3

A DEEP DIVE INTO MATPLOTLIB

OVERVIEW

This chapter describes the fundamentals of Matplotlib and teaches you how to create visualizations using the built-in plots that are provided by the library. Specifically, you will create various visualizations such as bar plots, pie charts, radar plots, histograms, and scatter plots through various exercises and activities. You will also learn basic skills such as loading, saving, plotting, and manipulating the color scale of images. You will also be able to customize your visualization plots and write mathematical expressions using TeX.

INTRODUCTION

In the previous chapter, we focused on various visualizations and identified which visualization is best suited to show certain information for a given dataset. We learned about the features, uses, and best practices for following various plots such as comparison plots, relation plots, composition plots, distribution plots, and geoplots.

Matplotlib is probably the most popular plotting library for Python. It is used for data science and machine learning visualizations all around the world. John Hunter was an American neurobiologist who began developing Matplotlib in 2003. It aimed to emulate the commands of the **MATLAB** software, which was the scientific standard back then. Several features, such as the global style of MATLAB, were introduced into Matplotlib to make the transition to Matplotlib easier for MATLAB users. This chapter teaches you how to best utilize the various functions and methods of Matplotlib to create insightful visualizations.

Before we start working with Matplotlib to create our first visualizations, we will need to understand the hierarchical structure of plots in Matplotlib. We will then cover the basic functionality, such as creating, displaying, and saving Figures. Before covering the most common visualizations, text and legend functions will be introduced. After that, layouts will be covered, which enable multiple plots to be combined into one. We will end the chapter by explaining how to plot images and how to use mathematical expressions.

OVERVIEW OF PLOTS IN MATPLOTLIB

Plots in Matplotlib have a hierarchical structure that nests Python objects to create a tree-like structure. Each plot is encapsulated in a **`Figure`** object. This **`Figure`** is the top-level container of the visualization. It can have multiple axes, which are basically individual plots inside this top-level container.

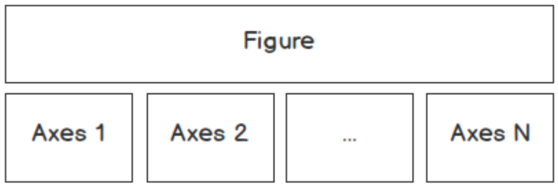

Figure 3.1: A Figure contains at least one axes object

Furthermore, we again find Python objects that control axes, tick marks, legends, titles, text boxes, the grid, and many other objects. All of these objects can be customized.

The two main components of a plot are as follows:

- **Figure**

  The Figure is an outermost container that allows you to draw multiple plots within it. It not only holds the **Axes** object but also has the ability to configure the **Title**.

- **Axes**

  The axes are an actual plot, or subplot, depending on whether you want to plot single or multiple visualizations. Its sub-objects include the x-axis, y-axis, spines, and legends.

Observing this design, we can see that this hierarchical structure allows us to create a complex and customizable visualization.

When looking at the "anatomy" of a Figure (shown in the following diagram), we get an idea about the complexity of a visualization. Matplotlib gives us the ability not only to display data, but also design the whole **Figure** around it by adjusting the **Grid**, **X** and **Y ticks**, **tick labels**, and the **Legend**.

This implies that we can modify every single bit of a plot, starting from the **Title** and **Legend**, right down to the major and minor ticks on the spines:

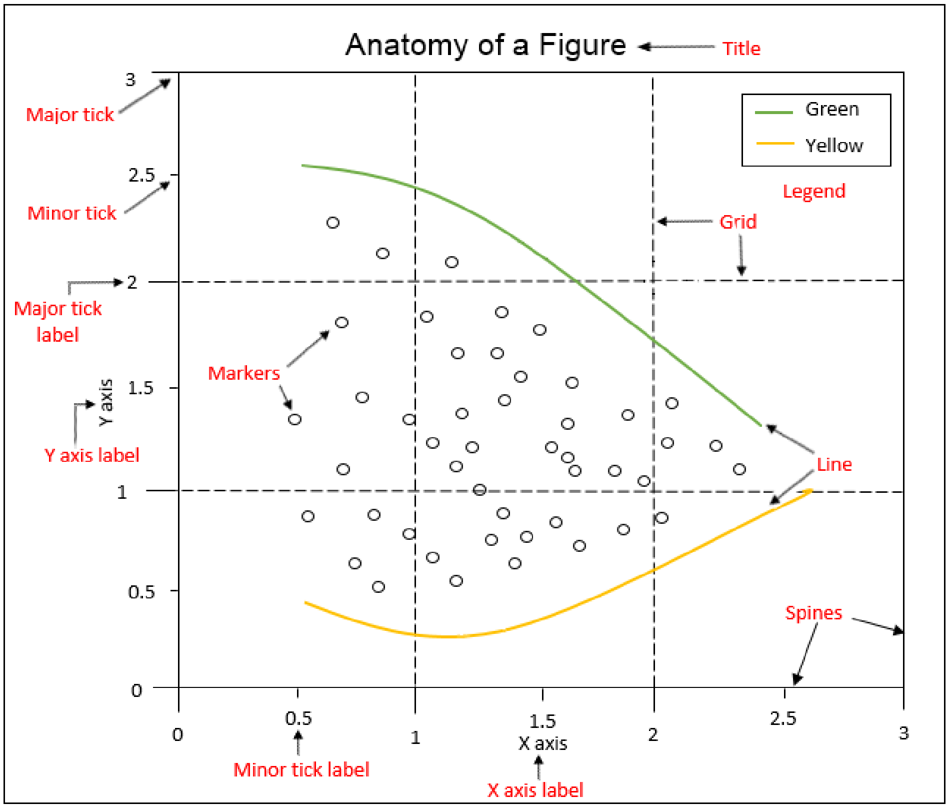

Figure 3.2: Anatomy of a Matplotlib Figure

Taking a deeper look into the anatomy of a **Figure** object, we can observe the following components:

- **Spines**: Lines connecting the axis tick marks
- **Title**: Text label of the whole Figure object
- **Legend**: Describes the content of the plot
- **Grid**: Vertical and horizontal lines used as an extension of the tick marks
- **X/Y axis label**: Text labels for the X and Y axes below the spines
- **Minor tick**: Small value indicators between the major tick marks
- **Minor tick label**: Text label that will be displayed at the minor ticks

- **Major tick**: Major value indicators on the spines
- **Major tick label**: Text label that will be displayed at the major ticks
- **Line**: Plotting type that connects data points with a line
- **Markers**: Plotting type that plots every data point with a defined marker

In this book, we will focus on Matplotlib's submodule, **pyplot**, which provides MATLAB-like plotting.

Pyplot Basics

pyplot contains a simpler interface for creating visualizations that allow the users to plot the data without explicitly configuring the **Figure** and **Axes** themselves. They are automatically configured to achieve the desired output. It is handy to use the alias **`plt`** to reference the imported submodule, as follows:

```
import matplotlib.pyplot as plt
```

The following sections describe some of the common operations that are performed when using pyplot.

Creating Figures

You can use **`plt.figure()`** to create a new **Figure**. This function returns a Figure instance, but it is also passed to the backend. Every Figure-related command that follows is applied to the current Figure and does not need to know the Figure instance.

By default, the Figure has a width of 6.4 inches and a height of 4.8 inches with a **`dpi`** (dots per inch) of 100. To change the default values of the Figure, we can use the parameters **`figsize`** and **`dpi`**.

The following code snippet shows how we can manipulate a Figure:

```
#To change the width and the height
plt.figure(figsize=(10, 5))

#To change the dpi
plt.figure(dpi=300)
```

Even though it is not necessary to explicitly create a Figure, this is a good practice if you want to create multiple Figures at the same time.

Closing Figures

Figures that are no longer used should be closed by explicitly calling **`plt.close()`**, which also cleans up memory efficiently.

If nothing is specified, the **`plt.close()`** command will close the current Figure. To close a specific Figure, you can either provide a reference to a Figure instance or provide the Figure number. To find the **number** of a Figure object, we can make use of the **`number`** attribute, as follows:

```
plt.gcf().number
```

The **`plt.close('all')`** command is used to close all active Figures. The following example shows how a Figure can be created and closed:

```
#Create Figure with Figure number 10
plt.figure(num=10)

#Close Figure with Figure number 10
plt.close(10)
```

For a small Python script that only creates a visualization, explicitly closing a Figure isn't required, since the memory will be cleaned in any case once the program terminates. But if you create lots of Figures, it might make sense to close Figures in between so as to save memory.

Format Strings

Before we actually plot something, let's quickly discuss **format strings**. They are a neat way to specify **colors**, **marker types**, and **line styles**. A format string is specified as **`[color][marker][line]`**, where each item is optional. If the **`color`** argument is the only argument of the format string, you can use **`matplotlib.colors`**. Matplotlib recognizes the following formats, among others:

- RGB or RGBA float tuples (for example, (0.2, 0.4, 0.3) or (0.2, 0.4, 0.3, 0.5))
- RGB or RGBA hex strings (for example, '#0F0F0F' or '#0F0F0F0F')

The following table is an example of how a color can be represented in one particular format:

Colors	Color
'b'	blue
'r'	red
'g'	green
'm'	magenta
'c'	cyan
'k'	black
'w'	white
'y'	yellow

Figure 3.3: Color specified in string format

All the available marker options are illustrated in the following figure:

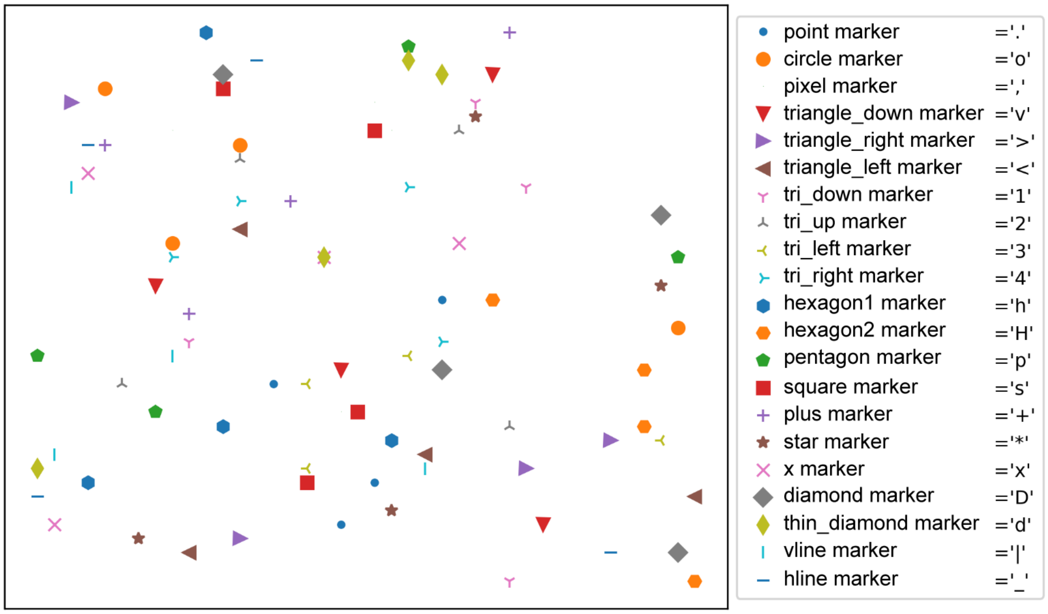

Figure 3.4: Markers in format strings

All the available line styles are illustrated in the following diagram. In general, solid lines should be used. We recommend restricting the use of dashed and dotted lines to either visualize some bounds/targets/goals or to depict uncertainty, for example, in a forecast:

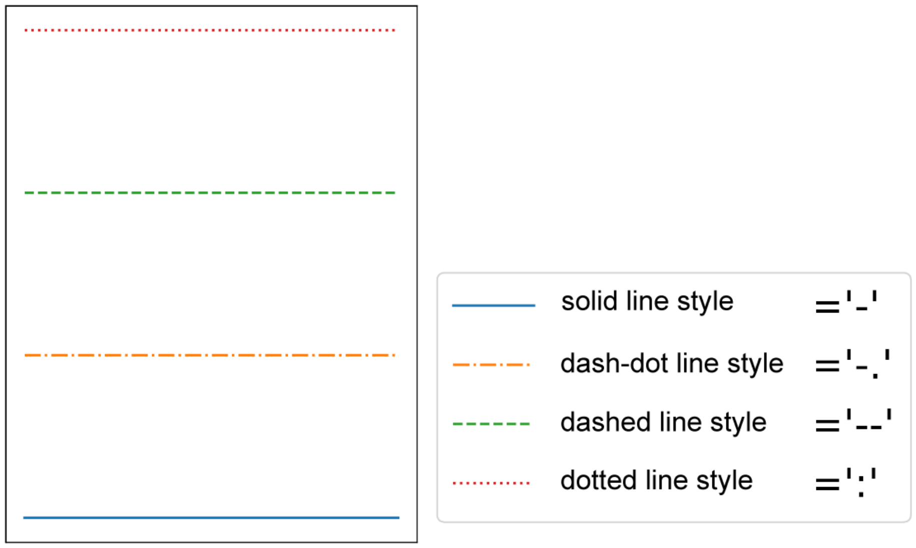

Figure 3.5: Line styles

To conclude, format strings are a handy way to quickly customize colors, marker types, and line styles. It is also possible to use arguments, such as **`color`**, **`marker`**, and **`linestyle`**.

PLOTTING

With **`plt.plot([x], y, [fmt])`**, you can plot data points as lines and/or markers. The function returns a list of **`Line2D`** objects representing the plotted data. By default, if you do not provide a format string (**`fmt`**), the data points will be connected with straight, solid lines. **`plt.plot([0, 1, 2, 3], [2, 4, 6, 8])`** produces a plot, as shown in the following diagram. Since **`x`** is optional and the default values are **`[0, …, N-1]`**, **`plt.plot([2, 4, 6, 8])`** results in the same plot:

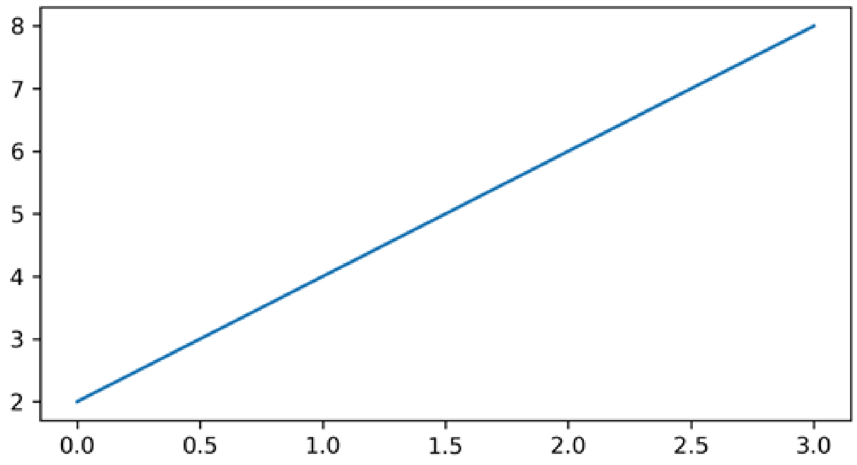

Figure 3.6: Plotting data points as a line

If you want to plot markers instead of lines, you can just specify a format string with any marker type. For example, **`plt.plot([0, 1, 2, 3], [2, 4, 6, 8], 'o')`** displays data points as circles, as shown in the following diagram:

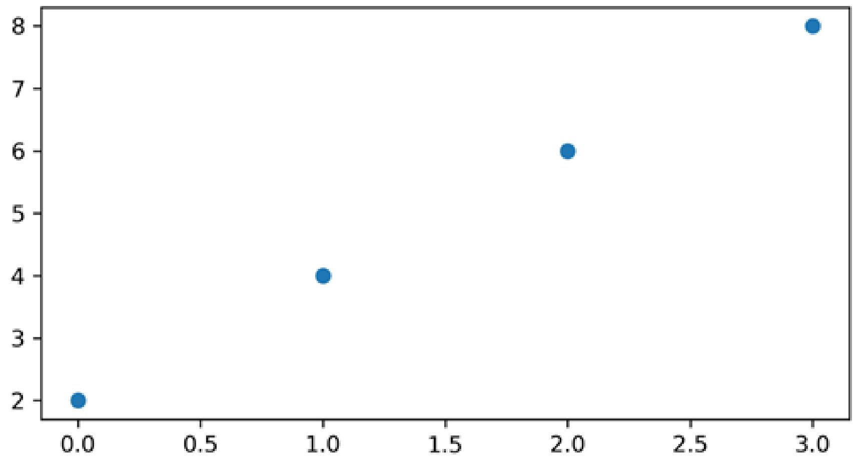

Figure 3.7: Plotting data points with markers (circles)

To plot multiple data pairs, the syntax **`plt.plot([x], y, [fmt], [x], y2, [fmt2], …)`** can be used. **`plt.plot([2, 4, 6, 8], 'o', [1, 5, 9, 13], 's')`** results in the following diagram. Similarly, you can use **`plt.plot`** multiple times, since we are working on the same Figure and Axes:

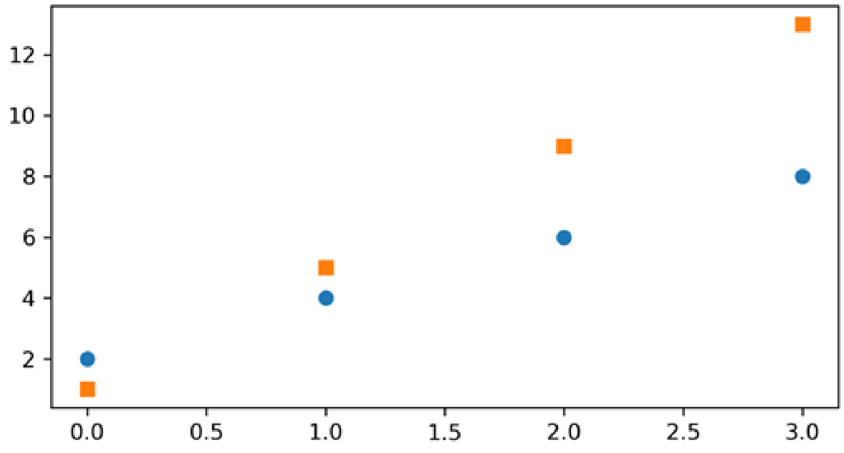

Figure 3.8: Plotting data points with multiple markers

Any **`Line2D`** properties can be used instead of format strings to further customize the plot. For example, the following code snippet shows how we can additionally specify the **`linewidth`** and **`markersize`** arguments:

```
plt.plot([2, 4, 6, 8], color='blue', marker='o', \
         linestyle='dashed', linewidth=2, markersize=12)
```

Besides providing data using lists or NumPy arrays, it might be handy to use pandas DataFrames, as explained in the next section.

PLOTTING USING PANDAS DATAFRAMES

It is pretty straightforward to use **`pandas.DataFrame`** as a data source. Instead of providing **`x`** and **`y`** values, you can provide the **`pandas.DataFrame`** in the data parameter and give keys for **`x`** and **`y`**, as follows:

```
plt.plot('x_key', 'y_key', data=df)
```

If your data is already a pandas DataFrame, this is the preferred way.

TICKS

Tick locations and labels can be set manually if Matplotlib's default isn't sufficient. Considering the previous plot, it might be preferable to only have ticks at multiples of ones at the x-axis. One way to accomplish this is to use **`plt.xticks()`** and **`plt.yticks()`** to either get or set the ticks manually.

`plt.xticks(ticks, [labels], [**kwargs])` sets the current tick locations and labels of the x-axis.

Parameters:

- **`ticks`**: List of tick locations; if an empty list is passed, ticks will be disabled.
- **`labels`** (optional): You can optionally pass a list of labels for the specified locations.
- **`**kwargs`** (optional): **`matplotlib.text.Text()`** properties can be used to customize the appearance of the tick labels. A quite useful property is **`rotation`**; this allows you to rotate the tick labels to use space more efficiently.

Example:

Consider the following code to plot a graph with custom ticks:

```
import numpy as np
plt.figure(figsize=(6, 3))
plt.plot([2, 4, 6, 8], 'o', [1, 5, 9, 13], 's')
plt.xticks(ticks=np.arange(4))
```

This will result in the following plot:

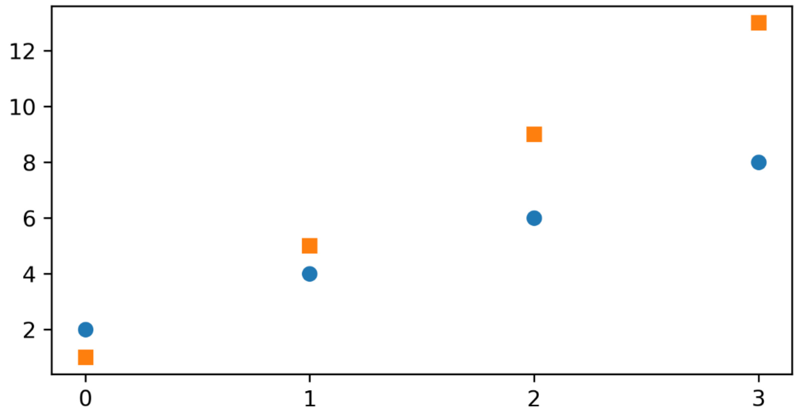

Figure 3.9: Plot with custom ticks

It's also possible to specify tick labels, as follows:

```
plt.figure(figsize=(6, 3))
plt.plot([2, 4, 6, 8], 'o', [1, 5, 9, 13], 's')
plt.xticks(ticks=np.arange(4), \
           labels=['January', 'February', 'March', 'April'], \
           rotation=20)
```

This will result in the following plot:

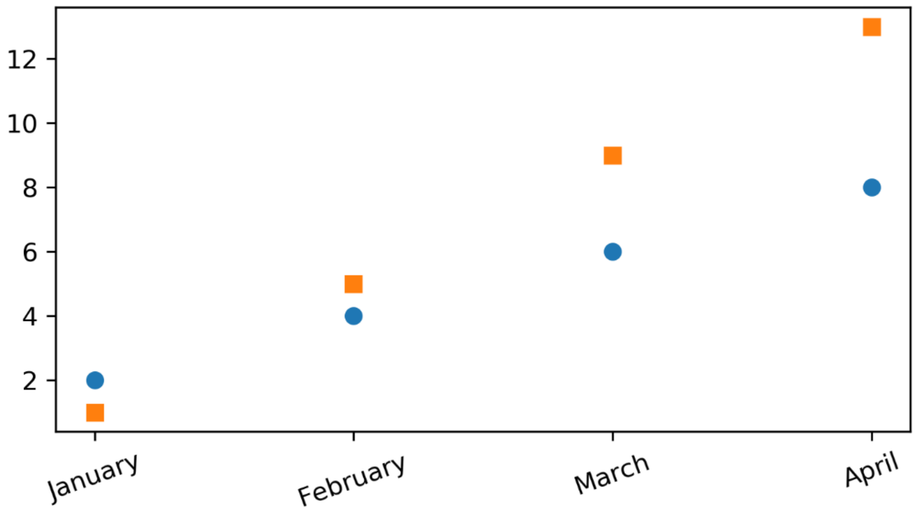

Figure 3.10: Plot with custom tick labels

If you want to do even more sophisticated things with ticks, you should look into tick locators and formatters. For example, **`ax.xaxis.set_major_locator(plt.NullLocator())`** would remove the major ticks of the x-axis, and **`ax.xaxis.set_major_formatter(plt.NullFormatter())`** would remove the major tick labels, but not the tick locations of the x-axis.

DISPLAYING FIGURES

`plt.show()` is used to display a Figure or multiple Figures. To display Figures within a Jupyter Notebook, simply set the **`%matplotlib inline`** command at the beginning of the code.

If you forget to use **`plt.show()`**, the plot won't show up. We will learn how to save the Figure in the next section.

Saving Figures

The **`plt.savefig(fname)`** saves the current Figure. There are some useful optional parameters you can specify, such as **`dpi`**, **`format`**, or **`transparent`**. The following code snippet gives an example of how you can save a Figure:

```
plt.figure()
plt.plot([1, 2, 4, 5], [1, 3, 4, 3], '-o')
#bbox_inches='tight' removes the outer white margins
plt.savefig('lineplot.png', dpi=300, bbox_inches='tight')
```

The following is the output of the code:

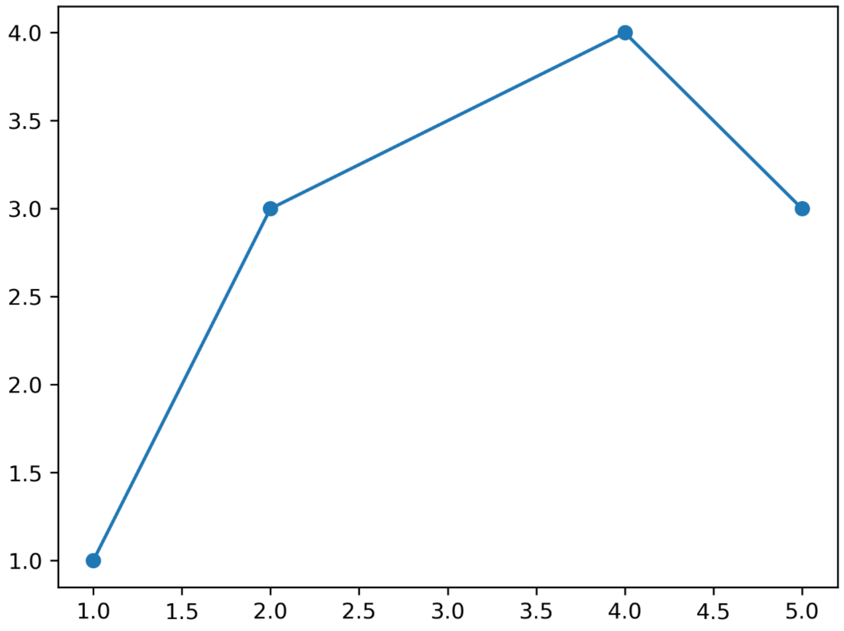

Figure 3.11: Saved Figure

Note

All exercises and activities will be developed in Jupyter Notebook. Please download the GitHub repository with all the prepared templates from https://packt.live/2HkTW1m. The datasets used in this chapter can be downloaded from https://packt.live/3bzApYN.

Let's create a simple visualization in our next exercise.

EXERCISE 3.01: CREATING A SIMPLE VISUALIZATION

In this exercise, we will create our first simple plot using Matplotlib. The purpose of this exercise is for you to create your first simple line plot using Matplotlib, including the customization of the plot with format strings.

1. Create a new **`Exercise3.01.ipynb`** Jupyter Notebook in the **`Chapter03/Exercise3.01`** folder to implement this exercise.
2. Import the necessary modules and enable plotting within the Jupyter Notebook:

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

%matplotlib inline
```

3. Explicitly create a Figure and set the **`dpi`** to 200:

```
plt.figure(dpi=200)
```

4. Plot the following data pairs **`(x, y)`** as circles, which are connected via line segments: **`(1, 1)`**, **`(2, 3)`**, **`(4, 4)`**, and **`(5, 3)`**. Then, visualize the plot:

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

   Your output should look similar to this:

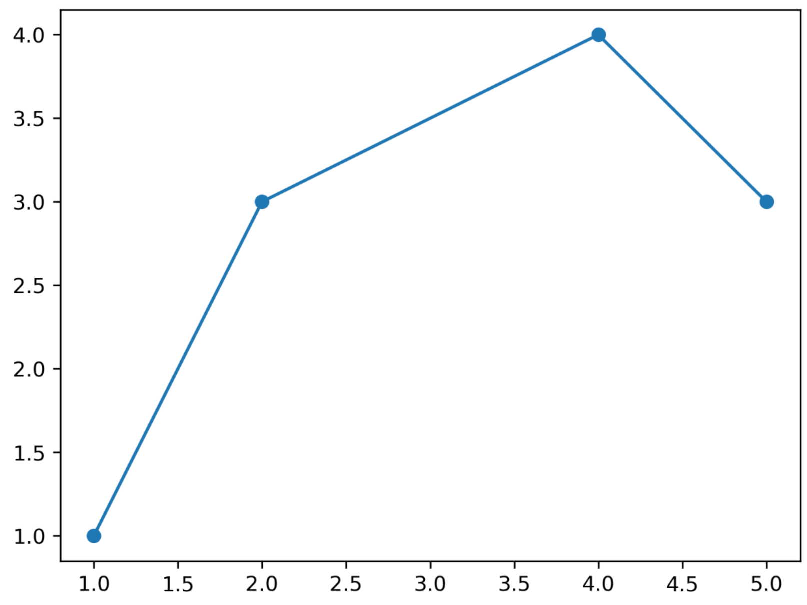

Figure 3.12: A simple visualization created with the help of given data pairs and connected via line segments

5. Save the plot using the **`plt.savefig()`** method. Here, we can either provide a filename within the method or specify the full path:

```
plt.savefig('Exercise3.01.png', bbox_inches='tight')
```

NOTE

To access the source code for this specific section, please refer to https://packt.live/2URkzlE.

You can also run this example online at https://packt.live/2YI3A6t.

This exercise showed you how to create a line plot in Matplotlib and how to use format strings to quickly customize the appearance of the specified data points. Don't forget to use **`bbox_inches='tight'`** to remove the outer white margins. In the following section, we will cover how to further customize plots by adding text and a legend.

BASIC TEXT AND LEGEND FUNCTIONS

All of the functions we discuss in this topic, except for the legend, create and return a **`matplotlib.text.Text()`** instance. We are mentioning it here so that you know that all of the properties discussed can be used for the other functions as well. All text functions are illustrated in *Figure 3.13*.

LABELS

Matplotlib provides a few **label** functions that we can use for setting labels to the x- and y-axes. The **`plt.xlabel()`** and **`plt.ylabel()`** functions are used to set the label for the current axes. The **`set_xlabel()`** and **`set_ylabel()`** functions are used to set the label for specified axes.

Example:

```
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
```

You should (always) add labels to make a visualization more self-explanatory. The same is valid for titles, which will be discussed now.

TITLES

A **title** describes a particular chart/graph. The titles are placed above the axes in the center, left edge, or right edge. There are two options for titles – you can either set the **Figure title** or the title of an **Axes**. The **`suptitle()`** function sets the title for the current and specified Figure. The **`title()`** function helps in setting the title for the current and specified axes.

Example:

```
fig = plt.figure()
fig.suptitle('Suptitle', fontsize=10, fontweight='bold')
```

This creates a bold Figure title with a text subtitle and a font size of 10:

```
plt.title('Title', fontsize=16)
```

The **`plt.title`** function will add a title to the Figure with text as **`Title`** and font size of **`16`** in this case.

TEXT

There are two options for **text** – you can either add text to a Figure or text to an Axes. The **`figtext(x, y, text)`** and **`text(x, y, text)`** functions add text at locations **`x`** or **`y`** for a Figure.

Example:

```
ax.text(4, 6, 'Text in Data Coords', \
        bbox={'facecolor': 'yellow', 'alpha':0.5, 'pad':10})
```

This creates a yellow text box with the text **Text in Data Coords**.

Text can be used to provide additional textual information to a visualization. To annotate something, Matplotlib offers annotations.

ANNOTATIONS

Compared to text that is placed at an arbitrary position on the Axes, **annotations** are used to annotate some features of the plot. In annotations, there are two locations to consider: the annotated location, **`xy`**, and the location of the annotation, text **`xytext`**. It is useful to specify the parameter **`arrowprops`**, which results in an arrow pointing to the annotated location.

Example:

```
ax.annotate('Example of Annotate', xy=(4,2), \
            xytext=(8,4), \
            arrowprops=dict(facecolor='green', shrink=0.05))
```

This creates a green arrow pointing to the data coordinates (4, 2) with the text **Example of Annotate** at data coordinates (8, 4):

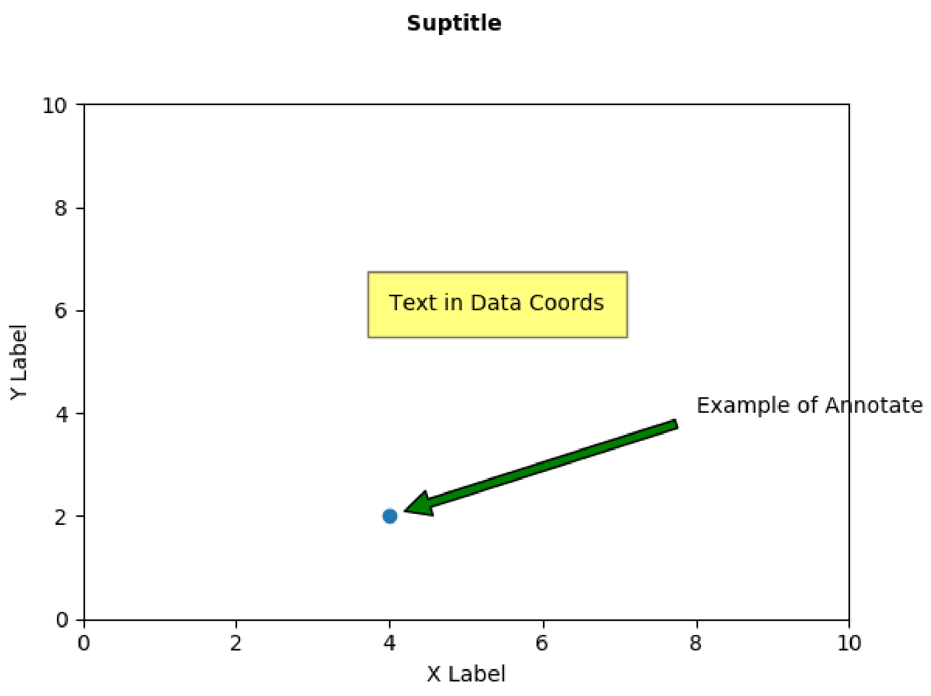

Figure 3.13: Implementation of text commands

LEGENDS

Legend describes the content of the plot. To add a **legend** to your Axes, we have to specify the **`label`** parameter at the time of plot creation. Calling **`plt.legend()`** for the current Axes or **`Axes.legend()`** for a specific Axes will add the legend. The **`loc`** parameter specifies the location of the legend.

Example:

```
plt.plot([1, 2, 3], label='Label 1')
plt.plot([2, 4, 3], label='Label 2')
plt.legend()
```

This example is illustrated in the following diagram:

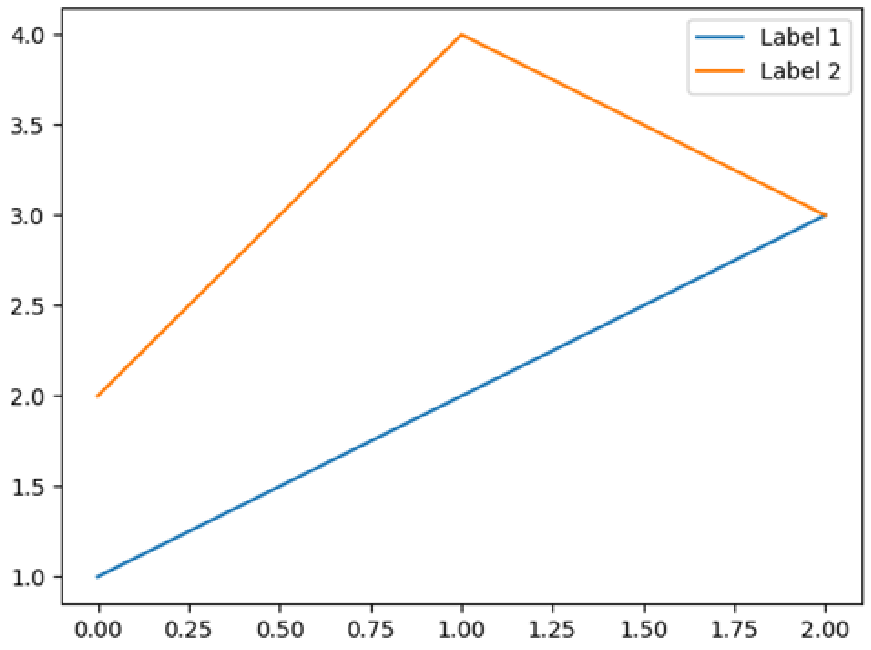

Figure 3.14: Legend example

Labels, titles, text, annotations, and a legend are great ways to add textual information to visualization and therefore make it more understandable and self-explanatory. But don't overdo it. Too much text can be overwhelming. The following activity gives you the opportunity to consolidate the theoretical foundations learned in this section.

ACTIVITY 3.01: VISUALIZING STOCK TRENDS BY USING A LINE PLOT

In this activity, we will create a line plot to show stock trends. The aim of this activity is to not just visualize the data but to use labels, a title, and a legend to make the visualization self-explanatory and "complete."

Let's look at the following scenario: you are interested in investing in stocks. You downloaded the stock prices for the "big five": Amazon, Google, Apple, Facebook, and Microsoft. You want to visualize the closing prices in dollars to identify trends. This dataset is available in the `Datasets` folder that you had downloaded initially. The following are the steps to perform:

1. Import the necessary modules and enable plotting within a Jupyter Notebook.
2. Use pandas to read the datasets (**`GOOGL_data.csv`**, **`FB_data.csv`**, **`AAPL_data.csv`**, **`AMZN_data.csv`**, and **`MSFT_data.csv`**) located in the **`Datasets`** folder. The **`read_csv()`** function reads a **`.csv`** file into a DataFrame.
3. Use Matplotlib to create a line chart visualizing the closing prices for the past 5 years (whole data sequence) for all five companies. Add labels, titles, and a legend to make the visualization self-explanatory. Use **`plt.grid()`** to add a grid to your plot. If necessary, adjust the ticks in order to make them readable.

   After executing the preceding steps, the expected output should be as follows:

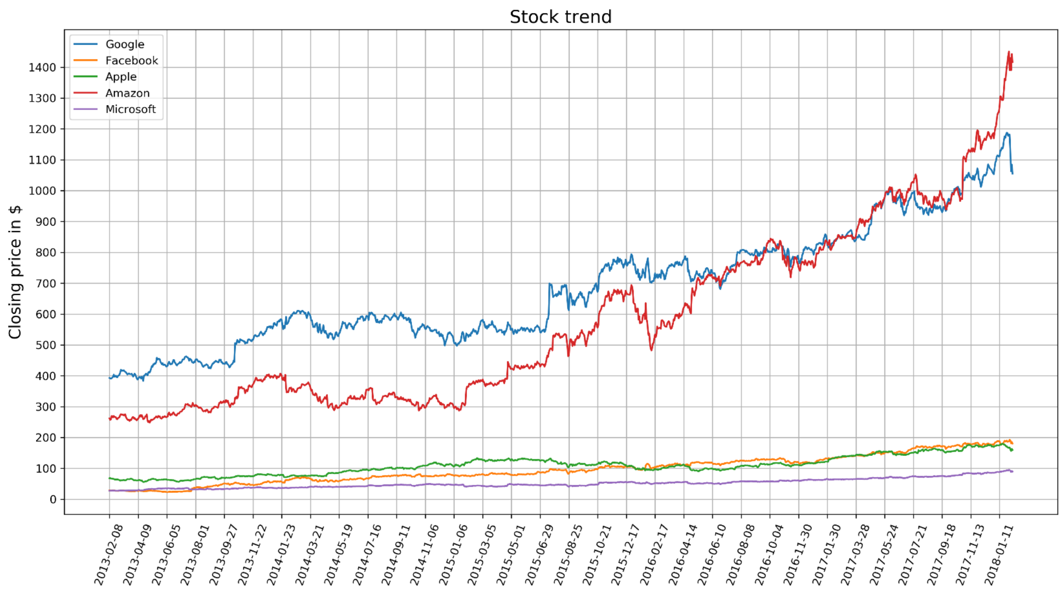

Figure 3.15: Visualization of stock trends of five companies

NOTE

The solution for this activity can be found via this link.

This covers the most important things about pyplot. In the following section, we will talk about how to create various plots in Matplotlib.

Basic Plots

In this section, we are going to go through the different types of simple plots. This includes bar charts, pie charts, stacked bar, and area charts, histograms, box plots, scatter plots and bubble plots. Please refer to the previous chapter to get more details about these plots. More sophisticated plots, such as violin plots, will be covered in the next chapter, using Seaborn instead of Matplotlib.

Bar Chart

The **`plt.bar(x, height, [width])`** creates a vertical bar plot. For horizontal bars, use the **`plt.barh()`** function.

Important parameters:

- **`x`**: Specifies the x coordinates of the bars
- **`height`**: Specifies the height of the bars
- **`width`** (optional): Specifies the width of all bars; the default is 0.8

Example:

```
plt.bar(['A', 'B', 'C', 'D'], [20, 25, 40, 10])
```

The preceding code creates a bar plot, as shown in the following diagram:

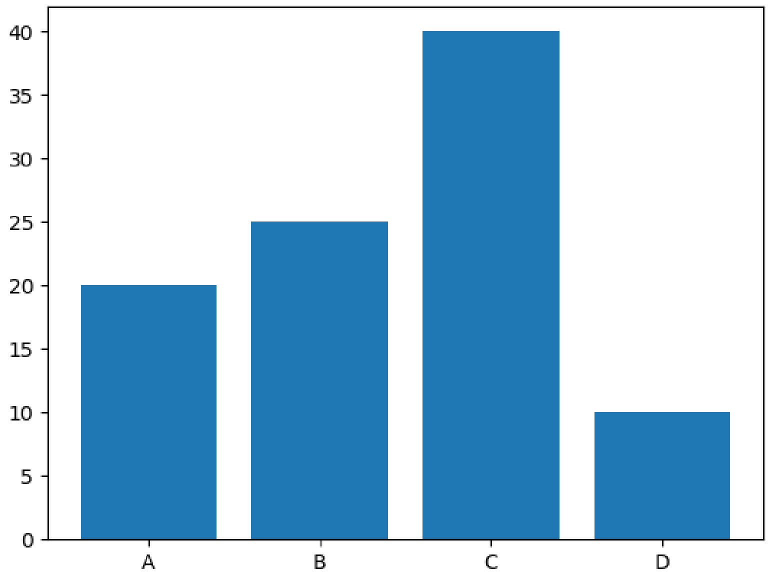

Figure 3.16: A simple bar chart

If you want to have subcategories, you have to use the **plt.bar()** function multiple times with shifted x-coordinates. This is done in the following example and illustrated in the figure that follows. The **arange()** function is a method in the **NumPy** package that returns evenly spaced values within a given interval. The **gca()** function helps in getting the instance of current axes on any current Figure. The **set_xticklabels()** function is used to set the x-tick labels with the list of given string labels.

Example:

```
labels = ['A', 'B', 'C', 'D']
x = np.arange(len(labels))
width = 0.4
plt.bar(x - width / 2, [20, 25, 40, 10], width=width)
plt.bar(x + width / 2, [30, 15, 30, 20], width=width)
# Ticks and tick labels must be set manually
plt.xticks(x)
ax = plt.gca()
ax.set_xticklabels(labels)
```

This creates a bar chart as shown in the following diagram:

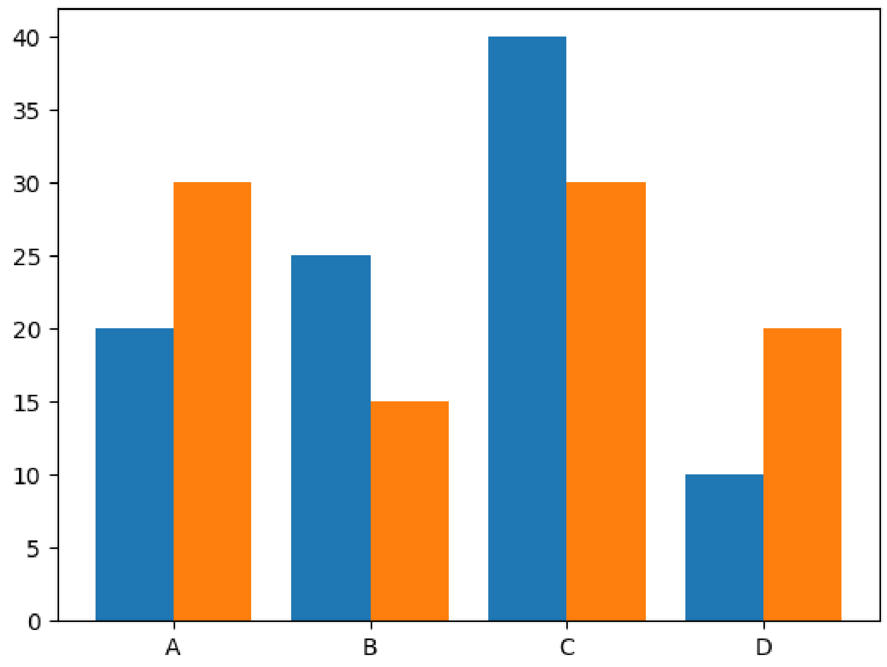

Figure 3.17: Bar chart with subcategories

After providing the theoretical foundation for creating bar charts in Matplotlib, you can apply your acquired knowledge in practice with the following activity.

ACTIVITY 3.02: CREATING A BAR PLOT FOR MOVIE COMPARISON

In this activity, we will create visually appealing bar plots. We will use a bar plot to compare movie scores. You are given five movies with scores from Rotten Tomatoes. The Tomatometer is the percentage of approved Tomatometer critics who have given a positive review for the movie. The Audience Score is the percentage of users who have given a score of 3.5 or higher out of 5. Compare these two scores among the five movies.

The following are the steps to perform:

1. Import the necessary modules and enable plotting within a Jupyter Notebook.
2. Use pandas to read the data located in the **`Datasets`** subfolder.
3. Use Matplotlib to create a visually appealing bar plot comparing the two scores for all five movies.
4. Use the movie titles as labels for the x-axis. Use percentages at intervals of 20 for the y-axis and minor ticks at intervals of 5. Add a legend and a suitable title to the plot.
5. Use functions that are required to explicitly specify the axes. To get the reference to the current axes, use **`ax = plt.gca()`**. To add minor y-ticks, use **`Axes.set_yticks([ticks], minor=True)`**. To add a horizontal grid for major ticks, use **`Axes.yaxis.grid(which='major')`**, and to add a dashed horizontal grid for minor ticks, use **`Axes.yaxis.grid(which='minor', linestyle='--')`**.

The expected output is as follows:

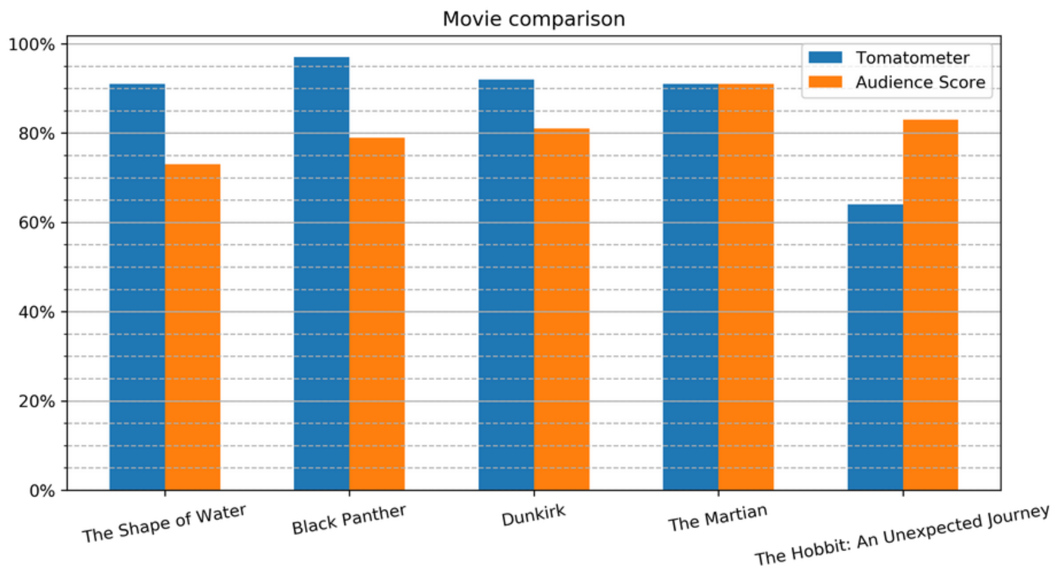

Figure 3.18: Bar plot comparing scores of five movies

NOTE

The solution for this activity can be found via this link.

After practicing the creation of bar plots, we will discuss how to create pie charts in Matplotlib in the following section.

PIE CHART

The `plt.pie(x, [explode], [labels], [autopct])` function creates a pie chart.

Important parameters:

- `x`: Specifies the slice sizes.
- `explode` (optional): Specifies the fraction of the radius offset for each slice. The `explode-array` must have the same length as the `x-array`.

- **`labels`** (optional): Specifies the labels for each slice.
- **`autopct`** (optional): Shows percentages inside the slices according to the specified format string. Example: **`'%1.1f%%'`**.

Example:

```
plt.pie([0.4, 0.3, 0.2, 0.1], explode=(0.1, 0, 0, 0), \
        labels=['A', 'B', 'C', 'D'])
```

The result of the preceding code is visualized in the following diagram:

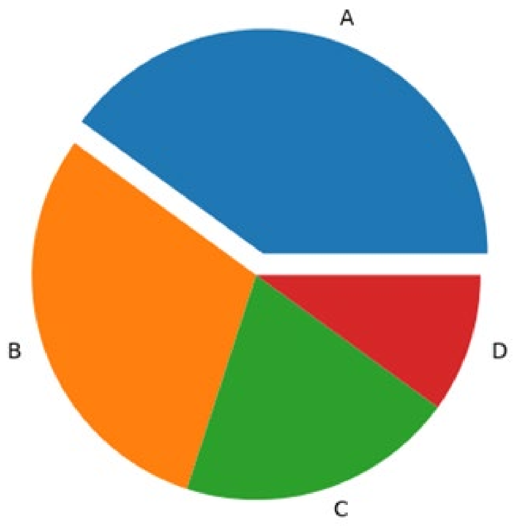

Figure 3.19: Basic pie chart

After this short introduction to pie charts, we will create a more sophisticated pie chart that visualizes the water usage in a common household in the following exercise.

EXERCISE 3.02: CREATING A PIE CHART FOR WATER USAGE

In this exercise, we will use a pie chart to visualize water usage. There has been a shortage of water in your locality in the past few weeks. To understand the reason behind it, generate a visual representation of water usage using pie charts.

The following are the steps to perform:

1. Create an **`Exercise3.02.ipynb`** Jupyter Notebook in the **`Chapter03/Exercise3.02`** folder to implement this exercise.
2. Import the necessary modules and enable plotting within the Jupyter Notebook:

```
# Import statements
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline
```

3. Use pandas to read the data located in the **`Datasets`** subfolder:

```
# Load dataset
data = pd.read_csv('../../Datasets/water_usage.csv')
```

4. Use a pie chart to visualize water usage. Highlight one usage of your choice using the **`explode`** parameter. Show the percentages for each slice and add a title:

```
# Create figure
plt.figure(figsize=(8, 8), dpi=300)
# Create pie plot
plt.pie('Percentage', explode=(0, 0, 0.1, 0, 0, 0), \
        labels='Usage', data=data, autopct='%.0f%%')
# Add title
plt.title('Water usage')
# Show plot
plt.show()
```

The output is as follows:

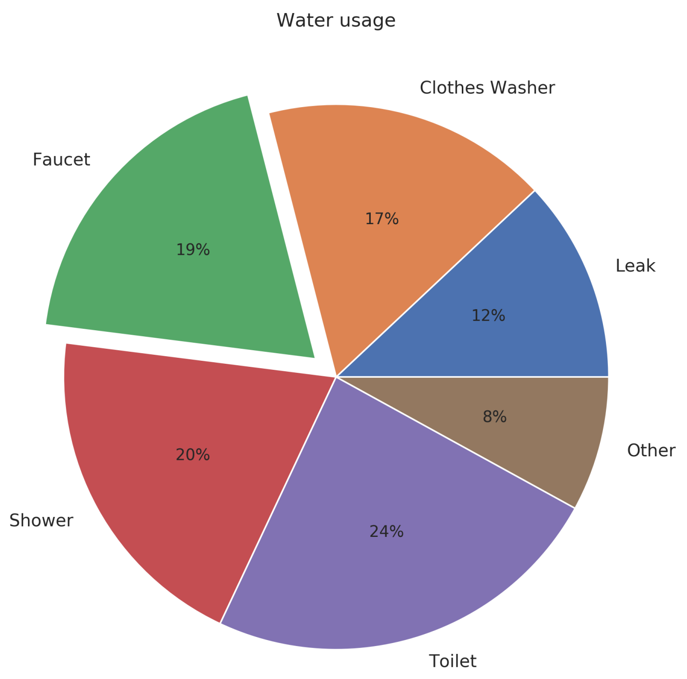

Figure 3.20: Pie chart for water usage

Pie charts are a common way to show part-of-a-whole relationships, as you've seen in the previous exercise. Another visualization that falls into this category are stacked bar charts.

NOTE

To access the source code for this specific section, please refer to https://packt.live/3frXRrZ.

You can also run this example online at https://packt.live/2Y4D1cd.

In the next section, we will learn how to generate a stacked bar chart and implement an activity on it.

STACKED BAR CHART

A **stacked bar chart** uses the same **`plt.bar`** function as bar charts. For each stacked bar, the **`plt.bar`** function must be called, and the **`bottom`** parameter must be specified, starting with the second stacked bar. This will become clear with the following example:

```
plt.bar(x, bars1)
plt.bar(x, bars2, bottom=bars1)
plt.bar(x, bars3, bottom=np.add(bars1, bars2))
```

The result of the preceding code is visualized in the following diagram:

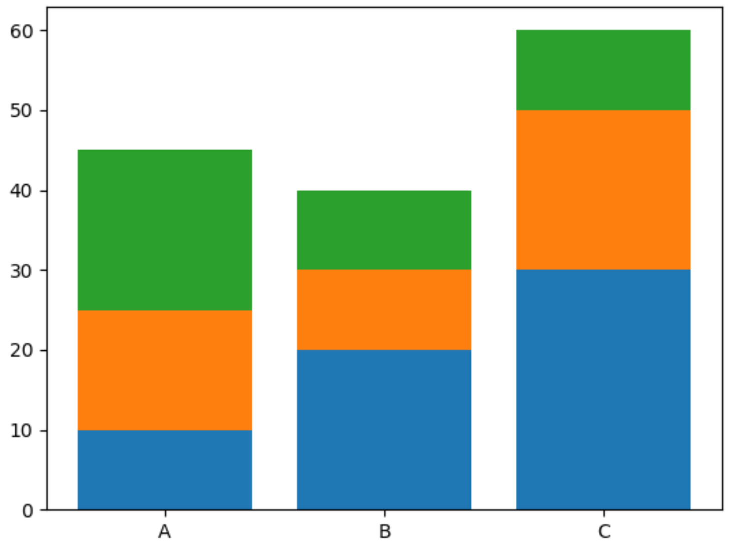

Figure 3.21: A stacked bar chart

Let's get some more practice with stacked bar charts in the following activity.

Activity 3.03: Creating a Stacked Bar Plot to Visualize Restaurant Performance

In this activity, we will use a stacked bar plot to visualize the performance of a restaurant. Let's look at the following scenario: you are the owner of a restaurant and, due to a new law, you have to introduce a *No Smoking Day*. To make as few losses as possible, you want to visualize how many sales are made every day, categorized by smokers and non-smokers.

Use the dataset tips from Seaborn, which contains multiple entries of restaurant bills, and create a matrix where the elements contain the sum of the total bills for each day and smokers/non-smokers:

> **Note**
>
> For this exercise, we will import the Seaborn library as **`import seaborn as sns`**. The dataset can be loaded using this code: **`bills = sns.load_dataset('tips')`**.
>
> We will learn in detail about this in *Chapter 4*, *Simplifying Visualizations Using Seaborn*.

1. Import all the necessary dependencies and load the **`tips`** dataset. Note that we have to import the Seaborn library to load the dataset.
2. Use the given dataset and create a matrix where the elements contain the sum of the total bills for each day and split according to smokers/non-smokers.
3. Create a stacked bar plot, stacking the summed total bills separated according to smoker and non-smoker for each day.
4. Add a legend, labels, and a title.

   After executing the preceding steps, the expected output should be as follows:

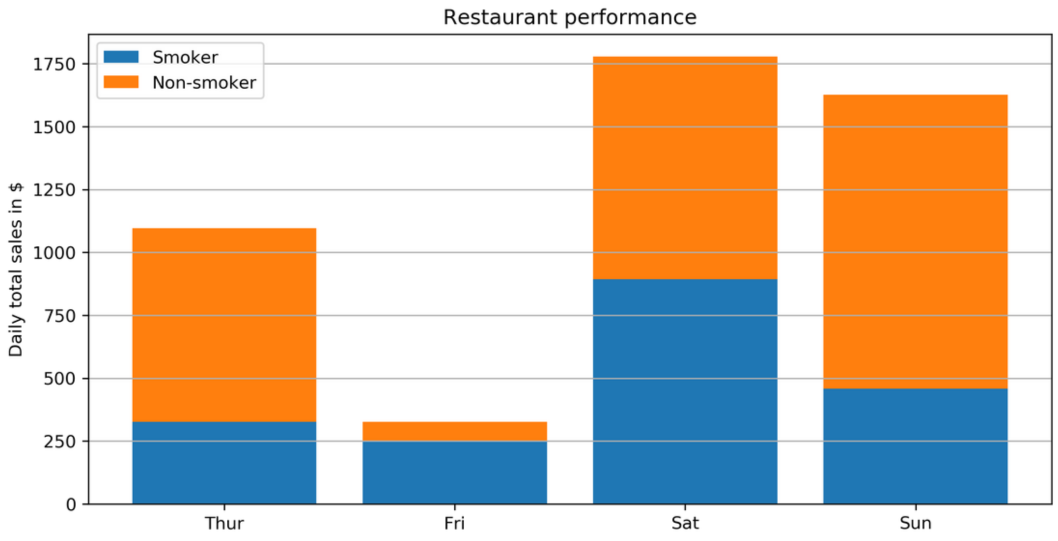

Figure 3.22: Stacked bar chart showing the performance of a restaurant on different days

NOTE

The solution for this activity can be found via this link.

In the following section, stacked area charts will be covered, which, in comparison to stacked bar charts, are suited to visualizing part-of-a-whole relationships for time series data.

STACKED AREA CHART

`plt.stackplot(x, y)` creates a stacked area plot.

Important parameters:

- **`x`**: Specifies the x-values of the data series.
- **`y`**: Specifies the y-values of the data series. For multiple series, either as a 2D array or any number of 1D arrays, call the following function: **`plt.stackplot(x, y1, y2, y3, …)`**.
- **`labels`** (optional): Specifies the labels as a list or tuple for each data series.

Example:

```
plt.stackplot([1, 2, 3, 4], [2, 4, 5, 8], [1, 5, 4, 2])
```

The result of the preceding code is shown in the following diagram:

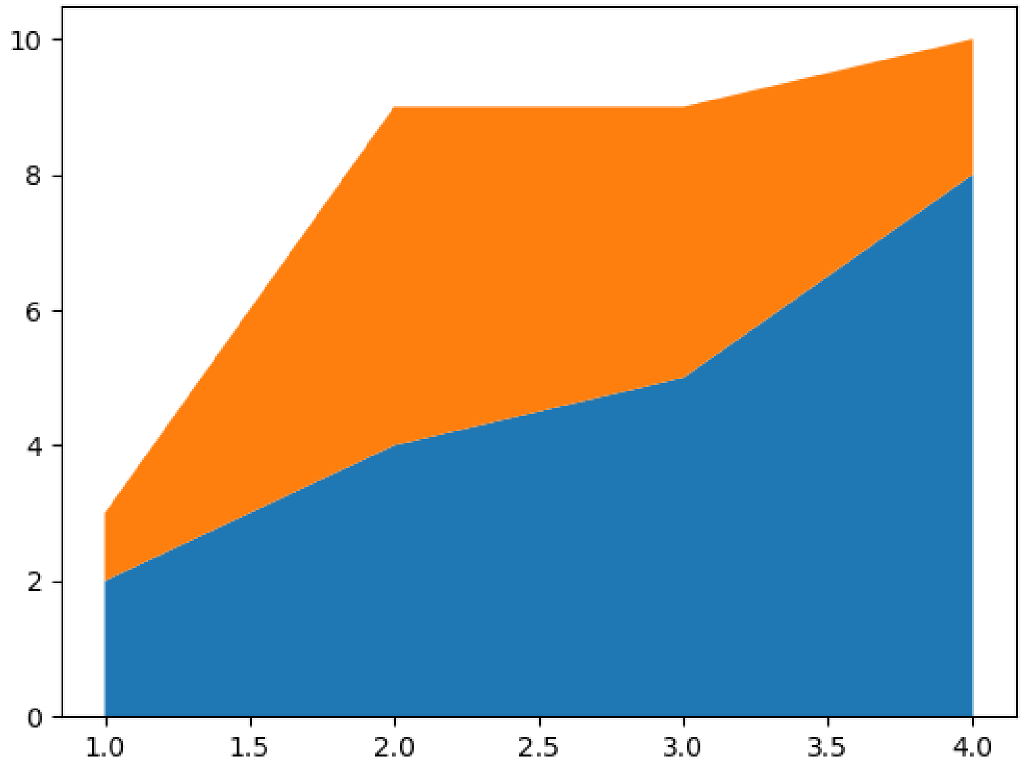

Figure 3.23: Stacked area chart

Let's get some more practice regarding stacked area charts in the following activity.

Activity 3.04: Comparing Smartphone Sales Units Using a Stacked Area Chart

In this activity, we will compare smartphone sales units using a stacked area chart. Let's look at the following scenario: you want to invest in one of the five biggest smartphone manufacturers. Looking at the quarterly sales units as part of a whole may be a good indicator of which company to invest in:

1. Import the necessary modules and enable plotting within a Jupyter Notebook.
2. Use pandas to read the **`smartphone_sales.csv`** dataset located in the **`Datasets`** subfolder.
3. Create a visually appealing stacked area chart. Add a legend, labels, and a title.

   After executing the preceding steps, the expected output should be as follows:

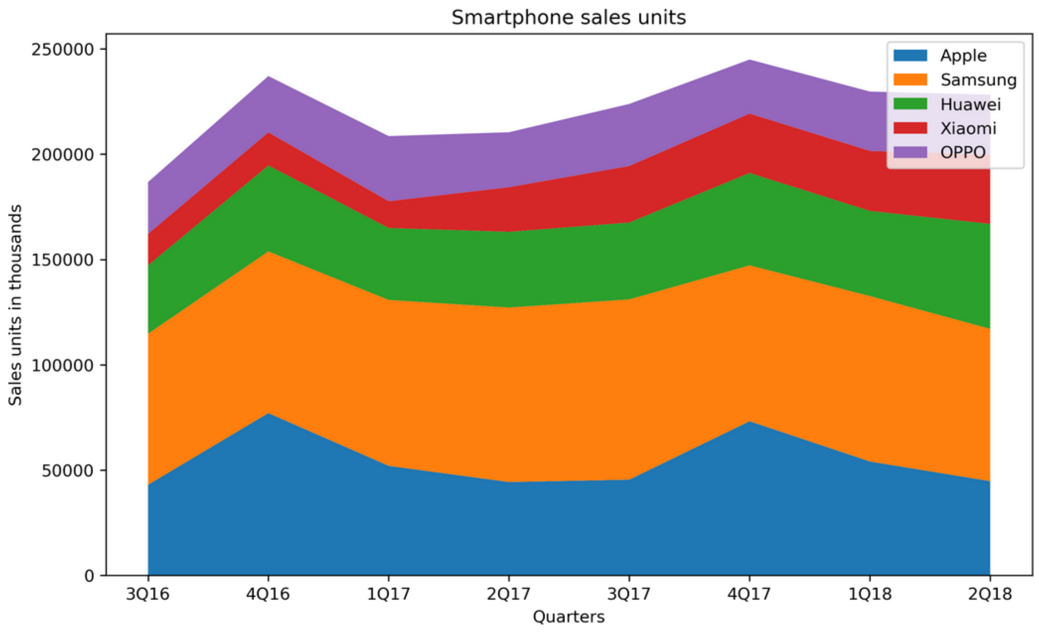

Figure 3.24: Stacked area chart comparing sales units of different smartphone manufacturers

Note

The solution for this activity can be found via this link.

In the following section, the histogram will be covered, which helps to visualize the distribution of a single numerical variable.

HISTOGRAM

A histogram visualizes the distribution of a single numerical variable. Each bar represents the frequency for a certain interval. The **`plt.hist(x)`** function creates a histogram.

Important parameters:

- **`x`**: Specifies the input values.
- **`bins`**: (optional): Specifies the number of bins as an integer or specifies the bin edges as a list.
- **`range`**: (optional): Specifies the lower and upper range of the bins as a tuple.
- **`density`**: (optional): If true, the histogram represents a probability density.

Example:

```
plt.hist(x, bins=30, density=True)
```

The result of the preceding code is shown in the following diagram:

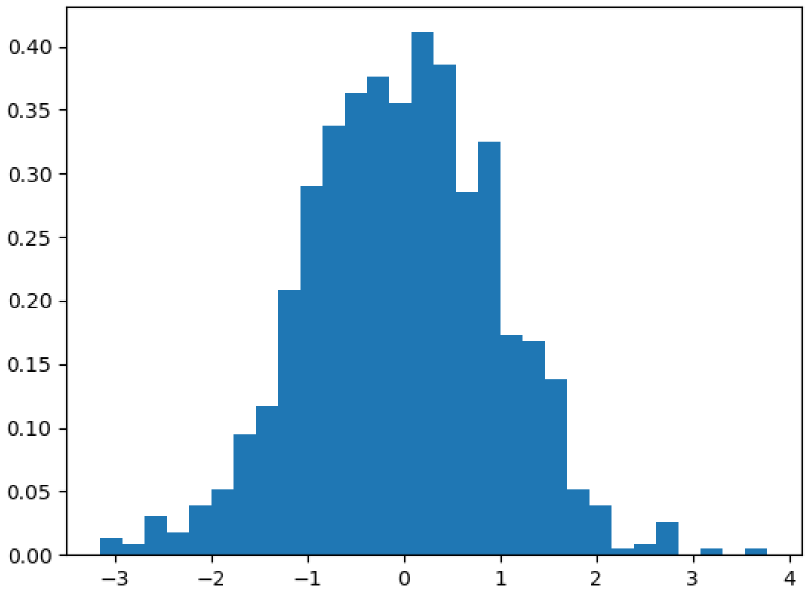

Figure 3.25: Histogram

`plt.hist2d(x, y)` creates a 2D histogram. 2D histograms can be used to visualize the frequency of two-dimensional data. The data is plotted on the xy-plane and the frequency is indicated by the color. An example of a 2D histogram is shown in the following diagram:

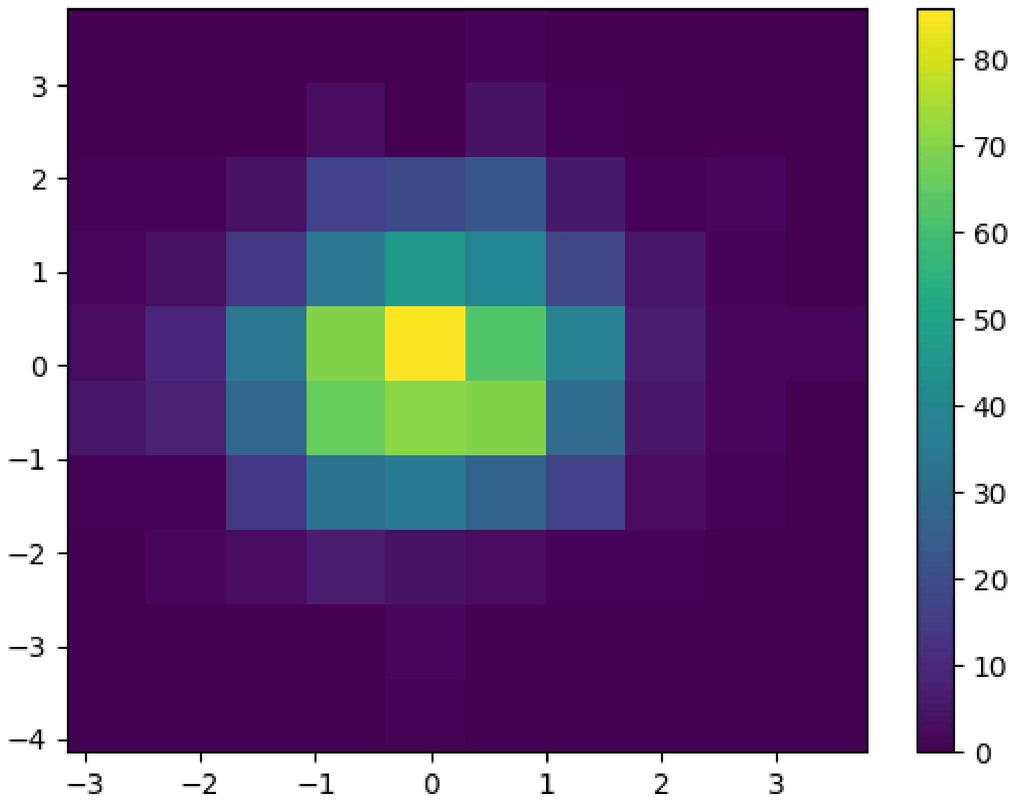

Figure 3.26: 2D histogram with color bar

Histograms are a good way to visualize an estimated density of your data. If you're only interested in summary statistics, such as central tendency or dispersion, the following covered box plots are more interesting.

BOX PLOT

The box plot shows multiple statistical measurements. The box extends from the lower to the upper quartile values of the data, thereby allowing us to visualize the interquartile range. For more details regarding the plot, refer to the previous chapter. The **`plt.boxplot(x)`** function creates a box plot.

Important parameters:

- **x**: Specifies the input data. It specifies either a 1D array for a single box, or a sequence of arrays for multiple boxes.
- **notch**: (optional) If true, notches will be added to the plot to indicate the confidence interval around the median.
- **labels**: (optional) Specifies the labels as a sequence.
- **showfliers**: (optional) By default, it is true, and outliers are plotted beyond the caps.
- **showmeans**: (optional) If true, arithmetic means are shown.

Example:

```
plt.boxplot([x1, x2], labels=['A', 'B'])
```

The result of the preceding code is shown in the following diagram:

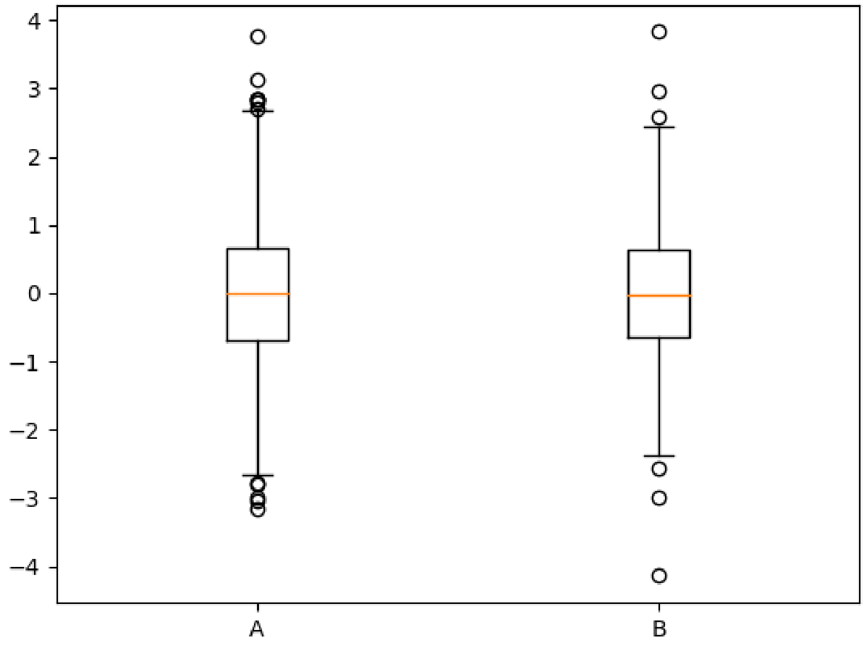

Figure 3.27: Box plot

Now that we've introduced histograms and box plots in Matplotlib, our theoretical knowledge can be practiced in the following activity, where both charts are used to visualize data regarding the intelligence quotient.

ACTIVITY 3.05: USING A HISTOGRAM AND A BOX PLOT TO VISUALIZE INTELLIGENCE QUOTIENT

In this activity, we will visualize the **intelligence quotient** (**IQ**) of 100 applicants using histogram and box plots. 100 people have come for an interview in a company. To place an individual applicant in the overall group, a histogram and a box plot shall be used.

> **NOTE**
>
> The **`plt.axvline(x, [color=…], [linestyle=…])`** function draws a vertical line at position **`x`**.

1. Import the necessary modules and enable plotting within a Jupyter Notebook.
2. Use the following IQ scores to create the plots:

```
# IQ samples
iq_scores = [126,  89,  90, 101, 102,  74,  93, 101,  66, \
             120, 108,  97,  98, 105, 119,  92, 113,  81, \
             104, 108,  83, 102, 105, 111, 102, 107, 103,  \
             89,  89, 110,  71, 110, 120,  85, 111,  83, 122, \
             120, 102, 84, 118, 100, 100, 114,  81, 109,  69,  \
             97,  95, 106, 116, 109, 114,  98,  90,  92,  98,  \
             91,  81,  85,  86, 102,  93, 112,  76, 89, 110,  \
             75, 100,  90,  96,  94, 107, 108,  95,  96,  96, \
             114, 93,  95, 117, 141, 115,  95,  86, 100, 121, \
             103,  66,  99,  96, 111, 110, 105, 110, 91, 112, \
             102, 112,  75]
```

3. Plot a histogram with 10 bins for the given IQ scores. IQ scores are normally distributed with a mean of 100 and a standard deviation of 15. Visualize the mean as a vertical solid red line, and the standard deviation using dashed vertical lines. Add labels and a title. The expected output is as follows:

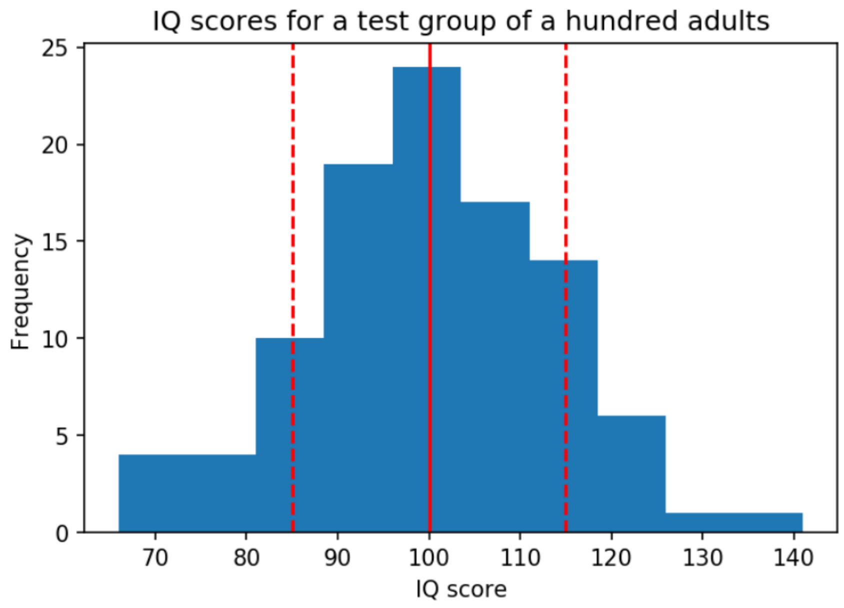

Figure 3.28: Histogram for an IQ test

4. Create a box plot to visualize the same IQ scores. Add labels and a title. The expected output is as follows:

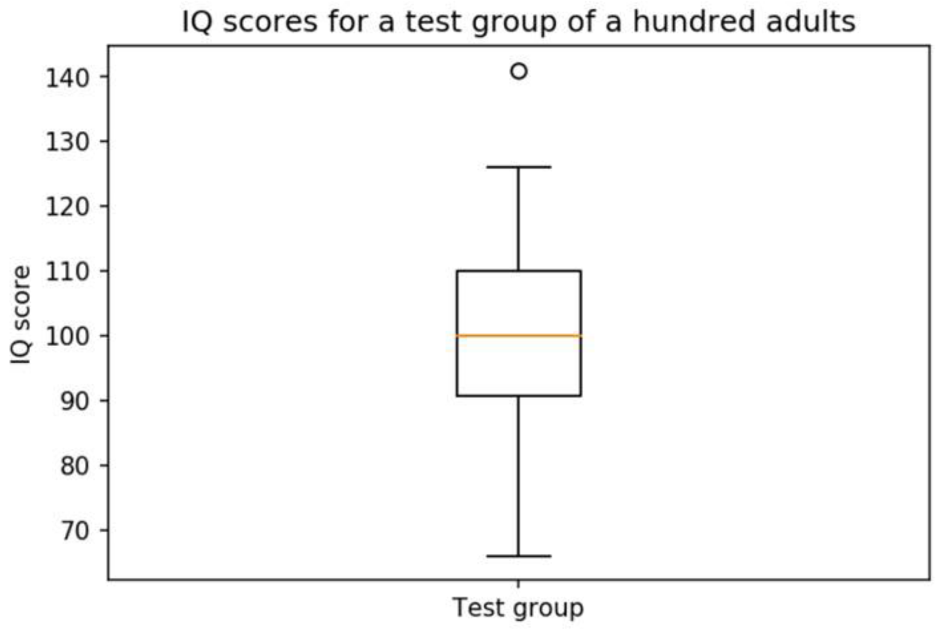

Figure 3.29: Box plot for IQ scores

5. Create a box plot for each of the IQ scores of the different test groups. Add labels and a title. The following are IQ scores for different test groups that we can use as data:

```
group_a = [118, 103, 125, 107, 111,  96, 104,  97,  96, \
           114,  96,  75, 114, 107,  87, 117, 117, 114, \
           117, 112, 107, 133,  94,  91, 118, 110, 117,  \
           86, 143,  83, 106,  86,  98, 126, 109,  91, 112, \
           120, 108, 111, 107,  98,  89, 113, 117,  81, 113, \
           112,  84, 115,  96,  93, 128, 115, 138, 121,  87, \
           112, 110,  79, 100,  84, 115,  93, 108, 130, 107, \
           106, 106, 101, 117,  93,  94, 103, 112,  98, 103,  \
           70, 139,  94, 110, 105, 122,  94,  94, 105, 129, \
           110, 112,  97, 109, 121, 106, 118, 131,  88, 122, \
           125,  93,  78]

group_b = [126,  89,  90, 101, 102,  74,  93, 101,  66, \
           120, 108,  97,  98, 105, 119,  92, 113,  81, \
           104, 108,  83, 102, 105, 111, 102, 107, 103,  \
           89,  89, 110,  71, 110, 120,  85, 111,  83, \
           122, 120, 102, 84, 118, 100, 100, 114,  81, \
           109,  69,  97,  95, 106, 116, 109, 114, 98,  \
           90,  92,  98,  91,  81,  85,  86, 102,  93, 112,  \
           76, 89, 110,  75, 100,  90,  96,  94, 107, 108,  \
           95,  96,  96, 114, 93,  95, 117, 141, 115,  95,  \
           86, 100, 121, 103,  66,  99,  96, 111, 110, 105, \
           110,  91, 112, 102, 112,  75]

group_c = [108,  89, 114, 116, 126, 104, 113,  96,  69, 121, \
           109, 102, 107, 122, 104, 107, 108, 137, 107, 116,  \
           98, 132, 108, 114,  82,  93, 89,  90,  86,  91,  \
           99,  98,  83,  93, 114,  96,  95, 113, 103, 81, \
           107,  85, 116,  85, 107, 125, 126, 123, 122, 124, \
           115, 114, 93,  93, 114, 107, 107,  84, 131,  91, \
           108, 127, 112, 106, 115, 82,  90, 117, 108, 115, \
           113, 108, 104, 103,  90, 110, 114,  92, 101,  72, \
           109,  94, 122,  90, 102,  86, 119, 103, 110,  96,  \
                90, 110,  96,  69,  85, 102,  69,  96, 101,  90]

    group_d = [93,  99,  91, 110,  80, 113, 111, 115,  98,  74,  \
                96,  80,  83, 102,  60,  91,  82,  90,  97, 101,  \
                89,  89, 117,  91, 104, 104, 102, 128, 106, 111,  \
                79,  92,  97, 101, 106, 110,  93,  93, 106, 108,  \
                85,  83, 108,  94,  79,  87, 113, 112, 111, 111,  \
                79, 116, 104,  84, 116, 111, 103, 103, 112,  68,  \
                54,  80,  86, 119,  81, 84,  91,  96, 116, 125,  \
                99,  58, 102,  77,  98, 100,  90, 106, 109, 114, \
                102, 102, 112, 103,  98,  96,  85,  97, 110, 131,  \
                92, 79, 115, 122,  95, 105,  74,  85,  85,  95]
```

The expected output is as follows:

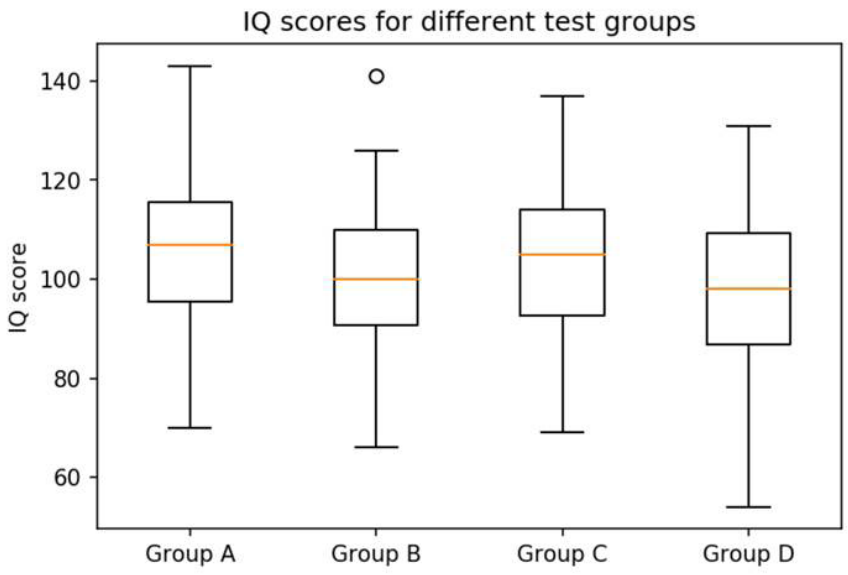

Figure 3.30: Box plot for IQ scores of different test groups

> **NOTE**
>
> The solution for this activity can be found via this link.

In the next section, we will learn how to generate a scatter plot.

Scatter Plot

Scatter plots show data points for two numerical variables, displaying a variable on both axes. **`plt.scatter(x, y)`** creates a scatter plot of **`y`** versus **`x`**, with optionally varying marker size and/or color.

Important parameters:

- **`x`**, **`y`**: Specifies the data positions.
- **`s`**: (optional) Specifies the marker size in points squared.
- **`c`**: (optional) Specifies the marker color. If a sequence of numbers is specified, the numbers will be mapped to the colors of the color map.

Example:

```
plt.scatter(x, y)
```

The result of the preceding code is shown in the following diagram:

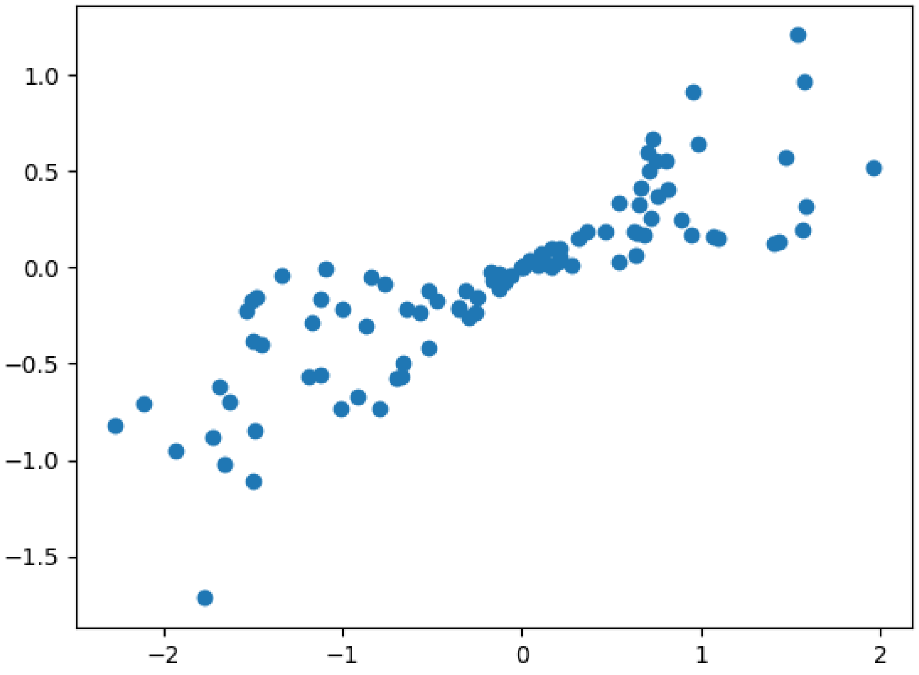

Figure 3.31: Scatter plot

Let's implement a scatter plot in the following exercise.

EXERCISE 3.03: USING A SCATTER PLOT TO VISUALIZE CORRELATION BETWEEN VARIOUS ANIMALS

In this exercise, we will use a scatter plot to show correlation within a dataset. Let's look at the following scenario: You are given a dataset containing information about various animals. Visualize the correlation between the various animal attributes such as *Maximum longevity in years* and *Body mass in grams*.

> **NOTE**
>
> The **`Axes.set_xscale('log')`** and the **`Axes.set_yscale('log')`** change the scale of the x-axis and y-axis to a logarithmic scale, respectively.

Let's visualize the correlation between various animals with the help of a scatter plot:

1. Create an **`Exercise3.03.ipynb`** Jupyter Notebook in the **`Chapter03/Exercise3.03`** folder to implement this exercise.
2. Import the necessary modules and enable plotting within the Jupyter Notebook:

```
# Import statements
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

3. Use pandas to read the data located in the **`Datasets`** folder:

```
# Load dataset
data = pd.read_csv('../../Datasets/anage_data.csv')
```

4. The given dataset is not complete. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Sort the data according to the animal class; here, the **`isfinite()`** function (to check whether the number is finite or not) checks for the finiteness of the given element:

```
# Preprocessing
longevity = 'Maximum longevity (yrs)'
mass = 'Body mass (g)'
data = data[np.isfinite(data[longevity]) \
       & np.isfinite(data[mass])]
```

```
# Sort according to class
amphibia = data[data['Class'] == 'Amphibia']
aves = data[data['Class'] == 'Aves']
mammalia = data[data['Class'] == 'Mammalia']
reptilia = data[data['Class'] == 'Reptilia']
```

5. Create a scatter plot visualizing the correlation between the body mass and the maximum longevity. Use different colors to group data samples according to their class. Add a legend, labels, and a title. Use a log scale for both the x-axis and y-axis:

```
# Create figure
plt.figure(figsize=(10, 6), dpi=300)
# Create scatter plot
plt.scatter(amphibia[mass], amphibia[longevity], \
            label='Amphibia')
plt.scatter(aves[mass], aves[longevity], \
            label='Aves')
plt.scatter(mammalia[mass], mammalia[longevity], \
            label='Mammalia')
plt.scatter(reptilia[mass], reptilia[longevity], \
            label='Reptilia')
# Add legend
plt.legend()
# Log scale
ax = plt.gca()
ax.set_xscale('log')
ax.set_yscale('log')
# Add labels
plt.xlabel('Body mass in grams')
plt.ylabel('Maximum longevity in years')
# Show plot
plt.show()
```

The following is the output of the code:

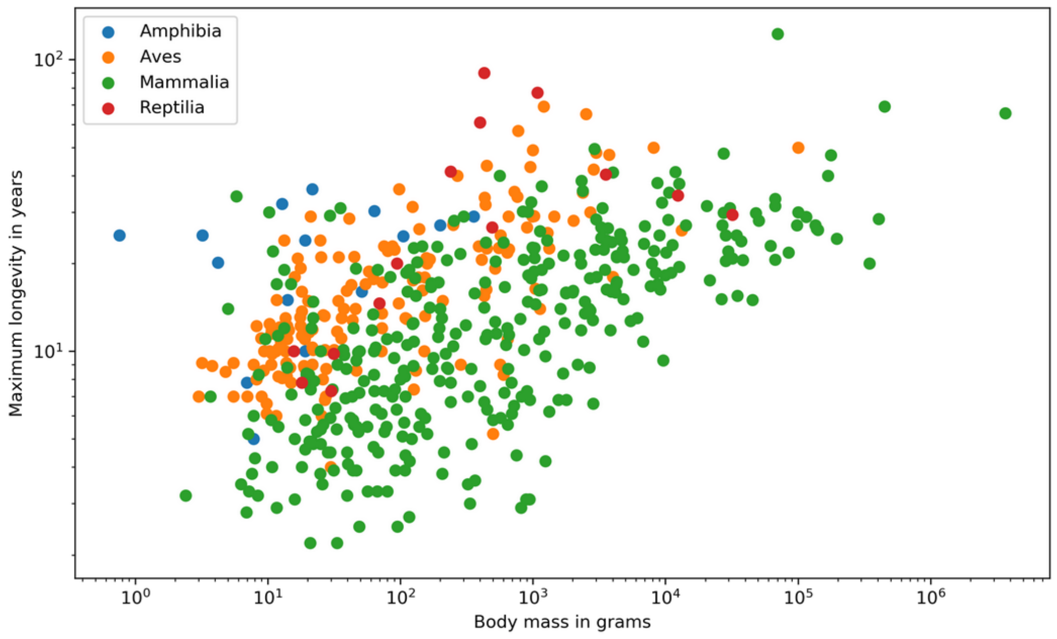

Figure 3.32: Scatter plot on animal statistics

From the preceding output, we can visualize the correlation between various animals based on the maximum longevity in years and body mass in grams.

NOTE

To access the source code for this specific section, please refer to https://packt.live/3fsozRf.

You can also run this example online at https://packt.live/37yk0C7.

Next, we will learn how to generate a bubble plot.

BUBBLE PLOT

The **`plt.scatter`** function is used to create a bubble plot. To visualize a third or fourth variable, the parameters **`s`** (scale) and **`c`** (color) can be used.

Example:

```
plt.scatter(x, y, s=z*500, c=c, alpha=0.5)
plt.colorbar()
```

The **colorbar** function adds a colorbar to the plot, which indicates the value of the color. The result is shown in the following diagram:

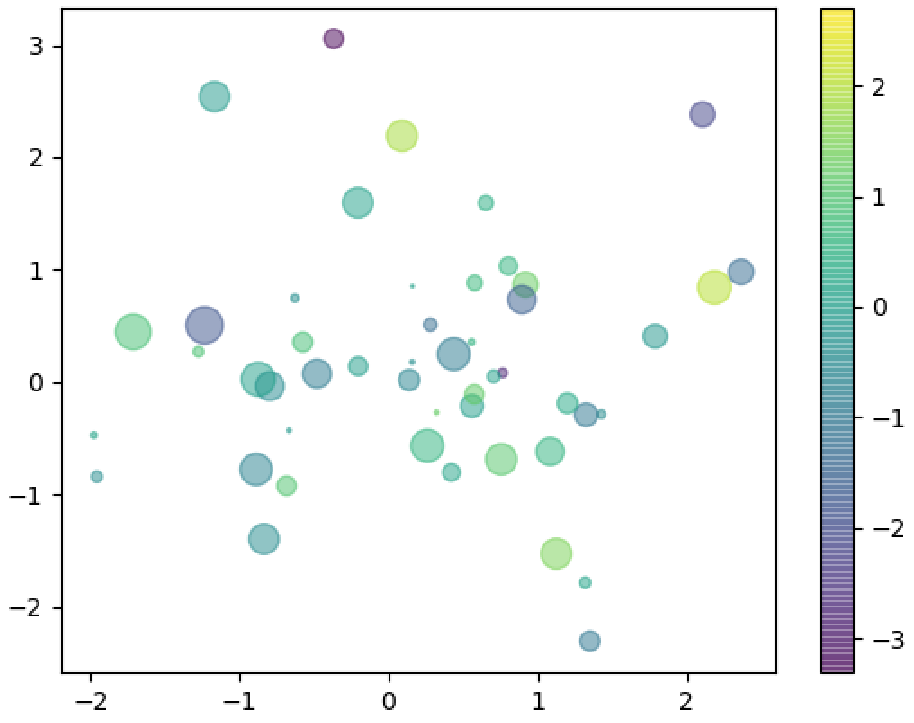

Figure 3.33: Bubble plot with color bar

LAYOUTS

There are multiple ways to define a visualization layout in Matplotlib. By layout, we mean the arrangement of multiple Axes within a Figure. We will start with **subplots** and how to use the **tight layout** to create visually appealing plots and then cover **GridSpec**, which offers a more flexible way to create multi-plots.

Subplots

It is often useful to display several plots next to one another. Matplotlib offers the concept of subplots, which are multiple Axes within a Figure. These plots can be grids of plots, nested plots, and so on.

Explore the following options to create subplots:

- The **`plt.subplots(, ncols)`** function creates a Figure and a set of subplots. **`nrows, ncols`** define the number of rows and columns of the subplots, respectively.
- The **`plt.subplot(nrows, ncols, index)`** function or, equivalently, **`plt.subplot(pos)`** adds a subplot to the current Figure. The index starts at 1. The **`plt.subplot(2, 2, 1)`** function is equivalent to **`plt.subplot(221)`**.
- The **`Figure.subplots(nrows, ncols)`** function adds a set of subplots to the specified Figure.
- The **`Figure.add_subplot(nrows, ncols, index)`** function or, equivalently, **`Figure.add_subplot(pos)`**, adds a subplot to the specified Figure.

To share the x-axis or y-axis, the parameters **`sharex`** and **`sharey`** must be set, respectively. The axis will have the same limits, ticks, and scale.

`plt.subplot` and **`Figure.add_subplot`** have the option to set a projection. For a polar projection, either set the **`projection='polar'`** parameter or the **`parameter polar=True`** parameter.

Example 1:

```
fig, axes = plt.subplots(2, 2)
axes = axes.ravel()
for i, ax in enumerate(axes):
    ax.plot(series[i])
# [...]
for i in range(4):
    plt.subplot(2, 2, i+1)
    plt.plot(series[i])
```

Both examples yield the same result, as shown in the following diagram:

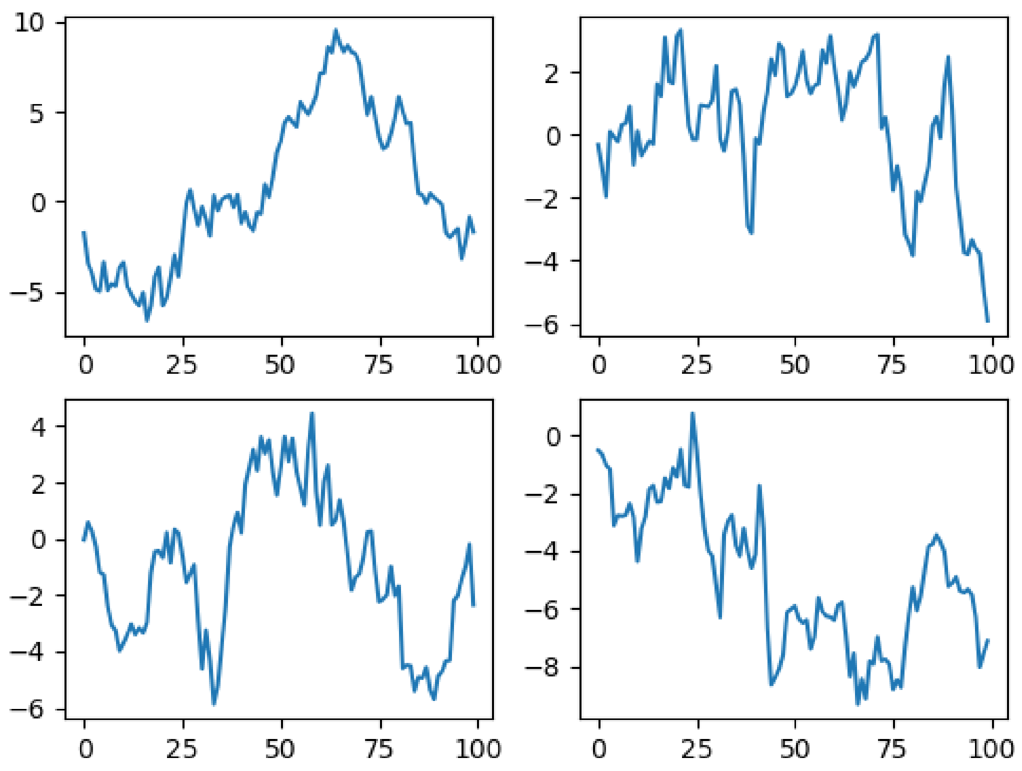

Figure 3.34: Subplots

Example 2:

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
axes = axes.ravel()
for i, ax in enumerate(axes):
    ax.plot(series[i])
```

Setting **`sharex`** and **`sharey`** to **`True`** results in the following diagram. This allows for a better comparison:

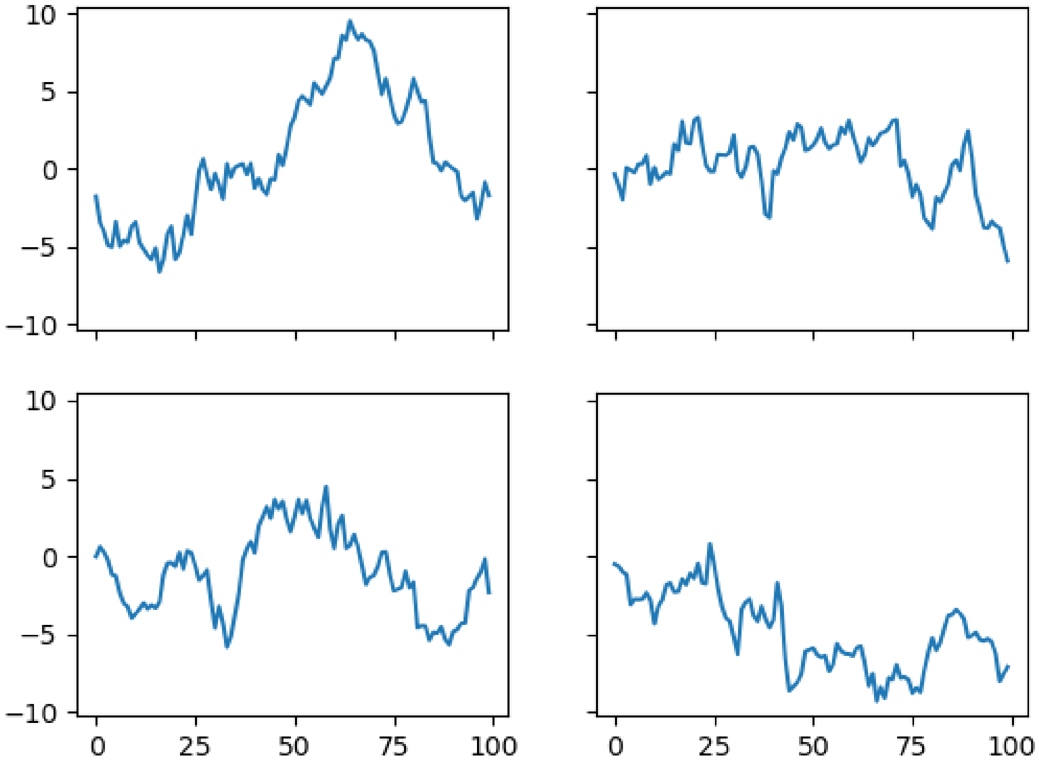

Figure 3.35: Subplots with a shared x- and y-axis

Subplots are an easy way to create a Figure with multiple plots of the same size placed in a grid. They are not really suited for more sophisticated layouts.

Tight Layout

The **`plt.tight_layout()`** adjusts subplot parameters (primarily padding between the Figure edge and the edges of subplots, and padding between the edges of adjacent subplots) so that the subplots fit well in the Figure.

Examples:

If you do not use **`plt.tight_layout()`**, subplots might overlap:

```
fig, axes = plt.subplots(2, 2)
axes = axes.ravel()
for i, ax in enumerate(axes):
    ax.plot(series[i])
    ax.set_title('Subplot ' + str(i))
```

The result of the preceding code is shown in the following diagram:

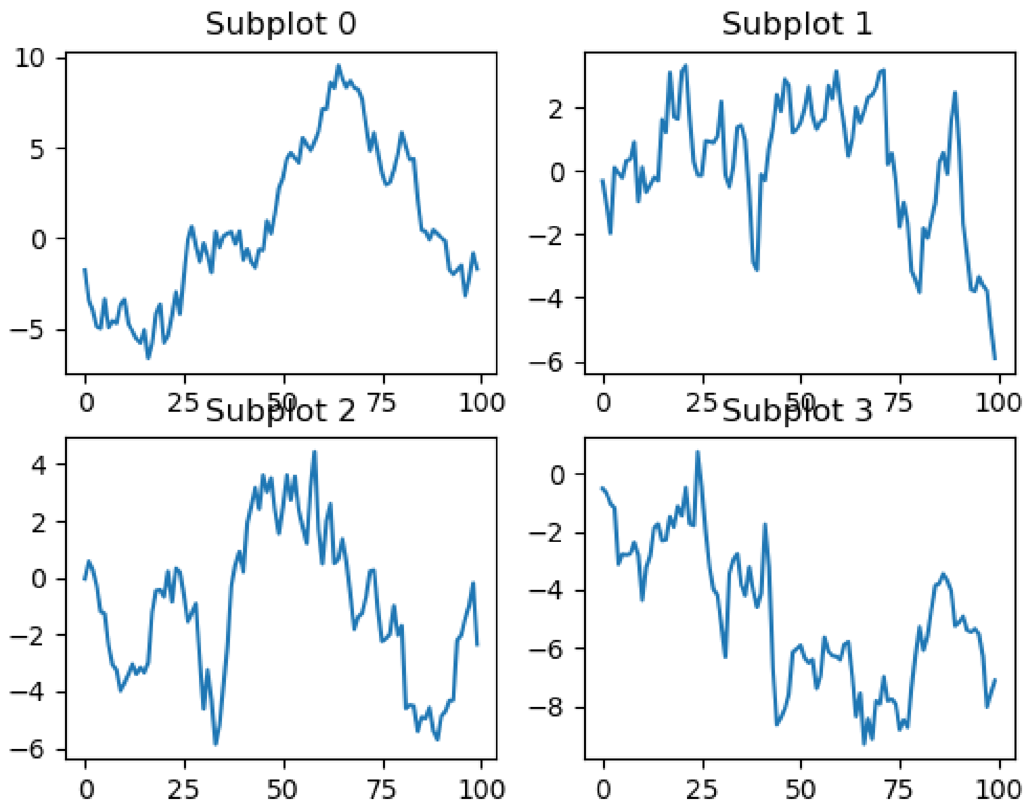

Figure 3.36: Subplots with no layout option

Using **`plt.tight_layout()`** results in no overlapping of the subplots:

```
fig, axes = plt.subplots(2, 2)
axes = axes.ravel()
for i, ax in enumerate(axes):
    ax.plot(series[i])
    ax.set_title('Subplot ' + str(i))
plt.tight_layout()
```

The result of the preceding code is shown in the following diagram:

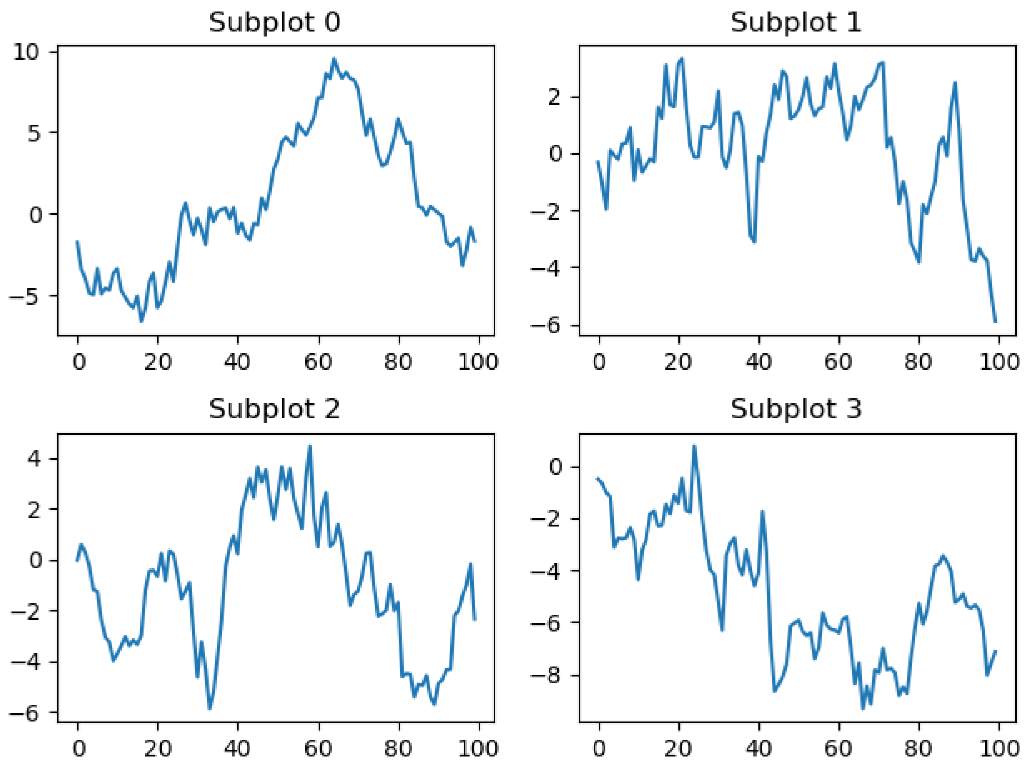

Figure 3.37: Subplots with a tight layout

RADAR CHARTS

Radar charts, also known as **spider** or **web charts**, visualize multiple variables, with each variable plotted on its own axis, resulting in a polygon. All axes are arranged radially, starting at the center with equal distance between each other, and have the same scale.

EXERCISE 3.04: WORKING ON RADAR CHARTS

As a manager of a team, you have to award a "Star Performer" trophy to an employee for the month of December. You come to the conclusion that the best way to understad the performance of your team members would be to visualize the performance of your team members in a radar chart. Thus, in this exercise, we will show you how to create a radar chart. The following are the steps to perform this exercise:

1. Create an **Exercise3.04.ipynb** Jupyter Notebook in the **Chapter03/Exercise3.04** folder to implement this exercise.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

```
# Import settings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

3. The following dataset contains ratings of five different attributes for four employees:

```
"""
Sample data
Attributes: Efficiency, Quality, Commitment, Responsible Conduct,
Cooperation
"""
data = \
pd.DataFrame({'Employee': ['Alex', 'Alice', \
                           'Chris', 'Jennifer'], \
              'Efficiency': [5, 4, 4, 3,],
              'Quality': [5, 5, 3, 3],
              'Commitment': [5, 4, 4, 4],
              'Responsible Conduct': [4, 4, 4, 3],
              'Cooperation': [4, 3, 4, 5]})
```

4. Create angle values and close the plot:

```
attributes = list(data.columns[1:])
values = list(data.values[:, 1:])
employees = list(data.values[:, 0])
angles = [n / float(len(attributes)) * 2 \
          * np.pi for n in range(len(attributes))]
# Close the plot
angles += angles[:1]
values = np.asarray(values)
values = np.concatenate([values, values[:, 0:1]], axis=1)
```

5. Create subplots with the polar projection. Set a tight layout so that nothing overlaps:

```
# Create figure
plt.figure(figsize=(8, 8), dpi=150)
# Create subplots
for i in range(4):
    ax = plt.subplot(2, 2, i + 1, polar=True)
    ax.plot(angles, values[i])
    ax.set_yticks([1, 2, 3, 4, 5])
    ax.set_xticks(angles)
    ax.set_xticklabels(attributes)
    ax.set_title(employees[i], fontsize=14, color='r')
# Set tight layout
plt.tight_layout()
# Show plot
plt.show()
```

The following diagram shows the output of the preceding code:

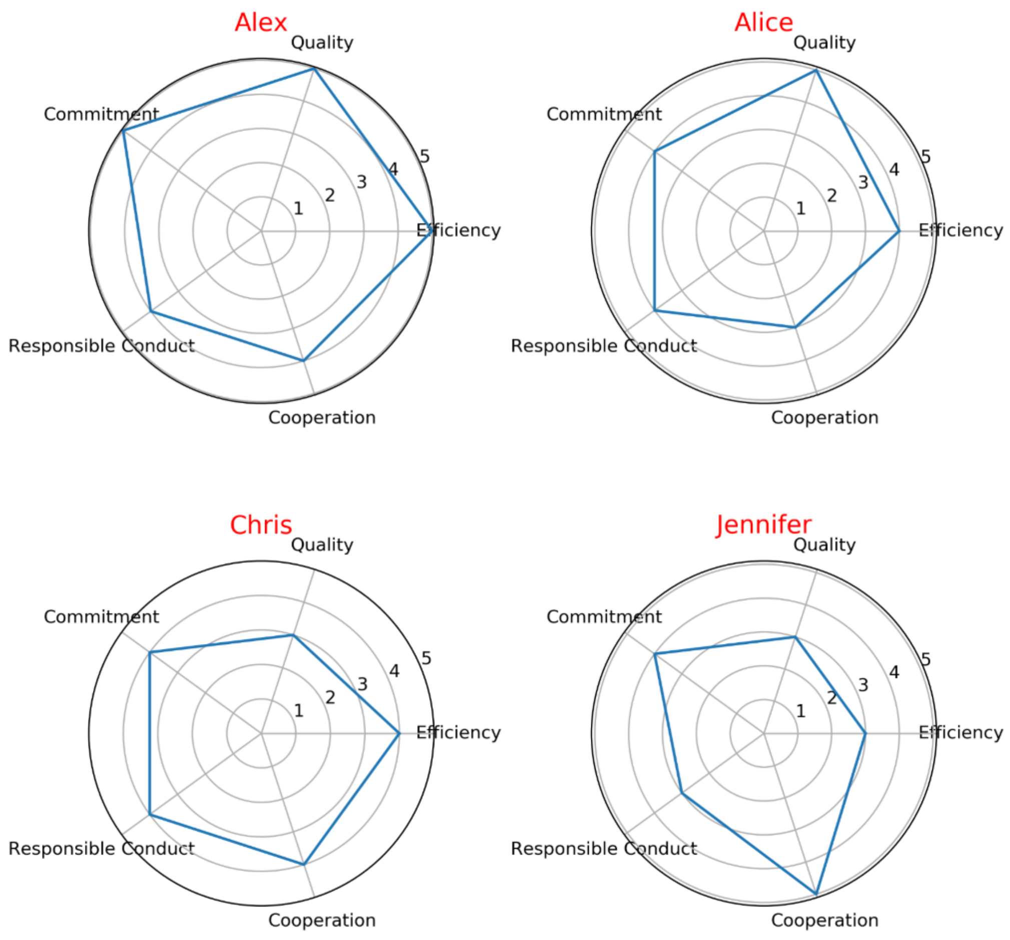

Figure 3.38: Radar charts

From the preceding output, we can clearly see how the various team members have performed in terms of metrics such as Quality, Efficiency, Cooperation, Responsible Conduct, and Commitment. You can easily draw the conclusion that Alex outperforms his collegues when all metrics are considered. In the next section, we will learn how to use the **`GridSpec`** function.

NOTE

To access the source code for this specific section, please refer to https://packt.live/3e6is4X.

You can also run this example online at https://packt.live/3hxeFjf.

GridSpec

The **`matplotlib.gridspec.GridSpec(nrows, ncols)`** function specifies the geometry of the grid in which a subplot will be placed. For example, you can specify a grid with three rows and four columns. As a next step, you have to define which elements of the gridspec are used by a subplot; elements of a gridspec are accessed in the same way as NumPy arrays. You could, for example, only use a single element of a gridspec for a subplot and therefore end up with 12 subplots in total. Another possibility, as shown in the following example, is to create a bigger subplot using 3x3 elements of the gridspec and another three subplots with a single element each.

Example:

```
gs = matplotlib.gridspec.GridSpec(3, 4)
ax1 = plt.subplot(gs[:3, :3])
ax2 = plt.subplot(gs[0, 3])
ax3 = plt.subplot(gs[1, 3])
ax4 = plt.subplot(gs[2, 3])
ax1.plot(series[0])
ax2.plot(series[1])
ax3.plot(series[2])
ax4.plot(series[3])
plt.tight_layout()
```

The result of the preceding code is shown in the following diagram:

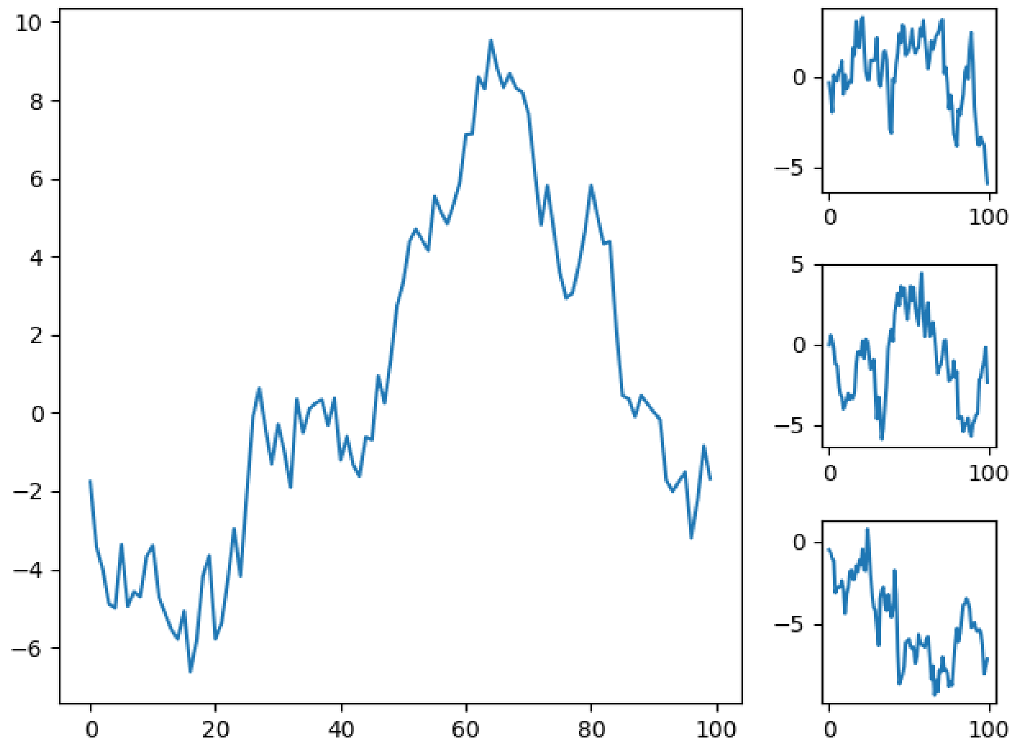

Figure 3.39: GridSpec

Next, we will implement an activity to implement **`GridSpec`**.

ACTIVITY 3.06: CREATING A SCATTER PLOT WITH MARGINAL HISTOGRAMS

In this activity, we will make use of **GridSpec** to visualize a **scatter plot** with **marginal histograms**. Let's look at the following scenario: you are given a dataset containing information about various animals. Visualize the correlation between the various animal attributes using scatter plots and marginal histograms.

The following are the steps to perform:

1. Import the necessary modules and enable plotting within a Jupyter Notebook.
2. Filter the data so that you end up with samples containing a body mass and maximum longevity as the given dataset, **`AnAge`**, which was used in the previous exercise, is not complete. Select all of the samples of the **`Aves`** class with a body mass of less than 20,000.

3. Create a Figure with a constrained layout. Create a gridspec of size 4x4. Create a scatter plot of size 3x3 and marginal histograms of size 1x3 and 3x1. Add labels and a Figure title.

   After executing the preceding steps, the expected output should be as follows:

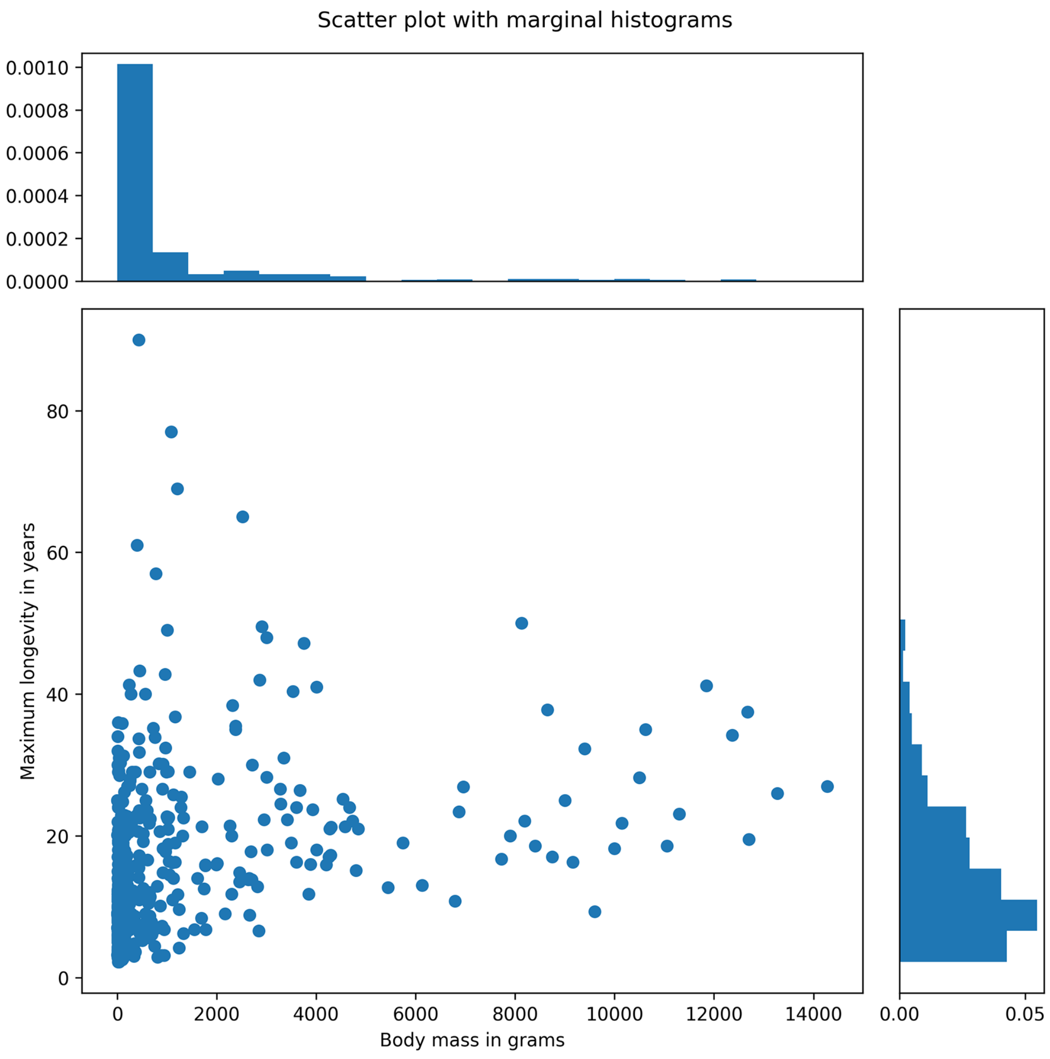

Figure 3.40: Scatter plots with marginal histograms

NOTE

The solution for this activity can be found via this link.

Next, we will learn how to work with image data in our visualizations.

Images

If you want to include images in your visualizations or work with image data, Matplotlib offers several functions for you. In this section, we will show you how to **load**, **save**, and **plot** images with Matplotlib.

> **NOTE**
>
> The images that are used in this section are sourced from https://unsplash.com/.

Basic Image Operations

The following are the basic operations for designing an image.

Loading Images

If you encounter image formats that are not supported by Matplotlib, we recommend using the **Pillow** library to load the image. In Matplotlib, loading images is part of the **`image`** submodule. We use the alias **`mpimg`** for the submodule, as follows:

```
import matplotlib.image as mpimg
```

The **`mpimg.imread(fname)`** reads an image and returns it as a **`numpy.array`** object. For grayscale images, the returned array has a shape (height, width), for RGB images (height, width, 3), and for RGBA images (height, width, 4). The array values range from 0 to 255.

We can also load the image in the following manner:

```
img_filenames = os.listdir('../../Datasets/images')
imgs = \
[mpimg.imread(os.path.join('../../Datasets/images', \
                           img_filename)) \
                           for img_filename in img_filenames]
```

The **`os.listdir()`** method in Python is used to get the list of all files and directories in the specified directory and then the **`os.path.join()`** function is used to join one or more path components intelligently.

Saving Images

The **`mpimg.imsave(fname, array)`** saves a **`numpy.array`** object as an image file. If the **`format`** parameter is not given, the format is deduced from the filename extension. With the optional parameters **`vmin`** and **`vmax`**, the color limits can be set manually. For a grayscale image, the default for the optional parameter, **`cmap`**, is **`'viridis'`**; you might want to change it to **`'gray'`**.

Plotting a Single Image

The **`plt.imshow(img)`** displays an image and returns an **`AxesImage`** object. For grayscale images with shape (height, width), the image array is visualized using a colormap. The default colormap is **`'viridis'`**, as illustrated in *Figure 3.41*. To actually visualize a grayscale image, the colormap has to be set to **`'gray'`** (that is, **`plt.imshow(img, cmap='gray')`**, which is illustrated in *Figure 3.42*. Values for grayscale, RGB, and RGBA images can be either **`float`** or **`uint8`**, and range from **`[0…1]`** or **`[0…255]`**, respectively. To manually define the value range, the parameters **`vmin`** and **`vmax`** must be specified. A visualization of an RGB image is shown in the following figures:

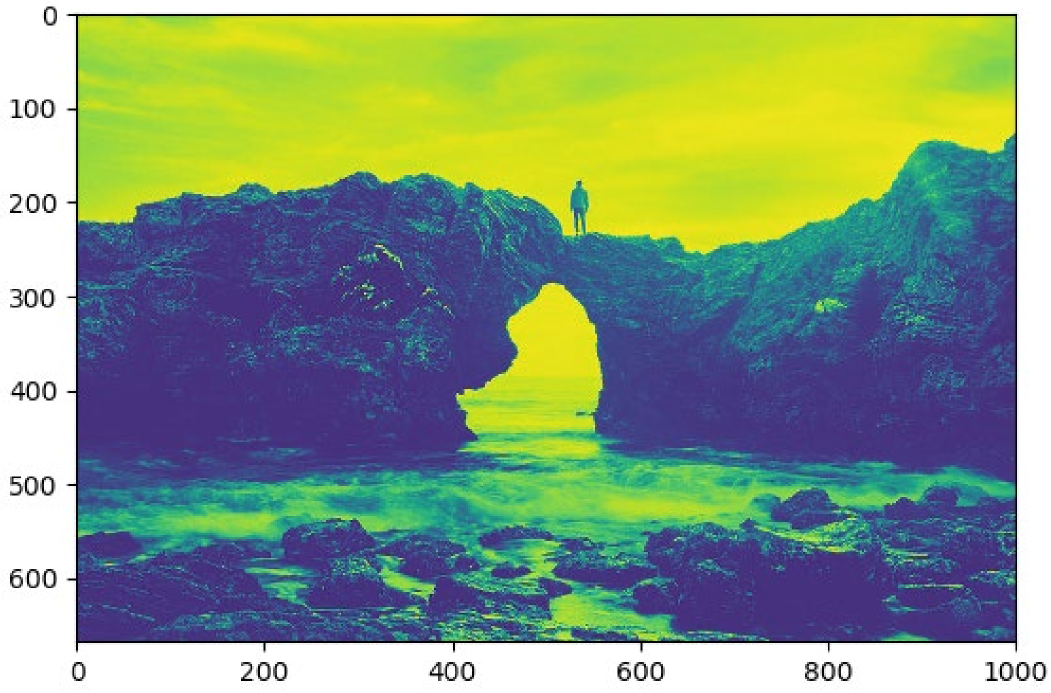

Figure 3.41: Grayscale image with a default viridis colormap

The following figure shows a grayscale image with a gray colormap:

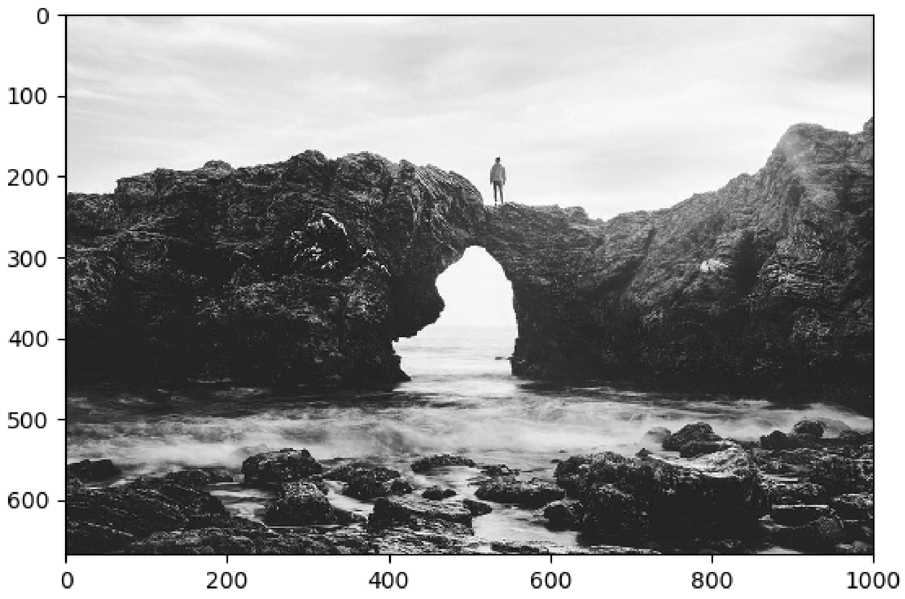

Figure 3.42: Grayscale image with a gray colormap

The following figure shows an RGB image:

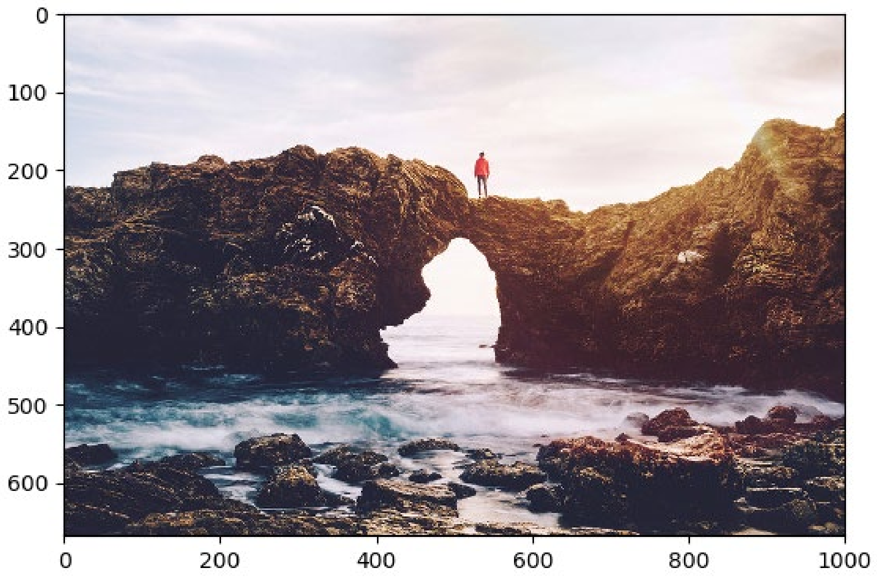

Figure 3.43: RGB image

Sometimes, it might be helpful to get an insight into the color values. We can simply add a color bar to the image plot. It is recommended to use a colormap with high contrast—for example, **`jet`**:

```
plt.imshow(img, cmap='jet')
plt.colorbar()
```

The preceding example is illustrated in the following figure:

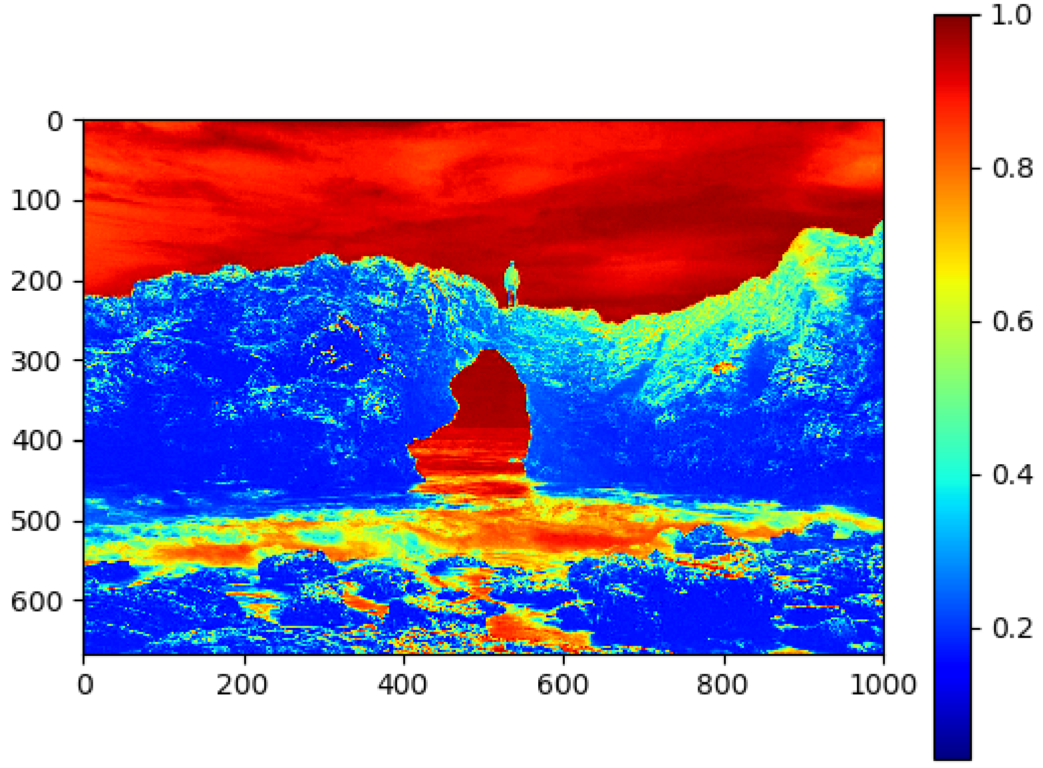

Figure 3.44: Image with a jet colormap and color bar

Another way to get insight into the image values is to plot a histogram, as shown in the following diagram. To plot the histogram for an image array, the array has to be flattened using **`numpy.ravel`**:

```
plt.hist(img.ravel(), bins=256, range=(0, 1))
```

The following diagram shows the output of the preceding code:

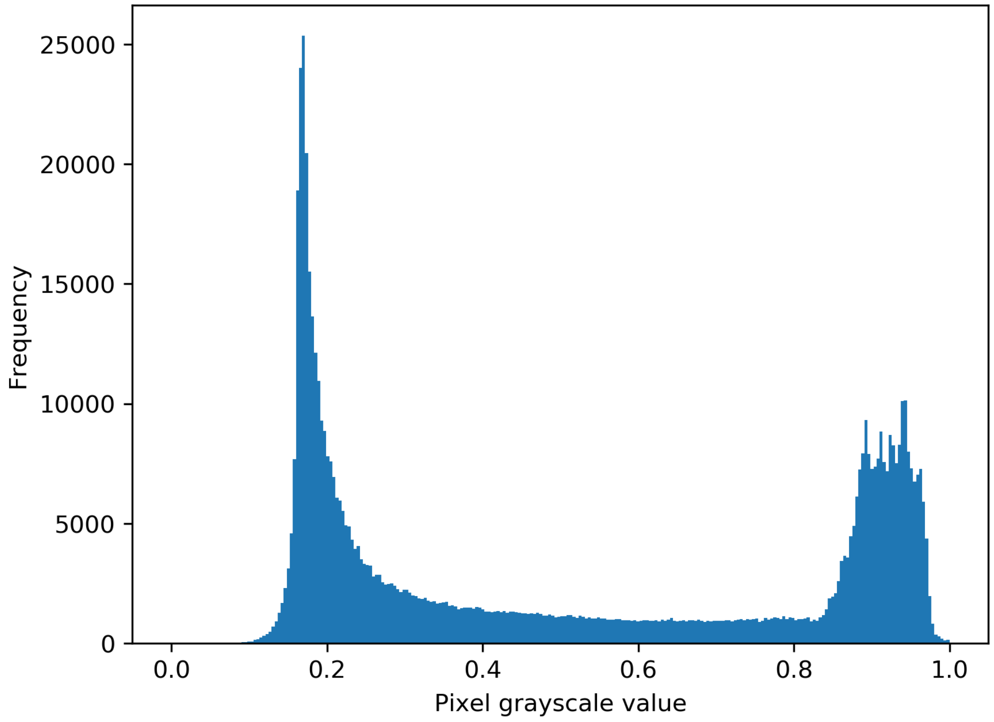

Figure 3.45: Histogram of image values

Plotting Multiple Images in a Grid

To plot multiple images in a grid, we can simply use **`plt.subplots`** and plot an image per **`Axes`**:

```
fig, axes = plt.subplots(1, 2)
for i in range(2):
    axes[i].imshow(imgs[i])
```

The result of the preceding code is shown in the following diagram:

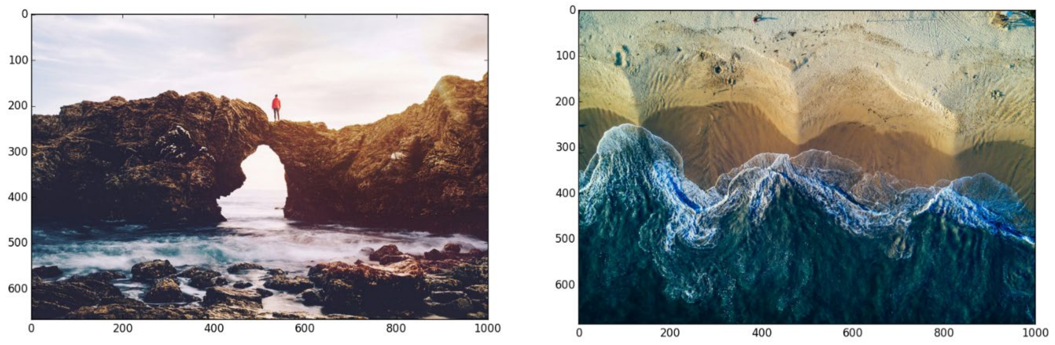

Figure 3.46: Multiple images within a grid

In some situations, it would be neat to remove the ticks and add labels. **`axes.set_xticks([])`** and **`axes.set_yticks([])`** remove x-ticks and y-ticks, respectively. **`axes.set_xlabel('label')`** adds a label:

```
fig, axes = plt.subplots(1, 2)
labels = ['coast', 'beach']
for i in range(2):
    axes[i].imshow(imgs[i])
    axes[i].set_xticks([])
    axes[i].set_yticks([])
    axes[i].set_xlabel(labels[i])
```

The result of the preceding code is shown in the following diagram:

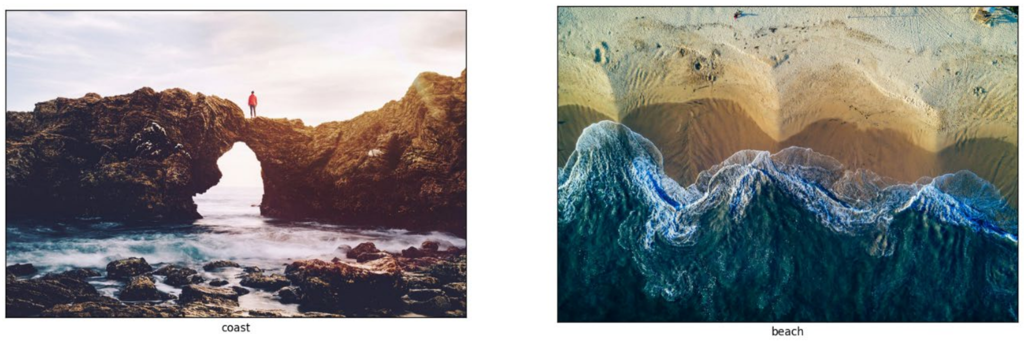

Figure 3.47: Multiple images with labels

Let's go through an activity for grid images.

ACTIVITY 3.07: PLOTTING MULTIPLE IMAGES IN A GRID

In this activity, we will plot images in a grid. You are a developer in a social media company. Management has decided to add a feature that helps the customer to upload images in a 2x2 grid format. Develop some standard code to generate grid-formatted images and add this new feature to your company's website.

The following are the steps to perform:

1. Import the necessary modules and enable plotting within a Jupyter Notebook.
2. Load all four images from the `Datasets` subfolder.
3. Visualize the images in a 2x2 grid. Remove the axes and give each image a label.

   After executing the preceding steps, the expected output should be as follows:

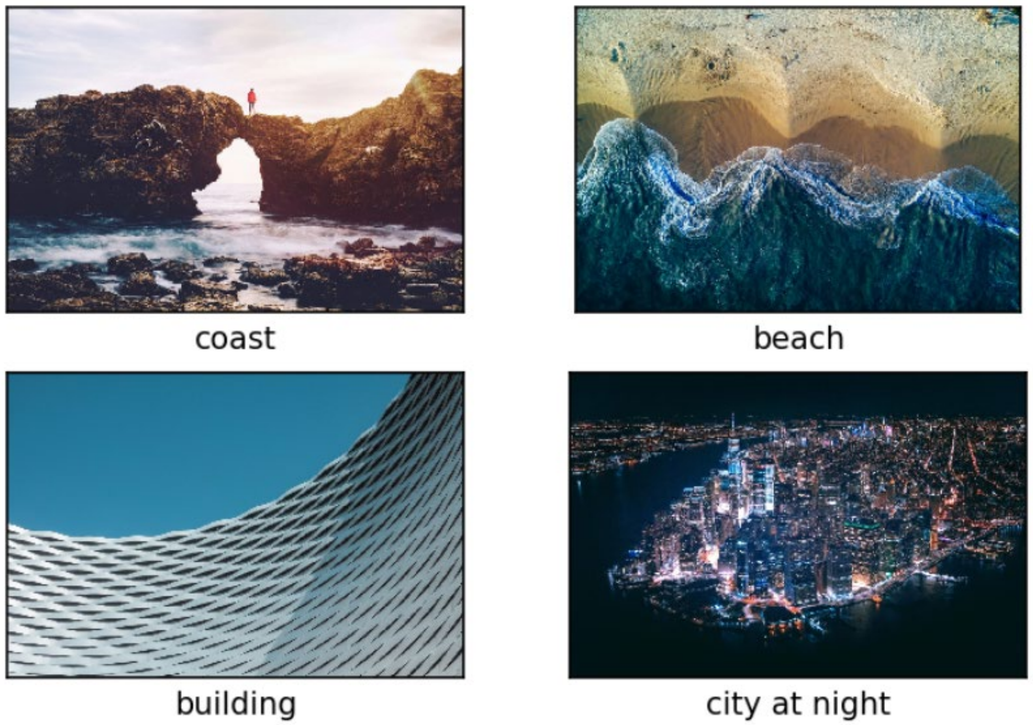

Figure 3.48: Visualizing images in a 2x2 grid

NOTE

The solution for this activity can be found via this link.

In this activity, we have plotted images in a 2x2 grid. In the next section, we will learn the basics of how to write and plot a mathematical expression.

WRITING MATHEMATICAL EXPRESSIONS

In case you need to write mathematical expressions within the code, Matplotlib supports **TeX**, one of the most popular typesetting systems, especially for typesetting mathematical formulas. You can use it in any text by placing your mathematical expression in a pair of dollar signs. There is no need to have TeX installed since Matplotlib comes with its own parser.

An example of this is given in the following code:

```
plt.xlabel(,$x$')
plt.ylabel(,$\cos(x)$')
```

The following diagram shows the output of the preceding code:

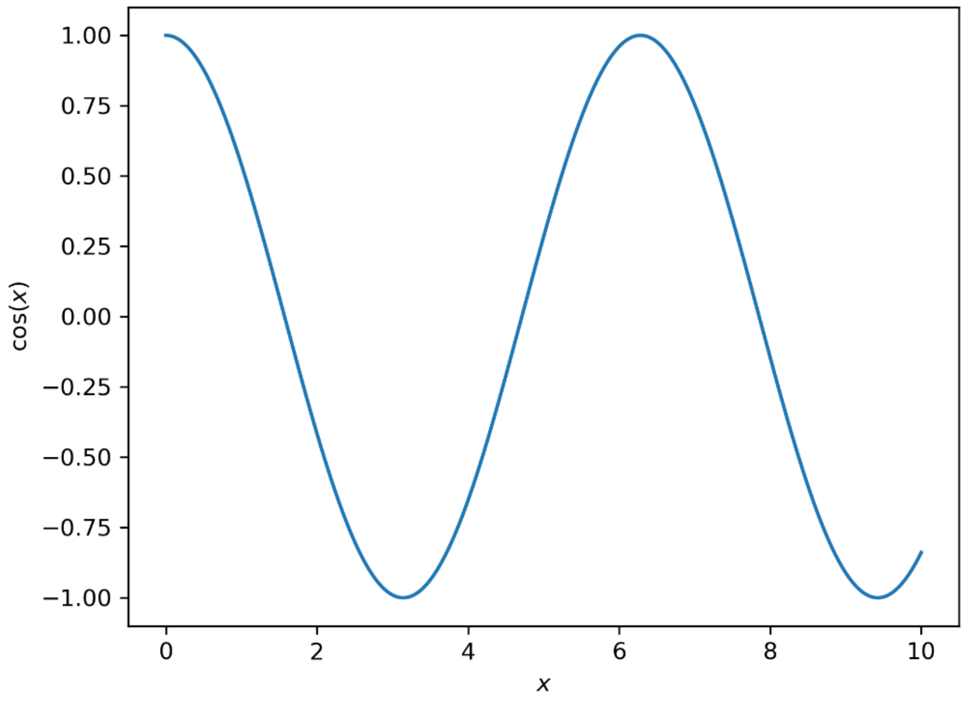

Figure 3.49: Diagram demonstrating mathematical expressions

TeX examples:

- **`'$\alpha_i>\beta_i$'`** produces $\alpha_i > \beta_i$.
- **`'$\sum_{i=0}^\infty x_i$'`** produces $\sum_{i=0}^{\infty} x_i$.
- **`'$\sqrt[3]{8}$'`** produces $\sqrt[3]{8}$.
- **`'$\frac{3 - \frac{x}{2}}{5}$'`** produces $\frac{3 - \frac{x}{2}}{5}$.

In this section, we learned how to write a basic mathematical expression and generate a plot using it.

SUMMARY

In this chapter, we provided a detailed introduction to Matplotlib, one of the most popular visualization libraries for Python. We started off with the basics of pyplot and its operations, and then followed up with a deep insight into the numerous possibilities that help to enrich visualizations with text. Using practical examples, this chapter covered the most popular plotting functions that Matplotlib offers, including comparison charts, and composition and distribution plots. It concluded with how to visualize images and write mathematical expressions.

In the next chapter, we will learn about the Seaborn library. Seaborn is built on top of Matplotlib and provides a higher-level abstraction to create visualizations in an easier way. One neat feature of Seaborn is the easy integration of DataFrames from the pandas library. Furthermore, Seaborn offers a few more plots out of the box, including more advanced visualizations, such as violin plots.