subplot
**fig_kw	Additional keywords to subplots are used when creating the figure, such as plt.subplots(2, 2, figsize=(8, 6))

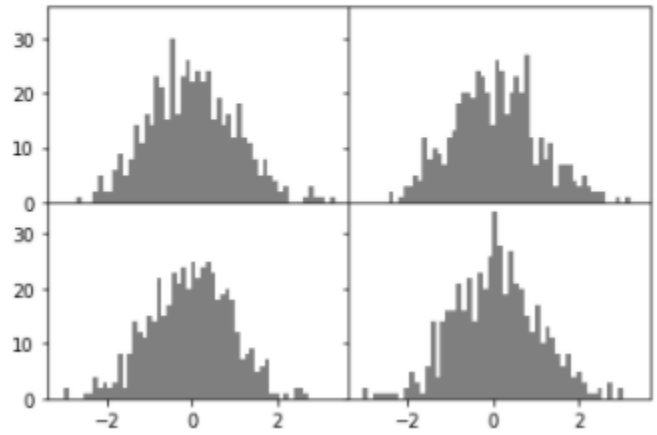
Adjusting the spacing around subplots

By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. You can change the spacing using the subplots_adjust method on Figure objects, also available as a top-level function:

subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None,hspace=None)

wspace and hspace controls the percent of the figure width and figure height, respectively, to use as spacing between subplots. Here is a small example where I shrink the spacing all the way to zero

```
In [11]: fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
         for i in range(2):
          for j in range(2):
             axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
         plt.subplots_adjust(wspace=0, hspace=0)
```



You may notice that the axis labels overlap. matplotlib doesn't check whether the labels overlap, so in a case like this you would need to fix the labels yourself by speci- fying explicit tick locations and tick labels.

Colors, Markers, and Line Styles

Matplotlib's main plot function accepts arrays of x and y coordinates and optionally a string abbreviation indicating color and line style. For example, to plot x versus y with green dashes, you would execute:

ax.plot(x, y, 'g--')

This way of specifying both color and line style in a string is provided as a conve- nience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

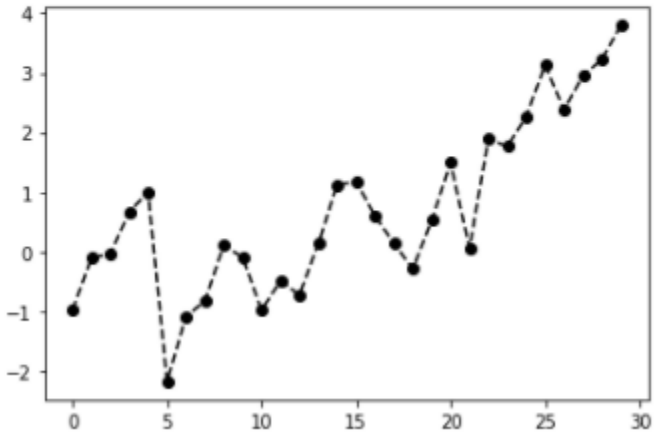
ax.plot(x, y, linestyle='--', color='g')

There are a number of color abbreviations provided for commonly used colors, but you can use any color on the spectrum by specifying its hex code (e.g., '#CECECE'). You can see the full set of line styles by looking at the docstring for plot (use plot? in IPython or Jupyter).

Line plots can additionally have markers to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style.

```
In [12]: from numpy.random import randn

In [13]: plt.plot(randn(30).cumsum(), 'ko--')
Out[13]: [<matplotlib.lines.Line2D at 0x2a55ce75eb0>]
```



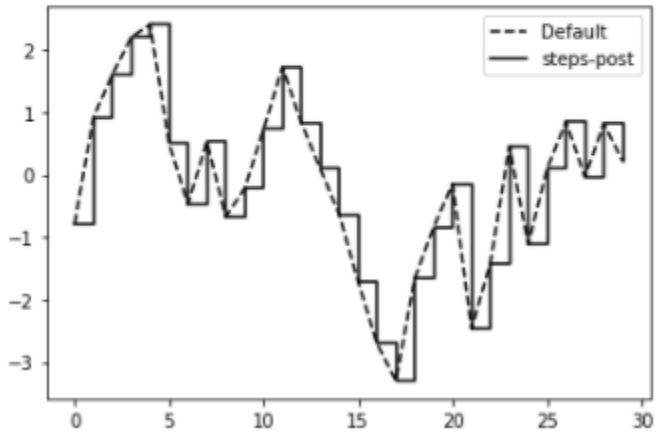
This could also have been written more explicitly as:

plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the drawstyle option

```
In [14]: data = np.random.randn(30).cumsum()
         plt.plot(data, 'k--', label='Default')
         plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
         plt.legend(loc='best')
```

```
Out[14]: <matplotlib.legend.Legend at 0x257b0481310>
```



You may notice output like when you run this. matplotlib returns objects that reference the plot subcomponent that was just added. A lot of the time you can safely ignore this output. Here, since we passed the label arguments to plot, we are able to create a plot legend to identify each line using plt.legend.

You must call plt.legend (or ax.legend, if you have a reference to the axes) to create the legend, whether or not you passed the label options when plotting the data.

Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural pyplot interface (i.e., matplotlib.pyplot) and the more object-oriented native matplotlib API. The pyplot interface, designed for interactive use, consists of methods like xlim, xticks, and xticklabels. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

- Called with no arguments returns the current parameter value (e.g., plt.xlim() returns the current x-axis plotting range)
- Called with parameters sets the parameter value (e.g., plt.xlim([0, 10]), sets the x-axis range to 0 to 10)

All such methods act on the active or most recently created AxesSubplot. Each of them corresponds to two methods on the subplot object itself; in the case of xlim these are ax.get_xlim and ax.set_xlim. I prefer to use the subplot instance methods myself in the interest

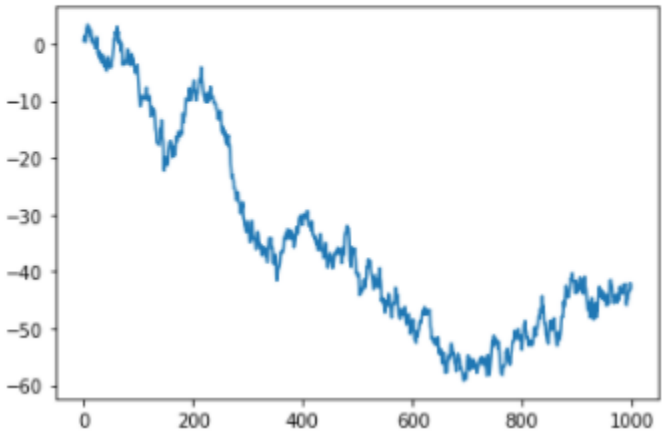
of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk

```
In [15]: fig = plt.figure()
         ax = fig.add_subplot(1, 1, 1)
         ax.plot(np.random.randn(1000).cumsum())
```

```
Out[15]: [<matplotlib.lines.Line2D at 0x257b07fd7c0>]
```



To change the x-axis ticks, it's easiest to use set_xticks and set_xticklabels. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using set_xticklabels:

```
In [17]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
         labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],rotation=30, fontsize='small')
```

```
In [19]: ax.set_title('My first matplotlib plot')
```

```
Out[19]: Text(0.5, 1.0, 'My first matplotlib plot')
```

```
In [20]: ax.set_xlabel('Stages')
```

```
Out[20]: Text(0.5, 3.1999999999999993, 'Stages')
```

Modifying the y-axis consists of the same process, substituting y for x in the above. The axes class has a set method that allows batch setting of plot properties. From the prior example, we could also have written:

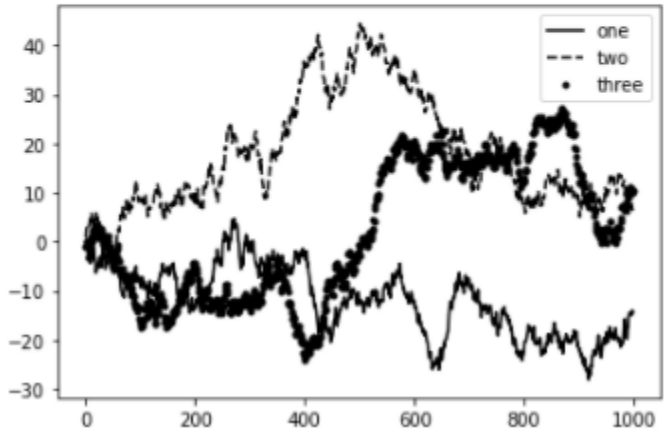
```
props = {
        'title': 'My first matplotlib plot',
        'xlabel': 'Stages'
        }
```

ax.set(**props)

Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the label argument when adding each piece of the plot:
Once you've done this, you can either call ax.legend() or plt.legend() to automat- ically create a legend. The resulting plot is

```
In [22]: from numpy.random import randn
         fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
         ax.plot(randn(1000).cumsum(), 'k', label='one')
         ax.plot(randn(1000).cumsum(), 'k--', label='two')
         ax.plot(randn(1000).cumsum(), 'k.', label='three')
         ax.legend(loc='best')
Out[22]: <matplotlib.legend.Legend at 0x257b08b1250>
```



The loc tells matplotlib where to place the plot. If you aren't picky, 'best' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or label='_nolegend_

Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annota- tions, which could consist of text, arrows, or other shapes. You can add annotations and text using the text, arrow, and annotate functions. text draws text at given coordinates (x, y) on the plot with optional custom styling:

ax.text(x, y, 'Hello world!', family='monospace', fontsize=10)

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True)
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor='black', headwidth=4, width=2,
                                headlength=4),
                horizontalalignment='left', verticalalignment='top')

# Zoom in on 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in the 2008-2009 financial crisis')
```

Output:

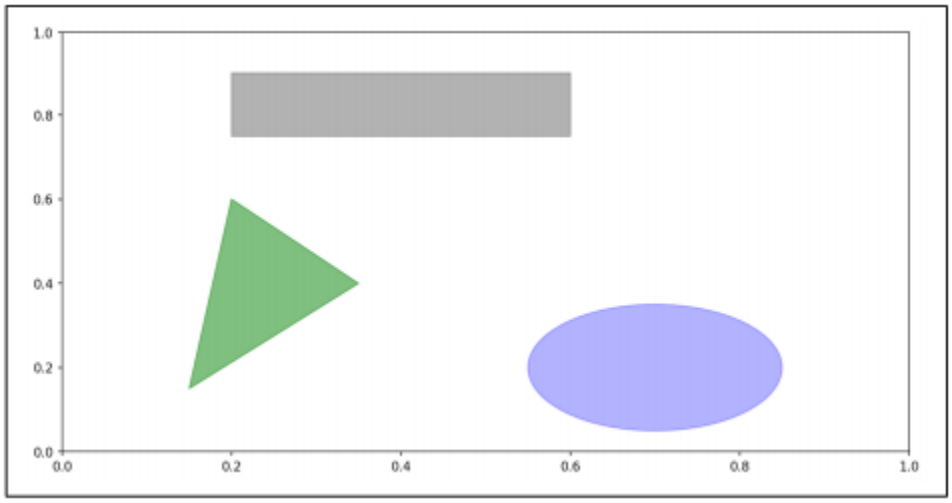
There are a couple of important points to highlight in this plot: the ax.annotate method can draw labels at the indicated x and y coordinates. We use the set_xlim and set_ylim methods to manually set the start and end boundaries for the plot rather than using matplotlib's default. Lastly, ax.set_title adds a main title to the plot.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as patches. Some of these, like Rectangle and Circle, are found in matplotlib.pyplot, but the full set is located in matplotlib.patches. To add a shape to a plot, you create the patch object shp and add it to a subplot by calling ax.add_patch(shp)

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```



If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

Saving Plots to File

You can save the active figure to file using plt.savefig. This method is equivalent to the figure object's savefig instance method. For example, to save an SVG version of a figure, you need only type:

plt.savefig('figpath.svg')

The file type is inferred from the file extension. So if you used .pdf instead, you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: dpi, which controls the dots-per-inch resolution, and bbox_inches, which can trim the whitespace around

the actual figure. To get the same plot as a PNG with minimal whitespace around the plot and at 400 DPI, you would do:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

savefig doesn’t have to write to disk; it can also write to any file-like object, such as a BytesIO:

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. One way to modify the configuration programmatically from Python is to use the rc method; for example, to set the global default figure size to be 10 × 10, you could enter:

```
plt.rc('figure', figsize=(10, 10))
```

The first argument to rc is the component you wish to customize, such as 'figure', 'axes', 'xtick', 'ytick', 'grid', 'legend', or many others. After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

```
font_options = {'family' : 'monospace', 'weight' : 'bold', 'size' : 'small'}
plt.rc('font', **font_options)
```

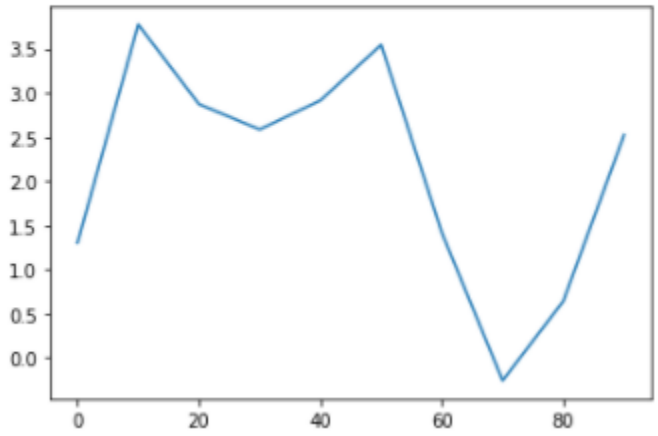
For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file matplotlibrc in the matplotlib/mpl-data directory. If you customize this file and place it in your home directory titled .matplotlibrc, it will be loaded each time you use matplotlib.

Line Plots

Series and DataFrame each have a plot attribute for making some basic plot types. By default, plot() makes line plots

```
In [24]: import pandas as pd
         s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
         s.plot()
```

```
Out[24]: <AxesSubplot:>
```



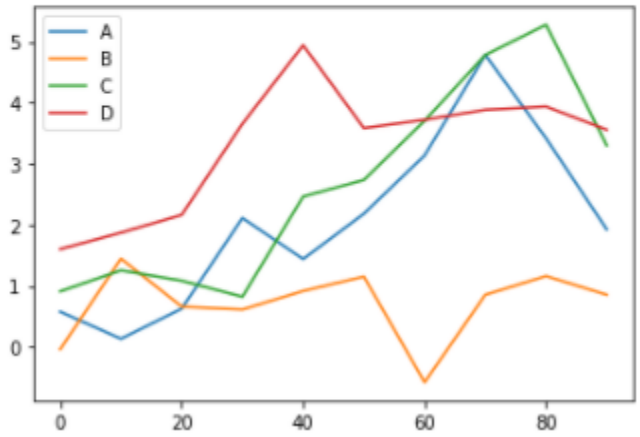
The Series object's index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing use_index=False. The x-axis ticks and limits can be adjusted with the xticks and xlim options, and y-axis respectively with yticks and ylim.

Most of pandas's plotting methods accept an optional ax parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout.

DataFrame's plot method plots each of its columns as a different line on the same subplot, creating a legend automatically

```
In [25]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
                           columns=['A', 'B', 'C', 'D'],
                           index=np.arange(0, 100, 10))
         df.plot()
```

```
Out[25]: <AxesSubplot:>
```



The plot attribute contains a “family” of methods for different plot types. For exam- ple, df.plot() is equivalent to df.plot.line().

Argument	Description
label	Label for plot legend
ax	matplotlib subplot object to plot on; if nothing passed, uses active matplotlib subplot
style	Style string, like 'ko--', to be passed to matplotlib
alpha	The plot fill opacity (from 0 to 1)
kind	Can be 'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie'
logy	Use logarithmic scaling on the y-axis
use_index	Use the object index for tick labels
rot	Rotation of tick labels (0 through 360)
xticks	Values to use for x-axis ticks
yticks	Values to use for y-axis ticks

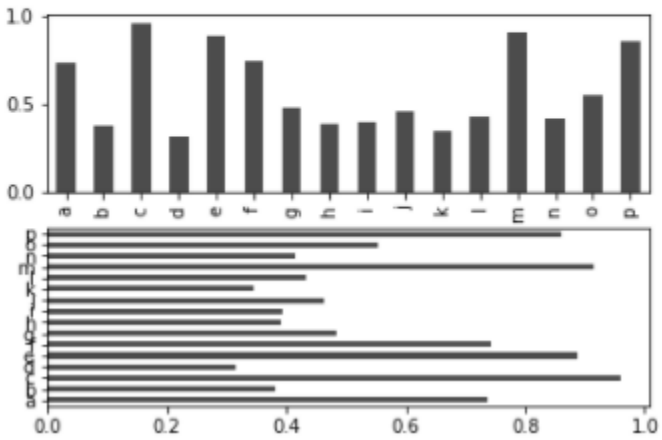
xlim	x-axis limits (e.g., [0, 10])
ylim	y-axis limits
grid	Display axis grid (on by default)

Bar Plots

The plot.bar() and plot.barh() make vertical and horizontal bar plots, respectively. In this case, the Series or DataFrame index will be used as the x (bar) or y (barh) ticks.

```
In [22]: fig, axes = plt.subplots(2, 1)
         data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
         data.plot.bar(ax=axes[0], color='k', alpha=0.7)
         data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

```
Out[22]: <AxesSubplot:>
```



The options color='k' and alpha=0.7 set the color of the plots to black and use par- tial transparency on the filling.
With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value.

```
In [23]: df = pd.DataFrame(np.random.rand(6, 4),index=['one', 'two', 'three', 'four', 'five', 'six'],
                           columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
```

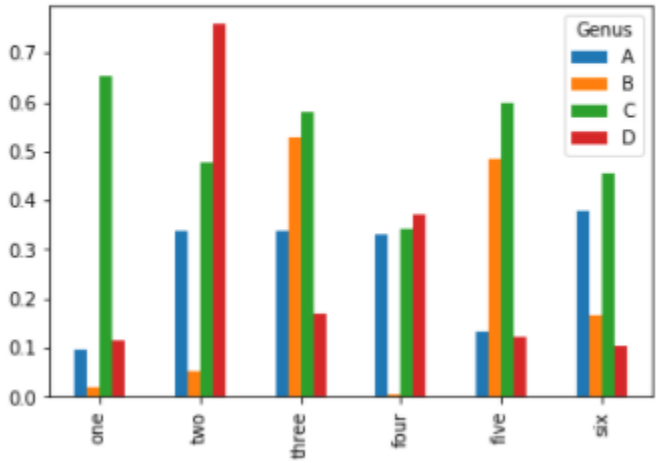
```
In [24]: df
```

Out[24]:

Genus	A	B	C	D
one	0.098331	0.020620	0.652337	0.115165
two	0.337624	0.053342	0.477606	0.758093
three	0.338230	0.528235	0.579451	0.168904
four	0.330170	0.004753	0.342217	0.370222
five	0.134297	0.484148	0.598823	0.123722
six	0.378607	0.165117	0.457301	0.104813

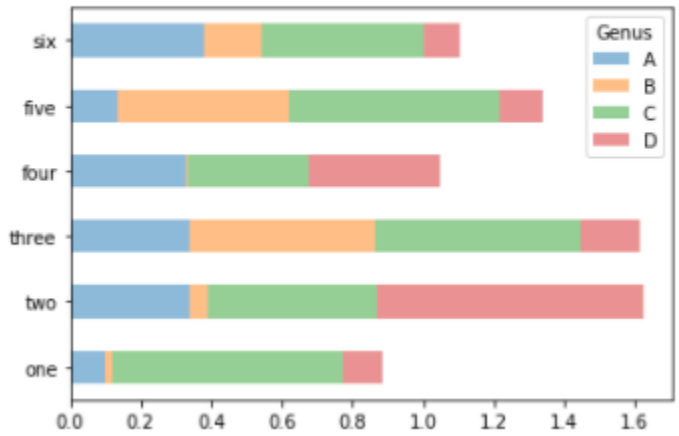
```
In [26]: df.plot.bar()
```

Out[26]: <AxesSubplot:>



Note that the name “Genus” on the DataFrame’s columns is used to title the legend.
We create stacked bar plots from a DataFrame by passing stacked=True, resulting in the value in each row being stacked together.

```
In [27]: df.plot.barh(stacked=True, alpha=0.5)
Out[27]: <AxesSubplot:>
```



In this article we have covered various elements of matplotlib including the figures, subplots,configuring the plots(adjusting space,colors,markers,line style,ticks,labels,legends),annotations and drawing on subplots,saving plots to file,line plot,bar plots etc