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 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 are instructions for installation 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.

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.

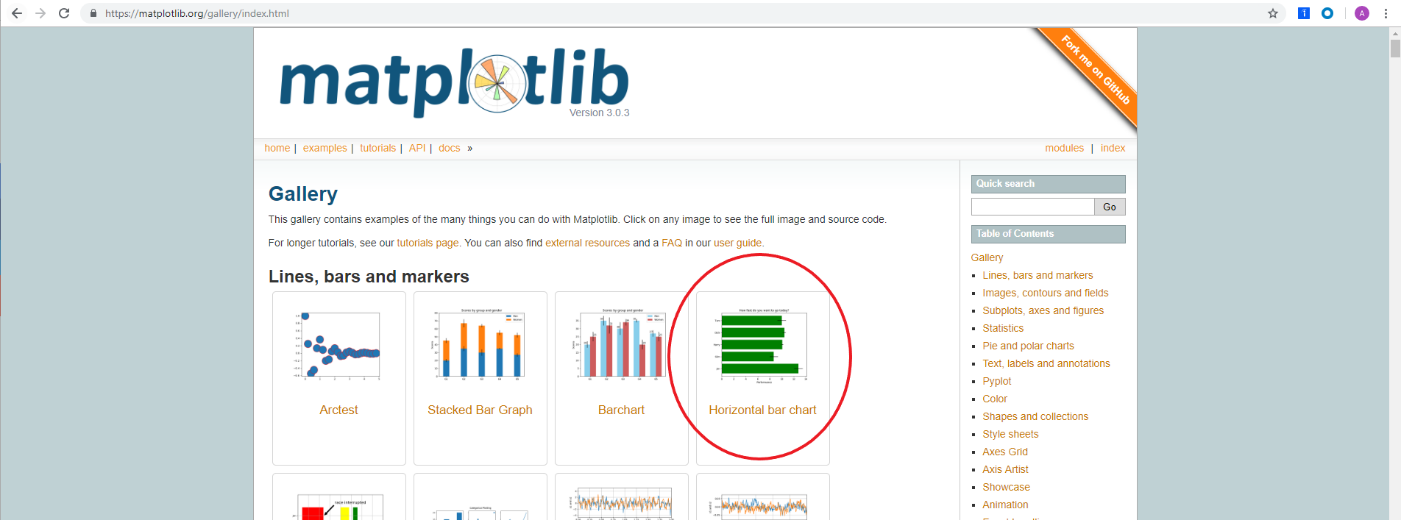


Figure 5.2. Matplotlib gallery.

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.

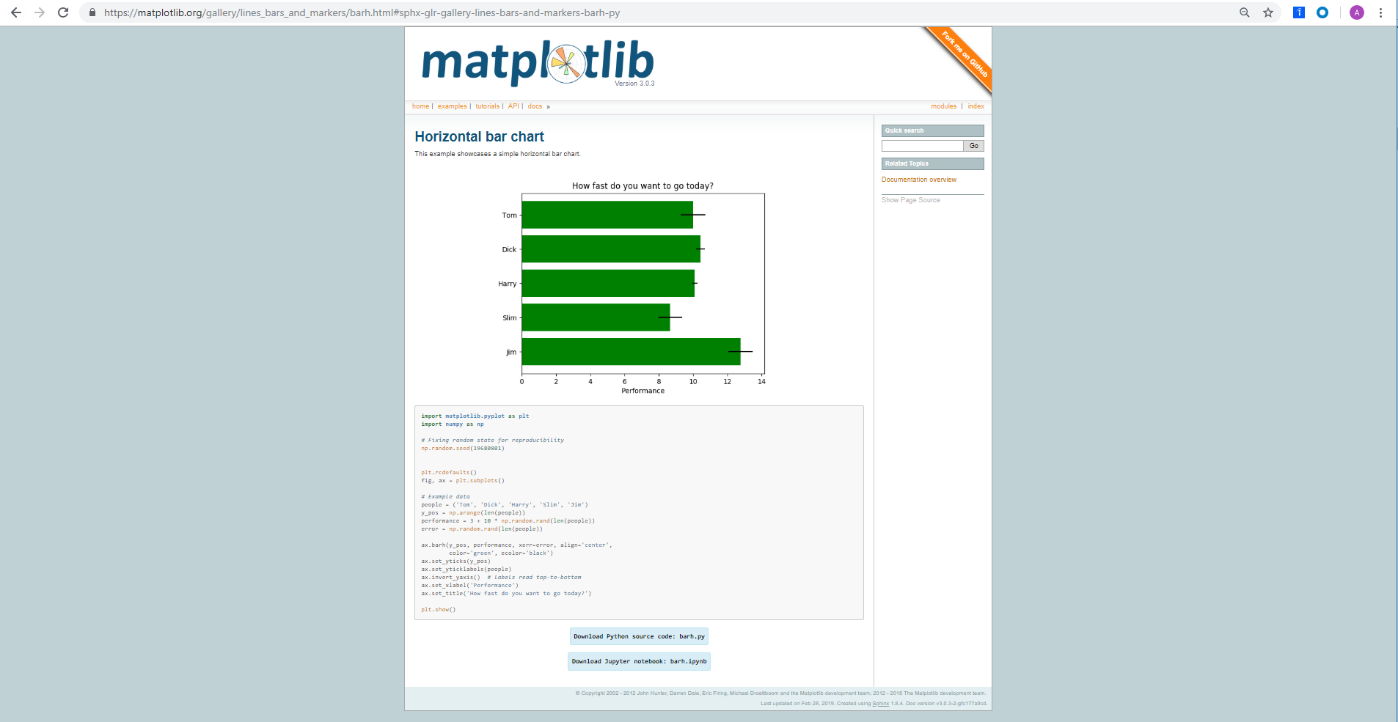


Figure 5.3. Matplotlib horizontal bar chart example.

*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

*Present x: Functional Method – Line Plot*

To learn matplotlib, we will begin by creating a line plot and progress from there.

1. Generate an array of numbers for the horizontal axis ranging 0 to 10 in 20 evenly spaced values using the following code:

import numpy as np

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.

y = x\*\*3

1. Create the plot as follows:

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.show()

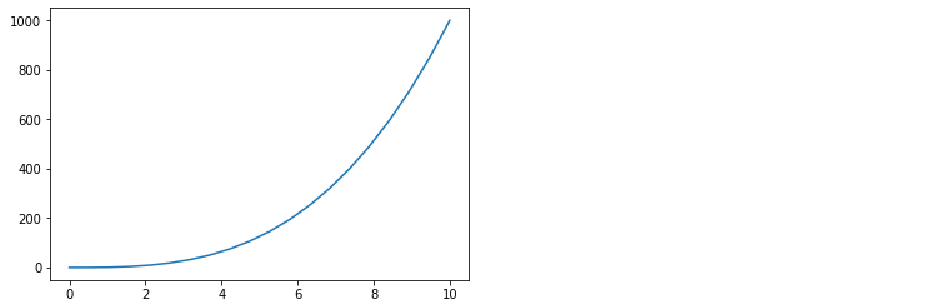


Figure 5.x: Line plot of y (x-cubed) by x

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

Add a y-axis label that reads ‘y Value’ using plt.ylabel(‘y Value’).

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

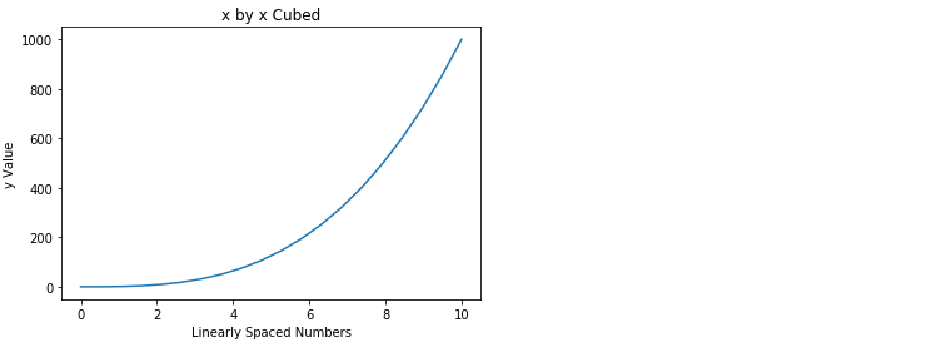


Figure 5.x: Line plot of y (x-cubed) by x with labeled axes and a title

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

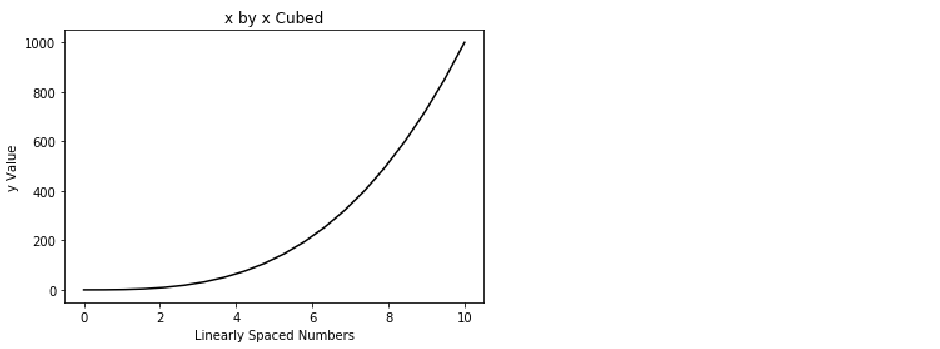


Figure 5.x: Line plot of y (x-cubed) by x with labeled axes, title, and a black line

1. To change the line characters into a diamond use a character argument (i.e., D) combined with the color character (i.e., k) as follows:

plt.plot(x, y, 'Dk')

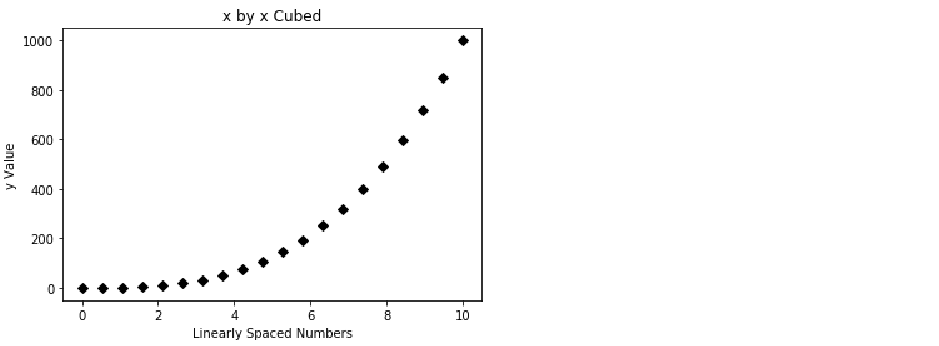


Figure 5.x: Line plot of y (x-cubed) by x with labeled axes, title, and unconnected, black diamond markers

1. Connect the diamonds with a solid line by placing ‘-’ between ‘D’ and ‘k’ using plt.plot(x, y, 'D-k').

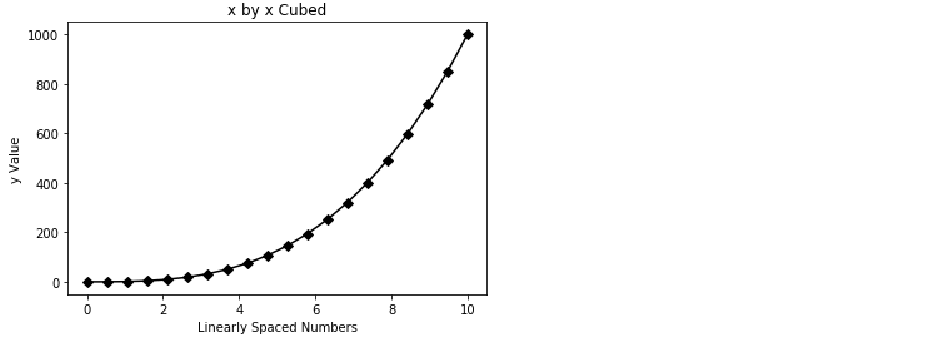


Figure 5.x: Line plot of y (x-cubed) by x with labeled axes, title, and connected, black diamond markers

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

plt.title('x by x Cubed', fontsize=22)

Print plot to the console:

plt.show()

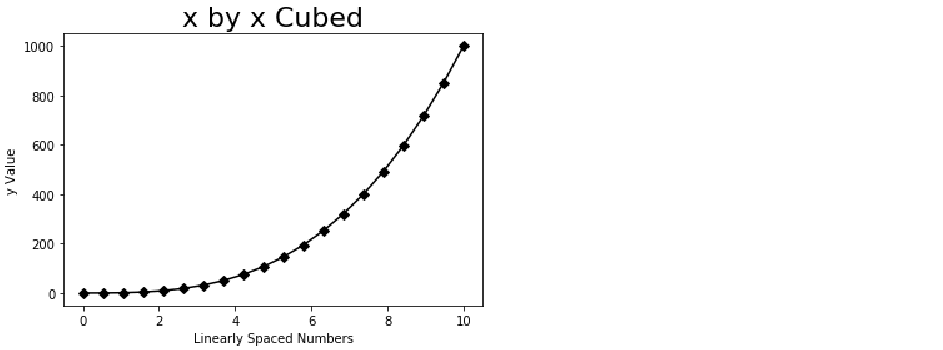


Figure 5.x: Line plot of y (x-cubed) by x with labeled axes, title, connected, black diamond markers, and a larger title

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

The previous steps detailed using the functional method to create a single-line line plot and use styling to make it more aesthetically pleasing. However, it is likely that the line plots will be comparing more than 1 trend. Thus, the next exercise will detail plotting multiple lines on a line plot and creating a legend to discern the lines.

Exercise 2: Add a Second Line to Line Plot

*Present x: Functional Method – Add a Second Line to Line Plot*

Matplotlib makes adding another line to a line plot very easy by simply specifying another plt.plot() instance. For this exercise we will be displaying x-cubed and x-squared as separate lines.

1. First, we must create another y object much as we did the first y object, but this time squaring x rather than cubing x. Refer to the code below:

y2 = x\*\*2

1. Now, plot y2 in the same plot as y by adding plt.plot(x, y2) to the existing plot.

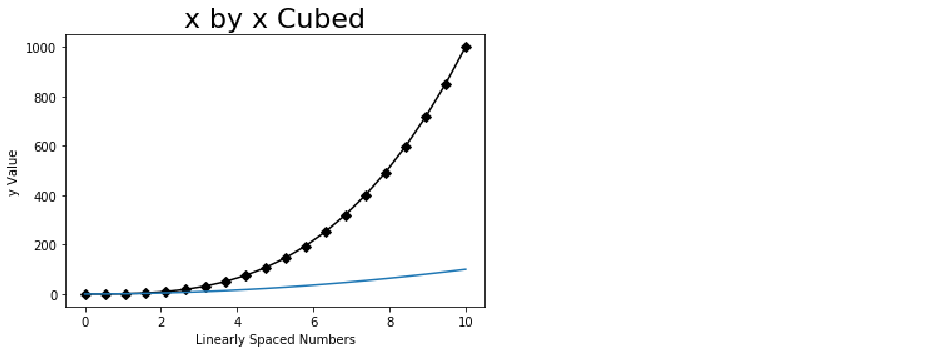


Figure 5.x: Line plot of y (x-cubed) and y2 (x-squared) by x

1. Change the color of y2 to dotted red line using the following code:

plt.plot(x, y2, ‘--r’)

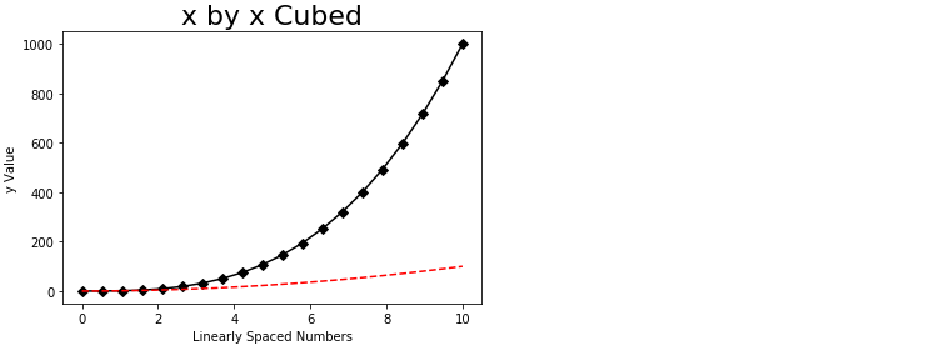


Figure 5.x: Line plot of y (x-cubed) and y2 (x-squared) by x with y2 as a red, dotted line

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').

Label y2 as ‘x squared’ using plt.plot(x, y2, '--r', label='x squared').

Use plt.legend(loc=’upper left’) to specify the location for the legend.

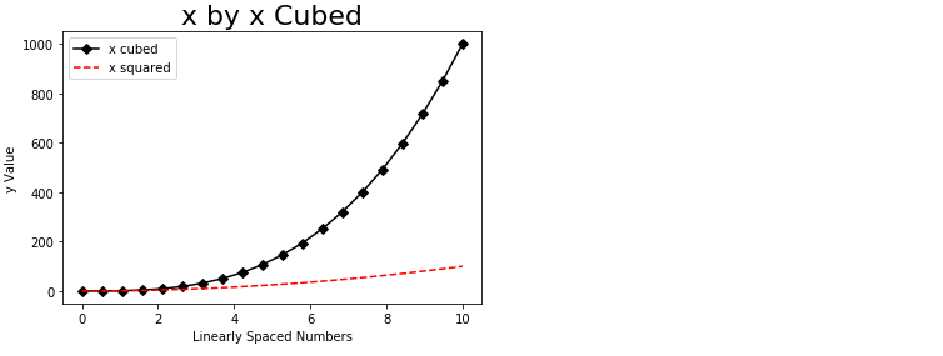


Figure 5.x: Line plot of y (x-cubed) and y2 (x-squared) by x with y2 as a red, dotted line, and a legend

1. It is helpful to the consumer when 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, by using the below code, we can create the title displayed below:

plt.title('As x increases, \nx Cubed (black) increases \nat a Greater Rate than \nx Squared (red)', fontsize=22)

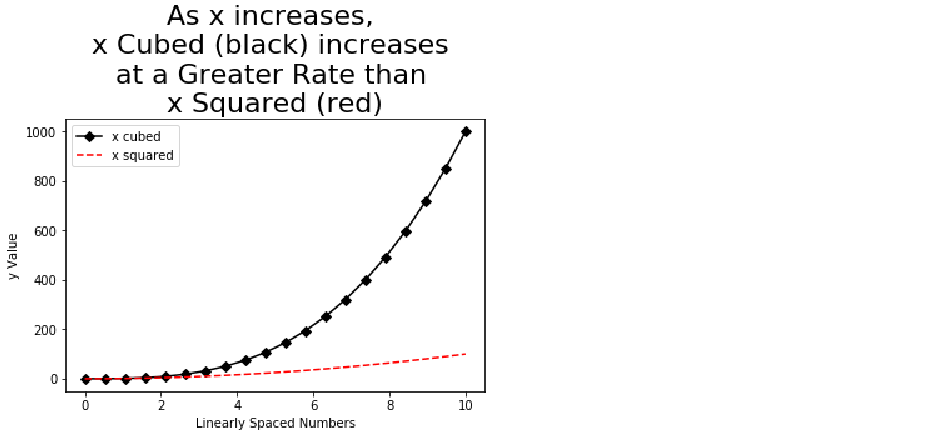


Figure 5.x: Line plot of y (x-cubed) and y2 (x-squared) by x with y2 as a red, dotted line, a legend, and a descriptive, multi-line title

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

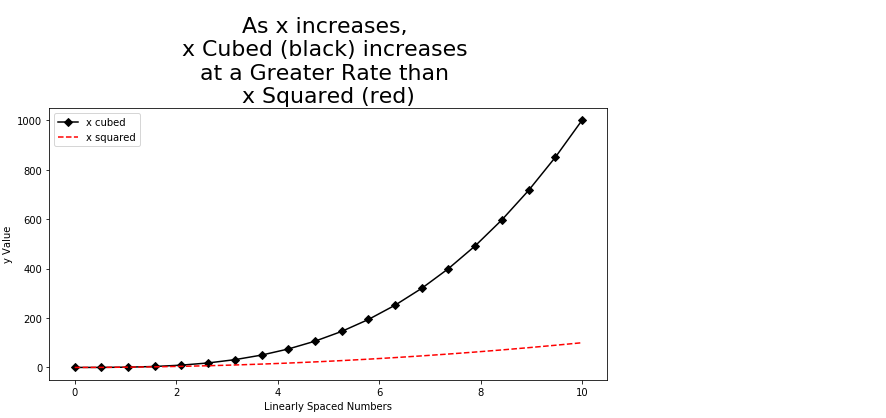


Figure 5.x: Line plot of y (x-cubed) and y2 (x-squared) by x with y2 as a red, dotted line, a legend, a descriptive, multi-line title, and increased figure size

In this exercise, we learned to create and style a single- and multi-line plot in Matplotlib using the functional method. To help solidify our learning, we will plot another single-line plot with slightly different styling.

Activity 1: Line Plot

*Present x: Functional Method – Line Plot (activity)*

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. 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. Create a list of 6 strings for x containing months January through June.
2. 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.
3. Generate the described plot.

**Solution:**

1. To create a list of 6 strings for each month January through June and save it as x using:

x = ['January','February','March','April','May','June']

1. 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 as follows:

y = [1000, 1200, 1400, 1600, 1800, 2000]

1. Plot y (‘Items Sold’) by x (‘Month’) with a dotted blue line and star markers using plt.plot(x, y, '\*:b').
2. Set the x-axis to ‘Month’ using the following code:

plt.xlabel('Month')

Set the y-axis to ‘Items Sold’ as follows:

plt.ylabel('Items Sold')

1. To set the title to read ‘Items Sold has been Increasing Linearly’ refer to the following code:

plt.title('Items Sold has been Increasing Linearly')

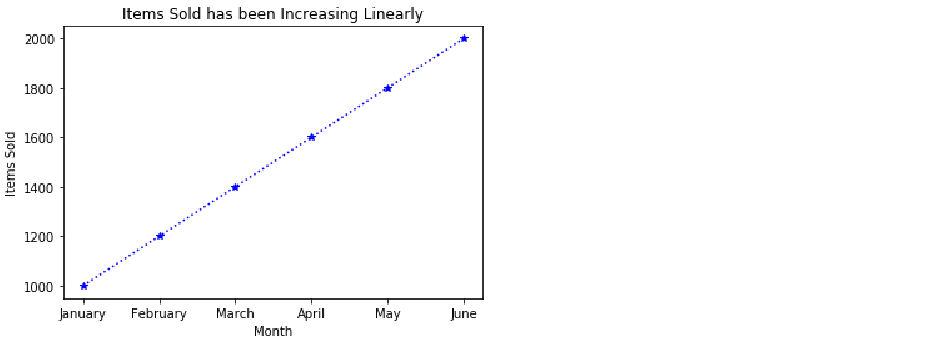


Figure 5.x: Line plot of items sold by month

To this point, we have gained a lot of practice creating and customizing line plots. Line plots are commonly used for displaying trends. However, if comparing values between and/or among groups, bar plots are traditionally the visualization of choice.

Exercise 3: Bar Plot

*Present x: Functional Method – 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. Create a list of item types and save it as x using the following code:

x = ['Shirts', 'Pants','Shorts','Shoes']

1. Create a list of Sales Revenue and save it as y as follows:

y = [1000, 1200, 800, 1800]

1. To create a bar plot and print it to the console refer to the code below:

import matplotlib.pyplot as plt

plt.bar(x, y) # plot revenue by group

plt.show()

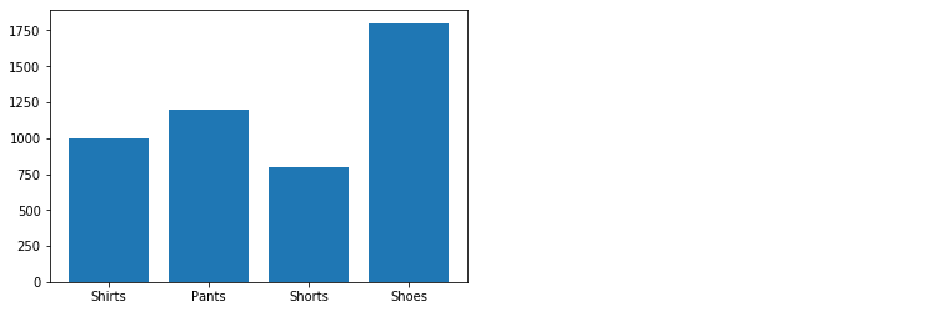


Figure 5.x: Bar plot of sales revenue by item type

1. Add a title reading ‘Sales Revenue by Item Type’ using plt.title('Sales Revenue by Item Type')

Create an x-axis label reading ‘Item Type’ using plt.xlabel('Item Type')

Add a y-axis label reading ‘Sales Revenue ($)’ using plt.ylabel('Sales Revenue ($)')

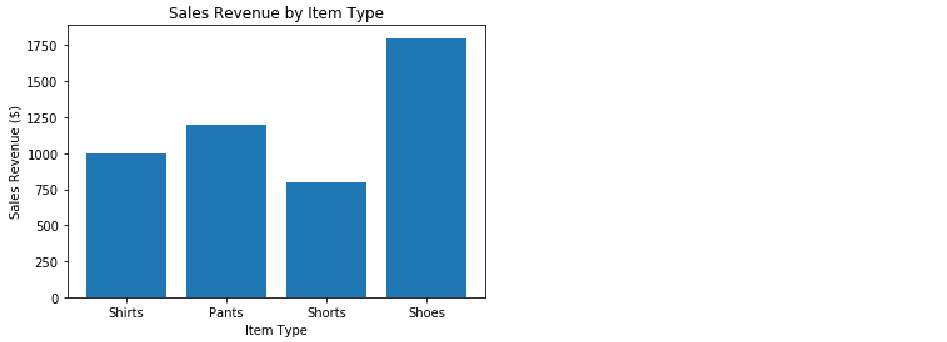


Figure 5.x: Bar plot of sales revenue by item type with customized axes and title

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 the following code:

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].
2. Make the title programmatic as follows:

plt.title('{} Produce the Most Sales Revenue'.format(most\_sold\_item))

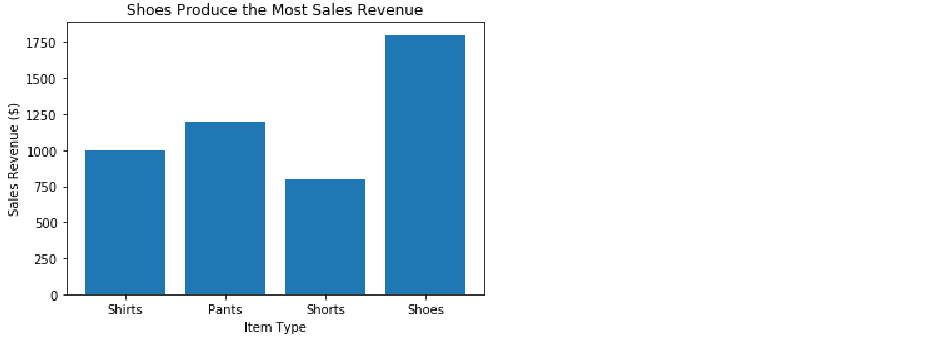


Figure 5.x: Bar plot of sales revenue by item type with customized axes and a detailed, programmatic title

1. If we wish to convert the plot into a horizontal bar plot we can do so by replacing plt.bar(x, y) with plt.barh(x, y).

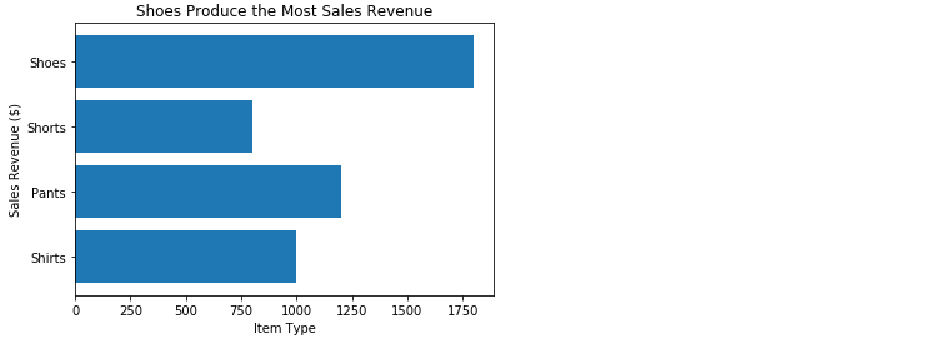


Figure 5.x: Horizontal bar plot displaying sales revenue by item type with incorrect customized axes, and a detailed, programmatic title

***Teaching Tip****: 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.

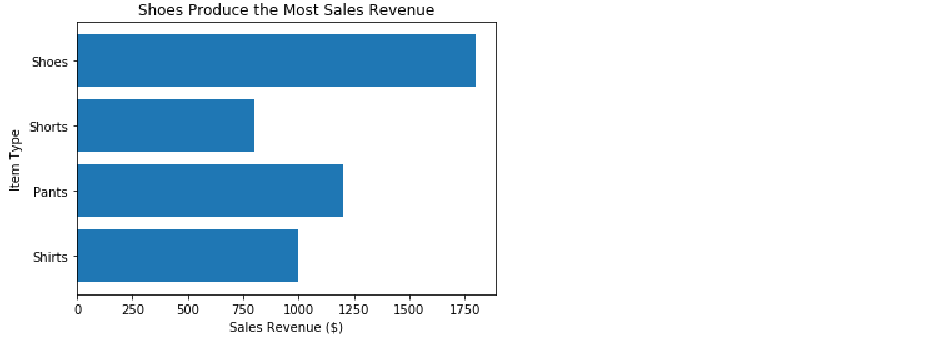


Figure 5.x: Horizontal bar plot displaying sales revenue by item type with correct customized axes, and a detailed, programmatic title

Building bar plots using Matplotlib is straightforward. In the following activity, we will continue to practice building bar plots.

Activity 2: Bar Plot

*Present x: Functional Method – Bar Plot (activity)*

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

**Solution:**

1. Create a list of 5 strings for x containing the names of NBA franchises with the most titles using the following code:

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].
2. Place x and y into a data frame with the column names ‘Team’ and ‘Titles’, respectively as follows:

import pandas as pd

df = pd.DataFrame({'Team': x,

'Titles': y})

1. To sort the data frame descending by ‘Titles’ and save it as df\_sorted refer to the code below:

df\_sorted = df.sort\_values(by=('Titles'), ascending=False)

***Teaching tip****: 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].

Lastly, 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).

1. Use a bar graph to plot the number of titles by team using the following code:

import matplotlib.pyplot as plt

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').

1. To prevent the x tick labels from overlapping by rotating them 45 degrees refer to the code below:

plt.xticks(rotation=45)

1. Set the title of the plot to the programmatic title object we created as follows:

plt.title(title)

1. Save the plot to our current working directory as ‘Titles\_by\_Team.png’ using the following code:

plt.savefig('Titles\_by\_Team)

Print the plot using plt.show().

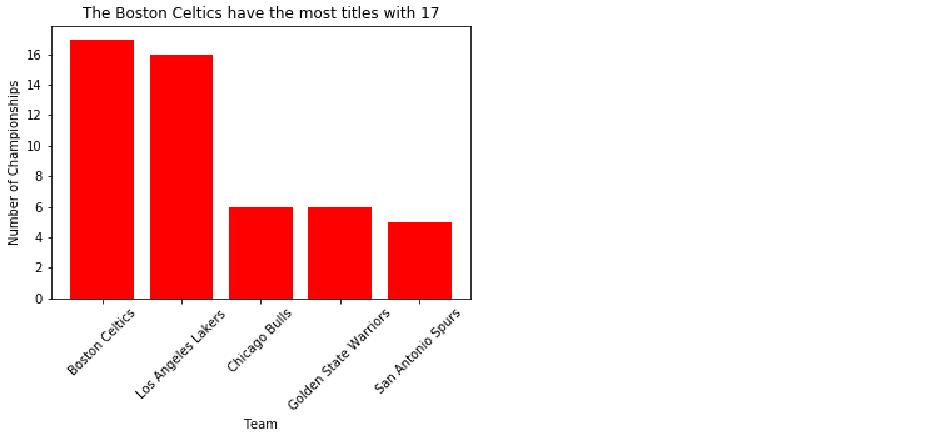


Figure 5.x: Bar plot of number of titles by NBA team

***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.png’ we see that it crops the x tick labels.

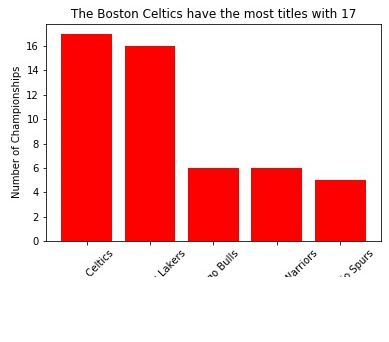


Figure 5.x: Saved bar plot of number of titles by NBA team with x tick labels cropped

1. To fix the cropping issue, add bbox\_inches=’tight’ as an argument inside of plt.savefig() as follows:

plt.savefig('Titles\_by\_Team', bbox\_inches='tight')

***Note***

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

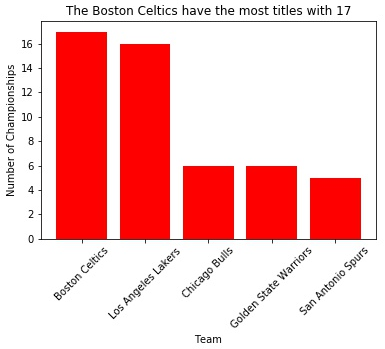


Figure 5.5. Saved figure with x tick labels fixed.

Line plots and bar plots are 2 very common and effective visualizations for reporting trends and comparing groups, respectively. However, for deeper statistical analyses, it is important to generate graphs that uncover characteristics of features not apparent by line plots and bar plots. Thus, in the following exercises, we will run-through create statistical plots.

Exercise 4: Histogram

*Present x: Functional Method – 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 for a given array.

1. To demonstrate the creation of a histogram we will generate an array of 100 normally distributed values with a mean of 0 and a standard deviation of 0.1 and save it as y using the code below:

import numpy as np

y = np.random.normal(loc=0, scale=0.1, size=100)

1. With matplotlib imported, create the histogram using plt.hist(y, bins=20).

Create a label for the x-axis titled ‘y Value’ using plt.xlabel('y Value').

Title the y-axis ‘Frequency’ using plt.ylabel(‘Frequency’).

Print it to the console using plt.show().

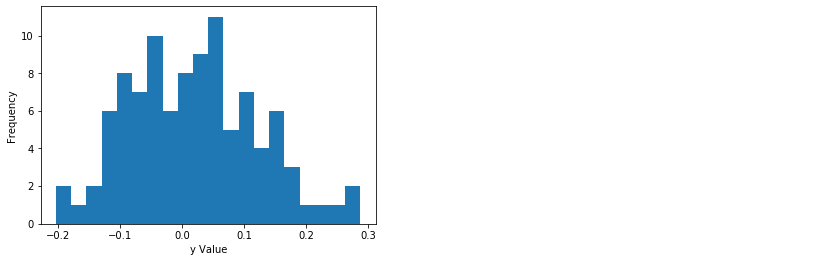


Figure 5.x: Histogram of y with labeled axes

***Note***

When we are looking at a histogram, often we are determining if the distribution is normal. Sometimes, a distribution may appear normal when it is not and sometimes a distribution may appear non-normal when it is normal. There is a 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. We will use the results from the Shapiro-Wilk test to create a programmatic title communicating to the reader if the distribution is normal or not.

1. Use tuple unpacking to save the W statistic and the p-value into the objects shap\_w and shap\_p, respectively, using the following code:

from scipy.stats import shapiro

shap\_w, shap\_p = shapiro(y)

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. Refer to the code below:

if shap\_p > 0.05:

normal\_YN = 'Fail to reject the null hypothesis. Data is normally distributed.'

else:

normal\_YN = 'Null hypothesis is rejected. Data is not normally distributed.'

1. Assign normal\_YN to our plot using plt.title(normal\_YN) and print it to the console using plt.show().

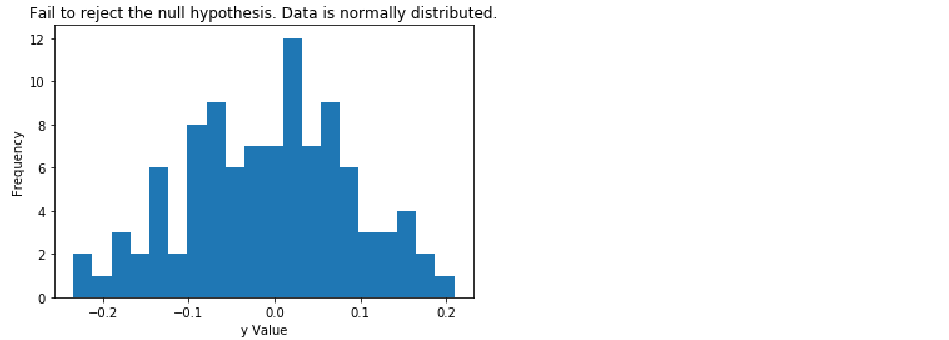


Figure 5.x: Histogram of y with labeled axes and programmatic title

As mentioned previously, histograms are used for displaying the distribution of an array. Another common statistical plot for exploring a numerical feature is the box-and-whisker plot, also referred to as a boxplot.

Exercise 5: Box-and-Whisker plot

*Present x: Functional Method – Box-and-Whisker Plot*

The last plot we will cover is the Box-and-Whisker plot. This plot can display the distribution of an array based on the minimum, first quartile, median, third quartile, and maximum. But it is primarily used for indicating the skew of a distribution and to identify outliers. In this exercise, we will portray this information in our title.

1. 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 the following code:

import numpy as np

y = np.random.normal(loc=0, scale=0.1, size=100)

1. Create and display the plot as follows:

import matplotlib.pyplot as plt

plt.boxplot(y)

plt.show()

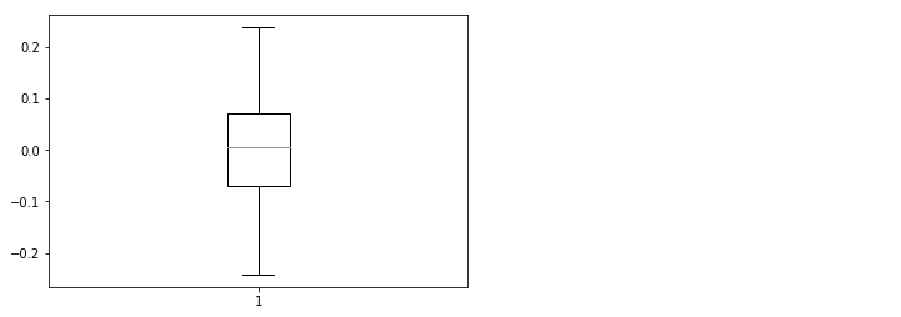


Figure 5.x: Boxplot of y

**Teaching tip:**

The plot displays a box which represents the interquartile range (IQR). The top of the box is the 25th percentile (i.e., Q1) while the bottom of the box is the 75th percentile (i.e., Q3). The orange line going through the box is the median. The two lines extending above and below the box are the whiskers. The top of the upper whisker is the “maximum” value which is calculated using Q1 – 1.5\*IQR. The bottom of the lower whisker is the “minimum” value which is calculated using Q3 + 1.5\*IQR. Outliers (or fringe outliers) are displayed as dots above the “maximum” whisker or below the “minimum” whisker.

1. Save the Shapiro W and p-value from the shapiro function as follows:

from scipy.stats import shapiro

shap\_w, shap\_p = shapiro(y)

1. Refer to the code below to convert y into z-scores:

from scipy.stats import zscore

y\_z\_scores = zscore(y)

1. Iterate through the y\_z\_scores array to find the number of outliers using the following code:

***Teaching Tip:***

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

total\_outliers = 0

for i in range(len(y\_z\_scores)):

if abs(y\_z\_scores[i]) >= 3:

total\_outliers += 1

1. Generate a title that communicates 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:

if shap\_p > 0.05:

title = 'Normally distributed with {} outlier(s).'.format(total\_outliers)

else:

title = 'Not normally distributed with {} outlier(s).'.format(total\_outliers)

1. Set our plot title as the programmatically named title using plt.title(title) and print it to the console using plt.show().

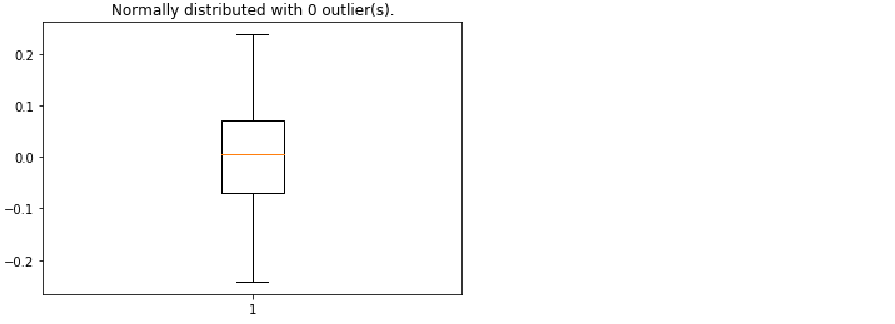


Figure 5.x: Boxplot of y with programmatic title

Histograms and Box-and-Whisker plots are effective in exploring characteristics of numerical arrays. However, they do not provide information in relationships between arrays. In the next exercise, we will learn to create a scatterplot, a common visualization to display the relationship between 2 continuous arrays.

Exercise 6: Scatterplot

*Present x: Functional Method – 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].
2. 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].
3. Create a basic scatterplot with weight on the x-axis and height on the y-axis using the following code:

import matplotlib.pyplot as plt

plt.scatter(x, y)

1. Label the x-axis ‘Weight’ as follows:

plt.xlabel('Weight')

Label the y-axis ‘Height’ as follows:

plt.ylabel('Height')

Print the plot to the console using plt.show()

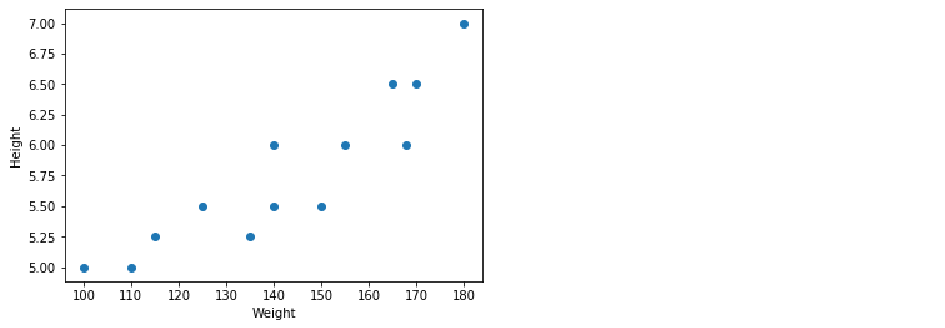


Figure 5.x: Plot of height by weight

1. 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. To compute the Pearson correlation coefficient refer to the code below:

from scipy.stats import pearsonr

correlation\_coeff, p\_value = pearsonr(x, y)

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

if correlation\_coeff == 1.00:

title = 'There is a perfect positive linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff >= 0.8:

title = 'There is a very strong, positive linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff >= 0.6:

title = 'There is a strong, positive linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff >= 0.4:

title = 'There is a moderate, positive linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff >= 0.2:

title = 'There is a weak, positive linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff > 0:

title = 'There is a very weak, positive linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff == 0:

title = 'There is no linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff <= -0.8:

title = 'There is a very strong, negative linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff <= -0.6:

title = 'There is a strong, negative linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff <= -0.4:

title = 'There is a moderate, negative linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

elif correlation\_coeff <= -0.2:

title = 'There is a weak, negative linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

else:

title = 'There is a very weak, negative linear relationship (r = {0:0.2f}).'.format(correlation\_coeff)

print(title)

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

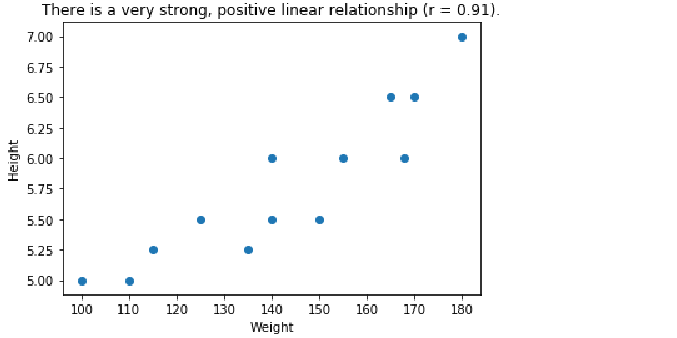


Figure 5.x: Plot of height by weight with programmatic title

Up to this point, we have learned to create and style an assortment of plots for several different purposes using the functional method. While this method of plotting is effective for generating visualizations it does not allow us to store the plot as an object in our environment. To save the plot as an object in our environment we must use the object-oriented method, which will be covered in the following exercises and activities.

Object-Oriented Method Using Subplots

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

Using the functional method of plotting in matplotlib does not allow the user to save the plot as an object in our environment. In the object-oriented method, we create a figure object which acts as an empty canvas and then we add a set of axes, or subplots, 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 7: Single Line Plot using Subplots

*Present x: Object-Oriented Method – Single Line Plot using Subplots*

When we learned about the functional method of plotting in matplotlib we began by creating and customizing a line plot. In this exercise, we will create and style a line plot using the functional plotting method.

1. Save x as an array ranging from 0 to 10 in 20 linearly spaced steps as follows:

import numpy as np

x = np.linspace(0, 10, 20)

Save y as x cubed using y = x\*\*3.

1. Create a figure and a set of axes as follows:

import matplotlib.pyplot as plt

fig, axes = plt.subplots()

plt.show()

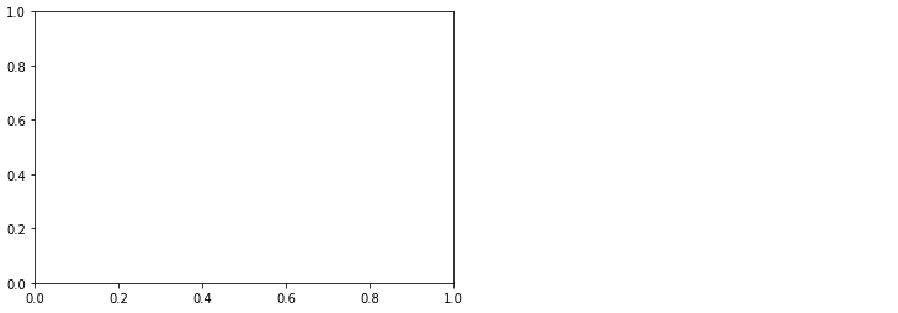


Figure 5.x: Callable figure and set of axes

***Note***

The fig object is now callable and returns the axis on which we can plot

1. Plot y (i.e., x squared) by x using axes.plot(x, y).

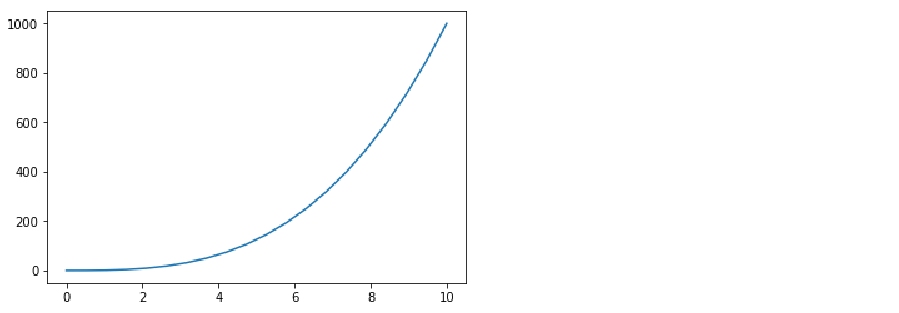


Figure 5.x: Callable line plot of y (x squared) by x

1. Style the plot much the same as in Exercise 1. First, change the line color and markers as follows:

axes.plot(x, y, 'D-k')

Set the x-axis label to ‘Linearly Spaced Numbers’ using axes.set\_xlabel('Linearly Spaced Numbers').

To set the y-axis to ‘y Value’ refer to the code below:

axes.set\_ylabel('y Value')

Set the title to ‘As x increases, y increases by x cubed’ using the following code:

axes.set\_title('As x increases, y increases by x cubed')

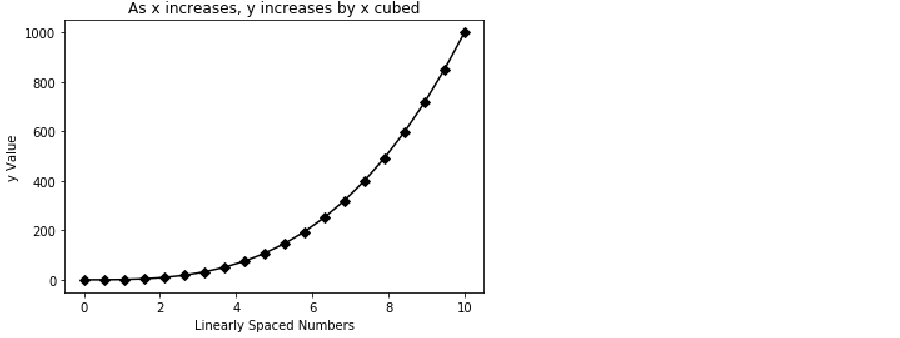


Figure 5.x: Styled, callable line plot of y (x squared) by x

In this exercise, we created a plot very similar to the first plot in Exercise 1, but now it is a callable object. Another advantage of using the object-oriented plotting method is the ability to create multiple subplots on a single figure object.

Exercise 8: Multiple Line Plots using Subplots

*Present x: Object-Oriented Method – Multiple Line Plots using Subplots*

Sometimes, we want to compare different views of data side-by-side. Thus, in this exercise, we will plot the same lines as in Exercise 2, but we will plot them on two subplots in the same, callable fig object.

1. Create x, y, and y2 using the following code:

import numpy as np

x = np.linspace(0, 10, 20)

y = x\*\*3

y2 = x\*\*2

1. Create a figure with 2 axes (I.e., subplots) that are side-by-side (I.e., 1 row with 2 columns) as follows:

import matplotlib.pyplot as plt

fig, axes = plt.subplots(nrows=1, ncols=2)

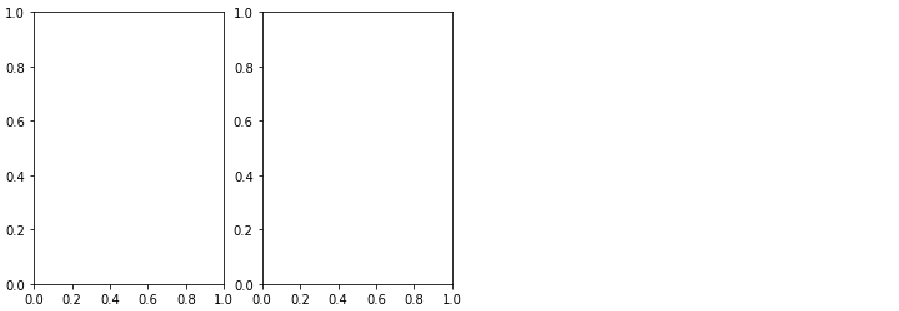


Figure 5.x: Figure object with 2 subplots

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).

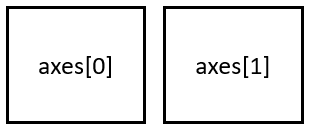


Figure 5.x: Axes index referencing in a figure object with 1 row and multiple columns.

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.x).

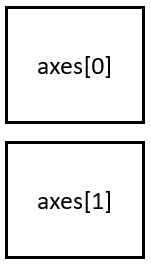


Figure 5.x: Axes index referencing in a figure object with multiple rows and 1 column.

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.

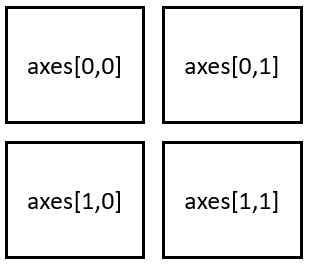


Figure 5.8. Axes index referencing in a figure object with multiple rows and multiple columns.

1. To access the subplot 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).

Add a title using axes[0].set\_title('x by x Cubed')

Generate an x-axis label using axes[0].set\_xlabel('Linearly Spaced Numbers')

Create a y-axis label using axes[0].set\_ylabel('y Value')

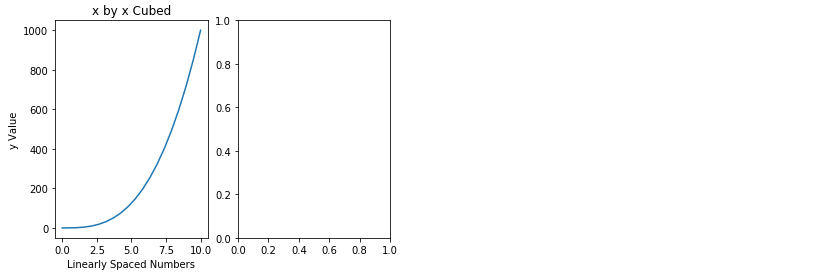


Figure 5.x: Figure object with 2 subplots where the left subplot has been created and styled

1. On the right axes, plot y2 by x using axes[1].plot(x, y2)

Add a title using axes[1].set\_title('x by x Squared')

Generate an x-axis label using axes[1].set\_xlabel('Linearly Spaced Numbers')

Create a y-axis label using axes[1].set\_ylabel('y Value')

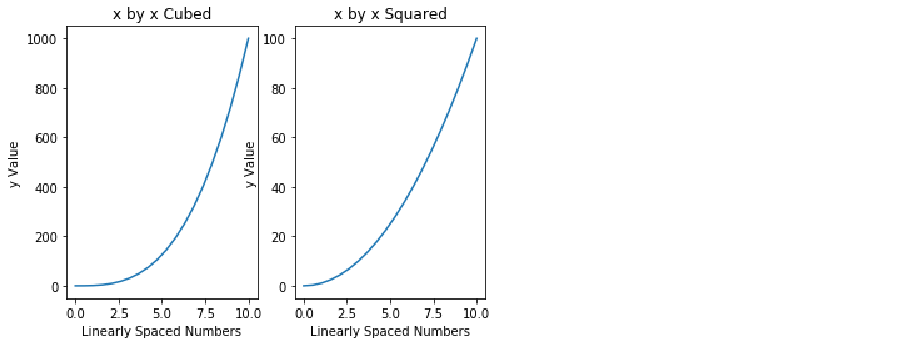


Figure 5.x: Figure object with 2 subplots where both subplots have been created and styled

1. We have successfully created 2 subplots. However, it looks like the y-axis of the plot on the right is overlapping onto the left plot. To prevent the overlapping of the plots, use plt.tight\_layout().

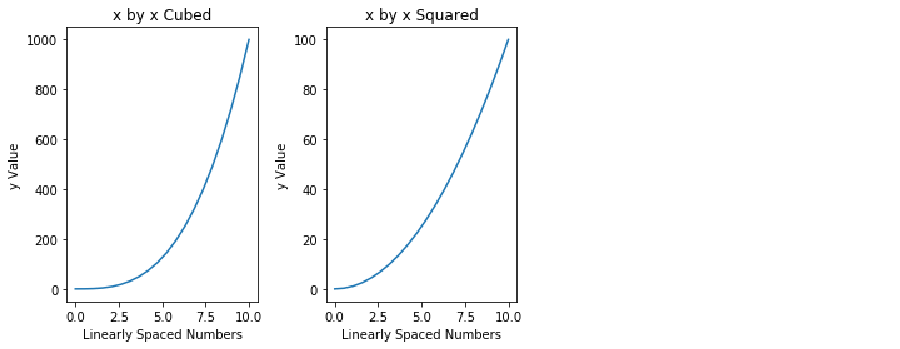


Figure 5.x: Figure object with 2 subplots where both subplots have been created and styled and the overlapping has been fixed

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

Activity 3: Multiple Plot Types using Subplots

*Present x: Object-Oriented Method – Multiple Plot Types using Subplots*

In exercises 1-6 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 7, we were introduced to the object-oriented Method and in exercise 8 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.x).

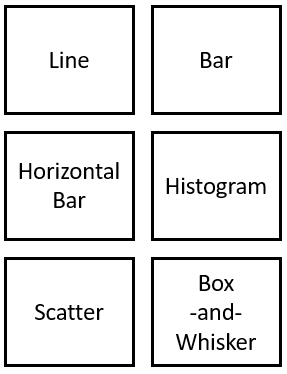


Figure 5.x: Layout for subplots.

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

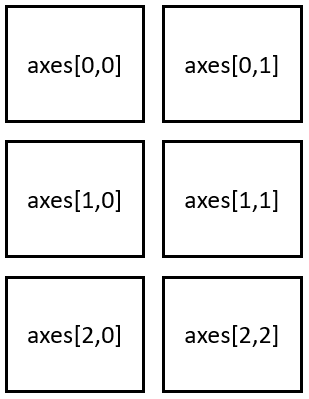


Figure 5.x: Axes index referencing in a figure object with 3 rows and 2 columns.

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 practicing this in the following activity.

1. Import the data and generate a normally distributed array of numbers
2. Generate a figure with 6 empty subplots using 3 rows and 2 columns that do not overlap.
3. Set the plot titles so they match with Figure 5.9.
4. In the ‘Line’, ‘Bar’, and ‘Horizontal Bar’ axes, plot ‘Items\_Sold’ by ‘Week’ from ‘Items\_Sold\_by\_Week.csv’.
5. In the ‘Histogram’ and ‘Box-and-Whisker’ axes, plot the array of 100 normally distributed numbers.
6. In the ‘Scatter’ axis, plot weight by height from ‘Weight\_by\_Height.csv’.
7. Label the x- and y-axis in each subplot.
8. Increase the size of the figure and save it.

**Solution:**

1. Import the ‘Items\_Sold\_by\_Week.csv’ file and save it as the data frame object Items\_by\_Week using the following code:

import pandas as pd

Items\_by\_Week = pd.read\_csv('Items\_Sold\_by\_Week.csv')

Import the Weight\_by\_Height.csv’ file and save it as the data frame object Weight\_by\_Height as follows:

Weight\_by\_Height = pd.read\_csv('Weight\_by\_Height.csv')

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. To generate a figure with 6 subplots organized in 3 rows and 2 columns that do not overlap refer to the code below:

import matplotlib.pyplot as plt

fig, axes = plt.subplots(nrows=3, ncols=2)

plt.tight\_layout()

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

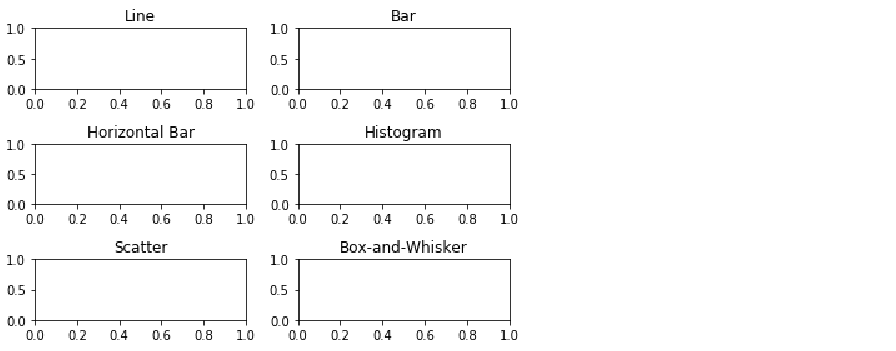


Figure 5.x: Figure object with 6 subplots where all subplots have been titled and the overlapping has been fixed

1. In the ‘Line’, ‘Bar’, and ‘Horizontal Bar’ axes, plot ‘Items\_Sold’ by ‘Week’ from ‘Items\_by\_Week’ using:
   1. axes[0,0].plot(Items\_by\_Week['Week'], Items\_by\_Week['Items\_Sold'])
   2. axes[0,1].bar(Items\_by\_Week['Week'], Items\_by\_Week['Items\_Sold'])
   3. axes[1,0].barh(Items\_by\_Week['Week'], Items\_by\_Week['Items\_Sold'])

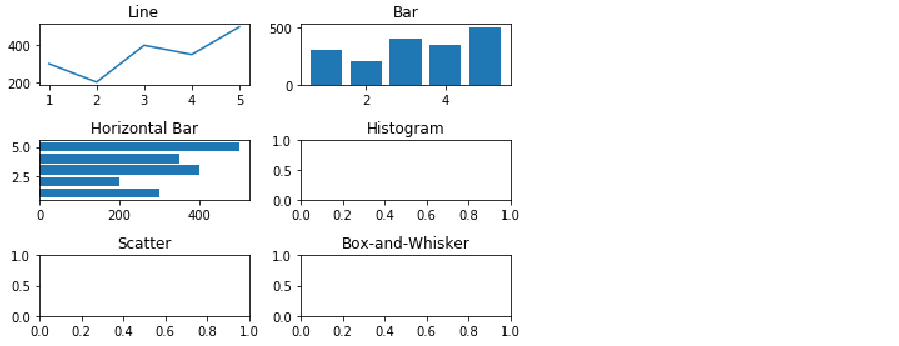


Figure 5.x: Figure object with 6 subplots where all subplots have been titled, the overlapping has been fixed, and the line, bar, and horizontal bar plots have been created.

1. In the ‘Histogram’ and ‘Box-and-Whisker’ axes, plot the array of 100 normally distributed numbers using the following code:
   1. axes[1,1].hist(y, bins=20)
   2. axes[2,1].boxplot(y)

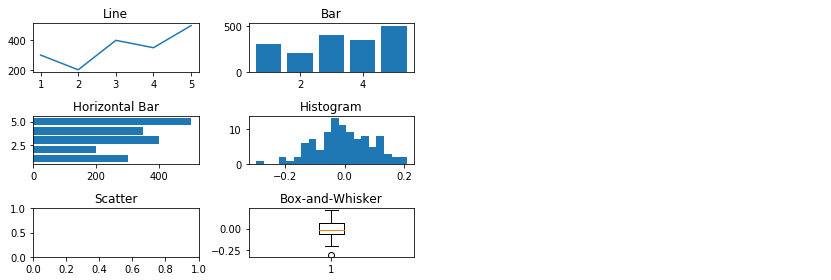


Figure 5.x: Figure object with 6 subplots where all subplots have been titled, the overlapping has been fixed, and the line, bar, horizontal bar, histogram, and box-and-whisker plots have been created.

1. Plot ‘Weight’ by ‘Height’ in the ‘Scatterplot’ axes from ‘Weight\_by\_Height’ data frame using the following code:

axes[2,0].scatter(Weight\_by\_Height['Height'], Weight\_by\_Height['Weight'])

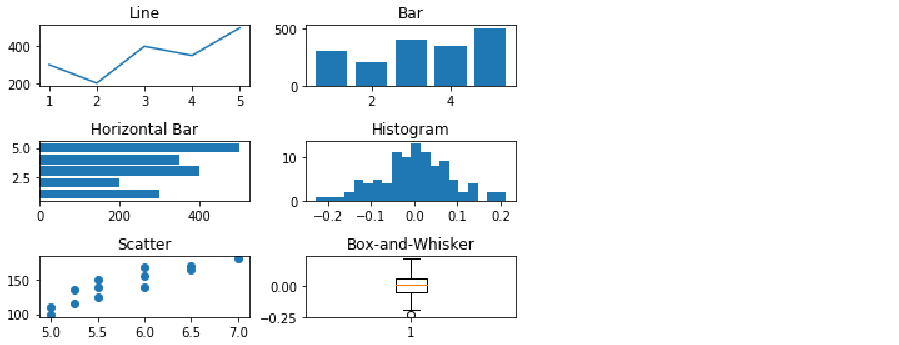


Figure 5.x: Figure object with 6 subplots where all subplots have been created and titled and the overlapping has been fixed.

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.

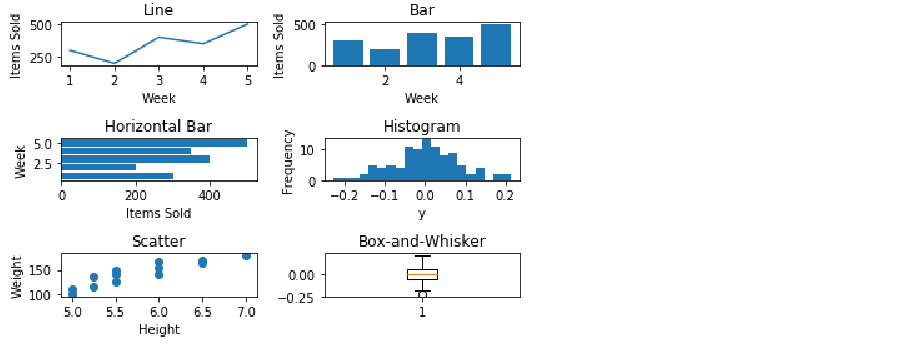
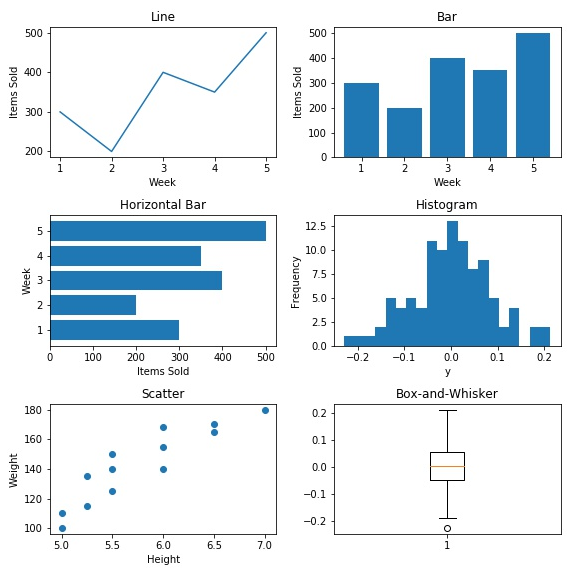


Figure 5.x: Figure object with 6 subplots where all subplots have been created and titled, the overlapping has been fixed, and the x- and y-axes have been named appropriately.

1. Increase the size of the figure with the figsize argument in the subplots function as follows:

fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(8,8))

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



Conclusion

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.

Assessment Questions:

*Present x: Assessment questions*

1. Which type of plot is commonly used to display trends over time?
   1. Bar plot
   2. Line plot
   3. Histogram
   4. Scatterplot
2. Which plot type is used to display a distribution?
   1. Bar plot
   2. Line plot
   3. Histogram
   4. Scatterplot
3. Which plot is used to compare groups?
   1. Bar plot
   2. Line plot
   3. Histogram
   4. Scatterplot
4. The 2 methods in which plots are created using matplotlib are:
   1. Functional and object-related
   2. Functional and object-oriented
   3. None of the above
   4. All of the above
5. Fill in the blanks: In the functional method, \_ figure(s) is/are created with \_ plot(s) and the figure is not a callable \_\_\_\_\_\_.
   1. 1, 1, plot
   2. 1, multiple, object
   3. 1, 1, object
   4. Multiple, 1, plot
6. True/False: In the object-oriented method, 1 figure is created with multiple subplots and the figure is a callable object.
   1. True
   2. False
7. If a figure with 4 subplots is created using 2 rows and 2 columns, how do we access the top right axes?
   1. axes[0,1]
   2. axes[1,1]
   3. axes[0,0]
   4. axes[1,0]
8. True/False: If the subplots overlap, we must lower the number of subplots because it is impossible to include code that prevents the overlap of subplots.
   1. True
   2. False
9. True/False: We are unable to save plots that were generated using the functional method because it does not return a callable object.
   1. True
   2. False
10. True/False: This chapter encompasses all possible plot types. If a plot type other than the ones covered in this lesson is needed, a different plotting library must be used.
    1. True
    2. False