5

Data Visualization

*Present 1-2: The title slide and the Learning Objectives slide. An overview of what we will achieve in this course*

Lesson Objectives:

By the end of this lesson, you will be able to:

* Create and customize line plots, bar plots, histograms, scatterplots, and box-and-whisker plots using the functional method
* Create a programmatic, descriptive title
* Describe the advantages of using the object-oriented method rather than the functional method in matplotlib
* Create a callable figure object containing a single axis or multiple axes
* Resize and save a figure object with numerous subplots

Introduction

*Present 3: Brief introduction to plotting*

Data visualizations are powerful tools allowing the user to digest large amounts of data very quickly. Plots come in a variety of shapes, sizes, and styles. In business, line plots and bar graphs are very common to display trends over time and compare metrics across groups, respectively. Statisticians, on the other hand, may be more interested in checking correlations between variables using a scatterplot or correlation heat map. They may also use histograms to check the distribution of a variable or boxplots to check for outliers. In politics, pie charts are widely used for comparing the total data between or among categories. Data visualizations can be very intricate and creative, being limited only by one’s imagination.

*Present 4: Introduction to Matplotlib*

The Python library Matplotlib (https://matplotlib.org/index.html) is a well-documented, 2-dimensional plotting library which can be used to create a variety of powerful data visualizations and aims to “...make easy things easy and hard things possible” ([https://matplotlib.org/index.html](https://matplotlib.org/3.0.3/index.html)). On the Matplotlib home page (<https://matplotlib.org/index.html>) are instructions for installation (https://matplotlib.org/users/installing.html) as well as the documentation. Additionally, Matplotlib provides a very comprehensive gallery of examples with the corresponding code.

For example, if we are wanting to create a bar chart and we want to see some of the examples that have already been completed we would begin by clicking on the *examples* link from the Matplotlib home page.



Figure 5.1. Matplotlib home page (https://matplotlib.org/index.html)

In the gallery, we can see hundreds of examples, but if we want to see an example of a horizontal bar chart, we can select the *Horizontal bar chart* example.

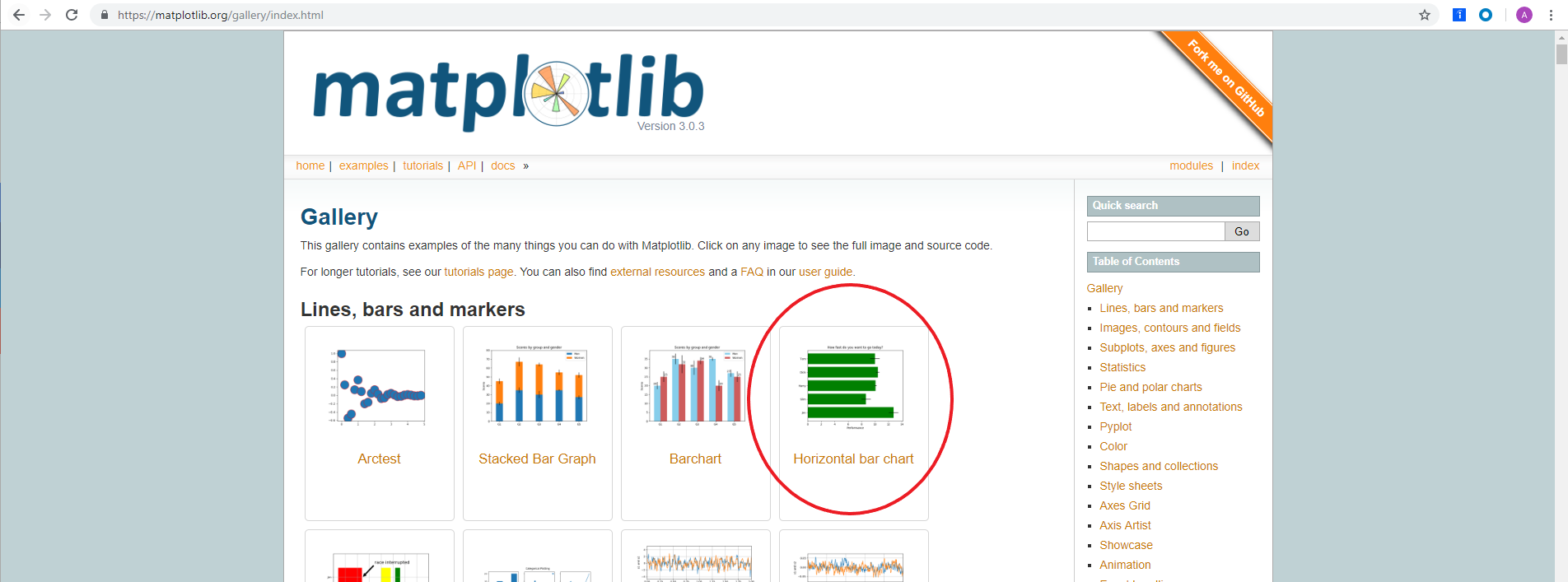


Figure 5.2. Matplotlib gallery (<https://matplotlib.org/gallery/index.html>)

This brings to a page displaying a picture of the chart as well as the code used to construct the plot which can be downloaded as a Python file or a Jupyter Notebook.

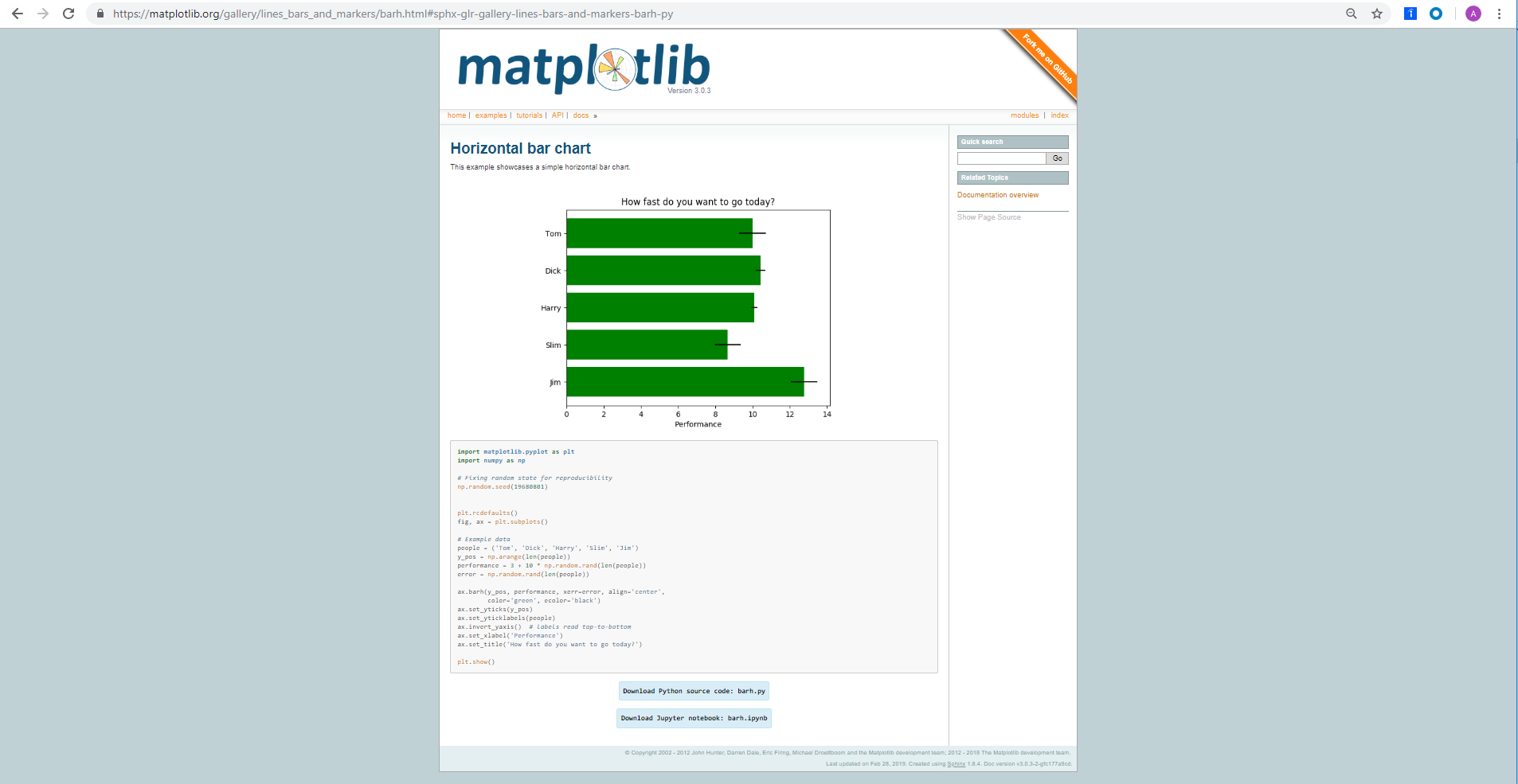


Figure 5.3. Matplotlib horizontal bar chart example (https://matplotlib.org/gallery/lines\_bars\_and\_markers/barh.html#sphx-glr-gallery-lines-bars-and-markers-barh-py)

*Present 5: Introduction to Matplotlib methods*

There are two methods in which plots are created using Matplotlib, the functional method and the object-oriented method.

In the functional method, 1 figure is created with 1 plot. The plots are created and customized by a collection of line-by-line functions. However, the functional method does not allow us to save the plot to our environment as an object; this is possible using the object-oriented method. In the object-oriented method we create a figure object and assign an axis or numerous axes for one plot or multiple subplots, respectively. We can then customize the axis or axes and call that single plot or set of multiple plots by calling the figure object.

In this Chapter, we will use the functional method to create and customize line plots, bar plots, histograms, scatterplots, and box-and-whisker plots. This will include 2 activities. In the first activity, we will create and customize a line plot to analyze the trend of Items Sold by Month. In the next activity we will create and customize a bar plot to display NBA titles by franchise. We will be introduced to programmatic titles that communicate to the reader what the plot is displaying and are able to adapt if the data changes. Using the object-oriented method, we will complete 2 exercises and 1 activity. The first exercise will include creating and customizing a single line plot on a callable figure object. The second exercise will demonstrate creating, referencing, and customizing multiple axes on a callable figure object. The third activity will include creating, referencing, and customizing multiple axes and plot types on a callable figure object. By the end of the chapter we will be experts in creating and customizing common plot types using matplotlib. We will also gain comfortability in navigating the matplotlib documentation so we can figure out how to make plots that are not covered in this chapter.

Functional Method

*Present 6: Functional Method*

Exercise 1: Line Plot

To learn matplotlib, we will begin with learning how to make a line plot and progress from there.

1. Import matplotlib using import matplotlib.pyplot as plt.



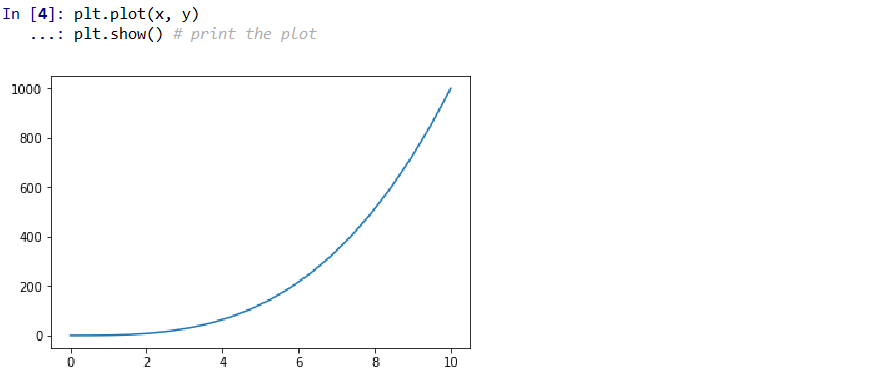
1. Create an array and save it as the object x. By first importing numpy, we are able to create an array of numbers ranging 0 to 10 in 20 evenly spaced values using x = np.linspace(0, 10, 20).



1. Create an array and save it as object y. The snippet of code below cubes the values of x and saves it to the array, y.



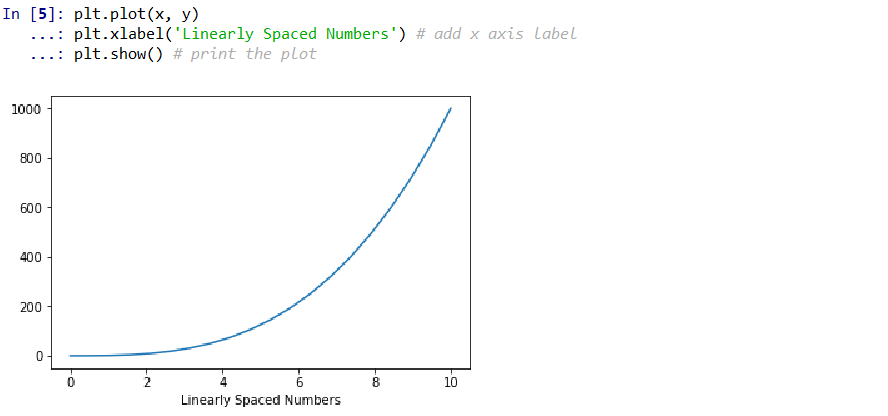
1. Create the plot using plt.plot(x, y) and print it to the console using plt.show().



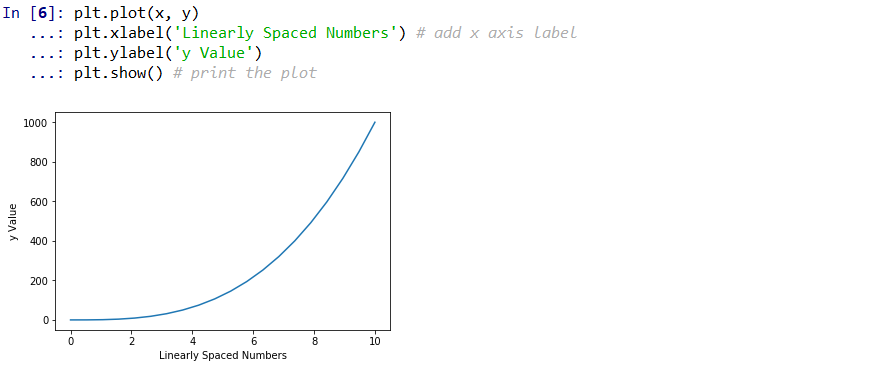
***Note***

Here, we have a plot which displays x on the x-axis and cubed values of x on the y-axis. Matplotlib makes it straightforward to add styling and labels to plots.

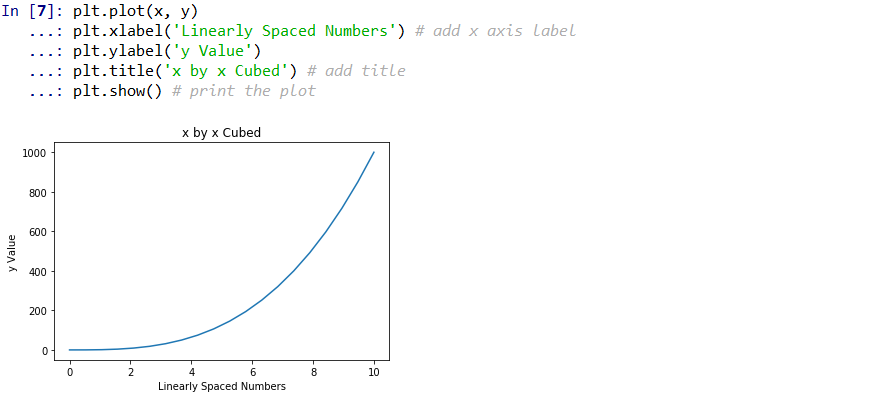
1. Add an x-axis label that reads ‘Linearly Spaced Numbers’ using plt.xlabel(‘Linearly Spaced Numbers’)



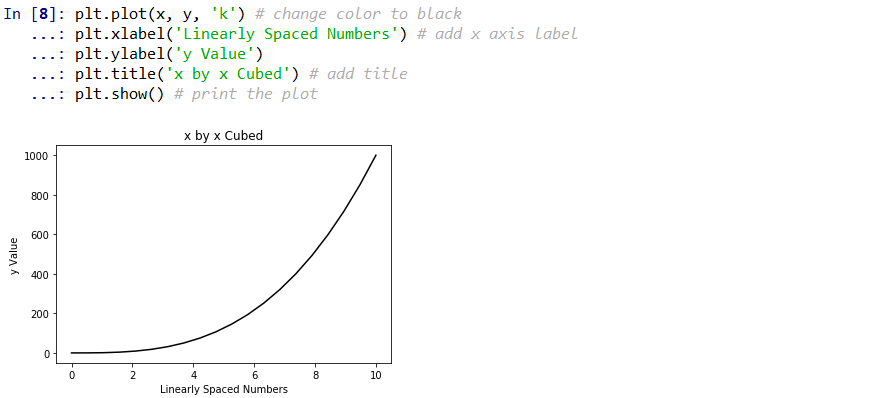
1. Add a y-axis label that reads ‘Cubed Values of x’ using plt.ylabel(‘Cubed Values of x’).



1. Add a title that reads ‘x by x cubed’ using plt.title(‘x by x Cubed’).



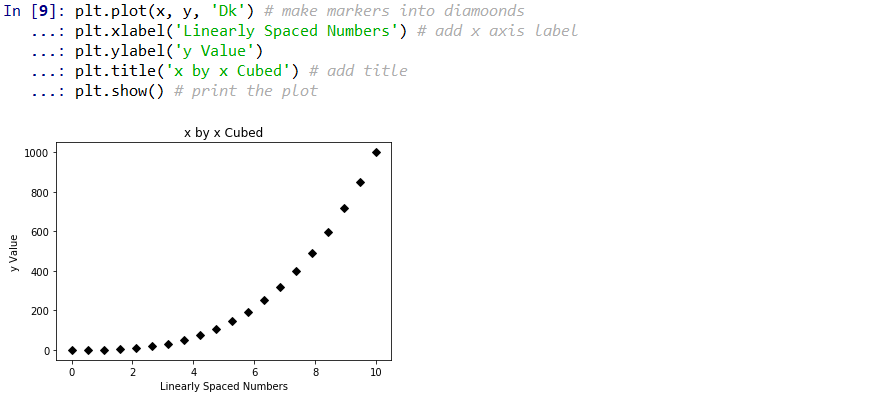
1. To change the line color specify a color argument in the plt.plot() function using plt.plot(x, y, ‘k’).



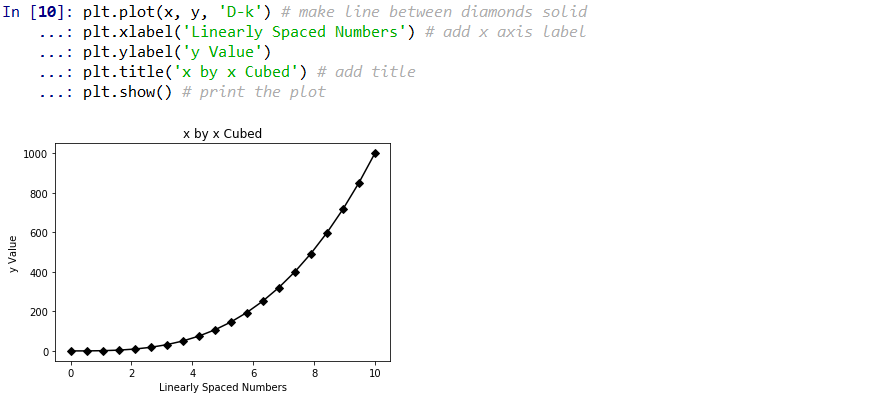
***Note***

We specified the color of the line to be black using the ‘k’ argument. For a full list of color arguments see <https://matplotlib.org/2.1.1/api/_as_gen/matplotlib.pyplot.plot.html>.

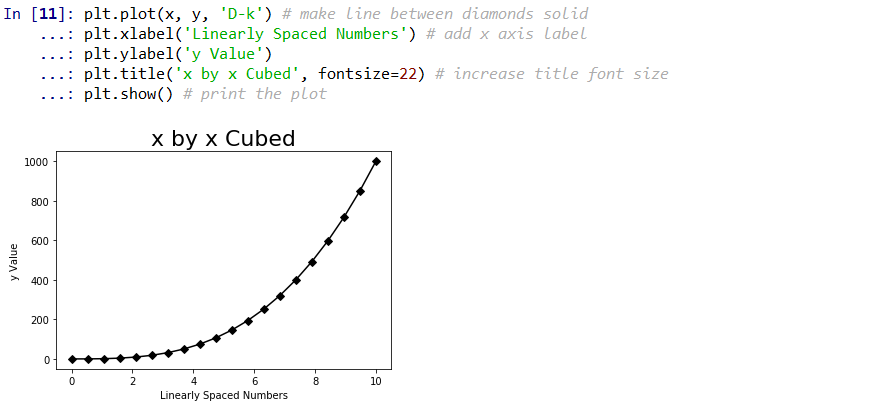
1. To make the line characters into a diamond use a character argument combined with the color character



1. We lost the solid line, so we will connect the diamonds with a solid line by placing ‘-’ between ‘D’ and ‘k’ using ‘D-k’.



1. Increase the font size of the title using the fontsize argument in the plt.title() function.



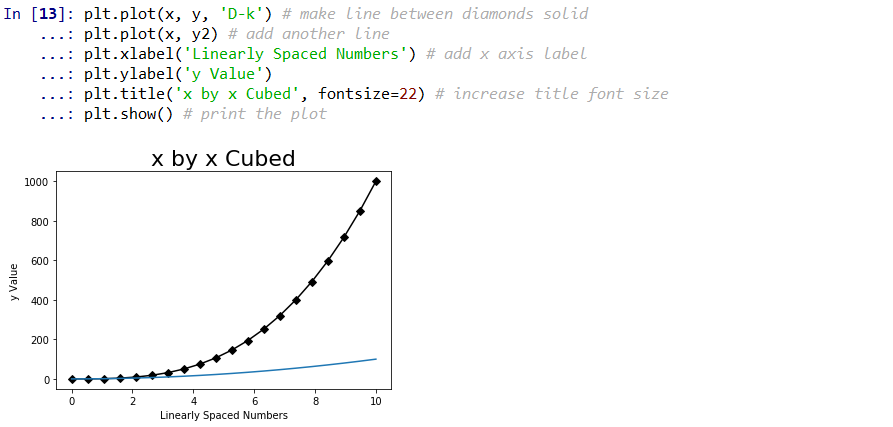
***Note***

The fontsize argument can be used in the xlabel() and xlabel() functions as well.

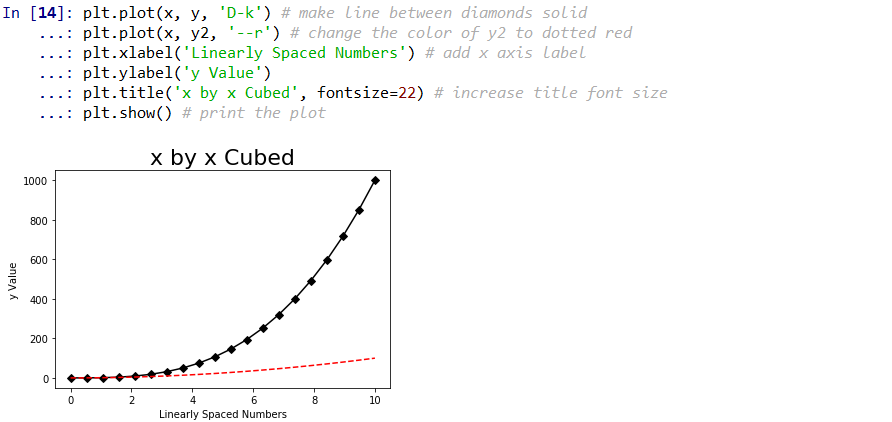
1. Add another line to the plot by simply specifying another plt.plot() function. For this exercise we will be displaying x-squared in another line. First, we must create another y object much as we did the first y object, but this time squaring x rather than cubing x.



1. Now, plot y2 in the same plot at y using plt.plot(x, y2).



1. Change the color of y2 to dotted red line using plt.plot(x, y2, ‘--r’) and adjust the title accordingly.



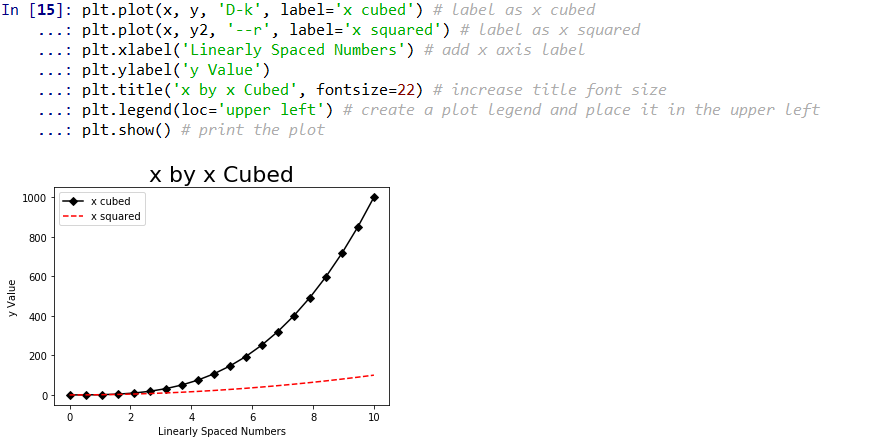
***Note***

At first glance, it is not intuitive as to what this plot is showing. We need to include a more descriptive title and a legend.

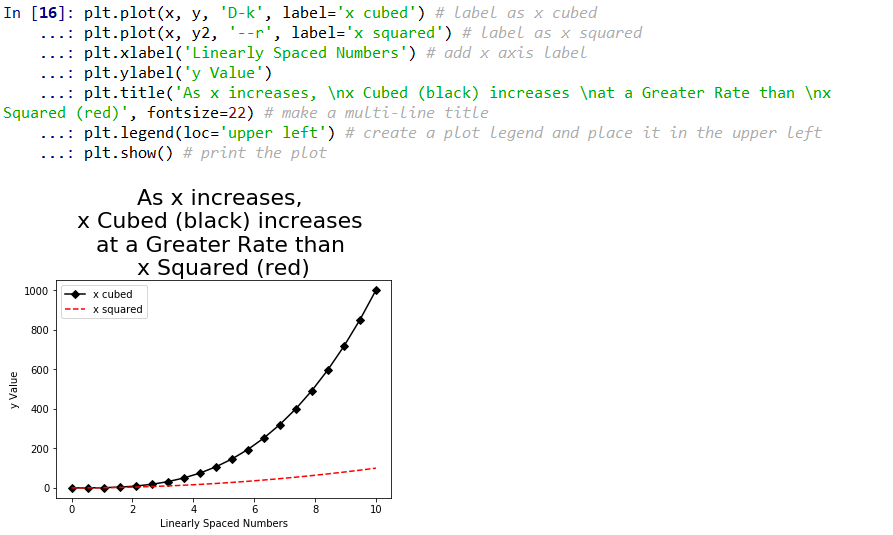
1. To create a legend we must first create labels for our lines using the label argument inside the plt.plot() functions. To label y as ‘x cubed’ use plt.plot(x, y, 'D-k', label='x cubed'). To label y2 as ‘x squared’ use plt.plot(x, y2, '--r', label='x squared'). Then, use plt.legend(loc=’upper left’) to specify the location for the legend.

***Note***

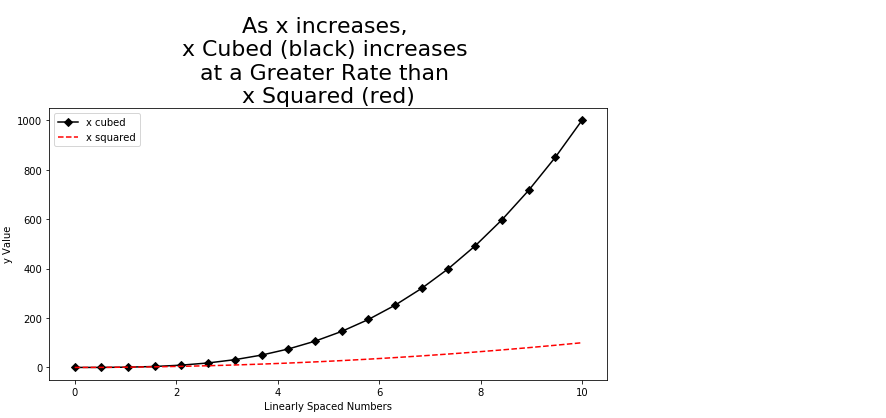
To view other arguments for legend locations see <https://matplotlib.org/api/_as_gen/matplotlib.pyplot.legend.html>.



1. It is helpful to the reader if the plot can be summarized in the title. In this plot, I want to communicate to the reader that, as x increases, x cubed increases at a greater rate than x squared. Sometimes, titles may get a little longer than we want. To break a line into new lines we use ‘\n’ at the beginning of a new line within our string. Thus, using plt.title('As x increases, \nx Cubed (black) increases \nat a Greater Rate than \nx Squared (red)', fontsize=22), we can create the title displayed below.



1. To change the dimensions of our plot we will need to add the function plt.figure(figsize=(10,5)) to the top of our plt functions. The figsize argument of 10 and 5 specify the width and height, respectively.



Activity 1: Line Plot

In this activity we will create a line plot to analyze month-to-month trends of Items Sold for months January through June. The trend will be positive, linear and will be represented using a dotted, blue line, with star markers (for styling see, <https://matplotlib.org/2.1.1/api/_as_gen/matplotlib.pyplot.plot.html>). The x-axis will be labeled ‘Month’ and the y-axis will be labeled ‘Items Sold.’ The title will say ‘Items Sold has been Increasing Linearly.’

1. Import the matplotlib library.
2. Create a list of 6 strings for x containing months January through June.
3. Create a list of 6 values for y containing values for ‘Items Sold’ that starts at 1000 and increases by 200 in each value, so the final value is 2000.
4. Generate the described plot.

**Solution:**

1. Import matplotlib using import matplotlib.pyplot as plt.



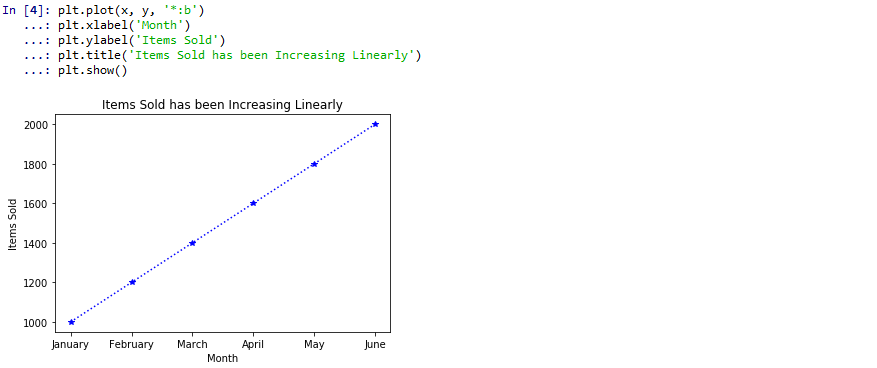
1. To create a list of 6 strings for each month January through June and save it as x use x = ['January','February','March','April','May','June'].



1. To create a list of 6 values for ‘Items Sold’ that starts at 1000 and increases by 200 so the final value is 2000 and save it as y use y = [1000, 1200, 1400, 1600, 1800, 2000].



1. To plot y (‘Items Sold’) by x (‘Month’) with a dotted blue line and star markers use plt.plot(x, y, '\*:b'). To set the x-axis to ‘Month’ use plt.xlabel('Month'). To set the y-axis to ‘Items Sold’ use plt.ylabel('Items Sold'). To set the title to ‘Items Sold has been Increasing Linearly’ use plt.title('Items Sold has been Increasing Linearly').



Exercise 2: Bar Plot

Now, that we have the basics of creating a line plot in matplotlib, we will explore how to create another very common type of plot, the bar plot. While line charts are most powerful when presenting trends, bar plots strive in displaying the differences between, or among, groups. Thus, in this example, we will be displaying Sales Revenue by Item type.

1. Import the matplotlib package using import matplotlib.pyplot as plt.



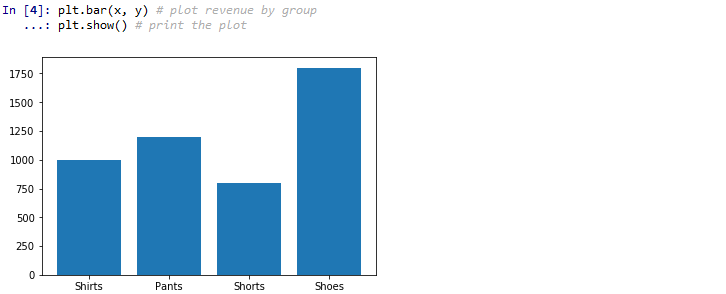
1. Create a list of item types and save it as x using x = ['Shirts', 'Pants','Shorts','Shoes'].



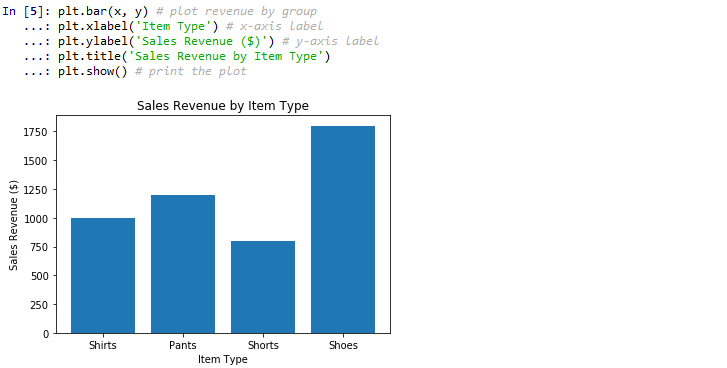
1. Create a list of Sales Revenue and save it as y using y = [1000, 1200, 800, 1800].



1. Create a bar plot using plt.bar(x, y) and print the plot using plt.show().



1. Add a title, an x label, and y label as we did in Exercise 1.



***Note***

While this title explains what the plot is displaying, it does not tell the reader what is going on in the plot. Next, we will go over how to make a programmatic title explaining to the reader what is happening in the plot.

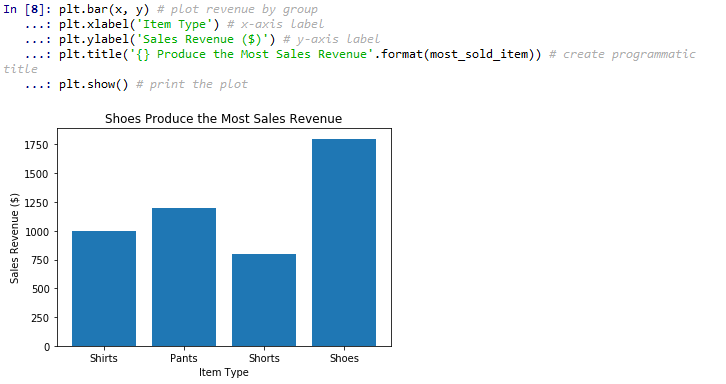
1. We are going to create a title that will change to the data which is plotted. For this example, it will read “Shoes Produce the Most Sales Revenue.” First, we will find the index of the maximum value in y and save it as the object index\_of\_max\_y using index\_of\_max\_y = y.index(max(y)).



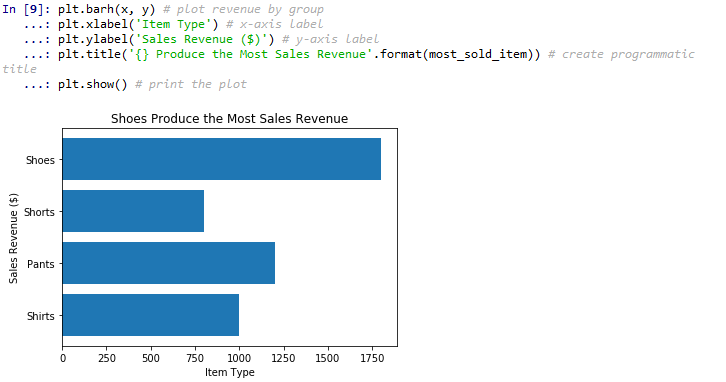
1. Save the item in list x with index equaling index\_of\_max\_y to the object most\_sold\_item using most\_sold\_item = x[index\_of\_max\_y].



1. Make the title programmatic using plt.title('{} Produce the Most Sales Revenue'.format(most\_sold\_item)). This code places the value for most\_sold\_item in place of {}.



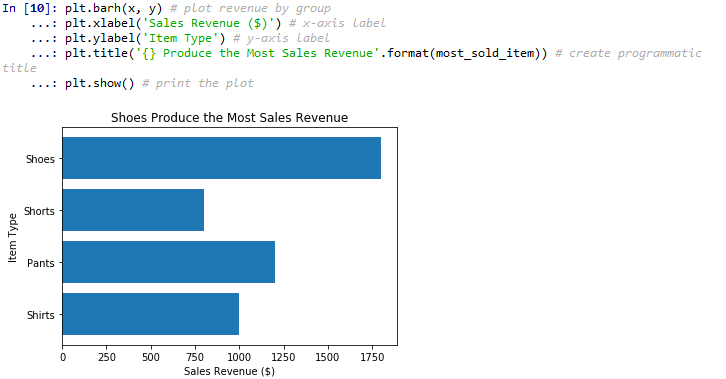
1. Turn the plot into a horizontal bar plot by preplacing plt.bar(x, y) with plt.barh(x, y).



***Note***

Remember, when a bar plot is transformed from vertical to horizontal that the x and y axes need to be switched.

1. Switch x and y labels from plt.xlabel('Item Type') and plt.ylabel('Sales Revenue ($)'), respectively, to plt.xlabel('Sales Revenue ($)') and plt.ylabel('Item Type'), respectively.



Activity 2: Bar Plot

In this exercise, we will be creating a bar plot comparing the number of NBA championships among the 5 franchises with the most championships. The plot will be sorted so the franchise with the greatest number of championships is on the left and the franchise with the least is on the right. The bars will be red, the x-axis will be titled ‘NBA Franchises’, the y-axis will be titled ‘Number of Championships’, and the title will be programmatic explaining which franchise has the most titles and how many they have. Additionally, we will rotate the x tick labels 45 degrees so they do not overlap, and we will save our plot to the current directory.

1. Import the matplotlib library.
2. Create a list of 5 strings for x containing the names of NBA franchises with the most titles.
3. Create a list of 5 values for y containing values for ‘Titles Won’ that corresponds with the strings in x.
4. Place x and y into a data frame with the column names ‘Team’ and ‘Titles’, respectively.
5. Sort the data frame descending by ‘Titles.’
6. Make a programmatic title and save it as title.
7. Generate the described plot.
8. Fix the cropping issue upon saving.

**Solution:**

1. Import the matplotlib library using import matplotlib.pyplot as plt.



1. Create a list of 5 strings for x containing the names of NBA franchises with the most titles using x = ['Boston Celtics','Los Angeles Lakers', 'Chicago Bulls', 'Golden State Warriors', 'San Antonio Spurs'].



1. Create a list of 5 values for y containing values for ‘Titles Won’ that corresponds with the strings in x using y = [17, 16, 6, 6, 5].



1. Place x and y into a data frame with the column names ‘Team’ and ‘Titles’, respectively by importing pandas using import pandas as pd. Once pandas has been imported, create the data frame using df = pd.DataFrame({'Team': x, 'Titles': y}).



1. Sort the data frame descending by ‘Titles’ and save it as df\_sorted using df\_sorted = df.sort\_values(by=('Titles'), ascending=False).



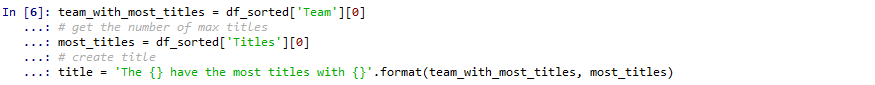
***Note***

If we sort with ascending=True the plot will have the larger values to the right. Since we want the larger values on the left we will be using ascending=False.

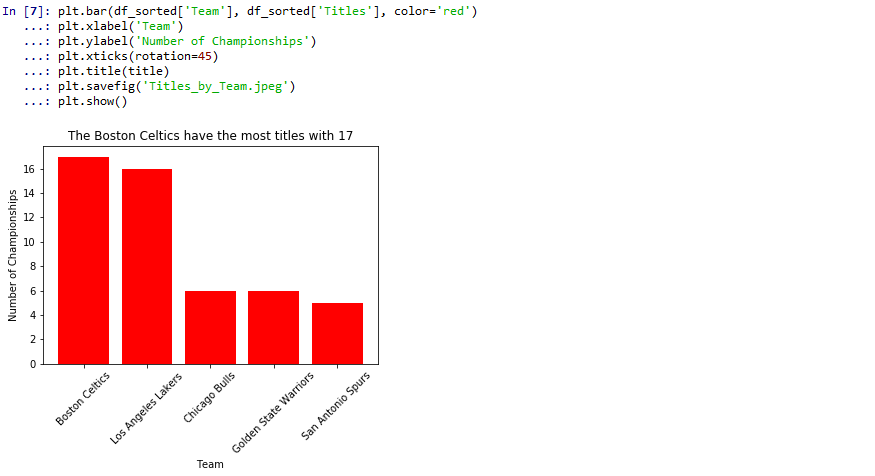
1. Make a programmatic title and save it as title by first finding the team with the most titles and saving it as the object team\_with\_most\_titles using team\_with\_most\_titles = df\_sorted['Team'][0]. Then, get the number of titles for the team with the most titles using most\_titles = df\_sorted['Titles'][0]. Then, with the format() function, create a string that reads 'The Boston Celtics have the most titles with 17' using title = 'The {} have the most titles with {}'.format(team\_with\_most\_titles, most\_titles).

***Note***

Since the data is sorted with the largest numbers at the top (I.e., descending) we can access the maximum values by referencing index 0.



1. To use a bar graph to plot the number of titles by team use plt.bar(df\_sorted['Team'], df\_sorted['Titles'], color='red'). Set the x-axis label to ‘Team’ using plt.xlabel('Team'). Set the y-axis label to ‘Number of Championships’ using plt.ylabel('Number of Championships'). Prevent the x tick labels from overlapping by rotating them 45 degrees using plt.xticks(rotation=45). Set the title of the plot to the programmatic title object we created using plt.title(title). Save the plot to our current working directory as ‘Titles\_by\_Team.jpeg’ using plt.savefig('Titles\_by\_Team.jpeg').



***Note***

When we print the plot to the console using plt.show() it appears as intended, however, when we open the file we created titled ‘Titles\_by\_Team.jpeg’ we see that it crops the x tick labels.

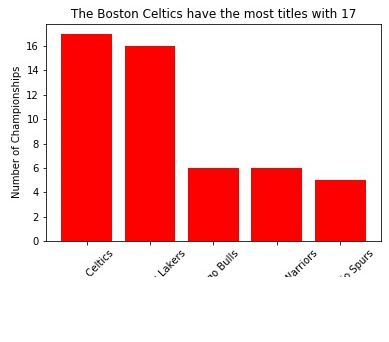
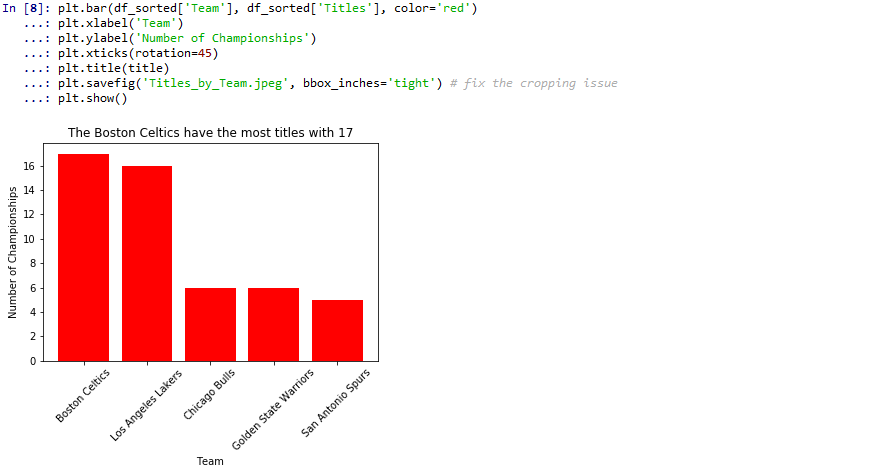


Figure 5.4. Saved figure with x tick labels cropped.

1. To fix the cropping issue use the argument bbox\_inches=’Tight’ inside the plt.savefig() function using plt.savefig('Titles\_by\_Team.jpeg', bbox\_inches='tight').



***Note***

Now, if we open the saved file ‘Titles\_by\_Team.jpeg’ from our working directory we see that the x tick labels are not cropped.

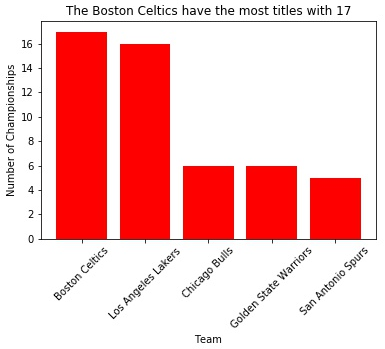


Figure 5.5. Saved figure with x tick labels fixed.

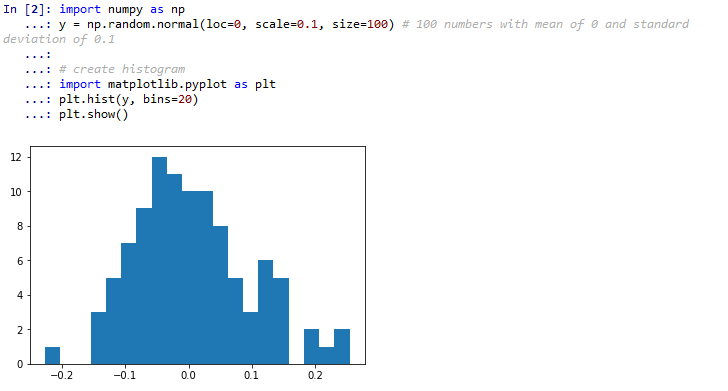
Exercise 3: Histogram

In statistics, it is essential to be aware of the distribution of continuous variables. To display the distribution we will use a histogram. Histograms display the frequency by the bin.

1. To demonstrate the creation of a histogram we must first generate an array of normally distributed values and save it as y. First, use import numpy as np. Then, using numpy’s random.normal function, use y = np.random.normal(loc=0, scale=0.1, size=100). This generates 100 normally distributed numbers with a mean of 0 and a standard deviation of 0.1.



1. With matplotlib imported, create the histogram using plt.hist(y, bins=20). The ‘bins’ argument determines the extent of the histogram’s granularity. Thus, the range of y is divided into 20 evenly spaced buckets, or bins.



***Note***

Since we covered styling arguments in great detail in Exercises 2 and 3, we will skip this portion and go right into making the title tell us what we need to know.

When we are looking at a histogram, we are determining whether or not the distribution is normal. There is a separate test for normality termed the Shapiro-Wilk test. The null hypothesis for the Shapiro-Wilk test is that the data is normally distributed. Thus, a p-value < 0.05 indicates a non-normal distribution while a p-value > 0.05 indicates a normal distribution.

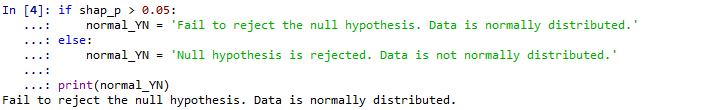
1. To return the W test statistic and the corresponding p-value from the Shapiro-Wilk test we must import the shapiro function using from scipy.stats import shapiro. The shapiro function returns a tuple containing two values. The first is the W statistic while the second is the p-value. Thus, we can use tuple unpacking to save the W statistic and the p-value into the objects shap\_w and shap\_p, respectively, using shap\_w, shap\_p = shapiro(y).



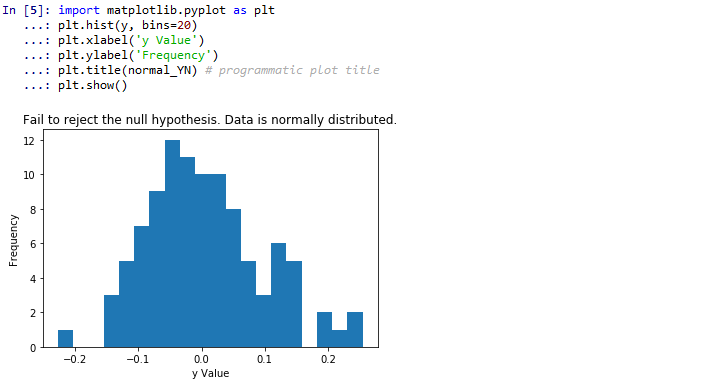
***Note***

The p-value is > 0.05, so, according to the Shapiro-Wilk test, y is normally distributed. We will set up some logic to make this decision programmatic.

1. We will use an if-else statement that assigns the string 'Fail to reject the null hypothesis. Data is normally distributed' to the object normal\_YN if shap\_p > 0.05 and assigns the string 'Null hypothesis is rejected. Data is not normally distributed' is shap\_p > 0.05.



1. Assign normal\_YN to our plot using plt.title(normal\_YN).



Exercise 4: Scatterplot

A common plot used for displaying a relationship between two continuous variables is the scatterplot. In this exercise, we will be creating a scatterplot of weight vs height. We will, again, create a title explaining the message for which the plot is portraying.

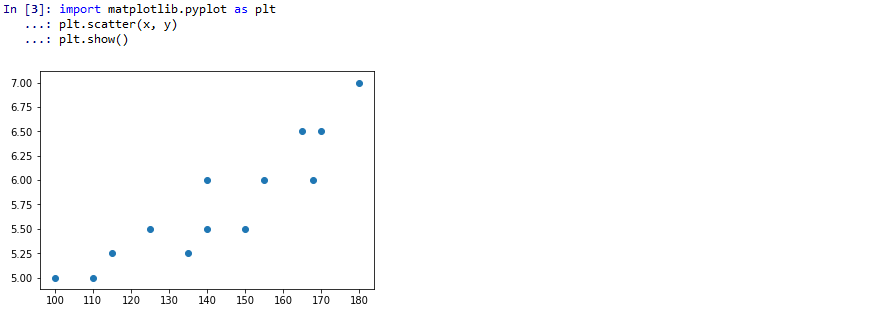
1. Generate a list of numbers representing height and save it as y using y = [5, 5.5, 5, 5.5, 6, 6.5, 6, 6.5, 7, 5.5, 5.25, 6, 5.25].



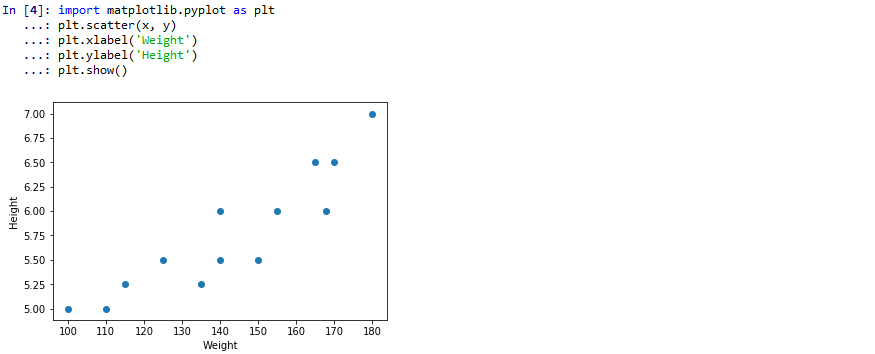
1. Generate a list of numbers representing weight and save it as x using x = [100, 150, 110, 140, 140, 170, 168, 165, 180, 125, 115, 155, 135].



1. Import matplotlib using import matplotlib.pyplot as plt. Create a basic scatterplot using plt.scatter(x, y).



1. Create an x-axis label using plt.xlabel('Weight'). Create a y-axis label using plt.ylabel('Height').



***Note***

We want our plot to tell us the strength of the relationship and the Pearson correlation coefficient, so we need to calculate this and set up some logic for our programmatic title.

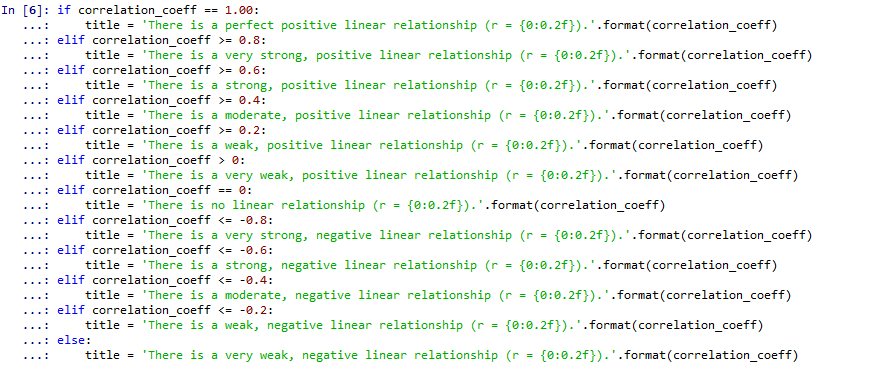
1. Import the pearsonr function (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html) using from scipy.stats import pearsonr. The pearsonr function returns a tuple; the first value in the tuple correlation coefficient and the second is the p-value. We can unpack the tuple using correlation\_coeff, p\_value = pearsonr(x, y).



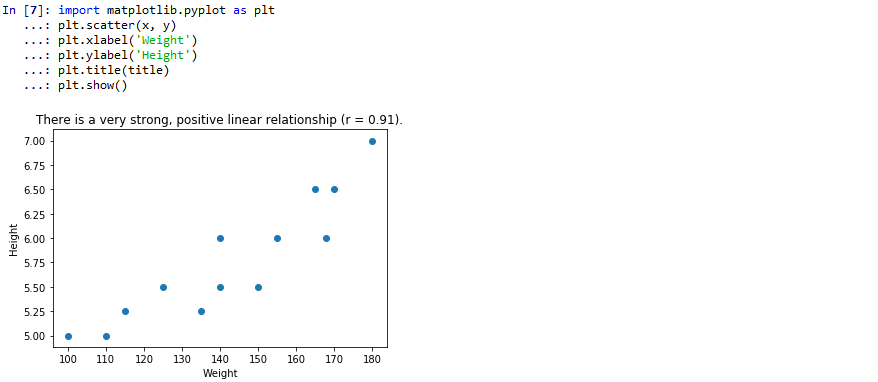
***Note***

Now that we have our correlation coefficient, we can set up some logic detailing the strength and direction of the relationship as well as the coefficient.

1. Set-up some logic to interpret the value of the correlation coefficient.



1. Now, we can use the object title as our title using plt.title(title).



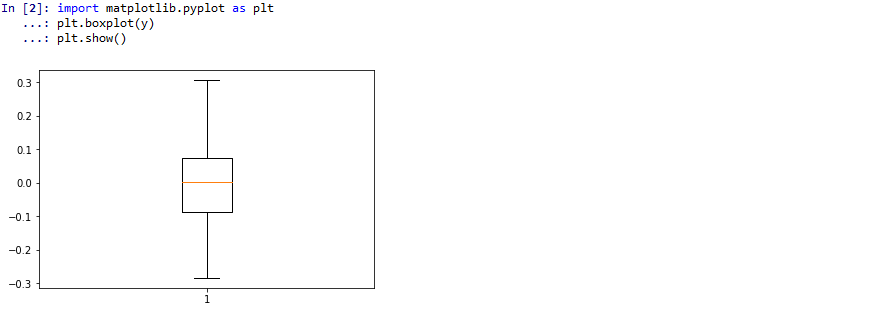
Exercise 5: Box-and-Whisker plot

The last plot we will cover is the Box-and-Whisker plot. This plot is used primarily to indicate skew of a distribution and to identify outliers. We will portray this information in our title.

1. Import numpy using import numpy as np. Generate an array of 100 normally distributed numbers with a mean of 0 and a standard deviation of 0.1 and save it as y using y = np.random.normal(loc=0, scale=0.1, size=100).



1. Import matplotlib using import matplotlib.pyplot as plt. Then, create the plot using plt.boxplot(y).



1. Calculate the p-value from the Shapiro-Wilk test by importing the shapiro function using from scipy.stats import shapiro. The, using tuple unpacking to save the Shapiro W and p-value from the shapiro function using shap\_w, shap\_p = shapiro(y).



1. Import the zscore function using from scipy.stats import zscore. Then, use the function to convert y into z-scores using y\_z\_scores = zscore(y).



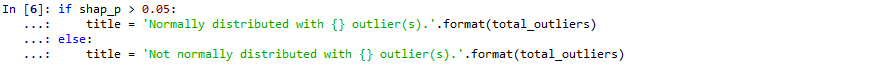
1. Iterate through the y\_z\_scores array to find how many outliers there are. Set the object total\_outliers to 0 using total\_outliers = 0. Then, iterating through every index of y\_z\_scores, check to see if the value is greater than or equal to 3. If the value is at least 3, increase the value for total\_outliers by 1.

***Note***

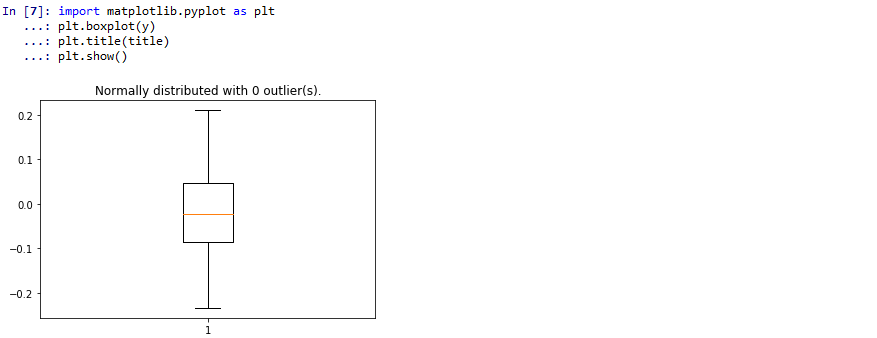
Because the array, y, was generated to be normally distributed we can expect that there are no outliers in the data.



1. Set-up some logic for our programmatic title. Our title will tell us if the data is normally distributed as well as the number of outliers. If shap\_p is greater than 0.05 then our data is normally distributed. If it is not greater than 0.05 then our data is not normally distributed. We can set this up and include the number of outliers by the logic below.



1. Set out plot title as the programmatically named title using plt.title(title).



Object-Oriented Method Using Subplots

*Present 17-18: Introduction to the Object-Oriented method*

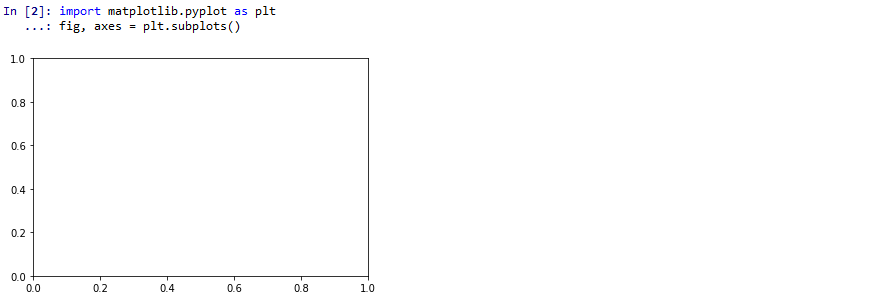
However, it is much more common to see the use of subplots in Matplotlib plots than to see the functional method. In the object-oriented method we create a figure object which acts as an empty canvas and then we add a set of axes to it. The figure object is callable and if called will return the figure to the console. We will demonstrate how this works by plotting the same x and y objects as we did in Exercise 1.

Exercise 6: Single Line Plot using Subplots

1. Save x as an array ranging from 0 to 10 in 20 linearly spaced steps using x = np.linspace(0, 10, 20). Then, save y as x cubed using y = x\*\*3.



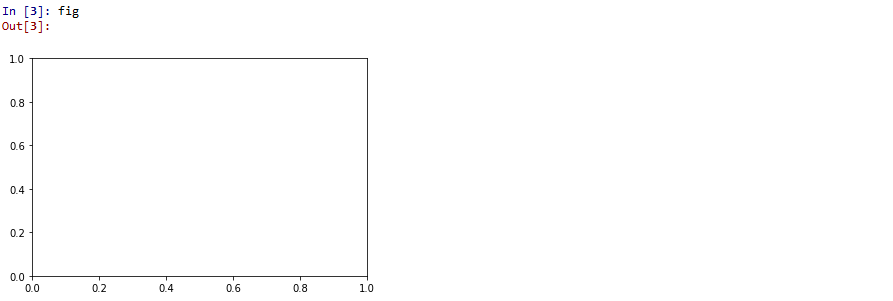
1. Import matplotlib using import matplotlib.pyplot as plt. Use tuple unpacking to create a figure and a set of axes using fig, axes = plt.subplots().



***Note***

By default, matplotlib creates a figure with one set of axes. To create multiple plots we will provide arguments inside the subplots() function.

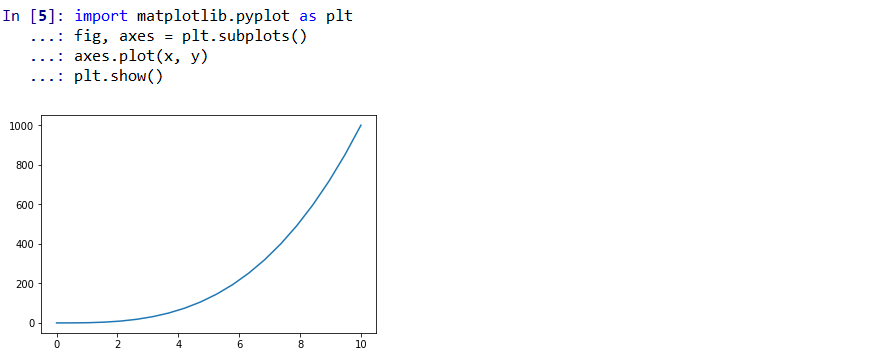
1. If we now just run fig we can see the axes that were created.



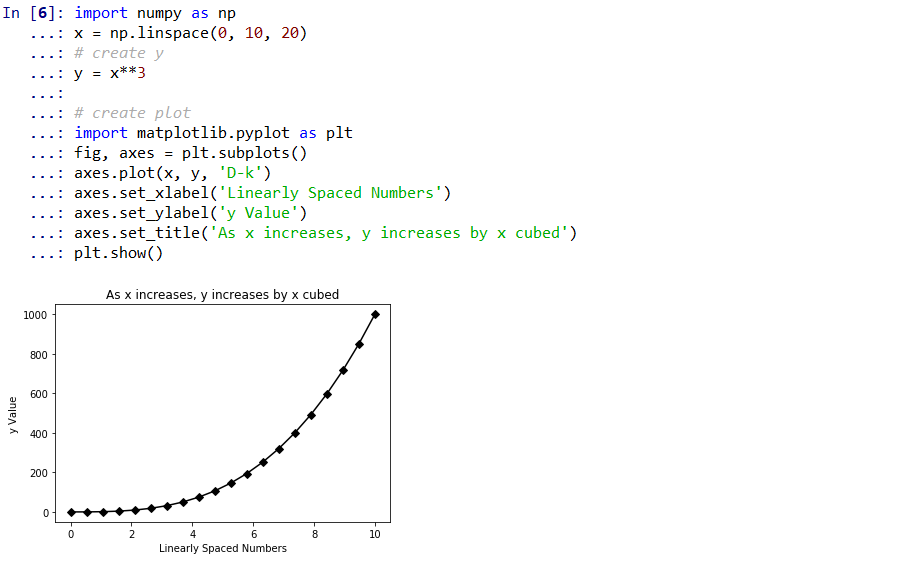
1. If we run axes we get:



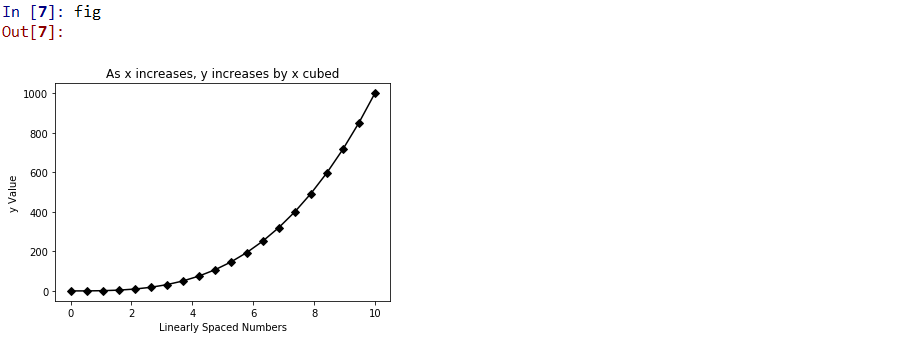
1. To plot y by x (i.e., x squared), we use axes.plot(x, y) and show the plot using plt.show().



1. Style the plot much the same as in Exercise 1. Set the x-axis label to ‘Linearly Spaced Numbers’ using axes.set\_xlabel('Linearly Spaced Numbers'). To set the y-axis to ‘y Value’ use axes.set\_ylabel('y Value'). To set the title to ‘As x increases, y increases by x cubed’ use axes.set\_title('As x increases, y increases by x cubed'). Lastly, to change the line color and markers use the argument ‘D-k’ in the axes.plot() function.



1. Call the fig object using fig



***Note***

We created a plot very similar to the first plot in Exercise 1, but now it is a callable object. A further advantage of using the object-oriented approach is the ability to create multiple sublots on a single figure object.

*Discuss: What might be some advantages to having a callable figure object?*

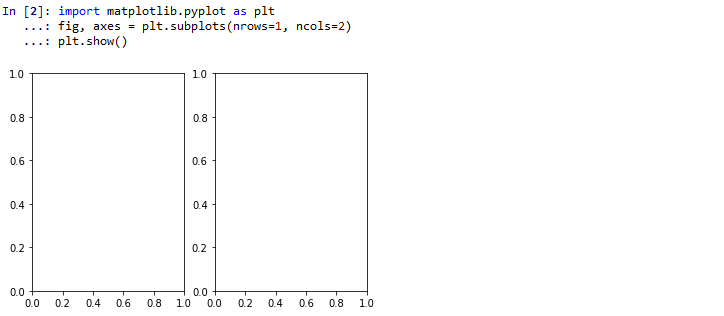
Exercise 7: Multiple Line Plots Using Subplots

In this exercise, we will plot the same lines as in Exercise 1, but we will plot them on two subplots.

After, x, y, and y2 have been created (for more details, see Exercise 1):



1. Import matplotlib using import matplotlib.pyplot as plt. Create a figure with 2 axes (I.e., plots) that are side-by-side (I.e., 1 row with 2 columns) using fig, axes = plt.subplots(nrows=1, ncols=2).



***Note***

To access the various axes we will treat the axes object as a list. The plot on the left is the first index of the axes object so we can reference it using axes[0]. The plot on the right is second index of the axes object so we reference it using axes[1] (see Figure 5.6).

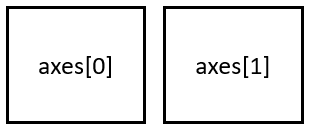


Figure 5.6. Axes index referencing in a figure object with 1 row and multiple columns.

If the figure contained one column of two axes (I.e., if there were two axes stacked on top of one another), the top plot would be index 0 while the bottom plot would be index 1 (see Figure 5.7).

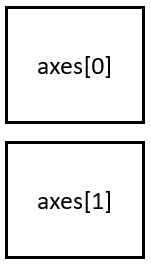


Figure 5.7. Axes index referencing in a figure object with multiple rows and 1 column.

However, if our figure contains multiple rows and multiple columns, we reference the plot using [row, column]. For example, if our figure object contains 4 subplots organized in 2 rows and 2 columns, we would index reference the top left plot using axes[0,0] and the bottom right plot using axes[1,1] (see Figure 5.8). This will become and extremely important concept to understand to complete activity 3.

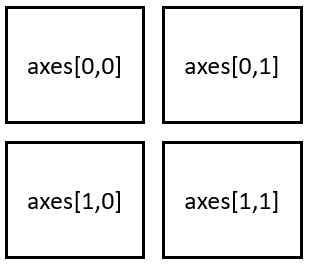
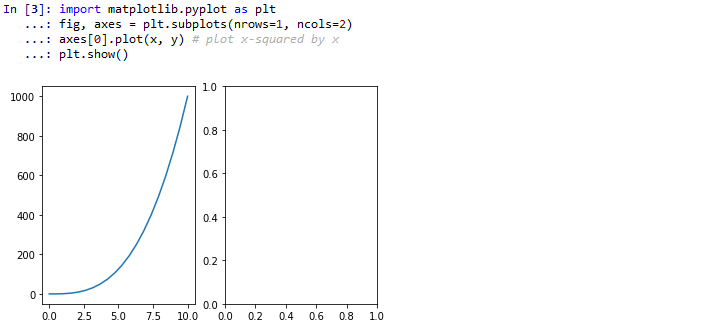
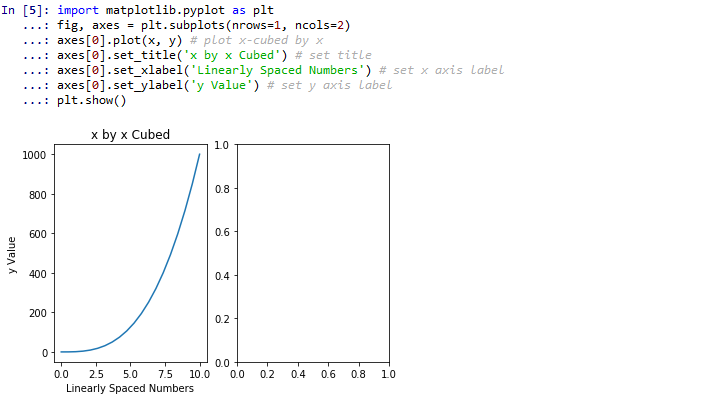


Figure 5.8. Axes index referencing in a figure object with multiple rows and multiple columns.

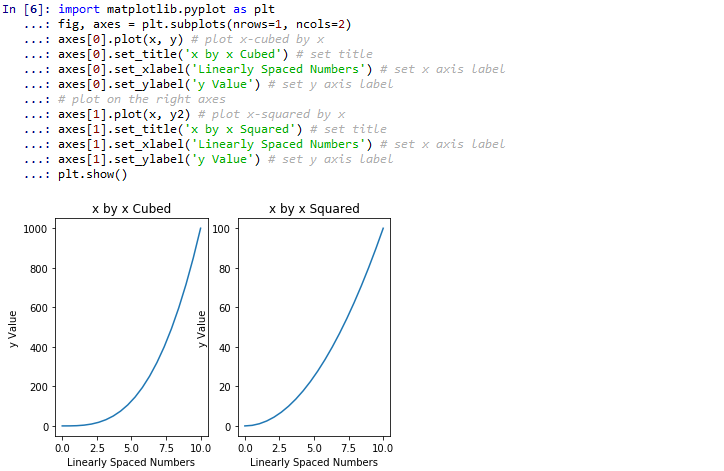
1. To access the plot on the left refer to it as axes[0]. To access the plot on the right refer to it as axes[1]. On the left axes, plot y by x using axes[0].plot(x, y).



1. Style it the same way we did in Exercise 3 by adding a title, x label, and y label.



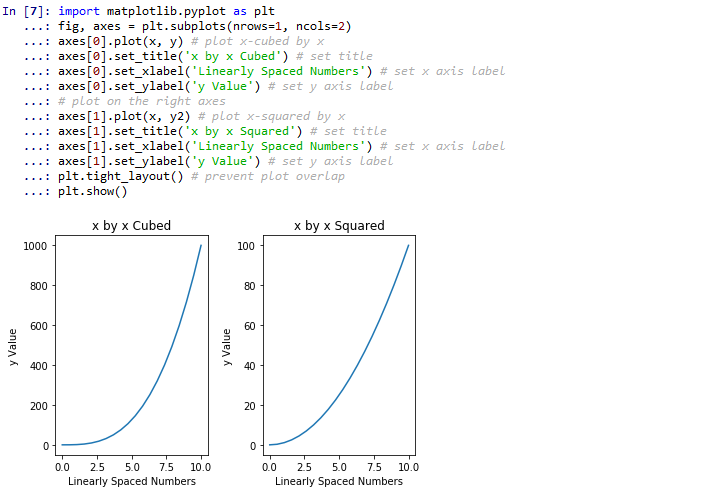
1. On the right axes, plot y by x using axes[1].plot(x, y2) and use the same styling as in step 5.



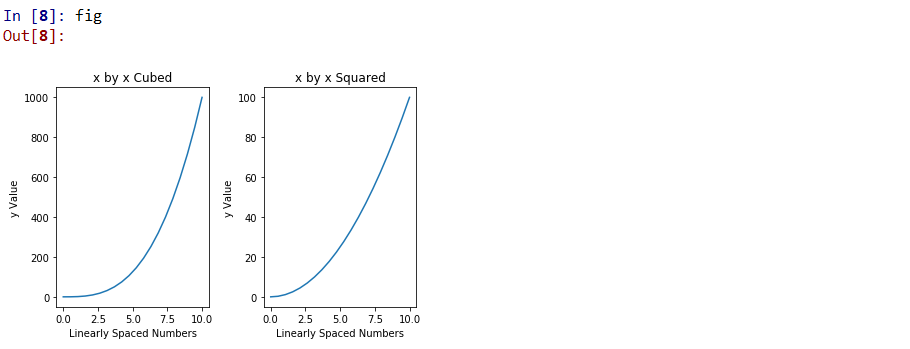
***Note***

We have successfully created two subplots. However, it looks like the y-axis of the plot on the right is overlapping onto the left plot.

1. Fix the overlapping of the plots using plt.tight\_layout().



1. Call the fig object using fig.



***Note***

Using the object-oriented approach, we are able to display both plots just by calling the fig object. We will get more practice doing this in Activity 3.

Activity 3: Multiple Plot Types using Subplots

In exercises 1-5 and activities 1-2, we learned to build, customize, and program line plots, bar plots, histograms, scatterplots, and box-and-whisker plots using the Functional method. In exercise 6, we were introduced to the Object-Oriented Method and in exercise 7 we learned how to create a figure with multiple plots using subplots. Thus, in this activity we will be leveraging subplots to create a figure with multiple plots and plot types. We will be creating a figure with 6 subplots. The subplots will be displayed in 3 rows and 2 columns (see Figure 5.9).

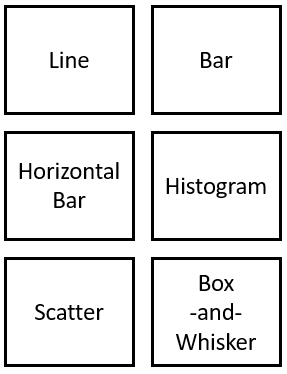


Figure 5.9. Layout for subplots.

***Note***

Once we have generated our figure of 6 subplots, we access each subplot using ‘row, column’ indexing (see Figure 5.10).

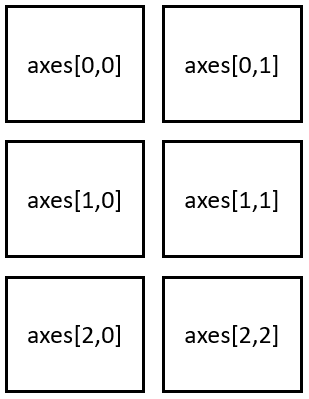


Figure 5.10. Axes index referencing in a figure object with 3 rows and 2 columns.

Thus, to access the Line plot (I.e., top left) use axes[0, 0]. To access the histogram (I.e., middle right) use axes[1, 1]. We will be using this in the following exercise.

1. Import the ‘Items\_Sold\_by\_Week.csv’ file and save it as the data frame object Items\_by\_Week.
2. Create 1 normally distributed array of numbers and save it as y to use for the histogram and box-and-whisker plots.
3. Import the ‘Weight\_by\_Height.csv’ file and save it as the data frame object Weight\_by\_Height.
4. Generate a figure with 6 subplots using 3 rows and 2 columns.
5. Make sure the plots do not overlap.
6. Set the plot titles so they match with Figure 5.9.
7. In the ‘Line’ axis, plot items sold by week from the Items\_Sold\_by\_Week data frame.
8. In the ‘Bar’ axis, plot items sold by week from the Items\_Sold\_by\_Week data frame.
9. In the ‘Horizontal Bar’ axis, plot items sold by week from the Items\_Sold\_by\_Week data frame.
10. In the ‘Histogram’ axis, plot the distribution of y.
11. In the ‘Scatter’ axis, plot weight by height from the Weight\_by\_Height data frame.
12. In the ‘Box-and-Whisker' axis, plot y.
13. Label the x- and y-axis in each subplot.
14. Increase the size of the figure.
15. Save the figure.

**Solution:**

1. Import pandas and alias it as pd using import pandas as pd. Import the ‘Items\_Sold\_by\_Week.csv’ file and save it as the data frame object Items\_by\_Week using Items\_by\_Week = pd.read\_csv('Items\_Sold\_by\_Week.csv').



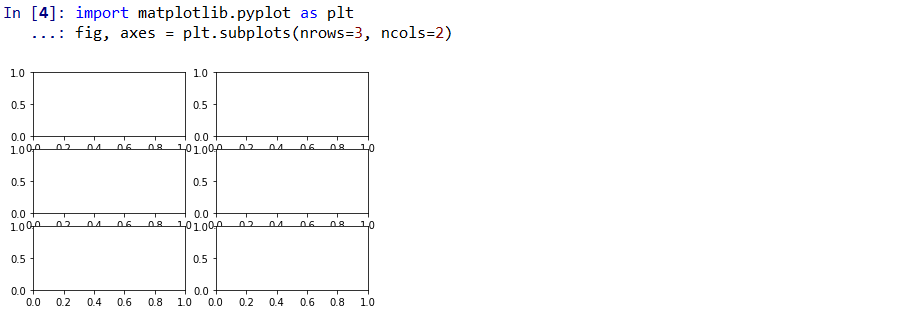
1. Generate an array of 100 normally distributed numbers to use as the data for the histogram and box-and-whisker plots and save it as y using y = np.random.normal(loc=0, scale=0.1, size=100).



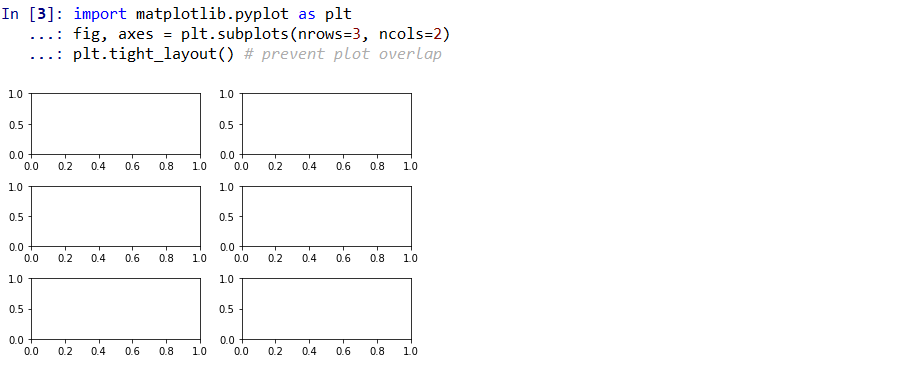
1. Import pandas and alias it as pd using import pandas as pd. Import the Weight\_by\_Height.csv’ file and save it as the data frame object Weight\_by\_Height using Weight\_by\_Height = pd.read\_csv('Weight\_by\_Height.csv').



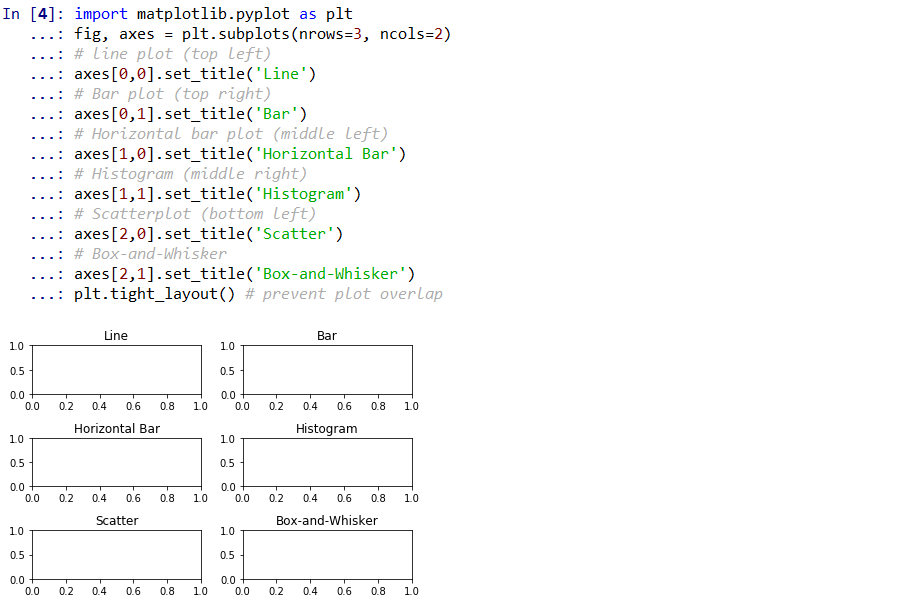
1. To generate a figure with 6 subplots organized in 3 rows and 2 columns we must first import matplotlib using import matplotlib.pyplot as plt. Then generate the fig object using fig, axes = plt.subplots(nrows=3, ncols=2).



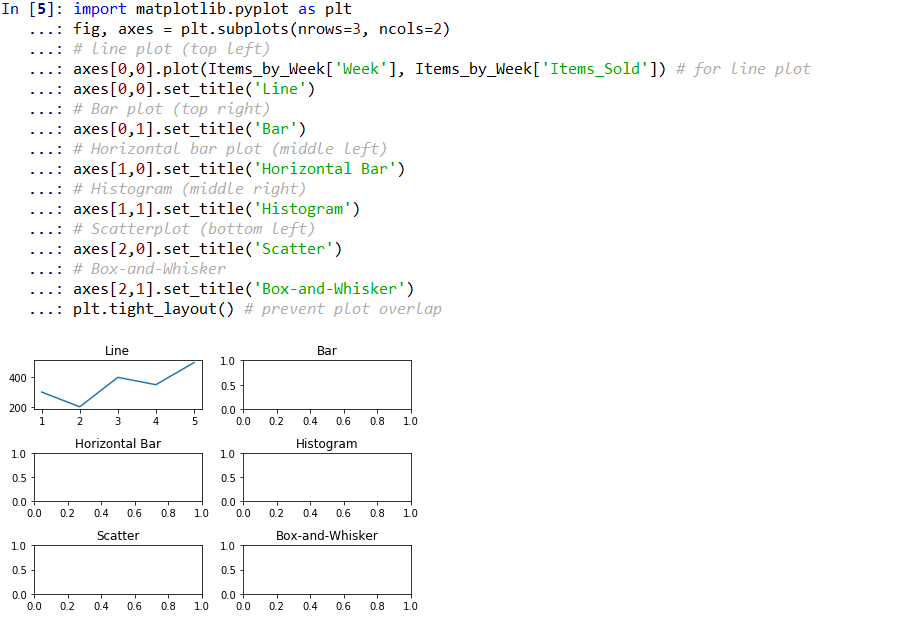
1. To make sure the plots do not overlap use plt.tight\_layout().



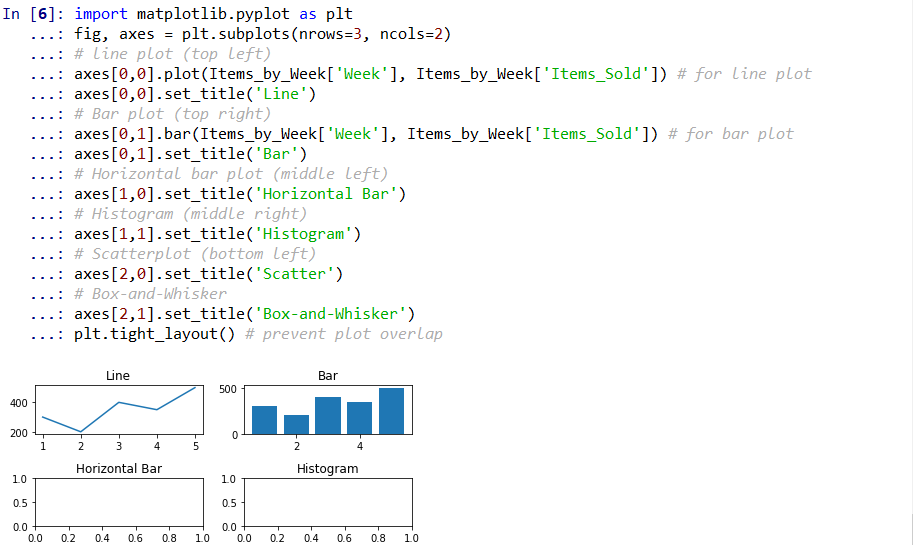
1. Set the respective axes titles to match those in Figure 5.9 using:
   1. axes[0,0].set\_title('Line') for the line plot
   2. axes[0,1].set\_title('Bar') for the bar plot
   3. axes[1,0].set\_title('Horizontal Bar') for the horizontal bar plot
   4. axes[1,1].set\_title('Histogram') for the histogram
   5. axes[2,0].set\_title('Scatter') for the scatterplot
   6. axes[2,1].set\_title('Box-and-Whisker') for the box-and-whisker plot



1. Plot ‘Items\_Sold’ by ‘Week’ from the ‘Items\_by\_Week’ data frame in the ‘Line’ axes using axes[0,0].plot(Items\_by\_Week['Week'], Items\_by\_Week['Items\_Sold']).



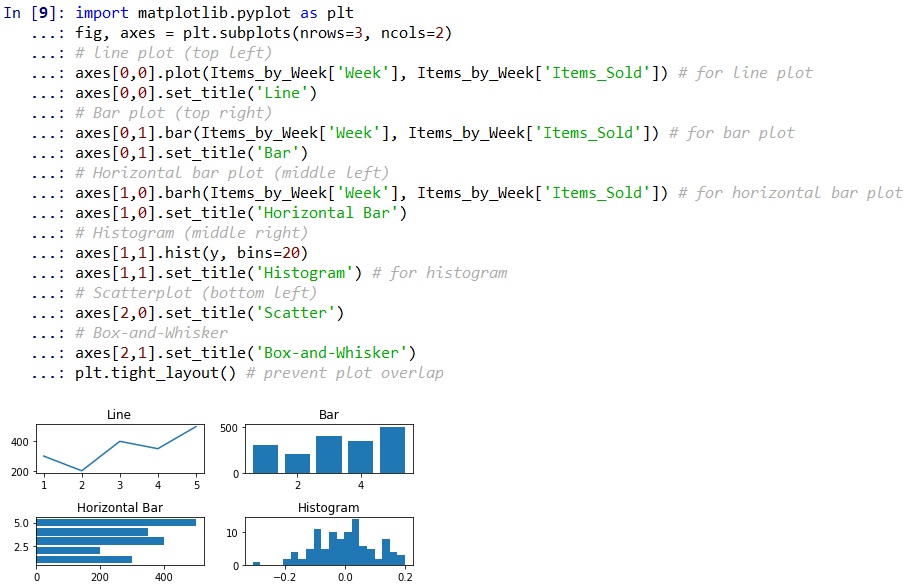
1. Plot ‘Items\_Sold’ by ‘Week’ from the ‘Items\_by\_Week’ data frame in the 'Bar’ axes using axes[0,1].bar(Items\_by\_Week['Week'], Items\_by\_Week['Items\_Sold']).



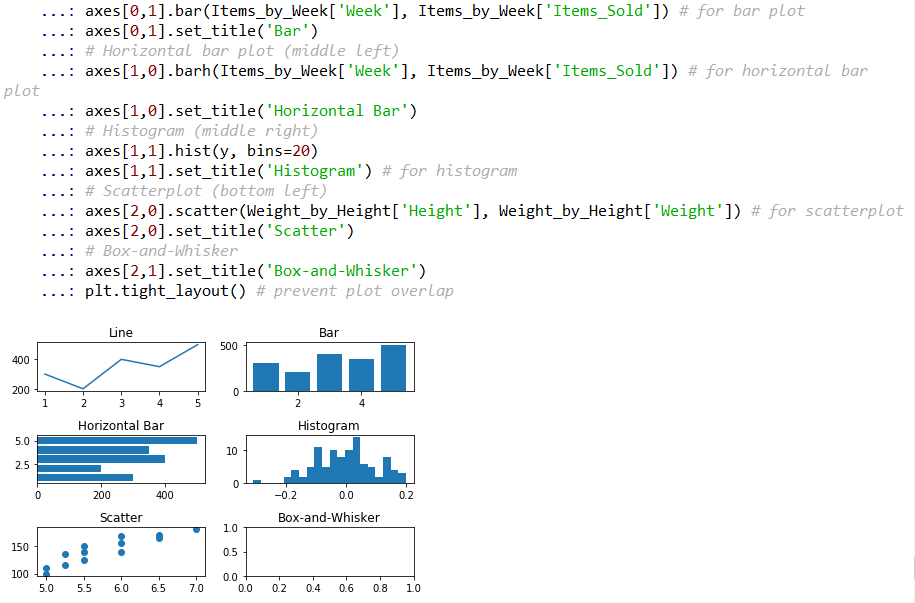
1. Plot ‘Items\_Sold’ by ‘Week’ from the ‘Items\_by\_Week’ data frame in the 'Horizontal Bar’ axes using axes[1,0].barh(Items\_by\_Week['Week'], Items\_by\_Week['Items\_Sold']).



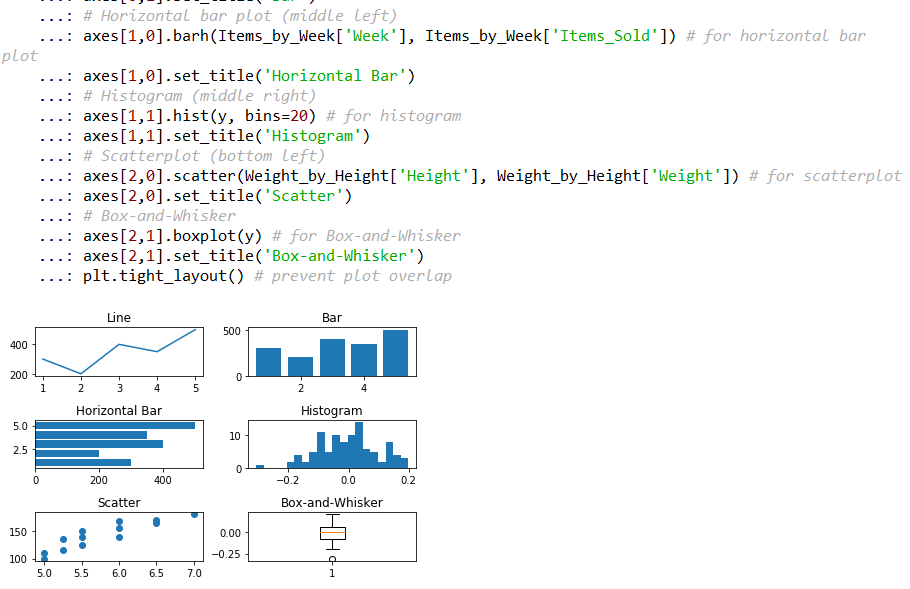
1. Plot the distribution of y using axes[1,1].hist(y, bins=20).



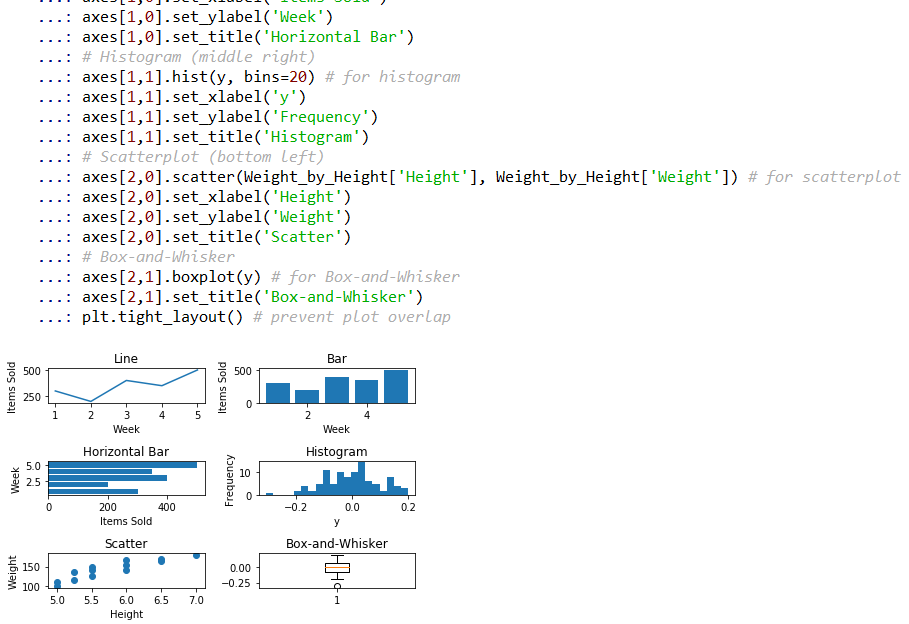
1. Plot Weight by Height in the ‘Scatterplot’ axes from the ‘Weight\_by\_Height’ data frame using axes[2,0].scatter(Weight\_by\_Height['Height'], Weight\_by\_Height['Weight']).



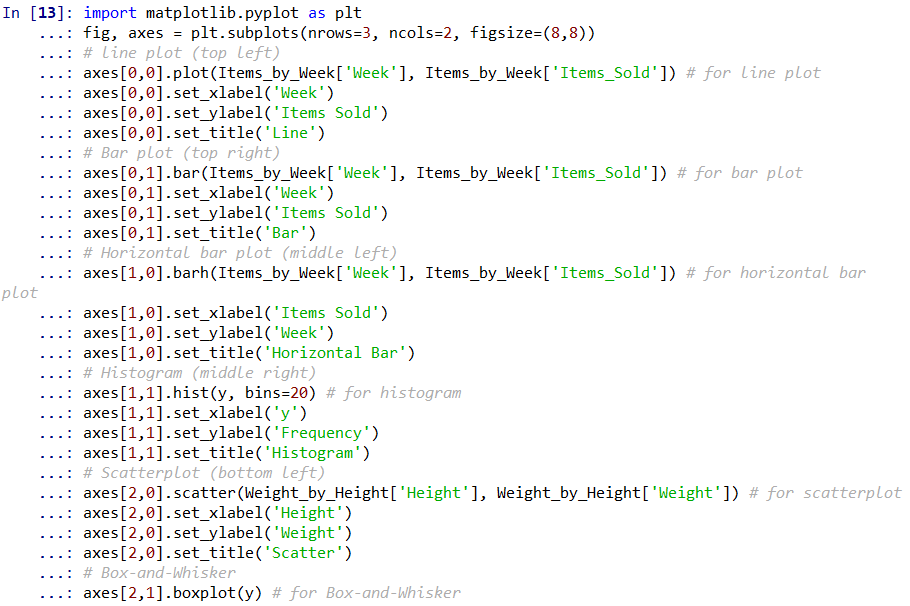
1. Check for outliers in the array y in the ‘Box-and-Whisker' axes using axes[2,1].boxplot(y).



1. Label the x- and y-axis for each subplot using axes[*row, column*].set\_xlabel(‘*X-Axis Label*’) and axes[*row, column*].set\_ylabel(‘*Y-Axis Label’*), respectively.



1. Increase the size of the figure with the figsize argument in the subplots function using fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(8,8)).



1. Save the figure to the current working directory as ‘Six\_Subplots.jpeg’ using fig.savefig('Six\_Subplots.jpeg').



***Note***

Remember, this figure with 6 subplots is now stored in the object fig.

In this chapter, we used the python plotting library matplotlib to create, customize, and save plots using the functional method. We then covered the importance of a descriptive title and created our own descriptive, programmatic titles. However, the functional method does not create a callable figure object and it does not return subplots. Thus, to create a callable figure object with the potential of numerous subplots we created, customized, and saved plots using the object-oriented method. Plotting needs can vary analysis to analysis, so covering every possible plot in this chapter is not practical. Rather, we learned to access, interpret, and leverage the matplotlib documentation and gallery to create plots which meet the need of each analysis.