FILE I/O

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.

File operation:

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

Opening a File

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

In [ ]:

```
file object = open(file_name [, access_mode][, buffering])
```

In [ ]:

```
f = open('example.txt') #open file in current direcotry
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

Python File Modes

'r' Open a file for reading. (default)

'w' Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.

'x' Open a file for exclusive creation. If the file already exists, the operation fails.

'a' Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.

't' Open in text mode. (default)

'b' Open in binary mode.

'+' Open a file for updating (reading and writing)

In [ ]:

```
f = open('example.txt') #equivalent to 'r'
f = open('example.txt', 'r')

f = open('test.txt', 'w')
```

The default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

Closing a File

Closing a file will free up the resources that were tied with the file and is done using the close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

In [1]:

```
f = open('example.txt')
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

In [ ]:

```
try:
   f = open("example.txt")
   # perform file operations

except : print
finally:
   f.close()
```

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

The best way to do this is using the with statement. This ensures that the file is closed when the block inside with is exited.

We don't need to explicitly call the close() method. It is done internally.

with open("example.txt",encoding = 'utf-8') as f:

```
    #perform file operations
```

Writing to a File

In order to write into a file we need to open it in **write 'w', append 'a' or exclusive creation 'x' mode**.

We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using **write()** method. This method returns the number of characters written to the file.

In [ ]:

```
f = open("test.txt", "w")    # with open("test.txt", "w") as fd:
f.write("This is a First File\n")
f.write("Contains two lines\n")
f.close()
```

This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten.

Reading From a File

There are various methods available for this purpose. We can use the read(size) method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.

In [ ]:

```
f = open("test.txt", "r")
f.read()
```

In [ ]:

```
f = open("test.txt", "r")
f.read(4)
```

In [ ]:

```
#f = open("test.txt","r")
f.read(10)
```

We can change our current file cursor (position) using the seek() method.

Similarly, the **tell()** method returns our current position (in number of bytes).

In [ ]:

```
f.tell()
```

In [ ]:

```
f.seek(0) #bring the file cursor to initial position
```

In [ ]:

```
print(f.read()) #read the entire file
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

In [ ]:

```
f.seek(0)
for line in f:
    print(line)
```

Alternately, we can use readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.

In [ ]:

```
f = open("test.txt", "r")
f.readline()
```

In [ ]:

```
f.readline()
```

In [ ]:

```
f.readline()
```

The **readlines()** method returns a list of remaining lines of the entire file. All these reading method return empty values when end of file (EOF) is reached.

In [ ]:

```
f.seek(0)
f.readlines()
```

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.

In [ ]:

```
import os

#Rename a file from test.txt to sample.txt
os.rename("test.txt", "sample.txt")
```

In [ ]:

```
f = open("sample.txt", "r")
f.readline()
```

In [ ]:

```
#Delete a file sample.txt
os.remove("sample.txt")
```

In [ ]:

```
f = open("sample.txt", "r")
f.readline()
```

Python Directory and File Management

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 the os module, which provides us with many useful methods to work with directories (and files as well).

Get current Directory

We can get the present working directory using the getcwd() method.

This method returns the current working directory in the form of a string.

In [ ]:

```
import os
os.getcwd()
```

Changing Directory

We can change the current working directory using the chdir() method.

The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash () to separate path elements.

In [ ]:

```
os.chdir("/Users/varma/")
```

In [ ]:

```
os.getcwd()
```

List Directories and Files

All files and sub directories inside a directory can be known using the listdir() method.

In [ ]:

```
os.listdir(os.getcwd())
```

Making New Directory

We can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

In [ ]:

```
os.mkdir('test')
```

However, note that rmdir() method can only remove empty directories.

In order to remove a non-empty directory we can use the rmtree() method inside the shutil module.

In [ ]:

```
os.rmdir('test')
```

In [ ]:

```
import shutil

os.mkdir('test')
os.chdir('./test')
f = open("testfile.txt",'w')
f.write("Hello World")
os.chdir("../")
os.rmdir('test')
```

In [ ]:

```
# remove an non-empty directory
shutil.rmtree('test')
```

In [ ]:

```
os.getcwd()
```

In [1]:

```
with open("test.txt",'w') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```