FILE HANDLING

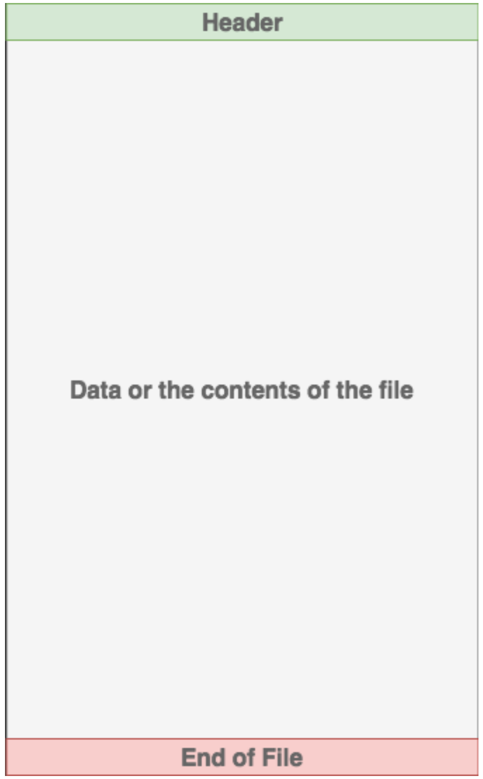
Files

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g., hard disk).

Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

What does a File contain?

- Files on most modern file systems are composed of three main parts:
  - Header: metadata about the contents of the file (file name, size, type, and so on)
  - Data: contents of the file as written by the creator or editor
  - End of file (EOF): special character that indicates the end of the file



File Paths

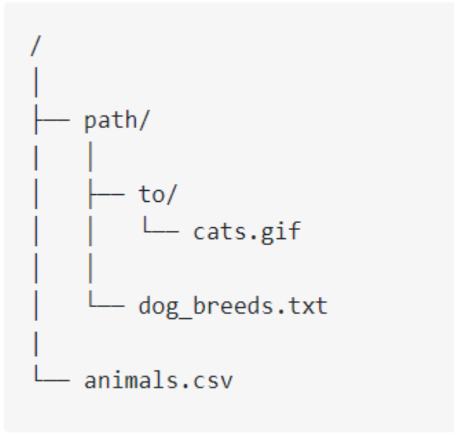
When you access a file on an operating system, a file path is required. **The file path is a string that represents the location of a file.**
It's broken up into three major parts:

- **Folder Path:** the file folder location on the file system where subsequent folders are separated by a forward slash / (Unix) or backslash \ (Windows)
- **File Name:** the actual name of the file
- **Extension:** the end of the file path pre-pended with a period (.) used to indicate the file type

Example – File path

- Let's say you have a file located within a file structure like this:

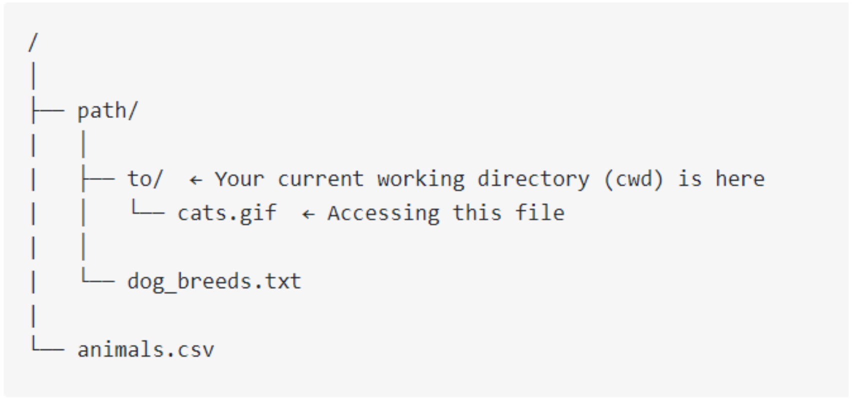
- Let's say you wanted to access the cats.gif file, and your current location was in the same folder as path. In order to access the file, you need to go through the path folder and then the to folder, finally arriving at the cats.gif file. The Folder Path is path/to/. The File Name is cats. The File Extension is .gif. So the full path is path/to/cats.gif.



```
/
|
├── path/
|   |
|   ├── to/
|   |   └── cats.gif
|   |
|   └── dog_breeds.txt
|
└── animals.csv
```

File Paths

- Now let’s say that your current location or current working directory (cwd) is in the to folder of our example folder structure. Instead of referring to the cats.gif by the full path of path/to/cats.gif, the file can be simply referenced by the file name and extension cats.gif.

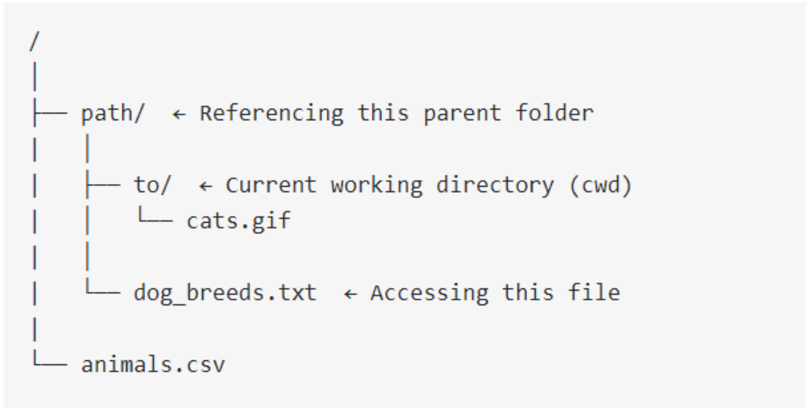


```
/
│
├── path/
|   │
|   ├── to/  ← Your current working directory (cwd) is here
|   │   └── cats.gif  ← Accessing this file
|   │
|   └── dog_breeds.txt
|
└── animals.csv
```

File Paths

- But what about dog_breeds.txt? How would you access that without using the full path? You can use the special characters double-dot (..) to move one directory up. This means that ../dog_breeds.txt will reference the dog_breeds.txt file from the directory path.

- The double-dot (..) can be chained together to traverse multiple directories above the current directory. For example, to access animals.csv from the to folder, you would use ../../animals.csv.

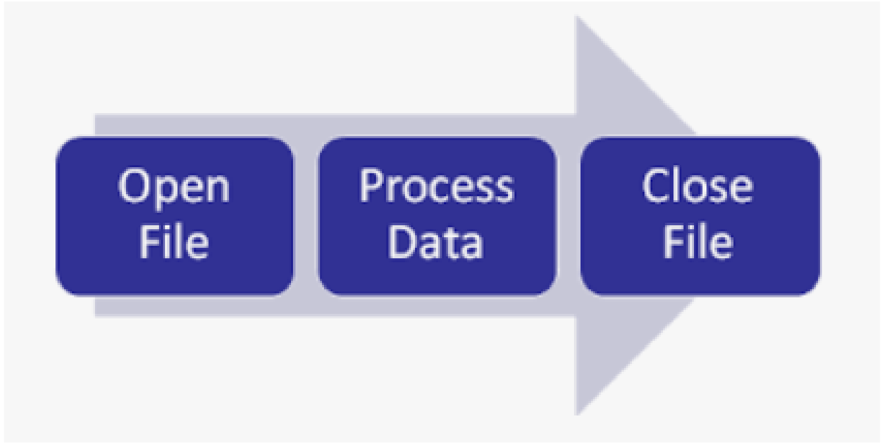


```
/
│
├── path/  ← Referencing this parent folder
|   │
|   ├── to/  ← Current working directory (cwd)
|   │   └── cats.gif
|   │
|   └── dog_breeds.txt  ← Accessing this file
|
└── animals.csv
```

Files

A file operation takes place in the following order:

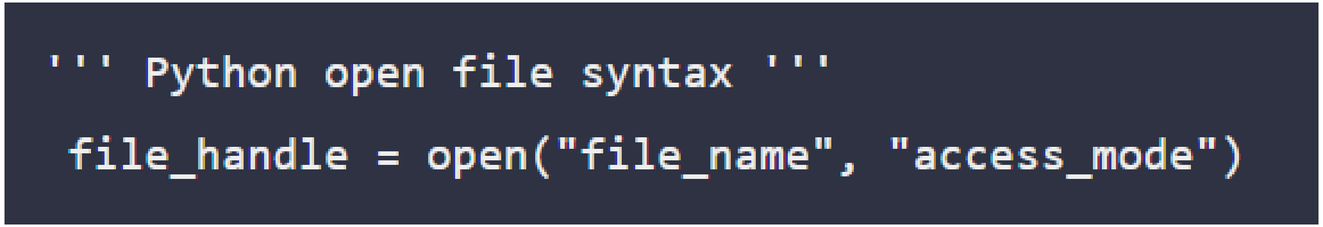
- Open a file
- Read or write (perform operation)
- Close the file



Opening Files in Python

- Python has a built-in **open()** function to open a file.

- This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.



```
''' Python open file syntax '''
 file_handle = open("file_name", "access_mode")
```

Opening a file

- The first argument is the name or path of the file (including file name). For example – sample_log.txt or /Users/john/home/sample_log.txt.

- And the second parameter (optional) represents a mode to open the file.

- The value of the “access_mode” defines the operation you want to perform on it. The **default value is the READ only** mode.

Opening a file - example

```
# Open a file named "sample_log.txt"
# It rests in the same directory as you are working in.
file_handlel = open("sample_log.txt")

# Let's open the file from a given path
file_handle2 = open("/Users/john/home/sample_log.txt")
```

Opening a File - modes

- There are a total of eight access modes available in python.
- We can also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode. In this mode, we get strings when reading from the file.
- On the other hand, binary mode returns bytes, and this is the mode to be used when dealing with non-text files like images or executable files.

“r” – It opens a text file for reading. It keeps the offset at the start of the file. If the file is missing, then it raises an I/O error. It is also the default mode.

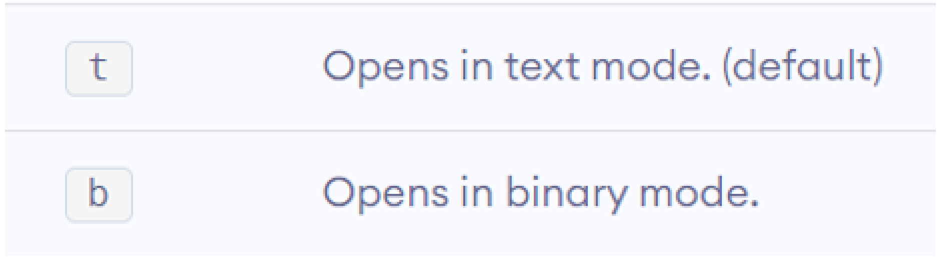
“r+” – It opens the file for both READ and WRITE operations. It sets the offset at the start of the file. An I/O error occurs for a non-existent file.

“w” – It opens a file for writing and overwrites any existing content. The handle remains at the start of the data. If the file doesn’t exist, then it creates one.

“w+” – It opens the file for both READ and WRITE operations. Rest, it works the same as the “w” mode.

“a” – It opens a file for writing or creates a new one if the file not found. The handle moves to the end (EOF). It preserves the existing content and inserts data to the end.

“a+” – It opens the file for both READ and WRITE operations. Rest, it works the same as the “a” mode.



t	Opens in text mode. (default)
b	Opens in binary mode.

Opening a File - modes

- Modes 'r+', 'w+' and 'a+' open the file for updating (reading and writing).
- Note that 'w+' truncates the file, if it exists. Hence If you wanted to read the previous data and add to it, you should use r+ instead of w+.
- For both the w+ and a+, if the file does not exist, it creates a new file for writing.
- r+ does not create a new file if it does not exist.

```
                  | r   r+   w   w+   a   a+
------------------|--------------------------
read              | +   +        +        +
write             |     +    +   +    +   +
write after seek  |     +    +   +
create            |          +   +    +   +
truncate          |          +   +
position at start | +   +    +   +
position at end   |                   +   +
```

where meanings are: (just to avoid any misinterpretation)

- read - reading from file is allowed
- write - writing to file is allowed
- create - file is created if it does not exist yet
- trunctate - during opening of the file it is made empty (all content of the file is erased)
- position at start - after file is opened, initial position is set to the start of the file
- position at end - after file is opened, initial position is set to the end of the file

Note: `a` and `a+` always append to the end of file - ignores any `seek` movements.

Examples

```
# Open a file named "sample_log.txt" in write mode
###
# It rests in the same directory as you are working in.
file_handlel = open("sample log.txt", "w")

# Open the file from a given path in append mode
file_handle2 = open("/Users/john/home/sample log.txt", "a")
```

File modes

- Python 3 added a new mode for exclusive creation so that you will not accidentally truncate or overwrite and existing file.

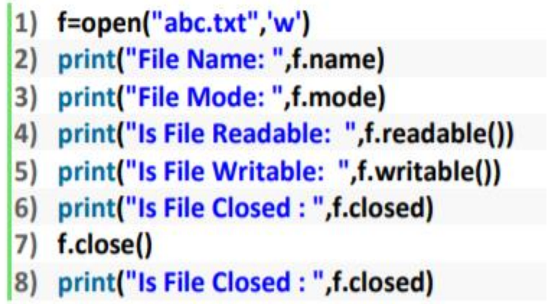
- 'x' - open for exclusive creation(writing), will **raise FileExistsError** if the file already exists.

- 'x+' - reading and writing mode. Similar to w+ as it will create a new file if the file does not exist. Otherwise, it will raise **FileExistsError**.

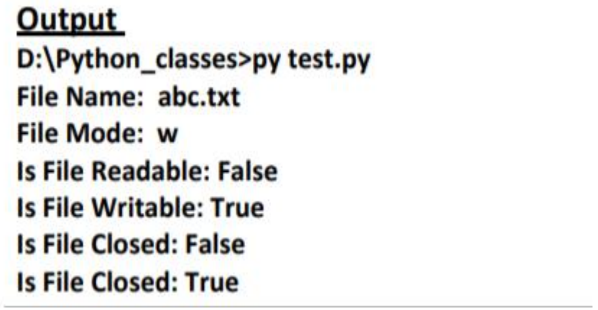
Various Properties of File Object:

- Once we open a file and we get file object, from which we can get various details related to that file by using its properties.

  - **name** - Name of opened file
  - **mode** - Mode in which the file is opened
  - **closed** - Returns boolean value indicates that file is closed or not
  - **readable()** - Returns boolean value indicates that whether file is readable or not
  - **writable()** - Returns boolean value indicates that whether file is writable or not.



```
f=open("abc.txt",'w')
print("File Name: ",f.name)
print("File Mode: ",f.mode)
print("Is File Readable:  ",f.readable())
print("Is File Writable:  ",f.writable())
print("Is File Closed : ",f.closed)
f.close()
print("Is File Closed : ",f.closed)
```



Output

```
D:\Python_classes>py test.py
File Name:  abc.txt
File Mode:  w
Is File Readable: False
Is File Writable: True
Is File Closed: False
Is File Closed: True
```

Writing to Files in Python

- In order to write into a file in Python, we need to open it in write w, append a or exclusive creation x mode.

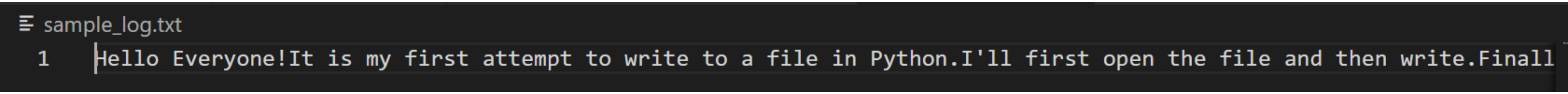
- We need to be careful with the w mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

- Python provides two functions to write into a text file: **write() and writelines().**

write()

- 1. write() – Let’s first use write() for writing to a file in Python. This function puts the given text in a single line.

```
# A simple example - Python write file
file_handle = open("sample_log.txt","w")

file_handle.write("Hello Everyone!")
file_handle.write("It is my first attempt to write to a file in Python.")
file_handle.write("I'll first open the file and then write.")
file_handle.write("Finally, I'll close it.")

file_handle.close()
```



sample_log.txt

```
Hello Everyone!It is my first attempt to write to a file in Python.I'll first open the file and then write.Finall
```

Writing to Files

- This program will create a new file if it does not exist. If it does exist, it is overwritten.
- We must include the newline characters ourselves to distinguish the different lines.
- In the previous example, we’ve opened the file in “w” mode, which means to overwrite anything written previously.

writelines()

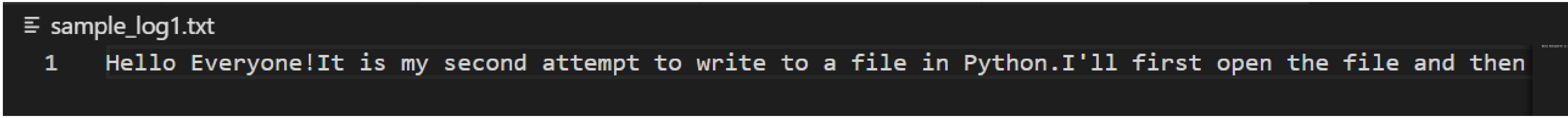
- **2. writelines() –** The writelines() function takes a list of strings as the input and inserts each of them in a file in one go.

- You can check its syntax below:

```
''' Python writelines() function'''
file_handle.writelines([strl, str2, str3, ...])
```

Example

```
# A simple example - Python write file

file_handle = open("sample_logl.txt","w")

file_handle.writelines(["Hello Everyone!","It is my second attempt to write to a file in Python.", "I'll first open the file and then write.","Finally, I'll close it."])

file_handle.close()
```

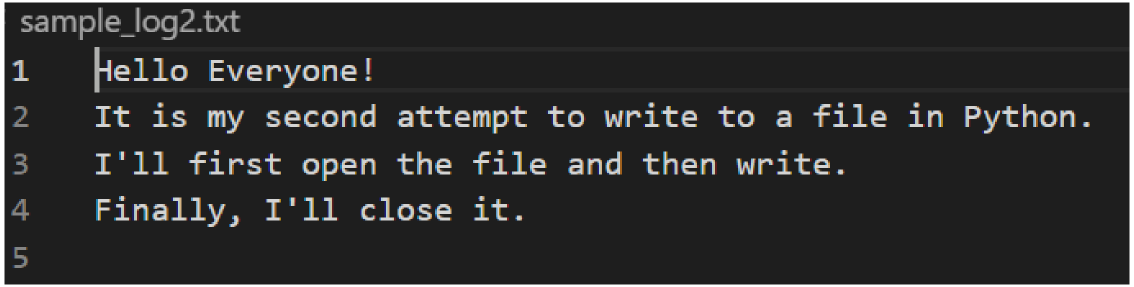


sample_log1.txt

```
Hello Everyone!It is my second attempt to write to a file in Python.I'll first open the file and then
```

Example

```
# A simple example - Python write file

file_handle = open("sample_log2.txt","w")

file_handle.writelines(["Hello Everyone!\n","It is my second attempt to write to a file in Python.\n ", "I'll first open the file and then write.\n","Finally, I'll close it.\n"])

file_handle.close()
```



sample_log2.txt

```
Hello Everyone!
It is my second attempt to write to a file in Python.
I'll first open the file and then write.
Finally, I'll close it.

```

Closing Files in Python

- When we are done with performing operations on the file, we need to properly close the file.

- Closing a file will free up the resources that were tied with the file. It is done using the close() method available in Python.

- Python has a garbage collector to clean up unreferenced objects, but we must not rely on it to close the file.

Closing a file

- This method of closing a file, as shown in the previous examples are not entirely safe.

- If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

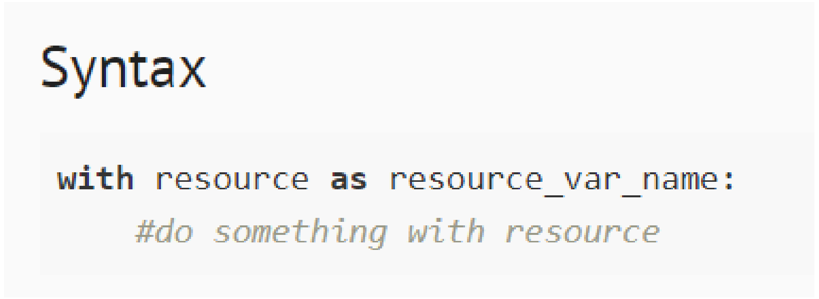
- A safer way is to use a **try...finally** block.

- This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

With statement

- If you want a cleaner and elegant way to write to a file in Python, then try using the **WITH** statement. It does the automatic clean up of the system resources like file handles.

- Also, it provides out of the box exception handling (**Python try-except**) while you are working with the files.



Syntax

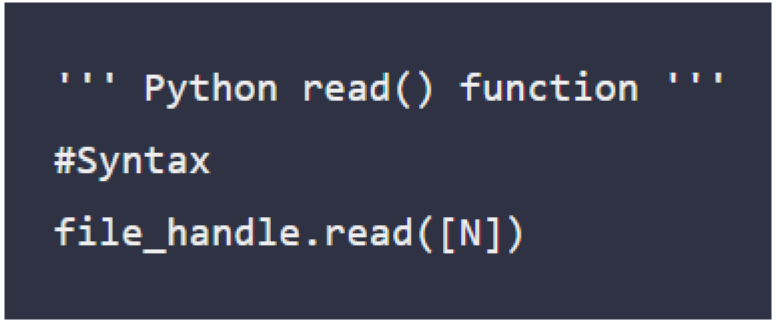
```
with resource as resource_var_name:
    #do something with resource
```

```
# Write File in Python using WITH statement
##
# Sample code(1) Write to a text file
fh = open("sample_log.txt", "w")

try:
        fh.write("I love Python Programming!")
finally:
        fh.close()


# Sample code(2) Write using with statement
with open("sample_log.txt", "w") as fh:
        fh.write("I love Python even morel!")
```

read functions to read a file

For reading a text file, Python bundles the following three functions: ***read()***, ***readline()***, and ***readlines()***

- **1. read()** – It reads the given no. of bytes (N) as a string. If no value is given, then it reads the file till the EOF.



```
''' Python read() function '''
#Syntax
file_handle.read([N])
```

read functions to read a file

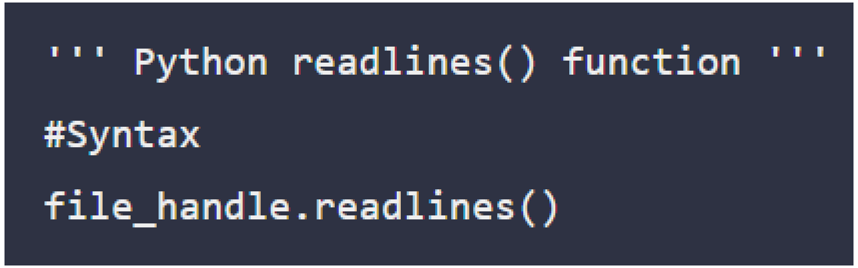
- **2. readline()** – It reads the specified no. of bytes (N) as a string from a single line in the file.

- It restricts to one line per call even if N is more than the bytes available in one line.

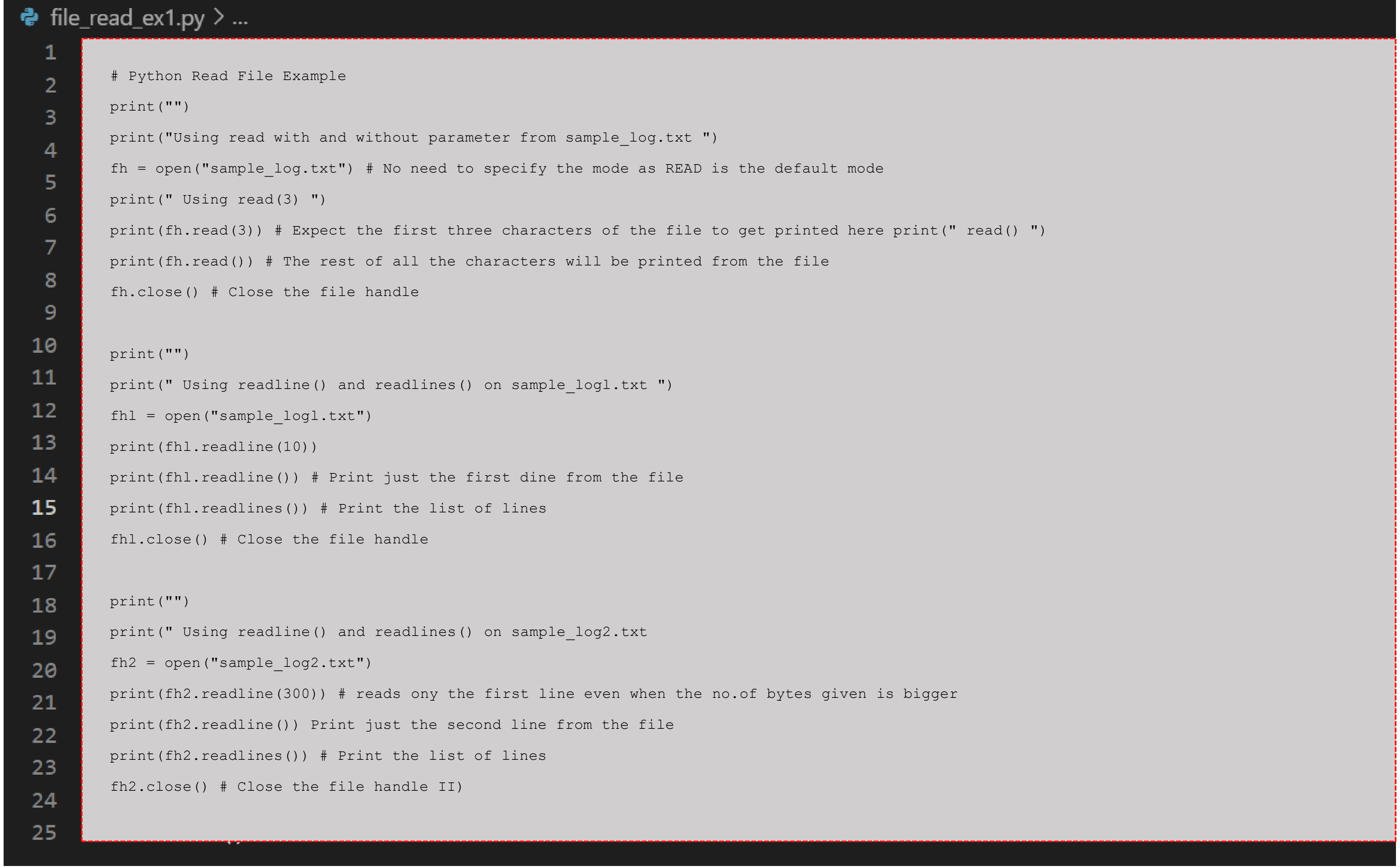
```
''' Python readline() function '''
#Syntax
file_handle.readline([N])
```

Read functions to read a file

- **3. readlines() –** It reads every line present in the text file and returns them as a list of strings.

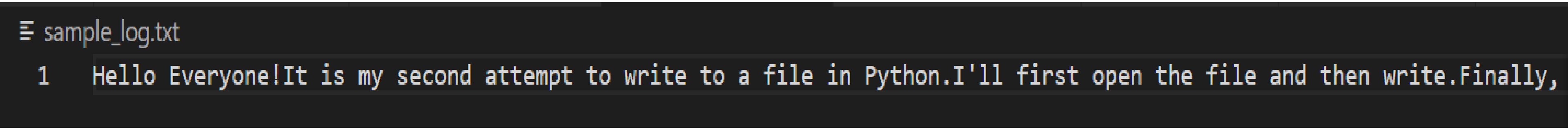


```
''' Python readlines() function '''
#Syntax
file_handle.readlines()
```



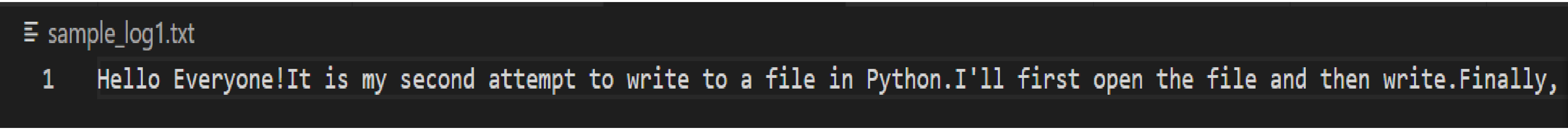
file_read_ex1.py > ...

```python
# Python Read File Example
print("")
print("Using read with and without parameter from sample_log.txt ")
fh = open("sample_log.txt") # No need to specify the mode as READ is the default mode
print(" Using read(3) ")
print(fh.read(3)) # Expect the first three characters of the file to get printed here print(" read() ")
print(fh.read()) # The rest of all the characters will be printed from the file
fh.close() # Close the file handle

print("")
print(" Using readline() and readlines() on sample_logl.txt ")
fhl = open("sample_logl.txt")
print(fhl.readline(10))
print(fhl.readline()) # Print just the first dine from the file
print(fhl.readlines()) # Print the list of lines
fhl.close() # Close the file handle

print("")
print(" Using readline() and readlines() on sample_log2.txt
fh2 = open("sample_log2.txt")
print(fh2.readline(300)) # reads ony the first line even when the no.of bytes given is bigger
print(fh2.readline()) Print just the second line from the file
print(fh2.readlines()) # Print the list of lines
fh2.close() # Close the file handle II)
```



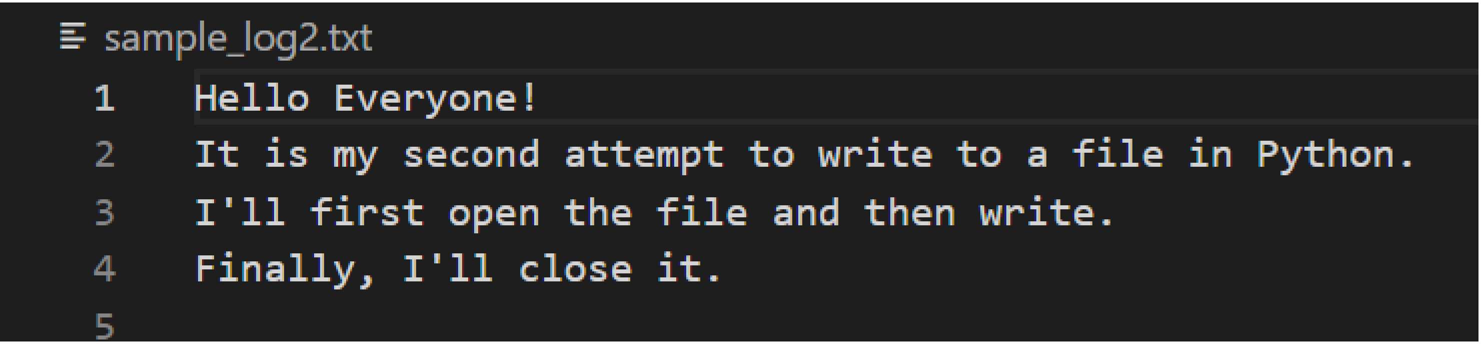
sample_log.txt

```
Hello Everyone!It is my second attempt to write to a file in Python.I'll first open the file and then write.Finally,
```



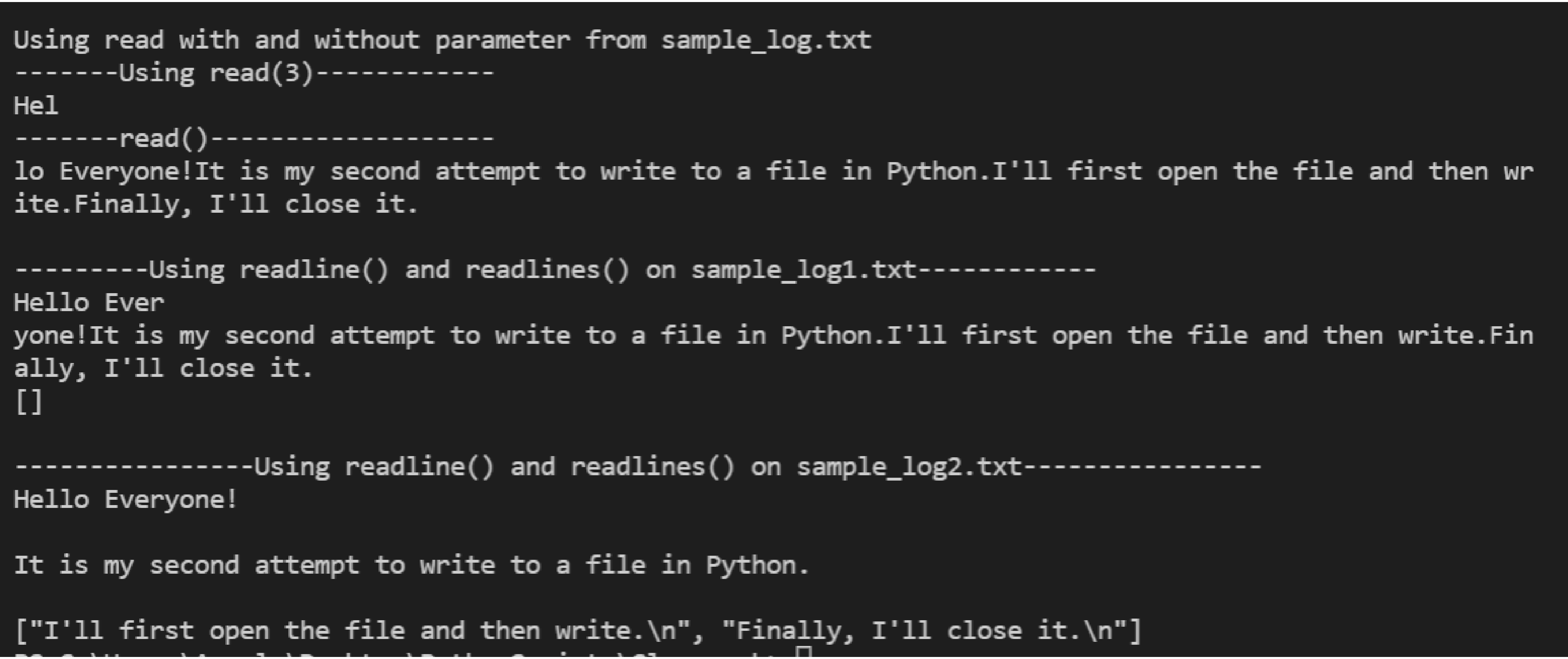
sample_log1.txt

```
Hello Everyone!It is my second attempt to write to a file in Python.I'll first open the file and then write.Finally,
```



sample_log2.txt

```
Hello Everyone!
It is my second attempt to write to a file in Python.
I'll first open the file and then write.
Finally, I'll close it.

```



```
Using read with and without parameter from sample_log.txt
-------Using read(3)------------
Hel
-------read()-------------------
lo Everyone!It is my second attempt to write to a file in Python.I'll first open the file and then wr
ite.Finally, I'll close it.

---------Using readline() and readlines() on sample_log1.txt------------
Hello Ever
yone!It is my second attempt to write to a file in Python.I'll first open the file and then write.Fin
ally, I'll close it.
[]

----------------Using readline() and readlines() on sample_log2.txt----------------
Hello Everyone!

It is my second attempt to write to a file in Python.

["I'll first open the file and then write.\n", "Finally, I'll close it.\n"]
```

seek and tell Functions

- We know that when you use the open function to open a file and read the contents of the file, it always starts with the first character (byte) of the file.

- So, is there a way to specify the starting position for reading? The answer is yes, this requires moving the position of the file pointer.

- By moving the position of the file pointer and then using the read and write functions, it can be easily implemented to read the data at the specified position in the file (or write data to the specified position in the file).

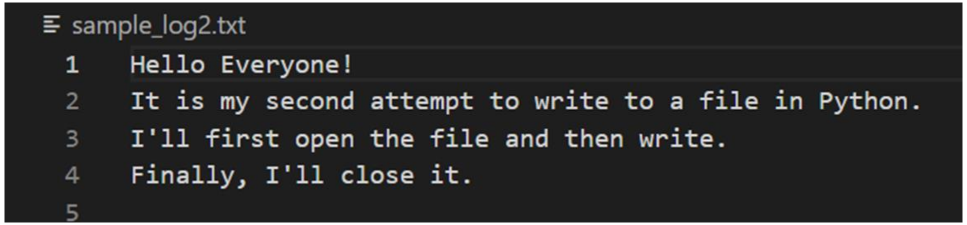
seek and tell Functions

- To implement the movement of the file pointer, the file object provides a tell function and a seek function.
- The **tell function is used to determine the current position of the file pointer.**
- The **seek function is used to move the file pointer to the specified position of the file.**

The seek() and tell() Methods:

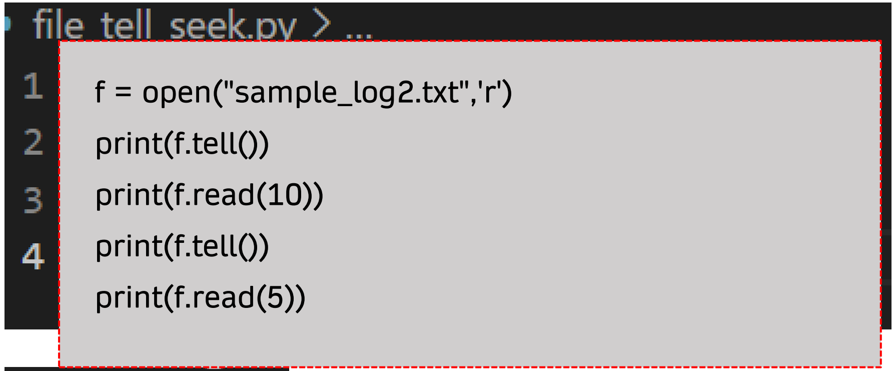
- When the file is opened using the open function, the starting position of the file pointer is 0, which means that it is located at the beginning of the file.

- After reading 10 characters from the file using the read function, the file pointer moves at the same time, and is moved by 10 characters.



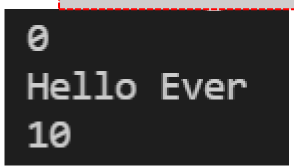
sample_log2.txt

```
Hello Everyone!
It is my second attempt to write to a file in Python.
I'll first open the file and then write.
Finally, I'll close it.

```



file tell seek.py > ...

```python
f = open("sample_log2.txt",'r')
print(f.tell())
print(f.read(10))
print(f.tell())
print(f.read(5))
```



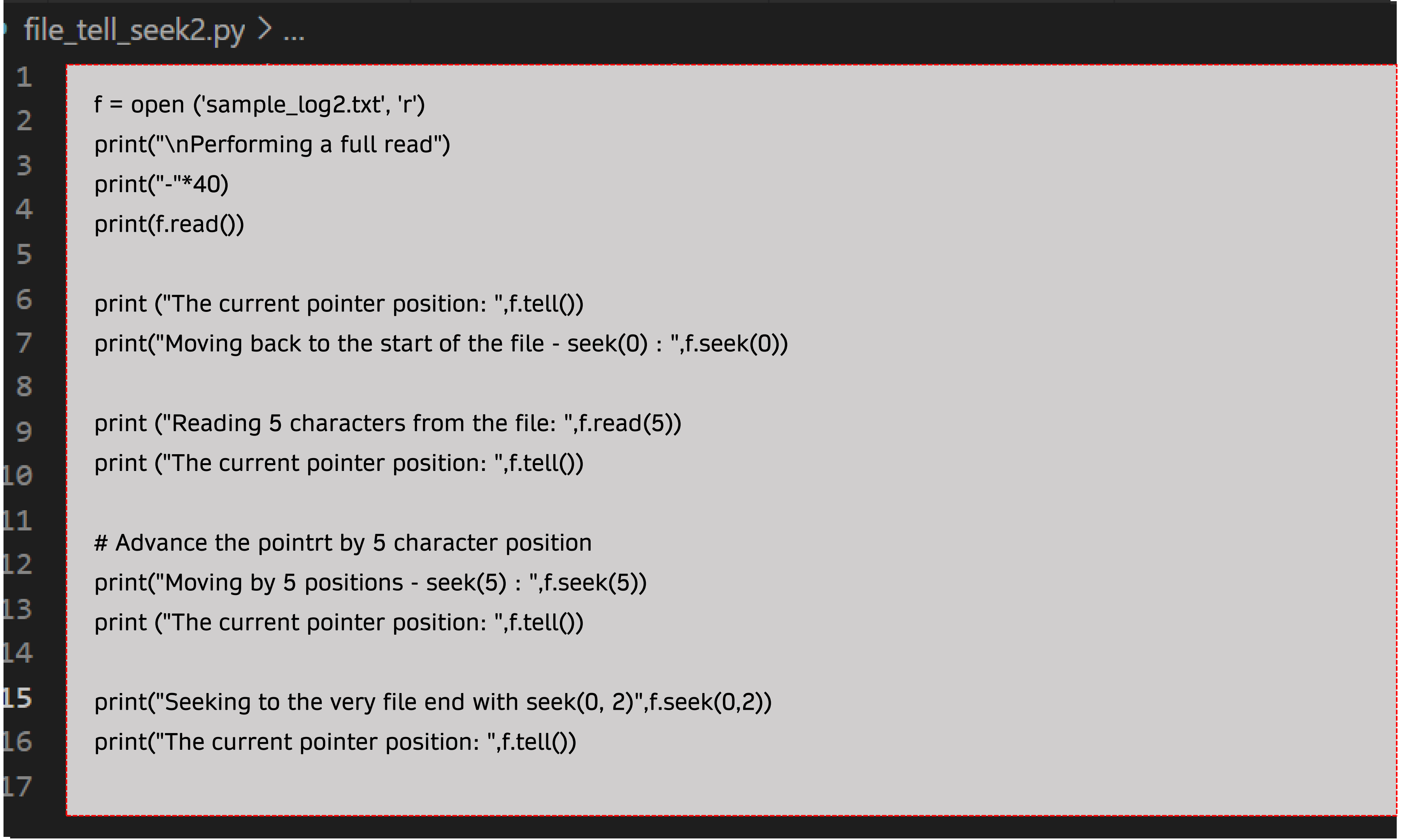
```
0
Hello Ever
10
```

The seek() and tell() Methods:

- To change the file object’s position, use **f.seek(offset, whence)**. The position is computed from adding offset to a reference point; the reference point is selected by the whence argument.

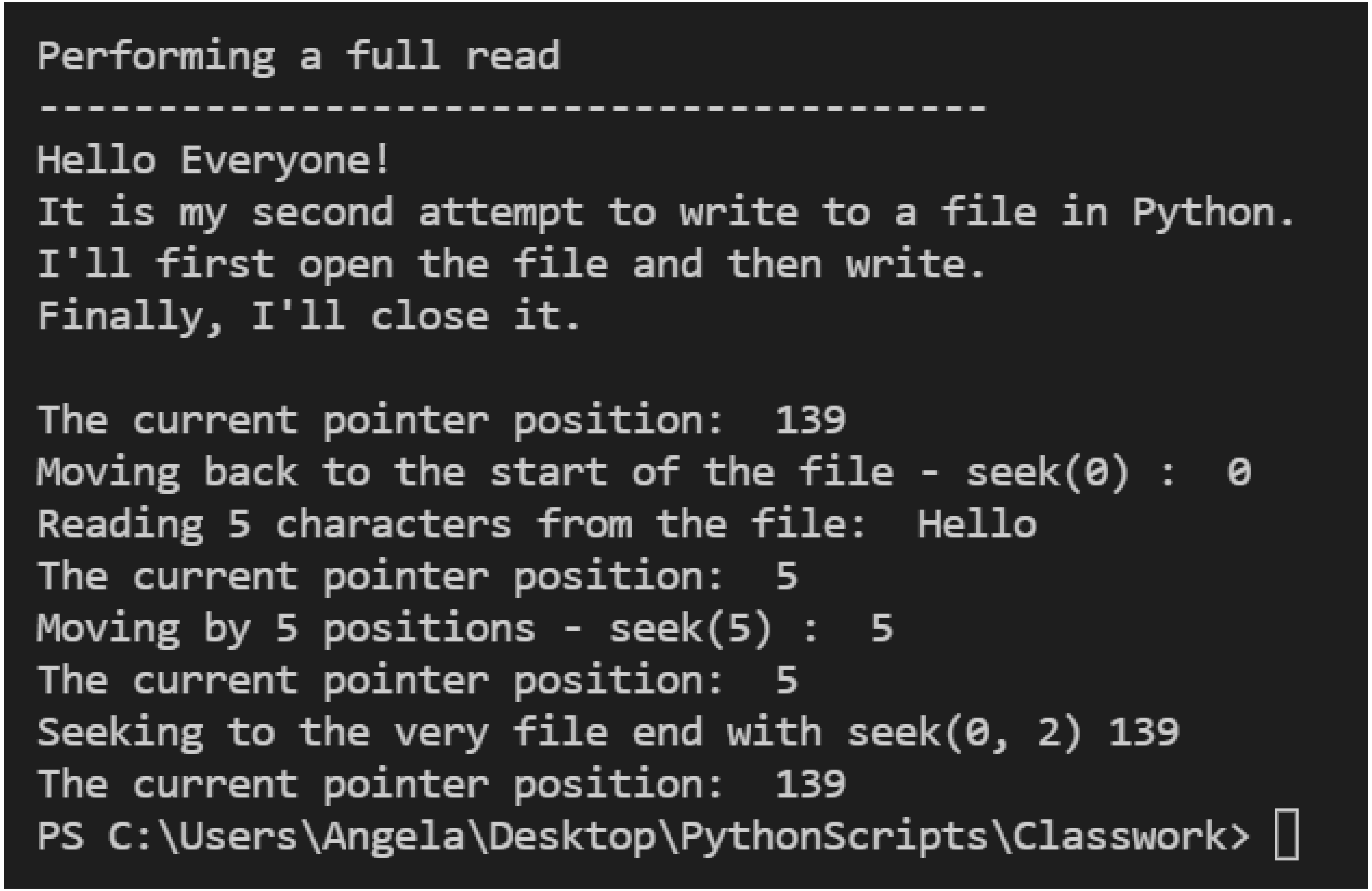
- A whence value of ***0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point.***

- whence can be omitted and defaults to 0, using the beginning of the file as the reference point. **In text files (those opened without a b in the mode string), only seeks relative to the beginning of the file are allowed** (the exception being seeking to the very file end with **seek(0, 2))**



file_tell_seek2.py > ...

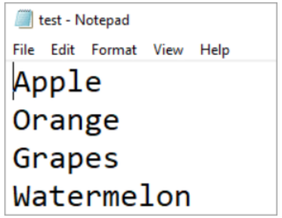
```
f = open ('sample_log2.txt', 'r')
print("\nPerforming a full read")
print("-"*40)
print(f.read())

print ("The current pointer position: ",f.tell())
print("Moving back to the start of the file - seek(0) : ",f.seek(0))

print ("Reading 5 characters from the file: ",f.read(5))
print ("The current pointer position: ",f.tell())

# Advance the pointrt by 5 character position
print("Moving by 5 positions - seek(5) : ",f.seek(5))
print ("The current pointer position: ",f.tell())

print("Seeking to the very file end with seek(0, 2)",f.seek(0,2))
print("The current pointer position: ",f.tell())
```



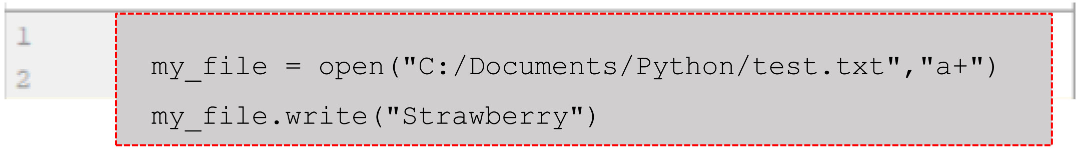
```
Performing a full read
----------------------------------------
Hello Everyone!
It is my second attempt to write to a file in Python.
I'll first open the file and then write.
Finally, I'll close it.

The current pointer position:  139
Moving back to the start of the file - seek(0) :  0
Reading 5 characters from the file:  Hello
The current pointer position:  5
Moving by 5 positions - seek(5) :  5
The current pointer position:  5
Seeking to the very file end with seek(0, 2) 139
The current pointer position:  139
PS C:\Users\Angela\Desktop\PythonScripts\Classwork> 
```

Appending to a File

- Sometimes, you may want to append to a file or start writing at the end of an already populated file. This is easily done by using the 'a' character for the mode argument:

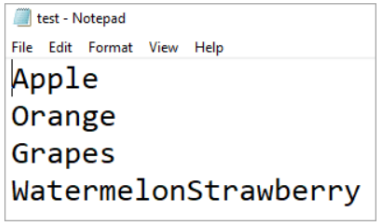


```
Apple
Orange
Grapes
Watermelon
```

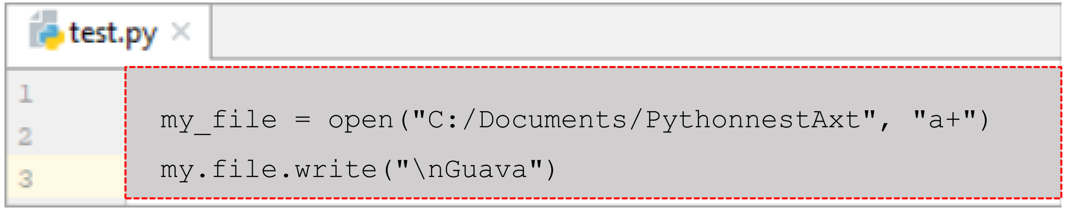


```
my_file = open("C:/Documents/Python/test.txt","a+")
my_file.write("Strawberry")
```

Output:

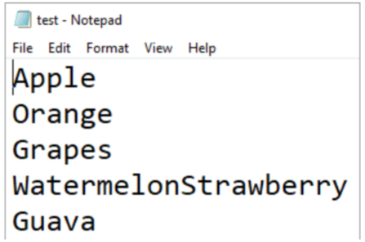


```
Apple
Orange
Grapes
WatermelonStrawberry
```



```
my_file = open("C:/Documents/PythonnestAxt", "a+")
my.file.write("\nGuava")
```

Output:

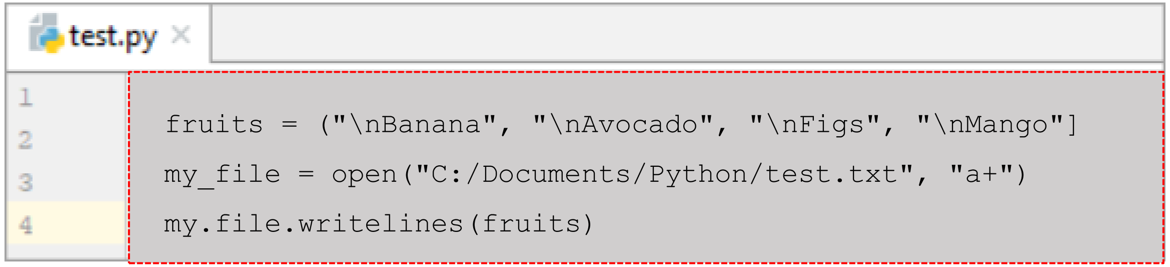


```
Apple
Orange
Grapes
WatermelonStrawberry
Guava
```

Appending to a File

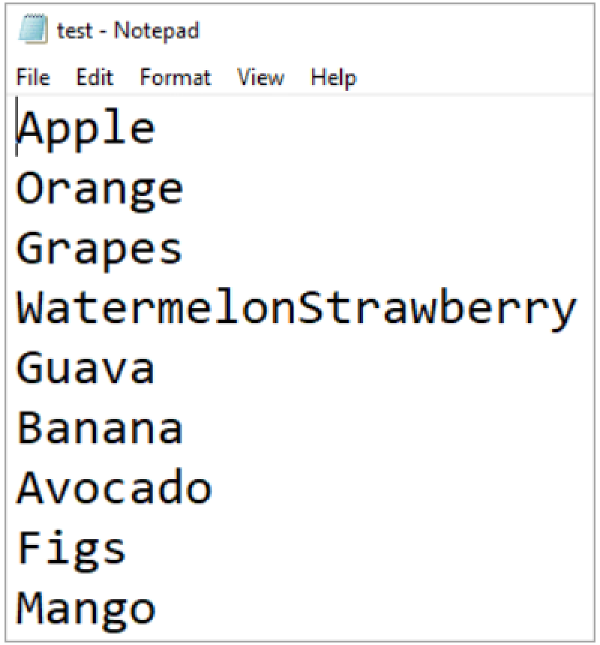
```
fruits = ["\nBanana", "\nAvocado", "\nFigs", "\nMango"]
my_file = open("C:/Documents/Python/test.txt", "a+")
my_file.writelines(fruits)
```

The above code **appends a list of data** into a 'test.txt' file.


test.py

```
fruits = ("\nBanana", "\nAvocado", "\nFigs", "\nMango"]
my_file = open("C:/Documents/Python/test.txt", "a+")
my.file.writelines(fruits)
```

Output:


test - Notepad

```
Apple
Orange
Grapes
WatermelonStrawberry
Guava
Banana
Avocado
Figs
Mango
```

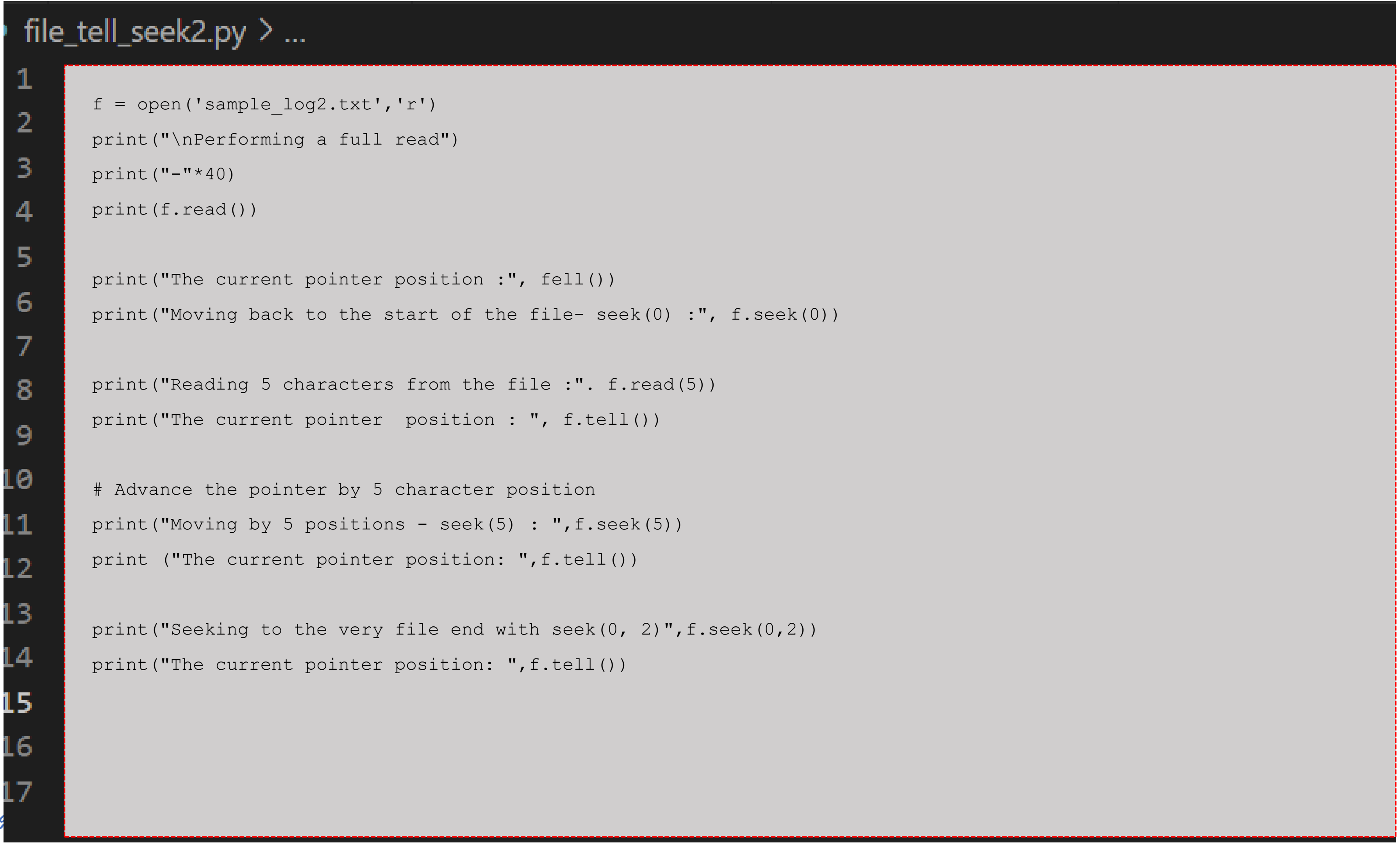
Appending to a File

- If we add a "+" to the mode argument of the open method, we can open the file for both appending and reading. This will enable both reading and writing to the file. Without the "+" an IOError exception will occur if we try and read from the file.

- By default, both reading and writing will occur at the end of the file. **Seek method does not have any effect while appending (Line 19)**

- Here are the commands to achieve this (note that we use the flush method to ensure the new content is written to the file before we try to read it back):

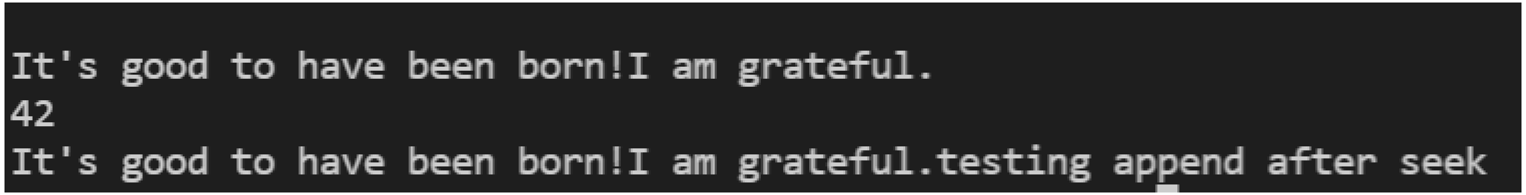
```
#open a file in append mode
f = open("helloworld.txt", "a")
f.write("It's good to have been born!")
f.close()

# open a file in read and write(append) mode
f1 = open("helloworld.txt", "a+") f1.write("I am grateful.")
content = f1.read()            # read and print does not output anything as the file pointer
                               # is at the end of the file after appending
print (content)
f1.flush()
f1.seek(0)                     # move the file pointer to the beginning of the file
content = f1.read()
print (content)
print(f1.tell())
f1.seek(0)
f1.write("testing append after seek")

f1.seek(0)
content = f1.read()
print(content)
f1.close()
```



file_tell_seek2.py > ...

```python
f = open('sample_log2.txt','r')
print("\nPerforming a full read")
print("-"*40)
print(f.read())

print("The current pointer position :", fell())
print("Moving back to the start of the file- seek(0) :", f.seek(0))

print("Reading 5 characters from the file :". f.read(5))
print("The current pointer  position : ", f.tell())

# Advance the pointer by 5 character position
print("Moving by 5 positions - seek(5) : ",f.seek(5))
print ("The current pointer position: ",f.tell())

print("Seeking to the very file end with seek(0, 2)",f.seek(0,2))
print("The current pointer position: ",f.tell())
```

Output



```
It's good to have been born!I am grateful.
42
It's good to have been born!I am grateful.testing append after seek
```

Types Of File

There are two types of files in Python, and they are:

- Binary file
- Text file

Types Of File

- All binary files follow a specific format. We can open some binary files in the normal text editor, but we can’t read the content present inside the file. That’s because all the binary files will be encoded in the binary format, which can be understood only by a computer or machine.

- For handling such binary files, we need a specific type of software to open it.

- For Example, you need a pdf reader software to open .pdf binary files and you need a photo editor software to read the image files and so on.

Text files in Python

Text files don’t have any specific encoding and it can be opened in normal text editor itself.
Example:

- Web standards: html, XML, CSS, JSON etc.
- Source code: c, cpp, js, py, java etc.
- Documents: txt, tex, RTF etc.
- Tabular data: csv, tsv etc.

Modes – Binary files

- While using binary files, we have to use the same modes with the letter 'b' at the end. So that Python can understand that we are interacting with binary files.

- 'wb' – Open a file for write only mode in the binary format.
- 'rb' – Open a file for the read-only mode in the binary format.
- 'ab' – Open a file for appending only mode in the binary format.
- 'rb+' – Open a file for read and write mode in the binary format.
- 'ab+' – Open a file for appending and read mode in the binary format.
- 'wb+' – Open a file for write and read mode in the binary format.

Opening a binary file

Example 1:

```
fo = open(“C:/Documents/Python/test.txt”, “r+”)
```

In the above example, we are opening the file named ‘test.txt’ present at the location ‘C:/Documents/Python/’ and we are opening the same file in a read-write mode which gives us more flexibility.

Example 2:

```
fo = open(“C:/Documents/Python/img.bmp”, “rb+”)
```

In the above example, we are opening the file named ‘img.bmp’ present at the location “C:/Documents/Python/”, But, here we are trying to open the binary file.

Working With Two Files at the Same Time

- There are times when you may want to read a file and write to another file at the same time. If you use the example that was shown when you were learning how to write to a file, it can actually be combined into the following:

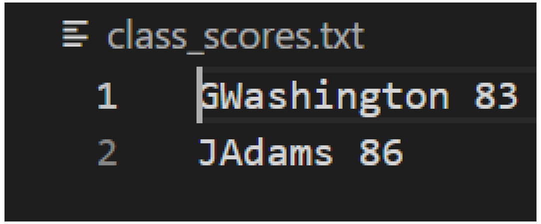
```
d_path = 'dog_breeds.txt'
d _ r _path = 'dog_breeds reversed.txt’

with open(d_path, 'r') as reader, open(d_r_path, 'w') as writer:
        dog_breeds = reader.readlines()
        writer.writelines(reversed(dog_breeds))
```

Exercise - 1

Write a file called class_scores.txt, where each line of the file contains a one-word username and a test score separated by spaces, like below:

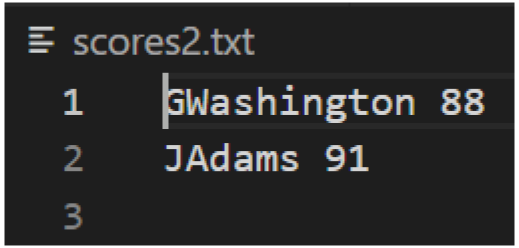
- GWashington 83
- JAdams 86

Write code that scans through the file, adds 5 points to each test score, and outputs the usernames and new test scores to a new file, scores2.txt



class_scores.txt

```
GWashington 83
JAdams 86
```



scores2.txt

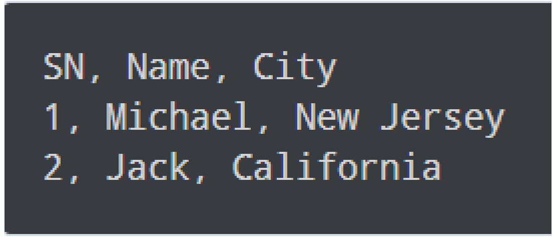
```
GWashington 88
JAdams 91

```

Reading and Writing CSV Files

A CSV (Comma Separated Values) format is one of the most simple and common ways to store tabular data. To represent a CSV file, it must be saved with the **.csv** file extension.

Let's take an example:

- If you open the above CSV file using a text editor, you will see:



```
SN, Name, City
1, Michael, New Jersey
2, Jack, California
```

What Is a CSV File?

- Notice how each piece of data is separated by a comma. Normally, the first line identifies each piece of data—in other words, the name of a data column. Every subsequent line after that is actual data and is limited only by file size constraints.

- In general, the separator character is called a delimiter, and the comma is not the only one used. Other popular delimiters include the tab (\t), colon (:) and semi-colon (;) characters. Properly parsing a CSV file requires us to know which delimiter is being used.

Parsing CSV Files With Python's Built-in CSV Library

- The csv module implements classes to read and write tabular data in CSV format.

- It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel.

- Before we can use the methods to the csv module, we need to import the module first using:

```
import csv
```

Writing CSV files Using csv.writer()

- To **write to a CSV file in Python, we can use the csv.writer() function.**
- The csv.writer() function returns a writer object.
- The writer.writerow() function is then used to write single rows to the CSV file.
- Let's see an example.

Example 1: Write into CSV files with csv.writer()

Suppose we want to write a CSV file with the following entries:

```
SN,Name,Contribution
1,Linus Torvalds,Linux Kernel
2,Tim Berners-Lee,World Wide Web
3,Guido van Rossum,Python Programming
```

```
import csv
with open('innovators.csv', 'w', newline=' ') as file:
        writer = csv.writer(file)
        writer.writerow(["SN", "Name", "Contribution"])
        writer.writerow([1, "Linus Torvalds", "Linux Kernel"])
        writer.writerow([2, "Tim Berners-Lee", "World Wide Web"])
        writer.writerow([3, "Guido van Rossum", "Python Programming"])
```

Example 2: Writing Multiple Rows with writerows()

If we need to write the contents of the 2-dimensional list to a CSV file, here's how we can do it.

```
import csv
row list = [["SN", "Name", "Contribution"],
                         [1, "Linus Torvalds", "Linux Kernel"],
                         [2, "Tim Berners-Lee", "World Wide Web"],
                         [3, "Guido van Rossum", "Python Programming"]]

with open('protagonist.csv', 'w', newline='') as file:
            writer = csv.writer(file)
            writer.writerows(row_list)
```

The output of the program is the same as in **Example 1.**

Here, our 2-dimensional list is passed to the `writer.writerows()` function to write the content of the list to the CSV file.

CSV Files with Custom Delimiters

- By default, a comma is used as a delimiter in a CSV file.
- However, some CSV files can use delimiters other than a comma. Few popular ones are | and \t.
- Suppose we want to use | as a delimiter in the innovators.csv file of Example 1. To write this file, we can pass an additional delimiter parameter to the csv.writer() function.

Example 3: Write CSV File Having Pipe Delimiter

```
import csv
data_list = [["SN", "Name", "Contribution"],
                    [1, "Linus Torvalds", "Linux Kernel"],
                    [2, "Tim Berners-Lee", "World Wide Web"],
                    [3, "Guido van Rossum", "Python Programming"]]

with open('innovators.csvi, 'w', newline=") as file:
         writer = csv.writer(file, delimiter=‘|’)
         writer.writerows(data_list)
```

Output

```
SN|Name|Contribution
1|Linus Torvalds|Linux Kernel
2|Tim Berners-Lee|World Wide Web
3|Guido van Rossum|Python Programming
```

As we can see, the optional parameter `delimiter = '|'` helps specify the `writer` object that the CSV file should have `|` as a delimiter.

Reading CSV Files With csv

- Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python's built-in open() function, which returns a file object.

- To read a CSV file in Python, we can use the csv.reader() function.

- This is then passed to the reader, which does the heavy lifting.

Suppose we have a CSV file with the following entries:

```
SN,Name,Contribution
1,Linus Torvalds,Linux Kernel
2,Tim Berners-Lee,World Wide Web
3,Guido van Rossum,Python Programming
```

We can read the contents of the file with the following program:

```
import csv
with open('innovators.csv', 'r') as file:
        reader = csv.reader(file)
        for row in reader:
                print(row)
```

Output

```
['SN', 'Name', 'Contribution']
['1', 'Linus Torvalds', 'Linux Kernel']
['2', 'Tim Berners-Lee', 'World Wide Web']
['3', 'Guido van Rossum', 'Python Programming']
```

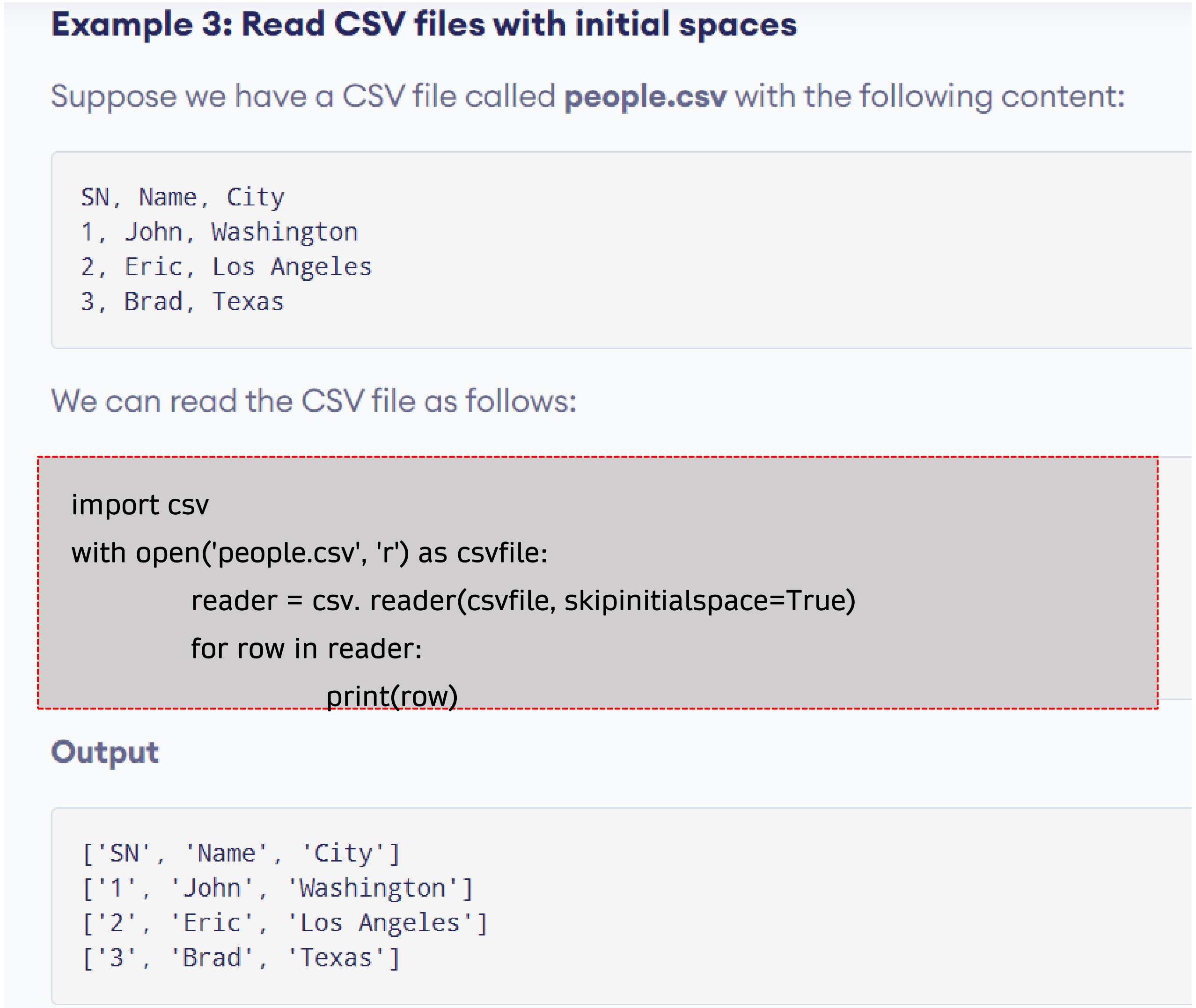
Example 2: Read CSV file Having Tab Delimiter

```
import csv
with open('innovators.csv’, 'r') as file:
          reader = csv.reader(file, delimiter = '\t')
          for row in reader:
                    print(row)
```

Output

```
['SN', 'Name', 'Contribution']
['1', 'Linus Torvalds', 'Linux Kernel']
['2', 'Tim Berners-Lee', 'World Wide Web']
['3', 'Guido van Rossum', 'Python Programming']
```

As we can see, the optional parameter `delimiter = '\t'` helps specify the `reader` object that the CSV file we are reading from, has **tabs** as a delimiter.



Example 3: Read CSV files with initial spaces

Suppose we have a CSV file called **people.csv** with the following content:

```
SN, Name, City
1, John, Washington
2, Eric, Los Angeles
3, Brad, Texas
```

We can read the CSV file as follows:

```
import csv
with open('people.csv', 'r') as csvfile:
        reader = csv. reader(csvfile, skipinitialspace=True)
        for row in reader:
                print(row)
```

Output

```
['SN', 'Name', 'City']
['1', 'John', 'Washington']
['2', 'Eric', 'Los Angeles']
['3', 'Brad', 'Texas']
```

Example 4: Read CSV files with quotes

```
import csv
with open('personl.csv', 'r') as file:
        reader = csv.reader(file, quoting=csv.QUOTE_ALL, skipinitialspace=True)
        for row in reader:
                print(row)
```

Output

```
['SN', 'Name', 'Quotes']
['1', 'Buddha', 'What we think we become']
['2', 'Mark Twain', 'Never regret anything that made you smile']
['3', 'Oscar Wilde', 'Be yourself everyone else is already taken']
```

https://docs.python.org/3/librar
y/csv.html

Angela