Inheritance in Python

We often come across different products that have a basic model and an advanced model with added features over and above basic model. A software modelling approach of OOP enables extending the capability of an existing class to build a new class, instead of building from scratch. In OOP terminology, this characteristic is called inheritance, the existing class is called base or parent class, while the new class is called child or sub class.

Inheritance comes into picture when a new class possesses the 'IS A' relationship with an existing class.

Dog IS an animal. Cat also IS an animal. Hence, animal is the base class, while dog and cat are inherited classes.

A quadrilateral has four sides. A rectangle IS a quadrilateral, and so IS a square. Quadrilateral is a base class (also called parent class), while rectangle and square are the inherited classes - also called child classes.

The child class inherits data definitions and methods from the parent class. This facilitates the reuse of features already available. The child class can add a few more definitions or redefine a base class method.

This feature is extremely useful in building a hierarchy of classes for objects in a system. It is also possible to design a new class based upon more than one existing classes. This feature is called multiple inheritance.

The general mechanism of establishing inheritance is illustrated below:

Syntax:

```
class parent:
    statements

class child(parent):
    statements
```

While defining the child class, the name of the parent class is put in the parentheses in front of it, indicating the relation between the two. Instance attributes and methods defined in the parent class will be inherited by the object of the child class.

To demonstrate a more meaningful example, a quadrilateral class is first defined, and it is used as a base class for the rectangle class.

A quadrilateral class having four sides as instance variables and a perimeter() method is defined below:

Example:

```
class quadriLateral:
    def __init__(self, a, b, c, d):
        self.side1=a
        self.side2=b
        self.side3=c
        self.side4=d

    def perimeter(self):
        p=self.side1 + self.side2 + self.side3 + self.side4
        print("perimeter=",p)
```

The constructor (the `__init__()` method) receives four parameters and assigns them to four instance variables. To test the above class, declare its object and invoke the `perimeter()` method.

```
>>>q1=quadriLateral(7,5,6,4)
>>>q1.perimeter()
perimeter=22
```

We now design a rectangle class based upon the `quadriLateral` class (rectangle IS a quadrilateral!). The instance variables and the `perimeter()` method from the base class should be automatically available to it without redefining it.

Since opposite sides of the rectangle are the same, we need only two adjacent sides to construct its object. Hence, the other two parameters of the `__init__()` method are set to none. The `__init__()` method forwards the parameters to the constructor of its base (quadrilateral) class using the **super()** function. The object is initialized with `side3` and `side4` set to none. Opposite sides are made equal by the constructor of rectangle class. Remember that it has automatically inherited the `perimeter()` method, hence there is no need to redefine it.

Example: Inheritance

```
class rectangle(quadriLateral):
    def __init__(self, a, b):
        super().__init__(a, b, a, b)
```

We can now declare the object of the rectangle class and call the `perimeter()` method.

```
>>> r1=rectangle(10, 20)
>>> r1.perimeter()
perimeter=60
```

Overriding in Python

In the above example, we see how resources of the base class are reused while constructing the inherited class. However, the inherited class can have its own instance attributes and methods.

Methods of the parent class are available for use in the inherited class. However, if needed, we can modify the functionality of any base class method. For that purpose, the inherited class

contains a new definition of a method (with the same name and the signature already present in the base class). Naturally, the object of a new class will have access to both methods, but the one from its own class will have precedence when invoked. This is called method overriding.

First, we shall define a new method named `area()` in the rectangle class and use it as a base for the `square` class. The area of rectangle is the product of its adjacent sides.

Example:

```
class rectangle(QuadriLateral):
    def __init__(self, a,b):
        super().__init__(a, b, a, b)

    def area(self):
        a = self.side1 * self.side2
        print("area of rectangle=", a)
```

Let us define the square class which inherits the rectangle class. The `area()` method is overridden to implement the formula for the area of the square as the square of its sides.

Example:

```
class square(rectangle):
    def __init__(self, a):
        super().__init__(a, a)
    def area(self):
        a=pow(self.side1, 2)
        print('Area of Square: ', a)
```

```
>>>s=Square(10)
>>>s.area()
Area of Square: 100
```