13. Files

13.1. About files

While a program is running, its data is stored in *random access memory* (RAM). RAM is fast and inexpensive, but it is also **volatile**, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time the computer is turned on and the program is started, it has to be written to a **non-volatile** storage medium, such a hard drive, usb drive, or CD-RW.

Data on non-volatile storage media is stored in named locations on the media called **files**. By reading and writing files, programs can save information between program runs.

Working with files is a lot like working with a notebook. To use a notebook, it has to be opened. When done, it has to be closed. While the notebook is open, it can either be read from or written to. In either case, the notebook holder knows where they are. They can read the whole notebook in its natural order or they can skip around.

All of this applies to files as well. To open a file, we specify its name and indicate whether we want to read or write.

13.2. Writing our first file

Let's begin with a simple program that writes three lines of text into a file:

```
1 myfile = open("test.txt", "w")
2 myfile.write("My first file written from Python\n")
3 myfile.write("---------------------------------\n")
4 myfile.write("Hello, world!\n")
5 myfile.close()
```

Opening a file creates what we call a file **handle**. In this example, the variable `myfile` refers to the new handle object. Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk.

On line 1, the open function takes two arguments. The first is the name of the file, and the second is the **mode**. Mode `"w"` means that we are opening the file for writing.

With mode `"w"`, if there is no file named `test.txt` on the disk, it will be created. If there already is one, it will be replaced by the file we are writing.

To put data in the file we invoke the `write` method on the handle, shown in lines 2, 3 and 4 above. In bigger programs, lines 2–4 will usually be replaced by a loop that writes many more lines into the file.

Closing the file handle (line 5) tells the system that we are done writing and makes the disk file available for reading by other programs (or by our own program).

A handle is somewhat like a TV remote control

We're all familiar with a remote control for a TV. We perform operations on the remote control — switch channels, change the volume, etc. But the real action happens on the TV. So, by simple analogy, we'd call the remote control our *handle* to the underlying TV.

Sometimes we want to emphasize the difference — the file handle is not the same as the file, and the remote control is not the same as the TV. But at other times we prefer to treat them as a single mental chunk, or abstraction, and we'll just say "close the file", or "flip the TV channel".

13.3. Reading a file line-at-a-time

Now that the file exists on our disk, we can open it, this time for reading, and read all the lines in the file, one at a time. This time, the mode argument is `"r"` for reading:

```
1  mynewhandle = open("test.txt", "r")
2  while True:                          # Keep reading forever
3      theline = mynewhandle.readline()  # Try to read next line
4      if len(theline) == 0:             # If there are no more lines
5          break                         #     leave the loop
6
7      # Now process the line we've just read
8      print(theline, end="")
9
10 mynewhandle.close()
```

This is a handy pattern for our toolbox. In bigger programs, we'd squeeze more extensive logic into the body of the loop at line 8 — for example, if each line of the file contained the name and email address of one of our friends, perhaps we'd split the line into some pieces and call a function to send the friend a party invitation.

On line 8 we suppress the newline character that `print` usually appends to our strings. Why? This is because the string already has its own newline: the `readline` method in line 3 returns everything up to *and including* the newline character. This also explains the end-of-file detection logic: when there are no more lines to be read from the file, `readline` returns an empty string — one that does not even have a newline at the end, hence its length is 0.

Fail first ...

In our sample case here, we have three lines in the file, yet we enter the loop *four* times. In Python, you only learn that the file has no more lines by failure to read another line. In some other programming languages (e.g. Pascal), things are different: there you read three lines, but you have what is called *look ahead* — after reading the third line you already know that there are no more lines in the file. You're not even allowed to try to read the fourth line.

So the templates for working line-at-a-time in Pascal and Python are subtly different!

When you transfer your Python skills to your next computer language, be sure to ask how you'll know when the file has ended: is the style in the language "try, and after you fail you'll know", or is it "look ahead"?

If we try to open a file that doesn't exist, we get an error:

```
>>> mynewhandle = open("wharrah.txt", "r")
IOError: [Errno 2] No such file or directory: "wharrah.txt"
```

13.4. Turning a file into a list of lines

It is often useful to fetch data from a disk file and turn it into a list of lines. Suppose we have a file containing our friends and their email addresses, one per line in the file. But we'd like the lines sorted into alphabetical order. A good plan is to read everything into a list of lines, then sort the list, and then write the sorted list back to another file:

```
1  f = open("friends.txt", "r")
2  xs = f.readlines()
3  f.close()
4
5  xs.sort()
6
7  g = open("sortedfriends.txt", "w")
8  for v in xs:
9      g.write(v)
10 g.close()
```

The `readlines` method in line 2 reads all the lines and returns a list of the strings.

We could have used the template from the previous section to read each line one-at-a-time, and to build up the list ourselves, but it is a lot easier to use the method that the Python implementors gave us!

13.5. Reading the whole file at once

Another way of working with text files is to read the complete contents of the file into a string, and then to use our string-processing skills to work with the contents.

We'd normally use this method of processing files if we were not interested in the line structure of the file. For example, we've seen the `split` method on strings which can break a string into words. So here is how we might count the number of words in a file:

```
1 f = open("somefile.txt")
2 content = f.read()
3 f.close()
4
5 words = content.split()
6 print("There are {0} words in the file.".format(len(words)))
```

Notice here that we left out the `"r"` mode in line 1. By default, if we don't supply the mode, Python opens the file for reading.

Your file paths may need to be explicitly named.

In the above example, we're assuming that the file `somefile.txt` is in the same directory as your Python source code. If this is not the case, you may need to provide a full or a relative path to the file. On Windows, a full path could look like `"C:\\temp\\somefile.txt"`, while on a Unix system the full path could be `"/home/jimmy/somefile.txt"`.

We'll return to this later in this chapter.

13.6. Working with binary files

Files that hold photographs, videos, zip files, executable programs, etc. are called **binary** files: they're not organized into lines, and cannot be opened with a normal text editor. Python works just as easily with binary files, but when we read from the file we're going to get bytes back rather than a string. Here we'll copy one binary file to another:

```
1  f = open("somefile.zip", "rb")
2  g = open("thecopy.zip", "wb")
3
4  while True:
5      buf = f.read(1024)
6      if len(buf) == 0:
7          break
8      g.write(buf)
9
10 f.close()
11 g.close()
```

There are a few new things here. In lines 1 and 2 we added a `"b"` to the mode to tell Python that the files are binary rather than text files. In line 5, we see `read` can take an argument which tells it how many bytes to attempt to read from the file. Here we chose to read and write up to 1024 bytes on each iteration of the loop. When we get back an empty buffer from our attempt to read, we know we can break out of the loop and close both the files.

If we set a breakpoint at line 6, (or print `type(buf)` there) we'll see that the type of `buf` is `bytes`. We don't do any detailed work with `bytes` objects in this textbook.

13.7. An example

Many useful line-processing programs will read a text file line-at-a-time and do some minor processing as they write the lines to an output file. They might number the lines in the output file, or insert extra

blank lines after every 60 lines to make it convenient for printing on sheets of paper, or extract some specific columns only from each line in the source file, or only print lines that contain a specific substring. We call this kind of program a **filter**.

Here is a filter that copies one file to another, omitting any lines that begin with `#`:

```
1  def filter(oldfile, newfile):
2      infile = open(oldfile, "r")
3      outfile = open(newfile, "w")
4      while True:
5          text = infile.readline()
6          if len(text) == 0:
7              break
8          if text[0] == "#":
9              continue
10
11         # Put any more processing logic here
12         outfile.write(text)
13
14     infile.close()
15     outfile.close()
```

The `continue` statement at line 9 skips over the remaining lines in the current iteration of the loop, but the loop will still iterate. This style looks a bit contrived here, but it is often useful to say *"get the lines we're not concerned with out of the way early, so that we have cleaner more focused logic in the meaty part of the loop that might be written around line 11."*

Thus, if `text` is the empty string, the loop exits. If the first character of `text` is a hash mark, the flow of execution goes to the top of the loop, ready to start processing the next line. Only if both conditions fail do we fall through to do the processing at line 11, in this example, writing the line into the new file.

Let's consider one more case: suppose our original file contained empty lines. At line 6 above, would this program find the first empty line in the file, and terminate immediately? No! Recall that `readline` always includes the newline character in the string it returns. It is only when we try to read *beyond* the end of the file that we get back the empty string of length 0.

13.8. Directories

Files on non-volatile storage media are organized by a set of rules known as a **file system**. File systems are made up of files and **directories**, which are containers for both files and other directories.

When we create a new file by opening it and writing, the new file goes in the current directory (wherever we were when we ran the program). Similarly, when we open a file for reading, Python looks for it in the current directory.

If we want to open a file somewhere else, we have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

```
>>> wordsfile = open("/usr/share/dict/words", "r")
>>> wordlist = wordsfile.readlines()
>>> print(wordlist[:6])
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

This (Unix) example opens a file named `words` that resides in a directory named `dict`, which resides in `share`, which resides in `usr`, which resides in the top-level directory of the system, called `/`. It then reads in each line into a list using `readlines`, and prints out the first 5 elements from that list.

A Windows path might be `"c:/temp/words.txt"` or `"c:\\temp\\words.txt"`. Because backslashes are used to escape things like newlines and tabs, we need to write two backslashes in a literal string to get one! So the length of these two strings is the same!

We cannot use `/` or `\` as part of a filename; they are reserved as a **delimiter** between directory and filenames.

The file `/usr/share/dict/words` should exist on Unix-based systems, and contains a list of words in alphabetical order.

13.9. What about fetching something from the web?

The Python libraries are pretty messy in places. But here is a very simple example that copies the contents at some web URL to a local file.

```
import urllib.request

url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
destination_filename = "rfc793.txt"

urllib.request.urlretrieve(url, destination_filename)
```

The `urlretrieve` function — just one call — could be used to download any kind of content from the Internet.

We'll need to get a few things right before this works:

- The resource we're trying to fetch must exist! Check this using a browser.
- We'll need permission to write to the destination filename, and the file will be created in the "current directory" - i.e. the same folder that the Python program is saved in.
- If we are behind a proxy server that requires authentication, (as some students are), this may require some more special handling to work around our proxy. Use a local resource for the purpose of this demonstration!

Here is a slightly different example. Rather than save the web resource to our local disk, we read it directly into a string, and return it:

```
import urllib.request

def retrieve_page(url):
    """ Retrieve the contents of a web page.
        The contents is converted to a string before returning it.
    """
    my_socket = urllib.request.urlopen(url)
    dta = str(my_socket.readall())
    my_socket.close()
    return dta

the_text = retrieve_page("http://xml.resource.org/public/rfc/txt/rfc793.txt")
print(the_text)
```

Opening the remote url returns what we call a **socket**. This is a handle to our end of the connection between our program and the remote web server. We can call read, write, and close methods on the socket object in much the same way as we can work with a file handle.

13.10. Glossary

delimiter
A sequence of one or more characters used to specify the boundary between separate parts of text.

directory
A named collection of files, also called a folder. Directories can contain files and other directories, which are referred to as *subdirectories* of the directory that contains them.

file
A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

file system
A method for naming, accessing, and organizing files and the data they contain.

handle
An object in our program that is connected to an underlying resource (e.g. a file). The file handle lets our program manipulate/read/write/close the actual file that is on our disk.

mode

A distinct method of operation within a computer program. Files in Python can be opened in one of four modes: read (`"r"`), write (`"w"`), append (`"a"`), and read and write (`"+"`).

non-volatile memory

Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

path

A sequence of directory names that specifies the exact location of a file.

text file

A file that contains printable characters organized into lines separated by newline characters.

socket

One end of a connection allowing one to read and write information to or from another computer.

volatile memory

Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

13.11. Exercises

1. Write a program that reads a file and writes out a new file with the lines in reversed order (i.e. the first line in the old file becomes the last one in the new file.)
2. Write a program that reads a file and prints only those lines that contain the substring `snake`.
3. Write a program that reads a text file and produces an output file which is a copy of the file, except the first five columns of each line contain a four digit line number, followed by a space. Start numbering the first line in the output file at 1. Ensure that every line number is formatted to the same width in the output file. Use one of your Python programs as test data for this exercise: your output should be a printed and numbered listing of the Python program.
4. Write a program that undoes the numbering of the previous exercise: it should read a file with numbered lines and produce another file without line numbers.