面向对象编程

1. Self

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，但是在调用这个方法的时候你不为这个参数赋值，Python 会提供这个值。这个特别的变量指对象本身，按照惯例它的名称是 `self`。
虽然你可以给这个参数任何名称，但是 强烈建议 你使用 `self` 这个名称——其他名称都是不赞成你使用的。使用一个标准的名称有很多优点——你的程序读者可以迅速识别它，如果使用 `self` 的话，还有些 IDE（集成开发环境）也可以帮助你。
给 C++/Java/C#程序员的注释 Python 中的 `self` 等价于 C++中的 `self` 指针和 Java、C#中的 `this` 参考。
你一定很奇怪 Python 如何给 `self` 赋值以及为何你不需要给它赋值。举一个例子会使此变得清晰。假如你有一个类称为 `MyClass` 和这个类的一个实例 `MyObject`。当你调用这个对象的方法 `MyObject.method(arg1, arg2)`的时候，这会由 Python 自动转为 `MyClass.method(MyObject, arg1, arg2)`——这就是 `self` 的原理了。
这也意味着如果你有一个不需要参数的方法，你还是得给这个方法定义一个 `self` 参数。

2. 类

一个尽可能简单的类如下面这个例子所示。

```
#!/usr/bin/python
# Filename: simplestclass.py

class Person:
    pass # An empty block

p = Person()
print p
```

输出

```
$ python simplestclass.py
<__main__.Person instance at 0xf6fcb18c>
```

它如何工作

我们使用 `class` 语句后跟类名，创建了一个新的类。这后面跟着一个缩进的语句块形成类体。在这个例子中，我们使用了一个空白块，它由 `pass` 语句表示。
接下来，我们使用类名后跟一对圆括号来创建一个对象/实例。（我们将在下面的章节中学习更多的如何创建实例的方法）。为了验证，我们简单地打印了这个变量的类型。它告诉我们我们已经在`__main__`模块中有了一个 `Person` 类的实例。
可以注意到存储对象的计算机内存地址也打印了出来。这个地址在你的计算机上会是另外一个值，因为 Python 可以在任何空位存储对象。

3. 对象的方法

我们已经讨论了类/对象可以拥有像函数一样的方法，这些方法与函数的区别只是一个额外的 `self` 变量。现在我们来学习一个例子。

```
#!/usr/bin/python
# Filename: method.py

class Person:
    def sayHi(self):
        print 'Hello, how are you?'

p = Person()
p.sayHi()

# This short example can also be written as Person().sayHi()
```

输出

```
$ python method.py
Hello, how are you?
```

它如何工作

这里我们看到了 `self` 的用法。注意 `sayHi` 方法没有任何参数，但仍然在函数定义时有 `self`。

4. __init__方法

在 Python 的类中有很多方法的名字有特殊的重要意义。现在我们将学习 `__init__`方法的意义。

`__init__`方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的 初始化 。注意，这个名称的开始和结尾都是双下划线。

```
#!/usr/bin/python
# Filename: class_init.py

class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print 'Hello, my name is', self.name

p = Person('Swaroop')
p.sayHi()

# This short example can also be written as Person('Swaroop').sayHi()
```

输出

```
$ python class_init.py
Hello, my name is Swaroop
```

它如何工作

这里，我们把`__init__`方法定义为取一个参数`name`（以及普通的参数`self`）。在这个`__init__`里，我们只是创建一个新的域，也称为`name`。注意它们是两个不同的变量，尽管它们有相同的名字。点号使我们能够区分它们。
最重要的是，我们没有专门调用`__init__`方法，只是在创建一个类的新实例的时候，把参数包括在圆括号内跟在类名后面，从而传递给`__init__`方法。这是这种方法的重要之处。
现在，我们能够在我们的方法中使用`self.name`域。这在`sayHi`方法中得到了验证。
给 C++/Java/C#程序员的注释 `__init__`方法类似于 C++、C#和 Java 中的 constructor 。

5. 类与对象的方法

我们已经讨论了类与对象的功能部分，现在我们来看一下它的数据部分。事实上，它们只是与类和对象的**名称空间** 绑定 的普通变量，即这些名称只在这些类与对象的前提下有效。

有两种类型的 域 ——类的变量和对象的变量，它们根据是类还是对象 拥有这个变量而区分。

类的变量 由一个类的所有对象（实例）共享使用。只有一个类变量的拷贝，所以当某个对象对类的变量做了改动的时候，这个改动会反映到所有其他的实例上。

对象的变量 由类的每个对象/实例拥有。因此每个对象有自己对这个域的一份拷贝，即它们不是共享的，在同一个类的不同实例中，虽然对象的变量有相同的名称，但是是互不相关的。通过一个例子会使这个易于理解。

```
#!/usr/bin/python
# Filename: objvar.py

class Person:
    '''Represents a person.'''
    population = 0

    def __init__(self, name):
        '''Initializes the person's data.'''
        self.name = name
        print '(Initializing %s)' % self.name

        # When this person is created, he/she
        # adds to the population
        Person.population += 1

    def __del__(self):
        '''I am dying.'''
        print '%s says bye.' % self.name

        Person.population -= 1

        if Person.population == 0:
            print 'I am the last one.'
        else:
            print 'There are still %d people left.' % Person.population

    def sayHi(self):
        '''Greeting by the person.

        Really, that's all it does.'''
        print 'Hi, my name is %s.' % self.name

    def howMany(self):
        '''Prints the current population.'''
        if Person.population == 1:
            print 'I am the only person here.'
```

```
        else:
            print 'We have %d persons here.' % Person.population

swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()

kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()

swaroop.sayHi()
swaroop.howMany()
```

输出

```
$ python objvar.py
(Initializing Swaroop)
Hi, my name is Swaroop.
I am the only person here.
(Initializing Abdul Kalam)
Hi, my name is Abdul Kalam.
We have 2 persons here.
Hi, my name is Swaroop.
We have 2 persons here.
Abdul Kalam says bye.
There are still 1 people left.
Swaroop says bye.
I am the last one.
```

它如何工作

这是一个很长的例子，但是它有助于说明类与对象的变量的本质。这里，`population` 属于 `Person` 类，因此是一个类的变量。`name` 变量属于对象（它使用 `self` 赋值）因此是对象的变量。

观察可以发现`__init__`方法用一个名字来初始化 `Person` 实例。在这个方法中，我们让 `population` 增加 `1`，这是因为我们增加了一个人。同样可以发现，`self.name` 的值根据每个对象指定，这表明了它作为对象的变量的本质。

记住，你只能使用 `self` 变量来参考同一个对象的变量和方法。这被称为 属性参考 。

在这个程序中，我们还看到 **docstring** 对于类和方法同样有用。我们可以在运行时使用 `Person.__doc__`和 `Person.sayHi.__doc__`来分别访问类与方法的文档字符串。

就如同`__init__`方法一样，还有一个特殊的方法`__del__`，它在对象消逝的时候被调用。对象消逝即对象不再被使用，它所占用的内存将返回给系统作它用。在这个方法里面，我们只是简单地把 `Person.population` 减 `1`。

当对象不再被使用时，`__del__`方法运行，但是很难保证这个方法究竟在 什么时候 运行。如果你想要指明它的运行，你就得使用`del`语句，就如同我们在以前的例子中使用的那样。
给 C++/Java/C#程序员的注释 Python 中所有的类成员（包括数据成员）都是 公共的 ，所有的方法都是 有效的 。 只有一个例外：如果你使用的数据成员名称以 双下划线前缀 比如`__privatevar`，Python 的名称管理体系会有效地把它作为私有变量。 这样就有一个惯例，如果某个变量只想在类或对象中使用，就应该以单下划线前缀。而其他的名称都将作为公共的，可以被其他类/对象使用。记住这只是一个惯例，并不是 Python 所要求的（与双下划线前缀不同）。 同样，注意`__del__`方法与 destructor 的概念类似。

6. 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。

假设你想要写一个程序来记录学校之中的教师和学生情况。他们有一些共同属性，比如姓名、年龄和地址。他们也有专有的属性，比如教师的薪水、课程和假期，学生的成绩和学费。

你可以为教师和学生建立两个独立的类来处理它们，但是这样做的话，如果要增加一个新的共有属性，就意味着要在这两个独立的类中都增加这个属性。这很快就会显得不实用。

一个比较好的方法是创建一个共同的类称为`SchoolMember`然后让教师和学生的类 继承 这个共同的类。即它们都是这个类型（类）的子类型，然后我们再为这些子类型添加专有的属性。
使用这种方法有很多优点。如果我们增加/改变了`SchoolMember`中的任何功能，它会自动地反映到子类型之中。例如，你要为教师和学生都增加一个新的身份证域，那么你只需简单地把它加到`SchoolMember`类中。然而，在一个子类型之中做的改动不会影响到别的子类型。另外一个优点是你可以把教师和学生对象都作为`SchoolMember`对象来使用，这在某些场合特别有用，比如统计学校成员的人数。一个子类型在任何需要父类型的场合可以被替换成父类型，即对象可以被视作是父类的实例，这种现象被称为**多态现象**。
另外，我们会发现在 重用 父类的代码的时候，我们无需在不同的类中重复它。而如果我们使用独立的类的话，我们就不得不这么做了。

在上述的场合中，`SchoolMember`类被称为 基本类 或 超类 。而`Teacher`和`Student`类被称为 导出类 或 子类 。
现在，我们将学习一个例子程序。

```
#!/usr/bin/python
# Filename: inherit.py
```

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember: %s)' % self.name

    def tell(self):
        '''Tell my details.'''
        print 'Name:"%s" Age:"%s"' % (self.name, self.age),

class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher: %s)' % self.name

    def tell(self):
        SchoolMember.tell(self)
        print 'Salary: "%d"' % self.salary

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student: %s)' % self.name

    def tell(self):
        SchoolMember.tell(self)
        print 'Marks: "%d"' % self.marks

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 22, 75)

print # prints a blank line

members = [t, s]
for member in members:
    member.tell() # works for both Teachers and Students
```

输出

```
$ python inherit.py
```

```
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"22" Marks: "75"
```

它如何工作

为了使用继承，我们把基本类的名称作为一个元组跟在定义类时的类名称之后。然后，我们注意到基本类的`__init__`方法专门使用`self`变量调用，这样我们就可以初始化对象的基本类部分。这一点十分重要——Python 不会自动调用基本类的 constructor，你得亲自专门调用它。

我们还观察到我们在方法调用之前加上类名称前缀，然后把`self`变量及其他参数传递给它。

注意，在我们使用`SchoolMember`类的`tell`方法的时候，我们把`Teacher`和`Student`的实例仅仅作为`SchoolMember`的实例。

另外，在这个例子中，我们调用了子类型的`tell`方法，而不是`SchoolMember`类的`tell`方法。可以这样来理解，Python 总是首先查找对应类型的方法，在这个例子中就是如此。如果它不能在导出类中找到对应的方法，它才开始到基本类中逐个查找。基本类是在类定义的时候，在元组之中指明的。

一个术语的注释——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 。