Reading and writing files

This lab covers

- [Opening files and [file] objects]
- [Closing files]
- [Opening files in different modes]
- [Reading and writing text or binary data]
- [Redirecting screen input/output]
- [Using the [struct] module]
- [Pickling objects into files]
- [Shelving objects]

Opening files and file objects

Probably the single most common thing you'll want to do with files is open and read them.

In Python, you open and read a file by using the built-in [open] function and various built-in reading operations. The following short Python program reads in one line from a text file named myfile:

```
with open('myfile', 'r') as file_object:
    line = file_object.readline()
```

[open] doesn't read anything from the file; instead, it returns an object called a [file] object that you can use to access the opened file. A [file] object keeps track of a file and how much of the file has been read or written. All Python file I/O is done using [file] objects rather than filenames.

The first call to [readline] returns the first line in the [file] object, everything up to and including the first newline character or the entire file if there's no newline character in the file; the next call to [readline] returns the second line, if it exists, and so on.

The first argument to the [open] function is a pathname. In the previous example, you're opening what you expect to be an existing file in the current working directory. The following opens a file at an absolute location---[c:\My Documents\test\myfile]:

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

Note also that this example uses the [with] keyword, indicating that the file will be opened with a context manager, which I explain more in [chapter 14]. For now, it's enough to note that this style of opening files better manages potential I/O errors and is generally preferred.

Closing files

After 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. For small scripts, not closing a [file] object generally doesn't have much of an effect; [file] objects are automatically closed when the script or program terminates. For larger programs, too many open [file] objects may exhaust system resources, causing the program to abort.

You close a [file] object by using the [close] method when the [file] object is no longer needed. The earlier short program then becomes this:

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

Using a context manager and the keyword [with] is also a good way to automatically close files when you're done:

```
with open("myfile", 'r') as file_object:
    line = file_object.readline()
    # . . . any further reading on the file_object . . .
```

Opening files in write or other modes

The second argument of the [open] command is a string 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 appending" (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:

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

Depending on the operating system, [open] may also have access to additional file modes. These modes aren't necessary for most purposes. As you write more advanced Python programs, you may want to consult the Python reference manuals for details.

[open] can take an optional third argument, which defines how reads or writes for that file are buffered. *Buffering* is the process of holding data in memory until enough data has been requested or written to justify the time cost of doing a disk access. Other parameters to [open] control the encoding for text files and the handling of newline characters in text files. Again, these features aren't things you typically need to worry about, but as you become more advanced in your use of Python, you may want to read up on them.

Functions to read and write text or binary data

I've already presented the most common text file--reading function, [readline]. This function reads and returns a single line from a [file] object, including any newline character on the end of the line. If there's nothing more to be read from the file, [readline] returns an empty string, which makes it easy to (for example) count the number of lines in a file:

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

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

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

If you happen to be counting all the lines in a huge file, of course, this method may cause your computer to run out of memory because it reads the entire file into memory at once. It's 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 situation 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.

Another way to iterate over all of the lines in a file is to treat the [file] object as an iterator in a [for] loop:

```
file_object = open("myfile", 'r')
count = 0
for line in file_object:
    count = count + 1
print(count)
file_object.close()
```

This method has the advantage that the lines are read into memory as needed, so even with large files, running out of memory isn't a concern. The other advantage of this method is that it's simpler and easier to read.

A possible problem with the [read] method may arise due to the fact that on Windows and Macintosh machines, text-mode translations occur if you use the [open] command in text mode---that is, without adding a [b] to the mode. In text mode, on a Macintosh any [\r] is converted to ["\n"], whereas on Windows ["\r\n"] pairs are converted to ["\n"]. You can specify the treatment of newline characters by using the newline parameter when you open the file and specifying [newline="\n"], ["\r"], or ["\r\n"], which forces only that string to be used as a newline:

```
input_file = open("myfile", newline="\n")
```

This example forces only ["\n"] to be considered to be a newline. If the file has been opened in binary mode, the newline parameter isn't needed, because all bytes are returned exactly as they are in the file.

The write methods that correspond to the [readline] and [readlines] methods are the [write] and [writelines] methods. Note that there's no [writeline] function. [write] writes a single string, which can span multiple lines if newline characters are embedded within the string, as in this example:

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

[write] doesn't 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 are mapped back to the platform-specific line endings (that is, ['\r\n'] on Windows or ['\r'] on Macintosh platforms). Again, opening the file with a specified newline prevents this situation.

[writelines] is something of a misnomer because it doesn't necessarily write lines; it 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're written as lines; otherwise, they're effectively concatenated in the file. But [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. Assuming that myfile.txt exists and is a text file, this bit of code creates an exact copy of myfile.txt called myfile2.txt:

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

Using binary mode

On some occasions, you may want to read all the data in a file into a single [bytes] object, especially if the data isn't a string, and you want to get it all into memory so you can treat it as a byte sequence. Or you may want to read data from a file as [bytes] objects of a fixed size. You may be reading data without explicit newlines, for example, where each line is assumed to be a sequence of characters of a fixed size. To do so, use the [read] method. Without any argument, this method reads all of a file from the current position and returns that data as a [bytes] object. With a single-integer argument, it reads that number of bytes (or less, if there isn't enough data in the file to satisfy the request) and returns a [bytes] object of the given size:

```
input_file = open("myfile", 'rb')
header = input_file.read(4)
data = input_file.read()
input_file.close()
```

The first line opens a file for reading in binary mode, the second line reads the first four bytes as a header string, and the third line reads the rest of the file as a single piece of data.

Keep in mind that files open in binary mode deal only in bytes, not strings. To use the data as strings, you must decode any [bytes] objects to [string] objects. This point is often important in dealing with network protocols, where data streams often behave as files but need to be interpreted as bytes, not strings.

Quick Check

What is the significance of adding a ["b"] to the file open mode string, as in [open("file", "wb")]?

Suppose that you want to open a file named [myfile.txt] and write additional data on the end of it. What command would you use to open [myfile.txt]? What command would you use to reopen the file to read from the beginning?

Reading and writing with pathlib

In addition to its path-manipulation powers discussed in [chapter 12], a [Path] object can be used to read and write text and binary files. This capability can be convenient because no open or close is required, and separate methods are used for text and binary operations. One limitation, however, is that you have no way to append by using [Path] methods, because writing replaces any existing content:

```
>>> from pathlib import Path
>>> p_text = Path('my_text_file')
>>> p_text.write_text('Text file contents')
18
>>> p_text.read_text()
'Text file contents'
>>> p_binary = Path('my_binary_file')
>>> p_binary.write_bytes(b'Binary file contents')
20
>>> p_binary.read_bytes()
b'Binary file contents'
```

Screen input/output and redirection

You can use the built-in [input] method to prompt for and read an input string:

```
>>> x = input("enter file name to use: ")
enter file name to use: myfile
```

```
>>> x
'myfile'
```

The prompt line is optional, and the newline at the end of the input line is stripped off. To read in numbers by using [input], you need to explicitly convert the string that [input] returns to the correct number type. The following example uses [int]:

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

[input] writes its prompt to the *standard output* and reads from the *standard input*. Lower-level access to these and *standard error* can be obtained by using the [sys] module, which has [sys.stdin], [sys.stdout], and [sys.stderr] attributes. These attributes can be treated as specialized [file] objects.

For [sys.stdin], you have the [read], [readline], and [readlines] methods. For [sys.stdout] and [sys.stderr], you can use the standard [print] function as well as the [write] and [writelines] methods, which operate as they do for other [file] objects:

```
>>> import sys
>>> print("Write to the standard output.")
Write to the standard output.
>>> sys.stdout.write("Write to the standard output.\n")
Write to the standard output.
30
>>> s = sys.stdin.readline()
An input line
>>> s
'An input line\n'
```

You can redirect standard input to read from a file. Similarly, standard output or standard error can be set to write to files and then programmatically restored to their original values by 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 second line.\n"])
>>> print("A line from the print function")
>>> 3 + 4
>>> sys.stdout = sys.__stdout__
>>> f.close()
>>> 3 + 4
7
```

The [print] function also can be redirected to any file without changing standard output:

```
>>> import sys
>>> f = open("outfile.txt", 'w')
>>> print("A first line.\n", "A second line.\n", file=f)
>>> 3 + 4
7
```

```
>>> f.close()
>>> 3 + 4
7
```

While the standard output is redirected, you receive prompts and tracebacks from errors but no other output. If you're using IDLE, these examples using [sys.__ stdout__] won't work as indicated; you have to use the interpreter's interactive mode directly.

You'd normally use this technique when you're running from a script or program. But if you're using the interactive mode on Windows, you may want to temporarily redirect standard output to capture what might otherwise scroll off the screen. The short module shown here implements a set of functions that provides this capability.

Listing 13.1. File mio.py

```
"""mio: module, (contains functions capture_output, restore_output,
     print_file, and clear_file )"""
import sys
_file_object = None
def capture_output(file="capture_file.txt"):
    """capture_output(file='capture_file.txt'): redirect the standard
    output to 'file'."""
    global _file_object
    print("output will be sent to file: {0}".format(file))
    print("restore to normal by calling 'mio.restore_output()'")
    _file_object = open(file, 'w')
    sys.stdout = _file_object

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

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

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

Here, [capture_output()] redirects standard output to a file that defaults to ["capture_file.txt"]. The function [restore_output()] restores standard output to the default. Assuming [capture_output] hasn't been executed, [print_file()] prints this file to the standard output, and [clear_file()] clears its current contents.

Try this: Redirecting input and output

Write some code to use the mio.py module in [listing 13.1] to capture all the print output of a script to a file named myfile.txt, reset the standard output to the screen, and print that file to screen.

Reading structured binary data with 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's usually best to use text or bytes input and output. For more sophisticated applications, Python provides the ability to easily read or write arbitrary Python objects (*pickling*, described in [section 13.8]). This ability is much less error-prone than directly writing and reading your own binary data and is highly recommended.

But there's at least one situation in which you'll likely need to know how to read or write binary data: when you're dealing with files that are generated or used by other programs. This section describes how to do this by using the [struct] module. Refer to the Python reference documentation for more details.

As you've seen, Python supports explicit binary input or output by using bytes instead of strings if you open the file in binary mode. But because most binary files rely on a particular structure to help parse the values, writing your own code to read and split them into variables correctly is often more work than it's worth. Instead, you can use the standard [struct] module to permit you to treat those strings as formatted byte sequences with some specific meaning.

Assume that you want 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. You want to read this data into a Python list of tuples, with each tuple containing an integer, a floating-point number, and a string.

The first thing to do is define a *format string* understandable to the [struct] module, which tells the module how the data in one of your records is packed. The format string uses characters meaningful to [struct] to indicate what type of data is expected where in a record. The character ['h'], for example, 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. Any of these may be preceded by an integer to indicate the number of values; in this case, ['4s'] indicates a string consisting of four characters. For your 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 you start reading records from your file, you need to know how many bytes to read at a time. Fortunately, [struct] includes a [calcsize] function, which takes your format string as an argument and returns the number of bytes used to contain data in such a format.

To read each record, you use the [read] method described earlier in this lab. Then the [struct.unpack] function conveniently returns a tuple of values by parsing a read record according to your format string. The program to read your binary data file is remarkably simple:

```
import struct
record_format = 'hd4s'
record_size = struct.calcsize(record_format)
result_list = []
input = open("data", 'rb')
while 1:
    record = input.read(record_size)
    if record == '':
        input.close()
        break
    result_list.append(struct.unpack(record_format, record))
```

If the record is empty, you're at the end of the file, so you quit the loop ***2***. Note that there's no checking for file consistency; if the last record is an odd size, the [struct.unpack] function raises an error.

As you may already have guessed, [struct] also provides the ability to take Python values and convert them to packed byte sequences. This conversion 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 whereas [struct.unpack] returns a tuple of Python values, [struct.pack] doesn't 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. To produce a binary record of the form used in the previous example, you might do something like this:

```
>>> import struct
>>> record_format = 'hd4s'
>>> struct.pack(record_format, 7, 3.14, b'gbye')
b'\x07\x00\x00\x00\x00\x00\x00\x00\x1f\x85\xebQ\xb8\x1e\t@gbye'
```

[struct] gets even better; 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 things like a C short integer should be sized either as native to the machine (the default) or as standard C sizes. If you need these features, it's nice to know that they exist. See the *Python Library Reference* for details.

Quick Check: struct

What use cases can you think of in which the struct module would be useful for either reading or writing binary data?

Pickling objects files

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

Python provides this capability via the [pickle] module. Pickling is powerful but simple to use. Assume that the entire state of a program is held in three variables: [a], [b], and [c]. You can save this state to a file called state as follows:

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

It doesn't matter what was stored in [a], [b], and [c]. The content might be as simple as numbers or as complex as a list of dictionaries containing instances of user-defined classes. [pickle.dump] saves everything.

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

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

Any data that was previously in the variables [a], [b], or [c] is restored to them by [pickle.load].

The [pickle] module can store almost anything in this manner. It 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. But code objects (what Python uses to store byte-compiled code) and system resources (like files or sockets) can't be pickled.

More often than not, you won't want to save your entire program state with [pickle]. Most applications can have multiple documents open at one time, for example. 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 that stores all data you want to save into a dictionary and then uses [pickle] to save the dictionary. Then you can use a complementary restore function to read the dictionary back in (again using [pickle]) and to assign the values in the dictionary to the appropriate program variables. This technique also has the advantage that there's no possibility of reading values back in an incorrect order---that is, an order different from the order in which the values were stored. Using this approach with the previous example, you get code looking something like this:

```
import pickle
.
.
.
def save_data():
    global a, b, c
    file = open("state", 'wb')
    data = {'a': a, 'b': b, 'c': c}
    pickle.dump(data, file)
    file.close()

def restore_data():
    global a, b, c
    file = open("state", 'rb')
    data = pickle.load(file)
    file.close()
    a = data['a']
    b = data['b']
    c = data['c']
    .
    .
```

This example is somewhat contrived. 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 [chapter 7]. In that chapter, you called a function that performed a time-intensive calculation based on its three arguments. During the course of a program run, many of your calls to that function ended up using the same set of arguments. You were able to obtain a significant performance improvement by caching the results in a dictionary, keyed by the arguments that produced them. But it was also the case that many sessions of this program were being run many times over the course of days, weeks, and months. Therefore, by pickling the cache, you can avoid having to start over with every session. Here is a pared-down version of the module you might use for this purpose.

Listing 13.2. File sole.py

```
"""sole module: contains functions sole, save, show"""
import pickle
_sole_mem_cache_d = {}
_sole_disk_file_s = "solecache"
file = open(_sole_disk_file_s, 'rb')
_sole_mem_cache_d = pickle.load(file)
file.close()

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

def save():
    """save(): save the updated cache to disk."""
    global _sole_mem_cache_d, _sole_disk_file_s
    file = open(_sole_disk_file_s, 'wb')
    pickle.dump(_sole_mem_cache_d, file)
    file.close()

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

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

```
>>> import pickle
>>> file = open("solecache",'wb')
>>> pickle.dump({}, file)
>>> file.close()
```

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

Reasons not to pickle

Although it may make some sense to use a pickled object in the previous scenario, you should also be aware of the drawbacks to pickles:

- [Pickling is neither particularly fast nor space-efficient as a means of serialization. Even using JSON to store serialized objects is faster and results in smaller files on disk.]
- [Pickling isn't secure, and loading a pickle with malicious content can result in the execution of arbitrary code on your machine. Therefore, you should avoid pickling if there's *any* chance at all that the pickle file

will be accessible to anyone who might alter it.]

Quick Check: Pickles

Think about why a pickle would or would not be a good solution in the following use cases:

- [Saving some state variables from one run to the next]
- [Keeping a high-score list for a game]
- [Storing usernames and passwords]
- [Storing a large dictionary of English terms]

Shelving objects

This topic is somewhat advanced but certainly not difficult. You can think of a [shelve] object as being a dictionary that stores its data in a file on disk rather than in memory, which means that you still have the convenience of access with a key, but you don't have the limitations of the amount of available RAM.

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: permits the reading or writing of pieces of data in large files without reading or writing the entire file. For applications that perform many accesses of large files (such as database applications), the savings in time can be spectacular. Like the [pickle] module (which it uses), the [shelve] module is simple.

In this section, you explore this module through an address book. This sort of thing usually is small enough that an entire address file can 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 example, it would be better to use [shelve] and not worry about it.

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

First, import the [shelve] module, and open the address book. [shelve.open] creates the address book file if it doesn't exist:

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

Now add a couple of entries. Notice that you're treating the object returned by [shelve.open] as a dictionary (although it's a dictionary that can use only 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, see whether what you 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 you entered before was stored to disk, even though you did no explicit disk writes. That's exactly what [shelve] does.

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

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 doesn't 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, which can improve your program in several ways. The program may start faster, because it doesn't need to read a potentially large file into memory. Also, the program may execute faster because more memory is available to the rest of the program; thus, less code must be swapped out into virtual memory. You can operate on data sets that are otherwise too large to fit in memory.

You have a few restrictions when using the [shelve] module. As previously mentioned, [shelf] object keys can be only strings, but any Python object that can be pickled can be stored under a key in a [shelf] object. Also, [shelf] objects aren't suitable for multiuser databases because they provide no control for concurrent access. Make sure that you close a [shelf] object when you're finished; closing is sometimes required for the changes you've made (entries or deletions) to be written back to disk.

As written, the cache example in [listing 13.1] is an excellent candidate to be handled with shelves. You wouldn't, for example, have to rely on the user to explicitly save their work to the disk. The only possible issue is that you wouldn't have the low-level control when you write back to the file.

Quick Check: Shelve

Using a [shelf] object looks very much like using a dictionary. In what ways is using a [shelf] object different? What disadvantages would you expect in using a [shelf] object?

Lab 13: Final fixes to wc

If you look at the [man] page for the [wc] utility, you see two command-line options that do very similar things. [-c] makes the utility count the bytes in the file, and [-m] makes it count characters (which in the case of some Unicode characters can be two or more bytes long). In addition, if a file is given, it should read from and process that file, but if no file is given, it should read from and process [stdin].

Rewrite your version of the [wc] utility to implement both the distinction between bytes and characters and the ability to read from files and standard input.

Summary

- [File input and output in Python uses various built-in functions to open, read, write, and close files.]
- [In addition to reading and writing text, the [struct] module gives you the ability to read or write packed binary data.]
- [The [pickle] and [shelve] modules provide simple, safe, and powerful ways of saving and accessing arbitrarily complex Python data structures.]