Chapter 6.

File I/O Handling and Exception Handling

12 Marks

I/O Operations:

Reading Keyboard Input:

- Python provides ***two built-in functions to read a line of text*** from standard input, which by default comes from the keyboard. These functions are −

1. input(prompt)
2. input()

The input(prompt) Function

- The *input([prompt])* function allows user input. It takes one argument. The syntax is as follows:

Syntax:

```
input(prompt)
```

Example:

```
x = input("Enter your name: ")

print('Hello ' + x)
```

Output−

```
Enter your name: Kedar
```

```
Hello Kedar
```

The input() Function

- The *input()* function always evaluate the input provided by user and return same type data**.** The syntax is as follows**:**

***Syntax*:**

x=input()

Example:

x=int(input())

print(type(x))

Output−

5

<class 'int'>

Printing to Screen:

- The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows –
- The syntax is

  Print(object(s),separator=separator,end=end,file=file,flush=flush)

Parameter values:

Object(s) : It can be any object but will be converted to string before primted.

sep='seperator' : Optional. Specify how to separate the objects, if there are more than one. Default is ''.

end='end': Optional. Specify what to print at the end. Default is '\n'(line feed).

file: Optional. An object with a write method, Default is sys.stdout

flush: Optional. A Boolean, specifying if the output is flushed(True) or buffered(False). Default is False.

Example

```
print("hello","how are you?",sep="---")
```

Output:

```
hello---how are you?
```

- Some other example for printing to screen.

Example

```
print("Python is really a great language,", "isn't it?")
```

Output:

```
Python is really a great language, isn't it?
```

- To make the output more attractive formatting is used. This can be done by using the str.format() method.

Example

```
a=10

b=20

print('Value of a is {} and b is {}'.format(a,b))
```

Output:

```
Value of a is 10 and b is 20
```

- We can format the number output by using % operator.

Example

```
a=12.3456789

print('Value of x is=%3.2f'%a)

print('Value of x is=%3.4f'%a)
```

Output:

```
Value of x is=12.35

Value of x is=12.3457
```

File Handling:

What is a file?

- ***File is a named location on disk to store related information*.** It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we *use files for future use of the data***.**

- When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- Hence, in Python, a *file operation* takes place in the following order.

Open a file

Read or write (perform operation)

Close the file

How to open a file?

- Python has a **built-in function open()** to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
f = open("test.txt")              # open file in current directory
f = open("F:\\PYTH prog\\README.txt")    # specifying full path
```

- We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.
- The default is reading in **text mode**. In this mode, ***we get strings when reading from the file***.
- On the other hand, **binary mode** ***returns bytes*** and this is the mode to be used ***when dealing with non-text files like image or exe files***.

The open function:

- Before you can read or write a file, you have to open it using Python's built-in **open()** function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

file object = open(file_name [, access_mode][, buffering])

The open function:

- **file_name** – The file_name argument is a string value that contains the ***name of the file that you want to access***.
- **access_mode** – The access_mode determines the mode in which the file has to be opened, i.e., **read, write, append**, etc. This is optional parameter and the **default file access mode is read (r)**.
- **buffering** – If the ***buffering value is set to* 0, *no buffering*** takes place. If the **buffering value is 1**, ***line buffering is performed*** while accessing a file. If you specify the buffering value as an integer **greater than 1**, then ***buffering action is performed with the indicated buffer size***. If **negative**, the ***buffer size is the system default***(default behavior).

Opening File in different Modes:

Modes & Description-

(if any question asked on this topic for 4 marks write only modes for text mode and skip modes for binary format)**

- **r**

Opens a file for ***reading only***. The file pointer is placed at the beginning of the file. This is the default mode.

- **rb**

Opens a file for ***reading only in binary format***. The file pointer is placed at the beginning of the file. This is the default mode.

- **r+**

Opens a file for ***both reading and writing***. The file pointer placed at the beginning of the file.

- **rb+**

Opens a file for ***both reading and writing in binary format***. The file pointer placed at the beginning of the file.

- **w**

Opens a file for ***writing only***. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

- **wb**

Opens a file for ***writing only in binary format***. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

- **w+**

Opens a file for ***both writing and reading***. ***Overwrites the existing file if the file exists***. If the file does not exist, creates a new file for reading and writing.

- **wb+**

Opens a file for ***both writing and reading in binary format***. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

- **a**

Opens a file for ***appending***. The ***file pointer is at the end of the file if the file exists***. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

- **ab**

Opens a file for ***appending in binary format***. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

- **a+**

Opens a file for ***both appending and reading***. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

- **ab+**

Opens a file for ***both appending and reading in binary format***. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Example-

```
# For different modes of opening a file

f=open("D:\python programs\Test.txt",'r')  # opening file in r reading mode

print(f.read())

f=open("Test.txt",'w') # Opening file in w writing mode
```

Output-

```
Hello
```

Accessing file contents using standard library functions:

- Once a file is opened and you have one *file* object, you can get various information related to that file.
- List of all attributes related to file object are-

Attribute & Description:

- **file.closed** -Returns true if file is closed, false otherwise.
- **file.mode** -Returns access mode with which file was opened.

- **file.name** -Returns name of the file.
- **file.encoding** -Returns encoding of the file.

Example:

```
fo = open("F:\\2019-2020\\PYTH prog\\README.txt","w)
print("Name of the file: ",fo.name)
print("Closed or not : ",fo.closed)
print("Opening mode : ",fo.mode)
print("File encoding : ",fo.encoding)
```

Output:

```
Name of the file:  F:\2019-2020\PYTH prog\README.txt
Closed or not :  False
Opening mode :  w
File encoding :  cp1252
```

Other File related standard functions-

- **file.flush()** -Flushes the internal buffer.
- **file.isatty()** -It returns true if file has a <tty> (teletypewriter for deaf) attached to it.
- **file.next** -Returns the next line from the last offset.

Reading data from files:

The read() Method-

- The *read()* method ***reads a string from an open file***. It is important to note that Python strings can have binary data. apart from text data.

Syntax

```
fileObject.read([count])
```

Here, passed ***parameter is the number of bytes to be read from the opened file***. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

```
fo = open("F:\\2019-2020\\PYTH prog\\README.txt","r

str=fo.read(5);

print("Read String is : ",str)

print(fo.read(3))

print(fo.read())

# Close opend file

fo.close()
```

Output-

```
Read String is :  Hello

 Wo

rld
```

The readline() Method-

- The *readline()* method ***output the entire line whereas readline(n) outputs at most n bytes of a single line of a file.***
- Once the end of line is reached, we get empty string on further reading.

Syntax

```
fileObject.readline([count])
```

Here, passed ***parameter is the number of bytes of a single line to be read from the opened file***.

Example

```
fo = open("F:\\2019-2020\\PYTH prog\\README.txt","r")
print(fo.readline())
print(fo.readline(3))
print(fo.readline())
print(fo.readline())
print(fo.readline())
```

Output-

```
The readline() method output the entire line whereas readline(n) outputs at most n bytes of a single line of a file.

Onc

e the end of line is reached, we get empty string on further reading.

Syntax

        fileObject.readline([count])
```

The readlines() Method-

- This method maintains a list of each line in the file.

Syntax

```
fileObject.readlines()
```

Example

```
fo = open("F:\\2019-2020\\PYTH prog\\README.txt","r")
print(fo.readlines())
```

Output-

```
['The\xa0readline()\xa0method output the entire line whereas readline(n) outputs at most n bytes of a single line of a file. \n', 'Once the end of line is reached, we get empty string on further reading.\n', 'Syntax\n', '\tfileObject.readline([count]) \n', 'Here, passed parameter is the number of bytes of a single line to be read from the opened file.\n']
```

Writing Files:

There are two ways to write in a file.

- **write(string) :** Inserts the string str1 in a single line in the text file.File_object.write(str1)

e.g.

# Program to write Hello World in a file

```
fo = open("README.txt","w+")
fo.write("Hello World\n")
fo = open("README.txt","r")
print(fo.read())
```

Output:

```
Hello World
```

- **writelines(List) :** For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

```
L = [str1, str2, str3]

File_object.writelines(L)
```

e.g.

# Program to write sequence of strings in a file

```
L=["str1\n","str2\n","str3\n"]

fo = open("README.txt","w+")

fo.writelines(L)

fo = open("README.txt","r")

print(fo.read())
```

Output:

```
str1

str2

str3
```

Closing a File:

When we are done with operations to the file we need to properly close the file.

Closing a file will free up the resources that were tied with the filed and is done using python close() method.

# Program closing a file

```
fo = open("README.txt")

print("Name of the file: ",fo.name)

fo.close()
```

Output:

```
Name of the file:  README.txt
```

File Position: (Not in syllabus read once)

We can change the current file cursor(position) using the seek() method. Similarly the tell() method returns the current position(in number of bytes) of the cursor.

# Program for file position

```
fo = open("README.txt","r")

print(fo.tell())

print(fo.read())

print(fo.tell())

print(fo.read())

print(fo.seek(0))

print(fo.read())
```

Output:

```
0

str1

str2

str3
```

```
18
0
str1
str2
str3
```

Renaming a File:

- Renaming a file in python is done with the help of **rename()** method. To rename a file in python the OS module needs to be imported.
- OS module allows us to perform operating system dependent operations such as ***making a folder, listing contents of a folder etc***.

The rename() method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_filename, new_filename)
```

# Program for renaming files

```
import os
print("Present working directory\n",os.listdir())
os.rename("README.txt","newREADME.txt")
print("Present working directory\n",os.listdir())
```

Output

```
['accept_n_add.py', 'add_two.py', 'area_circle.py', 'armstrong.py', 'array.py', 'check_alphabet.py', 'check_prime.py', 'cmp_list.py', 'combine_dict_values.py', 'compare_bar.py', 'dict_1.py', 'dict_concat.py', 'dict_text_asc.py', 'dict_unique.py', 'dic_asc_desc.py', 'display_exp2.py', 'display_message.py', 'even_no.py',
```

'exp6_1.py', 'exp8_4.py', 'exp9_1.py', 'ex_np.py', 'factorial.py', 'factorial_recursion.py', 'file_op.py', 'file_test.py', 'file_test2.py', 'for2_sample.py', 'for_ex.py', 'for_sample.py', 'hello.py', 'high_3_dict.py', 'incorrect_pswd.py', 'intro_python.docx', 'intro_python.pptx', 'list_concat.py', 'myfile.txt', 'mypkg', 'mypkg1', 'p1.py', 'pack1.py', 'passwd_check.py', 'pattern2_loop.py', 'pattern_loop.py', 'pkg_sample.py', 'prime.py', 'Python datatype.docx', 'python programs', 'pyth_asc_desc_numeric.py', **'README.txt'**, 'sample.py', 'set1_ex.py', 'set3_ex.py', 'set_ex.py', 'set_value_dict.py', 'simple_interest.py', 'sort_ex.py', 'sub_total_avg.py', 'sum_dic.py', 'test.py', 'trial.py', 'tup_ex.py', 'version_pyt.py', 'vowel.py', 'write_file.py']

['accept_n_add.py', 'add_two.py', 'area_circle.py', 'armstrong.py', 'array.py', 'check_alphabet.py', 'check_prime.py', 'cmp_list.py', 'combine_dict_values.py', 'compare_bar.py', 'dict_1.py', 'dict_concat.py', 'dict_text_asc.py', 'dict_unique.py', 'dic_asc_desc.py', 'display_exp2.py', 'display_message.py', 'even_no.py', 'exp6_1.py', 'exp8_4.py', 'exp9_1.py', 'ex_np.py', 'factorial.py', 'factorial_recursion.py', 'file_op.py', 'file_test.py', 'file_test2.py', 'for2_sample.py', 'for_ex.py', 'for_sample.py', 'hello.py', 'high_3_dict.py', 'incorrect_pswd.py', 'intro_python.docx', 'intro_python.pptx', 'list_concat.py', 'myfile.txt', 'mypkg', 'mypkg1', **'newREADME.txt**', 'p1.py', 'pack1.py', 'passwd_check.py', 'pattern2_loop.py', 'pattern_loop.py', 'pkg_sample.py', 'prime.py', 'Python datatype.docx', 'python programs', 'pyth_asc_desc_numeric.py', 'sample.py', 'set1_ex.py', 'set3_ex.py', 'set_ex.py', 'set_value_dict.py', 'simple_interest.py', 'sort_ex.py', 'sub_total_avg.py', 'sum_dic.py', 'test.py', 'trial.py', 'tup_ex.py', 'version_pyt.py', 'vowel.py', 'write_file.py']

Deleting a File:

- We can use the **remove()** method to delete files by supplying the name of the file to be deleted as the argument.
- To remove a file, **the OS module need to be imported**.

Syntax

os.remove(filename)

# Program for removing files

```
import os

print("Present working directory\n",os.listdir())

os.remove(“myfile.txt")

print("Present working directory\n",os.listdir())
```

Output

```
Present working directory

 ['2_area_perimeter_square.py', '2_area_rectangle.py', '2_cylinder.py', '2_kb_gb_tb.py', '2_square_root.py', '2_swap.py', '3_dollar_rupees.py', 'basic_array_numpy.py', 'built_in_package.py', 'class_ex.py', 'class_method.py', 'composition_ex.py', 'constructor_ex.py', 'constr_rect_area.py', 'const_circle.py', 'creating_constructor.py', 'for_ex.py', 'hello.py', 'inherited_methods.py', 'list_cmp.py', 'reshp.py', 'sample.py', 'sets1.py', 'Test.txt', 'trial.py', 'try_except.py', 'tuple_interchange.py', 'user_defined_except.py', 'user_def_example.py']

Present working directory

 ['2_area_perimeter_square.py', '2_area_rectangle.py', '2_cylinder.py', '2_kb_gb_tb.py', '2_square_root.py', '2_swap.py', '3_dollar_rupees.py', 'basic_array_numpy.py', 'built_in_package.py', 'class_ex.py', 'class_method.py', 'composition_ex.py', 'constructor_ex.py', 'constr_rect_area.py', 'const_circle.py', 'creating_constructor.py', 'for_ex.py', 'hello.py', 'inherited_methods.py', 'list_cmp.py', 'reshp.py', 'sample.py', 'sets1.py', 'trial.py', 'try_except.py', 'tuple_interchange.py', 'user_defined_except.py', 'user_def_example.py']
```

Directories in Python:

- If there is large number of files to handle in python program, we can arrange the code within different directories to make things more manageable.

- A directory or folder is a collection of files and sub directories. Python has the OS module, which provides us with many useful methods to work with directories.

Directory Related Standard Functions:

Create New Directory:

- We can make a new directory using the **mkdir()** method.
- This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

e.g.

```
import os

os.mkdir("testdir")
```

Get Current Directory

- We can get the present working directory using the **getcwd()** method.
- This method returns the current working directory in the form of a string.

e.g.

```
import os

print(os.getcwd())
```

Changing Directory:

- We can change the current working directory using the **chdir()** method.
- The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements.

Example-

```
import os

print(os.getcwd())

os.chdir("F:\\2019-2020")

print(os.getcwd())
```

Output:

```
F:\2019-2020\PYTH prog

F:\2019-2020
```

List Directories and Files:

- All files and sub directories inside a directory can be known using the **listdir()** method.
- This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

e.g.

```
import os

print(os.listdir())
```

Removing Directory:

- A file can be removed (deleted) using the remove() method.
- Similarly, the **rmdir()** method removes an empty directory.

e.g.

```
import os
os.rmdir("testdir")
```

PROGRAMS

# Program to create a simple file and write some content in it

```
print("Enter 'x' for exit")
f=input("Enter file name to create and write content: ")
if f=='x':
    exit()
else:
    c=open(f,'w+')
print("\n The file ",f," created successfully")
print("Enter sentences to write in the file:")
sent1=str(input())
c.write(sent1)
c=open(f,'r')
print(c.read())
print("\n Content successfully placed in the file")
```

Output:

```
Enter file name to create and write content: file123


 The file  file123  created successfully

Enter sentences to write in the file:

Hello

Hello


 Content successfully placed in the file
```

# Program to open a file in write mode and append some content at the end of a file

```
print("Enter 'x' for exit")

f=input("Enter file name to append content: ")

if f=='x':

    exit()

else:

    c=open(f,'a+')


print("Enter sentences to append in the file:")

sent1=str(input())

c.write("\n")
```

```
c.write(sent1)

c=open(f,'r')

print(c.read())

print("\n Content appended in the file")
```

Output:

```
Enter 'x' for exit

Enter file name to append content: test123

Enter sentences to append in the file:

World


World


 Content appended in the file
```

# Program to open a file in read mode and print number of occurences of characters

```
count = 0

char = input("ENTER CHARACTER : ")

file = open("Test.txt","r")

for i in file:

    for c in i:

        if c == char:
```

```
        count = count + 1
print("THE CHARACTER {} IS FOUND {} TIMES IN THE TEXT FILE".format(char,count))
```

Output:

```
ENTER CHARACTER : o
THE CHARACTER o IS FOUND 2 TIMES IN THE TEXT FILE
```

Exception Handling:

Introduction-

When we execute a python program there may be a few uncertain conditions which occur, known as errors. There are three types of errors.

Compile Time Error- Occurs at the time of compilation due to violation of syntax rules like missing of a colon.

Run Time Error- Occurs during the runtime of a program due to wrong input submitted to the program by user.

Logical Errors- Occurs due to wrong logic written in the program.

Errors occurs at runtime are known as exception.

- ***An exception is an event which occurs during the execution of a program that disrupts the normal flow of the program's instructions***.
- ***An exception is also called as runtime error that can halt the execution of the program***.
- Python has many ***built-in exceptions*** which ***forces your program to output an error when something in it goes wrong***.

- When these exceptions occur, ***it causes the current process to stop*** and ***passes it to the calling process until it is handled***. If not handled, our ***program will crash***.
- For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is spit out and our program come to a sudden, unexpected halt.
- For handling exception in python, ***the exception handler block needs to be written which consists of set of statements*** that need to be ***executed according to raised exception***.
- There are three blocks that are used in the exception handling process, namely ***try, except and finally***.

try Block- A set of statements that may cause error during runtime are to be written in the try block.

except Block- It is written to display the execution details to the user when certain exception occurs in the program. The except block executed only when a certain type as exception occurs in the execution statements written in the try block.

finally Block- This is the last block written while writing an exception handler in the program which indicates the set of statements that are used to clean up the resources used by the program.

Exception Handling in Python:

‘try : except’ statement-

- In Python**, exceptions** can be **handled** using a **try** statement.
- A critical operation which can raise ***exception is placed inside the try clause*** and the ***code that handles exception is written in except clause***.
- It is up to us, what operations we perform once we have caught the exception. Following is the example.

# Program for execption handling only using try:except block

```
import sys   # import module sys to get the type of exception

randomList = ['a', 0, 2]

for entry in randomList:

    try:

        print("The entry is", entry)

        r = 1/int(entry)

        break

    except:

        print(sys.exc_info()[0],"occured.")

        print("Next entry.")

print()

print("The reciprocal of",entry,"is",r)
```

Output:

```
The entry is a
```

```
<class 'ValueError'> occured.

Next entry.

The entry is 0

<class 'ZeroDivisionError'> occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5
```

Catching Specific Exceptions in Python

- In the above example, we did not mention any exception in the except clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.
- A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.
- We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

Example pseudocode-

```
try:

# do something

pass

 exceptValueError:
# handleValueError exception
pass
 except (TypeError, ZeroDivisionError):
# handle multiple exceptions
# TypeError and ZeroDivisionError
pass
 except:
# handle all other exceptions
pass
```

Example:

# Program to handle multiple errors with one except statement

```
try :
    a = 3
    if a < 4 :

        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print("Value of b = ", b)

# note that braces () are necessary here for multiple exceptions
```

```
except(ZeroDivisionError, NameError):
    print("\nError Occurred and Handled")
```

Output:

```
Error Occurred and Handled
```

Output: ***(When value of a is set to 5 in above same program)***

```
Error Occurred and Handled
```

# Program for use of try except statement

```
n=10
m=0
try:
    n/m
except ZeroDivisionError:
    print("Divide by zero error")
else:
    print(n/m)
```

Output:

```
Divide by zero error
```

try-except statement with no exception:

- We can use try-except clause with no exception.
- All types of exceptions that occur are caught by the try-except statement.
- However, because it catches all exceptions, the programmer cannot identify the root cause of a problem that may occur. Hence this type of programming approach is not considered good.

```
# Program for try-except statement with no exception
while True:
    try:
        a=int(input("Enter an integer: "))
        div=10/a
        break
    except:
        print("Error occured")
        print("Please enter valid value")
        print()
print("Division is: ",div)
```

Output:

```
Enter an integer: a
Error occured
Please enter valid value
Enter an integer: 3.5
Error occured
```

```
Please enter valid value
Enter an integer: 0
Error occured
Please enter valid value
Enter an integer: 5
Division is:  2.0
```

try..finally statement:

- The try statement in Python can have an **optional finally clause**. ***This clause is executed no matter what, and is generally used to release external resources***.
- For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).
- In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.
- Here is an example of file operations to illustrate this.

Example

```
try:
   f = open("test.txt",encoding = 'utf-8')
# perform file operations
finally:
f.close()
```

- This type of construct makes sure the file is closed even if an exception occurs

```
# Program to check for ZeroDivisionError Exception
x=int(input("Enter first value: "))
y=int(input("Enter second value: "))
try:
   result=x/y
except ZeroDivisionError:
   print("Divison by Zero")
else:
   print("Result is: ",result)
finally:
   print("Execute finally clause")
```

Output:

```
Enter first value: 5
Enter second value: 0
Divison by Zero
Execute finally clause
Enter first value: 10
Enter second value: 5
Result is:  2.0
Execute finally clause
```

raise statement:

- In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise.
- We can also optionally pass in value to the exception to clarify why that exception was raised.

Example:

# Program using raise statement to throw exception if age is less that 18 condition occurs

```
while True:
    try:
        age=int(input("Enter your age for election: "))
        if age<18:
            raise Exception
        else:
            print("You are eligible for election")
            break
    except Exception:
        print("This value is too small, try again")
```

Output:

```
Enter your age for election: 15
This value is too small, try again
Enter your age for election: 20
You are eligible for election
```

User Defined Exception:

- Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.
- However, sometimes you may ***need to create custom exceptions that serve your purpose***.
- In Python, ***users can define such exceptions*** by ***creating a new class***. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived from this class.
- User can also create and raise his/her own exception known as user defined exception.

Example:

# Program to raise user defined exception if age is less than 18

```
#define python user defined exception
class error(Exception):
    """Base class for other exceptions"""  #empty class
    pass
class AgeSmallException(error):
    """Raised when the input value is too small""" #empty class
    pass
# main program
while True:
    try:
        age=int(input("Enter your age for election: "))
```

```
if age<18:
            raise AgeSmallException
        else:
            print("You are eligible for election")
            break
    except AgeSmallException:
        print("This value is too small , try again")
        print()
```

Output:

```
Enter your age for election: 15
This value is too small , try again
Enter your age for election: 20
You are eligible for election
```

Program to guess the number entered by the user matches with the defined number.

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass
class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass
class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass
# main program user guesses a number until he gets it right you need to guess this number
number = 10
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num< number:
            raise ValueTooSmallError
        elif i_num> number:
            raise ValueTooLargeError
        elif i_num==number:
            break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()
print("Correct number")
```

Output:

```
Enter a number: 5

This value is too small, try again!


Enter a number: 15

This value is too large, try again!


Enter a number: 10

Correct number
```