

# Python 面向对象

## 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。
- **局部变量:** 定义在方法中的变量，只作用于当前实例的类。
- **实例变量:** 在类的声明中，属性是用变量来表示的。这种变量就称为实例变量，是在类声明的内部但是在类的其他成员方法之外声明的。
- **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：

```
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

```
#!/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
- `self` 代表类的实例，**`self` 在定义类的方法时是必须有的**，虽然在调用时不必传入相应的参数。

### `self` 代表类的实例，而非类

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称，按照惯例它的名称是 `self`。

从执行结果可以很明显的看出，`self` 代表的是类的实例，代表当前对象的地址，而 `self.__class__` 则指向类。

`self` 不是 python 关键字，我们把他换成 `runoob` 也是可以正常执行的

### 创建实例对象

实例化类其他编程语言中一般用关键字 `new`，但是在 `Python` 中并没有这个关键字，类的实例化类似函数调用方式。

以下使用类的名称 `Employee` 来实例化，并通过 `__init__` 方法接收参数。

```
"创建 Employee 类的第一个对象"  
emp1 = Employee("Zara", 2000)
```

### 访问属性

您可以使用点号 `.` 来访问对象的属性。使用如下类的名称访问类变量：

```
emp1.displayEmployee()
```

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
```

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 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。

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。

## 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。

通过继承创建的新类称为子类或派生类，被继承的类称为基类、父类或超类。

### 继承语法

```
class 派生类名(基类名)
    ...
```

在 python 中继承中的一些特点：

- 1、如果在子类中需要父类的构造方法就需要显示的调用父类的构造方法，或者不重写父类的构造方法。详细说明可查看：[python 子类继承父类构造函数说明](#)。
- 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。区别在于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, ...])
```

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,super)`
- `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`。

## 方法重写

如果你的父类的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

## 基础重载方法

序号	方法, 描述 & 简单的调用
1	<code>__init__ ( self [,args...])</code> 构造函数 简单的调用方法: <code>obj = className(args)</code>
2	<code>__del__( self )</code> 析构方法, 删除一个对象 简单的调用方法 : <code>del obj</code>
3	<code>__repr__( self )</code> 转化为供解释器读取的形式 简单的调用方法 : <code>repr(obj)</code>
4	<code>__str__( self )</code> 用于将值转化为适于人阅读的形式 简单的调用方法 : <code>str(obj)</code>
5	<code>__cmp__ ( self, x )</code> 对象比较 简单的调用方法 : <code>cmp(obj, x)</code>

## 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

在类的内部, 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, 不能在类的外部调用。在类的内部调用 `self.__private_methods` 实例不能访问私有变量

Python 不允许实例化的类访问私有数据，但你可以使用 `object._className__attrName`（对象名.\_类名\_\_私有属性名）访问属性，参考以下实例：

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

class Runoob:
    __site = "www.runoob.com"

runoob = Runoob()
print runoob._Runoob__site
```

执行以上代码，执行结果如下：

```
www.runoob.com
```

单下划线、双下划线、头尾双下划线说明：

- `__foo__`：定义的是特殊方法，一般是系统定义名字，类似 `__init__` 之类的。
- `_foo`：以单下划线开头的表示的是 `protected` 类型的变量，即保护类型只能允许其本身与子类进行访问，不能用于 `from module import *`
- `__foo`：双下划线的表示的是私有类型(private)的变量，只能是允许这个类本身进行访问了。