IN THIS CHAPTER

» Working with text and binary files

» Opening, closing, and reading the contents of a file

» Looping through a file

» Reading and copying a binary file

» Working with CSV files

» Importing CSV to objects and dictionaries

Chapter 1

Working with External Files

COPYRIGHTED MATERIAL

Pretty much everything stored in your computer, be it a document, program, movie, photograph, and more, is stored in a file. Most files are organized into *folders* (also called *directories*). You can browse through folders and files by using Finder (on a Mac) or File Explorer or Windows Explorer (in Windows).

Python offers many tools for creating, reading from, and writing to many different kinds of files. In this chapter, you learn the most important skills for using Python code to work with files.

Understanding Text and Binary Files

There are basically two types of files:

» **Text file:** Contains plain text characters. When you open a text file in a text editor, it displays human-readable content. The text may not be in a language

you know or understand, but you will see mostly normal characters that you can type at any keyboard.

» **Binary file:** Stores information in bytes that aren't quite so human readable. If you open the binary file in a text editor, what you see may resemble Figure 1-1. (We don't recommend that you do this.)

endstream
endobj
1111 0 obj
<</Filter/FlateDecode/Length 910>>stream

FIGURE 1-1: How a binary files looks in a program for editing text files.

WARNING

If you open a binary file in a text editor and see this gobbledygook, don't panic. Just close the file or program and choose No if asked to save it. The file will be fine, as long as you don't save it.

Figure 1-2 lists examples of different kinds of text and binary files, some of which you may have worked with before. Other files types are available; these are among the most widely used.

Text File

- **Plain text**: .txt, .csv
- **Source code:** .py, .html, .css, .js
- **Data**: .json, .xml

Binary File

- **Executable**: .exe, .dmg, .bin
- **Images**: .jpg, .png, .gif, .tiff, .ico
- **Video**: .mp4, .m4v, .mp4, .mov
- **Audio**: .aif, .mp3, .mpa, .wav
- **Compressed**: .zip, .deb, .tar.gz
- **Font**: .woff, .otf, .ttf
- **Document**: .pdf, .docx, .xlsx

FIGURE 1-2: Common text and binary files.

As with any Python code, you can use a Jupyter notebook, VS Code, or almost any coding editor to write your Python code. We use VS Code in this chapter simply because its Explorer pane (on the left, when it's open) displays the contents of the folder in which you're currently working.

Opening and Closing Files

To open a file from a Python app, use this syntax:

```
open(filename.ext[,mode])
```

Replace *filename.ext* with the filename of the file you want to open. If the file is not in the same directory as the Python code, you need to specify a path to the file, . For example, if you're using Windows and you want to open the foo.txt on your desktop and your user account name is Alan, you'd use the path C:/Users/Alan/Desktop/foo.txt rather than the more common Windows syntax with backslashes (C:\Users\Alan\Desktop\foo.txt).

The *,mode* is optional (as indicated by the square brackets). Use it to specify what kind of access you want your app to have, using the following single-character abbreviations:

- **r: (Read):** Opens the file but does not allow Python to make any changes. This is the default mode and is used if you don't specify a mode. If the file doesn't exist, Python raises a FileNotFoundError exception.
- **r+: (Read/Write):** Opens the file and allows Python to read and write to the file.
- **a: (Append):** Opens the file and allows Python to add content to the end of the file but not change existing content. If the file doesn't exist, this mode creates the file.
- **w: (Write):** Opens the file and overwrites its contents, or creates the file if it doesn't exist.
- **x: (Create):** Creates the file if it doesn't already exist. If the file does exist, it raises a FileExistsError exception.

REMEMBER

For more information on exceptions, see Book 2, Chapter 7.

You can also specify the type of file you're opening or creating. If you already specified one of the preceding modes, just add this specification as another letter. If you use just one of the following letters on its own, the file opens in Read mode:

- **t: (Text):** Opens the file as a text file and allows Python to read and write text.
- **b: (Binary):** Opens the file as a binary file and allows Python to read and write bytes.

You can use the `open` method in basically two ways. With one syntax, you assign a variable name to the file, and you use this variable name in code to refer to the file:

```
var = open(filename.ext[,mode])
```

Replace *var* with a name of your choosing (though it's common in Python to use just the letter `f` as the name).

After the file is open, you can access its content in a few ways, as we discuss a little later in the chapter. For now, we simply copy everything in the file to a variable named *filecontents*, and then we display this content using a simple `print()` function. So to open `quotes.txt`, read in all its content, and display that content on the screen, use this code:

```
f = open('quotes.txt')
filecontents = f.read()
print(filecontents)
```

With this method, the file remains open until you specifically close it using the file variable name and the `.close()` method, like this:

```
f.close()
```

REMEMBER

Make sure that your apps close any files they no longer need open. Failure to do so allows open file handlers to accumulate, which can eventually cause the app to throw an exception and crash, perhaps even corrupting some of the open files along the way.

The second way to open a file is by using a context manager or contextual coding. *Contextual coding* starts with the word `with`. You still assign a variable name, but you do so near the end of the line. The last thing on the line is a colon, which marks the beginning of the `with` block. All indented code below that is assumed to be relevant to the context of the open file (like code indented inside a loop). At the end of contextual coding, you don't need to close the file because Python does it automatically:

```
# ---------------- Contextual syntax
with open('quotes.txt') as f:
    filecontents = f.read()
    print(filecontents)

# The unindented line below is outside the with... block;
print('File is closed: ', f.closed)
```

The following code shows a single app that opens `quotes.txt`, reads and displays its content, and then closes the file. With the first method, you have to use `.close()` to close the file. With the second method, the file closes automatically, so no `.close()` is required:

```
# - Basic syntax to open, read, and display file contents.
f = open('quotes.txt')
filecontents = f.read()
print(filecontents)
# Returns True if the file is closed, otherwise False.
print('File is closed: ', f.closed)

# Closes the file.
f.close() # Close the file.
print() # Print a blank line.

# ---------------- Contextual syntax
with open('quotes.txt') as f:
    filecontents = f.read()
    print(filecontents)

# The unindented line below is outside the with... block;
print('File is closed: ', f.closed)
```

The output of this app follows:

```
I've had a perfectly wonderful evening, but this wasn't it.
Groucho Marx
The difference between stupidity and genius is that genius has its limits.
Albert Einstein
We are all here on earth to help others; what on earth the others are here
for, I have no idea.
W. H. Auden
Ending a sentence with a preposition is something up with which I will not
put.
Winston Churchill

File is closed:  False
```

```
I've had a perfectly wonderful evening, but this wasn't it.
Groucho Marx
The difference between stupidity and genius is that genius has its limits.
Albert Einstein
We are all here on earth to help others; what on earth the others are here
for, I have no idea.
W. H. Auden
Ending a sentence with a preposition is something up with which I will not
put.
Winston Churchill

File is closed:  True
```

(We can't vouch that these famous quotes were actually said by the people shown.) At the end of the first output, `.closed` is `False` because it's tested before the `close()` closes the file. At the end of the second output, `.closed` is `True` without executing a `.close()` because leaving the code indented under the `with:` line closes the file automatically.

For the rest of this chapter, we stick with contextual syntax because it's generally the preferred and recommended syntax and a good habit to acquire right from the start.

The previous example works fine because `quotes.txt` is a simple text file that contains only ASCII characters — the kinds of letters, numbers, and punctuation marks that you can type from a standard keyboard for the English language. Now consider the following code, which attempts to open a `.jpg` file, which is a graphic image, not a text file:

```
with open('happy_pickle.jpg') as f:
    filecontents = f.read()
    print(filecontents)
```

Attempting to run that code results in the following error:

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 40:
character maps to <undefined>
```

This message isn't the most helpful one in the world. Suppose we try to open `names.txt`, which (one would assume) is a text file like `quotes.txt`, using this code:

```
with open('names.txt') as f:
    filecontents = f.read()
    print(filecontents)
```

Running this code could produce another strange error message, worded something like this:

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x81 in position 45:
character maps to <undefined>
```

What the heck is going on here?

The first problem is caused because the file type is `.jpg`, a graphic image, which means the file is a binary file, not a text file. To open a `.jpg` file, you need `b` in the mode. Or use `rb`, which means *read binary,* like this:

```
with open('happy_pickle.jpg', 'rb') as f:
    filecontents = f.read()
    print(filecontents)
```

Running this code doesn't generate an error. But what it does display doesn't look anything like the actual picture:

```
\x07~}\xba\xe7\xd2\x8c\x00\x0e|\xbd\xa8\x12l+\xca\xf7\xae\xa5\x9e^\x8d\x89
\x7f\xde\xb4f>\x98\xc7\xfc\xcf46d\xcf\x1c\xd0\xa6\x98m$\xb6(U\x8c\xa6\x83
\x19\x17\xa6>\xe6\x94\x96|g\'4\xab\xdd\xb8\xc8=\xa9[\x8b\xcc`\x0e8\xa3
\xb0;\xc6\xe6\xbb(I.\xa3\xda\x91\xb8\xbd\xf2\x97\xdf\xc1\xf4\xefI\xcdy
\x97d\x1e`;\xf64\x94\xd7\x03
```

If we open `happy_pickle.jpg` in a graphics app or in VS Code, it looks nothing like that gibberish. Instead, it looks like Figure 1-3.

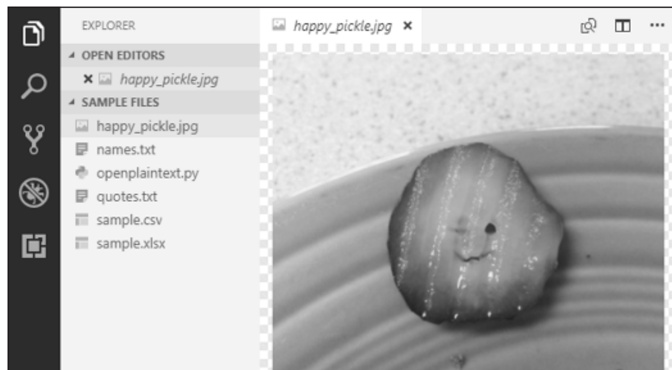

FIGURE 1-3: How `happy_pickle.jpg` is supposed to look.

So why does the file look so messed up in Python? The `print()` function displays the raw bytes that make up the file. Displaying raw bytes isn't a problem or an issue; it's just not a good way to work with a `.jpg` file right now.

The problem with `names.txt` is different. That file is a text file (`.txt`), just like `quotes.txt`. But if you open it and look at its contents, as in Figure 1-4, you'll see that it has a lot of unusual characters that you don't normally see in ASCII (the numbers, letters, and punctuation marks on your keyboard).

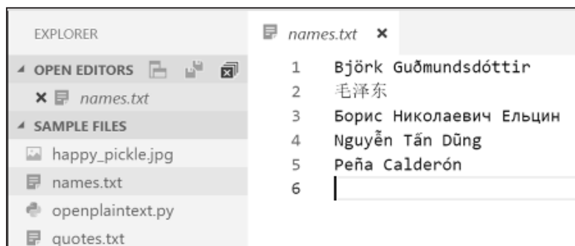

FIGURE 1-4: The `Names.txt` file is text, but with lots of non-English characters.

All those fancy-looking characters tell you that `names.txt` is not a simple ASCII text file. More likely it's a UTF-8 file, which is basically a text file that uses more than the standard ASCII text characters. To open this file, you have to tell Python to expect UTF-8 characters by using `encoding='utf-8'` in the `open()` statement, as shown in Figure 1-5. The output matches the contents of the `names.txt` file.

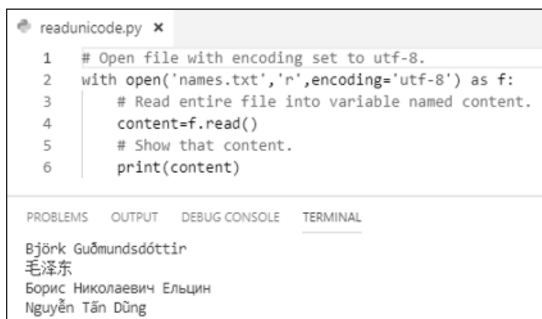

FIGURE 1-5: Contents of `names.txt` displayed.

When opening a file, you need to be aware of three things:

- For a plain-text file (ASCII), you can use `r` or nothing as the mode.
- For a binary file, you must specify `b` in the mode.
- For a text file with fancy characters, you most likely need to open it as a text file with `encoding` set to `utf-8` in the `open()` statement.

WHAT IS UTF-8?

UTF-8 is short for Unicode Transformation Format, 8-bit, and is a standardized way to represent letters and numbers on computers. The original ASCII set of characters, which contains mostly uppercase and lowercase letters, numbers, and punctuation marks, worked okay in the early days of computing. But when other languages were brought into the mix, these characters were just not enough. Many standards for dealing with other languages have been proposed and accepted over the years. Of those, UTF-8 has steadily grown in use whereas most others declined. Today, UTF-8 is pretty much the standard for all things internet, so it's a good choice if you have to choose a character set for a project.

If you get stuck trying to open a file that's supposed to be UTF-8 but isn't cooperating, do a web search for *convert file to utf-8 encoding.* Then look for a web page or an app that will work with your operating system to make the conversion.

Reading a File's Contents

Previously in this chapter, you saw how you can use `.read()` to read the contents of an open file. But that's not the only way to read a file. You have three choices:

- `read([size])`: Reads the entire file if you leave the parentheses empty. If you specify a size inside the parentheses, it reads that many characters (for a text file) or that many bytes (for a binary file).
- `readline()`: Reads one line of the contents from a text file — the line ends wherever there's a newline character. (The newline character, `\n`, ends the line that's displayed and moves the cursor down to the next line.)
- `readlines()`: Reads all the lines of a text file into a list.

TECHNICAL STUFF

People don't type binary files, so any newline characters in a binary file are arbitrary. Therefore, `readline()` and `readlines()` are useful only for text files.

Both the `read()` and `readline()` methods read in the entire file simultaneously. The only difference is that `read` reads in the file as one big chunk of data, whereas `readlines()` reads in the file one line at a time and stores each line as an item in a list. For example, the following code opens `quotes.txt`, reads in all the contents, and then displays it:

```
with open('quotes.txt') as f:
    # Read in entire file
```

Working with External Files

```
    content = f.read()
    print(content)
```

The `content` variable stores a copy of everything from the text file. We print the variable to display its contents. The newline character at the end of each line in the file starts a new line on the screen when printing.

Here is the same code using `readlines()` rather than `read`:

```
with open('quotes.txt') as f:
    content = f.readlines()
    print(content)
```

The output from this code is

```
["I've had a perfectly wonderful evening, but this wasn't it.\n",
'Groucho Marx\n', 'The difference between stupidity and genius is that
genius has its limits.\n', 'Albert Einstein\n', 'We are all here on earth to
help others; what on earth the others are here for, I have no idea.\n',
'W. H. Auden\n', 'Ending a sentence with a preposition is something up with
which I will not put.\n', 'Winston Churchill\n']
```

The square brackets surrounding the output tell you that it's a list. Each item in the list is surrounded by quotation marks and separated by commas. The `\n` at the end of each item is the newline character that ends the line in the file.

Unlike `readlines()` (plural), `readline()` reads just one line from the file. The line extends from the current position in the file to the next newline character. Executing another `readline()` reads the next line in the file, and so forth. For example, suppose you run this code:

```
with open('quotes.txt') as f:
    content = f.readline()
    print(content)
```

The output is

```
I've had a perfectly wonderful evening, but this wasn't it.
```

Executing another `readline()` after this would read the next line. As you may guess, when it comes to `readline()` and `readlines()`, you're likely to want to use loops to access all the data in a way that gives you more control.

Looping through a File

You can loop through a file using either `readlines()` or `readline()`. The `readlines()` method always reads in the file as a whole. So if the file is very large, your computer may run out of memory (RAM) before the file has been read in. But if you know the size of the file and it's relatively small (maybe a few thousand or fewer rows of data), `readlines()` is a speedy way to get all the data. That data will be in a list, so you will loop through the list rather than the file. You can also loop through binary files, but they don't have lines of text as text files do. So binary files are read in chunks, as you'll see at the end of this section.

Looping with readlines()

When you read a file with `readlines()`, you read the entire file in one fell swoop as a list. So you don't really loop through the file one row at a time. Rather, you loop through the list of items that `readlines()` stores in memory. The code to do so looks like this:

```
with open('quotes.txt') as f:
    # Reads in all lines first, then loops through.
    for one_line in f.readlines():
        print(one_line)
```

If you run this code, the output will be double-spaced because each list item ends with a newline, and then `print` normally adds its own newline with each pass through the loop. If you want to retain the single spacing, add `end=''` to the `print` statement (make sure you use two single or double quotation marks with no spaces after =). Here's an example:

```
with open('quotes.txt') as f:
    # Reads in all lines first, then loops through.
    for one_line in f.readlines():
        print(one_line, end='')
```

The output from this code follows:

```
I've had a perfectly wonderful evening, but this wasn't it.
Groucho Marx
The difference between stupidity and genius is that genius has its limits.
Albert Einstein
We are all here on earth to help others; what on earth the others are here
for, I have no idea.
W. H. Auden
```

```
Ending a sentence with a preposition is something up with which I will not
put.
Winston Churchill
```

Suppose you're happy with that output but want to improve it slightly. You want to indent the name below each quote a couple of spaces and add a blank line below the name. How could you do that? Well, Python has a built-in `enumerate()` function that, when used with a list, counts the number of passes through the loop, starting at `0`. So instead of the `for` loop shown in the preceding example, you write `for one_line in enumerate(f.readlines()):`. With each pass through the loop, `one_line[0]` contains the number of that line, `one_line[1]` contains its contents (the text of the line), and you can see whether the counter is an even number using the modulus operator, `%`, which returns the remainder after division. So when you calculate `% 2` (modulo 2) for an even number, you always get `0`. An odd number will always return 1 as the remainder when divided by 2. So you could write the code this way:

```
with open('quotes.txt') as f:
    # Reads in all lines first, then loops through.
    # Count each line starting at zero.
    for one_line in enumerate(f.readlines()):
        # If counter is even number, print with no extra newline
        if one_line[0] % 2 == 0:
            print(one_line[1], end='')
        # Otherwise print a couple spaces and an extra newline.
        else:
            print('  ' + one_line[1])
```

The output is as follows:

```
I've had a perfectly wonderful evening, but this wasn't it.
  Groucho Marx

The difference between stupidity and genius is that genius has its limits.
  Albert Einstein

We are all here on earth to help others; what on earth the others are here
for, I have no idea.
  W. H. Auden

Ending a sentence with a preposition is something up with which I will not
put.
  Winston Churchill
```

Looping with readline()

If you aren't too sure about the size of the file you're reading or the amount of RAM in the computer running your app, using `readlines()` to read in an entire file can be risky. If there isn't enough memory to hold the entire file, the app will crash when it runs out of memory. To play it safe, you can loop through the file one line at a time so that only one line of the contents from the file is in memory at any given time.

To use this method, you open the file, read one line, and put it in a variable. Then loop through the file *as long as* (`while`) the variable isn't empty. Because each line in the file contains some text, the variable won't be empty until after the last line is read. Here is the code for this approach to looping:

```
with open('quotes.txt') as f:
    one_line = f.readline()
    while one_line:
        print(one_line, end='')
        one_line = f.readline()
```

For larger files, this method is the way to go because at no point are you reading in the entire file. The only potential problem is forgetting to include `.readline()` inside the loop to advance to the next row. If you forget the `readline()`, you end up in an infinite loop that prints the first line over and over. If you ever find yourself in this situation, press Ctrl+C in the Terminal pane where the code is running to stop the loop.

You can accomplish the same format, in which you indent the name under each quote and add a blank line, by using `.readline()` in Python. In your code, start a counter at `1`. Create a loop that reads one row at a time from the text file. Within that loop, increment your counter variable by `1` with each pass through the loop. Then indent and do the extra space on even-numbered lines, like this:

```
# Store a number to use as a loop counter.
counter = 1
# Open the file.
with open('quotes.txt') as f:
    # Read one line from the file.
    one_line = f.readline()
    # As long as there are lines to read...
    while one_line:
        # If the counter is an even number, print a couple spaces.
        if counter % 2 == 0:
            print('  ' + one_line)
        # Otherwise print with no newline at the end.
        else:
```

```
            print(one_line, end='')
        # Increment the counter
        counter += 1
        # Read the next line.
        one_line = f.readline()
```

The output from this loop is the same as for the second `readlines()` loop, in which each author's name is indented and followed by an extra blank line caused by using `print()` without the `end=''`.

TIP

You can use generative AI to write code for opening and looping through files. For example, ask Copilot or ChatGPT to *write python code to open a text file named whatever.txt and loop through its contents. Include exception handler for file not found.* Then just modify the code it generates to suit your needs.

Appending versus overwriting files

WARNING

Anytime you work with files, it's important to understand the difference between *write* and *append.* If a file contains information and you open it in write mode and then write to it, your new content will overwrite (replace) whatever is already in the file. There is no undo for this. So if the content of the file is important, you want to make sure you don't make that mistake. To add content to the end of a file, open the file in append (`a`) mode, and then use `.write` to write to the file.

Suppose you want to add the name `Peña Calderón` to the `names.txt` file used earlier in this chapter. This name, as well as the names already in this file, use special characters beyond the English alphabet, so you need to set the encoding to UTF-8. Also, if you want to display each name in the file on a separate line, add a `\n` (newline) to the end of the name you're adding. Your code should look like this:

```
# New name to add with \n to mark end of line.
new_name = 'Peña Calderón\n'
# Open names.txt in append mode with encoding.
with open('names.txt', 'a', encoding='utf-8') as f:
    f.write(new_name)
```

To verify that it worked, start a new block of code, with no indents, so that `names.txt` file closes automatically. Then open the file in read (`r`) mode and print its contents. Figure 1-6 shows the code for adding the new name and the code to display the `names.txt` file after adding the name.

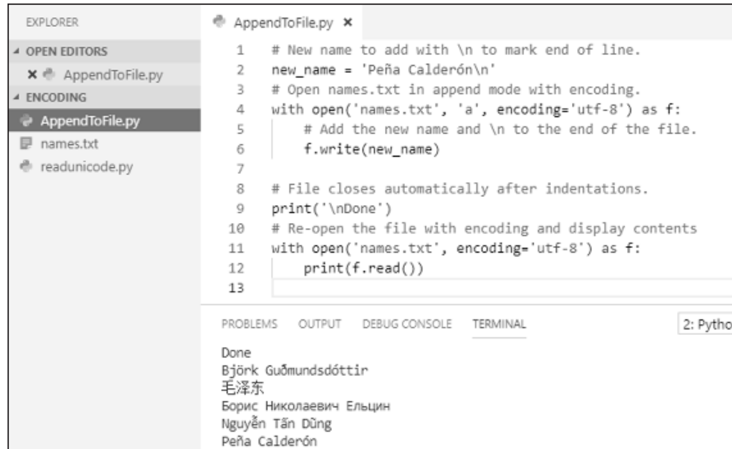

FIGURE 1-6: A new name appended to the end of the `names.txt` file.

TIP

Typing special characters such as *ñ* and *ó* usually involves holding down the Alt key and typing a three- or four-numeric digit; for example, Alt+164 for *ñ* or Alt+0243 for *ó*. Exactly how you do this depends on your operating system and editor. You can do a web search for something like *type tilde n on Windows* or *type accented o on Mac* for details.

Using tell() to determine the pointer location

When you loop through a file, its contents are read top to bottom and left to right. Python maintains a pointer to keep track of where it is in the file. When you're reading a text file with `readline()`, the pointer is always the character position of the next line in the file.

If all you've done so far is open the file, the character position will be `0`, the start of the file. Each time you execute `readline()`, the pointer advances to the start of the next row. Here is some code and its output to illustrate:

```
with open('names.txt', encoding='utf-8') as f:
    # Read first line to get started.
    print(f.tell())
    one_line = f.readline()
    # Keep reading one line at a time until there are no more.
    while one_line:
        print(one_line[:-1], f.tell())
        one_line = f.readline()
```

```
0
Björk Guðmundsdóttir 25
毛泽东 36
Борис Николаевич Ельцин 82
Nguyễn Tấn Dũng 104
Peña Calderón 121
```

The first `0` is the position of the pointer right after the file is opened. The `25` at the end of the next line is the position of the pointer after reading the first line. The `36` at the end of the next line is the pointer position at the end of the second line, and so forth, until the `121` at the end, when the pointer is at the end of the file.

If you try to do this with `readlines()`, you get a different result. Here is the code:

```
with open('names.txt', encoding='utf-8') as f:
    print(f.tell())
    # Reads in all lines first, then loops through.
    for one_line in f.readlines():
        print(one_line[:-1], f.tell())
```

Here is the output:

```
0
Björk Guðmundsdóttir 121
毛泽东 121
Борис Николаевич Ельцин 121
Nguyeễn Tấn Dũng 121
Peña Calderón 121
```

The pointer starts out at position `0`, as expected. But each line displays `121` at the end because `readlines()` reads in the entire file when executed, leaving the pointer at the end, position `121`. The loop is actually looping through the copy of the file in memory; it's no longer reading through the file.

Moving the pointer with seek()

Whereas the `tell()` method tells you where the pointer is in an external file, the `seek()` method enables you to reposition the pointer. The syntax is

```
file.seek(position[,whence])
```

Replace `file` with the variable name of the open file. Replace `position` to indicate where you want to put the pointer. For example, `0` moves the pointer back to the top of the file. The `whence` is optional; you can use it to indicate where in the file to set the pointer position. Your choices are

- `0`: Set the position relative to the start of the file.
- `1`: Set the position relative to the current pointer position.
- `2`: Set the position relative to the end of the file. Use a negative number for `position`.

If you omit the `whence` value, it defaults to `0`.

By far the most common use of `seek` is to just reset the pointer back to the top of the file for another pass through the file. The syntax for this is simply `.seek(0)`.

Reading and Copying a Binary File

Suppose you have an app that changes a binary file, and you want to always work with a copy of the original file to play it safe. Binary files can be huge, so rather than opening it all at once and risking running out of memory, you can read it in chunks and write it out in chunks. Binary files do not have human-readable content. Nor do they have lines of text. So `readline()` and `readlines()` aren't a good choice for looping through binary files, but you can use `.read()` with a specified size.

Figure 1-7 shows the `binarycopy.py` file, which makes a copy of any binary file. We take you through that code step-by-step so that you can understand how it works.

The first step is to specify the file you want to copy. We chose `happy_pickle.jpg`, which, as you can see in the figure, is in the same folder as the `binarycopy.py` file:

```
# Specify the file to copy.
file_to_copy = 'happy_pickle.jpg'
```

```
binarycopy.py > ...
1   # Specify the file to copy
2   file_to_copy = 'happy_pickle.jpg'
3   #Create new file nane wuth _copy before the extension.
4   name_parts = file_to_copy.split('.')
5   new_file = name_parts[0] + '_copy.' + name_parts[1]
6   # Open the priginal file as read-only binary.
7   with open(file_to_copy,'rb') as original_file:
8       # Create or open file to copy into.
9       with open(new_file,'wb') as copy_to:
10          # Grab a chunk of the original file (4MB).
11          chunk=original_file.read(4096)
12          # Loop though until no more chunks.
13          while len(chunk) > 0:
14              copy_to.write(chunk)
15              # Make sure you read the next chunk in this loop.
16              chunk = original_file.read(4096)
17
18  # Close is automatic after loops, show done message.
19  print('Done!')
```

FIGURE 1-7: The `binarycopy.py` file copies any binary file.

To make an empty file to copy into, you need a filename for the file. The following code takes care of that:

```
# Create new file name with _copy before the extension.
name_parts = file_to_copy.split('.')
new_file = name_parts[0] + '_copy.' + name_parts[1]
```

The first line after the copy splits the existing filename in two at the dot, so `name_parts[0]` contains `happy_pickle` and `name_parts[1]` contains `jpg`. Then the `new_file` variable gets a value consisting of the first part of the name with `_copy` and a dot attached, and then the last part of the name. So after this line executes, the `new_file` variable contains `happy_pickle_copy.jpg`.

To make the copy, open the original file in `rb` (read, binary file) mode. Then open the file into which you want to copy the original file in `wb` mode (write, binary). With *write,* Python creates a file of this name if the file doesn't already exist. If the file does exist, Python opens it with the pointer set at `0`, so anything that you write into the file will *replace* (not *add to*) the existing file.

In the code, you can see that we used `original_file` as the variable name from which to copy, and `copy_to` as the variable name of the file into which you copy data. Indentations, as always, are critical:

```
# Open the original file as read-only binary.
with open(file_to_copy, 'rb') as original_file:
    # Create or open file to copy into.
    with open(new_file, 'wb') as copy_to:
```

If you use .read() to read in the entire binary file, you run the risk of its being so large that it overwhelms the computer's RAM and crashes the program. To avoid this problem, we've written this program to read in a modest 4KB (4,096 bytes) of data at a time. This 4KB chunk is stored in a variable named chunk:

```
# Grab a chunk of original file (4KB).
chunk = original_file.read(4096)
```

The next line sets up a loop that keeps reading one chunk at a time. The pointer is automatically positioned at the next chunk with each pass through the loop. Eventually, it will hit the end of the file where it can't read anymore. When this happens, chunk will be empty, meaning it has a length of 0. So this loop keeps going through the file until it gets to the end:

```
# Loop through until no more chunks.
while len(chunk) > 0:
```

Within the loop, the first line copies the last-read chunk into the copy_to file. The second line reads the next 4KB chunk from the original file. And so it goes until everything from original_file has been copied to the new file:

```
    copy_to.write(chunk)
    # Make sure you read in the next chunk in this loop.
    chunk = original_file.read(4096)
```

All the indentations stop after this line. When the loop is done, the files close automatically, and the last line displays Done! as follows:

```
print('Done!')
```

Figure 1-8 shows the results of running the code. The Terminal pane simply shows Done!. But as you can see, there's now a file named happy_pickle_copy.jpg in the folder. Opening this file will prove that it is a copy of the original file.

```
# Specify the file to copy.
file_to_copy = 'happy_pickle.jpg'

# Create new file name with _copy before the extension.
name_parts = file_to_copy.split('.')
new_file = name_parts[0] + '_copy.' + name_parts[1]

# Open the original file as read-only binary.
with open(file_to_copy, 'rb') as original_file:

    # Create or open file to copy into.
    with open(new_file, 'wb') as copy_to:

        # Grab a chunk of original file (4MB).
        chunk = original_file.read(4096)

        # Loop through until no more chunks.
        while len(chunk) > 0:

            copy_to.write(chunk)
            # Make sure you read in the next chunk in this loop.
            chunk = original_file.read(4096)

# Close is automatic after loops, show done message.
print('Done!')
```

```
(base) C:\Users\acsimpson\Desktop\sample files>C:/Users/acsimpson/AppData/Local
Done!
```

FIGURE 1-8: Running `binarycopy.py` added `happy_pickle_copy.jpg` to the folder.

Conquering CSV Files

CSV (comma-separated values) is a widely used format for storing and transporting tabular data. *Tabular* means the data can generally be displayed in a table format consisting of rows and columns. In a spreadsheet app such as Microsoft Excel, Apple Numbers, or Google Sheets, the tabular format is obvious, as shown in Figure 1-9.

	A	B	C	D	E
1	Full Name	Birth Year	Date Joined	Is Active	Balance
2	Angst, Annie	1982	1/11/2011	TRUE	$300.00
3	Bónañas, Barry	1973	2/11/2012	FALSE	-$123.45
4	Schadenfreude, Sandy	2004	3/3/2003	TRUE	$0.00
5	Weltschmerz, Wanda	1995	4/24/1994	FALSE	$999,999.99
6	Malaise, Mindy	2006	5/5/2005	TRUE	$454.01
7	O'Possum, Ollie	1987	7/27/1997	FALSE	-$1,000.00
8					
9	Pusillanimity, Pamela	1979	8/8/2008	TRUE	$12,345.67

FIGURE 1-9: A CSV file in Microsoft Excel.

Without the aid of a special program to make the data in the file display in a neat tabular format, each row is just a line in the file. And each unique value is separated by a comma. For instance, opening the file shown in Figure 1-9 in a simple text editor such as Notepad or TextEdit shows what's really stored in the file, as you can see in Figure 1-10.

```
sample.csv
1   Full Name,Birth Year,Date Joined,Is Active,Balance
2   "Angst, Annie",1982,1/11/2011,TRUE,$300.00
3   "Bónañas, Barry",1973,2/11/2012,FALSE,-$123.45
4   "Schadenfreude, Sandy",2004,3/3/2003,TRUE,$0.00
5   "Weltschmerz, Wanda",1995,4/24/1994,FALSE,"$999,999.99"
6   "Malaise, Mindy",2006,5/5/2005,TRUE,$454.01
7   "O'Possum, Ollie",1987,7/27/1997,FALSE,"-$1,000.00"
8   ,,,,
9   "Pusillanimity, Pamela",1979,8/8/2008,TRUE,"$12,345.67"
10
```

FIGURE 1-10: A CSV file in a text editor.

In the text editor, the first row, often called the *header*, contains the column headings, or *field names*, that appear across the first row of the spreadsheet. If you look at the names in the second example, the raw CSV file, you'll see that they're enclosed in quotation marks, like this:

```
"Angst, Annie"
```

The quotation marks indicate that the stuff *between* them is all one thing. In other words, the comma between the last and first name is part of the name; it isn't the start of a new column. So the first two columns in this row one are

```
"Angst, Annie", 1982
```

and not

```
Angst, Annie
```

The same is true in all other rows: The name enclosed in quotation marks (including commas) is just one name, not two separate columns of data.

If a string contains an apostrophe, which is the same character as a single quotation mark, you have to use double quotation marks around the string. Otherwise, if you do this:

```
'O'Henry, Harry'
```

the first part of the string is `'O'` and then Python doesn't know what to do with the text after the second single quotation mark. Using double quotation marks alleviates any confusion because there are no other double quotation marks within the name:

```
"O'Henry, Harry"
```

Figure 1-10 also illustrates other considerations when creating CSV files. For example, the `Bónañas, Barry` name contains some non-ASCII characters. The

second-to-last row contains only a bunch of commas. (In a CSV file, if a cell is missing its data, you put the comma that ends the cell with nothing to its left.) The Balance column has dollar signs and commas in the numbers, which don't work with the Python `float` data type. We talk about how to deal with all these issues in the sections to follow.

Although you could work with CSV files using just what you've learned so far, the task will be a lot quicker and easier if you use the `csv` module, which you already have. To use it, just put this near the top of your program:

```
import csv
```

Remember, this line of code doesn't bring in a CSV *file.* It brings in the prewritten code that makes it easier for you to work with CSV files in your own Python code.

Opening a CSV file

Opening a CSV file is no different from opening any other file. Just remember that if the file contains special characters, you need to include `encoding='utf-8'` to avoid an error message. Optionally, when importing data, you probably don't want to read in the newline character at the end of each row, so you can add `newline=''` to the `open()` statement. Here is how you might comment and code this, except you'd replace `sample.csv` with the path to the CSV file you want to open:

```
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
```

To loop through a CSV file, you can use the built-in `reader()` function, which returns an object that can iterate through rows.\ Again, the syntax is simple:

```
reader = csv.reader(f)
```

Replace `f` with the name you used at the end of your `open` statement (without the colon at the very end).

Optionally, you can also count rows as you go. Just put everything to the right of = in `enumerate()`, as shown in the following (where we've also added a comment above the code):

```
# Create a CVS row counter and row reader.
reader = enumerate(csv.reader(f))
```

Next, you can set up your loop to read one row at a time. Because you put an enumerator on the loop, you can use two variable names in your `for` loop. The first variable (which we call `i`) keeps track of the counter (which starts at `0` and increases by `1` with each pass through the loop). The second variable, `row`, contains the entire row of data from the CSV file:

```
# Loop through one row at a time, i is counter, row is entire row.
for i, row in reader:
```

You can use a `print()` function to print the value of `i` and `row` with each pass through the loop, like this:

```
import csv
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Create a CVS row counter and row reader.
    reader = enumerate(csv.reader(f))
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
        print(i, row)
print('Done')
```

The output from this code, using the `sample.csv` file described earlier as input, is as follows:

```
0 ['Full Name', 'Birth Year', 'Date Joined', 'Is Active', 'Balance']
1 ['Angst, Annie', '1982', '1/11/2011', 'TRUE', '$300.00']
2 ['Bónañas, Barry', '1973', '2/11/2012', 'FALSE', '-$123.45']
3 ['Schadenfreude, Sandy', '2004', '3/3/2003', 'TRUE', '$0.00']
4 ['Weltschmerz, Wanda', '1995', '4/24/1994', 'FALSE', '$999,999.99']
5 ['Malaise, Mindy', '2006', '5/5/2005', 'TRUE', '$454.01']
6 ["O'Possum, Ollie", '1987', '7/27/1997', 'FALSE', '-$1,000.00']
7 ['', '', '', '', '']
8 ['Pusillanimity, Pamela', '1979', '8/8/2008', 'TRUE', '$12,345.67']
```


Note how the row of column names is row `0`. If you happen to see some weird characters like `\ufeff` before `Full Name` in that row, that's called a byte order mark (BOM), which is just something Excel might stick in there. Typically you don't care what's in that first row because the real data doesn't start until the second row. So don't give the BOM a second thought; it's of no value to you and it isn't doing any harm.

As you can see, each row is a list of five items separated by commas. In your code, you can refer to each column by its position. For example, `row[0]` is the first column in the row (the person's name). Then, `row[1]` is the birth year, `row[2]` is date joined, `row[3]` is whether the person is active, and `row[4]` is the balance.

All the data in the CSV file consists of strings — even if they don't look like strings. But anything and everything coming from a CSV file is a string because a CSV file is a type of text file, and a text file contains only strings (text) — no integers, dates, Booleans, or floats.

In your app, you'll probably want to convert the incoming data to Python data types so that you can work with them more effectively or even transfer them to a database. In the next sections, we look at how to do the conversion for each data type.

Converting strings

Technically, you don't have to convert anything from the CSV file to a string. But you may want to chop the file up a bit or deal with empty strings in some way. First, as mentioned, we care only about the data here, not that first row. So inside the loop, you can start with an `if` that doesn't do anything if the current row is row `0`. Replace the `print(i, row)` like this:

```
# Row 0 is just column headings, ignore it.
if i > 0:
    full_name = row[0].split(',')
    last_name = full_name[0].strip()
    first_name = full_name[1].strip()
```

When the `if` condition is `True`, the indented lines below it execute. The first line creates a list named `full_name` that contains the person's last name in `full_name[0]`, and that person's first name in `full_name[1]`. Then the code puts the person's last name in a variable named `last_name`, and the person's first name in a variable named `first_name`. The `.strip()` method ensures that any leading or trailing whitespace is removed before the value is stored in the variable. But if you run the code that way, it will bomb, because row 7 doesn't have a name, and Python can't split an empty string at a comma (because the empty string contains no comma).

To get around this problem, you can tell Python to *try* to split the name at the comma, if it can. But if it bombs when trying, just store an empty string in the `full_name`, `last_name`, and `first_name` variables. Here's that code with some extra comments thrown in to explain what's going on. Instead of printing `i` and the entire row, the code prints the first name and last name (and nothing for the row whose information is missing). The output appears below the code:

```
import csv
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
```

```
    # Create a CVS row counter and row reader.
    reader = enumerate(csv.reader(f))
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
        # Row 0 is just column headings, ignore it.
        if i > 0:
            # Whole name split into two at comma.
            try:
                full_name = row[0].split(',')
                # Last name, strip extra spaces.
                last_name = full_name[0].strip()
                # First name, strip extra spaces.
                first_name = full_name[1].strip()
            except IndexError:
                full_name = last_name = first_name = ""
            print(first_name, last_name)
print('Done!')
```

```
Annie Angst
Barry Bónañas
Sandy Schadenfreude
Wanda Weltschmerz
Mindy Malaise
Ollie O'Possum

Pamela Pusillanimity
Done!
```

Converting to integers

The second column in each row, `row[1]`, is the birth year. As long as the string contains something that can be converted to a number, you can use the simple built-in `int()` function to convert it to an integer. We do have a problem in row `7`, which is empty. Python won't automatically convert this to a `0`; you have to help it along a bit, as follows:

```
# Birth year integer, zero for empty string.
birth_year = int(row[1] or 0)
```

The code looks surprisingly simple, but that is the beauty of Python: It *is* surprisingly simple. This line of code says "Create a variable named `birth_year` and put in it the second column value, if you can, or if there is nothing to convert to an integer, then just put in a zero."

Converting to date

The third column in our CSV file, `row[2]`, is the date joined, and it appears to have a reasonable date in each row (except the row whose data is missing). To convert the textual date to a Python date, you need to import the `datetime` module by adding `import datetime as dt` near the top of the program. Then the simple conversion is

```
date_joined = dt.datetime.strptime(row[2], "%m/%d/%Y").date()
```

A lot is going on here. First, you create a variable named `date_joined`. The `strptime` code means "string parse for date time." The `[row[2]` code means the third column (because the first column is always column `0`). The `"%m/%d/%Y"` tells `strptime` that the string date contains the month, a slash, the day of the month, a slash, and then the four-digit year (`%Y`). The `.date()` at the end means "just take the date from the date and time."

One small problem. When the program gets to the row whose date is missing, it will bomb. So once again we use a `try` block to convert the date; if it can't come up with a date, it puts in the value `None`, which is Python's word for an empty object.

REMEMBER

In Python, `datetime` is a class, so any date and time you create is an object (of the `datetime` type). You use `''` for an empty string, but `None` for an empty object.

Here is the code as it stands now with the `import` at top for `datetime`, and `try ... except` to convert the string date to a Python date:

```
import csv
import datetime as dt
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Create a CVS row counter and row reader.
    reader = enumerate(csv.reader(f))
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
        # Row 0 is just column headings, ignore it.
        if i > 0:
            # Whole name split into two at comma.
            try:
                full_name = row[0].split(',')
                # Last name, strip extra spaces.
                last_name = full_name[0].strip()
                # First name, strip extra spaces.
                first_name = full_name[1].strip()
            except IndexError:
                full_name = last_name = first_name = ""
```

```
            # Birth year integer, zero for empty string.
            birth_year = int(row[1] or 0)
            # Date_joined is a date.
            try:
                date_joined = dt.datetime.strptime(row[2], "%m/%d/%Y").date()
            except ValueError:
                date_joined = None
            print(first_name, last_name, birth_year, date_joined)
print('Done!')
```

Here is the output from this code, which now prints `first_name`, `last_name`, `birth_year`, and `date_joined` with each pass through the data rows in the table:

```
Annie Angst 1982 2011-01-11
Barry Bónañas 1973 2012-02-11
Sandy Schadenfreude 2004 2003-03-03
Wanda Weltschmerz 1995 1994-04-24
Mindy Malaise 2006 2005-05-05
Ollie O'Possum 1987 1997-07-27
  0 None
Pamela Pusillanimity 1979 2008-08-08
Done!
```

Converting to Boolean

The fourth column, `row[3]` in each row, contains `TRUE` or `FALSE`, or possibly nothing at all. Excel uses all uppercase letters for `True` and `False`, which are automatically carried over to the CSV file when saving as CSV in Excel. To convert to a Boolean, set `is_active` to `True` if the fourth column contains `TRUE`. Otherwise, set `is_active` to `False`. Here is a comment and some code you can add to accomplish that:

```
    # is_active is a Boolean. Convert TRUE to True, anything else to False.
    is_active = True if row[3] == 'TRUE' else False
```

So after the line of code executes, the Python `is_active` variable contains a Boolean `True` if `row` contained `TRUE`. Otherwise, `is_active` is set to `False` if the row contained `FALSE` or nothing.

Converting to floats

The fifth column in each row contains the balance, which is a dollar amount. In Python, you want the dollar amount to be an actual numeric value, so you can do math. The float data type is good because you can include decimal points for the pennies. But there's one potential snag. Python floats can't contain a currency

symbol ($) or a comma (,), so you must remove those from the string. As a general rule of thumb, it's also a good idea to remove any leading and trailing spaces in numbers. These you can remove easily with the strip() method. The following line of code creates a variable named str_balance (which is still a string) but with the dollar sign, comma, and any trailing leading spaces removed:

```
# Remove $, commas, leading and trailing spaces.
str_balance = (row[4].replace('$', '').replace(',', '')).strip()
```

You can read this second line as "The new string named str_balance consists of whatever is in the fifth column after removing currency symbols, commas, and any unnecessary blank spaces."

That line of code creates a string that's in a format that can be reliably converted to a number. You can use the Python float() function to convert that string to a valid floating-point number. Here is the exact code. The "or 0" part just means set the value to zero if the string is empty or contains content that can't be converted to a number.

```
            balance = float(str_balance or 0)
```

USING REGULAR EXPRESSIONS IN PYTHON

Although we assume that you're not familiar with other programming languages, some readers will be and may be wondering why we didn't use a regular expression instead of the replace() method to remove the dollar sign and comma from balance. Well, regular expressions aren't built into Python. So if you want to use them, you need to put from re import sub near the top of your code. The re module provides support for regular expressions. The sub() function from that module lets you do that actual substitution.

```
from re import sub
```

Later in the code, you can replace

```
str_balance = (row[4].replace('$', '').replace(',', '')).strip()
```

with

```
str_balance = (sub(r'[\s\$,]', '', row[4])).strip()
```

This line does the same thing as the original line. It removes the dollar sign, commas, and any leading and trailing spaces from the fifth column value.

The code in Figure 1-11 shows everything in place, including a `print()` line that displays the values of all five columns after the conversion.

```
1   import csv
2   import datetime as dt
3
4   # Open CSV file with UTF-8 encoding, don't read in newline characters.
5   with open("sample.csv", encoding="utf-8", newline="") as f:
6       # Create a CVS row counter and row reader.
7       reader = enumerate(csv.reader(f))
8       # Loop through one row at a time, i is counter, row is entire row.
9       for i, row in reader:
10          # Row 0 is just column headings, ignore it.
11          if i > 0:
12              # Whole name split into two at comma.
13              try:
14                  full_name = row[0].split(",")
15                  # Last name, strip extra spaces.
16                  last_name = full_name[0].strip()
17                  # First name, strip extra spaces.
18                  first_name = full_name[1].strip()
19              except IndexError:
20                  full_name = last_name = first_name = ""
21              # Birth year integer, zero for empty string.
22              birth_year = int(row[1] or 0)
23              # Date_joined is a date.
24              try:
25                  date_joined = dt.datetime.strptime(row[2], "%m/%d/%Y").date()
26              except ValueError:
27                  date_joined = None
28              # is_active is a Boolean. Convert TRUE to True, anything else to False.
29              is_active = True if row[3] == "TRUE" else False
30              # Balance is a float, zero for empty string.
31              str_balance = row[4].replace("$", "").replace(",", "")
32              balance = float(str_balance or 0)
33              print(first_name, last_name, birth_year, date_joined, is_active, balance)
34  print("Done!")
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

Sandy Schandenfreud 2004 2003-03-03 True 0.0
Wanda Weltschmerz 1995 1994-04-24 False 999999.99
Mindy Malaise 2006 2005-05-05 True 454.01
Ollie O'Possum 1987 1997-07-27 False -1000.0
  0 None False 0.0
Pamela Pusillanimity 1979 2008-08-08 True 12345.67
Done!
```

FIGURE 1-11: Reading a CSV file and converting it to Python data types.

Converting from CSV to Objects and Dictionaries

You've seen how to read in data from a CSV file, and how to convert that data from the default string data type to an appropriate Python data type. Chances are, in addition to all this, you may want to organize the data into a group of objects generated from the same class or perhaps into a set of dictionaries inside a larger dictionary.

All the code you've learned so far will be useful because it's necessary to get the job done. To reduce the code clutter in these examples, we've taken the various

bits of code for converting the data and put them into their own functions. This allows you to convert a data item using just the function name with the value to convert in parentheses, such as `balance(row[4])`.

Importing CSV to Python objects

If you want the data from your CSV file to be organized into a list of objects, write your code as shown here:

```
import datetime as dt
import csv
# Use these functions to convert any string to appropriate Python
# data type.

# Get just the first name from full name.
def fname(any):
    try:
        nm = any.split(',')
        return nm[1]
    except IndexError:
        return ''

# Get just the last name from full name.
def lname(any):
    try:
        nm = any.split(',')
        return nm[0]
    except IndexError:
        return ''

# Convert string to integer or zero if no value.
def integer(any):
    return int(any or 0)

# Convert mm/dd/yyyy date to date or None if no valid date.
def date(any):
    try:
        return dt.datetime.strptime(any,"%m/%d/%Y").date()
    except ValueError:
        return None

# Convert any TRUE to True, anything else to False.
def boolean(any):
    return True if any == "TRUE" else False

# Convert string to float, or to zero if no value.
def floatnum(any):
    s_float = any.replace("$", "").replace(",", "")
```

```
    return float(s_float or 0)

# Create an empty list of people.
people = []
# Define a class where each person is an object.
class Person:
    def __init__(self, id, first_name, last_name, birth_year, date_joined,
                 is_active, balance):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.birth_year = birth_year
        self.date_joined = date_joined
        self.is_active = is_active
        self.balance = balance

# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Set up a csv reader with a counter.
    reader = enumerate(csv.reader(f))
    # Skip the first row, which is column names.
    f.readline()
    # Loop through remaining rows one at a time, i is counter,
    # row is entire row.
    for i, row in reader:
        # From each data row in the CSV file, create a Person object with
        # unique id and appropriate data types, add to people list.
        people.append(Person(i, fname(row[0]), lname(row[0]),
                      integer(row[1]), date(row[2]), boolean(row[3]),
                      floatnum(row[4])))

# When above loop is done, show all objects in the people list.
for p in people:
    print(p.id, p.first_name, p.last_name, p.birth_year, p.date_joined,
          p.is_active, p.balance)
```


Here's how the code works: The first few lines are the required `imports`, followed by a number of functions to convert the incoming string data to Python data types. This code is similar to previous examples in this chapter. We just separated the conversion code into separate functions to compartmentalize everything a bit:

```
import datetime as dt
import csv
# Use these functions to convert any string to appropriate Python data type.
# Get just the first name from full name.
def fname(any):
```

```
    try:
        nm = any.split(',')
        return nm[1]
    except IndexError:
        return ''

# Get just the last name from full name.
def lname(any):
    try:
        nm = any.split(',')
        return nm[0]
    except IndexError:
        return ''

# Convert string to integer or zero if no value.
def integer(any):
    return int(any or 0)

# Convert mm/dd/yyyy date to date or None if no valid date.
def date(any):
    try:
        return dt.datetime.strptime(any,"%m/%d/%Y").date()
    except ValueError:
        return None

# Convert any TRUE to True, anything else to False.
def boolean(any):
    return True if any == "TRUE" else False


# Convert string to float, or to zero if no value.
def floatnum(any):
    s_float = any.replace("$", "").replace(",", "").strip()
    return float(s_float or 0)
```

The next line creates an empty list named `people` to provide a place to store the objects that the program will create from the CSV file:

```
# Create an empty list of people.
people = []
```

Next, the code defines a class that will be used to generate each `Person` object from the CSV file:

```
# Define a class where each person is an object.
class Person:
    def __init__(self, id, first_name, last_name, birth_year, date_joined,
```

```
                 is_active, balance):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.birth_year = birth_year
        self.date_joined = date_joined
        self.is_active = is_active
        self.balance = balance
```

The reading of the CSV file starts in the next lines. The code opens the `sample.csv` file with UTF-8 encoding. The `newline=''` just prevents the code from sticking the newline character at the end of each row to the last item of data in each row. The `reader` uses an enumerator to keep a count while reading the rows. The `f.readline()` reads the first row, which is just column heads, so that the `for` that follows starts on the second row. The `i` variable in the `for` loop is the incrementing counter, and the `row` is the entire row of data from the CSV file:

```
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Set up a csv reader with a counter.
    reader = enumerate(csv.reader(f))
    # Skip the first row, which is column names.
    f.readline()
    # Loop through remaining rows one at a time,
    # i is counter, row is entire row.
    for i, row in reader:
```

With each pass through the loop, the code creates a single `Person` object from the incrementing counter (`i`) and appends the data in the `row`. Note how we've called on the functions defined earlier in the code to do the data type conversions. This makes this code more compact and a little easier to read and work with:

```
# From each data row in the CSV file, create a Person object with unique id
# and appropriate data types, add to people list.
people.append(Person(i, fname(row[0]), lname(row[0]), integer(row[1]),
              date(row[2]), boolean(row[3]), floatnum(row[4])))
```

When the loop is complete, the next code simply displays each object on the screen to verify that the code worked correctly:

```
# When above loop is done, show all objects in the people list.
for p in people:
    print(p.id, p.first_name, p.last_name, p.birth_year, p.date_joined,
        p.is_active, p.balance)
```

Figure 1-12 shows the output from running this program. Of course, subsequent code in the program can do anything you need to do with each object; the printing is there to test and verify that the program worked.

```
34  # Create an empty list of people.
35  people = []
36  # Define a class where each person is an object.
37  class Person:
38      def __init__(self, id, first_name, last_name, birth_year, date_joined, is_active, balance):
39          self.id = id
40          self.first_name = first_name
41          self.last_name = last_name
42          self.birth_year = birth_year
43          self.date_joined = date_joined
44          self.is_active = is_active
45          self.balance = balance
46
47  # Open CSV file with UTF-8 encoding, don't read in newline characters.
48  with open('sample.csv', encoding='utf-8', newline='') as f:
49      # Set up a csv reader with a counter.
50      reader = enumerate(csv.reader(f))
51      # Skip the first row, which is column names.
52      f.readline()
53      # Loop through remaining rows one at a time, i is counter, row is entire row.
54      for i, row in reader:
55          # From each data row in the CSV file, create a Person object with unique id and appropriate data types, add to people list.
56          people.append(Person(i, fname(row[0]), lname(row[0]), integer(row[1]), date(row[2]), boolean(row[3]), floatnum(row[4])))
57
58  # When above loop is done, show all objects in the people list.
59  for p in people:
60      print(p.id, p.first_name, p.last_name, p.birth_year, p.date_joined, p.is_active, p.balance)
61

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

0  Annie Angst 1982 2011-01-11 True 300.0
1  Barry Bónañas 1973 2012-02-11 True -123.45
2  Sandy Schadenfreude 2004 2003-03-03 True 0.0
3  Wanda Weltschmerz 1995 1994-04-24 True 999999.99
4  Mindy Malaise 2006 2005-05-05 True 454.01
5  Ollie O'Possum 1987 1997-07-27 True -1000.0
6   0 None False 0.0
7  Pamela Pusillanimity 1979 2008-08-08 True 12345.67
```

FIGURE 1-12: Reading a CSV file into a list of objects.

Importing CSV to Python dictionaries

If you prefer to store each row of data from the CSV file in its own dictionary, you can use code that's similar to the preceding code for creating objects. You don't need the class definition code, because you won't be creating objects here. Instead of creating a `people` list, you can create an empty `people` dictionary to hold all the individual "person" dictionaries, like this:

```
# Create an empty dictionary of people.
people = {}
```

As far as the loop goes, again you can use an enumerator (`i`) to count rows, and you can also use this unique value as the key for each new dictionary you create. The line that starts with `newdict=` creates a dictionary with the data from one CSV file row, using the built-in Python `dict()` function. The next line assigns the value of `i` plus `1` to each newly created dictionary (to start the counting at `1` rather than `0`):

```
# Loop through remaining rows one at a time, i is counter,
# row is entire row.
for i, row in reader:
    # From each data row in the CSV file, create a dictionary item with
    # unique id and appropriate data types, add to people list.
    newdict = dict({'first_name': fname(row[0]),
        'last_name': lname(row[0]), 'birth_year': integer(row[1]),
        'date_joined' : date(row[2]), 'is_active' : boolean(row[3]),
        'balance' : floatnum(row[4])})
    people[i + 1] = newdict
```

To verify that the code ran correctly, you can loop through the dictionaries in the `people` dictionary and show the *`key`*`:`*`value`* pair for each item of data in each row. Figure 1-13 shows the result of running that code in VS Code:

```
30  # Convert string to float, or to zero if no value.
31  def floatnum(any):
32      s_balance = (any.replace('$', '').replace(',', '')).strip()
33      return float(s_balance or 0)
34  # Create an empty dictionary of people.
35  people = {}
36  # Open CSV file with UTF-8 encoding, don't read in newline characters.
37  with open('sample.csv', encoding='utf-8', newline='') as f:
38      # Set up a csv reader with a counter.
39      reader = enumerate(csv.reader(f))
40      # Skip the first row, which is column names.
41      f.readline()
42      # Loop through remaining rows one at a time, i is counter,
43      # row is entire row.
44      for i, row in reader:
45          # From each data row in the CSV file, create a Person object with
46          # unique id and appropriate data types, add to people dictionary.
47          newdict = dict({'first_name': fname(row[0]),
48              'last_name': lname(row[0]), 'birth_year': integer(row[1]),
49              'date_joined' : date(row[2]), 'is_active' : boolean(row[3]),
50              'balance' : floatnum(row[4])})
51          people[i + 1] = newdict
52
53  # When above loop is done, show all objects in the people list.
54  for person in people.keys():
55      id = person
56
57      print(id, people[person]['first_name'],
58                people[person]['last_name'],
59                people[person]['birth_year'],
60                people[person]['date_joined'],
61                people[person]['is_active'],
62                people[person]['balance'])
63

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Alan\Desktop\py311> & C:/Users/Alan/AppData/Local/Microsoft/WindowsApps/pyt
1  Annie Angst 1982 2011-01-11 True 300.0
2  Barry Bonanas 1973 2012-02-11 True -123.45
3  Sandy Schandenfreud 2004 2003-03-03 True 0.0
4  Wanda Weltschmerz 1995 1994-04-24 True 999999.99
5  Mindy Malaise 2006 2005-05-05 True 454.01
6  Ollie O'Possum 1987 1997-07-27 True -1000.0
7   0 None False 0.0
8  Pamela Pusillanimity 1979 2008-08-08 True 12345.67
```

FIGURE 1-13: Reading a CSV file into a dictionary of dictionaries.

Here is all the code that reads the data from the CSV files into the dictionaries:

```
import datetime as dt
import csv
# Use these functions to convert any string to appropriate
# Python data type.
# Get just the first name from full name.
def fname(any):
    try:
        nm = any.split(',')
        return nm[1]
    except IndexError:
        return ''
# Get just the last name from full name.
def lname(any):
    try:
        nm = any.split(',')
        return nm[0]
    except IndexError:
        return ''
# Convert string to integer or zero if no value.
def integer(any):
    return int(any or 0)
# Convert mm/dd/yyyy date to date or None if no valid date.
def date(any):
    try:
        return dt.datetime.strptime(any, "%m/%d/%Y").date()
    except ValueError:
        return None
# Convert any TRUE to True, anything else to False.
def boolean(any):
    return True if any == "TRUE" else False
# Convert string to float, or to zero if no value.
def floatnum(any):
    s_float = (any.replace('$', '').replace(',', '')).strip()
    return float(s_float or 0)
# Create an empty dictionary of people.
people = {}
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Set up a csv reader with a counter.
    reader = enumerate(csv.reader(f))
    # Skip the first row, which is column names.
    f.readline()
    # Loop through remaining rows one at a time, i is counter,
    # row is entire row.
    for i, row in reader:
        # From each data row in the CSV file, create a person with
```

```
        # unique id and appropriate data types, add to people dictionary.
        newdict = dict({'first_name': fname(row[0]),
            'last_name': lname(row[0]), 'birth_year': integer(row[1]),
            'date_joined' : date(row[2]), 'is_active' : boolean(row[3]),
            'balance' : floatnum(row[4])})
        people[i + 1] = newdict

# When above loop is done, show all objects in the people list.
for person in people.keys():
    id = person
    print(id, people[person]['first_name'],
              people[person]['last_name'],
              people[person]['birth_year'],
              people[person]['date_joined'],
              people[person]['is_active'],
              people[person]['balance'])
```

CSV files are widely used because it's easy to export data from spreadsheets and database tables to this format. Getting data from those files can be tricky at times, but you'll find Python's `csv` module a big help. The `csv` module takes care of many of the details, makes it relatively easy to loop through one row at a time, and handles the data however you see fit in your Python app.

Another format for transporting and storing data in a simple textual format is JSON, or JavaScript Object Notation. You learn all about JSON in the next chapter.