

继承和多态

1444次阅读

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

```
class Animal(object):
    def run(self):
        print 'Animal is running...'
```

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和Dog类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于Animal实现了run()方法，因此，Dog和Cat作为它的子类，什么事也没干，就自动拥有了run()方法：

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
Animal is running...
```

当然，也可以对子类增加一些方法，比如Dog类：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'
    def eat(self):
        print 'Eating meat...'
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是Dog还是Cat，它们run()的时候，显示的都是Animal is running...，符合逻辑的做法是分别显示Dog is running...和Cat is running...，因此，对Dog和Cat类改进如下：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'

class Cat(Animal):
    def run(self):
        print 'Cat is running...'
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

当子类 and 父类都存在相同的`run()`方法时，我们说，子类的`run()`覆盖了父类的`run()`，在代码运行的时候，总是会调用子类的`run()`。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个`class`的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如`str`、`list`、`dict`没什么两样：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用`isinstance()`判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来`a`、`b`、`c`确实对应着`list`、`Animal`、`Dog`这3种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来`c`不仅仅是`Dog`，`c`还是`Animal`！

不过仔细想想，这是有道理的，因为`Dog`是从`Animal`继承下来的，当我们创建了一个`Dog`的实例`c`时，我们认为`c`的数据类型是`Dog`没错，但`c`同时也是`Animal`也没错，`Dog`本来就是`Animal`的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

`Dog`可以看成`Animal`，但`Animal`不可以看成`Dog`。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个`Animal`类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入`Animal`的实例时，`run_twice()`就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入`Dog`的实例时，`run_twice()`就打印出：

```
>>> run_twice(Dog())
```

```
Dog is running...
Dog is running...
```

当我们传入Cat的实例时，run_twice()就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Tortoise类型，也从Animal派生：

```
class Tortoise(Animal):
    def run(self):
        print 'Tortoise is running slowly...'
```

当我们调用run_twice()时，传入Tortoise的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个Animal的子类，不必对run_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

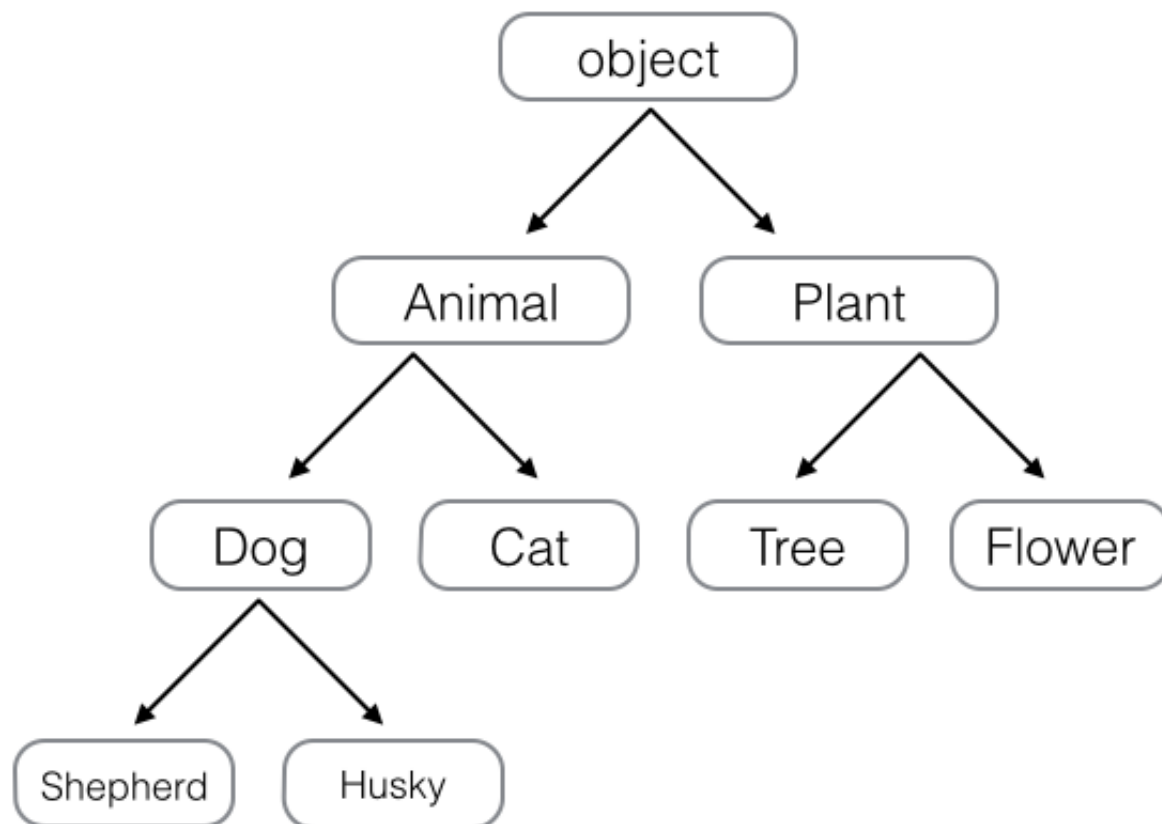
多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Animal子类；

对修改封闭：不需要修改依赖Animal类型的run_twice()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



小结

继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写；

有了继承，才能有多态。在调用类实例方法的时候，尽量把变量视作父类类型，这样，所有子类类型都可以正常被接收；

旧的方式定义Python类允许不从object类继承，但这种编程方式已经严重不推荐使用。任何时候，如果没有合适的类可以继承，就继承自object类。
