CH7 再谈抽象

7.2 类

7.2.1 类到底是什么

本书前面反复提到了类，并将其用作类型的同义词。从很多方面来说，这正是类的定义——一种对象。每个对象都属于特定的类，并被称为该类的实例。

注：在Python中，类约定使用单数并将首字母大写，如Bird和Lark。

在较新的Python 2版本中，类型和类之间这种差别不那么明显。在Python 3中，已不再区分类和类型了。

7.2.2 创建自定义类

终于要创建自定义类了！下面是一个简单的示例：

```
__metaclass__ = type # 如果你使用的是Python 2，请包含这行代码
class Person:
    def set_name(self, name):
        self.name = name
    def get_name(self):
        return self.name
    def greet(self):
        print("Hello, world! I'm {}.".format(self.name))
```

显然，self很有用，甚至必不可少。如果没有它，所有的方法都无法访问对象本身——要操作的属性所属的对象。与以前一样，也可以从外部访问这些属性。

```
>>> foo.name
'Luke Skywalker'
>>> bar.name = 'Yoda'
>>> bar.greet()
Hello, world! I'm Yoda.
```

注：如果foo是一个Person实例，可将foo.greet()视为Person.greet(foo)的简写，但后者的多态性更低。

7.2.3 属性、函数和方法

实际上，方法和函数的区别表现在前一节提到的参数self上。方法（更准确地说是关联的方法）将其第一个参数关联到它所属的实例，因此无需提供这个参数。**无疑可以将属性关联到一个普通函数，但这样就没有特殊的self参数了。**

```
>>> class Class:
...     def method(self):
...     print('I have a self!')
...
>>> def function():
...     print("I don't...")
...
>>> instance = Class()
>>> instance.method() I have a self!
>>> instance.method = function
>>> instance.method() I don't...
```

注意: 有没有参数self并不取决于是否以刚才使用的方式（如instance.method）调用方法。

实际上，完全可以让另一个变量指向同一个方法。

```
>>> class Bird:
...     song = 'Squaawk!'
...     def sing(self):
...         print(self.song)
...
>>> bird = Bird()
>>> bird.sing()
Squaawk!
>>> birdsong = bird.sing
>>> birdsong()
Squaawk!
```

虽然最后一个方法调用看起来很像函数调用，但变量birdsong指向的是关联的方法`bird.sing`，这意味着它也能够访问参数self（即它也被关联到类的实例）。

7.2.4 再谈隐藏

默认情况下，可从外部访问对象的属性。

```
>>> c.name
'Sir Lancelot'
>>> c.name = 'Sir Gumby'
>>> c.get_name()
'Sir Gumby'
```

为避免这类问题，可将属性定义为私有。私有属性不能从对象外部访问，而只能通过存取器方法（如`get_name`和`set_name`）来访问。

Python没有为私有属性提供直接的支持，而是要求程序员知道在什么情况下从外部修改属性是安全的。

要让方法或属性成为私有的（不能从外部访问），只需让其名称以两个下划线打头即可。

```
class Secretive:
    def __inaccessible(self):
        print("Bet you can't see me ...")
    def accessible(self):
        print("The secret message is:")
        self.__inaccessible()
```

现在从外部不能访问__inaccessible，但在类中（如accessible中）依然可以使用它。

```
>>> s = Secretive()
>>> s.__inaccessible()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: Secretive instance has no attribute '__inaccessible'
>>> s.accessible()
The secret message is:
Bet you cant see me ...
```

如果你不希望名称被修改，又想发出不要从外部修改属性或方法的信号，可用一个下划线打头。这虽然只是一种约定，但也有些作用。例如，`from module import *`不会导入以一个下划线打头的名称。

7.2.5 类的命名空间

在class语句中定义的代码都是在一个特殊的命名空间（类的命名空间）内执行的，而类的所有成员都可访问这个命名空间。类定义其实就是要执行的代码段，并非所有的Python程序员都知道这一点，但知道这一点很有帮助。例如，在类定义中，并非只能包含def语句。

```
class MemberCounter:
    members = 0
    def init(self):
        MemberCounter.members += 1
>>> m1 = MemberCounter()
>>> m1.init()
>>> MemberCounter.members
1
>>> m2 = MemberCounter()
>>> m2.init()
>>> MemberCounter.members
2
```

注：这里把member定义在函数之外，访问时用的也是类的名字而不是实例的名字。

每个实例都可访问这个类作用域内的变量，就像方法一样。

```
>>> m1.members
2
>>> m2.members
2
```

如果你在一个实例中给属性members赋值，结果将如何呢？

```
>>> m1.members = 'Two'
>>> m1.members
'Two'
>>> m2.members
2
```

新值被写入m1的一个属性中，这个属性遮住了类级变量。这类似于第6章的旁注“遮盖的问题”所讨论的，函数中局部变量和全局变量之间的关系。

7.2.6 指定超类

要指定超类，可在class语句中的类名后加上超类名，并将其用圆括号括起。

```
class Filter:
    def init(self):
        self.blocked = []
    def filter(self, sequence):
        return [x for x in sequence if x not in self.blocked]

class SPAMFilter(Filter): # SPAMFilter是Filter的子类
    def init(self): # 重写超类Filter的方法init
        self.blocked = ['SPAM']
```

Filter是一个过滤序列的通用类。实际上，它不会过滤掉任何东西。

```
>>> f = Filter()
>>> f.init()
>>> f.filter([1, 2, 3])
[1, 2, 3]
```

Filter类的用途在于可用作其他类（如将'SPAM'从序列中过滤掉的SPAMFilter类）的基类（超类）。

```
>>> s = SPAMFilter()
>>> s.init()
>>> s.filter(['SPAM', 'SPAM', 'SPAM', 'SPAM', 'eggs', 'bacon', 'SPAM'])
['eggs', 'bacon']
```

请注意SPAMFilter类的定义中有两个要点。

- 以提供新定义的方式重写了Filter类中方法init的定义。
- 直接从Filter类继承了方法filter的定义，因此无需重新编写其定义。
  第二点说明了继承很有用的原因：可以创建大量不同的过滤器类，它们都从Filter类派生而来，并且都使用已编写好的方法filter。这就是懒惰的好处。

7.2.7 深入探讨继承

要确定一个类是否是另一个类的子类，可使用内置方法`issubclass`。

```
>>> issubclass(SPAMFilter, Filter)
True
>>> issubclass(Filter, SPAMFilter)
False
```

如果你有一个类，并想知道它的基类，可访问其特殊属性**bases**。

```
>>> SPAMFilter.__bases__
(<class __main__.Filter at 0x171e40>,)
>>> Filter.__bases__
(<class 'object'>,)
```

同样，要确定对象是否是特定类的实例，可使用`isinstance`。

```
>>> s = SPAMFilter()
>>> isinstance(s, SPAMFilter)
True
>>> isinstance(s, Filter)
True
>>> isinstance(s, str)
False
```

所有SPAMFilter对象都是Filter对象。从前一个示例可知，isinstance也可用于类型，如字符串类型（str）。

注意: 使用`isinstance`通常不是良好的做法，依赖多态在任何情况下都是更好的选择。一个重要的例外情况是使用抽象基类和模块abc时。
如果你要获悉对象属于哪个类，可使用属性**class**。对于新式类（无论是通过用`__metaclass__ = type`还是通过从object继承创建的）的实例，还可使用`type(s)`来获悉其所属的类

```
>>> s.__class__
<class __main__.SPAMFilter at 0x1707c0>
```

7.2.8 多个超类

你肯定注意到了一个有点奇怪的细节：复数形式的**bases**。前面说过，你可使用它来获悉类的基类，而基类可能有多个。为说明如何继承多个类，下面来创建几个类。

```
class Calculator:
    def calculate(self, expression):
        self.value = eval(expression)

class Talker:
    def talk(self):
        print('Hi, my value is', self.value)

class TalkingCalculator(Calculator, Talker):
    pass

>>> tc = TalkingCalculator()
>>> tc.calculate('1 + 2 * 3')
>>> tc.talk()
Hi, my value is 7
```

这被称为多重继承。然而，除非万不得已，否则应避免使用多重继承，因为在有些情况下，它可能带来意外的“并发症”。

使用多重继承时，有一点务必注意：如果多个超类以不同的方式实现了同一个方法（即有多个同名方法），必须在class语句中小心排列这些超类，因为**位于前面的类的方法将覆盖位于后面的类的方法**。

多个超类的超类相同时，查找特定方法或属性时访问超类的顺序称为**方法解析顺序（MRO）**，它使用的算法非常复杂。所幸其效果很好，你可能根本无需担心。

7.2.9 接口和内省

接口这一概念与多态相关。处理多态对象时，你只关心其接口（协议）——对外暴露的方法和属性。在Python中，不显式地指定对象必须包含哪些方法才能用作参数。

通常，你要求对象遵循特定的接口（即实现特定的方法），但如果需要，也可非常灵活地提
出要求：不是直接调用方法并期待一切顺利，而是检查所需的方法是否存在；如果不存在，就改
弦易辙。

```
# 使用hasattr来检查所需的方法是否存在。
>>> hasattr(tc, 'talk')
True
>>> hasattr(tc, 'fnord')
False
```

在上述代码中，你发现tc（本章前面介绍的TalkingCalculator类的实例）包含属性talk（指向一个方法），但没有属性fnord。如果你愿意，还可以检查属性talk是否是可调用的。

```
>>> callable(getattr(tc, 'talk', None))
True
>>> callable(getattr(tc, 'fnord', None))
False
```

请注意，这里没有在if语句中使用hasattr并直接访问属性，而是使用了getattr（它让我能够指定属性不存在时使用的默认值，这里为None），然后对返回的对象调用callable。

注意: setattr与getattr功能相反，可用于设置对象的属性：

`Python >>> setattr(tc, 'name', 'Mr. Gumby') >>> tc.name 'Mr. Gumby'`

要查看对象中存储的所有值，可检查其**dict**属性。如果要确定对象是由什么组成的，应研究模块inspect。这个模块主要供高级用户创建对象浏览器（让用户能够以图形方式浏览Python对象的程序）以及其他需要这种功能的类似程序。

7.2.10 抽象基类

一般而言，**抽象类是不能（至少是不应该）实例化的类**，其职责是定义子类应实现的一组抽象方法。

```
from abc import ABC, abstractmethod

class Talker(ABC):
    @abstractmethod
    def talk(self):
        pass
```

形如@this的东西被称为装饰器。这里的要点是你使用@abstractmethod来将方法标记为抽象的——在子类中必须实现的方法。

假设像下面这样从它派生出一个子类：

```
class Knigget(Talker):
pass
```

由于没有重写方法talk，因此这个类也是抽象的，不能实例化。然而，你可重新编写这个类，使其实现要求的方法。

```
class Knigget(Talker):
    def talk(self):
        print("Ni!")
```

这是抽象基类的主要用途，而且只有在这种情形下使用isinstance才是妥当的：如果先检查给定的实例确实是Talker对象，就能相信这个实例在需要的情况下有方法talk。

```
>>> k = Knigget()
>>> isinstance(k, Talker)
True
>>> k.talk()
Ni!
```

鸭子类型的精神:因此，只要实现了方法talk，即便不是Talker的子类，依然能够通过类型检查。

```
class Herring:
    def talk(self):
        print("Blub.")
```

```
>>> h = Herring()
>>> isinstance(h, Talker)
False
```

为解决这个问题，你可将Herring注册为Talker（而不从Herring和Talker派生出子类），这样所有的Herring对象都将被视为Talker对象。

```
>>> Talker.register(Herring)
<class '__main__.Herring'>
>>> isinstance(h, Talker)
True
>>> issubclass(Herring, Talker)
True
```

这种做法存在一个缺点，就是直接从抽象类派生提供的保障没有了。

```
>>> class Clam:
...     pass
...
>>> Talker.register(Clam)
<class '__main__.Clam'>
>>> issubclass(Clam, Talker)
True
>>> c = Clam()
>>> isinstance(c, Talker)
True
>>> c.talk()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Clam' object has no attribute 'talk'
```

换而言之，应将isinstance返回True视为一种意图表达。在这里，Clam有成为Talker的意图。本着鸭子类型的精神，我们相信它能承担Talker的职责，但可悲的是它失败了。

7.4 小结

本章不仅介绍了有关Python语言的知识，还介绍了多个你可能一点都不熟悉的概念。下面来
总结一下。

- **对象**：对象由属性和方法组成。属性不过是属于对象的变量，而方法是存储在属性中的函数。相比于其他函数，（关联的）方法有一个不同之处，那就是它总是将其所属的对象作为第一个参数，而这个参数通常被命名为self。
- **类**：类表示一组（或一类）对象，而每个对象都属于特定的类。类的主要任务是定义其实例将包含的方法。
- **多态**：多态指的是能够同样地对待不同类型和类的对象，即无需知道对象属于哪个类就可调用其方法。
- **封装**：对象可能隐藏（封装）其内部状态。在有些语言中，这意味着对象的状态（属性）只能通过其方法来访问。在Python中，所有的属性都是公有的，但直接访问对象的状态时程序员应谨慎行事，因为这可能在不经意间导致状态不一致。
- **继承**：一个类可以是一个或多个类的子类，在这种情况下，子类将继承超类的所有方法。你可指定多个超类，通过这样做可组合正交（独立且不相关）的功能。为此，一种常见的做法是使用一个核心超类以及一个或多个混合超类。
- **接口和内省**：一般而言，你无需过于深入地研究对象，而只依赖于多态来调用所需的方法。然而，如果要确定对象包含哪些方法或属性，有一些函数可供你用来完成这种工作。
- **抽象基类**：使用模块abc可创建抽象基类。抽象基类用于指定子类必须提供哪些功能，却不实现这些功能。
- **面向对象设计**：关于该如何进行面向对象设计以及是否该采用面向对象设计，有很多不同的观点。无论你持什么样的观点，都必须深入理解问题，进而创建出易于理解的设计。

7.4.1 本章介绍的新函数

函 数	描 述
:	:
callable(object)	判断对象是否是可调用的（如是否是函数或方法）
getattr(object,name[,default])	获取属性的值，还可提供默认值
hasattr(object, name)	确定对象是否有指定的属性
isinstance(object, class)	确定对象是否是指定类的实例

issubclass(A, B)	确定A是否是B的子类
random.choice(sequence)	从一个非空序列中随机地选择一个元素
setattr(object, name, value)	将对象的指定属性设置为指定的值
type(object)	返回对象的类型