DEPARTMENT OF ARTIFICIAL

INTELLIGENCE & MATHEMATICAL SCIENCES

# PROGRAMMING LANGUAGE-II

# Python Laboratory Manual

# LABORATORY MANUAL CONTENTS

**Example:-1 Create a class with name Parrot and some object**

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class (Objects of the class)
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

---

```
class Parrot:

    # class attribute
    species = "bird"
    Age=10;
    Name="Blu";

# instantiate the Parrot class
blu = Parrot()
woo = Parrot()

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

print("Blu age is  {}".format(blu.__class__.Age))
print("Its name is {}".format(woo.__class__.Name))
```

Example:-2 Create a class with name Human and their objects Amir and Aisha

```
class Human:

    # class attribute
    species = "Animal"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
Amir = Human("Amir", 25)
Aisha = Human("Aisha", 20)

# access the class attributes
print("Amir is a {}".format(Amir.__class__.species))
print("Aisha is also a {}".format(Aisha.__class__.species))

# access the instance attributes
print("{} is {} years old".format( Amir.name, Amir.age))
print("{} is {} years old".format( Aisha.name, Aisha.age))
```

Exercise-3: Create a class with name Human and with some characteristics

```
class Human:

    # class attribute
    species = "Animal"

    # instance attribute
    def __init__(self, name, age, profession):
        self.name = name
        self.age = age
        self.profession= profession
```

```
# instantiate the Parrot class
Amir = Human("Amir", 25, "Doctor")
Aisha = Human("Aisha", 20, "Teacher")



# access the class attributes print("Amir is a
{}".format(Amir.__class__.species)) print("Aisha is also a
{}".format(Aisha.__class__.species))

# access the instance attributes print("{} is {} years old and {}".format(
Amir.name, Amir.age, Amir.profession))
print("{} is {} years old and {}".format( Aisha.name, Aisha.age, Aisha.profession))
```

==================================================================
===============================================Lec # (Method in python+++++++++++++++++++++++++++++++++

**Lab#3**
**Methods in Object Oriented Programming Using Python.**

**Method/Function:It is a line/group of code which is used to achieve the specific task as per programmer desire and it returns a value.**

**This group of code is called "Function" in procedural languages, while in OOP it is known as behavior or method.**

## Method Types in Python

There are three types of methods:

1. Instance Methods.
2. Class Methods
3. Static Methods

## (1) Instance Method

```
class Student:

    def __init__(self, a, b):
self.a = a
```

```
        self.b = b

    def avg(self):
        return (self.a + self.b) / 2

s1 = Student(10, 20)
print( s1.avg() )
```

---

```
class volume:
  def calculate(self,a,b):
    self.a = a
    self.b = b
    c = a*b
    print(c)

z = volume()
z.calculate(10,20)
```

---

```
class volume:
  def calculate(self):
    a = 9
    b = 7
    c = a*b
    print(c)

z = volume()
z.calculate()
```

**Example#2**

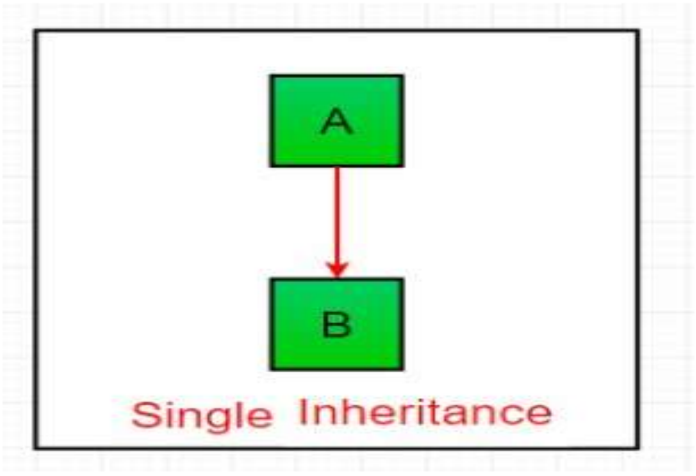
```
class Parrot:

    # instance attributes     def
__init__(self, name, age):         self.name
= name
        self.age = age

    # instance method     def sing(self, song):         return
"{} sings {}".format(self.name, song)

#    def dance(self):
 #       return "{} is now dancing".format(self.name)

# instantiate the object blu =
Parrot("Blu", 10)
print("{} is {} years old".format( blu.name, blu.age))

# call our instance methods print(blu.sing("'Happy'"))
#print(blu.dance())

==============================Method2==================================
```

Example#3: Create a class with Name Teacher which have two behaviors “Eat() and Teach()”.

=

===============================Inheritance===================================

**Single Inheritance:** Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

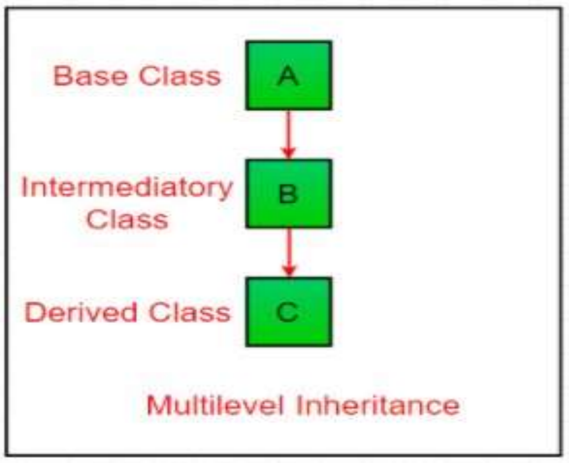


Exercise # 1
Write a program which has a parent class named "Bird" with at-least one method and then a child class named Penguin. This class extends the parent lass "Bird".

Exercise # 2
Write a program with a parent class named "Car" having two different behaviors and then its child class named "Suzuki" which inherits parent's class methods and it also has its own methods.

## Multilevel Inheritance

When a child class becomes a parent class for another child class.
In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



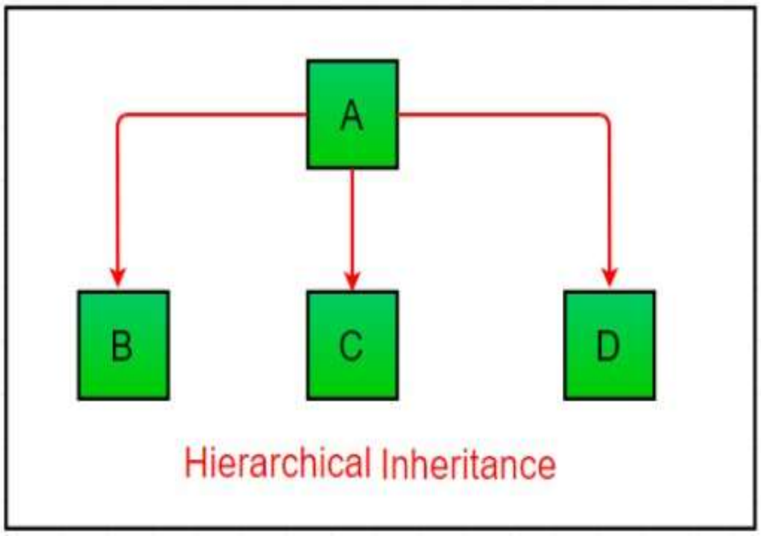
Exercise # 1
Program with three classes one class with "Parent" class 2nd class which is child of parent class (Inherits to parent class properties / behaviors)and 3rd class which is the child of 2nd class (2nd class is the parent of 3rd class) and this class may inherit the properties and behaviors of 2nd class.

Exercise # 2
Write a program which represents your personal family information using Multilevel Inheritance

## Hierarchical Inheritance

Hierarchical inheritance involves multiple inheritance from the same base or parent class.

When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



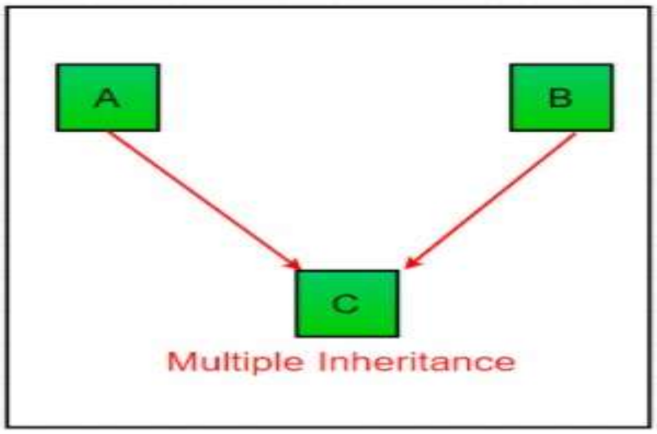
Exercise # 1

Write a program in which one PARENT CLASS may have more than one derived classes (Child Classes).

## Multiple Inheritance

When a child class inherits from more than one parent class.

When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.



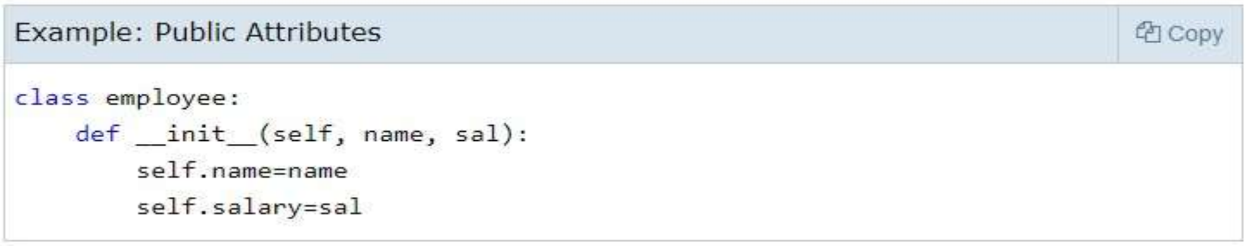
Exercises # 1

Write a program in which one child / derived class may extends two parent classes at a time?

## Access Modifier in Python
## (public, private and protected)

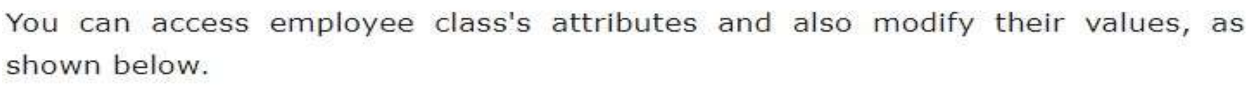
Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behaviour of protected and private access specifiers.

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

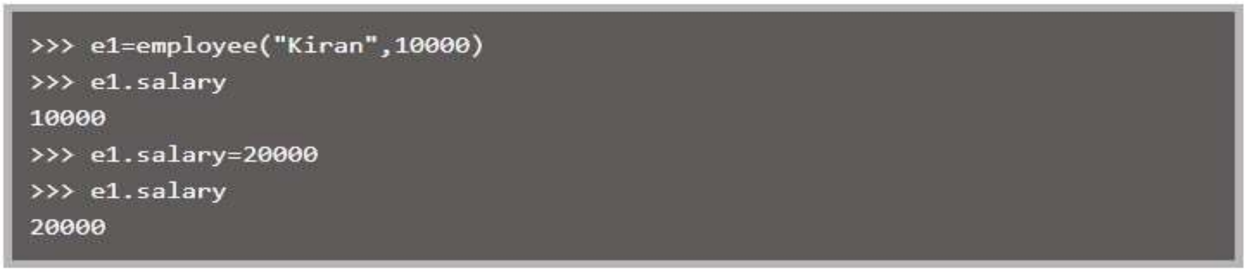


Example: Public Attributes

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self.salary=sal
```

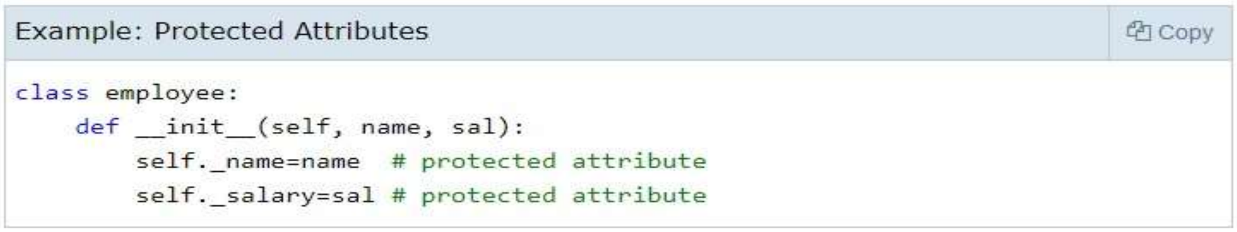


You can access employee class's attributes and also modify their values, as shown below.



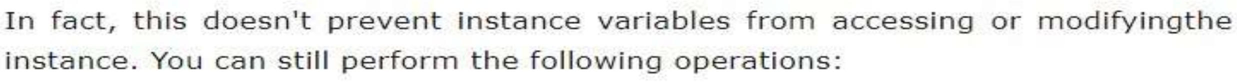
```
>>> e1=employee("Kiran",10000)
>>> e1.salary
10000
>>> e1.salary=20000
>>> e1.salary
20000
```

Python's convention to make an instance variable **protected** is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.

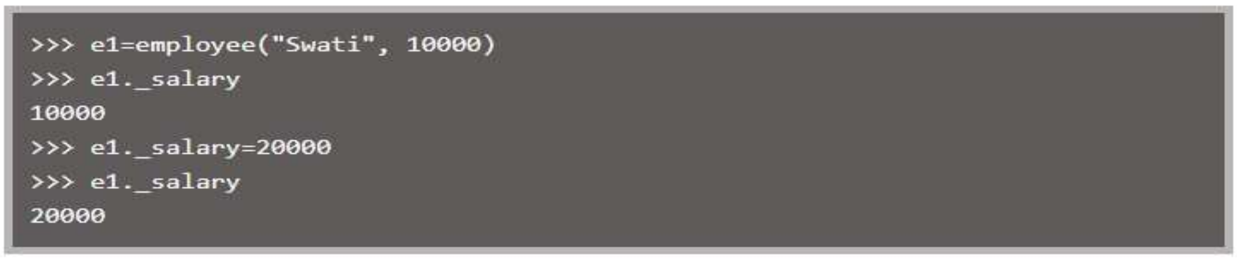


Example: Protected Attributes

```
class employee:
    def __init__(self, name, sal):
        self._name=name  # protected attribute
        self._salary=sal # protected attribute
```



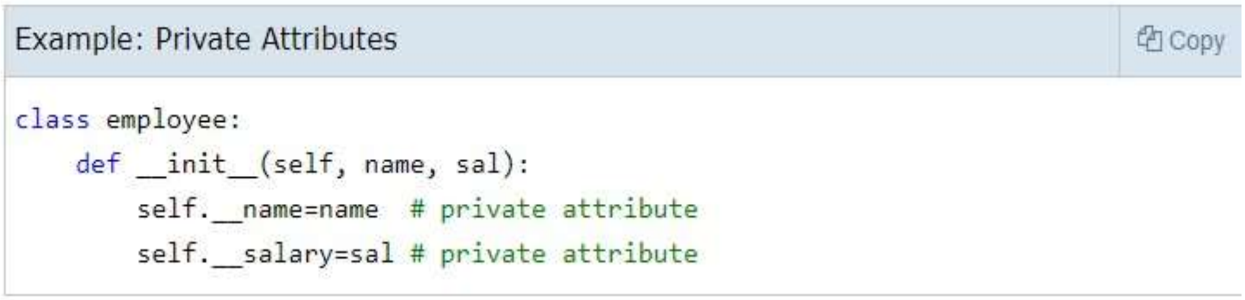
In fact, this doesn't prevent instance variables from accessing or modifyingthe instance. You can still perform the following operations:



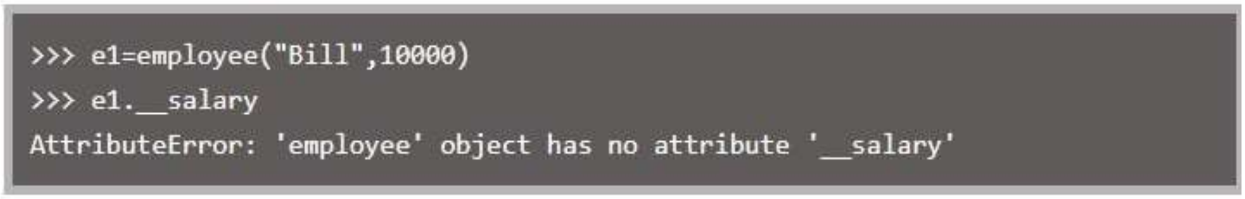
```
>>> e1=employee("Swati", 10000)
>>> e1._salary
10000
>>> e1._salary=20000
>>> e1._salary
20000
```

Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class.

Similarly, a double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:



```
class employee:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        self.__salary=sal # private attribute
```



```
>>> e1=employee("Bill",10000)
>>> e1.__salary
AttributeError: 'employee' object has no attribute '__salary'
```

## Public Access Modifier:

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

```
# program to illustrate public access modifier in a class

class Geek:

    # constructor       def
__init__(self, name, age):

        # public data mambers
self.geekName = name
        self.geekAge = age

    # public memeber function
def displayAge(self):

        # accessing public data member
        print("Age: ", self.geekAge)

# creating object of the class
obj = Geek("Aamir", 20)

# accessing public data member
print("Name: ", obj.geekName)

# calling public member function of the class  obj.displayAge()
```

Example # 2

```
class Parent:   # Public Access Modifier
def func1(self):
      print("this is function 1") class
Parent2:
  def func2(self):
      print("this is function 2")
class Child(Parent , Parent2):
def func3(self):
print("this is function 3")

ob = Child()   # object of child class ob.func1()   #
Accessing the method of Parent1 class ob.func2()      #
Accessing the method of Parent2 class ob.func3()  #
Chhild class method
```

## Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

```
# program to illustrate protected access modifier in a class

# super class
class Student:

    # protected data members
    _name = None
    _roll = None
    _branch = None

    # constructor
    def __init__(self, name, roll, branch):
        self._name = name
        self._roll = roll
        self._branch = branch

    # protected member function
    def _displayRollAndBranch(self):

        # accessing protected data members
        print("Roll: ", self._roll)
        print("Branch: ", self._branch)


# derived class
class Geek(Student):

    # constructor
    def __init__(self, name, roll, branch):
        Student.__init__(self, name, roll, branch)

    # public member function
    def displayDetails(self):

        # accessing protected data members of super class
        print("Name: ", self._name)

        # accessing protected member functions of super class
        self._displayRollAndBranch()

# creating objects of the derived class
obj = Geek("Aamir", 1706256, "Information Technology")

# calling public member functions of the class
obj.displayDetails()
```

## Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

```
class volume:   # normal Program
  name='Saeed Ahmed'
def calculate(self):
    a = 9
b = 7     c
= a*b
    print(c)

#z = volume()
#z.calculate() class
volume2(volume):   def
vol(self):
    print("this is test message")


obj=volume2() obj.calculate()
obj.vol() obj.name
```

```
# Same above program with Private modifier
```

```
class volume:
 __name='Saeed Ahmed'
def calculate(self):
  a = 9
b = 7    c
= a*b
  print(c)

#z = volume() #z.calculate()
class volume2(volume):   def
vol(self):    print("this is test
message")
```

```
obj=volume2() obj.calculate()
obj.vol() obj._volume__name
```

```
# program to illustrate private access modifier in a class

class Geek:

    # private members
    __name = None
    __roll = None
    __branch = None

    # constructor       def __init__(self,
name, roll, branch):
        self.__name = name
self.__roll = roll          self.__branch
= branch

    # private member function
def __displayDetails(self):

        # accessing private data members
print("Name: ", self.__name)
print("Roll: ", self.__roll)          print("Branch:
", self.__branch)

    # public member function
def accessPrivateFunction(self):

        # accesing private member function
self.__displayDetails()

# creating object
obj = Geek("Aamir", 1706256, "Information Technology")

# calling public member function of the class  obj.accessPrivateFunction()
```

**Below is a program to illustrate the use of all the above three access modifiers (public, protected and private) of a class in Python:**

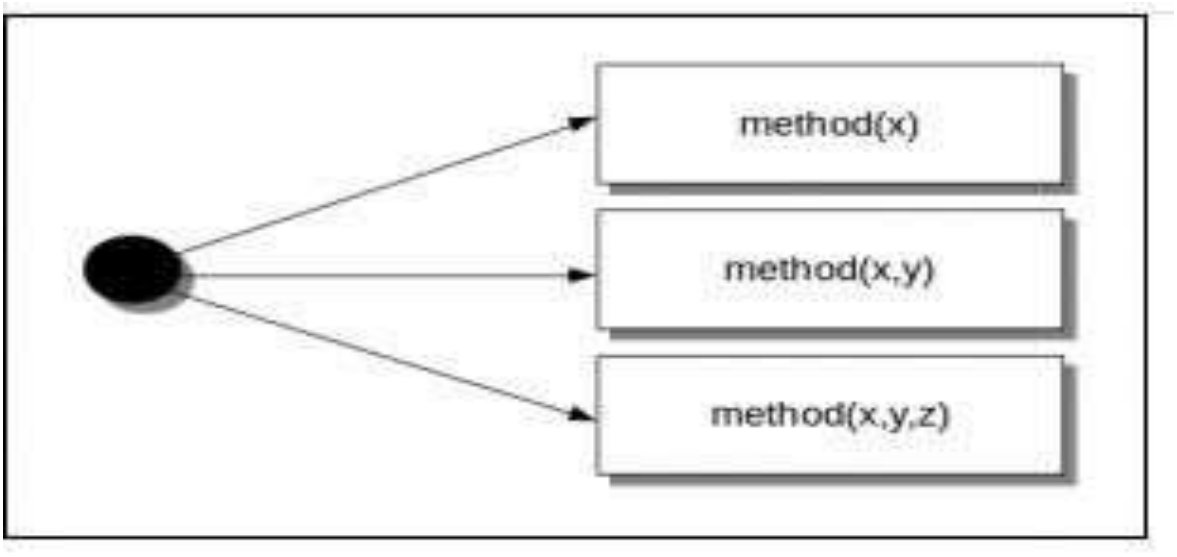
```
# program to illustrate access modifiers of a class

# super class  class
Super:

    # public data member
var1 = None

    # protected data member
    _var2 = None

    # private data member
    __var3 = None

    # constructor       def __init__(self,
var1, var2, var3):
        self.var1 = var1
self._var2 = var2
        self.__var3 = var3

   # public member function
def displayPublicMembers(self):

        # accessing public data members
        print("Public Data Member: ", self.var1)

    # protected member function
def _displayProtectedMembers(self):

        # accessing protected data members
        print("Protected Data Member: ", self._var2)

    # private member function
def __displayPrivateMembers(self):

        # accessing private data members
        print("Private Data Member: ", self.__var3)

    # public member function
def accessPrivateMembers(self):

        # accessing private memeber function
self.__displayPrivateMembers()
```

```
# derived class  class
Sub(Super):

   # constructor        def
__init__(self, var1, var2, var3):
        Super.__init__(self, var1, var2, var3)

   # public member function        def
accessProtectedMemebers(self):

        # accessing protected member functions of super class
self._displayProtectedMembers()

# creating objects of the derived class      obj
= Sub("Geeks", 4, "Geeks !")

# calling public member functions of the class
obj.displayPublicMembers()  obj.accessProtectedMemebers()
obj.accessPrivateMembers()

# Object can access protected member
print("Object is accessing protected member:", obj._var2)

# object can not access private member, so it will generate Attribute error  #print(obj.__var3)
```

Polymorphism

## Method Overloading:

- Method Overloading is the class having methods that are the same name with different arguments.
- Arguments different will be based on a number of arguments and types of arguments.
- It is used in a single class.
- It is also used to write the code clarity as well as reduce complexity.



## Method Overloading in Python

In Python, you can create a method that can be called in different ways. So, you can have a method that has zero, one, or more parameters. Depending on the method definition, we can call it with zero, one or more arguments.

Given a single method or function, the number of parameters can be specified by you. This process of calling the same method in different ways is called method overloading.

Program to demonstrate Method Overloading-1

```
#Method Overloading class Myclass:   def
sum(self, a=None, b=None, c=None):     if
a!=None and b!=None and c!=None:
      result=a+b+c     elif
a!=None and b!=None:
      result=a*b
   return result obj=Myclass()
```

```
print(obj.sum(10,10))
```

---

Program of method overloading with different 3 numbers of parameters

```
#Method Overloading class Myclass:   def sum(self, a=None,
b=None, c=None):     if a!=None and b!=None and c!=None:
# 1st action calling      result=a+b+c
    elif a!=None and b!=None:     # 2nd action calling
      result=a*b
    else:                        # 3rd action calling
      result=a
return result
obj=Myclass()
print(obj.sum(10)
)
```

---

```
#Method Overloading (with float number) class
Myclass:
  def  sum(self,  a=None,  b=None,  c=None):
if a!=None and b!=None and c!=None:
      result=a+b+c     elif
a!=None and b!=None:
      result=a*b
return result
obj=Myclass()
print(obj.sum(10,10.5))
```

---

Program to demonstrate Method Overloading-2

```
class Human:

    def sayHello(self, name=None):

        if name!=None:
            print('Hello ' + name)
else:
            print('Hello ')



# Create instance
obj = Human()

# Call the method
```

```
obj.sayHello()

# Call the method with a parameter obj.sayHello('Guido')
```

```
# Program to computer area with method overloading-3
class Compute: # area method   def area(self, x = None,
y = None):     if x != None and y != None:
      return x * y
elif x != None:
return x * x
else:       return 0
# object obj =
Compute() #
zero argument
print("Area Value:", obj.area())
# one argument
print("Area Value:", obj.area(4))
# two argument
print("Area Value:", obj.area(3, 5))
```

## Method Overriding in Python

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

```
#Program for Method Overriding-1
class Add:   def result(self,a,b):
    print('Addition:',a+b)

class Multi(Add):
  def result(self, a,b):
    print('Multiplication:',a*b)
obj=Multi() obj.result(10,10)
```

In above program only child class method is accessed. In that case how we can access the parent class method [which has same name like child class method] There are two ways:

1 We can access the parent class method making its (Parent class) object, see example below [But in this case there is no use of Inheritance ]

```
#Program for Method Overriding to call parent class method
       #Method Overriding
       class Add:   def
       result(self,a,b):
          print('Addition:',a+b)

       class Multi(Add):
       def result(self, a,b):
          print('Multiplication:',a*b)
       obj=Multi()
       obj.result(10,10)

       obj1=Add()
       obj1.result(10,5)
```

2 By using Super key word in child class

```
#Program of Method Overriding using super key word class
Add:
  def result(self,x,y):
    print('Addition:',x+y)
```

```
class Multi(Add):
def result(self, a,b):
   super().result(10,5) # Accessing Parent class method
print('Multiplication:',a*b)
obj=Multi() obj.result(10,10)
```

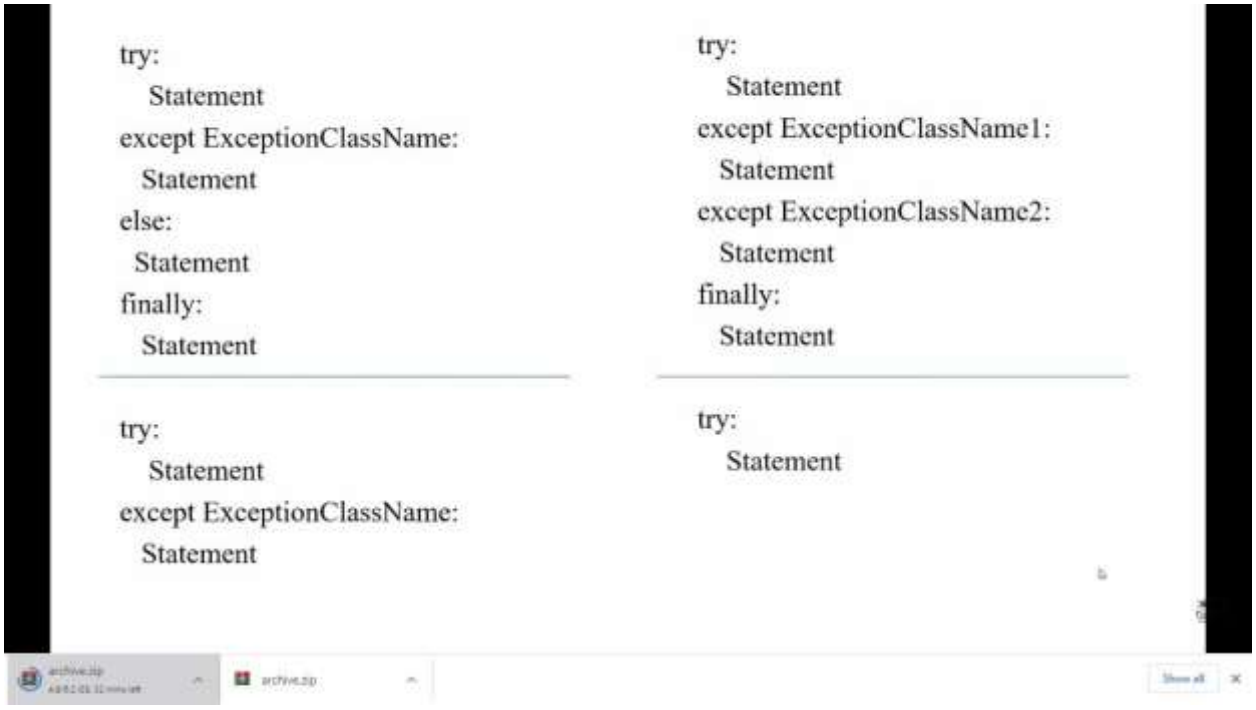
Method overriding with multiple inheritance

```
# Python program to demonstrate overriding in multiple inheritance


# Defining parent class 1  class
Parent1():
    # Parent's show method
  def show(self):
  print("Inside Parent1")

  # Defining Parent class 2  class
  Parent2():

    # Parent's show method
  def display(self):
       print("Inside Parent2")



  # Defining child class  class
  Child(Parent1, Parent2):

    # Child's show method
  def display(self):
      print("Child display method")
  def show(self):
  super().show()        print("Inside
  Child")
       super().display()



  # Driver's code
  obj = Child()

  obj.show()
  obj.display()
```

Method overriding with multilevel inheritance

```
  # Python program to demonstrate overriding in multilevel inheritance



  class Parent():

    # Parent's show method
  def display(self):
       print("Inside Parent")
```

```
# Inherited or Sub class (Note Parent in bracket)   class
Child(Parent):

    # Child's show method
def show(self):
print("Inside Child")

# Inherited or Sub class (Note Child in bracket)   class
GrandChild(Child):

    # Child's show method      def show(self):          super().show()
# Accessing the child class method with super         print("Inside
GrandChild")

    def display(self):
        super().display()       # Accessing the parent class method with super
print("Inside display of GrandChild")

# Driver code   g
= GrandChild()
g.show()
g.display()
```



```
try:
    Statement
except ExceptionClassName:
   Statement
else:
  Statement
finally:
   Statement
```

```
try:
    Statement
except ExceptionClassName:
   Statement
```

```
try:
    Statement
except ExceptionClassName1:
   Statement
except ExceptionClassName2:
   Statement
finally:
   Statement
```

```
try:
    Statement
```

```
a=10 b=5
c=a/b
print(c)
```

---

```
# program to handle exception -1
a=10 b=0 try:   c=a/b
print(c) except
ZeroDivisionError:
  print('Division by zero not Allowed')

print("End of program")
```

---

```
# program to handle exception with else


a=10 b=5 try:   c=a/b
print(c) except
ZeroDivisionError:
  print('Division by zero not Allowed')

else:
  print('No exception raised')
print("End of program")
```

# program to handle exception with Finally block

```
a=10 b=0 try:   c=a/b
print(c) except
ZeroDivisionError:
  print('Division by zero not Allowed')

else:
  print('No exception raised')

finally:   print("Inside
Finally")
print("End of program")
```

## Abstract Classes in Python

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

**Why use Abstract Base Classes :**
By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

## Define Abstract Class in Python

Python comes with a module called `abc` which provides useful stuff for abstract class.

We can define a class as an abstract `abc.ABC` and define a method as an by abstract method `abc.abstractmethod` by is the abbreviation of abstract base class.

: h

from `abc.ABC` ns

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):
@abstractmethod
def move(self):
        pass
#a = Animal()
# TypeError: Can't instantiate abstract class Animal with abstract methods move
```

```
class Animal():
@abstractmethod
def move(self):
        pass
a = Animal() # No errors
```

Simple Example

```
from abc import ABC, abstractmethod
```

```
class Father(ABC):
@abstractmethod
def disp(self):
      pass    def
show(self):
     print('Conrete Method')

class child(Father):
def disp(self):
     print("Child Class")
     print("Defining Abstract Method")
obj=child() obj.disp()
obj.show()
```

**How Abstract Base classes work :**

By default, Python does not provide abstract classes. Python comes with a module which provides the base for defining Abstract Base classes(ABC) and that module name is ABC. ***ABC*** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

Example

```
# Python program showing
# abstract base class work

from abc import ABC, abstractmethod

class Polygon(ABC):

    # abstract method
def noofsides(self):
      pass

class Triangle(Polygon):

    # overriding abstract method
def noofsides(self):
print("I have 3 sides")

class Pentagon(Polygon):
   # overriding abstract method
def noofsides(self):
print("I have 5 sides")

class Hexagon(Polygon):

   # overriding abstract method
def noofsides(self):
print("I have 6 sides")

class Quadrilateral(Polygon):

   # overriding abstract method
def noofsides(self):
print("I have 4 sides")

# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

Exmaple=-2

```
# Python program showing
# abstract base class work

from abc import ABC, abstractmethod  class
Animal(ABC):

   def move(self):
      pass

class Human(Animal):
    def move(self):
        print("I can walk and run")

class Snake(Animal):

    def move(self):
print("I can crawl")

class Dog(Animal):

    def move(self):
print("I can bark")

class Lion(Animal):

    def move(self):
print("I can roar")

# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()
```

**Implementation Through Subclassing :**
By subclassing directly from the base, we can avoid the need to register the class explicitly. In this case, the Python class management is used to recognize PluginImplementation as implementing the abstract PluginBase.

```
# Python program showing
# implementation of abstract
# class through subclassing

import abc
```

```
class parent:
def geeks(self):
        pass

class child(parent):
def geeks(self):
print("child class")

# Driver code  print(
issubclass(child, parent))  print(
isinstance(child(), parent))
```

**Concrete Methods in Abstract Base Classes :**

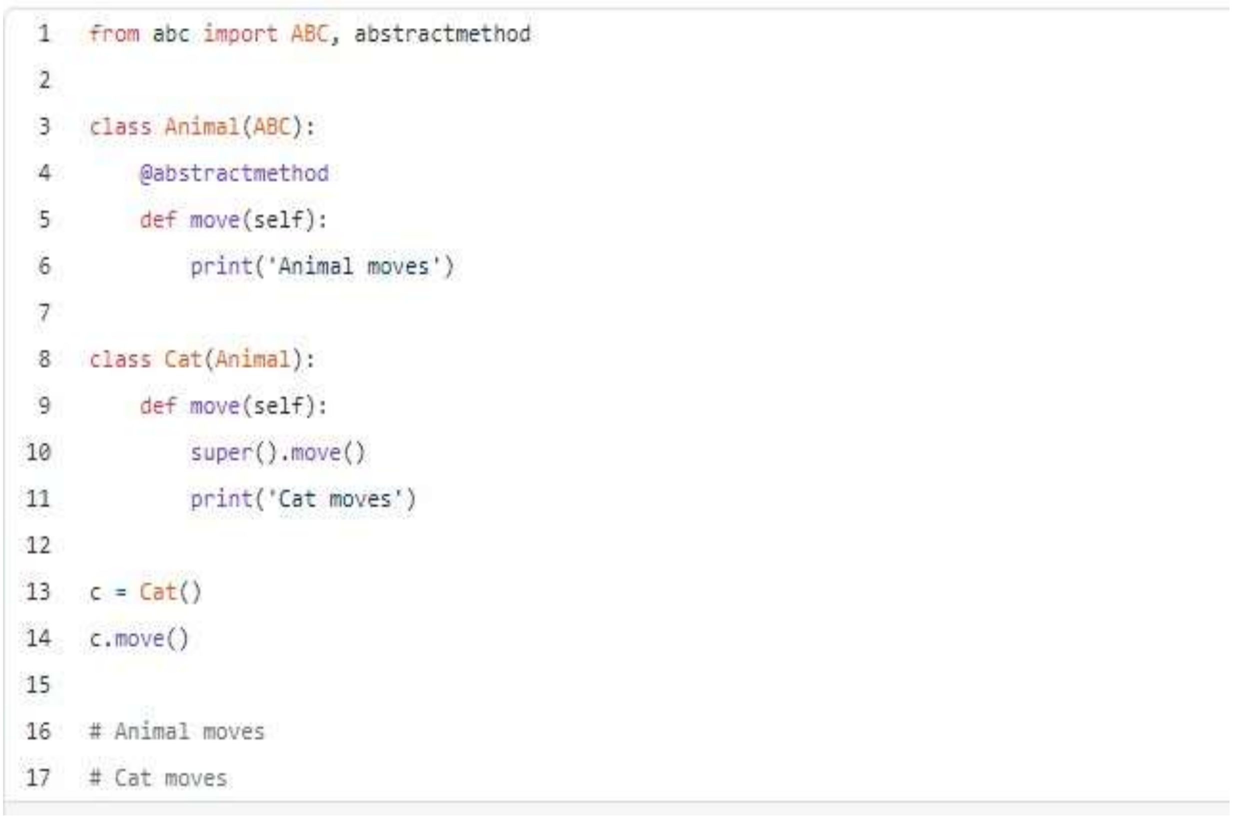
Concrete classes contain only concrete (normal)methods whereas abstract classes may contains both concrete methods and abstract methods. Concrete class provide an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via super().

Let look over the example to invoke the method using super():

```
# Python program invoking a
# method using super()

import abc
from abc import ABC, abstractmethod

class R(ABC):
def rk(self):
        print("Abstract Base Class")

class K(R):      def
rk(self):
super().rk()
print("subclass ")

# Driver code
r = K()  r.rk()
```

## <u>Invoke Methods from Abstract Classes</u>

Actually an abstract method is not needed to be “totally abstract” in Python, which is different with some other object-oriented programming language. We can define some common stuff in an abstract method and use `super()` to invoke it in subclasses.



```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        print('Animal moves')

class Cat(Animal):
    def move(self):
        super().move()
        print('Cat moves')

c = Cat()
c.move()

# Animal moves
# Cat moves
```