Overview of OOP Terminology

Class − A user-defined prototype for an object that **defines a set of attributes** called **data members** (class variables and instance variables) and methods, accessed via dot notation.

Class variable − Defined within a class

Instance variable − A variable that is **defined inside a method** and **belongs only to the current instance** of a class

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 behaviour** to a particular function. The operation performed **varies by the types of arguments** involved.

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**.

Instantiation − The **creation of an instance** of a class.

Method − A special kind of function that is **defined in a class definition.**

Operator overloading − Using same operator in **different ways**

Like + for addition and concatenation

* for multiplication and repetition

Creating Classes

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_suite*** consists of all the **component statements defining class members, data attributes and functions.**

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

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

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

Creating Instance Objects

```
emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

Access the **object's attributes using the dot operator** with object.

```
emp1.displayEmployee()
emp2.displayEmployee()
```

Class variable would be accessed **using class name** as follows

```
print ("Total Employee %d" % Employee.empCount)
```

Result

Name : Zara ,Salary: 2000
Name : Manni ,Salary: 5000
Total Employee 2

```
class Employee:
   'Common base class for all employees'
   empCount = 0
   def __init__(self,na,sal):
      self.name = na
      self.salary = sal
      Employee.empCount += 1
   def displayEmployee(self):
      print ("Name : ",self.name,  ", Salary: ",self.salary)
emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

You can **add, remove, or modify attributes** of classes and objects at any time

```
emp1.age = 27  # Add an 'age' attribute.
emp1.name = ‘sara'  # Modify 'name' attribute.
del emp1.salary  # Delete 'salary' attribute.
```

```
class Employee:
   'Common base class for all employees'
   empCount = 0
   def __init__(self,na,sal):
      self.name = na
      self.salary = sal
      Employee.empCount += 1
   def displayEmployee(self):
      print ("Name : ",self.name,"Salary: ",self.salary)
   def deldisp(self):
        print ("Name : ",self.name)
emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
emp1.age = 27  # Add an 'age' attribute.
emp1.name = ‘sara'  # Modify 'name' attribute.
emp1.displayEmployee()
print("Age:",emp1.age)
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)

del emp1.salary  # Delete 'salary' attribute.
emp1.deldisp()
```

```
Name :  sara Salary:  2000
Age: 27
Name :  Manni Salary:  5000
Total Employee 2
Name :  sara
```

Built-in functions −

The **getattr(obj, name[, default])** − to **access the attribute** of object.

The **hasattr(obj,name)** − to check if an **attribute exists or not.**

The **setattr(obj,name,value)** − to **set an attribute**.

The **delattr(obj, name)** − to **delete** an attribute.

```
print(hasattr(emp1,'salary'))
print(getattr(emp1,'age'))
print(getattr(emp2,'age',30))
setattr(emp2,'salary',30000)
emp2.displayEmployee()
delattr(emp2,'salary')
#emp2.displayEmployee()
emp2.deldisp()
```

Output

```
False
27
30
Name :  Manni Salary:  30000
Name :  Manni
```

Built-In Class Attributes

can be **accessed using dot operator** like any other attribute

__doc__ – **Class documentation** string or none, if undefined.

__name__ – **Class name**.

__module__ – **Module name** in which the class is defined.

__bases__ – **tuple containing the base classes**

__dict__ – **Dictionary** containing the class's namespace.

```
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__ )
```

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {
     '__module__': '__main__', '__doc__': 'Common base class for all
employees',    'empCount': 2, '__init__':
  <function Employee.__init__ at 0x0124F810>, 'displayEmployee':
  <function Employee.displayEmployee at 0x0160D300>,
  '__weakref__':     <attribute '__weakref__' of 'Employee' objects>,
'__dict__':    <attribute '__dict__' of 'Employee' objects>}
```

Destroying Objects (Garbage Collection)

Python **deletes unneeded objects automatically to free the memory space**.

Periodically reclaiming blocks of memory that no longer in use is termed as **Garbage Collection**.

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

__del__() destructor instance of a class that is **about to be destroyed**

```
class Employee:
   'Common base class for all employees'
   empCount = 0
   def __init__(self,na,sal):
      self.name = na
      self.salary = sal
      Employee.empCount += 1
   def displayEmployee(self):
      print ("Name : ",self.name,"Salary: ",self.salary)
   def __del__(self):
         print ("in destructor")
emp1 = Employee("Zara", 2000)
emp1.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
del emp1 #calls destructor __del__()
emp1.displayEmployee()
```

OUTPUT:

```
Name :  Zara Salary:  2000
Total Employee 1
in destructor
Traceback (most recent call last):
  File "C:\Users\LENOVO\tryfinally.py", line 17,
in <module>
    emp1.displayEmployee()
NameError: name 'emp1' is not defined
```

Overloading Functions:

Same function with multiple ways to call

Depending on the function definition, it can be called with zero, one, two ,or more arguments

The argument list could differ in—

1. Number of arguments
2. Data type of parameters
3. Sequence of data type of parameters

But Python does not supports function overloading

Like other languages (for example function overloading in C++) do, python does not supports function overloading. We may overload the function but can only use the latest defined function.

Problem

```
def product(a, b):
    p = a * b
    print(p)
def product(a, b, c):
    p = a * b*c
    print(p)
product(4, 5)
product(4, 5, 5)
```

Solution

```
def product(a, b):
    p = a * b
    print(p)
product(4, 5)
def product(a, b, c):
    p = a * b*c
    print(p)
product(4, 5, 5)
```

We may define many functions of same name and different arguments but we can only use the latest defined function.

Calling the other function will produce an error. Like here calling product(4,5) will produce an error as the latest defined product function takes three arguments.

However we may use other implementation in python to make the same function work differently i.e. as per the arguments.

Operator Overloading

We can use + **operator** for adding numbers and at the same time to concatenate strings.

It is **possible** because + **operator** is **overloaded** by both **int class and str class.** The **operators** are actually methods defined in respective classes.

```
def add(a,b):
    print(a+b)
def mul(a,b):
    print(a*b)
```

add(3,4) → 7

add('hi','hello') → hihello

mul(3,4) → 12

mul('hi',3) → hihihi

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__()

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)

Class Inheritance

create a class by **deriving it from a pre-existing class** by **listing the parent class in parentheses** after the new class name.

The child class **inherits the attributes of its parent class.**

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

Syntax

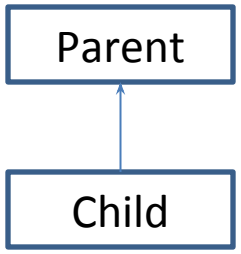
```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

TYPES:

1. Single
2. Multiple
3. Multilevel
4. Hierarchical
5. Hybrid

1. Single Inheritance:

```
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)
```

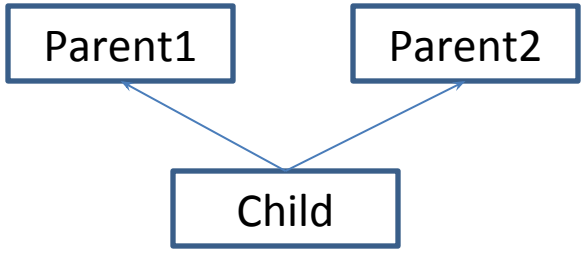


```
class Child(Parent): # define child class
   def __init__(self):
      print ("Calling child constructor")

   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
```

OUTPUT:
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200

Multiple Inheritance

a class can be derived from multiple parent classes (Multiple Inheritance)

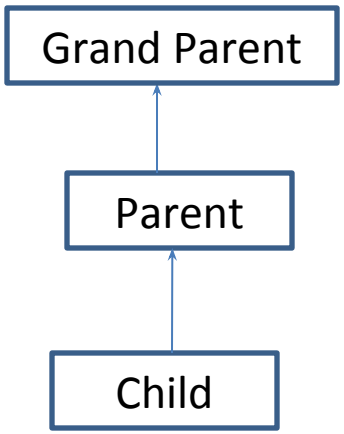



```
class A:        # define your class A
.....
class B:        # define your class B
.....
class C(A, B):   # subclass of A and B
.....
```

- Multiple Inheritance ex1–for example see the programs word doc

Multiple Inheritance ex2

Multi level Inheritance



a class can be derived from other derived class

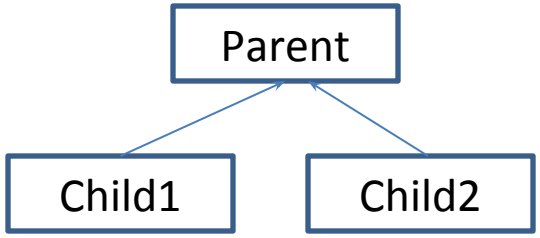
```
class A:        # define your class A
.....
class B(A):         # define your class B subclass of A
.....
class C(B):   # subclass of B
.....
```

- Multilevel Inheritance-for example see the programs word doc

Multilevel Inheritance ex2

Hierarchical Inheritance

Two child classes can be derived from one parent class



class A: # define your class A

.....

class B(A): # define your class B subclass of A

.....

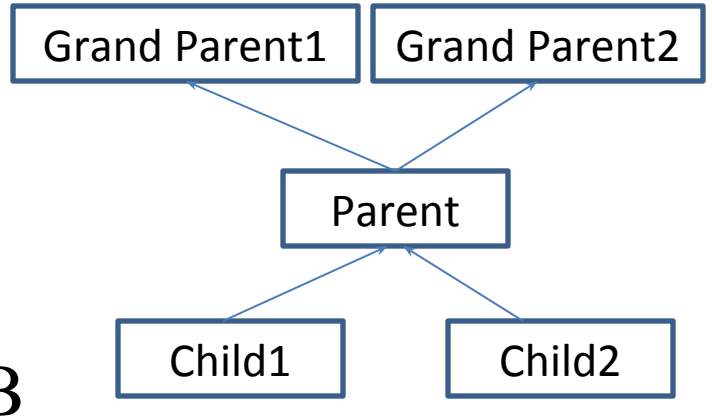
class C(A): # define your class C subclass of A

.....

 Hierarchical Inheritance-for example see the programs word doc

Hybrid Inheritance

A combination of multiple and hierarchical inheritance




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

- Hybrid Inheritance-for example see the programs word doc

issubclass() or isinstance() functions used 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

Overriding Methods

overriding parent's methods is that you may want **special or different functionality** in your subclass.

```
class Parent:
    def __init__(self,a,b):
        self.a=a
        self.b=b
    def display(self):
        print("a+b: ",self.a+self.b)
class Child(Parent):
    def __init__(self,c,d):
        super().__init__(10,20)
        self.c=c
        self.d=d
    def display(self):
        super().display()
        print("c-d:  ",self.c-self.d)
o=Child(60,40)
o.display()
```

Result −

```
a+b:  30
c-d:   20
```

```
class Parent:        # define parent class
   def myMethod(self):
      a=10
      b=20
      print ('Parent add:',a+b)

class Child(Parent): # define child class
   def myMethod(self):
      super().myMethod()
      a='hi'
      b='hello'
      print ('Child add:',a+b)

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

OUTPUT:

```
Parent add: 30
Child add: hihello
```

Method overriding, in object-oriented programming, is a language feature that **allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses** or parent classes.

Rules for method overriding:

- In Python, a method can **only be written in subclass** not in same class.

- The **argument list should be exactly the same** as that of the overridden method.

- The **access level cannot be more restrictive** than the overridden method's access level.
  - For example: if the **super class method is declared public** then the over-ridding method in the sub class **cannot be either private or protected.**

- Instance methods can **be overridden only if they are inherited by the subclass**.

- If a method **cannot be inherited** then it **cannot be overridden.**

- **A subclass** within the **same package** as the instance's superclass **can override any superclass method that is not declared private**.

Data Hiding

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

attributes with a **double underscore prefix will not be directly visible to outsiders**.--- Private attributes

```
class Datahide:
    __hide=20
    data=30
    def method1(self):
        __secret=10
        print("Hide value",Datahide.__hide)
        print("Data value:",Datahide.data)
        print("secret value of method1:",__secret)
    def method2(self):
        print("Method2 secret value:",__secret)
d=Datahide()
d.method1()
#d.method2()
print(Datahide.data)
#print(Datahide.__hide)
print(d._Datahide__hide)
```

access to such attributes as **object._className__attrName**

OUTPUT:

Hide value 20
Data value: 30
secret value of method1: 10
30
20