



PYTHON



FILE HANDLING



Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.



The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character

Hence, a file operation can be done in the following order.

1. Open a file
2. Read or write - Performing operation
3. Close the file

OPENING A FILE

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

```
file object = open(<file-name>, <access-mode>, <buffering>)
```

FILE MODE TABLE

Sr.	Mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
3	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
Note	b	Treated as binary file (rb = read binary, wb= write binary, ab = append binary)
	+	For read and write both
	b+	Binary file read and write




```
#opens the file file.txt in read mode
```

```
fileptr = open("file.txt","r")
```

```
if fileptr:
```

```
    print("file is opened successfully")
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

THE CLOSE() METHOD

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

```
fileobject.close()
```



```
# opens the file file.txt in read mode
```


```
fileptr = open("file.txt","r")
```

```
if fileptr:
```

```
    print("file is opened successfully")
```

```
#closes the opened file
```

```
fileptr.close()
```



After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.



We should use the following method to overcome such type of problem.

try:

```
fileptr = open("file.txt")
```

```
# perform file operations
```

finally:

```
fileptr.close()
```


THE WITH STATEMENT

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.


```
with open(<file name>, <access mode>) as <file-pointer>:
```

```
    #statement suite
```



The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.




```
with open("file.txt",'r') as f:  
    content = f.read();  
    print(content)
```


WRITING THE FILE

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.



```
# open the file.txt in append mode. Create a new file if no such file exists.  
fileptr = open("file2.txt", "w")  
  
# appending the content to the file  
fileptr.write("""Python is the modern day language. It makes things so simple.  
It is the fastest-growing programming language""")  
  
# closing the opened the file  
fileptr.close()
```



We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.



```
#open the file.txt in write mode.
```


```
fileptr = open("file2.txt","a")
```

```
#overwriting the content of the file
```

```
fileptr.write(" Python has an easy syntax and user-friendly interaction.")
```


```
#closing the opened file
```

```
fileptr.close()
```



We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.



The syntax of the **read()** method is given below.

fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.



#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r")


#stores all the data of the file into the variable content

content = fileptr.read(10)

print(type(content)) # prints the type of the data stored in the file

print(content) #prints the content of the file

fileptr.close() #closes the opened file



In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

```
content = fileptr.read()
```

```
print(content)
```


READ FILE THROUGH FOR LOOP

#open the file.txt in read mode. causes an error if no such file exists.

```
fileptr = open("file2.txt","r");
```

#running a for loop

```
for i in fileptr:
```

```
    print(i) # i contains each line of the file
```

READ LINES OF THE FILE

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the **readline()** method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file "**file2.txt**" containing three lines. Consider the following example.



#open the file.txt in read mode. causes error if no such file exists.

```
fileptr = open("file2.txt","r");
```

#stores all the data of the file into the variable content

```
content = fileptr.readline()
```

```
content1 = fileptr.readline()
```

```
print(content)    #prints the content of the file
```


```
print(content1)
```

```
fileptr.close()   #closes the opened file
```



We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.



```
#open the file.txt in read mode. causes error if no such file exists.  
fileptr = open("file2.txt","r");  
content = fileptr.readlines()    #stores all the data of the file into content  
print(content)    #prints the content of the file  
fileptr.close()    #closes the opened file
```

CREATING A NEW FILE

The new file can be created by using one of the following access modes with the function `open()`.

1. **x**: it creates a new file with the specified name. It causes an error a file exists with the same name.
2. **a**: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.
3. **w**: It creates a new file with the specified name if no such file exists. It overwrites the existing file



#open the file.txt in read mode. causes error if no such file exists.

```
fileptr = open("file2.txt","x")
```


```
print(fileptr)
```

```
if fileptr:
```

```
    print("File created successfully")
```

FILE POINTER POSITIONS

Python provides the `tell()` method which is used to print the byte number at which the file pointer currently exists. Consider the following example.



```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")
print("The filepointer is at byte :",fileptr.tell())
content = fileptr.read();
print("After reading, the filepointer is at:",fileptr.tell())
```

MODIFYING FILE POINTER POSITION

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the `seek()` method which enables us to modify the file pointer position externally.


The syntax to use the `seek()` method is given below.

```
<file-ptr>.seek(offset[, from])
```



The `seek()` method accepts two parameters:

1. **offset:** It refers to the new position of the file pointer within the file.
2. **from:** It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.



```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")
print("The filepointer is at byte :",fileptr.tell())
fileptr.seek(10);
print("After reading, the filepointer is at:",fileptr.tell())
```



CONTINUE IN NEXT UNIT