5.6 WORKING WITH BINARY FILES

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

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

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

PICKLING & UNPICKLING

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

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

```
import pickle
```

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

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

Following sub-sections are going to make it clear.

5.6.1 Creating/Opening/Closing Binary Files

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

```
Dfile = open("stu.dat", "wb+")
```
← *Binary file opened in **write** mode with file handle as **Dfile***

Or

Notice 'b' is used with the file modes

```
File1 = open("stu.dat", "rb")
```
← *Binary file opened in **read mode** with file handle as **File1***

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

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

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

```
Dfile.close()
```

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

IMPORTANT

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

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

5.6.2 Writing onto a Binary File – **Pickling**

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

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

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

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

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

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

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

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

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

NOTE

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

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

Now consider some example programs given below.

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

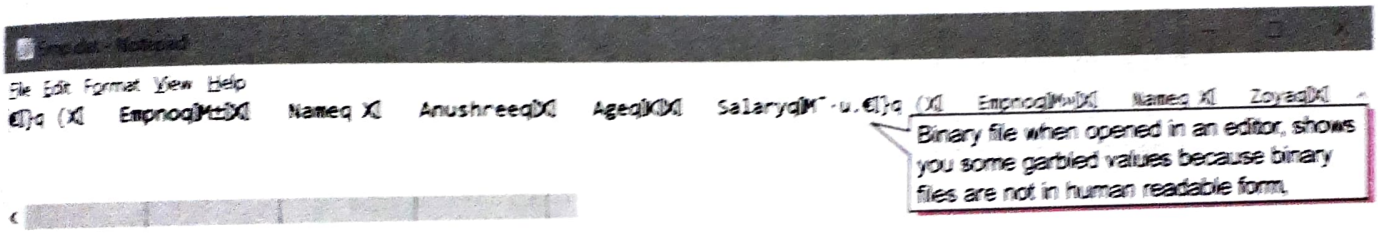
```
import pickle
# dictionary objects to be stored in the binary file
emp1 = {'Empno' : 1201, 'Name' : 'Anushree', 'Age' : 25, 'Salary' : 47000}
emp2 = {'Empno' : 1211, 'Name' : 'Zoya', 'Age' : 30, 'Salary' : 48000}
emp3 = {'Empno' : 1251, 'Name' : 'Simarjeet', 'Age' : 27, 'Salary' : 49000}
emp4 = {'Empno'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

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

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

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

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



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

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

Output

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

The sample run of above program is as shown here.

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

5.6.2A Appending Records in Binary Files

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

NOTE

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

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

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

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

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

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

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

5.6.3 Reading from a Binary File – UnPickling

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

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

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

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

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

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

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

IMPORTANT

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

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

(i) Use try and except blocks

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

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

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

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

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

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

(ii) Using with statement

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

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

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

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

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

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

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

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

This statement would unpickle the object being read from file

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

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

The output produced by the above program will be :

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

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

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

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

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

The output produced by above program will be :

```
File Stu.dat stores these records
{'Rollno': 11, 'Name': 'Sia', 'Marks': 83.5}
{'Rollno': 12, 'Name': 'Guneet', 'Marks': 80.5}
{'Rollno': 13, 'Name': 'James', 'Marks': 81.0}
{'Rollno': 14, 'Name': 'Ali', 'Marks': 80.5}
```

See, the appended record is also here

Program 5.13 Create a binary file namely myfile.info and write a string having two lines in it.

```
import pickle
string = "This is my first line. This is second line."
with open ("myfile.info", "wb") as fh :
    pickle.dump(string, fh)
print("File successfully created.")
```

*File **myfile.info** is opened in file handle **fh***

*All file processing statements inside the **with** block.*

The above program has created binary file namely *myfile.info* that stored the given string in the binary format.

Program 5.14 Write a program to read from the file myfile.info created in previous program and display the string until letter 'o' is encountered, *i.e.*, display all the text before the letter 'o'.

```
import pickle
st =""
with open("myfile.info", "rb") as fh :
    st = pickle.load(fh)
    lst = st.split('o')
    print( lst[0] )
```

*All file processing statements inside the **with** block.*

The output produced by above program is as shown below :

```
This is my first line. This is sec
```

See, the text before the letter 'o' is displayed

5.6.4 Searching in a File

There are multiple ways of searching for a value stored in a file. The simplest being the sequential search whereby you read the records from a file one by one and then look for the search key in the read record. We are covering the same method here. That is, in order to search for some value(s) in a file, you need to do the following :

(*Please note that objects read from the file are being referred to as records in this chapter.*)

1. Open the file in read mode.
2. Read the file contents record by record (*i.e.*, object by object).
3. In every read record, look for the desired search-key.
4. If found, process as desired.
5. If not found, read the next record and look for the desired search-key.
6. If search-key is not found in any of the records, report that no such value found in the file.

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

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

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

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

Searching for in the read record

This block will get executed when the search is successful.

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

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

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

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

5.6.5 Updating in a Binary File

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

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

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

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

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

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

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

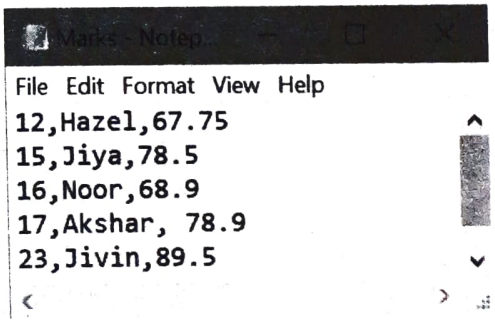
The tell( ) Function

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

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

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

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



Now consider the following code snippet :

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

To get current position of file pointer

3 bytes read

The output produced by the above code will be :

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

Initially file-pointer is at 0th byte

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

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

Now consider the following modified code :

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

The output produced by above code will be :

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

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

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

The seek( ) Function

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

The syntax for using this function is :

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

where

offset is a number specifying number-of-bytes

mode is a number 0 or 1 or 2 signifying

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

file object is the handle of open file.

NOTE

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

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

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

Consider the following examples :

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

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

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

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

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

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

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

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

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

NOTE

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

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

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

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

The output produced by above code is :

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

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

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

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

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

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

The output produced by above code is :

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

NOTE

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

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

5.6.5B Updating Record(s) in a File

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

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

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

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

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

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

(*ii*) Locate the record :

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

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

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

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

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

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

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

Following example program illustrates this process.

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

Note. Important statements have been highlighted.

```
import pickle
stu = {}                    # declare empty dictionary object to hold read records
found = False
# open binary file in read and write mode
fin = open('Stu.dat', 'rb+')
# read from the file
try:
    while True :
        rpos = fin.tell()       # store file-pointer position before reading the record
        stu = pickle.load(fin)

        if stu['Marks'] > 81 :
            stu['Marks'] += 2           # changes made in the record; 2 bonus marks added
            fin.seek(rpos)      # place the file-pointer at the exact location of the record
            pickle.dump(stu, fin) # now write the updated record on the exact location
            found = True
except EOFError:
    if found == False:
        print("Sorry, no matching record found.")
    else:
        print("Record(s) successfully updated.")
    fin.close()                         # close file
```

*It is important to open the file in read as well as write mode ; hence **rb+***

*Before reading any record, firstly store its beginning position – its **exact position***

Locating the desired record through search condition

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

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

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

It will show you a message :

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

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

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

Program

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

The output produced by above program is :

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

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

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

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

Program

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

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

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

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

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

IMPORTANT

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

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

(*i*) create a new file ;

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

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

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

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

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

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

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

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