16

DOWNLOADING DATA

In this chapter, you'll download data sets from online sources and create working visualizations of that data. You can find an incredible variety of data online, much of which hasn't been examined thoroughly. The ability to analyze this data allows you to discover patterns and connections that no one else has found.

We'll access and visualize data stored in two common data formats, CSV and JSON. We'll use Python's `csv` module to process weather data stored in the CSV (comma-separated values) format and analyze high and low temperatures over time in two different locations. We'll then use Matplotlib to generate a chart based on our downloaded data to display variations in temperature in two dissimilar environments: Sitka, Alaska, and Death Valley, California. Later in the chapter, we'll use the `json` module to access earthquake data stored in the JSON format and use Plotly to draw a world map showing the locations and magnitudes of recent earthquakes.

By the end of this chapter, you'll be prepared to work with different types and data set formats, and you'll have a deeper understanding of how to build complex visualizations. Being able to access and visualize online data of different types and formats is essential to working with a wide variety of real-world data sets.

The CSV File Format

One simple way to store data in a text file is to write the data as a series of values separated by commas, which is called *comma-separated values*. The resulting files are called *CSV* files. For example, here's a chunk of weather data in CSV format:

```
"USW00025333","SITKA AIRPORT, AK US","2018-01-01","0.45",,"48","38"
```

This is an excerpt of some weather data from January 1, 2018 in Sitka, Alaska. It includes the day's high and low temperatures, as well as a number of other measurements from that day. CSV files can be tricky for humans to read, but they're easy for programs to process and extract values from, which speeds up the data analysis process.

We'll begin with a small set of CSV-formatted weather data recorded in Sitka, which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. Make a folder called *data* inside the folder where you're saving this chapter's programs. Copy the file *sitka_weather_07-2018_simple.csv* into this new folder. (After you download the book's resources, you'll have all the files you need for this project.)

NOTE

The weather data in this project was originally downloaded from https://ncdc.noaa.gov/cdo-web/.

Parsing the CSV File Headers

Python's `csv` module in the standard library parses the lines in a CSV file and allows us to quickly extract the values we're interested in. Let's start by examining the first line of the file, which contains a series of headers for the data. These headers tell us what kind of information the data holds:

sitka_highs.py

```
import csv

filename = 'data/sitka_weather_07-2018_simple.csv'
❶ with open(filename) as f:
❷     reader = csv.reader(f)
❸     header_row = next(reader)
      print(header_row)
```

After importing the `csv` module, we assign the name of the file we're working with to `filename`. We then open the file and assign the resulting file object to `f` ❶. Next, we call `csv.reader()` and pass it the file object as an argument to create a reader object associated with that file ❷. We assign the reader object to `reader`.

The `csv` module contains a `next()` function, which returns the next line in the file when passed the reader object. In the preceding listing, we call `next()` only once so we get the first line of the file, which contains the file headers ❸. We store the data that's returned in `header_row`. As you can see, `header_row` contains meaningful, weather-related headers that tell us what information each line of data holds:

```
['STATION', 'NAME', 'DATE', 'PRCP', 'TAVG', 'TMAX', 'TMIN']
```

The `reader` object processes the first line of comma-separated values in the file and stores each as an item in a list. The header `STATION` represents the code for the weather station that recorded this data. The position of this header tells us that the first value in each line will be the weather station code. The `NAME` header indicates that the second value in each line is the name of the weather station that made the recording. The rest of the headers specify what kinds of information were recorded in each reading. The data we're most interested in for now are the date, the high temperature (`TMAX`), and the low temperature (`TMIN`). This is a simple data set that contains only precipitation and temperature-related data. When you download your own weather data, you can choose to include a number of other measurements relating to wind speed, direction, and more detailed precipitation data.

Printing the Headers and Their Positions

To make it easier to understand the file header data, we print each header and its position in the list:

sitka_highs.py

```
--snip--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

❶   for index, column_header in enumerate(header_row):
        print(index, column_header)
```

The `enumerate()` function returns both the index of each item and the value of each item as you loop through a list ❶. (Note that we've removed the line `print(header_row)` in favor of this more detailed version.)

Here's the output showing the index of each header:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TAVG
5 TMAX
6 TMIN
```

Here we see that the dates and their high temperatures are stored in columns 2 and 5. To explore this data, we'll process each row of data in *sitka_weather_07-2018_simple.csv* and extract the values with the indexes 2 and 5.

Extracting and Reading Data

Now that we know which columns of data we need, let's read in some of that data. First, we'll read in the high temperature for each day:

sitka_highs.py

```
--snip--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get high temperatures from this file.
❶   highs = []
❷   for row in reader:
❸       high = int(row[5])
        highs.append(high)

print(highs)
```

We make an empty list called `highs` ❶ and then loop through the remaining rows in the file ❷. The `reader` object continues from where it left off in the CSV file and automatically returns each line following its current position. Because we've already read the header row, the loop will begin at the second line where the actual data begins. On each pass through the loop, we pull the data from index 5, which corresponds to the header `TMAX`, and assign it to the variable `high` ❸. We use the `int()` function to convert the data, which is stored as a string, to a numerical format so we can use it. We then append this value to `highs`.

The following listing shows the data now stored in `highs`:

```
[62, 58, 70, 70, 67, 59, 58, 62, 66, 59, 56, 63, 65, 58, 56, 59, 64, 60, 60,
 61, 65, 65, 63, 59, 64, 65, 68, 66, 64, 67, 65]
```

We've extracted the high temperature for each date and stored each value in a list. Now let's create a visualization of this data.

Plotting Data in a Temperature Chart

To visualize the temperature data we have, we'll first create a simple plot of the daily highs using Matplotlib, as shown here:

sitka_highs.py

```
import csv

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
with open(filename) as f:
  --snip--

# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(highs, c='red')

# Format plot.
❷ ax.set_title("Daily high temperatures, July 2018", fontsize=24)
❸ ax.set_xlabel('', fontsize=16)
ax.set_ylabel("Temperature (F)", fontsize=16)
ax.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

We pass the list of highs to `plot()` and pass `c='red'` to plot the points in red ❶. (We'll plot the highs in red and the lows in blue.) We then specify a few other formatting details, such as the title, font size, and labels ❷, which you should recognize from Chapter 15. Because we have yet to add the dates, we won't label the x-axis, but `ax.set_xlabel()` does modify the font size to make the default labels more readable ❸. Figure 16-1 shows the resulting plot: a simple line graph of the high temperatures for July 2018 in Sitka, Alaska.

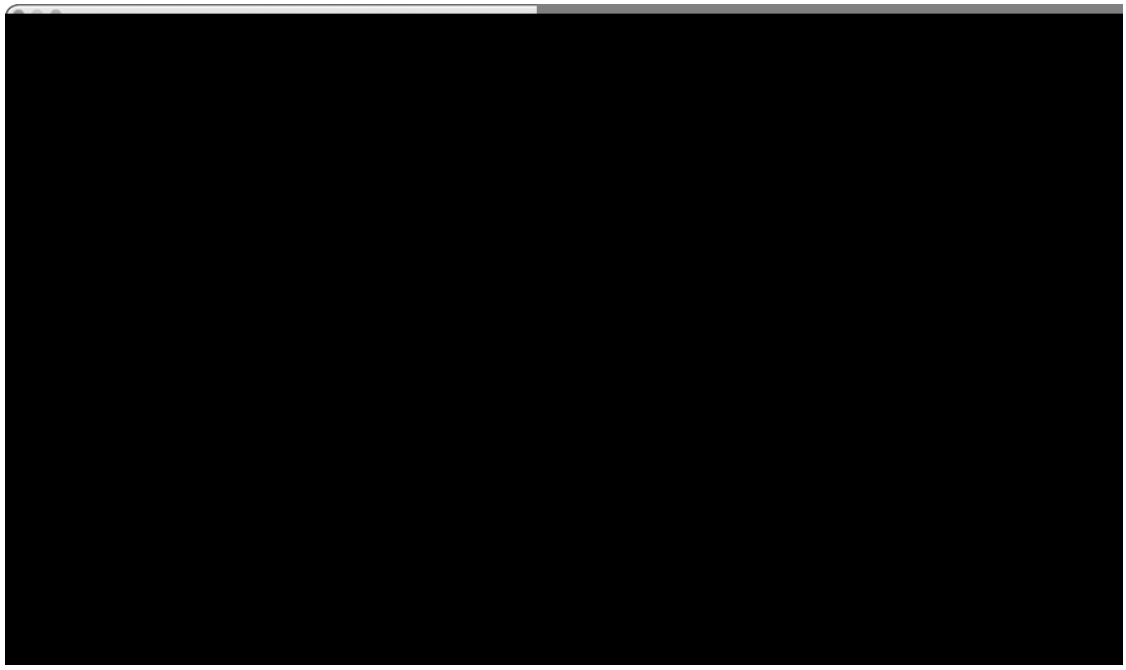

Figure 16-1: A line graph showing daily high temperatures for July 2018 in Sitka, Alaska

The datetime Module

Let's add dates to our graph to make it more useful. The first date from the weather data file is in the second row of the file:

```
"USW00025333","SITKA AIRPORT, AK US","2018-07-01","0.25",,"62","50"
```

The data will be read in as a string, so we need a way to convert the string `"2018-07-01"` to an object representing this date. We can construct an object representing July 1, 2018 using the `strptime()` method from the `datetime` module. Let's see how `strptime()` works in a terminal session:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2018-07-01', '%Y-%m-%d')
>>> print(first_date)
2018-07-01 00:00:00
```

We first import the `datetime` class from the `datetime` module. Then we call the method `strptime()` using the string containing the date we want to work with as its first argument. The second argument tells Python how the date is formatted. In this example, Python interprets `'%Y-'` to mean the part of

the string before the first dash is a four-digit year; `'%m-'` means the part of the string before the second dash is a number representing the month; and `'%d'` means the last part of the string is the day of the month, from 1 to 31.

The `strptime()` method can take a variety of arguments to determine how to interpret the date. Table 16-1 shows some of these arguments.

Table 16-1: Date and Time Formatting Arguments from the `datetime` Module

Argument	Meaning
`%A`	Weekday name, such as *Monday*
`%B`	Month name, such as *January*
`%m`	Month, as a number (01 to 12)
`%d`	Day of the month, as a number (01 to 31)
`%Y`	Four-digit year, such as 2019
`%y`	Two-digit year, such as 19
`%H`	Hour, in 24-hour format (00 to 23)
`%I`	Hour, in 12-hour format (01 to 12)
`%p`	AM or PM
`%M`	Minutes (00 to 59)
`%S`	Seconds (00 to 61)

Plotting Dates

Now we can improve our temperature data plot by extracting dates for the daily highs and passing those highs and dates to `plot()`, as shown here:

sitka_highs.py

```
import csv
from datetime import datetime

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get dates and high temperatures from this file.
❶   dates, highs = [], []
    for row in reader:
❷       current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[5])
        dates.append(current_date)
        highs.append(high)

# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❸ ax.plot(dates, highs, c='red')

# Format plot.
ax.set_title("Daily high temperatures, July 2018", fontsize=24)
ax.set_xlabel('', fontsize=16)
❹ fig.autofmt_xdate()
ax.set_ylabel("Temperature (F)", fontsize=16)
ax.tick_params(axis='both', which='major', labelsize=16)
```

```
plt.show()
```

We create two empty lists to store the dates and high temperatures from the file ❶. We then convert the data containing the date information (`row[2]`) to a `datetime` object ❷ and append it to `dates`. We pass the dates and the high temperature values to `plot()` ❸. The call to `fig.autofmt_xdate()` ❹ draws the date labels diagonally to prevent them from overlapping. Figure 16-2 shows the improved graph.

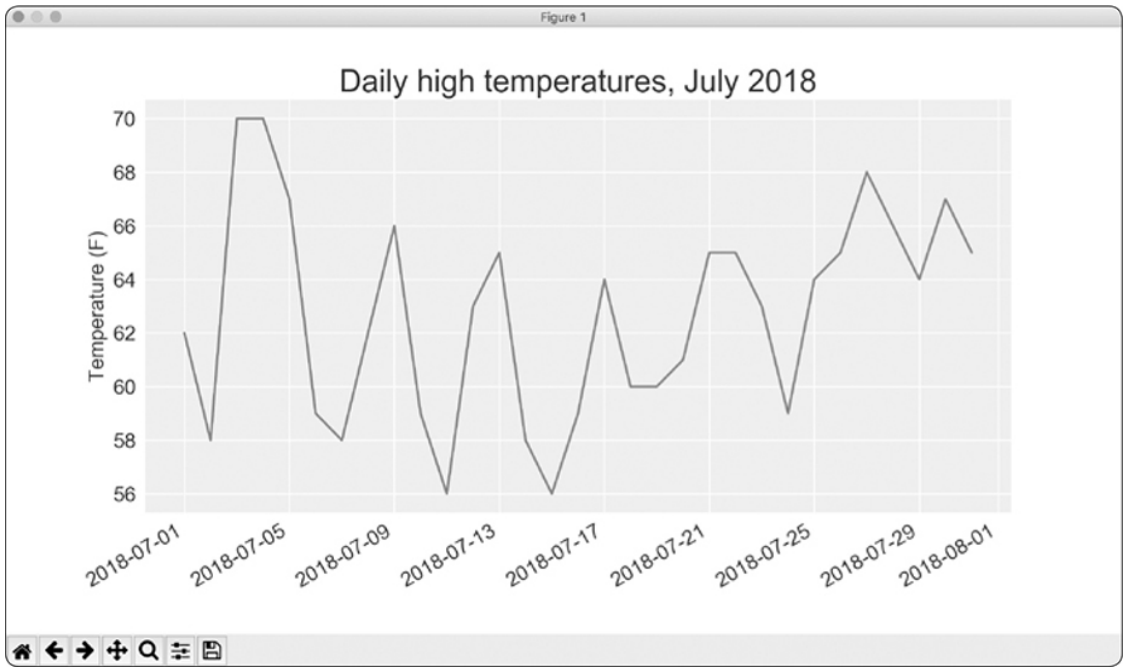

Figure 16-2: The graph is more meaningful now that it has dates on the x-axis.

Plotting a Longer Timeframe

With our graph set up, let's add more data to get a more complete picture of the weather in Sitka. Copy the file *sitka_weather_2018_simple.csv*, which contains a full year's worth of weather data for Sitka, to the folder where you're storing the data for this chapter's programs.

Now we can generate a graph for the entire year's weather:

sitka_highs.py

```
--snip--
❶ filename = 'data/sitka_weather_2018_simple.csv'
with open(filename) as f:
--snip--
# Format plot.
❷ ax.set_title("Daily high temperatures - 2018", fontsize=24)
ax.set_xlabel('', fontsize=16)
--snip--
```

We modify the filename to use the new data file *sitka_weather_2018_simple.csv* ❶, and we update the title of our plot to reflect the change in its content ❷. Figure 16-3 shows the resulting plot.

Figure 16-3: A year's worth of data

Plotting a Second Data Series

We can make our informative graph even more useful by including the low temperatures. We need to extract the low temperatures from the data file and then add them to our graph, as shown here:

sitka_highs_lows.py

```
--snip--
filename = 'sitka_weather_2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get dates, and high and low temperatures from this file.
❶   dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[5])
❷       low = int(row[6])
        dates.append(current_date)
        highs.append(high)
        lows.append(low)

# Plot the high and low temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.plot(dates, highs, c='red')
❸ ax.plot(dates, lows, c='blue')

# Format plot.
❹ ax.set_title("Daily high and low temperatures - 2018", fontsize=24)
--snip--
```

At ❶ we add the empty list `lows` to hold low temperatures, and then extract and store the low temperature for each date from the seventh position in each row (`row[6]`) ❷. At ❸ we add a call to `plot()` for the low temperatures and color these values blue. Finally, we update the title ❹. Figure 16-4 shows the resulting chart.

Figure 16-4: Two data series on the same plot

Shading an Area in the Chart

Having added two data series, we can now examine the range of temperatures for each day. Let's add a finishing touch to the graph by using shading to show the range between each day's high and low temperatures. To do so, we'll use the `fill_between()` method, which takes a series of x-values and two series of y-values, and fills the space between the two y-value series:

sitka_highs_lows.py

```
--snip--
# Plot the high and low temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(dates, highs, c='red', alpha=0.5)
ax.plot(dates, lows, c='blue', alpha=0.5)
❷ ax.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
--snip--
```

The `alpha` argument at ❶ controls a color's transparency. An `alpha` value of 0 is completely transparent, and 1 (the default) is completely opaque.

By setting `alpha` to 0.5, we make the red and blue plot lines appear lighter.

At ❷ we pass `fill_between()` the list `dates` for the x-values and then the two y-value series `highs` and `lows`. The `facecolor` argument determines the color of the shaded region; we give it a low `alpha` value of 0.1 so the filled region connects the two data series without distracting from the information they represent. Figure 16-5 shows the plot with the shaded region between the highs and lows.

Figure 16-5: The region between the two data sets is shaded.

The shading helps make the range between the two data sets immediately apparent.

Error Checking

We should be able to run the *sitka_highs_lows.py* code using data for any location. But some weather stations collect different data than others, and some occasionally malfunction and fail to collect some of the data they're supposed to. Missing data can result in exceptions that crash our programs unless we handle them properly.

For example, let's see what happens when we attempt to generate a temperature plot for Death Valley, California. Copy the file

death_valley_2018_simple.csv to the folder where you're storing the data for this chapter's programs.

First, let's run the code to see the headers that are included in this data file:

death_valley_highs_lows.py

```
import csv

filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    for index, column_header in enumerate(header_row):
        print(index, column_header)
```

Here's the output:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TMAX
5 TMIN
6 TOBS
```

The date is in the same position at index 2. But the high and low temperatures are at indexes 4 and 5, so we'd need to change the indexes in our code to reflect these new positions. Instead of including an average temperature reading for the day, this station includes TOBS, a reading for a specific observation time.

I removed one of the temperature readings from this file to show what happens when some data is missing from a file. Change

sitka_highs_lows.py to generate a graph for Death Valley using the indexes we just noted, and see what happens:

death_valley_highs_lows.py

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    # Get dates, and high and low temperatures from this file.
    dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
❶       high = int(row[4])
        low = int(row[5])
        dates.append(current_date)
--snip--
```

At ❶ we update the indexes to correspond to this file's `TMAX` and `TMIN` positions.

When we run the program, we get an error, as shown in the last line in the following output:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 15, in <module>
    high = int(row[4])
ValueError: invalid literal for int() with base 10: ''
```

The traceback tells us that Python can't process the high temperature for one of the dates because it can't turn an empty string (`''`) into an integer. Rather than look through the data and finding out which reading is missing, we'll just handle cases of missing data directly.

We'll run error-checking code when the values are being read from the CSV file to handle exceptions that might arise. Here's how that works:

death_valley_highs_lows.py

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
❶       try:
            high = int(row[4])
            low = int(row[5])
        except ValueError:
❷           print(f"Missing data for {current_date}")
❸       else:
            dates.append(current_date)
            highs.append(high)
            lows.append(low)

# Plot the high and low temperatures.
--snip--

# Format plot.
❹ title = "Daily high and low temperatures - 2018\nDeath Valley, CA"
ax.set_title(title, fontsize=20)
ax.set_xlabel('', fontsize=16)
--snip--
```

Each time we examine a row, we try to extract the date and the high and low temperature ❶. If any data is missing, Python will raise a `ValueError` and we handle it by printing an error message that includes the date of the missing data ❷. After printing the error, the loop will continue processing the next row. If all data for a date is retrieved without error, the `else` block will run and the data will be appended to the appropriate lists ❸. Because we're plotting information for a new location, we update the title to include the location on the plot, and we use a smaller font size to accommodate the longer title ❹.

When you run *death_valley_highs_lows.py* now, you'll see that only one date had missing data:

```
Missing data for 2018-02-18 00:00:00
```

Because the error is handled appropriately, our code is able to generate a plot, which skips over the missing data. Figure 16-6 shows the resulting plot.

Figure 16-6: Daily high and low temperatures for Death Valley

Comparing this graph to the Sitka graph, we can see that Death Valley is warmer overall than southeast Alaska, as we expect. Also, the range of temperatures each day is greater in the desert. The height of the shaded region makes this clear.

Many data sets you work with will have missing, improperly formatted, or incorrect data. You can use the tools you learned in the first half of this book to handle these situations. Here we used a `try-except-else` block to handle missing data. Sometimes you'll use `continue` to skip over some data or use `remove()` or `del` to eliminate some data after it's been extracted. Use any approach that works, as long as the result is a meaningful, accurate visualization.

Downloading Your Own Data

If you want to download your own weather data, follow these steps:

1. Visit the NOAA Climate Data Online site at *https://www.ncdc.noaa.gov/cdo-web/*. In the *Discover Data By* section, click **Search Tool**. In the *Select a Dataset* box, choose **Daily Summaries**.
2. Select a date range, and in the *Search For* section, choose **ZIP Codes**. Enter the ZIP Code you're interested in, and click **Search**.
3. On the next page, you'll see a map and some information about the area you're focusing on. Below the location name, click **View Full Details**, or click the map and then click **Full Details**.
4. Scroll down and click **Station List** to see the weather stations that are available in this area. Choose one of the stations, and click **Add to Cart**. This data is free, even though the site uses a shopping cart icon. In the upper-right corner, click the cart.
5. In *Select the Output*, choose **Custom GHCN-Daily CSV**. Make sure the date range is correct, and click **Continue**.
6. On the next page, you can select the kinds of data you want. You can download one kind of data, for example, focusing on air temperature, or you can download all the data available from this station. Make your choices, and then click **Continue**.
7. On the last page, you'll see a summary of your order. Enter your email address, and click **Submit Order**. You'll receive a confirmation that your order was received, and in a few minutes you should receive another email with a link to download your data.

The data you download will be structured just like the data we worked with in this section. It might have different headers than those you saw in this section. But if you follow the same steps we used here, you should be able to generate visualizations of the data you're interested in.

TRY IT YOURSELF

16-1. Sitka Rainfall: Sitka is in a temperate rainforest, so it gets a fair amount of rainfall. In the data file *sitka_weather_2018_simple.csv* is a header called PRCP, which represents daily rainfall amounts. Make a visu-

alization focusing on the data in this column. You can repeat the exercise for Death Valley if you're curious how little rainfall occurs in a desert.

16-2. Sitka–Death Valley Comparison: The temperature scales on the Sitka and Death Valley graphs reflect the different data ranges. To accurately compare the temperature range in Sitka to that of Death Valley, you need identical scales on the y-axis. Change the settings for the y-axis on one or both of the charts in Figures 16-5 and 16-6. Then make a direct comparison between temperature ranges in Sitka and Death Valley (or any two places you want to compare).

16-3. San Francisco: Are temperatures in San Francisco more like temperatures in Sitka or temperatures in Death Valley? Download some data for San Francisco, and generate a high-low temperature plot for San Francisco to make a comparison.

16-4. Automatic Indexes: In this section, we hardcoded the indexes corresponding to the `TMIN` and `TMAX` columns. Use the header row to determine the indexes for these values, so your program can work for Sitka or Death Valley. Use the station name to automatically generate an appropriate title for your graph as well.

16-5. Explore: Generate a few more visualizations that examine any other weather aspect you're interested in for any locations you're curious about.

Mapping Global Data Sets: JSON Format

In this section, you'll download a data set representing all the earthquakes that have occurred in the world during the previous month. Then you'll make a map showing the location of these earthquakes and how significant each one was. Because the data is stored in the JSON format, we'll work with it using the `json` module. Using Plotly's beginner-friendly mapping tool for location-based data, you'll create visualizations that clearly show the global distribution of earthquakes.

Downloading Earthquake Data

Copy the file *eq_1_day_m1.json* to the folder where you're storing the data for this chapter's programs. Earthquakes are categorized by their magnitude on the Richter scale. This file includes data for all earthquakes with a magnitude M1 or greater that took place in the last 24 hours (at the time of this writing). This data comes from one of the United States Geological Survey's earthquake data feeds, which you can find at *https://earthquake.usgs.gov/earthquakes/feed/*.

Examining JSON Data

When you open *eq_1_day_m1.json*, you'll see that it's very dense and hard to read:

```
{"type":"FeatureCollection","metadata":{"generated":1550361461000,...
{"type":"Feature","properties":{"mag":1.2,"place":"11km NNE of Nor...
{"type":"Feature","properties":{"mag":4.3,"place":"69km NNW of Ayn...
{"type":"Feature","properties":{"mag":3.6,"place":"126km SSE of Co...
{"type":"Feature","properties":{"mag":2.1,"place":"21km NNW of Teh...
{"type":"Feature","properties":{"mag":4,"place":"57km SSW of Kakto...
--snip--
```

This file is formatted more for machines than it is for humans. But we can see that the file contains some dictionaries, as well as information that we're interested in, such as earthquake magnitudes and locations.

The `json` module provides a variety of tools for exploring and working with JSON data. Some of these tools will help us reformat the file so we can look at the raw data more easily before we begin to work with it programmatically.

Let's start by loading the data and displaying it in a format that's easier to read. This is a long data file, so instead of printing it, we'll rewrite the data to a new file. Then we can open that file and scroll back and forth easily through the data:

eq_explore_data.py

```
import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
❶    all_eq_data = json.load(f)

❷ readable_file = 'data/readable_eq_data.json'
with open(readable_file, 'w') as f:
❸    json.dump(all_eq_data, f, indent=4)
```

We first import the `json` module to load the data properly from the file, and then store the entire set of data in `all_eq_data` ❶. The `json.load()` function converts the data into a format Python can work with: in this case, a giant dictionary. At ❷ we create a file to write this same data into a more readable format. The `json.dump()` function takes a JSON data object and a file object, and writes the data to that file ❸. The `indent=4` argument tells `dump()` to format the data using indentation that matches the data's structure.

When you look in your *data* directory and open the file *readable_eq_data.json*, here's the first part of what you'll see:

readable_eq_data.json

```
{
   "type": "FeatureCollection",
❶   "metadata": {
      "generated": 1550361461000,
      "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
      "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
      "status": 200,
      "api": "1.7.0",
      "count": 158
   },
```

```
❷  "features": [
   --snip--
```

The first part of the file includes a section with the key `"metadata"`. This tells us when the data file was generated and where we can find the data online. It also gives us a human-readable title and the number of earthquakes included in this file. In this 24-hour period, 158 earthquakes were recorded.

This *geoJSON* file has a structure that's helpful for location-based data. The information is stored in a list associated with the key `"features"` ❷. Because this file contains earthquake data, the data is in list form where every item in the list corresponds to a single earthquake. This structure might look confusing, but it's quite powerful. It allows geologists to store as much information as they need to in a dictionary about each earthquake, and then stuff all those dictionaries into one big list.

Let's look at a dictionary representing a single earthquake:

readable_eq_data.json

```
   --snip--
     {
       "type": "Feature",
❶       "properties": {
         "mag": 0.96,
         --snip--
❷         "title": "M 1.0 - 8km NE of Aguanga, CA"
       },
❸       "geometry": {
         "type": "Point",
         "coordinates": [
❹           -116.7941667,
❺           33.4863333,
           3.22
         ]
```

```
        },
        "id": "ci37532978"
    },
```

The key `"properties"` contains a lot of information about each earthquake ❶. We're mainly interested in the magnitude of each quake, which is associated with the key `"mag"`. We're also interested in the title of each earthquake, which provides a nice summary of its magnitude and location ❷.

The key `"geometry"` helps us understand where the earthquake occurred ❸. We'll need this information to map each event. We can find the longitude ❹ and the latitude ❺ for each earthquake in a list associated with the key `"coordinates"`.

This file contains way more nesting than we'd use in the code we write, so if it looks confusing, don't worry: Python will handle most of the complexity. We'll only be working with one or two nesting levels at a time. We'll start by pulling out a dictionary for each earthquake that was recorded in the 24-hour time period.

NOTE

When we talk about locations, we often say the location's latitude first, followed by the longitude. This convention probably arose because humans discovered latitude long before we developed the concept of longitude. However, many geospatial frameworks list the longitude first and then the latitude, because this corresponds to the (x, y) *convention we use in mathematical representations. The geoJSON format follows the (longitude, latitude) convention, and if you use a different framework it's important to learn what convention that framework follows.*

Making a List of All Earthquakes

First, we'll make a list that contains all the information about every earthquake that occurred.

eq_explore_data.py

```
import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
    all_eq_data = json.load(f)

all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

We take the data associated with the key `'features'` and store it in `all_eq_dicts`. We know this file contains records about 158 earthquakes, and the output verifies that we've captured all of the earthquakes in the file:

```
158
```

Notice how short this code is. The neatly formatted file *readable_eq_data.json* has over 6,000 lines. But in just a few lines, we can read through all that data and store it in a Python list. Next, we'll pull the magnitudes from each earthquake.

Extracting Magnitudes

Using the list containing data about each earthquake, we can loop through that list and extract any information we want. Now we'll pull the magnitude of each earthquake:

eq_explore_data.py

```
--snip--
all_eq_dicts = all_eq_data['features']
```

```
❶ mags = []
  for eq_dict in all_eq_dicts:
❷     mag = eq_dict['properties']['mag']
      mags.append(mag)

  print(mags[:10])
```

We make an empty list to store the magnitudes, and then loop through the list `all_eq_dicts` ❶. Inside this loop, each earthquake is represented by the dictionary `eq_dict`. Each earthquake's magnitude is stored in the `'properties'` section of this dictionary under the key `'mag'` ❷. We store each magnitude in the variable `mag`, and then append it to the list `mags`.

We print the first 10 magnitudes, so we can see whether we're getting the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
```

Next, we'll pull the location data for each earthquake, and then we can make a map of the earthquakes.

Extracting Location Data

The location data is stored under the key `"geometry"`. Inside the geometry dictionary is a `"coordinates"` key, and the first two values in this list are the longitude and latitude. Here's how we'll pull this data:

eq_explore_data.py

```
--snip--
all_eq_dicts = all_eq_data['features']

mags, lons, lats = [], [], []
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag']
❶   lon = eq_dict['geometry']['coordinates'][0]
    lat = eq_dict['geometry']['coordinates'][1]
```

```
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)

print(mags[:10])
print(lons[:5])
print(lats[:5])
```

We make empty lists for the longitudes and latitudes. The code `eq_dict['geometry']` accesses the dictionary representing the geometry element of the earthquake ❶. The second key, `'coordinates'`, pulls the list of values associated with `'coordinates'`. Finally, the 0 index asks for the first value in the list of coordinates, which corresponds to an earthquake's longitude.

When we print the first five longitudes and latitudes, the output shows that we're pulling the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
[-116.7941667, -148.9865, -74.2343, -161.6801, -118.5316667]
[33.4863333, 64.6673, -12.1025, 54.2232, 35.3098333]
```

With this data, we can move on to mapping each earthquake.

Building a World Map

With the information we've pulled so far, we can build a simple world map. Although it won't look presentable yet, we want to make sure the information is displayed correctly before focusing on style and presentation issues. Here's the initial map:

eq_world_map.py

```
    import json

❶ from plotly.graph_objs import Scattergeo, Layout
    from plotly import offline
```

```
--snip--
for eq_dict in all_eq_dicts:
    --snip--

# Map the earthquakes.
❷ data = [Scattergeo(lon=lons, lat=lats)]
❸ my_layout = Layout(title='Global Earthquakes')

❹ fig = {'data': data, 'layout': my_layout}
offline.plot(fig, filename='global_earthquakes.html')
```

We import the `Scattergeo` chart type and the `Layout` class, and then import the `offline` module to render the map ❶. As we did when making a bar chart, we define a list called `data`. We create the `Scattergeo` object inside this list ❷, because you can plot more than one data set on any visualization you make. A `Scattergeo` chart type allows you to overlay a scatter plot of geographic data on a map. In the simplest use of this chart type, you only need to provide a list of longitudes and a list of latitudes.

We give the chart an appropriate title ❸ and create a dictionary called `fig` that contains the data and the layout ❹. Finally, we pass `fig` to the `plot()` function along with a descriptive filename for the output. When you run this file, you should see a map that looks like the one in Figure 16-7. Earthquakes usually occur near plate boundaries, which matches what we see in the chart.

Figure 16-7: A simple map showing where all the earthquakes in the last 24 hours occurred

We can do a lot of modifications to make this map more meaningful and easier to read, so let's make some of these changes.

A Different Way of Specifying Chart Data

Before we configure the chart, let's look at a slightly different way to specify the data for a Plotly chart. In the current chart, the `data` list is defined in one line:

```
data = [Scattergeo(lon=lons, lat=lats)]
```

This is one of the simplest ways to define the data for a chart in Plotly. But it's not the best way when you want to customize the presentation. Here's an equivalent way to define the data for the current chart:

```
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
}]
```

In this approach, all the information about the data is structured as key-value pairs in a dictionary. If you put this code into *eq_plot.py*, you'll see the same chart we just generated. This format allows us to specify customizations more easily than the previous format.

Customizing Marker Size

When we're figuring out how to improve the map's styling, we should focus on aspects of the data that we want to communicate more clearly. The current map shows the location of each earthquake, but it doesn't communicate the severity of any earthquake. We want viewers to immediately see where the most significant earthquakes occur in the world.

To do this, we'll change the size of markers depending on the magnitude of each earthquake:

eq_world_map.py

```
import json
--snip--
# Map the earthquakes.
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❶   'marker': {
❷       'size': [5*mag for mag in mags],
    },
}]
my_layout = Layout(title='Global Earthquakes')
--snip--
```

Plotly offers a huge variety of customizations you can make to a data series, each of which can be expressed as a key-value pair. Here we're using the key `'marker'` to specify how big each marker on the map should be

❶. We use a nested dictionary as the value associated with `'marker'`, because you can specify a number of settings for all the markers in a series.

We want the size to correspond to the magnitude of each earthquake. But if we just pass in the `mags` list, the markers would be too small to easily see the size differences. We need to multiply the magnitude by a scale factor to get an appropriate marker size. On my screen, a value of 5 works well; a slightly smaller or larger value might work better for your map. We use a list comprehension, which generates an appropriate marker size for each value in the `mags` list ❷.

When you run this code, you should see a map that looks like the one in Figure 16-8. This is a much better map, but we can still do more.

Figure 16-8: The map now shows the magnitude of each earthquake.

Customizing Marker Colors

We can also customize each marker's color to provide some classification to the severity of each earthquake. We'll use Plotly's colorscales to do this. Before you make these changes, copy the file *eq_data_30_day_m1.json* to your data directory. This file includes earthquake data for a 30-day period, and the map will be much more interesting to look at using this larger data set.

Here's how to use a colorscale to represent the magnitude of each earthquake:

eq_world_map.py

```
--snip--
❶ filename = 'data/eq_data_30_day_m1.json'
--snip--
# Map the earthquakes.
data = [{
    --snip--

    'marker': {
        'size': [5*mag for mag in mags],
❷       'color': mags,
❸       'colorscale': 'Viridis',
❹       'reversescale': True,
❺       'colorbar': {'title': 'Magnitude'},
    },
}]
--snip--
```

Be sure to update the filename so you're using the 30-day data set ❶. All the significant changes here occur in the `'marker'` dictionary, because we're only modifying the markers' appearance. The `'color'` setting tells Plotly what values it should use to determine where each marker falls on the colorscale ❷. We use the `mags` list to determine the color that's used. The `'colorscale'` setting tells Plotly which range of colors to use: `'Viridis'` is a colorscale that ranges from dark blue to bright yellow and works well for this data set ❸. We set `'reversescale'` to `True`, because we want to use bright yellow for the lowest values and dark blue for the most severe earthquakes ❹. The `'colorbar'` setting allows us to control the appearance of the colorscale shown on the side of the map. Here we title the colorscale `'Magnitude'` to make it clear what the colors represent ❹.

When you run the program now, you'll see a much nicer-looking map. In Figure 16-9, the colorscale shows the severity of individual earthquakes. Plotting this many earthquakes really makes it clear where the tectonic plate boundaries are!

Figure 16-9: In 30 days' worth of earthquakes, color and size are used to represent the magnitude of each earthquake.

Other Colorscales

You can also choose from a number of other colorscales. To see the available colorscales, save the following short program as *show_color_scales.py*:

show_color_scales.py

```
from plotly import colors

for key in colors.PLOTLY_SCALES.keys():
    print(key)
```

Plotly stores the colorscales in the `colors` module. The colorscales are defined in the dictionary `PLOTLY_SCALES`, and the names of the colorscales serve as the keys in the dictionary. Here's the output showing all the available colorscales:

```
Greys
YlGnBu
Greens
--snip--
Viridis
```

Feel free to try out these colorscales; remember that you can reverse any of these scales using the `reversescale` setting.

> **NOTE**
>
> *If you print the `PLOTLY_SCALES` dictionary, you can see how colorscales are defined. Every scale has a beginning color and an end color, and some scales have one or more intermediate colors defined as well. Plotly interpolates shades between each of these defined colors.*

Adding Hover Text

To finish this map, we'll add some informative text that appears when you hover over the marker representing an earthquake. In addition to showing the longitude and latitude, which appear by default, we'll show the magnitude and provide a description of the approximate location as well.

To make this change, we need to pull a little more data from the file and add it to the dictionary in `data` as well:

eq_world_map.py

```
--snip--
❶ mags, lons, lats, hover_texts = [], [], [], []
for eq_dict in all_eq_dicts:
    --snip--
    lat = eq_dict['geometry']['coordinates'][1]
❷   title = eq_dict['properties']['title']
```

```
        mags.append(mag)
        lons.append(lon)
        lats.append(lat)
        hover_texts.append(title)
    --snip--

    # Map the earthquakes.
    data = [{
        'type': 'scattergeo',
        'lon': lons,
        'lat': lats,
❸       'text': hover_texts,
        'marker': {
            --snip--
        },
    }]
    --snip--
```

We first make a list called `hover_texts` to store the label we'll use for each marker ❶. The "title" section of the earthquake data contains a descriptive name of the magnitude and location of each earthquake in addition to its longitude and latitude. At ❷ we pull this information and assign it to the variable `title`, and then append it to the list `hover_texts`.

When we include the key `'text'` in the `data` object, Plotly uses this value as a label for each marker when the viewer hovers over the marker. When we pass a list that matches the number of markers, Plotly pulls an individual label for each marker it generates ❸. When you run this program, you should be able to hover over any marker, see a description of where that earthquake took place, and read its exact magnitude.

This is impressive! In approximately 40 lines of code, we've created a visually appealing and meaningful map of global earthquake activity that also illustrates the geological structure of the planet. Plotly offers a wide range of ways you can customize the appearance and behavior of your vi-

sualizations. Using Plotly's many options, you can make charts and maps that show exactly what you want them to.

TRY IT YOURSELF

16-6. Refactoring: The loop that pulls data from `all_eq_dicts` uses variables for the magnitude, longitude, latitude, and title of each earthquake before appending these values to their appropriate lists. This approach was chosen for clarity in how to pull data from a JSON file, but it's not necessary in your code. Instead of using these temporary variables, pull each value from `eq_dict` and append it to the appropriate list in one line. Doing so should shorten the body of this loop to just four lines.

16-7. Automated Title: In this section, we specified the title manually when defining `my_layout`, which means we have to remember to update the title every time the source file changes. Instead, you can use the title for the data set in the metadata part of the JSON file. Pull this value, assign it to a variable, and use this for the title of the map when you're defining `my_layout`.

16-8. Recent Earthquakes: You can find data files containing information about the most recent earthquakes over 1-hour, 1-day, 7-day, and 30-day periods online. Go to *https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php* and you'll see a list of links to data sets for various time periods, focusing on earthquakes of different magnitudes. Download one of these data sets, and create a visualization of the most recent earthquake activity.

16-9. World Fires: In the resources for this chapter, you'll find a file called *world_fires_1_day.csv*. This file contains information about fires burning in different locations around the globe, including the latitude and longitude, and the brightness of each fire. Using the data processing work from the first part of this chapter and the mapping work from this section, make a map that shows which parts of the world are affected by fires.

You can download more recent versions of this data at *https://earthdata.nasa.gov/earth-observation-data/near-real-time/firms/active-fire-data/*. You can find links to the data in CSV format in the *TXT* section.

Summary

In this chapter, you learned to work with real-world data sets. You processed CSV and JSON files, and extracted the data you want to focus on. Using historical weather data, you learned more about working with Matplotlib, including how to use the `datetime` module and how to plot multiple data series on one chart. You plotted geographical data on a world map in Plotly and styled Plotly maps and charts as well.

As you gain experience working with CSV and JSON files, you'll be able to process almost any data you want to analyze. You can download most online data sets in either or both of these formats. By working with these formats, you'll be able to learn how to work with other data formats more easily as well.

In the next chapter, you'll write programs that automatically gather their own data from online sources, and then you'll create visualizations of that data. These are fun skills to have if you want to program as a hobby and critical skills if you're interested in programming professionally.