**Python inheritance and polymorphism**

Updated on Jan 07, 2020

Inheritance allows programmer to create a general class first then later extend it to more specialized class. It also allows programmer to write better code.

Using inheritance you can inherit all access data fields and methods, plus you can add your own methods and fields, thus inheritance provide a way to organize code, rather than rewriting it from scratch.

In object-oriented terminology when class X extend class Y, then Y is called *super class* or *base class* and X is called *subclass or derived class*. One more point to note that only data fields and method which are not private are accessible by child classes, private data fields and methods are accessible only inside the class.

Syntax to create a subclass is:

|  |  |
| --- | --- |
| 1  2  3 | class SubClass(SuperClass):  # data fields  # instance methods |

Let take an example to illustrate the point.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class Vehicle:      def \_\_init\_\_(self, name, color):          self.\_\_name = name      # \_\_name is private to Vehicle class  self.\_\_color = color      def getColor(self):         # getColor() accessible to class Car          return self.\_\_color      def setColor(self, color):  # setColor accessible outside the class          self.\_\_color = color      def getName(self): # getName()accessible outside the class          return self.\_\_name  class Car(Vehicle):      def \_\_init\_\_(self, name, color, model):  # call parent constructor to set name and color          super().\_\_init\_\_(name, color)          self.\_\_model = model      def getDescription(self):          return self.getName() + self.\_\_model + " in " + self.getColor() + " color"  # in method getDescription we are able to call getName(), getColor() because they are  # accessible to child class through inheritance  c = Car("Ford Mustang", "red", "GT350")  print(c.getDescription())  print(c.getName()) # car has no method getName()  but it is accessible through class Vehicle |

**Expected Output:**

|  |  |
| --- | --- |
| 1  2 | Ford MustangGT350 in red color  Ford Mustang |

Here we have created base class Vehicle and it's subclass Car. Notice that we have not defined getName() in the Car class but we are still able to access it, because the class Car inherits it from the Vehicle class. In the above code super() method is used to call method of the base class. Here is the how super() works

Suppose you need to call a method called get\_information() in the base class from child class , you can do so using the following code.

super().get\_information()

Similarly, you can call base class constructor from child class constructor using the following code.

super().\_\_init\_\_()

**Multiple inheritance**

Unlike languages like Java and C#, python allows multiple inheritance i.e you can inherit from multiple classes at the same time like this,

|  |  |
| --- | --- |
| 1  2  3 | class Subclass(SuperClass1, SuperClass2, ...):     # initializer    # methods |

Let's take an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class MySuperClass1():      def method\_super1(self):          print("method\_super1 method called")  class MySuperClass2():      def method\_super2(self):          print("method\_super2 method called")  class ChildClass(MySuperClass1, MySuperClass2):      def child\_method(self):          print("child method")  c = ChildClass()  c.method\_super1()  c.method\_super2() |

**Expected Output:**

|  |  |
| --- | --- |
| 1  2 | method\_super1 method called  method\_super2 method called |

As you can see becuase ChildClass inherited MySuperClass1, MySuperClass2, object of ChildClass is now able to access method\_super1() and method\_super2().

**Overriding methods**

To override a method in the base class, sub class needs to define a method of same signature. (i.e same method name and same number of parameters as method in base class).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class A():      def \_\_init\_\_(self):          self.\_\_x = 1      def m1(self):          print("m1 from A")  class B(A):      def \_\_init\_\_(self):          self.\_\_y = 1      def m1(self):          print("m1 from B")  c = B()  c.m1() |

**Expected Output:**

m1 from B

Here we are overriding m1() method from the base class. Try commenting m1() method in B class and now m1() method from Base class i.e class A will run.

**Expected Output:**

m1 from A

**isinstance() function**

The isinstance() function is used to determine whether the object is an instance of the class or not.

**Syntax:** isinstance(object, class\_type)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> isinstance(1, int)  True  >>> isinstance(1.2, int)  False  >>> isinstance([1,2,3,4], list)  True |