Python Functions

A Function is a self block of code.

A Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code re usability

A Function is a subprogram that works on data and produce some output.

Types of Functions:

There are two types of Functions.

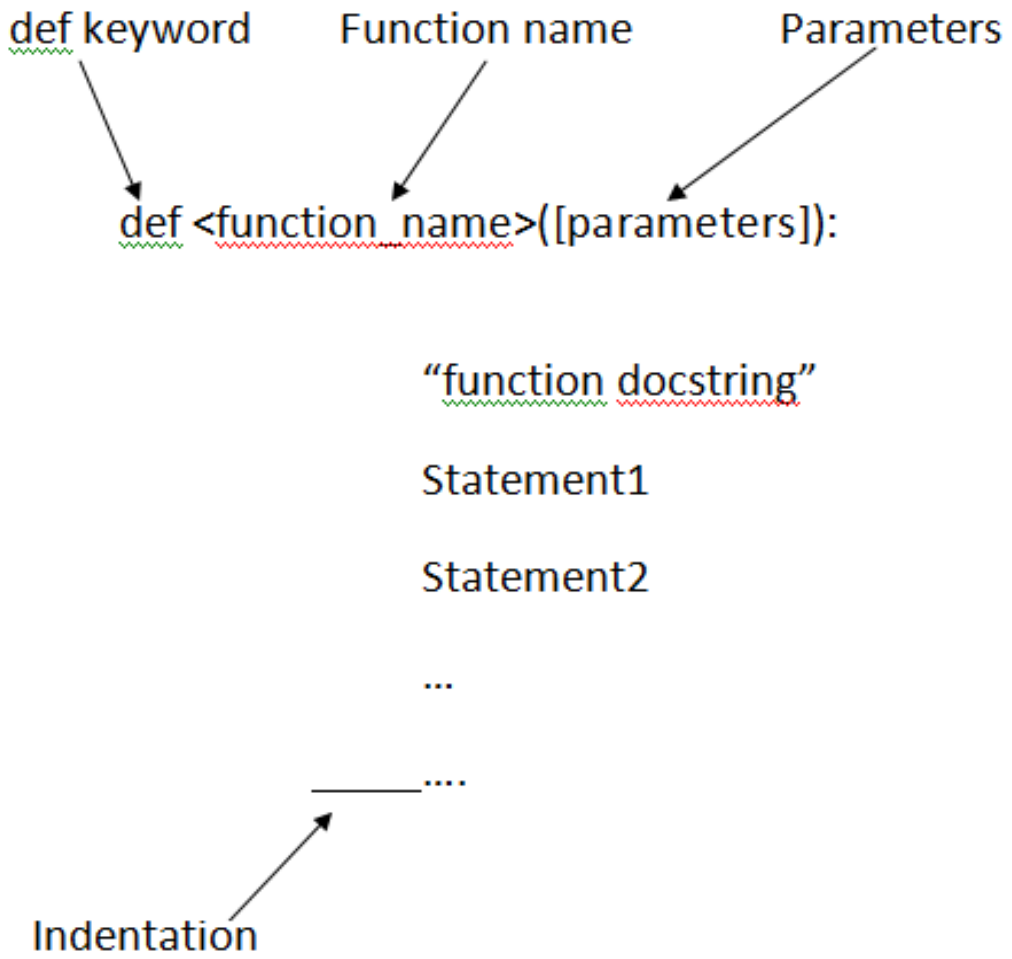
a) **Built-in Functions:** Functions that are predefined. We have used many predefined functions in Python.

b) **User- Defined**: Functions that are created according to the requirements.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on large project can divide the workload by making different functions.

Syntax of Function




```
Note : function docstring can be used like below also
                        """docstring"""
```

Function definition consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Docstring

The first string after the function header is called the docstring and is short for documentation string.

It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines.

This string is available to us as __doc__ attribute of the function.

For example:

```
def greet(name):
    """This function greets to
    the person passed in as
    parameter"""
    print("Hello, " + name + ". Good morning!")


>>> print(greet.__doc__)
This function greets to
    the person passed into the
    name parameter
```

Function Call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Ram')
Hello, Ram. Good morning!
```

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

return [expression_list]

This statement can contain expression which gets evaluated and the value is returned.

If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Example of a user-defined function

```
# Program to illustrate
# the use of user-defined functions

def my_addition(x,y):
   """This function adds two
   numbers and return the result"""
   sum = x + y
   return sum
num1 = float(input("Enter a number: "))
num2 = float(input("Enter another number: "))
print("The sum is", my_addition(num1,num2))
```

```
Output
Enter a number: 2.4
Enter another number: 6.5
The sum is 8.9
```

In the above example, input(), print() and float() are built-in functions of the Python programming language.

Argument and Parameter:

There can be two types of data passed in the function.

1) The First type of data is the data passed in the function call. This data is called ? arguments?.

2) The second type of data is the data received in the function definition. This data is called ? parameters?.

Arguments can be literals, variables and expressions.

Parameters must be variable to hold incoming values.

Alternatively, arguments can be called as actual parameters or actual arguments and parameters can be called as formal parameters or formal arguments.

Eg:

```
def addition(x,y):
        print x+y
x=15
addition(x ,10)
addition(x,x)
y=20
addition(x,y)
```

Output:

```
>>>
25
30
35
>>>
```

Passing Parameters

Apart from matching the parameters, there are other ways of matching the parameters.

Python supports following types of formal argument:

1) Positional argument (Required argument).

2) Default argument.

3) Keyword argument (Named argument)

4) Arbitrary Arguments

Positional / Required Arguments:

When the function call statement must match the number and order of arguments as defined in the function definition it is Positional Argument matching.

```
#Function definition of sum
def sum(a,b):
        "Function having two parameters"
    c=a+b
        print c


sum(10,20)
sum(20)
```

Output:

```
>>>
30
Traceback (most recent call last):
  File "C:/Python27/su.py", line 8, in <module>
    sum(20)
TypeError: sum() takes exactly 2 arguments (1 given)
>>>
```

Explanation:

1) In the first case, when sum() function is called passing two values i.e., 10 and 20 it matches with function definition parameter and hence 10 and 20 is assigned to a and b respectively. The sum is calculated and printed.

2) In the second case, when sum() function is called passing a single value i.e., 20 , it is passed to function definition. Function definition accepts two parameters whereas only one value is being passed, hence it will show an error.

Default Arguments

Default Argument is the argument which provides the default values to the parameters passed in the function definition, in case value is not provided in the function call.

Eg:

```
def greet(name, msg = "Good morning!"):
    """This function greets to
    the person with the provided message.
    If message is not provided, it defaults
    to "Good morning!" """

    print("Hello",name + ', ' + msg)

greet("Ram")
greet("Sham","How do you do?")
```

Output

```
('Hello', 'Ram, Good morning!')
('Hello', 'Sham, How do you do?')
```

Keyword Arguments:

Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

Eg:

```
def msg(id,name):
        "Printing passed value"
            print id
            print name
            return
msg(id=100,name='Raj')
msg(name='Rahul',id=101)
```

Output:

```
>>>
100
Raj
101
Rahul
>>>
```

Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.

Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello",name)


greet("Ram","Sham","Bhim")
```

Output

```
('Hello', 'Ram')
('Hello', 'Sham')
('Hello', 'Bhim')
```

Anonymous Function:

Anonymous Functions are the functions that are not bond to name.

Anonymous Functions are created by using a keyword "lambda".

Lambda takes any number of arguments and returns an evaluated expression.

Lambda is created without using the def keyword.

Syntax:

lambda arg1,args2,args3,....,argsn :expression

```
#Function Definiton
Pro=lambda x1,y1: x1*y1

#Calling square as a function
print "Product of numbers is",Pro(10,2)
```

Output:

```
>>>
Product of numbers is 20
>>>
```

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized.

Lifetime of a variable is the period throughout which the variable exits in the memory.

There are two types of variables based on Scope:

1) Local Variable.

2) Global Variable.

1) Local Variables:

Variables declared inside a function body is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Eg:

```
def msg():
        a=10
        print "Value of a is",a
        return
msg()
 print a #it will show error since variable is local
```

Output:

```
>>>
Value of a is 10
  Traceback (most recent call last):
    File "C:/Python27/lam.py", line 7, in <m
odule>
     print a #it will show error since variable
 is local
NameError: name 'a' is not defined
```

2) Global Variable:

Variable defined outside the function is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.

Eg:

```
b=20
def msg():
        a=10
        print "Value of a is",a
        print "Value of b is",b
        return

        msg()
        print b
```

Output:

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

Python Recursion

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Example of recursive function

```
# An example of a recursive function to
# find the factorial of a number

def recur_fact(x):
   """This is a recursive function
   to find the factorial of an integer"""

   if x == 1:
       return 1
   else:
       return (x * recur_fact(x-1))

num = int(input("Enter a number: "))

if num >= 1:
   print("The factorial of", num, "is", recur_fact(num))
```

Explanation

In the above example, recur_fact() is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number-1 until the number is equal to one.

This recursive call can be explained in the following steps.

```
recur_fact(4)                # 1st call with 4
4 * recur_fact(3)            # 2nd call with 3
4 * 3 * recur_fact(2)        # 3rd call with 2
4 * 3 * 2 * recur_fact(1)    # 4th call with 1
4 * 3 * 2 * 1                # retrun from 4th call as number=1
4 * 3 * 2                    # return from 3rd call
4 * 6                        # return from 2nd call
24                           # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely. We must avoid infinite recursion.

Advantages of recursion

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

Disadvantages of recursion

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

Python Modules

- Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.
- We use modules to break down large programs into small manageable and organized files.
- Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

```
# Python Module example

def add(a, b):
    """This program adds two
    numbers and return the result"""
    result = a + b
    return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

Importing modules

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the **import** keyword to do this. To import our previously defined module example we type the following in the Python prompt.

```
>>> import example
```

This does not enter the names of the functions defined in example directly in the current symbol table.

It only enters the module name example there. Using the module name we can access the function using dot (.) operation.

For example:

>>> example.add(4,5.5)

9.5

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) into a list of directories defined in sys.path. The search is in this order.

- The current directory.
- PYTHONPATH (an environment variable with a list of directory).
- The installation-dependent default directory.

```
>>> import sys
>>> sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can add modify this list to add our own path.

append() method can be used to set our own path to sys path.

Ways to import modules.

1) The import statement

We can import a module using import statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example
# to import standard module math

import math
print("The value of pi is", math.pi)

Output
The value of pi is 3.141592653589793
```

2) Import with renaming

We can import a module by renaming it as follows.

```
import math as m
print("The value of pi is", m.pi)
```

The output of this is same as above. We have renamed the math module as m. This can save us typing time in some cases. Note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

3) The from...import statement

We can import specific names form a module without importing the module as a whole.

```
# import only pi from math module
from math import pi
print("The value of pi is", pi)
```

```
The output of this is same as above. We imported only the attribute pi
form the module. In such case we don't use the dot operator.
We could have imported multiple attributes as follows.
      >>> from math import pi, e
      >>> pi
      3.141592653589793
      >>> e
      2.718281828459045
```

4) Import all names

We can import all names(definitions) form a module using the following construct.

```
from math import *   # import all names form the standard module math
print("The value of pi is", pi)
```

```
Importing everything with the asterisk (*) symbol is not a good
programming practice.

This can lead to duplicate definitions for an identifier. It also
hampers the readability of our code.
```

Consider the Module prnt.py

```
def sum1(x, y):
        """ This will find the sum"""
        x=x+y
        return x


def recur_fact(x):
   """This is a recursive function
   to find the factorial of an integer"""

   if x == 1:
       return 1
   else:
       return (x * recur_fact(x-1))

num = int(input("Enter a number: "))
num2 = int(input("Enter a number: "))

if num >= 1:
   print("The factorial of", num, "is", recur_fact(num))


print("sum is", sum1(num,num2))
```

Example: Importing prnt module methods

```
>>> import sys
>>> sys.path.append('/home/nmamit/Python')
```

```
>>> import prnt
Enter a number: 4
Enter a number: 5
('The factorial of', 4, 'is', 24)
('sum is', 9)
```

```
>>> import prnt as P
>>> P.recur_fact(6)
720
>>> P.sum1(5,5)
10
>>>
```

```
>>> from prnt import sum1
>>> sum1(5,6)
11
```

```
>>> from prnt import *
>>> recur_fact(5)
120
>>> sum1(6,7)
13
>>>
```

```
>>> from prnt import recur_fact, sum1
>>> recur_fact(3)
6
>>> sum1(7,8)
15
```

Reloading a module

The Python interpreter imports a module only once during a session. This makes things more efficient.

Now if our module changed during the course of the program, we would have to reload it.

One way to do this is to restart the interpreter. But this does not help much.

Python provides a neat way of doing this. We can use the **reload()** function to reload a module.

Syntax: reload(module_name)

For example assume that the following function definition is added to the module prnt.

```
def deff(x, y):
    return x-y
print("Difference is", deff(num,num2))
print("..... Thank u.....");
```

Then when you import prnt the output will be..

```
>>> import prnt
>>>
```

So functions will not run..and therefore we need to reload the module as

```
>>> reload(prnt)
Enter a number: 5
Enter a number: 3
('The factorial of', 5, 'is', 120)
('sum is', 8)
('Difference is', 2)
..... Thank u.....
<module 'prnt' from '/home/nmamit/Python/prnt.py'>
>>>
```

Python Package

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages.

This makes a project (program) easy to manage and conceptually clear. Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

A directory must contain a file named __init__.py in order for Python to consider it as a package.

This file can be left empty but we generally place the initialization code for that package in this file.

Importing module from a package

We can import modules from packages using the dot (.) operator

```
>>>from MyPack import prnt

>>> prnt.deff(5,3)

2

>>> from pack import prnt

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ImportError: No module named pack
```

Here a Package called MyPack is created with a file named __init__.py and module prnt is placed inside that package

But pack is a directory without __init__.py file in it. So it is not treated as a package.

Assuming dat u r in previously set path.

```
>>> from MyPack.prnt import deff
>>> deff(5,6)
-1
>>>


>>> from MyPack.prnt import deff
>>> deff(5,6)
-1
>>> from MyPack.prnt import *
>>> sum1(4,5)
9
>>> deff(6,3)
3
>>> recur_fact(5)
120
>>>
```

Function Exercises

1. Python Program To Display Powers of 2 Using Anonymous Function
2. Python Program to Find Numbers Divisible by Another Number
3. Python Program to Convert Decimal to Binary, Octal and Hexadecimal
4. Python Program to Find ASCII Value of Character
5. Python Program to Find GCD
6. Python Program to Find LCM
7. Python Program to Make a Simple Calculator
8. Python Program to Display Calendar (use Python Module)
9. Python Program to Display Fibonacci Sequence Using Recursion
10. Python Program to Find Sum of Natural Numbers Using Recursion
11. Python Program to Find Factorial of Number Using Recursion
12. Python Program to Convert Decimal to Binary Using Recursion

Python Object Oriented

Python is an object oriented programming language. Unlike procedure oriented programming, in which the main emphasis is on functions, object oriented programming stress on objects. Object is simply a collection of data (variables) and methods (functions) that act on those data.

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.

Creating Classes

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows −

```
class ClassName:
    '''Optional class documentation string'''
    class_suite
```

- The class has a documentation string, which can be accessed via ClassName.__doc__.
- The class_suite consists of all the component statements defining class members, data attributes and functions.
- A class creates a new local namespace where all its attributes are defines.
- Attributes may be data or functions.
- There are also special attributes in it that begins with double underscores (__).
- For example, __doc__ gives us the docstring of that class.
- As soon as 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.

Creating an Object in Python

To create instances of a class, you call the class using class name

Syntax: **objectname = Classname()**

This will create a new instance object named objectname.

Accessing Attributes

You access the object's attributes(data member or function) using the dot operator with object.

Syntax: **objectname.datamemeber**

Objectname.functionname(parameters)

You declare other class methods like normal functions with the exception that the first argument to each method is ***self***.

Python adds the self argument to the list for you; you do not need to include it when you call the methods .

Therefore even if your function does not take any parameter, we have to pass ***self*** as a parameter to the function.

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def read(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee()

name=raw_input("Enter the name")
salary=input("Enter the salary")
emp1.read(name,salary)

"This would create second object of Employee class"
emp2 = Employee()
emp2.read("Sham", 50000)

emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Output:

```
Enter the name Ram
Enter the salary 40000
Name :   Ram , Salary:  40000
Name :  Sham , Salary:  50000
Total Employee 2
```

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: Consider previous example class Employee.

```
print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

Output:

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount': <function displayCount at
0xb7496684>, 'read': <function read at 0xb748de9c>, 'empCount': 2, 'displayEmployee':
<function displayEmployee at 0xb7496294>, '__doc__': 'Common base class for all
employees'}
```

Constructors in Python

Class functions that begins with double underscore (__) are called special functions as they have special meaning.

Of one particular interest is the __init__() function.

This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP).

We normally use it to initialize all the variables.

Synatx:

```
class Classname:
    def __init__(self,otherparameters):
        Initialization statements
```

The following demonstrates the use of constructors and also **passing objects as parameters and returning object concepts**

Example:

```
class ComplexCompute:
    def __init__(self, realPart, imagPart):
        self.realPart = realPart
        self.imagPart = imagPart

    def add(self, other):
        resultR = self.realPart+other.realPart
        resultI = self.imagPart+other.imagPart
        result = complex(resultR, resultI)
        return result

    def sub(self, other):
        resultR = self.realPart-other.realPart
        resultI = self.imagPart-other.imagPart
        result = complex(resultR, resultI)
        return result

c1 = ComplexCompute(2,2)
c2 = ComplexCompute(1,1)

print "sum is:", c1.add(c2)
print "Difference is:",c1.sub(c2)
```

Output:

```
sum is: (3+3j)
Difference is: (1+1j)
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space.

The process by which Python periodically reclaims blocks of memory that no longer are in use is termed **Garbage Collection**.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.

An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary).

The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope.

When an object's reference count reaches zero, Python collects it automatically.

Example:

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space.

But a class can implement the special method **__del__()**, called a **destructor**, that is invoked when the instance is about to be destroyed.

This method might be used to clean up any non memory resources used by an instance.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed

```
class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1

print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

When the above code is executed, it produces following result −

```
3083401324 3083401324 3083401324
Point destroyed
```

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class.

A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
    class_suite
```

Example

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

```
#calling parent class constructor through child constructor

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self,a):
      print "Calling parent constructor"
      Parent.parentAttr=a

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr


class Child(Parent): # define child class
   def __init__(self,parent):
      print "Calling child constructor"
      Parent.__init__(self,parent)

   def childMethod(self):
      print 'Calling child method'
```

```
c = Child(300)          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.getAttr()
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Output

```
Calling child constructor
Calling parent constructor
Calling child method
Calling parent method
Parent attribute : 300
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....
class B:         # define your calss B
.....
class C(A, B):   # subclass of A and B
.....
```

You can use **issubclass()** or **isinstance()** functions to check a relationships of two classes and instances.

The **issubclass(sub, sup)** boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.

The **isinstance(obj, Class)** boolean function returns true if obj is an instance of class Class or is an instance of a subclass of Class

Example:consider the previous example

```
print issubclass(Child,Parent)
print isinstance(c,Child)
```

Output:

```
True
True
```

Data Hiding

An object's attributes may or may not be visible outside the class definition.

You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

Python protects those members by internally changing the name to include the class name.

You can access such attributes as ***object._className__attrName.***

Example:

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount


    def __hidden(self):
         print "Am hidden..."


    def accesshidden(self):
        self.__hidden()
```

```
counter = JustCounter()
counter.count()
counter.count()
print counter._JustCounter__secretCount  #Alternative way to access hidden variable
counter.accesshidden()
counter._JustCounter__hidden()           #Alternative way to access hidden method

#The following statements will give error
counter.__hidden()
print counter.__secretCount
```

Output

```
1
2
2
Am hidden...
Am hidden...
Traceback (most recent call last):
  File "opoverload.py", line 77, in <module>
    counter.__hidden()
AttributeError: JustCounter instance has no attribute '__hidden'
```

Data Encapuslation

If an identifier doesn't start with an underscore character "_" it can be accessed from outside, i.e. the value can be read and changed. Data can be protected by making members private or protected. Instance variable names starting with two underscore characters cannot be accessed from outside of the class. At least not directly, but they can be accessed through private name mangling. That means, private data __A can be accessed by the following name construct: instance_name._classname__A. If an identifier is only preceded by one underscore character, it is a protected member. Protected members can be accessed like public members from outside of class. This is illustrated in the following example:

example

```
class Encapsulation(object):
    def __init__(self, a, b, c):
        self.public = a               #public variable no two undersccore
        self._protected = b            # protected varible one underscore
        self.__private = c             #private variable two underscore
```

Inheritance

Single level

example

```
class Instrument:

    def __init__(self, name):

        self.name = name

    def has_strings(self):

        return True

class PercussionInstrument(Instrument):

    def has_strings(self):

        return False

guitar = Instrument('guitar')

drums = PercussionInstrument('drums')
```

```
print ('Guitar has strings: {0}'.format(guitar.has_strings()))
print ('Guitar name: {0}'.format(guitar.name))
print ('Drums have strings: {0}'.format(drums.has_strings()))
print ('Drums name: {0}'.format(drums.name))
```

Multilevel inheritance

example

```
class Instrument:
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True
class StringInstrument(Instrument):
    def __init__(self, name, count):
        super(StringInstrument, self).__init__(name)
        self.count = count
class Guitar(StringInstrument):
    def __init__(self):
        super(Guitar, self).__init__('guitar', 6)
guitar = Guitar()
print ('Guitar name: {0}'.format(guitar.name))
print ('Guitar count: {0}'.format(guitar.count))
```

Multiple inheriatance

example

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True
class Analyzable(object):
    def analyze(self):
        print ('I am a {0}'.format(self.__class__.__name__))
class Flute(Instrument, Analyzable):
    def has_strings(self):
        return False
flute = Flute('flute')
flute.analyze()
```

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

```
class Parent:        # define parent class
  def myMethod(self):
    print ('Calling parent method')

class Child(Parent): # define child class
  def myMethod(self):
    print ('Calling child method')

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result −

```
Calling child method
```

abstract class

example

```
from abc import ABCMeta, abstractmethod
import sys
import traceback
class Instrument(object):
    __metaclass__ = ABCMeta
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def has_strings(self):
        pass
class StringInstrument(Instrument):
    def has_strings(self):
return True
guitar = StringInstrument('guitar')
print ('Guitar has strings: {0}'.format(guitar.has_strings()))
try:
    guitar = Instrument('guitar')
except:
    traceback.print_exc(file=sys.stdout)
```

Persistent storage of objects

It is used for serializing and de-serializing a Python object structure. Any object in python can be pickled so that it can be saved on disk. What pickle does is that it “serialises” the object first before writing it to file. Pickling is a way to convert a python object (list, dict, etc.) into a character stream. The idea is that this character stream contains all the information necessary to reconstruct the object in another python script. The pickle.load() function takes a stream object, reads the serialized data from the stream, creates a new Python object, recreates the serialized data in the new Python object, and returns the new Python object.

Example

```
import pickle

a = 'test value’

fileObject = open("sample",'wb')

# this writes the object a to the

# file named 'testfile'

pickle.dump(a,fileObject)

# here we close the fileObject

fileObject.close()

# we open the file for reading

fileObject = open("sample",'rb')

# load the object from the file into var b

while True:

    try:

        b1 = pickle.load(fileObject)

        print(b1)

    except EOFError:

            fileObject.close()

            break
```

Python 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.

This feature in Python, that allows same operator to have different meaning according to the context is called **operator overloading**.

So what happens when we use them with objects of a user-defined class?

Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

```
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
```

Now when we try to add two points that we create as follows.

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> p1 + p2

Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

TypeError was raised since Python didn't know how to add two Point objects together.

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__()

Example:

```
class Complex1:
    def __init__(self, realPart=0, imagPart=0):
        self.realPart = realPart
        self.imagPart = imagPart

    def __add__(self, other):

        resultR = self.realPart+other.realPart
        resultI = self.imagPart+other.imagPart
        result = Complex1(resultR, resultI)
        return result

    def __sub__(self, other):
        resultR = self.realPart-other.realPart
        resultI = self.imagPart-other.imagPart
        result = Complex1(resultR, resultI)
        return result

c1 = Complex1(2,3)
c2 = Complex1(1,4)
c3 = Complex1()
c4 = Complex1()

c3 = c1+c2
print "sum is:",complex(c3.realPart,c3.imagPart)
c4 = c1 - c2
print "Difference is:",complex(c4.realPart,c4.imagPart)
```

Output:

```
sum is: (3+7j)
Difference is: (1-1j)
```

Eample 2

```
class Point:
    # previous definitions...
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
p1 = Point(2,3)
p2 = Point(-1,1)
p3=p1 + p2
print(p3)
```

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:

```
class test:
	def __init__(self,a):
		self.a=a

	def __gt__(self,other):
		if self.a > other.a:
			return True

		else:
			return False


t1=test(15)
t2=test(70)

print t1>t2
print t2>t1
```

Output:

```
False
True
```

Example

Overloading == operator(__eq__)

```
class test:
    def __init__(self,a):
        self.a=a
    def __eq__(self,other):
        if self.a ==other.a:
            return True
        else:
            return False
t1=test(15)
t2=test(15)
print (t1==t2)
```

Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

SN	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__repr__(self)** Evaluatable string representation Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*


Thank You

EXCEPTION HANDLING

- Exception can be said to be any abnormal condition in a program resulting to the disruption in the flow of the program.
- Whenever an exception occurs the program halts the execution and thus further code is not executed. Thus exception is that error which python script is unable to tackle with.
- Exception in a code can also be handled. In case it is not handled, then the code is not executed further and hence execution stops when exception occurs.

- Hierarchy Of Exception:

  ZeroDivisionError: Occurs when a number is divided by zero.

  NameError: It occurs when a name is not found. It may be local or global.

  IndentationError: If incorrect indentation is given.

  IOError: It occurs when Input Output operation fails.

  EOFError: It occurs when end of file is reached and yet operations are being performed

etc..

Exception Handling:

The suspicious code can be handled by using the try block. Enclose the code which raises an exception inside the try block. The try block is followed except statement. It is then further followed by statements which are executed during exception and in case if exception does not occur.

Syntax:

try:

malicious code

except Exception1:

execute code

except Exception2:

execute code

....

....

except ExceptionN:

execute code

else:

In case of no exception, execute the else block code.

eg:

```
try:
    a=10/0
    print a
except ArithmeticError:
        print "This statement is raising an exception"
else:
    print "Welcome"
```

Output:

```
>>>
This statement is raising an exception
>>>
```

Explanation:

- The malicious code (code having exception) is enclosed in the try block.
- Try block is followed by except statement. There can be multiple except statement with a single try block.
- Except statement specifies the exception which occurred. In case that exception is occurred, the corresponding statement will be executed.
- At the last you can provide else statement. It is executed when no exception is occurred.

Except with no Exception:

Except statement can also be used without specifying Exception.

Syntax:

try:

code

except:

code to be executed in case exception occurs.

else:

code to be executed in case exception does not occur.

Eg:

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

Output:

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

Declaring Multiple Exception

Multiple Exceptions can be declared using the same except statement:

Syntax:

```
try:
    code
except Exception1,Exception2,Exception3,..,ExceptionN:
    execute this code in case any Exception of these occur.
else:
    execute code in case no exception occurred.
```

eg:

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

Output:

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

Finally Block:

In case if there is any code which the user want to be executed, whether exception occurs or not then that code can be placed inside the finally block.

Finally block will always be executed irrespective of the exception.

Syntax:

try:

Code

finally:

code which is must to be executed.

eg:

```
try:
    a=10/0;
    print "Exception occurred"
finally:
    print "Code to be executed"
```

Output:

```
Code to be executed
Traceback (most recent call last):
  File "C:/Python27/noexception.py", line 2, in <module>
    a=10/0;
ZeroDivisionError: integer division or modulo by zero
```

In the above example finally block is executed. Since exception is not handled therefore exception occurred and execution is stopped.

Raise an Exception:

You can explicitly throw an exception in Python using ?raise? statement. raise will cause an exception to occur and thus execution control will stop in case it is not handled.

Syntax:

raise Exception_class,<value>

eg:

```
try:
    a=10
    print a
    raise NameError("Hello")
except NameError as e:
        print "An exception occurred"
        print e
```

Output:

```
>>>
10
An exception occurred
Hello
>>>
```

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.

Standard Exceptions –

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.

AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError KeyError	Raised when an index is not found in a sequence. Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError EnvironmentError	Raised when trying to access a local variable in a function or method but no value has been assigned to it. Base class for all exceptions that occur outside the Python environment.
IOError IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.

SyntaxError IndentationError	Raised when there is an error in Python syntax. Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Python User-Defined Exception

Users can define their own exception by creating a new class in Python.

This exception class has to be derived, either directly or indirectly, from **Exception class**.

Most of the built-in exceptions are also derived form this class.

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise.

Most implementations declare a custom base class and derive others exception classes from this base class.

This concept is made clearer in the following example.

Example:

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""


class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
```

```
class ValueTooLargeError(Error):
   """Raised when the input value is too large"""


# our main program
# user guesses a number until he/she gets it right
# you need to guess this number

number = 10
while True:
   try:
      i_num = int(input("Enter a number: "))
      if i_num < number:
         raise ValueTooSmallError
      elif i_num > number:
         raise ValueTooLargeError
      break

   except ValueTooSmallError:
      print("This value is too small, try again!")

   except ValueTooLargeError:
      print("This value is too large, try again!")

print("Congratulations! You guessed it correctly.")
```

Output:

```
Enter a number: 6
This value is too small, try again!

Enter a number: 13
This value is too large, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```