Files in Python

We use different files in our daily life. For example, we may keep all our school certificates in a file and call it a ‘certificates’ file. This file contains several certificates or pages and each page some data. From this example can simply say that a file is a place for storing data. Similarly, when we go to an office, we can see a book that contains the attendance of the employees. This book is called ‘attendance’ file since it contains attendance data of employees. Let's understand that if the data is stored in a place, it is called a file.

Files

Data is very important. Every organization depends on its data for continuing its business operations. If the data is lost, the organization has to be closed. This is the reason computers are primarily created for handling data, especially for storing and retrieving data. In later days, programs are developed to process the data that is stored in the computer.

To store data in a computer, we need files. For example, we can store employee data like employee number, name and salary in a file in the computer and later use it whenever we want. Similarly, we can store student data like student roll number, name and marks in the computer. In computers' view, a file is nothing but collection of data that is available to a program. Once we store data in a computer file, we can retrieve it and use it depending on our requirements.

Types of Files in Python

In Python, there are two types of files. They are:

- Text files
- Binary files

Text files store the data in the form of characters. For example, if we store the employer name "Ganesh", it will be stored as 6 characters and the employee salary 8900.75 is stored as 7 characters. Normally, text files are used to store characters or strings.

Binary files store entire data in the form of bytes, i.e. a group of 8 bits each. For example a character is stored as a byte and an integer is stored in the form of 8 bytes (on a 64 bit machine). When the data is retrieved from the binary file, the programmer can retrieve the data as bytes. Binary files can be used to store text, images, audio and video.

Image files are generally available in jpg, gif or .png formats. We cannot use text files to store images as the images do not contain characters. On the other hand, images contain pixels which are minute dots with which the picture is composed of. Each pixel can be represented by a bit, i.e. either 1 or 0. Since these bits can be handled by binary files, we can say that they are highly suitable to store images.

It is very important to know how to create files, store data in the files and retrieve the data from the files in Python. To do any operation on files, first of all we should open the files.

Opening a File

We should use the open() function to open a file. This function accepts ‘filename' and 'open mode' in which to open the file.

file handler = open("file name", "open mode", "buffering")

Here, the file name' represents a name on which the data is stored. We can use any name to reflect the actual data. For example, we can use 'empdata' as a file name to represent the employee data. The file 'open mode' represents the purpose of opening the file specifies the file open modes and their meanings:

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.

+	Opens a file for updating (reading and writing)

A buffer represents a temporary block of memory. 'buffering' is an optional integer used to set the size of the buffer for the file. In the binary mode, we can pass 0 as a buffering integer to inform us not to use any buffering. In text mode, we can use 1 for buffering to retrieve data from the file one line at a time. Apart from these, we can use any positive integer for buffering. Suppose, we use 500, then a buffer of 500 bytes size is used via which the data is read or written. If we do not mention any buffering integer, then the default buffer size used is 4096 or 8192 bytes.

When the open() function is used to open a file, it returns a pointer to the beginning of the file. This is called 'file handler' or 'file object'. As an example, to open a file for storing data into it, we can written the open() function as:

```
f = open("myfile.txt", "w")
```

Here, T represents the file handler or file object. It refers to the file with the name "myfile.txt" that is opened in "w" mode. This means, we can write data into the file but we cannot read data from this file. If this file exists already, then its contents are deleted and the present data is stored into the file.

Closing a File

A file which is opened should be closed using the close() method. Once a file is opened but not closed, then the data of the file may be corrupted or deleted in some cases. Also, if the file is not closed, the memory utilized by the file is not freed, leading to problems like insufficient memory. This happens when we are working with several files simultaneously. Hence it is mandatory to close the file.

```
f.close()
```

Here, the file represented by the file object T is closed. It means f is deleted from the memory. Once the file object is lost, the file data will become inaccessible. If we want to do any work with the file again, we should once again open the file using the open) function.

In the program below, we are creating a file where we want to store some characters. We know that a group of characters represent a string. After entering a string from the keyboard using input() function, we store the string into the file using write() method as:

f.write(str)

In this way, write() can be used to store a character or a group of characters (string) into a file represented by the file object ‘f’.

A Python program to create a text file to store individual characters.

```
f = open('myfile.txt', 'w')
str =input('Enter text: ')
f.write(str)
f.close()
```

Output:

```
Enter text: Hi
>>>
```

In the program above, characters typed by us are stored into the file 'myfile.txt'. This file can be checked in the current directory where the program is executed. If we run this program again, the already entered data in the file is lost and the file will be created as a fresh file without any data. In this way, new data entered by us will only be stored into the file and the previous data is lost. If we want to retain the previous data and want to add the new data at the end of the file, then we have to open the file in append mode as:

f = open('myfile.txt', 'a')

The next step is to read the data from 'myfile.txt' and display it on the monitor. To read data from a text file, we can use read() method as:

str = f.read()

This will read all characters from the file f and return them into the string 'str'. We can also use the read() method to read only a specified number of bytes from the file as:

str= f.read (n)

where 'n' represents the number of bytes to be read from the beginning of the file.

A Python program to read characters from a text file.

```
f = open('myfile.txt', 'r')
str = f.read()
print(str)
f.close()
```

Output:

```
==== RESTART: (
Hi
>>>
```

If in the above program, we use the read() method as:

str = f.read (4)

Then it will read only the first 4 bytes of the file and hence the output will be:

```
==== RESTAR
Hi
>>>
```

Working with Text Files Containing Strings

To store a group of strings into a text file, we have to use the write() method inside a loop For example, to store strings into the file as long as the user does not type 'g' symbol, we can write while loop as:

```
while str!='@':
    if(str!='@'):
        f.write(str+"\n")
```

Please observe the "\n" at the end of the string inside write() method. The write() method writes all the strings sequentially in a single line. To write the strings in different lines, we are supposed to add the "\n" character at the end of each string.

A Python program to store a group of strings into a text file.

```
f=open('myfile.txt', 'w')
print("Enter text (@ at end): ")
while str!='@':
    str=input()
    if(str!='@'):
        f.write(str+"\n")
f.close()
```

Output:

```
==== RESTART: C:/Users/user.
Enter text (@ at end):
This is my file line one
This is line two
@
```

Now, to read the strings from the file, we can use the read() method in the following way:

f.read()

This method reads all the lines of the text file and displays them line by line as they were stored in the "myfile.txt". Consider the following output:

This is my file line one.
This is line two.

There is another method by the name readlines() that reads all the lines into a list. This can be used as:

f.readlines()

This method displays all the strings as elements in a list. The "\n" character is visible at the end of each string, as:

['This is my file line one. \n', 'This is line two.\n']

If we want to suppress the "\n" characters, then we can use read() method with

splitlines() method as:

f.read().splitlines ()

In this case the output will be:

['This is my file line one.', 'This is line two.']

A Python program to read all the strings from the text file and display them.

```
f = open('myfile.txt', 'r')
print('The file contents are:')
str = f.read()
print (str)
f.close()
```

Output:

```
The file contents are:
Hi
How are you?
```

Knowing Whether a File Exists or Not

The operating system (os) module has a sub module by the name 'path' that contains a method isfilet). This method can be used to know whether a file that we are opening really casts or not. For example, os.path.isfile(fname) gives True if the file exists otherwise False. We can use it as:

```
if os.path.isfile(fname):
    f=open(fname,'r')
else:
    print (fname+' does not exist')
    sys.exit()
```

In the program below, we are accepting the name of a file from the keyboard, checking whether the file exists or not. If the file exists, then we display the contents of the file, otherwise. we display a message that the file does not exist and then terminate the program.

A Python program to know whether a file exists or not.

```
import os, sys
fname=input("Enter filename: ")

if os.path.isfile(fname):
    f=open(fname,'r')
else:
    print(fname+'does not exist')
    sys.exit()

print("The file contents are: ")
str=f.read()
print(str)

f.close()
```

Output:

```
==== RESTART: C:/Users/user/.
Enter filename: myfile.txt
The file contents are:
Hi
How are you?

==== RESTART: C:/Users/
Enter filename: file3
file3does not exist
>>>
```

Working with Binary Files

Binary files handle data in the form of bytes. Hence, they can be used to read or write text, images or audio and video files. To open a binary file for reading purposes, we can use ‘rb' mode. Here, b' is attached to T to represent that it is a binary file. Similarly to write bytes into a binary file, we can use 'wb' mode. To read bytes from a binary file, we can use the read() method and to write bytes into a binary file, we can use the write() methods. Let's write a program where we want to open an image file like jpg, gif or png file and read bytes from that file. These bytes are then written into a new binary file. It means we are copying an image file as another file.

A Python program to copy an image file into another file.

```
f1=open('dog .png', 'rb')
f2= open('new.png', 'wb')

bytes=f1.read()
f2.write(bytes)

f1.close()
f2.close()
```

Output:

```
===== RESTART: C:/Users/user/AppData/Local/Programs/Python/Python38/copy.py ====
>>>
```

When the above program is run, it will copy the image file ‘dog.png' into another file ‘new.png' In this program, our assumption is that the file ‘dog.png' is already available in the current directory. Current directory is the directory where our program is running. Suppose, if the dog.png is not available in the current directory, but it is in another directory, then we have to supply the path of that directory to open() function, as:

f1= open("C:\\Users\\user\\Desktop\\dog.png', 'rb')

The preceding statement tells that the dog .jpg file is available in 'Users' sub directory in 'C drive. Please remember, the double backslashes in the path above are interpreted as single backslashes only.

The with Statement

The 'with’ statement can be used while opening a file. The advantage of with statement is that it will take care of closing a file which is opened by it. Hence, we need not close the file explicitly.

In case of an exception also, ‘with’ statement will close the file before the exception is handled. The format of using 'with' is:

with open("filename", "openmode") as fileobject:

A Python program to use with' to open a file and write some strings into the file.

```
with open('sample.txt', 'w') as f:
    f.write('I am a learner\n')
    f.write('Python is Interesting\n')
```

Output:

```
==== RESTART: C:/Users/user/AppData/Local/Programs/Python/Python38/with1.py ====
>>>
```

Pickle in Python

So far, we wrote some programs where we stored only text into the files and retrieved the same text from the files. These text files are useful when we do not want to perform any calculations on the data. What happens if we want to store some structured data in the files? For example, we want to store some employee details like employee identification number (int type), name (string type) and salary (float type) in a file. This data is well structured and has different types. To store such data, we need to create a class Employee with the instance variables id, name and sal as shown here:

```
class Emp:
    def __init__(self,id, name, sal):
        self.id = id
        self.name = name
        self.sal=sal

    def display (self):
        print("{:5d} {:20s} {:10.2f}". format (self.id, self.name,self.sal))
```

Then we create an object to this class and store actual data into that object. Later, this object should be stored into a binary file in the form of bytes. This is called pickle or serialization. So, let's understand that pickle is a process of converting a class object into a byte stream so that it can be stored into a file. This is also called object serialization. Pickling is done using the dump() method of 'pickle' module as:

pickle.dump(object, file)

The preceding statement stores the 'object' into the binary file. Once the objects are stored into a file, we can read them from the file at any time. Unpickle is a process whereby a byte stream is converted back into a class object. It means, unpickling represents reading the class objects

from the file. Unpickling is also called deserialization. Unpickling is done using the load() method of 'pickle' module as:

object = pickle.load(file)

Here, the load() method reads an object from a binary 'file' and returns it into 'object". Let's remember that pickling and unpickling should be done using binary files since they support byte streams. The word stream represents data flow. So, the byte stream represents the flow of bytes.

We will understand pickling and unpickling more clearly with the help of an example. First of all let's create an Emp class that contains employee identification number, name and salary. This is shown in the program below.

A Python program to create an Emp class with employee details as instance variables.

```
class Emp:
    def __init__(self,id, name, sal):
        self.id = id
        self.name = name
        self.sal=sal

    def display (self):
        print("{:5d} {:20s} {:10.2f}". format (self.id, self.name,self.sal))
```

Output:

```
===== RESTART: C:/Users/user/AppData/Local/Programs/Python/Python38/Emp.py =====
>>>
```

Our intention is to pickle Emp class objects. For this purpose, we have to import the Emp.py file as a module since Emp class is available in that file.

import Emp

Now, an object to Emp class can be created as:

e=Emp.Emp(id, name, sal)

Please observe that this is the object of Emp class. Since the Emp class belongs to the Emp module, we referred to it as Emp.Emp(). This object 'e' should be written to the file using dump() method of pickle module, as:

pickle.dump(e. f)

A Python program to pickle Emp class objects.

```
import Emp, pickle
f=open('emp.dat', 'wb')
n = int(input('How many employees? '))

for i in range(n):
    id= int(input('Enter id: '))
    name=input('Enter name: ')
    sal=float(input('Enter salary: '))
    e=Emp.Emp(id,name,sal)
    pickle.dump(e,f)

f.close()
```

Output:

```
How many employees? 2
Enter id: 100
Enter name: Sai
Enter salary: 9000.00
Enter id: 101
Enter name: Ravi
Enter salary: 8900.50
```

In the previous program, we stored 2 Emp class objects into the emp.dat file. If we want to get back those objects from the file, we have to unpickle them. To unpickle, we should use load() method of pickle module as:

obj pickle.load(f))

This method reads an object from the file and returns it into 'obj' Since this object obj belongs to Emp class, we can use the display method by using the Emp class to display the data of the object as:

obj.display()

In this way, using a loop, we can read objects from the emp.dat file. When we reach the end of the file and could not find any more objects to read, then the exception EOFError will occur. When this exception occurs, we should break the loop and come out of it. This is shown in the program below.

A Python program to unpickle Emp class objects.

```
import Emp, pickle
f= open('emp.dat', 'rb')
print('Employee Details:')

while True:
    try:
        obj=pickle.load(f)
        obj.display()

    except EOFError:
        print('End of file reached...')
        break

f.close()
```

Output:

```
===== RESTART: C:/Users/user/AppData/Loc
Employee Details:
  100 Sai                    9000.00
  101 Ravi                   8900.50
End of file reached...
```

The seek() and tell() Methods

We know that data in the binary files is stored in the form of bytes. When we conduct reading or writing operations on a binary file, a file pointer moves inside the file depending on how many bytes are written or read from the file. For example, if we read 10 bytes of data from a file, the file pointer will be positioned at the 10th byte so that it is possible to continue reading from the 11 byte onwards. To know the position of the file pointer, we can use the tell method. It returns the current position of the file pointer from the beginning of the file. It is used in the form:

n = f.tell()

Here, T represents file handler or file object. 'n' is an integer that represents the byte position where the file pointer is positioned.
In case, we want to move the file pointer to another position, we can use the seek() method. This method takes two arguments:

f.seek(offset, fromwhere)

Here, "offset" represents how many bytes to move. ‘fromwhere’ represents from which position to move. For example, ‘fromwhere' can be 0, 1 or 2. Here, 0 represents the beginning of the file, 1 represents the current position and 2 represents the ending of the file. The default value of 'fromwhere' is 0, i.e. beginning of the file.

f.seek(10)

This will move the file pointer to the 11th byte (i.e. 10+1) from the beginning of the file (0 represents beginning of the file). So, any reading operation will read data from 11 byte onwards.

f.seek(-10, 2)

This will move the file pointer to the 9th byte (-10+1) from the ending of the file (2 represents ending of the file). The negative sign before 10 represents moving back in the file.

We will take an example to understand how these methods work in case of a binary file.Observe the following code snippet where we open a binary file 'line.txt' in ‘r+b’ mode so that it is possible to write data into the file and read data from the file.

```
with open('line.txt', 'r+b') as f:
    f.write(b'Amazing Python')
```

The file object is ‘f’. The write() method is storing a string ‘Amazing Python’ into the file. Observe ‘b' prefixed with this string to consider it as a binary string. The string is stored in the file

First, we will move the file pointer to the 4th byte using seek() as:

f.seek (3)

The preceding method will put the file pointer at 3+1=4th position. So, the file pointer will be at 2. If we print 2 bytes using read() method as:

print(f.read(2))

This will display ‘zi’. Now, to know the position of the file pointer we can use the tell() method as:
print(f.tell())

This will display 5. Now, to move the file pointer to 5 position from the ending of the file, we can use seek() method as:

f.seek(-6, 2)

This will move the file pointer to -6+1=5th position. It means it will be positioned at the character ‘y'. We can display this character using read() method as:

print (f.read(1))

This will display ‘y'. Now, we can find the position of the file pointer using the tell() method as:

print(f.tell())

This will display 10 as the character y' is at 10 position from the beginning of the file. Please remember the tell() method always gives the positions from the beginning of the file.

Random Accessing of Binary Files

Data the binary files is stored in the form of continuous bytes. Let's take a binary file having 1000 bytes of data. If we want to access the last 10 bytes of data, it is not needed to search the file byte by byte from the beginning. It is possible to directly go to 991 byte using the seek() method as:

f.seek(900)

Then read the last 10 bytes using the read() method as:

f.read(10)

In this way, directly going to any byte in the binary file is called random accessing. This is possible by moving the file pointer to any location in the file and then performing reading or writing operations on the file as required. A problem with binary files is that they accept data in the form of bytes or in binary format. For example, if we store a string into a binary file, as shown in the following statement, we will end up with an error.

```
str="Dear"
with open('data.bin', 'wb') as f:
    f.write(str)
    f.write('Hello')
```

The preceding code will display the following error:

TypeError: 'str does not support the buffer interface. The reason behind this error is that we are trying to store strings into a binary file without converting them into binary format. So, the solution is to convert the ordinary strings into binary format before they are stored into the binary file. Now consider the following code:

```
str="Dear"
with open('data.bin', 'wb') as f:
    f.write(str.encode())
    f.write(b'Hello')
```

Converting a string literal into binary format can be done by prefixing the character before the string, as: b'Hello'. On the other hand, to convert a string variable into binary format, we have to use the encode() method, as:

str.encode().

The encode() method represents the string in byte format so that it can be stored into the binary file. Similarly, when we read a string from a binary file, it is advisable to convert it into ordinary text format using decode() method, as:

str.decode().

In the following program, we are creating a binary file and storing a group of strings. One way to view the data of a binary file is as a group of records with fixed length. For example, we want to take 20 bytes (or characters) as one record and store several such records into the file. Program 14 demonstrates how to store names of cities as a group of records into cities.bin file. In this program, each record length is taken as 20. So, if the city name entered by the user is less than 20 characters long, then the remaining characters in the record will be filled with spaces. For example, if the user stores the city name 'Delhi', since 'Delhi' has only 5 characters, another 15 spaces will be attached at the end of this name and then it is stored into the file. In this way, each record is maintained with fixed length. Consider the following code:

```
ln=len(city)
city=city+ (20-ln)*' '
```

A Python program to create a binary file and store a few records. create cities.bin file with cities names

```
reclen=20
with open("cities.bin", "wb") as f:
    n = int(input('How many entries? '))
    for i in range(n):
        city=input('Enter city name: ')
        ln=len(city)
        city = city +(reclen-ln)*' '
        city = city.encode()
        f.write(city)
```

Output:

```
==== RESTART: C:/Users/user/I
How many entries? 2
Enter city name: Hyderabad
Enter city name: Pune
```

Now, the cities.bin file is created with 2 records. In each record, we stored the name of a city.The next step is to display a specific record from the file. This is shown in the program below. In this program, we accept the record number and display that record. Suppose, we want to display the 2nd record, first we should put the file pointer at the end of the 1st record. This is done by seek() method, as:

```
reclen=20
f.seek(reclen*1)
```

Once this is done, we can read the next 20 bytes using the read() method that gives the 2nd record.

A Python program to randomly access a record from a binary file.

```
reclen = 20
with open('cities.bin', 'rb') as f:
    n = int(input('Enter record number: '))
    f.seek(reclen * (n-1))
    str = f.read(reclen)
    print (str.decode())
```

Output:

```
Enter record number: 2
Pune
```

Random Accessing of Binary Files using mmap

mmap - ‘memory mapped file' is a module in Python that is useful to map or link to a binary file and manipulate the data of the file as we do with the strings. It means, once binary file is created with some data, that data is viewed as strings and can manipulated using mmap module. The first step to use the mmap module is to map the file using the mmap() method as:

mm = mmap.mmap (f. fileno(), 0)

This will map the currently opened file (i.e. 'f') with the file object 'mm'. Please observe the arguments of the mmap() method. The first argument is 'f.fileno().This indicates that filen is a handle to the file object 'f'. This represents the actual binary file that is being mapped. The

second argument is zero(0) represents the total size of the file should be considered for mapping. So, the entire file represented by the file object 'f' is mapped in memory to the object 'mm'. This means, 'mm' will now onwards behave like the file 'f'.

Now, we can read the data from the file using read() or readline() methods as:

```
print (mm. read())
print (mm. readline())
```

Also, we can retrieve data from the file using slicing operator as:

print (mm[5:])
print (mm[5:10])

It is also possible to modify or replace the data of the file using slicing as:

mm [5:10]=str

We can also use find() method that returns the first position of a string in the file as:

n = mm. find (name)

We can also use seek() method to position the file pointer to any position we want as:

mm. seek (10, 0)

Let's remember that the memory mapping is done only for the binary files and not for the text files. So, first of all let's create a binary file where we want to store some strings. Suppose we want to store name and phone number into the file, we can use write method as:

f.write(name+phone)

This will store name and phone number as one concatenated string into the file object ‘f’. But we are supposed to convert these strings into binary (bytes) format before they are stored into the file. For this purpose, we should use encode() method as:

name = name.encode()

The encode() method converts the strings into bytes so that they can be written into the file Similarly, while reading the data from the file, we get only binary strings from the file. This binary string can be converted into ordinary string using decode() method as:

ph=ph.decode()

Now, let's write a program to create a phone book that contains names of persons and phone numbers in the form of a binary file.

A Python program to create a phone book with names and phone numbers.

```
with open ("phonebook.dat", "wb") as f:
    n = int(input('How many entries? '))
    for i in range(n):
        name = input('Enter name: ')
        phone=input('Enter phone no: ')
        name = name.encode()
        phone=phone.encode()
        f.write(name+phone)
```

Output:

```
How many entries? 3
Enter name: Tree
Enter phone no: 12345
Enter name: Apple
Enter phone no: 09876
Enter name: Newton
Enter phone no: 56789
```

A Python program to use mmap and performing various operations

```
import mmap, sys #display a menu
print("1 to display all the entries")
print('2 to display phone number')
print('3 to modify an entry')
print('4 exit')
ch=input('Your choice: ')
if ch=='4':
    sys.exit()
with open("phonebook.dat", "r+b") as f:
    mm=mmap.mmap(f.fileno(), 0)
    if(ch == '1'):
        print(mm.read().decode())
    if(ch == '2'):
        name=input('Enter name: ')
        n=mm.find(name.encode())
        n1=n+len(name)
        ph = mm[n1: n1+10]
        print('Phone no: ', ph.decode())
    if(ch=='3'):
        name=input('Enter name: ')
        n=mm.find(name.encode())
        n1 = n+len(name)
        ph1=input('Enter new phone number:')
        mm[n1: n1+10]=ph1.encode()
    mm.close()
```

Output:

```
1 to display all the entries
2 to display phone number
3 to modify an entry
4 exit
Your choice: 2
Enter name: Newton
Phone no:  56789
```