CHAPTER 13

Reading and writing files

13.1 Opening files and file objects

Use open *to open a file for reading or writing.* open *returns a file object.*

Probably the single most common thing you will want to do with files is open and read them. In Python, opening and reading a file is accomplished using the built-in `open` function, and various different built-in reading operations. This following short Python program reads in one line from a text file named "`myfile`":

```
fileobject = open("myfile", 'r')
line = fileobject.readline()
```

`open` doesn't actually read anything from the file; instead it returns an abstract object called a *file object*, representing the opened file. All Python file I/O is done using file objects, rather than file names.

The first call to `readline` returns the first line in the file object, everything up to and including the first newline character, or all of the file if there is no newline character in the file; the next call to `readline` would return the second line, and so on. So, a file object is something that keeps track of a file and how much of the file has been read, or written.

The first argument to the open function is a pathname. In the above example we are opening what we expect to be an already existing file in the current working directory. The following would open a file at the given absolute location:

```
import os
fileName = os.path.join("c:", "My Documents", "test", "myfile")
fileobject = open(fileName,'r')
```

13.2 Closing files

Once all data has been read from or written to a file object, it should be closed. Closing a file object frees up system resources, allows the underlying file to be read or written to by other code, and, in general, makes the program more reliable. In small scripts, not closing a file object will generally not have much of an effect; file objects are automatically closed when the script or program terminates. In larger programs, too many open file objects may exhaust system resources, causing the program to abort.

Use `close` to close a file object once you are done with it.

Closing file objects is done using the `close` method, after the file object is no longer needed. The short program above then becomes this:

```
fileobject = open("myfile", 'r')
line = fileobject.readline()
# . . . any further reading on the fileobject . . .
fileobject.close()
```

13.3 Opening files in write or other modes

`open` will open a file for reading, (over)writing, or appending, depending on the value if its second argument, which can be `'r'`, `'w'`, or `'a'`.

The second argument of the open command is a single character denoting how the file should be opened. `'r'` means open the file for reading: `'w'` means open the file for writing (any data already in the file will be erased): and `'a'` means open the file for writing (new data will be appended to the end of any data already in the file). If you want to open the file for reading, you can leave out the second argument. `'r'` is the default. The following short program writes "Hello, World" to a file:

```
fileobject = open("myfile", 'w')
fileobject.write("Hello, World\n")
fileobject.close()
```

Depending on the operating system, `open` may also have access to additional file modes. In general, these are not necessary for most purposes. As you write more advanced Python programs, you may wish to consult the Python reference manuals for details.

As well, `open` can take an optional third argument, which defines how `read`s or `write`s for that file are *buffered*. Buffering is the process of holding data in memory until enough has been requested or written to justify the time cost of doing a disk access. Again, this is not something you typically need to worry about, but as you become more advanced in your use of Python, you may wish to read up on it.

13.4 Functions to read and write text or binary data

The `readline` method reads and returns a single line from a file object, up to and including the next newline.

The most common text file-reading function, `readline`, was presented above. It reads and returns a single line from a file object, including any newline character on the end of the line. If there is nothing more to be read from the file, `readline` returns an empty string. This makes it easy to, for example, count the number of lines in a file:

```
fileobject = open("myfile", 'r')
count = 0
while fileobject.readline() != "":
   count = count + 1
print count
fileobject.close()
```

`readlines` can be used to read in all lines from a file object, and return them as a list of strings.

For this particular problem, an even shorter way of counting all of the lines is using the built-in `readlines` method, which reads *all* of the lines in a file, and returns them as a list of strings, one string per line (with trailing newlines still included):

```
fileobject = open("myfile", 'r')
print len(fileobject.readlines())
fileobject.close()
```

An optional argument to `readline` or `readlines` can limit the amount of data they read in at any one time.

Of course, if you happen to be counting all of the lines in a particularly huge file, this might cause your computer to run out of memory, since it does read the entire file into memory at once. It is also possible to overflow memory with `readline`, if you have the misfortune to try to read a line from a huge file that contains no newline characters, although this is highly unlikely. To handle such circumstances, both `readline` and `readlines` can take an optional argument affecting the amount of data they read at any one time. See the Python reference documentation for details.

Use the `read` method to read a byte sequence from a file—either the entire file, or a fixed number of bytes.

Use binary mode (i.e. `"rb"` or `"wb"`) when working with binary files.

On some occasions, you might wish to read all of the data in a file into a single string, especially if the data is not actually a string, and you simply want to get it all into memory so you can treat it as a byte sequence. Or you might wish to read data from a file as strings of a fixed size. For example, you might be reading data without explicit newlines, where each line is assumed to be a sequence of characters of a fixed size. To do this, use the `read` method. Without any argument, it will read all of the rest of a file and return that data as a string. With a single integer argument, it will read that number of bytes, or less, if there is not enough data in the file to satisfy the request, and return a string of the given size. A possible problem may arise due to the fact that on Windows and Macintosh machines text mode translations will occur if you use the `open` command. On Macintosh any `"\r"` will be converted to `"\n"` while on Windows `"\r\n"` pairs will be converted to `"\n"` and `"\32"` will be taken as an EOF character. Use the `'b'` (binary) argument `open("file",'rb')` or `open("file",'wb')`, to open the file in binary mode to eliminate this issue. This will work transparently on UNIX platforms.

```
# Open a file for reading.
input = open("myfile", 'rb')
# Read the first four bytes as a header string.
header = input.read(4)
```

```
# Read the rest of the file as a single piece of data.
data = input.read()
input.close()
```

Use `write` to write a string or byte sequence to a file, and `writelines` to write a list of strings to a file.

The converses of the `readline` and `readlines` methods are the `write` and `writelines` methods. Note that there is no writeline function. `write` writes a single string, which could span multiple lines if newline characters are embedded within the string, for example something like:

```
myfile.write("Hello")
```

`write` does not write out a newline after it writes its argument; if you want a newline in the output, you must put it there yourself. If you open a file in text mode (using w), any '\n' characters will be mapped back to the platform-specific line endings (i.e., '\r\n' on Windows or '\r' on Macintosh platforms). Again opening the file in binary mode (i.e., 'wb') will avoid this.

`writelines` is something of a misnomer; it doesn't necessarily write lines—it simply takes a list of strings as an argument, and writes them, one after the other, to the given file object, without writing newlines. If the strings in the list end with newlines, they will be written as lines, otherwise they will be effectively concatenated together in the file. However, `writelines` is a precise inverse of `readlines`, in that it can be used on the list returned by `readlines` to write a file identical to the file `readlines` read from. For example, assuming `myfile.txt` exists and is a text file, this bit of code will create an exact copy of `myfile.txt` called `myfile2.txt`:

```
input = open("myfile.txt", 'r')
lines = input.readlines()
input.close()
output = open("myfile2.txt", 'w')
output.writelines(lines)
output.close()
```

13.5 Screen input/output and redirection

`raw_input` can prompt for and read in a string.

The built-in `raw_input` method can be use to prompt for and read an input string.

```
>>> x = raw_input("enter file name to use:")
enter file name to use:myfile
>>> s
⇒  'myfile'
```

The prompt line is optional and the newline at the end of the input line is stripped off. We could read in numbers with `raw_input`, obtaining a string version that we convert.

```
>>> x = int(raw_input("enter your number:"))
enter your number: 39
>>> x
⇒  39
```

A more general approach is to use another built-in function, `input`.

```
>>> x = input("enter your number:")
enter your number: 39
```

```
>>> x
⇒  39
```

With `input` we have excellent flexibility as it can actually read in any valid Python expression. Thus, we can read in a floating point or complex number or a delimited string. Anything that is not a valid expression will result in a `syntaxError` exception being raised.

`input` can prompt for and read in any valid Python expression.

```
>>> x = input("enter your number:")
enter your number: 47+3j
>>> x
 ⇒ (47+3j)
>>> x = input()
4 + 10/2.0
>>> x
 ⇒ 9.0
>>> x = input("enter a delimited string:")
enter a delimited string: 'Here is my delimited string.'
>>> x
 ⇒ 'Here is my delimited string.'
>>> input("enter expression:")
enter expression:an undelimited string
Traceback (innermost last):
  File "<stdin>", line 1, in
  File "<string>", line 1
    an undelimited string
                ^
SyntaxError: invalid syntax
```

● **Any valid Python expression can be input.**

● **An undelimited string is not a valid Python expression so a `SyntaxError` is raised.**

The specialized file objects `sys.stdin`, `sys.stdout`, and `sys.stderr` are set to the standard input, standard output and standard error, respectively.

Both `raw_input` and `input` write their prompt to the *standard output* and read from the *standard input*. Lower level access to these and *standard error* can be had using the `sys` module. It has `sys.stdin`, `sys.stdout`, and `sys.stderr` attributes. These can be treated as specialized file objects.

For `sys.stdin` we have `read`, `readline`, and `readlines` methods. For `sys.stdout` and `sys.stderr` there are the `write` and `writelines` methods. These operate as they do for other file objects.

```
>>> import sys
>>> sys.stdout.write("Write to the standard output.\n")
Write to the standard output.
>>> s = sys.stdin.readline()
An input line
>>> s
⇒  'An input line\012'
```

We can redirect standard input to read from a file. Similarly, standard output or standard error can be set to write to files. They can also be subsequently programmatically restored to their original values using `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__`:

```
>>> import sys
>>> f= open("outFile.txt",W)
>>> sys.stdout = f
```

```
>>> sys.stdout.writelines(["A first line.\n","A seconde line.\n"])
>>> print "A line from the print statement""
>>> 3+4
>>> sys.stdout = sys.__stdout__
>>> f.close()
>>> 3+4
⇒  7
```

sys.stdin, sys.stdout, and sys.stderr can be redirected to files.

While the standard output was redirected, we received prompts and we would have received any tracebacks from errors, but no other output. If you are using IDLE, these examples using `sys.__stdout__` will not work as indicated. You will have to use the interpreter's interactive mode directly.

This would normally be used when you are running from a script or program. However, if you are using the interactive mode on Windows you might want to temporarily redirect standard output in order to capture what might otherwise scroll off the screen. The following short module implements a set of functions that provide this capability. Here, `CaptureOutput()` will redirect standard output to a file that defaults to `"captureFile.txt"`. The function `RestoreOutput()` will restore standard output to the default. Also `PrintFile()` will print this file to the standard output and `ClearFile()` will clear it of its current contents.

File mio.py

```
"""myIO: module, (contains functions CaptureOutput, RestoreOutput,
         PrintFile, and ClearFile )"""
import sys
_fileObject = None

def CaptureOutput(file="captureFile.txt"):
   """CaptureOutput(file='captureFile.txt'): redirect the standard
   output to 'file'."""
   global _fileObject
   print "output will be sent to file: %s" % (file)
   print "restore to normal by calling 'mu.RestoreOutput()'"
   _fileObject= open(file, 'w')
   sys.stdout = _fileObject

def RestoreOutput():
   """RestoreOutput(): restore the standard output back to the
         default (also closes the capture file)"""
   global _fileObject
   sys.stdout = sys.__stdout__
   _fileObject.close()
   print "standard output has been restored back to normal"

def PrintFile(file="captureFile.txt"):
   """PrintFile(file="captureFile.txt"): print the given file to the
         standard output"""
   f = open(file,'r')
   print f.read()

def ClearFile(file="captureFile.txt"):
```

```
    """ClearFile(file="captureFile.txt"): clears the contents of the
          given file"""
    f = open(file,'w')
    f.close()
```

13.6 The struct module

Generally speaking, when working with your own files, you probably don't want to read or write binary data in Python. For very simple storage needs, it is usually best to use textual input and output as described above. For more sophisticated applications, Python provides the ability to easily read or write arbitrary Python objects, pickling, described later in this chapter. This ability is much less error-prone than directly writing and reading your own binary data, and so is highly recommended.

However, there is at least one situation in which you will likely need to know how to read or write binary data, and that is when dealing with files which are generated or used by other programs. This section gives a short description of how to do this using the struct module. Refer to the Python reference documentation for more details.

Use the above-mentioned read/write methods to read or write byte sequences, and process those sequences with the struct module.

Python does not actually support explicit binary input or output. Instead, in keeping with its philosophy of modularization, it simply reads and writes strings, which are really just byte sequences, and provides the standard `struct` module to permit you to treat those strings as formatted byte sequences with some specific meaning.

Assume that we wish to read in a binary file called `data`, containing a series of records generated by a C program. Each record consists of a C short integer, a C double float, and a sequence of four characters that should be taken as a four-character string. We wish to read this data into a Python list of tuples, with each tuple containing an integer, floating-point number, and a string.

`struct` functions understand format strings which define how binary data is packed.

The first thing to do is to define a *format string* understandable to the `struct` module, which tells the module how the data in one of our records is packed. The format string uses characters meaningful to `struct` to indicate what type of data is expected where in a record. For example, the character `'h'` indicates the presence of a single C short integer, and the character `'d'` indicates the presence of a single C double-precision floating-point number. Not surprisingly, `'s'` indicates the presence of a string, and may be preceded by an integer to indicate the length of the string; `'4s'` indicates a string consisting of four characters. For our records, the appropriate format string is therefore `'hd4s'`. `struct` understands a wide range of numeric, character, and string formats. See the *Python Library Reference* for details.

Before we start reading records from our file, we need to know how many bytes to read at a time. Fortunately, `struct` includes a `calcsize` function, which simply takes our format string as argument and returns the number of bytes used to contain data in such a format.

To read each record, we will simply use the `read` method described previously. Then, the `struct.unpack` function conveniently returns a tuple of values by parsing a read record according to our format string. The program to read our binary data file is remarkably simple:

```
import struct
    recordFormat = 'hd4s'
    recordSize = struct.calcsize(recordFormat)
```

```
resultList = []
input = open("data", 'rb')
while 1:
      # Read in a single record.
      record = input.read(recordSize)
      # If the record is empty, it indicates we have reached the end of file, so quit the loop.
      # Note that we have made no provision for checking for file consistency,
      # i.e. that the file contains a number of bytes which is an integer multiple
      # of the record size. However, if the last record is an "odd" size, the
      # struct.unpack function will raise an error.
      if record == '':
         input.close()
         break
      # Unpack the record into a tuple, and append that tuple to the result list.
      resultList.append(struct.unpack(recordFormat, record))
```

As you might already have guessed, `struct` also provides the ability to take Python values and convert them into packed byte sequences. This is accomplished through the `struct.pack` function, which is almost, but not quite, an inverse of `struct.unpack`. The "almost" comes from the fact that while `struct.unpack` returns a tuple of Python values, `struct.pack` does not take a tuple of Python values; rather, it takes a format string as its first argument, and then enough additional arguments to satisfy the format string. So, to produce a binary record of the form used in the above example, we might do something like this:

```
>>>
```

```
>>> import struct
>>> recordFormat = 'hd4s'
>>> struct.pack(recordFormat, 7, 3.14, 'gbye')
⇒  '\007\000\000\000\000\000\000\037\205\353Q\270\036\011@gbye'
```

`struct` gets even better than this; you can insert other special characters into the format string to indicate that data should be read/written in big-endian, little-endian, or machine-native-endian format (default is machine-native), and to indicate that sizes of things like C short integer should either be sized as native to the machine (the default), or as standard C sizes. But, if you need these features, it's nice to know they exist. See the *Python Library Reference* for details.

13.7 Pickling objects into files

Pickling is a *major* benefit in Python. Use this ability!

Python can write any data structure into a file, and read that data structure back out of a file and re-create it, with just a few commands. This is an unusual ability, but one that is highly useful. It can save the programmer many pages of code which do nothing but dump the state of a program into a file (and can save a similar amount of code which does nothing but read that state back in).

Write an arbitrary Python object with `cPickle.dump`.

Python provides this ability via the `cPickle` module. `cPickle` is actually a C language rewrite of the original `pickle` module. We are using it in our examples here as it is a thousand times faster than the `pickle` module. Pickling is very powerful but very simple

to use. For example, assume that the entire state of a program is held in three variables, a, b, and c. We can save this state to a file called "state" as follows:

```
import cPickle
.
.
.
file = open("state", 'w')
cPickle.dump(a, file)
cPickle.dump(b, file)
cPickle.dump(c, file)
file.close()
```

It doesn't matter what was stored in a, b, and c. It might be as simple as numbers, or as complex as a list of dictionaries containing instances of user-defined classes. cPickle.dump will save everything.

Now, to read that data back in on a later run of the program, just say

Retrieve an arbitrary Python object with cPickle.load.

```
import cPickle
file = open("state", 'r')
a = cPickle.load(file)
b = cPickle.load(file)
c = cPickle.load(file)
file.close()
```

Any data that was previously in the variables a, b, or c will have been restored to them by cPickle.load.

cPickle can handle just about any Python object.

The cPickle module can store almost anything in this manner. The cPickle module can handle lists, tuples, numbers, strings, dictionaries, and just about anything made up of these types of objects, which includes all class instances. It also handles shared objects, cyclic references, and other complex memory structures correctly, storing shared objects only once, and restoring them as shared objects, not as identical copies. However, code objects (what Python stores byte compiled code in) and system resources like files or sockets can not be pickled.

A convenient way of using cPickle *is to save your state variables into a dictionary, and then* cPickle *the dictionary.*

More often than not, you won't want to save your entire program state with cPickle. For example, most applications can have multiple documents open at one time. If you saved the entire state of the program, you would effectively save all open documents in one file. An easy and effective way of saving and restoring only data of interest is to write a save function which stores all data you wish to save into a dictionary, and then uses cPickle to save the dictionary. Then, a complementary restore function can be used to read the dictionary back in (again using cPickle), and to assign the values in the dictionary to the appropriate program variables. This also has the advantage that there is no possibility of reading values back in an incorrect order, that is, an order different from the order in which they were stored. Using this approach with the above example, we would get code looking something like this:

```
import cPickle
.
.
.
```

```
def saveData():
    global a, b, c
    file = open("state", 'w')
    data = {'a' : a, 'b' : b, 'c' : c}
    cPickle.dump(data, file)
    file.close()

def restoreData():
    global a, b, c
    file = open("state", 'r')
    data = cPickle.load(file)
    file.close()
    a = data['a']
    b = data['b']
    c = data['c']
    .
    .
```

Now this is a somewhat contrived example. You probably won't be saving the state of the top-level variables of your interactive mode very often.

A real life application is an extension of the cache example given in the dictionary chapter. Recall that there we were calling a function that performed a time intensive calculation based on its three arguments. During the course of a program run many of our calls to it ended up being with the same set of arguments. We were able to obtain a significant performance improvement by caching the results in a dictionary, keyed by the arguments that produced them. However, it was also the case that many different sessions of this program were being run many times over the course of days, weeks, and months. Therefore, by pickling the cache we were able to keep from having to start over with every session. Following is a pared down version of the module for doing this.

File sole.py

```
"""sole module: contains function sole, save, show"""

import cPickle

_soleMemCacheD = {}
_soleDiskFileS = "solecache"

# This initialization code will be executed when this module is first loaded.
file = open(_soleDiskFileS, 'r')
_soleMemCacheD = cPickle.load(file)
file.close()

# Public functions
def sole(m, n, t):
    """sole(m,n,t): perform the sole calculation using the cache."""
    global _soleMemCacheD
    if _soleMemCacheD.has_key((m, n, t)):
         return _soleMemCacheD[(m, n, t)]
    else:
         # . . . do some time-consuming calculations . . .
         _soleMemCacheD[(m, n, t)] = result
```

```
        return result

def save():
    """save(): save the updated cache to disk."""
    global _soleMemCacheD, _soleDiskFileS
    file = open(_soleDiskFileS, 'w')
    cPickle.dump(_soleMemCacheD, file)
    file.close()

def show():
    """show(): print the cache"""
    global _soleMemCacheD
    print _soleMemCacheD
```

This code assumes the cache file already exists. If you want to play around with it, use the following to initialize the cache file:

```
>>> import cPickle
>>> file = open("solecache",W)
>>> cPickle.dump({}, file)
```

You will also, of course, need to replace the comment *"# . . . do some time-consuming calculations"* with an actual calculation. Note that for production code, this is a situation where you probably would use an absolute pathname for your cache file. Also, concurrency is not being handled here. If two people run overlapping sessions, you will only end up with the additions of the last person to save. If this were an issue, you could limit this overlap window significantly by using the dictionary update method in the `save` function.

13.8 Shelving objects

The shelve module permits you to pickle Python objects into files which appear as dictionaries. This is very useful for large data sets.

This is a somewhat advanced topic, but certainly not a difficult one. This section is likely of most interest to people whose work involves storing or accessing pieces of data in large files, because the Python `shelve` module does exactly that—it permits the reading or writing of pieces of data in large files, without reading or writing the entire file. For applications which perform many accesses of large files (such as database applications), the savings in time can be spectacular. Like the `cPickle` module (which it makes use of), the `shelve` module is very simple.

Let's explore it through an address book. This is the sort of thing that is usually small enough so that an entire address file could be read in when the application is started, and written out when the application is done. If you're an extremely friendly sort of person, and your address book is too big for this, better to use `shelve` and not worry about it.

We'll assume that each entry in our address book consists of a tuple of three elements, giving the first name, phone number, and address of a person. Each entry will be indexed by the last name of the person the entry refers to. This is so simple that our application will just be an interactive session with the Python shell.

First, import the `shelve` module, and open the address book. `shelve.open` will create the address book file if it does not exist:

```
>>>
```

```
>>> import shelve
```

```
>>> book = shelve.open("addresses")
```

Now, add a couple of entries. Notice that we're treating the object returned by `shelve.open` as a dictionary (though it is a dictionary which can only use strings as keys):

```
...
```

```
>>> book['flintstone'] = ('fred', '555-1234', '1233 Bedrock Place')
>>> book['rubble'] = ('barney', '555-4321', '1235 Bedrock Place')
```

Finally, close the file and end the session:

```
...
```

```
>>> book.close()
```

So what? Well, in that same directory, start Python again, and open the same address book:

```
>>>
```

```
>>> import shelve
>>> book = shelve.open("addresses")
```

But now, instead of entering something, let's see if what we put in before is still around:

```
...
```

```
>>> book['flintstone']
⇒  ('fred', '555-1234', '1233 Bedrock Place')
```

The "addresses" file created by `shelve.open` in the first interactive session has acted just like a persistent dictionary. The data we entered before was stored to disk, even though we did no explicit disk writes. That's exactly what `shelve` does.

Shelf objects are effectively persistent, string-indexed dictionaries.

More generally, `shelve.open` returns a shelf object which permits basic dictionary operations, key assignment or lookup, `del`, and the `has_key` and `keys` methods. However, unlike a normal dictionary, shelf objects store their data on disk, not in memory. Unfortunately, shelf objects do have one significant restriction as compared to dictionaries. They can only use strings as keys, versus the wide range of key types allowable in dictionaries.

Shelf objects do NOT read an entire file into memory—this can be a major advantage.

It's important to understand the advantage shelf objects give you over dictionaries when dealing with large data sets. `shelve.open` makes the file accessible; it does not read an entire shelf object file into memory. File accesses are done only when needed, typically when an element is looked up, and the file structure is maintained in such a manner that lookups are very fast. Even if your data file is really large, only a couple of disk accesses will be required to locate the desired object in the file. This can improve your program in a number of ways. It may start faster, since it does not need to read a potentially large file into memory. It may execute faster, since there is more memory available to the rest of the program, and thus less code will need to be swapped out into virtual memory. You can operate on data sets that are otherwise too large to fit in memory at all.

Shelf objects are not necessarily written to disk until they are closed.

There are a few restrictions when using the `shelve` module. As previously mentioned, shelf object keys can only be strings; however, any Python object that can be pickled can

be stored under a key in a shelf object. Also, shelf objects are not really suitable for multiuser databases, because they provide no control for concurrent access.

!!! Finally, make sure to `close` a shelf object when you are done—this is sometimes required in order for changes you've made (entries or deletions) to be written back to disk.

As written, the cache example of the previous section would be an excellent candidate to be handled using shelves. You would not, for example, have to rely on the user to explicitly save his work to the disk. The only possible issue is that it would not have the low-level control when you write back to the file.

13.9 Summary

File input and output in Python is a remarkably simple but powerful feature of the language. You can use various built-in functions to open, read, write, and close files. For very simple uses, you'll probably want to stick with reading and writing text, but the `struct` module does give you the ability to read or write packed binary data. Even better, the `cPickle` and `shelve` modules provide simple, safe, and powerful ways of saving and accessing arbitrarily complex Python data structures, which means you may never again need to worry about defining file formats for your programs.