**File Handling**

* A file is an object on a computer that stores [data](https://www.computerhope.com/jargon/d/data.htm), [information](https://www.computerhope.com/jargon/i/informat.htm), settings, or commands used with a computer [program](https://www.computerhope.com/jargon/p/program.htm).
* I/O operations are the costliest operations where a program can stumble.
* Hence, you should be quite careful while implementing file handling for reporting or any other purpose.
* Optimizing a single file operation can help you produce a high-performing application or a robust solution for automated software testing.
* Let’s take an example, say, you are going to create a big project in Python which contains a no. of workflows. Then, it’s inevitable for you not to create a log file. And you’ll also be doing both the read/write operations on the log file. Log files are a great tool to debug large programs. It’s always better to think about a scalable design from the beginning, as you won’t regret it later that you didn’t do it.

**Note:**

**File is a named location on the system storage which records data for later access. It enables persistent storage in a non-volatile memory i.e. hard disk.**

**Open a File:**

**Filename=open (filename [, access\_mode][, buffering])**

**<access\_mode>**

**File open mode e.g. read, write, append, etc.**

**Optional, by default it is read mode**

**<Buffering>**

**Default is 0 🡪 Buffer won’t happen**

**1 🡪 buffering will take place while accessing the file**

**>1 🡪 buffering action will run as per the buffer size**

**If Negative value, the default behaviour is considered**

**<Filename>**

**File name to be accessed.**

**File Open modes in Python:**

**<r>** read-only mode while the file offset stays at the root.

**<rb>** (binary + read-only) modes

**<r+>** Read and Write mode

**<rb+>** Read+Write+Binary

**<w>** It will create a new file if not exists or it will overwrite the existing file.

**<wb>** Write + Binary

**<w+>** Read + Write

**<a>** Append mode, offset goes to the end of the file, if file not exists it will create.

**<ab>** Append+ Binary

**<ab+>** Append + Read+ Binary

**<a+>** Append + Read

**Python File object attributes:**

File.closed It returns true for closed file

File.mode It returns the access mode used to open a file

File.name It returns the name of the file

File.softspace It returns a Boolean to suggest if a space char will get added before printing another value in the output of a <print> command.

**Python File Encoding:**

* In Python 3.X, we have clear difference between strings (text) and a byte (8-bits).
* It states that the character ‘a’ does not represent the ASCII value 97 until it is specified. So it is best to mention its encoding type.
* Also, Python stores a file in the form of bytes on the disk, so you need to decode them in strings before reading. And, similarly, encode them while writing texts to the file.
* For a note, Python enables platform-dependent encoding by default. Hence, if you don’t change it, then it’s set to <cp1252> for Windows and <utf-8> for Linux.
* Thus, the documentation says to quote the desired encoding while opening a file in Python. See the below Python code snippet.
  + **f = open('app.log', mode = 'r', encoding = 'utf-8')**
* For a note, you should import the <io> module in Python 2.x to enable the encoding feature. Python 3.x does it implicitly.

**Close a File:**

* While closing a file, the system frees up all resources allocated to it.

f = open ("app.log", encoding = 'utf-8')

# do file operations.

f.close ()

**Close with Try…Catch**

* If any error occurs when opening a file, the code exists without closing the file

try:

f = open('app.log', encoding = 'utf-8')

# do file operations.

finally:

f.close()

**Auto Close Using ‘With’**

 It ensures that the file gets closed when the block inside the **WITH** clause executes.

with open('app.log', encoding = 'utf-8') as f:

#do any file operation.

**Write Operations:**

with open('app.log', 'w', encoding = 'utf-8') as f:

#first line

f.write('my first file\n')

#second line

f.write('This file\n')

#third line

f.write('contains three lines\n')

with open('app.log', 'r', encoding = 'utf-8') as f:

content = f.readlines()

for line in content:

print(line)

**Read Operations:**

with open('app.log', 'w', encoding = 'utf-8') as f:

#first line

f.write('my first file\n')

#second line

f.write('This file\n')

#third line

f.write('contains three lines\n')

f = open('app.log', 'r', encoding = 'utf-8')

print(f.read(10)) # read the first 10 data

#'my first f'

print(f.read(4)) # read the next 4 data

#'ile\n'

print(f.read()) # read in the rest till end of file

#'This file\ncontains three lines\n'

print(f.read()) # further reading returns empty sting

#''

**File. tell() 🡺** This method gives you the current offset of the file pointer in a file.

**Seek() Method**

* + Change the position of a file pointer in a file.

file.seek(offset[, from])

The <offset> argument represents the size of the displacement.

The <from> argument indicates the start point.

If from is 0, then the shift will start from the root level.

If from is 1, then the reference position will become the current position.

It from is 2, then the end of the file would serve as the reference position.

**Example:**

with open('app.log', 'w', encoding = 'utf-8') as f:

#first line

f.write('It is my first file\n')

#second line

f.write('This file\n')

#third line

f.write('contains three lines\n')

#Open a file

f = open('app.log', 'r+')

data = f.read(19);

print('Read String is : ', data)

#Check current position

position = f.tell();

print('Current file position : ', position)

#Reposition pointer at the beginning once again

position = f.seek(0, 0);

data = f.read(19);

print('Again read String is : ', data)

#Close the opened file

f.close()

**output:**

Read String is : It is my first file

Current file position : 19

Again read String is : It is my first file

**Renaming the File**

os.rename(cur\_file, new\_file)

**Remove the File**

os.remove(file\_name)