10.2. Reading a File

As an example, suppose we have a text file called olympics.txt that contains the data representing about olympians across different years. The contents of the file are shown at the bottom of the page.

To open this file, we would call the open function. The variable, fileref, now holds a reference to the file object returned by open. When we are finished with the file, we can close it by using the close method. After the file is closed any further attempts to use fileref will result in an error.

fileref = open("olympics.txt","r")

## other code here that refers to variable fileref

fileref.close()

​

**Note**

A common mistake is to get confused about whether you are providing a variable name or a string literal as an input to the open function. In the code above, “olympics.txt” is a string literal that should correspond to the name of a file on your computer. If you put something without quotes, like open(x, "r"), it will be treated as a variable name. In this example, x should be a variable that’s already been bound to a string value like “olympics.txt”.

10.3. Alternative File Reading Methods

Once you have a file “object”, the thing returned by the open function, Python provides three methods to read data from that object. The read() method returns the entire contents of the file as a single string (or just some characters if you provide a number as an input parameter. The readlines method returns the entire contents of the entire file as a list of strings, where each item in the list is one line of the file. The readline method reads one line from the file and returns it as a string. The strings returned by readlines or readline will contain the newline character at the end. [Table 2](https://fopp.umsi.education/runestone/static/fopp/Files/AlternativeFileReadingMethods.html#filemethods2a) summarizes these methods and the following session shows them in action.

| **Method Name** | **Use** | **Explanation** |
| --- | --- | --- |
| write | filevar.write(astring) | Add astring to the end of the file. filevar must refer to a file that has been opened for writing. |
| read(n) | filevar.read() | Reads and returns a string of n characters, or the entire file as a single string if n is not provided. |
| readline(n) | filevar.readline() | Returns the next line of the file with all text up to and including the newline character. If n is provided as a parameter than only n characters will be returned if the line is longer than n. |
| readlines(n) | filevar.readlines() | Returns a list of strings, each representing a single line of the file. If n is not provided then all lines of the file are returned. If n is provided then n characters are read but n is rounded up so that an entire line is returned. |

In this course, we will generally either iterate through the lines returned by readlines() with a for loop, or use read() to get all of the contents as a single string.

# 10.4. Iterating over lines in a file

We will now use this file as input in a program that will do some data processing. In the program, we will examine each line of the file and print it with some additional text. Because readlines() returns a list of lines of text, we can use the for loop to iterate through each line of the file.

A **line** of a file is defined to be a sequence of characters up to and including a special character called the **newline** character. If you evaluate a string that contains a newline character you will see the character represented as \n. If you print a string that contains a newline you will not see the \n, you will just see its effects (a carriage return).

As the for loop iterates through each line of the file the loop variable will contain the current line of the file as a string of characters. The general pattern for processing each line of a text file is as follows:

**for** line **in** myFile.readlines():

statement1

statement2

...

To process all of our olypmics data, we will use a for loop to iterate over the lines of the file. Using the split method, we can break each line into a list containing all the fields of interest about the athlete. We can then take the values corresponding to name, team and event to construct a simple sentence.

olypmicsfile = open("olypmics.txt","r")

​

for aline in olypmicsfile.readlines():

values = aline.split(",")

print(values[0], "is from", values[3], "and is on the roster for", values[4])

​

olypmicsfile.close()

​

To make the code a little simpler, and to allow for more efficient processing, Python provides a built-in way to iterate through the contents of a file one line at a time, without first reading them all into a list. Some students find this confusing initially, so we don’t recommend doing it this way, until you get a little more comfortable with Python. But this idiom is preferred by Python programmers, so you should be prepared to read it. And when you start dealing with big files, you may notice the efficiency gains of using it.

olypmicsfile = open("olypmics.txt","r")

​

for aline in olypmicsfile:

values = aline.split(",")

print(values[0], "is from", values[3], "and is on the roster for", values[4])

​

olypmicsfile.close()

​

**Check your Understanding**

Sad upset blue down melancholy somber bitter troubled

Angry mad enraged irate irritable wrathful outraged infuriated

Happy cheerful content elated joyous delighted lively glad

Confused disoriented puzzled perplexed dazed befuddled

Excited eager thrilled delighted

Scared afraid fearful panicked terrified petrified startled

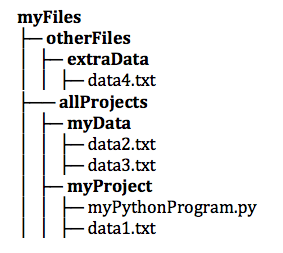
Nervous anxious jittery jumpy tense uneasy apprehensive

1. Write code to find out how many lines are in the file emotion\_words.txt as shown above. Save this value to the variable num\_lines. Do not use the len method.

# 10.5. Finding a File in your Filesystem

In the examples we have provided, and in the simulated file system that we’ve built for this online textbook, all files sit in a single directory, and it’s the same directory where the python program is stored. Thus, we can just write open('myfile.txt','r').

If you have installed python on your local computer and you are trying to get file reading and writing operations to work, there’s a little more that you may need to understand. Computer operating systems (like Windows and the Mac OS) organize files into a hierarchy of folders, with some folders containing other folders.



If your file and your Python program are in the same directory you can simply use the filename. For example, with the file hierarchy in the diagram, the file myPythonProgram.py could contain the code open('data1.txt','r').

If your file and your Python program are in different directories, however, then you need to specify a **path**. You can think of the filename as the short name for a file, and the path as the full name. Typically, you will specify a relative file path, which says where to find the file to open, relative to the directory where the code is running from. For example, the file myPythonProgram.py could contain the code open('../myData/data2.txt','r'). The ../ means to go up one level in the directory structure, to the containing folder (allProjects); myData/ says to descend into the myData subfolder.

There is also an option to use an absolute file path. For example, suppose the file structure in the figure is stored on a computer in the user’s home directory,``/Users/joebob01/myFiles``. Then code in any python program running from any file folder could open data2.txt via open('/Users/joebob01/myFiles/allProjects/myData/data2.txt','r'). You can tell an absolute file path because it begins with a /. If you will ever move your programs and data to another computer (e.g., to share them with someone else), it will be much more convenient if your use relative file paths rather than absolute. That way, if you preserve the folder structure when moving everything, you won’t need to change your code. If you use absolute paths, then the person you are sharing with probably not have the same home directory name, /Users/joebob01/. Note that python pathnames follow the UNIX conventions (Mac OS is a UNIX variant), rather than the Windows file pathnames that use : and ‘’. The python interpreter will translate to Windows pathnames when running on a Windows machine; you should be able to share your python program between a Windows machine and a MAC without having to rewrite the file open commands.

**Note**

For security reasons, our code running in your browser doesn’t read or write files to your computer’s file system. Later, when you run python natively on your own computer, you will be able to truly read files, using path names as suggested above. To get you started, we have faked it by providing a few files that you can read as if they were on your hard disk. In this chapter, we simulate the existence of one textfile; you can’t open any other files from your local computer from textbook code running in your browser.

**Check Your Understanding**

files-2-1: Say you are in a directory called Project. In it, you have a file with your python code. You would like to read in data from a file called “YearlyProjections.csv” which is in a folder called CompanyData, which is inside of Project. What is the best way to open the file in your python program?

Top of Form

A. open("YearlyProjections.csv", "r")  
B. open("../CompanyData/YearlyProjections.csv", "r")  
C. open("CompanyData/YearlyProjections.csv", "r")  
D. open("Project/CompanyData/YearlyProjections.csv", "r")  
E. open("../YearlyProjections.csv", "r")

Bottom of Form

files-2-2: Which of the following paths are relative file paths?

Top of Form

A. "Stacy/Applications/README.txt"  
B. "/Users/Raquel/Documents/graduation\_plans.doc"  
C. "/private/tmp/swtag.txt"  
D. "ScienceData/ProjectFive/experiment\_data.csv"

# 10.6. Using with for Files

**Note**

This section is a bit of an advanced topic and can be easily skipped. But with statements are becoming very common and it doesn’t hurt to know about them in case you run into one in the wild.

Now that you have seen and practiced a bit with opening and closing files, there is another mechanism that Python provides for us that cleans up the often forgotten close. Forgetting to close a file does not necessarily cause a runtime error in the kinds of programs you typically write in an introductory programing course. However if you are writing a program that may run for days or weeks at a time that does a lot of file reading and writing you may run into trouble.

Python has the notion of a context manager that automates the process of doing common operations at the start of some task, as well as automating certain operations at the end of some task. For reading and writing a file, the normal operation is to open the file and assign it to a variable. At the end of working with a file the common operation is to make sure that file is closed.

The Python with statement makes using context managers easy. The general form of a with statement is:

**with** <create some object that understands context> **as** <some name>:

do some stuff **with** the object

...

When the program exits the with block, the context manager handles the common stuff that normally happens at the end, in our case closing a file. A simple example will clear up all of this abstract discussion of contexts. Here are the contents of a file called “mydata.txt”.

*Data file: mydata.txt*

1 2 3

4 5 6

with open('mydata.txt', 'r') as md:

for line in md:

print(line)

# continue on with other code

​

The first line of the with statement opens the file and assigns it to the variable md. Then we can iterate over the file in any of the usual ways. When we are done we simply stop indenting and let Python take care of closing the file and cleaning up. The final line print(md)

This is equivalent to code that specifically closes the file at the end, but neatly marks the set of code that can make use of the open file as an indented block, and ensures that the programmer won’t forget to include the .close() invocation.

md = open('mydata.txt', 'r')

for line in md:

print(line)

md.close()

# continue with other code

# 10.7. Recipe for Reading and Processing a File

Here’s a foolproof recipe for processing the contents of a text file. If you’ve fully digested the previous sections, you’ll understand that there are other options as well. Some of those options are preferable for some situations, and some are preferred by python programmers for efficiency reasons. In this course, though, you can always succeed by following this recipe.

#1. Open the file using with and open.

#2. Use .readlines() to get a list of the lines of text in the file.

#3. Use a for loop to iterate through the strings in the list, each being one line from the file. On each iteration, process that line of text

#4. When you are done extracting data from the file, continue writing your code outside of the indentation. Using with will automatically close the file once the program exits the with block.

fname = "yourfile.txt"

**with** open(fname, 'r') **as** fileref: *# step 1*

lines = fileref.readlines() *# step 2*

**for** line **in** lines: *# step 3*

*#some code that references the variable line*

*#some other code not relying on fileref # step 4*

However, this will not be good to use when you are working with large data. Imagine working with a datafile that has 1000 rows of data. It would take a long time to read in all the data and then if you had to iterate over it, even more time would be necessary. This would be a case where programmers prefer another option for efficiency reasons.

This option involves iterating over the file itself while still iterating over each line in the file:

fname = "yourfile.txt"

**with** open(fname, 'r') **as** fileref: *# step 1*

**for** lin **in** lines: *# step 2*

*## some code that reference the variable lin*

*#some other code not relying on fileref # step 3*

# 10.8. Writing Text Files

One of the most commonly performed data processing tasks is to read data from a file, manipulate it in some way, and then write the resulting data out to a new data file to be used for other purposes later. To accomplish this, the open function discussed above can also be used to create a new file prepared for writing. Note in [Table 1](https://fopp.umsi.education/runestone/static/fopp/Files/intro-WorkingwithDataFiles.html#filemethods1a) that the only difference between opening a file for writing and opening a file for reading is the use of the 'w' flag instead of the 'r' flag as the second parameter. When we open a file for writing, a new, empty file with that name is created and made ready to accept our data. If an existing file has the same name, its contents are overwritten. As before, the function returns a reference to the new file object.

[Table 2](https://fopp.umsi.education/runestone/static/fopp/Files/AlternativeFileReadingMethods.html#filemethods2a) shows one additional method on file objects that we have not used thus far. The write method allows us to add data to a text file. Recall that text files contain sequences of characters. We usually think of these character sequences as being the lines of the file where each line ends with the newline \n character. Be very careful to notice that the write method takes one parameter, a string. When invoked, the characters of the string will be added to the end of the file. This means that it is the programmer’s job to include the newline characters as part of the string if desired.

Assume that we have been asked to provide a file consisting of all the squared numbers from 1 to 12.

First, we will need to open the file. Afterwards, we will iterate through the numbers 1 through 12, and square each one of them. This new number will need to be converted to a string, and then it can be written into the file.

The program below solves part of the problem. We first want to make sure that we’ve written the correct code to calculate the square of each number.

for number in range(1, 13):

square = number \* number

print(square)

​

When we run this program, we see the lines of output on the screen. Once we are satisfied that it is creating the appropriate output, the next step is to add the necessary pieces to produce an output file and write the data lines to it. To start, we need to open a new output file by calling the open function, outfile = open("squared\_numbers.txt",'w'), using the 'w' flag. We can choose any file name we like. If the file does not exist, it will be created. However, if the file does exist, it will be reinitialized as empty and you will lose any previous contents.

Once the file has been created, we just need to call the write method passing the string that we wish to add to the file. In this case, the string is already being printed so we will just change the print into a call to the write method. However, there is an additional step to take, since the write method can only accept a string as input. We’ll need to convert the number to a string. Then, we just need to add one extra character to the string. The newline character needs to be concatenated to the end of the line. The entire line now becomes outfile.write(str(square)+ '\n'). The print statement automatically outputs a newline character after whatever text it outputs, but the write method does not do that automatically. We also need to close the file when we are done.

The complete program is shown below.

**Note**

As with file reading, for security reasons the runestone interactive textbook environment does not write files to the file system on your local computer. In an activecode window, we simulate writing to a file. The contents of the written file are shown and you can do a subsequent read of the contents of that filename. If you try to overwrite a file that’s built in to the page, it may not let you; don’t try to get too fancy with our file system simulator!

Below, we have printed the first 10 characters to the output window.

filename = "squared\_numbers.txt"

outfile = open(filename, "w")

​

for number in range(1, 13):

square = number \* number

outfile.write(str(square) + "\n")

​

outfile.close()

​

infile = open(filename, "r")

print(infile.read()[:10])

# 10.9. CSV Format

CSV stands for Comma Separated Values. If you print out tabular data in CSV format, it can be easily imported into other programs like Excel, Google spreadsheets, or a statistics package (R, stata, SPSS, etc.).

For example, we can make a file with the following contents. If you save it as a file name grades.csv, then you could import it into one of those programs. The first line gives the column names and the later lines each give the data for one row.

Name,score,grade

Jamal,98,A+

Eloise,87,B+

Madeline,99,A+

Wei,94,A

# 10.10. Reading in data from a CSV File

We are able to read in CSV files the same way we have with other text files. Because of the standardized structure of the data, there is a common pattern for processing it. To practice this, we will be using data about olympic events.

Typically, CSV files will have a header as the first line, which contains column names. Then, each following row in the file will contain data that corresponds to the appropriate columns.

All file methods that we have mentioned - read, readline, and readlines, and simply iterating over the file object itself - will work on CSV files. In our examples, we will iterate over the lines. Because the values on each line are separated with commas, we can use the .split() method to parse each line into a collection of separate value.

fileconnection = open("olympics.txt", 'r')

lines = fileconnection.readlines()

header = lines[0]

field\_names = header.strip().split(',')

print(field\_names)

for row in lines[1:]:

vals = row.strip().split(',')

if vals[5] != "NA":

print("{}: {}; {}".format(

vals[0],

vals[4],

vals[5]))

# 10.11. Writing data to a CSV File

The typical pattern for writing data to a CSV file will be to write a header row and loop through the items in a list, outputting one row for each. Here we a have a list of tuples, each representing one Olympian, a subset of the rows and columns from the file we have been reading from.

olympians = [("John Aalberg", 31, "Cross Country Skiing"),

("Minna Maarit Aalto", 30, "Sailing"),

("Win Valdemar Aaltonen", 54, "Art Competitions"),

("Wakako Abe", 18, "Cycling")]

​

outfile = open("reduced\_olympics.csv","w")

# output the header row

outfile.write('Name,Age,Sport')

outfile.write('\n')

# output each of the rows:

for olympian in olympians:

row\_string = '{},{},{}'.format(olympian[0], olympian[1], olympian[2])

outfile.write(row\_string)

outfile.write('\n')

outfile.close()

​

There are a few things worth noting in the code above.

First, using .format() makes it really clear what we’re doing when we create the variable row\_string. We are making a comma separated set of values; the {} curly braces indicated where to substitute in the actual values. The equivalent string concatenation would be very hard to read. An alternative, also clear way to do it would be with the .join method: row\_string = ','.join(olympian[0], olympian[1], olympian[2]).

Second, unlike the print statement, remember that the .write() method on a file object does not automatically insert a newline. Instead, we have to explicitly add the character \n at the end of each line.

Third, we have to explicitly refer to each of the elements of olympian when building the string to write. Note that just putting .format(olympian) wouldn’t work because the interpreter would see only one value (a tuple) when it was expecting three values to try to substitute into the string template. Later in the book we will see that python provides an advanced technique for automatically unpacking the three values from the tuple, with .format(\*olympian).

As described previously, if one or more columns contain text, and that text could contain commas, we need to do something to distinguish a comma in the text from a comma that is separating different values (cells in the table). If we want to enclose each value in double quotes, it can start to get a little tricky, because we will need to have the double quote character inside the string output. But it is doable. Indeed, one reason Python allows strings to be delimited with either single quotes or double quotes is so that one can be used to delimit the string and the other can be a character in the string. If you get to the point where you need to quote all of the values, we recommend learning to use python’s csv module.

olympians = [("John Aalberg", 31, "Cross Country Skiing, 15KM"),

("Minna Maarit Aalto", 30, "Sailing"),

("Win Valdemar Aaltonen", 54, "Art Competitions"),

("Wakako Abe", 18, "Cycling")]

​

outfile = open("reduced\_olympics2.csv","w")

# output the header row

outfile.write('"Name","Age","Sport"')

outfile.write('\n')

# output each of the rows:

for olympian in olympians:

row\_string = '"{}", "{}", "{}"'.format(olympian[0], olympian[1], olympian[2])

outfile.write(row\_string)

outfile.write('\n')

outfile.close()

# 10.12. Glossary

**open**

You must open a file before you can read its contents.

**close**

When you are done with a file, you should close it.

**read**

Will read the entire contents of a file as a string. This is often used in an assignment statement so that a variable can reference the contents of the file.

**readline**

Will read a single line from the file, up to and including the first instance of the newline character.

**readlines**

Will read the entire contents of a file into a list where each line of the file is a string and is an element in the list.

**write**

Will add characters to the end of a file that has been opened for writing.

Bottom of Form