Chapter 10

Reading and Writing Files

Up to this point we have entered data into our programs either literally (i.e., hardwired into the code when the code is written) or obtained it from the user via an `input()` statement. However, we often need to work with data stored in a file. Furthermore, rather than displaying results on the screen, sometimes the desired action is to store the results in a file. In this chapter we explore the reading and writing of files. In general, the content of a file is in the form of (readable) text or in the form of binary or "raw" data. In this chapter we will only consider the reading and writing of text, i.e., files that consist of characters from the ASCII character set.

10.1 Reading a File

To open a file for reading we use the built-in function `open()`. The first argument is the name of the file and the second argument is the *mode*. When we want to read from an existing file, the mode is set to the character `'r'` (as opposed to the character `'w'` which indicates we want to write to a new file).[1] For the file to be opened successfully, it must be in one of the directories where Python searches, i.e., the file must be somewhere in Python's *path*. Controlling Python's path for the reading and writing of files is no different than controlling the path for importing modules. Thus, the discussion in Sec. 8.6 is directly relevant to the material in this chapter. As a reminder, perhaps the simplest way to ensure Python will find an existing file is to set the *current working directory* to the directory where the file resides. For example, assume there is a file called `info.txt`[2] in the `Documents` directory of a Macintosh or Linux machine or in the `My Documents` folder of a Windows machine.[3] The statements shown in Listing 10.1 are appropriate for opening this file for the user `guido`. The first two lines of both sets of instructions serve to set the current working directory to the desired location. Even if more than one file in this directory is opened, the first two statements are issued just once (however, there must be one `open()` statement for each file).

From the file: `files.tex`

[1] Actually `open()` can be used with a single argument—the file name—in which case it is understood that the file should be opened for reading, i.e., `'r'` is the default mode.

[2] There is no restriction on a file name. Often files will have an extension of `.txt` or `.dat`, but this is not necessary.

[3] We use the terms *folder* and *directory* interchangeably.

Listing 10.1 Demonstration of setting the current working directory and opening a file within this directory. It is assumed the user's account name is `guido` and the file system has been configured in a "typical" manner.

The following are appropriate for a Macintosh or Linux machine:

```
import os     # Import the "operating system" module os.
os.chdir("/Users/guido/Documents")
file = open("info.txt", "r")
```

Analogous statements on a Windows machine are:

```
import os     # Import the "operating system" module os.
os.chdir("C:/Users/guido/My Documents")
file = open("info.txt", "r")
```

The `open()` function returns a *file object*.[4] We will typically assign the file object to an identifier although, as will be shown, this isn't strictly necessary to access the contents of a file.

A list of a file object's methods is obtained by giving the object as the argument to `dir()`. This is done in Listing 10.2. In the subsequent output we observe the methods `read()`, `readline()`, and `readlines()`. Of some interest is the fact that, even though we have opened the file for reading, the object has methods `write()` and `writelines()`. We will consider the write-related methods in Sec. 10.2.

Listing 10.2 Methods for a file object. The methods discussed in this chapter are shown in slanted bold type.

```
>>> file = open("info.txt", "r")
>>> dir(file)
['_CHUNK_SIZE', '__class__', '__delattr__', '__doc__', '__enter__',
  '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__',
  '__getstate__', '__gt__', '__hash__', '__init__', '__iter__',
  '__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__',
  '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
  '__str__', '__subclasshook__', '_checkClosed', '_checkReadable',
  '_checkSeekable', '_checkWritable', 'buffer', 'close', 'closed',
  'detach', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
  'line_buffering', 'name', 'newlines', 'read', 'readable',
  'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate',
  'writable', 'write', 'writelines']
```

Throughout this section we assume the existence of the file `info.txt` located in a directory in Python's path. The file is assumed to contain the following text:

```
This file contains some numbers (and text).
4,   5,   6,    12
```

[4] More technically, the object is a *stream* or a *file-like* object. However, if you issue the command `help(open)` the documentation states that `open()` returns a file object. On the other hand, if you use `type()` to check the type of one of these file objects, you get something that looks rather mysterious.

```
12.3 37.2 -15.7

This is the last line.
```

After opening a file, the contents can be obtained by invoking the `read()`, `readline()`, or `readlines()` methods (or a combination of these methods). `read()` and `readline()` both return a string. `readlines()` returns a list. Alternatively, we can also simply use the file object directly as the iterable in a `for`-loop. In the following sections we consider each of these ways of reading a file.

10.1.1 `read()`, `close()`, and `tell()`

When one uses the `read()` method, the entire file is read and its contents are returned as a single string. This is demonstrated in Listing 10.3. In line 1 the file is opened. In line 2 the `read()` method is called and the result is stored to the identifier `all`. Line 3 serves to echo the contents of `all` which we see in lines 4 and 5.[5] The output in lines 4 and 5 is not formatted, e.g., newlines are indicated by `\n`. To obtain output that mirrors the contents of the file in the way it is typically displayed, we can simply print this string as is done in line 6. Note that the `print()` statement has the optional argument `end` set to the empty string. This is done because the string `all` already contains all the newline characters that were contained in the file itself. Thus, because `all` ends with a newline character, if we do not set `end` to the empty string, there will be an additional blank line at the end of the output.

Listing 10.3 Use of the `read()` method to read an entire file as one string.

```
>>> file = open("info.txt", "r")   # Open file for reading.
>>> all = file.read()              # Read the entire file.
>>> all                            # Show the resulting string.
'This file contains some numbers (and text).\n4,   5,   6,    12\n
12.3 37.2 -15.7\n\nThis is the last line.\n'
>>> print(all, end="")             # Print the contents of the file.
This file contains some numbers (and text).
4,   5,   6,    12
12.3 37.2 -15.7

This is the last line.
```

After reading the contents of a file with `read()`, we can call `read()` again. *However,* rather than obtaining the entire contents of the file, we get an empty string. Python maintains a *stream position* that indicates where it is in terms of reading a file, i.e., the index of the next character to be read. When a file is first opened, this position is at the very start of the file. After invoking the `read()` method the position is at the end of the file, i.e., there are no more characters to read. If you invoke the `read()` method with the stream position at the end of the file, you merely obtain

[5] An explicit line break has been added for display purposes—Python tries to display the string on a single line but the output is "wrapped" to the following line at the edge of the window.

an empty string—`read()` does not automatically return to the start of the file. If you must read a file multiple times, you can *close* the file, using the `close()` method, and then open it again using `open()`. In fact, there is no need to `close()` the file before issuing the second `open()` because if a file is already open, Python will close it before opening it again. However, it is good practice to close files when you are done with them. When you `close()` a file that has been opened for reading, you free some resources and you ensure that accidental reads of a file do not occur. Listing 10.4 demonstrates reading from a file multiple times.

Listing 10.4 If the `read()` method is invoked more than once on a file, the method returns an empty string for all but the first invocation. To reposition the stream position back to the start of the file the simplest approach is to close the file and reopen it. It is an error to try to read from a closed file.

```
1  >>> file = open("info.txt", "r")   # Open file for reading.
2  >>> all = file.read()              # Read the entire file.
3  >>> everything = file.read()       # Attempt to read entire file again.
4  >>> everything                     # There is nothing in everything.
5  ''
6  >>> file.close()                   # Close file.
7  >>> everything = file.read()       # Attempt to read a closed file.
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 ValueError: I/O operation on closed file.
11 >>> file = open("info.txt", "r")   # Open file for reading (again).
12 >>> everything = file.read()       # Now we obtain everything.
13 >>> everything
14 'This file contains some numbers (and text).\n4,   5,   6,    12\n
15 12.3 37.2 -15.7\n\nThis is the last line.\n'
```

The first two lines of Listing 10.4 are the same as those of Listing 10.3, i.e., the file is opened and the entire file is read. In line 3 the `read()` method is again called. This time the return value is assigned to the identifier `everything`. Lines 4 and 5 show that, rather than containing a copy of the entire file, `everything` is the empty string. In line 6 the `close()` method is used to close the file. In line 7 we again attempt to use `read()` to read the file. However, it is an error to read from a closed file and hence a `ValueError` exception is raised as shown in lines 8 through 10. In line 11 the file is opened again. In line 12 the `read()` method is used to read the entire file. As shown in lines 14 through 15, `everything` now corresponds to the contents of the entire file.

Although it is generally something we ignore, it can be instructive to observe the stream position, i.e., the value that indicates the index of the next character to be read. As mentioned, when a file is first opened, the stream position is zero, i.e., the next character to be read is the first one in the file. Or, thought of another way, this zero indicates that we are offset zero from the start of the file. We can obtain the value of the stream position with the `tell()` method. This is demonstrated in Listing 10.5. The code is discussed following the listing.

Listing 10.5 The `tell()` method returns the current stream position, i.e., the index of the next character to be read.

```
1  >>> file = open("info.txt", "r")   # Open file for reading.
2  >>> file.tell()                    # Stream position at start of file.
3  0
4  >>> all = file.read()              # Read entire file.
5  >>> file.tell()                    # Stream position at end of file.
6  103
7  >>> # The following shows that, after reading the entire file, the
8  >>> # length of all the characters in the file and the value of the
9  >>> # stream position are the same.
10 >>> len(all)
11 103
```

In line 1 the file is opened for reading. In lines 2 and 3 `tell()` reports the stream position is `0`, i.e., we are at the start of the file. In line 4 the entire file is read using `read()` and stored as the string `all`. In lines 5 and 6 we see the stream position is now `103`. Checking the length of `all` in lines 10 and 11 we see that it is the same as the current stream position. (Recall that the last valid index of a sequence is one less than its length, i.e., the last valid index for `all` is `102`.)

10.1.2 `readline()`

Unlike `read()`, which reads the entire contents of a file in one shot, the `readline()` method reads a single line and returns this line as a string. This is illustrated in Listing 10.6 where the file `info.txt` is read one line at a time. To illustrate how Python keeps track of where it is within the file, the `tell()` method is also called to show the stream position. Typically there is no reason to display this information—it is sufficient that Python is keeping track of it. The code is discussed following the listing.

Listing 10.6 Demonstration of the use of `readline()` to read the contents of a file one line at a time. The `tell()` method is used merely to show how the stream position advances with each invocation of `readline()`.

```
1  >>> file = open("info.txt", "r")
2  >>> file.readline()       # Read one line.
3  'This file contains some numbers (and text).\n'
4  >>> file.tell()           # Check on stream position.
5  44
6  >>> file.readline()       # Read second line.
7  '4,   5,   6,    12\n'
8  >>> file.tell()
9  63
10 >>> file.readline()       # Read third line.
11 '12.3 37.2 -15.7\n'
12 >>> file.tell()
```

```
79
>>> file.readline()       # Fourth line is "blank" but not empty.
'\n'
>>> file.tell()
80
>>> file.readline()       # Read fifth line.
'This is the last line.\n'
>>> file.tell()
103
>>> # It is not an error to call readline() when at the end of the file.
>>> # Instead, readline merely returns an empty string.
>>> file.readline()
''
```

In line 1 the file is opened for reading. In line 2 the `readline()` method is invoked. This returns, as a string, the first line of the file. Note that the newline character is part of this string (i.e., the last character of the string). In line 4 the `tell()` method is called to check the current value of the stream position. Line 5 reports it is `44`. This corresponds to the number of characters in the first line of the file (the newline character counts as a single character). The next several lines repeat the process of calling `readline()` and then showing the stream position.

In line 15 of Listing 10.6, note the result of reading the fourth line of `info.txt` (i.e., note the result of the statement in line 14). The fourth line of `info.txt` is a "blank" line but, in fact, that does not mean it is empty. Within the file this line consists solely of the newline character. We see this in the output on line 15.

In line 24, we attempt to read a sixth line. Although there isn't a sixth line, this does not produce an error. Instead, the return value is the empty string. Thus, we know (and Python knows) that the end of the file has been reached when `readline()` returns an empty string.

Obviously it can be awkward if one has to explicitly call the `readline()` method for each individual line of a long file. Fortunately there are a couple other constructs that facilitate reading the entire file on a line-by-line basis.

10.1.3 `readlines()`

An alternative approach for conveniently reading the entire contents of a file is offered by the `readlines()` method. The `readlines()` method is somewhat like `read()` in that it reads the entire file.[6] However, rather than returning a single string, `readlines()` returns a `list` in which each element of the `list` is a line from the file. This is demonstrated in Listing 10.7 which is discussed following the listing.

Listing 10.7 Demonstration of the use of the `readlines()` method .

```
>>> file = open("info.txt", "r")
>>> lines = file.readlines()       # Read all lines into a list.
```

[6] Actually these methods read from the current stream position to the end of the file as discussed in more detail later.

```
>>> lines                                # Display list.
['This file contains some numbers (and text).\n', '4,   5,   6,    12\n',
 '12.3 37.2 -15.7\n', '\n', 'This is the last line.\n']
>>> count = 0
>>> for line in lines:     # Show lines together with a count.
...     count = count + 1
...     print(count, ": ", line, sep="", end="")
...
1: This file contains some numbers (and text).
2: 4,   5,   6,    12
3: 12.3 37.2 -15.7
4:
5: This is the last line.
```

In line 1 the file is opened for reading. In line 2 the `readlines()` method is invoked. This returns a `list`, which is assigned to the identifier `lines`, in which each element corresponds to a line of the file. In this particular case the list `lines` contains all the lines of the file. The entire file is stored within the `list` because the stream position was `0` when the method was invoked, i.e., `readlines()` starts from the current location of the stream position and reads until the end of the file. This type of behavior (with the reading starting from the current stream position and continuing to the end of the file) also pertains to `read()` and is discussed further in Sec. 10.1.5. If the `readlines()` method is invoked when the stream position is at the end of a file, the method returns an empty `list`.

Line 3 echoes `lines`. The output on lines 4 and 5 shows that this `list` does indeed contain all the lines of the file. Note that each string in the `list` (i.e., each line) is terminated by a newline character. The code in lines 6 through 9 displays each line of the file with the line preceded by a line number (starting from one). The output is shown in lines 11 through 15. Note that the `print()` statement in line 9 has the optional argument `end` set to the empty string. If this were not done, each numbered line would be followed by a blank line since one newline is produced by the newline character contained within the string for the line and another newline generated by the `print()` function, thus resulting in a blank line. The optional argument `sep` is also set to the empty string so that there is no space between the count at the start of the line and the colon.

10.1.4 File Object Used as an Iterable

Rather than using `readlines()` to read the lines of file into a `list` and *then* using a `for`-loop to cycle through the elements of this `list`, one can use the file object itself as the iterable in the `for`-loop header. In this case the file will be read line-by-line, i.e., for each iteration of the `for`-loop the loop variable will be assigned a string corresponding to a line of the file. Lines are read, in order, from the current stream position until the end of the file.

In some applications input files are huge. It thus requires significant memory to read and store the entirety of these files. However, by using the file object as the `for`-loop iterable, only one line is read (and processed) at a time. In this way it is possible to process files that are tremendously large without using much memory.

The use of a file object as the iterable of a `for`-loop is illustrated in Listing 10.8.

Listing 10.8 Demonstration that a file object can be used as the iterable in the header of a `for`-loop. In this case the loop variable takes on the value of each line of the file.

```
>>> file = open("info.txt", "r")
>>> count = 0
>>> for line in file:
...     count = count + 1
...     print(count, ": ", line, sep="", end="")
...
1: This file contains some numbers (and text).
2: 4,   5,   6,    12
3: 12.3 37.2 -15.7
4:
5: This is the last line.
```

In this code the integer variable `count` is used to keep track of the line number. The `print()` statement in the `for`-loop prints both the line number and the corresponding line from the file. `end` is set to the empty string to suppress the newline generated by the `print()` function. Alternatively, rather than setting `end` to the empty string, to avoid repeating the newline character, we can strip the newline character from the `line` variable using the `rstrip()` method.

As a slight twist to the implementation in Listing 10.8, one may use the open statement directly as the iterable in the `for`-loop header. Thus, an alternate implementation of Listing 10.8 is given in Listing 10.9. Here, however, we made the further modification of enclosing the `open()` function in the `enumerate()` function. This allows us, in line 1 of Listing 10.9, to use simultaneous assignment to obtain both a "count" and a line from the file. However, the count starts from zero. Because we want the line number to start from one, we add `1` to `count` in line 2 in the first argument of the `print()` function.

Listing 10.9 An alternate implementation of 10.8 where the `open()` function is used as the iterable of the `for`-loop. In this implementation the `enumerate()` function is used to obtain the "count" directly, although we have to add one to this value to obtain the desired line number.

```
>>> for count, line in enumerate(open("info.txt", "r")):
...     print(count + 1, ": ", line, sep="", end="")
...
1: This file contains some numbers (and text).
2: 4,   5,   6,    12
3: 12.3 37.2 -15.7
4:
5: This is the last line.
```

Both Listing 10.8 and Listing 10.9 accomplish the same thing but five lines of code were required in Listing 10.8 whereas only two lines were needed in Listing 10.9. Reducing the number of lines of code is, in itself, neither a good thing nor a bad thing. Code should strive for both

efficiency and readability. In the case of Listings 10.8 and 10.9, efficiency is not an issue and the "readability" will partly be a function of the experience of the programmer. Listing 10.9 is rather "dense" and hence may be difficult for beginning Python programmers to understand. On the other hand, experienced Python programmers may prefer this density to the comparatively "verbose" code in Listing 10.8. Recognizing that readability is partially in the eye of the reader, we try to use common Python idioms in the listings, but often tend toward a verbose implementation when it is easier to understand than a more terse or dense implementation.

10.1.5 Using More than One Read Method

As mentioned in Sec. 10.1.3, when the `read()` or `readlines()` methods are called, the reading starts from the current stream position. To help illustrate this, consider a situation in which the first two lines of a file are comment lines or provide header information or simply need to be handled differently than the rest of the file. In this case these two lines could be read using the `readline()` method and then the remainder of the file could be conveniently read using either `readlines()` or `read()`.

The "mixing" of read methods is demonstrated in Listing 10.10. The file is opened in line 1. The first and second lines are read in lines 2 and 3, respectively. Then, in line 4, the `readlines()` method is invoked as the iterable of the `for`-loop. The strings returned by `readlines()` are assigned to the loop variable `line` and printed using the `print()` statement in line 5. The fact that there are only three lines of output (shown in lines 7 through 9) shows that `readlines()` starts reading the file from its third line.

Listing 10.10 The first two lines of a file are read using `readline()` and the remainder of the file is read using `readlines()`.

```
1 >>> file = open("info.txt", "r")
2 >>> line_one = file.readline()
3 >>> line_two = file.readline()
4 >>> for line in file.readlines():
5 ...     print(line, end="")
6 ...
7 12.3 37.2 -15.7
8
9 This is the last line.
```

10.2 Writing to a File

The converse of reading from a file is writing to a file. To do this, we again open the file with the `open()` function. As with opening a file for reading, the first argument is the file name. Now, however, the second argument (the mode) must be `'w'`. **Caution:** *When you open a file for writing, you destroy any previous file that existed with this file name in the current working directory.* Because `open()` can destroy existing files, any time you plan to open a file for writing

or reading, make doubly sure you have set the mode and the file name correctly. After opening the file we can use the `write()` or `writelines()` methods to write to it. Additionally, there is an optional argument for the `print()` function that allows us to specify a file to which the output can be written (i.e., rather than the output appearing on the screen, it is written to the file).

10.2.1 `write()` and `print()`

The `write()` method takes a single argument which is the string to be written to the file. The `write()` method is much less flexible than the `print()` function. With the `print()` function we can use any number of arguments and the arguments can be of any type. Furthermore, using the optional `sep` and `end` parameters, we can specify the separator to appear between objects and how the line should be terminated. In contrast, the `write()` method only accepts a single *string* as its argument (or, of course, any expression that returns a string). However, this is not quite as restrictive as it might first appear. Using string formatting, we can easily represent many objects as a single string. As an example of such a construction, first consider the code in Listing 10.11 which uses the `print()` function to display three variables.

Listing 10.11 Use of the `print()` function to display three values.

```
1 >>> a = 1.0
2 >>> b = "this"
3 >>> c = [12, -4]
4 >>> print(a, b, c)
5 1.0 this [12, -4]
```

The code in Listing 10.12 produces the same output as Listing 10.11, but in this case the output goes to the file `foo.dat`. In line 1 the file is opened and assigned to the identifier `file_out`. In line 5 the `write()` method is called with a single string argument. However, this string is the one produced by the `format()` method acting on the given format string which contains three replacement fields (see the discussion of replacement fields in Sec. 9.7). The resulting string is the same as representation of the three objects displayed in line 5 of Listing 10.11, i.e., after this code has been run, the file `foo.dat` will contain the same output as shown in line 5 of Listing 10.11. Line 7 of Listing 10.12 invokes the `close()` method on `file_out`. If you look at the contents of the file `foo.dat` before calling the `close()` method, you will probably *not* see the output generated by the `write()` method. The reason is that, in the interest of speed and efficiency, the output is *buffered*. Writing from internal computer memory to external memory (such as a disk drive) can be slow. The fewer times such an operation occurs, the better. When output is buffered, it is stored internally and not written to the file until there are "many" characters to write or until the file is closed.[7]

Listing 10.12 Use of the `write()` method to write a single string. Use of a format string and the `format()` method gives the same representation of the three objects as produced by the `print()` statement in Listing 10.11.

[7] Alternatively, there is a `flush()` method that forces the "flushing" of output from the buffer to the output file.

```
1 >>> file_out = open("foo.dat", "w")
2 >>> a = 1.0
3 >>> b = "this"
4 >>> c = [12, -4]
5 >>> file_out.write("{} {} {}\n".format(a, b, c))
6 18
7 >>> file_out.close()
```

The `18` that appears in line 6 is the return value of the `write()` method and corresponds to the number of characters written to the file. This number is shown in the interactive environment, but if the statement on line 5 appeared within a program (i.e., within a `.py` file), the return value would not be visible. A programmer is not obligated to use a return value and when a return value is not assigned to an identifier or printed, it simply "disappears."

Now, having discussed the `write()` method, we actually *can* use the `print()` function to write to a file! By adding an optional `file=<file_object>` argument to `print()`'s argument list, the output will be written to the file rather than to the screen.[8] Thus, the code in Listing 10.13 is completely equivalent to the code in Listing 10.12.

Listing 10.13 The `print()` function's optional `file` argument is used to produce the same result as in Listing 10.12. This code and the code of Listing 10.12 differ only in line 5.

```
1 >>> file_out = open("foo.dat", "w")
2 >>> a = 1.0
3 >>> b = "this"
4 >>> c = [12, -4]
5 >>> print(a, b, c, file=file_out)
6 >>> file_out.close()
```

As we are already familiar with the `print()` function, we will not consider it further in the remainder of this chapter.

As an example of the use of the `write()` method, let's read from one file and write its contents to another. Specifically, let's copy the contents of the file `info.txt` to a new file called `info_lined.txt` and insert the line number at the start of each line. This can be accomplished with the code shown in Listing 10.14. Lines 1 and 2 open the files. In line 3 the variable `count` is initialized to `0`. The `for`-loop in lines 4 through 6 cycles through each line of the input file. Pay close attention to the argument of the `write()` method in line 6. The first replacement field in the format string specifies that two spaces should be allocated to the integer corresponding to the first argument of the `format()` method. The second replacement field merely serves as a placeholder for the second argument of the `format()` method, i.e., the string corresponding to a line of the input file. Note that this format string does not end with a newline character and the write method does not add a newline character. *However*, the line read from the input file does end with a newline character. Thus, one should not add another newline. The numerical values shown in lines 8 through 12 are the numbers of characters written on each line. (We can prevent these

[8] Or, more technically, the default output is directed to `sys.stdout` which is usually the screen.

values from being displayed simply by assigning the return value of the `write()` method to a variable. However, when this code is run from a `.py` file, these values are not displayed unless we explicitly do something to display them.)

Listing 10.14 Code to copy the contents of the file `info.txt` to the (new) file `info_lined.txt`.

```
>>> file_in = open("info.txt", "r")
>>> file_out = open("info_lined.txt", "w")
>>> count = 0
>>> for line in file_in:
...     count = count + 1
...     file_out.write("{:2d}: {}".format(count, line))
...
48
23
20
5
27
>>> file_out.close()
```

After running the code in Listing 10.14, the file `info_lined.txt` contains the following:

```
 1: This file contains some numbers (and text).
 2: 4,   5,   6,    12
 3: 12.3 37.2 -15.7
 4:
 5: This is the last line.
```

10.2.2 `writelines()`

The `writelines()` method takes a sequence as an argument, e.g., a tuple or a `list`. Each element of the sequence must be a string. In some sense `writelines()` is misnamed in that it doesn't necessarily write *lines*. Instead, it writes elements of a sequence (but a method name of `writeelementsofsequence()` isn't very appealing). If all the elements of the sequence end with the newline character, then the output will indeed be as if `writelines()` writes lines.

Consider the code shown in Listing 10.15 which creates two files, `out_1.txt` and `out_2.txt`. The `writelines()` method is used to write the `list values_1` to `out_1.txt` and the `tuple values_2` to `out_2.txt`. The significant difference between these two statements is not that one argument is a `tuple` and the other is a `list` (this distinction is of no concern to `writelines()`). Rather, it is that the strings in `values_2` are terminated with newline characters but the strings in `values_1` are not. The discussion continues following the listing.

Listing 10.15 Demonstration of the use of the `writelines()` method.

```
>>> out_1 = open("out_1.txt", "w")
>>> out_2 = open("out_2.txt", "w")
>>> values_1 = ["one", "two", "three"]
>>> values_2 = ("one\n", "two\n", "three\n")
>>> out_1.writelines(values_1)
>>> out_2.writelines(values_2)
>>> out_1.close()
>>> out_2.close()
```

After the code in Listing 10.15 is run, the file `out_1.txt` contains the following:

```
onetwothree
```

This text is not terminated by a newline character. On the other hand the file `out_2.txt` contains:

```
one
two
three
```

This text is terminated by a newline character.

10.3 Chapter Summary

Files are opened for reading or writing using the `open()` function. The first argument is the file name and the second is the mode (`'r'` for read, `'w'` for write). Returns a *file object*.

The `stream position` indicates the next character to be read from a file. When the file is first opened, the stream position is zero.

Contents of a file can be obtained using the following file-object methods:

- **`read()`**: Returns a string corresponding to the contents of a file from the current stream position to the end of the file.
- **`readline()`**: Returns, as a string, a single line from a file.
- **`readlines()`**: Returns a `list` of strings, one for each line of a file, from the current stream position to the end of the file (newline characters are not removed).

If the stream position is at the end of a file, `read()` and `readline()` return empty strings while `readlines()` returns an empty `list`.

A file object can be used as the iterable in a `for`-loop.

The **`close()`** method is used to close a file object. It is an error to read from a closed file.

The **`write()`** method can be used to write a string to a file.

A `print()` statement can also be used to print to a file (i.e., write a string to a file) using the optional `file` argument. A file object must be provided with this argument.

The **`writelines()`** method takes a sequence of strings as its argument and writes them to the given file as a continuous string.

10.4 Review Questions

1. Assume the file `input.txt` is opened successfully in the following. What is the type of the variable `z` after the following code is executed:

```
file = open("input.txt", "r")
z = file.readlines()
```

   (a) `list`

   (b) `str`

   (c) `file object`

   (d) None of the above.

   (e) This code produces an error.

For problems 2 through 9, assume the file `foo.txt` contains the following:

```
This is
a test.
Isn't it?  I think so.
```

2. What output is produced by the following?

```
file = open("foo.txt", "r")
s = file.read()
print(s[2])
```

3. What output is produced by the following?

```
file = open("foo.txt", "r")
file.readline()
print(file.readline(), end="")
```

4. What output is produced by the following?

```
file = open("foo.txt", "r")
for line in file:
    print(line[0], end="")
```

5. What output is produced by the following?

```
file = open("foo.txt", "r")
s = file.read()
xlist = s.split()
print(xlist[1])
```

6. What output is produced by the following?

```
file = open("foo.txt")
s = file.read()
xlist = s.split('\n')
print(xlist[1])
```

7. What output is produced by the following?

```
file = open("foo.txt")
xlist = file.readlines()
ylist = file.readlines()
print(len(xlist), len(ylist))
```

8. What output is produced by the following?

```
file = open("foo.txt")
xlist = file.readlines()
file.seek(0)
ylist = file.readlines()
print(len(ylist), len(ylist))
```

9. What output is produced by the following?

```
file = open("foo.txt")
for x in file.readline():
   print(x, end=":")
```

10. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
print(s, "is a test", file=file)
file.close()
```

11. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
file.write(s, "is a test")
file.close()
```

12. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
file.write(s + "is a test")
file.close()
```

13. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
file.write("{} {}\n".format(s, "is a test"))
file.close()
```

ANSWERS: 1) a; 2) `i`; 3) `a test.`; 4) `TaI`; 5) `is`; 6) `a test.`; 7) `3 0`; 8) `3 3`; 9) `T:h:i:s:  :i:s:`; 10) `this is a test` (terminated with a newline character); 11) The file will be empty since there is an error that prevents the `write()` method from being used—`write()` takes a single string argument and here is is given two arguments.; 12) `thisis a test` (this is *not* terminated with a newline character); 13) `this is a test` (terminated with a newline character).