Python 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. Hence, in Python, a file operation takes place in the following order.

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.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python33/README.txt")  # specifying full path
```

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. The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Python File Modes	
Mode	**Description**
'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)

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Since the version 3.x, Python has made a clear distinction between `str` (text) and `bytes` (8-bits). Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using `ASCII` (or other equivalent encodings). Hence, when working with files in text mode, it is recommended to specify the encoding type. Files are stored in bytes in the disk, we need to decode them into `str` when we read into Python. Similarly, encoding is performed while writing texts to the file.

The default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux. Hence, we must not rely on the default encoding otherwise, our code will behave differently in different platforms. Thus, this is the preferred way to open a file for reading in text mode.

```
f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

Closing a File

When we are done with operations to the file, we need to properly close it. Python has a garbage collector to clean up unreferenced objects. But we must not rely on it to close the file. Closing a file will free up the resources that were tied with the file and is done using the `close()` method.

```
f = open("test.txt",encoding = 'utf-8')
# perform file operations
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.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
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("test.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.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten. We must include the newline characters ourselves to distinguish different lines.

Reading From a File

To read the content of a file, we must open the file in reading mode. 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.

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '
```

```
>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'

>>> f.read()  # further reading returns empty sting
''
```

We can see, that `read()` method returns newline as '\n'. Once the end of file is reached, we get empty string on further reading. We can change our current file cursor (position) using the `seek()` method. Similarly, the `tell()` method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position
56

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

>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

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

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

The lines in file itself has a newline character '\n'. Moreover,the `print()` function also appends a newline by default. Hence, we specify the *end* parameter to avoid two newlines when printing.

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.

```
>>> f.readline()
'This is my first file\n'

>>> f.readline()
```

```
'This file\n'

>>> f.readline()
'contains three lines\n'

>>> f.readline()
''
```

Lastly, 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.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

Python File Methods

There are various methods available with the file object. Some of them have been used in above examples. Here is the complete list of methods in text mode with a brief description.

Python File Methods	
Method	**Description**
close()	Close an open file. It has no effect if the file is already closed.
detach()	Separate the underlying binary buffer from the `TextIOBase` and return it.
fileno()	Return an integer number (file descriptor) of the file.
flush()	Flush the write buffer of the file stream.
isatty()	Return `True` if the file stream is interactive.
read(*n*)	Read atmost *n* characters form the file. Reads till end of file if it is negative or `None`.
readable()	Returns `True` if the file stream can be read from.
readline(*n*=-1)	Read and return one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Read and return a list of lines from the file. Reads in at most *n* bytes/characters if specified.

seek(*offset*,*from*=`SEEK_SET`)	Change the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=`None`)	Resize the file stream to *size* bytes. If *size* is not specified, resize to current location.
writable()	Returns `True` if the file stream can be written to.
write(*s*)	Write string *s* to the file and return the number of characters written.
writelines(*lines*)	Write a list of *lines* to the file.