

## # 第7章 python类与对象

在Python中，所有数据类型都可以视为对象。面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。下面是面向对象编程(OOP)的一个名称介绍：

- 类：用于定义表示用户定义对象的一组属性的原型。属性是通过点符号访问的数据成员（类变量和实例变量）和方法。
- 类变量：由类的所有实例共享的变量。类变量在类中定义，但在类的任何方法之外。类变量不像实例变量那样频繁使用。
- 数据成员：保存和类及其对象相关联的数据的类变量或实例变量。
- 函数重载：将多个行为分配给特定函数。执行的操作因涉及的对象或参数的类型而异。
- 实例变量：在方法中定义并仅属于类的当前实例的变量。
- 继承：将类的特征传递给从其派生的其他类。
- 实例：某个类的单个对象。
- 实例化：创建类的实例。
- 方法：在类定义中定义的一种特殊类型的函数。
- 对象：由其类定义的数据结构的唯一实例。对象包括数据成员（类变量和实例变量）和方法。
- 运算符重载：将多个函数分配给特定的运算符。

# 第7章 python类与对象

在Python中，所有数据类型都可以视为对象。面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。下面是面向对象编程(OOP)的一个名称介绍：

- 类：用于定义表示用户定义对象的一组属性的原型。属性是通过点符号访问的数据成员（类变量和实例变量）和方法。
- 类变量：由类的所有实例共享的变量。类变量在类中定义，但在类的任何方法之外。类变量不像实例变量那样频繁使用。
- 数据成员：保存和类及其对象相关联的数据的类变量或实例变量。
- 函数重载：将多个行为分配给特定函数。执行的操作因涉及的对象或参数的类型而异。
- 实例变量：在方法中定义并仅属于类的当前实例的变量。
- 继承：将类的特征传递给从其派生的其他类。
- 实例：某个类的单个对象。
- 实例化：创建类的实例。
- 方法：在类定义中定义的一种特殊类型的函数。
- 对象：由其类定义的数据结构的唯一实例。对象包括数据成员（类变量和实例变量）和方法。
- 运算符重载：将多个函数分配给特定的运算符。

本章将从以下几方面来介绍：

### 一、类和实例

- 创建类和实例对象
- 数据封装

## 二、访问限制

## 三、类继承

- 继承
- 重载方法
- 多重继承
- 四、获取对象信息
- type()
- isinstance()

## 1. 类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

### 1.1 创建类和实例对象

这里以Student类为例，在Python中，通过class关键字来定义类：

```
[1] class Student(object):  
    pass
```

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的。通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的

```
[2] bart = Student() # 创建Student类的实例  
print(bart)  
print(Student)
```

```
<__main__.Student object at 0x00000182D5B8FCF8>  
<class '__main__.Student'>
```

可以看到，变量**bart**指向的就是一个**Student**的实例，后面的**0x000001FB57EBDCF8**是内存地址，每个object的地址都不一样，而**Student**本身则是一个类

可以自由地给一个实例变量绑定属性

```
[3] # 比如：给实例bart绑定一个name属性：  
    bart.name = 'Bart Simpson'  
    bart.name  
  
    'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的**\_\_init\_\_**方法，在创建实例的时候，就把**name**，**score**等属性绑上去：

```
[4] class Student(object):  
  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score
```

注意到**\_\_init\_\_**方法的第一个参数永远是**self**，表示创建的实例本身，因此，在**\_\_init\_\_**方法内部，就可以把各种属性绑定到**self**，因为**self**就指向创建的实例本身。有了**\_\_init\_\_**方法，在创建实例的时候，就不能传入空的参数了，必须传入与**\_\_init\_\_**方法匹配的参数，但**self**不需要传，Python解释器自己会把实例变量传进去

```
[5] bart = Student('Bart Simpson',60)  
    print(bart.name)  
    print(bart.score)
```

```
Bart Simpson  
60
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量**self**，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区

别，所以，仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

## 1.2 数据封装

面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例就拥有各自的name和score这些数据。访问这些数据，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

```
[6] class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：

```
[7] bart = Student('Kobe', 89) # 创建实例
    bart.print_score() # 实例变量调用类方法
```

Kobe: 89

## 2. 访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、score属性，如下所示：

```
[8] bart = Student('Bart', 88)
    print(bart.score)
    bart.score=90
    print(bart.score)
```

如果要想让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们Student类改为以下形式

```
[13] class Student(object):

    def __init__(self, name, score):
        self.__name = name    # 变为私有变量
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))
```

修改完后，对于外部代码来说，已经无法从外部访问实例变量.\_\_name和实例变量.\_\_score了。这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

要想在外部获取name和score，可以给类添加get\_name和get\_score方法；同理，想在外部修改，则可添加set方法，如下所示：

```
[15] class Student(object):

    def __init__(self, name, score):
        self.__name = name    # 变为私有变量
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score

    def set_name(self, name):
        self.__name = name

    def set_score(self, score):
        self.__score = score
```

值得注意的是，在Python中，变量名为\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是私有变量。

最后再注意下面这种错误的写法：

```
[17]  bart = Student('Bart',88)
      print(bart.get_name())
      bart.__name = 'New Name' # 设置__name变量
      print(bart.__name)
```

```
Bart
New Name
```

表面上看，外部代码“成功”地设置了\_\_name变量，但实际上这个\_\_name变量和class内部的\_\_name变量不是一个变量！内部的\_\_name变量已经被Python解释器保护起来，而外部代码给bart新增了一个\_\_name变量。调用get\_name()方法即可知道：

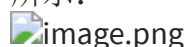
```
[18]  bart.get_name()
```

```
'Bart'
```

### 3. 类继承

继承用于指定一个类将从其父类获取其大部分或全部功能。它是面向对象编程的一个特征。这是一个非常强大的功能，方便用户对现有类进行几个或多个修改来创建一个新的类。新类称为子类或派生类，从其继承属性的主类称为基类或父类。

子类或派生类继承父类的功能，向其添加新功能。它有助于代码的可重用性。如下图所示：

image.png

#### 3.1 继承

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类。

比如，我们已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

```
[19] class Animal(object):  
      def run(self):  
          print('I am a animal')
```

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

```
[20] # 对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和  
      Dog类似。  
      class Dog(Animal):  
          pass  
  
      class Cat(Animal):  
          pass
```

对于子类来说，它获得了父类的所有功能。由于Animal实现了run()方法，因此，Dog和Cat作为它的子，就自动拥有了run()方法：

```
[21] dog = Dog()  
      dog.run()  
  
      cat = Cat()  
      cat.run()
```

```
I am a animal  
I am a animal
```

除此之外，还可以对子类增加一些方法，比如：

```
[43] class Dog(Animal):  
      def eat(self):    # 添加新方法  
          print('I like eating bones')
```

```
[44] dog = Dog()  
      dog.run()  
      dog.eat()
```

```
I am a animal
I like eating bones
```

### 3.2 重载方法

可以随时重载父类的方法。


```
[45] class Animal(object):
      def run(self):
          print('I am a animal')

      class Dog(object):
          def run(self): # 重载方法
              print('I am a dog')

      dog = Dog()
      dog.run()
```

I am a dog

下表列出了可以在自己的类中覆盖的一些通用方法:

image.png

### 3.3 多重继承

在多重继承中，所有基类的特征都被继承到派生类中。多重继承的语法类似于单继承。

```
[46] # 多重继承的示例
      class Base1:
          pass

      class Base2:
          pass

      class MultiDerived(Base1, Base2): # MultiDerived继承Base1和Base2
          pass
```

image.png



```
[51] # 多重继承举例
class Animal(object):
    def run(self):
        print('I am a animal')

class Dog(object):
    def eat(self):
        print('I like eat bones')

class Dog_son(Animal,Dog): # Dog_son类同时继承Animal类和Dog
    # Dog_son拥有Animal和Dog中所有的方法
    def run(self):
        print('666666666') # 而且多重继承中，同样可以重写父类的方法
    def p(self):
        print('hello')

dog_son = Dog_son()
dog_son.p()
dog_son.eat()
dog_son.run()
```

```
hello
I like eat bones
666666666
```


### 3.4 多级继承

另一方面，我们也可以继承一个派生类的形式。这被称为多级继承。它可以在Python中有任何的深度(层级)。在多级继承中，基类和派生类的特性被继承到新的派生类中。

```
[47] # 多级继承示例
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

image.png

```
[54] # 多级继承举例
```

```

class Animal(object):
    def run(self):
        print('I am a animal')

class Dog(Animal):
    def eat(self):
        print('I like eat bones')

class Dog_son(Dog): # Dog_son继承Dog类,Dog类继承Animal类
    def run(self):
        print('666666666') # 多级继承中,同样可以重写父类的方法
    def p(self):
        print('hello')
# 与多重继承相似, dog_son同样拥有Dog和Animal类中的方法
dog_son = Dog_son()
dog_son.p()
dog_son.eat()
dog_son.run()

hello
I like eat bones
666666666

```

## 4. 获取对象信息

当我们拿到一个对象的引用时,可以通过以下方法得知对象的类型及包含的方法

### 4.1 type()

```

[27] # 基本类型都可以用type()判断
print(type(123))
print(type('str'))
print(type(None))

```

```

<class 'int'>
<class 'str'>
<class 'NoneType'>

```

```

[28] # 如果一个变量指向函数或者类,也可以用type()判断
print(type(abs))
print(type(dog))

```

```
<class 'builtin_function_or_method'>
<class '__main__.Dog'>
```

## 4.2 isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数

isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

```
[29] # 引用上面的dog类继承Animal类
      print(isinstance(dog,Dog))
      print(isinstance(dog,Animal))
```

```
True
True
```

```
[30] # 能用type()判断的基本类型也可以用isinstance()判断
      print(isinstance('a',str))
      print(isinstance(123,str))
```

```
True
False
```

```
[31] # 还可以判断一个变量是否是某些类型中的一种
      print(isinstance([1,2,3],(list,tuple)))
      print(isinstance(1,2,3),(list,tuple)))
```

```
True
True
```

## 4.3 dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list

```
[34] # 比如，获得一个str对象的所有属性和方法
      dir('ABC')[0:5] # 值显示前5个结果
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__']
```

## 5. 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。给实例绑定属性的方法是通过实例变量，或者通过self变量：

```
[55] class Animal():
      def __init__(self,name,weight):
          self.name = name
          self.weight = weight
dog = Animal('hasky',90)
dog.height = 20 # 这里的height为实例属性
print(dog.height)
```

```
20
```

在class中定义属性，这种属性是类属性，如下面的tt,归Animal类所有：

```
[58] class Animal():
      tt = 'ss'
      def __init__(self,name,weight):
          self.name = name
          self.weight = weight
print(Animal.tt) # 类属性
dog = Animal('hasky',90)
print(dog.tt)
dog.tt = 'aa'
print(dog.tt)
print(Animal.tt)
```

```
ss
ss
aa
ss
```

可以看到，我们创建了Animal类的一个类属性tt，所有实例都可以访问该属性，而在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例

属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

[ ]