Python Programming - VI

By Nimesh Kumar Dagur, CDAC, India

Working with Files in Python Programming

- File is a named location on disk to store related information
- While a program is running, its data is in memory.
- When the program ends, or the computer shuts down, data in memory disappears.
- To store data permanently, you have to put it in a File.
- file 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.

- Files are usually stored on a hard drive, floppy drive, or CD-ROM.
- When there are a large number of files, they are often organized into directories (also called \folders").
- Each file is identified by a unique name, or a combination of a file name and a directory name.
- By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.

Working with Files in Python Programming

- Working with files is a lot like working with books.
- To use a book, you have to open it. When you're done, you have to close it.
- While the book is open, you can either write in it or read from it.
- In either case, you know where you are in the book.
- Most of the time, you read the whole book in its natural order, but you can also skip around.
- All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.
- Opening a file creates a file object.

Working with Files in Python Programming

- In Python, a file operation takes place in the following order.
  - Open a file
  - Read or write (perform operation)
  - 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
```

- Opening a file creates what we call a file **handle**.
- In this example, the variable **f** refers to the new handle object.
- Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk.

Opening a File

- 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 text 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)

Python File Modes

```
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
```

Python File Modes

- 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.

Python File Modes

- Thus, this is the preferred way to open a file for reading in text mode.

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

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.

Writing to a File

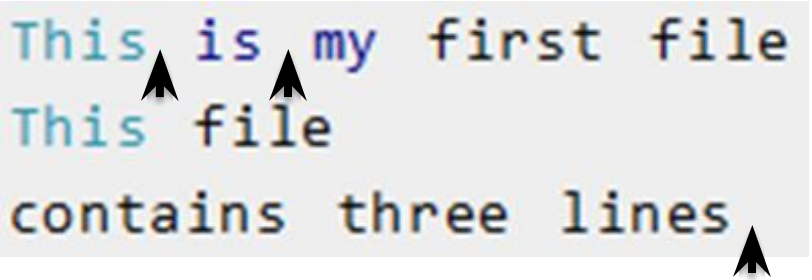
```
with open("test.txt",'w',encoding = 'utf-8') as f:
    f.write("my first file\n")
    f.write("This file\n\n")
    f.write("contains three lines\n")
```

- 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
- read() method returns newline as '\n'.
- Once the end of file is reached, we get empty string on further reading.

Reading From a 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
''
```

Reading From a File

- We can change our current file cursor (position) using the seek() method.
- Similarly, the tell() method returns our current position (in number of bytes).

Reading From a File

```
>>> 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
```

Reading From a File

- 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.

Reading From a File

- 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()
''
```

Reading From a File

- 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.

```
This is my first file
This file
contains three lines
```

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

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()
```

Closing a File

- Previous 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.

Closing a File

- 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
```

Python Directory and Files Management

- If there are large number of files in Python, we can place related files in different directories to make things more manageable.
- A directory or folder is a collection of files and sub directories.
- Files on non-volatile storage media are organized by a set of rules known as a file system.
- File systems are made up of files and directories, which are containers for both files and other 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.
- We can also use the **getcwdb()** method to get it as bytes object.

Get Current Directory

```
>>> import os

>>> os.getcwd()
'C:\\Program Files\\PyScripter'

>>> os.getcwdb()
b'C:\\Program Files\\PyScripter'
```

- The extra backslash implies escape sequence.
- The print() function will render this properly.

```
>>> print(os.getcwd())
C:\Program Files\PyScripter
```

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.
- It is safer to use escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())
C:\Python33
```

List Directories and Files

- All files and sub directories inside a directory can be known using the listdir() method.
- This method takes in a path and returns a list of sub directories and files in that path.
- If no path is specified, it returns from the current working directory.

```
>>> print(os.getcwd())
C:\Python33

>>> os.listdir()
['DLLs',
'Doc',
'include',
'Lib',
'libs',
'LICENSE.txt',
'NEWS.txt',
'python.exe',
'pythonw.exe',
'README.txt',
'Scripts',
'tcl',
'Tools']
```

```
>>> os.listdir('G:\\')
['$RECYCLE.BIN',
'Movies',
'Music',
'Photos',
'Series',
'System Volume Information']
```

Making a 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.

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

>>> os.listdir()
['test']
```

Renaming a Directory or a File

- The rename() method can rename a directory or a file.
- The first argument is the old name and the new name must be supplies as the second argument.

```
>>> os.listdir()
['test']

>>> os.rename('test','new_one')

>>> os.listdir()
['new_one']
```

Removing Directory or File

- A file can be removed (deleted) using the remove() method.
- Similarly, the rmdir() method removes an empty directory.

```
>>> os.listdir()
['new_one', 'old.txt']

>>> os.remove('old.txt')
>>> os.listdir()
['new_one']

>>> os.rmdir('new_one')
>>> os.listdir()
[]
```

Removing Directory or File

- 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.

```
>>> os.listdir()
['test']

>>> os.rmdir('test')
Traceback (most recent call last):
...
OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')
>>> os.listdir()
[]
```