UNIT-V (Chapter-1)

File Handling

File

A file is a collection of data stored on a secondary storage device like hard disk.

A file is basically used because real-life applications involve large amounts of data and in such situations the console oriented I/O operations pose two major problems:

- First, it becomes cumbersome and time consuming to handle huge amount of data through terminals.
- Second, when doing I/O using terminal, the entire data is lost when either the program is terminated or computer is turned off. Therefore, it becomes necessary to store data on a permanent storage (the disks) and read whenever necessary, without destroying the data.

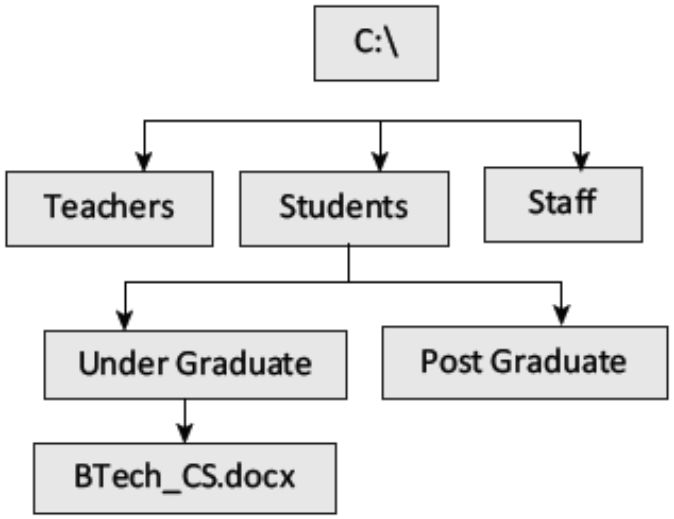
File Path

Files that we use are stored on a storage medium like the hard disk in such a way that they can be easily retrieved as and when required.

Every file is identified by its path that begins from the root node or the root folder. In Windows, C:\ (also known as C drive) is the root folder but you can also have a path that starts from other drives like D:\, E:\, etc. The file path is also known as ***pathname***.

Relative Path and Absolute Path

A file path can be either *relative* or *absolute*. While an absolute path always contains the root and the complete directory list to specify the exact location the file, relative path needs to be combined with another path in order to access a file. It starts with respect to the current working directory and therefore lacks the leading slashes. For example, C:\Students\Under Graduate\BTech_CS.docx but Under Graduate\BTech_CS.docx is a relative path as only a part of the complete path is specified.



ASCII Text Files

A *text file* is a stream of characters that can be sequentially processed by a computer in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Because text files can process characters, they can only read or write data one character at a time. In Python, a text stream is treated as a special kind of file.

Depending on the requirements of the operating system and on the operation that has to be performed (read/write operation) on the file, the newline characters may be converted to or from carriage-return/linefeed combinations. Besides this, other character conversions may also be done to satisfy the storage requirements of the operating system. However, these conversions occur transparently to process a text file. In a text file, each line contains zero or more characters and ends with one or more characters

Another important thing is that when a text file is used, there are actually two representations of data- internal or external. For example, an integer value will be represented as a number that occupies 2 or 4 bytes of memory internally but externally the integer value will be represented as a string of characters representing its decimal or hexadecimal value.

Binary Files

A ***binary file*** is a file which may contain any type of data, encoded in binary form for computer storage and processing purposes. It includes files such as word processing documents, PDFs, images, spreadsheets, videos, zip files and other executable programs. Like a text file, a binary file is a collection of bytes. A binary file is also referred to as a character stream with following two essential differences.

- A binary file does not require any special processing of the data and each byte of data is transferred to or from the disk unprocessed.
- Python places no constructs on the file, and it may be read from, or written to, in any manner the programmer wants.

While text files can be processed sequentially, binary files, on the other hand, can be either processed sequentially or randomly depending on the needs of the application.

The Open() Function

Before reading from or writing to a file, you must first open it using Python's built-in open() function. This function creates a file object, which will be used to invoke methods associated with it. The syntax of open() is:

fileObj = open(file_name [, access_mode])

Here,

***file_name* is a string value that specifies name of the file that you want to access.**

***access_mode* indicates the mode in which the file has to be opened, i.e., read, write, append, etc.**

Example:

```
file = open("File1.txt", "rb")
print(file)
```

OUTPUT

```
<open file 'File1.txt', mode 'rb' at 0x02A850D0>
```

The open() Function – Access Modes

Mode	Purpose
r	This is the default mode of opening a file which opens the file for reading only. The file pointer is placed at the beginning of the file.
rb	This mode opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.
r+	This mode opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
rb+	This mode opens the file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
w	This mode opens the file for writing only. When a file is opened in w mode, two things can happen. If the file does not exist, a new file is created for writing. If the file already exists and has some data stored in it, the contents are overwritten.
wb	Opens a file in binary format for writing only. When a file is opened in this mode, two things can happen. If the file does not exist, a new file is created for writing. If the file already exists and has some data stored in it, the contents are overwritten.
w+	Opens a file for both writing and reading. When a file is opened in this mode, two things can happen. If the file does not exist, a new file is created for reading as well as writing. If the file already exists and has some data stored in it, the contents are overwritten.
wb+	Opens a file in binary format for both reading and writing. When a file is opened in this mode, two things can happen. If the file does not exist, a new file is created for reading as well as writing. If the file already exists and has some data stored in it, the contents are overwritten.
a	Opens a file for appending. The file pointer is placed at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
ab	Opens a file in binary format for appending. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both reading and appending. The file pointer is placed at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file in binary format for both reading and appending. The file pointer is placed at the end of the file if the file exists. If the file does not exist, a new file is created for reading and writing.

The close () Method

The *close() method* is used to close the file object. Once a file object is closed, you cannot further read from or write into the file associated with the file object. While closing the file object the close() flushes any unwritten information. Although, Python automatically closes a file when the reference object of a file is reassigned to another file, but as a good programming habit you should always explicitly use the close() method to close a file. The syntax of close() is **fileObj.close()**

The close() method frees up any system resources such as file descriptors, file locks, etc. that are associated with the file. Moreover, there is an upper limit to the number of files a program can open. If that limit is exceeded then the program may even crash or work in unexpected manner. Thus, you can waste lots of memory if you keep many files open unnecessarily and also remember that open files always stand a chance of corruption and data loss.

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

The File Object Attributes

Once a file is successfully opened, a *file* object is returned. Using this file object, you can easily access different type of information related to that file. This information can be obtained by reading values of specific attributes of the file.

Attribute	Information Obtained
`fileObj.closed`	Returns true if the file is closed and false otherwise
`fileObj.mode`	Returns access mode with which file has been opened
`fileObj.name`	Returns name of the file

Example:

```
file = open("File1.txt", "wb")
print("Name of the file: ", file.name)
print("File is closed.", file.closed)

print("File has been opened in ", file.mode, "mode")
```

OUTPUT

```
Name of the file:  File1.txt
File is closed. False
File has been opened in wb mode
```

The read() Method

The *read() method* is used to read a string from an already opened file. As said before, the string can include, alphabets, numbers, characters or other symbols. The syntax of read() method is fileObj.read([count])

In the above syntax, count is an optional parameter which if passed to the read() method specifies the number of bytes to be read from the opened file. The read() method starts reading from the beginning of the file and if *count* is missing or has a negative value then, it reads the entire contents of the file (i.e., till the end of file).

Example:

```
file = open("File1.txt", "r")
print(file.read(10))
file.close()
```

OUTPUT

```
Hello All,
```

The readline() & readlines() Methods

The *readline() method* only reads a single line of the file. This means that if the file contains three lines long, the readline() function would only parse (or iterate/operate) on the first line of the file.

The *readlines() method* reads all lines of the file. If the file contains three lines long, the readlines() would read all lines as like a list. Each line reads as a single item of the list

The write() and writelines() Methods

The *write() method* is used to write a string to an already opened file. Of course this string may include numbers, special characters or other symbols. While writing data to a file, you must remember that the write() method does not add a newline character ('\n') to the end of the string. The syntax of write() method is: fileObj.write(string)

The *writelines() method* is used to write a list of strings.

Examples:

```
file = open("File1.txt", "w")
file.write("Hello All, hope you are enjoying learning Python")
file.close()
print("Data Written into the file.......")
```

OUTPUT

```
Data Written into the file......."
```

```
file = open("File1.txt", "w")
lines = ["Hello World, ", "Welcome to the world of Python", "Enjoy Learning
Python"]
file.writelines(lines)
file.close()
print("Data written to file........")
```

OUTPUT

```
Data written to file........
```

append() Method

Once you have stored some data in a file, you can always open that file again to write more data or append data to it. To append a file, you must open it using 'a' or 'ab' mode depending on whether it is a text file or a binary file. Note that if you open a file in 'w' or 'wb' mode and then start writing data into it, then its existing contents would be overwritten. So always open the file in 'a' or 'ab' mode to add more data to existing data stored in the file.

Appending data is especially essential when creating a log of events or combining a large set of data into one file.

Example:

```
file = open("File1.txt", "a")
file.write("\n Python is a very simple yet powerful language")
file.close()
print("Data appended to file........")
```

OUTPUT

```
Data appended to file........"
```

Opening Files using "with" Keyword

It is good programming habit to use the with keyword when working with file objects. This has the advantage that the file is properly closed after it is used even if an error occurs during read or write operation or even when you forget to explicitly close the file. (No need to close the file explicitly).

Examples:

```
with open("sample.txt","r") as file:
    print(file.read())
    print(file.closed)

print(file.closed)
```

Output:

```
The Quick Brown Fox jumps over a lazy dog.
False
True
```

```
file=open("sample.txt",'r')
print(file.read())
print(file.closed)
file.close()
```

Output:

```
The Quick Brown Fox jumps over a lazy dog.
False
```

Splitting Words

Python allows you to read line(s) from a file and splits the line (treated as a string) based on a character. By default, this character is space but you can even specify any other character to split words in the string.

Example:

```
with open('File1.txt', 'r') as file:
    line = file.readline()
    words = line.split()
    print(words)
```

OUTPUT

```
['Hello', 'World,', 'Welcome', 'to', 'the', 'world', 'of', 'Python', 'Programming']
```

File Positions

With every file, the file management system associates a pointer often known as ***file pointer*** that facilitate the movement across the file for reading and/ or writing data. The file pointer specifies a location from where the current read or write operation is initiated. Once the read/write operation is completed, the pointer is automatically updated.

Python has various methods that tells or sets the position of the file pointer.

- The **tell()** method tells the current position within the file at which the next read or write operation will occur. It is specified as number of bytes from the beginning of the file. When you just open a file for reading, the file pointer is positioned at location 0, which is the beginning of the file.

File Positions

seek() method

In Python, seek() function 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, whence), *where f is file pointer*

Offset: *Number of postions to move forward*

whence: It defines point of reference.

The reference point is selected by the **whence** 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

Note: By default from_what argument is set to 0.

File Positions - Example

```
file = open("File1.txt", "rb")
print("Position of file pointer before reading is : ", file.tell())
print(file.read(10))
print("Position of file pointer after reading is : ", file.tell())
print("Setting 3 bytes from the current position of file pointer")
file.seek(3,1)
print(file.read())
file.close()
```

OUTPUT

```
Position of file pointer before reading is : 0
Hello All,
Position of file pointer after reading is : 10
Setting 3 bytes from the current position of file pointer
pe you are enjoying learning Python
```

Renaming and Deleting Files

The os module in Python has various methods that can be used to perform file-processing operations like renaming and deleting files. To use the methods defined in the os module, you should first import it in your program then call any related functions.

The rename() Method: The *rename()* method takes two arguments, the current filename and the new filename. Its syntax is: os.rename(old_file_name, new_file_name)

The remove() Method: This method can be used to delete file(s). The method takes a filename (name of the file to be deleted) as an argument and deletes that file. Its syntax is: os.remove(file_name)

Examples:

```
import os
os.rename("File1.txt", "Students.txt")
print("File Renamed")
```

OUTPUT

```
File Renamed
```

```
import os
os.remove("File1.txt")
print("File Deleted")
```

OUTPUT

```
File Deleted
```

Directory Methods

The **mkdir()** Method: The mkdir()method of the OS module is used to create directories in the current directory. The method takes the name of the directory (the one to be created) as an argument. The syntax of mkdir() is, **os.mkdir("new_dir_name")**

The **getcwd()** Method:The getcwd() method is used to display the current working directory (cwd).

os.getcwd()

The **chdir()** Method: The chdir() method is used to change the current directory. The method takes the name of the directory which you want to make the current directory as an argument. Its syntax is

os.chdir("dir_name")

The **rmdir()** Method: The *rmdir()* method is used to remove or delete a directory. For this, it accepts name of the directory to be deleted as an argument. However, before removing a directory, it should be absolutely empty and all the contents in it should be removed.The syntax of remove() method is **os.rmdir(("dir_name")**

The **makedirs()** method:The method mkdirs() is used to create more than one folder.

Directory Methods - Examples

```
import os
os.rename("File1.txt", "Students.txt")
print("File Renamed")
```

OUTPUT

```
File Renamed
```

```
import os
os.remove("File1.txt")
print("File Deleted")
```

OUTPUT

```
File Deleted
```

```
import os
os.mkdir("New Dir")
print("Directory Created")
```

OUTPUT

```
Directory Created
```

```
import os
print("Current Working Directory is : ", os.getcwd())
os.chdir("New Dir")
print("After chdir, the current Directory is now..... ",
end = ' ')
print(os.getcwd())
```

OUTPUT

```
Current Working Directory is : C:\Python27
After chdir, the current Directory is now.....  C:\Python27\New Dir
```

Programming Tip: OS Object Methods: This provides methods to process files as well as directories.