

File Handling in Python

- A file is a collection of data stored on a secondary storage device like hard disk.
- File path: Files that we use are stored on a storage medium like hard disk. You have to give the file path to open that file.

There are two types of paths:

- **Relative path:** is specified relative to the program's current working directory. **For example: file1.txt**
- **Absolute Path:** always contain root and the complete directory list to specify the exact location of the file. **For example: C:\student\documents\file1.txt**

Types of files:

- Like C, Python also supports two types of files: **text files and binary files.**
- Contents of a binary file are not human readable. If you want that data stored in the file must be human readable then store the data in a text file.
- In a text file, each line of data ends with a newline character i.e \n. Each file ends with a special character called the end-of-file (EOF) marker. Each line in a text file can have maximum of 255 characters.

Python File Handling

- Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- [Python](#) treats files differently as text or binary and this is important. Each line of code includes a sequence of characters and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

Advantages of File Handling

- **Versatility:** File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.
- **Flexibility:** File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files, etc.), and to perform different operations on files (e.g. read, write, append, etc.).

- **User-friendly:** Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.
- **Cross-platform:** Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling

- **Error-prone:** File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks:** File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity:** File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.
- **Performance:** File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.

The open() Function

This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are the parameter details –

- **file_name** – The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** – The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).

- **buffering** – If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

File Opening Modes

Following are the file opening modes –

Sr.No.	Modes & Description
1	r Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	rb Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	r+ Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	rb+ Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. C
5	w Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	b Opens the file in binary mode
7	t Opens the file in text mode (default)
8	+ open file for updating (reading and writing)

9	wb Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
10	w+ Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
11	wb+ Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
12	a Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
13	ab Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
14	a+ Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
15	ab+ Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
16	x open for exclusive creation, failing if the file already exists

Explain different file object attribute with example.

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object –

Sr.No.	Attribute & Description
1	file.closed Returns true if file is closed, false otherwise.
2	file.mode Returns access mode with which file was opened.
3	file.name Returns name of the file.
4	file.softspace Returns false if space explicitly required with print, true otherwise.

Exapmle:

```
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

This produces the following result –

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

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.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is good practice to close the file once all the operations are done.

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

Syntax

1. `fileobject.close()`

Consider the following example.

1. `# opens the file file.txt in read mode`
2. `fileptr = open("file.txt","r")`
3. `if fileptr:`
4. `print("file is opened successfully")`
5. `#closes the opened file`
6. `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.

1. `try:`
2. `fileptr = open("file.txt")`
3. `# perform file operations`
4. `finally:`
5. `fileptr.close()`

NOTE: Once the file is closed using the `close()` method, any attempt to use the file object will result in an error.

The with statement

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.

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

2. `#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.

Consider the following example.

Example

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

The `read()` method

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.

Syntax:

1. `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.

Consider the following example.

Example

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt","r")`
3. `#stores all the data of the file into the variable content`
4. `content = fileptr.read(10)`

5. `# prints the type of the data stored in the file`
6. `print(type(content))`
7. `#prints the content of the file`
8. `print(content)`
9. `#closes the opened file`
10. `fileptr.close()`

Output:

```
<class 'str'>
Python is
```

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.

1. `content = fileptr.read()`
2. `print(content)`

Output:

```
Python is the modern-day language. It makes things so simple.
It is the fastest-growing programming language Python has easy an syntax and
user-friendly interaction.
```

Read file through for loop

We can read the file using for loop. Consider the following example.

1. `#open the file.txt in read mode. causes an error if no such file exists.`
2. `fileptr = open("file2.txt","r");`
3. `#running a for loop`
4. `for i in fileptr:`
5. `print(i) # i contains each line of the file`

Output:

```
Python is the modern day language.
```

```
It makes things so simple.
```

```
Python has easy syntax and user-friendly interaction.
```


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.

Example 1: Reading lines using readline() function

1. *#open the file.txt in read mode. causes error if no such file exists.*
2. `fileptr = open("file2.txt","r");`
3. *#stores all the data of the file into the variable content*
4. `content = fileptr.readline()`
5. `content1 = fileptr.readline()`
6. *#prints the content of the file*
7. `print(content)`
8. `print(content1)`
9. *#closes the opened file*
10. `fileptr.close()`

Output:

```
Python is the modern day language.
```

```
It makes things so simple.
```

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.

Example 2: Reading Lines Using readlines() function

1. *#open the file.txt in read mode. causes error if no such file exists.*
2. `fileptr = open("file2.txt","r");`
- 3.
4. *#stores all the data of the file into the variable content*
5. `content = fileptr.readlines()`
- 6.

7. #prints the content of the file
8. **print**(content)
- 9.
10. #closes the opened file
11. fileptr.close()

Output:

```
['Python is the modern day language.\n', 'It makes things so simple.\n',  
'Python has easy syntax and user-friendly interaction.']
```

The *write()* Method

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The *write()* method does not add a newline character ('\n') to the end of the string –

Syntax

```
fileObject.write(string)
```

Here, passed parameter is the content to be written into the opened file.

```
fo = open("foo.txt", "wb")  
fo.write( "Python is a great language.\nYeah its great!!\n")  
  
# Close opened file  
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.  
Yeah its great!!
```

writelines() method:

It is used to write a list of strings. Whereas *write()* method does not accept list, number type argument only string type argument is accepted.

Example:

```
f1=open("file3.txt", "w")  
lines=["hello world", "welcome to programmming", "enjoy  
learning"]
```

```
f1.writelines(lines)
f1=open("file3.txt","r")
print(f1.read())
```

Output:

hello worldwelcome to programminglearning

append() method

Once you have stored some data in a file, you can always open file again to write more data. For appending a data, mode is used “a”. while write() overwrites existing data to write the new data in the file.

Example:

```
f=open("deepika.txt","a")
f.write("\nmy name is deepika")
f.close()
f=open("deepika.txt","r")
print(f.read())
```

Output:

```
i live in meerut
my name is Deepika
```

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.

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
- 3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
- 6.
7. #reading the content of the file
8. content = fileptr.read();

- 9.
10. #after the read operation file pointer modifies. tell() returns the location of the fileptr.
- 11.
12. **print**("After reading, the filepointer is at:",fileptr.tell())

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

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.

Syntax:

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

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

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.

Consider the following example.

Example 1

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
- 3.
4. #initially the filepointer is at 0
5. **print**("The filepointer is at byte :",fileptr.tell())

- 6.
7. `#changing the file pointer location to 10.`
8. `fileptr.seek(10);`
- 9.
10. `#tell()` returns the location of the `fileptr`.
11. `print("After reading, the filepointer is at:",fileptr.tell())`

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

Example 2

```
fo = open("foo.txt", "r+")
str = fo.read(10)
print "Read String is : ", str

# Check current position
position = fo.tell()
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10)
print "Again read String is : ", str
# Close opened file
fo.close()
```

This produces the following result –

```
Read String is : Python is
Current file position : 10
Again read String is : Python is
```

Python OS module

Renaming the file

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the `rename()` method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

Syntax:

1. `rename(current-name, new-name)`

The first argument is the current file name and the second argument is the modified name. We can change the file name by passing these two arguments.

Example 1:

1. **import** os
- 2.
3. *#rename file2.txt to file3.txt*
4. `os.rename("file2.txt", "file3.txt")`

Output:

The above code renamed current **file2.txt** to **file3.txt**

Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

1. `remove(file-name)`

Example 1

1. **import** os;
2. *#deleting the file named file3.txt*
3. `os.remove("file3.txt")`

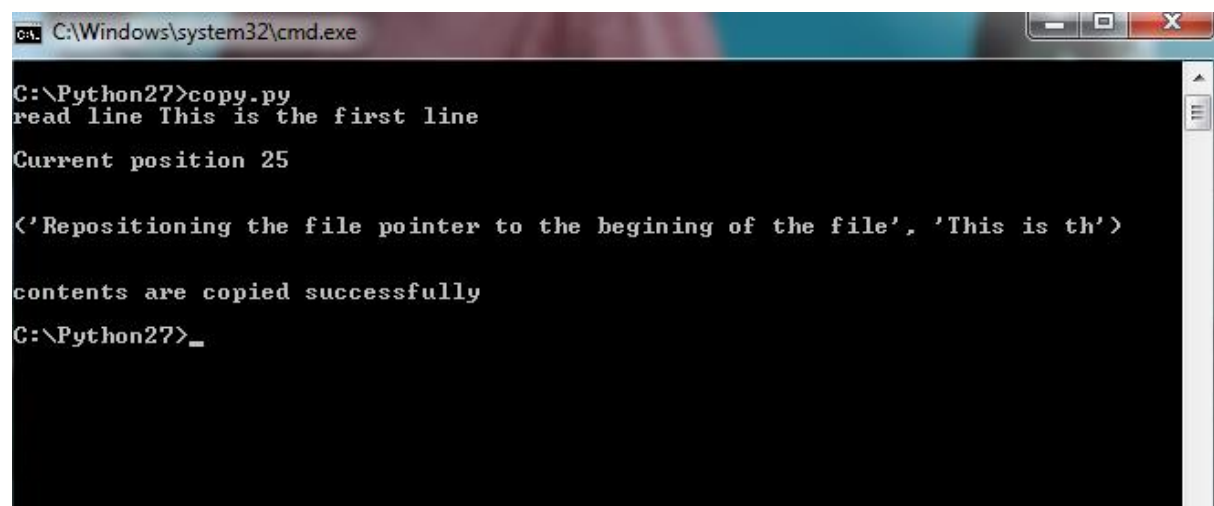
1. Program to demonstrate the use of `seek()`, `tell()` and `readline()`.

Example:

```

fob=open("texting.txt","w")
fob.write("This is the first line \n This is the second line")
fob=open("texting.txt")
line=fob.readline()
print("read line %s" %(line))
#To get the current position of the file
position=fob.tell()
print("Current position %d" %(position))
print("\n")
#Repositioning pointer at the begining once again
position1=fob.seek(0)
ran=fob.read(10)
print("Repositioning the file pointer to the begining of the file",ran)
print("\n")
#copying contents of one file to another.
with open("texting.txt")as fa:
    with open("out1.txt","w")as fa1:
        for line in fa:
            fa1.write(line)
print("contents are copied successfully")

```



```

C:\Windows\system32\cmd.exe
C:\Python27>copy.py
read line This is the first line
Current position 25
<'Repositioning the file pointer to the begining of the file', 'This is th'>
contents are copied successfully
C:\Python27>_

```

2.Program to write numbers from 1 to 20 to the output file “t.txt”

```
f=open("t.txt","w")
for i in range(1,21):
    x=str(i)+" "
    f.write(x)
f.close()
f=open("t.txt","r")
print(f.read())
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

3. Program to read the contents of a file.

Ans:

```
a=str(input("Enter the name of the file with .txt extension:"))
file2=open(a,'r')
line=file2.readline()
while(line!=""):
    print(line)
    line=file2.readline()
file2.close()
```

4. Program to check End of file is reached.

```
f=open("deepika.txt","r")
eof=f.read()
if(eof=="") :
    print("END OF FILE")
```

Output:

```
END OF FILE
```

5. Program to demonstrate the use of r+ mode

```
f=open("file1.txt","w")
f.write("python programming language\n")
f.write("object oriented language\n")
f.write("c programming language")
f.close()
f=open("file1.txt","r+")
f.write("\ni work in meerut")
print(f.read())
```

Output:

```
python programming language
object oriented language
```



```
c programming language
i work in meerut
```

6. Program to use the split() method in the file

```
f=open("deepika.txt","r+")
b=f.read()
print(b)
c=b.split("\n")
print(c)
for i in c:
    print(i)
```

Output:

```
i live in meerut
my name is deepika
my name is deepika
['i live in meerut', ' my name is deepika', 'my name is deepika']
i live in meerut
my name is deepika
my name is deepika
```

7. Program that accepts filename as an input from the user. Open the file and count the number of times a character appears in the file.

```
filename=input("enter file name ")
with open(filename) as f:
    text=f.read()
    letter=input("enter letter to be searched ")
    count=0
    for i in text:
        if i==letter:
            count=count+1
print(letter,"appears",count,"times in file")
```

Output:

```
enter file name file13.txt
enter letter to be searched a
a appeaars 8 times in file
```

8. Program that copies contents of one file into another file in such a way that all comments lines are skipped and not copied to second file.

```
with open("first.txt","w") as f:
    f.write("# this is the first file\n")
    f.write("i am deepika gupta\n")
    f.write("i am a teacher")

with open("first.txt","r") as f:
    with open("second.txt","w") as f1:
        while True:
            a=f.readline()
            if len(a)!=0:
```

```

    if a[0]=='#':
        continue
    else:
        f1.write(a)
else:
    break

```

or using another open() method

```

f1=open("first.txt","r")
f2=open("second.txt","w")
while True:
    a=f1.readline()
    if len(a)!=0:
        if a[0]=='#':
            continue
        else:
            f2.write(a)
    else:
        break

```

```

f=open("second.txt","r")
print(f.read())

```

Output:

```

i am deepika gupta
i am a teacher

```

9. Program that reads a file line by line. Each line read from the file is copied to another file with line numbers specified at the beginning of the line.

```

f1=open("one.txt","r")
f2=open("second.txt","w")
num=1
for i in f1:
    print(i)
    f2.write(str(num)+":"+i)
    num=num+1
f1.close()
f2.close()

```

Output:

```

1:Mango
2:banana
3:Grapes

```

10. Program to count the number of words in a text file.

Ans:

```
fname = input("Enter file name: ")

num_words = 0

with open(fname, 'r') as f:
    for line in f:
        words = line.split()
        num_words += len(words)
print("Number of words:")
print(num_words)
```

12. Program to read a string from the user and appends it into a file.

Ans:

13. Program to count the occurrences of a word in a text file

Ans:

14. Python Program to copy the contents of one file into another.

Ans: