Chapter 5:

FILE HANDLING

Introduction:

A file 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.

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:

(01) Text files (02) Binary files

Text Files:

A text file stores information in 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 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').

Binary Files:

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 encodings). In binary file, there is no delimiter for a line. Also, no translation occur in binary files. As a result, binary files are faster and easier for a program to read and write than are text file. If 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.

Opening and Closing Files:

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.

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.text”)

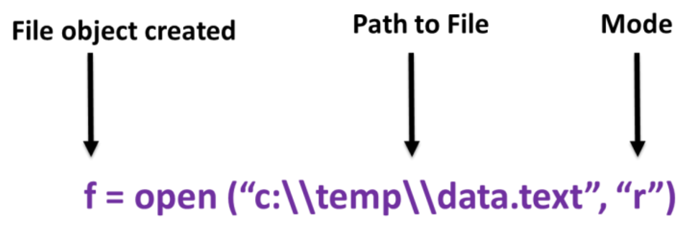
The above statement opens file “taxes.text” in file mode as read mode (default mode) and attaches it to file object namely myfile.

Consider another statement:

file2 = open (“data.txt”, “r”)

The above statement opens the file “data.txt” in real mode (because of “r” given as mode and attaches it to file object namely file2

- 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 would 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.



f = open (“c:\\temp\\data.text”, “r”)

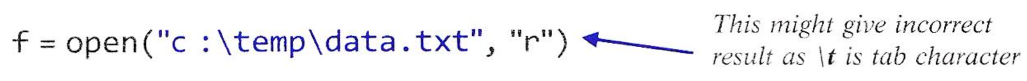
Figure summarizes the open () function of Python for you.

As you can see in figure. 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.

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

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")
```

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.text", "r")

(ii) Give raw string by prefixing the file-path string with an r e.g.,
  f = open (r"c:\temp\data.text", "r")

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 several different tasks.

File object (also called file-handle) is 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 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 is opened. File modes supported by Python are being given in the table.

Text File Mode	Binary File Mode	Description	Notes
'r'	'rb'	read only	❖ File must exist already, otherwise Python raises I/O error
'w'	'wb'	write only	❖ If the file doesn't 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 doesn't 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 it doesn't exist ❖ If file exists, file is truncated ❖ Both reading and writing can take place
'a+'	'a+b' or 'ab+'	write and read	❖ File is created if does not exist ❖ If file exists, files existing data is returned new data is appended ❖ Both reading and writing can take place

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).

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?

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

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 file handle

Reading from a text file:

Python provides three types of read functions to read from a data file.

read()

readline()

readlines()

read():

Method	Syntax	Description
read()	**<filehandle>.read([n])**	Reads at most n bytes ; if no n is specified, reads the entire file. Returns the read byte in form of a string. In [11]: file1=open("C:\\mydata\\info.txt") In [12]: readinfo=file1.read(15) In [13]: print(readinfo) It's time to go In [14]: type(readinfo) Out [14]: str

Example:

myfile = open(r’E\poem.text’,”r”)

str = myfile.read(30)

File object being used

file (r”E :\poem.txt’, “r”).read (30)

Would give the same result as that of above code.

Example:

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) $\longleftarrow$ **reading 30 bytes**

print(str)

str2 = myfile.read(50) $\longleftarrow$ **reading next 50 bytes**

print (str2)

myfile. close( )

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.

Example:

Reading a file’s entire content

myfile = open(r’E : \poem. Txt’, “r”)

str = myfile.read( )

print(str)

myfile.close( )

readline( ):

Method	Syntax	Description
readline()	<filehandle>.readline([n])	Reads a line of input; if N is specified reads at most N bytes. returns the re read bytes in the form of a string ending with In(line) character returns if blank string if no more bytes are left for reading in the file. In [20]: file1 = open("E:\\mydata\\info.txt") In [21]: readInfo = file1.readline() In [22]: print(readInfo) It's time to work with files.

readline():

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

Example:

Reading a file's first three lines – line by line

```
myfile = open(r ‘E:\poem.txt’, “r”)
str = myfile.readline( )
print(str, end = ‘ ‘ )
str = myfilel. readline( )
print(str, end = ‘ ‘)
str = myfile. readline( )
print(str, end = ‘ ‘)
myfile.close ( )
```

Example:

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 code will print the entire content of file poem. Txt.

The readline( ) function reads the leading and trailing space (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.

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 model>])
 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)
```

Example:

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 + ien(str1.strip( ) )
print(“Size of file after removing all EOL characters & blank lines:”, size)
print(“The TOTAL size of the file:”, tsize)
myfile.close( )
```

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”)
```

readlines():

Method	Syntax	Description
readlines()	<filehandle>.readlines([n])	reads all lines and returns them in a list In [23]: file1=open("E:\\mydata\\info.txt") In [24]: readInfo = file1.readlines() In [25]: print(readInfo) "It's time to work with files.\n", "Files offer

		ease and power to store to store your work/data/Information for later use."\n", "Simply create afile and store(write) in it.\n", "Or open an existing file and read from it.\n" In [26]: type(readInfo) Out[26]: list

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( )
```

Writing onto files:

After working with file-reading functions, let us talk about the writing functions for data files available in Python. 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.

Sl. No.	Name	Syntax	Description
1	write( )	<filehandle>.write(str1)	Write 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 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.

Example:

Create a file to hold some data

```
fileout = open("student.dat", "w")
for I in range (s) :
    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( )

Example:

Create a file to hold some data, separated as lines.

```
fileout = open("student.dat", "w")
for I in range(s) :
    name = input("Enter name of student : ")
    fileout.write(name)

    fileout.write('\n') ←——— The newline chapter '\n' written after every name

fileout.close( )
```

Example:

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

(For this, we shall use writelines( ) in place of write( ) function which writes the content of a list to a file. Function writelines( ) also, does not add newline character, so you must 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( )
```

Example:

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.det".

```
count = int(input("How many students are there in the class:") )

fileout = open ("Marks.det", "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

Example:

Write a program to add two more students' details to a file already created .

```
    fileout = open ("Marks.det", "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

Example:

Write a program to display the contents of file "Marks .det" created above.

```
      fileinp = open("Marks.det","r")

      while str :

          str = fileinp.readline( )

          print(str)

    fileinp.close( )
```

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 to buffer onto storage. You can use flush( ) function.

Python automatically flushes the files 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( )

Consider the following example code :

```
f = open ('out.log', 'wt')
f = write('The output is \n')
f.write ("My" + "work – status "+")
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

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 strip( ) removes the given character from trailing end i.e., right end.**
- **The istrip( ) 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')

2. **Removing the leading whilespaces from the line read from the file.**
   Line = file("poem.txt,"r"). readline( )

   Line = line.1strip( )

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.

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

>>> line = fh.readline( )

>>> len(line)

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

>>> len(line2)

>>> line3 = line.strip( )

>>> len(line3)
```

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:

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.

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 step1.

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.

Close the file:

This is important 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.

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

Modes and Opening Position of File-Pointer:

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

Table: 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.

Pogram: Write a program to display all the records in a file along with line/record number.

Solution:

```
fh = open("Result,det","r")
count = 0
rec = " "
while True:
      rec = fh.readline( )
      if rec = = " " :
          break
      count = count + 1
  print(count, rec, end = ' ')              # to suppress extra newline by print
  fh. Close( )
```

Python Data Files: With Statement

Python's with statement for files is very handy when you have two related operations which you would 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 :

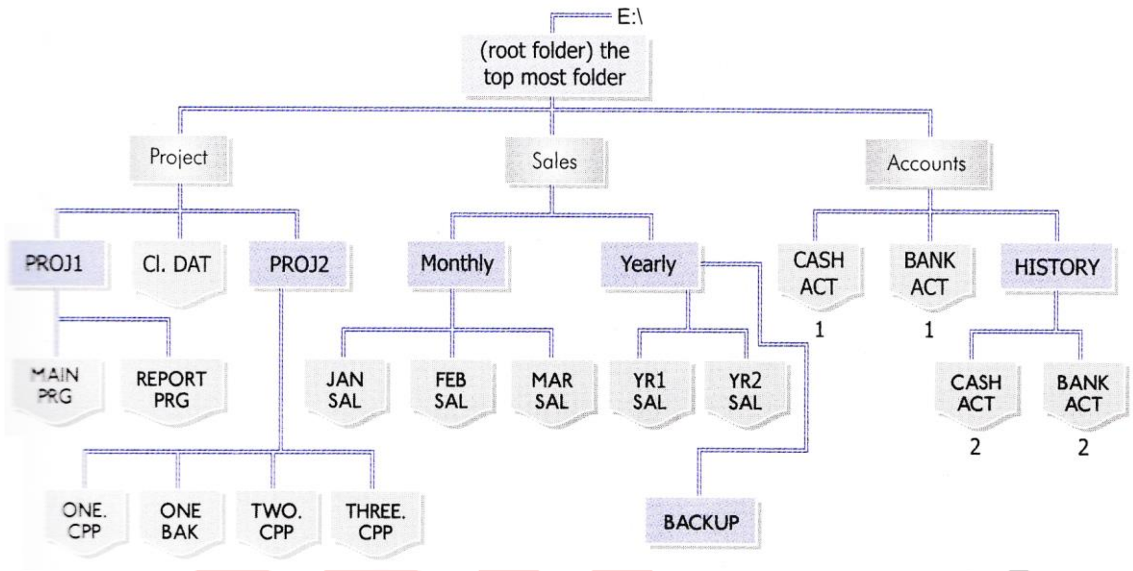
```
with open('output.txt', 'w') as f:
          f.write('HI there!')
```

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, it will close the file.

Absolute and Relative Path:

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 particular directory or file name. Let us consider the following structure:



Now the full name of directions 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 BACKUPS path will be:

E:\SALES\YEARLY\BACKUP

As to reach BACKUP the sequence one must 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

$\downarrow$

Refers to root directory

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 name, but their path names differ, therefore, these can exist on system. Therefore, CASH.ACT1's path will be

E:\ACCOUNTS\CASH.ACT

And CASH.ACT2'S path will be

E: \ACCOUTNS\HISOTRY\CASH.ACT

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

Binary File Operations:

Most of the files that we see in our computer system are called binary files.

Example:

- **Document files:** .pdf, .doc, .xls etc.
- **Image files:** .png, .jpg, .gif, .bmp etc.
- **Video files:** .mp4, .3gp, .mkv, .avi etc.
- **Audio files:** .mp3, .wav, .mka, .aac etc.
- **Database files:** .mdb, .accde, .frm, .sqlite etc.
- **Archive files:** .zip, .rar, .iso, .7z etc.
- **Executable files:** .exe, .dll, .class etc.

All binary files follow a specific format. We can open some binary files in the normal text editor, but we cannot read the content present inside the file. That is because all the binary files will be encoded in the binary format, which can be understood only by a computer or a machine.

In binary files, there is no delimiter to end a line. Since they are directly in the form of binary, hence there is no need to translate them. That is why these files are easy and fast in working.

operations performed using a binary file are:

- Inserting/Appending record in a binary file.
- Read records from a binary file.
- Search a record in a binary file.
- Update a record in a binary file

Reading and Writing to a Binary File:

Pickle module:

Python pickle module is used for serializing and de-serializing python object structures. The process to converts any kind of python objects (list, dict, etc.) into byte streams (0s and 1s) is called pickling or serialization or flattening or marshalling. Pickling is used to store python objects. This means things like lists, dictionaries, class objects, and more.

We can convert the byte stream (generated through pickling) back into python objects by a process called as unpickling.

Only after importing pickle module we can do pickling and unpickling. Importing pickle can be done using the following command –

```
import pickle
```

Pickle module is used for **Serialization** and **Deserialization**.

Serialization:

- Process to convert any kind of python objects (list, dict, etc.) into byte streams (0s and 1s) is called **pickling or serialization or flattening or marshalling.**
- It is used to store python objects like lists, dictionaries, class objects, and more.

De-Serialization:

- It is the reverse process of serialization.
- Process to convert byte streams (0s and 1s) back into python object is called **unpicking or De-serialization.**
- It is used to read from binary file.

pickle():

The two main methods of pickle module used for pickling and unpickling are:

dump()

load()

Writing To A Binary File: dump()

- The open() function opens a file in text format by default.

- To open a file in binary format, add 'b' to the mode parameter.
- The **"wb"** mode opens the file in binary format for writing.
- Use dump() to write records.

Pickle.dump(Rec, filehandle)

Example 1:(Write a list into a file)

```
import  pickle

f=open("binfile.bin","wb")

num=[5, 10, 15, 20, 25]

Pickle.dump(num,f)

f.close()
```

Example 2: (Writing 5 students records into a binary file)

```
import pickle

file = open('important', ‘wb')

for i in range(5):

        name= input('Enter name: ')

        roll= input('Enter roll: ')

        per= input('Enter per: ')

        str=[name,roll,per]

        pickle.dump(str, file)

file.close()
```

Example:

```
import pickle
file = open('important', 'wb')
```

```
n= int(input('Enter the number of data : '))
data = [ ]
for i in range(n):
   str = input('Enter data : ')
   data.append(str)
   pickle.dump(data, file)
file.close()
```

Reading from a binary file: load()

- load() function is used to retrieve pickled data.
- The argument of load function is the file object.
- It reads a single record from the file and returns it.
- **Syntax:**

  Rec= pickle. load(file handle)

Example: (Read and display 1st 3 records from a binary file):

```
import pickle
file = open('important.dat', 'rb')
rec1= pickle.load(file)
rec2= pickle.load(file)
rec3= pickle.load(file)
Print(rec1)
Print(rec2)
Print(rec3)
file.close()
```

Insert/Append a Record in Binary File:

Inserting or adding (appending) a record into a binary file requires importing pickle module into your program followed by dump() method to write onto the file.

Practical Implementation:

Program to insert/append 2 records in the binary file "student.dat"

```
import pickle

file = open('important', ‘ab')

for i in range(2):

        name= input('Enter name: ')

        roll= input('Enter roll: ')

        per= input('Enter per: ')

        str=[name,roll,per]

        pickle.dump(str, file)

file.close()
```

Searching a record in a binary file:

Searching the binary file "student" is carried out based on the roll number entered by the user. The file is opened in the read-binary mode and gets stored in the file object, f. load() method is used to read the object from the opened file. A variable ‘found’ is used which will tell the status of the search operation being successful or unsuccessful. Each record from the file is read and the contents of the field, roll no, is compared with the roll number to be searched. Upon the search being successful, appropriate message is displayed to the user as shown in the practical implementation that follows.

Practical Implementation:

Program to search a record from the binary file "student.dat" based on roll number.

```
import pickle
f = open("D:/student","rb")
stud_rec = pickle.load(f)
found = 0
rno = int(input("Enter the roll number to search:"))
```

```
for R in stud_rec:
    if R[0] == rno:
        print("Successful Search",R[1],"Found!")
        found  = 1
        break
if found == 0:
    print("Sorry, record not found")
f.close()
```

Updating a record in a binary file:

Updating a record in the file requires roll number to be fetched from the user whose name is to be updated.

Once the record is found, the file pointer is moved to the beginning of the file using seek(0) statement, and then the changed name is written to the file and the record is updated. seek() method is used for random access to the file.

Practical Implementation:

```
import pickle
f = open("D:/student","rb+")
stud_rec = pickle.load(f)
found = 0
rollno = int(input("Enter the roll number to search:"))
for R in stud_rec:
    rno = R[0]
    if rno == rollno:
        print("Current Name is:", R[1])
        R[1]=input("New Name:")
        found = 1
        break
```

```
if found == 1:
        f.seek(0)
pickle.dump(stud_rec,f)
print("Name Updated!!!")
f.close()
```

Random Access In Files Using tell() And seek():

Till now in all our programs we laid stress on the sequential processing of data in a text and binary file. But files in Python allow random access of the data as well using built-in methods seek() and tell().

seek()

seek() function is used to change the position of the file handle (file pointer) to a given specific position. File pointer is like a cursor, which defines from where the data has to be read or written in the file.

Python file method seek() sets the file's current position at the offset. This argument is optional and defaults to 0, which means absolute file positioning. Other values are: 1, which signifies seek is relative (may change) to the current position and 2 means seek is relative to the end of file. There is no return value.

The reference point is defined by the "from what" argument. It can have any of the three values.

0: sets the reference point at the beginning of the file, which is by default.
1: sets the reference point at the current file position.
2: sets the reference point at the end of the file.

seek() can be done in two ways:

- **Absolute Positioning**
- **Relative Positioning**

Absolute referencing using seek() gives the file number on which the file pointer has to position itself. The syntax for seek() is—

f.seek(file_location) **#where f is the file pointer**

For example, f.seek(20) will give the position or file number where the file pointer has been placed. This statement shall move the file pointer to 20th byte in the file no matter where you are.

Relative referencing/positioning has two arguments, offset and the position from which it must traverse. The syntax for relative referencing is:

f.seek(offset,from_what) **#where f is file pointer**

For example:

f.seek(–10,1) from current position, move 10 bytes backward

f.seek(10,1) from current position, move 10 bytes forward

f.seek(–20,1) from current position, move 20 bytes backward

f.seek(10,0) from beginning of file, move 10 bytes forward

tell()

tell() returns the current position of the file read/write pointer within the file. Its syntax is:

f.tell() **#where f is file pointer**

- When you open a file in reading/writing mode, the file pointer rests at 0th byte.
- When you open a file in append mode, the file pointer rests at last byte.

Introduction to CSV:

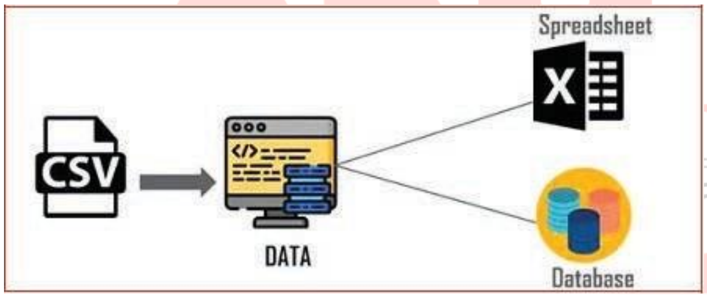
In every span of today's organizational working environment, data sharing is one of the major tasks to be carried out, largely through spreadsheets or databases.

A basic approach to share data is through the comma separated values (CSV) file.

CSV is a simple flat file in a human readable format which is extensively used to store tabular data, in a spreadsheet or database. A CSV file stores tabular data (numbers and text) in plain text.

Files in the CSV format can be imported to and exported from programs that store data in tables, such as Microsoft Excel or OpenOffice Calc.

Already defined, CSV stands for "**comma separated values**". Thus, we can say that a comma-separated file is a delimited text file that uses a comma to separate values.

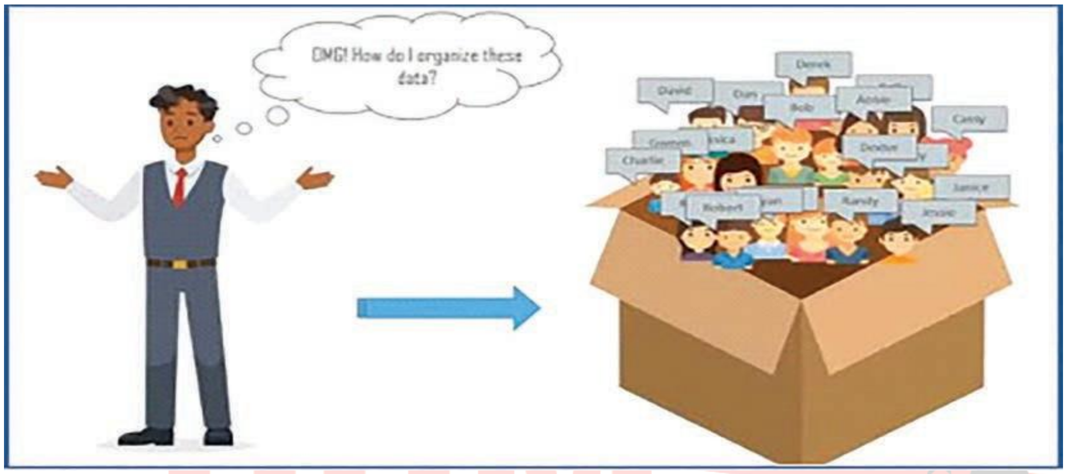


CSV Stands for Comma Separated Values
It is used for storing tabular data in spreadsheet or database
CSV
Each record consists of fields separated by commas(delimiter)
Each line of the file is called a record

Each line in a file is known as **data/record**. Each record consists of one or more fields, separated by commas (also known as delimiters), *i.e.*, each of the records is also a part of this file. Tabular data is stored as **text** in a CSV file. The use of comma as a field separator is the source of the name for this file format. It stores our data into a spreadsheet or a database.

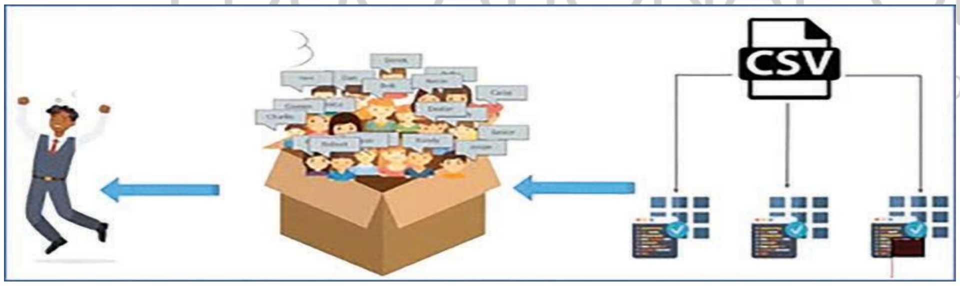
Why Use CSV:

With the use of social networking sites and its various associated applications being extensively used requires the handling of huge data. The problem arises as to how to handle and organize this large unstructured data as shown in the Figure given.



Problem of Huge Data

The solution to the above problem is CSV . Thus, the CSV organizes data into a structured form and, hence, the proper and systematic organization of this large amount of data is done by CSV. Since CSV files formats are of plain text format, it makes it quite easy for website developers to create applications that implement CSV.



CSV files are commonly used because they are easy to read and manage, small, and fast to process/transfer. Because of these salient features, they are frequently used in software applications, ranging anywhere from online e-commerce stores to mobile apps to desktop tools. *For example*, Magento, an e-commerce platform, is known for its support of CSV.

Thus, in a nutshell, the several advantages that are offered by CSV files are as follows:

- CSV is faster to handle.
- CSV is smaller in size.

- CSV is easy to generate and import onto a spreadsheet or database.
- CSV is human readable and easy to edit manually.
- CSV is simple to implement and parse.
- CSV is processed by almost all existing applications.

After understanding the concept and importance of using CSV files, we will now discuss the read and write operations on CSV files.

CSV File Handling In Python:

For working with CSV files in Python, there is an inbuilt module called **CSV**. It is used to read and write tabular data in CSV format.

Therefore, to perform read and write operation with CSV file, we must import **CSV module**. CSV module can handle CSV files correctly regardless of the operating system on which the files were created.

Along with this module, open() function is used to open a CSV file, and return file object. We load the module in the usual way using import:

>>> import csv

Like other files (text and binary) in Python, there are two basic operations that can be carried out on a CSV file.

- Reading a CSV

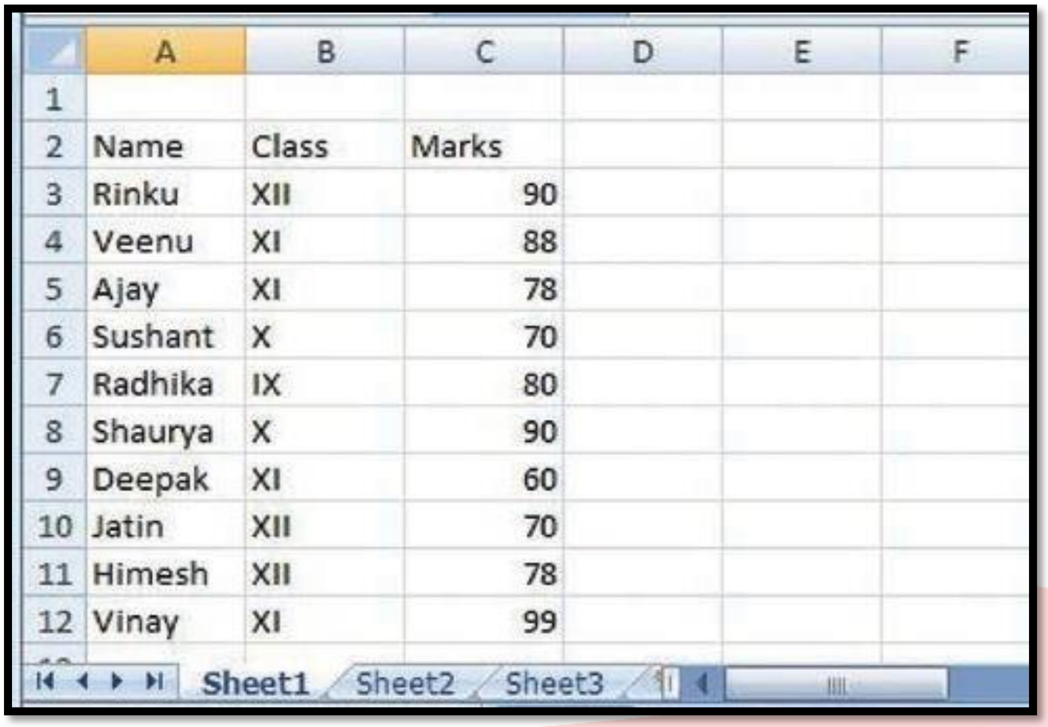
1. Writing to a CSV.

Let us discuss these CSV operations.

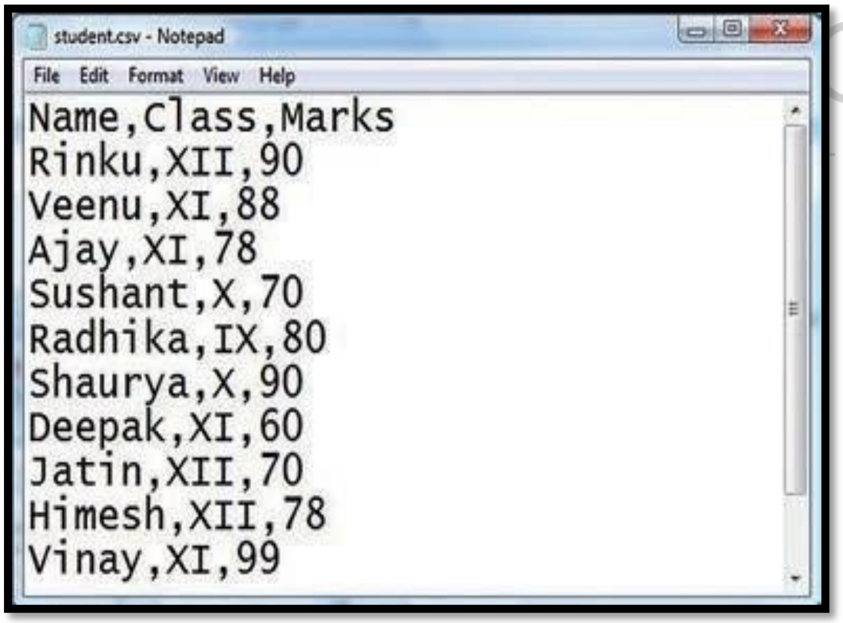
Reading from CSV File:

- Reading from a CSV file is done using the reader object.
- The CSV file is opened as a text file with Python's built-in open() function, which returns a file object.
- This creates a special type of object to access the CSV file (reader object), using the reader() function.
- The reader object is an iterable that gives us access to each line of the CSV file as a list of fields.
- You can also use next() directly on it to read the next line of the CSV file, or you can treat it like a li st in a for loop to read all the lines of the file (as lists of the file's fields).
- This is shown in the practical implementation given below.
- Before this, enter the student details in spreadsheet and save this file as shown:

- Next step is to open the Notepad and enter the data for student.csv, which will be the equivalent for student.xls.



	A	B	C	D	E	F
1						
2	Name	Class	Marks			
3	Rinku	XII	90			
4	Veenu	XI	88			
5	Ajay	XI	78			
6	Sushant	X	70			
7	Radhika	IX	80			
8	Shaurya	X	90			
9	Deepak	XI	60			
10	Jatin	XII	70			
11	Himesh	XII	78			
12	Vinay	XI	99			



```
Name,Class,Marks
Rinku,XII,90
Veenu,XI,88
Ajay,XI,78
Sushant,X,70
Radhika,IX,80
Shaurya,X,90
Deepak,XI,60
Jatin,XII,70
Himesh,XII,78
Vinay,XI,99
```

- In student.csv (notepad) file, first line is the header and remaining lines are the data/records.

- The fields are separated by comma, or we may say the separator character.
- In general, the separator character is called a delimiter, and the comma is not the only one used.
- Other popular delimiters include the tab (\t), colon (:) and semi-colon (;) characters.

Practical Implementation:

Write a program to read the contents of "student.csv" file.

```
import csv

f = open(C:/Users/admin/Documents/student.csv",'r')

csv_reader = csv.reader(f)

for row in csv_reader:

    print(row)

f.close()
```

Explanation:

- As seen from the above output, every record is stored in reader object in the form of a List.
- In the above code, we first open the CSV file in READ mode.
- The file object is named as **f**.
- The file object is converted to csv.reader object.
- We save the csv.reader object as **csv_reader**.
- The reader object is used to read records as lists from a csv file.
- Now, we iterate through all the rows using a for loop. When we try to print each row.
- One can find that row is nothing but a list containing all the field values.
- Thus, all the records are displayed as lists separated by comma.

Practical Implementation:

Write a program to read the contents of "student.csv" file using with open().

```
import csv
with open ("student.csv",'r')  as  csv_file:
        reader = csv.reader(csv_file)
        rows = []
        for rec in reader:
                rows.append(rec)
                print(rows)
```

Practical Implementation:

Write a program to count the number of records present in "student.csv" file.

```
import csv
f = open("student.csv",'r')
csv_reader = csv.reader(f)
columns = next(csv_reader)
c = 0
for row in csv_reader:
        c = c + 1
print("No. Of records are:",c)
```

Writing to CSV File:

- To write to a CSV file in Python, we can use the csv.writer() function.
- The csv.writer() function returns a writer object that converts the user's data into a delimited string.
- This string can later be used to write into CSV files using the writerow() function.
- In order to write to a CSV file, we create a special type of object to write to the CSV file "writer" .

writerow():

- Writer object, which is defined in the CSV module is created using the writer() function.
- The writerow() method allows us to write a list of fields to the file.
- The fields can be strings or numbers or both.

- Also, while using writerow(), you do not need to add a new line character (or other EOL indicator) to indicate the end of the line, writerow() does it for you as necessary.

Practical Implementation:

Program to write data onto “student” CSV file using writerow() method.

```
import csv
fields = ['Name', 'Class', 'Year', 'Percent']
rows = [ ['Rohit', 'XII', '2008', '92'],
         ['Shaurya', 'XI', '2004', '82'],
         ['Deep', 'XII', '2002', '80'],
         ['Prerna', 'XI', '2006', '85'],
         ['Lakshya', 'XII', '2005', '72'] ]
filename = "D:/marks.csv"
with open(filename,'w',newline=' ') as f:
    csv_w = csv.writer(f,delimiter = ',')
    csv_w.writerow(fields)
    for i in rows:
        csv_w.writerow(i)
```

Explanation:

- The very first line is for importing csv file into your program.
- Next, the column headings for our data.
- All the data stored inside these fields are placed inside variable rows.
- student.csv will be created and stored inside your current working directory or the path that you mentioned.
- ‘w’ stands for write mode and we are using the file by opening it using "with open statement”.
- csv.writer() is called to obtain a writer object and store it in the variable csv_w .
- writer() takes the name of file object ‘f’ as the argument. By default, the delimiter is comma (,).
- writerow(fields) is going to write the fields which are the column headings into the file.
- Using for loop, rows are traversed from the list of rows from the file. writerow(i) is writing the data row-wise in the file.

- While giving csv.writer(), the delimiter taken is comma.
- We can change the delimiter whenever and wherever required by changing the argument passed to delimiter attribute.

writerows():

- writerows() can write multiple rows into a CSV file provided it must be in the form of a nested sequence.

rows=[[“akash”,1,20], [“ankita”,2,30], [“azhar”,3,20], [“ahmed”,4,40]]

csv_w.writerows(rows)

Program to write data onto "student" csv file using writerows() method (modification of Practical Implementation).

```
import csv

fields = ['Name', 'Class', 'Year', 'Percent']

rows = [ ['Rohit', 'XII', '2008', '92'],['Shaurya', 'XI', '2004', '82'],['Deep', 'XII', '2002', '80'],

                    ['Prerna', 'XI', '2006', '85'],['Lakshya', 'XII', '2005', '72'] ]

filename = "D:/newmarks.csv"

with open(filename,'w',newline=' ') as f:

    csv_w = csv.writer(f,delimiter = ',')

    csv_w.writerow(fields)

    csv_w.writerows(rows)
```

****