python编程之面向对象编程

* 1. 定义类并创建实例

类通过class关键字定义。以Person为例，定义一个Person类如下：

class Person(object):

pass

按照Python的编程习惯，类名以大写字母开头，紧接着是(object)，表示该类是从哪个类继承下来的。类的继承将在后面的章节讲解，现在我们只需要简单地从object类继承。

有了Person类的定义，就可以创建出具体的xiaoming、xiaohong等实例。创建实例使用类名+()，类似函数调用的形式创建：

xiaoming = Person()

xiaohong = Person()

* 1. 属性

虽然可以通过Person类创建出xiaoming、xiaohong等实例，但是这些实例看上除了地址不同外，没有什么其他不同。在现实世界中，区分xiaoming、xiaohong要依靠他们各自的名字、性别、生日等属性。

如何让每个实例拥有各自不同的属性？由于Python是动态语言，对每一个实例，都可以直接给他们的属性赋值，例如，给xiaoming这个实例加上name、gender和birth属性：

xiaoming = Person()

xiaoming.name = 'Xiao Ming'

xiaoming.gender = 'Male'

xiaoming.birth = '1990-1-1'

* 1. 初始化实例属性

在定义 Person 类时，可以为Person类添加一个特殊的\_\_init\_\_()方法，当创建实例时，\_\_init\_\_()方法被自动调用，我们就能在此为每个实例都统一加上以下属性：

class Person(object):

def \_\_init\_\_(self, name, gender, birth):

self.name = name

self.gender = gender

self.birth = birth

\_\_init\_\_()方法的第一个参数必须是self（也可以用别的名字，但建议使用习惯用法），后续参数则可以自由指定，和定义函数没有任何区别。

相应地，创建实例时，就必须要提供除self以外的参数：

xiaoming = Person('Xiao Ming', 'Male', '1991-1-1')

xiaohong = Person('Xiao Hong', 'Female', '1992-2-2')

* 1. 访问限制

Python对属性权限的控制是通过属性名来实现的，如果一个属性由双下划线开头(\_\_)，该属性就无法被外部访问。看例子：

class Person(object):

def \_\_init\_\_(self, name):

self.name = name

self.\_title = 'Mr'

self.\_\_job = 'Student'

p = Person('Bob')

print p.name

# => Bob

print p.\_title

# => Mr

print p.\_\_job

# => Error

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Person' object has no attribute '\_\_job'

可见，只有以双下划线开头的"\_\_job"不能直接被外部访问。

但是，如果一个属性以"\_\_xxx\_\_"的形式定义，那它又可以被外部访问了，以"\_\_xxx\_\_"定义的属性在Python的类中被称为特殊属性，有很多预定义的特殊属性可以使用，通常我们不要把普通属性用"\_\_xxx\_\_"定义。

以单下划线开头的属性"\_xxx"虽然也可以被外部访问，但是，按照习惯，他们不应该被外部访问。

类属性

类是模板，而实例则是根据类创建的对象。

绑定在一个实例上的属性不会影响其他实例，但是，类本身也是一个对象，如果在类上绑定一个属性，则所有实例都可以访问类的属性，并且，所有实例访问的类属性都是同一个！也就是说，实例属性每个实例各自拥有，互相独立，而类属性有且只有一份。

定义类属性可以直接在class中定义：

class Person(object):

address = 'Earth'

def \_\_init\_\_(self, name):

self.name = name

因为类属性是直接绑定在类上的，所以，访问类属性不需要创建实例，就可以直接访问：

print Person.address

# => Earth

对一个实例调用类的属性也是可以访问的，所有实例都可以访问到它所属的类的属性：

p1 = Person('Bob')

p2 = Person('Alice')

print p1.address

# => Earth

print p2.address

# => Earth

由于Python是动态语言，类属性也是可以动态添加和修改的：

Person.address = 'China'

print p1.address

# => 'China'

print p2.address

# => 'China'

因为类属性只有一份，所以，当Person类的address改变时，所有实例访问到的类属性都改变了。

任务

请给 Person 类添加一个类属性 count，每创建一个实例，count 属性就加 1，这样就可以统计出一共创建了多少个 Person 的实例。

class Person(object):

count = 0

def \_\_init\_\_(self, name):

Person.count = Person.count + 1

self.name = name

p1 = Person('Bob')

print Person.count

# => 1

p2 = Person('Alice')

print Person.count

# => 2

p3 = Person('Tim')

print Person.count

# => 3

类属性与实例属性

修改类属性会导致所有实例访问到的类属性全部都受影响，但是，如果在实例变量上修改类属性会发生什么问题呢？

class Person(object):

address = 'Earth'

def \_\_init\_\_(self, name):

self.name = name

p1 = Person('Bob')

p2 = Person('Alice')

print 'Person.address = ' + Person.address

p1.address = 'China'

print 'p1.address = ' + p1.address

print 'Person.address = ' + Person.address

print 'p2.address = ' + p2.address

结果如下：

Person.address = Earth

p1.address = China

Person.address = Earth

p2.address = Earth

我们发现，在设置了 p1.address = 'China' 后，p1访问 address 确实变成了 'China'，但是，Person.address和p2.address仍然是'Earch'，怎么回事？

原因是 p1.address = 'China'并没有改变 Person 的 address，而是给 p1这个实例绑定了实例属性address ，对p1来说，它有一个实例属性address（值是'China'），而它所属的类Person也有一个类属性address，所以:

访问 p1.address 时，优先查找实例属性，返回'China'。

访问 p2.address 时，p2没有实例属性address，但是有类属性address，因此返回'Earth'。

可见，当实例属性和类属性重名时，实例属性优先级高，它将屏蔽掉对类属性的访问。

当我们把 p1 的 address 实例属性删除后，访问 p1.address 就又返回类属性的值 'Earth'了：

del p1.address

print p1.address

# => Earth

可见，千万不要在实例上修改类属性，它实际上并没有修改类属性，而是给实例绑定了一个实例属性。

定义实类方法

实例的方法就是在类中定义的函数，它的第一个参数永远是self，指向调用该方法的实例本身，其他参数和一个普通函数是完全一样的：

class Person(object):

def \_\_init\_\_(self, name):

self.\_\_name = name

def get\_name(self):

return self.\_\_name

get\_name(self)就是一个实例方法，它的第一个参数是self。\_\_init\_\_(self, name)其实也可看做是一个特殊的实例方法。

调用实例方法必须在实例上调用：

p1 = Person('Bob')

print p1.get\_name() # self不需要显式传入

# => Bob

在实例方法内部，可以访问所有实例属性，这样，如果外部需要访问私有属性，可以通过方法调用获得，这种数据封装的形式除了能保护内部数据一致性外，还可以简化外部调用的难度。

方法也是属性

因为方法也是一个属性，所以，它也可以动态地添加到实例上，只是需要用 types.MethodType() 把一个函数变为一个方法：

import types

def fn\_get\_grade(self):

if self.score >= 80:

return 'A'

if self.score >= 60:

return 'B'

return 'C'

class Person(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

p1 = Person('Bob', 90)

p1.get\_grade = types.MethodType(fn\_get\_grade, p1, Person)

print p1.get\_grade()

# => A

p2 = Person('Alice', 65)

print p2.get\_grade()

# ERROR: AttributeError: 'Person' object has no attribute 'get\_grade'

# 因为p2实例并没有绑定get\_grade

给一个实例动态添加方法并不常见，直接在class中定义要更直观

定义类方法

和属性类似，方法也分实例方法和类方法。

在class中定义的全部是实例方法，实例方法第一个参数 self 是实例本身。

要在class中定义类方法，需要这么写：

class Person(object):

count = 0

@classmethod

def how\_many(cls):

return cls.count

def \_\_init\_\_(self, name):

self.name = name

Person.count = Person.count + 1

print Person.how\_many()

p1 = Person('Bob')

print Person.how\_many()

通过标记一个 @classmethod，该方法将绑定到 Person 类上，而非类的实例。类方法的第一个参数将传入类本身，通常将参数名命名为 cls，上面的 cls.count 实际上相当于 Person.count。

因为是在类上调用，而非实例上调用，因此类方法无法获得任何实例变量，只能获得类的引用。