CHAPTER 11

Files and Stuff

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`, `raw_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 chapter 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:

```
open(name[, mode[, buffering]])
```

The `open` function takes a file name as its only mandatory argument, and returns a file object. The mode and buffering arguments are both optional and will be explained in the following sections.

Assuming that you have a text file (created with your text editor, perhaps) called `somefile.txt` stored in the directory `C:\text` (or something like `~/text` in UNIX), you can open it like this:

```
>>> f = open(r'C:\text\somefile.txt')
```

If the file doesn't exist, you may see an exception traceback like this:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in ?
IOError: [Errno 2] No such file or directory: "C:\\text\\somefile.txt"
```

You'll see what you can do with such file objects in a little while, but first, let's take a look at the other two arguments of the `open` function.

File Modes

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 must state that explicitly, supplying a *mode*. (Be patient—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 11-1.

Table 11-1. *Most Common Values for the Mode Argument of the open Function*

Value	Description
`'r'`	Read mode
`'w'`	Write mode
`'a'`	Append 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. (For this to be useful, you will probably want to use `seek` as well; see the sidebar "Random Access" later in this chapter.)

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.

If you find it shocking that Python manipulates your text files, don't worry. The only "trick" it employs is to standardize your line endings. Generally, in Python, you end your lines with a newline character (`\n`), as is the norm in UNIX systems. This is not standard in Windows, however. In Windows, a line ending is marked with `\r\n`. To hide this from your program (so it can work seamlessly across different platforms), Python does some automatic conversion here. When you read text from a file in text mode in Windows, it converts `\r\n` to `\n`. Conversely, when you write text to a file in text mode in Windows, it converts `\n` to `\r\n`. (The Macintosh version does the same thing, but converts between `\n` and `\r`.)

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 mentioned in the previous paragraph, and if you are using text mode, Python performs 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.

Note that this distinction is not important on platforms (such as UNIX) where the newline character is the standard line terminator, because no conversion is performed there anyway.

■ **Note** Files can be opened in universal newline support mode, using the mode character `U` together with, for example, `r`. In this mode, all line-ending characters/strings (`\r\n`, `\r`, or `\n`) are then converted to newline characters (`\n`), regardless of which convention is followed on the current platform.

Buffering

The `open` function takes a third (optional) parameter, which controls the *buffering* of the file. If the parameter is 0 (or `False`), input/output (I/O) 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 Files," later in this chapter). 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 about some basic methods of file objects (and some other file-like objects, sometimes called *streams*).

■ **Note** You will probably run into the term *file-like* repeatedly in your Python career (I've used it a few times already). A file-like object is simply one supporting a few of the same methods as a file, most notably either `read` or `write` or both. The objects returned by `urllib.urlopen` (see Chapter 14) are a good example of this. They support methods such as `read`, `readline`, and `readlines`, but not (at the time of writing) methods such as `isatty`, for example.

THREE STANDARD STREAMS

In Chapter 10, in the section about the `sys` module, I mentioned three standard streams. These are actually files (or file-like objects), and you can apply most of what you learn about files to them.

A standard source of data input is `sys.stdin`. When a program reads from standard input, you can supply text by typing it, or you can link it with the standard output of another program, using a *pipe*, as demonstrated in the section "Piping Output." (This is a standard UNIX concept.)

The text you give to `print` appears in `sys.stdout`. The prompts for `input` and `raw_input` also go there. Data written to `sys.stdout` typically appears on your screen, but can be rerouted to the standard input of another program with a pipe, as mentioned.

Error messages (such as stack traces) are written to `sys.stderr`. In many ways, it is similar to `sys.stdout`.

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:

```
>>> f = open('somefile.txt', 'w')
>>> f.write('Hello, ')
>>> f.write('World!')
>>> f.close()
```

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 chapter.

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

Here's an example (continuing where I left off):

```
>>> f = open('somefile.txt', 'r')
>>> f.read(4)
'Hell'
>>> f.read()
'o, World!'
```

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.

Piping Output

In a UNIX shell (such as GNU bash), you can write several commands after one another, linked together with *pipes*, as in this example (assuming GNU bash):

```
$ cat somefile.txt | python somescript.py | sort
```

■ **Note** GNU bash is also available in Windows. For more information, visit `http://www.cygwin.com`. In Mac OS X, the shell is available out of the box, through the Terminal application, for example.

This pipeline consists of three commands:

- `cat somefile.txt`: This command simply writes the contents of the file `somefile.txt` to standard output (`sys.stdout`).
- `python somescript.py`: This command executes the Python script `somescript`. The script presumably reads from its standard input and writes the result to standard output.
- `sort`: This command reads all the text from standard input (`sys.stdin`), sorts the lines alphabetically, and writes the result to standard output.

But what is the point of these pipe characters (|), and what does `somescript.py` do?

The pipes link up the standard output of one command with the standard input of the next. Clever, eh? So you can safely guess that `somescript.py` reads data from its `sys.stdin` (which is what `cat somefile.txt` writes) and writes some result to its `sys.stdout` (which is where `sort` gets its data).

A simple script (`somescript.py`) that uses `sys.stdin` is shown in Listing 11-1. The contents of the file `somefile.txt` are shown in Listing 11-2.

Listing 11-1. *Simple Script That Counts the Words in sys.stdin*

```
# somescript.py
import sys
text = sys.stdin.read()
words = text.split()
wordcount = len(words)
print 'Wordcount:', wordcount
```

Listing 11-2. *A File Containing Some Nonsensical Text*

```
Your mother was a hamster and your
father smelled of elderberries.
```

Here are the results of `cat somefile.txt | python somescript.py`:

```
Wordcount: 11
```

RANDOM ACCESS

In this chapter, I treat files only as streams—you can read data only from start to finish, strictly in order. In fact, you can also move around a file, accessing only the parts you are interested in (called *random access*) by using the two file-object methods `seek` and `tell`.

The method `seek(offset[, whence])` moves the current position (where reading or writing is performed) to the position described by `offset` and `whence`. `offset` is a byte (character) count. `whence` defaults to 0, which means that the offset is from the beginning of the file (the offset must be nonnegative). `whence` may also be set to 1 (move relative to current position; the offset may be negative), or 2 (move relative to the end of the file). Consider this example:

```
>>> f = open(r'c:\text\somefile.txt', 'w')
>>> f.write('0123456789012345678 9')
>>> f.seek(5)
>>> f.write('Hello, World!')
>>> f.close()
>>> f = open(r'c:\text\somefile.txt')
>>> f.read()
'01234Hello, World!89'
```

The method `tell()` returns the current file position, as in the following example:

```
>>> f = open(r'c:\text\somefile.txt')
>>> f.read(3)
'012'
>>> f.read(2)
'34'
>>> f.tell()
5L
```

Note that the number returned from `f.tell` in this case was a long integer. That may not always be the case.

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.

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 need to add those yourself. Also, there is no `writeline` method because you can just use `write`.

■ **Note** On platforms that use other line separators, substitute "carriage return" (Mac) or "carriage return and newline" (Windows) for "newline" (as determined by `os.linesep`).

Closing 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. However, closing those files can't hurt, and might help to avoid keeping the file uselessly "locked" against modification in some operating systems and settings. It also avoids using up any quotas for open files your system might have.

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:

```
# Open your file here
try:
    # Write data to your file
finally:
    file.close()
```

There is, in fact, a statement designed specifically for this situation (introduced in Python 2.5)—the `with` statement:

```
with open("somefile.txt") as somefile:
    do_something(somefile)
```

The `with` statement lets you open a file and assign it to a variable name (in this case, `soefile`). You then write data to your file (and, perhaps, do other things) in the body of the statement, and the file is automatically closed when the end of the statement is reached, even if that is caused by an exception.

In Python 2.5, the `with` statement is available only after the following import:

```
from __future__ import with_statement
```

In later versions, the statement is always available.

■ **Tip** After writing something to a file, you usually want the changes to appear in that file, so other programs reading the same file can see the changes. Well, isn't that what happens, you say? Not necessarily. As mentioned, the data may be *buffered* (stored temporarily somewhere in memory), and not written until you close the file. If you want to keep working with the file (and not close it) but still want to make sure the file on disk is updated to reflect your changes, call the file object's `flush` method. (Note, however, that `flush` might not allow other programs running at the same time to access the file, due to locking considerations that depend on your operating system and settings. Whenever you can conveniently close the file, that is preferable.)

CONTEXT MANAGERS

The `with` statement is actually a quite general construct, allowing you to use so-called *context managers*. A context manager is an object that supports two methods: `__enter__` and `__exit__`.

The `__enter__` method takes no arguments. It is called when entering the `with` statement, and the return value is bound to the variable after the `as` keyword.

The `__exit__` method takes three arguments: an exception type, an exception object, and an exception traceback. It is called when leaving the method (with any exception raised supplied through the parameters). If `__exit__` returns false, any exceptions are suppressed.

Files may be used as context managers. Their `__enter__` methods return the file objects themselves, while their `__exit__` methods close the files. For more information about this powerful, yet rather advanced, feature, check out the description of context managers in the Python Reference Manual. Also see the sections on context manager types and on `contextlib` in the Python Library Reference.

Using the Basic File Methods

Assume that `somefile.txt` contains the text in Listing 11-3. What can you do with it?

Listing 11-3. *A Simple Text File*

```
Welcome to this file
There is nothing here except
This stupid haiku
```

Let's try the methods you know, starting with `read(n)`:

```
>>> f = open(r'c:\text\somefile.txt')
>>> f.read(7)
'Welcome'
>>> f.read(4)
' to '
>>> f.close()
```

Next up is `read()`:

```
>>> f = open(r'c:\text\somefile.txt')
>>> print f.read()
Welcome to this file
There is nothing here except
This stupid haiku
>>> f.close()
```

Here's `readline()`:

```
>>> f = open(r'c:\text\somefile.txt')
>>> for i in range(3):
        print str(i) + ': ' + f.readline(),
0: Welcome to this file
1: There is nothing here except
2: This stupid haiku
>>> f.close()
```

And here's `readlines()`:

```
>>> import pprint
>>> pprint.pprint(open(r'c:\text\somefile.txt').readlines())
['Welcome to this file\n',
'There is nothing here except\n',
'This stupid haiku']
```

Note that I relied on the file object being closed automatically in this example.

Now let's try writing, beginning with `write(string)`:

```
>>> f = open(r'c:\text\somefile.txt', 'w')
>>> f.write('this\nis no\nhaiku')
>>> f.close()
```

After running this, the file contains the text in Listing 11-4.

Listing 11-4. *The Modified Text File*

```
this
is no
haiku
```

Finally, here's `writelines(list)`:

```
>>> f = open(r'c:\text\somefile.txt')
>>> lines = f.readlines()
>>> f.close()
>>> lines[1] = "isn't a\n"
>>> f = open(r'c:\text\somefile.txt', 'w')
>>> f.writelines(lines)
>>> f.close()
```

After running this, the file contains the text in Listing 11-5.

Listing 11-5. *The Text File, Modified Again*

```
this
isn't a
haiku
```

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 you can certainly just find your favorite and stick to that. However, 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`); others I'll introduce here (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. Here's one simple example:

```
def process(string):
    print 'Processing: ', string
```

More useful implementations 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.

Also, to try out the examples, you should set the variable `filename` to the name of some actual file.

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 in Listing 11-6.

Listing 11-6. *Looping over Characters with read*

```
f = open(filename)
char = f.read(1)
while char:
    process(char)
    char = f.read(1)
f.close()
```

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). As long as `char` is true, you know that you aren't finished yet.

As you can see, I have repeated the assignment `char = f.read(1)`, and code repetition is generally considered a bad thing. (Laziness is a virtue, remember?) To avoid that, I can use the `while True`/`break` technique introduced in Chapter 5. The resulting code is shown in Listing 11-7.

Listing 11-7. *Writing the Loop Differently*

```
f = open(filename)
while True:
    char = f.read(1)
    if not char: break
    process(char)
f.close()
```

As mentioned in Chapter 5, you shouldn't use the `break` statement too often (because it tends to make the code more difficult to follow). Even so, the approach shown in Listing 11-7 is usually preferred to that in Listing 11-6, precisely because you avoid duplicated code.

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 in Listing 11-8.

Listing 11-8. *Using readline in a while Loop*

```
f = open(filename)
while True:
    line = f.readline()
    if not line: break
    process(line)
f.close()
```

Reading Everything

If the file isn't too large, you can just read the whole file in one go, using the `read` method with no parameters (to read the entire file as a string), or the `readlines` method (to read the file into a list of strings, in which each string is a line). Listings 11-9 and 11-10 show how easy it is to iterate over characters and lines when you read the file like this. Note that reading the contents of a file into a string or a list like this can be useful for other things besides iteration. For example, you might apply a regular expression to the string, or you might store the list of lines in some data structure for further use.

Listing 11-9. *Iterating over Characters with read*

```
f = open(filename)
for char in f.read():
    process(char)
f.close()
```

Listing 11-10. *Iterating over Lines with readlines*

```
f = open(filename)
for line in f.readlines():
    process(line)
f.close()
```

Lazy Line Iteration with fileinput

Sometimes you need to iterate over the lines in a very large file, and `readlines` would use too much memory. You could use a `while` loop with `readline`, of course, but in Python, `for` loops are preferable when they are available. It just so happens that they are in this case. You can use a method called *lazy line iteration*—it's lazy because it reads only the parts of the file actually needed (more or less).

You have already encountered `fileinput` in Chapter 10. Listing 11-11 shows how you might use it. Note that the `fileinput` module takes care of opening the file. You just need to give it a file name.

Listing 11-11. *Iterating over Lines with fileinput*

```
import fileinput
for line in fileinput.input(filename):
    process(line)
```

■ **Note** In older code, you may also see lazy line iteration performed using the `xreadlines` method. It works almost like `readlines` except that it doesn't read all the lines into a list. Instead it creates an `xreadlines` object. Note that `xreadlines` is somewhat old-fashioned, and you should instead use `fileinput` or file iterators (explained next) in your own code.

File Iterators

It's time for the coolest (and, perhaps, the most common) technique of all. If Python had had this since the beginning, I suspect that several of the other methods (at least `xreadlines`) would never have appeared. So what is this cool technique? In current versions of Python (from version 2.2), files are *iterable,* which means that you can use them directly in `for` loops to iterate over their lines. See Listing 11-12 for an example. Pretty elegant, isn't it?

Listing 11-12. *Iterating over a File*

```
f = open(filename)
for line in f:
    process(line)
f.close()
```

In these iteration examples, I have explicitly closed my files. Although this is generally a good idea, it's not critical, as long as I don't write to the file. If you are willing to let Python take care of the closing, you could simplify the example even further, as shown in Listing 11-13. Here, I don't assign the opened file to a variable (like the variable `f` I've used in the other examples), and therefore I have no way of explicitly closing it.

Listing 11-13. *Iterating over a File Without Storing the File Object in a Variable*

```
for line in open(filename):
    process(line)
```

Note that `sys.stdin` is iterable, just like other files, so if you want to iterate over all the lines in standard input, you can use this form:

```
import sys
for line in sys.stdin:
    process(line)
```

Also, you can do all the things you can do with iterators in general, such as converting them into lists of strings (by using `list(open(filename))`), which would simply be equivalent to using `readlines`.

Consider the following example:

```
>>> f = open('somefile.txt', 'w')
>>> f.write('First line\n')
>>> f.write('Second line\n')
>>> f.write('Third line\n')
>>> f.close()
>>> lines = list(open('somefile.txt)')
>>> lines
['First line\n', 'Second line\n', 'Third line\n']
>>> first, second, third = open('somefile.txt')
>>> first
'First line\n'
>>> second
'Second line\n'
>>> third
'Third line\n'
```

In this example, it's important to note the following:

- I've used `print` to write to the file. This automatically adds newlines after the strings I supply.
- I use sequence unpacking on the opened file, putting each line in a separate variable. (This isn't exactly common practice because you usually won't know the number of lines in your file, but it demonstrates the "iterability" of the file object.)
- I close the file after having written to it, to ensure that the data is flushed to disk. (As you can see, I haven't closed it after reading from it. Sloppy, perhaps, but not critical.)

A Quick Summary

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

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. If you want to make sure your file is closed, even if something goes wrong, you can use the `with` statement.

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, but might be prudent elsewhere, too.)

Standard streams: The three standard files (`stdin`, `stdout`, and `stderr`, found in the `sys` module) are file-like objects that implement the UNIX *standard I/O* mechanism (also available in 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 `xreadlines`, that are compatible with older versions of Python.

New Functions in This Chapter

Function	Description
`file(name[, mode[, buffering]])`	Opens a file and returns a file object.
`open(name[, mode[, buffering]])`	Alias for `file`; use `open` rather than `file` when opening a file.

What Now?

So now you know how to interact with the environment through files, but what about interacting with the user? So far we've used only `input`, `raw_input`, and `print`, and unless the user writes something in a file that your program can read, you don't really have any other tools for creating user interfaces. That changes in the next chapter, where I cover graphical user interfaces, with windows, buttons, and so on.