UNIT-II

PART-I
FUNCTIONS

TOPICS:-

1. Defining a function
2. Calling a function
3. returning multiple values from a function
4. functions are first class objects
5. formal and actual arguments
6. positional arguments
7. recursive functions

TOPIC 2.1.1 & 2.1.2 DEFINING and CALLING A FUNCTION

A function in programming is a block of code organized by a set of rules to accomplish a specific task.

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Types of Functions:-

There are many categories based on which we can categorize the functions. This categorization is based on who created it.

1. Pre-defined or built-in functions
2. User-defined functions

Predefined or built-in functions: The functions which come installed along with python software are called predefined or built-in functions. We have covered some inbuilt functions in examples of earlier chapters. Some of them are id( ), type( ), input( ), print( ) etc.
User-defined functions: The functions which are defined by the developer as per the requirement are called user-defined functions. In this chapter, we shall concentrate on these kinds of functions that are user-defined.

A user can define function in four ways:-

1. Function without argument and without return value

2. Function with argument and with return value

3. Function without argument and with return value

4. Function with argument and without return value

How to Create and call a function in Python?

From creating a function to using it, these are the two things that are done.

1. Defining function
2. Calling function

Defining a function

A function is defined by using the block keyword **"def"** , followed by the function's name, followed by a set of parentheses which hold any parameters the function will take and ending with a colon. Next follows the **block of statements** that are part of this function. Functions may return a value to the caller, using the keyword- 'return'

To summarize the definition contains – def keyword, name for the function, parentheses, parameters(optional), colon (:), body, return(optional).

Syntax:-

```
def function_name(parameters):
   """docstring"""
   statement1
   statement2
   ...
   ...
   return [expr]
```

Example:-

```
def display( ):
        print("welcome to function")
```

In the above demo1.py example we defined a function with the name 'display' with a print statement in it. But when we execute the demo1.py it will not display any output because the function is not called. Hence, function calling is also important along with function definition. After defining a function, we need to call to execute the function. While calling the function, we have to call with the same name of the function which we used while defining it, otherwise we will get an error.

Example:-

```
def display( ):
    print("welcome to function")
display( )
```

Output:-

```
welcome to function
```

Function without argument and without return value

```
# defining a function
def sum():
  a,b=20,30
  print("Sum of two values=", (a+b))
# calling function
sum()
```

Output:-

```
Sum of two values= 50
```

Function with argument and with return value

```
# defining a function

def sum(a,b):

  return(a+b)

# calling function

result=sum(20,30)

print('sum of two values = ' ,result)
```

Output:-

```
 sum of two values =  50
```

Function without argument and with return value

```
# defining a function

def sum():

  a,b=20,30

  return(a+b)
```

```
# calling function

result=sum()

print('sum of two values = ' ,result)
```

Output:-

```
sum of two values =  50
```

Function with argument and without return value

```
# defining a function

def sum(a,b):

  result=a+b

  print('sum of two values = ',result)

# calling function

sum(20,30)
```

Output:-

```
sum of two values =  50
```

TOPIC 2.1.3 RETURNING MULTIPLE VALUES FROM A FUNCTION

In python, a function can return multiple values. If a function is returning multiple values then the same should be handled at the function calling statement

Example: Define a function that can return multiple values

```
def m1(a, b):

  c = a+b

  d = a-b
  return c, d

#calling function

x, y = m1(10, 5)

print("sum of a and b: ", x)

print("subtraction of a and b: ", y)
```

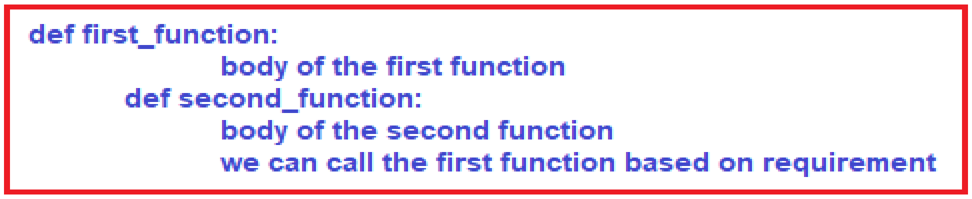
Output:-

```
sum of a and b:  15

subtraction of a and b:  5
```

Note:- The function can call another function in Python:

It is also possible in python that a function can call another function. The syntax to call a function from another function is given below.



```
def first_function:
                body of the first function
        def second_function:
                body of the second function
                we can call the first function based on requirement
```

Example:-

```
def m1( ):

  print("first function information")

def m2( ):

  print("second function information")

  m1( )

m2( )
```

Output:-

```
second function information

first function information
```

TOPIC 2.1.4 FUNCTIONS AS FIRST CLASS OBJECTS

Functions are considered as first-class objects. In python, below things are possible to

1. Assign a function to variables
2. Pass function as a parameter to another function
3. Define one function inside another function
4. The function can return another function

Assigning a function to a variable in Python:

Example: Assign a function to a variable

```
def add( ):
   print("We assigned function to variable")
#Assign function to variable
sum=add
#calling function
sum( )
```

Output:-

```
We assigned function to variable
```

Pass function as a parameter to another function in Python

Example: Pass function as a parameter to another function

```
def display(x):
  print("This is display function")
def message( ):
  print("This is message function")
# calling function
display(message( ))
```

Output:-

```
This is message function
This is display function
```

Define one function inside another function in Python:
Example: function inside another function

```
def first( ):
  print("This is outer function")
  def second( ):
     print("this is inner function")
  second( )
#calling outer function
first( )
```

Output:-

```
This is outer function
this is inner function
```

The function can return another function in Python:
Example: function can return another function

```
def first( ):
  def second( ):
     print("This function is return type to outer function")
  return second
x=first( )
x( )
```

Output:-

```
This function is return type to outer function
```

TOPIC 2.1.5: ACTUAL AND FORMAL ARGUMENTS

Actual Parameter	Formal Parameter
The arguments that are passed in a function call are called actual arguments.	The formal arguments are the parameters/arguments in a function declaration.
Data type not required. But data type should match with corresponding data type of formal parameters.	Data types needs to be mentioned.
Actual parameters are the parameters which you specify when you call the Functions or Procedures.	A formal parameter is a parameter which you specify when you define the function.
The actual parameters are passed by the calling function.	The formal parameters are in the called function.

Ex:
Void main()
{ Int a,b;
a= sum(4,5); //function call
}
// 4,5 are actual parameter
	Ex:
	Int sum(int a, int b)
	{ Int s;
	s=a+b;
	return(s);
	}
	//a&b are formal parameter

TOPIC 2.1.6 TYPES OF ARGUMENTS

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Keyword Arguments:-

In python when functions are called, the values passed are assigned to the arguments based on their positions, which means the first value passed will be assigned to the first argument. This position based distribution sometimes leaves the user confused because the user never knows the format in which function was defined.

If user wants to change order of arguments from function call to function definition , then user has to define keyword arguments.

Example Program:-

```
def person(name,age,height,weight):
   print(name,age,height,weight)
person(height=5.8,name='hari',weight=65.8,age=21) # keyword arguments
```

Output:-

```
hari 21 5.8 65.8
```

Default Arguments:-

In some cases, we may want to build a function that has some default values for parameters in case if the user doesn’t provide values for them.

In such cases, we can use the default arguments. A default argument assumes a default value if a value is not supplied as an argument while calling the function.

This can be done using the assignment operator ‘=’. We can provide default values for one or more parameters.

Note:- if user forgotten to define arguments in function call, then user can define default arguments in function definition. But always default arguments should follow non default arguments.

Example:-

```
def person(age=21,weight=65.8,height,name):
   print(name,age,height,weight)
person(height=5.8,name='hari')
```

Output:- error will occur

```
File "<ipython-input-24-9b036b9fb4f8>", line 1
  def person(age=21,weight=65.8,height,name):
         ^
SyntaxError: non-default argument follows default argument
```

Example:-

```
def person(height,name,age=21,weight=65.8):
   print(name,age,height,weight)
person(height=5.8,name='hari')
```

Output:-

```
hari 21 5.8 65.8
```

Variable Length Arguments:-

Sometimes, we may not know the possible number of arguments that the function can have while defining the function. If the number of arguments needed is unknown to us, then we can define the function with a flexible number of arguments which are called variable length arguments.

We can create functions that take any amount of arguments using a * symbol before the keyword inside the parentheses. These are also called variable-length arguments.

Example Program:-

```
def add(*args):
   total=0
   for a in args:
      total+=a
   print(total)
add(3,5)
add(3,4,5,1,2)
```

Output:-

```
8
15
```

Positional Arguments:-

Positional arguments are arguments that need to be included in the proper position or order. The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second and the third positional argument listed third, etc.

When we pass the values during the function call, they are assigned to the respective arguments according to their position. For example, if a function is defined as

```
def demo(name, age):
```

and we are calling the function like :

```
demo("Keshav", "35")
```

then the value “Keshav” is assigned to the argument name and the value “35” is assigned to the argument age. Such arguments are called **positional arguments**.

In other words, a positional argument is an argument that has a given position in the list of arguments passed into your function.

Example Program:-

```
def abc(a,b,c=2):
   return a+b+c
x=abc(1,2) #both positional argument and c is default
print(x)
y=abc(2, b=3) # positional, named and again c is default
print(y)
z=abc(a=2,b=4) # both named argument and c is default
print(z)
```

Output:-

```
5
7
8
```

TOPIC 2.1.7 RECURISVE FUNCTION

A recursive function is a function that calls itself until it doesn’t.

And a recursive function always has a condition that stops calling itself.

Example Program:-

```
def factorial(x):
   """This is a recursive function
   to find the factorial of an integer"""
   if x == 1:
       return 1
   else:
       return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:-

```
The factorial of 3 is 6
```

-----******-----

UNIT-II
PART-2
EXCEPTIONS

Topics:-

1. Errors in a Python program
2. Exceptions
3. Exception handling
4. Types of exceptions
5. The except block
6. The assert statement
7. User-defined exceptions.

TOPIC 2.2.1 ERRORS IN PYTHON

We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

1. Syntax errors
2. Logical errors
3. Run time errors(Exceptions)

Python Syntax Errors

Error caused by not following the proper structure (syntax) of the language is called **syntax error** or **parsing error**.

Example:-

```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

As shown in the example, an arrow indicates where the parser ran into the syntax error.

We can notice here that a colon `:` is missing in the `if` statement.

Python Logical Errors

Logical errors are the most difficult to fix. They occur when the program runs without crashing, **but produces an incorrect result**. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred.

Python Runtime Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.
For instance, they occur when we try to open a file(for reading) that does not exist (`FileNotFoundError`), try to divide a number by zero (`ZeroDivisionError`), or try to import a module that does not exist (`ImportError`).

TOPIC 2.2.2, 2.2.3 & 2.2.5 EXCEPTION HANDLING & EXCEPT BLOCK IN PYTHON

Exception :-

An unwanted/unexpeceted event that disturbs the normal flow of the program.

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

The problem without handling exceptions

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

Example

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)


#other code:
print("Hi I am other part of the program")
```

Output:

```
Enter a:10

Enter b:0

Traceback (most recent call last):

  File "exception-test.py", line 3, in <module>

    c = a/b;

ZeroDivisionError: division by zero
```

Exception handling statements in python

1. The try-expect statement

2. The try-expect-else statement

3. The except statement with no exception

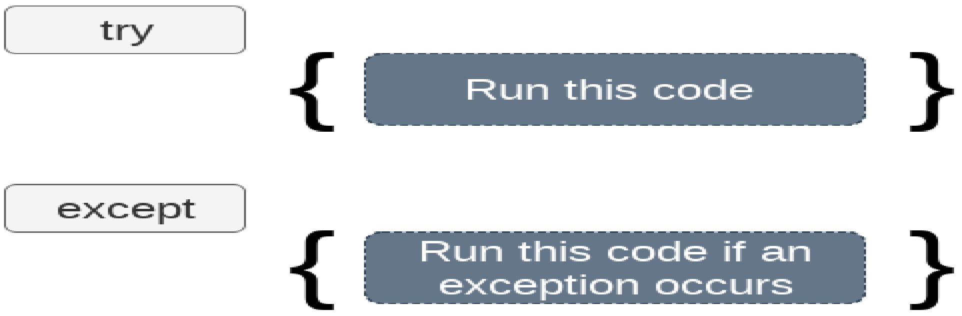
4. Multiple exceptions

5. The except statement using with exception variable

6. The try...finally block

1. The try-expect statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block. The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax

```
try:
   #block of code

except Exception1:
   #block of code

except Exception2:
   #block of code

#other code
```

Example 1

```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b
```

```
except:
  print("Can't divide with zero")
```

Output:

```
Enter a:10

Enter b:0

Can't divide with zero
```

2. The try-expect-else statement

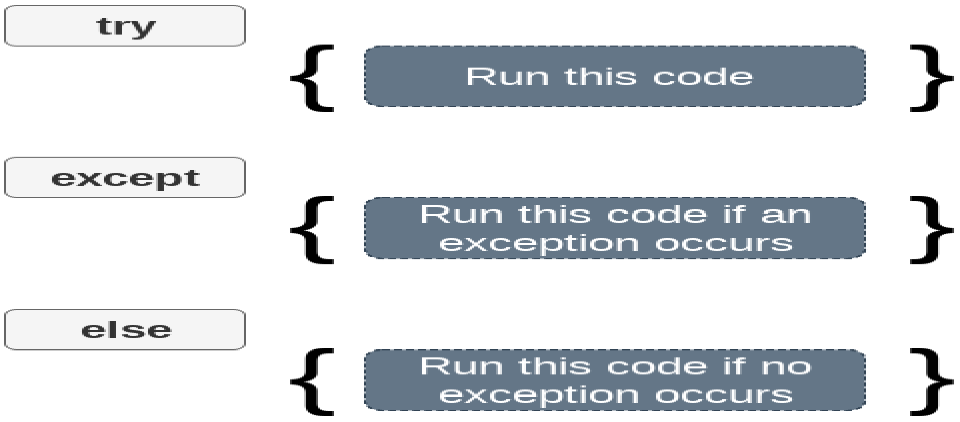
We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

Syntax:-

```
try:
   #block of code

except Exception1:
   #block of code

else:
   #this code executes if no except block is executed
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs

Example

```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b
   print("a/b = %d"%c)
```

```
# Using Exception with except statement. If we print(Exception) it will return exception cla
ss
except Exception:
   print("can't divide by zero")
   print(Exception)
else:
   print("Hi I am else block")
```

Output:

```
Enter a:10

Enter b:0

can't divide by zero

<class 'Exception'>
```

3. The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement.

Example

```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b;
   print("a/b = %d"%c)
except:
   print("can't divide by zero")
else:
   print("Hi I am else block")
```

4. The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
```

```
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10

Enter b:0

can't divide by zero

division by zero
```

5. <u>Declaring Multiple Exceptions</u>

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

Syntax

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
```

Consider the following example.

```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Output:

Arithmetic Exception

6. The try...finally block

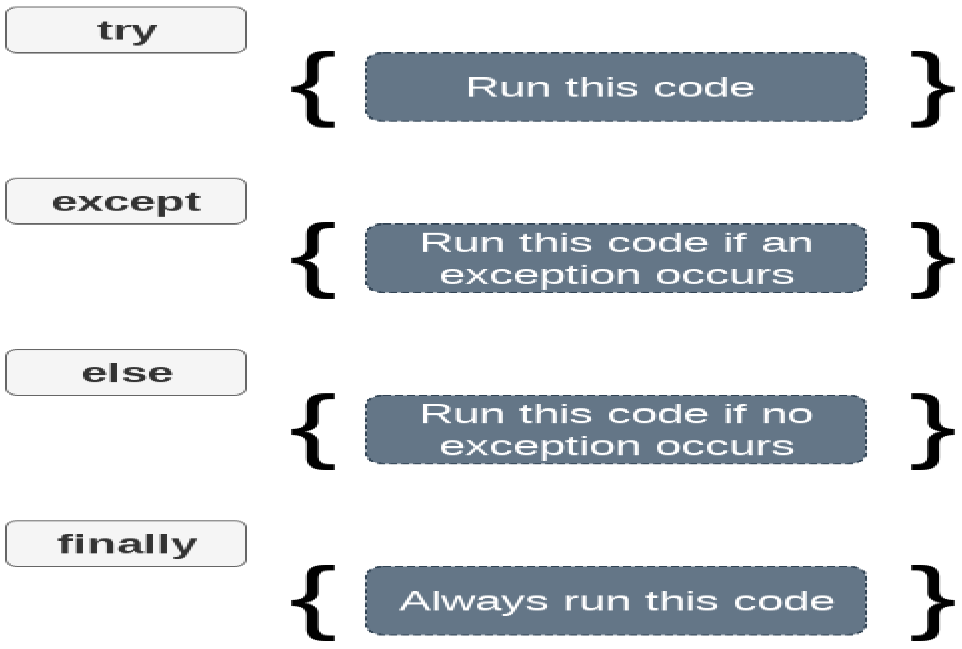
Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax

```
try:
   # block of code
   # this may throw an exception
finally:
   # block of code
   # this will always be executed
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

Example

try:

```
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Output:

```
file closed

Error
```

TOPIC 2.2.3 TYPES OF EXCEPTIONS

There are two types of exceptions in Python.

- Built-in Exceptions
- User-Defined Exceptions

Built-in Exceptions:-

Built-in exceptions are the standard exceptions defined in Python.

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

So, these are some of the built-in exceptions in Python. These all are derived from a base exception class. Every exception is derived from a base exception class.

User-Defined Exceptions:-

As the name goes by the exceptions defined by a user are called User-Defined Exceptions.

Sometimes in a program, we need create custom exceptions to serve our own purposes. Python also allows us to create our own exceptions by deriving classes from the standard built-in exceptions.

TOPIC 2.2.6 ASSERT STATEMENT

In Python, the `assert` statement is used to continue the execute if the given condition evaluates to True. If the assert condition evaluates to False, then it raises the `AssertionError` exception with the specified error message.

Syntax

```
assert condition [, Error Message]
```

The following example demonstrates a simple assert statement.

Example: assert

```
x = 10
assert x > 0
print('x is a positive number.')
```

Output

```
x is a positive number.
```

In the above example, the assert condition, `x > 0` evalutes to be True, so it will continue to execute the next statement without any error.

The assert statement can optionally include an error message string, which gets displayed along with the `AssertionError`. Consider the following assert statement with the error message.

Example: Assert Statement with Error Message

```
x = 0
assert x > 0, 'Only positive numbers are allowed'
print('x is a positive number.')
```

Output

```
Traceback (most recent call last):
    assert x > 0, 'Only positive numbers are allowed'
AssertionError: Only positive numbers are allowed
```

Above, `x=0`, so the assert condition `x > 0` becomes False, and so it will raise the `AssertionError` with the specified message 'Only positive numbers are allowed'. It does not execute `print('x is a positive number.')` statement.

TOPIC 2.2.7 USER DEFINED EXCEPTIONS

How to define a Custom Exception(user defined exception) in Python?

We can create user-defined exceptions by creating a new class in Python. Directly or indirectly the custom exception class has to be derived from the base exception class. Most of the built-in exceptions are derived from this base class to enforce their exceptions.

Python Exception Handling: Example to illustrate User-defined exception

Let's create a user-defined exception where the program will ask the user to input a number again and again until the user enters the correct stored number.

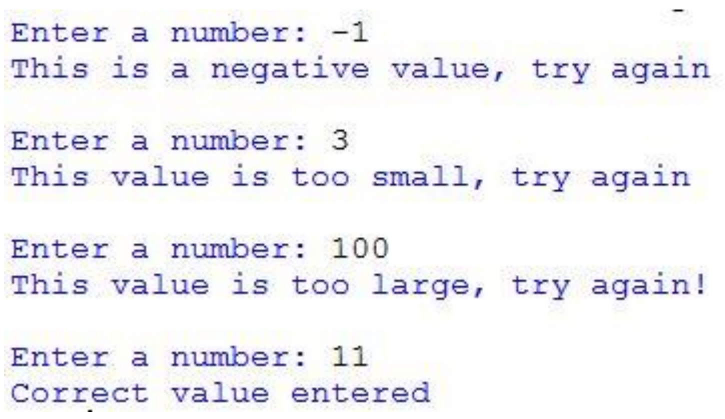
Here we will demonstrate how to raise an user-defined exception and catch errors in a program.

Example Program:-

```
#User-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass
#Define class for NegativeValueError
class NegativeValueError(Error):
  """Raised when the input is negative"""
  pass
#Define class for ValueTooSmallError
class ValueTooSmallError(Error):
   """Raised when the value is too small"""
  pass
#Define class for ValueTooLargeError
class ValueTooLargeError(Error):
  """Raised when the value is too large"""
  pass
#main program
#Takes input till the user inputs correct value
#Stored number
number = 11
while True:
  try:
    num = int(input("Enter a number: "))
    if num < 0:
      raise NegativeValueError
    elif num < number:
      raise ValueTooSmallError
    elif num > number:
      raise ValueTooLargeError
    break
  except NegativeValueError:
    print("This is a negative value, try again")
    print("")
  except ValueTooSmallError:
    print("This value is too small, try again")
    print("")
  except ValueTooLargeError:
    print("This value is too large, try again!")
    print("")
print("Correct value entered")
```

Here in this program, we have defined a base class Error from which three other exceptions `NegativeValueError`, `ValueTooSmallError` and `ValueTooLargeError` are derived.

Let’s try this program with multiple test values.



```
Enter a number: -1
This is a negative value, try again

Enter a number: 3
This value is too small, try again

Enter a number: 100
This value is too large, try again!

Enter a number: 11
Correct value entered
```

------***** THE END *****------