MODULE 2 – FILES

2.3 FILES

File handling is an important requirement of any programming language, as it allows us to store the data permanently on the secondary storage and read the data from a permanent source. Here, we will discuss how to perform various operations on files using the programming language Python.

2.3.1 Persistence

The programs that we have considered till now are based on console I/O. That is, the input was taken from the keyboard and output was displayed onto the monitor. When the data to be read from the keyboard is very large, console input becomes a laborious job. Also, the output or result of the program has to be used for some other purpose later, it has to be stored permanently. Hence, reading/writing from/to files are very essential requirement of programming.

We know that the programs stored in the hard disk are brought into main memory to execute them. These programs generally communicate with CPU using conditional execution, iteration, functions etc. But, the content of main memory will be erased when we turn-off our computer. We have discussed these concepts in Module1 with the help of Figure 1.1. Here we will discuss about working with secondary memory or files. The files stored on the secondary memory are permanent and can be transferred to other machines using pen-drives/CD.

2.3.2 Opening Files

To perform any operation on a file, one must open a file. File opening involves communication with operating system. In Python, a file can be opened using a built-in function ***open()***. While opening a file, we must specify the name of the file to be opened. Also, we must inform the OS about the purpose of opening a file, which is termed as *file opening mode*. The syntax of ***open()*** function is as below –

```
fhand= open("filename", "mode")
```

Here, `filename` is name of the file to be opened. This string may be just a name of the file, or it may include pathname also. Pathname of the file is optional when the file is stored in current working directory

`mode` This string indicates the purpose of opening a file. It takes a pre-defined set of values as given in Table 2.1

`fhand` It is a reference to an object of ***file*** class, which acts as a handler or tool for all further operations on files.

When our Python program makes a request to open a specific file in a particular mode, then OS will try to serve the request. When a file gets opened successfully, then a file object is returned. This is known as *file handle* and is as shown in Figure 2.1. It will help to

perform various operations on a file through our program. If the file cannot be opened due to some reason, then error message (*traceback*) will be displayed.

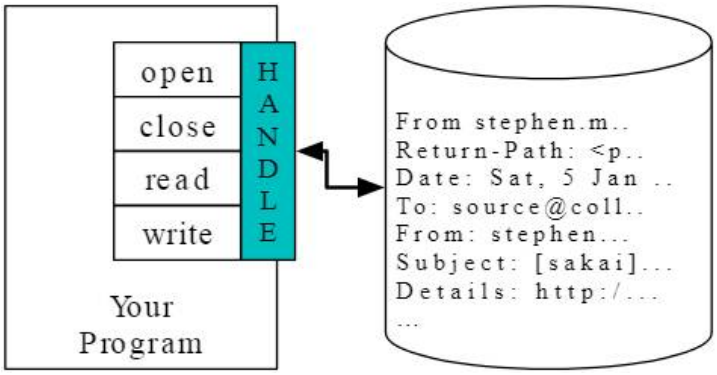

Figure 2.1 A File Handle

A file opening may cause an error due to some of the reasons as listed below –

- File may not exist in the specified path (when we try to read a file)
- File may exist, but we may not have a permission to read/write a file
- File might have got corrupted and may not be in an opening state

Since, there is no guarantee about getting a file handle from OS when we try to open a file, it is always better to write the code for file opening using *try-except* block. This will help us to manage error situation.

Mode	Meaning
r	Opens a file for reading purpose. If the specified file does not exist in the specified path, or if you don't have permission, error message will be displayed. This is the default mode of *open()* function in Python.
w	Opens a file for writing purpose. If the file does not exist, then a new file with the given name will be created and opened for writing. If the file already exists, then its content will be over-written.
a	Opens a file for appending the data. If the file exists, the new content will be appended at the end of existing content. If no such file exists, it will be created and new content will be written into it.
r+	Opens a file for reading and 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.
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.
rb	Opens a file for reading only in binary format
wb	Opens a file for writing only in binary format
ab	Opens a file for appending only in binary format

2.3.3 Text Files and Lines

A text file is a file containing a sequence of lines. It contains only the plain text without any images, tables etc. Different lines of a text file are separated by a newline character `\n`. In the text files, this newline character may be invisible, but helps in identifying every line in the file. There will be one more special entry at the end to indicate end of file (EOF).

NOTE: There is one more type of file called binary file, which contains the data in the form of bits. These files are capable of storing text, image, video, audio etc. All these data will be stored in the form of a group of bytes whose formatting will be known. The supporting program can interpret these files properly, whereas when opened using normal text editor, they look like messy, unreadable set of characters.

2.3.4 Reading Files

When we successfully open a file to read the data from it, the ***open()*** function returns the file handle (or an object reference to *file* object) which will be pointing to the first character in the file. A text file containing lines can be iterated using a for-loop starting from the beginning with the help of this file handle. Consider the following example of counting number of lines in a file.

NOTE: Before executing the below given program, create a text file (using Notepad or similar editor) *myfile.txt* in the current working directory (The directory where you are going store your Python program). Open this text file and add few random lines to it and then close. Now, open a Python script file, say *countLines.py* and save it in the same directory as that of your text file *myfile.txt*. Then, type the following code in Python script *countLines.py* and execute the program. (You can store text file and Python script file in different directories. But, if you do so, you have to mention complete path of text file in the *open()* function.)

Sample Text file *myfile.txt:*

```
hello how are you?
I am doing fine
what about you?
```

Python script file *countLines.py*

```
fhand=open('myfile.txt','r')
count =0
for line in fhand:
    count+=1
    print("Line Number ",count, ":", line)

print("Total lines=",count)
fhand.close()
```

Output:

```
Line Number 1 : hello how are you?
Line Number 2 : I am doing fine
```

```
Line Number 3 : what about you?
Total lines= 3
```

In the above program, initially, we will try to open the file `'myfile.txt`. As we have already created that file, the file handler will be returned and the object reference to this file will be stored in `fhand`. Then, in the for-loop, we are using `fhand` as if it is a sequence of lines. For each line in the file, we are counting it and printing the line. In fact, a line is identified internally with the help of new-line character present at the end of each line. Though we have not typed `\n` anywhere in the file `myfile.txt`, after each line, we would have pressed enter-key. This act will insert a `\n`, which is invisible when we view the file through notepad. Once all lines are over, `fhand` will reach end-of-file and hence terminates the loop. Note that, when end of file is reached (that is, no more characters are present in the file), then an attempt to read will return `None` or empty character `''` (two quotes without space in between).

Once the operations on a file is completed, it is a practice to close the file using a function *close()*. Closing of a file ensures that no unwanted operations are done on a file handler. Moreover, when a file was opened for writing or appending, closure of a file ensures that the last bit of data has been uploaded properly into a file and the end-of-file is maintained properly. If the file handler variable (in the above example, `fhand`) is used to assign some other file object (using *open()* function), then Python closes the previous file automatically.

If you run the above program and check the output, there will be a gap of two lines between each of the output lines. This is because, the new-line character `\n` is also a part of the variable `line` in the loop, and the *print()* function has default behavior of adding a line at the end (due to default setting of *end* parameter of *print()*). To avoid this double-line spacing, we can remove the new-line character attached at the end of variable `line` by using built-in string function *rstrip()* as below –

```
print("Line Number ",count, ":", line.rstrip())
```

It is obvious from the logic of above program that from a file, each line is read one at a time, processed and discarded. Hence, there will not be a shortage of main memory even though we are reading a very large file. But, when we are sure that the size of our file is quite small, then we can use ***read()*** function to read the file contents. This function will read entire file content as a single string. Then, required operations can be done on this string using built-in string functions. Consider the below given example –

```
fhand=open('myfile.txt')
s=fhand.read()
print("Total number of characters:",len(s))
print("String up to 20 characters:", s[:20])
```

After executing above program using previously created file *myfile.txt*, then the output would be –

```
Total number of characters:50
String up to 20 characters: hello how are you?
I
```

2.3.5 Writing Files

To write a data into a file, we need to use the mode ***w*** in *open()* function.

```
>>> fhand=open(“mynewfile.txt","w")
>>> print(fhand)
<_io.TextIOWrapper name='mynewfile.txt' mode='w' encoding='cp1252'>
```

If the file specified already exists, then the old contents will be erased and it will be ready to write new data into it. If the file does not exists, then a new file with the given name will be created.

The ***write()*** method is used to write data into a file. This method returns number of characters successfully written into a file. For example,

```
>>> s="hello how are you?"
>>> fhand.write(s)
     18
```

Now, the file object keeps track of its position in a file. Hence, if we write one more line into the file, it will be added at the end of previous line. Here is a complete program to write few lines into a file –

```
fhand=open('f1.txt','w')
for i in range(5):
     line=input("Enter a line: ")
     fhand.write(line+"\n")

fhand.close()
```

The above program will ask the user to enter 5 lines in a loop. After every line has been entered, it will be written into a file. Note that, as ***write()*** method doesn't add a new-line character by its own, we need to write it explicitly at the end of every line. Once the loop gets over, the program terminates. Now, we need to check the file `f1.txt` on the disk (in the same directory where the above Python code is stored) to find our input lines that have been written into it.

2.3.6 Searching through a File

Most of the times, we would like to read a file to search for some specific data within it. This can be achieved by using some string methods while reading a file. For example, we may be interested in printing only the line which starts with a character *h***.** Then we can use *startswith()* method.

```
fhand=open('myfile.txt')
for line in fhand:
    if line.startswith('h'):
        print(line)
fhand.close()
```

Assume the input file *myfile.txt* is containing the following lines –

hello how are you?
I am doing fine
how about you?

Now, if we run the above program, we will get the lines which starts with *h* –

hello how are you?
how about you?

2.3.7 Letting the User Choose the File Name

In a real time programming, it is always better to ask the user to enter a name of the file which he/she would like to open, instead of hard-coding the name of a file inside the program.

```
fname=input("Enter a file name:")
fhand=open(fname)

count =0
for line in fhand:
    count+=1
    print("Line Number ",count, ":", line)

print("Total lines=",count)
fhand.close()
```

In this program, the user input filename is received through variable `fname`, and the same has been used as an argument to `open()` method. Now, if the user input is *myfile.txt* (discussed before), then the result would be

```
Total lines=3
```

Everything goes well, if the user gives a proper file name as input. But, what if the input filename cannot be opened (Due to some reason like – file doesn't exists, file permission denied etc)? Obviously, Python throws an error. The programmer need to handle such run-time errors as discussed in the next section.

2.3.8 Using *try*, *except* to Open a File

It is always a good programming practice to write the commands related to file opening within a *try* block. Because, when a filename is a user input, it is prone to errors. Hence, one should handle it carefully. The following program illustrates this –

```
fname=input("Enter a file name:")
try:
    fhand=open(fname)
except:
```

```
        print("File cannot be opened")
        exit()

    count =0
    for line in fhand:
        count+=1
        print("Line Number ",count, ":", line)

    print("Total lines=",count)
    fhand.close()
```

In the above program, the command to open a file is kept within *try* block. If the specified file cannot be opened due to any reason, then an error message is displayed saying `File cannot be opened`, and the program is terminated. If the file could able to open successfully, then we will proceed further to perform required task using that file.

2.3.9 Debugging

While performing operations on files, we may need to extract required set of lines or words or characters. For that purpose, we may use string functions with appropriate delimiters that may exist between the words/lines of a file. But, usually, the invisible characters like white-space, tabs and new-line characters are confusing and it is hard to identify them properly. For example,

```
>>> s="1 2\t 3\n 4"
>>> print(s)
1 2   3
 4
```

Here, by looking at the output, it may be difficult to make out where there is a space, where is a tab etc. Python provides a utility function called as ***repr()*** to solve this problem. This method takes any object as an argument and returns a string representation of that object. For example, the *print()* in the above code snippet can be modified as –

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Note that, some of the systems use \n as new-line character, and few others may use \r (carriage return) as a new-line character. The ***repr()*** method helps in identifying that too.