Text File

A **text file**  is a kind of [computer file](https://en.wikipedia.org/wiki/Computer_file) that is structured as a sequence of [lines](https://en.wikipedia.org/wiki/Line_(text_file)) of [electronic text](https://en.wikipedia.org/wiki/Electronic_text). A text file exists within a [computer file system](https://en.wikipedia.org/wiki/Computer_file_system). Ex: For example, web pages are plain text with [HTML](https://en.wikipedia.org/wiki/HTML) tags , log files,  Python source code files are a type of plain text file.

**Use** : In [computing](https://en.wikipedia.org/wiki/Computing), a **file system**  is used to control how data is [stored](https://en.wikipedia.org/wiki/Computer_data_storage) and retrieved. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified.

Opening and Closing of Files

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

The *open* Function

Before we can read or write a file, we 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](http://www.tutorialspoint.com/)

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

**fo = open('school.txt','w')**

Here are 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 optional parameter and the default file access mode is read (r).
* **buffering:** Sometimes it is better to [load](http://stackoverflow.com/questions/29712445/what-is-the-use-of-buffering-in-pythons-built-in-open-function) a bunch of the file into a buffer in memory, then consume it at will. 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](http://www.tutorialspoint.com/) default(default behavior).

Here is a list of the different modes of opening a file −

|  |  |
| --- | --- |
| **Modes** | **Description** |
| r | Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| 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. |
| r+ | Opens a file for both reading and writing. The file pointer placed at the beginning of the file. |
| rb+ | Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| file.closed | Returns true if file is closed, false otherwise. |
| file.mode | Returns access mode with which file was opened. |
| file.name | Returns name of the file. |
|  |  |

Example

# Open a file

fo = open("foo.txt", "w")

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 : w

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.

It is a good practice to use the close() method to close a file.

Syntax

fileObject.close();

Example

# Open a file

fo = open("foo.txt", "w")

print "Name of the file: ", fo.name

# Close opend file

fo.close()

This produces the following result −

Name of the file: foo.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](http://www.tutorialspoint.com/). 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

# Open a file

fo = open("foo.txt", "w")

fo.write( "Python is a great language.\nYeah its great!!\n");

# Close opend 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!!

## 

## 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

Let's take a file *foo.txt*, which we created above.

# Open a file

fo = open("foo.txt", "r+")

str = fo.read(10);

print "Read String is : ", str

# Close opend file

fo.close()

This produces the following result −

Read String is : Python is

## ReadLine Function

To read a file line by line you could just keep reading one character at a time with .read(1), until you run into a newline character \n.

There's an easier way though, which is to use the .readline() method in place of .read().

## 

## 

## 

## File Positions

**The *tell()*** method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

**The *seek(offset[, from])*** method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The  *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

### Example

Let us take a file *foo.txt*, which we created above.

# Open a file

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 opend file

fo.close()

This produces the following result −

Read String is : Python is

Current file position : 10

Again read String is : Python is

## 

## OS Module

The OS module in Python provides a way of using operating system dependent

functionality.

The functions that the OS module provides allows you to interface with the

underlying operating system that Python is running on

Python **os** module provides methods that help you perform file/directories-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.

## Renaming and Deleting Files

## 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

Following is the example to rename an existing file *test1.txt*:

import os

# Rename a file from test1.txt to test2.txt

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

Following is the example to delete an existing file *test2.txt* −

import os

# Delete file test2.txt

os.remove("text2.txt")

## Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change 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

Following is the example to create a directory *test* in the current directory −

import os

# Create a directory "test"

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

Following is the example to go into "/home/newdir" directory −

import os

# Changing a directory to "/home/newdir"

os.chdir("/home/newdir")

## The *getcwd()* Method

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

### Syntax

os.getcwd()

### Example

Following is the example to give current directory −

import os

# This would give location of the current directory

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.

import os

# This would remove "/tmp/test" directory.

os.rmdir( "/tmp/test" )



