# Python 面向对象

Python从设计之初就已经是一门面向对象的语言，正因为如此，在Python中创建一个类和对象是很容易的。本章节我们将详细介绍Python的面向对象编程。

如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习Python的面向对象编程。

接下来我们先来简单的了解下面向对象的一些基本特征。

## 面向对象技术简介

* **类(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
* **类变量：**类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
* **数据成员：**类变量或者实例变量用于处理类及其实例对象的相关的数据。
* **方法重写：**如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
* **实例变量：**定义在方法中的变量，只作用于当前实例的类。
* **继承：**即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）。
* **实例化：**创建一个类的实例，类的具体对象。
* **方法：**类中定义的函数。
* **对象：**通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

## 创建类

使用class语句来创建一个新类，class之后为类的名称并以冒号结尾，如下实例:

class ClassName:

'类的帮助信息' #类文档字符串

class\_suite #类体

类的帮助信息可以通过ClassName.\_\_doc\_\_查看。

class\_suite 由类成员，方法，数据属性组成。

### 实例

以下是一个简单的Python类实例:

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class Employee:

'所有员工的基类'

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

* empCount变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用Employee.empCount访问。
* 第一种方法\_\_init\_\_()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法

## 创建实例对象

要创建一个类的实例，你可以使用类的名称，并通过\_\_init\_\_方法接受参数。

"创建 Employee 类的第一个对象"

emp1 = Employee("Zara", 2000)"创建 Employee 类的第二个对象"

emp2 = Employee("Manni", 5000)

## 访问属性

您可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量:

emp1.displayEmployee()

emp2.displayEmployee()print "Total Employee %d" % Employee.empCount

完整实例：

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class Employee:

'所有员工的基类'

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

"创建 Employee 类的第一个对象"

emp1 = Employee("Zara", 2000)"创建 Employee 类的第二个对象"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()print "Total Employee %d" % Employee.empCount

执行以上代码输出结果如下：

Name : Zara ,Salary: 2000Name : Manni ,Salary: 5000Total Employee 2

你可以添加，删除，修改类的属性，如下所示：

emp1.age = 7 # 添加一个 'age' 属性

emp1.age = 8 # 修改 'age' 属性del emp1.age # 删除 'age' 属性

你也可以使用以下函数的方式来访问属性：

* getattr(obj, name[, default]) : 访问对象的属性。
* hasattr(obj,name) : 检查是否存在一个属性。
* setattr(obj,name,value) : 设置一个属性。如果属性不存在，会创建一个新属性。
* delattr(obj, name) : 删除属性。

hasattr(emp1, 'age') # 如果存在 'age' 属性返回 True。

getattr(emp1, 'age') # 返回 'age' 属性的值

setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8

delattr(emp1, 'age') # 删除属性 'age'

## Python内置类属性

* \_\_dict\_\_ : 类的属性（包含一个字典，由类的数据属性组成）
* \_\_doc\_\_ :类的文档字符串
* \_\_name\_\_: 类名
* \_\_module\_\_: 类定义所在的模块（类的全名是'\_\_main\_\_.className'，如果类位于一个导入模块mymod中，那么className.\_\_module\_\_ 等于 mymod）
* \_\_bases\_\_ : 类的所有父类构成元素（包含了一个由所有父类组成的元组）

Python内置类属性调用实例如下：

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class Employee:

'所有员工的基类'

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

print "Employee.\_\_doc\_\_:", Employee.\_\_doc\_\_print "Employee.\_\_name\_\_:", Employee.\_\_name\_\_print "Employee.\_\_module\_\_:", Employee.\_\_module\_\_print "Employee.\_\_bases\_\_:", Employee.\_\_bases\_\_print "Employee.\_\_dict\_\_:", Employee.\_\_dict\_\_

执行以上代码输出结果如下：

Employee.\_\_doc\_\_: 所有员工的基类Employee.\_\_name\_\_: EmployeeEmployee.\_\_module\_\_: \_\_main\_\_Employee.\_\_bases\_\_: ()Employee.\_\_dict\_\_: {'\_\_module\_\_': '\_\_main\_\_', 'displayCount': <function displayCount at 0x10a939c80>, 'empCount': 0, 'displayEmployee': <function displayEmployee at 0x10a93caa0>, '\_\_doc\_\_': '\xe6\x89\x80\xe6\x9c\x89\xe5\x91\x98\xe5\xb7\xa5\xe7\x9a\x84\xe5\x9f\xba\xe7\xb1\xbb', '\_\_init\_\_': <function \_\_init\_\_ at 0x10a939578>}

## python对象销毁(垃圾回收)

Python 使用了引用计数这一简单技术来跟踪和回收垃圾。

在 Python 内部记录着所有使用中的对象各有多少引用。

一个内部跟踪变量，称为一个引用计数器。

当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也就是说， 这个对象的引用计数变为0 时， 它被垃圾回收。但是回收不是"立即"的， 由解释器在适当的时机，将垃圾对象占用的内存空间回收。

a = 40 # 创建对象 <40>

b = a # 增加引用， <40> 的计数

c = [b] # 增加引用. <40> 的计数

del a # 减少引用 <40> 的计数

b = 100 # 减少引用 <40> 的计数

c[0] = -1 # 减少引用 <40> 的计数

垃圾回收机制不仅针对引用计数为0的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充， 垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。 在这种情况下， 解释器会暂停下来， 试图清理所有未引用的循环。

### 实例

析构函数 \_\_del\_\_ ，\_\_del\_\_在对象销毁的时候被调用，当对象不再被使用时，\_\_del\_\_方法运行：

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class Point:

def \_\_init\_\_( self, x=0, y=0):

self.x = x

self.y = y

def \_\_del\_\_(self):

class\_name = self.\_\_class\_\_.\_\_name\_\_

print class\_name, "销毁"

pt1 = Point()

pt2 = pt1

pt3 = pt1print id(pt1), id(pt2), id(pt3) # 打印对象的iddel pt1del pt2del pt3

以上实例运行结果如下：

3083401324 3083401324 3083401324Point 销毁

**注意：**通常你需要在单独的文件中定义一个类，

## 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。

需要注意的地方：**继承语法** class 派生类名（**基类名**）：//... 基类名写在括号里，基本类是在类定义的时候，在元组之中指明的。

在python中继承中的一些特点：

* 1：在继承中基类的构造（\_\_init\_\_()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
* 2：在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。区别于在类中调用普通函数时并不需要带上self参数
* 3：Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。

如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" 。

**语法：**

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

实例：

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class Parent: # 定义父类

parentAttr = 100

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

print "调用父类构造函数"

def parentMethod(self):

print '调用父类方法'

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

print "父类属性 :", Parent.parentAttr

class Child(Parent): # 定义子类

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

print "调用子类构造方法"

def childMethod(self):

print '调用子类方法 child method'

c = Child() # 实例化子类

c.childMethod() # 调用子类的方法

c.parentMethod() # 调用父类方法

c.setAttr(200) # 再次调用父类的方法

c.getAttr() # 再次调用父类的方法

以上代码执行结果如下：

调用子类构造方法调用子类方法 child method调用父类方法父类属性 : 200

你可以继承多个类

class A: # 定义类 A.....

class B: # 定义类 B.....

class C(A, B): # 继承类 A 和 B.....

你可以使用issubclass()或者isinstance()方法来检测。

* issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)

print issubclass(Child,Parent)

print issubclass(Parent,Child)

* isinstance(obj, Class) 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。

print isinstance(c, Child)

## 方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class Parent: # 定义父类

def myMethod(self):

print '调用父类方法'

class Child(Parent): # 定义子类

def myMethod(self):

print '调用子类方法'

c = Child() # 子类实例

c.myMethod() # 子类调用重写方法

执行以上代码输出结果如下：

调用子类方法

## 基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

|  |  |
| --- | --- |
| **序号** | **方法, 描述 & 简单的调用** |
| 1 | **\_\_init\_\_ ( self [,args...] )** 构造函数 简单的调用方法: *obj = className(args)* |
| 2 | **\_\_del\_\_( self )** 析构方法, 删除一个对象 简单的调用方法 : *dell obj* |
| 3 | **\_\_repr\_\_( self )** 转化为供解释器读取的形式 简单的调用方法 : *repr(obj)* |
| 4 | **\_\_str\_\_( self )** 用于将值转化为适于人阅读的形式 简单的调用方法 : *str(obj)* |
| 5 | **\_\_cmp\_\_ ( self, x )** 对象比较 简单的调用方法 : *cmp(obj, x)* |

## 运算符重载

Python同样支持运算符重载，实例如下：

#!/usr/bin/python

class Vector:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_str\_\_(self):

return 'Vector (%d, %d)' % (self.a, self.b)

def \_\_add\_\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)print v1 + v2

以上代码执行结果如下所示:

Vector(7,8)

## 类属性与方法

### 类的私有属性

**\_\_private\_attrs**：两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。在类内部的方法中使用时 **self.\_\_private\_attrs**。

### 类的方法

在类地内部，使用def关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数self,且为第一个参数

### 类的私有方法

**\_\_private\_method**：两个下划线开头，声明该方法为私有方法，不能在类地外部调用。在类的内部调用 **self.\_\_private\_methods**

### 实例

#!/usr/bin/python# -\*- coding: UTF-8 -\*-

class JustCounter:

\_\_secretCount = 0 # 私有变量

publicCount = 0 # 公开变量

def count(self):

self.\_\_secretCount += 1

self.publicCount += 1

print self.\_\_secretCount

counter = JustCounter()

counter.count()

counter.count()print counter.publicCountprint counter.\_\_secretCount # 报错，实例不能访问私有变量

Python 通过改变名称来包含类名:

122Traceback (most recent call last):

File "test.py", line 17, in <module>

print counter.\_\_secretCount # 报错，实例不能访问私有变量AttributeError: JustCounter instance has no attribute '\_\_secretCount'

Python不允许实例化的类访问私有数据，但你可以使用 **object.\_className\_\_attrName** 访问属性，将如下代码替换以上代码的最后一行代码：

.........................print counter.\_JustCounter\_\_secretCount

执行以上代码，执行结果如下：

1222