Python Inheritance

Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more.

It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

```
class BaseClass:

  Body of base class

class DerivedClass(BaseClass):

  Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

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

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

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

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

A better option would be to use the built-in function `super()`. So, `super().__init__(3)` is equivalent to `Polygon.__init__(self,3)`

Two built-in functions `isinstance()` and `issubclass()` are used to check inheritances.

The function `isinstance()` returns `True` if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class `object`.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Similarly, `issubclass()` is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

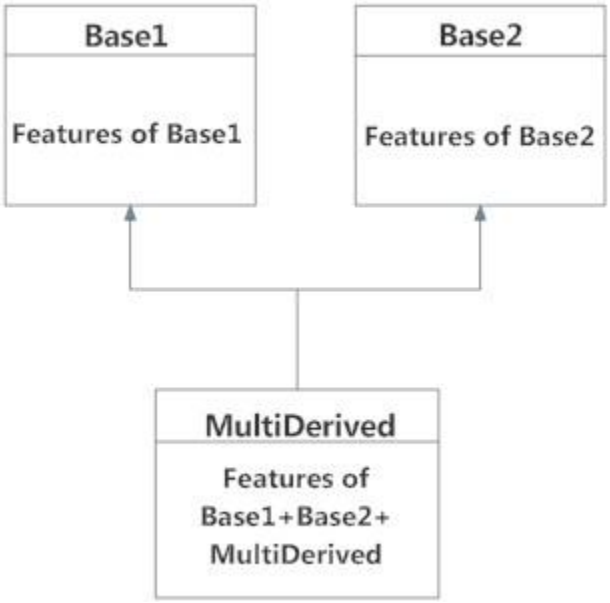
Python Multiple Inheritance

A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.
In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Example

```
class Base1:
```

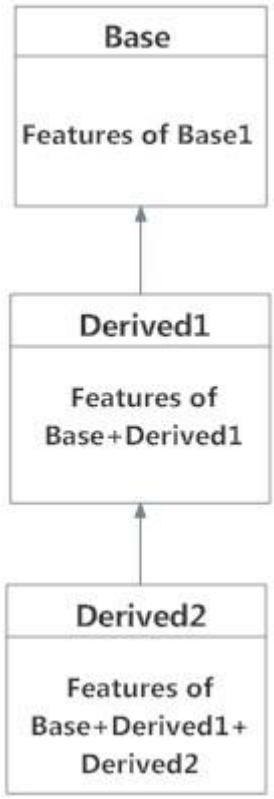
```
    pass

class Base2:
    pass

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




```
1. class Calculation1:
2.     def Summation(self,a,b):
3.         return a+b;
4. class Calculation2:
5.     def Multiplication(self,a,b):
6.         return a*b;
7. class Derived(Calculation1,Calculation2):
8.     def Divide(self,a,b):
9.         return a/b;
10.        d = Derived()
11.        print(d.Summation(10,20))
12.        print(d.Multiplication(10,20))
13.        print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```




```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
            def eat(self):
                print("Eating bread...")
        d = DogChild()
        d.bark()
        d.speak()
        d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Method Overriding

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class

```
1. class Animal:
2.     def speak(self):
3.         print("speaking")
4. class Dog(Animal):
5.     def speak(self):
6.         print("Barking")
7. d = Dog()
8. d.speak()
```

```
1. class Bank:
2.     def getroi(self):
3.         return 10;
4. class SBI(Bank):
5.     def getroi(self):
6.         return 7;
7.
8. class ICICI(Bank):
9.     def getroi(self):
10.            return 8;
11.        b1 = Bank()
12.        b2 = SBI()
13.        b3 = ICICI()
14.        print("Bank Rate of interest:",b1.getroi());
```

```
15.         print("SBI Rate of interest:",b2.getroi());
16.         print("ICICI Rate of interest:",b3.getroi());
```

Data abstraction in python

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

```
1. class Employee:
2.     __count = 0;
3.     def __init__(self):
4.         Employee.__count = Employee.__count+1
5.     def display(self):
6.         print("The number of employees",Employee.__count)
7. emp = Employee()
8. emp2 = Employee()
9. try:
10.         print(emp.__count)
11.     finally:
12.         emp.display()
```

Output:

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```