Python File Operation

What Is File:-

File is a named location on disk to store related information. It is 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 computer is turned off, we use files for future use of the data.When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

Type of File:-

In Windows, for example, a file can be any item manipulated, edited or created by the user/OS. That means files can be images, text documents, executables, and much more. Most files are organized by keeping them in individual folders.

In Python, a file is categorized as either text or binary, and the difference between the two file types is important.

Text files are structured as a sequence of lines, where each line includes a sequence of characters. This is what you know as code or syntax. Each line is terminated with a special character, called the **EOL or End of Line character**. There are several types, but the most common is the comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. A backslash character can also be used, and it tells the interpreter that the next character – following the slash – should be treated as a new line. This character is useful when you don't want to start a new line in the text itself but in the code.

A binary file is any type of file that is not a text file. Because of their nature, binary files can only be processed by an application that know or understand the file's structure. In other words, they must be applications that can read and interpret binary.

Opening and Closing Files

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

The *open* Function

Before you can read or write a file, you have to open it using Python's built-in *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 parameter details −

- **file_name** − Thefile_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − Theaccess_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 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).

different modes of opening a file −

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.
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	**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.
7	**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.
8	**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.
9	**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.
10	**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.

11	**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.
12	**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.

The *file* Object Attributes

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.

Example

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

Output:-

Name of the file: ECSIII.txt
Closed or not : False
Opening mode :wb
Softspaceflag : 0

The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close();
```

Example

```
es1=open("ECSIII.txt","wb")
print"Name of the file: ",es1.name
es1.close()
```

Output:-

```
Name of the file:  ECSIII.txt
```

Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

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.

Example

```
es= open("ECSIII.txt","wb")
es.write("Python is a great language.\nYeah its great!!\n");
es.close()
```

Output:-

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

The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

```
obj= open("ECSIII.txt","r+")
str=obj.read(10);
print"Read String is : ",str
obj.close()
```

Output:-

```
Read String is :  Python is
```

Renaming and Deleting Files

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.To use this module you need to import it first and then you can call any related functions.

The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax

os.rename(current_file_name, new_file_name)

Example

```
import os
os.rename( "test1.txt", "test2.txt" )
```

The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

os.remove(file_name)

Example

```
import os
os.remove("text2.txt")
```

File Methods

There are various methods available with the file object. Some of them have been used in above examples.

Method	Description
close()	Close an open file. It has no effect if the file is already closed.
detach()	Separate the underlying binary buffer from the TextIOBase and return it.
fileno()	Return an integer number (file descriptor) of the file.
flush()	Flush the write buffer of the file stream.
isatty()	Return True if the file stream is

	interactive.
read(*n*)	Read atmost *n* characters form the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(*n*=-1)	Read and return one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Read and return a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*,*from*=SEEK_SET)	Change the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=None)	Resize the file stream to *size* bytes. If *size* is not specified, resize to current location.
writable()	Returns True if the file stream can be written to.
write(*s*)	Write string *s* to the file and return the number of characters written.
writelines(*lines*)	Write a list of *lines* to the file.

Example for fileno() method-

```
e1 = open("ECSIII.txt", "wb")
```

```
print "Name of the file: ", e1.name
fid = e1.fileno()
print "File Descriptor: ", fid
e1.close()
```

Example for readline() method-

```
e = open("ECSIII.txt", "rw+")
print "Name of the file: ", e.name
line = e.readline()
print "Read Line: %s" % (line)

line = e.readline(5)
print "Read Line: %s" % (line)
e.close()
```

Example for tell() method-

```
f1 = open("ECSIII.txt", "rw+")
print "Name of the file: ", f1.name
line = f1.readline()
print "Read Line: %s" % (line)
pos = f1.tell()
print "Current Position: %d" % (pos)
f1.close()
```

Example for seek() method-

is used to change the position of the File Handle to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax:

f.seek(offset, from_what), where f is file pointer

Parameters:

Offset: Number of positions to move forward

from_what: It defines point of reference.

Returns: Return the new absolute position.

The reference point is selected by the **from_what** argument. It accepts three values:

- 0: sets the reference point at the beginning of the file
- 1: sets the reference point at the current file position
- 2: sets the reference point at the end of the file

By default from_what argument is set to 0.

Note: Reference point at current position / end of file cannot be set in text mode except when offset is equal to 0.

```
# Python program to demonstrate

f = open("GfG.txt", "r")
f.seek(20)
print(f.tell())
 print(f.readline())
f.close()
```

Example 2:

Seek() function with negative offset only works when file is opened in binary mode. Let's suppose the binary file contains the following text.

```
f = open("data.txt", "rb")
# sets Reference point to tenth  position to the left from end
f.seek(-10, 2)
```

```
print(f.tell())
print(f.readline().decode('utf-8'))
f.close()
```

zipping and unzipping files;-

What is a zip file

ZIP file is a file format that is used for compressing multiple files together into a single file. It is used in an archive file format that supports lossless data compression and reduces storage requirements it also improves data transfer over standard connections. Zip files make the task of sharing multiple files easy by comprising them into one. The ZipFile class contains extractall() and extract() methods which are used for unzipping the files.

Why do we need zip files?

- To reduce storage requirements.
- To improve transfer speed over standard connections.

extractall()

The extractall() method is used to extract all the files present in the zip file to the current working directory. Files can also be extracted to a different location bypassing the path parameter.

Syntax: ZipFile.extractall(file_path , members=None, pwd=None)

Parameters:

- **file_path:** location where archive file needs to be extracted, if file_path is None then contents of zip file will be extracted to current working directory

- **members:** It specifies the list of files to be extracted, if not specified, all the files in the zip will be extracted. members must be a subset of the list returned by namelist()
- **pwd:** the password used for encrypted files, By default pwd is None.

extract()

The extract() method is used to Extract a member from the zip to the current working directory. The file can also be extracted to a different location bypassing the path parameter.

Syntax: ZipFile.extract(member, file_path=None , pwd=None)

- members: It specifies the name of files to be extracted.
- file_path: location where archive file needs to be extracted, if file_path is None then contents of zip file will be extracted to the current working directory
- pwd: the password used for encrypted files, By default pwd is None.

Example 1: Extracting all the files present in the zip

Import the zipfile module Create a zip file object using ZipFile class. Call the extractall() method on zip file object and pass the path where the files needed to be extracted and Extracting the specific file present in the zip.

```
# importing the zipfile module
from zipfile import ZipFile
with ZipFile("C:\\Users\\sai mohan pulamolu\\\
        Desktop\\geeks_dir\\temp\\temp.zip", 'r') as zObject:
    zObject.extractall(
```

path="C:\\Users\\sai mohan pulamolu\\Desktop\\geeks_dir\\temp")

Program

```
from zipfile import ZipFile
file_name = "my_python_files.zip"
with ZipFile(file_name, 'r') as zip:
    zip.printdir()
    print('Extracting all the files now...')
    zip.extractall()
    print('Done!')
```

Pickling and Unpickling:-

To serialize and de-serialize a Python object structure, we have the Pickle module in Python. The pickle module implements binary protocols for serializing and de-serializing a Python object structure.

Pickling is the process through which a Python object hierarchy is converted into a byte stream. To serialize an object hierarchy, you simply call the dumps() function.

Unpickling is the inverse operation. A byte stream from a binary file or bytes-like object is converted back into an object hierarchy. To de-serialize a data stream, you call the loads() function.

Pickling and unpickling are alternatively known as serialization.

What can be pickled and unpickled?

In Python, the following types can be pickled −

- None, True, and False.
- integers, floating-point numbers, complex numbers.
- strings, bytes, bytearrays.
- tuples, lists, sets, and dictionaries containing only picklable objects.
- functions, built-in and user-defined.

Pickle Module Constants

pickle.HIGHEST_PROTOCOL − The highest protocol version available. Integer value.

pickle.DEFAULT_PROTOCOL − The default protocol version used for pickling. Integer value. Currently the default protocol is 4

Pickle Module Functions

pickle.dump() − Write the pickled representation of the object to the open file object file.

pickle.dumps() − Return the pickled representation of the object as a bytes object, instead of writing it to a file.

pickle.load() − Read the pickled representation of an object from the open file object file.

pickle.loads() − Return the reconstituted object hierarchy of the pickled representation data of an object.

Program:-

```
import pickle
my_data = { 'BMW', 'Audi', 'Toyota', 'Benz'}
with open("demo.pickle","wb") as file_handle:
  pickle.dump(my_data, file_handle,
pickle.HIGHEST_PROTOCOL)
```

```
with open("demo.pickle","rb") as file_handle:
  res = pickle.load(file_handle)
  print(my_data) # display the output
```

Working directories :-

If there are a large number of files to handle in your Python program, you can arrange your code within different directories to make things more manageable.
A directory or folder is a collection of files and sub directories. Python has theos module, which provides us with many useful methods to work with directories

The *mkdir()* Method

You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

```
importos
os.mkdir("test")
```

The *chdir()* Method

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

```
os.chdir("newdir")
```

Example

```
importos
os.chdir("/home/newdir")
```

The *getcwd()* Method

The *getcwd()* method displays the current working directory.

Syntax

```
os.getcwd()
```

Example

```
importos
os.getcwd()
```

The *rmdir()* Method

The *rmdir()* method deletes the directory, which is passed as an argument in the method.Before removing a directory, all the contents in it should be removed.

Syntax

```
os.rmdir('dirname')
```

Example

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
importos
```

```
os.rmdir("/tmp/test")
```

Delete a directory

To delete a directory, you use the os.rmdir() function as follows:

```
import os
dir = os.path.join("C:\\","temp","python")
if os.path.exists(dir):
    os.rmdir(dir)
    print(dir + ' is removed.')
```