

# B Sc Computer Science – VI Semester

## Elective Papers - CS6PET01: Python and LaTeX

### Module IV - Files and user I/O

User input, Reading files, Writing to files, Directories, Interactive programming, Pickling, Exceptions

---

### INTRODUCTION TO FILES

We have so far created programs in Python that accept the input, manipulate it and display the output. But that output is available only during execution of the program and input is to be entered through the keyboard. This is because the variables used in a program have a lifetime that lasts till the time the program is under execution. What if we want to store the data that were input as well as the generated output permanently so that we can reuse it later? Usually, organisations would want to permanently store information about employees, inventory, sales, etc. to avoid repetitive tasks of entering the same data. Hence, data are stored permanently on secondary storage devices for reusability. We store Python programs written in script mode with a .py extension. Each program is stored on the secondary device as a file. Likewise, the data entered, and the output can be stored permanently into a file.

A file is a named location on a secondary storage media where data are permanently stored for later access.

### TYPES OF FILES

Computers store every file as a collection of 0s and 1s i.e., in binary form. Therefore, every file is basically just a series of bytes stored one after the other. There are mainly two types of data files — text file and binary file. A text file consists of human readable characters, which can be opened by any text editor. On the other hand, binary files are made up of non-human readable characters and symbols, which require specific programs to access its contents.

#### 1. Text file

A text file can be understood as a sequence of characters consisting of alphabets, numbers and other special symbols. Files with extensions like .txt, .py, .csv, etc. are some examples of text files. When we open a text file using a text editor (e.g., Notepad), we see several lines of text. However, the file contents are not stored in such a way internally. Rather, they are stored in sequence of bytes consisting of 0s and 1s. In ASCII, UNICODE or any other encoding scheme, the value of each character of the text file is stored as bytes. So, while opening a text file, the text editor translates each ASCII value and shows us the equivalent character that is readable by the human being. For example, the ASCII value 65 (binary equivalent 1000001) will be displayed by a text editor as the letter 'A' since the number 65 in ASCII character set represents 'A'. Each line

of a text file is terminated by a special character, called the End of Line (EOL). For example, the default EOL character in Python is the newline (`\n`). However, other characters can be used to indicate EOL. When a text editor or a program interpreter encounters the ASCII equivalent of the EOL character, it displays the remaining file contents starting from a new line. Contents in a text file are usually separated by whitespace, but comma (,) and tab (`\t`) are also commonly used to separate values in a text file.

## 2. Binary Files

Binary files are also stored in terms of bytes (0s and 1s), but unlike text files, these bytes do not represent the ASCII values of characters. Rather, they represent the actual content such as image, audio, video, compressed versions of other files, executable files, etc. These files are not human readable. Thus, trying to open a binary file using a text editor will show some garbage values. We need specific software to read or write the contents of a binary file. Binary files are stored in a computer in a sequence of bytes. Even a single bit change can corrupt the file and make it unreadable to the supporting application. Also, it is difficult to remove any error which may occur in the binary file as the stored contents are not human readable. We can read and write both text and binary files through Python programs.

## OPENING AND CLOSING A TEXT FILE

In real world applications, computer programs deal with data coming from different sources like databases, CSV files, HTML, XML, JSON, etc. We broadly access files either to write or read data from it. But operations on files include creating and opening a file, writing data in a file, traversing a file, reading data from a file and so on. Python has the `io` module that contains different functions for handling files.

### 1. Opening a file

To open a file in Python, we use the `open()` function. The syntax of `open()` is as follows:

```
file_object= open(file_name, access_mode)
```

This function returns a file object called file handle which is stored in the variable `file_object`. We can use this variable to transfer data to and from the file (read and write) by calling the functions defined in the Python's `io` module. If the file does not exist, the above statement creates a new empty file and assigns it the name we specify in the statement.

The `file_object` has certain attributes that tells us basic information about the file, such as:

- `<file.closed>` returns true if the file is closed and false otherwise.
- `<file.mode>` returns the access mode in which the file was opened.
- `<file.name>` returns the name of the file.

The `file_name` should be the name of the file that has to be opened. If the file is not in the current working directory, then we need to specify the complete path of the file along with its name.

The `access_mode` is an optional argument that represents the mode in which the file has to be accessed by the program. It is also referred to as processing mode. Here mode means the operation for which the file has to be opened like `<r>` for reading, `<w>` for writing, `<+>` for both reading and writing, `<a>` for appending at the end of an existing file. The default is the read mode. In addition, we can specify whether the file will be handled as binary (`<b>`) or text mode. By default, files are opened in text mode that means strings can be read or written. Files containing non-textual data are opened in binary mode that means read/write are performed in terms of bytes. The following Table

lists various file access modes that can be used with the `open()` method. The file offset position in the table refers to the position of the file object when the file is opened in a particular mode.

### File Open Modes

File Mode	Description	File Offset position
<r>	Opens the file in read-only mode.	Beginning of the file
<rb>	Opens the file in binary and read-only mode.	Beginning of the file
<r+> or <+r>	Opens the file in both read and write mode.	Beginning of the file
<w>	Opens the file in write mode. If the file already exists, all the contents will be overwritten. If the file doesn't exist, then a new file will be created.	Beginning of the file
<wb+> or <+wb>	Opens the file in read,write and binary mode. If the file already exists, the contents will be overwritten. If the file doesn't exist, then a new file will be created.	Beginning of the file
<a>	Opens the file in append mode. If the file doesn't exist, then a new file will be created.	End of the file
<a+> or <+a>	Opens the file in append and read mode. If the file doesn't exist, then it will create a new file.	End of the file

Consider the following example.

```
myObject=open("myfile.txt", "a+")
```

In the above statement, the file *myfile.txt* is opened in append and read modes. The file object will be at the end of the file. That means we can write data at the end of the file and at the same time we can also read data from the file using the file object named *myObject*.

## 2. Closing a file

Once we are done with the read/write operations on a file, it is a good practice to close the file. Python provides a `close()` method to do so. While closing a file, the system frees the memory allocated to it. The syntax of `close()` is:

```
file_object.close()
```

Here, file object is the object that was returned while opening the file. Python makes sure that any unwritten or unsaved data is flushed off (written) to the file before it is closed. Hence, it is always advised to close the file once our work is done. Also, if the file object is re-assigned to some other file, the previous file is automatically closed.

## 3. Opening a file using with clause

In Python, we can also open a file using with clause. The syntax of with clause is:

```
with open (file_name, access_mode) as file_ object:
```

The advantage of using with clause is that any file that is opened using this clause is closed automatically, once the control comes outside the with clause. In case the user forgets to close the file explicitly or if an exception occurs, the file is closed automatically. Also, it provides a simpler syntax.

```
with open("myfile.txt","r+") as myObject:
    content = myObject.read()
```

Here, we don't have to close the file explicitly using `close()` statement. Python will automatically close the file.

#### 4. WRITING TO A TEXT FILE

For writing to a file, we first need to open it in write or append mode. If we open an existing file in write mode, the previous data will be erased, and the file object will be positioned at the beginning of the file. On the other hand, in append mode, new data will be added at the end of the previous data as the file object is at the end of the file. After opening the file, we can use the following methods to write data in the file.

- `write()` - for writing a single string
- `writelines()` - for writing a sequence of strings

##### The `write()` method

`write()` method takes a string as an argument and writes it to the text file. It returns the number of characters being written on single execution of the `write()` method. Also, we need to add a newline character (`\n`) at the end of every sentence to mark the end of line. Consider the following piece of code:

```
>>> myobject=open("myfile.txt", 'w')
>>> myobject.write("Hey I have started #using files in
Python\n") 41
>>> myobject.close()
```

On execution, `write()` returns the number of characters written on to the file. Hence, 41, which is the length of the string passed as an argument, is displayed. **Note:** '`\n`' is treated as a single character. If numeric data are to be written to a text file, the data need to be converted into string before writing to the file. For example:

```
>>>myobject=open("myfile.txt", 'w')
>>> marks=58 #number 58 is converted to a string using #str()
>>> myobject.write(str(marks))
2
>>>myobject.close()
```

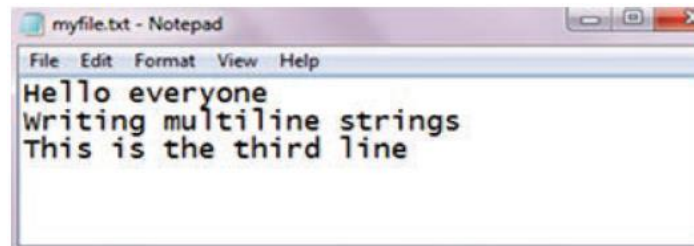
The `write()` actually writes data onto a buffer. When the `close()` method is executed, the contents from this buffer are moved to the file located on the permanent storage.

##### The `writelines()` method

This method is used to write multiple strings to a file. We need to pass an iterable object like lists, tuple, etc. containing strings to the `writelines()` method. Unlike `write()`, the `writelines()` method does not return the number of characters written in the file. The following code explains the use of `writelines()`.

```
>>> myobject=open("myfile.txt", 'w')
>>> lines = ["Hello everyone\n", "Writing #multiline strings\n", "This
is the #third line"]
>>> myobject.writelines(lines)
>>>myobject.close()
```

On opening `myfile.txt`, using notepad, its content will appear as shown in Figure :



*Contents of myfile.txt*

## 5. READING FROM A TEXT FILE

We can write a program to read the contents of a file. Before reading a file, we must make sure that the file is opened in “r”, “r+”, “w+” or “a+” mode. There are three ways to read the contents of a file:

### 1. The read() method

This method is used to read a specified number of bytes of data from a data file. The syntax of read() method is:

```
file_object.read(n)
```

Consider the following set of statements to understand the usage of read() method:

```
>>> myobject=open("myfile.txt", 'r')
>>> myobject.read(10)
'Hello ever'
>>> myobject.close()
```

If no argument or a negative number is specified in read(), the entire file content is read.

For example,

```
>>> myobject=open("myfile.txt", 'r')
>>> print(myobject.read())
Hello everyone Writing multiline strings This is the third line
>>> myobject.close()
```

### 2. The readline([n]) method

This method reads one complete line from a file where each line terminates with a newline (\n) character. It can also be used to read a specified number (n) of bytes of data from a file but maximum up to the newline character (\n). In the following example, the second statement reads the first ten characters of the first line of the text file and displays them on the screen.

```
>>> myobject=open("myfile.txt", 'r')
>>> myobject.readline(10)
'Hello ever'
>>> myobject.close()
```

If no argument or a negative number is specified, it reads a complete line and returns string.

```
>>> myobject=open("myfile.txt", 'r')
>>> print (myobject.readline())
'Hello everyone\n'
```

To read the entire file line by line using the readline(), we can use a loop. This process is known as looping/ iterating over a file object. It returns an empty string when EOF is reached.

### 3. The readlines() method

The method reads all the lines and returns the lines along with newline as a list of strings. The following example uses readlines() to read data from the text file *myfile.txt*.

```
>>> myobject=open("myfile.txt", 'r')
>>> print(myobject.readlines())
['Hello everyone\n', 'Writing multiline strings\n', 'This is
the third line']
>>> myobject.close()
```

As shown in the above output, when we read a file using readlines() function, lines in the file become members of a list, where each list element ends with a newline character ('\n'). In case we want to display each word of a line separately as an element of a list, then we can use split() function. The following code demonstrates the use of split() function.

```
>>> myobject=open("myfile.txt", 'r')
>>> d=myobject.readlines()
>>> for line in d:
    words=line.split()
    print(words)
['Hello', 'everyone']
['Writing', 'multiline', 'strings']
['This', 'is', 'the', 'third', 'line']
```

In the output, each string is returned as elements of a list. However, if *splitlines()* is used instead of split(), then each line is returned as element of a list, as shown in the output below:

```
>>> for line in d:
    words=line.splitlines()
    print(words)
['Hello everyone']
['Writing multiline strings']
['This is the third line']
```

**Write a program that accepts a string from the user and writes it to a text file.**

**#Program : Writing and reading to a text file**

```
fobject=open("testfile.txt","w") # creating a data file
sentence=input("Enter the contents to be written in the file: ")
fobject.write(sentence) # Writing data to the file
fobject.close() # Closing a file
print("Now reading the contents of the file: ")
fobject=open("testfile.txt","r")
#looping over the file object to read the file
for str in fobject:
    print(str)
fobject.close()
```

The file named *testfile.txt* is opened in write mode and the file handle named *fobject* is returned. The string is accepted from the user and written in the file using write(). Then the file is closed and again opened in read mode. Data is read from the file and displayed till the end of file is reached.

## 6. SETTING OFFSETS IN A FILE

The functions that we have learnt till now are used to access the data sequentially from a file. But if we want to access data in a random fashion, then Python gives us `seek()` and `tell()` functions to do so.

### 1 The `tell()` method

This function returns an integer that specifies the current position of the file object in the file. The position so specified is the byte position from the beginning of the file till the current position of the file object. The syntax of using `tell()` is:

```
file_object.tell()
```

### 2 The `seek()` method

This method is used to position the file object at a particular position in a file. The syntax of `seek()` is:

```
file_object.seek(offset [, reference_point])
```

In the above syntax, `offset` is the number of bytes by which the file object is to be moved. `reference_point` indicates the starting position of the file object. That is, with reference to which position, the offset has to be counted. It can have any of the following values: 0 - beginning of the file 1 - current position of the file 2 - end of file.

By default, the value of `reference_point` is 0, i.e. the offset is counted from the beginning of the file. For example, the statement `fileObject.seek(5, 0)` will position the file object at 5th byte position from the beginning of the file.

**The code in Program given below demonstrates the usage of `seek()` and `tell()`.**

**Program - Application of `seek()` and `tell()`**

```
print("Learning to move the file object")
fileobject=open("testfile.txt","r+")
str=fileobject.read()
print(str)
print("Initially, the position of the file object is: ",
fileobject.tell())
fileobject.seek(0)
print("Now the file object is at the beginning of the file:
",fileobject.tell())
fileobject.seek(10)
print("We are moving to 10th byte position from the beginning of
file")
print("The position of the file object is at",
fileobject.tell()) str=fileobject.read()
print(str)
```

### Output of Program

```
Learning to move the file object
roll_numbers = [1, 2, 3, 4, 5, 6]
```



Initially, the position of the file object is: 33  
Now the file object is at the beginning of the file: 0  
We are moving to 10th byte position from the beginning of file  
The position of the file object is at 10  
numbers = [1, 2, 3, 4, 5, 6] >>>

### Program : To display data from a text file

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

### Program :To perform reading and writing operation in a text file

```
fileobject=open("report.txt", "w+")
print ("WRITING DATA IN THE FILE")
print() # to display a blank line
while True:
    line= input("Enter a sentence ")
    fileobject.write(line)
    fileobject.write('\n')
    choice=input("Do you wish to enter more data? (y/n): ")
    if choice in ('n','N'):
        break
    print("The byte position of file object is ",fileobject.tell())
    fileobject.seek(0) #places file object at beginning of file
    print()
    print("READING DATA FROM THE FILE")
    str=fileobject.read()
    print(str)
    fileobject.close()
```

In the Program the file will be read till the time end of file is not reached and the output as shown in below is displayed.

#### WRITING DATA IN THE FILE

```
Enter a sentence I am a student of class XII
Do you wish to enter more data? (y/n): y
Enter a sentence my school contact number is 4390xxx8
Do you wish to enter more data? (y/n): n
The byte position of file object is 67
```

#### READING DATA FROM THE FILE

```
I am a student of class XII my school contact number is 4390xxx8
>>>
```



## 7. THE PICKLE MODULE

We know that Python considers everything as an object. So, all data types including list, tuple, dictionary, etc. are also considered as objects. During execution of a program, we may require to store current state of variables so that we can retrieve them later to its present state. Suppose you are playing a video game, and after some time, you want to close it. So, the program should be able to store the current state of the game, including current level/stage, your score, etc. as a Python object. Likewise, you may like to store a Python dictionary as an object, to be able to retrieve later. To save any object structure along with data, Python provides a module called Pickle. The module Pickle is used for serializing and de-serializing any Python object structure. Pickling is a method of preserving food items by placing them in some solution, which increases the shelf life. In other words, it is a method to store food items for later consumption.

Serialization is the process of transforming data or an object in memory (RAM) to a stream of bytes called byte streams. These byte streams in a binary file can then be stored in a disk or in a database or sent through a network. **Serialization process is also called pickling.**

**De-serialization or unpickling** is the inverse of pickling process where a byte stream is converted back to Python object.

The pickle module deals with binary files. Here, data are not written but dumped and similarly, data are not read but loaded. The Pickle Module must be imported to load and dump data. The pickle module provides two methods - dump() and load() to work with binary files for pickling and unpickling, respectively.

### 1 The dump() method

This method is used to convert (pickling) Python objects for writing data in a binary file. The file in which data are to be dumped, needs to be opened in binary write mode (wb).

Syntax of dump() is as follows:

```
dump(data_object, file_object)
```

where data\_object is the object that has to be dumped to the file with the file handle named file\_object. For example, the following Program writes the record of a student (roll\_no, name, gender and marks) in the binary file named mybinary.dat using the dump(). We need to close the file after pickling.

#### Program : Pickling data in Python

```
import pickle
listvalues=[1,"Geetika",'F', 26]
fileobject=open("mybinary.dat", "wb")
pickle.dump(listvalues,fileobject)
fileobject.close()
```

### 2 The load() method

This method is used to load (unpickling) data from a binary file. The file to be loaded is opened in binary read (rb) mode.

Syntax of load() is as follows:

```
Store_object = load(file_object)
```

Here, the pickled Python object is loaded from the file having a file handle named `file_object` and is stored in a new file handle called `store_object`. The program 2-7 demonstrates how to read data from the file *mybinary.dat* using the `load()`.

### Program - Unpickling data in Python

```
import pickle
print("The data that were stored in file are: ")
fileobject=open("mybinary.dat","rb")
objectvar=pickle.load(fileobject)
fileobject.close()
print(objectvar)
```

#### Output of Program :

```
>>> RESTART:
The data that were stored in file are:
[1, 'Geetika', 'F', 26]
>>>
```

## 3. File handling using pickle module

As we read and write data in a text file, similarly we will be adding and displaying data for a binary file. The following Program accepts a record of an employee from the user and appends it in the binary file *tv*. Thereafter, the records are read from the binary file and displayed on the screen using the same object. The user may enter as many records as they wish to. The program also displays the size of binary files before starting with the reading process.

### Program : To perform basic operations on a binary file using pickle module

#### # Program to write and read employee records in a binary file

```
import pickle
print("WORKING WITH BINARY FILES")
bfile=open("empfile.dat","ab")
recno=1
print ("Enter Records of Employees")
print()
#taking data from user and dumping in the file as list object

while True:
    print("RECORD No.", recno)
    eno=int(input("\tEmployee number : "))
    ename=input("\tEmployee Name : ")
    ebasic=int(input("\tBasic Salary : "))
    allow=int(input("\tAllowances : "))
    totalsal=ebasic+allow
    print("\tTOTAL SALARY : ", totalsal)
    edata=[eno,ename,ebasic,allow,totsal]
```

```
pickle.dump(edata,bfile)
ans=input("Do you wish to enter more records (y/n)? ")
recno=recno+1
if ans.lower()=='n':
    print("Record entry OVER ")
    print()
    break
# retrieving the size of file
print("Size of binary file (in bytes):",bfile.tell())
bfile.close()
# Reading the employee records from the file using load() module
print("Now reading the employee records from the file")
print()
readrec=1
try:
    with open("empfile.dat","rb") as bfile:
        while True:
            edata=pickle.load(bfile)
            print("Record Number : ",readrec)
            print(edata)
            readrec=readrec+1
except EOFError:
    pass
bfile.close()
```

### Output of Program :

#### WORKING WITH BINARY FILES

Enter Records of Employees

RECORD No. 1

```
Employee number : 11
Employee Name : D N Ravi
Basic Salary : 32600
Allowances : 4400
TOTAL SALARY : 37000
```

Do you wish to enter more records (y/n)? y

RECORD No. 2

```
Employee number : 12
Employee Name : Farida Ahmed
Basic Salary : 38250
Allowances : 5300
TOTAL SALARY : 43550
```

Do you wish to enter more records (y/n)? n

Record entry OVER

Size of binary file (in bytes): 216

Now reading the employee records from the file

```
Record Number : 1
11, 'D N Ravi', 32600, 4400, 37000]
Record Number : 2
12, 'Farida Ahmed', 38250, 5300, 43550]
>>>
```

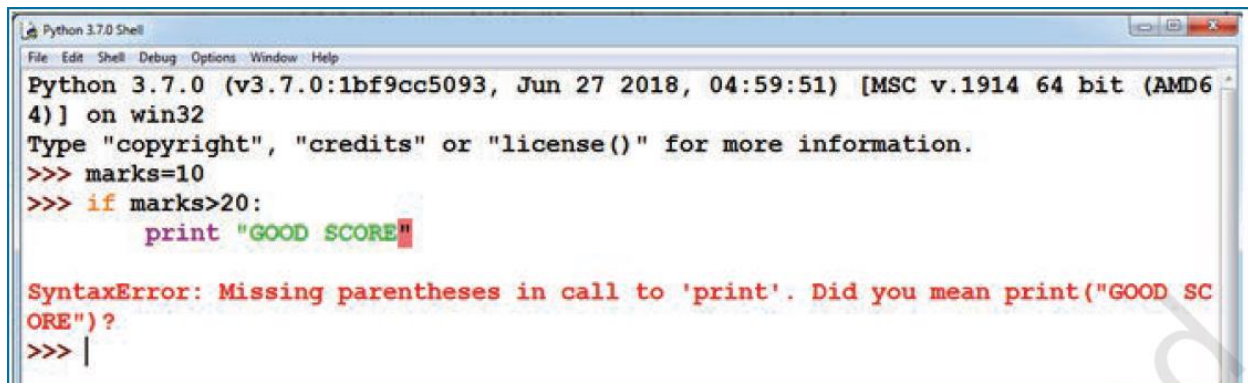
As each employee record is stored as a list in the file `empfile.dat`, hence while reading the file, a list is displayed showing record of each employee. Notice that in Program 2-8, we have also used `try.. except` block to handle the end-of-file exception.

## Exception Handling in Python

Sometimes while executing a Python program, the program does not execute at all or the program executes but generates unexpected output or behaves abnormally. These occur when there are syntax errors, runtime errors or logical errors in the code. In Python, exceptions are errors that get triggered automatically. However, exceptions can be forcefully triggered and handled through program code.

### 1 SYNTAX ERRORS

Syntax errors are detected when we have not followed the rules of the particular programming language while writing a program. These errors are also known as *parsing errors*. On encountering a syntax error, the interpreter does not execute the program unless we rectify the errors, save and rerun the program. When a syntax error is encountered while working in shell mode, Python displays the name of the error and a small description about the error.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> marks=10
>>> if marks>20:
>>>     print "GOOD SCORE"
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("GOOD SCORE")?
>>> |
```

### 2 EXCEPTIONS

Even if a statement or expression is syntactically correct, there might arise an error during its execution. For example, trying to open a file that does not exist, division by zero and so on. Such types of errors might disrupt the normal execution of the program and are called exceptions. An exception is a Python object that represents an error. When an error occurs during the execution of a program, an exception is said to have been raised. Such an exception needs to be handled by the programmer so that the program does not terminate abnormally. Therefore, while designing a program, a programmer may anticipate such erroneous situations

that may arise during its execution and can address them by including appropriate code to handle that exception.

### 3 BUILT-IN EXCEPTIONS

Commonly occurring exceptions are usually defined in the compiler/interpreter. These are called built-in exceptions. Python's standard library is an extensive collection of built-in exceptions that deals with the commonly occurring errors (exceptions) by providing the standardized solutions for such errors. On the occurrence of any built-in exception, the appropriate exception handler code is executed which displays the reason along with the raised exception name. The programmer then has to take appropriate action to handle it. Some of the commonly occurring built-in exceptions that can be raised in Python are explained in Table.

S. No	Name of the Built-in Exception	Explanation
1.	SyntaxError	It is raised when there is an error in the syntax of the Python code.
2.	ValueError	It is raised when a built-in method or operation receives an argument that has the right data type but mismatched or inappropriate values.
3.	IOError	It is raised when the file specified in a program statement cannot be opened.
4	KeyboardInterrupt	It is raised when the user accidentally hits the Delete or Esc key while executing a program due to which the normal flow of the program is interrupted.
5	ImportError	It is raised when the requested module definition is not found.
6	EOFError	It is raised when the end of file condition is reached without reading any data by input().
7	ZeroDivisionError	It is raised when the denominator in a division operation is zero.
8	IndexError	It is raised when the index or subscript in a sequence is out of range.
9	NameError	It is raised when a local or global variable name is not defined.
10	IndentationError	It is raised due to incorrect indentation in the program code.
11	TypeError	It is raised when an operator is supplied with a value of incorrect data type.
12	OverflowError	It is raised when the result of a calculation exceeds the maximum limit for numeric data type.

```

Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print (50/0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print (50/0)
ZeroDivisionError: division by zero
>>> print (var+40)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print (var+40)
NameError: name 'var' is not defined
>>> 10+'5'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    10+'5'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> |

```

A programmer can also create custom exceptions to suit one's requirements. These are called user-defined exceptions.

## 4 RAISING EXCEPTIONS

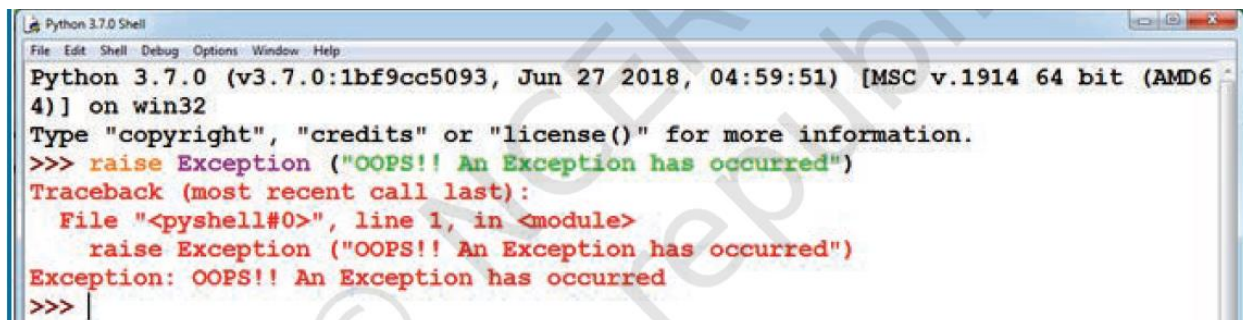
Each time an error is detected in a program, the Python interpreter raises (throws) an exception. Exception handlers are designed to execute when a specific exception is raised. Programmers can also forcefully raise exceptions in a program using the raise and assert statements. Once an exception is raised, no further statement in the current block of code is executed. So, raising an exception involves interrupting the normal flow execution of program and jumping to that part of the program (exception handler code) which is written to handle such exceptional situations.

### The raise Statement

The raise statement can be used to throw an exception. The syntax of raise statement is:

```
raise exception-name[(optional argument)]
```

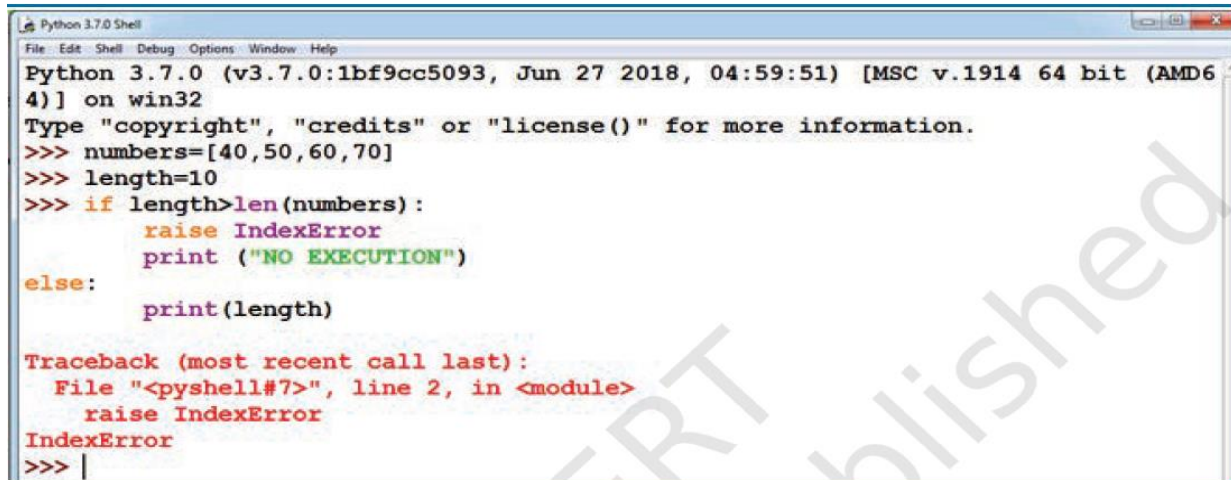
The argument is generally a string that is displayed when the exception is raised. For example, when an exception is raised as shown in the following Figure, the message "OOPS : An Exception has occurred" is displayed along with a brief description of the error.

A screenshot of a Python 3.7.0 Shell window. The window title is "Python 3.7.0 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The text in the window shows the Python version and build information: "Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32". It then prompts the user to type "copyright", "credits", or "license()". The user enters the command ">>> raise Exception ('OOPS!! An Exception has occurred')". The output shows a traceback: "Traceback (most recent call last):", "File '<pyshell#0>', line 1, in <module>", "raise Exception ('OOPS!! An Exception has occurred')", and "Exception: OOPS!! An Exception has occurred". The prompt ">>>|" is visible at the bottom.

```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> raise Exception ('OOPS!! An Exception has occurred')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise Exception ('OOPS!! An Exception has occurred')
Exception: OOPS!! An Exception has occurred
>>> |
```

The error detected may be a built-in exception or may be a user-defined one. Consider the example given in the following Figure that uses the raise statement to raise a built-in exception called `IndexError`.



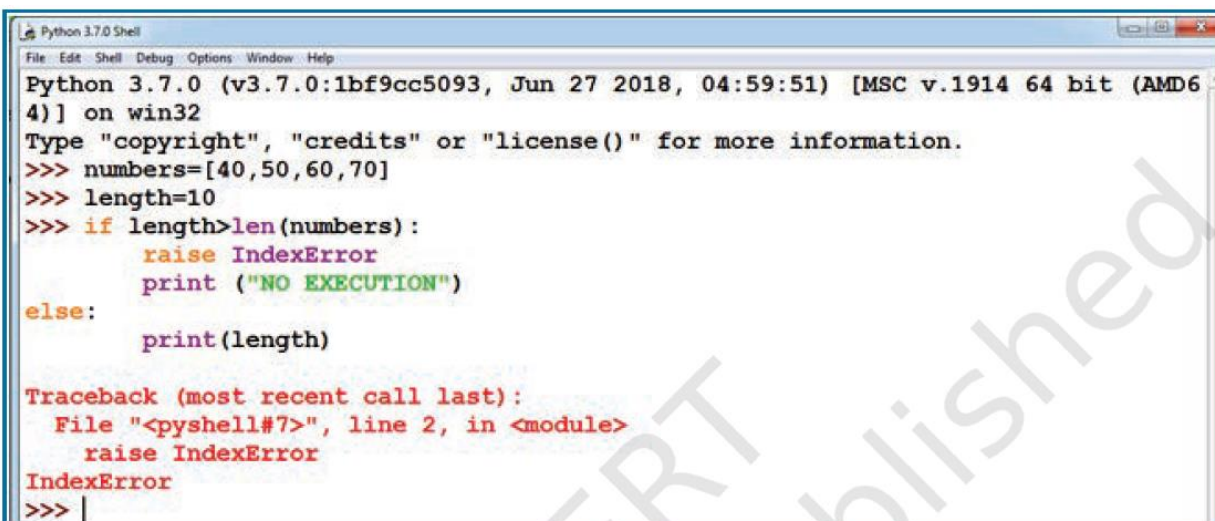


```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> numbers=[40,50,60,70]
>>> length=10
>>> if length>len(numbers):
>>>     raise IndexError
>>>     print ("NO EXECUTION")
>>> else:
>>>     print(length)

Traceback (most recent call last):
  File "<pyshell#7>", line 2, in <module>
    raise IndexError
IndexError
>>> |
```

**Note:** In this case, the user has only raised the exception but has not displayed any error message explicitly.

In the following Figure, since the value of variable *length* is greater than the length of the list *numbers*, an `IndexError` exception will be raised. The statement following the `raise` statement will not be executed. So the message "NO EXECUTION" will not be displayed in this case.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> numbers=[40,50,60,70]
>>> length=10
>>> if length>len(numbers):
>>>     raise IndexError
>>>     print ("NO EXECUTION")
>>> else:
>>>     print(length)

Traceback (most recent call last):
  File "<pyshell#7>", line 2, in <module>
    raise IndexError
IndexError
>>> |
```

## The assert Statement

An `assert` statement in Python is used to test an expression in the program code. If the result after testing comes false, then the exception is raised. This statement is generally used in the beginning of the function or after a function call to check for valid input. The syntax for `assert` statement is:

```
assert Expression[,arguments]
```

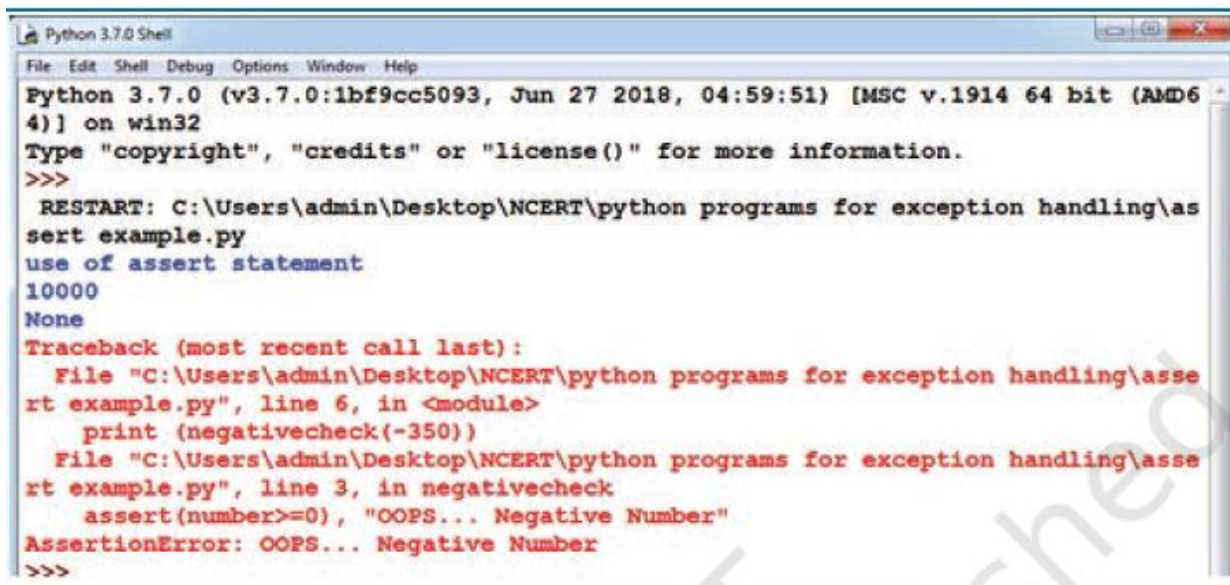


On encountering an assert statement, Python evaluates the expression given immediately after the assert keyword. If this expression is false, an `AssertionError` exception is raised which can be handled like any other exception. Consider the code given below:

Program :

```
#Use of assert statement
print("use of assert statement")
def negativecheck(number):
    assert(number>=0), "OOPS... Negative Number"
print(number*number)
print (negativecheck(100))
print (negativecheck(-350))
```

In the code, the assert statement checks for the value of the variable number. In case the number gets a negative value, `AssertionError` will be thrown, and subsequent statements will not be executed. Hence, on passing a negative value (-350) as an argument, it results in `AssertionError` and displays the message "OOPS.... Negative Number".



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\Users\admin\Desktop\NCERT\python programs for exception handling\assert example.py
use of assert statement
10000
None
Traceback (most recent call last):
  File "C:\Users\admin\Desktop\NCERT\python programs for exception handling\assert example.py", line 6, in <module>
    print (negativecheck(-350))
  File "C:\Users\admin\Desktop\NCERT\python programs for exception handling\assert example.py", line 3, in negativecheck
    assert(number>=0), "OOPS... Negative Number"
AssertionError: OOPS... Negative Number
>>>
```

## HANDLING EXCEPTIONS

Each and every exception has to be handled by the programmer to avoid the program from crashing abruptly. This is done by writing additional code in a program to give proper messages or instructions to the user on encountering an exception. This process is known as *exception handling*.

### Need for Exception Handling

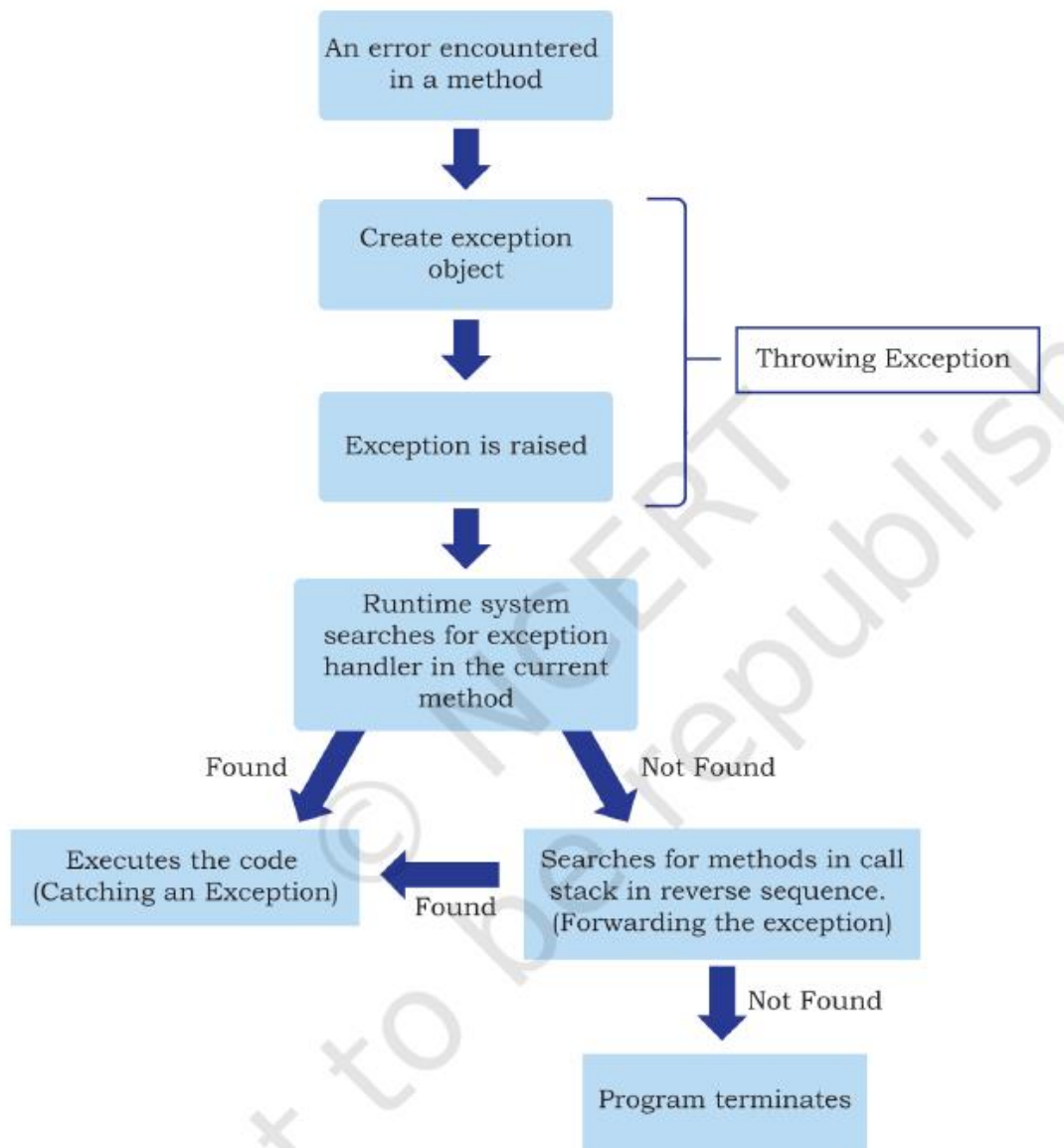
Exception handling is being used not only in Python programming but in most programming languages like C++, Java, Ruby, etc. It is a useful technique that helps in capturing runtime errors and handling them so as to avoid the program getting crashed. Following are some of the important points regarding exceptions and their handling:

- Python categorises exceptions into distinct types so that specific exception handlers (code to handle that particular exception) can be created for each type.

- Exception handlers separate the main logic of the program from the error detection and correction code. The segment of code where there is any possibility of error or exception, is placed inside one block. The code to be executed in case the exception has occurred, is placed inside another block. These statements for detection and reporting the exception do not affect the main logic of the program.
- The compiler or interpreter keeps track of the exact position where the error has occurred.
- Exception handling can be done for both user-defined and built-in exceptions.

### Process of Handling Exception

When an error occurs, Python interpreter creates an object called the *exception object*. This object contains information about the error like its type, file name and position in the program where the error has occurred. The object is handed over to the runtime system so that it can find an appropriate code to handle this particular exception. This process of creating an exception object and handing it over to the runtime system is called *throwing* an exception. It is important to note that when an exception occurs while executing a particular program statement, the control jumps to an exception handler, abandoning execution of the remaining program statements. The runtime system searches the entire program for a block of code, called the *exception handler* that can handle the raised exception. It first searches for the method in which the error has occurred and the exception has been raised. If not found, then it searches the method from which this method (in which exception was raised) was called. This hierarchical search in reverse order continues till the exception handler is found. This entire list of methods is known as *call stack*. When a suitable handler is found in the call stack, it is executed by the runtime process. This process of executing a suitable handler is known as *catching the exception*. If the runtime system is not able to find an appropriate exception after searching all the methods in the call stack, then the program execution stops. The following flowchart describes the exception handling process.



### Catching Exceptions

An exception is said to be caught when a code that is designed to handle a particular exception is executed. Exceptions, if any, are caught in the `try` block and handled in the `except` block. While writing or debugging a program, a user might doubt an exception to occur in a particular part of the code. Such suspicious lines of codes are put inside a `try` block. Every `try` block is followed by an `except` block. The appropriate code to handle each of the possible exceptions (in the code inside the `try` block) are written inside the `except` clause.

While executing the program, if an exception is encountered, further execution of the code inside the `try` block is stopped and the control is transferred to the `except` block. The syntax of `try ... except` clause is as follows:

```
try:
    [ program statements where exceptions might occur ]
except [exception-name]:
```

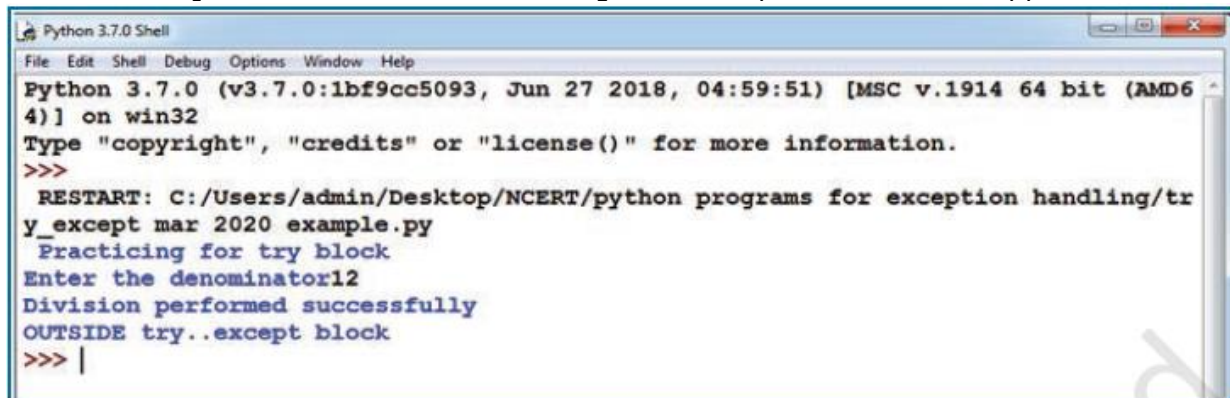
```
[ code for exception handling if the exception-name error  
is  
encountered]
```

Consider the Program 1-2 given below:

# Program Using try..except block

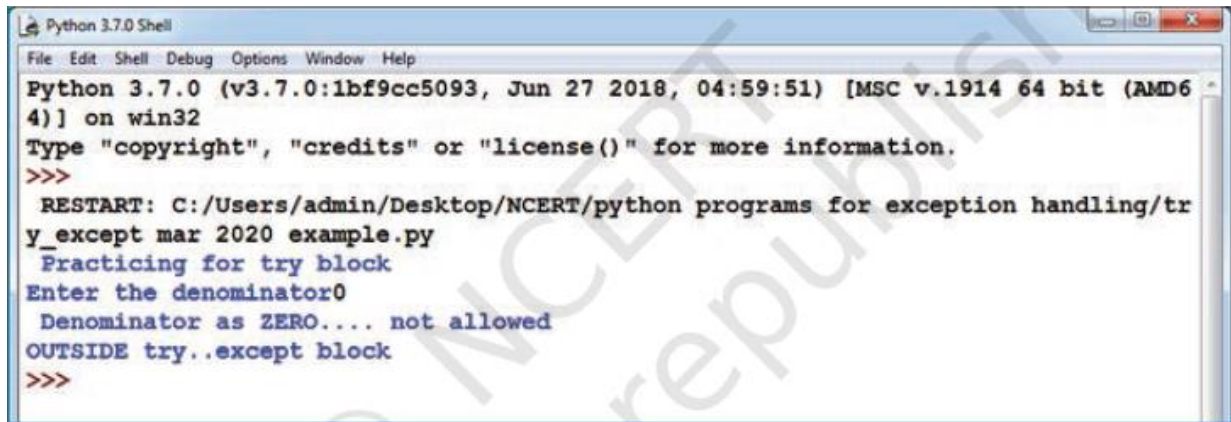
```
print ("Practicing for try block")  
try:  
    numerator=50  
    denom=int(input("Enter the denominator"))  
    quotient=(numerator/denom)  
    print(quotient)  
    print ("Division performed successfully")  
except ZeroDivisionError:  
    print ("Denominator as ZERO.... not allowed")  
print("OUTSIDE try..except block")
```

In the above Program the `ZeroDivisionError` exception is handled. If the user enters any non-zero value as denominator, the quotient will be displayed along with the message "Division performed successfully". The except clause will be skipped in this case.



```
Python 3.7.0 Shell  
File Edit Shell Debug Options Window Help  
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>>  
RESTART: C:/Users/admin/Desktop/NCERT/python programs for exception handling/tr  
y_except mar 2020 example.py  
Practicing for try block  
Enter the denominator:12  
Division performed successfully  
OUTSIDE try..except block  
>>> |
```

So, the next statement after the try..except block is executed and the message "OUTSIDE try.. except block" is displayed. However, if the user enters the value of denom as zero (0), then the execution of the try block will stop. The control will shift to the except block and the message "Denominator as Zero.... not allowed" will be displayed. Thereafter, the statement following the try..except block is executed and the message "OUTSIDE try..except block" is displayed in this case also.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/admin/Desktop/NCERT/python programs for exception handling/try_except
mar 2020 example.py
Practicing for try block
Enter the denominator0
Denominator as ZERO... not allowed
OUTSIDE try..except block
>>>
```

Sometimes, a single piece of code might be suspected to have more than one type of error. For handling such situations, we can have multiple except blocks for a single try block as shown in the following Program.

**#Program : Use of multiple except clauses**

```
print ("Handling multiple exceptions")
try:
    numerator=50
    denom=int(input("Enter the denominator: "))
    print (numerator/denom)
    print ("Division performed successfully")
except ZeroDivisionError:
    print ("Denominator as ZERO is not allowed")
except ValueError:
    print ("Only INTEGERS should be entered")
```

In the code, two types of exceptions (`ZeroDivisionError` and `ValueError`) are handled using two `except` blocks for a single `try` block. When an exception is raised, a search for the matching `except` block is made till it is handled. If no match is found, then the program terminates.

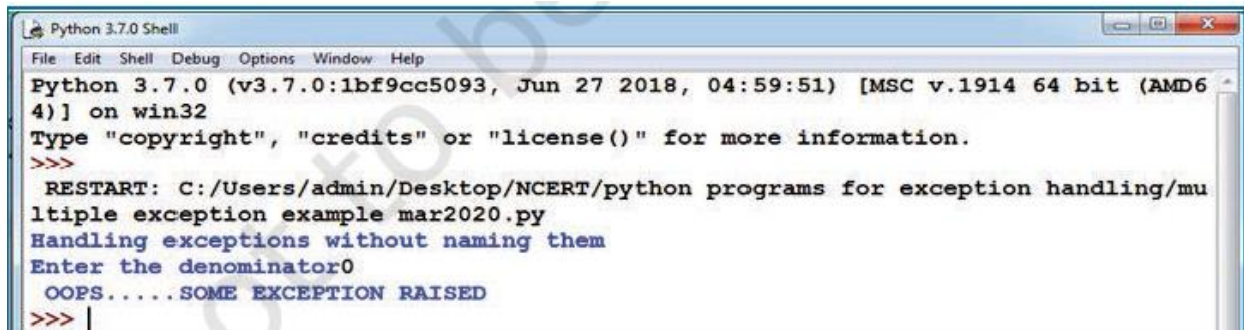
However, if an exception is raised for which no handler is created by the programmer, then such an exception can be handled by adding an `except` clause without specifying any exception. This `except` clause should be added as the last clause of the `try..except` block.

**# Program : Use of except without specifying an exception**

```
print ("Handling exceptions without naming them")
try:
    numerator=50
    denom=int(input("Enter the denominator"))
    quotient=(numerator/denom)
    print ("Division performed successfully")
except ValueError:
    print ("Only INTEGERS should be entered")
except: print(" OOPS.....SOME EXCEPTION RAISED")
```



If the above code is executed, and the denominator entered is 0 (zero) , the handler for `ZeroDivisionError` exception will be searched. Since it is not present, the last except clause (without any specified exception) will be executed , so the message “OOPS.....SOME EXCEPTION RAISED” will be displayed.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/admin/Desktop/NCERT/python programs for exception handling/multiple exception example mar2020.py
Handling exceptions without naming them
Enter the denominator0
OOPS.....SOME EXCEPTION RAISED
>>>
```

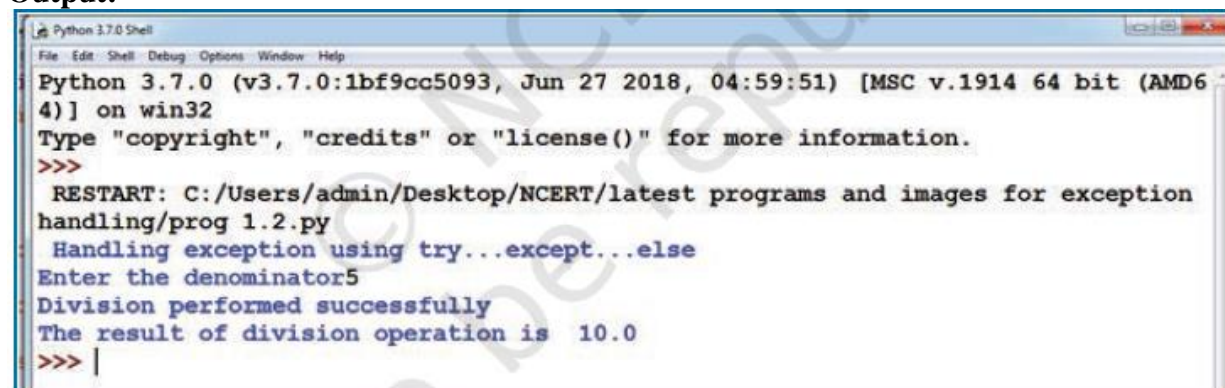
### try...except...else clause

We can put an optional else clause along with the try...except clause. An except block will be executed only if some exception is raised in the try block. But if there is no error then none of the except blocks will be executed. In this case, the statements inside the else clause will be executed. Program 1-5 along with its output explains the use of else block with the try...except block.

#Program : Use of else clause

```
print ("Handling exception using try...except...else")
try:
    numerator=50
    denom=int(input("Enter the denominator: "))
    quotient=(numerator/denom)
    print ("Division performed successfully")
except ZeroDivisionError:
    print ("Denominator as ZERO is not allowed")
except ValueError:
    print ("Only INTEGERS should be entered")
else:
    print ("The result of division operation is ", quotient)
```

Output:



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/admin/Desktop/NCERT/latest programs and images for exception handling/prog 1.2.py
Handling exception using try...except...else
Enter the denominator5
Division performed successfully
The result of division operation is 10.0
>>>
```

## FINALLY CLAUSE

The try statement in Python can also have an optional finally clause. The statements inside the finally block are always executed regardless of whether an exception has occurred in the try block or not. It is a common practice to use finally clause while working with files to ensure that the file object is closed. If used, finally should always be placed at the end of try clause, after all except blocks and the else block.

### #Program : Use of finally clause

```
print ("Handling exception using try...except...else...finally")
try:
    numerator=50
    denom=int(input("Enter the denominator: "))
    quotient=(numerator/denom)
    print ("Division performed successfully")
except ZeroDivisionError:
    print ("Denominator as ZERO is not allowed")
except ValueError:
    print ("Only INTEGERS should be entered")
else:
    print ("The result of division operation is ", quotient)
finally:
    print ("OVER AND OUT")
```

In the above program, the message “OVER AND OUT” will be displayed irrespective of whether an exception is raised or not.

## Recovering and continuing with finally clause

If an error has been detected in the try block and the exception has been thrown, the appropriate except block will be executed to handle the error. But if the exception is not handled by any of the except clauses, then it is re-raised after the execution of the finally block. For example, in the following Program, it contains only the except block for ZeroDivisionError. If any other type of error occurs for which there is no handler code (except clause) defined, then also the finally clause will be executed first.

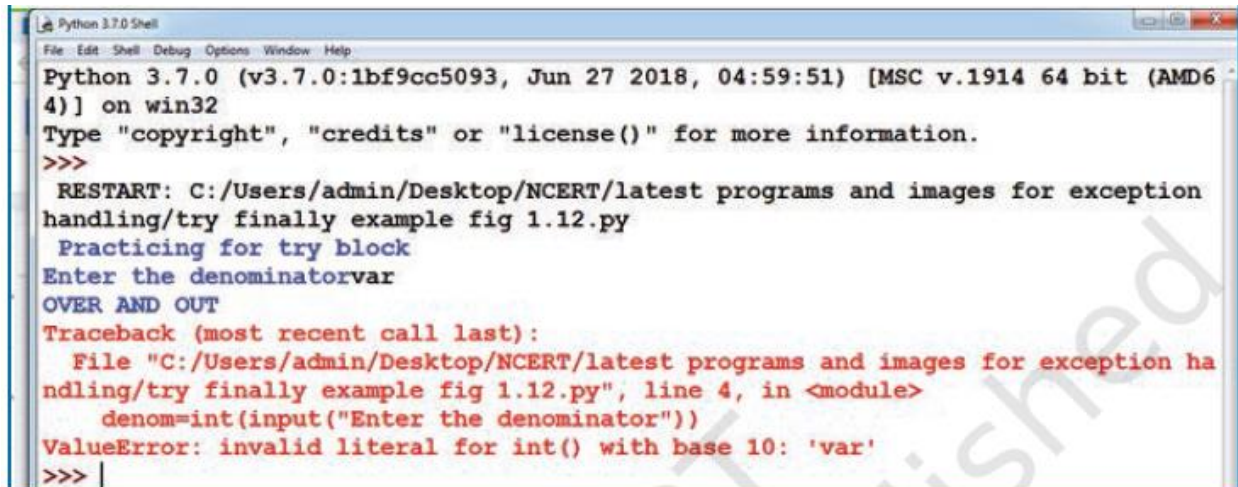
### # Program : Recovering through finally clause

```
print (" Practicing for try block")
try:
    numerator=50
    denom=int(input("Enter the denominator"))
    quotient=(numerator/denom)
    print ("Division performed successfully")
except ZeroDivisionError:
    print ("Denominator as ZERO is not allowed")
else:
    print ("The result of division operation is ", quotient)
```



```
finally:
    print ("OVER AND OUT")
```

While executing the above code, if we enter a non-numeric data as input, the finally block will be executed. So, the message “OVER AND OUT” will be displayed. Thereafter the exception for which handler is not present will be re-raised.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/admin/Desktop/NCERT/latest programs and images for exception handling/try finally example fig 1.12.py
Practicing for try block
Enter the denominatorvar
OVER AND OUT
Traceback (most recent call last):
  File "C:/Users/admin/Desktop/NCERT/latest programs and images for exception handling/try finally example fig 1.12.py", line 4, in <module>
    denom=int(input("Enter the denominator"))
ValueError: invalid literal for int() with base 10: 'var'
>>>
```

After execution of finally block, Python transfers the control to a previously entered try or to the next higher level default exception handler. In such a case, the statements following the finally block is executed. That is, unlike except, execution of the finally clause does not terminate the exception. Rather, the exception continues to be raised after execution of finally.

If we put a piece of code where there are possibilities of errors or exceptions to occur inside a try block. Inside each except clause we define handler codes to handle the matching exception raised in the try block. The optional else clause contains codes to be executed if no exception occurs. The optional finally block contains codes to be executed irrespective of whether an exception occurs or not.

**Note :** Syntax errors or parsing errors are detected when we have not followed the rules of the particular programming language while writing a program

## RENAMING A FILE

The **os module** Python provides methods that help to perform file-processing operations such as renaming and deleting files. To use this **os module** we need to import it first and then we can call any related functions. To rename an existing file, we can use the rename() method. This method takes two arguments, current filename and new filename. The following shows the syntax for rename () method

Syntax :

```
os.rename (current file name, new file name)
```

# Example Program

```
import os
```

```
# Rename a file from test.txt to Newtest.txt
```

```
os.rename ("test.txt", "Newtest.txt")
```

```
print ("File renamed. ")
```

Output :  
File renamed.

## DELETING A FILE

We can delete a file by using the `remove()` method available in `os` module. This method receives one argument which is the filename to be deleted. The following gives the syntax for `remove()` method.

Syntax :

```
os.remove (filename)
```

```
#Example Program
import os
#Delete a file
os.remove ("Newtest.txt")
print ("File Deleted.")
```

Output :  
File Deleted.

## DIRECTORIES IN PYTHON

All files will be saved in various directories, and Python has efficient methods for handling directories and files. The `os` module has several methods that help to create, remove, and change directories.

### 1 `mkdir()` method

The `mkdir()` method of the `os` module is used to create directories in the current directory. We need to supply an argument to this method which contains the name of the directory to be created. The following shows the syntax of `mkdir()` method.

Syntax :

```
os.mkdir ("dirname")
```

#Example

```
import os
#Create a directory "Fruits"
os.mkdir ("Fruits")
```

Output :  
The above example creates a directory named Fruits.

### 2 `chdir()` method

To change the current directory, we can use the `chdir()` method. The `chdir()` method takes an argument, which is the name of the directory that we want to make the current directory. The following shows the syntax of `chdir()` method.

Syntax:

```
os.chdir ("dirname")
```

```
#Example
import os
# Change a directory
os.chdir("/home/abc")
```

Output :

The above example goes to the directory `/home/abc`.

### 3 `getcwd()` method

The `getcwd()` method displays the current working directory. The following shows the syntax of `getcwd()` method.

Syntax :

```
os.getcwd ()
```

```
#Example
import os
#Displays the location of current directory
os.getcwd ()
```

### 4 `rmdir()` method

The `rmdir()` method deletes the directory, which is passed as an argument in the method.

Syntax :

```
os.rmdir( "dirname")
```

```
#Example
import os
#Removes the directory
os.rmdir ( "Fruits")
```

It is intended to give the exact path of a directory. Otherwise it will search the current working directory.

### Write a program to open a file in write mode

```
#Open a file in write mode
```

```
fo = Open ( "Demo.txt" w")
seq= ["First Line \n", " Second Line\n", "Third Line\n", "Fourth
Line\n", "Fifth Line\n"]
fo.writelines (seq)
```

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

### Write a program to copy a text file to another file

```
# Open the file to be copied in read mode
file1 =input ( "Enter the source file to be copied: ")
file2=input ("Enter the destination file name: ")
fr=open (file1, "r") # Open a file to be copied in write mode
fw=open (file2, "w")
for line in fr.readlines ():
    fw. write (line)
#close the files
fr.close ()
fw.close ()
print ("1 File Copied")
```

Output :  
1 File Copied

### Program to count the number of lines in a file.

```
fr=open ( "Demo.txt", "r")
countlines=0
for line in fr.readlines ():
    countlines=countlines+1
    print ("Number of Lines:", countlines)
#close the file
fr.close ()
```

Output :  
Number of Lines: 5

### Program to count the frequencies of each word from a file.

```
#Open a file in write mode

fo = Open ( "Demo.txt" w")
seq= ["Programming in Python is fun \n", "Python is easy\n"]
fo.writelines (seq)
#close the file
fo.close()
```

```
#Open the file to be read in read mode
fr=open ( "test.txt", "r")
```

```
wordcount={}

for word in fr.read().split ():
    if word not in wordcount :
        wordcount[word]= 1
    else :
        wordcount[word] += 1
for k,v in wordcount.items ():
    print (k, v)
#close the file
fr.close ()
```

Output:  
Programming 1  
in 1  
Python 2  
is 2  
fun 1  
easy 1

### Write a program to append a file with the contents of another file.

```
file1=input ( "Enter the file to be opened for appending:")
file2=input ("Enter the file name to be appended: ")
#Open the file to in append mode
fa=open (file1, "a")
#Open a file in read mode
fr=open (file2, "r")
for line in fr.readlines () :
    fa.write (line)
#close the files
fr.close ()
fa.close ()
print ("1 File Appended")
```

Output :

```
Enter the file to be opened for
appending: TruncateDemoCopy.py
Enter the file name to be
appended: TruncateDemo.py
1 File Appended
```