PYTHON PROGRAMMING

UNIT - II

FUNCTIONS

FUNCTIONS

Definition:

- Functions are blocks of organized, reusable code used to perform single or related set of actions.
- Provide better modularity and high degree of re usability.
- Python supports:
  - – Built-in functions like print(),math functions,type conversion functions etc..
  - – User - defined functions

BUILT-IN FUNCTIONS

Python provides a number of important built-in functions that we can use without using function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The max and min functions give us the largest and smallest values in a list, respectively:

Example:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
```

The max function tells us the "largest character" in the string (which turns out to be the letter "w") and the min function shows us the smallest character which turns out to be a space.

Another very common built-in function is the **len** function which tells us how many items are in its argument. If the argument to len is a string, it returns the number of characters in the string.

Example:

```
>>> len('Hello world')
11
```

1.TYPE CONVERSION FUNCTIONS

Python also provides built-in functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

Examples:

```
>>> int('32')
32
```

```
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

- int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

- float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

- Finally, str converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

2.RANDOM NUMBERS

The random module provides functions that generate pseudorandom numbers .The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series.

Example 1:

```
import random
for i in range(10):
    x = random.random()
    print x
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

OUTPUT:

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
```

```
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

The random function is only one of many functions which handle random numbers. The function randint takes parameters low and high and returns an integer between low and high (including both).

Example 2:

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Example 3:

```
import random

print ("A random number from list is : ",random.choice([1, 4, 8, 10, 3]),)

print ("A random number from range is : ",end="")

print (random.randrange(20, 50, 1))

print (random.random())

li = [1, 4, 5, 10, 2]

print ("The list before shuffling is : ", end="")

for i in range(0, len(li)):

    print (li[i], end=" ")

print("\r")

random.shuffle(li)
```

```
print ("The list after shuffling is : ", end="")
for i in range(0, len(li)):
    print (li[i], end=" ")
print("\r")
print ("The random floating point number between 5 and 10 is : ",end="")
print (random.uniform(5,10))
```

Output:

```
A random number from list is :  10
A random number from range is : 42
0.4890781390951835
The list before shuffling is : 1 4 5 10 2
The list after shuffling is : 1 4 10 5 2
The random floating point number between 5 and 10 is : 9.518753303584672
```

3.MATH FUNCTIONS

Python has a math **module** that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a **module object** named math. If you print the module object, you get some information about it:

```
>>> print math
<module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

Example 1:

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called log that computes logarithms base e.

Example 2:

The second example finds the sine of radians. The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2 p:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

USER DEFINED FUNCTIONS

Defining a function:

- Function blocks starts with a keyword **"def"** followed by function_name, parenthesis (())

and a colon :

- Arguments are placed inside these parenthesis
- The first statement of a function can be an optional statement - the documentation string of the function or "docstring".
- Every line inside code block is indented
- return [expression] statement exits the function by returning an expression to the called function.
- return statement with no expression is same as return None.

SYNTAX:

```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

Example 1:

```
def print_lyrics():
    print ("I'm a lumberjack, and I'm okay.")
    print ('I sleep all night and I work all day.')
```

Explanation:

- def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names.
- The empty parentheses after the name indicate that this function doesn't take any arguments.
- The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented.

Calling a Function:

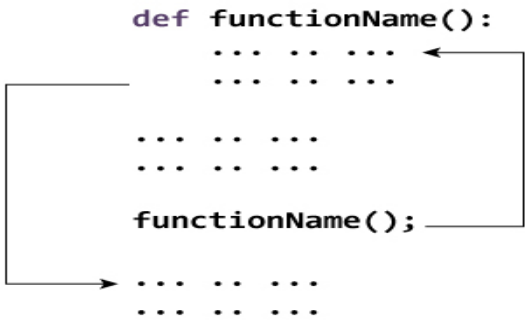
- Defining a function gives it a name, specifies function parameters and structures the

blocks of code.

- Functions are invoked by a function call statement/code which may be part of another

function

How Function works in Python?



Example 2:

```
def print_str(str1):          # Defining function print_str(str1)
    print("This function prints string passed as an argument")
    print(str1)
print_str("Calling the user defined function", print_str("welcome"))
# Calling user-defined function
```

OUTPUT:

Welcome

Note:

This function prints string passed as an argument

Calling the user defined function print_str(str1)

Example 3:

```
def printme( str ):                    # Function definition
        "This prints a passed string into this function"
         print (str)
         return
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

OUTPUT:

I'm first call to user defined function!

Again second call to the same function

Pass arguments to functions:

- Arguments are passed by reference in Python
- Any change made to parameter passed by reference in the called function will reflect in the calling function based on whether data type of argument passed is mutable or immutable
- In Python
  - -Mutable Data types include Lists, Sets, Dictionary
  - -Immutable Data types include Number, Strings, Tuples

EXAMPLE 1:

Pass arguments to functions: Immutable Data Type – Number

```
def change(cust_id):
    #Function Definition
        cust_id += 1
        print("Customer Id in function definition: ", cust_id)
        return
cust_id = 100
  # Function Invocation with arguments of immutable data type
print("Customer Id before function invocation: ", cust_id)
```

```
change(cust_id)
print("Customer Id after function invocation: ", cust_id)
```

OUTPUT:

```
Customer Id before function invocation: 100
Customer Id in function definition: 101
Customer Id after function invocation: 100
```

EXAMPLE 2:

Pass arguments to functions: Mutable Data Type – List

```
def change(list_cust_id):                    #Function Definition
    list_cust_id.append([10, 20, 30])
    #Assign new values inside the function
    print("Customer Id in function definition: ", list_cust_id)
    return
list_cust_id = [100, 101, 102]          # Function Invocation with arguments of immutable data type
print("List of Customer Id before function invocation: ", list_cust_id)
change(list_cust_id)
print("Customer Id after function invocation: ", list_cust_id)
```

OUTPUT:

```
List of Customer Id before function invocation: [100, 101, 102]
List of Customer Id in function definition: [100, 101, 102, [10, 20, 30]]
List of Customer Id after function invocation: [100, 101, 102, [10, 20, 30]]
```

FUNCTION ARGUMENTS TYPES:

- Different types of formal arguments:
  - – Required arguments
  - – Keyword arguments
  - – Default arguments
  - – Variable – length arguments

1. Required arguments

– Arguments follow positional order

– No. of arguments and the order of arguments in the function call should be exactly same as that in function definition

Example 1:

```
def print_str(str1):                    # Function Definition
        print("This function prints the string passed as an argument")
        print(str1)
        return
print_str()                 # Function Invocation without required arguments
```

OUTPUT:

TypeError: print_str() missing 1 required positional argument: 'str1'

2. Keyword arguments

- when used in function call, the calling function identifies the argument by parameter name
- Allows you to skip arguments or place them out of order
- Python Interpreter uses the keyword provided to match the values with parameters
- Once the value for keyword is provided in argument ,all value to its right side must hold the value for defined keyword(refer Example 2: below)

Example 1:

```
def customer_details (cust_id, cust_name):          # Function Definition
        print("This function prints Customer details")
        print("Customer Id: ",cust_id)
        print("Customer Name: ",cust_name)
        return
customer_details(cust_name = "John", cust_id = 101)
   # Function Invocation with Keyword arg
```

Output:

This function prints Customer details

Customer Id: 101

Customer Name: John

Example 2:

```
def customer_details (cust_id, cust_name=”hari”,cust_age=32):          # Function Definition
        print("This function prints Customer details")
        print("Customer Id: ",cust_id)
        print("Customer Name: ",cust_name)
        print("Customer Name: ",cust_age)
        return
customer_details(cust_id = 101)
   # Function Invocation with Keyword arg
```

Output:

This function prints Customer details

Customer Id: 101

Customer Name: John

Cutomer age:32

NOTE: if age keyword is not defined with values it shows error as : non default argument follows default argumen

3. Default Arguments:

- Assumes a default value if the value is not specified for that argument in the function call

Example 1:

```
def customer_details (cust_name, cust_age = 30):        # Function Definition
        print("This function prints Customer details")
        print("Customer Name: ",cust_name)
        print("Customer Age: ",cust_age)
        return
customer_details(cust_age = 25, cust_name = "John")   # Function Invocation with Default arg
```

customer_details(cust_name = "John")

Output:

This function prints Customer details

Customer Name: John

Customer Age: 25

This function prints Customer details

Customer Name: John

Customer Age: 30

4. Variable-length arguments

- Used to execute functions with more arguments than specified during function definition
- unlike required and default arguments, variable arguments are not named while defining

a function

Syntax:

def functionname([formal_args,] *var_args_tuple):

optional: Any print statement for documentation"

function_suite

return [expression]

- An asterisk „*“ is placed before variable name to hold all non-keyword variable arguments

- *var_args_tuple is empty if no additional arguments are specified during function call.

Example 1:

def customer_details (cust_name, *var_tuple): **# Function Definition**

print("This function prints Customer Names")

print("Customer Name: ",cust_name)

for var in var_tuple:

print(var)

return

customer_details("John", "Joy", "Jim", "Harry") **# Function Invocation with Variable length arg**

```
customer_details("Mary")
```

Output:

```
This function prints Customer Names
Customer Name: John
Joy
Jim
Harry
This function prints Customer Names
Customer Name: Mary
```

Example 2:

```
def fn(rank, **names):
    print ("rank", rank)
    for n in names:
        print ("students name is %s:%s"%(n, names[n]))
        print(n,names.keys())
        print(n,names.values())
        print(rank,names)
fn(1, name1="ram", name2="raj")
```

Output:

```
rank 1
students name is name2:raj
name2 dict_keys(['name2', 'name1'])
name2 dict_values(['raj', 'ram'])
1 {'name2': 'raj', 'name1': 'ram'}
students name is name1:ram
name1 dict_keys(['name2', 'name1'])
name1 dict_values(['raj', 'ram'])
```

1 {'name2': 'raj', 'name1': 'ram'}

SCOPE OF VARIABLES:

- Determines accessibility of a variable at various portions of the program
- Different types of variables
- **Local variables:-** Variables defined inside the function have local scope..It can be accessed only inside the function in which it is defined.
- **Global variables:-**Variables defined outside the function have global scope.Variables can be accessed throughout the program by all other functions as well.

Example 1:

```
total = 0
def add( arg1, arg2 ):                    # Function Definition
        total = arg1 + arg2               # total is local variable
        print ("Value of Total(Local Variable): ", total)
        return total
add( 25, 12 )                             # Function Invocation
print("Value of Total(Global Variable): ", total)
```

OUTPUT:

```
Value of Total(Local Variable): 37
Value of Total(Global Variable): 0
```

Usage of keyword ”Global”:

- Used to access the variable outside the function**.**

EXAMPLE 2:

```
total = 0
def add( arg1, arg2 ):                    # Function Definition
        global total
        TOTAL = arg1 + arg2; # Here total is made global variable
        print ("Value of Total(inside the function): ", total)
        return total
```

```
add( 25, 12 )                    # Function Invocation
print("Value of Total(outside the function): ", total)
```

OUTPUT:

Value of Total(inside the function): 37

Value of Total(outside the function): 37

Example 3:

```
n1=10
print("global",n1)
def fun(n2):
    print("local",n2)
    global n3
    n3=15
fun(20)
print(n1)
print(n3)
```

Output:

global 10

local 20

10

15

Recursion:

- A method invoking itself is referred to as Recursion
- Typically, when a program employs recursion the function invokes itself with a smaller argument..
- Computing factorial(5) involves computing factorial(4), computing factorial(4) involves computing factorial(3) and so on..
- Often results in compact representation of certain types of logic and is used as

substitute for iteration.

Example 1:

```
def calc_factorial(x):
    """This is a recursive function  to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

OUTPUT:

The factorial of 4 is 24.

Advantages of recursion:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

The Anonymous Functions:

- These functions are called anonymous because they are not declared in the standard manner by using the def keyword.
- We can use the lambda keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local name space and cannot access variables other than those in their parameter list and those in the global name space.

Syntax:

lambda [arg1 [,arg2,.....argn]]:expression

Example 1:

```
sum = lambda arg1, arg2: arg1 + arg2              # Function definition


print "Value of total : ", sum( 10, 20 )            #function call
print "Value of total : ", sum( 20, 20 )
```

Output:

```
Value of total :  30
Value of total :  40
```

Use of Lambda Function:

- We use lambda functions when we require a nameless function for a short period of time.
- we generally use it as an argument to a higher-order function

The return Statement:

- The statement return [expression] exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.

Example 1:

```
def sum( arg1, arg2 ):                # Function definition
   # Add both the parameters and return them."
   total = arg1 + arg2
   print ("Inside the function : ", total)
   return total
total = sum( 10, 20 )                #function call
print ("Outside the function : ", total)
```

Output:

```
Inside the function :  3
Outside the function :  30
```

FRUITFUL FUNCTIONS AND VOID FUNCTIONS

Fruitful Functions:- Functions which yields results .

Example 1:

```
import math
x=math.sqrt(100)/2
print(x)
```

Void Functions:- It performs action but doesn't return any value.

Example 1:

```
def print():              #function def
    print("hai")
print()                   #function call
```

Generators:

We can create our own generator using yield functions.

Example 1:

```
def natural(n):
    i=1
    while(i<n):
       yield i
       i=i+1
n=eval(input("enter value"))
nat_iter=natural(n)
print(next(nat_iter))
print(next(nat_iter))
```

Output:

```
>>>
enter value3
1
2
```

SAMPLE PROGRAMS USING FUNCTIONS:

1. Program to sum the elements in the tuple and also find the max and min element, length from tuple.

SOLUTION:

```
def func_sum(t):
   print("sum:",sum(t))

def func_max(t):
   print("max:",max(t))

def func_min(t):
   print("min:",min(t))

def func_len(t):
   print("len:",len(t))

t=(4,5,6)
func_sum(t)
func_max(t)
func_min(t)
func_len(t)
```

OUTPUT:

```
>>>
sum: 15
max: 6
min: 4
len: 3
```

2.Consider the sample string "Welcome To Python World".Develop a Python code with function that accepts a string and calculate the number of upper case letters and lower case letters.

Sample String : Welcome To Python World
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 16

SOLUTION:

```
def calc(s):
```

```
    ucount=lcount=0
    for i in s:
        if i.isupper():
            ucount=ucount+1
        if i.islower():
            lcount=lcount+1

    print("uppercase:",ucount)
    print("lowercase:",lcount)

x=input("Enter the string:")
calc(x)
```

OUTPUT:

3. Program to write two functions that should return the square and cube of number

SOLUTION:

```
def square(n):
    return n*n

def cube(n):
    return n*n*n
num=int(input("enter the num"))
print("square:",square(num))
print("cube:",cube(num))
```

OUTPUT:

4. Program to display a random number:

SOLUTION:

```
import random
print(random.random())
print(random.randint(3,10))
t=(1,2,4,5)
print(random.choice(t))
```

OUTPUT:

```
>>>
0.7853489859084744
3
2
```

CLASSES AND OBJECTS

INTRODUCTION

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy.

OVERVIEW OF OOP TERMINOLOGY

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation:** The creation of an instance of a class.
- **Method :** A special kind of function that is defined in a class definition.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

CLASS:

- A class is a blueprint for the object.
- An object is also called an instance of a class and the process of creating this object is known as instantiation.
- A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
- When we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Syntax:

```
class ClassName:
   <statement-1>
   .
   .
    <statement-N>
```

Example 1:

```
class Myclass:
        "My first class program"
        a=10
        def func():
                print("hello")
```

When we define a class, a new class object is created with the same class name. This new class object provides a facility to access the different attributes as well as to instantiate new objects of that class.

There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class.

Create an Object in Python

SYNTAX: ObjectName=className()

Example 1:

```
Obj=Myclass()
```

Constructors in Python

- Class functions that begins with double underscore (__) are called special functions as they have special meaning.
- __init__() function This special function gets called whenever a new object of that class is instantiated.

Example 1:

```
class First:
                def __init__(self,a=0,b=0):
                    self.a=a
                    self.b=b
                def getData(self):
                    print("{0},{1}".format(self.a,self.b))
```

```
First1=First(2,3)
First1.getData()
```

Output:

```
>>>
2,3
>>>
```

Example 2:

```
class Student:
  def __init__(self, rollno, name):
    self.rollno = rollno
    self.name = name
  def displayStudent(self):
    print "rollno : ", self.rollno,  ", name: ", self.name
emp1 = Student(121, "Asha")
emp2 = Student(122, "Sono")
emp1.displayStudent()
emp2.displayStudent()
```

Output:

```
>>>
rollno :  121 , name:  Asha
rollno :  122 , name:  Sono
>>>
```

Difference between class variable Versus Object variable.

Class variable-

- is shared and accessed by all objects of class.
- When any one object makes a change to class variable,the change is reflected in all instances.

Object Variable-

- Owned by individual object/instance
- Each object has its own copy
- Not Shared

Example 3:

```
class Myclass:
   cv=0
   def __init__(self,var):
      Myclass.cv+=1          #class variable
      self.var=var           #Object variable
      print("Object variable",var)
      print("class variable",Myclass.cv)
m1=Myclass(10)
m2=Myclass(20)
m3=Myclass(30)
```

```
print(Myclass.var)
```

Output:

```
>>>
Object variable 10
class variable 1
Object variable 20
class variable 2
Object variable 30
class variable 3
```

Example 4:

```
class Number:
    even=[]
    odd=[]
    def __init__(self,no):
        self.no=no
        if no%2==0:
            Number.even.append(no)
        else:
            Number.odd.append(no)
n1=Number(21)
n2=Number(32)
n3=Number(45)
n4=Number(18)
print(Number.even)
print(Number.odd)
```

Output:

```
>>>
[32, 18]
[21, 45]
>>>
```

To add Modify values In instance variables

Example 1:

```
class Test:
    def fun(self):
        print(self.n)
Obj1=Test()
Obj1.n=30
print(Obj1.n)
Obj1.n=20    # n value is modified
print(Obj1.n)
```

Output:

```
>>>
30
20
>>>
```

Deleting Attributes and Objects

- similar to destructor.
- automatically called when object is going out of scope

To delete an attributname:

Syntax: **del objname.attributename**

To delete an object:

Syntax: del objectname

Example 1:

```
class Bank:
   bank_name="bank"
   def __init__(self,name):
      self.name=name
   def fun(self):
      print(self.name+" "+Bank.bank_name)

   def __del__(self):
      print("object out of scope")
o1=Bank('Axis')
o1.fun()

o2=Bank('ICICI')
o2.fun()
del o1.a
print(o1.name)
```

Output:

```
Axis bank
ICICI bank
Traceback (most recent call last):
  File "/home/students/Desktop/w.py", line 15, in <module>
    del o1.a
AttributeError: a
```

Object Oriented Concepts:

- CLASS
- INHERITANCE
- OBJECTS
- ABSTRACTION

- ENCAPSULATION
- POLYMORPHISM

Inheritance

- When one object acquires all the properties and behaviours of parent object i.e. known as inheritance.

Polymorphism

- When one task is performed by different ways

Abstraction

- Hiding internal details and showing functionality is known as abstraction.

Encapsulation.

- Binding (or wrapping) code and data together into a single unit is known as encapsulation

Data Encapsulation:

- It is a process to restrict the access of data members
- Some objects are not visible from outside of object definition
- Public,protected,private

The following table shows the different behaviour:

Name	Notation	Behaviour
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside.
__name	Private	Can't be seen and accessed from outside

Example 1:

```
class Sample:
   v1=10
   _v2=20
   __v3=30
   def fun1(self):
      print("v3",Sample.__v3)
m1=Sample()
m1.fun1()
print(m1.v1)
print(m1._v2)
```

print(m1.__v3)

Output:

```
v3 30
10
20
Traceback (most recent call last):
  File "/home/students/Desktop/w.py", line 11, in <module>
    print(m1.__v3)
AttributeError: 'Sample' object has no attribute '__v3'
```

In order to access __v3 outside the class, use class name
print(m1._Sample__v3)
output:
30

BUILT-IN CLASS ATTRIBUTES

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Example 1:

```
class Employee:
   'employee'
   empCount=0
   def __init__(self,name,salary):
      self.name=name
      self.salary=salary
      Employee.empCount+=1
   def displayCount(self):
      print("empid",Employee.empCount)
   def displayEmployee(self):
      print(self.name,self.salary)
print(Employee.__doc__)
print(Employee.__name__)
print(Employee.__module__)
```

Output:

```
employee
Employee
__main__
```

Passing Objects as Arguments

Example 1:

```
class Square:
    pass
s=Square()
s.side=4
def area(obj):
    a=obj.side*obj.side
    print(a)
area(s)
```

Output:

```
>>>
16
>>>
```

Passing Objects as return

```
class sq:
    pass
s1=sq()
s1.side=4

def area(obj):
    print(s1.side)
    s1.side=obj.side*obj.side
    return(s1)
obj=area(s1)
print(obj.side)
```

Output:

4
16

METHOD OVERLOADING IN PYTHON

- **Python does not support method overloading.**
- **Can be achieved by default arguments**

Example 1:

```
class class_A:
    def fun_add(self,val1,val2,val3=None,val4=None):
```

```
    if val3 is None and val4 is None:
        print(val1+val2)
    elif(val4 is None):
        print(val1*val2*val3)
    else:
        print(val1+val2+val3+val4)

obj=class_A()
obj.fun_add(23, 12)
obj.fun_add(23, 12,10)
obj.fun_add(23, 12,22,10)
```

Output:

```
35
2760
67
```

Operator Overloading

Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.

Operator Overloading Special Functions in Python

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators in Python

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Comparision Operator Overloading in Python		
Operator	**Expression**	**Internally**
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Example 1:

```
class class_A:

    def __init__(self,val1,val2):
        self.val1=val1
        self.val2=val2

    def __add__(self,n):
        print(self.val1*n.val2)

obj=class_A(3,4)
obj1=class_A(10,20)
obj+obj1
```

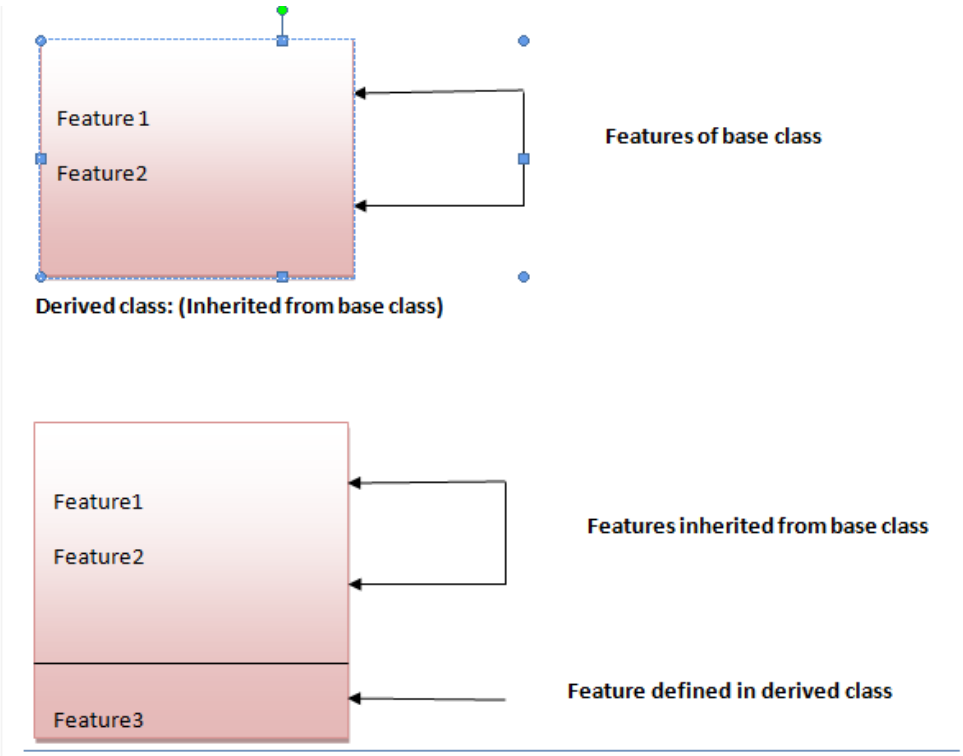
Output:

60

Inheritance

Inheritance is used to specify that one class will get most or all of its features from its parent class. It is a feature of Object Oriented Programming.

It facilitates re-usability of code.



Types of inheritance

- Single inheritance
- Multiple inheritance
- Multilevel inheritance

Basic Inheritance program

Example 1:

```
class A:
    def print1(self):
        print("hello")
class B(A):
    def print2(self):
        print("hai")
ob=B()
ob.print2()
```

ob.print1()

Output:

```
>>>
hai
hello
>>>
```

SINGLE INHERITANCE IN PYTHON

Python Inheritance Syntax:

```
class BaseClass:
  Body of base class

class DerivedClass(BaseClass):
  Body of derived class
```

Example 1:

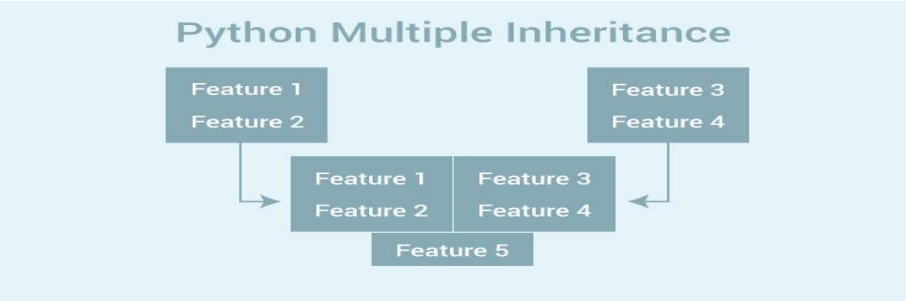
```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
t = Triangle()
t.inputSides()
t.dispSides()
t.findArea()
```

Output:

```
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4
```

Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0
The area of the triangle is 6.00

Multiple Inheritance:



- In multiple inheritance, the features of all the base classes are inherited into the derived class.
- The syntax for multiple inheritance is similar to single inheritance.

Syntax:

```
class Base1:
   pass

class Base2:
   pass

class MultiDerived(Base1, Base2):
   pass
```

Example 1:

SOLUTION:

```
class Base1(object):
   def __init__(self):
      self.str1 = "python"
      print("Base1")

class Base2(object):
   def __init__(self):
      self.str2 = "programming"
      print("Base2")

class Derived(Base1, Base2):
   def __init__(self):

      # Calling constructors of Base1
      # and Base2 classes
      Base1.__init__(self)
      Base2.__init__(self)
```

```
        print("Derived")

    def printStrs(self):
        print(self.str1, self.str2)


ob = Derived()
ob.printStrs()
```

OUTPUT:

```
>>>
Base1
Base2
Derived
python programming
```

Multilevel Inheritance :

- In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

Syntax:

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

Example 1:

SOLUTION:

```
# Define a class as 'student'
class student:
    # Method
    def getStudent(self):
        self.name = input("Name: ")
        self.age = input("Age: ")
        self.gender = input("Gender: ")

# Define a class as 'test' and inherit base class 'student'
class test(student):
    # Method
    def getMarks(self):
        self.stuClass = input("Class: ")
        print("Enter the marks of the respective subjects")
```

```
        self.literature = int(input("Literature: "))
        self.math = int(input("Math: "))
        self.biology = int(input("Biology: "))
        self.physics = int(input("Physics: "))

# Define a class as 'marks' and inherit derived class 'test'
class marks(test):
    # Method
    def display(self):
        print("\n\nName: ",self.name)
        print("Age: ",self.age)
        print("Gender: ",self.gender)
        print("Study in: ",self.stuClass)
        print("Total Marks: ", self.literature + self.math + self.biology + self.physics)


m1 = marks()
# Call base class method 'getStudent()'
m1.getStudent()
# Call first derived class method 'getMarks()'
m1.getMarks()
# Call second derived class method 'display()'
m1.display()
```

OUTPUT:

```
Name: ram
Age: 24
Gender: male
Class: 10th
Enter the marks of the respective subjects
Literature: 45
Math: 99
Biology: 88
Physics: 67


Name:  ram
Age:  24
Gender:  male
Study in:  10th
Total Marks:  299
```

Sample programs in class Example 1:

```
class Base:
    x=20
    def __init__(self,a):
        self.a=a
        print("base cons",a)
    def disp(self):
        print("Inside base Class Display")
```

```
        print(self.a)

class Derived(Base):
    def __init__(self,a,b):
        Base.__init__(self,a)
        super().__init__(a)
        super(Derived,self).__init__(a)
        self.b=b
        print("Derived",self.a,self.b)
    def disp(self):
        super(Derived,self).disp()
        print("Inside Derived Class Display")

Derived1=Derived(7,8)
Derived1.disp()
```

Output:

```
>>>
base cons 7
base cons 7
base cons 7
Derived 7 8
Inside base Class Display
7
Inside Derived Class Display
>>>
```

Program 2:

```
#----- Parent class: person --------
class person:
    def __init__(self,first,last):
        self.firstName = first
        self.lastName = last

    def getName(self):
        return self.firstName + " " + self.lastName

#----- child class: Employee inherited from person --------
class Employee(person):
    def __init__(self,first,last,staffNum):
        person.__init__(self,first,last)
        self.staffNum = staffNum

    def getEmployee(self):
        return self.getName() + ": " + self.staffNum
#----- child class: Employee inherited from person --------
class Boss(person):
    def __init__(self,first,last,title):
```

```
        person.__init__(self,first,last)
        self.title = title

    def getBoss(self):
        return self.getName() + ": Employee ID: " + self.title

#----- instantiate class --------
person_1 = person("Math", "Hoang")
Employee_1 = Employee("Math", "Hoang", "12344")
Employee_2 = Employee("Blog", "spot", "43211")
Boss_ = Boss("Mathhoang", "Blogspot.com", "CEO")

#------ print out information ------
print(person_1.getName())
print(Employee_1.getEmployee())
print(Employee_2.getEmployee())
print(Boss_.getBoss())
```

Output:

```
>>>
Math Hoang
Math Hoang: 12344
Blog spot: 43211
Mathhoang Blogspot.com: Employee ID: CEO
```

3.Program to depict built-in attributes:

SOLUTION:

```
class Person:
    age = 23
    name = 'Adam'

    def disp(self):
        print(self.age,self.name)

person = Person()
print('Person has age?:', hasattr(person, 'age'))
print(getattr(person,'age'))
person.disp()
setattr(person,'age',44)
person.disp()
print('Person has salary?:', hasattr(person, 'salary'))
```

Output:

```
>>>
Person has age?: True
23
23 Adam
44 Adam
Person has salary?: False
```

4.Program to depict __init__ and __del__
SOLUTION:

```
class Student:
    def __init__(self,rollno,name):
        self.rollno=rollno
        self.name=name

    def dispdetails(self):
        print(self.rollno,self.name)

    def __del__(self):
        print("Object destroyed")
roll=int(input("enter rollno"))
name=input("enter name")
s1=Student(roll,name)
s1.dispdetails()
del s1
```

Output:

```
>>>
enter rollno25
enter nameram
25 ram
Object destroyed
```

5. The Titan company planned to maintain branch office details such as Officer name,Age,City and phone number.

Name	Age	City	Phone
Vijay kumar	27	Chennai	9876548787
Siddharth	26	Coimbatore	8765767600
Abinanth	27	Mysore	7865767645
Sharukh	27	Mumbai	8989797877

Ashiek	26	Banglore	78987654911

Develop a python code to perform the following:
1.Display all the Branch officer's name and age alone.
2.Identify the branch office city name.
3.Display the contact number of the person Abinanth.
4.Find the created Class name.
5.Find all the branch office details.

CONSTRAINTS :

Class Name	Method Name	Operation
Person	def __init__(self,name=None, age=0,city=None,phone=None):	Initialize the attributes which are given in the function arguments.

- ✓ Create an object as an empty list.
  For Example :
  personList = []
- ✓ Append the branch office information in the list object.

SOLUTION:

```
class Person:
    def __init__(self,name=None, age=0,city=None,phone=None):
        self.name=name
        self.age=age
        self.city=city
        self.phone=phone
    def dispdetails(listob):
        print("All the details");
        for i  in listob:
            print(getattr(i,'name'),getattr(i,'age'),getattr(i,'city'),getattr(i,'phone'))
        print("Branch office");
        for i  in listob:
            print(getattr(i,'city'))
        print("Name and age");
        for i  in listob:
            print(getattr(i,'name'),getattr(i,'age'))
        print("contact num of abinanth:")
        for i  in listob:
            if (getattr(i,'name')=='abinanth'):
                print(getattr(i,'name'),getattr(i,'age'))
        print(Person.__name__)

    def __del__(self):
```

```
        print("Object destroyed")
l=[]
n=int(input("enter the count"))
for i in range(n):
    name=input("name")
    age=int(input("age"))
    city=input("city")
    phone=int(input("phone"))
    ob=Person(name,age,city,phone)
    l.append(ob)
Person.dispdetails(l)
```

Output:

```
>>>
enter the count2
nameram
age24
citycbe
phone79890789994
namesita
age23
citycbe
phone79009494804
All the details
ram 24 cbe 79890789994
sita 23 cbe 79009494804
Branch office
cbe
cbe
Name and age
ram 24
sita 23
contact num of abinanth:
Person
```

6.Program to store even and odd nums in list

SOLUTION:

```
class Number:
    even=[]
    odd=[]
    def __init__(self,no):
        if(no%2==0):
            Number.even.append(no)
        else:
            Number.odd.append(no)

n1=Number(21)
n2=Number(34)
n3=Number(45)
n4=Number(90)
print("Even:",Number.even)
print("Odd:",Number.odd)
```

Output:

```
>>>
Even: [34, 90]
Odd: [21, 45]
```

7.Mr. Rithick, student of **"Government College of Technology"**, Coimbatore is maintaining the book details in the college library. Consider the given book details in the:

S.No	Author	Title	Publisher	Year of Publication	Available
1	Graham Glass	Web Technology	Pearson Education	2003	2
2	Arnold Robins	Operating systems	O REILLY	2005	3
3	Sumitabha Das	Shell scriptig	Tata McGraw-Hill Education	2006	5
4	Stephen G.Kochan	UI concepts	Sams publishing	2005	1
5	Siva	Basics of Python	PHI Learning pvt.lmt	2009	2

The Librarian checks the following using python Code :

- ✓ Create two classes Author and book.
- ✓ Inherits book from Author
- ✓ Count the total number of books in the library.
- ✓ Identify which book count is below 2.
- ✓ Find the book name of the author "siva".

CONSTRAINTS :

Class Name	Method Name	Operation
Author	def __init__(self,sno=0,bauthor=None,Bname=None):	Initialize the attributes which are given in the function arguments.

Book **Inherits** **Author**	def get_details: def dis_details:	Read and display the details of Publication

SOLUTION:

```
class Author:
    def __init__(self,sno=0,bauthor=None,Bname=None):
        self.sno=sno
        self.bauthor=bauthor
        self.Bname=Bname

class Book(Author):
    count=0
    countb=[]
    def getdetails(self):

        self.publisher=input("Enter publication")
        self.yop=int(input("enter year of publication"))
        self.avail=int(input("enter availability"))
        Book.count+=self.avail
        if(self.avail<2):
            Book.countb.append(self.Bname)

    def dispdetails(self):
        print(self.sno,self.bauthor,self.Bname,self.publisher,self.yop,self.avail)

l=[]
n=int(input("enter the num of books"))
for i in range(n):
    sno=int(input("sno:"))
    bauth=input("Enter book author")
    bname=input("Enter book name")
    ob=Book(sno,bauth,bname)
    ob.getdetails()
    l.append(ob)

print("-----BOOK DETAILS-----")
for i in l:
    i.dispdetails()

print("-----BOOK COUNT----",Book.count)

print(Book.countb)
```

```
for i in l:
    if(i.bauthor=="siva"):
        print(i.Bname)
```

Output:

```
>>>
enter the num of books2
sno:124
Enter book authorrowling
Enter book nameharry potter
Enter publication1995
enter year of publication1995
enter availability100
sno:276
Enter book authormeyer
Enter book nametwilight
Enter publicationwb
enter year of publication2001
enter availability200
-----BOOK DETAILS-----
124 rowling harry potter 1995 1995 100
276 meyer twilight wb 2001 200
-----BOOK COUNT---- 300
[]
```

FILES

Definition:

- Files are used to store data permanently on a non volatile memory such as hard disk.
- File is categorized as either text or binary.
- **Text File**- Text files are structured as a sequence of lines, where each line includes a sequence of characters
- **Binary File**- A binary file is any type of file that is not a text file.

File operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

How to open a file?

Python's built-in *open()* function is used to open the file.

Syntax: fileobject=open("filename","mode","buffering")

- **File name:** The file name argument is a string value

- **Access mode:** The access mode determines the mode in which the file has to be opened, i.e., read, write, append, etc.
- **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.

The *file* Object Attributes

Once a file is opened and you have one *file* object, We can get various information related to that file.

Here is a list of all attributes related to file object:

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.softspace	Returns false if space explicitly required with print, true otherwise.

List of the different modes of opening a file:

Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Example 1:

```
fo = open("foo.txt", "w")
print("Name of the file: ", fo.name)
print("Closed or not : ", fo.closed)
print("Opening mode : ", fo.mode)
```

Output:

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

Closed or not : False
Opening mode : w

The *close ()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done. Python automatically closes a file when the reference object of a file is reassigned to another file.

Syntax: fileobject.close()

Example 1:

```
# Open a file
fo = open("foo.txt", "w")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

Output:

Name of the file: foo.txt

Writing and Reading file:

Writing into file:

1. Write () Method: The *write()* method writes any string to an open file**.**

Syntax: fileObject.write(string)

Example 1:

```
fo = open("foo.txt", "w")                          # Open a file
fo.write( "Python is a great language.\nYeah its great!!\n");
fo.close()                                          # Close opend file
```

Output:

Python is a great language.
Yeah its great!!

2**. Writelines() Method:-** Writes a sequence of strings to the file.

Syntax : file. *Writelines (iterable)*

Sequence:

Any iterable object producing strings, typically a list of strings.

Example 1:

```
f=open("s.txt",'w')
f.writelines(['foo','bar'])
f.close()
```

Reading a file:

1. Read () Method:-It is used to read the data from the file.

Syntax: fileobject.read(value)

Example 1:

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

Output:
Read String is : Python is

2.**Readlines() Method**:-It is used to read many number of lines from the file.

Syntax: fileobject.readlines()

Example 1:

```
f=open("s.txt",'w')
f.writelines(['foo','bar'])
f.close()
f=open("s.txt",'r')
print(f.readlines())
```

Output:
['foo','bar']

File Positions Methods(tell() &seek())

- The ***tell()*** method tells the current position within the file.

Syntax:fileobject.tell()

- The *seek(offset[, from])* method changes the current file position.
- The *offset* argument indicates the number of bytes to be moved.
- The *from* argument specifies the reference position from where the bytes are to be moved.

Syntax:fileobject.seek(offset,[from])

Example 1:

```
fo = open("foo.txt", "r+")                    # Open a file
str = fo.read(10);
print("Read String is : ", str)
```

```
position = fo.tell()                    # Check current position
print ("Current file position : ", position)

position = fo.seek(0, 0)
str = fo.read(10);
print ("Again read String is : ", str)
fo.close()                              # Close opend file
```

Output:

```
Read String is :  Python is
Current file position:  10
Again read String is :  Python is
```

Os Modules:

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

1. The rename() Method

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

Syntax: os.rename(current_file_name, new_file_name)

Example 1:

```
import os
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

2. The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax:os.remove(file_name)

Example 1:

```
import os
# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

1. The *mkdir()* Method

We can use the *mkdir()* method of the **os** module to create directories in the current directory.

Syntax:os.mkdir("newdir")

Example 1:

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

2. The *chdir()* Method

We can use the *chdir()* method to change the current directory

Syntax:os.chdir("newdir")

Example 1:

```
import os
# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

3. The *getcwd()* Method

The *getcwd()* method displays the current working directory.

Syntax:os.getcwd()

Example 1:

```
import os
# This would give location of the current directory
os.getcwd()
```

4. The *rmdir()* Method

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

Syntax:os.rmdir('dirname')

Example 1:

```
import os
os.rmdir( "/tmp/test"  )
```

SAMPLE PROGRAMS IN FILES:

1.Calculate the total marks of each student stored in a file.

Name	Rollno	Mark1	Mark2
ARAV	102	99	98
RAM	103	98	78

SOLUTION:

```
f=open("d.txt",'r')
sum=0
for i in f:
    sum=0
    x,y,z,a=i.split()
    print(x,y,z,a)
    if z.isnumeric() and a.isnumeric():
        print("name",x,"roll",y)
        sum=sum+int(z)+int(a)
        print(sum)
```

2. Write a python code to write multiple lines to a file.

SOLUTION:

```
fh = open(“hello.txt”,”w”)
lines_of_text = [“One line of text here”, “and another line here”, “and yet another here”, “and so on and so forth”]
fh.writelines(lines_of_text)
fh.close()
```

OUTPUT:

[“One line of text here”, “and another line here”, “and yet another here”, “and so on and so forth”]

3. Write a python program to find the longest words.

SOLUTION:

```
 def longest_word(filename):
    with open(filename, 'r') as infile:
            words = infile.read().split()
    max_len = len(max(words, key=len))
    return [word for word in words if len(word) == max_len]

print(longest_word('test.txt'))
```

OUTPUT:

PYTHON PROGRAMMING

4.Program to find number of lines,characters,words in a file

SOLUTION:

```
f=open("file3.txt",'w')
f.write("Karpagam college of engineering\n")
f.write("Karpagam college")
f.close()
f=open("file3.txt",'r')
```

```
print(f.read())
f.seek(0)
l=f.readlines()
print("lines:",len(l))
wcount=chcount=0
for i in l:
   word=i.split()
   wcount+=len(word)
   for i in word:
      chcount+=len(i)

print("words:",wcount)
print("characters:",chcount)
```

OUTPUT:

```
>>>
Karpagam college of engineering
Karpagam college
lines: 2
words: 6
characters: 43
```

PICKLE:

- It is used for serializing and de-serializing a Python object structure.
- Pickling is a way to convert a python object (list, dict, etc.) into a character stream
- import the module through this command:
  import pickle
- pickle has two main methods:dump() and load().
- Dump()-Used to dumps an object to a file object
- Load()-Used to loads an object from a file object.

Syntax:pickle.dump(value,fileobject)
Variable=pickle.load(fileobject)

Example 1:

```
import pickle
f1=open("f1.txt","wb")
l1=[10,20,30]
l2=[40,50]
l_obj=(l1,l2)
pickle.dump(l_obj,f1)
f1.close()
f2=open("f1.txt","rb")
#(list1,list2)=pickle.load(f2)
#print("in list",list1,list2)
liste=pickle.load(f2)
print("in tuple",liste)
```

f2.close()
Output:

```
>>>
in tuple ([10, 20, 30], [40, 50])
>>>
```

Example 2:

```
import pickle
fo=open("f1.txt","wb")
l1=[10,20,30]
pickle.dump(l1,fo)
fo.close()
f1=open("f1.txt","rb")
l2=pickle.load(f1)
print(l2)
```

Output:

```
>>>
in tuple ([10, 20, 30], [40, 50])
>>>
```

SHELVE:
Example 1:

```
import shelve
s=shelve.open("file1.txt","c")
l1=[10,20]
l2=[30,40]
s["lis1"]=l1
s["lis2"]=l2
for i,j in s.items():
   print (i,j)
```

OUTPUT:

```
>>>
lis1 [10, 20]
lis2 [30, 40]
>>>
```

MODULES AND PACKAGES

Module:

- A module allows you to logically organize your Python code.
- As the program gets longer,we may split several fies for easier maintenance
- Python put definitions into file and use them in a script such a file is called module.

Example 1:

```
def fibonacci(n):
```

```
    a,b=0,1
    while b<n:
        print(b)
        a,b=b,a+b

import M1
M1.fibonacci(5)
```

M1.py

```
def display():
    print("Python Program")
def square(len):
    print(len*len)
```

M2.py

```
import M1
from M1 import display,square
display()
square(5)
```

OUTPUT:
Fibonacci:120
Python program
Square:25

From external Path-if two files are in different directories then import sys

Example 1:

```
import sys
sys.path.append("C:\M1")
import M1 as cal
cal.display()
cal.square(5)
```

OUTPUT:
Python program
Square:25

Reload

Example 1:

```
import M1
import imp
from M1 import display,square
display()
square(5)
imp.reload(M1)
```

display()
square(5)

EXCEPTION HANDLING

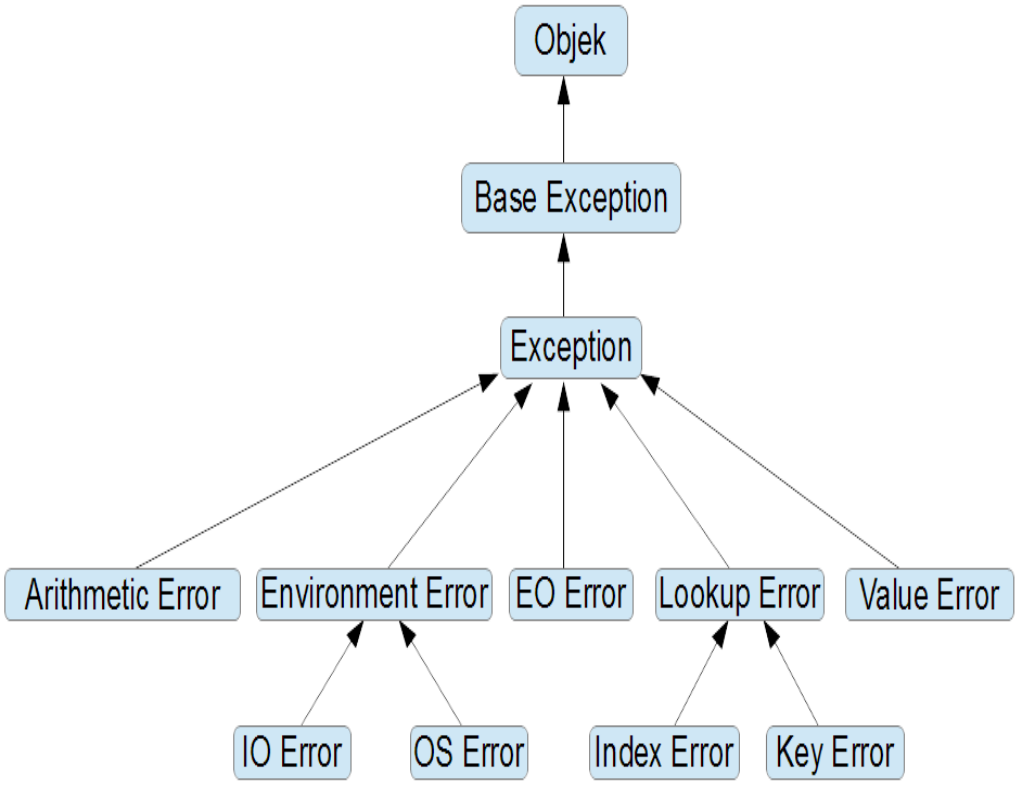
Definition:

- Exception can be said to be any abnormal condition in a program resulting to the disruption in the flow of the program.
- Two important features

1. **Exception handling**-Error that happens during execution of program, non programmers view. Exception is an object
2. **Assertions**

- Used to handle unexpected error in python program.

EXCEPTION HIERARCHY:



Example 1:
Without Exception:

```
a=10
b=20
print (a+b)
print (b+2)
```

With Exception:

```
a=10
b=20
print (a+b)
try:
   print (a/0)
except:
   print("Exception")
print (b+2)
```

OUTPUT:

```
>>>
30
Exception
22
```

Few Standard Exceptions:

Exception Name	Description
Exception	Base class for all exceptions
Arithmetic Error	Base class for all errors that occur for numeric calculation
Floating Point Error	Raised when a floating point calculation fails.
Zero Division Error	Raised when division or modulo by zero takes place for all numeric types.
IO Error	Raised when an input / output operation fails, such as print() or open() functions when trying to open a file that does not exist.
Syntax error	Raised when there is a error on Python syntax
Indentation error	Raised when indentation is not specified properly
Value Error	Raised when built-in-function for a data type has a valid type of arguments, but the arguments have invalid values specified
Runtime Error	Raised when a generated error does not fall into any category

Handling an Exception:

• Exception is an event, which occurs during the execution of program and disrupts the normal flow of program‟s instructions.

• When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

• If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block.

• After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

• Different ways of Exception Handling in Python are:

– try....except...else

– try...except

– try...finally

1.try...except...else

– A single try statement can have multiple except statements

– Useful when we have a try block that may throw different types of exceptions

– Code in else-block executes if the code in the try: block does not raise an exception

Syntax:

```
try:
You do your operations here;
.....................
except ExceptionA:
If there is ExceptionA, then execute this
block.
except ExceptionB:
If there is ExceptionB, then execute this
block.
.....................
else:
If there is no exception then execute
this block
```

Example 1:

```
try:
        fh = open("testfile", "w")
```

```
        fh.write("This is my test file")
except IOError:
        print ("Error: can\'t find file or read data")
else:
        print ("Written content to file successfully")
        fh.close()
```

Output:
Written content to file successfully

2.try...except..

– Catches all exceptions that occur

– It is not considered as good programming practice though it catches all exceptions as it does help the programmer in identifying the root cause of the problem that may occur.

Syntax:

```
try:
You do your operations here;
.....................
except ExceptionA:
If there is ExceptionA, then execute this
block.
except ExceptionB:
If there is ExceptionB, then execute this
block.
.....................
```

Example 1:

```
try:
        fh = open("testfile", "w")
        fh.write("This is my test file for exceptionhandling!!")
except IOError:
        print ("Error: can\'t find file or write data")
```

Output:
Error: can't find file or write data

3.try...finally..

– finally block is a place to put any code that must execute irrespective of try-block raised an exception or not.

– else block can be used with finally block

Syntax:

```
try:
You do your operations here;
.....................
```

Due to any exception, this may be
skipped.
finally:
This would always be executed.
.....
Example 1:

```
try:
        fh = open("testfile", "w")
        fh.write("This is my test file for exceptionhandling!!")
finally:
        print("Error: can\'t find file or write data")
```

Output:
Error: can't find file or write data

SAMPLE PROGRAMS :
Example for Type Error 1:

```
try:
    A=5+'T'
except TypeError:
    print("not possible")
finally:
    print("final")
```

Output: unsupported operand type(s) for +: 'int' and 'str'

Example for Key Error 2:

```
try:
    d={1:"one",2:"three"}
    print(d[3])
except KeyError:
    print("not possible")
finally:
    print("final")
```

```
try:
    a=10/0
except ArithmeticError:
    print("not possible")
finally:
    print("final")
```

Example for Zero Division Error 3:

Example for Index Error 4:

```
try:
    s="hello"
    print(s[8])
except IndexError:
    print("not possible")
finally:
    print("final")
```

Example for Value Error 5:

```
while True:
    try:
        n=input("integer")
        n=int(n)
        break
    except ValueError:
        print("no valid integer")
```

Example 6:

```
try:
   a=10/0;
except:
   print "Arithmetic Exception"
else:
   print "Successfully Done"
```

Output:

```
>>>
Arithmetic Exception
>>>
```

Example 7:

```
try:
   a=10/0;
   print ("Exception occurred")
finally:
```

```
print ("Code to be executed"  )
```

Output:

```
Code to be executed
Traceback (most recent call last):
  File "/home/students/Desktop/1111.py", line 2, in <module>
    a=10/0;
ZeroDivisionError: division by zero
```

Example 8:

```
try:
   raise NameError("hai")
   print ("Exception occurred")
except NameError as e:
   print (e )
```

output:

```
>>>
hai
```

Explanation:

i) To raise an exception, raise statement is used. It is followed by exception class name.

ii) Exception can be provided with a value that can be given in the parenthesis. (here, Hello)

iii) To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

Custom Exception:

Creating your own Exception class or User Defined Exceptions are known as Custom Exception.

Example 1:

```
class Invalid(Exception):
   def check(self,age):
      self.age=age
      if age<18:
         print("sorry")
try:

      raise Invalid()
except Invalid as e:
       e.check(12)
```

Output:

```
>>>
sorry
>>>
```

Assertion:

Way of telling a program to test a condition and trigger an error if condition is false

Syntax: Assert<condition>,<optional message>

Example 1:

```
def KelvinToFahrenheit(Temperature):
  assert (Temperature >= 0),"Colder than absolute zero!"
  return ((Temperature-273)*1.8)+32

print(KelvinToFahrenheit(273))
print(int(KelvinToFahrenheit(505.78)))
print(KelvinToFahrenheit(-5))
```

Output:

```
>>>
32.0
451
```

sample programs:

1.Program to create custom Exception InvalidLogin.

SOLUTION:

```
class InvalidLogin(Exception):
    def disp(self):
        print("Invalid Username/Password")
uname=input("Enter the Username")
password=input("Enter the password")
try:
    if uname=='admin' and password=='admin':
        print("Admin successfully Logged IN!!!")
    else:
        raise InvalidLogin()

except InvalidLogin as e:
    e.disp()

finally:
    print("Thank you")
```

output:

```
>>>
Enter the UsernameRAM
Enter the passwordRAMM
Invalid Username/Password
Thank you
>>>
```

2.Program to create custom Exception InvalidAge

Solution:

```
class InvalidAge(Exception):
    def __init__(self):
        print("Age should be greater than 18")

age=int(input("Enter the age"))
try:
    if age>18:
        print("You will be able to vote")
    else:
        raise InvalidAge()
except InvalidAge as e:
    print(e)
finally:
    print("Thank you")
```

Output:

```
Enter the age24
You will be able to vote
Thank you
```