Python的面向对象 (二)

封装、继承和多态

在面向对象的编程设计中的三个特点：封装、继承、多态。在此我们就学习一下Python语言中面向对象编程的三个特点。

封装

封装就是在由类创建对象时候，每个被创建的对象中都有这个类的所有的属性和方法，当方法需要执行时候，需要使用的变量保存在对象中，这样对象所使用的方法对对象来说就是隐藏的，看不见的。简单来说：就是私有化方法，把一些属性和方法私有化，外部无法调用，增强了代码的安全性。
那么如何将属性或者方法进行私有化呢？在要进行私有化的属性名前面添加两个下划线`"__"`，或者在要进行私有化的方法名字前面添加两个下划线`“__”`。
下面举例说明：

未进行封装（私有化）处理

```
# 定义一个BOY类，
class BOY():
    '''
    利用初始化函数初始化类的共同特性，name，age
    此处的属性名字均未作处理，保持原样
    '''
    def __init__(self,name,age):
        # 此处的属性name,age都未作私有化处理
        self.name = name
        self.age = age
    # 此处定义一个打印函数printer，同样未封装
    def printer(self):
    print self.age
# 对类进行了实例化，出入了相应的参数
Me = BOY("Lynn",25)
print Me.name
Me.printer()
```

因为上面函数中定义的属性和方法军未封装，那么在类的外面我们是可以调用类中的属性和方法的，即通过实例调用。
下面为上面程序运行的结果：

```
>>> Lynn
>>> 25
```

进行封装（私有化）处理后

```
# 定义一个BOY类，
class BOY():
    '''
    利用初始化函数初始化类的共同特性，name，age
    此处的属性名字均添加双下划线'__'，封装私有化处理
    '''
    def __init__(self,name,age):
        # 此处的属性name,age都进行私有化处理
        self.__name = name
        self.__age = age
    # 此处定义的方法也进行封装
    def __printer(self):
        print self.age
# 对类进行了实例化，出入了相应的参数
Me = BOY("Lynn",25)
print Me.name
Me.printer()
```

按照Python私有化定义来看，对类的属性和方法进行私有化，类外部是无法正常对其调用的，那么程序运行的结果是：

```
Traceback (most recent call last):
  File "E:/PycharmProjects/MD/test.py", line 14, in <module>
    print Me.name
AttributeError: BOY instance has no attribute 'name'
```

此时控制台提示属性错误(AttributeError)，*BOY instance has no attribute 'name'*即BOY实例中不含有*name*属性。说明我们在累的外面不能通过实例去访问类里面的私有化属性，这个属性已经被类私有化，通过这种方式提高了代码的安全性、私密性。

由于程序的执行的顺序性，并没有抛出调用*printer*的错误，如果将属性不做私有化处理（即去掉属性*name*，*age*前面的两条下滑线s），只对方法做私有化封装处理，那么程序的运行结果就是：

```
Lynn
Traceback (most recent call last):
  File "E:/PycharmProjects/MD/test.py", line 15, in <module>
    Me.printer()
AttributeError: BOY instance has no attribute 'printer'
```

因为*name*，*age*属性为能够正常调用，所以能够正常显示名字，但是*printer*方法由于做了私有化封装不能正常调用，抛出异常。

将代码封装私有化之后代码的健壮性安全性提高了，那么问题又来了，如果我们需要得到*name*，*age*该怎么办？这个时候就需要在类的内部添加一个方法用于外部调用以得到*name*，*age*属性。同样，如果类在初始化的时候给了*name*赋值，当我们需要另外再给*name*赋新值的时候，这个时候我们也可以在类的内部增加一个修改*name*的方法，然后通过外部调用来修改值。

就像下面：

```
#-*- coding:utf-8 -*-

class BOY():

    def __init__(self,age, name = "David"):
        self.__name = name
        self.__age = age

    # 在类的内部定义一个专门用来获取name的方法
    def get_name(self):
        print self.__name
        return None
    # 在类的内部定义一个专门用来获取age的方法
    def get_age(self):
        print self.__age
        return None
    # 定义修改name值得方法
    def change_name(self,name):
        self.name = name

# 对方法进行实例化
Me = BOY(25)
'''
 name,age私有化封装之后无法直接调用，在此调用打印
 name,age的方法 查看初始化name，age的值
'''
Me.get_name()
Me.get_age()
'''
 调用修改name的方法，将初始化的 name = "David"
 修改为 name = "Lynn",调用方法如下，将新值作为
 参数传入name
'''
Me.change_name("Lynn")
# 再次调用打印name方法，查看修改是否成功
Me.get_name()
```

上面程序的运行结果如下：

```
>>> David
>>> 25
>>> Lynn
```

说明调用打印*name*和*age*的方法成功，并成功修改了*name*。

继承

在面向对象程序设计中，当我们定义一个类的时候，这个类的属性可以使用别的已定义类的属性，那么这种行为就是继承。

举例说明：

> 机器人具有一些其与人类不同的属性，比如，不用吃饭、不用喝水、可以帮助人类干一些人类不能胜任的工作。那么，做饭的机器人也是机器人的一个类别，同理它也具有机器人的属性。这些属性只需要在定义机器人的时候添加一次就可以使其被重用，这就是继承。

那么Python中的那些对象能够被继承呢？
类，类中的属性，类中的方法 这些都可以被继承。

属性继承

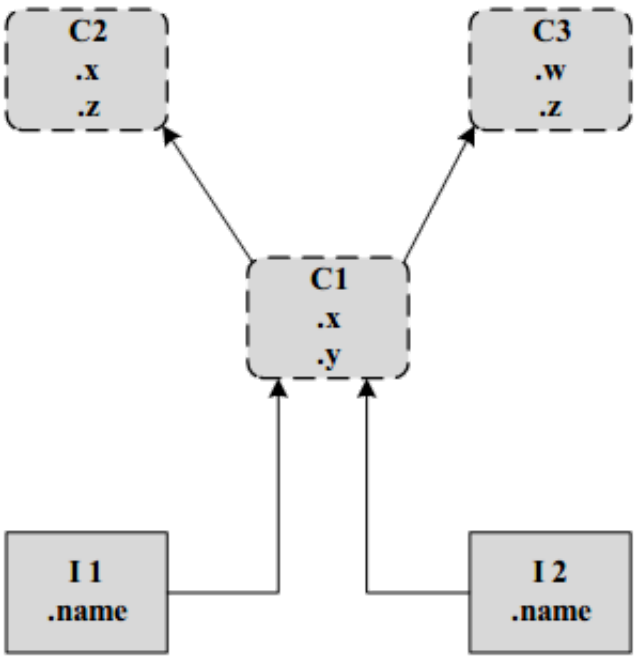
在Python中面向对象的程序设计读取器调用的属性都可以化成以下简单的句子表达式:

```
object.attribute
```

当我们对*class*语句产生的对象使用这种表达方式的时候，这个表达式就会启动Python搜索，搜索对象连接的书，进而寻找*attribute*首次出现的地方。等于下面的语言：

> 找出attribute首次出现的地方，先搜索object，然后是对该对象之上的所有类，从下到上，从左到右。

在Python中就是这样，我们通过代码建立连接对象树，每次使用*object.attribute*表达式的时候，Python确实会在运行期间去“爬树”，来搜索属性。



如上图所示，是一个类树图，地段有两个实例（*I1*和*I2*），在它上面有一个类*C1*，而顶端有两个超类（*C2*和*C3*），继承就是由下到上搜索此树，来寻找属性名称所出现的最低的地方，代码中隐含了这种数的形状。（关于“超类，子类”下面会举例进行介绍）

假设我们已经创建了上图中的类树，然后在代码中出现*I2.w*这个代码就会立即启动继承，因为这是一个*attribute.object*的表达形式，于是就会触发上图中对树的搜索，Python就会查看I2和其上的对象来搜索属性*w*，就是按照以下顺序进行搜索连接的对象：

> I2,C1,C2,C3

找到首个*w*之后就会停止搜索（如果找不到，就抛出错误）。在此例子中，一致搜索到*C3*时才会找到*w*，即通过自动搜索，*I2.w*会解析为*C3.w*,通过OOP术语而言，*I2*从*C3*处继承了属性*w*。
同理可得：

- *I1.x*和*I2.x*都会在*C1*中找到*x*并停止搜索，因为*C1*比*C2*位置更低；
- *I1.y*和*I2.y*都会在*C1*中找到*y*，因为这是*y*唯一出现的位置；
- *I1.z*和*I2.z*都会在*C2*中找到*z*，因为*C2*比*C3*更靠左；
- *I2.name*会找到*I2*中的*name*，不需要爬树。

下面举一个简单的例子

```
# 定义一个Animal类，包含一个方法
class Animal(object):
    def __init__(self,name):
        self.name = name
    def eat(self):
        print 'I am eating……'
'''
定义了一个Dog类，后面括号内写的是Animal
说明这个Dog类继承了Animal类，包括其属性
方法，等。
'''
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

上面例子中*Dog*和*Cat*新类就被称为子类（**Subclass**），*Animal*类就被称为基类、父类和超类（**Baseclass**，**Super Class**），并且两个子类*Dog*,*Cat*均继承了*Animal*的属性。

方法的继承

上面我们介绍了Python是如何继承属性的，其实对于类中方法的继承与属性相似。上面介绍了应用*I2.w*通过继承搜索传到*C3.w*，如果上面的*I2.w*引用的是一个类函数的调用，那么当其通过继承搜索变成*C3.w*时其含义是“调用*C3.w*函数以处理I2”。也就是说，Python会自动将*I2.w()*调用映射为*C3.w(I2)*调用，传入该实例作为继承函数的第一个参数。
下面举例进行说明：

```
#-*- coding:utf-8 -*-
'''
 定义超类 Animal并且具有name属性
 包含一个类方法eat
'''
class Animal(object):
    def __init__(self,name):
        self.name = name
    def eat(self):
        print 'I am eating……'

# 定义了一个子类Dog
class Dog(Animal):
    pass
# 定义了一个子类Cat
class Cat(Animal):
    pass
# 实例化Dog并给其一个名字'Alpha'
gou = Dog('Alpha')

# 测试能不能继承属性和方法
print gou.name
gou.eat()
```

上面代码运行结果：

```
>>>Alpha
>>> I am eating……
```

说明两个子类分别成功的继承了超类的属性、方法。
此外，继承除了上面介绍的子类从单个父类继承属性和方法，子类还可以从多个父类那里继承属性和方法，这种集成方式称为多重继承

```
class C2: ...:
class C3: ...:
class C1(C2,C3):

    def __init__(self,...)
        pass
    def ...:
        pass
I1 = C1()
I2 = C1()
```

多重继承的父类们写在子类小括号里面，搜索顺寻从左到右。在上面的例子中就是先搜索*C2*，再搜索*C3*。
关于多重继承在此不多介绍。