If you have ever created graphs in MATLAB, you will find that matplotlib in many ways directly emulates this experience. The similarities between MATLAB and matplotlib are intentional. The MATLAB plotting interface was a direct inspiration for matplotlib. Even if you haven't used MATLAB, you will likely find creating plots with matplotlib to be simple and straightforward.

Let's dive in!

Install matplotlib

You can install matplotlib from your terminal with pip3:

```
pip3 install matplotlib
```

You can then view some details about the package with pip3 show:

```
$ pip3 show matplotlib
Name: matplotlib
Version: 2.2.3
Summary: Python plotting package
Home-page: http://matplotlib.org
Author: John D. Hunter, Michael Droettboom
Author-email: matplotlib-users@python.org
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires: python-dateutil, pytz, kiwisolver, numpy,
          cycler, six, pyparsing
Required-by:
```

In particular, note that the latest version at the time of writing is version 2.2.3.

Basic Plotting With pyplot

The matplotlib package provides two distinct means of creating plots. The first, and simplest, method is through the pyplot interface. This is the interface that MATLAB users will find the most familiar.

The second method for plotting in matplotlib is through what is known as the object oriented API. The object-oriented approach offers more control over your plots than is available through the pyplot interface. However, the concepts are generally more abstract.

In this section, you'll learn how to get up and running with the pyplot interface. You'll be pumping out some great looking plots in no time!

Note

The developers of matplotlib suggest you try to use the object-oriented API instead of the pyplot interface. In practice, if the pyplot interface offers you everything you need, then don't be ashamed to stick with it!

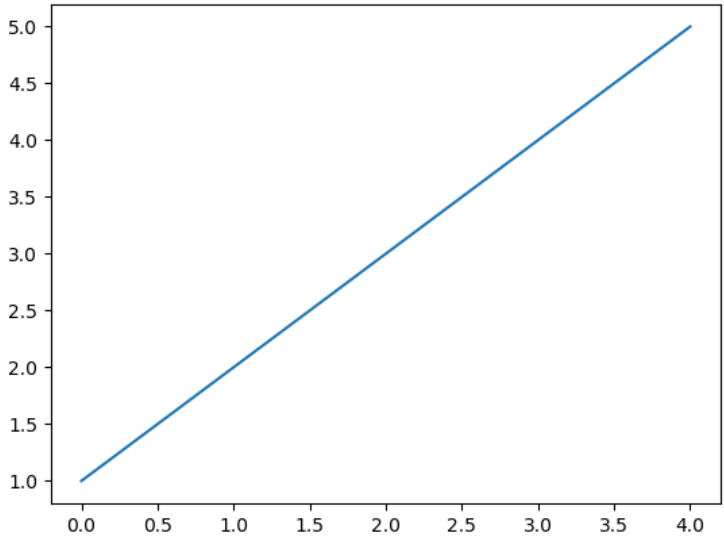
That said, if you are interested in learning more about the object-oriented approach, check out Real Python's Python Plotting With Matplotlib (Guide).

Let's start by creating a simple plot. Open IDLE and run the following script:

```
from matplotlib import pyplot as plt


plt.plot([1, 2, 3, 4, 5])
plt.show()
```

A new window appears displaying the following plot:



In this simple script, you created a plot with just a single line of code. The line `plt.plot([1, 2, 3, 4, 5])` creates a plot with a line through the points (0, 1), (1, 2), (2, 3), (3, 4), and (4, 5). The list `[1, 2, 3, 4, 5]` that you passed to the `plt.plot()` function represents the y-values of the points in the plot. Since you didn't specify any x-values, `matplotlib` automatically uses the indices of the list elements which, since Python starts counting at 0, are 0, 1, 2, 3 and 4.

The `plt.plot()` function creates a plot, but it does not display anything. The `plot.show()` function must be called to display the plot.

Note

If you are working in Windows, you should have no problem recreating the above plot from IDLE's interactive window. However, some operating systems have trouble displaying plots with `plot.show()` when called from the interactive window. We recommend working through each example in a new script.

If `plt.show()` works from the interactive window on your machine and you decide to follow along that way, be aware that once the figure is displayed in the new window, control isn't returned to the interactive window until you close the figure's window. That is, you won't see a new >>> prompt until the figure's window has been closed.

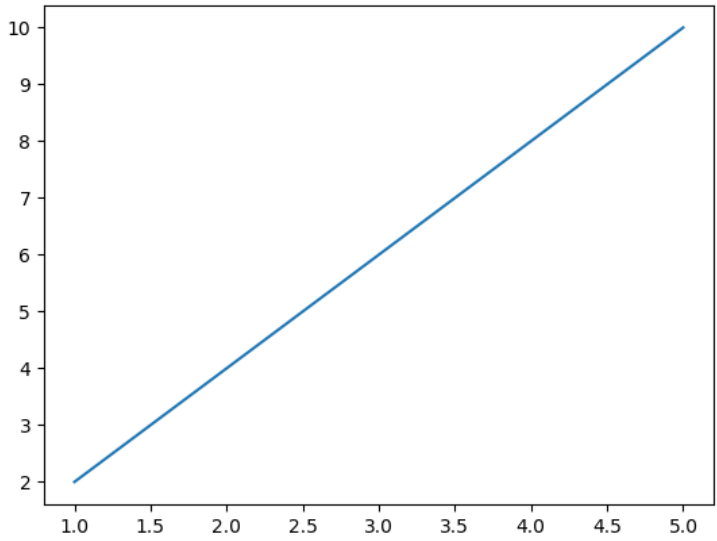
You can specify the x-values for the points in your plot by passing two lists to the `plt.plot()` function. When two arguments are provided to `plt.plot()`, the first list specifies the x-values and the second list specifies the y-values:

```
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
ys = [2, 4, 6, 8, 10]

plt.plot(xs, ys)
plt.show()
```

Running the above script produces the following plot:



At first glance, this figure may look exactly like the first. However, the labels on the axes now reflect the new x- and y-coordinates of the points.

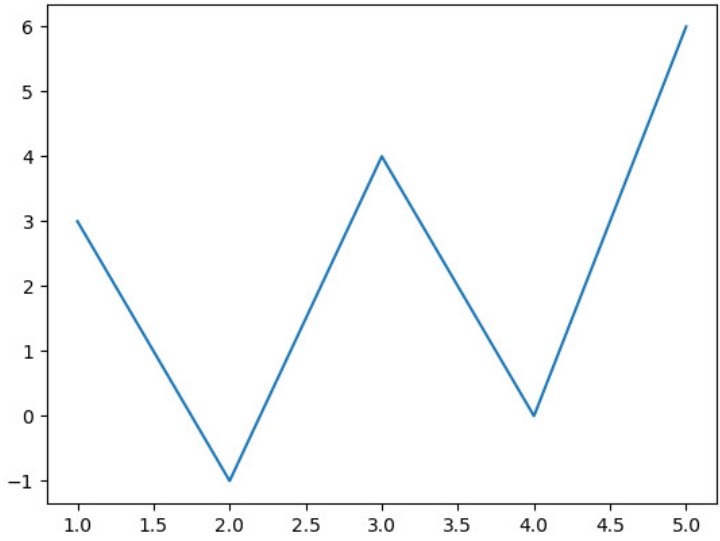
You can use `plot()` to plot more than lines. In the graphs above, the points being plotted just happen to all fall on the same line. By default, when plotting points with `.plot()`, each pair of consecutive points being plotted is connected with a line segment.

The following plot displays some data that doesn't fall on a line:

```
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
ys = [3, -1, 4, 0, 6]

plt.plot(xs, ys)
plt.show()
```

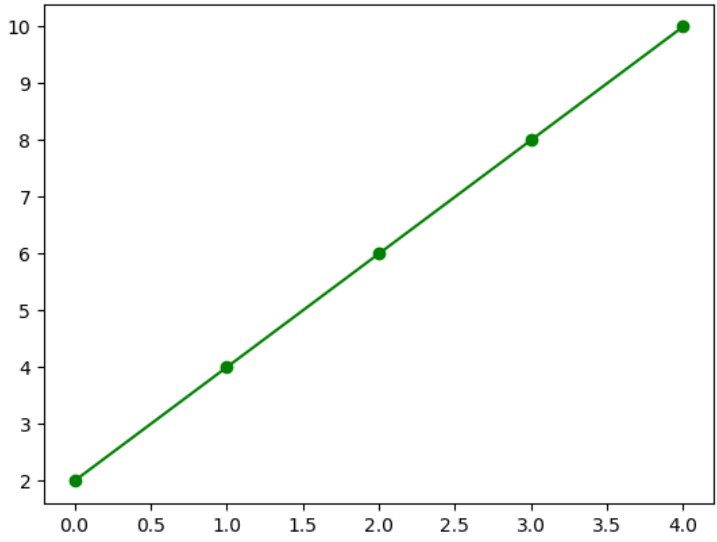


There is an optional "formatting" argument that can be inserted into `plot()` after specifying the points to be plotted. This argument specifies the color and style of lines or points to draw.

Unfortunately, the standard is borrowed from MATLAB and (compared to most Python) the formatting is not very intuitive to read or remember. The default value is "solid blue line," which would be represented by the format string `b-`. If we wanted to plot green circular dots connected by solid lines instead, we would use the format string `g-o` like so:

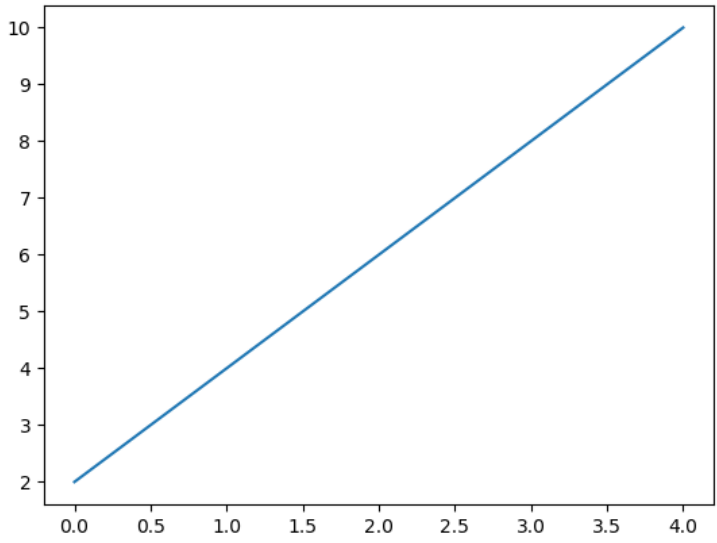
```
from matplotlib import pyplot as plt


plt.plot([2, 4, 6, 8, 10], "g-o")
plt.show()
```



Note

For reference, the full list of possible formatting combinations can be found here.



Plot Multiple Graphs in the Same Window

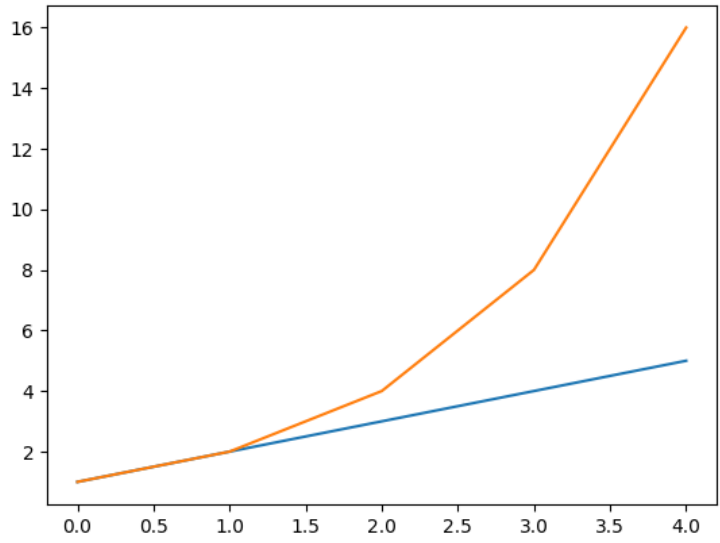
If you need to plot multiple graphs in the same window, you can do so a few different ways.

You can pass multiple pairs of x- and y-value lists:

```
from matplotlib import pyplot as plt

xs = [0, 1, 2, 3, 4]
y1 = [1, 2, 3, 4, 5]
y2 = [1, 2, 4, 8, 16]

plt.plot(xs, y1, xs, y2)
plt.show()
```



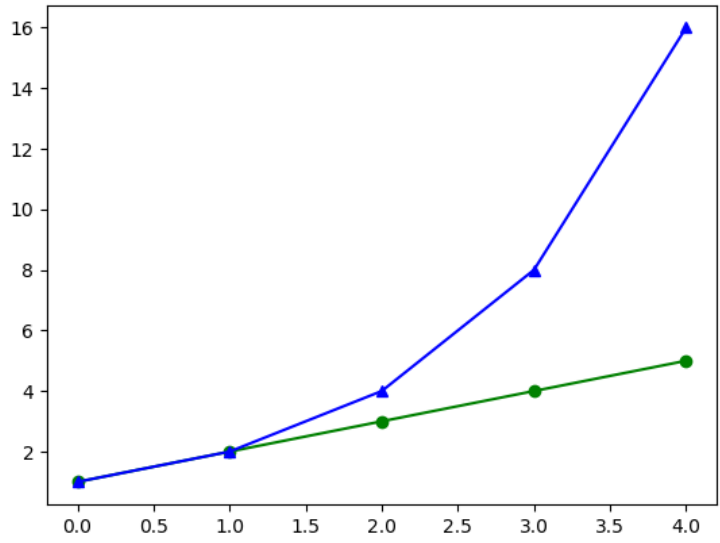
Notice that each graph is displayed in a different color. This built-in functionality of the `plot()` function is convenient for making easy-to-read plots very quickly.

If you want to control the style of each graph, you can pass the formatting strings to the `plot()` in addition to the x- and y-values:

```
from matplotlib import pyplot as plt

xs = [0, 1, 2, 3, 4]
y1 = [1, 2, 3, 4, 5]
y2 = [1, 2, 4, 8, 16]

plt.plot(xs, y1, "g-o", xs, y2, "b-^")
plt.show()
```



Passing multiple sets of points to `plot()` may work well when you only have a couple of graphs to display, but if you need to show many, it might make more sense to display each one with its own `plot()` function.

For example, the following script displays the same plot as the previous example:

```
from matplotlib import pyplot as plt


plt.plot([1, 2, 3, 4, 5], "g-o")
plt.plot([1, 2, 4, 8, 16], "b-^")
plt.show()
```

Plot Data From NumPy Arrays

Up to this point, you have been storing your data points in pure Python lists. In the real world, you will most likely be using some-

thing like a NumPy array to store your data. Fortunately, matplotlib plays nicely with array objects.

Note

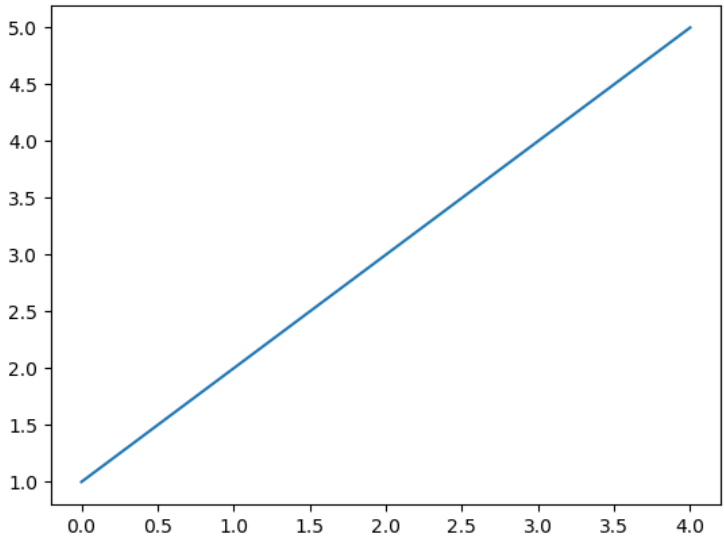
If you do not currently have NumPy installed, you need to install it with pip. For more information, please refer to the previous section in this chapter.

For example, instead of a list, you can use NumPy's arange() function to define your data points and then pass the resulting array object to the plot() function:

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

array = np.arange(1, 6)

plt.plot(array)
plt.show()
```



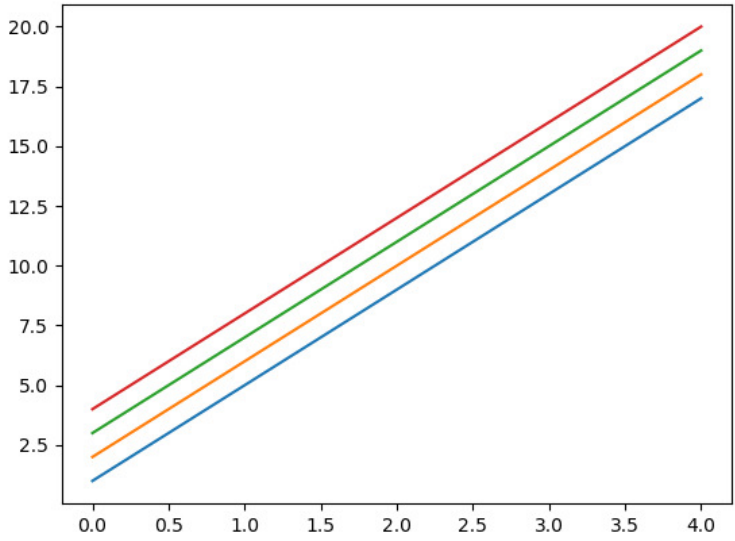
Passing a two-dimensional `array` plots each *column* of the array as the y-values for a graph. For example, the following script plots four lines:

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

data = np.arange(1, 21).reshape(5, 4)

# data now contains the following array:
# array([[ 1,  2,  3,  4],
#        [ 5,  6,  7,  8],
#        [ 9, 10, 11, 12],
#        [13, 14, 15, 16],
#        [17, 18, 19, 20]])

plt.plot(data)
plt.show()
```

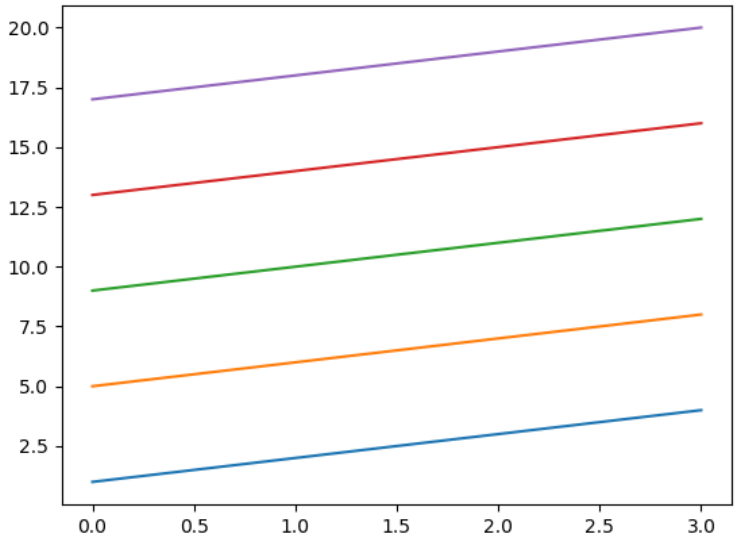


If instead you want to plot the rows of the matrix, you need to plot the transpose of the array. The following script plots the five rows of the same array from the previous example:

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

data = np.arange(1, 21).reshape(5, 4)

plt.plot(data.transpose())
plt.show()
```



Format Your Plots to Perfection

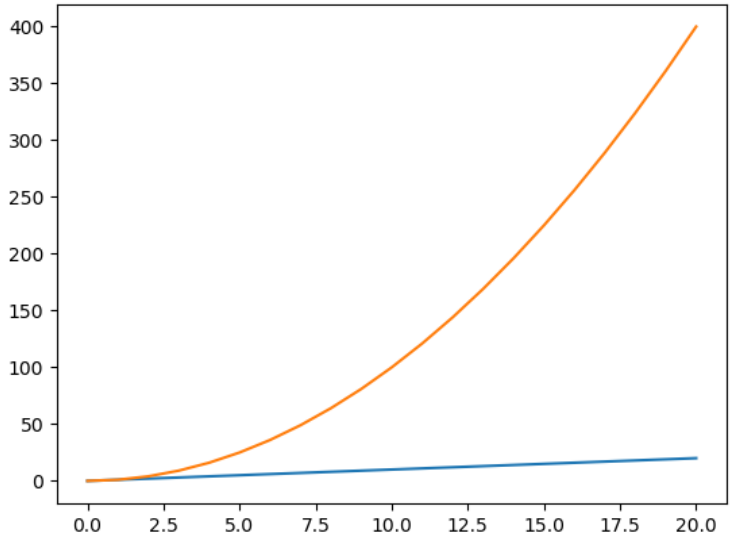
So far, the plots you have seen don't provide any information about what the plot represents. In this section, you will learn how to change the format and layout of your plots to make them easier to understand.

Let's start by plotting the amount of Python learned in the first 20 days of reading Real Python versus another website:

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

days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.show()
```



As you can see, the gains from reading Real Python are exponential! However, if you showed this graph to someone else, they may not understand what's going on.

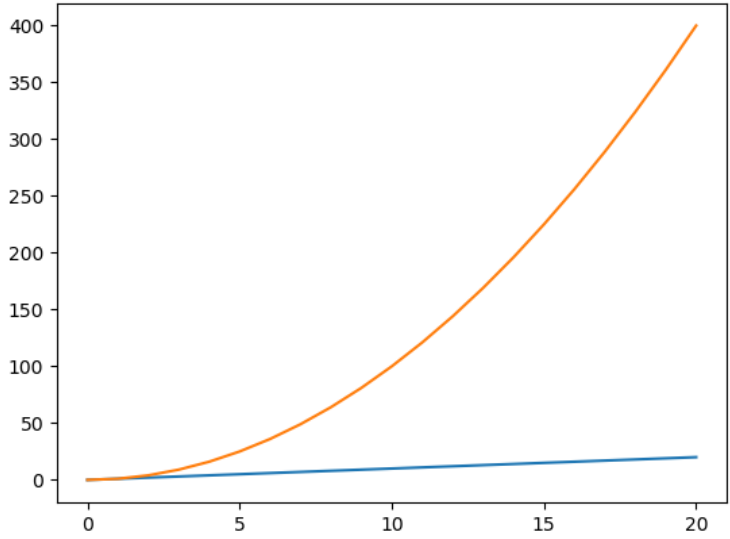
First of all, the x-axis is a little weird. It is supposed to represent days but is displaying half days instead. It would also be helpful to know what each line and axis represents. A title describing the plot wouldn't hurt, either.

Let's start with adjusting the x-axis. You can use the `plt.xticks()` function to specify where the ticks should be located by passing a `list` of locations. If we pass the list `[0, 5, 10, 15, 20]`, the ticks should mark days 0, 5, 10, 15 and 20:

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

```
days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.show()
```



Nice! That's a little easier to read, but it still isn't clear what each axis represents.

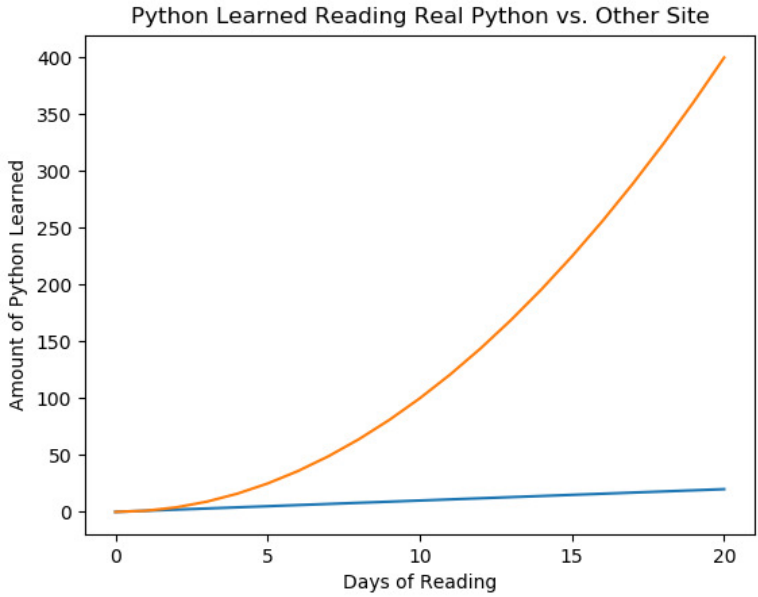
You can use the `plt.xlabel()` and `plt.ylabel()` to label the x- and y-axes, respectively. Just provide a string as an argument, and `matplotlib` displays the label on the corresponding axis.

While we're labeling things, let's go ahead and give the plot a title with the `plt.title()` function:

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

days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.xlabel("Days of Reading")
plt.ylabel("Amount of Python Learned")
plt.title("Python Learned Reading Real Python vs Other Site")
plt.show()
```



Now we're starting to get somewhere!

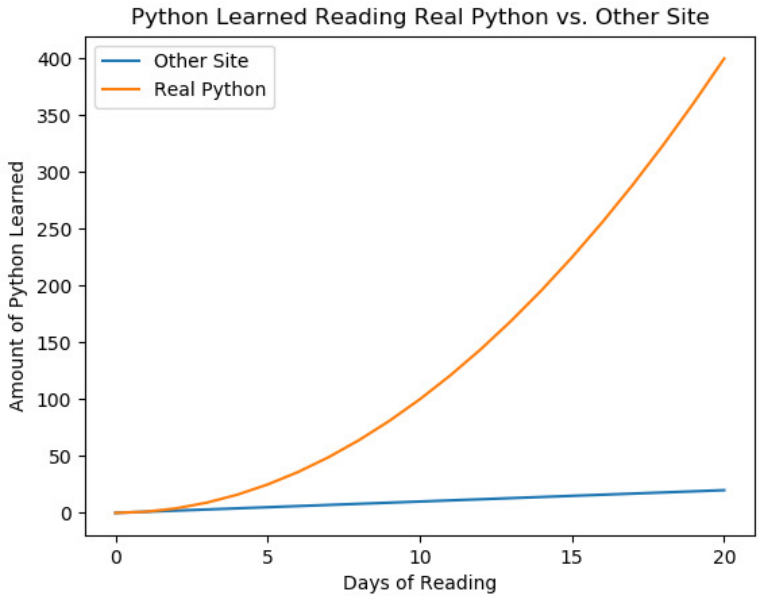
There's only one problem. It's not clear which graph represents Real Python and which one represents the other website.

To clarify which graph is which, you can add a legend with the `plt.legend()` function. The primary argument of the `legend()` function is a list of strings identifying each graph in the plot. These strings must be ordered in the same order the graphs were added to the plot:

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

days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.xlabel("Days of Reading")
plt.ylabel("Amount of Python Learned")
plt.title("Python Learned Reading Real Python vs Other Site")
plt.legend(["Other Site", "Real Python"])
plt.show()
```



Note

There are many ways to customize legends. For more information, check out the Legend Guide in the `matplotlib` documentation.

Other Types of Plots

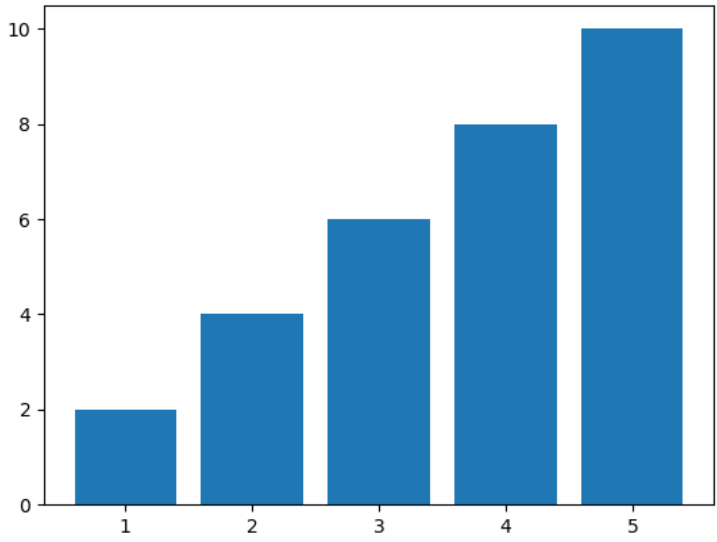
Aside from line charts, which up until now you have seen exclusively, `matplotlib` provides simple methods for creating other kinds of charts.

One frequently used type of plot in basic data visualization is the bar chart. You can easily create bar charts using the `plt.bar()` function. You must provide at least two arguments to `bar()`. The first is a list of x-values for the center point for each bar, and the second is the value for the top of each bar:

```
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
tops = [2, 4, 6, 8, 10]

plt.bar(xs, tops)
plt.show()
```



Just like the `plot()` function, you can use a NumPy `array` instead of a list. The following script produces a plot identical to the previous one:

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

xs = np.arange(1, 6)
tops = np.arange(2, 12, 2)
```

```
plt.bar(xs, tops)
plt.show()
```

The `bar()` function is more flexible than it lets on. For example, the first argument doesn't need to be a list of numbers. It could be a list of strings representing categories of data.

Suppose you wanted to plot a bar chart representing the data contained in the following dictionary:

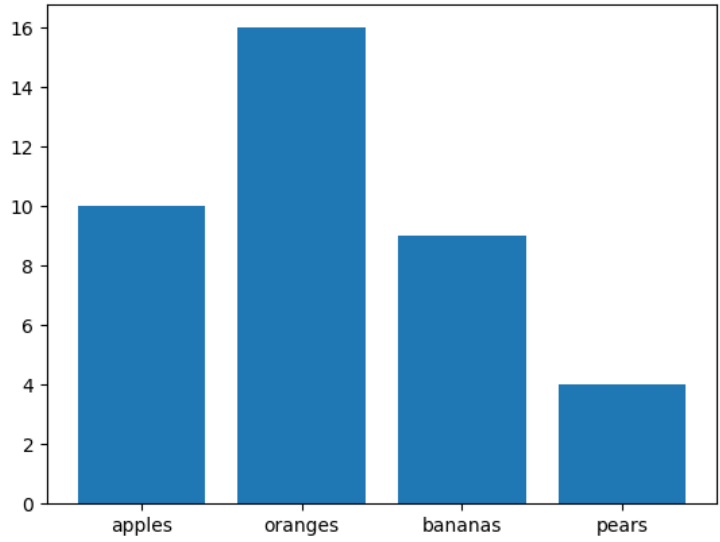
```
fruits = {
    "apples": 10,
    "oranges": 16,
    "bananas": 9,
    "pears": 4,
}
```

You can get a list of the names of the fruits using `fruits.keys()`, and the corresponding values using `fruits.values()`. Check out what happens when you pass these to the `bar()` function

```
from matplotlib import pyplot as plt

fruits = {
    "apples": 10,
    "oranges": 16,
    "bananas": 9,
    "pears": 4,
}

plt.bar(fruits.keys(), fruits.values())
plt.show()
```



The names of the fruits are conveniently used as the tick labels along the x-axis.

Note

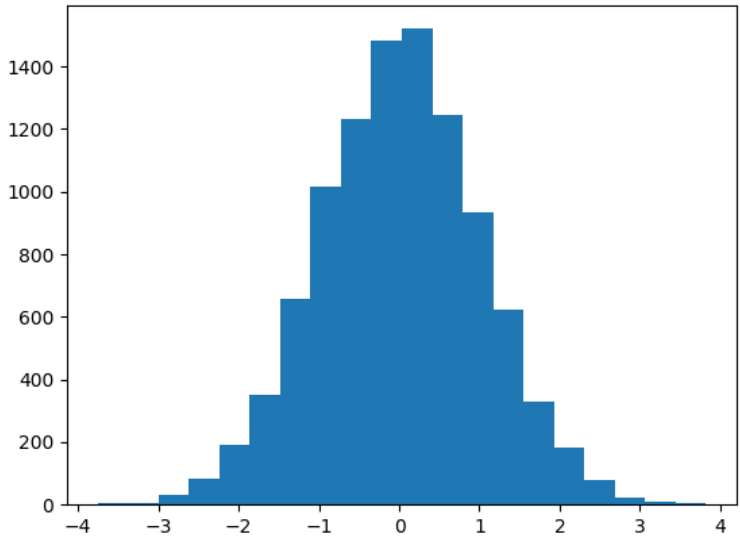
Using a list of strings as x-values works for the `plot()` function as well, although it often makes less sense to do so.

Another commonly used type of graph is the histogram, which shows how data is distributed. You can make simple histograms easily with the `plt.hist()` function. You must supply `hist()` with a list (or array) of values and a number of bins to use.

For instance, we can create a histogram of 10,000 normally distributed random numbers binned across 20 possible bars with the following, which uses NumPy's `random.randn()` function to generate an array of normally distributed random numbers:

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

plt.hist(random.randn(10000), 20)
plt.show()
```



Note

For a detailed discussion of creating histograms with Python, check out Python Histogram Plotting: NumPy, Matplotlib, Pandas & Seaborn on Real Python.

Save Figures as Images

You may have noticed that the window displaying your plots has a toolbar at the bottom. You can use this toolbar to save your plot as an image file.

More often than not, you probably don't want to have to sit at your computer and click on the save button for each plot you want to export. Fortunately, `matplotlib` makes it easy to save your plots programmatically.

To save your plot, use the `plt.savefig()` function. Pass the path to where you would like to save your plot as a string. The example below saves a simple bar chart as `bar.png` to the current working directory. If you would like to save to somewhere else, you must provide an absolute path.

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

xs = np.arange(1, 6)
tops = np.arange(2, 12, 2)

plt.bar(xs, tops)
plt.savefig("bar.png")
```

Note

If you want to both save a figure and display it on the screen, make sure that you save it first before displaying it!

The `show()` function pauses execution of your code and closing the display window destroys the graph, so trying to save the figure after calling `show()` results in an empty file.

Work With Plots Interactively

When you are initially tweaking the layout and formatting of a particular graph, it can be helpful to change parts of the graph without having to re-run an entire script just to see the results.

One of the easiest ways to do this is with a Jupyter Notebook, which creates an interactive Python interpreter session that runs in your

browser.

Jupyter notebooks have become a staple for interacting with and exploring data, and work great with both NumPy and `matplotlib`.

For an interactive tutorial on how to use Jupyter Notebooks, check out Jupyter's IPython In Depth tutorial.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Recreate as many of the graphs shown in this section as you can by writing your own scripts without referring to the provided code.
2. It is a well-documented fact that the number of pirates in the world is correlated with a rise in global temperatures. Write a script `pirates.py` that visually examines this relationship:
   - Read in the file `pirates.csv` from the Chapter 17 practice files folder.
   - Create a line graph of the average world temperature in degrees Celsius as a function of the number of pirates in the world—that is, graph Pirates along the x-axis and Temperature along the y-axis.
   - Add a graph title and label your graph's axes.
   - Save the resulting graph out as a PNG image file.
   - Bonus: Label each point on the graph with the appropriate Year. You should do this programmatically by looping through the actual data points rather than specifying the individual position of each annotation.

Leave feedback on this section »