Python Programming

Unit 5

Topics to be covered

Object Oriented Programming OOP in Python: Classes, 'self variable', Methods, Constructor Method, Inheritance, Overriding Methods, Datahiding.
Error and Exceptions: Difference between an error and Exception, Handling Exception, try except block, Raising Exceptions, User Defined Exceptions

Objective: Understanding Object Oriented Concepts and exception handling
Outcome: Students are able to **apply** Object Oriented Concepts and exception handling.

Introduction to Object Oriented Programming

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are very easy. The OOP is more value as the program size is growing. The ***procedural programming*** may produce some side affect and have no data security. The main advantage of the OOP is that, the data can be combined with functions that are acting on the data into a single unit. This process is called "***encapsulation***". The basic unit that provides this encapsulation is the "***class***" keyword.

OOP Principles

The four major principles of object oriented programming they are:

Encapsulation
Data Abstraction
Polymorphism
Inheritance

Encapsulation – It is the process of wrapping or binding the data and member function into a single unit.
Data Abstraction – Data Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. This can be achieved by making the data as private or protected.
Polymorphism - The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
Inheritance - The process of acquiring the properties or members of one class to another class.

Class

It is 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.

In Python, everything is an object or an instance of the some class. In general, the class can be defined as follow:

It is a ***Prototype*** from which objects are created.
It is a ***Model*** from which objects are created.
It is a ***Blueprint*** from which objects are created.
It is a ***Template*** from which objects are created.

Defining the class

Defining the class in python is very easy. The class definition can appear anywhere in the program, but usually they are written in the beginning. The definition contains two things: ***class header and class body***. The class header begins with "**class**" keyword followed by the class_name, and colon (:) at the end. The body of the class contains one or more statements. These statements are normally data members and member function. The class variables and methods together called "C***lass Members***". The data members are also called as instance variables.

```
#defining the class
class Employee:  # class header
  class variable
  Data Member1
  Data Member2
  Member function1
  Member function 2
```

Class Body

Where *class* is the keyword, Employee is the class name. The class header contains class keyword, class name and colon (:) operator. The class body contains class variables, data members and member functions.

Defining the _init_ method (constructor)

There is a special method "__init__" which is used to initialize the instance variables or data members of the class. This is also called as "**constructor**". It is defined as follows:

```
def __init__(self, n, s):   # constructor, where n and s are parameters
    self.name=n         #initialization of instance variables- name and sal
    self.sal=s
```

The "__init__" method has one argument "self" and every method has at least one argument that "self". This '***self***' argument is a reference to the current object on which the method is being called. This "__init__" method is called with the help of class constructor.

Adding the Member Functions or Methods to class

We can add any number of functions or methods to the class as we like. The function that is written inside the class is called "***member function or method***". Writing the method is quite similar to the ordinary function with just one difference. The methods must have one argument named as "self". This is the first argument that added to the beginning of parameters list. The method or function definition is written inside the class as shown in the syntax:

```
#Defining class
class Employee:  # class header
    #declaring the class variable
count=0
  #defining the constructor
  def __init__(self,n,s):
    self.name=n
    self.sal=s
    Employee.count+=1
  #adding the method  to the class
  def dispemp(self):
    print("The employee name is:",self.name,"Salary is:",self.sal)
```

self variable

The self argument refers to the current object.

Python takes care of passing the current object as argument to the method while calling.

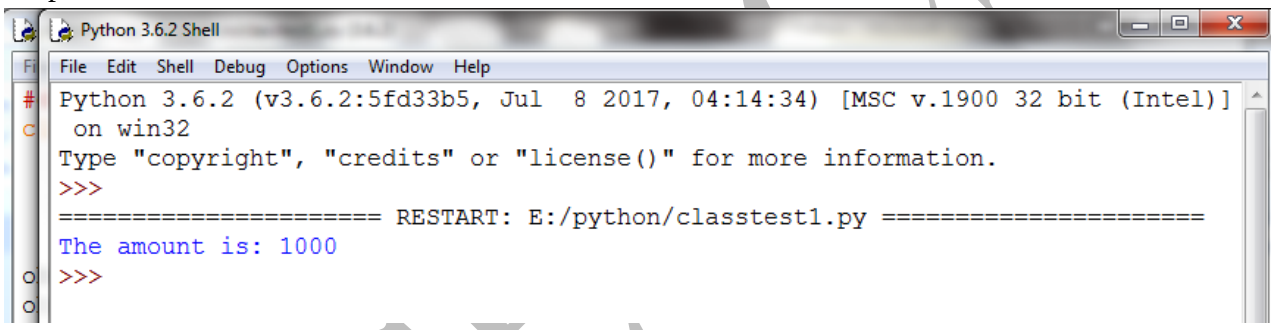
Even if the method does not contain the argument, Python passes this "current object" that called the method as argument, which in turn is assigned to the self variable in the method definition.

Similarly a method defined to take one argument will actually take two arguments: self and parameter.

```
#defining class and creating the object
class ABC:
   def __init__(self,b):
      self.balance=b
   def disp(self):  # method with self argument
      print("The amount is:",self.balance)
ob=ABC(1000);
```

ob.disp() //method is called without argument, python passes 'ob' as argument at the background.

Output:



Creating the Object from the class

The procedure for creating the object is similar to the function call. The class name and the arguments mentioned in the "__init__" method should be specified. The syntax is as follow:

```
#creating the object
emp1=Employee("Ramesh",23000)
```

Where, the **"Employee"** is class name. This is used as constructor name. The actual parameters are passed to the formal parameters present in the __init__ method that in turn assigns to the instance variable. This statement will create a new instance (object) of class, named emp1. We can access the members of objects using the object name as prefix along with dot (.) operator. Creating the object or instance of the class is called "***Instantiation***".

Accessing the members of the object

Once the object is created, it is very easy and straight forward to access the members of the object. The object name, dot operator and member name is used. The syntax is as follow:

```
#accessing the member function
```

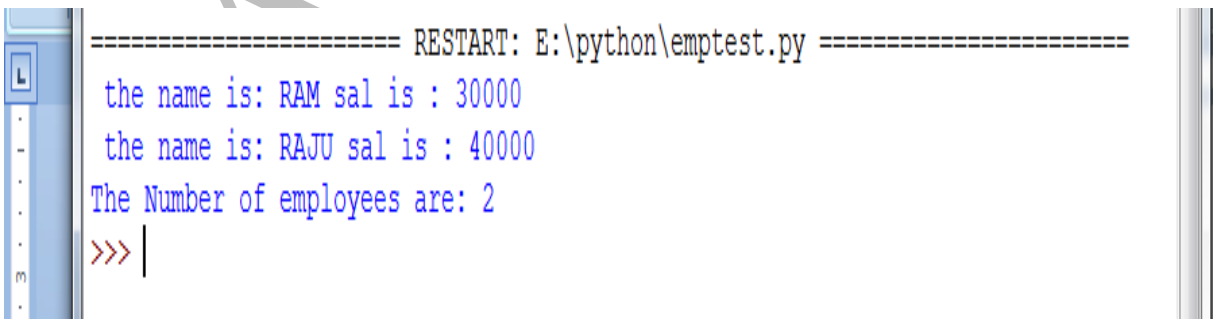
```
emp1.dispemp()
```

Putting all the things together

emptiest.py

```
#defining the class
class Employee:
   'doc string'
   #declare class variables
   count=0
   def __init__(self,n,s): #constructor
      self.name=n
      self.sal=s
      Employee.count+=1
   def dispemp(self):
      print(" the name is:",self.name,"sal is :",self.sal)
#end of the class
#creating object
emp1=Employee("RAM",30000)
emp2=Employee("RAJU",40000)
#access the member function
emp1.dispemp()
emp2.dispemp()
print("The Number of employees are:",Employee.count)
```

Output:



Data Abstraction and Hiding through Classes

Data Encapsulation is also known as Data Hiding. It is the process of binding the data members and member function together into a single unit. This encapsulation defines different ***access*** levels for members of the class. These access levels are specified as follow:

Any data or member function with access level "***public***" can be accessed by any functions belonging to any class. This is the ***lowest*** level of protection.

Any data or member function with access level "***private***" can be accessed by member functions of the same class in which it is declared. This is the ***highest*** level of protection. In Python, private variable are declared with the help of (__) double underscore prefixed to the variable. For example, "__balance" is the private variable.

Class Variable and Instance Variable

A class can have variable defined in it. Basically, these variables are of two types: class variables and instance (object) variable. The class variables are always associated with class and instance variables are always associated with object.

The class variables are shared among all the objects. There exists a single copy of the class variables.

Any change made to the class variable will be seen by all the objects.

The instance variables are not shared between objects.

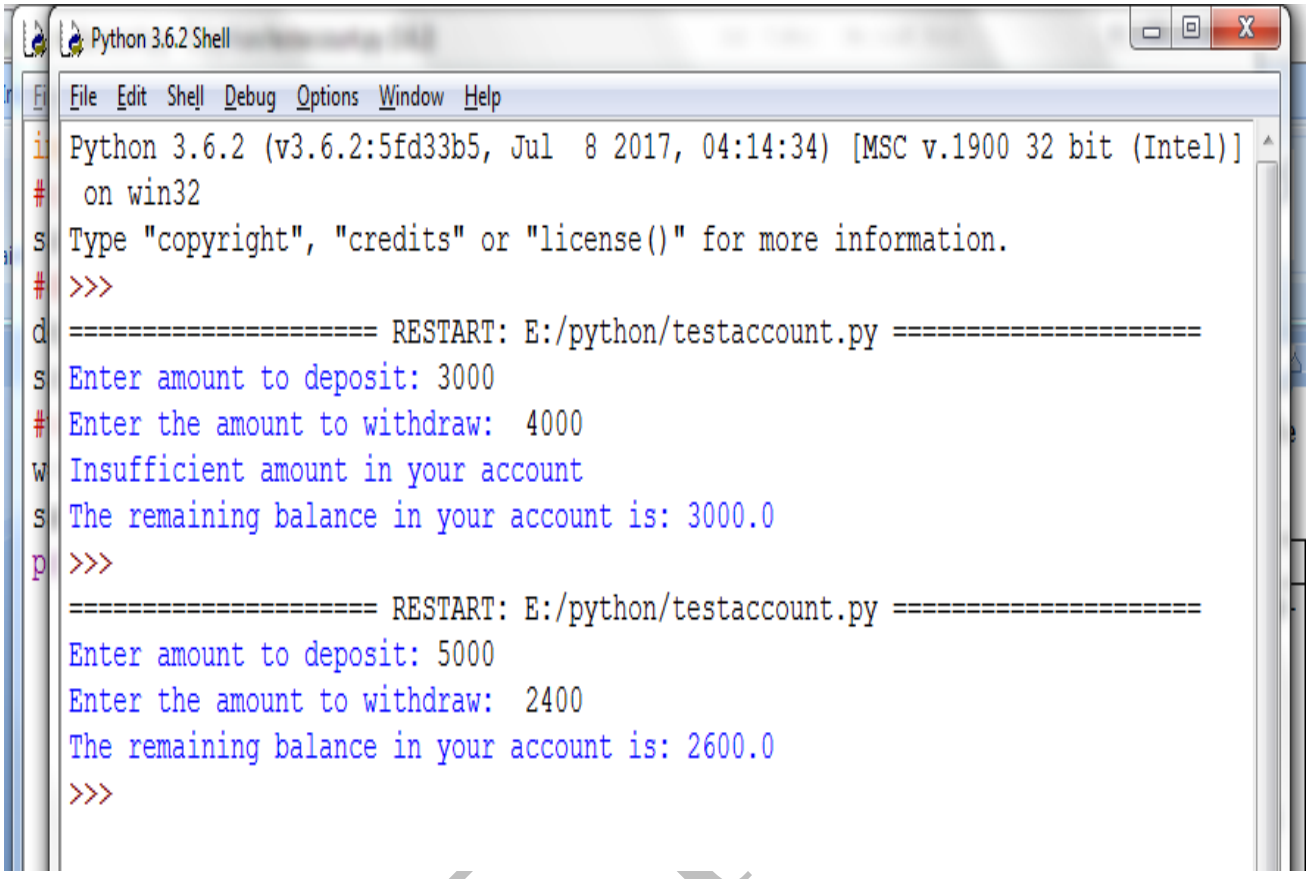
A change made to the instance variable will not reflect in other objects.

Create a class name "BankAccount" and perform operations like deposit and withdraw from the same account. Save this class in a module named Account.py and import it in the saving.py file. (Use Instance variable)

account.py (module)	testaccount.py

class BankAccount:
  def __init__(self,bal):
  self.balance=bal
  def deposit(self,bal):
  self.balance+=bal
  def withdraw(self,amount):
  if(self.balance>=amount):
  self.balance-=amount
  else:
  print("Insufficient amount in your account")
	import account #here account is the module - contains bankaccount class
	#create object from bankaccount class
	savings=account.BankAccount(0)
	#depositing amount
	dep=float(input("Enter amount to deposit: "))
	savings.deposit(dep)
	#withdraw amount
	w=float(input("Enter the amount to withdraw: "))
	savings.withdraw(w)
	print("The remaining balance in your account is:",savings.balance)

Output:



Write a Python program that defines a class named "BankAccount" in account.py module. Create three ATM Objects from this class and perform deposit and withdraw transactions at every atm. Display the balance after every deposit and withdrawal of amount. (use class class variable for balance)

account.py (module)

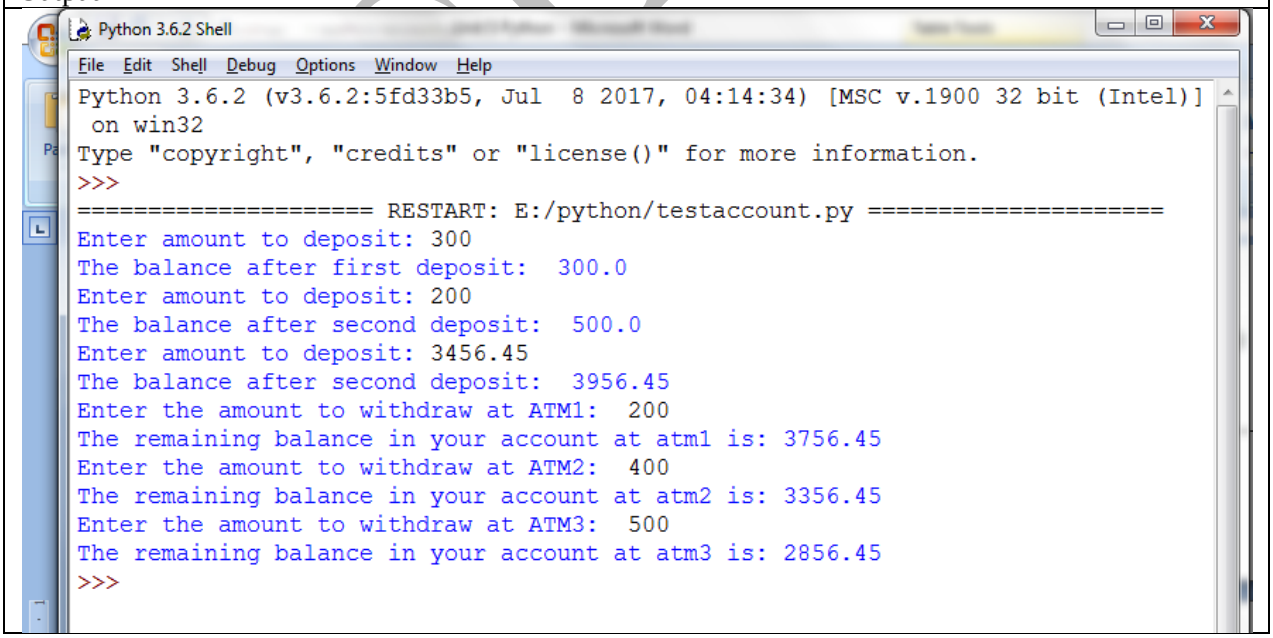
```
class BankAccount:
#class variable
   balance=0
   def __init__(self,bal):
      BankAccount.balance=bal
   def deposit(self,bal):
      BankAccount.balance+=bal
   def withdraw(self,amount):
      if(BankAccount.balance>=amount):
         BankAccount.balance-=amount
      else:
         print("Insufficient amount in your account")
   def balenquiry(self):
      return(BankAccount.balance)
```

testaccount.py

```
import account  #here account is the module -contains bankaccount class
#create object from bankaccount class
atm1=account. BankAccount (0)
```

```
atm2=account. BankAccount (0)
atm3=account. BankAccount (0)
#depositing amount at atm1and balance enquiry
dep=float(input("Enter amount to deposit: "))
atm1.deposit(dep)
print("The balance after first deposit: ",atm1.balenquiry())
#depositing amount at atm2and balance enquiry
dep=float(input("Enter amount to deposit: "))
atm2.deposit(dep)
print("The balance after second deposit: ",atm1.balenquiry())
#depositing amount at atm3and balance enquiry
dep=float(input("Enter amount to deposit: "))
atm2.deposit(dep)
print("The balance after second deposit: ",atm1.balenquiry())
#withdraw amount at atm1
w=float(input("Enter the amount to withdraw at ATM1:  "))
atm1.withdraw(w)
print("The remaining balance in your account at atm1 is:",atm1.balenquiry())
#withdraw amount at atm2
w=float(input("Enter the amount to withdraw at ATM2:  "))
atm2.withdraw(w)
print("The remaining balance in your account at atm2 is:",atm2.balenquiry())
#withdraw amount at atm3
w=float(input("Enter the amount to withdraw at ATM3:  "))
atm3.withdraw(w)
print("The remaining balance in your account at atm3 is:",atm3.balenquiry())
```

Output

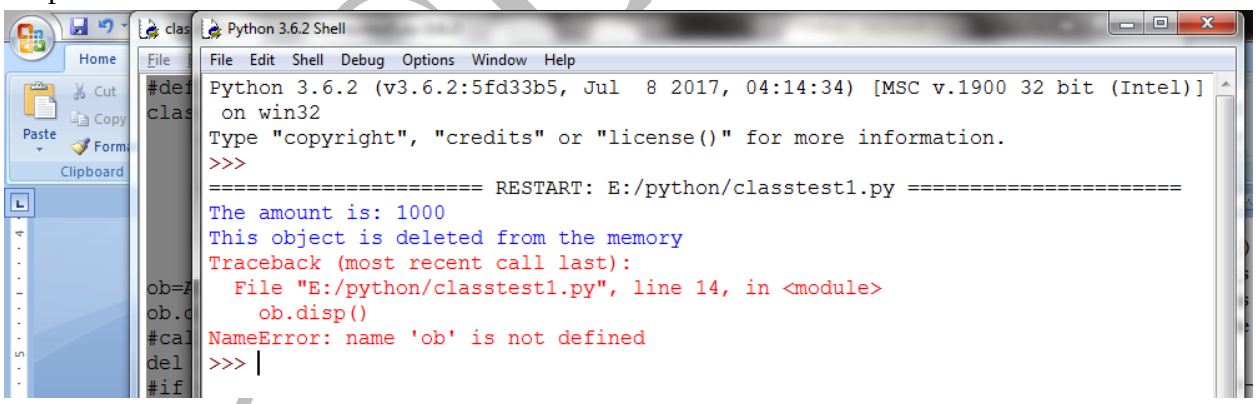


The _del_() method (Garbage Collection)

The __init__() method initializes an object instance variables. Similarly we have, __del__() method which will do the opposite work. This method frees the memory of the object when ti is no longer needed and the freed memory is returned back to the system. This process is known as "***Garbage Collection***". This method is called automatically when an object is going out of the scope. We can use "del" statement for deleting the object as shown in the program.

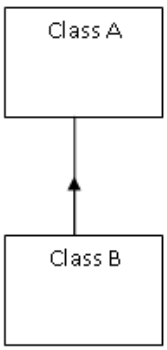
```
#defining class and creating the object
class ABC:
    def __init__(self,b):
        self.balance=b
    def disp(self):  # emthod with self argument
        print("The amount is:",self.balance)
    def __del__(self):
        print("This object is deleted from the memory")
ob=ABC(1000);
ob.disp()
#callinf the __del__() method
del ob
#if you try to call the method disp it raises error
ob.disp()
```

Output:



Inheritance

Inheritance is the one of the most and essential concept of the Object Oriented Programming. It is the process by which one class acquires the properties from one or more classes. Here the properties are the Data members and member functions. The new classes are created from the existing classes. The newly created class is called "Derived" class. The



existing class is called "Base" class. The Derived class also called with other names such as sub class, child class and descendent. The existing class is also called with other names such as super class, parent class and ancestor. The concept of inheritance therefore, frequently used to implement is-a relationship. The relationship between base and derived class is called "Kind of Relationship".

REUSABILITY

The main reason to go to the concept of the Inheritance is reusing the existing properties, which is called **reusability**. The reusability permits us to get the properties from the previous classes. We can also add the extra features to the existing class. This is possible by creating new class from existing class. The new class will have its own features and features acquired from the base class.

> ***Note: The new class will have its own properties and properties acquired from the base class.***

Inheriting classes in Python

The Syntax to inherit the properties from one class to another will be as follow:

class Derived_Class (Base_Class):

#body of the Derived_class

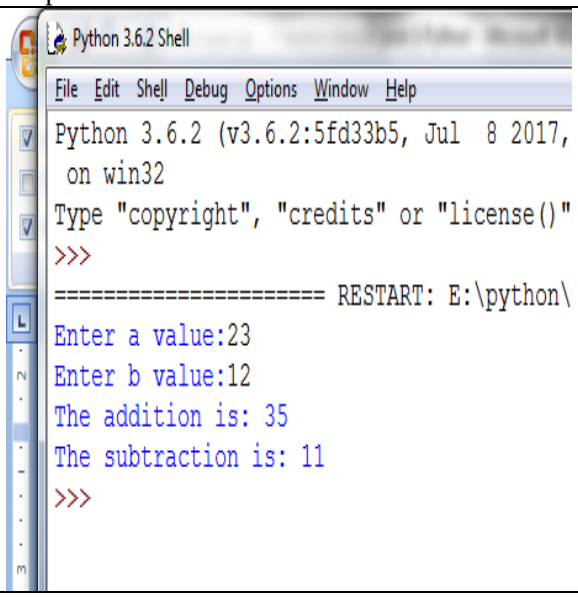
We can even write the base class name along with name of the module instead of writing it again.

Example Program to Implement the Inheritance (si.py)

#Single Inheritances

```
class A:
    def add(self,x,y):
        self.x=x
        self.y=y
        print("The addition is:",self.x+self.y)
class B(A):   #Single Inheritance
    def sub(self,x,y):
        self.x=x
        self.y=y
        print("The subtraction is:",self.x-self.y)
#read data into a and b
a=int(input("Enter a value:"))
b=int(input("Enter b value:"))
#create object from derived object
ob=B()
ob.add(a,b)
ob.sub(a,b)
```

Output



Polymorphism and Method Overriding

Polymorphism in its simple terms refers to have different forms. It is the key feature of OOP. It enables program to assign different version of the function based on the context. In Python, method overriding is a way to implement Polymorphism.

If the base class and derived classes are having the same method, when we create object from the derived class, the derived class version of the method is executed always. This is known as method overriding.

Example program for method overriding

Method overriding	Output

#method overriding -super() method
class BC:
  def disp(self):
  print("Base class Method")
class DC(BC):
  def disp(self):
  print("Derived class method")
ob=DC()
ob.disp()
	Derived class method

TYPES OF INHERITANCES

Python has various types of Inheritances. The Process of Inheritance can be either Simple or complex. This depends on the following points:

The Number of base classes used in the inheritance.

Nested derivation

Based on the above points the inheritances are classified in to the six different types.

Single Inheritance

Multilevel Inheritance

Multiple Inheritance

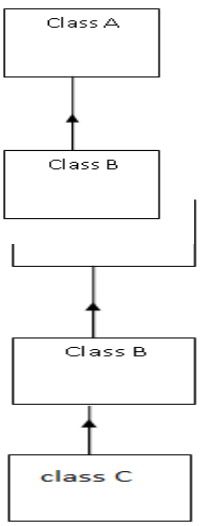
Hierarchical Inheritance

Hybrid Inheritance

Multi-path Inheritance

Single Inheritance

When only one class is derived from a single base class, such derivation is called single inheritance. It is the simplest form of Inheritance. The New class is termed as ***"Derived"*** class and the existing class is called ***"Base"*** class. The newly created class contains the entire characteristics from its base class. ***The Example program is already is given above.***



Multilevel Inheritance

The process of deriving a new class from a derived class is known as "Multilevel Inheritance". The intermediate derived class is also known as middle base class. C is derived from B. The class B is derived from A. Here B is called ***Intermediate base class***. The series of classes A, B and C is called "**Inheritance Pathway**".

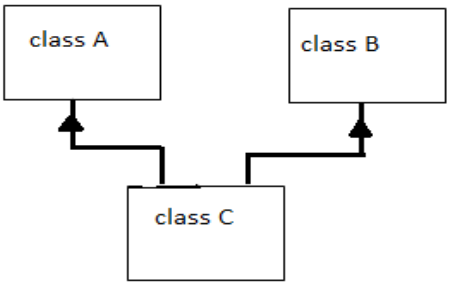
Example Program on Multi-Level Inheritance

Multi-Level Inheritance	Output

class A:
  def add(self,x,y):
  self.x=x
  self.y=y
  print("The addition is:",self.x+self.y)
class B(A): #Single Inheritance
  def sub(self,x,y):
  self.x=x
  self.y=y
  print("The subtraction is:",self.x-self.y)
class C(B):
  def mul(self,x,y):
  self.x=x
  self.y=y
  print("The product is:",self.x*self.y)
#read data into a and b
a=int(input("Enter a value:"))
b=int(input("Enter b value:"))
#create object from derived object
ob=C()
ob.add(a,b)
ob.sub(a,b)
ob.mul(a,b)
	Enter a value:23
	Enter b value:12
	The addition is: 35
	The subtraction is: 11
	The product is: 276

Multiple Inheritances

When two or more base classes are used in the derivation of new class, it is called "***Multiple Inheritance***". The derived class C has all the properties of both class A and class B.



Example program on Multiple –Inheritance:

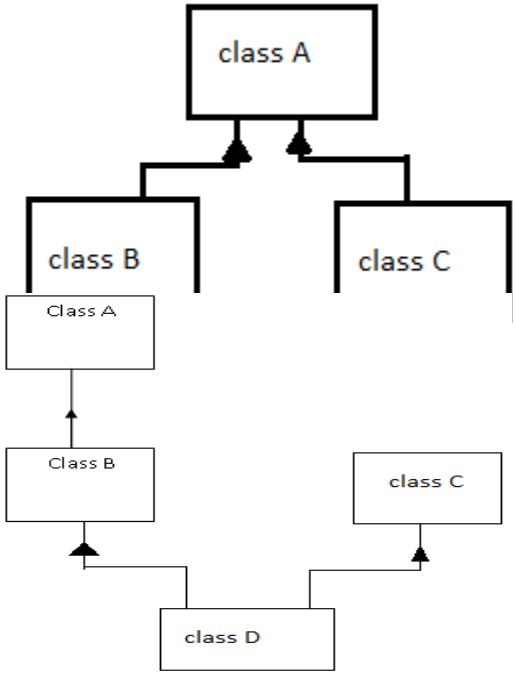
Multiple Inheritance	Output

class A:
  def add(self,x,y):
  self.x=x
  self.y=y
  print("The addition is:",self.x+self.y)
class B: #Single Inheritance
  def sub(self,x,y):
  self.x=x
  self.y=y
	Enter a value:23
	Enter b value:12
	The addition is: 35
	The subtraction is: 11
	The product is: 276

```
        print("The subtraction is:",self.x-self.y)
#Multiple Inheritance
class C(A,B):
    def mul(self,x,y):
        self.x=x
        self.y=y
        print("The product is:",self.x*self.y)
#read data into a and b
a=int(input("Enter a value:"))
b=int(input("Enter b value:"))
#create object from derived object
ob=C()
ob.add(a,b)
ob.sub(a,b)
ob.mul(a,b)
```

Hierarchical Inheritance

The Process of splitting the base class into several sub classes is called, "**Hierarchical Inheritance**". All the sub classes have the same properties of those in base class. Here The classes B and C acquire properties from the class A.



Hybrid Inheritance

A Combination of two or more types of inheritances is called "***Hybrid Inheritance***". Sometimes it is essential to derive a class using more types of inheritance. Here There are 4 classes A, B, C and D. The class B acquires the properties from A, hence there exist a single inheritance. Class D acquires properties from B (which is a derived class) and C, hence uses multiple inheritance. Here two different types of Inheritances used are: Single and Multiple.

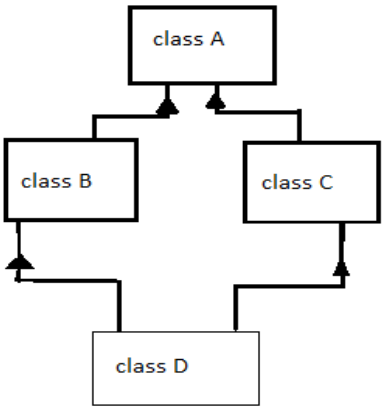
Example Program on Hybrid-Inheritance

Hybrid-Inheritance	Output

#types of Inheritances
class A:
  def add(self,x,y):
  self.x=x
  self.y=y
  print("The addition is:",self.x+self.y)
class B(A): #Single Inheritance
  def sub(self,x,y):
  self.x=x
	Enter a value:12
	Enter b value:3
	The addition is: 15
	The subtraction is: 9
	The product is: 36
	The division is: 4.0

```
        self.y=y
        print("The subtraction is:",self.x-self.y)
class C:
    def mul(self,x,y):
        self.x=x
        self.y=y
        print("The product is:",self.x*self.y)
#Hybrid Inheritance
class D(B,C):
    def div(self,x,y):
        self.x=x
        self.y=y
        print("The division is:",self.x/self.y)
#read data into a and b
a=int(input("Enter a value:"))
b=int(input("Enter b value:"))
#create object from derived object
ob=D()
ob.add(a,b)
ob.sub(a,b)
ob.mul(a,b)
ob.div(a,b)
```

Multi-Path Inheritance



Deriving a class from two derived classes that are in turn derived from the same base class is called "***Multi-Path Inheritance*".** In this context the derived class has two immediate base classes, which are derived from one base class, there by forming the ***grandparent, parent and child relationship***. The derived class inherits the features from base class (grandparent) via two separate paths. Therefore, the base class is also known as the indirect base class.

Problem in Multi-Path Inheritance (Diamond Problem)

The derived class has the members of the base class twice, via parent1 and parent 2. This results in ambiguity because a duplicate set of members is created. This is avoided in python using the dynamic algorithm (C3 and MRO) linearizes the search order in such as way that left-to-right ordering is specified to avoid duplication.

Composition or Container or Complex objects

The complex objects are objects that are created from smaller or simple objects. For example, a car is built from metal frame, an engine, some tyres, a steering and several other parts. The process of building complex objects from simpler objects is known as ***Composition*** or ***Containership***. The relationship is also called "***has-a***" or "***part-of***" relationship.

Example Program on Composition:

Composition	Output

class A1:
  def add(self,x,y):
  self.x=x
  self.y=y
  print("The addition is:",self.x+self.y)
#composition
class B1:
  def sub(self,x,y):
  self.a=A1() #object of class A1()
  self.x=x
  self.y=y
  print("The subtraction is:",self.x-self.y)
  self.a.add(x,y) #calling the method of another class
ob=B1()
ob.sub(12,3)
	The subtraction is: 9
	The addition is: 15

The Difference between Inheritance and composition

Inheritance	Composition
A class Inherits properties from another class	A class contains objects of different classes as data members
The derived class may override base class functions	The container class cannot override the base class functions
The derived class may add data or functionality to the base class	The container class cannot add anything to the contained class
This represents a "is-a" relationship	This represents "has-a" relationship

Abstract classes and Inheritances

In python, it is possible to create a class which cannot be instantiated. We can create a class from which objects are not created. This class is used as interface or template only. The derived can override the features of the base class. In python, we use "**NotImplementedError**" to restrict the instantiation. Any class that has the **NotImplementedError** inside the method definitions cannot be instantiated.

Example Program on Abstract class:

#abstract class	Output

class A:
  def disp(self):
  #abstract class method
  raise NotImplementedError()
class B(A):
  def disp(self):
	The method of class B
	The method of class C

```
        print("The method of class B")
class C(A):
    def disp(self):
        print("The method of class C")
#create object
#ob=A()     # we cannot create object
#ob.disp()   #we cannot call the method
ob1=B()
ob1.disp()
ob2=C()
ob2.disp()
```

Error and Exceptions: Difference between an error and Exception

There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

Syntax Error: These errors are occur when we violate the rules of Python and these are most common kind of errors that we get while we learn any new programming language.

Examples: (1) if x>=10

SyntaxError: invalid syntax

Here at the end of, if statement (:) colon should be written, error will be generated otherwise.

Violating Indentation, missing (:) colon at the end of loop statements etc.

Exception Error: Even if a statement is syntactically correct, it may still cause an error when executed. Such errors that occur at run-time are known as "Exceptions". The logical error may occur due to wrong algorithm or logic while solving a particular problem. These logical errors may lead to Exceptions. Examples of Exceptions are as follow:

```
>>>y+2
```

NameError: name 'y' is not defined

```
>>>2+'x'
```

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

```
>>>10*(1/0)
```

ZeroDivisionError: division by zero

```
>>>l=[1,2,3]
>>> l[3]
```

IndexError: list index out of range

```
>>>d={1:'x',2:'y',3:'z'}
>>> d[4]
```

KeyError: 4

Problem with Exception:

The normal execution of the program will be abruptly terminated because of the exception. In general, just because of a single run-time error, it is not reasonable to terminate the program abruptly, even if program contains some legal lines. To solve this problem the exceptions must be handled. In python these exceptions are handled using the ***try*** and ***except*** blocks.

Handling Exceptions

The statements that can raise the exception are placed inside the ***try*** block, and the code that handles is placed inside ***except*** block. Here **try** and **except** are keywords. The syntax for try-except can be as given bellow:

```
try:
        Statements
except ExceptionName:
        Statements
```

The try statement works as follows.

First, the *try block* (the statement(s) between the try and except keywords) is executed.

If no exception occurs, the *except block* is skipped and execution of the try statement is finished.

If an exception occurs during execution of the try block, the rest of the block is skipped. Then if its type matches the exception named after the except keyword, the except block is executed, and then execution continues after the try statement.

If an exception occurs which does not match the exception named in the except block, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message.

Example Program

Exception1.py	Exception1.py

#program without exception handler
x=int(input("Enter x value"))
y=int(input("Enter y value"))
print("The sum is:",x+y)
print("The suntraction is:",x-y)
print("The quotient is:",x/y)
print("The product is:",x*y)
print("The remainder is:",x%y)
print("The power of x^y si:",x**y)
	x=int(input("Enter x value"))
	y=int(input("Enter y value"))
	print("The sum is:",x+y)
	print("The suntraction is:",x-y)
	try:
	  print("The quotient is:",x/y)
	  print("The remainder is:",x%y)
	except ZeroDivisionError:
	  print("You should not divide number with zero")
	#legal code
	print("The product is:",x*y)
	print("The power of x^y si:",x**y)

Output	Output

Enter x value4
Enter y value0
The sum is: 4
The suntraction is: 4
Traceback (most recent call last):
  File "E:/python/exception1.py", line 5, in <module>
  print("The quotient is:",x/y)
ZeroDivisionError: division by zero
	Enter x value4
	Enter y value0
	The sum is: 4
	The suntraction is: 4
	You should not divide number with zero
	The product is: 0
	The power of x^y si: 1

Multiple except blocks

Python allows you to have multiple except blocks for single try block. The block that matches with exception will be executed. A try block can be associated with one or more except block. However, only one block will be executed at a time.

The Syntax for multiple except blocks for single try will be as follow:

try:
    Operations are done in this block
except Exception1:
    If exception is matched, this block will be executed.
except Exception2:
    If exception is matched, this block will be executed.
else:
    If there is no exception matched, this block will be executed.

Example program to Handle multiple exceptions

Mulexcept.py	Output

#read the data
try:
  x=int(input("Enter value of x:"))
  y=int(input("Enter value of y:"))
  print(x**2/y)
except (ValueError):
  print("Check before you enter value:")
except ZeroDivisionError:
  print("The value of y should not be zero")
except (KeyboardInterrupt):
  print("Please eneter number:")
print("End of the program")
	Enter value of x:2
	Enter value of y:0
	The value of y should not be zero
	End of the program
	>>>
	== RESTART: E:/python/multiexcep.py =======
	Enter value of x:
	Check before you enter value:
	End of the program
	>>>

Multiple Exceptions in a single except block

It is possible to list number of exceptions in the except block. When any one of the exception is raised, then the except block is executed as shown in the program.

Multexcept1.py	output

#read the data
try:
  x=int(input("Enter value of x:"))
  y=int(input("Enter value of y:"))
  print(x**2/y)
except (ValueError,ZeroDivisionError,KeyboardInterrupt):
  print("Check before you enter values foer x and y. Y should not be zero")
print("End of the program")
	Enter value of x:3
	Enter value of y:2
	4.5
	End of the program
	>>>
	======= RESTART: E:/python/multiexcep1.py
	=======
	Enter value of x:4
	Enter value of y:
	Check before you enter values for x and y. Y should not be zero

	End of the program

Except block without exception

- We can even specify except block without mentioning any exception.
- In large software programs, many a times, it is difficult to anticipate (guessing) all types of possible exceptional conditions.
- Therefore a programmer may not be able to write a different handler for every exception.
- In such a situation, a better idea is to write a handler that would catch all types of exceptions.
- This must me the last one that can serve as ***wildcard***.
- Syntax will be as follow:

```
try:
   statements
except Exception1:
        statements
except Exception2:
        statement
except:
   execute this block, if an exception match is found
```

Example Program

Testexcept.py	Output
#read the data **try:** x=int(input("Enter value of x:")) y=int(input("Enter value of y:")) print(x**2/y) **except (TypeError):** print("Choose the correct type of value:") **except (ZeroDivisionError):** print("The value of y should not be zero") **except:** print("Unexpected error ....Terminating the program:") print("End of the program")	Enter value of x:4 Enter value of y: Unexpected error ....Terminating the program: End of the program

The else clause

The **try- except** block can also have an else clause, which, when present must follow all except blocks. The statements in the else block only executed, if the try clause does not raise an exception.

Testexcept.py	Output
try: x=int(input("Enter value of x:")) y=int(input("Enter value of y:")) print(x**2/y) **except (TypeError):** print("Choose the correct type of value:") except (ZeroDivisionError): print("The value of y should not be zero") **except (ValueError):** print("Unexpected error ....Terminating the program:") **else:** print("Program execution is successful....") print("End of the program")	Enter value of x:4 Enter value of y:2 8.0 Program execution is successful.... End of the program

Raising the exception

We can also deliberately raise the exception using the raise keyword. The syntax is as follow:

```
raise [exception-name]
```

Example Program:

Testexcept.py	Output
try: x=int(input("Enter value of x:")) y=int(input("Enter value of y:")) print(x**2/y) **raise ZeroDivisionError** **except:** print("The value of y should not be zero")	Enter value of x:4 Enter value of y:0 The value of y should not be zero End of the program

Handling exceptions in Invoked Functions

We can also handle exceptions inside the functions using the try-except blocks.

Funexcep.py	Output

def division(num,deno):
try:
r=num/deno
print("The quotient is:",r)
except ZeroDivisionError:
print("You cannot divide number by zero")
#function call
x=int(input("Enter x value:"))
y=int(input("Enter y value:"))
division(x,y)
	Enter x value:4
	Enter y value:0
	You cannot divide number by zero
	>>>
	======= RESTART: E:/python/funexcep.py ========
	Enter x value:4
	Enter y value:2
	The quotient is: 2.0
	>>>

Built-in Exceptions:

Exception	Description
Exception	Base class for all exceptions
StopIteration	Generated when next() method does not point to any object
SystemExit	Raised by sys.exit() function
StandardError	Base class for all built-in exceptions(except StopIteration and SystemExit)
ArithmeticError	Base class for mathematical errors
OverflowError	Raised when maximum limit of the number is exceeded
FloatingPointError	Raised when floating point operations could not performed
ZeroDivisionError	Raised when number is divided by zero
AssertionError	Raised when assert condition fails
AttributeError	Raised when attribute reference is failed
EOFError	Raised when end of the file is reached
ImportError	Raised when import statement is failed
KeyboardInterrupt	Raised when user interrupts the keyboard (pressing ctrl+c)
IndexError	Raised when index is not found
KeyError	Raised when key is not found
NameError	Raised when an identifier is not defined
SyntaxError	Raised when rules of language are violated
ValueError	Raised when arguments are invalid type
TypeError	Raised when two or more data types are mixed

User Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *Exception*. Here, a class is created that is subclassed from *Exception*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable 'e' is used to create an instance of the class *Networkerror*.

```
class NetworkError(Exception):
  def __init__(self, arg):
    self.name = arg
```

```
try:
  raise NetworkError('Network Connection Failed')
except NetworkError as e:
  print("The network error is:", e.name)
```

Output:

The network error is: Network Connection Failed