16
WORKING WITH CSV FILES AND JSON DATA

In Chapter 15, you learned how to extract text from PDF and Word documents. These files were in a binary format, which required special Python modules to access their data. CSV and JSON files, on the other hand, are just plaintext files. You can view them in a text editor, such as Mu. But Python also comes with the special `csv` and `json` modules, each providing functions to help you work with these file formats.

CSV stands for "comma-separated values," and CSV files are simplified spreadsheets stored as plaintext files. Python's `csv` module makes it easy to parse CSV files.

JSON (pronounced "JAY-sawn" or "Jason"—it doesn't matter how because either way people will say you're pronouncing it wrong) is a format that stores information as JavaScript source code in plaintext files. (JSON is short for JavaScript Object Notation.) You don't need to know the JavaScript programming language to use JSON files, but the JSON format is useful to know because it's used in many web applications.

The csv Module

Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. For example, the spreadsheet *example.xlsx* from *https://nostarch.com/automatestuff2/* would look like this in a CSV file:

```
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
4/8/2015 8:59,Oranges,52
4/10/2015 2:07,Apples,152
4/10/2015 18:10,Bananas,23
4/10/2015 2:40,Strawberries,98
```

I will use this file for this chapter's interactive shell examples. You can download *example.csv* from *https://nostarch.com/automatestuff2/* or enter the text into a text editor and save it as *example.csv*.

CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, CSV files:

- Don't have types for their values—everything is a string
- Don't have settings for font size or color
- Don't have multiple worksheets
- Can't specify cell widths and heights
- Can't have merged cells
- Can't have images or charts embedded in them

The advantage of CSV files is simplicity. CSV files are widely supported by many types of programs, can be viewed in text editors (including Mu), and are a straightforward way to represent spreadsheet data. The CSV format is exactly as advertised: it's just a text file of comma-separated values.

Since CSV files are just text files, you might be tempted to read them in as a string and then process that string using the techniques you learned in Chapter 9. For example, since each cell in a CSV file is separated by a comma, maybe you could just call `split(',')` on each line of text to get the comma-separated values as a list of strings. But not every comma in a CSV file represents the boundary between two cells. CSV files also have their own set of escape characters to allow commas and other characters to be included *as part of the values*. The `split()` method doesn't handle these escape characters. Because of these potential pitfalls, you should always use the `csv` module for reading and writing CSV files.

reader Objects

To read data from a CSV file with the `csv` module, you need to create a `reader` object. A `reader` object lets you iterate over lines in the CSV file. Enter the following into the interactive shell, with *example.csv* in the current working directory:

```
❶ >>> import csv
❷ >>> exampleFile = open('example.csv')
❸ >>> exampleReader = csv.reader(exampleFile)
❹ >>> exampleData = list(exampleReader)
❺ >>> exampleData
   [['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
   ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
   ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
   ['4/10/2015 2:40', 'Strawberries', '98']]
```

The `csv` module comes with Python, so we can import it ❶ without having to install it first.

To read a CSV file with the `csv` module, first open it using the `open()` function ❷, just as you would any other text file. But instead of calling the `read()` or `readlines()` method on the `File` object that `open()` returns, pass it to the `csv.reader()` function ❸. This will return a `reader` object for you to use. Note that you don't pass a filename string directly to the `csv.reader()` function.

The most direct way to access the values in the `reader` object is to convert it to a plain Python list by passing it to `list()` ❹. Using `list()` on this `reader` object returns a list of lists, which you can store in a variable like `exampleData`. Entering `exampleData` in the shell displays the list of lists ❺.

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression `exampleData[row][col]`, where `row` is the index of one of the lists in `exampleData`, and `col` is the index of the item you want from that list. Enter the following into the interactive shell:

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
>>> exampleData[0][2]
'73'
```

```
>>> exampleData[1][1]
'Cherries'
>>> exampleData[6][1]
'Strawberries'
```

As you can see from the output, `exampleData[0][0]` goes into the first list and gives us the first string, `exampleData[0][2]` goes into the first list and gives us the third string, and so on.

Reading Data from reader Objects in a for Loop

For large CSV files, you'll want to use the `reader` object in a `for` loop. This avoids loading the entire file into memory at once. For example, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
        print('Row #' + str(exampleReader.line_num) + ' ' + str(row))

Row #1 ['4/5/2015 13:34', 'Apples', '73']
Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
Row #4 ['4/8/2015 8:59', 'Oranges', '52']
Row #5 ['4/10/2015 2:07', 'Apples', '152']
Row #6 ['4/10/2015 18:10', 'Bananas', '23']
Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

After you import the `csv` module and make a `reader` object from the CSV file, you can loop through the rows in the `reader` object. Each row is a list of values, with each value representing a cell.

The `print()` function call prints the number of the current row and the contents of the row. To get the row number, use the `reader` object's `line_num` variable, which contains the number of the current line.

The `reader` object can be looped over only once. To reread the CSV file, you must call `csv.reader` to create a `reader` object.

writer Objects

A `writer` object lets you write data to a CSV file. To create a `writer` object, you use the `csv.writer()` function. Enter the following into the interactive shell:

```
>>> import csv
❶ >>> outputFile = open('output.csv', 'w', newline='')
❷ >>> outputWriter = csv.writer(outputFile)
>>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
21
>>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
32
>>> outputWriter.writerow([1, 2, 3.141592, 4])
16
>>> outputFile.close()
```

First, call `open()` and pass it `'w'` to open a file in write mode ❶. This will create the object you can then pass to `csv.writer()` ❷ to create a `writer` object.

On Windows, you'll also need to pass a blank string for the `open()` function's `newline` keyword argument. For technical reasons beyond the scope of this book, if you forget to set the `newline` argument, the rows in *output.csv* will be double-spaced, as shown in Figure 16-1.

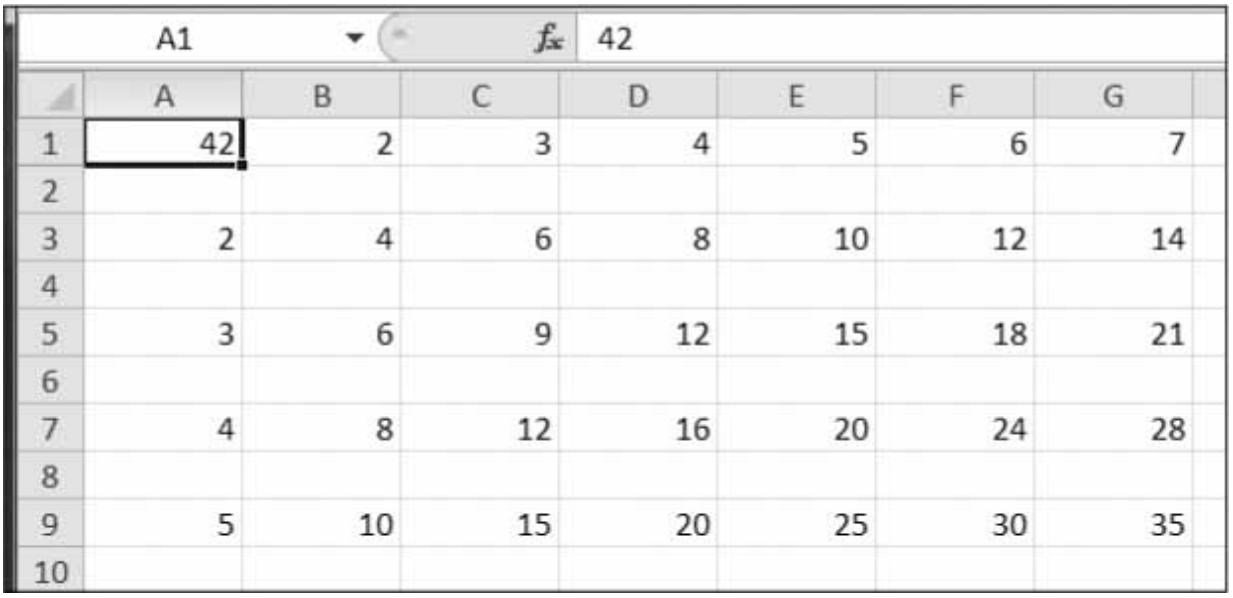

Figure 16-1: If you forget the `newline=''` keyword argument in `open()`, the CSV file will be double-spaced.

The `writerow()` method for `writer` objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file. The return value of `writerow()` is the

number of characters written to the file for that row (including newline characters).

This code produces an *output.csv* file that looks like this:

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

Notice how the `writer` object automatically escapes the comma in the value `'Hello, world!'` with double quotes in the CSV file. The `csv` module saves you from having to handle these special cases yourself.

The delimiter and lineterminator Keyword Arguments

Say you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced. You could enter something like the following into the interactive shell:

```
>>> import csv
>>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

This changes the delimiter and line terminator characters in your file. The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the `delimiter` and `lineterminator` keyword arguments with `csv.writer()`.

Passing `delimiter='\t'` and `lineterminator='\n\n'` ❶ changes the character between cells to a tab and the character between rows to two newlines. We then call `writerow()` three times to give us three rows.

This produces a file named *example.tsv* with the following contents:

```
apples  oranges grapes

eggs    bacon   ham

spam    spam    spam    spam    spam    spam
```

Now that our cells are separated by tabs, we're using the file extension *.tsv*, for tab-separated values.

DictReader and DictWriter CSV Objects

For CSV files that contain header rows, it's often more convenient to work with the `DictReader` and `DictWriter` objects, rather than the `reader` and `writer` objects.

The `reader` and `writer` objects read and write to CSV file rows by using lists. The `DictReader` and `DictWriter` CSV objects perform the same functions but use dictionaries instead, and they use the first row of the CSV file as the keys of these dictionaries.

Go to *https://nostarch.com/automatestuff2/* and download the *exampleWithHeader.csv* file. This file is the same as *example.csv* except it has Timestamp, Fruit, and Quantity as the column headers in the first row.

To read the file, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('exampleWithHeader.csv')
>>> exampleDictReader = csv.DictReader(exampleFile)
>>> for row in exampleDictReader:
...     print(row['Timestamp'], row['Fruit'], row['Quantity'])
...
4/5/2015 13:34 Apples 73
4/5/2015 3:41 Cherries 85
4/6/2015 12:46 Pears 14
4/8/2015 8:59 Oranges 52
4/10/2015 2:07 Apples 152
4/10/2015 18:10 Bananas 23
4/10/2015 2:40 Strawberries 98
```

Inside the loop, `DictReader` object sets `row` to a dictionary object with keys derived from the headers in the first row. (Well, technically, it sets `row` to an `OrderedDict` object,

which you can use in the same way as a dictionary; the difference between them is beyond the scope of this book.) Using a `DictReader` object means you don't need additional code to skip the first row's header information, since the `DictReader` object does this for you.

If you tried to use `DictReader` objects with *example.csv*, which doesn't have column headers in the first row, the `DictReader` object would use `'4/5/2015 13:34'`, `'Apples'`, and `'73'` as the dictionary keys. To avoid this, you can supply the `DictReader()` function with a second argument containing made-up header names:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleDictReader = csv.DictReader(exampleFile, ['time', 'name',
'amount'])
>>> for row in exampleDictReader:
...     print(row['time'], row['name'], row['amount'])
...
4/5/2015 13:34 Apples 73
4/5/2015 3:41 Cherries 85
4/6/2015 12:46 Pears 14
4/8/2015 8:59 Oranges 52
4/10/2015 2:07 Apples 152
4/10/2015 18:10 Bananas 23
4/10/2015 2:40 Strawberries 98
```

Because *example.csv*'s first row doesn't have any text for the heading of each column, we created our own: `'time'`, `'name'`, and `'amount'`.

`DictWriter` objects use dictionaries to create CSV files.

```
>>> import csv
>>> outputFile = open('output.csv', 'w', newline='')
>>> outputDictWriter = csv.DictWriter(outputFile, ['Name', 'Pet', 'Phone'])
>>> outputDictWriter.writeheader()
>>> outputDictWriter.writerow({'Name': 'Alice', 'Pet': 'cat', 'Phone': '555-
1234'})
20
>>> outputDictWriter.writerow({'Name': 'Bob', 'Phone': '555-9999'})
15
```

```
>>> outputDictWriter.writerow({'Phone': '555-5555', 'Name': 'Carol', 'Pet':
'dog'})
20
>>> outputFile.close()
```

If you want your file to contain a header row, write that row by calling `writeheader()`. Otherwise, skip calling `writeheader()` to omit a header row from the file. You then write each row of the CSV file with a `writerow()` method call, passing a dictionary that uses the headers as keys and contains the data to write to the file.

The *output.csv* file this code creates looks like this:

```
Name,Pet,Phone
Alice,cat,555-1234
Bob,,555-9999
Carol,dog,555-5555
```

Notice that the order of the key-value pairs in the dictionaries you passed to `writerow()` doesn't matter: they're written in the order of the keys given to `DictWriter()`. For example, even though you passed the `Phone` key and value before the `Name` and `Pet` keys and values in the fourth row, the phone number still appeared last in the output.

Notice also that any missing keys, such as `'Pet'` in `{'Name': 'Bob', 'Phone': '555-9999'}`, will simply be empty in the CSV file.

Project: Removing the Header from CSV Files

Say you have the boring job of removing the first line from several hundred CSV files. Maybe you'll be feeding them into an automated process that requires just the data and not the headers at the top of the columns. You *could* open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the *.csv* extension in the current working directory, read in the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

> **WARNING**

As always, whenever you write a program that modifies files, be sure to back up the files first, just in case your program does not work the way you expect it to. You don't want to accidentally erase your original files.

At a high level, the program must do the following:

1. Find all the CSV files in the current working directory.
2. Read in the full contents of each file.
3. Write out the contents, skipping the first line, to a new CSV file.

At the code level, this means the program will need to do the following:

1. Loop over a list of files from `os.listdir()`, skipping the non-CSV files.
2. Create a CSV `reader` object and read in the contents of the file, using the `line_num` attribute to figure out which line to skip.
3. Create a CSV `writer` object and write out the read-in data to the new file.

For this project, open a new file editor window and save it as *removeCsvHeader.py*.

Step 1: Loop Through Each CSV File

The first thing your program needs to do is loop over a list of all CSV filenames for the current working directory. Make your *removeCsvHeader.py* look like this:

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current

# working directory.

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Loop through every file in the current working directory.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
     ❶ continue    # skip non-csv files
```

```
    print('Removing header from ' + csvFilename + '...')

    # TODO: Read the CSV file in (skipping first row).

    # TODO: Write out the CSV file.
```

The `os.makedirs()` call will create a `headerRemoved` folder where all the headless CSV files will be written. A `for` loop on `os.listdir('.')` gets you partway there, but it will loop over *all* files in the working directory, so you'll need to add some code at the start of the loop that skips filenames that don't end with `.csv`. The `continue` statement ❶ makes the `for` loop move on to the next filename when it comes across a non-CSV file.

Just so there's *some* output as the program runs, print out a message saying which CSV file the program is working on. Then, add some `TODO` comments for what the rest of the program should do.

Step 2: Read in the CSV File

The program doesn't remove the first line from the CSV file. Rather, it creates a new copy of the CSV file without the first line. Since the copy's filename is the same as the original filename, the copy will overwrite the original.

The program will need a way to track whether it is currently looping on the first row. Add the following to *removeCsvHeader.py*.

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--

    # Read the CSV file in (skipping first row).
    csvRows = []
    csvFileObj = open(csvFilename)
    readerObj = csv.reader(csvFileObj)
    for row in readerObj:
        if readerObj.line_num == 1:
            continue    # skip first row
        csvRows.append(row)
```

```
    csvFileObj.close()

    # TODO: Write out the CSV file.
```

The `reader` object's `line_num` attribute can be used to determine which line in the CSV file it is currently reading. Another `for` loop will loop over the rows returned from the CSV `reader` object, and all rows but the first will be appended to `csvRows`.

As the `for` loop iterates over each row, the code checks whether `readerObj.line_num` is set to `1`. If so, it executes a `continue` to move on to the next row without appending it to `csvRows`. For every row afterward, the condition will be always be `False`, and the row will be appended to `csvRows`.

Step 3: Write Out the CSV File Without the First Row

Now that `csvRows` contains all rows but the first row, the list needs to be written out to a CSV file in the *headerRemoved* folder. Add the following to *removeCsvHeader.py*:

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--

# Loop through every file in the current working directory.
❶ for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue    # skip non-CSV files

    --snip--

    # Write out the CSV file.
    csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
                      newline='')
    csvWriter = csv.writer(csvFileObj)
    for row in csvRows:
        csvWriter.writerow(row)
    csvFileObj.close()
```

The CSV `writer` object will write the list to a CSV file in `headerRemoved` using `csvFilename` (which we also used in the CSV reader). This will overwrite the original file.

Once we create the `writer` object, we loop over the sublists stored in `csvRows` and write each sublist to the file.

After the code is executed, the outer `for` loop ❶ will loop to the next filename from `os.listdir('.')`. When that loop is finished, the program will be complete.

To test your program, download *removeCsvHeader.zip* from *https://nostarch.com/automatestuff2/* and unzip it to a folder. Run the *removeCsvHeader.py* program in that folder. The output will look like this:

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
--snip--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

This program should print a filename each time it strips the first line from a CSV file.

Ideas for Similar Programs

The programs that you could write for CSV files are similar to the kinds you could write for Excel files, since they're both spreadsheet files. You could write programs to do the following:

- Compare data between different rows in a CSV file or between multiple CSV files.
- Copy specific data from a CSV file to an Excel file, or vice versa.
- Check for invalid data or formatting mistakes in CSV files and alert the user to these errors.
- Read data from a CSV file as input for your Python programs.

JSON AND APIS

JavaScript Object Notation is a popular way to format data as a single human-readable string. JSON is the native way that JavaScript programs write their data structures and usually resembles what Python's `pprint()` function would produce. You don't need to know JavaScript in order to work with JSON-formatted data.

Here's an example of data formatted as JSON:

```
{"name": "Zophie", "isCat": true,
 "miceCaught": 0, "napsTaken": 37.5,
 "felineIQ": null}
```

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an *application programming interface (API)*. Accessing an API is the same as accessing any other web page via a URL. The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

Many websites make their data available in JSON format. Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs for programs to use. Some of these sites require registration, which is almost always free. You'll have to find documentation for what URLs your program needs to request in order to get the data you want, as well as the general format of the JSON data structures that are returned. This documentation should be provided by whatever site is offering the API; if they have a "Developers" page, look for the documentation there.

Using APIs, you could write programs that do the following:

- Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.)
- Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook.
- Create a "movie encyclopedia" for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

You can see some examples of JSON APIs in the resources at *https://nostarch.com/automatestuff2/*.

JSON isn't the only way to format data into a human-readable string. There are many others, including XML (eXtensible Markup Language), TOML (Tom's Obvious, Minimal Language), YML (Yet another Markup Language), INI (Initialization), or even the outdated ASN.1 (Abstract Syntax Notation One) formats, all of which provide a structure for representing data as human-readable text. This book won't cover these,

because JSON has quickly become the most widely used alternate format, but there are third-party Python modules that readily handle them.

The json Module

Python's `json` module handles all the details of translating between a string with JSON data and Python values for the `json.loads()` and `json.dumps()` functions. JSON can't store *every* kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`. JSON cannot represent Python-specific objects, such as `File` objects, CSV `reader` or `writer` objects, `Regex` objects, or Selenium `WebElement` objects.

Reading JSON with the loads() Function

To translate a string containing JSON data into a Python value, pass it to the `json.loads()` function. (The name means "load string," not "loads.") Enter the following into the interactive shell:

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,
"felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
>>> jsonDataAsPythonValue
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

After you import the `json` module, you can call `loads()` and pass it a string of JSON data. Note that JSON strings always use double quotes. It will return that data as a Python dictionary. Python dictionaries are not ordered, so the key-value pairs may appear in a different order when you print `jsonDataAsPythonValue`.

Writing JSON with the dumps() Function

The `json.dumps()` function (which means "dump string," not "dumps") will translate a Python value into a string of JSON-formatted data. Enter the following into the interactive shell:

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie',
'felineIQ': None}
>>> import json
```

```
>>> stringOfJsonData = json.dumps(pythonValue)
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

The value can only be one of the following basic Python data types: dictionary, list, integer, float, string, Boolean, or `None`.

Project: Fetching Current Weather Data

Checking the weather seems fairly trivial: Open your web browser, click the address bar, type the URL to a weather website (or search for one and then click the link), wait for the page to load, look past all the ads, and so on.

Actually, there are a lot of boring steps you could skip if you had a program that downloaded the weather forecast for the next few days and printed it as plaintext. This program uses the `requests` module from Chapter 12 to download data from the web.

Overall, the program does the following:

1. Reads the requested location from the command line
2. Downloads JSON weather data from OpenWeatherMap.org
3. Converts the string of JSON data to a Python data structure
4. Prints the weather for today and the next two days

So the code will need to do the following:

1. Join strings in `sys.argv` to get the location.
2. Call `requests.get()` to download the weather data.
3. Call `json.loads()` to convert the JSON data to a Python data structure.
4. Print the weather forecast.

For this project, open a new file editor window and save it as *getOpenWeather.py*. Then visit *https://openweathermap.org/api/* in your browser and sign up for a free account to obtain an *API key*, also called an app ID, which for the OpenWeatherMap service is a string code that looks something like `'30144aba38018987d84710d0e319281e'`. You don’t need to pay for this service unless you plan on making more than 60 API calls per minute. Keep the API key secret; anyone who knows it can write scripts that use your account’s usage quota.

Step 1: Get Location from the Command Line Argument

The input for this program will come from the command line. Make *getOpenWeather.py* look like this:

```
#! python3
# getOpenWeather.py - Prints the weather for a location from the command line.

APPID = 'YOUR_APPID_HERE'

import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: getOpenWeather.py city_name, 2-letter_country_code')
    sys.exit()
location = ' '.join(sys.argv[1:])

# TODO: Download the JSON data from OpenWeatherMap.org's API.

# TODO: Load JSON data into a Python variable.
```

In Python, command line arguments are stored in the `sys.argv` list. The `APPID` variable should be set to the API key for your account. Without this key, your requests to the weather service will fail. After the `#!` shebang line and `import` statements, the program will check that there is more than one command line argument. (Recall that `sys.argv` will always have at least one element, `sys.argv[0]`, which contains the Python script's filename.) If there is only one element in the list, then the user didn't provide a location on the command line, and a "usage" message will be provided to the user before the program ends.

The OpenWeatherMap service requires that the query be formatted as the city name, a comma, and a two-letter country code (like "US" for the United States). You can find a list of these codes at *https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2*. Our script displays the weather for the first city listed in the retrieved JSON text. Unfortunately, cities that share a name, like Portland, Oregon, and Portland, Maine, will both be included, though the JSON text will include longitude and latitude information to differentiate between the cities.

Command line arguments are split on spaces. The command line argument `San Francisco, US` would make `sys.argv` hold `['getOpenWeather.py', 'San', 'Francisco,', 'US']`. Therefore, call the `join()` method to join all the strings except for the first in `sys.argv`. Store this joined string in a variable named `location`.

Step 2: Download the JSON Data

OpenWeatherMap.org provides real-time weather information in JSON format. First you must sign up for a free API key on the site. (This key is used to limit how frequently you make requests on their server, to keep their bandwidth costs down.) Your program simply has to download the page at *https://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3&APPID=<APIkey>*, where *<Location>* is the name of the city whose weather you want and *<API key>* is your personal API key. Add the following to *getOpenWeather.py*.

```
#! python3
# getOpenWeather.py - Prints the weather for a location from the command line.

--snip--

# Download the JSON data from OpenWeatherMap.org's API.
url ='https://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3&APPID=%s ' %
(location,
APPID)
response = requests.get(url)
response.raise_for_status()

# Uncomment to see the raw JSON text:
#print(response.text)

# TODO: Load JSON data into a Python variable.
```

We have `location` from our command line arguments. To make the URL we want to access, we use the `%s` placeholder and insert whatever string is stored in `location` into that spot in the URL string. We store the result in `url` and pass `url` to `requests.get()`. The `requests.get()` call returns a `Response` object, which you can check for errors by calling

`raise_for_status()`. If no exception is raised, the downloaded text will be in `response.text`.

Step 3: Load JSON Data and Print Weather

The `response.text` member variable holds a large string of JSON-formatted data. To convert this to a Python value, call the `json.loads()` function. The JSON data will look something like this:

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
          'country': 'United States of America',
          'id': '5391959',
          'name': 'San Francisco',
          'population': 0},
 'cnt': 3,
 'cod': '200',
 'list': [{'clouds': 0,
           'deg': 233,
           'dt': 1402344000,
           'humidity': 58,
           'pressure': 1012.23,
           'speed': 1.96,
           'temp': {'day': 302.29,
                    'eve': 296.46,
                    'max': 302.29,
                    'min': 289.77,
                    'morn': 294.59,
                    'night': 289.77},
           'weather': [{'description': 'sky is clear',
                        'icon': '01d',
--snip--
```

You can see this data by passing `weatherData` to `pprint.pprint()`. You may want to check *https://openweathermap.org/* for more documentation on what these fields mean. For example, the online documentation will tell you that the `302.29` after `'day'` is the daytime temperature in Kelvin, not Celsius or Fahrenheit.

The weather descriptions you want are after `'main'` and `'description'`. To neatly print them out, add the following to *getOpenWeather.py*.

```
! python3
# getOpenWeather.py - Prints the weather for a location from the command line.

--snip--

# Load JSON data into a Python variable.
weatherData = json.loads(response.text)

# Print weather descriptions.
❶ w = weatherData['list']
print('Current weather in %s:' % (location))
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])
print()
print('Tomorrow:')
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])
print()
print('Day after tomorrow:')
print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

Notice how the code stores `weatherData['list']` in the variable `w` to save you some typing ❶. You use `w[0]`, `w[1]`, and `w[2]` to retrieve the dictionaries for today, tomorrow, and the day after tomorrow's weather, respectively. Each of these dictionaries has a `'weather'` key, which contains a list value. You're interested in the first list item, a nested dictionary with several more keys, at index 0. Here, we print the values stored in the `'main'` and `'description'` keys, separated by a hyphen.

When this program is run with the command line argument `getOpenWeather.py San Francisco, CA`, the output looks something like this:

```
Current weather in San Francisco, CA:
Clear - sky is clear

Tomorrow:
Clouds - few clouds
```

```
Day after tomorrow:
Clear - sky is clear
```

(The weather is one of the reasons I like living in San Francisco!)

Ideas for Similar Programs

Accessing weather data can form the basis for many types of programs. You can create similar programs to do the following:

- Collect weather forecasts for several campsites or hiking trails to see which one will have the best weather.
- Schedule a program to regularly check the weather and send you a frost alert if you need to move your plants indoors. (Chapter 17 covers scheduling, and Chapter 18 explains how to send email.)
- Pull weather data from multiple sites to show all at once, or calculate and show the average of the multiple weather predictions.

Summary

CSV and JSON are common plaintext formats for storing data. They are easy for programs to parse while still being human readable, so they are often used for simple spreadsheets or web app data. The `csv` and `json` modules greatly simplify the process of reading and writing to CSV and JSON files.

The last few chapters have taught you how to use Python to parse information from a wide variety of file formats. One common task is taking data from a variety of formats and parsing it for the particular information you need. These tasks are often specific to the point that commercial software is not optimally helpful. By writing your own scripts, you can make the computer handle large amounts of data presented in these formats.

In Chapter 18, you'll break away from data formats and learn how to make your programs communicate with you by sending emails and text messages.

Practice Questions

1. What are some features Excel spreadsheets have that CSV spread-sheets don't?
2. What do you pass to `csv.reader()` and `csv.writer()` to create `reader` and `writer` objects?

3. What modes do `File` objects for `reader` and `writer` objects need to be opened in?
4. What method takes a list argument and writes it to a CSV file?
5. What do the `delimiter` and `lineterminator` keyword arguments do?
6. What function takes a string of JSON data and returns a Python data structure?
7. What function takes a Python data structure and returns a string of JSON data?

Practice Project

For practice, write a program that does the following.

Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the `openpyxl` module from Chapter 12, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

A single Excel file might contain multiple sheets; you'll have to create one CSV file per *sheet*. The filenames of the CSV files should be *<excel filename>_<sheet title>.csv*, where *<excel filename>* is the filename of the Excel file without the file extension (for example, `'spam_data'`, not `'spam_data.xlsx'`) and *<sheet title>* is the string from the `Worksheet` object's `title` variable.

This program will involve many nested `for` loops. The skeleton of the program will look something like this:

```
for excelFile in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheetName in wb.get_sheet_names():
        # Loop through every sheet in the workbook.
        sheet = wb.get_sheet_by_name(sheetName)

        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for rowNum in range(1, sheet.max_row + 1):
            rowData = []    # append each cell to this list
```

```
        # Loop through each cell in the row.
        for colNum in range(1, sheet.max_column + 1):
            # Append each cell's data to rowData.

        # Write the rowData list to the CSV file.

    csvFile.close()
```

Download the ZIP file *excelSpreadsheets.zip* from *https://nostarch.com/automatestuff2/* and unzip the spreadsheets into the same directory as your program. You can use these as the files to test the program on.