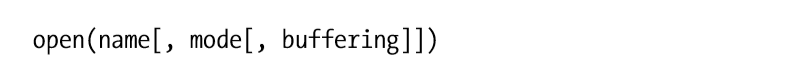
**Python File Handling**

**Overview**

|  |
| --- |
| So far we’ve mainly been working with data structures that reside in the interpreter itself. What little interaction our programs have had with the outside world has been through input, and print. In this chapter, we go one step further and let our programs catch a  glimpse of a larger world: the world of files and streams. The functions and objects described in this section will enable you to store data between program invocations and to process data from other programs. |

**Opening Files**

You can open files with the ***open*** function, which has the following syntax:



The open function takes a file name as its only required argument, and returns a file object. The mode and buffering arguments are both optional and will be explained below.

**The Mode Argument**

If you use *open* with only a file name as a parameter, you get a *file object* you can read from. If you want to *write* to the file, you have to state that explicitly, supplying a mode. (I get to the actual reading and writing in a little while.) The mode argument to the open function can have several values, as summarized in table below:

|  |  |
| --- | --- |
| **Values** | **Description** |
| ‘r’ | Read mode |
| ‘w’ | Write mode |
| ‘b’ | Binary mode (added to other mode) |
| ‘+’ | Read/write mode (added to other mode) |

Explicitly specifying read mode has the same effect as not supplying a mode string at all. The write mode enables you to write to the file.

The '+' can be added to any of the other modes to indicate that both reading and writing is allowed. So, for example, 'r+' can be used when opening a text file for reading and writing. The 'b' mode changes the way the file is handled. Generally, Python assumes that you are dealing with text files (containing characters).

Typically, this is not a problem. But if you are processing some other kind of file (called a binary file) such as a sound clip or an image, you should add a 'b' to your mode: for example, 'rb' to read a binary file.

**Why Use Binary Mode?**

If you use binary mode when you read (or write) a file, things won’t be much different. You are still able to read a number of bytes (basically the same as characters), and perform other operations associated with text files.

The main point is that when you use binary mode, Python gives you exactly the contents found in the file— and in text mode it won’t necessarily do that. The problem occurs when you work with a binary file, such as a sound clip. It may contain bytes that can be interpreted as the line-ending characters, Python will perform its automatic conversion. However, that will probably destroy your binary data. So, to avoid that, you simply use binary mode, and no conversions are made.

**Buffering**

The open function takes a third (optional) parameter, which controls the buffering of the file. If the parameter is 0 (or False), I/O (input/output) is unbuffered (all *reads* and *writes* go directly from/to the disk); if it is 1 (or *True*), I/O is buffered (meaning that Python may use memory instead of disk space to make things go faster, and only update when you use flush or close— see the section “Closing Your Files,” later in this section). Larger numbers indicate the buffer size (in bytes), while 1 (or any negative number) sets the buffer size to the default.

**The Basic File Methods**

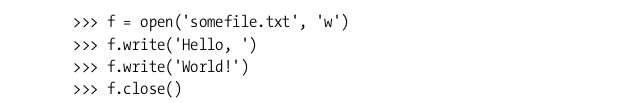
Now you know how to open files; the next step is to do something useful with them. In this section, you learn some basic methods that file objects (and some other ‘file-like’ objects, sometimes called streams) have.

**Reading and Writing**

The most important capabilities of files (or streams) are supplying and receiving data. If you

have a file-like object named f, you can write data (in the form of a string) with the method f.*write*, and *read* data (also as a string) with the method f.*read*.

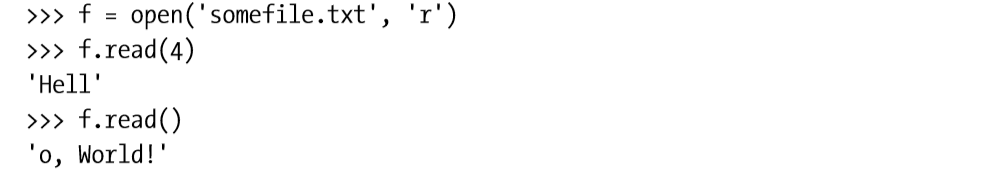
Each time you call f.*write*(string), the string you supply is written to the file after those you have written previously:



Notice that I call the close method when I’m finished with the file. You learn more about it in the section “Closing Your Files” later in this section.

Reading is just as simple. Just remember to tell the stream how many characters (bytes) you want to read.

**Example** (continuing where I left off):



First, I specify how many characters to read (4), and then I simply read the rest of the file (by not supplying a number). Note that I could have dropped the mode specification from the call to open because ‘r’ is the default.

**Reading and Writing Lines**

Actually, what I’ve been doing until now is a bit impractical. Usually, I could just as well be reading in the lines of a stream as reading letter by letter. You can read a single line (text from where you have come so far, up to and including the first line separator you encounter) with the method file.readline.

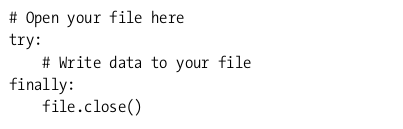
You can either use it without any arguments (in which case a line is simply read and returned) or with a nonnegative integer, which is then the maximum number of characters (or bytes) that readline is allowed to read. So if *someFile.readline()* returns 'Hello, World!\n', *someFile.readline(5)* returns 'Hello'. To read all the lines of a file and have them returned as a list, use the *readlines* method.

|  |
| --- |
| **Note** - An alternative to readlines that can be useful when iterating is ***xreadlines***. For more information, see the section “I**terating Over File Contents**,” later in this chapter. |

The method writelines is the opposite of readlines: Give it a list (or, in fact, any sequence or iterable object) of strings, and it writes all the strings to the file (or stream). Note that newlines are not added: you have to add those yourself. Also, there is no writeline method because you can just use write.

**Closing Your Files**

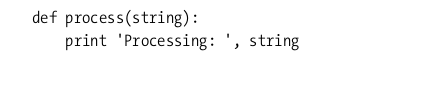
You should remember to close your files by calling their close method. Usually, a file object is closed automatically when you quit your program (and possibly before that), and not closing files you have been reading from isn’t really that important (although it can’t hurt, and might help to avoid keeping the file uselessly “locked” against modification in some operating systems and settings). But you should always close a file you have written to because Python may buffer (keep stored temporarily somewhere, for efficiency reasons) the data you have written, and if your program crashes for some reason, the data might not be written to the file at all. The safe thing is to close your files after you’re finished with them. If you want to be certain that your file is closed, you should use a try/finally statement with the call to close in the finally clause:



**Iterating Over File Contents**

Now you’ve seen some of the methods file objects present to us, and you’ve learned how to acquire such file objects. One of the common operations on files is to iterate over their contents, repeatedly performing some action as you go. There are many ways of doing this, and although you can find your favorite and stick to that, others may have done it differently, and to understand their programs, you should know all the basic techniques. Some of these techniques are just applications of the methods you’ve already seen (*read*, *readline*, and *readlines*), while some are new in this chapter (for example, *xreadlines* and *file iterators*).

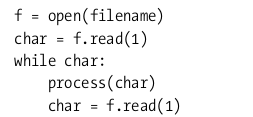
In all the examples in this section, I use a fictitious function called *process* to represent the processing of each character or line. Feel free to implement it in any way you like. One simple example would be the following:



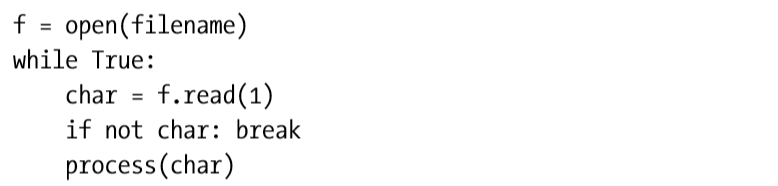
More useful implementation could do such things as storing data in a data structure, computing a sum, replacing patterns with the **re** module, or perhaps adding line numbers.

**Doing It Byte by Byte**

One of the most basic (but probably least common) ways of iterating over file contents is to use the read method in a while loop. For example, you might want to loop over every character (byte) in the file. You could do that as shown below:

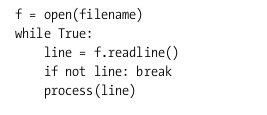


This program works because when you have reached the end of the file, the read method returns an empty string, but until then, the string always contains one character (and thus has the Boolean value true). So as long as char is true, you know that you aren’t finished yet.



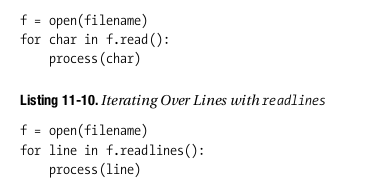
**One Line at a Time**

When dealing with text files, you are often interested in iterating over the lines in the file, not each individual character. You can do this easily in the same way as we did with characters, using the ***readline*** method (described earlier, in the section “Reading and Writing Lines”), as shown below:



**Reading Everything**

When dealing with text files, you are often interested in iterating over the lines in the file, not each individual character. You can do this easily in the same way as we did with characters, using the readline method (described earlier, in the section “**Reading and Writing Lines**”), as shown below:



**A Quick Summary**

In this section, you’ve seen how to interact with the environment through files and file-like objects, one of the most important techniques for I/O (input/output) in Python. Here are some of the highlights from this section:

**File-like objects.** A file-like object is (informally) an object that supports a set of methods such as read and readline (and possibly write and writelines).

**Opening and closing files.** You open a file with the open function (in newer versions of Python, actually just an alias for file), by supplying a file name.

**Modes and file types**. When opening a file, you can also supply a mode, such as 'r' for read mode or 'w' for write mode. By appending 'b' to your mode, you can open files as binary files. (This is necessary only on platforms where Python performs line-ending conversion, such as Windows.)

**Reading and writing.** You read from a file or file-like object using the method read. You write with the method write.

**Reading and writing lines**. You can read lines from a file using readline, readlines, and (for efficient iteration) xreadlines. You can write files with writelines.

**Iterating over file contents.** There are many ways of iterating over file contents. It is most common to iterate over the lines of a text file, and you can do this by simply iterating over the file itself. There are other methods too, such as readlines and writelines, that are compatible with older versions of Python.

One of programming little annoyances is that microsoft windows uses a backslash character between folder names while almost every other computer uses a forward slash.



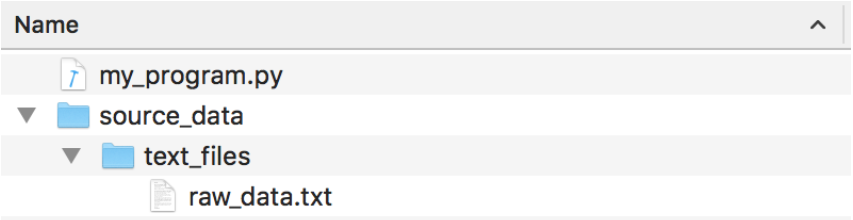
This is an accident of early [1980’s computer history](https://docs.microsoft.com/en-us/archive/blogs/larryosterman/why-is-the-dos-path-character). The first version of MS-DOS used the forward slash character for specifying command-line options. When Microsoft added support for folders in MS-DOS 2.0, the forward slash character was already taken so they used a backslash instead. Thirty-five years later, we are still stuck with this incompatibility.

If you want your Python code to work on both Windows and Mac/Linux, you’ll need to deal with these kinds of platform-specific issues. Luckily, Python 3 has a new module named **pathlib** that makes working with files nearly painless.

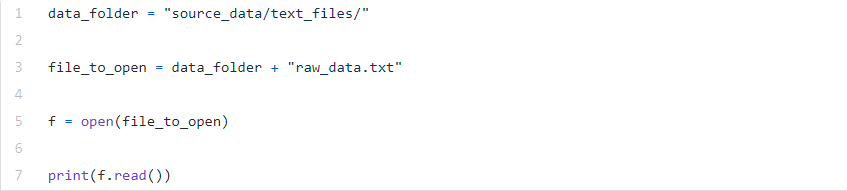
Let’s take a quick look at the different ways of handling filename paths and see how pathlib can make your life better!

The Wrong Solution: Building File Paths by Hand

Let’s say you have a data folder that contains a file that you want to open in your Python program:



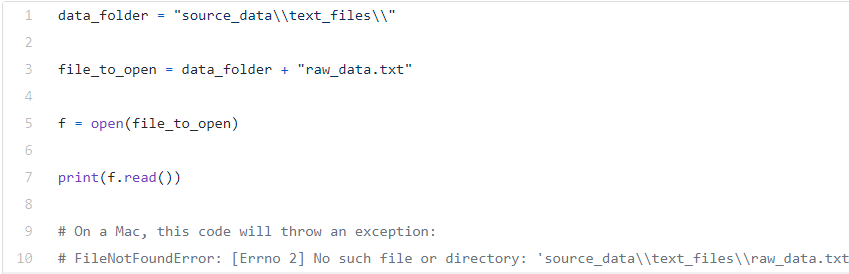
This is the wrong way to code it in Python:



Notice that I’ve hardcoded the path using Unix-style forward slashed since I’m on a Mac. This will make Windows users angry.

Technically this code will still work on Windows because Python has a hack where it will recognize either kind of slash when you call **open()** on Windows. But even still, you shouldn’t depend on that. Not all Python libraries will work if you use the wrong kind of slash on the wrong operating system - especially if they interface with external programs or libraries.

And Python’s support for mixing slash types is a Windows-only hack that doesn't work in reverse. Using backslashes in code will totally fail on a Mac:



For all these reasons and more, writing code with hardcoded path strings is the kind of thing that will make other programmers look at you with great suspicion. In general you should try to avoid it.

**Compulsory Task- After the quiz**