python

UNIT – III

Contents

- **Modules**

  Importing module, Math module, Random module, Packages,

- **Exception Handling**

  Exception, Exception Handling, except clause, Try? finally clause User Defined Exceptions

- A module is a file consisting of Python code. A module can define functions, classes, and variables. A module can also include runnable code.
- Grouping related code into a module makes the code easier to understand and use.
- Any Python source file as a module by executing an import statement in some other Python source file**.**
- import has the following syntax:

  import module1[, module2[,... moduleN]

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- The search path is a list of directories that the interpreter searches before importing a module.

1. import module name

Ex

hello.py

```
def                greet(str):
            print( “Hello”, str)
```

test.py

```
import   hello                    # imports the  module  hello.py
hello. greet(“python”)
```

Output:

```
hello python
```

2. from...import Statement

- ➢ Python's from statement lets you import specific attributes from a module into the current namespace.
- ➢ The from...import has the following syntax:

```
from  modname  import  name1[, name2[, ... nameN]]
```

EX calc.py

```
def  add(x,y):
        return(x+y)
def  sub(x,y):
        return(x-y)
def  mul(x,y):
        return(x*y)
def  div(x,y):
        return(x/y)
```

EX test.py

```
from  calc  import  add, sub
print(add(10,20))
print(sub(10,20))
```

Output

```
30
-10
```

3. from...import * Statement:

- ➢ It is also possible to import all names from a module into the current namespace by using the following import statement:

from modname import *

- ➢ This provides an easy way to import all the items from a module into the current namespace.

Ex: test.py

```
from calc import *
print(add(10,20))
print(sub(10,20))
print(mul(10,20))
print(div(10,20))
```

output

```
30
-10
200
0.5
```

4. using alias name

- It is possible to modify the names of modules and their functions within Python by using the as keyword.
- You may want to change a name because you have already used the same name for something else in your program, another module you have imported also uses that name, or you may want to abbreviate a longer name that you are using a lot.

Syntax

import [module] **as** [another_name]

Ex **test.py**

```
import  calc  as  c
print(c.add(10,20))
print(c.sub(10,20))
print(c.mul(10,20))
print(c.div(10,20))
```

output

```
30
-10
200
0.5
```

dir( )

- ➢ The dir() built-in function returns a sorted list of strings containing the names defined by a module.
- ➢ The list contains the names of all the modules, variables and functions that are defined in a module.

Ex

```
import   calc
print(dir(calc))
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add', 'div', 'mul', 'sub']
```

Random module

- The random module provides access to functions that support many operations.
- The most important thing is that it allows you to generate random numbers.

Random Functions

randint

- Generates a random integer
- Randint accepts two parameters: a lowest and a highest number.

Ex

```
import random          # generates a number between 2 to 5
random.randint(1,5)
```

choice

- *Generates a* random value from the sequence sequence.

EX

```
random. choice( ['red', 'black', 'green'] )
>>> 'green'
```

shuffle

The shuffle function, shuffles the elements in list in place, so they are in a random order.

Ex

```
l=[i for i in range(10)]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random. shuffle(l)
>>> l
[7, 6, 9, 0, 5, 2, 4, 1, 8, 3]
```

Randrange

Generate a randomly selected element from range(start, stop, step)

Syntax: random.randrange(start, stop[, step])

Ex

```
for i in range(3):
        print(random.randrange(0,100,5))
45
67
83
```

random

This number i used to generate a **float random number less than 1 and greater to 0. or equal** .

Ex
```
import  random
for i in range(2):
        print(random.random())
```
```
0.9844129726131 41
0.48528597775151017
```

Uniform This function is used to generate a floating point random number between the numbers mentioned in its arguments. It takes two arguments, lower limit and upper limit(not included in generation).

Ex :
```
import random
for i in range(2):
        print(random.uniform(5,10))
```

```
8.02302579895293
6.4517099857187175
```

Example program to generate OTP

```
from random import *
for i in range(5):
    print(randint(1,9),randint(0,9),randint(0,9),
        randint(0,9),randint(0,9),randint(0,9),sep='')
```

Example program to generate password

```
import random,string
pwd_chrs=string.printable+string.punctuation
for i in range(8):
    print(random.choice(pwd_chrs),end='')
```

Math Module

The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using import math.

Functions in math module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)

isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

packages

- A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional __init__.py file.
- Each package in Python is a directory which **MUST** contain a special file called __init__.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.
- when you import a package, only variables/functions/classes in the __init__.py file of that package are directly visible, not sub-packages or modules.

Steps to Create a Python Package

- Create a directory and give it your package's name.
- Put your classes in it.
- Create a **__init__.py** file in the directory

The __init__.py file is necessary because with this file, Python will know that this directory is a Python package directory other than an ordinary directory

1.Create a directory PacItf4
2. Put modules cal , area and some other files in PacItf4
3. Create a **__init__.py** file in the PacItF4

```
import PacItf4.cal
import PacItf4.area
print(PacItf4.cal.add(4,5))
print(PacItf4.area.aofs(4))
```

Errors in a python program: In general, we can classify errors in a program into one of these three types

- Compile- time errors
- Runtime errors
- Logical errors

Compile – Time Errors

These are syntactical errors found in the code, due to which a program fails to compile, for example forgetting a colon

Ex:

```
x=1
if x==1
        print("where's is colon?")
```

Runtime Errors

When PVM (python virtual matchine) can not execute the byte code, it flags runtime error. For example insufficient memory to store something.

Ex:

```
def concat(a,b):
        Print(a+b)
concat("python",1)
```

Output

```
Type Error
```

Logical Errors

These errors depict flaws in the logic of the program. The programmer might be using a wrong formula or the design of the program itself is wrong.

EX

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x+y/2
print ('The average of the two numbers you have entered is:',z)
```

Output:

```
Enter a number: 3
Enter a number: 4
The average of the two numbers you have entered is: 5.0
```

Exception

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it can't cope with, it raises an exception.
- An exception is a Python object that represents a runtime error which can be handled by the programmer.
- The exceptions which are already available in python are called built- in exceptions.
- The Base class for all the exceptions is ‘BaseException’ class, the sub class Exception is derived.
- From Exception class, the sub classes ‘standardError’ and ‘Warning’ are derived.
- A programmer can also create his own exceptions are called ‘user-defined’ exceptions.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

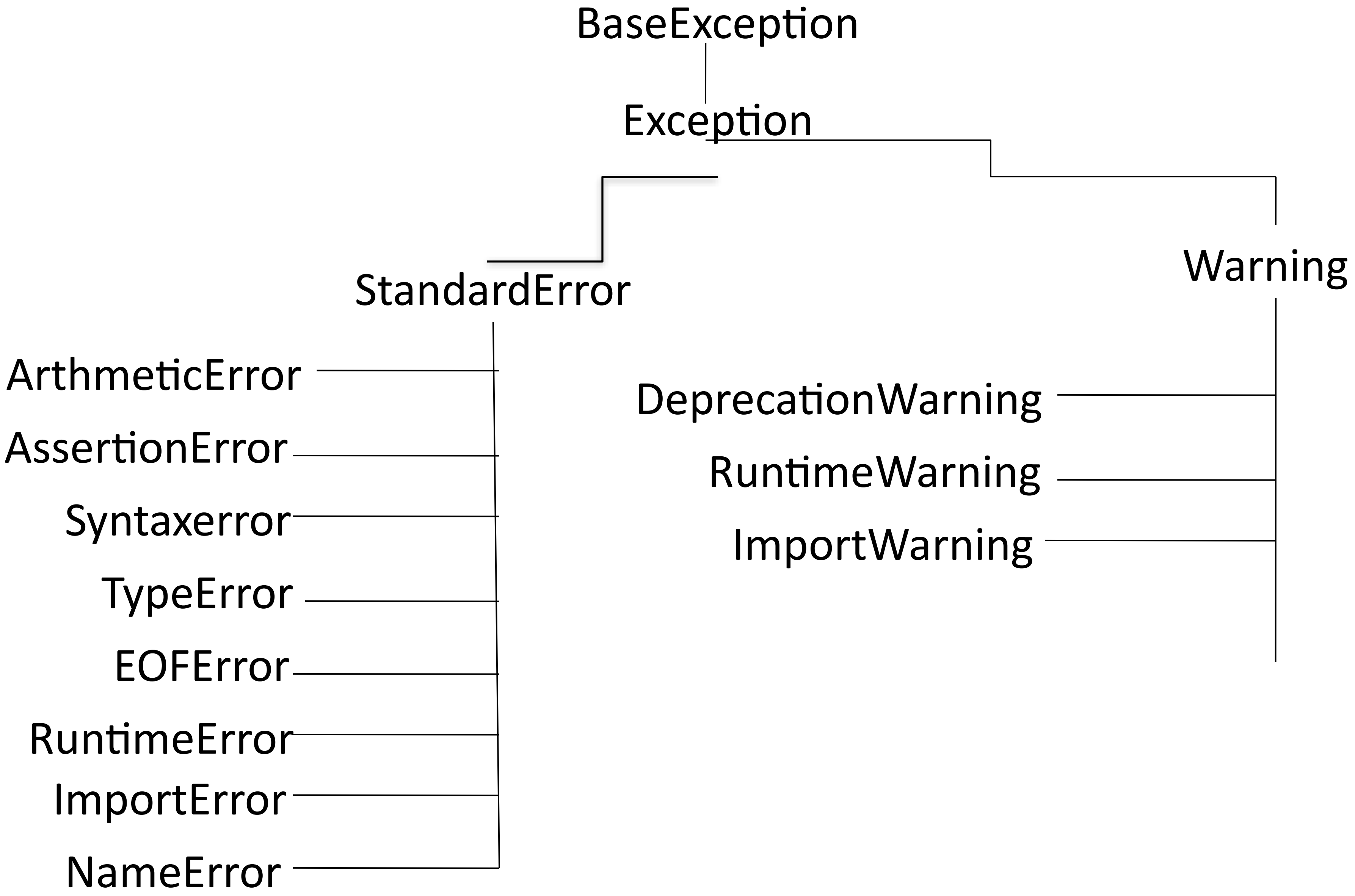


Figure: Exception classes in python

Exception Handling

- ➢ The purpose of handling errors is to make the program robust.
- ➢ When there is an error in the program, it will display an appropriate message to the user and continue execution.
- ➢ To handle exceptions, the programmer should perform the following three steps.

Step 1: Where there may be a possibility of exceptions should be written inside a try block.

Syntax:

```
try:
        statements
```

Step 2: The programmer should write the 'except' block where he should display the exception details to the user.

Syntax:

```
except  exceptionname:
        statements
```

Step 3: The programmer should perform clean up actions like closing the files and terminating any other processes which are running.
The statements inside the finally block are executed irrespective of whether there is an exception or not.
Syntax: finally:

```
        statements
```

The complete exception handling syntax will be in the following format.

```
        try:
                statements
        except Exception1:
                handler1
        except Exception2:
                handler2
        else:
                statements
        finally:
                statements
```

- A single try block can be followed by several except blocks.
- Multiple except blocks can be used to handle multiple exceptions.
- We cannot write except blocks without a try block.
- We can write a try block without any except blocks.
- else block and finally blocks are not compulsory.
- When there is no exception, else block is executed after try block.
- finally block is always executed.

Example program

```
a=int(input("enter a"))
b=int(input("enter b"))
print("division=",a/b)
print("add=",a+b)
print("mul=",a*b)
print("operations done")
```

Example program for Exception handling

```
a=int(input("enter a"))
b=int(input("enter b"))
try:
    print("division=",a/b)
except ZeroDivisionError:
    print(“zerodivision error")
print("add=",a+b)
print("mul=",a*b)
print("operations done")
```

Example program for Exception handling complete syntax

```
a=int(input("enter a"))
b=int(input("enter b"))
try:
    print("division=",a/b)
except ZeroDivisionError:
    print("denominator should not be o")
else:
    print("there is no exception")
finally:
    print("this will execute always")
print("operations done")
```

Types of Exceptions

ArithmeticError - Represents any type of exception. All exceptions are sub classes of this class.

AssertionError – raised when an assert statement gives error.

AttributeError – raised when an attribute reference or assignment fails.

EOFError – raised when input() function reaches end of file condition without reading any data.

FloatingPointError – raised when a floating point operation fails.

GeneratorExit – raised when generator's close method is called.

IOError – raised when an input or output operation failed. It raises when the file opened is not found or when writing data disk is full.
ImportError – raised when an import statement fails to find the module being imported.

IndexError – raised when a sequence index or subscript is out of range.

KeyError – raised when a mapping key is not found in the set of existing keys.

KeyboardInterrupt – raised when the user hits the interrupt key

NameError – raised when an identifier is not found locally or globally.

OverflowError – raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers.

RuntimeError – raised when an error is detected that doesn’t fall in any of the other categories.

StopIteration- raised by an iterator’s next() method to signal that there are no more elements.

SyntaxError - raised when the compiler encounters a syntax error. Import or exec statements and input() and eval() functions may raise this exception.

IndentationError – raised when indentation is not specified properly.

SystemExit – raised by the sys.exit() function is applied to an object of inappropriate data type.

TypeError – raised when an operation or function is applied to an object of inappropriate data type.

ValueError – raised when a built-in operation or function receives an argument that has right data type but wrong value.

ZeroDivisionError – raised when the denominator is zero in a division or modulus operation.

Handling SyntaxError

```
try:
        date = eval(input(“Enter  date:”))
except  SyntaxError:
        print(“Invalid  date entered”)
else:
        print(“you  entered:”,date)
```

Output:

```
Enter  date: 2019,02,20
you entered: (2019,02,20)
Enter  date: 2019,02b,20
Invalid date  entered
```

Handling IOError

```
try:
        name=input(“Enter  file  name:”)
        f = open(name, ’r’)
except  IOError:
        print(“File  not  found:”,name)
else:
        n=len(f.readlines())
        print(name,’has’,n,’lines’)
        f.close()
```

Output:

```
Enter file name:ex.py
Ex.py has 11 lines
Enter file name:abcd
File not found:abcd
```

The Except Block

It can be written in various formats.

➢ To catch the exception which is raised in the try block, we can write except block with the Exceptionclass name as

except Exceptionclass:

➢ We can catch the exception as an object that contains some description about the exception.

except Exceptionclass as obj:

➢ To catch multiple exceptions, we can write multiple catch blocks. We can also use a single except block and write all the exceptions as a tuple inside parentheses as

except (Exceptionclass1, Exceptionclass2,………)

➢ To catch any type of exception we can write except block without mentioning any Exceptionclass name as

except:

Handling more than one Exception

```
try:
    n = int(input("Enter  number : "))
    print("Result is", 10/n)
    print(“snist”+n)
except ZeroDivisionError:
    print("Division by zero is error !!")
except  ValueError:
    print(“give only digits")
except:
    print(“Exception")
else:
    print("No exceptions")
finally:
    print("This will execute no matter what")
```

Output:

Case 1

```
Enter  number : 0
Division by zero is error !!
This will execute no matter what
```

Case 2

```
Enter  number : two
give only digits
This will execute no matter what
```

Case 3

```
Enter  number : 2
Result is 5.0
Exception
This will execute no matter what
```

Exception with reference variable.

```
try:
    x = int(input("Your number: "))
    r= 10 / x
    print(r)
except ValueError as e:
    print ("You should have given either an int or a float",e)
except ZeroDivisionError as a:
    print ("Infinity : entered value is zero",a)
print("Rest of the Application.....")
```

Output:

case 1:

```
Your number:  10
1
Rest of the Application.....
```

case 2:

```
Your number:  0
Infinity : entered value is zero  division by zero
Rest of the Application......
```

case 3:

```
Your number:  snist
You should have given either an int or
a float could not convert string to
float: ‘snist'
Rest of the Application.....
```

Handling more than one exception by using single except block.

```
try:
    n = int(input("enter a number"))
    print(10/n)
    str="snist"
    print(str[10])
except (IndexError, ZeroDivisionError) as e:
    print("An ZeroDivisionError or a IndexError occurred :",e)
else:
    print(“No exception”)
```

Output:

case 1:

```
enter a number 0
An ZeroDivisionError or a IndexError occurred : division by zero
```

case 2:

```
enter a number2
5.0
An ZeroDivisionError or a IndexError occurred : string index out of range
```

Example program for Super class Exception

```
try:
    n = int(input("Enter  number : "))
    r = 10 / n
    print("Result is", r)
    print(‘25’+n)
except ZeroDivisionError:
    print("Division by zero is error !!")
except  ValueError:
    print(“give only digits")
except Exception:
    print(“ super class Exception")
else:
    print("No exceptions")
finally:
    print("This will execute no matter what")
```

Output:

Case 1

```
Enter  number : 0
Division by zero is error !!
This will execute no matter what
```

Case 2

```
Enter  number : two
give only digits
This will execute no matter what
```

Example program for Super class Exception

```
try:
    n = int(input("Enter  number : "))
    r = 10 / n
    print("Result is", r)
    print('25'+n)
except Exception:
    print(" super class Exception")
except ZeroDivisionError:
    print("Division by zero is error !!")
except  ValueError:
    print("give only digits")
else:
    print("No exceptions")
finally:
    print("This will execute no matter what")
```

Output:

Case 1

```
Enter  number : 0
 super class Exception
This will execute no matter what
```

Case 2

```
Enter  number : two
 super class Exception
This will execute no matter what
```

In two cases finally block is not executed

1. the control is not entered in try block
2. we use os._exit(0)

case 1:The control is not entered in try block so finally block is not executed.

```
print(10/0)
try:
        print("try block")
finally:
        print("finally block")
```

case 2:when we use os._exit(0) virtual machine is shutdown so finally block is not executed.

```
import os
try:
        print("try block")
        os._exit(0)
finally:
        print("finally block")
```

Raising Exceptions:

Exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword **raise.**

Ex:

```
try:
    age=int(input("enter age"))
    if(age<0:
            raise valueError
    elif(age>18):
            print("eligible for vote")
    else:
            print("not eligible for vote")
except  ValueError:
print("give  positive values only")
```

Output:

case1

```
enter age-15
give  positive values only
```

Case2

```
enter age12
not eligible for vote
```

```
try:
    age=int(input("enter age"))
    if(age<0:
        raise valueError("give positive values onl
    elif(age>18):
         print("eligible for vote")
    else:
        print("not eligible for vote")
except  ValueError as e:
print(e)
```

Nested Try-Except Blocks:

A try-except block can be surrounded by another try-except block.

Syntax

```
try:
        ---------
        ---------
        try:
                ----------
                ----------
        except:
                ----------
                ----------
except:
        -----------
        -----------
```

```
try:
        -----------
        -----------
        try:
                ---------
                ---------
        except:
                ---------
                ---------
        else:
                ---------
        finally:
                ---------
                ---------
except:
        ----------
        ----------
else:
        ---------
finally:
        ----------
        ----------
```

Example program for nested try-except block

```
try:
    n = int(input("enter a number"))
    print(10/n)
    try:
        str="snist"
        print(str[10])
    except IndexError as e:
        print(e)
except  ZeroDivisionError as e:
    print("An ZeroDivisionError :",e)
else:
    print("No exception")
```

Output:

Case 1

```
enter a number0
An ZeroDivisionError : division by zero
```

Case 2

```
enter a number2
5.0
string index out of range
No exception
```

Exception raised in inner try block will be handled by either inner except or outer except block

```
try:
   n = int(input("enter a number"))
   print(10/n)
   try:
      str=input(“enter string”)
      print(str[4])
      print(n+str)
   except IndexError as e:
      print(e)
except  ZeroDivisionError as e:
   print("An ZeroDivisionError :",e)
except  TypeError:
   print(“no operation b/w int and str”)
else:
   print("No exception")
```

Output

Case 1

```
enter a number0
An ZeroDivisionError : division by zero
```

Case 2

```
enter a number2
5.0
enter stringhif4
string index out of range
No exception
```

Case 3

```
enter a number2
5.0
enter stringhello
o
no operation b/w int and str
```

The assert statement

The assert statement is useful to ensure that a given condition is True. If it is not ture raise AssertionError.

Syntax

```
        assert  condition, message
```

If the condition is False, then the exception by the name AssertionError is raised along with the message written in the assert statement.

EX

```
try:
        x=int(input('Enter a number between 5 and 10:'))
        assert  x>=5  and  x<=10, "your  input  is  not  correct"
        print('The  number  entered:',x)
except   AssertionError  as  obj:
        print(obj)
```

User-Defined Exceptions

- The programmer create his own exceptions are called ‘user-defined exceptions’ or ‘custom exceptions’
- since all exceptions are classes, the programmer is supposed to create his own exception as a class and make this class as a subclass to the built in ‘Exception’ class.

```
class MyException(Exception):
        def __init__(self,arg):
                self.msg=arg
```

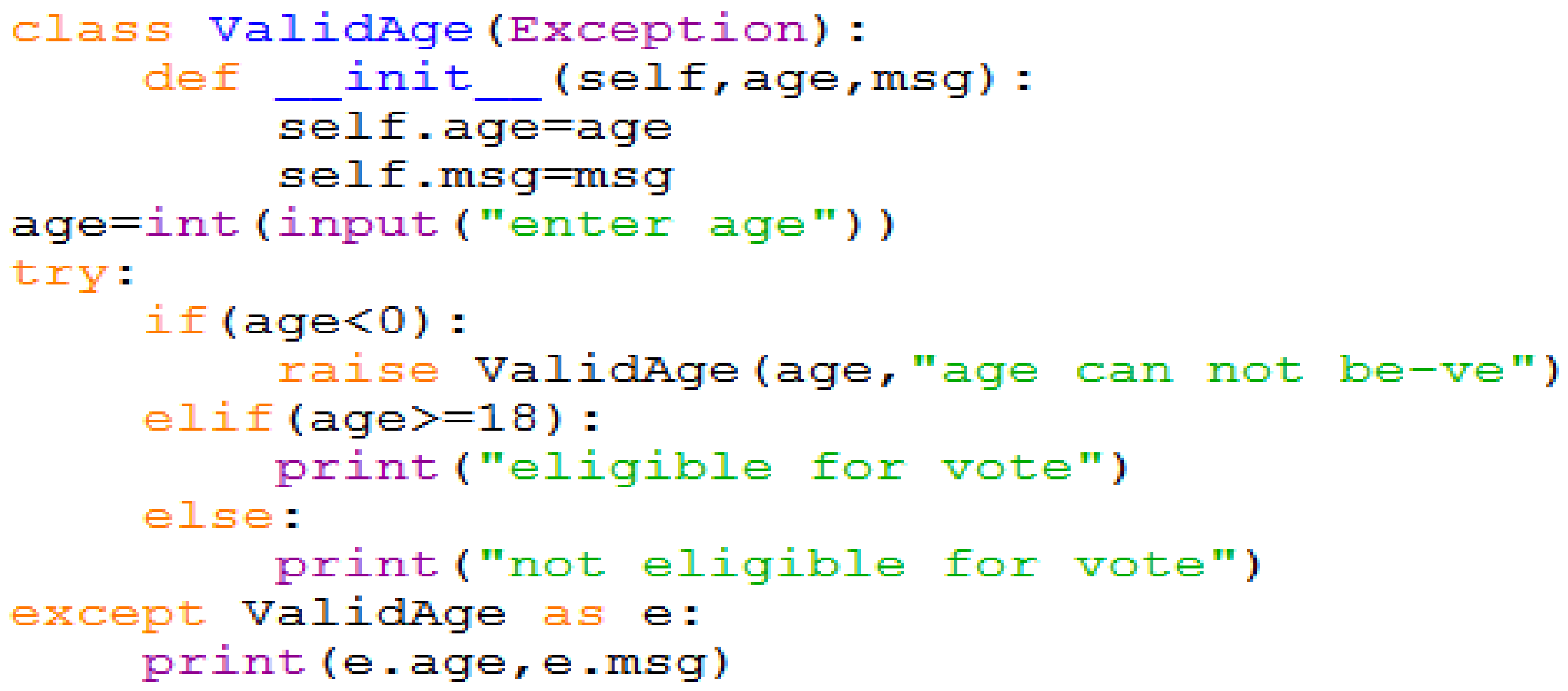
- When the programmer suspects the possibility of exception , he should raise his own exception using ‘raise’ statement as

```
raise MyException(‘message’)
```

- The programmer can insert the code inside a ‘try’ block and catch the exception using except block as

```
try:
        code
except MyException as me:
        pritnt(me)
```

Example program



```python
class ValidAge(Exception):
    def __init__(self,age,msg):
        self.age=age
        self.msg=msg
age=int(input("enter age"))
try:
    if(age<0):
        raise ValidAge(age,"age can not be-ve")
    elif(age>=18):
        print("eligible for vote")
    else:
        print("not eligible for vote")
except ValidAge as e:
    print(e.age,e.msg)
```

```
enter age-12
-12 age can not be-ve
>>>
enter age45
eligible for vote
>>>
enter age9
not eligible for vote
```