File Handling

chapter 5

In This Chapter

5.1 Introduction

Most computer programs work with files. This is because files help in storing information permanently. Word processors create document files ; Database programs create files of information ; Compilers read source files and generate executable files. So, we see, it is the files that are mostly worked with, inside programs. *A file in itself is a bunch of bytes stored on some storage device like hard-disk, thumb-drive etc.* Every programming language offers some provision to use and create files through programs. Python is no exception and in this chapter, you shall be learning to work with data files through Python programs.

5.2 Data Files

The data files are the files that store data pertaining to a specific application, for later use. The data files can be stored in *two* ways :

- Text files
- Binary files

1. Text Files

A text file stores information in the form of **a stream of ASCII or Unicode characters** (the one which is default for your programming platform). In text files, each line of text is terminated, (delimited) with a special character (as per the Operating System) known as EOL (End of Line) character. In text files, some internal translations take place when this EOL character is read or written. In Python, by default, this EOL character is the newline character ('\n') or carriage-return, newline combination ('\r\n').

The text files can be of following types :

(*i*) **Regular Text files.** These are the text files which store the text in the same form as typed. Here the newline character ends a line and the text translations take place. These files have a file extension as **.txt**.

(*ii*) **Delimited Text files.** In these text files, a specific character is stored to separate the values, *i.e.*, after each value, *e.g.*, a *tab* or a *comma* after every value.

- When a *tab character* is used to separate the values stored, these are called **TSV files** (**Tab Separated Values** files). These files can take the extension as **.txt** or **.csv**.
- When the comma is used to separate the values stored, these are called **CSV files** (**Comma Separated Values** files). These files take the extension as **.csv**.

For instance, have a look at following example :

Regular text file content : I am simple text.

TSV file content : I → am → simple → text.

CSV file content : I, am, simple, text.

Some setup files (*e.g.*, **.INI files**) and rich text format files (**.RTF files**) are also text files.

NOTE

The CSV (*Comma Separated Values*) format is a popular import and export format for spreadsheets and databases.

Most commonly used delimiter in a CSV file is comma (,), but it can also use other delimiter characters like tab (→), pipe (|), tilde (~) etc.

2. Binary Files

A binary file stores the information in the form of **a stream of bytes.** A binary file contains information in the same format in which the information is held in memory, *i.e.*, the file content that is returned to you is raw (with no translation or no specific encoding). In binary file, there is no delimiter for a line. Also no translations occur in binary files. As a result, binary files are faster and easier for a program to read and write than are text files. As long as the file doesn't need to be read by people or need to be ported to a different type of system, binary files are the best way to store program information.

The binary files can take variety of extensions. Most non-text file extensions (any application defined extension) are binary files.

NOTE

The text files can be opened in any text editor and are in human readable form while binary files are not in human readable form.

5.3 Opening and Closing Files

In order to work with a file from within a Python program, you need to open it in a specific mode as per the file manipulation task you want to perform. The most basic file manipulation tasks include *adding*, *modifying* or *deleting data* in a file, which in turn include any one or combination of the following operations :

- reading data from files
- writing data to files
- appending data to files

Python provides built-in functions to perform each of these tasks. But before you can perform these functions on a file, you need to first open the file.

5.3.1 Opening Files

In data file handling through Python, the first thing that you do is open the file. It is done using *open( )* function as per one of the following syntaxes :

```
<file_objectname> = open(<filename>)
<file_objectname> = open(<filename>, <mode>)
```

For example, `myfile = open("taxes.txt")` ← *Python will look for this file in current working directory*

The above statement opens file **"taxes.txt"** in **file mode** as ***read mode*** (default mode) and attaches it to ***file object namely myfile***. Consider another statement :

```
file2 = open("data.txt", "r")
```

The above statement opens the file **"data.txt"** in ***read mode*** (because of "r" given as mode) and attaches it to file object namely ***file2***. Consider one more file-open statement :

```
file3 = open("e:\\main\\result.txt", "w")
```
← *Python will look for this file in E:\main folder*

The above statement opens file **"result.txt"** (stored in folder **E:\ main**) in write mode (because of "w" given as mode) and attaches it to file object namely ***file3***.

(Notice that in above file open statement, the file path contains double slashes in place of single slashes.)

NOTE

A ***file-object*** ***is*** also known as ***file-handle***.

The above given *three* file-open statements must have raised these questions in your mind :

(*i*) *What is file-object* ? (*ii*) *What is mode or file-mode* ?

The coming lines will have answers to all your questions, but for now, let us summarize the file *open( )* function.

- Python's *open( )* function creates a *file object* which serves as a link to a file residing on your computer.
- The first parameter for the *open( )* function is a path to the file you'd like to open. If just the file name is given, then Python searches for the file in the current folder.
- The second parameter of the open function corresponds to a mode which is typically read ('r'), write ('w'), or append ('a'). If no second parameter is given, then by default it opens it in read ('**r**') *mode*.

NOTE

Please note that when you open a file in *readmode*, the given file must exist in the folder, otherwise Python will raise **FileNotFoundError**.

Figure 5.1 summarizes the *open( )* function of Python for you.

As you can see in Fig. 5.1, the slashes in the path are doubled. This is because the slashes have special meaning and to suppress that special meaning escape sequence for slash *i.e.*, \\ is given.

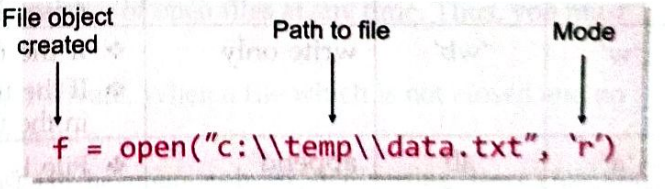


Figure 5.1 Working of file open( ) function.

However, if you want to write with single slash, you may write in raw string as :

with r here, you can give single slashes in pathnames

```
f = open(r"c :\temp\data.txt", "r")
```

The prefix ***r*** in front of a string makes it ***raw string*** that means there is no special meaning attached to any character. Remember, following statement might give you incorrect result :

```
f = open("c :\temp\data.txt", "r")
```
← *This might give incorrect result as \t is tab character*

NOTE

The prefix ***r*** in front of a string makes it ***raw string*** that means there is no special meaning attached to any character.

Reason being that '\t' will be treated as **tab character** and '\d' on some platforms as numeric-digit or some error.

Thus the *two* ways to give paths in filenames correctly are :

(*i*) Double the slashes *e.g.*,

```
f = open("c:\\temp\\data.txt", "r")
```

(*ii*) Give **raw string** by prefixing the file-path string with an *r e.g.*,

```
f = open(r"c:\temp\data.txt", "r")
```

NOTE

The default file-open mode is read mode, *i.e.*, if you do not provide any file open mode, Python will open it in read mode ("r").

NOTE

If you just give the file name without its path (*e.g.*, just "data.txt" rather than "c :\\temp\\data.txt"), Python will open/create the file in the same directory in which the module file is stored.

5.3.1A File Object/File Handle

File objects are used to read and write data to a file on disk. The file object is used to obtain a reference to the file on disk and open it for a number of different tasks.

File object (also called ***file-handle***) is very important and useful tool as through a *file-object* only, a Python program can work with files stored on hardware. All the functions that you perform on a data file are performed through file-objects.

When you use file **open( )**, Python stores the reference of mentioned file in the file-object. A file-object of Python is a stream of bytes where the data can be read either byte by byte or line by line or collectively. All this will be clear to you in the coming lines (under topic – *File access modes*).

FILE OBJECT

A **file object** is a reference to a file on disk. It opens and makes it available for a number of different tasks.

5.3.1B File Access Modes

When Python opens a file, it needs to know the file-mode in which the file is being opened. A file-mode governs the type of operations (such as *read* or *write* or *append*) possible in the opened file *i.e.*, it refers to how the file will be used once it's opened. File modes supported by Python are being given in Table 5.1.

Table 5.1 *File-modes*

Text File Mode	Binary File Mode	Description	Notes
'r'	'rb'	read only	❖ **Default mode** ; File must exist already, otherwise Python raises I/O error.
'w'	'wb'	write only	❖ If the file does not exist, file is created. ❖ If the file exists, Python will truncate existing data and over-write in the file. So this mode must be used with caution.
'a'	'ab'	append	❖ File is in **write only** mode. ❖ If the file exists, the data in the file is retained and new data being written will be appended to the end. ❖ If the file does not exist, Python will create a new file.
'r+'	'r+b' or 'rb+'	read and write	❖ File must exist otherwise error is raised. ❖ Both reading and writing operations can take place.
'w+'	'w+b' or 'wb+'	write and read	❖ File is created if does not exist. ❖ If file exists, file is truncated (past data is lost). ❖ Both reading and writing operations can take place.
'a+'	'a+b' or 'ab+'	write and read	❖ File is created if does not exist. ❖ If file exists, file's existing data is retained ; new data is appended. ❖ Both reading and writing operations can take place.

Adding a *'b'* to *text-file mode* makes it *binary-file mode*.

To create a file, you need to open a file in a mode that supports *write* mode (*i.e.*, 'w', or 'a' or 'w+' or 'a+' modes).

> **NOTE**
>
> A **file-mode** governs the type of operations (*e.g.*, read/write/append) possible in the opened file *i.e.*, it refers to how the file will be used once it's opened.

5.3.2 Closing Files

An open file is closed by calling the ***close( )*** method of its file-object. Closing of file is important. In Python, files are automatically closed at the end of the program but it is good practice to get into the habit of closing your files explicitly. Why ? Well, the operating system may not write the data out to the file until it is closed (this can boost performance). What this means is that if the program exits unexpectedly there is a danger that your precious data may not have been written to the file! So the moral is : once you finish writing to a file, close it.

The *close( )* function accomplishes this task and it takes the following general form :

```
<fileHandle>.close( )
```

> **NOTE**
>
> A close( ) function breaks the link of file-object and the file on the disk. After close( ), no tasks can be performed on that file through the file-object (or file-handle).

For instance, if a file **Master.txt** is opened *via* file-handle *outfile*, it may be closed by the following statement :

```
outfile.close( )
```

← *The close( ) must be used with filehandle*

Please remember, **open( )** is a built-in function (used stand-alone) while **close( )** is a method used with file-handle object.

Info Box 5.1

Why should you close files in your program ?

You know that you can close an open file using the close( ) function. What if you don't use close( ) function on the file handle? Well, in that case Python will eventually close your file anyway. But then why is it recommended every time that one should close the open file using close( ) ? The reasons for this are :

(i) Closing files using close() is a good programming practice.

(ii) Sometimes the data written onto files using write functions is held in memory until the next write or the file is closed. So for the last write operation, the data may remain in memory until close() is used. This may lead to data loss sometimes.

(iii) Sometimes there are operating system restrictions on number of open files at any time. Thus, you must close your file when its work is done.

(iv) Some operating systems treat open files as locked and private. When a file which is not closed and no longer in use, it leads to unnecessary blockage of memory.

For all the above reasons, it is advised that files should be closed in the programs by explicitly using close( ) function.

5.4 Working with Text Files

Python provides many functions for reading and writing the open files. In this section, we are going to explore these functions. Most common file reading and writing functions are being discussed in coming lines.

5.4.1 Reading from Text Files

Python provides mainly *three* types of *read functions* to read from a data file. But before you can read from a file, the file must be opened and linked via a ***file-object*** or ***file handle***. Most common file reading functions of Python are listed below in Table 5.2.

Table 5.2 *Python data files — reading writing functions*

S.No.	Method/Syntax	Description

1.	read( )
	
	Syntax
	<filehandle>.read([n])
		reads at most *n* bytes ; if no *n* is specified, reads the entire file. Returns the read bytes in the form of a *string*.
		`>>> file1 = open("E:\\mydata\\info.txt")`
		`>>> readInfo = file1.read(15)`
		`>>> print(readInfo)`
		`It's time to wo`
		15 bytes read
		`>>> type(readInfo)`
		`str`
		Bytes read into **string** type

2.	readline( )
	
	Syntax
	<filehandle>.readline([n])
		reads a line of input ; if *n* is specified reads at most *n* bytes. Returns the read bytes in the form of a *string* ending with line character ('\n') or returns a blank string if no more bytes are left for reading in the file.
		`>>> file1 = open("E:\\mydata\\info.txt")`
		`>>> readInfo = file1.readline()`
		`>>> print(readInfo)`
		`It's time to work with files.`
		1 line read
		If you check the size of *readInfo* using **len( )** funcion, you will find that it has one byte extra than the number of characters indicating the presence of '\n'.

3.	readlines( )
	
	Syntax
	<filehandle>.readlines()
		reads all lines and returns them in a *list*
		`>>> file1 = open("E:\\mydata\\info.txt")`
		`>>> readInfo = file1.readline()`
		`>>> print(readInfo)`
		`["It's time to work with files.\n", 'Files offer and ease and power to store your work/data/information for later use.\n', 'simply create a file and store(write) in it .\n', 'Or open an existing file and read from it.\n']`
		All lines read
		`>>> type(readInfo)`
		`List`
		Read into **list** type

The <filehandle> in above syntaxes is the file-object holding open file's reference.

Let us consider some examples now.

For the examples and explanations below, we are using a file namely ***poem.txt*** storing the content shown in Fig. 5.2.

WHY ?

We work, we try to be better
We work with full zest
But, why is that we just don't know any letter.

We still give our best.
We have to steal,
But, why is that we still don't get a meal.

We don't get luxury,
We don't get childhood,
But we still work,
Not for us, but for all the others.

Why is it that some kids wear shoes, BUT we make them ?

by Mythili, class 5

Figure 5.2 Contents of a sample file *poem.txt*.

Code Snippet 1 *Reading a file's first 30 bytes and printing it*

File-object created → `myfile = open(r'E:\poem.txt', "r")`

`str = myfile.read( 30 )` ← *See the number of bytes to be read specified as argument of read( )*

File-object being used → `print(str)`

When we specify number of bytes with read( ), Python will read only the specified number of bytes from the file. The next read( ) will start reading onwards from the last position read. ***Code snippet 2*** illustrates this fact.

The output produced by above code is :

```
>>>
    WHY?
We work, we try
```

TIP

You may combine the file( ) or open( ) with the file-object's function, if you need to perform just a single task on the open file.

If need to perform just a single task with the open file, then you may combine the two steps of opening the file and performing the task, *e.g.*, following statement :

```
>>> file(r"E :\poem.txt", "r").read(30)
```

← *First function will open the file and the second function will perform with the result of first function, i.e., the reference of open file*

would give the same result as that of above code.

For example, the following code will return the first line of file *poem.txt* :

```
open("poem.txt", "r").readline()
```

Code Snippet 2 *Reading n bytes and then reading some more bytes from the last position read*

```
myfile = open(r'E:\poem.txt', 'r')
str = myfile.read( 30 )
print(str)
str2 = myfile.read( 50 )
print(str2)
myfile.close()
```

← *reading 30 bytes*

← *reading next 50 bytes*

To see **reading from file** in action

Scan QR Code

The output produced by above code is :

```
>>>
    WHY?
We work, we try

  to be better
We work with full zest
But, why is t
```

← *Output by first print statement, i.e., **print(str)***

← ***print** has entered a new line after its output i.e., here*

← *Output by second print statement, i.e., **print(str2)***

If you do not want to have the newline characters inserted by print statement in the output then use **end = ' '** argument in the end of *print( )* statement so that print does not put any additional newline characters in the output. Refer to ***code snippets 3 and 4*** below.

Code Snippet 3 *Reading a file's entire content*

```
myfile = open(r'E :\poem.txt', "r")
str = myfile.read()
print(str)
myfile.close()
```

See when no value is specified as read( )'s argument, entire file is read

The output produced by above code is :

```
>>>
    WHY?
We work, we try to be better
We work with full zest
But, why is that we just don't know any letter.

We still give our best.
We have to steal,
But, why is that we still don't get a meal.

We don't get luxury,
We don't get childhood,
But we still work,
Not for us, but for all the others.

Why is it that some kids wear shoes, BUT we make them ?

by      Mythili, class 5
```

NOTE

You can also combine the **open( )** and **read( )** functions as :

`file("filename", <mode>).read( )`

Solved problem 6 is based on this.

Code Snippet 4 *Reading a file's first three lines - line by line*

```
myfile = open(r'E:\poem.txt', "r")
str = myfile.readline()
print(str, end = ' ')
str = myfile.readline()
print(str, end = ' ')
str = myfile.readline()
print(str, end = ' ')
myfile.close()
```

*Argument **end = ' '** at the end of print( ) statement will ensure that output shows exact content of the data file and no print-inserted newline characters are*

The *readline( )* function will read a full line. A line is considered till a newline character (EOL) is encountered in the data file.

The output produced by above code is :

```
>>>
    WHY?
We work, we try to be better
```

Line 1

Line 2

Line 3

Code Snippet 5 *Reading a complete file - line by line*

```
myfile = open(r'E:\poem.txt', "r")
str = " "              #initially storing a space (any non-None value)
while str :
    str = myfile.readline( )
    print(str, end = ' ')
myfile.close( )
```

The output produced by the above code will print the entire content of file **poem.txt.**

```
>>>
    WHY?

We work, we try to be better
We work with full zest
But, why is that we just don't know any letter.

We still give our best.
We have to steal,
But, why is that we still don't get a meal.

We don't get luxury,
We don't get childhood,
But we still work,
Not for us, but for all the others.

Why is it that some kids wear shoes, BUT we make them ?

by      Mythili, class 5
```

The ***readline( )*** function reads the leading and trailing spaces (if any) along with trailing newline character('\n') also while reading the line. You can remove these leading and trailing white spaces (spaces or tabs or newlines) using ***strip( )*** (without any argument) as explained below. Recall that ***strip( )*** without any argument removes leading and trailing whitespaces.

There is another way of printing a file line by line. This is a simpler way where after opening a file you can browse through the file using its file handle line by line by writing following code :

```
<filehandle> = open(<filename>, [<any read mode>])
for <var> in <filehandle> :
    print(<var>)
```

For instance, for the above given file *poem.txt*, if you write following code, it will print the entire file line by line :

```
myfile = open(r'E:\poem.txt', "r")
for line in myfile :
    print(line)
```

The output produced by above code is just the same as the output produced by above program. The reason behind this output is that when you iterate over a file-handle using a for loop, then

the for loop's variable moves through the file line by line where a line of a file is considered as a sequence of characters up to and including a special character called the newline character (\n). So the *for loop's* variable starts with first line and with each iteration, it moves to the next line. As the *for loop* iterates through each line of the file the loop variable will contain the current line of the file as a string of characters.

Code Snippet 6 *Displaying the size of a file after removing EOL characters, leading and trailing white spaces and blank lines*

```
myfile = open(r'E:\poem.txt', "r")
str1 = " "                  #initially storing a space (any non-None value)
size = 0
tsize = 0
while str1 :
    str1 = myfile.readline()
    tsize = tsize + len(str1)
    size = size + len(str1.strip())
print("Size of file after removing all EOL characters & blank lines:", size)
print("The TOTAL size of the file:", tsize)
myfile.close()
```

The output produced by the above code fragment is :

```
Size of file after removing all EOL characters & blank lines : 360
The TOTAL size of the file : 387
```

All the above code fragments read the contents from file in a string. However, if you use *readlines( )* (notice 's' in the end of the function), then the contents are read in a List. Go through next code fragment.

Code Snippet 7 *Reading the complete file-in a list*

```
myfile = open(r'E:\poem.txt', "r")
s = myfile.readlines()
print(s)
myfile.close()
```

*notice it is **readlines( )** not **readline( )***

Now carefully look at the output. The ***readlines( )*** has read the entire file in a list of strings where each line is stored as one string :

```
['  WHY?\n', '\n', 'We work, we try to be better\n', 'We work with
full zest\n', "But, why is that we just don't know any letter.\n",
'\n', 'We still give our best.\n', 'We have to steal,\n', "But, why
is that we still don't get a meal.\n", '\n', "We don't get luxury,
\n", "We don't get childhood,\n", 'But we still work,\n', 'Not for
us, but for all the others.\n', '\n', 'Why is it that some kids wear
shoes, BUT we make them ?\n', '\n', 'by     Mythili, class 5\n']
```

Now that you are familiar with these reading functions' working, let us write some programs.

Program 5.1 Write a program to display the size of a file in bytes.

```
myfile = open(r'E:\poem.txt', "r")
str = myfile.read()
size = len(str)
print("Size of the given file poem.txt is")
print(size, "bytes")
```

Output

```
>>>
Size of the given file poem.txt is
387 bytes
```

Program 5.2 Write a program to display the number of lines in the file.

```
myfile = open(r'E:\poem.txt', "r")
s = myfile.readlines()
linecount = len(s)
print("Number of lines in poem.txt is", linecount)
myfile.close()
```

Output

```
>>>
Number of lines in poem.txt is 18
```

5.4.2 Writing onto Text Files

After working with file-reading functions, let us talk about the writing functions for data files available in Python. (See Table 5.3). Like reading functions, the writing functions also work on open files, *i.e.*, the files that are opened and linked via a *file-object* or *file-handle*.

Table 5.3 *Python Data Files – Writing Functions*

S.No.	Name	Syntax	Description
1.	write( )	`<filehandle>.write(str1)`	writes string *str1* to file referenced by ***<filehandle>***
2.	writelines( )	`<filehandle>.writelines(L)`	writes all strings in list **L** as lines to file referenced by ***<filehandle>***
	The `<filehandle>` in above syntaxes is the *file-object* holding open file's reference		

Appending a File

When you open a file in "**w**" or ***write mode***, Python overwrites an existing file or creates a non-existing file. That means, for an existing file with the same name, the earlier data gets lost. If, however, you want to write into the file while retaining the old data, then you should open the file in "**a**" or ***append mode***. A file opened in *append mode* retains its previous data while allowing you to add newer data into. You can also add a plus symbol (+) with file read mode to facilitate reading as well as writing.

That means, in Python, writing in files can take place in following forms :

(*i*) In an existing file, while retaining its content

(*a*) if the file has been opened in append mode ("a") to retain the old content.

(*b*) if the file has been open in 'r+' or 'a+' modes to facilitate reading as well as writing.

(*ii*) to create a new file or to write on an existing file after truncating/ overwriting its old content

(*a*) if the file has been opened in write-only mode ("w")

(*b*) if the file has been open in 'w+' mode to facilitate writing as well as reading

(*iii*) Make sure to use **close( )** function on ***file-object*** after you have finished writing as sometimes, the content remains in memory buffer and to force-write the content on file and closing the link of ***file-handle*** from file, *close( )* is used.

Let us consider some examples now.

Code Snippet 8 *Create a file to hold some data*

```
fileout = open("Student.dat", "w")
for i in range(5) :
    name = input("Enter name of student :")
    fileout.write(name)
fileout.close()
```

The write() will simply write the content in file without adding any extra character.

It's important to use close()

The sample run of above code is as shown below :

```
>>>
Enter name of student : Riya
Enter name of student : Rehan
Enter name of student : Ronaq
Enter name of student : Robert
Enter name of student : Ravneet
```

Now you can see the file created in the same folder where this Python script/program is saved (see Fig. 5.3(*a*) below). However, if you want to create the file in specific folder then you must specify the file-path as per the guidelines discussed earlier.

Also, you can open the created file ("student.dat" in above case) in *Notepad* to see its contents. Fig. 5.3(*b*) shows the contents of file created through *code snippet 5*.

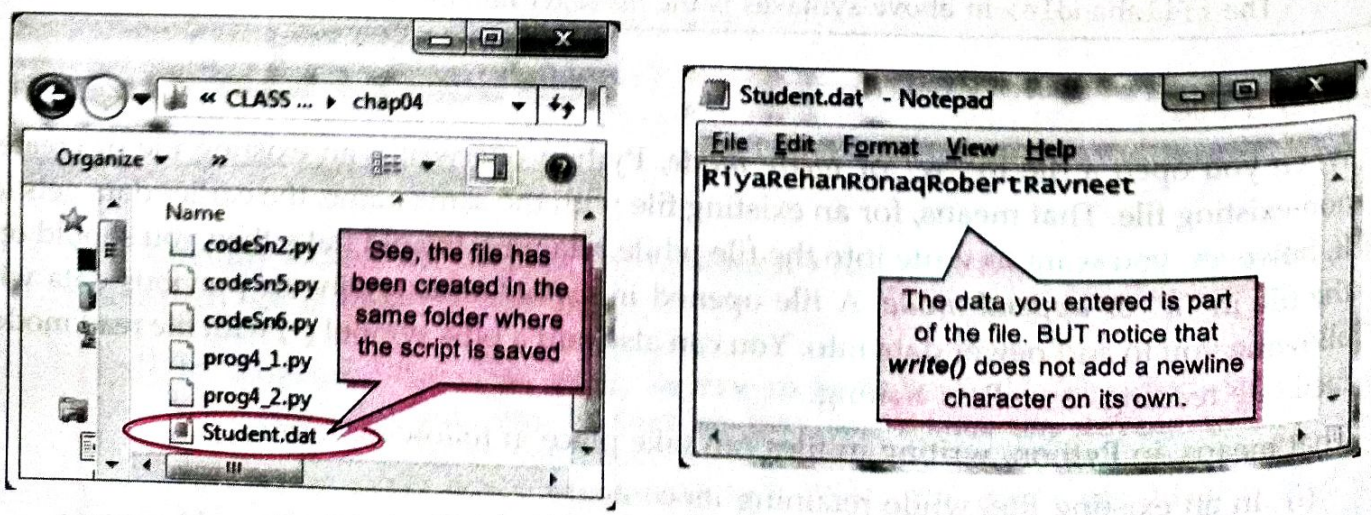


Figure 5.3 (*a*) File created through open( ) with "w" mode is stored in the same folder as that of script file (*b*) The write( ) function does not add any extra character in the file

IMPORTANT

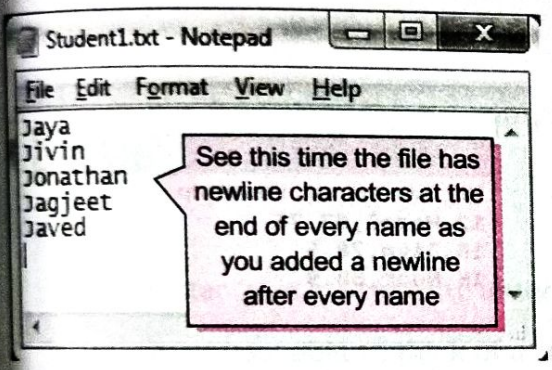
Carefully look at Fig. 5.3(*b*) that is showing contents of a file created through **write( )**. It is clear from the file-contents that ***write( )*** does not add any extra character like newline character('\n') after every write operation. Thus to group the contents you are depicted in ***code snippet 9***.

Code Snippet 9 *Create a file to hold some data, separated as lines*

(This code is creating a different file than created with code snippet 8)

```
fileout = open("Student1.txt", "w")
for i in range(5) :
    name = input("Enter name of student :")
    fileout.write(name)
    fileout.write('\n')
fileout.close()
```

The newline character '\n' written after every name



The sample run of the above code is as shown below :

```
>>>
Enter name of student : Jaya
Enter name of student : Jivin
Enter name of student : Jonathan
Enter name of student : Jagjeet
Enter name of student : Javed
```

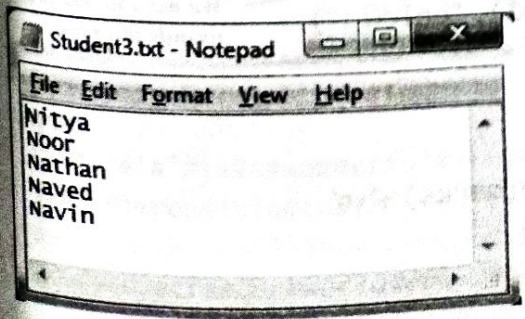
The file created by ***code snippet 9*** is shown above.

Code snippet 10 *Creating a file with some names separated by newline characters without using write( ) function.*

*(For this, we shall use **writelines( )** inplace of **write( )** function which writes the content of a list to a file. Function writelines( ) also, does not add newline character, so you have to take care of adding newlines to your file.)*

```
fileout = open("Student3.txt", "w")
List1 = []
for i in range(5) :
    name = input("Enter name of student :")
    List1.append(name + '\n')
fileout.writelines(List1)
fileout.close()
```

Responsibility to add newline character is of programmer's



Sample run of the above code is as shown below :

```
>>>
Enter name of student : Nitya
Enter name of student : Noor
Enter name of student : Nathan
Enter name of student : Naved
Enter name of student : Navin
```

Now that you are familiar with these writing functions' working, let us write some programs based on the same.

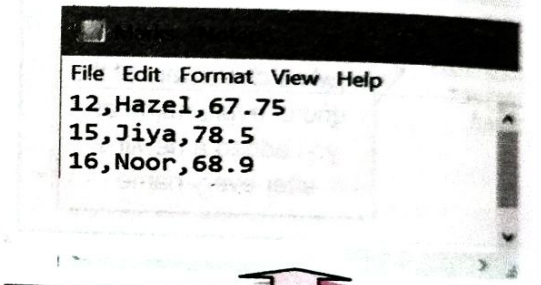
Program 5.3 Write a program to get roll numbers, names and marks of the students of a class (get from user) and store these details in a file called "Marks.txt".

```
count = int(input("How many students are there in the class?"))
fileout = open ("Marks.txt", "w")
for i in range(count) :
    print("Enter details for student", (i+1), "below:")
    rollno = int(input("Rollno:))"
    name = input("Name :")
    marks = float(input("Marks:"))
    rec = str(rollno) + "," + name + "," + str(marks) +'\n'
    fileout.write(rec)
fileout.close()
```

Joining individual information by adding commas in between

Output

```
>>>
How many students are there in the class? 3
Enter details for student 1 below :
Rollno : 12
Name : Hazel
Marks : 67.75
Enter details for student 2 below :
Rollno : 15
Name : Jiya
Marks : 78.5
Enter details for student 3 below :
Rollno : 16
Name : Noor
Marks : 68.9
```



File created by above program (program 5.3)

Figure 5.4 File created through program.

If you carefully notice, we have created comma separated values in one student record while writing in file. So we can say that the file created by program 5.4 is in a format similar to CSV (comma separated values) format or it is a delimited file where comma is the delimiter.

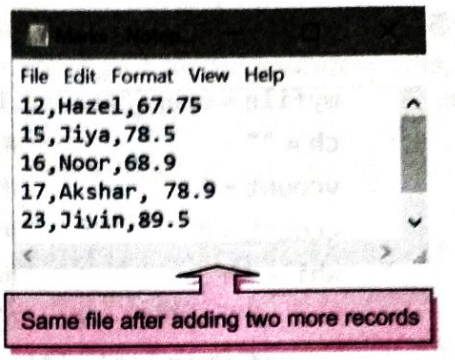
Program 5.4 Write a program to add two more students' details to the file created in program 5.3.

```
fileout = open ("Marks.txt", "a")
for i in range(2) :
    print("Enter details for student", (i+1), "below :")
    rollno = int(input("Rollno:"))
    name = input("Name :")
    marks = float(input("Marks:"))
    rec = str(rollno) + "," + name + "," + str(marks) +'\n'
    fileout.write(rec)
fileout.close()
```

Notice the file is opened in append mode ("a") this time so as to retain old content

We want to add two records this time

```
>>>
Enter details for student 1 below :
Rollno : 17
Name : Akshar
Marks : 78.9
Enter details for student 2 below :
Rollno : 23
Name : Jivin
Marks : 89.5
```



Same file after adding two more records

5.5 Write a program to display the contents of file "Marks.txt" created through programs 5.3 and 5.4.

Program

```
fileinp = open("Marks.txt", "r")
while str :
    str = fileinp.readline()
    print(str)
fileinp.close()
```

Output

```
>>>
12,Hazel,67.75
15,Jiya,78.5
16,Noor,68.9
17,Akshar,78.9
23,Jivin,89.5
```

Have a look at some more examples of working with text files where we are using the following text file (Fig. 5.5)

File : *Answer.txt*

Letter 'a' is a wonderful letter.

It is impossible to think of a sentence without it.

We know this will never occur.

How mediocre our world would be without this single most powerful letter.

Figure 5.5 The contents of file Answer.txt.

5.6 Write a program to read a text file line by line and display each word separated by a '#'.

Program

```
myfile = open("Answer.txt", "r")
line = ""                          #initially stored a space (a non-None value)
while line :
    line = myfile.readline()  # one line read from file
    # printing the line word by word using split()
    for word in line.split():
        print(word, end = '#')
    print()
#close the file
myfile.close()
```

Output

```
Letter#'a'#is#a#wonderful#letter.#
It#is#impossible#to#think#of#a#sentence#without#it.#
We#know#this#will#never#occur.#
How#mediocre#our#world#would#be#without#this#single#most#powerful#letter.#
```

Program 5.7 Write a program to read a text file and display the count of vowels and consonants in the file.

```
myfile = open("Answer.txt", "r")
ch = ""                    #initially stored a space (a non-None value)
vcount = 0                 #variable to store count of vowels
ccount = 0                 #variable to store count of consonants
while ch :                 #while ch stores a Non-None value
    ch = myfile.read(1)    #one character read from file
    if ch in ['a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U']:
        vcount = vcount + 1
    else :
        ccount = ccount + 1
print("Vowels in the file :", vcount)
print("Consonants in the file :", ccount)
# close the file
myfile.close()
```

5.4.3 The flush( ) Function

When you write onto a file using any of the write functions, Python holds everything to write in the file in buffer and pushes it onto actual file on storage device a later time. If however, you want to force Python to write the contents of buffer onto storage, you can use **flush( )** function.

Python automatically flushes the file buffers when closing them ***i.e.,*** this function is implicitly called by the **close( )** function. But you may want to flush the data before closing any file. The syntax to use flush( ) function is :

```
<fileObject>.flush()
```

FLUSH( )

The flush( ) function forces the writing of data on disc still pending in *output buffer*.

Consider the following example code :

```
f = open( 'out.log', 'w+')
f.write( 'The output is \n')
f.write( "My" + "work-status "+" is ")
f.flush()
s = 'OK.'
f.write(s)
f.write( '\n')
# some other work
f.write( 'Finally Over\n')
f.flush()
f.close()
```

With this statement, the strings written so far, i.e., 'The output is' and 'My work-status is' have been pushed on to actual file on disk.

These write statements' strings may still be pending to be written on to disk

The flush( ) function ensures that whatever is held in Output buffer, is written on to the actual file on disk

5.4.4 Removing Whitespaces after Reading from File

The ***read( )*** and ***readline( )*** functions discussed above, read data from file and return it in string form and the ***readlines( )*** function returns the entire file content in a list where each line is one item of the list.

All these read functions also read the leading and trailing whitespaces *i.e.*, spaces or tabs or newline characters. If you want to remove any of these trailing and leading whitespaces, you can use strip( ) functions [rstrip( ), lstrip( ) and strip( )] as shown below.

Recall that :

- the *strip*( ) removes the given character from both ends.
- the *rstrip*( ) removes the given character from trailing end *i.e., right end.*
- the *lstrip*( ) removes the given character from leading end *i.e., left end.*

To understand this, consider the following examples :

1. **Removing EOL '\n' character from the line read from the file.**

```
fh = file("poem.txt, "r")
line = fh.readline()
line = line.rstrip('\n')
```

Will remove the end of line ***newline*** *character '\n'*

```
>>> fh = file("e :\\poem.txt", "r")
>>> line = fh.readline()
>>> len(line)
14

>>> line2 = line.rstrip('\n')
>>> len(line2)
13

>>> line3 = line.strip()
>>> len(line3)
4
```

2. **Removing the leading whilespaces from the line read from the file**

```
line = file("poem.txt, "r").readline()
line = line.lstrip()
```

Now can you justify the output of following code that works with first line of file *poem.text* shown above where first line containing leading 8 spaces followed by word 'WHY?' and a '\n' in the end of line.

Info Box 5.2

Steps to Process a File

Following lines list the steps that need to be followed in the order as mentioned below. The five steps to use files in your Python program are :

1. Determine the type of file usage

Under this step, you need to determine whether you need to open the file for reading purpose (input type of usage) or writing purpose (output type of usage). If the data is to be brought in from a file to memory, then the file must be opened in a mode that supports reading such as "r" or "r+" etc.

Similarly, if the data (after some processing) is to be sent from memory to file, the file must be opened in a mode that supports writing such as "w" or "w+" or "a" etc.

2. Open the file and assign its reference to a *file-object* or file-handle

Next, you need to open the file using **open( )** and assign it to a file-handle on which all the file-operations will be performed. Just remember to open the file in the file-mode that you decided in step 1.

3. Now process as required

As per the situation, you need to write instructions to process the file as desired. For example, you might need to open the file and then read it one line at a time while making some computation, and so on.

4. Close the file

This is very important step especially if you have opened the file in write mode. This is because, sometimes the last lap of data remains in buffer and is not pushed on to disk until a close( ) operation is performed.

5.4.5 Significance of File Pointer in File Handling

Every file maintains a *file pointer* which tells the current position in the file where writing or reading will take place. (A file pointer in this context works like a book-mark in a book).

Whenever you read something from the file or write onto a file, then these *two* things happen involving file-pointer :

(*i*) this operation takes place at the position of *file-pointer* and

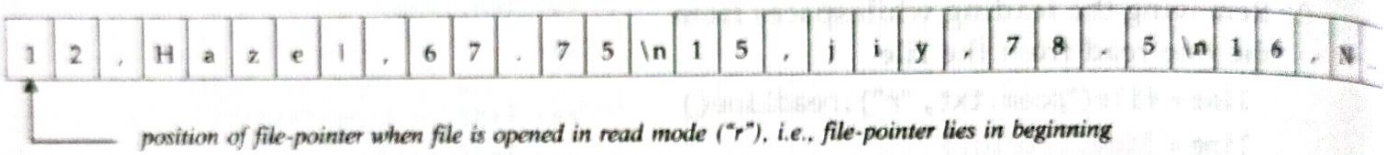
(*ii*) *file-pointer* advances by the specified number of bytes

To see working of file pointer in action

Figure 5.6 illustrates this process. It is important to understand how a *file pointer* works in Python files. The working of file-pointer has been described in Fig. 5.6.

1. `fh = open("Marks.txt, "r")`

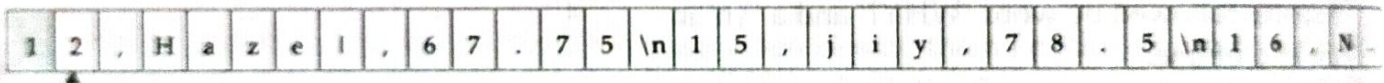
will open the file and place the file-pointer at the beginning of the file (see below)



position of file-pointer when file is opened in read mode ("r"), i.e., file-pointer lies in beginning

2. `ch = fh.read(1)`

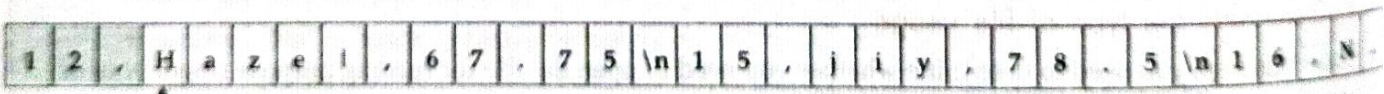
will read 1 byte from the file from the position the file-pointer is currently at ; and the file pointer advances by one byte. That is, now the ***ch*** will hold '1' and the file pointer will be at next byte holding value '2' (see below)



See now, the file-pointer has advanced by 1 position as previous read operation read 1 byte only. Now the next read will take place at the current position of file-pointer.

3. `str = fh.read(2)`

will read 2 bytes from the file from the position the file-pointer is currently at and the file pointer advances by 2 bytes. That is, now the ***str*** will hold '2', and the file pointer will be at next byte holding value 'H' (see below)



See now, the file-pointer has advanced by 2 positions as previous read operation read 2 bytes only. Now the next read will take place at the current position of file-pointer.

Figure 5.6 **Working of a file-pointer.**

5.4.5A File Modes and the Opening Position of File-Pointer

The position of a file-pointer is governed by the *filemode* it is opened in. Following table lists the opening position of a file-pointer as per *filemode*.

Table 5.4 *File modes and opening position of file-pointer*

File Modes	Opening position of file-pointer
r, rb, r+, rb+, r+b	*beginning of the file*
w, wb, w+, wb+, w+b	*beginning of the file* (Overwrites the file if the file exists).
a, ab, a+, ab+, a+b	*at the end of the file* if the file exists otherwise creates a new file.

5.5 Standard Input, Output and Error Streams

If someone asks you to give input to a program interactively or by typing, you know what device you need for it – ***the keyboard***. Similarly, if someone says that the output is to be displayed, you know which device it will be displayed on – ***the monitor***. So, we can safely say that the ***Keyboard*** is the **standard input device** and the ***monitor*** is **standard output device**. Similarly, any error if occurs is also displayed on the monitor. So, *the* ***monitor*** is also **standard error device**. That is,

- standard input device (**stdin**) – reads from the keyboard
- standard output device (**stdout**) – prints to the display and can be redirected as standard input.
- standard error device (**stderr**) – Same as *stdout* but normally only for errors. Having error output separately allows the user to divert regular output to a file and still be able to read error messages.

Do you know internally how these devices are implemented in Python? These standard devices are implemented as files called *standard streams*. In Python, you can use these standard stream files by using **sys** module. After importing, you can use these standard streams (**stdin**, **stdout** and **stderr**) in the same way as you use other files.

Interesting : Standard Input, Output Devices as Files

If you import **sys** module in your program then, **sys.stdin.read( )** would let you read from keyboard. This is because the keyboard is the *standard input device* linked to **sys.stdin**. Similarly, **sys.stdout.write( )** would let you write on the standard output device, *the monitor*. **sys.stdin** and **sys.stdout** are standard input and standard output devices respectively, treated as files.

Thus **sys.stdin** and **sys.stdout** are like files which are opened by the Python when you start Python. The **sys.stdin** is always opened in read mode and **sys.stdout** is always opened in write mode. Following code fragment shows you interesting use of these. It prints the contents of a file on monitor without using **print** statement :

These statements would write on file/device associated with sys.stdout, which is the monitor

```
import sys
fh = open(r"E:\poem.txt")
line1 = fh.readline()
line2 = fh.readline()
sys.stdout.write(line1)
sys.stdout.write(line2)
sys.stderr.write("No errors occurred\n")
```

Output produced is :

```
>>> ========================
>>>
        WHY ?
We work, we try to be better
No errors occurred
>>>
```

*See, **stderr** also displayed its text on monitor.*

Info Box 5.3
Python Data Files : with Statement

Python's **with** statement for files is very handy when you have two related operations which you'd like to execute as a pair, with a block of code in between. The syntax for using with statement is :

```
with open(<filename>, <filemode>) as <filehandle> :
    <file manipulation statements>
```

The classic example is opening a file, manipulating the file, then closing it :

```
with open('output.txt', 'w') as f:
    f.write('Hi there!')
```

The above **with** statement **will automatically close the file** after the nested block of code. The advantage of using a ***with statement*** is that it is guaranteed to close the file no matter how the nested block exits. **Even if an exception (a runtime error) occurs before the end of the block, the with statement will handle it and close the file.**

Absolute and Relative Paths

Full name of a file or directory or folder consists of path\primaryname.extension.

Path is a sequence of directory names which give you the hierarchy to access a particul directory or file name. Let us consider the following directory structure :

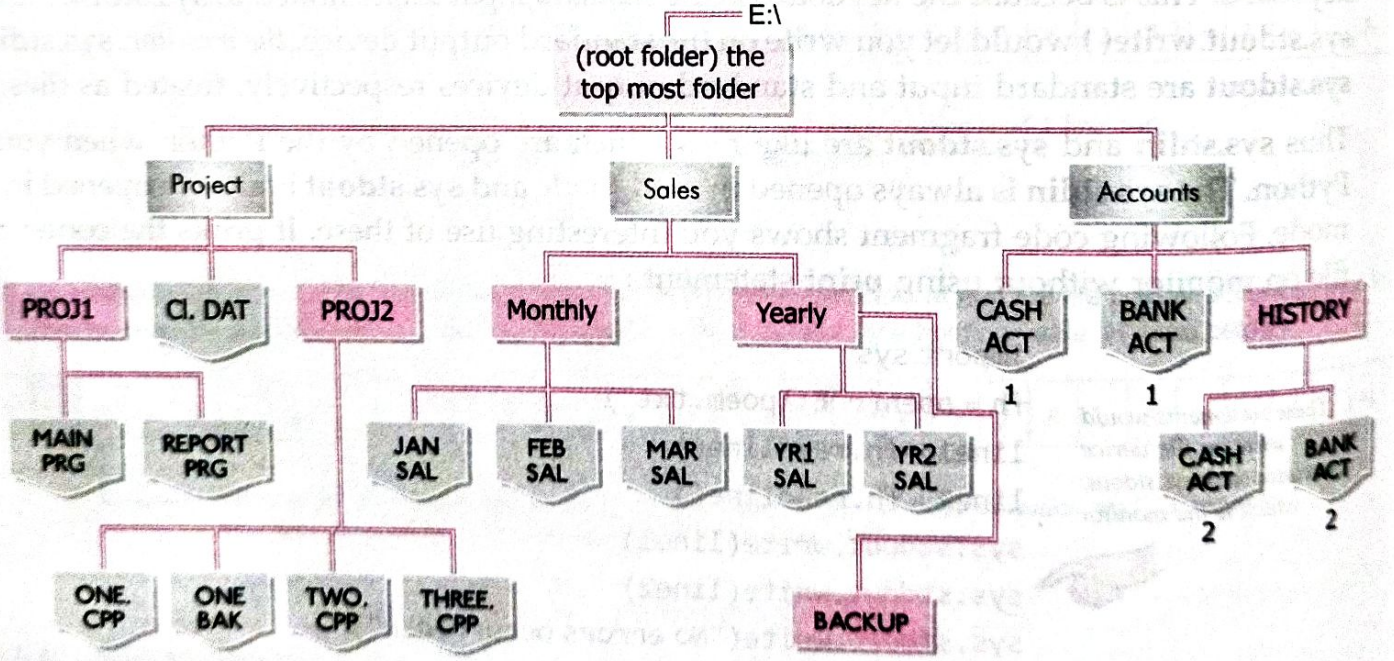


Figure 5.7 A Sample Directory Structure.

Now the *full name* of directories PROJECT, SALES and ACCOUNTS will be

`E:\PROJECT, E:\SALES` and `E:\ACCOUNTS` **respectively.**

So format of path can be given as

```
Drive-letter:\directory [\directory...]
```

where **first \ (backslash) refers to root directory** and other ('\'s) separate a directory name from the previous one.

Now the directory BACKUP'S path will be :

```
E:\SALES\YEARLY\BACKUP
```

As to reach BACKUP the sequence one has to follow is : under drive E, under root directory (first \), under SALES subdirectory of root, under YEARLY subdirectory of SALES, there lies BACKUP directory.

Similarly full name of ONE.VBP file under PROJ2 subdirectory will be :

```
E:\ PROJECT\PROJ2\ONE.VBP
      ↓
   refers to
root directory
```

PATHNAME

The full name of a file or a directory is also called *pathname*.

Now see there are *two* files with the same name CASH.ACT, one under ACCOUNTS directory and another under HISTORY directory but according to Windows rule no two files can have same names (*path names*). Both these files have same names but their path names differ, therefore, these can exist on system. Therefore, $CASH.ACT_1$'s path will be

```
E:\ACCOUNTS\CASH.ACT
```

and $CASH.ACT_2$'s path will be

```
E:\ACCOUNTS\HISTORY\CASH.ACT
```

Above mentioned *path names* are ***Absolute Pathnames*** as they mention the *paths* from the top most level of the directory structure.

NOTE

The absolute paths are from the topmost level of the directory structure. The relative paths are relative to **current working directory** denoted as a **dot(.)** while its **parent directory** is denoted with two dots(..).

Relative pathnames mention the paths relative to current working directory. Let us assume that current working directory now is, say, PROJ2. The symbols . *(one dot)* and .. *(two dots)* can be used now in relative paths and pathnames. The symbol . can be used in place of current directory and .. denotes the parent directory.

So, with PROJ2 as current working folder, pathname of TWO.DOC will be :

```
.\TWO.DOC   ← TWO.DOC in current folder
```

(PROJ2 being working Folder)

which means under current working folder, there is file TWO.PAS. Similarly, path name for file CL.DAT will be

```
..\CL.DAT   ← CL.DAT in parent folder
```

(PROJ2 being working Folder)

which means under parent folder of current folder, there lies a file CL.DAT. Similarly, path name for REPORT.PRG will be

```
..\PROJ1\REPORT.PRG
```

(PROJ2 being working Folder)

that is under parent folder's subfolder PROJ1, there lies a file REPORT.PRG.

Check Point 5.1

1. In which of the following file modes, the existing data of file will not be lost ?
   (a) 'rb' (b) ab (c) w
   (d) w + b (e) 'a + b' (f) wb
   (g) wb+ (h) w+ (i) r+
2. What would be the data type of variable data in following statements ?
   (a) data = f.read( )
   (b) data = f.read(10)
   (c) data = f.readline( )
   (d) data = f.readlines( )
3. How are following statements different ?
   (a) f.readline( )
   (b) f.readline( ).rstrip( )
   (c) f.readline( ).strip( )
   (d) f.readline.rstrip('\n')

5.6 Working with Binary Files

Till now you have learnt to write lines/strings and lists on files. Sometimes you may need to write and read non-simple objects like *dictionaries, tuples, lists* or *nested lists* and so forth on to the files. Since objects have some structure or hierarchy associated, it is important that they are stored in way so that their structure/hierarchy is maintained. For this purpose, objects are often *serialized* and then stored in binary files.

- **Serialisation** (also called **Pickling**) is the process of converting Python object hierarchy into a ***byte stream*** so that it can be written into a file. Pickling converts an object in byte stream in such a way that it can be reconstructed in original form when unpickled or de-serialised.
- **Unpickling** is the inverse of *Pickling* where a byte stream is converted into an object hierarchy. Unpickling produces the exact replica of the original object.

Python provides the **pickle** module to achieve this. As per Python's documentation, *"The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure."* In this section, you shall learn to use pickle module for reading/writing objects in binary files.

PICKLING & UNPICKLING

"Pickling" is the process whereby a Python object hierarchy is converted into a byte-stream, and "unpickling" is the inverse operation, whereby a byte-stream is converted back into an object hierarchy.

In order to work with the *pickle module*, you must first import it in your program using *import* statement :

```
import pickle
```

And then, you may use **dump( )** and **load( )** methods[1] of *pickle module* to write and read from an open binary file respectively. Process of working with binary files is similar to as you have been doing so far with a little difference that you work with *pickle module* in binary files, *i.e.*,

(*i*) Import ***pickle*** module.
(*ii*) Open binary file in the required file mode (read mode or write mode).
(*iii*) Process binary file by writing/reading objects using *pickle* module's methods.
(*iv*) Once done, close the file.

Following sub-sections are going to make it clear.

5.6.1 Creating/Opening/Closing Binary Files

A binary file is opened in the same way as you open any other file (as explained in section 5.3 earlier), but **make sure to use "b" with file modes to open a file in *binary mode*** *e.g.*,

```
Dfile = open("stu.dat", "wb+")
```

*Binary file opened in **write** mode with file handle as Dfile*

Or

Notice 'b' is used with the file modes

```
File1 = open("stu.dat", "rb")
```

*Binary file opened in **read mode** with file handle as File1*

1. There are two similar functions **dumps( )** and **loads( )** of pickle module but these serialise/de-serialise objects in string form while **load( )** and **dump( )** serialise objects for an open binary file. But as per syllabus, we shall only cover only load( ) and dump( ) functions in this chapter.

Like text files, a binary file will get created when opened in an output file mode and it does not exist already. That is, for the file modes, "*w*", "*w+*", "*a*", the file will get created if it does not exist already but if the file exists already, then the file modes "*w*" and "*w+*" will overwrite the file and the file mode "*a*" will retain the contents of the file.

An open binary file is closed in the same manner as you close any other file, *i.e.*, as :

```
Dfile.close()
```

Let us now learn to work with **pickle module's** methods to write/read into binary files.

IMPORTANT

If you are opening a binary file in the *read mode*, then the file must exist otherwise an exception (a run time error) is raised. Also, in an existing file, when the last record is reached and end of file (EOF) is reached, if not handled properly, it may raise **EOFError** exception. Thus it is important to handle exceptions while opening a file for reading. For this purpose, it is advised to open a file in *read mode* either in **try** and **except** blocks or using **with** statement.

We shall talk about both these methods (reading inside **try .. except** blocks and using **with** statement) when we talk about reading from binary files in section 5.6.3.

5.6.2 Writing onto a Binary File – Pickling

In order to write an object on to a binary file opened in the *write mode*, you should use **dump( )** function of *pickle module* as per the following syntax :

```
pickle.dump(<object-to-be-written>, <file handle-of-open-file> )
```

For instance, if you have a file open in handle **file1** and you want to write a list namely ***list1*** in the file, then you may write :

```
pickle.dump(list1, file1)
```

*Object **list1** is being written on file opened with file handle as **File1***

In the same way, you may write *dictionaries*, *tuples* or any other Python object in binary file using **dump( )** function.

For instance, to write a *dictionary* namely **student1** in a file open in handle **file2**, you may write :

```
pickle.dump(student1, file2)
```

*Object **student1** is being written on file opened with file handle as **File2***

NOTE

Python allows you to pickle objects with the following data types :

Booleans, Integers, Floats, Complex numbers, Strings, Tuples, Lists, Sets, Dictionaries containing picklable elements, and classes' objects etc.

Now consider some example programs given below.

Program 5.8 Write a program to a binary file called emp.dat and write into it the employee details of some employees, available in the form of dictionaries.

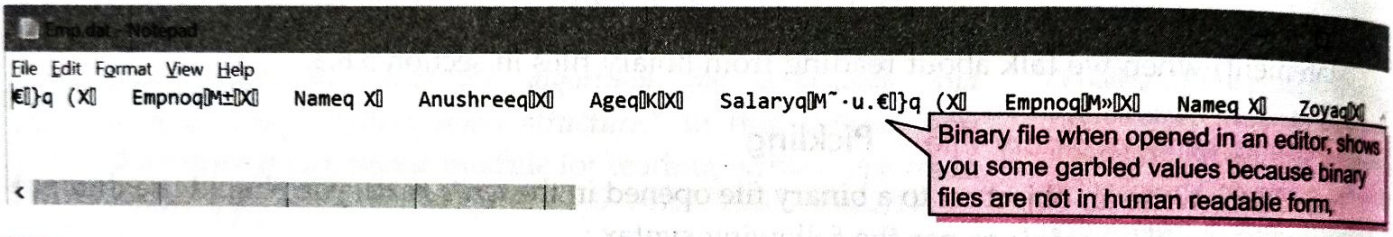
```
import pickle
# dictionary objects to be stored in the binary file
emp1 = {'Empno' : 1201, 'Name' : 'Anushree', 'Age' : 25, 'Salary' : 47000}
emp2 = {'Empno' : 1211, 'Name' : 'Zoya', 'Age' : 30, 'Salary' : 48000}
emp3 = {'Empno' : 1251, 'Name' : 'Simarjeet', 'Age' : 27, 'Salary' : 49000}
emp4 = {'Empno' : 1266, 'Name' : 'Alex', 'Age' : 29, 'Salary' : 50000}
```

```
# open file in write mode
empfile = open('Emp.dat', 'wb')
# write onto the file
pickle.dump(emp1, empfile)
pickle.dump(emp2, empfile)
pickle.dump(emp3, empfile)
pickle.dump(emp4, empfile)
print("Successfully written four dictionaries")
empfile.close( )                    # close file
```

See, 'w' for write mode and 'b' for the binary file

*Dictionary objects being written on file opened with file handle as **empfile***

The above program will create a file namely ***Emp.dat*** in your program's folder and if you try to open the created file with an editor such as *Notepad*, it will show you some garbled values (as shown below) because binary files are not in human readable form.



Program 5.9 Write a program to get student data (roll no., name and marks) from user and write onto a binary file. The program should be able to get data from the user and write onto the file as long as the user wants.

```
import pickle
stu = { }                           # declare empty dictionary
stufile = open('Stu.dat', 'wb')     # open file
# get data to write onto the file
ans = 'y'
while ans == 'y' :
    rno = int(input("Enter roll number : "))
    name = input("Enter name : ")
    marks = float(input("Enter marks : "))
    # add read data into dictionary
    stu['Rollno'] = rno
    stu['Name'] = name
    stu['Marks'] = marks
    # now write into the file
    pickle.dump(stu, stufile)
    ans = input("Want to enter more records? (y/n)...")
stufile.close()                     # close file
```

Output

```
Enter roll number : 11
Enter name : Sia
Enter marks : 83.5
Want to enter more records? (y/n)...y
Enter roll number : 12
Enter name : Guneet
Enter marks : 80.5
Want to enter more records? (y/n)...y
Enter roll number : 13
Enter name : James
Enter marks : 81
Want to enter more records? (y/n)...n
```

The sample run of above program is as shown here.

These 3 student records are written to the file stu.dat

5.6.2A Appending Records in Binary Files

Appending records in binary files is similar to writing, only thing you have to ensure is that you must open the file in append mode (*i.e.*, "ab"). A file opened in append mode will retain the previous records and append the new records written in the file. Just as you normally write in a binary file, you write records while appending using the same **dump( )** function of the *pickle module*.

> **NOTE**
>
> To append records in a binary file, make sure to open the file in append mode ("ab" or "ab+").

5.10 Write a program to append student records to file created in previous program, by getting data from user.

Program

```
import pickle
# declare empty dictionary
stu = { }
# open file in append mode
stufile = open('Stu.dat', 'ab')
# get data to write onto the file
ans = 'y'
while ans == 'y' :
    rno = int(input("Enter roll number : "))
    name = input("Enter name :")
    marks = float( input("Enter marks : "))
    # add read data into dictionary
    stu['Rollno'] = rno
    stu['Name'] = name
    stu['Marks'] = marks
    # now write into the file
    pickle.dump(stu, stufile)
    ans = input("Want to append more records? (y/n)...")
# close file
stufile.close()
```

For appending the records, the file is opened in append mode. Rest of the program is similar to that of writing records.

The sample run of the above program is as shown below :

```
Enter roll number : 14
Enter name : Ali
Enter marks : 80.5
Want to append more records? (y/n)...n
```

1 more student record is appended to the file stu.dat

5.6.3 Reading from a Binary File – UnPickling

Once you have written onto a file using **pickle** module's **dump( )** (as we did in the previous last section), you need to read from the file using **load( )** function of the **pickle** module as it would then **unpickle** the data coming from the file.

The **load( )** function is used as per the following syntax :

```
<object> = pickle.load(<filehandle>)
```

For instance, to read an object in **nemp** from a file open in file-handle **fout**, you would write :

```
nemp = pickle.load(fout)
```

*Read from the file opened with file handle as **fout** and store the read data in an object namely **nemp***

Following program 5.11 does the same for you. It reads the objects written by program 5.8 from the file **Emp.dat** and displays them. But before the program 5.11, read the following box. **(Important)**

IMPORTANT

But before you move onto the program code, it is important to know that **pickle.load( )** function would raise ***EOFError*** (a run time exception) when you reach end-of-file while reading from the file. You can handle this by following one of the below given two methods.

- ❖ Use **try** and **except** blocks
- ❖ Using with statement

(i) Use try and except blocks

Thus, you must write **pickle.load( )** enclosed in **try** and **except** statements as shown below. The **try** and **except** statements together, can handle runtime exceptions. In the **try** block, *i.e.*, between the **try** and **except** keywords, you write the code that can generate an exception and in the **except** block, *i.e.*, below the **except** keyword, you write what to do when the exception (**EOF – end of file** in our case) has occurred. (See below)

```
<filehandle> = open (<filename>, <readmode>)
try :
    <object> = pickle.load(<filehandle>)
    # other processing statements
except EOFError :
    <filehandle>.close()
```

*In the **try** block, write the **pickle.load( )** statement and other processing statements. In order to read all the records, read inside a loop as shown in the following program.*

Use this keyword with except keyword for checking EOF (end of file)

*In the **except** block, write code for what to do when EOF exception has occurred.*

Here, you just need to just concentrate on the syntax; you need not go in further details of **try** and **except** as it is beyond the scope of the book.

(ii) Using with statement

The **with** statement is a compact statement which combines the opening of file and processing of file along with inbuilt exception handling. (Refer to Info box 5.3 given earlier where we have talked about the **with** statement.) The **with** statement will also close the file automatically after **with block** is over. You can use the **with** statement as :

```
with open(<filename>, <mode>) as <file handle>:
    # use pickle.load here in this with block
    # perform other file manipulation task in this with block
```

Notice that you need not mention any exception with the **with** statement, explicitly.

Please note that while writing onto file, the exceptions like **"File does not exist"** or the EOF error do not arise as most write modes create the file if it does not exist already and you can write onto them as long as you want, *i.e.*, there is no restricting EOF marker for writing.

In the program below, we have used both the **try..except** block (in *programs 5.11* and *5.12*) and the **with** statement (in programs *5.13* and *5.14*) for working with the files. Now consider the following program that is reading from the file you created in the previous program.

5.11 Write a program to open the file **Emp.dat** (created in program 5.8), read the objects written in it and display them.

Program

```
import pickle
#declare empty dictionary object; it will hold the read record
emp = { }
empfile = open('Emp.dat', 'rb')          # open binary file in read mode
#read from the file
try:
    while True :  # it will become False when the end of file is reached (EOF exception).
        emp = pickle.load(empfile)
        print(emp)
except EOFError:
    empfile.close()                      # close file
```

*Read repeatedly; when no more records, it will give EOF and take you to **except** block where file is closed.*

This statement would unpickle the object being read from file

You can now use the object in usual way (e.g., we printed its contents)

The statements in this block will get executed when EOF has occurred, i.e., the file will be closed on reaching EOF

The output produced by the above program will be :

```
{'Empno': 1201, 'Name': 'Anushree', 'Age': 25, 'Salary': 47000}
{'Empno': 1211, 'Name': 'Zoya', 'Age': 30, 'Salary': 48000}
{'Empno': 1251, 'Name': 'Simarjeet', 'Age': 27, 'Salary': 49000}
{'Empno': 1266, 'Name': 'Alex', 'Age': 29, 'Salary': 50000}
```

It is returning the same data as we wrote in the previous program (Compare with the data of the previous program)

Now that you have an idea of how to read and write in binary files, let us write a few more programs before we talk about searching in and updating the binary files.

5.12 Write a program to open file created and used in programs 5.9 and 5.10 and display the student records stored in it.

Program

```
import pickle
stu = {}                                 # declare empty dictionary object to hold read records
fin = open('Stu.dat', 'rb')              # open binary file in read mode
# read from the file
try:
    print("File Stu.dat stores these records")
    while True :                         # it will become False upon EOF
        stu = pickle.load(fin)           # read record in stu dictionary from fin file handle
        print(stu)                       # print the read record
except EOFError:
    fin.close()                          # close file
```

Following program is just doing the same. It is using a Boolean variable namely **found** that is *False* initially and stores *True*, as soon as the search is successful. In the end, this variable is tested for its value and accordingly the message is reported.

5.15 Write a program to open file Stu.dat and search for records with roll numbers as 12 or 14. If found, display the records.

Program

```
import pickle
stu = {}                    # declare empty dictionary object to hold read records
found = False
fin = open('Stu.dat', 'rb')     # open binary file in read mode
searchkeys = [12, 14]           # list contains key values to be searched for

# read from the file
try:
   print("Searching in File Stu.dat ...")
   while True :                 # it will become False upon EOF exception
       stu = pickle.load(fin)   # read record in stu dictionary from fin file handle
       if stu['Rollno'] in searchkeys :
           print(stu)           #print the record
           found = True
except EOFError:
   if found == False :
       print("No such records found in the file")
   else:
       print("Search successful.")
   fin.close()                  # close file
```

Searching for in the read record

This block will get executed when the search is successful.

```
Searching in File Stu.dat ...
{'Rollno': 12, 'Name': 'Guneet', 'Marks': 80.5}
{'Rollno': 14, 'Name': 'Ali', 'Marks': 80.5}
Search successful.
```

5.16 Read file stu.dat created in earlier programs and display records having marks > 81.

Program

```
import pickle
stu = {}                    # declare empty dictionary object to hold read records
found = False
print("Searching in file Stu.dat ...")
# open binary file in read mode and process with the with block
with open('Stu.dat', 'rb') as fin :
    stu = pickle.load(fin)  # read record in stu dictionary from fin file handle
    if stu['Marks'] > 81 :
        print(stu)          # print the read record
        found = True
if found == False :
    print("No records with Marks > 81")
else:
    print("Search successful.")
```

```
Searching in file Stu.dat for Marks > 81 ...
  {'Rollno': 11, 'Name': 'Sia', 'Marks': 83.5}
  Search successful.
```

5.6.5 Updating in a Binary File

You know that updating an object means changing its value(s) and storing it again. Updating records in a file is similar and is a *three-step* process, which is :

(*i*) Locate the record to be updated by searching for it

(*ii*) Make changes in the loaded record in memory (the read record)

(*iii*) Write back onto the file at the exact location of old record.

You can easily perform the first two steps by whatever you have learnt so far. But for the third step, you need to know the location of the record in the file and then ensuring that the record being written is written at the exact location. Thus, we shall first talk about in the following sub-section how you can obtain the location of a record and how you can place the file pointer on a specific location (requirements of updating in a file).

5.6.5A Accessing and Manipulating Location of File Pointer – Random Access

Python provides two functions that help you manipulate the position of file-pointer and thus you can read and write from desired position in the file. The two file-pointer location functions of Python are : **tell( )** and **seek( )**. These functions work identically with the text files as well as the binary files.

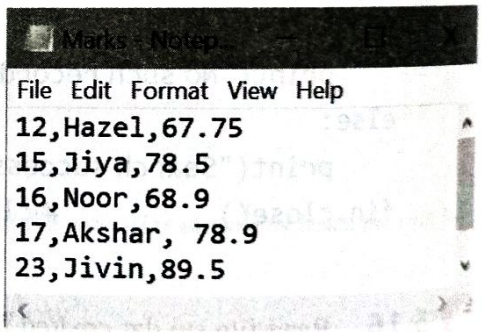
The tell( ) Function

The tell( ) function returns the current position of file pointer in the file. It is used as per the following syntax :

```
<file-object>.tell()
```

where ***file-object*** is the handle of the open file, *e.g.*, if the file is opened with handle **fin**, then **fin.tell( )** will give you the position of file pointer in the file opened with the handle **fin**.

Consider some examples, given below. We are using the same file *"Marks.txt"* (shown on the right) that we created in earlier examples.



Now consider the following code snippet :

```
fh = open("Marks.txt", "r")
print("Initially file-pointer's position is at:", fh.tell())
print ("3 bytes read are:", fh.read(3))
print ("After previous read, Current position of file-pointer:", fh.tell())
```

To get current position of file pointer

3 bytes read

The output produced by the above code will be :

```
>>>
Initially file-pointer's position is at : 0
3 bytes read are : 12,
After previous read, Current position of file-pointer : 3
```

Initially file-pointer is at 0th byte

Notice first 3 bytes of the file contain 1, 2 and a comma

After reading 3 bytes file-pointer's position is 3.

Now consider the following modified code :

```
fh = open("Marks.txt", "r")
print ("3 bytes read are:", fh.read(3))
print ("After previous read, Current position of file-pointer:", fh.tell())
print ("Next 5 bytes read:", fh.read(5))
print ("After previous read, Current position of file-pointer:", fh.tell())
```

The output produced by above code will be :

```
>>>
3 bytes read are: 12,
After previous read, Current position of file-pointer: 3
Next 5 bytes read: Hazel
After previous read, Current position of file-pointer: 8
```

Returned by first ***fh.tell( )***

Returned by second ***fh.tell( )***

The seek( ) Function

The **seek( )** function changes the position of the file-pointer by placing the file-pointer at the specified position in the open file.

The syntax for using this function is :

```
<file-object>.seek( offset[, mode] )
```

where

offset is a number specifying number-of-bytes

mode is a number 0 or 1 or 2 signifying

- 0 for beginning of file (to move file-pointer w.r.t. beginning of file) it is default position (*i.e.*, when no mode is specified)
- 1 for current position of file-pointer (to move file-pointer w.r.t. current position of it)
- 2 for end of file (to move file-pointer w.r.t. end of file)

file object is the handle of open file.

The seek( ) function changes the file pointer's position to a *new file position = start + offset* with respect to the start position as specified by the **mode** specified.

Consider the following examples :

```
fh = open("Marks.txt", "r")
:
fh.seek(30)
fh.seek(30, 1)
fh.seek(-30, 2)
fh.seek(30, 0)
fh.seek (-5, 1)
```

will place the file pointer at 30th byte from the beginning of the file (default)

will place the file pointer at 30th byte ahead of current file-pointer position (mode = 1)

will place file pointer at 30 bytes behind (backward direction) from end-of file (mode = 2)

will place the file pointer at 30th byte from the beginning of the file (mode = 0)

will place file pointer at 5 bytes behind (backward direction) from current file-pointer position (mode = 1)

NOTE

The **<file-object>.tell( )** function returns the current position of file pointer in an open file.

And the **<file-object>.seek( )** function places the file pointer at the specified by in an open file.

With the above examples, it is clear that you can move the file-pointer in **forward direction** (**with positive value for bytes**) as well as the **backward direction** (by giving **negative value for bytes**).

However, one thing that you should bear in mind is that :

- Backward movement of file-pointer is not possible from the **beginning of the file** (BOF).
- Forward movement of file-pointer is not possible from the **end of file** (EOF).

NOTE

You can move the file-pointer in forward direction (positive value for bytes) as well as the backward direction (by giving negative value for bytes).

Now consider some examples based on the above-discussed file pointer location functions.

Code Snippet 11 *Check the position of file pointer after read( ) function*

```
fh = open("Marks.txt", "r")
print(fh.read())
print("File-pointer is now at byte :", fh.tell())
```

The output produced by above code is :

```
12,Hazel,67.75
15,Jiya,78.5
16,Noor,68.9
17,Akshar,78.9
23,Jivin,89.5
File-pointer is now at byte : 75
```

*This value is returned by **fh.tell( )***

As you can make out that file has 74 characters including '\n' at the end of every line and thus after reading the entire file, the file-pointer is at the end-of file and thus showing 75.

Code Snippet 12 *Read the last 15 bytes of the file "Marks.txt"*

```
fh = open("Marks.txt", "r")
fh.seek(-15, 2)
str1 = fh.read(15)
print("Last 15 bytes of file contain :", str1)
```

Place the file pointer 15 bytes before the end of file (thus mode = 2)

The output produced by above code is :

```
Last 15 bytes of file contain : 23,Jiv in,89.5
```

NOTE

Functions **seek( )** and **tell( )** work identically in text and binary files.

Armed with the knowledge of file-pointer location functions, you can now easily update a file. Following sub-section will explain this.

5.6.5B Updating Record(s) in a File

Let us recall the three-step updation process mentioned earlier, which is :

(*i*) Locate the record to be updated by searching for it.

(*ii*) Make changes in the loaded record in memory (the read record).

(*iii*) Write back onto the file **at the exact location** of old record.

To determine the exact location, the enhanced version of the updation process would be :

(*i*) Open file in read as well as write mode. **(Important)**

(*ii*) Locate the record :

(*a*) Firstly store the position of file pointer (say **rpos**) before reading a record

(*b*) Read record from the file and search the key in it through appropriate test condition.

(*c*) If found, your desired record's start position is available in **rpos** (stored in step *a*)

(*iii*) Make changes in the record by changing its values in memory, as desired.

(*iv*) Write back onto the file **at the exact location** of old record.

(*a*) Place the file pointer at the stored record position (**the exact location**) using **seek( )**, *i.e.*, at **rpos**, which was stored in step a (the exact location of the record being updated)

(*b*) Write the modified record now. **The previous step is important and necessary as any operation read or write takes place at the current file pointer's position.** So the file pointer must be at the beginning of the record to be over-written.

Following example program illustrates this process.

Program 5.17(*a*) Consider the binary file **Stu.dat** storing student details, which you created in earlier programs. Write a program to update the records of the file Stu.dat so that those who have scored more than 81.0, get additional bonus marks of 2.

Note. Important statements have been highlighted.

```
import pickle
stu = {}                    # declare empty dictionary object to hold read records
found = False
# open binary file in read and write mode
fin = open('Stu.dat', 'rb+')          <-- It is important to open the file in read as well as write mode ; hence rb+
# read from the file
try:
    while True :                      <-- Before reading any record, firstly store its beginning position – its exact position
        rpos = fin.tell()       # store file-pointer position before reading the record
        stu = pickle.load(fin)
        if stu['Marks'] > 81 :        <-- Locating the desired record through search condition
            stu['Marks'] += 2         # changes made in the record; 2 bonus marks added
            fin.seek(rpos)            # place the file-pointer at the exact location of the record
            pickle.dump(stu, fin)     # now write the updated record on the exact location
            found = True
except EOFError:
    if found == False:
        print("Sorry, no matching record found.")
    else:
        print("Record(s) successfully updated.")
    fin.close()                       # close file
```

placing the file-pointer at the exact location of the record you stored earlier

After placing the file-pointer at the exact location, now write the updated record

The above program will look for the desired matching record (with marks > 81) and make the changes in it and write back into the file.

It will show you a message :

```
Record(s) successfully updated.
```

Following program reads the modified file and displays its records. You can see yourself if the record is modified.

Program 5.17(*b*) Display the records of file **Stu.dat**, which you modified in of program 5.17(*a*).

```
import pickle
stu = { }                    # declare empty dictionary object to hold read records
# open binary file in read mode
fin = open('Stu.dat', 'rb')
# read from the file
try:
    print("File Stu.dat stores these records")
    while True :
        stu = pickle.load(fin)      # read record in stu dictionary from fin
        print(stu)                  #print the read record
except EOFError:
    fin.close()                     # close file
```

The output produced by above program is :

```
File Stu.dat stores these records
{'Rollno': 11, 'Name': 'Sia', 'Marks': 85.5}
{'Rollno': 12, 'Name': 'Guneet', 'Marks': 80.5}
{'Rollno': 13, 'Name': 'James', 'Marks': 81.0}
{'Rollno': 14, 'Name': 'Ali', 'Marks': 80.5}
```

See, the matching record's marks have been modified (compare with the output of program 5.12)

You can also place the file pointer backwards using negative values in bytes BUT for that you need to get the size of record (*i.e.,* the object stored in the file) in bytes. Getting the size of a record in bytes is not straight forward in Python. For that you need to import a different module (*e.g.,* sys or cpickle etc.). Covering these modules here is beyond the scope of the book and thus we advise you to use the method covered above.

Program 5.18 Write a program to modify the name of rollno 12 as *Gurnam* in file *Stu.dat* (created in earlier programs)

```
import pickle
stu = {}                                # declare empty dictionary object to hold read records
found = False
fin = open('Stu.dat', 'rb+')            # open binary file in read and write mode
# read from the file
```

```
    try:
        while True :                    # it will become False upon EOF exception
            rpos = fin.tell()           # store file-pointer position before reading the record
            stu = pickle.load(fin)      # read record in stu dictionary from fin file handle
            if stu['Rollno'] == 12 :        #locate matching record
                stu['Name'] = 'Gurnam'      # changes made in the record
                fin.seek(rpos)              #place the file-pointer at the exact location of the record
                pickle.dump(stu, fin)
                found = True
    except EOFError:
        if found == False:
            print("Sorry, no matching record found.")
        else:
            print("Record(s) successfully updated.")
        fin.close()                 # close file
```

If run the code of program 5.17(*b*) to display the contents of modified file, it will show you the contents as :

```
File Stu.dat stores these records
{'Rollno': 11, 'Name': 'Sia', 'Marks': 87.5}
{'Rollno': 12, 'Name': 'Gurnam', 'Marks': 80.5}
{'Rollno': 13, 'Name': 'James', 'Marks': 81.0}
{'Rollno': 14, 'Name': 'Ali', 'Marks': 80.5}
```

See, the name of record with Rollno 12 is modified Compare with the output of previous program.

IMPORTANT

While modifying a binary file, make sure that the data types do not get changed for the value being modified. For example, if you modify **an integer field** as *value +value *0.25*; then the result may be **a floating point number**. Such a change may affect pickling and unpickling process and sometimes it leads to the **Unpickling Error**. Thus, make sure that modification of file data does not change the data type of the value being modified.

If you still need to have such modification then you can do it in a different way –

(*i*) create a new file ;

(*ii*) write records into the new file until the record to be modified is reached;

(*iii*) modify the record in memory and write the modified record in the new file;

(*iv*) Once done. Delete the old file and rename the new file with the old name.

(*v*) Deleting and renaming of files can be done through the **os module's** remove( ) and rename( ) functions as os.remove(<filename>) and os.rename(<old filename>, <new filename>).

(Make sure to import the **os module** before using its functions).

Following exceptions may arise while working with the *pickle module*.

pickle.PicklingError Raised when an unpicklable object is encountered while writing.

pickle.UnpicklingError Raised during unpickling of an object, if there is any problem (such as data corruption, access violation, etc).

5.7 Working with CSV Files

You know that CSV files are delimited files that store tabular data (data stored in rows and columns as we see in spreadsheets or databases) where comma delimits every value, *i.e.*, the values are separated with comma. **Typically the default delimiter of CSV files is the comma (,)** but modern implementations of CSV files allow you to choose a different delimiter character other than comma. Each line in a CSV file is a **data record**. Each record consists of one or more fields, separated by commas (or the chosen delimiter).

Since CSV files are the *text files*, you can apply text file procedures on these and then split values using **split( )** function BUT there is a better way of handling CSV files, which is – using ***csv module of Python***. In this section, we are going to talk about how you can read and write in CSV files using the *csv module* methods.

NOTE

The separator character of CSV files is called a delimiter. Default and most popular delimiter is comma. Other popular delimiters include the tab (\t), colon (:), pipe (|) and semi-colon (;) characters.

Python csv Module

The **csv** module of Python provides functionality to read and write tabular data in CSV format. It provides *two* specific types of objects – the **reader** and **writer** objects – to read and write into CSV files. The csv module's **reader** and **writer** objects read and write delimited sequences as records in a CSV file.

You will practically learn to use the reader and writer objects and the functions to read and write in CSV files in the coming sub-sections. But before you use **csv** module, make sure to import it in your program by giving a statement as :

```
import csv
```

5.7.1 Opening/Closing CSV Files

A CSV file is opened in the same way as you open any other text file (as explained in section 5.3 earlier), but make sure to do the following *two* things :

(*i*) Specify the file extension as **.csv**

(*ii*) Open the file like other text files, *e.g.*,

```
Dfile = open("stu.csv", "w")
```

*CSV file opened in **write mode** with file handle as Dfile*

Or

```
File1 = open("stu.csv", "r")
```

*CSV file opened in **read mode** with file handle as File1*

An open CSV file is closed in the same manner as you close any other file, *i.e.*, as :

```
Dfile.close()
```

Like text files, a CSV file will get created when opened in an output file mode and if it does not exist already. That is, for the file modes, "*w*", "*w +*", "*a*" "*a +*", the file will get created if it does not exist already and if it exists already, then the file modes "*w*" and "*w +*" will overwrite the existing file and the file mode "*a*" or "*a +*" will retain the contents of the file.

NOTE

The csv files are popular because of these reasons : (*i*) Easier to create, (*ii*) preferred export and import format for databases and spreadsheets, (iii) capable of storing large amounts of data.

5.7.1A Role of Argument newline in Opening of csv Files

While opening csv files, in the **open( )**, you can specify an additional argument **newline**, which is an optional but important argument. The role of **newline** argument is to specify how would Python handle newline characters while working with csv files.

As csv files are text files, while storing them, some types of translations occur – such as translation of **end of line (EOL) character** as per the operating system you are working on etc. Different operating systems store EOL characters differently. The MS-DOS (including Windows and OS/2), UNIX, and Macintosh operating systems all use different characters to designate the end of a line within a text file. Following table 5.4 lists the EOL characters used by different operating systems.

Table 5.4 *EOL characters used in different operating systems.*

Symbol/Char	Meaning	Operating System
CR [\r]	Carriage Return	Macintosh
LF [\n]	Line Feed	UNIX
CR/LF [\r \n]	Carriage Return/Line Feed	MS-DOS, Windows, OS/2
NULL [\0]	Null character	Other OSs

Now what would happen if you create a csv file on one operating system and use it on another. The EOL of one operating system will not be treated as EOL on another operating system. Also, if you have given a character, say '\r', in your text string (not as EOL but as part of a text string) and if you try to open and read from this file on a Macintosh system, what would happen ? Mac OS will treat every '\r' as EOL – yes, including the one you gave as part of the text string.

So what is the solution? Well, the solution is pretty simple. Just suppress the EOL translation by specify third argument of **open( )** as newline = ' ' (null string – no space in quotes).

If you specify the **newline** argument while writing onto a csv file, it will create a csv file with no EOL translation and you will be able to use csv file in normal way on any platform.

NOTE

Additional optional argument as newline = '' (null string; no space in between) with file **open( )** will ensure that ***no translation of end of line (EOL) character*** takes place.

That is, open your csv file as :

```
Dfile = open("stu.csv", "w", newline = '')
```

null string ; no space in between

*CSV file opened in **write mode** with file handle as **Dfile** (no EOL translation)*

Or

```
File1 = open("stu.csv", "r", newline = '')
```

*CSV file opened in **read mode** with file handle as **File1** (no EOL translation)*

Not only this is useful for working across different platforms, it is also useful when you work on the same platform for writing and reading. You will be able to appreciate the role of the *newline* argument when we talk about reading from the csv files. Program 5.21 onwards it will be clear to you.

Let us now learn to work with **csv module's** methods to write / read into CSV files.

5.7.2 Writing in CSV Files

Writing into csv files involves the conversion of the user data into the writable delimited form and then storing it in the form of csv file. For writing onto a csv files, you normally use the following *three* key players functions[2].

2. Other than the above shown functions, there are other functions too like ***DictWriter( )*** function but these are beyond the scope of the syllabus, hence we shall not cover these in this chapter.

These are :

`csv.writer()`	returns a writer object which writes data into CSV file
`<writerobject>.writerow()`	writes one row of data onto the ***writer object***
`<writerobject>.writerows()`	writes multiple rows of data onto the ***writer object***

Before we proceed, it is important for you to understand the significance of the **writer object**. Since csv files are delimited flat files and before writing onto them, the data must be in **csv-writable-delimited-form**[3] , it is important to convert the received user data into the form appropriate for the csv files. **This task is performed by the writer object.** The data row written to a writer object (written using **writerow( )** or **writerows( )** functions) gets converted to csv writable delimited form and then written on to the linked csv file on the disk.

Following figure 5.8 illustrates this process.

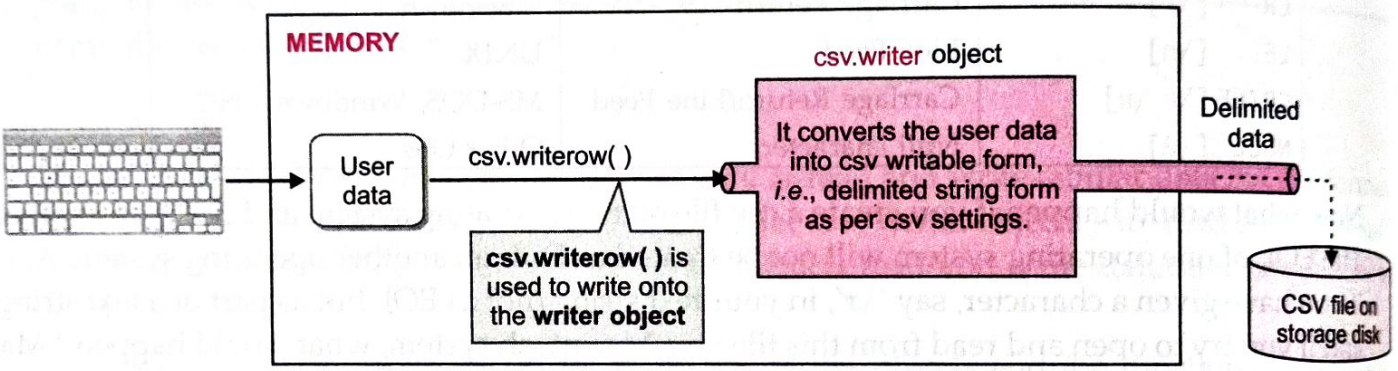


Figure 5.8 Role of the csv writer object.

Let us now learn to write onto csv files. In order to write onto a csv file, you need to do the following (**Note**, in order to appreciate the use of *newline* argument, we shall initially create csv files without the *newline* argument first and later we shall create csv files with *newline* argument too.)

(*i*) Import csv module

(*ii*) Open csv file in a file-handle (just as you open other text files), *e.g.*,

```
fh = open("student.csv", "w")
```

(*iii*) Create the **writer object** by using the syntax as shown below:

```
<name-of-writer-object> = csv.writer(<file-handle>, [delimiter = <delimiter character>])
```

If you skip this argument, comma will be used as the delimiter

e.g.,

```
stuwriter = csv.writer(fh)
```

you opened the file with this file handle in previous step

In the above statement, delimiter argument is not specified. When you skip the delimiter argument, the default delimiter character, which is comma, is used for separating characters. But if you specify a character with **delimiter** argument then the specified character is used as the delimiter, *e.g.*,

```
stuwriter = csv.writer(fh, delimiter = '|')
```

The above statement will create file with delimiter character as pipe symbol ('|').

3. It depends on something called ***dialect***, which by default is set for producing ***excel*** format like csv files. It can even be ***excel-tab*** where tab character is the delimiter. You can set the dialect using ***dialect*** argument but we shall only work with the default dialect and hence won't use *dialect* argument here. For further details on *dialect* argument, you may refer to the Python's documentation.

(iv) Obtain user data and form a Python sequence (list or tuple) out of it, *e.g.*,

```
Sturec = (11, 'Neelam', 79.0)
```

(v) Write the Python sequence containing user data onto the writer object using csv.writerow( ) or csv.writerows( ) functions, *e.g.*,

```
csv.writerow(Sturec)
```

Both the **writerow( )** and **writerrows( )** functions can be used for writing onto the writer object. We shall discuss about how to use **writerows( )** function little later. For now, let us focus on writing through **writerow( )** function, *i.e.*, writing single row at a time.

CSV files can also take the names of columns as header rows, which are written in the same way as any row of data is written, *e.g.*, to give column headings as "Rollno", "Name" and "Marks", you may write :

```
stuwriter.writerow( ['Rollno', 'Name', 'Marks'])
```

(vi) Once done, close the file.

You only need to write into the writer object. Rest of the work it will do itself, *i.e.*, converting the data into delimited form and then writing it onto the linked csv file.

Let us now do it practically. Following program illustrates this process.

5.19 Write a program to create a CSV file to store student data (Rollno., Name, Marks). Obtain data from user and write 5 records into the file.

Program

```
import csv
fh = open("Student.csv", "w")                    #open file
stuwriter = csv.writer(fh)
stuwriter.writerow(['Rollno', 'Name', 'Marks'] )        #write header row
for i in range(5):
    print("Student record", (i+1) )
    rollno = int(input("Enter rollno:"))
    name = input("Enter name:")
    marks = float(input("Enter marks:"))
    sturec = [rollno, name, marks]              #create sequence of user data
    stuwriter.writerow(sturec)
fh.close()                  #close file
```

Column headings written

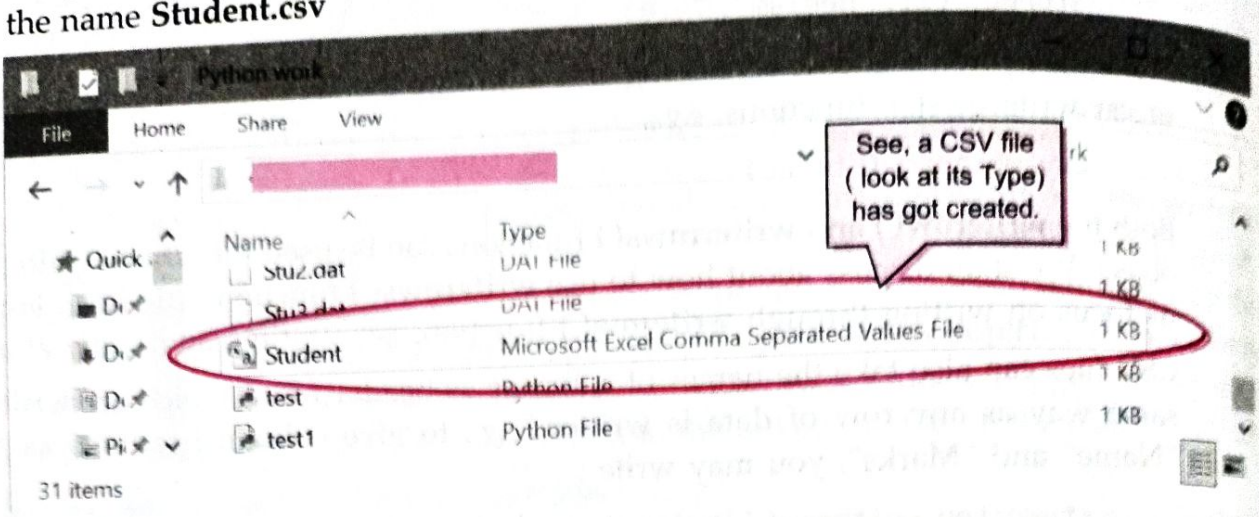
Python sequence created from user data

*Python data sequence written on the writer object using **writerow( )***

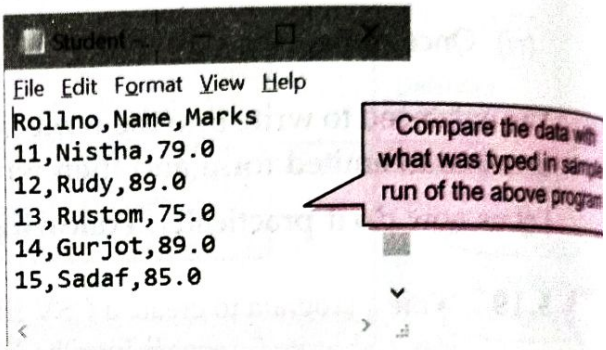
The sample run of the above program is as shown below :

```
Student record 1
Enter rollno:11
Enter name:Nistha
Enter marks:79
Student record 2
Enter rollno:12
Enter name:Rudy
Enter marks:89
Student record 3
Enter rollno:13
Enter name:Rustom
Enter marks:75
Student record 4
Enter rollno:14
Enter name:Gurjot
Enter marks:89
Student record 5
Enter rollno:15
Enter name:Sadaf
Enter marks:85
```

Now if you open the folder of your Python program, you will see that Python has created a file by the name **Student.csv**



And if you open this file in Notepad or any other ASCII editor, it shows :



The writerows( ) Function

If you have all the data available and data is not much lengthy then it is possible to write all data in one go. All you need to do is create a nested sequence out of the data and then write using the **writerows( )** function.

The **writerows( )** method writes all given rows to the CSV file, *e.g.*, to write following nested sequence, you can use the **writerows( )** function :

```
Sturec = [ [11,Nistha,79.0], [12,Rudy,89.0], [13,Rustom,75.0] ]
<writerobject>.writerows(Sturec)
```

Each inner list will now be written as a separate record in the csv file.

Following program illustrates this.

Program 5.20 The data of winners of four rounds of a competitive programming competition is given on right side.

Write a program to create a csv file (compresult.csv) and write the above data into it.

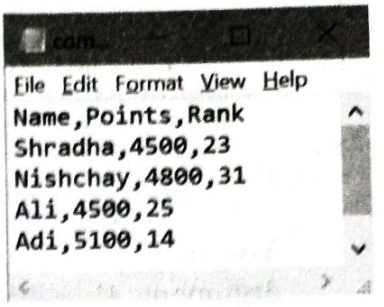
```
['Name', 'Points', 'Rank']
['Shradha', 4500, 23]
['Nishchay', 4800, 31]
['Ali', 4500, 25]
['Adi', 5100, 14]
```

```
import csv
fh = open("compresult.csv", "w")
cwriter = csv.writer(fh)
compdata = [
    ['Name', 'Points', 'Rank'],
    ['Shradha', 4500, 23],
    ['Nishchay', 4800, 31],
    ['Ali', 4500, 25],
    ['Adi', 5100, 14] ]
cwriter.writerows(compdata)
fh.close()
```

This nested sequence contains multiple records' data in the form of inner lists

*Nested list written in one go using **writerows( )***

The above program will create a csv file and in Notepad, it will look like :



You can also create the nested sequence programmatically by appending one record to a list while writing using **writerows( )** function. ***Solved problem 48 performs the same.***

Please note that till now we have created csv files without using the **newline** argument, which means that EOL translation has taken place internally. How this impacts a csv file, will become clear to you soon when we start programming in the following section.

5.7.3 Reading in CSV Files

Reading from a csv file involves loading of a csv file's data, **parsing** it (*i.e.*, removing its delimitation) , loading it in a ***Python iterable*** and then reading from this iterable. Recall that any Python sequence that can be iterated over in a for-loop is a *Python iterable, e.g., lists, tuples,* and *strings* are all Python iterables.

For reading from a csv files, you normally use following *function* :

`csv.reader()`	returns a reader object which loads data from CSV file into an iterable after parsing delimited data

The **csv.reader** object does the opposite of **csv.writer** object. The *csv.reader* object loads data from the csv file, parses it, *i.e.*, removes the delimiters and returns the data in the form of a Python iterable wherefrom you can fetch one row of data at a time. (Fig. 5.9)

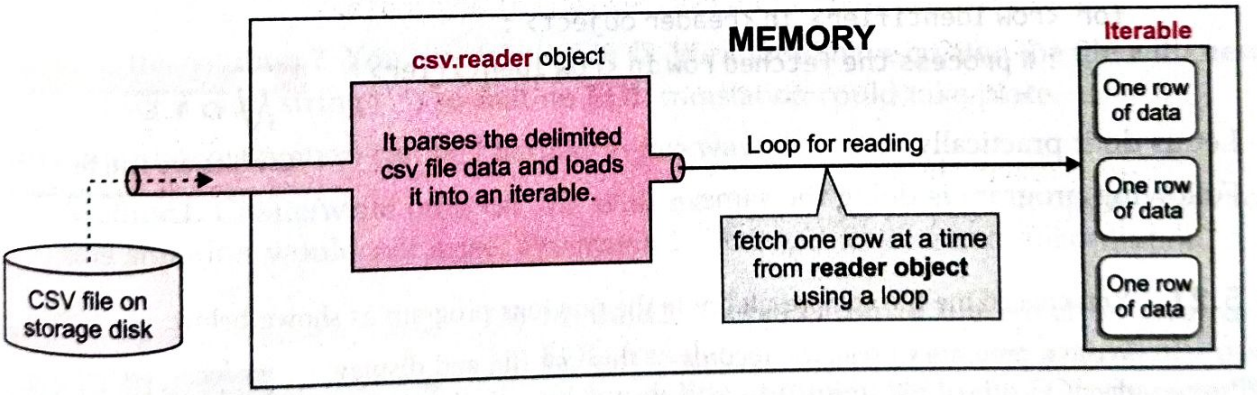


Figure 5.9 Role of csv.reader object.

Let us now learn to read from csv files. In order to read from a csv file, you need to do the following :

(*i*) Import csv module

(*ii*) Open csv file in a file-handle in read mode (just as you open other text files),

```
<file handle> = open (<csv-file>, <read mode>)
```

e.g.,

```
fh = open("student.csv", "r")
```

The file being opened must already exist otherwise an exception will get raised. Your code should be able to handle the exception *i.e.*, through **try..except**. (Alternatively you can use the **with** statement as mentioned below.)

(*iii*) Create the **reader object** by using the syntax as shown below :

```
<name-of-reader-object> = csv.reader(<file-handle>,[delimiter =<delimiter character>])
```

e.g.,

```
stureader = csv.reader(fh)
```

← *you opened the file with this file handle in previous step*

You may also specify a delimiter character other than a comma using the **delimiter** argument, *e.g.*,

```
stureader = csv.reader(fh, delimiter = '|')
```

(*iv*) The reader object stores the parsed data in the form of iterable and thus you can fetch from it row by row through a traditional for loop, one row at a time :

← *Loop to fetch one row at a time in* ***rec*** *from the iterable*

```
for rec in stureader :
    print (rec)          # or do any other processing
```

(*v*) Process the fetched single row of data as required.

(*vi*) Once done, close the file.

All the above mentioned steps are best performed through **with** statement as the with statement will also take care of any exception that may arise while opening/reading a file. Thus you should process the csv file as per the following syntax, which combines all the above mentioned steps :

```
with open (<csv-file>, <read mode>) as <file handle> :
    <name-of-reader-object> = csv.reader(<file-handle>)
    for <row identifier> in <reader object> :
        : # process the fetched row in <row identifier>
```

NOTE

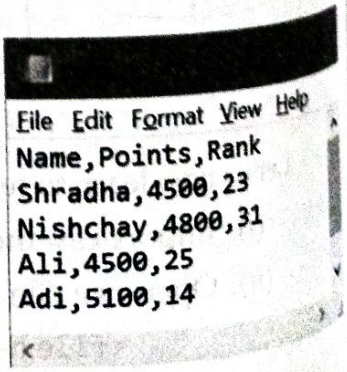
Csv files are flat, text files.

Let us do it practically.

Following program is doing the same.

Program 5.21 You created the file compresult.csv in the previous program as shown below.

Write a program to read the records of this csv file and display them.



Note. *Before we write the code for this program, recall that the given file created in the previous program,* ***was created without specifying the newline argument*** *in the file* ***open( )*** *function. So have a look at the previous program's (program 5.20) code before starting with this program's code.*

```
import csv
with open("compresult.csv", "r") as fh :
    creader = csv.reader(fh)
    for rec in creader :
        print(rec)
```

← *Loop to fetch one row at a time in* ***rec*** *from the iterable in* ***creader***

The above program will produce the result as :

```
['Name', 'Points', 'Rank']
[]
['Shradha', '4500', '23']
[]
['Nishchay', '4800', '31']
[]
['Ali', '4500', '25']
[]
['Adi', '5100', '14']
[]
```

Where have these empty rows come from ?

We never entered empty rows. Refer to the sample run of program 5.20.

Then what is the source / reason of these empty rows ?

Compare the above output with the sample run of previous program (program 5.20). We never entered empty rows. Then where have these empty rows come from? Any idea?

You guessed it right. We did not specify the **newline** argument in the file **open( )** function while creating/writing this csv file (*compresult.csv*, in this case) and thus EOL translation took place, which resulted in the blank rows after every record while reading. In fact, the above shown file data was internally stored as :

```
['Name', 'Points', 'Rank']\r
\n
['Shradha', '4500', '23']\r
\n
['Nishchay', '4800', '31']\r
\n
:
```

Every data record line was appended with EOL character '\r\n ' on Windows OS because of EOL translation

So what is the solution ? You are right again ☺. We should have created the file with **newline** argument set to null string (' ') so that no EOL translation could take place.

In fact, the above situation can be handled in *two* ways :

(*i*) **Method1.** Create/write onto csv file with **newline = ' '** argument in open( ) function so this situation would not arise. ***Programs 5.23(a) and (b)*** illustrate this solution.

(*ii*) **Method2.** To read from a file which is created with EOL translation (*i.e.*, no newline argument), open it with **newline** argument set to the EOL character of the OS where the csv file was created. For instance, if you created the csv file on a Windows OS, open file for reading with **newline** argument set as '\r\n' because the EOL on Windows is stored as '\r\n', *i.e.*, as :

```
open(<csvfilename>, <readmode>, newline = '\r\n')
```

Now the file will be read considering every '\r\n' character as newline.

Recall that the table 5.4 lists EOL characters on various operating systems.

Check Point 5.2

1. What are text files ?
2. What are binary files ?
3. What are CSV files ?
4. Name the functions used to read and write in plain text files.
5. Name the functions used to read and write in binary files.
6. Name the functions used to read and write in CSV files.
7. What is the full form of :
   (*i*) CSV (*ii*) TSV

[CBSE Sample Paper 2020-21]

Program 5.22 handles this situation using method 2 listed here. Let us have a look at it.

Program 5.22 The csv file (compresult.csv) used in the previous program was created on Windows OS where the EOL character is '\r\n'. Modify the code of the previous program so that blank lines for every EOL are not displayed.

```
import csv
with open("compresult.csv", "r", newline = '\r\n') as fh :
    creader = csv.reader(fh)
    for rec in creader :
        print(rec)
```

*Now the file will be read considering every '\r\n' character as **end of line** and not as a separate line*

The newline argument can be specified in independent open() statement as well as in the open() of with statement.

The output produced by above program is :

```
['Name', 'Points', 'Rank']
['Shradha', '4500', '23']
['Nishchay', '4800', '31']
['Ali', '4500', '25']
['Adi', '5100', '14']
```

See, no blank lines this time

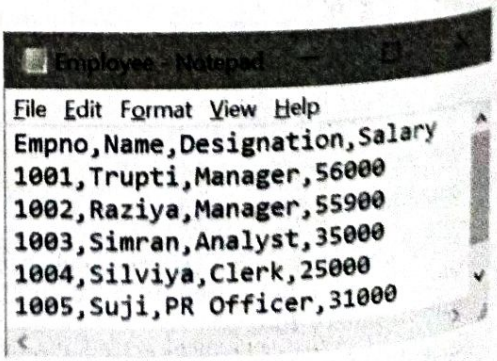
Let us now treat the earlier listed problem of blank lines in between records using the method 1 listed above. For this, we shall create the file with newline argument and then we shall not need any newline argument while reading as no EOL translation would have taken place.

Programs 5.23(*a*) and (*b*) are illustrating the same.

Program 5.23(a) Write a program to create a csv file by suppressing the EOL translation.

```
import csv
fh = open("Employee.csv", "w", newline = '')
ewriter = csv.writer(fh)
empdata = [
    ['Empno','Name','Designation','Salary'],
    [1001,'Trupti','Manager',56000],
    [1002,'Raziya','Manager',55900],
    [1003,'Simran','Analyst',35000],
    [1004,'Silviya','Clerk',25000],
    [1005,'Suji','PR Officer',31000 ]
    ]
ewriter.writerows(empdata)
print("File successfully created")
fh.close()
```

This argument ensures that no EOL translation takes place



The above program created a file as shown above.

Let us now read from the above created file. This time we need not specify the newline argument as no EOL translation has taken place. Program 5.23(*b*) is doing the same.

Program 5.23(b) Write a program to read and display the contents of Employee.csv created in the previous program.

```
import csv
with open("Employee.csv", "r") as fh :
    ereader = csv.reader(fh)
    print("File Employee.csv contains :")
    for rec in ereader :
        print(rec)
```

See, we did not specify newline argument in the open( ) as no EOL translation took place when the file was created.

The output produced by the above program is :

```
File Employee.csv contains :
['Empno', 'Name', 'Designation', 'Salary']
['1001', 'Trupti', 'Manager', '56000']
['1002', 'Raziya', 'Manager', '55900']
['1003', 'Simran', 'Analyst', '35000']
['1004', 'Silviya', 'Clerk', '25000']
['1005', 'Suji', 'PR Officer', '31000']
```

See, no blank lines in between the records this time because the csv file was created with no EOL translation.

With this we have come to the end of this chapter.

DATA FILES IN PYTHON — Progress In Python 5.1

PriP

This PriP session aims at giving you practical exposure to file handling in Python.

Fill it in PriP 5.1 under Chapter 5 of practical component-book – Progress in Computer Science with Python after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *A file in itself is a bunch of bytes stored on some storage devices like hard-disk, thumb-drive etc.*
- *The data files can be stored in three ways : (i) Text files (ii) Binary files (iii) CSV files.*
- *A text file stores information in ASCII or Unicode characters, where each line of text is terminated, (delimited) with a special character known as EOL (End of Line) character. In text files some internal manipulations take place when this EOL character is read or written.*
- *A binary file is just a file that contains information in the same format in which the information is held in memory, i.e., the file content that is returned to you is raw (with no translation or no specific encoding).*
- *The open( ) function is used to open a data file in a program through a file-object (or a file-handle).*
- *A file-mode governs the type of operations (e.g., read / write / append) possible in the opened file i.e., it refers to how the file will be used once it's opened.*
- *A text file and a csv file can be opened in these file modes : 'r' , 'w', 'a', 'r+', 'w+', 'a+'*
- *A binary file can be opened in these file modes : 'rb' , 'wb', 'ab', 'r+b' ('rb+'), 'w+b'('wb+'), 'a+b'('ab+').*
- *The three file reading functions of text files are : read( ), readline( ), readlines( )*
- *While read( ) reads some bytes from the file and returns it as a string, readline( ) reads a line at a time and readlines( ) reads all the lines from the file and returns it in the form of a list.*

- *The two writing functions for Python text files are write( ) and writelines( ).*
- *While write( ) writes a string in file, writelines( ) writes a list in a file.*
- *Pickle module's dump( ) and load( ) functions write and read into binary files.*
- *The input and output devices are implemented as files, also called standard streams.*
- *There are three standard streams : stdin (standard input), stdout (standard output) and stderr (standard error)*
- *The absolute paths are from the topmost level of the directory structure. The relative paths are relative to current working directory denoted as a dot(.) while its parent directory is denoted with two dots(..).*
- *Every open file maintains a file-pointer to determine and control the position of read or write operation in file.*
- *The seek( ) function places the file pointer at specified position.*
- *The tell( ) function returns the current position of file-pointer in open file.*
- *CSV files are delimited files that store tabular data (data stored in rows and columns as we see in spreadsheets or databases) where comma delimits every value.*
- *For writing onto a csv file, user data is written on a csv.writer object which converts the user data into delimited form and writes it on to the csv file.*
- *For reading from a csv file, **csv.reader** loads data from the csv file, parses it and makes it available in the form of an iterator wherefrom the records can be fetched row by row.*
- *CSV files should be opened with newline argument to suppress EOL translation.*

OBJECTIVE TYPE QUESTIONS OTQs

MULTIPLE CHOICE QUESTIONS

1. Information stored on a storage device with a specific name is called a _____ .
   (*a*) array (*b*) dictionary (*c*) file (*d*) tuple
2. Which of the following format of files can be created programmatically through Python to store some data ?
   (*a*) Data files (*b*) Text files (*c*) Video files (*d*) Binary files
3. To open a file c:\ss.txt for appending data, we use
   (*a*) file = open("c:\\ss.txt", "a") (*b*) file = open("c:\\ss.txt", "rw")
   (*c*) file = open(r"c:\ss.txt", "a") (*d*) file = open(file = "c:\ss.txt", "w")
   (*e*) file = open(file = "c:\\ss.txt", "w") (*f*) file = open("c:\res.txt")
4. To read the next line of the file from a file object *infi*, we use
   (*a*) infi.read(all) (*b*) infi.read( ) (*c*) infi.readline( ) (*d*) infi.readlines( )
5. To read the remaining lines of the file from a file object *infi*, we use
   (*a*) infi.read(all) (*b*) infi.read( ) (*c*) infi.readline( ) (*d*) infi.readlines( )
6. The readlines( ) method returns
   (*a*) str (*b*) a list of lines (*c*) a list of single characters (*d*) a list of integers
7. Which of the following mode will refer to binary data ?
   (*a*) r (*b*) w (*c*) + (*d*) b
8. Which of the following statement is not correct ? [CBSE Paper 2021 (Term 1)]
   (*a*) We can write content into a text file opened using 'w' mode.
   (*b*) We can write content into a text file opened using 'w+' mode.
   (*c*) We can write content into a text file opened using 'r' mode.
   (*d*) We can write content into a text file opened using 'r+' mode.