CHAPTER 17

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 'certificates file'. This file contains several certificates or pages and each page contains some data. From this example, we 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. Figure 17.1 shows the file in our daily life and the file stored in the computer:

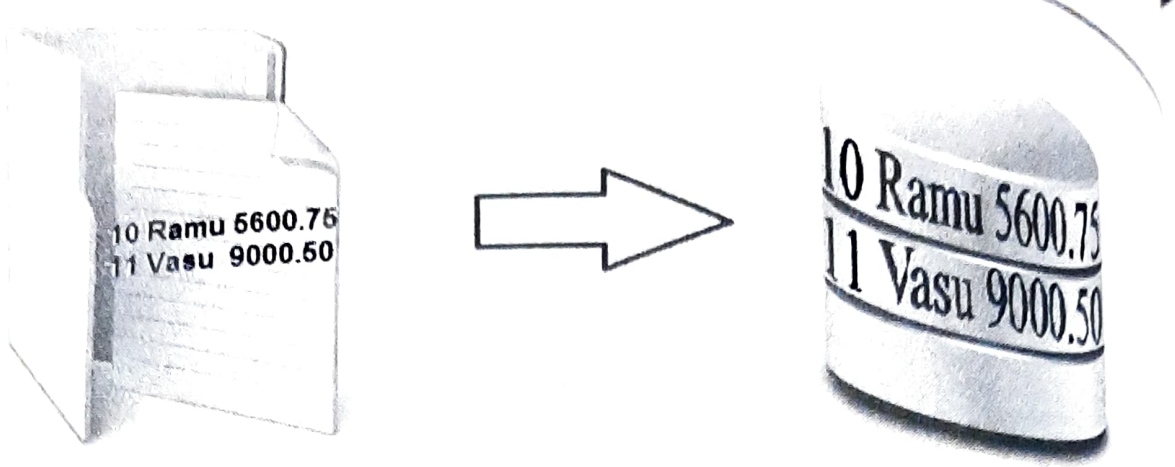

Figure 17.1: A file in our daily life is similar to a file in the computer

There are four important advantages of storing data in a file:

- When the data is stored in a file, it is stored permanently. This means that even though the computer is switched off, the data is not removed from the memory but the file is stored on hard disk or CD. This file data can be utilized later, whenever required.
- It is possible to update the file data. For example, we can add new data to the existing file, delete unnecessary data from the file and modify the available data of the file. This makes the file more useful.
- Once the data is stored in a file, the same data can be shared by various programs. For example, once employee data is stored in a file, it can be used in a program to calculate employees' net salaries or in another program to calculate income tax payable by the employees.
- Files are highly useful to store huge amount of data. For example, voters' list, census data.

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 employee name "Ganesh", it will be stored as 6 characters and the employee salary 8900 will be 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

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 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 file name to represent the employee data. The file 'open mode' represents the purpose of opening the file. Table 17.1 specifies the file open modes and their meanings:

Table 17.1: The File Opening Modes

File open mode	Description
w	To write data into file. If any data is already present in the file, it would be deleted and the present data will be stored.
r	To read data from the file. The file pointer is positioned at the beginning of the file.
a	To append data to the file. Appending means adding at the end of existing data. The file pointer is placed at the end of the file. If the file does not exist, it will create a new file for writing data.
w+	To write and read data of a file. The previous data in the file will be deleted.
r+	To read and write data into a file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.
a+	To append and read data of a file. The file pointer will be at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.

File open mode	Description
x	To open the file in exclusive creation mode. The file creation fails if the file already exists.

Table 17.1 represents file open modes for text files. If we attach 'b' for them represent modes for binary files. For example, wb, rb, ab, w+b, r+b, a+b are the modes binary files.

A buffer represents a temporary block of memory. 'buffering' is an optional integer to set the size of the buffer for the file. In the binary mode, we can pass 0 as buffer integer to inform not to use any buffering. In text mode, we can use 1 for buffer 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 which the data is read or written. If we do not mention any buffering integer, 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 the file. This is called 'file handler' or 'file object'. As an example, to open a file for writing data into it, we can written the open() function as:

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

Here, 'f' 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 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 'f' is closed. It means 'f' is deleted from 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 Program 1, 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'.

Program 1: A Python program to create a text file to store individual characters.

```
# creating a file to store characters
# open the file for writing data
f = open('myfile.txt', 'w')

# enter characters from keyboard
str = input('Enter text: ')

# write the string into file
f.write(str)

# closing the file
f.close()
```

Output:

```
C:\>python create.py
Enter text: This is my first line.
```

In Program 1, 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 returns 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.

Program 2: A Python program to read characters from a text file.

```
# reading characters from file
# open the file for reading data
f = open('myfile.txt', 'r')

# read all characters from file
str = f.read()

# display them on the screen
```

```
print(str)

# closing the file
f.close()
```

Output:

```
C:\>python read.py
This is my first line.
```

If in Program 2, 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:

```
This
```

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 '@' symbol, we can write while loop as:

```
while str != '@':
    # write the string into file
    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 "\n" character at the end of each string.

Program

Program 3: A Python program to store a group of strings into a text file.

```
# creating a file with strings
# open the file for writing data
f = open('myfile.txt', 'w')

# enter strings from keyboard
print('Enter text (@ at end): ')
while str != '@':
    str = input()   # accept string into str
    # write the string into file
    if(str != '@'):
        f.write(str+"\n")

# closing the file
f.close()
```

Output:

```
C:\>python create1.py
This is my file line one.
This is line two.
@
```

Now, to read the strings from the "myfile.txt", 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.']
```

Program

Program 4: A Python program to read all the strings from the text file and display them.

```
# reading strings from a file
# open the file for reading data
f = open('myfile.txt', 'r')

# read strings from the file
print('The file contents are: ')
str = f.read()
print(str)

# closing the file
f.close()
```

Output:

```
C:\>python read1.py
The file contents are:
This is my file line one.
This is line two.
```

We will plan another program to append data to the existing "myfile.txt" and then to display the data. For this purpose, we should open the file in 'append and read' mode as:

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

Once the file is opened in 'a+' mode, as usual we can use write() method to append the strings to the file. After writing the data, without closing the file, we can read strings from

the file. But first of all, we should place the file handler to the beginning of the file u... seek() method as:

```
f.seek(offset, fromwhere)
```

Here, 'offset' represents how many bytes to move. 'fromwhere' represents from w... position to move. For example, 'fromwhere' is 0 represents from beginning of fi... represents from the current position in the file and 2 represents from the ending of fi...

```
f.seek(10, 0)
```

This will position the file handler at 10th byte from the beginning of the file. So, a... reading operation will read data from 10th byte onwards.

```
f.seek(0, 0)
```

This will position the file handler at the 0th byte from the beginning of the file. It mea... any reading operation will read the data quite from the beginning of the file.

Program

Program 5: A Python program to append data to an existing file and then displaying ... entire file.

```
# appending and then reading strings
# open the file for reading data
f = open('myfile.txt', 'a+')

print('Enter text to append(@ at end): ')
while str != '@':
    str = input()   # accept string into str

    # write the string into file
    if(str != '@'):
        f.write(str+"\n")

# put the file pointer to the beginning of file
f.seek(0,0)

# read strings from the file
print('The file contents are: ')
str = f.read()
print(str)

# closing the file
f.close()
```

Output:

```
C:\>python append.py
Enter text to append(@ at end):
This line is added.
@
The file contents are:
This is my file line one.
This is line two.
This line is added.
```

Knowing Whether a File Exists or Not

The operating system (os) module has a sub module by the name 'path' that contains a method isfile(). This method can be used to know whether a file that we are opening really exists 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):    # if file exists,
    f = open(fname, 'r')     # open it
else:
    print(fname+' does not exist')
    sys.exit()   # terminate the program
```

In Program 6, 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.

Program

Program 6: A Python program to know whether a file exists or not.

```
# checking if file exists and then reading data
import os, sys

# open the file for reading data
fname = input('Enter filename: ')

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

# read strings from the file
print('The file contents are: ')
str = f.read()
print(str)

# closing the file
f.close()
```

Output:

```
C:\>python check.py
Enter filename: myfile.txt
The file contents are:
This is my file line one.
This is line two.
This line is added.

C:\>python check.py
Enter filename: yourfile.txt
yourfile.txt does not exist
```

Please observe that while running Program 6, we can enter the same program name 'check.py' to display the source code of the above program.

We can write another program to count the number of lines, words and characters in a text file. The logic is simple. We can use a for loop to read line by line from the file as:

```
for line in f:   # repeat the loop for all lines
    print(line)   # display one line at a time
```

Each line may contain some words. To find out the words, we have to split the line wherever there are spaces. This is done using line.split() method. But this method returns a list with the words.

```
for line in f:
    print(line)
    words = line.split()    # words is a list that contains all words in
                            # a line
    print(words)   # display the list
```

Now, to find out the number of words in a line, we can simply use len() function on 'words' list as len(words). Similarly, to find the number of characters in a line, we can use len(line). So, the total logic will be:

```
for line in f:
    print(line)
    words = line.split()
    print(words)
    cl +=1              # for each line add 1 to cl
    cw += len(words) # in each line add number of words to cw
    cc += len(line)  # in each line, add number of chars to cc
```

Program

Program 7: A Python program to count number of lines, words and characters in a text file.

```
# counting number of lines, words and characters in a file
import os, sys

# open the file for reading data
fname = input('Enter filename: ')

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

# initialize the counters to 0
cl= cw= cc = 0

# read line by line from the file
for line in f:
    words = line.split()
    cl +=1
    cw += len(words)
    cc += len(line)

print('No. of lines: ', cl)
print('No. of words: ', cw)
print('No. of characters: ', cc)
```

```
# close the file
f.close()
```

Output:

```
C:\>python count.py
Enter filename: myfile.txt
No. of lines:  3
No. of words:  13
No. of characters:  61
```

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 purpose, we can use 'rb' mode. Here, 'b' is attached to 'r' 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. This is shown in Program 8.

Program

Program 8: A Python program to copy an image file into another file.

```
# copying an image into a file
# open the files in binary mode
f1 = open('cat.jpg', 'rb')
f2 = open('new.jpg', 'wb')

# read bytes from f1 and write into f2
bytes = f1.read()
f2.write(bytes)

# close the files
f1.close()
f2.close()
```

Output:

```
C:\>python copy.py
C:\>
```

When the above program is run, it will copy the image file 'cat.jpg' into another file 'new.jpg'. In this program, our assumption is that the file 'cat.jpg' is already available in the current directory. Current directory is the directory where our program is running. Suppose, if the cat.jpg 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:\\rnr\\cat.jpg', 'rb')
```

The preceding statement tells that the cat.jpg file is available in 'rnr' sub directory of drive. Please remember, the double backslashes in the path above are interpreted single backslashes only.

The with Statement

The 'with' statement can be used while opening a file. The advantage of with statement that it will take care of closing a file which is opened by it. Hence, we need not close file explicitly. In case of an exception also, 'with' statement will close the file before exception is handled. The format of using 'with' is:

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

Program

Program 9: A Python program to use 'with' to open a file and write some strings into file.

```
# with statement to open a file
with open('sample.txt', 'w') as f:
    f.write('I am a learner\n')
    f.write('Python is attractive\n')
```

Output:

```
C:\>python with1.py
C:\>
```

In Program 9, we opened the file 'sample.txt' and stored two lines of text. This file is also opened using 'with' statement in Program 10 and the contents are displayed.

Program

Program 10: A Python program to use 'with' to open a file and read data from it.

```
# using with statement to open a file
with open('sample.txt', 'r') as f:
    for line in f:
        print(line)
```

Output:

```
C:\>python with2.py
I am a learner
Python is attractive
```

Pickle in Python

So far, we wrote some programs where we stored only text into the files and retrieved same text from the files. These text files are useful when we do not want to perform calculations on the data. What happens if we want to store some structured data in 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

structured and got 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 *de-searialization*. 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, byte stream represents flow of bytes.

We will understand pickling and unpickling more clearly with the help of an example. First of all let's create Emp class that contains employee identification number, name and salary. This is shown in Program 11.

Program

Program 11: A Python program to create an Emp class with employee details as instance variables.

```
# Emp class - save this as Emp.py
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:

```
C:\>python Emp.py
C:\>
```

Our intention is to pickle Emp class objects. For this purpose, we have to import 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 'e' is the object of Emp class. Since Emp class belongs to 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)
```

This logic is shown in Figure 17.2 and expressed in Program 12.

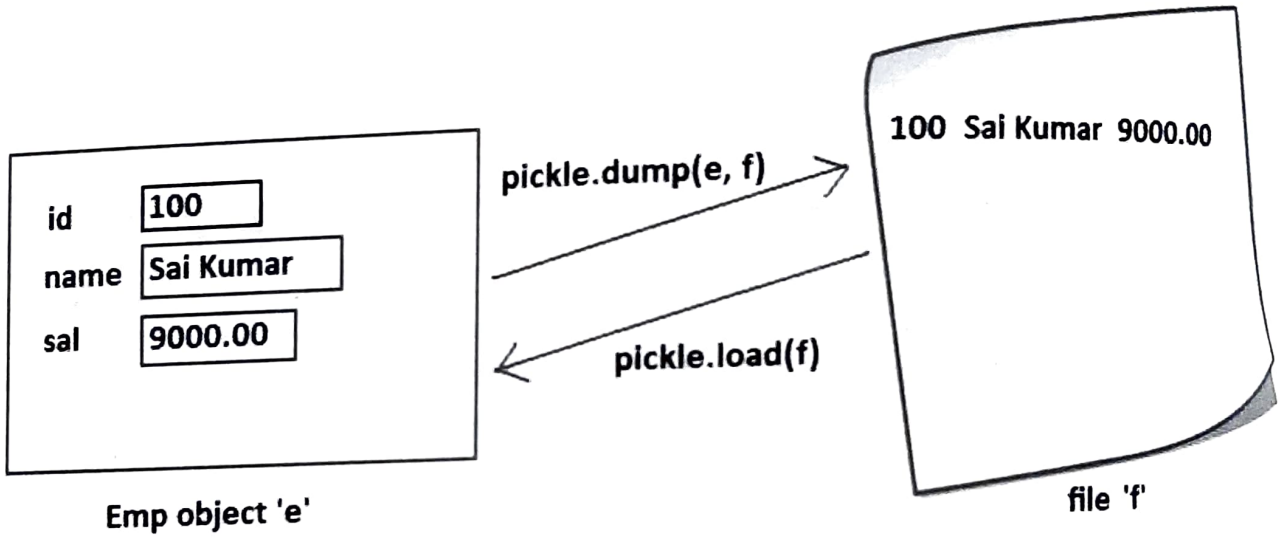

Figure 17.2: Pickling and unpickling a class object

Program

Program 12: A Python program to pickle Emp class objects.

```
# pickle - store Emp class objects into emp.dat file
import Emp, pickle

# open emp.dat file as a binary file for writing
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: '))

    # create Emp class object
    e = Emp.Emp(id, name, sal)

    # store the object e into the file f
```

```
    pickle.dump(e, f)

# close the file
f.close()
```

Output:

```
C:\>python pick.py
How many employees? 3
Enter id: 100
Enter name: Sai Kumar
Enter salary: 9000.00
Enter id: 101
Enter name: Ravi Teja
Enter salary: 8900.50
Enter id: 102
Enter name: Harsh Deep
Enter salary: 12000.75
```

In the previous program, we stored 3 Emp class objects into 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 'f' 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 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 Program 13.

Program 13: A Python program to unpickel Emp class objects.

```
# unpickle or object de-serialization
import Emp, pickle

# open the file to read objects
f = open('emp.dat', 'rb')

print('Employees details: ')
while True:
    try:
        # read object from file f
        obj = pickle.load(f)
        # display the contents of employee obj
        obj.display()

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

```
# close the file
f.close()
```

Output:

```
C:\>python unpick.py
Employees details:
  100  Sai Kumar          9000.00
  101  Ravi Teja          8900.50
  102  Harsh Deep        12000.75
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 are reading or writing operations on a binary file, a file pointer moves internally 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. Now it is possible to continue reading from the 11th 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, 'f' represents file handler or file object. 'n' is an integer that represents the 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 the position to move. For example, 'fromwhere' can be 0, 1 or 2. Here, 0 represents the beginning of the file, 1 represents from 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)    # same as f.seek(10, 0)
```

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

```
f.seek(-10, 2)
```

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

We will take an example to understand how these methods work. In case of the following code snippet where we open a binary file 'line.txt' in read mode, observe 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. Now, the string is stored in the file as shown in the Figure 17.3:

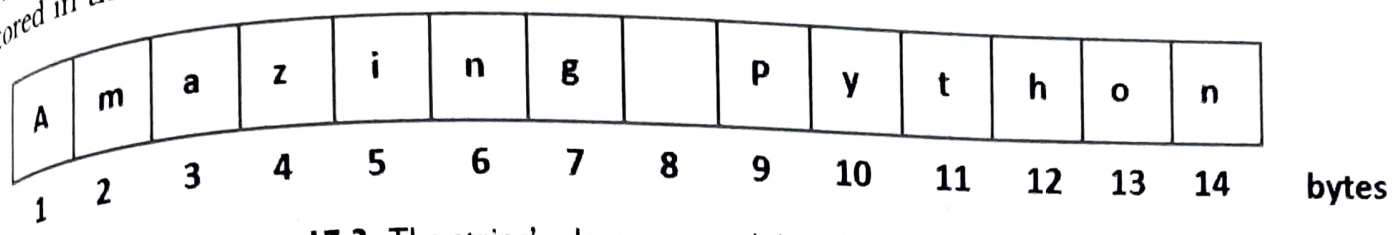

Figure 17.3: The string's characters and their byte positions in the file.

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, file pointer will be at 'z'. 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 tell() method as:

```
print(f.tell())
```

This will display 5. Now, to move the file pointer to 5th 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 10th 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 in 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 991st 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 possible by moving the file pointer to any location in the file and then performing read 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)   # store str
    f.write('Hello')   # store '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 encode() method, as: str.encode().

The encode() method represents the string in byte format so that it can be stored into a binary file. Similarly, when we read a string from a binary file, it is advisable to convert 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 length, then the remaining characters in the record will be filled with spaces. For example, if the user stores the name 'Delhi', since 'Delhi' has only 5 characters, another 15 spaces will be attached to 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)   # find length of city name. suppose it is 5
city = city + (20-ln)*' '          # add 15 spaces at the end of city name
```

Program 14: A Python program to create a binary file and store a few records.

```
# create cities.bin file with cities names
# take the record size as 20 bytes
reclen = 20

# open the file in wb mode as binary file
with open("cities.bin", "wb") as f:
    # write data into the file
    n = int(input('How many entries? '))

    for i in range(n):
        city = input('Enter city name: ')
        # find the length of city
        ln = len(city)
        # increase the city name to 20 chars
        # by adding remaining spaces
        city = city + (reclen-ln)*' '
        # convert city name into byte string
        city = city.encode()
        # write the city name into the file
        f.write(city)
```

Output:

```
C:\>python create.py
How many entries? 4
Enter city name: Delhi
Enter city name: Hyderabad
Enter city name: Pune
Enter city name: Ahmedabad
```

Now, cities.bin file is created with 4 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 Program 15. In this program, we accept the record number and display that record. Suppose, we want to display 3rd record, first we should put the file pointer at the end of 2nd record. This is done by seek() method, as:

```
reclen = 20   # this is record length
f.seek(reclen * 2)
```

Once this is done, we can read the next 20 bytes using read() method that gives the 3rd record.

Program 15: A Python program to randomly access a record from a binary file.

```
# reading city name based on record number
# take record length as 20 characters
reclen = 20

# open the file in binary mode for reading
with open('cities.bin', 'rb') as f:
    n = int(input('Enter record number: '))
    # move file pointer to the end of n-1 th record
    f.seek(reclen * (n-1))
```

```
# get the n th record with 20 chars
str = f.read(reclen)
# convert the byte string into ordinary string
print(str.decode())
```

Output:

```
C:\>python read.py
Enter record number: 3
Pune
```

The next question is how to search for a particular record in the binary file. Suppose, we want to know whether a particular city name is existing in the file or not, we should search record by record using the seek() method.

```
position = 0
f.seek(position)   # initially place the file pointer at 0th byte
position+=20   # every time increase it by 20 bytes
```

The preceding code should be written in a for loop that looks like this:

```
for i in range(n):
```

Here, 'i' indicates record number that changes from 0 to n-1 where n is the total number of records. Then, how to find the number of records in the file? For this purpose, first we should find the file size in bytes. This is given by getsize() method of 'path' sub module of 'os' module.

```
size = os.path.getsize('cities.bin')
```

By dividing the size of the file by the record length, we can get the number of records in the file as:

```
n = int(size/reclen)   # reclen is 20.
```

Program

Program 16: A Python program to search for city name in the file and display the record number that contains the city name.

```
# searching the city name in the file
import os

# take record length as 20 characters
reclen = 20

# find size of the file
size = os.path.getsize('cities.bin')
print('Size of file = {} bytes'.format(size))

# find number of records in the file
n = int(size/reclen)
print('No. of records = {}'. format(n))

# open the file in binary mode for reading
with open('cities.bin', 'rb') as f:
    name = input('Enter city name: ')
    # convert name into binary string
    name = name.encode()
```

```
    # position represents the position of file pointer
    position = 0

    # found becomes True if city is found in the file
    found = False

# repeat for n records in the file
for i in range(n):
        # place the file pointer at position
        f.seek(position)
        # read 20 characters
        str = f.read(20)
        # if found
        if name in str:
            print('Found at record no: ', (i+1))
            found=True
        # go to the next record
        position+=reclen

if not found :
    print('City not found')
```

Output:

```
C:\>python read1.py
Size of file = 80 bytes
No. of records = 4
Enter city name: Hyderabad
Found at record no: 2
```

To update the city name to a new name, first we should know the record number where the city name is found. This logic is presented already in Program 16. Suppose, the city name is found in record number 2 and after reading that record, the file pointer will be at the end of the record. Hence, we should get the file pointer back to the beginning of the record as:

```
f.seek(-20, 1)   # go back 20 bytes from current position
```

Then write the new city name in that place as:

```
f.write(newname)
```

Program 17: A Python program to update or modify a record in a binary file.

```
# updating the city name in the file
import os

# take record length as 20 characters
reclen = 20

# find size of the file
size = os.path.getsize('cities.bin')
print('Size of file = {} bytes'.format(size))

# find number of records in the file
n = int(size/reclen)
print('No. of records = {}'. format(n))
```

```
# open the file in binary mode for reading
with open('cities.bin', 'r+b') as f:
    name = input('Enter city name: ')
    # convert name into binary string
    name = name.encode()

    newname = input('Enter new name: ')
    # find length of newname
    ln = len(newname)
    # add spaces to make its length to be 20
    newname = newname+ (20-ln)*' '
    # convert newname into binary string
    newname = newname.encode()

    # position represents the position of file pointer
    position = 0

    # found becomes True if city is found in the file
    found = False

    # repeat for n records in the file
    for i in range(n):
        # place the file pointer at position
        f.seek(position)
        # read 20 characters
        str = f.read(20)
        # if found
        if name in str:
            print('Updated record no: ', (i+1))
            found=True
            # go back 20 bytes from current position
            f.seek(-20, 1)
            # update the record
            f.write(newname)

        # go to the next record
        position+=reclen

    if not found :
        print('City not found')
```

Output:

```
C:\>python update.py
Size of file = 80 bytes
No. of records = 4
Enter city name: Hyderabad
Enter new name: Bangalore
Updated record no: 2
```

The next step is to develop logic to delete a record from the binary file. There is no way to delete some part of the data from a binary file. The only way to do this is to follow the steps shown here:

1. Suppose we want to delete a record that contains city name 'Bangalore' from cities.bin file. Copy all the records of this file into a temporary file, say file2.bin except the record which matches the city name 'Bangalore'. Now file2.bin contains all the records except the 'Bangalore' record.

2. The next step is to delete the original cities.bin file. This is done by using the remove() method of 'os' module as: os.remove("filename")

3. Rename the file2.bin file as cities.bin file. This is done by using the rename() method of 'os' module as: os.rename("oldfile", "newfile"). Now, this new cities.bin file contains all the records except the one which we deleted.

This logic is presented in Program 18.

Program 18: A Python program to delete a specific record from the binary file.

```
# deleting a record from the file
import os

# take record length as 20 characters
reclen = 20

# find size of the file
size = os.path.getsize('cities.bin')

# find number of records in the file
n = int(size/reclen)

# open the cities.bin for reading
f1 = open('cities.bin', 'rb')

# open file2.bin for writing
f2 = open('file2.bin', 'wb')

# accept city name from keyboard
city = input('Enter city name to delete: ')

# add spaces so that it will have 20 characters length
ln = len(city)
city = city+(reclen-ln)*' '

# convert city name to binary string
city = city.encode()

# repeat for all the n records
for i in range(n):
    # read one record from f1 file
    str = f1.read(reclen)
    # if it is not the city name, store into f2 file
    if(str != city):
        f2.write(str)

print('Record deleted...')

# close the files
f1.close()
f2.close()
```

```
# delete the cities.bin file
os.remove("cities.bin")

# rename file2.bin as cities.bin
os.rename("file2.bin", "cities.bin")
```

Output:

```
C:\>python del.py
Enter city name to delete: Bangalore
Record deleted...
```

Random Accessing of Binary Files using mmap

mmap – 'memory mapped file' is a module in Python that is useful to map or link a binary file and manipulate the data of the file as we do with the strings. It means, when a binary file is created with some data, that data is viewed as strings and can be 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 mmap() method. The first argument is 'f.fileno()' .This indicates that file is a handle to the file object 'f'. This 'f' represents the actual binary file that is to be 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 to 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())     # displays entire file data
print(mm.readline())   # displays the first line of the file
```

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

```
print(mm[5:])   # display from 5th byte till the end
print(mm[5:10])   # display from 5th to 9th bytes
```

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

```
mm[5:10] = str   # replace from 5th to 9th characters by string 'str'
```

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

```
n = mm.find(name)   # return the position of name in the file
```

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

```
mm.seek(10, 0)   # position the file pointer to 10th byte from
# beginning of file
```

Let's remember that the memory mapping is done only for the binary files and not for text files. So, first of all let's create a binary file where we want to store some data. 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()    # convert name from string to binary string
```

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()    # convert bytes into a string
```

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.

Program 19: A Python program to create phone book with names and phone numbers.

```
# create phonebook.dat file
# open the file in wb mode as binary file
with open("phonebook.dat", "wb") as f:
    # write data into the file
    n = int(input('How many entries? '))

    for i in range(n):
        name = input('Enter name: ')
        phone = input('Enter phone no: ')
        # convert name and phone from strings to bytes
        name = name.encode()
        phone = phone.encode()
        # write the data into the file
        f.write(name+phone)
```

Output:

```
C:\>python demo.py
How many entries? 3
Enter name: Viswajith
Enter phone no: 9880125567
Enter name: Manoj
Enter phone no: 7700177001
Enter name: Dheerendra
Enter phone no: 8989189891
```

In Program 19, we have created the phonebook.dat file with 3 persons' names and phone numbers. We can go to the current directory and view the file contents in Notepad and it will look something like this:

```
Viswajith9880125567Manoj7700177001Dheerendra8989189891
```

The next step is to perform some operations like displaying all the entries of the file, display required information from the file or modify the contents of the file. This is done in Program 20.

Program

Program 20: A Python program to use mmap and performing various operations on binary file.

```
# using mmap on a binary file.
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:

    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)

    # display the entire file
    if(ch == '1'):
        print(mm.read().decode())

    # display phone number
    if(ch == '2'):
        name = input('Enter name: ')
        # find position of name
        n = mm.find(name.encode())
        # go to end of name
        n1 = n+len(name)
        # display the next 10 bytes
        ph = mm[n1: n1+10]
        print('Phone no: ', ph.decode())

    # modify phone number
    if(ch == '3'):
        name = input('Enter name: ')
        # find position of name
        n = mm.find(name.encode())
        # go to end of name
        n1 = n+len(name)
        # enter new phone number
        ph1 = input('Enter new phone number: ')
        # the old phone number is 10 bytes after n1
        mm[n1: n1+10] = ph1.encode()

    # close the map
    mm.close()
```

Output:

```
C:\>python demo1.py
1 to display all the entries
2 to display phone number
3 to modify an entry
4 exit
Your choice: 2
```

```
Enter name: Manoj
Phone no: 7700177001
C:\> python demo1.py
1 to display all the entries
2 to display phone number
3 to modify an entry
4 exit
Your choice: 3
Enter name: Dheerendra
Enter new phone number: 9999911111
```

Zipping and Unzipping Files

We know that some softwares like 'winzip' provide zipping and unzipping of file data. In zipping the file contents, following two things could happen:

- The file contents are compressed and hence the size will be reduced.
- The format of data will be changed making it unreadable.

While zipping a file content, a zipping algorithm (logic) is used in such a way that the algorithm first finds out which bit pattern is most often repeated in the original file and replaces that bit pattern with a 0. Then the algorithm searches for the next bit pattern which is most often repeated in the input file. In its place, a 1 is substituted. The third repeated bit pattern will be replaced by 10, the fourth by 11, the fifth by 100, and so on. In this way, the original bit patterns are replaced by lesser number of bits. This file with lesser number of bits is called 'zipped file' or 'compressed file'.

To get back the original data from the zipped file, we can follow a reverse algorithm, which substitutes the original bit pattern wherever particular bits are found. This is shown in Figure 17.4:

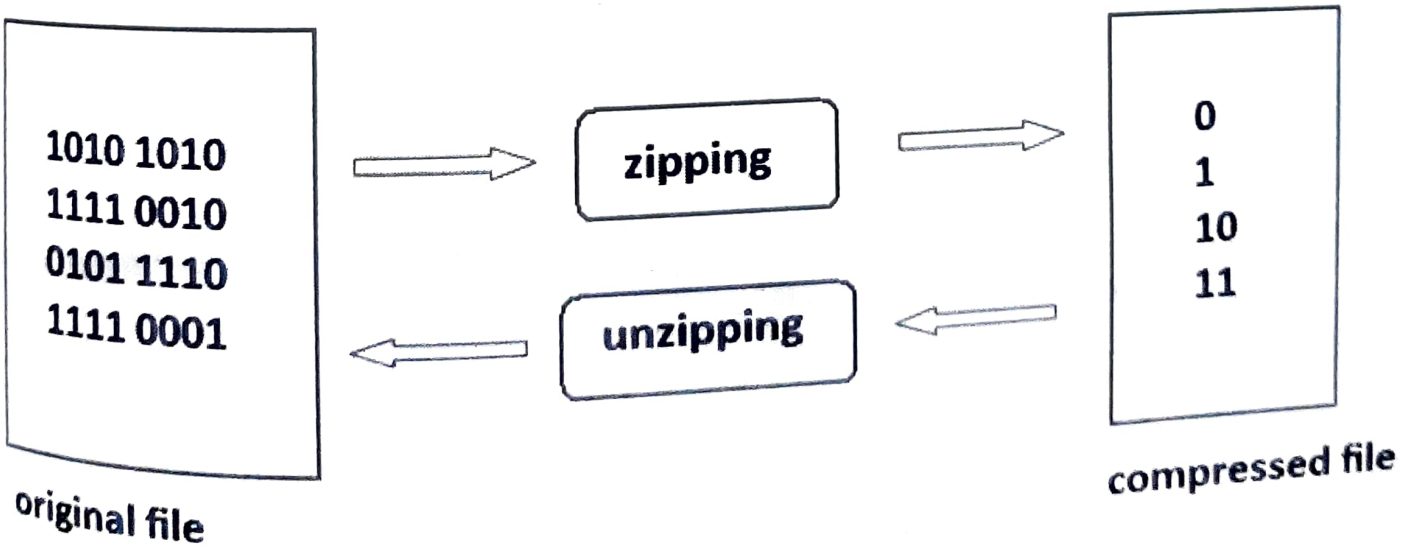

Figure 17.4: Zipping and unzipping a file

In Python, the module *zipfile* contains ZipFile class that helps us to zip or unzip contents. For example, to zip the files, we should first pass the zip file name in mode with an attribute ZIP_DEFLATED to the ZipFile class object as:

```
f = ZipFile('test.zip', 'w', ZIP_DEFLATED)
```

Here, 'f' is the ZipFile class object to which test.zip file name is passed. This is the that is created finally. The next step is to add the filenames that are to be zipped write() method as:

```
f.write('file1.txt')
f.write('file2.txt')
```

Here, we are writing two files: file1.txt and file2.txt into the object 'f'. Hence, the files are compressed and stored into test.zip file. This is shown in Program 21.

Program

Program 21: A Python program to compress the contents of files

```
# zipping the contents of files
from zipfile import *

# create zip file
f = ZipFile('test.zip', 'w', ZIP_DEFLATED)

# add some files. these are zipped
f.write('file1.txt')
f.write('file2.txt')
f.write('file3.txt')

# close the zip file
print('test.zip file created...')
f.close()
```

Output:

```
C:\>python compress.py
test.zip file created...
```

In Program 21, we assumed that the three files: file1.txt, file2.txt and file3.txt are available in the current directory where this program is run. To unzip the content compressed files and get back their original contents, we can use ZipFile class read mode as:

```
z = ZipFile('test.zip', 'r')
```

Here, test.zip is the filename that contains the compressed files. To extract all from the zip file object 'z', we can use the extractall() method as:

```
z.extractall()
```

This will extract all the files in uncompressed format into the current directory. If we want to extract them to another directory, we can mention the directory path in the extractall() method as:

```
z.extractall('f:\\python\core\sub')
```

Program 22: A Python program to unzip the contents of the files that are available in a zip file.

```
# to view contents of zipped files
from zipfile import *

# open the zip file
z = ZipFile('test.zip', 'r')

# extract all the file names which are in the zip file
z.extractall()
```

Output:

```
C:\>python uncompress.py
Contents of file1.txt
.....
.....
Contents of file2.txt
.....
.....
Contents of file3.txt
.....
.....
```

Working with Directories

The os (operating system) module represents operating system dependent functionality. This module is useful to perform some simple operations on directories. Let's assume we are working with the following directory path:

```
F:\py\enum
```

And we are currently in the directory F:\py where we are running our programs. If we want to know the currently working directory, we can use getcwd() method of 'os' module as shown in Program 23.

Program 23: A Python program to know the currently working directory.

```
import os
# get current working directory
current = os.getcwd()
print('Current directory= ', current)
```

Output:

```
F:\py>python os.py
Current directory= F:\py
```

If we want to create our own directory in the present directory, we can use the
method. This method can be used as:

```
os.mkdir('mysub')
```

This will create mysub in the present directory. Suppose, we write the
statement:

```
os.mkdir('mysub/mysub2')
```

This will create mysub2 in the mysub directory. Consider Program 24.

Program

Program 24: A Python program to create a sub directory and then sub-sub dire
the current directory.

```
import os
# create a sub directory by the name mysub
os.mkdir('mysub')

# create a sub-sub directory by the name mysub2
os.mkdir('mysub/mysub2')
```

Output:

```
F:\py>python os.py
```

After executing Program 24, we can observe mysub directory in our current di
(F:\py) and inside the mysub directory, there would be another directory by the
mysub2. This can be verified by visiting the directory structure in our compute
problem with mkdir() method is that it cannot create a sub directory unless the
directory exists. For example in Program 24, since mysub is existing, it is able to
mysub2 inside that. If mysub does not exist, then it cannot create mysub2. In thi
makedirs() is a useful method. This method recursively creates the sub director
example, consider the following statement:

```
os.makedirs('newsub/newsub2')
```

This creates first newsub directory if it does not exist, and then it will create n
inside it. Consider Program 25.

Program

Program 25: A Python program to use the makedirs() function to create sub and s
directories.

```
import os
# create sub and sub-sub directories
os.makedirs('newsub/newsub2')
```

Output:

```
F:\py>python os.py
```

We can change our current working directory to another existing directory. For this purpose, we can use chdir() method. For example, if we want to change our directory to newsub2 which is in newsub, we can use chdir() method as:

```
goto = os.chdir('newsub/newsub2')
```

Here, our assumption is that newsub is in the current directory and newsub contains newsub2. After executing the preceding statement, our current working directory will be newsub2. This is shown in Program 26.

Program 26: A Python program to change to another directory.

```
import os
# change to newsub2 directory
goto = os.chdir('newsub/newsub2')

# get current working directory
current = os.getcwd()
print('Current directory= ', current)
```

Output:

```
F:\py>python os.py
Current directory= F:\py\newsub\newsub2
```

To remove a directory, we can use rmdir() method of 'os' module. Let's suppose, our current directory is F:\py. In this, we have newsub directory and inside newsub, we have newsub2 directory. To remove newsub directory, we can write:

```
os.rmdir('newsub')
```

The rmdir() method will remove newsub since it is found in the current directory. Suppose, we write the following statement:

```
os.rmdir('newsub2')
```

This cannot remove newsub2 as it is not in the current directory (i.e. F:\py). newsub2 is inside newsub that is in the current directory. In this case, path should be given properly as:

```
os.rmdir('newsub/newsub2')
```

This will remove the newsub2 directory. Consider Program 27.

Program

Program 27: A Python program to remove a sub directory that is inside another directory.

```
import os
# to remove newsub2 directory
os.rmdir('newsub/newsub2')
```

Output:

```
F:\py>python os.py
```

There is another method removedirs() that recursively removes all the directories. For example, we write the following statement:

```
os.removedirs('mysub/mysub2/mysub3')
```

This will remove mysub3 first, then mysub2 and then mysub. In this way it can remove several sub-sub directories in a single stretch. In Program 28, our assumption is that we have mysub directory inside the current directory (i.e. F:\py). Also, mysub contains another sub directory mysub2 that again contains mysub3. We want to remove all the three sub directories using the removedirs() method.

Program

Program 28: A Python program to remove a group of directories in the path.

```
import os
# to remove mysub3, mysub2 and then mysub.
os.removedirs('mysub/mysub2/mysub3')
```

Output:

```
F:\py>python os.py
```

To give a new name to an existing directory, we can use rename() method in the following format:

```
os.rename('oldname', 'newname')
```

Here, 'oldname' of the directory is changed as 'newname'. In Program 29, we are renaming 'enum' as 'newenum'. The directory 'enum' is available in the current directory.

Program

Progarm 29: A Python program to rename a directory.

```
import os
# to rename enum as newenum
os.rename('enum', 'newenum')
```

Output:

```
F:\py>python os.py
```

We can go to our F:\py directory and check that the enum directory became newenum.

Sometimes, we want to know the contents of a directory. A directory may contain other directories or files. To display all the contents of a directory, we are equipped with walk() method in 'os' module. This method is used in the following format:

```
os.walk(path, topdown=True, onerror=None, followlinks=False)
```

This method returns an iterator object whose contents can be displayed using a for loop. This iterator object contains directory path, directory names and filenames found in a given directory path.

'path' represents the directory name. For current directory, we can use dot (.). Giving path is enough for walk() method.

If 'topdown' is True, the directory and its sub directories are traversed in top-down manner. If it is False, then the traversal will be in bottom-up manner.

'onerror' represents what to do when an error is detected. With 'onerror', we can specify a function to be executed if an error is encountered.

By default, walk() will not walk down into symbolic links that resolve to directories. We should set 'followlinks' to True to visit directories pointed to by symbolic links, on systems that support them.

In Program 30, we are going to display the contents of the current directory with the help of walk() method. Our assumption is that our current directory is F:\py that contains mysub directory and several files. mysub has another sub directory by the name mysub1 and a file. mysub1 has only one file. These directories and files can be identified from the output of this program.

Program

Program 30: A Python program to display all contents of the current directory.

```
import os
for dirpath, dirnames, filenames in os.walk('.'):
    print('Current path: ', dirpath)
    print('Directories: ', dirnames)
    print('Files: ', filenames)
    print()
```

Output:

```
F:\py>python os.py
Current path:  .
Directories:  ['mysub']
Files:  ['demo.py', 'demo1.py', 'emp.dat', 'Emp.py', 'files.py',
'new.jpg', 'os.py', 'os1.txt', 'output.txt', 'phonebook.dat', 'random.py',
'test.py', 'update.py']
```

```
Current path:  .\mysub
Directories:  ['mysub1']
Files:  ['myfile.txt']

Current path:  .\mysub\mysub1
Directories:  []
Files:  ['myfile1.txt']
```

In Program 30, we used os.walk('.') to display the contents of the current directory. If we want to display the contents of only the mysub directory which is in current directory, we can use walk() method as:

```
os.walk('mysub')
```

Running Other Programs from Python Program

The 'os' module has the system() method that is useful to run an executable program from our Python program. This method is similar to system() function of C language. It is used as system('string') where 'string' represents any command or executable file name. See the following examples:

```
os.system('dir')   # display directory contents on DOS operating system
os.system('python demo.py')   # run demo.py program
```

In Program 31, we are going to display the file names in the current directory that have .py extension. That means we are not displaying all files in the directory but filtering only python program files. This can be done using DOS operating system command *.py. We should remember that '*' is called wild card character that represents 'all'. *.py indicates all files that have .py extension name.

Program

Program 31: A Python program to display Python program files available in the current directory.

```
import os
# execute dir command of DOS operating system
os.system('dir *.py')
```

Output:

```
F:\py>python os.py
Volume in drive F is New Volume
 Volume Serial Number is 8683-5D92
 Directory of F:\py
04/09/2016  09:09 PM               483 demo.py
04/09/2016  09:52 PM             1,203 demo1.py
04/08/2016  11:20 PM               260 Emp.py
04/14/2016  09:31 PM               482 files.py
04/14/2016  11:27 PM                85 os.py
```

```
04/10/2016  08:26 PM                 82 test.py
04/10/2016  09:02 PM              1,513 update.py
               8 File(s)          4,664 bytes
               0 Dir(s)  136,625,213,440 bytes free
```

oints to Remember

- The data stored on a secondary storage media like hard disk or CD is called a file.
- Once the data is stored in a file, the same data can be shared by several programs.
- There are two types of files supported by Python: text files and binary files.
- Text files store data in the form of text or characters. Binary files store data in the form of bytes.
- The open() method opens a file for some operation like writing, reading or appending The close() method closes the file.
- The read() method reads the file content while write() method stores data into file.
- For binary files, we have to add 'b' at the end of file open mode. For example, 'w represents write mode for a binary file.
- Binary files are useful to handle text files, image files or audio or video files.
- We need not close the file if we open the file using with statement.
- Pickle or serialization is the concept of storing objects into a binary file. Unpickle de-serialization is the concept of retrieving objects from a binary file.
- In pickling, we use dump() method and in unpickling we use load() method.
- In a binary file, to move the file pointer to any position, we can use seek() method.
- In a binary file, to know the position of the file pointer, we can use tell() method.
- The encode() method converts the string into bytes so that it can be written int binary file.
- A binary string can be converted into ordinary string using decode() method.
- Accessing the contents of a file randomly by moving the file pointer to any byte in file is called random accessing. seek() and tell() methods are used in ran accessing of a file.
- Random accessing is also possible using mmap (memory mapped file) module.
- Zipping and unzipping of files can be done using ZipFile class of zipfile module.

- The os (operating system) module is useful to perform several operations directories like finding the currently working directory, changing to a par directory, renaming or deleting directories and listing the contents of a directory
- The system() method of os module is useful to run commands or executable prog from our Python program.