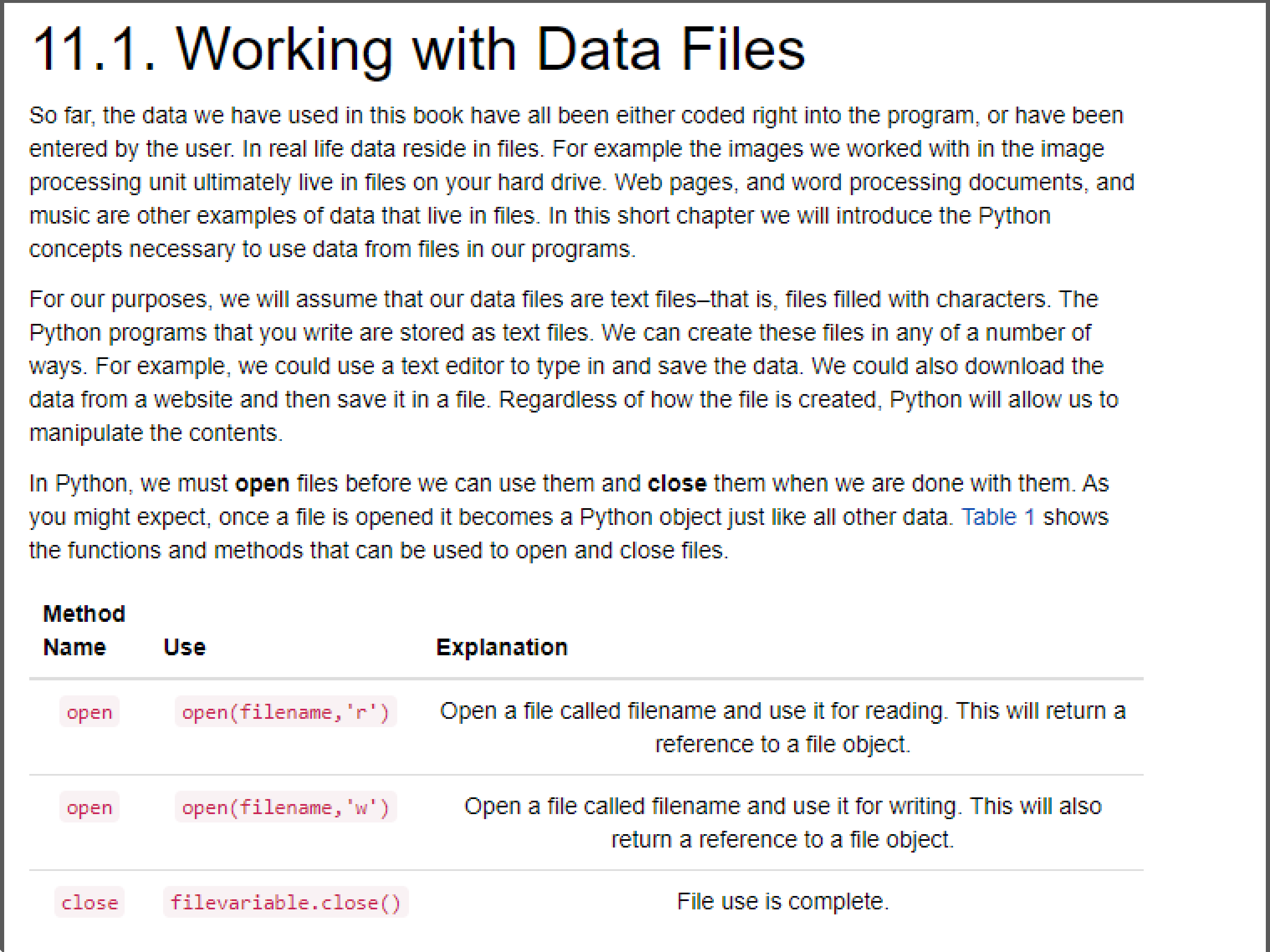
# Network Programming in Python I

Justin Ellis MBA

# Review

Any Questions?

# Files



## 11.1. Working with Data Files

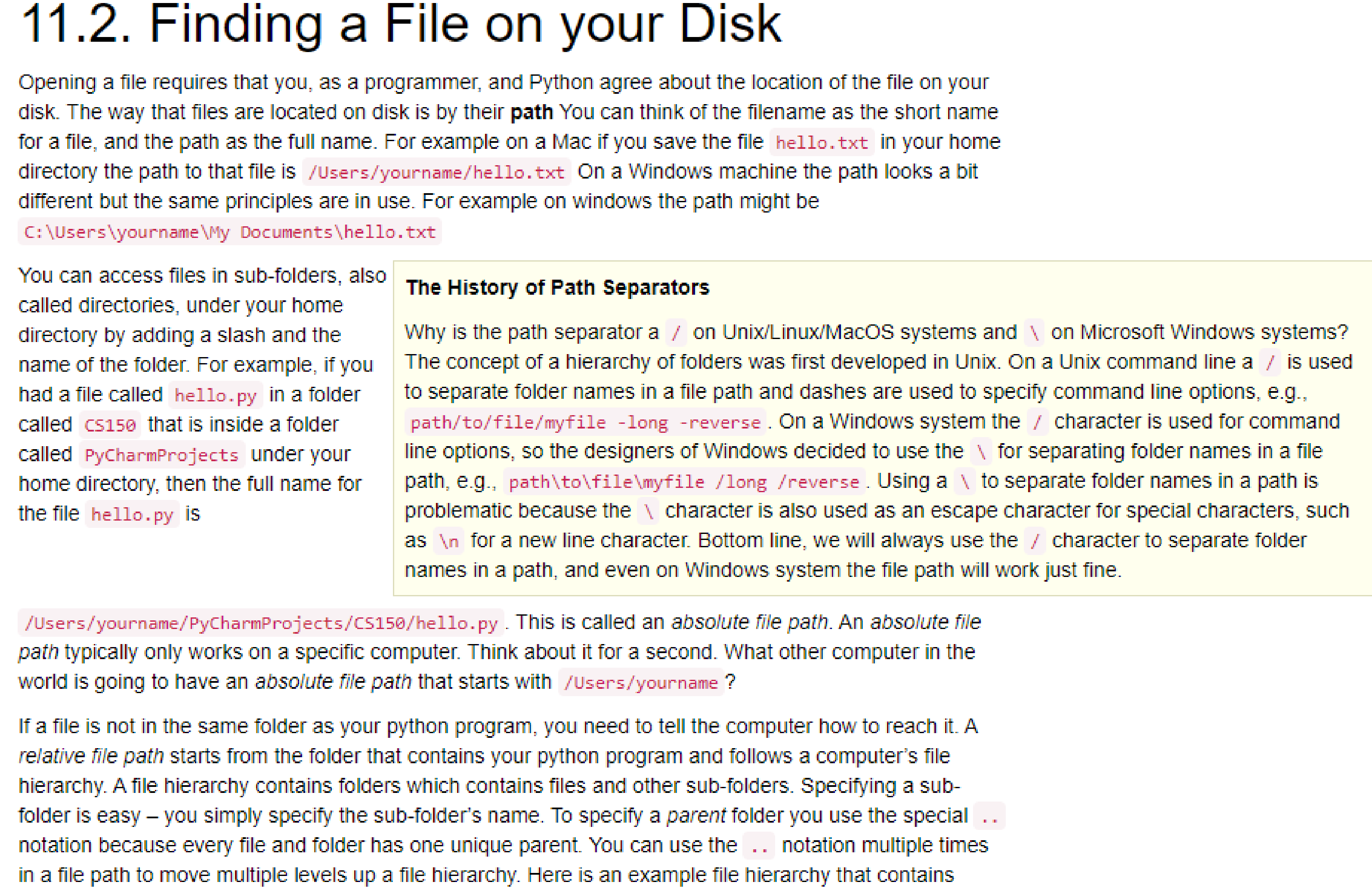
So far, the data we have used in this book have all been either coded right into the program, or have been entered by the user. In real life data reside in files. For example the images we worked with in the image processing unit ultimately live in files on your hard drive. Web pages, and word processing documents, and music are other examples of data that live in files. In this short chapter we will introduce the Python concepts necessary to use data from files in our programs.

For our purposes, we will assume that our data files are text files–that is, files filled with characters. The Python programs that you write are stored as text files. We can create these files in any of a number of ways. For example, we could use a text editor to type in and save the data. We could also download the data from a website and then save it in a file. Regardless of how the file is created, Python will allow us to manipulate the contents.

In Python, we must **open** files before we can use them and **close** them when we are done with them. As you might expect, once a file is opened it becomes a Python object just like all other data. Table 1 shows the functions and methods that can be used to open and close files.

| Method Name | Use | Explanation |
|---|---|---|
| `open` | `open(filename,'r')` | Open a file called filename and use it for reading. This will return a reference to a file object. |
| `open` | `open(filename,'w')` | Open a file called filename and use it for writing. This will also return a reference to a file object. |
| `close` | `filevariable.close()` | File use is complete. |

# Files



## 11.2. Finding a File on your Disk

Opening a file requires that you, as a programmer, and Python agree about the location of the file on your disk. The way that files are located on disk is by their **path** You can think of the filename as the short name for a file, and the path as the full name. For example on a Mac if you save the file `hello.txt` in your home directory the path to that file is `/Users/yourname/hello.txt` On a Windows machine the path looks a bit different but the same principles are in use. For example on windows the path might be `C:\Users\yourname\My Documents\hello.txt`

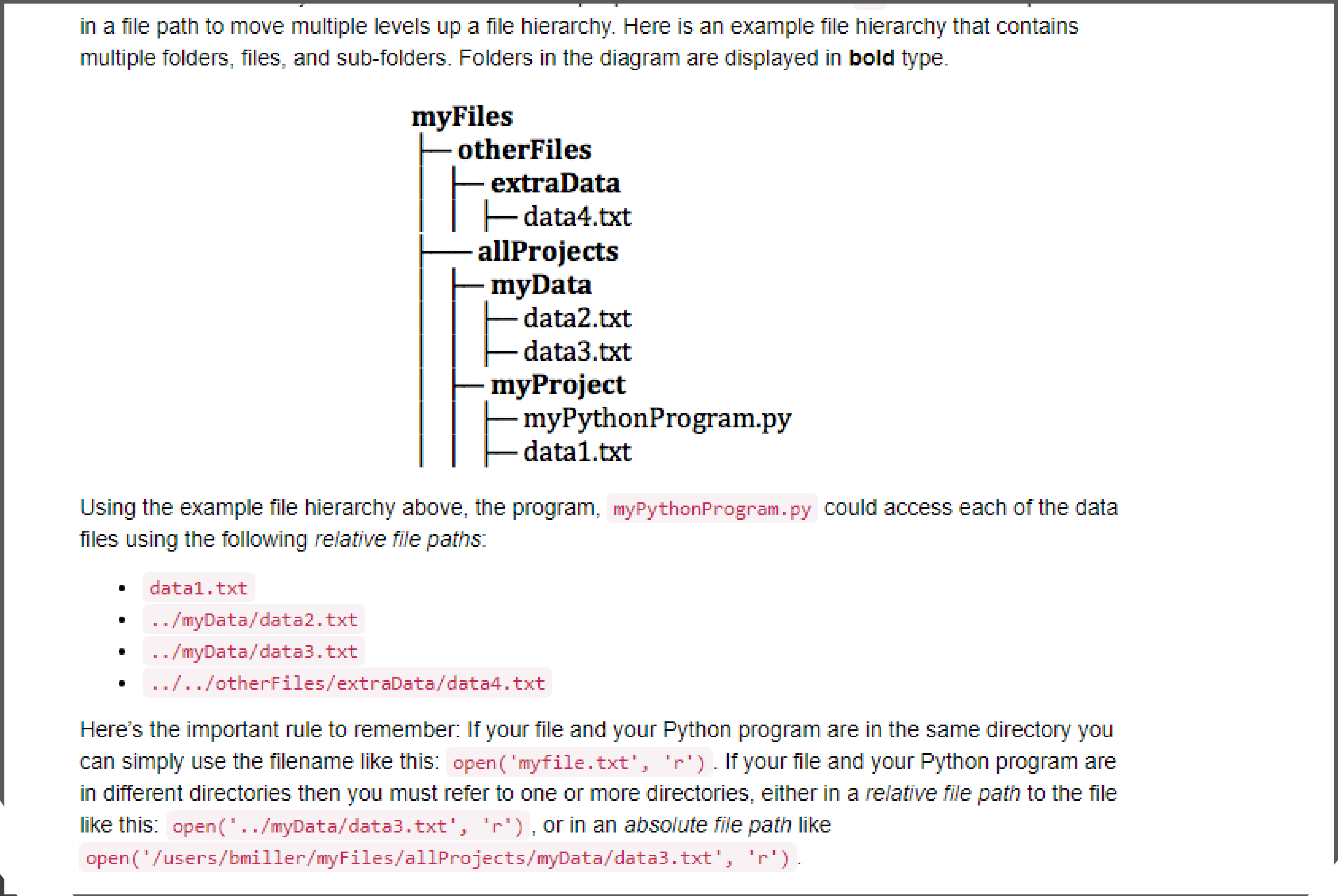
You can access files in sub-folders, also called directories, under your home directory by adding a slash and the name of the folder. For example, if you had a file called `hello.py` in a folder called `CS150` that is inside a folder called `PyCharmProjects` under your home directory, then the full name for the file `hello.py` is `/Users/yourname/PyCharmProjects/CS150/hello.py`. This is called an *absolute file path*. An *absolute file path* typically only works on a specific computer. Think about it for a second. What other computer in the world is going to have an *absolute file path* that starts with `/Users/yourname`?

**The History of Path Separators**

Why is the path separator a `/` on Unix/Linux/MacOS systems and `\` on Microsoft Windows systems? The concept of a hierarchy of folders was first developed in Unix. On a Unix command line a `/` is used to separate folder names in a file path and dashes are used to specify command line options, e.g., `path/to/file/myfile -long -reverse`. On a Windows system the `/` character is used for command line options, so the designers of Windows decided to use the `\` for separating folder names in a file path, e.g., `path\to\file\myfile /long /reverse`. Using a `\` to separate folder names in a path is problematic because the `\` character is also used as an escape character for special characters, such as `\n` for a new line character. Bottom line, we will always use the `/` character to separate folder names in a path, and even on Windows system the file path will work just fine.

If a file is not in the same folder as your python program, you need to tell the computer how to reach it. A *relative file path* starts from the folder that contains your python program and follows a computer's file hierarchy. A file hierarchy contains folders which contains files and other sub-folders. Specifying a sub-folder is easy – you simply specify the sub-folder's name. To specify a *parent* folder you use the special `..` notation because every file and folder has one unique parent. You can use the `..` notation multiple times in a file path to move multiple levels up a file hierarchy. Here is an example file hierarchy that contains

# Files



in a file path to move multiple levels up a file hierarchy. Here is an example file hierarchy that contains multiple folders, files, and sub-folders. Folders in the diagram are displayed in **bold** type.

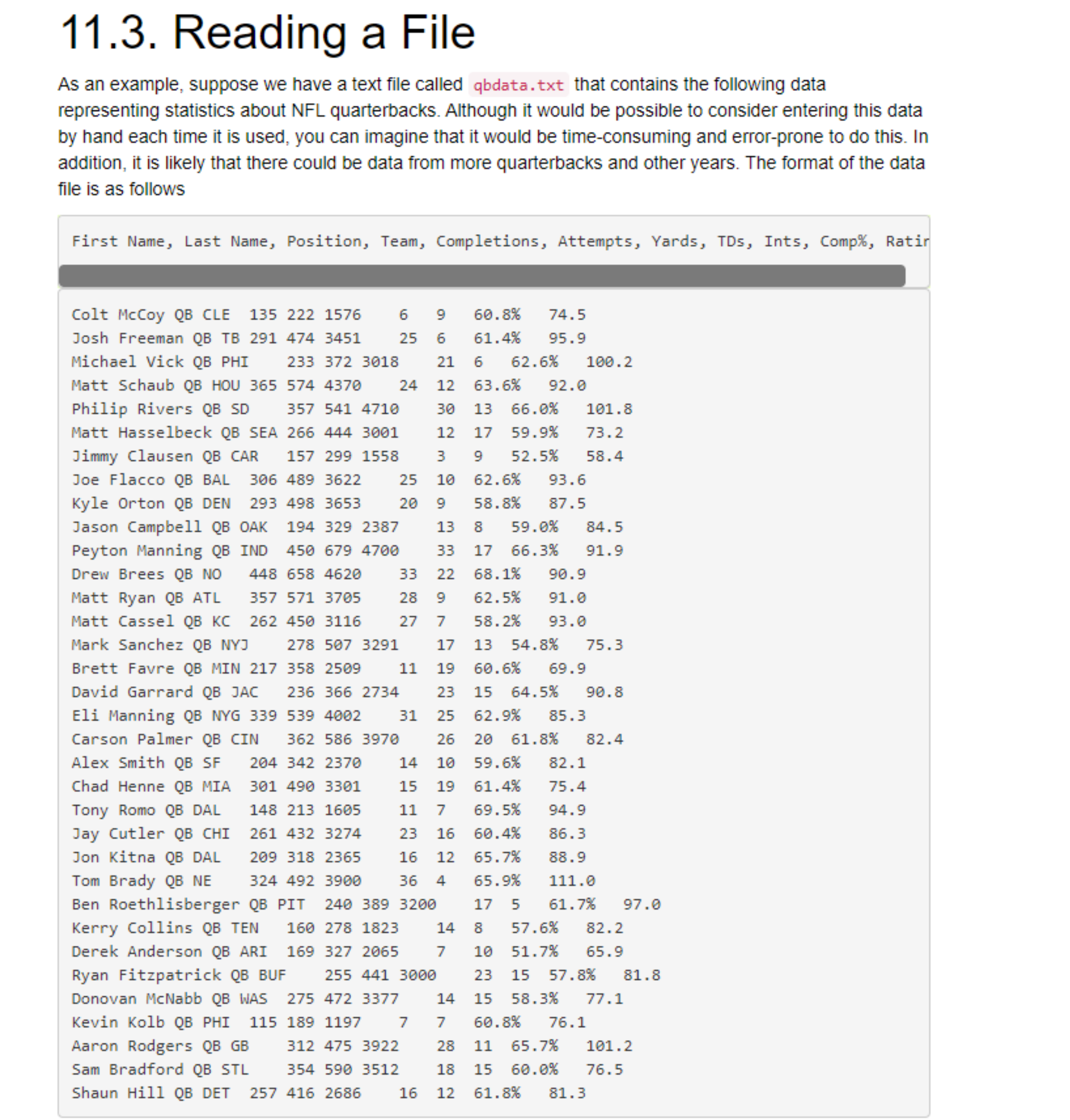
```
myFiles
├─ otherFiles
│  ├─ extraData
│  │  ├─ data4.txt
├── allProjects
│  ├─ myData
│  │  ├─ data2.txt
│  │  ├─ data3.txt
│  ├─ myProject
│  │  ├─ myPythonProgram.py
│  │  ├─ data1.txt
```

Using the example file hierarchy above, the program, `myPythonProgram.py` could access each of the data files using the following *relative file paths*:

- `data1.txt`
- `../myData/data2.txt`
- `../myData/data3.txt`
- `../../otherFiles/extraData/data4.txt`

Here's the important rule to remember: If your file and your Python program are in the same directory you can simply use the filename like this: `open('myfile.txt', 'r')`. If your file and your Python program are in different directories then you must refer to one or more directories, either in a *relative file path* to the file like this: `open('../myData/data3.txt', 'r')`, or in an *absolute file path* like `open('/users/bmiller/myFiles/allProjects/myData/data3.txt', 'r')`.

# Files



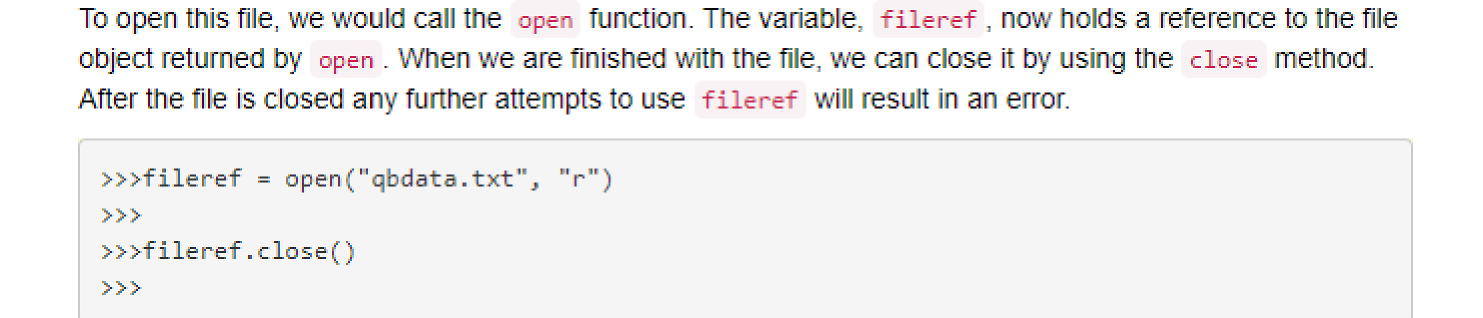
## 11.3. Reading a File

As an example, suppose we have a text file called `qbdata.txt` that contains the following data representing statistics about NFL quarterbacks. Although it would be possible to consider entering this data by hand each time it is used, you can imagine that it would be time-consuming and error-prone to do this. In addition, it is likely that there could be data from more quarterbacks and other years. The format of the data file is as follows

```
First Name, Last Name, Position, Team, Completions, Attempts, Yards, TDs, Ints, Comp%, Ratir
```

```
Colt McCoy QB CLE  135 222 1576    6   9   60.8%   74.5
Josh Freeman QB TB 291 474 3451    25  6   61.4%   95.9
Michael Vick QB PHI    233 372 3018    21  6   62.6%   100.2
Matt Schaub QB HOU 365 574 4370    24  12  63.6%   92.0
Philip Rivers QB SD    357 541 4710    30  13  66.0%   101.8
Matt Hasselbeck QB SEA 266 444 3001    12  17  59.9%   73.2
Jimmy Clausen QB CAR   157 299 1558    3   9   52.5%   58.4
Joe Flacco QB BAL  306 489 3622    25  10  62.6%   93.6
Kyle Orton QB DEN  293 498 3653    20  9   58.8%   87.5
Jason Campbell QB OAK  194 329 2387    13  8   59.0%   84.5
Peyton Manning QB IND  450 679 4700    33  17  66.3%   91.9
Drew Brees QB NO   448 658 4620    33  22  68.1%   90.9
Matt Ryan QB ATL   357 571 3705    28  9   62.5%   91.0
Matt Cassel QB KC  262 450 3116    27  7   58.2%   93.0
Mark Sanchez QB NYJ    278 507 3291    17  13  54.8%   75.3
Brett Favre QB MIN 217 358 2509    11  19  60.6%   69.9
David Garrard QB JAC   236 366 2734    23  15  64.5%   90.8
Eli Manning QB NYG 339 539 4002    31  25  62.9%   85.3
Carson Palmer QB CIN   362 586 3970    26  20  61.8%   82.4
Alex Smith QB SF   204 342 2370    14  10  59.6%   82.1
Chad Henne QB MIA  301 490 3301    15  19  61.4%   75.4
Tony Romo QB DAL   148 213 1605    11  7   69.5%   94.9
Jay Cutler QB CHI  261 432 3274    23  16  60.4%   86.3
Jon Kitna QB DAL   209 318 2365    16  12  65.7%   88.9
Tom Brady QB NE    324 492 3900    36  4   65.9%   111.0
Ben Roethlisberger QB PIT  240 389 3200    17  5   61.7%   97.0
Kerry Collins QB TEN   160 278 1823    14  8   57.6%   82.2
Derek Anderson QB ARI  169 327 2065    7   10  51.7%   65.9
Ryan Fitzpatrick QB BUF    255 441 3000    23  15  57.8%   81.8
Donovan McNabb QB WAS  275 472 3377    14  15  58.3%   77.1
Kevin Kolb QB PHI  115 189 1197    7   7   60.8%   76.1
Aaron Rodgers QB GB    312 475 3922    28  11  65.7%   101.2
Sam Bradford QB STL    354 590 3512    18  15  60.0%   76.5
Shaun Hill QB DET  257 416 2686    16  12  61.8%   81.3
```

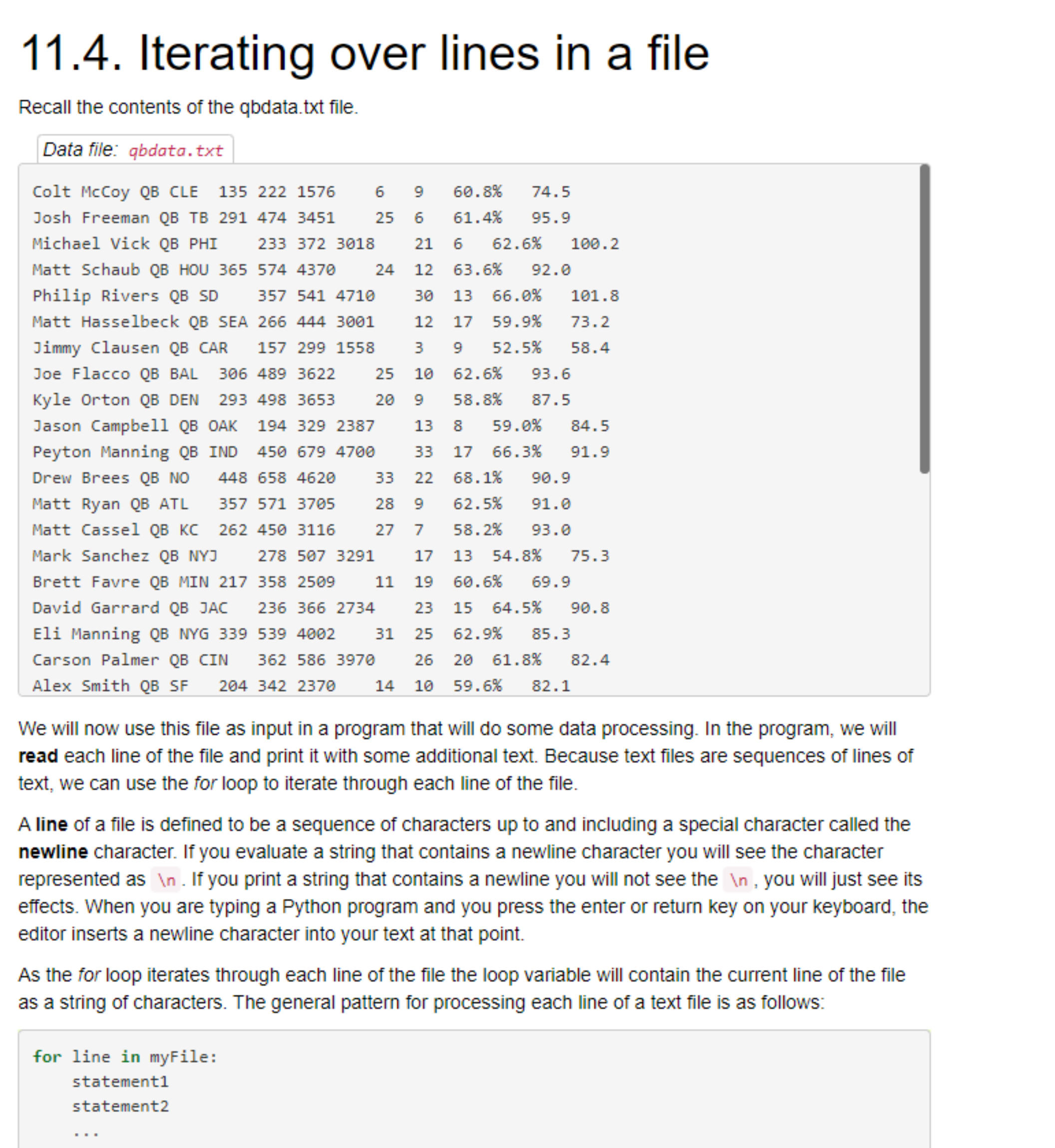
# Files



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("qbdata.txt", "r")
>>>
>>>fileref.close()
>>>
```

# Files



## 11.4. Iterating over lines in a file

Recall the contents of the qbdata.txt file.

*Data file:* `qbdata.txt`

```
Colt McCoy QB CLE  135 222 1576    6   9   60.8%   74.5
Josh Freeman QB TB 291 474 3451    25  6   61.4%   95.9
Michael Vick QB PHI    233 372 3018    21  6   62.6%   100.2
Matt Schaub QB HOU 365 574 4370    24  12  63.6%   92.0
Philip Rivers QB SD    357 541 4710    30  13  66.0%   101.8
Matt Hasselbeck QB SEA 266 444 3001    12  17  59.9%   73.2
Jimmy Clausen QB CAR   157 299 1558    3   9   52.5%   58.4
Joe Flacco QB BAL  306 489 3622    25  10  62.6%   93.6
Kyle Orton QB DEN  293 498 3653    20  9   58.8%   87.5
Jason Campbell QB OAK  194 329 2387    13  8   59.0%   84.5
Peyton Manning QB IND  450 679 4700    33  17  66.3%   91.9
Drew Brees QB NO   448 658 4620    33  22  68.1%   90.9
Matt Ryan QB ATL   357 571 3705    28  9   62.5%   91.0
Matt Cassel QB KC  262 450 3116    27  7   58.2%   93.0
Mark Sanchez QB NYJ    278 507 3291    17  13  54.8%   75.3
Brett Favre QB MIN 217 358 2509    11  19  60.6%   69.9
David Garrard QB JAC   236 366 2734    23  15  64.5%   90.8
Eli Manning QB NYG 339 539 4002    31  25  62.9%   85.3
Carson Palmer QB CIN   362 586 3970    26  20  61.8%   82.4
Alex Smith QB SF   204 342 2370    14  10  59.6%   82.1
```

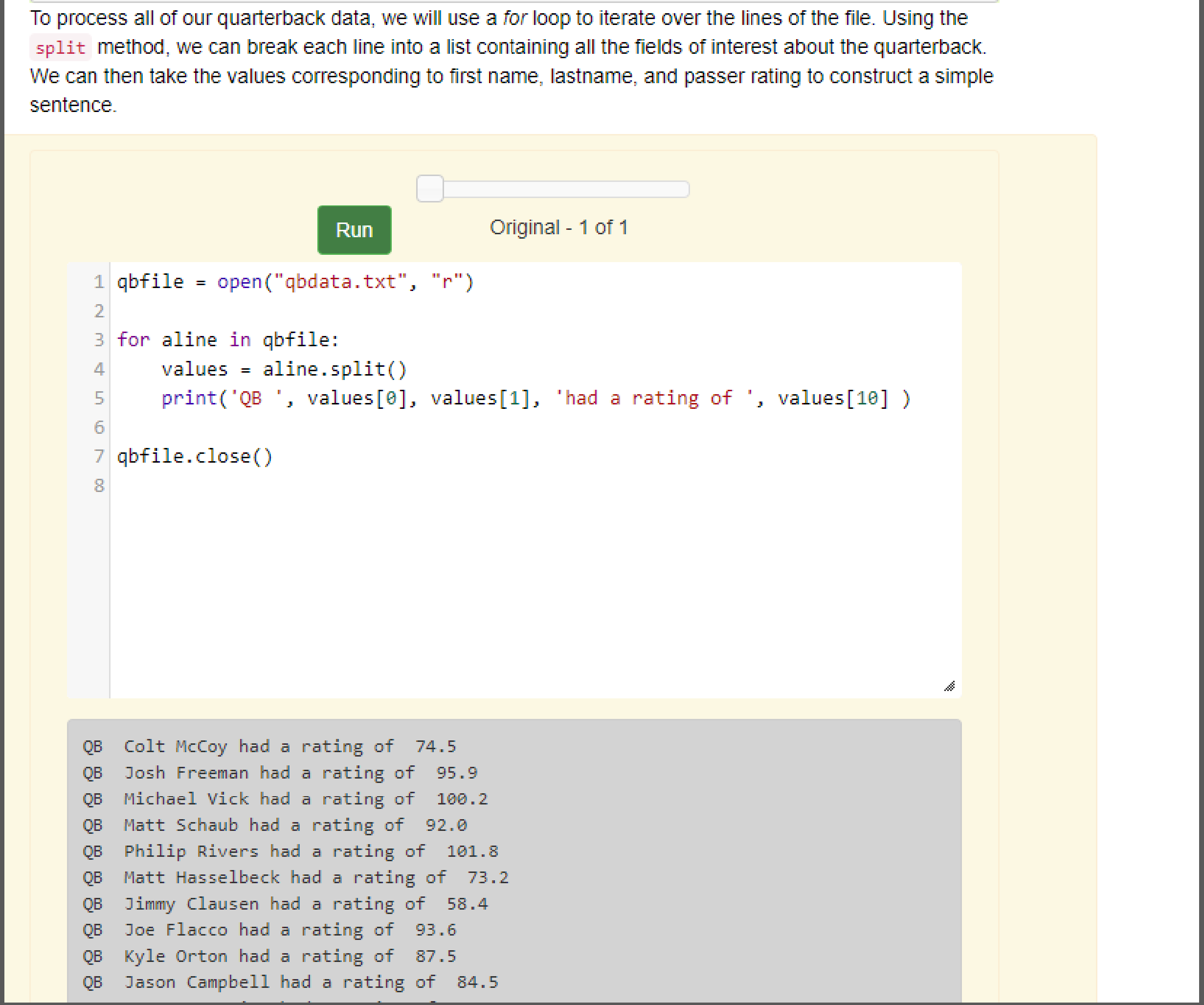
We will now use this file as input in a program that will do some data processing. In the program, we will **read** each line of the file and print it with some additional text. Because text files are sequences of lines of text, we can use the *for* loop to iterate through each line of the file.

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. When you are typing a Python program and you press the enter or return key on your keyboard, the editor inserts a newline character into your text at that point.

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:
    statement1
    statement2
    ...
```

# Files

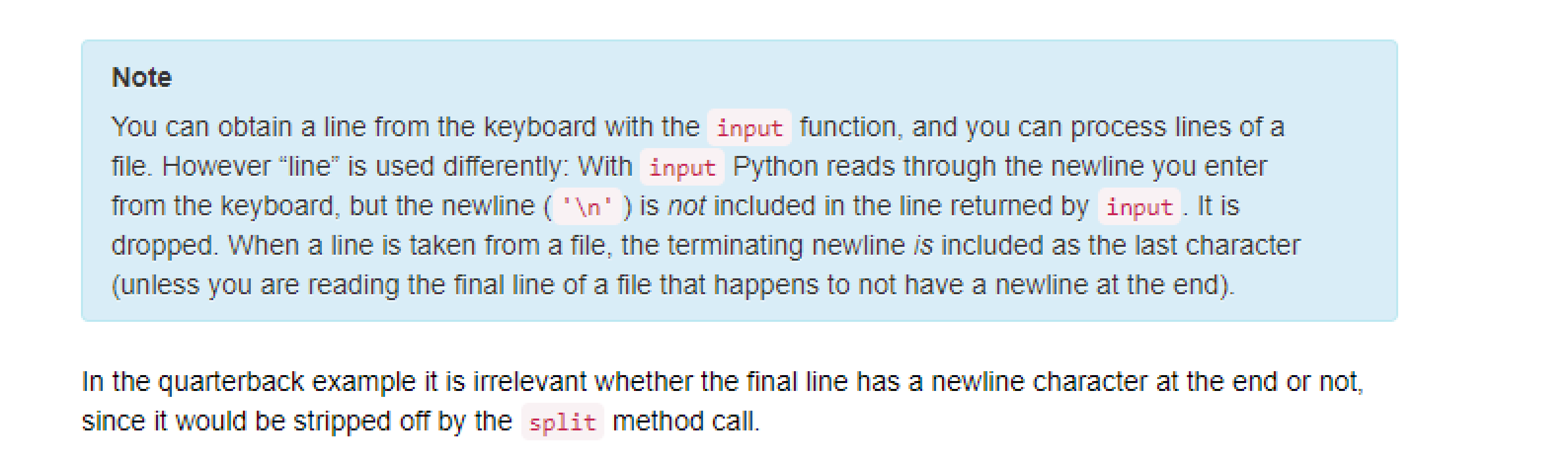


To process all of our quarterback data, we will use a *for* loop to iterate over the lines of the file. Using the `split` method, we can break each line into a list containing all the fields of interest about the quarterback. We can then take the values corresponding to first name, lastname, and passer rating to construct a simple sentence.

```python
qbfile = open("qbdata.txt", "r")

for aline in qbfile:
    values = aline.split()
    print('QB ', values[0], values[1], 'had a rating of ', values[10] )

qbfile.close()
```

```
QB  Colt McCoy had a rating of  74.5
QB  Josh Freeman had a rating of  95.9
QB  Michael Vick had a rating of  100.2
QB  Matt Schaub had a rating of  92.0
QB  Philip Rivers had a rating of  101.8
QB  Matt Hasselbeck had a rating of  73.2
QB  Jimmy Clausen had a rating of  58.4
QB  Joe Flacco had a rating of  93.6
QB  Kyle Orton had a rating of  87.5
QB  Jason Campbell had a rating of  84.5
```

# Files

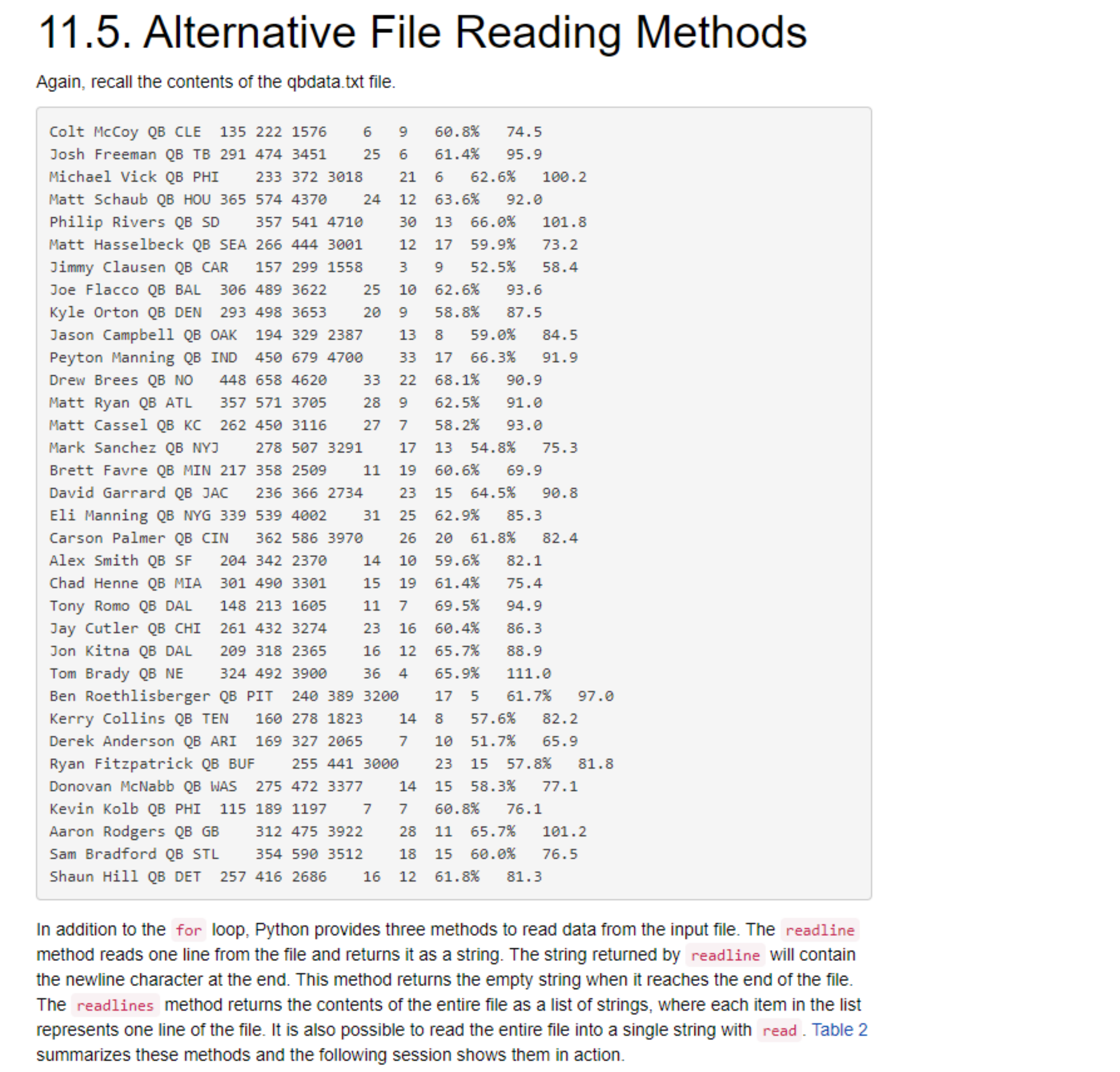


**Note**

You can obtain a line from the keyboard with the `input` function, and you can process lines of a file. However “line” is used differently: With `input` Python reads through the newline you enter from the keyboard, but the newline (`'\n'`) is *not* included in the line returned by `input`. It is dropped. When a line is taken from a file, the terminating newline *is* included as the last character (unless you are reading the final line of a file that happens to not have a newline at the end).

In the quarterback example it is irrelevant whether the final line has a newline character at the end or not, since it would be stripped off by the `split` method call.

# Files



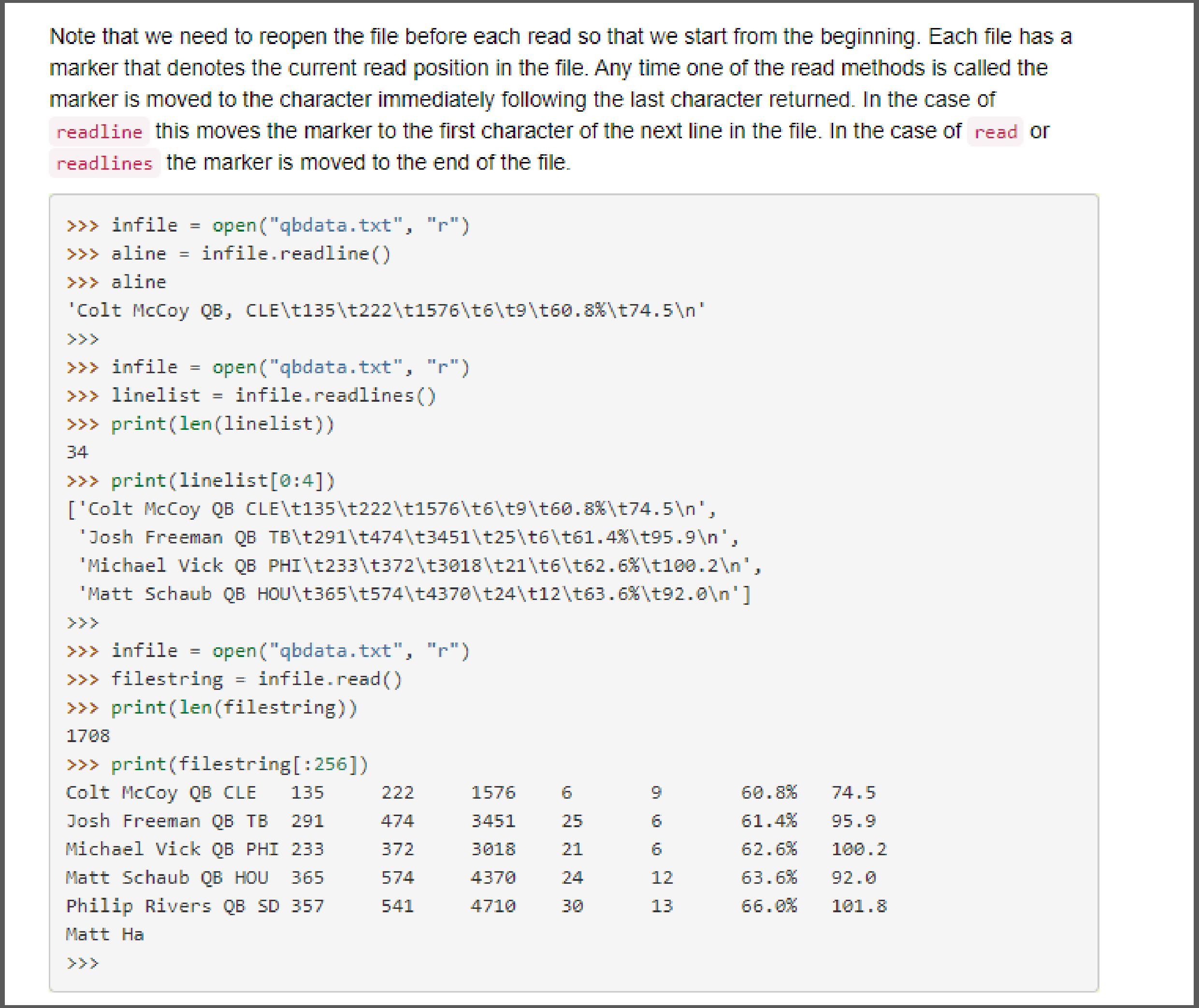
## 11.5. Alternative File Reading Methods

Again, recall the contents of the qbdata.txt file.

```
Colt McCoy QB CLE  135 222 1576    6   9   60.8%   74.5
Josh Freeman QB TB 291 474 3451    25  6   61.4%   95.9
Michael Vick QB PHI    233 372 3018    21  6   62.6%   100.2
Matt Schaub QB HOU 365 574 4370    24  12  63.6%   92.0
Philip Rivers QB SD    357 541 4710    30  13  66.0%   101.8
Matt Hasselbeck QB SEA 266 444 3001    12  17  59.9%   73.2
Jimmy Clausen QB CAR   157 299 1558    3   9   52.5%   58.4
Joe Flacco QB BAL  306 489 3622    25  10  62.6%   93.6
Kyle Orton QB DEN  293 498 3653    20  9   58.8%   87.5
Jason Campbell QB OAK  194 329 2387    13  8   59.0%   84.5
Peyton Manning QB IND  450 679 4700    33  17  66.3%   91.9
Drew Brees QB NO   448 658 4620    33  22  68.1%   90.9
Matt Ryan QB ATL   357 571 3705    28  9   62.5%   91.0
Matt Cassel QB KC  262 450 3116    27  7   58.2%   93.0
Mark Sanchez QB NYJ    278 507 3291    17  13  54.8%   75.3
Brett Favre QB MIN 217 358 2509    11  19  60.6%   69.9
David Garrard QB JAC   236 366 2734    23  15  64.5%   90.8
Eli Manning QB NYG 339 539 4002    31  25  62.9%   85.3
Carson Palmer QB CIN   362 586 3970    26  20  61.8%   82.4
Alex Smith QB SF   204 342 2370    14  10  59.6%   82.1
Chad Henne QB MIA  301 490 3301    15  19  61.4%   75.4
Tony Romo QB DAL   148 213 1605    11  7   69.5%   94.9
Jay Cutler QB CHI  261 432 3274    23  16  60.4%   86.3
Jon Kitna QB DAL   209 318 2365    16  12  65.7%   88.9
Tom Brady QB NE    324 492 3900    36  4   65.9%   111.0
Ben Roethlisberger QB PIT  240 389 3200    17  5   61.7%   97.0
Kerry Collins QB TEN   160 278 1823    14  8   57.6%   82.2
Derek Anderson QB ARI  169 327 2065    7   10  51.7%   65.9
Ryan Fitzpatrick QB BUF    255 441 3000    23  15  57.8%   81.8
Donovan McNabb QB WAS  275 472 3377    14  15  58.3%   77.1
Kevin Kolb QB PHI  115 189 1197    7   7   60.8%   76.1
Aaron Rodgers QB GB    312 475 3922    28  11  65.7%   101.2
Sam Bradford QB STL    354 590 3512    18  15  60.0%   76.5
Shaun Hill QB DET  257 416 2686    16  12  61.8%   81.3
```

In addition to the `for` loop, Python provides three methods to read data from the input file. The `readline` method reads one line from the file and returns it as a string. The string returned by `readline` will contain the newline character at the end. This method returns the empty string when it reaches the end of the file. The `readlines` method returns the contents of the entire file as a list of strings, where each item in the list represents one line of the file. It is also possible to read the entire file into a single string with `read`. Table 2 summarizes these methods and the following session shows them in action.

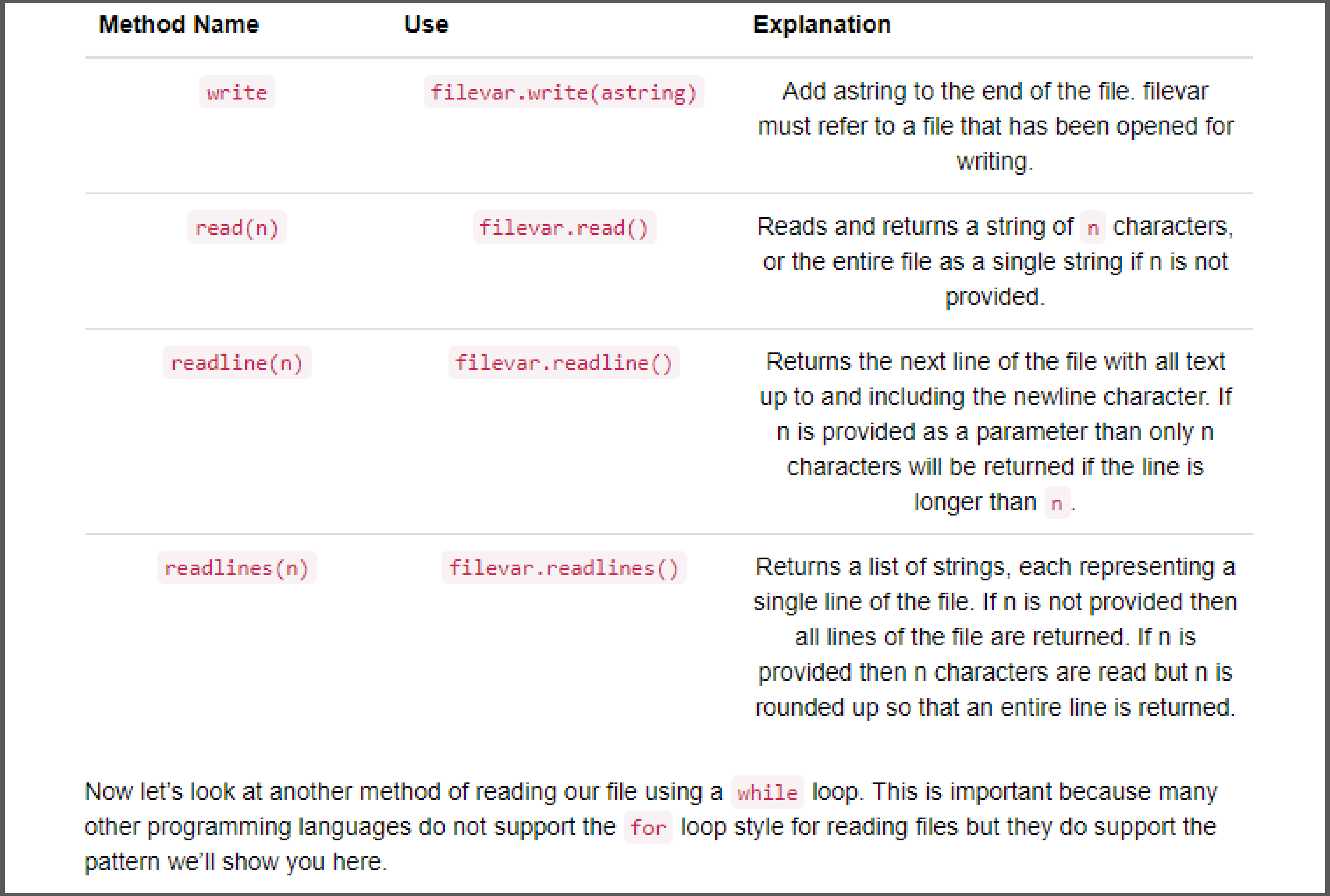
# Files



Note that we need to reopen the file before each read so that we start from the beginning. Each file has a marker that denotes the current read position in the file. Any time one of the read methods is called the marker is moved to the character immediately following the last character returned. In the case of `readline` this moves the marker to the first character of the next line in the file. In the case of `read` or `readlines` the marker is moved to the end of the file.

```
>>> infile = open("qbdata.txt", "r")
>>> aline = infile.readline()
>>> aline
'Colt McCoy QB, CLE\t135\t222\t1576\t6\t9\t60.8%\t74.5\n'
>>>
>>> infile = open("qbdata.txt", "r")
>>> linelist = infile.readlines()
>>> print(len(linelist))
34
>>> print(linelist[0:4])
['Colt McCoy QB CLE\t135\t222\t1576\t6\t9\t60.8%\t74.5\n',
 'Josh Freeman QB TB\t291\t474\t3451\t25\t6\t61.4%\t95.9\n',
 'Michael Vick QB PHI\t233\t372\t3018\t21\t6\t62.6%\t100.2\n',
 'Matt Schaub QB HOU\t365\t574\t4370\t24\t12\t63.6%\t92.0\n']
>>>
>>> infile = open("qbdata.txt", "r")
>>> filestring = infile.read()
>>> print(len(filestring))
1708
>>> print(filestring[:256])
Colt McCoy QB CLE   135     222     1576    6       9       60.8%   74.5
Josh Freeman QB TB  291     474     3451    25      6       61.4%   95.9
Michael Vick QB PHI 233     372     3018    21      6       62.6%   100.2
Matt Schaub QB HOU  365     574     4370    24      12      63.6%   92.0
Philip Rivers QB SD 357     541     4710    30      13      66.0%   101.8
Matt Ha
>>>
```

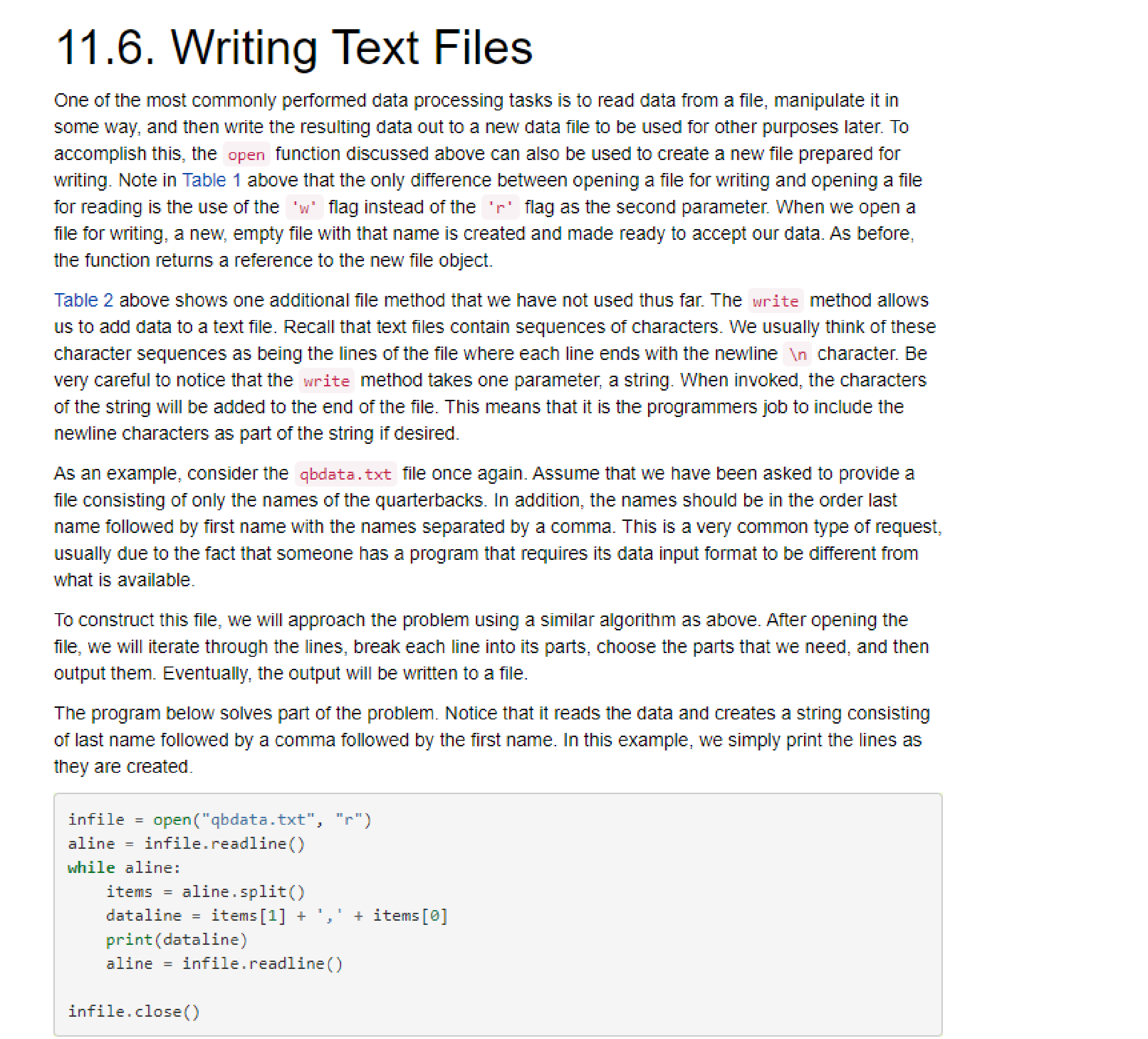
# Files



| 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. |

Now let's look at another method of reading our file using a `while` loop. This is important because many other programming languages do not support the `for` loop style for reading files but they do support the pattern we'll show you here.

# Files



## 11.6. 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 above 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. As before, the function returns a reference to the new file object.

Table 2 above shows one additional file method 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 programmers job to include the newline characters as part of the string if desired.

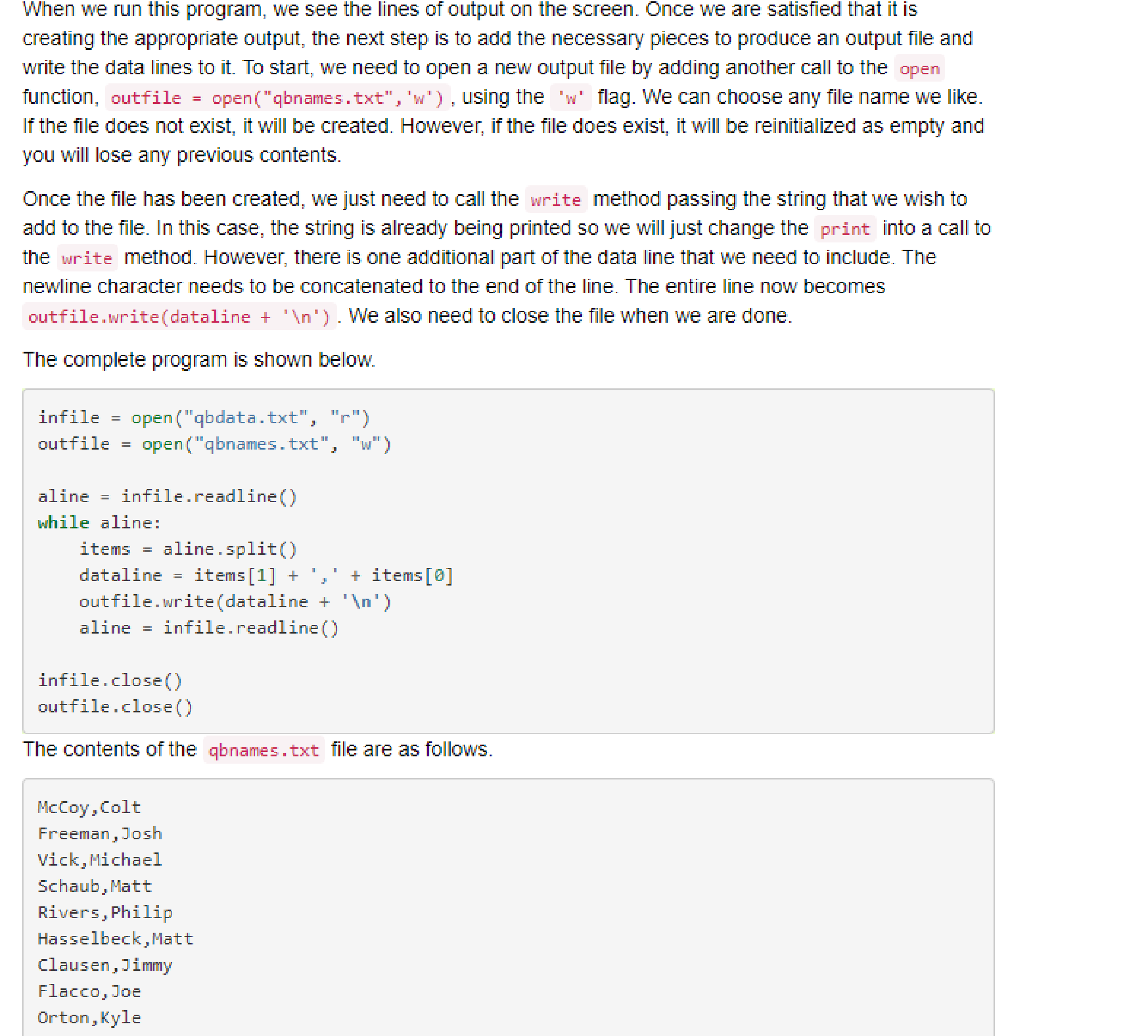
As an example, consider the `qbdata.txt` file once again. Assume that we have been asked to provide a file consisting of only the names of the quarterbacks. In addition, the names should be in the order last name followed by first name with the names separated by a comma. This is a very common type of request, usually due to the fact that someone has a program that requires its data input format to be different from what is available.

To construct this file, we will approach the problem using a similar algorithm as above. After opening the file, we will iterate through the lines, break each line into its parts, choose the parts that we need, and then output them. Eventually, the output will be written to a file.

The program below solves part of the problem. Notice that it reads the data and creates a string consisting of last name followed by a comma followed by the first name. In this example, we simply print the lines as they are created.

```
infile = open("qbdata.txt", "r")
aline = infile.readline()
while aline:
    items = aline.split()
    dataline = items[1] + ',' + items[0]
    print(dataline)
    aline = infile.readline()

infile.close()
```

# Files



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 adding another call to the `open` function, `outfile = open("qbnames.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 one additional part of the data line that we need to include. The newline character needs to be concatenated to the end of the line. The entire line now becomes `outfile.write(dataline + '\n')`. We also need to close the file when we are done.

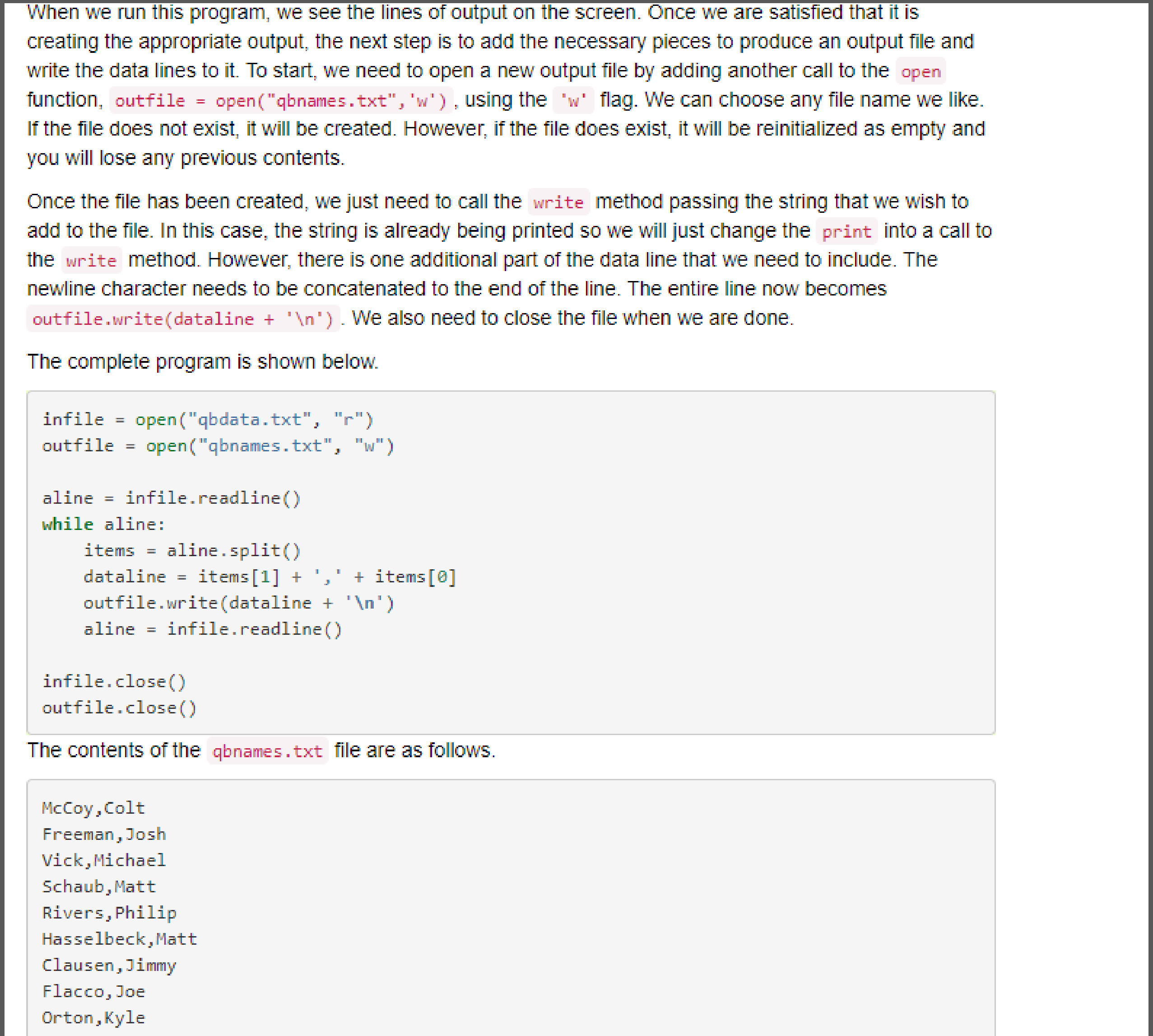
The complete program is shown below.

```
infile = open("qbdata.txt", "r")
outfile = open("qbnames.txt", "w")

aline = infile.readline()
while aline:
    items = aline.split()
    dataline = items[1] + ',' + items[0]
    outfile.write(dataline + '\n')
    aline = infile.readline()

infile.close()
outfile.close()
```

The contents of the `qbnames.txt` file are as follows.

```
McCoy,Colt
Freeman,Josh
Vick,Michael
Schaub,Matt
Rivers,Philip
Hasselbeck,Matt
Clausen,Jimmy
Flacco,Joe
Orton,Kyle
```

# Files



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 adding another call to the `open` function, `outfile = open("qbnames.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 one additional part of the data line that we need to include. The newline character needs to be concatenated to the end of the line. The entire line now becomes `outfile.write(dataline + '\n')`. We also need to close the file when we are done.

The complete program is shown below.

```
infile = open("qbdata.txt", "r")
outfile = open("qbnames.txt", "w")

aline = infile.readline()
while aline:
    items = aline.split()
    dataline = items[1] + ',' + items[0]
    outfile.write(dataline + '\n')
    aline = infile.readline()

infile.close()
outfile.close()
```

The contents of the `qbnames.txt` file are as follows.

```
McCoy,Colt
Freeman,Josh
Vick,Michael
Schaub,Matt
Rivers,Philip
Hasselbeck,Matt
Clausen,Jimmy
Flacco,Joe
Orton,Kyle
```

# Final Project

Make python do something

Requirements

- Must work on your machine (Can take code but must cite)

Rubric

- 15 points source code
- 10 points video demonstration
- Submit code from github

Fun Python Projects for Beginners to Try in 2021 | Career Karma