**PYTHON FILE HANDLING: OPEN A FILE IN PYTHON.**

To read or write to a file, you need to open it first. To open a file in Python, use its built open() function. This function returns a file object i.e. a handle. You can use it to read or modify the file.

file object = open(file\_name [, access\_mode][, buffering])

below are the parameter details.

**<access\_mode>-** It’s an integer representing the file open mode e.g. read, write, append, etc. It’s an optional parameter. By default, it is set to read-only <r>. In this mode, we get data in text form after reading from the file. On the other hand, binary mode returns bytes. It’s preferable for accessing the non-text files like an image or the Exe files. See the table in the next section. It lists down the available access modes.

**<buffering>-** The default value is 0 which means buffering won’t happen. If the value is 1, then line buffering will take place while accessing the file. If it’s greater than 1, then the buffering action will run as per the buffer size. In the case of a negative value, the default behavior is considered.

**<file\_name>-** It’s a string representing the name of the file you want to access.

|  |  |
| --- | --- |
| **<r>** | It opens a file in read-only mode while the file offset stays at the root. |
| **rb** | It opens the file in both (read + write) modes while the file offset is again at the root level. |
| **r+** | It opens the file in both (read + write) modes while the file offset is again at the root level. |
| **rb+** | It opens the file in (read + write + binary) modes. The file offset is again at the root level. |
| **w** | It allows write-level access to a file. If the file already exists, then it’ll get overwritten. It’ll create a new file if the same doesn’t exist. |
| **wb** | Use it to open a file for writing in binary format. Same behavior as for write-only mode |
| **w+** | It opens a file in both (read + write) modes. Same behavior as for write-only mode. |
| **wb+** | It opens a file in (read + write + binary) modes. Same behavior as for write-only mode. |
| **a** | It opens the file in append mode. The offset goes to the end of the file. If the file doesn’t exist, then it gets created. |
| **ab** | It opens a file in (append + binary) modes. Same behavior as for append mode. |
| **a+** | It opens a file in (append + read) modes. Same behavior as for append mode. |
| **ab+** | It opens a file in (append + read + binary) modes. Same behavior as for append mode. |

|  |  |
| --- | --- |
| **<file.closed>** | For a closed file, it returns true whereas false otherwise. |
| **<file.mode>** | It returns the access mode used to open a file. |
| **<file.name>** | It returns the name of a 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. |

Enter your input: Hello Python

#Open a file in write and binary mode.

fob = open("app.log", "wb")

#Display file name.

print "File name: ", fob.name

#Display state of the file.

print "File state: ", fob.closed

#Print the opening mode.

print "Opening mode: ", fob.mode

#Output the softspace value.

print "Softspace flag: ", fob.softspace

**PYTHON FILE ENCODING.**

In Python 3.x, there is a clear difference between strings (text) and a byte (8-bits). It states that the char ‘a’ doesn’t represent the ASCII value 97 until you specify it like that. So, while you want to use a file in text mode, then better you mention the correct 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.

Python

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

##### EXAMPLE-1: BASIC CLOSE OPERATION IN PYTHON.

The most basic way is to call the Python close() method.

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

# do file operations.

f.close()

##### EXAMPLE-2: USING EXCEPTION WITH CLOSE OPERATION IN PYTHON.

Say, if an exception occurs while performing some operations on the file. In such a case, the code exits without closing the file. So it’s better to put the code inside a <try-finally> block.

try:

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

# do file operations.

finally:

f.close()# do file operations.

### PYTHON FILE HANDLING: PERFORM WRITE OPERATION.

#### THE WRITE() FILE METHOD:

Python provides the <write()> method to write a string or sequence of bytes to a file. This function returns a number which is the size of data written in a single Write call.

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)

Output:

Python 3.5.1

[GCC 4.8.2] on Linux

my first file

This file

contains three lines

### PYTHON FILE HANDLING: PERFORM READ OPERATION.

To read data from a file, first of all, you need to open it in read mode. Then, you can call anyone of the methods that Python provides for reading from a file.

Usually, you can use Python <read(size)> function to read the content of a file up to the size. If you don’t pass the size, then it’ll read the whole file.

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

#''

### PYTHON FILE HANDLING: FILE READ POSITIONS

The <tell()> method gives you the current offset of the file pointer in a file.

The <seek(offset[, from])> method can help you change the position of a file pointer in a file.

The <offset> argument represents the size of the displacement while the <from> argument indicates the start point.

If the <from> value is 0, then the shift will start from the root level. If it’s value is 1, then the reference position will become the current position. Also, if it is 2, then the end of file would serve as the reference position.

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()

### PYTHON FILE HANDLING: RENAMING AND DELETING FILES IN PYTHON

While you were using the <read/write> functions, you may also need to <rename/delete> a file in Python. So, there comes a <os> module in Python which brings the support of file <rename/delete> operations.

So, to continue, first of all, you should import the <os> module in your Python script.

#### THE RENAME() FILE METHOD.

import os

#Rename a file from <app.log> to <app1.log>

os.rename( "app.log", "app1.log" )

#Close the opened file

f.close()

#### THE REMOVE() FILE METHOD

import os

os.remove( "app1.log" )

|  |  |
| --- | --- |
| **<file.close()>** | Close the file. You need to reopen it for further access. |
| **<file.flush()>** | Flush the internal buffer. It’s same as the <stdio>’s <fflush()> function. |
| **<file.fileno()>** | Returns an integer file descriptor. |
| **<file.isatty()>** | It returns true if file has a <tty> attached to it. |
| **<file.next()>** | Returns the next line from the last offset. |
| **<file.read(size)>** | Reads the given no. of bytes. It may read less if EOF is hit. |
| **<file.readline(size)>** | It’ll read an entire line (trailing with a new line char) from the file. |
| **<file.readlines(size\_hint)>** | It calls the <readline()> to read until EOF. It returns a list of lines read from the file. If you pass <size\_hint>, then it reads lines equalling the <size\_hint> bytes. |
| **<file.seek(offset[, from])>** | Sets the file’s current position. |
| **<file.tell()>** | Returns the file’s current position. |
| **<file.truncate(size)>** | Truncates the file’s size. If the optional size argument is present, the file is truncated to (at most) that size. |
| **<file.write(string)>** | It writes a string to the file. And it doesn’t return any value. |
| **<file.writelines(sequence)>** | Writes a sequence of strings to the file. The sequence is possibly an iterable object producing strings, typically a list of strings. |