CHAPTER

4

Data Visualization Using matplotlib

What Is matplotlib?

As the adage goes, "A picture is worth a thousand words." This is probably most true in the world of machine learning. No matter how large or how small your dataset, it is often very useful (and many times, essential) that you are able to visualize the data and see the relationships between the various features within it. For example, given a dataset containing a group of students with their family details (such as examination results, family income, educational background of parents, and so forth), you might want to establish a relationship between the students' results with their family income. The best way to do this would be to plot a chart displaying the related data. Once the chart is plotted, you can then use it to draw your own conclusions and determine whether the results have a positive relationship to family income.

In Python, one of the most commonly used tools for plotting is matplotlib. *Matplotlib* is a Python 2D plotting library that you can use to produce publication-quality charts and figures. Using matplotlib, complex charts and figures can be generated with ease, and its integration with Jupyter Notebook makes it an ideal tool for machine learning.

In this chapter, you will learn the basics of matplotlib. In addition, you will also learn about Seaborn, a complementary data visualization library that is based on matplotlib.

Python® Machine Learning, First Edition. Wei-Meng Lee.
© 2019 John Wiley & Sons, Inc. Published 2019 by John Wiley & Sons, Inc.

Plotting Line Charts

To see how easy it is to use matplotlib, let's plot a line chart using Jupyter Notebook. Here is a code snippet that plots a line chart:

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2]
)
```

Figure 4.1 shows the line chart plotted.

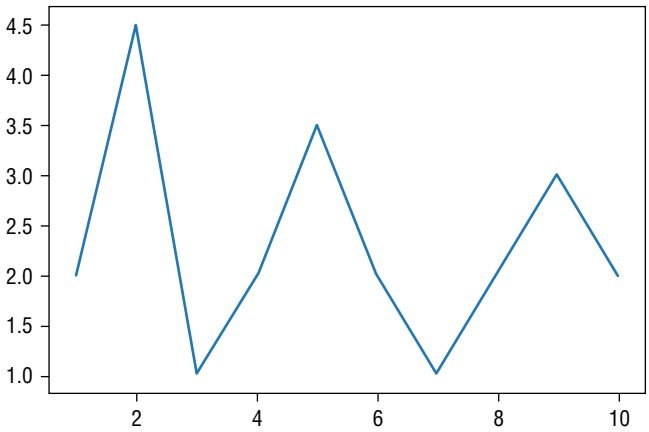

Figure 4.1: A line graph plotted using matplotlib

The first statement tells matplotlib to display the output of the plotting commands in line within front-ends likes Jupyter Notebook. In short, it means display the chart within the same page as your Jupyter Notebook:

```
%matplotlib inline
```

To use matplotlib, you import the `pyplot` module and name it *`plt`* (its commonly used alias):

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

To plot a line chart, you use the `plot()` function from the `pyplot` module, supplying it with two arguments as follows:

1. A list of values representing the x-axis
2. A list of values representing the y-axis

```
[1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2]
```

That's it. The chart will be shown in your Jupyter Notebook when you run it.

Adding Title and Labels

A chart without title and labels does not convey meaningful information. Matplotlib allows you to add a title and labels to the axes using the `title()`, `xlabel()`, and `ylabel()` functions as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2]
)
plt.title("Results")      # sets the title for the chart
plt.xlabel("Semester")    # sets the label to use for the x-axis
plt.ylabel("Grade")       # sets the label to use for the y-axis
```

Figure 4.2 shows the chart with the title, as well as the labels for the x- and y-axes.

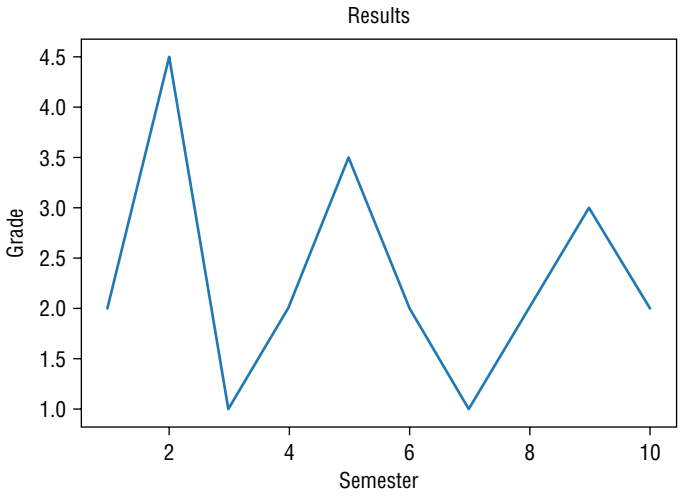

Figure 4.2: The line chart with the title and the labels for the x- and y-axes added

Styling

Matplotlib lets you adjust every aspect of your plot and create beautiful charts. However, it is very time consuming to create really beautiful charts and plots. To help with this, matplotlib ships with a number of predefined styles. Styles

allow you to create professional-looking charts using a predefined look-and-feel without requiring you to customize each element of the chart individually.

The following example uses the `ggplot` style, based on a popular data visualization package for the statistical programming language R:

TIP **The "gg" in ggplot comes from Leland Wilkinson's landmark 1999 book, *The Grammar of Graphics: Statistics and Computing*, (Springer, 2005).**

```
%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import style
style.use("ggplot")

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2]
)
plt.title("Results")      # sets the title for the chart
plt.xlabel("Semester")    # sets the label to use for the x-axis
plt.ylabel("Grade")       # sets the label to use for the y-axis
```

The chart styled using `ggplot` is shown in Figure 4.3.

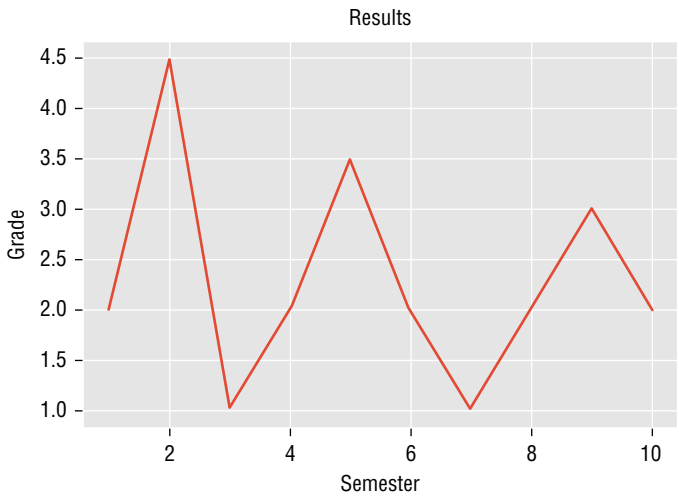

Figure 4.3: The chart with the ggplot style applied to it

Figure 4.4 shows the same chart with the `grayscale` styled applied.

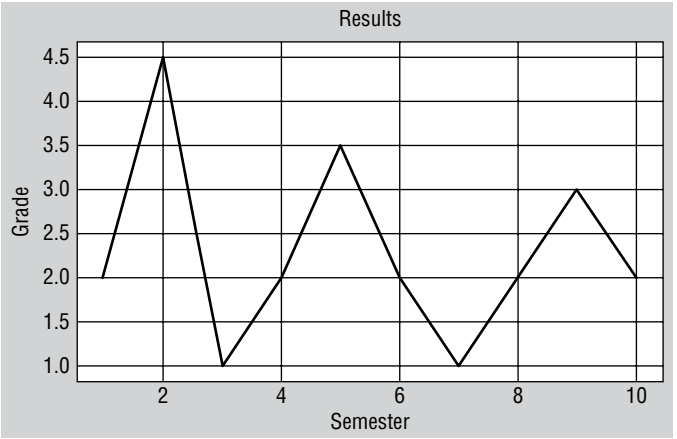

Figure 4.4: The chart with the grayscale style applied to it

You can use the `style.available` property to see the list of styles supported:

```
print(style.available)
```

Here is a sample output:

```
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',
'seaborn-whitegrid', 'classic', '_classic_test', 'fast', 'seaborn-talk',
'seaborn-dark-palette', 'seaborn-bright', 'seaborn-pastel', 'grayscale',
'seaborn-notebook', 'ggplot', 'seaborn-colorblind', 'seaborn-muted',
'seaborn', 'Solarize_Light2', 'seaborn-paper', 'bmh', 'seaborn-white',
'dark_background', 'seaborn-poster', 'seaborn-deep']
```

Plotting Multiple Lines in the Same Chart

You can plot multiple lines in the same chart by calling the `plot()` function one more time, as the following example shows:

```
%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import style
style.use("ggplot")

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2]
)

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [3,4,2,5,2,4,2.5,4,3.5,3]
)
```

```
plt.title("Results")      # sets the title for the chart
plt.xlabel("Semester")    # sets the label to use for the x-axis
plt.ylabel("Grade")       # sets the label to use for the y-axis
```

Figure 4.5 shows the chart now containing two line graphs.

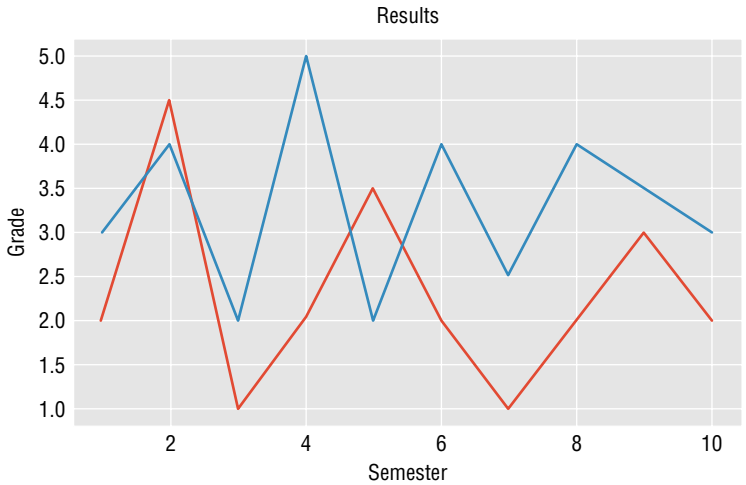

Figure 4.5: The chart with two line graphs

Adding a Legend

As you add more lines to a chart, it becomes more important to have a way to distinguish between the lines. Here is where a legend is useful. Using the previous example, you can add a label to each line plot and then show a legend using the `legend()` function as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import style
style.use("ggplot")

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2],
    label="Jim"
)

plt.plot(
    [1,2,3,4,5,6,7,8,9,10],
    [3,4,2,5,2,4,2.5,4,3.5,3],
    label="Tom"
)
```

```
plt.title("Results")      # sets the title for the chart
plt.xlabel("Semester")    # sets the label to use for the x-axis
plt.ylabel("Grade")       # sets the label to use for the y-axis
plt.legend()
```

Figure 4.6 shows the chart with a legend displayed.

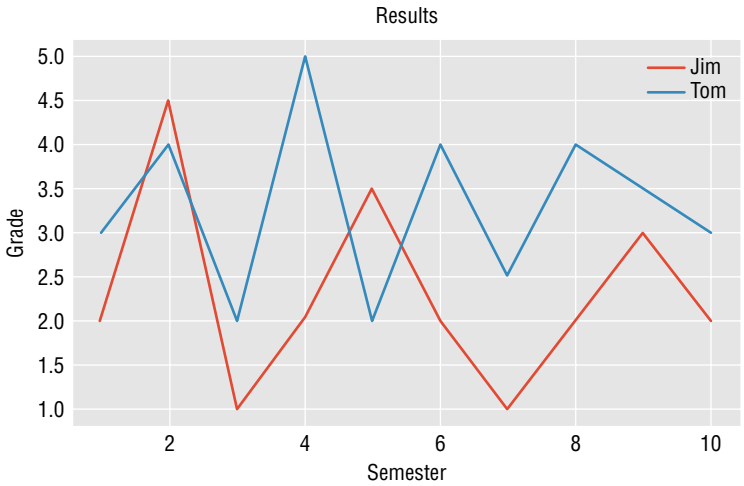

Figure 4.6: The chart with a legend displayed

Plotting Bar Charts

Besides plotting line charts, you can also plot bar charts using matplotlib. *Bar charts* are useful for comparing data. For example, you want to be able to compare the grades of a student over a number of semesters.

Using the same dataset that you used in the previous section, you can plot a bar chart using the `bar()` function as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import style

style.use("ggplot")

plt.bar(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2],
    label = "Jim",
    color = "m",                    # m for magenta
    align = "center"
)
```

```
plt.title("Results")
plt.xlabel("Semester")
plt.ylabel("Grade")

plt.legend()
plt.grid(True, color="y")
```

Figure 4.7 shows the bar chart plotted using the preceding code snippet.

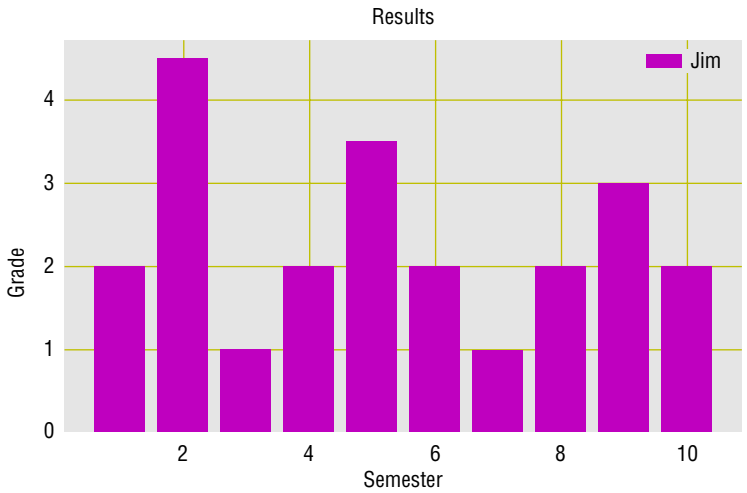

Figure 4.7: Plotting a bar chart

Adding Another Bar to the Chart

Just like adding an additional line chart to the chart, you can add another bar graph to an existing chart. The following statements in bold do just that:

```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import style

style.use("ggplot")

plt.bar(
    [1,2,3,4,5,6,7,8,9,10],
    [2,4.5,1,2,3.5,2,1,2,3,2],
    label = "Jim",
    color = "m",                    # for magenta
    align = "center",
    alpha = 0.5
)
```

```
plt.bar(
     [1,2,3,4,5,6,7,8,9,10],
     [1.2,4.1,0.3,4,5.5,4.7,4.8,5.2,1,1.1],
     label = "Tim",
     color = "g",                    # for green
     align = "center",
     alpha = 0.5
)


plt.title("Results")
plt.xlabel("Semester")
plt.ylabel("Grade")

plt.legend()
plt.grid(True, color="y")
```

Because the bars might overlap each with other, it is important to be able to distinguish them by setting their alpha to 0.5 (making them translucent). Figure 4.8 shows the two bar graphs in the same chart.

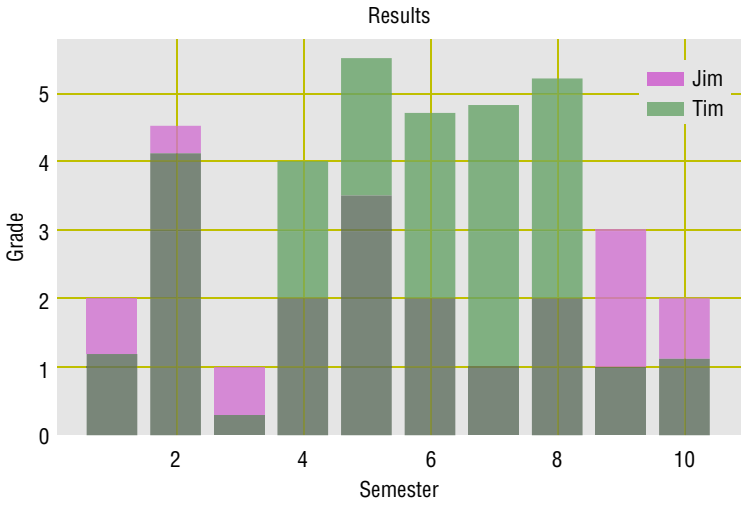

Figure 4.8: Plotting two overlapping bar charts on the same figure

Changing the Tick Marks

So far in our charts, the tick marks on the x-axis always displays the value that was supplied (such as 2, 4, 6, and so on). But what if your x-axis label is in the form of strings like this?

```
rainfall = [17,9,16,3,21,7,8,4,6,21,4,1]
months = ['Jan','Feb','Mar','Apr','May','Jun',
          'Jul','Aug','Sep','Oct','Nov','Dec']
```

In this case, you might be tempted to plot the chart directly as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt

rainfall = [17,9,16,3,21,7,8,4,6,21,4,1]
months = ['Jan','Feb','Mar','Apr','May','Jun',
          'Jul','Aug','Sep','Oct','Nov','Dec']

plt.bar(months, rainfall, align='center', color='orange' )
plt.show()
```

The preceding code snippet will create the chart shown in Figure 4.9.

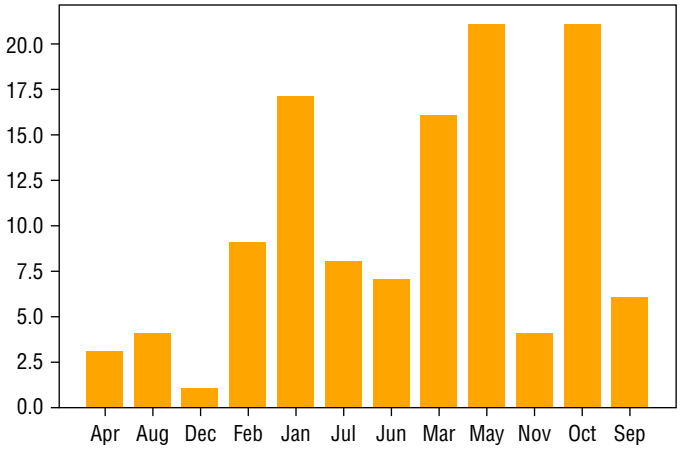

Figure 4.9: The bar chart with the alphabetically arranged x-axis

Look carefully at the x-axis: the labels have been sorted alphabetically, and hence the chart does not show the amount of rainfall from Jan to Dec in the correct order. To fix this, create a `range` object matching the size of the rainfall list, and use it to plot the chart. To ensure that the month labels are displayed correctly on the x-axis, use the `xticks()` function:

```
%matplotlib inline
import matplotlib.pyplot as plt

rainfall = [17,9,16,3,21,7,8,4,6,21,4,1]
months = ['Jan','Feb','Mar','Apr','May','Jun',
          'Jul','Aug','Sep','Oct','Nov','Dec']

plt.bar(range(len(rainfall)), rainfall, align='center', color='orange' )
plt.xticks(range(len(rainfall)), months, rotation='vertical')
plt.show()
```

The `xticks()` function sets the tick labels on the x-axis, as well the positioning of the ticks. In this case, the labels are displayed vertically, as shown in Figure 4.10.

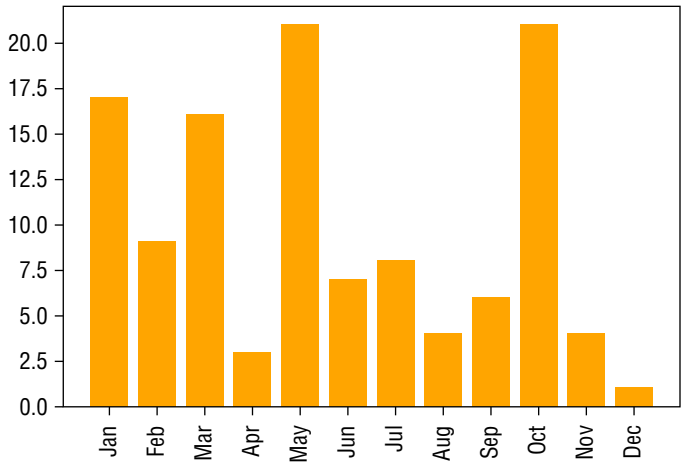

Figure 4.10: The bar chart with the correct x-axis

Plotting Pie Charts

Another chart that is popular is the pie chart. A *pie chart* is a circular statistical graphic divided into slices to illustrate numerical proportions. A pie chart is useful when showing percentage or proportions of data. Consider the following sets of data representing the various browser market shares:

```
labels      = ["Chrome", "Internet Explorer", "Firefox",
               "Edge","Safari","Sogou Explorer","Opera","Others"]
marketshare = [61.64, 11.98, 11.02, 4.23, 3.79, 1.63, 1.52, 4.19]
```

In this case, it would be really beneficial to be able to represent the total market shares as a complete circle, with each slice representing the percentage held by each browser.

The following code snippet shows how you can plot a pie chart using the data that we have:

```
%matplotlib inline
import matplotlib.pyplot as plt

labels      = ["Chrome", "Internet Explorer",
               "Firefox", "Edge","Safari",
               "Sogou Explorer","Opera","Others"]
```

```
marketshare = [61.64, 11.98, 11.02, 4.23, 3.79, 1.63, 1.52, 4.19]
explode     = (0,0,0,0,0,0,0,0)

plt.pie(marketshare,
        explode = explode,  # fraction of the radius with which to
                            # offset each wedge
        labels = labels,
        autopct="%.1f%%",   # string or function used to label the
                            # wedges with their numeric value
        shadow=True,
        startangle=45)      # rotates the start of the pie chart by
                            # angle degrees counterclockwise from the
                            # x-axis

plt.axis("equal")           # turns off the axis lines and labels
plt.title("Web Browser Marketshare - 2018")
plt.show()
```

Figure 4.11 shows the pie chart plotted. Note that matplotlib will decide on the colors to use for each of the slices in the pie chart.

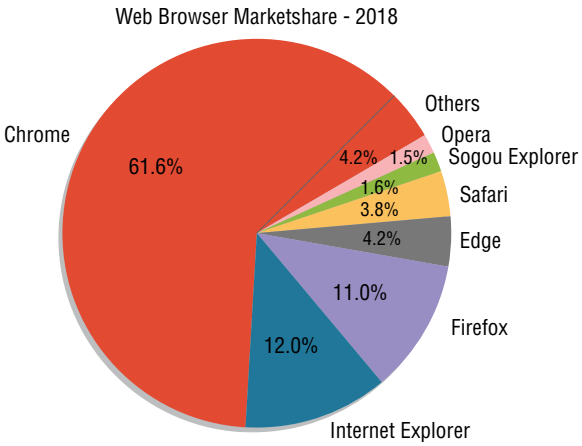

Figure 4.11: Plotting a pie chart

Exploding the Slices

The *explode parameter* specifies the fraction of the radius with which to offset each wedge. In the preceding example, we have set the `explode` parameter to all zeros:

```
explode     = (0,0,0,0,0,0,0,0)
```

Say that we need to highlight the market share of the Firefox and Safari browsers. In that case, we could modify the `explode` list as follows:

```
explode     = (0,0,0.5,0,0.8,0,0,0)
```

Refreshing the chart, you will see the two slices exploding (separating) from the main pie (see Figure 4.12).

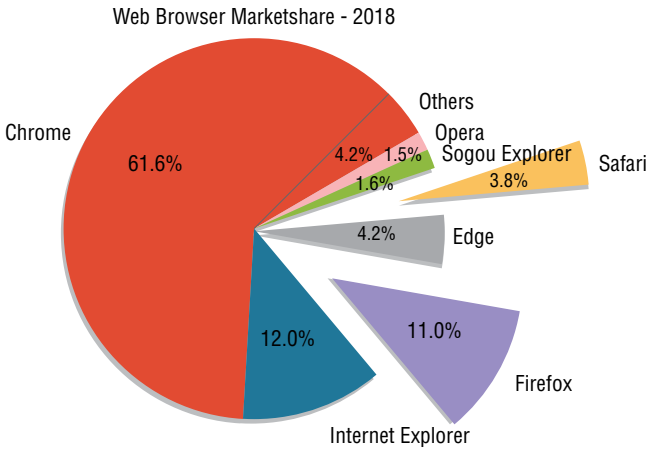

Figure 4.12: The pie chart with two exploded slices

Displaying Custom Colors

By default, matplotlib will decide on the colors to use for each of the slices in the pie chart. Sometimes the colors that are selected may not appeal to you. But you can certainly customize the chart to display using your desired colors.

You can create a list of colors and then pass it to the `colors` parameter:

```
%matplotlib inline
import matplotlib.pyplot as plt

labels      = ["Chrome", "Internet Explorer",
               "Firefox", "Edge","Safari",
               "Sogou Explorer","Opera","Others"]

marketshare = [61.64, 11.98, 11.02, 4.23, 3.79, 1.63, 1.52, 4.19]
explode     = (0,0,0.5,0,0.8,0,0,0)
colors      = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']

plt.pie(marketshare,
        explode = explode,  # fraction of the radius with which to
                            # offset each wedge
        labels = labels,
        colors = colors,
        autopct="%.1f%%",   # string or function used to label the
                            # wedges with their numeric value
        shadow=True,
```

```
        startangle=45)      # rotates the start of the pie chart by
                            # angle degrees counterclockwise from the
                            # x-axis
plt.axis("equal")           # turns off the axis lines and labels
plt.title("Web Browser Marketshare - 2018")
plt.show()
```

Since there are more slices than the colors you specified, the colors will be recycled. Figure 4.13 shows the pie chart with the new colors.

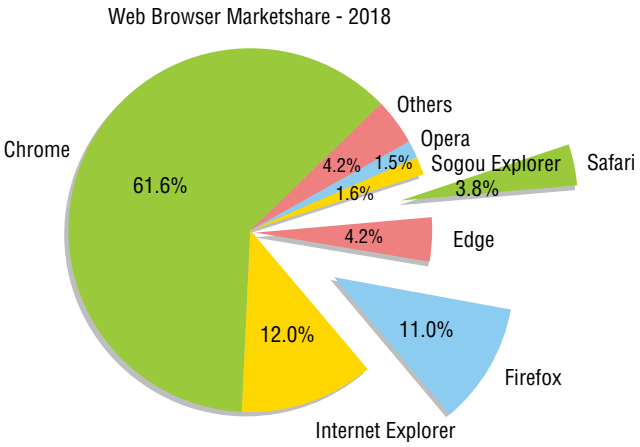

Figure 4.13: Displaying the pie chart with custom colors

Rotating the Pie Chart

Observe that we have set the `startangle` parameter to 45. This parameter specifies the degrees by which to rotate the start of the pie chart, counterclockwise from the x-axis. Figure 4.14 shows the effect of setting the `startangle` to 0 versus 45.

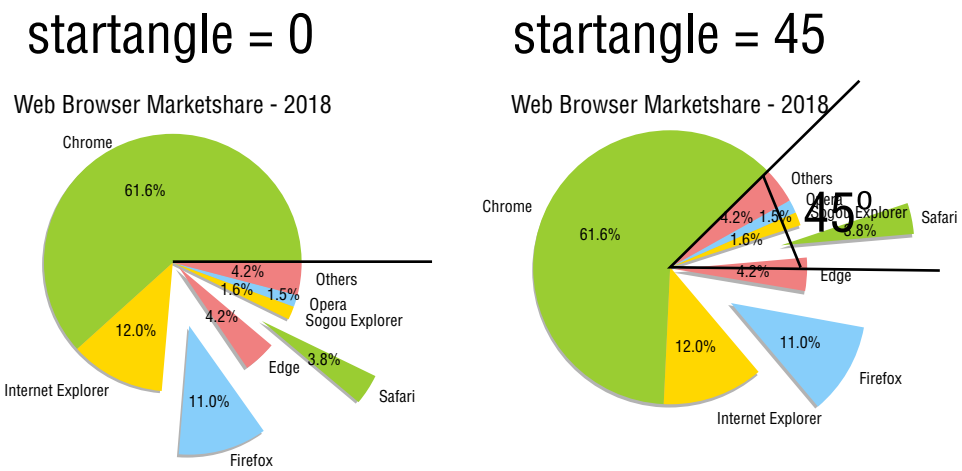

Figure 4.14: Setting the start angle for the pie chart

Displaying a Legend

Like the line and bar charts, you can also display a legend in your pie charts. But before you can do that, you need to handle the return values from the `pie()` function:

```
pie = plt.pie(marketshare,
        explode = explode,  # fraction of the radius with which to
                            # offset each wedge
        labels = labels,
        colors = colors,
        autopct="%.1f%%",   # string or function used to label the
                            # wedges with their numeric value
        shadow=True,
        startangle=45)      # rotates the start of the pie chart by
                            # angle degrees counterclockwise from the
                            # x-axis
```

The `pie()` function returns a tuple containing the following values:

`patches`: A list of `matplotlib.patches.Wedge` instances.
`texts`: A list of the label `matplotlib.text.Text` instances.
`autotexts`: A list of `Text` instances for the numeric labels. This will only be returned if the parameter `autopct` is not `None`.

To display the legend, use the `legend()` function as follows:

```
plt.axis("equal")           # turns off the axis lines and labels
plt.title("Web Browser Marketshare - 2018")
plt.legend(pie[0], labels, loc="best")
plt.show()
```

Figure 4.15 shows the legend displaying on the pie chart.

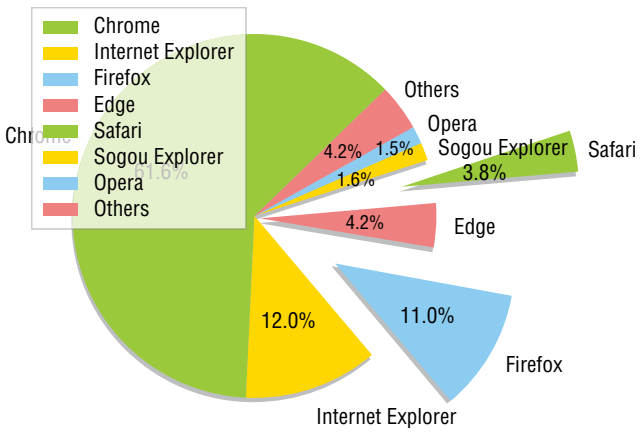

Figure 4.15: Displaying the legend on the pie chart

TIP **If the `autopct` parameter is not set to `None`, the `pie()` function returns the tuple (`patches`, `texts`, `autotexts`).**

The positioning of the legend can be modified through the `loc` parameter. It can take in either a string value or an integer value. Table 4.1 shows the various values that you can use for the `loc` parameter.

Table 4.1: Location Strings and Corresponding Location Codes

LOCATION STRING	LOCATION CODE
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

Saving the Chart

So far, you have been displaying the charts in a browser. At times, it is useful to be able to save the image to disk. You can do so using the `savefig()` function as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt

labels      = ["Chrome", "Internet Explorer",
               "Firefox", "Edge","Safari",
               "Sogou Explorer","Opera","Others"]

...
plt.axis("equal")           # turns off the axis lines and labels
plt.title("Web Browser Marketshare - 2018")
plt.savefig("Webbrowsers.png", bbox_inches="tight")
plt.show()
```

Setting the `bbox_inches` parameter to `tight` removes all of the extra white space around your figure.

Plotting Scatter Plots

A *scatter plot* is a two-dimensional chart that uses dots (or other shapes) to represent the values for two different variables. Scatter plots are often used to show how much the value of one variable is affected by another.

The following code snippet shows a scatter plot with the x-axis containing a list of numbers from 1 to 4, and the y-axis showing the cube of the x-axis values:

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot([1,2,3,4],         # x-axis
         [1,8,27,64],       # y-axis
         'bo')              # blue circle marker
plt.axis([0, 4.5, 0, 70])  # xmin, xmax, ymin, ymax
plt.show()
```

Figure 4.16 shows the scatter plot.

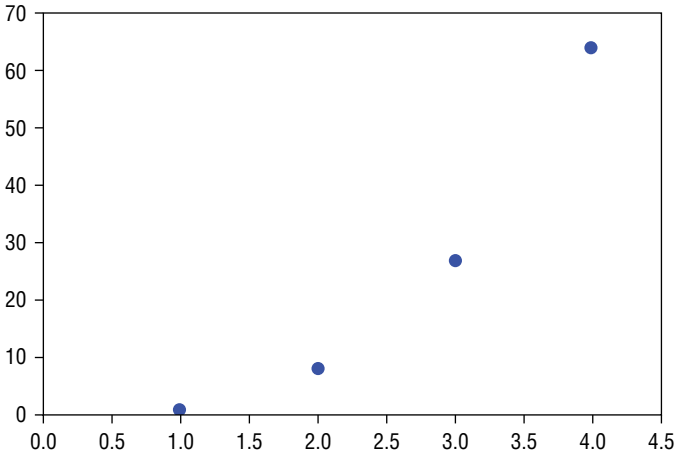

Figure 4.16: Plotting a scatter plot

Combining Plots

You can combine multiple scatter plots into one chart as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np

a = np.arange(1,4.5,0.1)    # 1.0, 1.1, 1.2, 1.3...4.4
plt.plot(a, a**2, 'y^',     # yellow triangle_up marker
```

```
          a, a**3, 'bo',     # blue circle
          a, a**4, 'r--',)   # red dashed line

plt.axis([0, 4.5, 0, 70])  # xmin, xmax, ymin, ymax
plt.show()
```

Figure 4.17 shows the chart displaying three scatter plots. You can customize the shape of the points to draw on the scatter plot. For example, `y^` indicates a yellow triangle-up marker, `bo` indicates a blue circle, and so on.

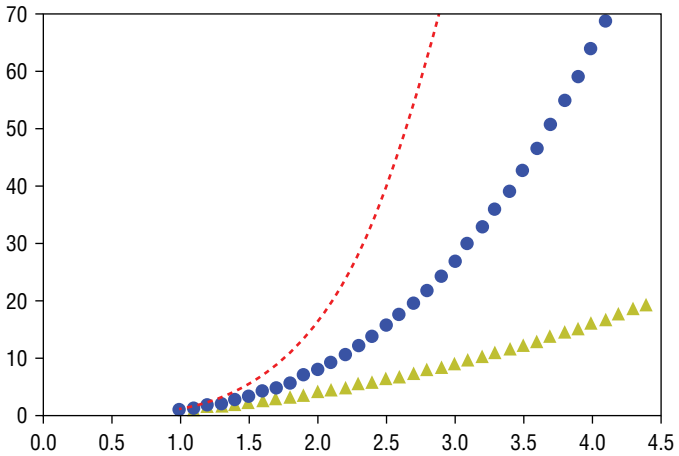

Figure 4.17: Combining multiple scatter plots into a single chart

Subplots

You can also plot multiple scatter plots separately and combine them into a single figure:

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

a = np.arange(1,5,0.1)

plt.subplot(121)            # 1 row, 2 cols, chart 1
plt.plot([1,2,3,4,5],
         [1,8,27,64,125],
         'y^')

plt.subplot(122)            # 1 row, 2 cols, chart 2
plt.plot(a, a**2, 'y^',
         a, a**3, 'bo',
         a, a**4, 'r--',)
```

```
plt.axis([0, 4.5, 0, 70])    # xmin, xmax, ymin, ymax
plt.show()
```

Figure 4.18 shows two charts displayed in a single figure.

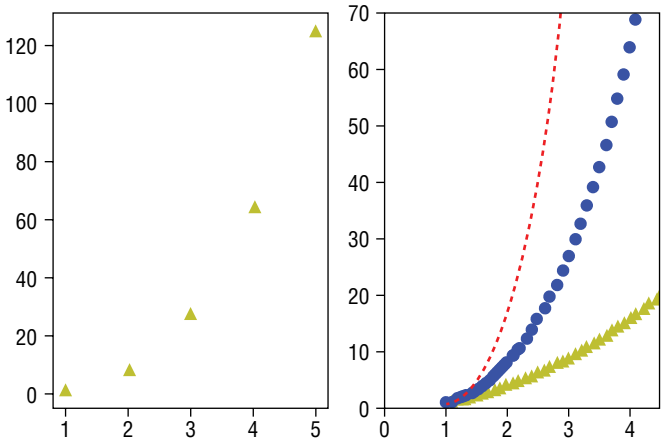

Figure 4.18: Combining two charts into a single figure

The `subplot()` function adds a subplot to the current figure. One of the arguments it takes in has the following format: *nrow,ncols,index*. In the preceding example, the 121 means "1 row, 2 columns, and chart 1." Using this format, you can have up to a maximum of nine figures. The `subplot()` function can also be called with the following syntax:

```
plt.subplot(1,2,1)              # 1 row, 2 cols, chart 1
```

Using this syntax, you can now have more than 10 charts in a single figure.

TIP **The `scatter()` function draws points without lines connecting them, whereas the `plot()` function may or may not plot the lines, depending on the arguments.**

Plotting Using Seaborn

While matplotlib allows you to plot a lot of interesting charts, it takes a bit of effort to get the chart that you want. This is especially true if you are dealing with a large amount of data and would like to examine the relationships between multiple variables.

Introducing *Seaborn*, a complementary plotting library that is based on the matplotlib data visualization library. Seaborn's strength lies in its ability to

make statistical graphics in Python, and it is closely integrated with the Pandas data structure (covered in Chapter 3). Seaborn provides high-level abstractions to allow you to build complex visualizations for your data easily. In short, you write less code with Seaborn than with matplotlib, while at the same time you get more sophisticated charts.

Displaying Categorical Plots

The first example that you will plot is called a categorical plot (formerly known as a factorplot). It is useful in cases when you want to plot the distribution of a certain group of data. Suppose that you have a CSV file named `drivinglicense.csv` containing the following data:

```
gender,group,license
men,A,1
men,A,0
men,A,1
women,A,1
women,A,0
women,A,0
men,B,0
men,B,0
men,B,0
men,B,1
women,B,1
women,B,1
women,B,1
women,B,1
```

This CSV file shows the distribution of men and women in two groups, A and B, with 1 indicating that the person has a driver's license and a 0 indicating no driver's license. If you are tasked with plotting a chart to show the proportion of men and women in each group that has a driver's license, you can use Seaborn's categorical plot.

First, import the relevant modules:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

Load the data into a Pandas dataframe:

```
#---load data---
data = pd.read_csv('drivinglicense.csv')
```

Call the `catplot()` function with the following arguments:

```
#---plot a factorplot---
g = sns.catplot(x="gender", y="license", col="group",
        data=data, kind="bar", ci=None, aspect=1.0)
```

You pass in the dataframe through the `data` parameter, and you specify the *`gender`* as the x-axis. The y-axis will tabulate the proportion of men and women who have a driver's license, and hence you set `y` to *`license`*. You want to separate the chart into two groups based on group, hence you set `col` to *`group`*.

Next, you set the labels on the chart:

```
#---set the labels---
g.set_axis_labels("", "Proportion with Driving license")
g.set_xticklabels(["Men", "Women"])
g.set_titles("{col_var} {col_name}")

#---show plot---
plt.show()
```

Figure 4.19 shows the categorical plot drawn by Seaborn. As you can see, 2/3 of the men and 1/3 of the women have driver's licenses in Group A, while in Group B, 1/4 of the men and all the women have driver's licenses. Neat, isn't it?

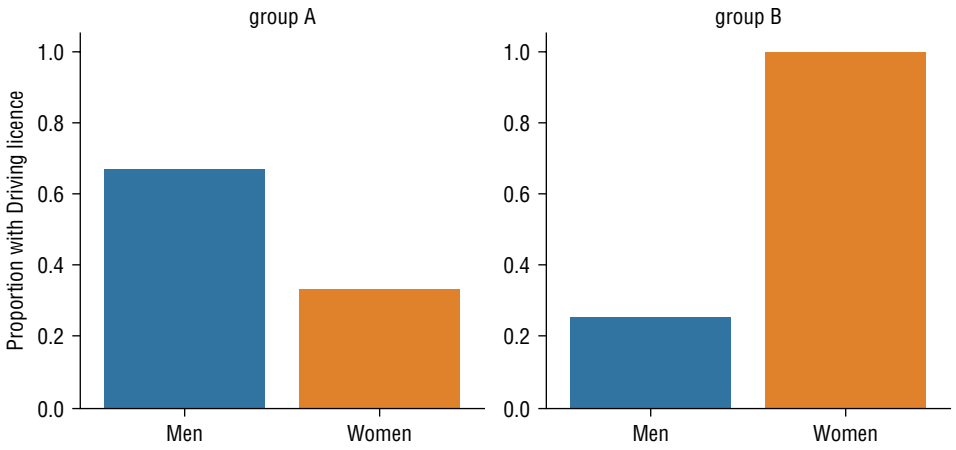

Figure 4.19: Displaying a factorplot showing the distribution of men and women who have driver's licenses in each group

Let's take a look at another example of catplot. Using the Titanic dataset, let's plot a chart and see what the survival rate of men, women, and children looks like in each of the three classes.

TIP **Seaborn has a built-in dataset that you can load directly using the `load_dataset()` function. To see the names of the dataset that you can load, use the `sns.get_dataset_names()` function. Alternatively, if you want to download the dataset for offline use, check out `https://github.com/mwaskom/seaborn-data`. Note that you would need to have an Internet connection, as the `load_dataset()` function loads the specified dataset from the online repository.**

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic = sns.load_dataset("titanic")
g = sns.catplot(x="who", y="survived", col="class",
        data=titanic, kind="bar", ci=None, aspect=1)

g.set_axis_labels("", "Survival Rate")
g.set_xticklabels(["Men", "Women", "Children"])
g.set_titles("{col_name} {col_var}")

#---show plot---
plt.show()
```

Figure 4.20 shows the distribution of the data based on classes. As you can see, both women and children have a higher chance of survival if they are in the first- and second-class cabins.

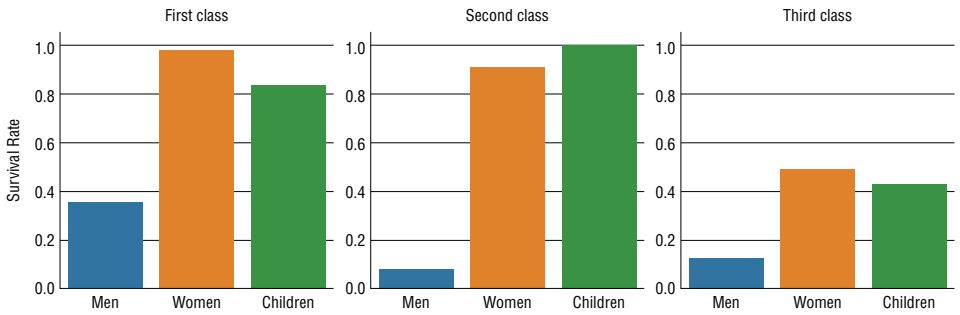

Figure 4.20: A factorplot showing the survival rate of men, women, and children in each of the cabin classes in the Titanic dataset

Displaying Lmplots

Another plot that is popular in Seaborn is the lmplot. An *lmplot* is a scatter plot. Using another built-in dataset from Seaborn, you can plot the relationships between the petal width and petal length of an iris plant and use it to determine the type of iris plants: setosa, versicolor, or virginica.

```
import seaborn as sns
import matplotlib.pyplot as plt

#---load the iris dataset---
iris = sns.load_dataset("iris")

#---plot the lmplot---
sns.lmplot('petal_width', 'petal_length', data=iris,
           hue='species', palette='Set1',
           fit_reg=False, scatter_kws={"s": 70})

#---get the current polar axes on the current figure---
ax = plt.gca()
ax.set_title("Plotting using the Iris dataset")

#---show the plot---
plt.show()
```

Figure 4.21 shows the scatter plot created using the `lmplot()` function.

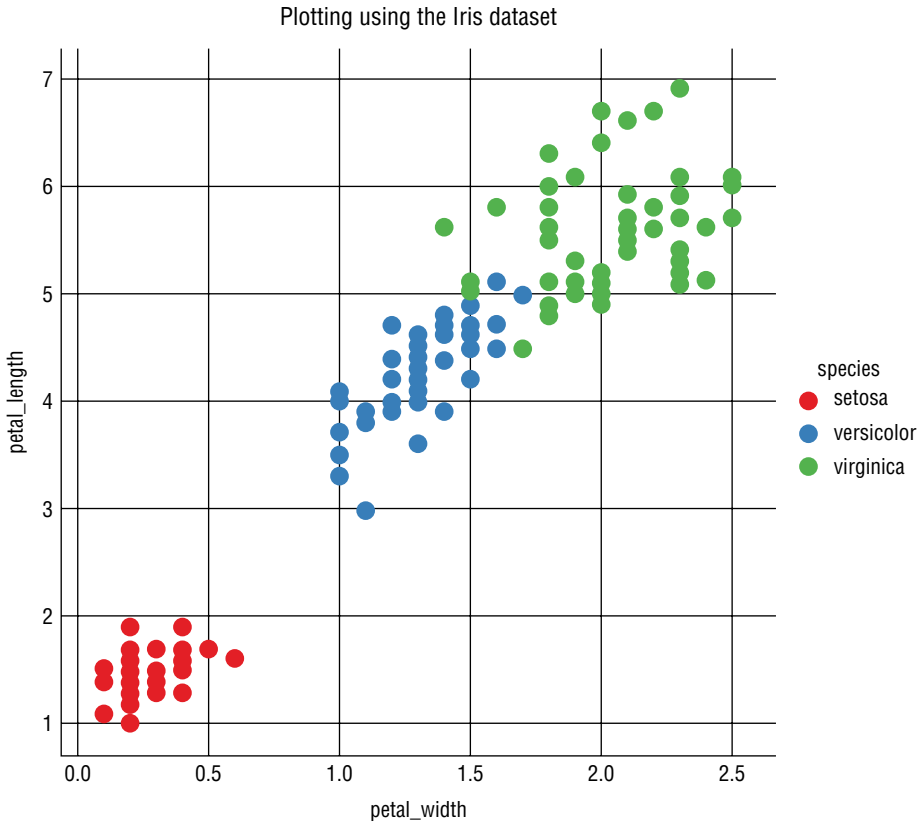

Figure 4.21: An lmplot showing the relationship between the petal length and width of the iris dataset

Displaying Swarmplots

A *swarmplot* is a categorical scatterplot with nonoverlapping points. It is useful for discovering the distribution of data points in a dataset. Consider the following CSV file named `salary.csv`, which contains the following content:

```
gender,salary
men,100000
men,120000
men,119000
men,77000
men,83000
men,120000
men,125000
women,30000
women,140000
women,38000
women,45000
women,23000
women,145000
women,170000
```

You want to show the distribution of salaries for men and women. In this case, a swarmplot is an ideal fit. The following code snippet does just that:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

sns.set_style("whitegrid")

#---load data---
data = pd.read_csv('salary.csv')

#---plot the swarm plot---
sns.swarmplot(x="gender", y="salary", data=data)

ax = plt.gca()
ax.set_title("Salary distribution")

#---show plot---
plt.show()
```

Figure 4.22 shows that, in this group, even though women have the highest salary, it also has the widest income disparity.

Figure 4.22: A swarmplot showing the distribution of salaries for men and women

Summary

In this chapter, you learned how to use matplotlib to plot the different types of charts that are useful for discovering patterns and relationships in a dataset. A complementary plotting library, Seaborn, simplifies plotting more sophisticated charts. While this chapter does not contain an exhaustive list of charts that you can plot with matplotlib and Seaborn, subsequent chapters will provide more samples and uses for them.