15

GENERATING DATA

Data visualization involves exploring data through visual representations. It's closely associated with *data analysis*, which uses code to explore the patterns and connections in a data set. A data set can be made up of a small list of numbers that fits in one line of code or it can be many gigabytes of data.

Making beautiful data representations is about more than pretty pictures. When a representation of a data set is simple and visually appealing, its meaning becomes clear to viewers. People will see patterns and significance in your data sets that they never knew existed.

Fortunately, you don't need a supercomputer to visualize complex data. With Python's efficiency, you can quickly explore data sets made of millions of individual data points on just a laptop. Also, the data points don't have to be numbers. With the basics you learned in the first part of this book, you can analyze nonnumerical data as well.

People use Python for data-intensive work in genetics, climate research, political and economic analysis, and much more. Data scientists have written an impressive array of visualization and analysis tools in Python, many of which are available to you as well. One of the most popular tools is Matplotlib, a mathematical plotting library. We'll use Matplotlib to make simple plots, such as line graphs and scatter plots. Then we'll create a more interesting data set based on the concept of a random walk—a visualization generated from a series of random decisions.

We'll also use a package called Plotly, which creates visualizations that work well on digital devices. Plotly generates visualizations that automatically resize to fit a variety of display devices. These visualizations can also include a number of interactive features, such as emphasizing particular aspects of the data set when users hover over different parts of the visualization. We'll use Plotly to analyze the results of rolling dice.

Installing Matplotlib

To use Matplotlib for your initial set of visualizations, you'll need to install it using `pip`, a module that downloads and installs Python packages. Enter the following command at a terminal prompt:

```
$ python -m pip install --user matplotlib
```

This command tells Python to run the `pip` module and install the `matplotlib` package to the current user's Python installation. If you use a command other than `python` on your system to run programs or start a terminal session, such as `python3`, your command will look like this:

```
$ python3 -m pip install --user matplotlib
```

NOTE

If this command doesn't work on macOS, try running the command again without the `--user` flag.

To see the kinds of visualizations you can make with Matplotlib, visit the sample gallery at *https://matplotlib.org/gallery/*. When you click a visualization in the gallery, you'll see the code used to generate the plot.

Plotting a Simple Line Graph

Let's plot a simple line graph using Matplotlib, and then customize it to create a more informative data visualization. We'll use the square number sequence 1, 4, 9, 16, 25 as the data for the graph.

Just provide Matplotlib with the numbers, as shown here, and Matplotlib should do the rest:

mpl_squares.py

```
  import matplotlib.pyplot as plt

  squares = [1, 4, 9, 16, 25]
❶ fig, ax = plt.subplots()
  ax.plot(squares)

  plt.show()
```

We first import the `pyplot` module using the alias `plt` so we don't have to type `pyplot` repeatedly. (You'll see this convention often in online examples, so we'll do the same here.) The `pyplot` module contains a number of functions that generate charts and plots.

We create a list called `squares` to hold the data that we'll plot. Then we follow another common Matplotlib convention by calling the `subplots()` function ❶. This function can generate one or more plots in the same figure. The variable `fig` represents the entire figure or collection of plots that are generated. The variable `ax` represents a single plot in the figure and is the variable we'll use most of the time.

We then use the `plot()` method, which will try to plot the data it's given in a meaningful way. The function `plt.show()` opens Matplotlib's viewer and displays the plot, as shown in Figure 15-1. The viewer allows you to zoom and navigate the plot, and when you click the disk icon, you can save any plot images you like.

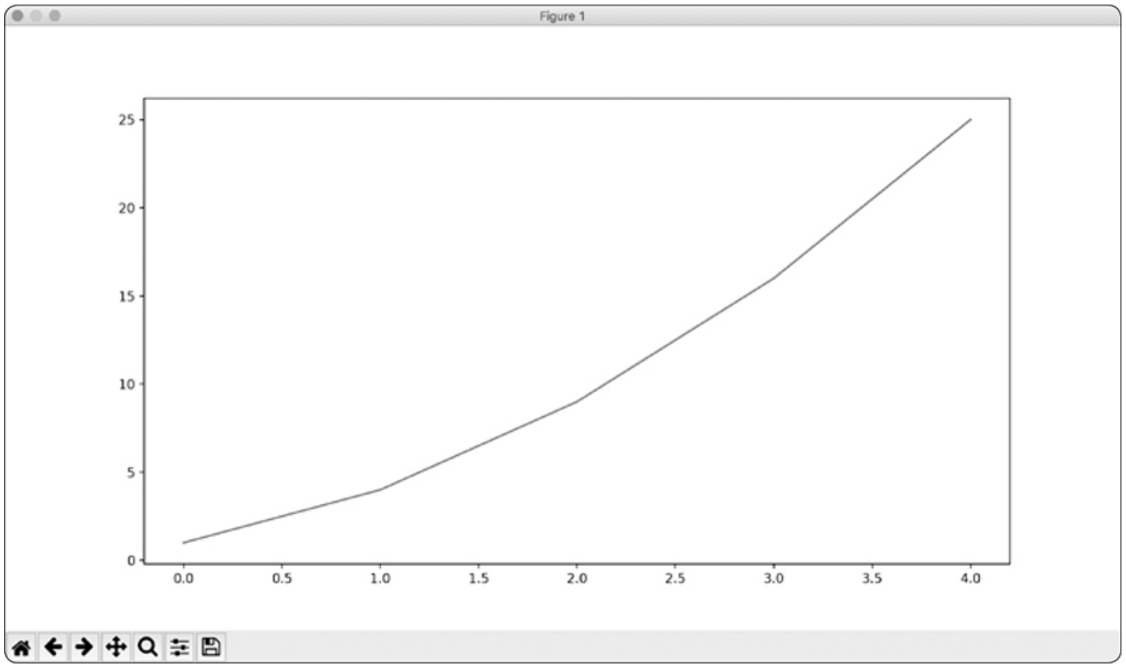

Figure 15-1: One of the simplest plots you can make in Matplotlib

Changing the Label Type and Line Thickness

Although the plot in Figure 15-1 shows that the numbers are increasing, the label type is too small and the line is a little thin to read easily. Fortunately, Matplotlib allows you to adjust every feature of a visualization.

We'll use a few of the available customizations to improve this plot's readability, as shown here:

mpl_squares.py

```
  import matplotlib.pyplot as plt

  squares = [1, 4, 9, 16, 25]

  fig, ax = plt.subplots()
❶ ax.plot(squares, linewidth=3)

  # Set chart title and label axes.
❷ ax.set_title("Square Numbers", fontsize=24)
❸ ax.set_xlabel("Value", fontsize=14)
  ax.set_ylabel("Square of Value", fontsize=14)
  # Set size of tick labels.
❹ ax.tick_params(axis='both', labelsize=14)

  plt.show()
```

The `linewidth` parameter at ❶ controls the thickness of the line that `plot()` generates. The `set_title()` method at ❷ sets a title for the chart. The `fontsize` parameters, which appear repeatedly throughout the code, control the size of the text in various elements on the chart.

The `set_xlabel()` and `set_ylabel()` methods allow you to set a title for each of the axes ❸, and the method `tick_params()` styles the tick marks ❹. The arguments shown here affect the tick marks on both the x- and y-axes (`axis='both'`) and set the font size of the tick mark labels to 14 (`labelsize=14`).

As you can see in Figure 15-2, the resulting chart is much easier to read. The label type is bigger, and the line graph is thicker. It's often worth experimenting with these values to get an idea of what will look best in the resulting graph.

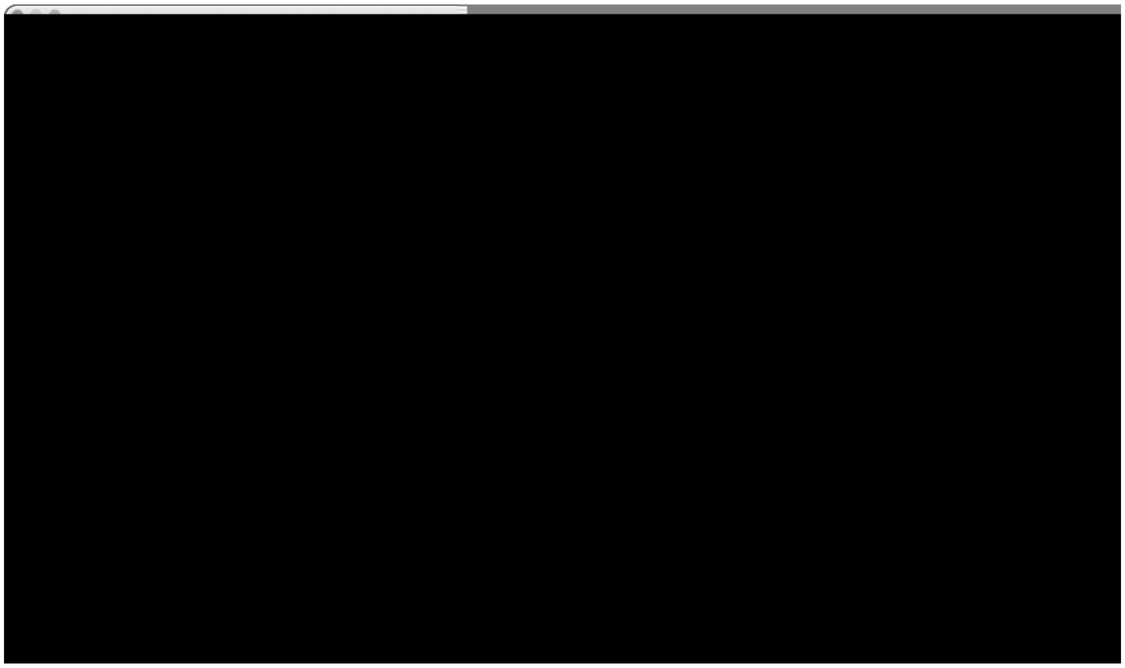

Figure 15-2: The chart is much easier to read now.

Correcting the Plot

But now that we can read the chart better, we see that the data is not plotted correctly. Notice at the end of the graph that the square of 4.0 is shown as 25! Let's fix that.

When you give `plot()` a sequence of numbers, it assumes the first data point corresponds to an x-coordinate value of 0, but our first point corresponds to an x-value of 1. We can override the default behavior by giving `plot()` the input and output values used to calculate the squares:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Set chart title and label axes.
--snip--
```

Now `plot()` will graph the data correctly because we've provided the input and output values, so it doesn't have to assume how the output numbers were generated. The resulting plot, shown in Figure 15-3, is correct.

Figure 15-3: The data is now plotted correctly.

You can specify numerous arguments when using `plot()` and use a number of functions to customize your plots. We'll continue to explore these customization functions as we work with more interesting data sets throughout this chapter.

Using Built-in Styles

Matplotlib has a number of predefined styles available, with good starting settings for background colors, gridlines, line widths, fonts, font sizes, and more that will make your visualizations appealing without requiring much customization. To see the styles available on your system, run the following lines in a terminal session:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',
--snip--
```

To use any of these styles, add one line of code before starting to generate the plot:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
--snip--
```

This code generates the plot shown in Figure 15-4. A wide variety of styles is available; play around with these styles to find some that you like.

Figure 15-4: The built-in seaborn style

Plotting and Styling Individual Points with scatter()

Sometimes, it's useful to plot and style individual points based on certain characteristics. For example, you might plot small values in one color and larger values in a different color. You could also plot a large data set with one set of styling options and then emphasize individual points by replotting them with different options.

To plot a single point, use the `scatter()` method. Pass the single (*x*, *y*) values of the point of interest to `scatter()` to plot those values:

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Let’s style the output to make it more interesting. We’ll add a title, label the axes, and make sure all the text is large enough to read:

```
  import matplotlib.pyplot as plt

  plt.style.use('seaborn')
  fig, ax = plt.subplots()
➊ ax.scatter(2, 4, s=200)

  # Set chart title and label axes.
  ax.set_title("Square Numbers", fontsize=24)
  ax.set_xlabel("Value", fontsize=14)
  ax.set_ylabel("Square of Value", fontsize=14)

  # Set size of tick labels.
  ax.tick_params(axis='both', which='major', labelsize=14)

  plt.show()
```

At ➊ we call `scatter()` and use the `s` argument to set the size of the dots used to draw the graph. When you run *scatter_squares.py* now, you should see a single point in the middle of the chart, as shown in Figure 15-5.

Figure 15-5: Plotting a single point

Plotting a Series of Points with scatter()

To plot a series of points, we can pass `scatter()` separate lists of x- and y-values, like this:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Set chart title and label axes.
--snip--
```

The `x_values` list contains the numbers to be squared, and `y_values` contains the square of each number. When these lists are passed to `scatter()`, Matplotlib reads one value from each list as it plots each point. The points to be plotted are (1, 1), (2, 4), (3, 9), (4, 16), and (5, 25); Figure 15-6 shows the result.

Figure 15-6: A scatter plot with multiple points

Calculating Data Automatically

Writing lists by hand can be inefficient, especially when we have many points. Rather than passing our points in a list, let's use a loop in Python to do the calculations for us.

Here's how this would look with 1000 points:

scatter_squares.py

```
  import matplotlib.pyplot as plt

❶ x_values = range(1, 1001)
  y_values = [x**2 for x in x_values]

  plt.style.use('seaborn')
  fig, ax = plt.subplots()
❷ ax.scatter(x_values, y_values, s=10)

  # Set chart title and label axes.
  --snip--

  # Set the range for each axis.
❸ ax.axis([0, 1100, 0, 1100000])

  plt.show()
```

We start with a range of x-values containing the numbers 1 through 1000 ❶. Next, a list comprehension generates the y-values by looping through the x-values (`for x in x_values`), squaring each number (`x**2`) and storing the results in `y_values`. We then pass the input and output lists to `scatter()` ❷. Because this is a large data set, we use a smaller point size.

At ❸ we use the `axis()` method to specify the range of each axis. The `axis()` method requires four values: the minimum and maximum values for the x-axis and the y-axis. Here, we run the x-axis from 0 to 1100 and the y-axis from 0 to 1,100,000. Figure 15-7 shows the result.

Figure 15-7: Python can plot 1000 points as easily as it plots 5 points.

Defining Custom Colors

To change the color of the points, pass `c` to `scatter()` with the name of a color to use in quotation marks, as shown here:

```
ax.scatter(x_values, y_values, c='red', s=10)
```

You can also define custom colors using the RGB color model. To define a color, pass the `c` argument a tuple with three decimal values (one each for red, green, and blue in that order), using values between 0 and 1. For example, the following line creates a plot with light-green dots:

```
ax.scatter(x_values, y_values, c=(0, 0.8, 0), s=10)
```

Values closer to 0 produce dark colors, and values closer to 1 produce lighter colors.

Using a Colormap

A *colormap* is a series of colors in a gradient that moves from a starting to an ending color. You use colormaps in visualizations to emphasize a pattern in the data. For example, you might make low values a light color and high values a darker color.

The `pyplot` module includes a set of built-in colormaps. To use one of these colormaps, you need to specify how `pyplot` should assign a color to each point in the data set. Here's how to assign each point a color based on its y-value:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = range(1, 1001)
y_values = [x**2 for x in x_values]

ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)

# Set chart title and label axes.
--snip--
```

We pass the list of y-values to `c`, and then tell `pyplot` which colormap to use using the `cmap` argument. This code colors the points with lower y-values light blue and colors the points with higher y-values dark blue. Figure 15-8 shows the resulting plot.

NOTE

*You can see all the colormaps available in `pyplot` at https://matplotlib.org/; go to **Examples**, scroll down to **Color**, and click **Colormap reference**.*

Figure 15-8: A plot using the `Blues` colormap

Saving Your Plots Automatically

If you want your program to automatically save the plot to a file, you can replace the call to `plt.show()` with a call to `plt.savefig()`:

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

The first argument is a filename for the plot image, which will be saved in the same directory as *scatter_squares.py*. The second argument trims extra whitespace from the plot. If you want the extra whitespace around the plot, just omit this argument.

TRY IT YOURSELF

15-1. Cubes: A number raised to the third power is a *cube*. Plot the first five cubic numbers, and then plot the first 5000 cubic numbers.

15-2. Colored Cubes: Apply a colormap to your cubes plot.

Random Walks

In this section, we'll use Python to generate data for a random walk, and then use Matplotlib to create a visually appealing representation of that data. A *random walk* is a path that has no clear direction but is determined by a series of random decisions, each of which is left entirely to chance. You might imagine a random walk as the path a confused ant would take if it took every step in a random direction.

Random walks have practical applications in nature, physics, biology, chemistry, and economics. For example, a pollen grain floating on a drop of water moves across the surface of the water because it's constantly pushed around by water molecules. Molecular motion in a water drop is random, so the path a pollen grain traces on the surface is a random walk. The code we'll write next models many real-world situations.

Creating the RandomWalk Class

To create a random walk, we'll create a `RandomWalk` class, which will make random decisions about which direction the walk should take. The class needs three attributes: one variable to store the number of points in the walk and two lists to store the x- and y-coordinate values of each point in the walk.

We'll only need two methods for the `RandomWalk` class: the `__init__()` method and `fill_walk()`, which will calculate the points in the walk. Let's start with `__init__()` as shown here:

random_walk.py

```
❶ from random import choice

  class RandomWalk:
      """A class to generate random walks."""

❷     def __init__(self, num_points=5000):
          """Initialize attributes of a walk."""
          self.num_points = num_points

          # All walks start at (0, 0).
❸         self.x_values = [0]
          self.y_values = [0]
```

To make random decisions, we'll store possible moves in a list and use the `choice()` function, from the `random` module, to decide which move to make each time a step is taken ❶. We then set the default number of points in a walk to 5000, which is large enough to generate some interesting patterns but small enough to generate walks quickly ❷. Then at ❸ we make two lists to hold the x- and y-values, and we start each walk at the point (0, 0).

Choosing Directions

We'll use the `fill_walk()` method, as shown here, to fill our walk with points and determine the direction of each step. Add this method to *random_walk.py*:

random_walk.py

```
      def fill_walk(self):
          """Calculate all the points in the walk."""

          # Keep taking steps until the walk reaches the desired length.
❶         while len(self.x_values) < self.num_points:

              # Decide which direction to go and how far to go in that direction.
```

```
❷       x_direction = choice([1, -1])
        x_distance = choice([0, 1, 2, 3, 4])
❸       x_step = x_direction * x_distance

        y_direction = choice([1, -1])
        y_distance = choice([0, 1, 2, 3, 4])
❹       y_step = y_direction * y_distance

        # Reject moves that go nowhere.
❺       if x_step == 0 and y_step == 0:
            continue

        # Calculate the new position.
❻       x = self.x_values[-1] + x_step
        y = self.y_values[-1] + y_step

        self.x_values.append(x)
        self.y_values.append(y)
```

At ❶ we set up a loop that runs until the walk is filled with the correct number of points. The main part of the `fill_walk()` method tells Python how to simulate four random decisions: will the walk go right or left? How far will it go in that direction? Will it go up or down? How far will it go in that direction?

We use `choice([1, -1])` to choose a value for `x_direction`, which returns either 1 for right movement or –1 for left ❷. Next, `choice([0, 1, 2, 3, 4])` tells Python how far to move in that direction (`x_distance`) by randomly selecting an integer between 0 and 4. (The inclusion of a 0 allows us to take steps along the y-axis as well as steps that have movement along both axes.)

At ❸ and ❹ we determine the length of each step in the *x* and *y* directions by multiplying the direction of movement by the distance chosen. A positive result for `x_step` means move right, a negative result means move left, and 0 means move vertically. A positive result for `y_step` means move up, negative means move down, and 0 means move horizontally. If the value of both `x_step` and `y_step` are 0, the walk doesn't go anywhere, so we continue the loop to ignore this move ❺.

To get the next x-value for the walk, we add the value in `x_step` to the last value stored in `x_values` ❻ and do the same for the y-values. When we have these values, we append them to `x_values` and `y_values`.

Plotting the Random Walk

Here's the code to plot all the points in the walk:

rw_visual.py

```
  import matplotlib.pyplot as plt

  from random_walk import RandomWalk

  # Make a random walk.
❶ rw = RandomWalk()
  rw.fill_walk()

  # Plot the points in the walk.
  plt.style.use('classic')
  fig, ax = plt.subplots()
❷ ax.scatter(rw.x_values, rw.y_values, s=15)
  plt.show()
```

We begin by importing `pyplot` and `RandomWalk`. We then create a random walk and store it in `rw` ❶, making sure to call `fill_walk()`. At ❷ we feed the walk's x- and y-values to `scatter()` and choose an appropriate dot size. Figure 15-9 shows the resulting plot with 5000 points. (The images in this section omit Matplotlib's viewer, but you'll continue to see it when you run *rw_visual.py*.)

Figure 15-9: A random walk with 5000 points

Generating Multiple Random Walks

Every random walk is different, and it’s fun to explore the various patterns that can be generated. One way to use the preceding code to make multiple walks without having to run the program several times is to wrap it in a `while` loop, like this:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Keep making new walks, as long as the program is active.
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    ax.scatter(rw.x_values, rw.y_values, s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    if keep_running == 'n':
        break
```

This code generates a random walk, displays it in Matplotlib's viewer, and pauses with the viewer open. When you close the viewer, you'll be asked whether you want to generate another walk. Press `y` to generate walks that stay near the starting point, that wander off mostly in one direction, that have thin sections connecting larger groups of points, and so on. When you want to end the program, press `n`.

Styling the Walk

In this section, we'll customize our plots to emphasize the important characteristics of each walk and deemphasize distracting elements. To do so, we identify the characteristics we want to emphasize, such as where the walk began, where it ended, and the path taken. Next, we identify the characteristics to deemphasize, such as tick marks and labels. The result should be a simple visual representation that clearly communicates the path taken in each random walk.

Coloring the Points

We'll use a colormap to show the order of the points in the walk, and then remove the black outline from each dot so the color of the dots will be clearer. To color the points according to their position in the walk, we pass the `c` argument a list containing the position of each point. Because the points are plotted in order, the list just contains the numbers from 0 to 4999, as shown here:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
❶    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers,
cmap=plt.cm.Blues,
        edgecolors='none', s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    --snip--
```

At ❶ we use `range()` to generate a list of numbers equal to the number of points in the walk. Then we store them in the list `point_numbers`, which we'll use to set the color of each point in the walk. We pass `point_numbers` to the `c` argument, use the `Blues` colormap, and then pass `edgecolors='none'` to get rid of the black outline around each point. The result is a plot of the walk that varies from light to dark blue along a gradient, as shown in Figure 15-10.

Figure 15-10: A random walk colored with the `Blues` colormap

Plotting the Starting and Ending Points

In addition to coloring points to show their position along the walk, it would be useful to see where each walk begins and ends. To do so, we can plot the first and last points individually after the main series has been plotted. We'll make the end points larger and color them differently to make them stand out, as shown here:

rw_visual.py

```
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers,
cmap=plt.cm.Blues,
        edgecolors='none', s=15)

    # Emphasize the first and last points.
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    plt.show()
    --snip--
```

To show the starting point, we plot point (0, 0) in green in a larger size (`s=100`) than the rest of the points. To mark the end point, we plot the last x- and y-value in the walk in red with a size of 100. Make sure you insert this code just before the call to `plt.show()` so the starting and ending points are drawn on top of all the other points.

When you run this code, you should be able to spot exactly where each walk begins and ends. (If these end points don't stand out clearly, adjust their color and size until they do.)

Cleaning Up the Axes

Let's remove the axes in this plot so they don't distract from the path of each walk. To turn off the axes, use this code:

rw_visual.py

```
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    # Remove the axes.
❶   ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()
    --snip--
```

To modify the axes, we use the `ax.get_xaxis()` and `ax.get_yaxis()` methods ❶ to set the visibility of each axis to `False`. As you continue to work with visualizations, you'll frequently see this chaining of methods.

Run *rw_visual.py* now; you should see a series of plots with no axes.

Adding Plot Points

Let's increase the number of points to give us more data to work with. To do so, we increase the value of `num_points` when we make a `RandomWalk` instance and adjust the size of each dot when drawing the plot, as shown here:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers,
cmap=plt.cm.Blues,
        edgecolor='none', s=1)
    --snip--
```

This example creates a random walk with 50,000 points (to mirror real-world data) and plots each point at size `s=1`. The resulting walk is wispy and cloud-like, as shown in Figure 15-11. As you can see, we've created a piece of art from a simple scatter plot!

Experiment with this code to see how much you can increase the number of points in a walk before your system starts to slow down significantly or the plot loses its visual appeal.

Figure 15-11: A walk with 50,000 points

Altering the Size to Fill the Screen

A visualization is much more effective at communicating patterns in data if it fits nicely on the screen. To make the plotting window better fit your screen, adjust the size of Matplotlib's output, like this:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots(figsize=(15, 9))
    --snip--
```

When creating the plot, you can pass a `figsize` argument to set the size of the figure. The `figsize` parameter takes a tuple, which tells Matplotlib the dimensions of the plotting window in inches.

Matplotlib assumes that your screen resolution is 100 pixels per inch; if this code doesn't give you an accurate plot size, adjust the numbers as necessary. Or, if you know your system's resolution, pass `plt.subplots()` the resolution using the `dpi` parameter to set a plot size that makes effective use of the space available on your screen, as shown here:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

TRY IT YOURSELF

15-3. Molecular Motion: Modify *rw_visual.py* by replacing `ax.scatter()` with `ax.plot()`. To simulate the path of a pollen grain on the surface of a drop of water, pass in the `rw.x_values` and `rw.y_values`, and include a `linewidth` argument. Use 5000 instead of 50,000 points.

15-4. Modified Random Walks: In the `RandomWalk` class, `x_step` and `y_step` are generated from the same set of conditions. The direction is chosen randomly from the list `[1, -1]` and the distance from the list `[0, 1, 2, 3, 4]`. Modify the values in these lists to see what happens to the overall shape of your walks. Try a longer list of choices for the distance, such as 0 through 8, or remove the –1 from the *x* or *y* direction list.

15-5. Refactoring: The `fill_walk()` method is lengthy. Create a new method called `get_step()` to determine the direction and distance for each step, and then calculate the step. You should end up with two calls to `get_step()` in `fill_walk()`:

```
x_step = self.get_step()
y_step = self.get_step()
```

This refactoring should reduce the size of `fill_walk()` and make the method easier to read and understand.

Rolling Dice with Plotly

In this section, we'll use the Python package Plotly to produce interactive visualizations. Plotly is particularly useful when you're creating visualizations that will be displayed in a browser, because the visualizations will scale automatically to fit the viewer's screen. Visualizations that Plotly generates are also interactive; when the user hovers over certain elements on the screen, information about that element is highlighted.

In this project, we'll analyze the results of rolling dice. When you roll one regular, six-sided die, you have an equal chance of rolling any of the numbers from 1 through 6. However, when you use two dice, you're more likely to roll certain numbers rather than others. We'll try to determine which numbers are most likely to occur by generating a data set that represents rolling dice. Then we'll plot the results of a large number of rolls to determine which results are more likely than others.

The study of rolling dice is often used in mathematics to explain various types of data analysis. But it also has real-world applications in casinos and other gambling scenarios, as well as in the way games like Monopoly and many role-playing games are played.

Installing Plotly

Install Plotly using `pip`, just as you did for Matplotlib:

```
$ python -m pip install --user plotly
```

If you used `python3` or something else when installing Matplotlib, make sure you use the same command here.

To see what kind of visualizations are possible with Plotly, visit the gallery of chart types at *https://plot.ly/python/*. Each example includes source code, so you can see how Plotly generates the visualizations.

Creating the Die Class

We'll create the following `Die` class to simulate the roll of one die:

die.py

```
from random import randint

class Die:
    """A class representing a single die."""

❶   def __init__(self, num_sides=6):
        """Assume a six-sided die."""
        self.num_sides = num_sides

    def roll(self):
        """"Return a random value between 1 and number of sides."""
❷       return randint(1, self.num_sides)
```

The `__init__()` method takes one optional argument. With the `Die` class, when an instance of our die is created, the number of sides will always be six if no argument is included. If an argument *is* included, that value will set the number of sides on the die ❶. (Dice are named for their number of sides: a six-sided die is a D6, an eight-sided die is a D8, and so on.)

The `roll()` method uses the `randint()` function to return a random number between 1 and the number of sides ❷. This function can return the starting value (1), the ending value (`num_sides`), or any integer between the two.

Rolling the Die

Before creating a visualization based on the `Die` class, let's roll a D6, print the results, and check that the results look reasonable:

die_visual.py

```
  from die import Die

  # Create a D6.
❶ die = Die()

  # Make some rolls, and store results in a list.
  results = []
❷ for roll_num in range(100):
      result = die.roll()
      results.append(result)

  print(results)
```

At ❶ we create an instance of `Die` with the default six sides. At ❷ we roll the die 100 times and store the results of each roll in the list `results`. Here's a sample set of results:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4,
 1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
1, 5, 1, 2]
```

A quick scan of these results shows that the `Die` class seems to be working. We see the values 1 and 6, so we know the smallest and largest possible values are being returned, and because we don't see 0 or 7, we know all the results are in the appropriate range. We also see each number from 1 through 6, which indicates that all possible outcomes are represented. Let's determine exactly how many times each number appears.

Analyzing the Results

We'll analyze the results of rolling one D6 by counting how many times we roll each number:

die_visual.py

```
  --snip--
  # Make some rolls, and store results in a list.
  results = []
❶ for roll_num in range(1000):
      result = die.roll()
      results.append(result)

  # Analyze the results.
  frequencies = []
❷ for value in range(1, die.num_sides+1):
❸     frequency = results.count(value)
❹     frequencies.append(frequency)

  print(frequencies)
```

Because we're no longer printing the results, we can increase the number of simulated rolls to 1000 ❶. To analyze the rolls, we create the empty list `frequencies` to store the number of times each value is rolled. We loop through the possible values (1 through 6 in this case) at ➋, count how many times each number appears in `results` ➌, and then append this value to the `frequencies` list ➍. We then print this list before making a visualization:

```
[155, 167, 168, 170, 159, 181]
```

These results look reasonable: we see six frequencies, one for each possible number when you roll a D6, and we see that no frequency is significantly higher than any other. Now let's visualize these results.

Making a Histogram

With a list of frequencies, we can make a *histogram* of the results. A histogram is a bar chart showing how often certain results occur. Here's the code to create the histogram:

die_visual.py

```
  from plotly.graph_objs import Bar, Layout
  from plotly import offline

  from die import Die
  --snip--

  # Analyze the results.
  frequencies = []
  for value in range(1, die.num_sides+1):
      frequency = results.count(value)
      frequencies.append(frequency)

  # Visualize the results.
➊ x_values = list(range(1, die.num_sides+1))
➋ data = [Bar(x=x_values, y=frequencies)]

➌ x_axis_config = {'title': 'Result'}
```

```
  y_axis_config = {'title': 'Frequency of Result'}
❹ my_layout = Layout(title='Results of rolling one D6 1000 times',
          xaxis=x_axis_config, yaxis=y_axis_config)
❺ offline.plot({'data': data, 'layout': my_layout}, filename='d6.html')
```

To make a histogram, we need a bar for each of the possible results. We store these in a list called `x_values`, which starts at 1 and ends at the number of sides on the die ❶. Plotly doesn't accept the results of the `range()` function directly, so we need to convert the range to a list explicitly using the `list()` function. The Plotly class `Bar()` represents a data set that will be formatted as a bar chart ❷. This class needs a list of x-values, and a list of y-values. The class must be wrapped in square brackets, because a data set can have multiple elements.

Each axis can be configured in a number of ways, and each configuration option is stored as an entry in a dictionary. At this point, we're just setting the title of each axis ❸. The `Layout()` class returns an object that specifies the layout and configuration of the graph as a whole ❹. Here we set the title of the graph and pass the x- and y-axis configuration dictionaries as well.

To generate the plot, we call the `offline.plot()` function ❺. This function needs a dictionary containing the data and layout objects, and it also accepts a name for the file where the graph will be saved. We store the output in a file called *d6.html*.

When you run the program *die_visual.py*, a browser will probably open showing the file *d6.html*. If this doesn't happen automatically, open a new tab in any web browser, and then open the file *d6.html* (in the folder where you saved *die_visual.py*). You should see a chart that looks like the one in Figure 15-12. (I've modified this chart slightly for printing; by default, Plotly generates charts with smaller text than what you see here.)

Figure 15-12: A simple bar chart created with Plotly

Notice that Plotly has made the chart interactive: hover your cursor over any bar in the chart, and you'll see the associated data. This feature is particularly useful when you're plotting multiple data sets on the same chart. Also notice the icons in the upper right, which allow you to pan and zoom the visualization, and save your visualization as an image.

Rolling Two Dice

Rolling two dice results in larger numbers and a different distribution of results. Let's modify our code to create two D6 dice to simulate the way we roll a pair of dice. Each time we roll the pair, we'll add the two numbers (one from each die) and store the sum in `results`. Save a copy of *die_visual.py* as *dice_visual.py*, and make the following changes:

dice_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create two D6 dice.
die_1 = Die()
```

```
  die_2 = Die()

  # Make some rolls, and store results in a list.
  results = []
  for roll_num in range(1000):
❶     result = die_1.roll() + die_2.roll()
      results.append(result)

  # Analyze the results.
  frequencies = []
❷ max_result = die_1.num_sides + die_2.num_sides
❸ for value in range(2, max_result+1):
      frequency = results.count(value)
      frequencies.append(frequency)

  # Visualize the results.
  x_values = list(range(2, max_result+1))
  data = [Bar(x=x_values, y=frequencies)]

❹ x_axis_config = {'title': 'Result', 'dtick': 1}
  y_axis_config = {'title': 'Frequency of Result'}
  my_layout = Layout(title='Results of rolling two D6 dice 1000 times',
          xaxis=x_axis_config, yaxis=y_axis_config)
  offline.plot({'data': data, 'layout': my_layout}, filename='d6_d6.html')
```

After creating two instances of `Die`, we roll the dice and calculate the sum of the two dice for each roll ❶. The largest possible result (12) is the sum of the largest number on both dice, which we store in `max_result` ❷. The smallest possible result (2) is the sum of the smallest number on both dice. When we analyze the results, we count the number of results for each value between 2 and `max_result` ❸. (We could have used `range(2, 13)`, but this would work only for two D6 dice. When modeling real-world situations, it's best to write code that can easily model a variety of situations. This code allows us to simulate rolling a pair of dice with any number of sides.)

When creating the chart, we include the `dtick` key in the `x_axis_config` dictionary ❹. This setting controls the spacing between tick marks on the x-axis. Now that we have more bars on the histogram, Plotly's default settings will only label some of the bars. The `'dtick': 1` setting tells Plotly to label every tick mark. We also update the title of the chart and change the output filename as well.

After running this code, you should see a chart that looks like the one in Figure 15-13.

Figure 15-13: Simulated results of rolling two six-sided dice 1000 times

This graph shows the approximate results you're likely to get when you roll a pair of D6 dice. As you can see, you're least likely to roll a 2 or a 12 and most likely to roll a 7. This happens because there are six ways to roll a 7, namely: 1 and 6, 2 and 5, 3 and 4, 4 and 3, 5 and 2, or 6 and 1.

Rolling Dice of Different Sizes

Let's create a six-sided die and a ten-sided die, and see what happens when we roll them 50,000 times:

dice_visual.py

```
  from plotly.graph_objs import Bar, Layout
  from plotly import offline

  from die import Die

  # Create a D6 and a D10.
  die_1 = Die()
❶ die_2 = Die(10)

  # Make some rolls, and store results in a list.
  results = []
  for roll_num in range(50_000):
      result = die_1.roll() + die_2.roll()
      results.append(result)

  # Analyze the results.
  --snip--

  # Visualize the results.
  x_values = list(range(2, max_result+1))
  data = [Bar(x=x_values, y=frequencies)]

  x_axis_config = {'title': 'Result', 'dtick': 1}
  y_axis_config = {'title': 'Frequency of Result'}
❷ my_layout = Layout(title='Results of rolling a D6 and a D10 50000 times',
          xaxis=x_axis_config, yaxis=y_axis_config)
  offline.plot({'data': data, 'layout': my_layout}, filename='d6_d10.html')
```

To make a D10, we pass the argument `10` when creating the second `Die` instance ❶ and change the first loop to simulate 50,000 rolls instead of 1000. We change the title of the graph and update the output filename as well ❷.

Figure 15-14 shows the resulting chart. Instead of one most likely result, there are five. This happens because there's still only one way to roll the smallest value (1 and 1) and the largest value (6 and 10), but the smaller die limits the number of ways you can generate the middle numbers: there are six ways to roll a 7, 8, 9, 10, and 11. Therefore, these are the most common results, and you're equally likely to roll any one of these numbers.

Figure 15-14: The results of rolling a six-sided die and a ten-sided die 50,000 times

Our ability to use Plotly to model the rolling of dice gives us considerable freedom in exploring this phenomenon. In just minutes you can simulate a tremendous number of rolls using a large variety of dice.

TRY IT YOURSELF

15-6. Two D8s: Create a simulation showing what happens when you roll two eight-sided dice 1000 times. Try to picture what you think the visualization will look like before you run the simulation; then see if your intuition was correct. Gradually increase the number of rolls until you start to see the limits of your system's capabilities.

15-7. Three Dice: When you roll three D6 dice, the smallest number you can roll is 3 and the largest number is 18. Create a visualization that shows what happens when you roll three D6 dice.

15-8. Multiplication: When you roll two dice, you usually add the two numbers together to get the result. Create a visualization that shows what happens if you multiply these numbers instead.

15-9. Die Comprehensions: For clarity, the listings in this section use the long form of `for` loops. If you're comfortable using list comprehensions, try writing a comprehension for one or both of the loops in each of these programs.

15-10. Practicing with Both Libraries: Try using Matplotlib to make a die-rolling visualization, and use Plotly to make the visualization for a random walk. (You'll need to consult the documentation for each library to complete this exercise.)

Summary

In this chapter, you learned to generate data sets and create visualizations of that data. You created simple plots with Matplotlib and used a scatter plot to explore random walks. You also created a histogram with Plotly and used a histogram to explore the results of rolling dice of different sizes.

Generating your own data sets with code is an interesting and powerful way to model and explore a wide variety of real-world situations. As you continue to work through the data visualization projects that follow, keep an eye out for situations you might be able to model with code. Look at the visualizations you see in news media, and see if you can identify those that were generated using methods similar to the ones you're learning in these projects.

In Chapter 16, you'll download data from online sources and continue to use Matplotlib and Plotly to explore that data.