第 9 章　类

面向对象编程 是最有效的软件编写方法之一。在面向对象编程中，你编写表示现实世界中的事物和情景的类，并基于这些类来创建对象。编写类时，你定义一大类对象都有的通用行为。基于类创建**对象** 时，每个对象都自动具备这种通用行为，然后可根据需要赋予每个对象独特的个性。使用面向对象编程可模拟现实情景，其逼真程度达到了令你惊讶的地步。

根据类来创建对象被称为**实例化** ，这让你能够使用类的实例。在本章中，你将编写一些类并创建其实例。你将指定可在实例中存储什么信息，定义可对这些实例执行哪些操作。你还将编写一些类来扩展既有类的功能，让相似的类能够高效地共享代码。你将把自己编写的类存储在模块中，并在自己的程序文件中导入其他程序员编写的类。

理解面向对象编程有助于你像程序员那样看世界，还可以帮助你真正明白自己编写的代码：不仅是各行代码的作用，还有代码背后更宏大的概念。了解类背后的概念可培养逻辑思维，让你能够通过编写程序来解决遇到的几乎任何问题。

随着面临的挑战日益严峻，类还能让你以及与你合作的其他程序员的生活更轻松。如果你与其他程序员基于同样的逻辑来编写代码，你们就能明白对方所做的工作；你编写的程序将能被众多合作者所理解，每个人都能事半功倍。

9.1　创建和使用类

使用类几乎可以模拟任何东西。下面来编写一个表示小狗的简单类`Dog` ——它表示的不是特定的小狗，而是任何小狗。对于大多数宠物狗，我们都知道些什么呢？它们都有名字和年龄；我们还知道，大多数小狗还会蹲下和打滚。由于大多数小狗都具备上述两项信息（名字和年龄）和两种行为（蹲下和打滚），我们的`Dog` 类将包含它们。这个类让Python知道如何创建表示小狗的对象。编写这个类后，我们将使用它来创建表示特定小狗的实例。

9.1.1　创建Dog 类

根据`Dog` 类创建的每个实例都将存储名字和年龄。我们赋予了每条小狗蹲下（`sit()` ）和打滚（`roll_over()` ）的能力：

dog.py

```
❶ class Dog():
❷     """一次模拟小狗的简单尝试"""

❸     def __init__(self, name, age):
          """初始化属性name和age"""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """模拟小狗被命令时蹲下"""
          print(self.name.title() + " is now sitting.")

      def roll_over(self):
          """模拟小狗被命令时打滚"""
          print(self.name.title() + " rolled over!")
```

这里需要注意的地方很多，但你也不用担心，本章充斥着这样的结构，你有大把的机会熟悉它。在❶处，我们定义了一个名为`Dog` 的类。根据约定，在Python中，首字母大写的名称指的是类。这个类定义中的括号是空的，因为我们要从空白创建这个类。在❷处，我们编写了一个文档字符串，对这个类的功能作了描述。

1. 方法`__init__()`

类中的函数称为**方法** ；你前面学到的有关函数的一切都适用于方法，就目前而言，唯一重要的差别是调用方法的方式。❸处的方法`__init__()` 是一个特殊的方法，每当你根据`Dog` 类创建新实例时，Python都会自动运行它。在这个方法的名称中，开头和末尾各有两个下划线，这是一种约定，旨在避免Python默认方法与普通方法发生名称冲突。

我们将方法`__init__()` 定义成了包含三个形参：`self` 、`name` 和`age` 。在这个方法的定义中，形参`self` 必不可少，还必须位于其他形参的前面。为何必须在方法定义中包含形参`self` 呢？因为Python调用这个`__init__()` 方法来创建`Dog` 实例时，将自动传入实参`self` 。每个与类相关联的方法调用都自动传递实参`self` ，它是一个指向实例本身的引用，让实例能够访问类中的属性和方法。我们创建`Dog` 实例时，Python将调用`Dog` 类的方法`__init__()` 。我们将通过实参向`Dog()` 传递名字和年龄；`self` 会自动传递，

因此我们不需要传递它。每当我们根据Dog 类创建实例时，都只需给最后两个形参（name 和age ）提供值。

❹处定义的两个变量都有前缀self 。以self 为前缀的变量都可供类中的所有方法使用，我们还可以通过类的任何实例来访问这些变量。self.name = name 获取存储在形参name 中的值，并将其存储到变量name 中，然后该变量被关联到当前创建的实例。self.age = age 的作用与此类似。像这样可通过实例访问的变量称为**属性** 。

Dog 类还定义了另外两个方法：sit() 和roll_over() （见❺）。由于这些方法不需要额外的信息，如名字或年龄，因此它们只有一个形参self 。我们后面将创建的实例能够访问这些方法，换句话说，它们都会蹲下和打滚。当前，sit() 和roll_over() 所做的有限，它们只是打印一条消息，指出小狗正蹲下或打滚。但可以扩展这些方法以模拟实际情况：如果这个类包含在一个计算机游戏中，这些方法将包含创建小狗蹲下和打滚动画效果的代码。如果这个类是用于控制机器狗的，这些方法将引导机器狗做出蹲下和打滚的动作。

2. 在Python 2.7中创建类

在Python 2.7中创建类时，需要做细微的修改——在括号内包含单词object ：

```
class ClassName(object):
    --snip--
```

这让Python 2.7类的行为更像Python 3类，从而简化了你的工作。

在Python 2.7中定义Dog 类时，代码类似于下面这样：

```
class Dog(object):
    --snip--
```

9.1.2 根据类创建实例

可将类视为有关如何创建实例的说明。Dog 类是一系列说明，让Python知道如何创建表示特定小狗的实例。

下面来创建一个表示特定小狗的实例：

```
  class Dog():
      --snip--

❶ my_dog = Dog('willie', 6)

❷ print("My dog's name is " + my_dog.name.title() + ".")
❸ print("My dog is " + str(my_dog.age) + " years old.")
```

这里使用的是前一个示例中编写的Dog 类。在❶处，我们让Python创建一条名字为'willie' 、年龄为6 的小狗。遇到这行代码时，Python使用实参'willie' 和6 调用Dog 类中的方法__init__() 。方法__init__() 创建一个表示特定小狗的示例，并使用我们提供的值来设置属性name 和age 。方法__init__() 并未显式地包含return 语句，但Python自动返回一个表示这条小狗的实例。我们将这个实例存储在变量my_dog 中。在这里，命名约定很有用：我们通常可以认为首字母大写的名称（如Dog ）指的是类，而小写的名称（如my_dog ）指的是根据类创建的实例。

1. 访问属性

要访问实例的属性，可使用句点表示法。在❷处，我们编写了如下代码来访问my_dog 的属性name 的值：

```
my_dog.name
```

句点表示法在Python中很常用，这种语法演示了Python如何获悉属性的值。在这里，Python先找到实例my_dog ，再查找与这个实例相关联的属性name 。在Dog 类中引用这个属性时，使用的是self.name 。在❸处，我们使用同样的方法来获取属性age 的值。在前面的第1条print 语句中，my_dog.name.title() 将my_dog 的属性name 的值'willie' 改为首字母大写的；在第2条print 语句中，str(my_dog.age) 将my_dog 的属性age 的值6 转换为字符串。

输出是有关my_dog 的摘要：

```
My dog's name is Willie.
My dog is 6 years old.
```

2. 调用方法

根据Dog 类创建实例后，就可以使用句点表示法来调用Dog 类中定义的任何方法。下面来让小狗蹲下和打滚：

```
class Dog():
    --snip--

my_dog = Dog('willie', 6)
my_dog.sit()
my_dog.roll_over()
```

要调用方法，可指定实例的名称（这里是my_dog ）和要调用的方法，并用句点分隔它们。遇到代码my_dog.sit() 时，Python在类Dog 中查找方法sit() 并运行其代码。Python以同样的方式解读代码my_dog.roll_over() 。

Willie按我们的命令做了：

```
Willie is now sitting.
Willie rolled over!
```

这种语法很有用。如果给属性和方法指定了合适的描述性名称，如name 、age 、sit() 和roll_over() ，即便是从未见过的代码块，我们也能够轻松地推断出它是做什么

的。

3. 创建多个实例

可按需求根据类创建任意数量的实例。下面再创建一个名为`your_dog`的实例：

```
class Dog():
    --snip--

my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")
my_dog.sit()

print("\nYour dog's name is " + your_dog.name.title() + ".")
print("Your dog is " + str(your_dog.age) + " years old.")
your_dog.sit()
```

在这个实例中，我们创建了两条小狗，它们分别名为Willie和Lucy。每条小狗都是一个独立的实例，有自己的一组属性，能够执行相同的操作：

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

就算我们给第二条小狗指定同样的名字和年龄，Python依然会根据`Dog`类创建另一个实例。你可按需求根据一个类创建任意数量的实例，条件是将每个实例都存储在不同的变量中，或占用列表或字典的不同位置。

动手试一试

9-1 餐馆：创建一个名为`Restaurant`的类，其方法`__init__()`设置两个属性：`restaurant_name`和`cuisine_type`。创建一个名为`describe_restaurant()`的方法和一个名为`open_restaurant()`的方法，其中前者打印前述两项信息，而后者打印一条消息，指出餐馆正在营业。

根据这个类创建一个名为`restaurant`的实例，分别打印其两个属性，再调用前述两个方法。

9-2 三家餐馆：根据你为完成练习9-1而编写的类创建三个实例，并对每个实例调用方法`describe_restaurant()`。

9-3 用户：创建一个名为`User`的类，其中包含属性`first_name`和`last_name`，还有用户简介通常会存储的其他几个属性。在类`User`中定义一个名为`describe_user()`的方法，它打印用户信息摘要；再定义一个名为`greet_user()`的方法，它向用户发出个性化的问候。

创建多个表示不同用户的实例，并对每个实例都调用上述两个方法。

9.2 使用类和实例

你可以使用类来模拟现实世界中的很多情景。类编写好后，你的大部分时间都将花在使用根据类创建的实例上。你需要执行的一个重要任务是修改实例的属性。你可以直接修改实例的属性，也可以编写方法以特定的方式进行修改。

9.2.1 `Car`类

下面来编写一个表示汽车的类，它存储了有关汽车的信息，还有一个汇总这些信息的方法：

car.py

```
  class Car():
      """一次模拟汽车的简单尝试"""

❶     def __init__(self, make, model, year):
          """初始化描述汽车的属性"""
          self.make = make
          self.model = model
          self.year = year

❷     def get_descriptive_name(self):
          """返回整洁的描述性信息"""
          long_name = str(self.year) + ' ' + self.make + ' ' + self.model
          return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2016)
  print(my_new_car.get_descriptive_name())
```

在❶处，我们定义了方法`__init__()`。与前面的`Dog`类中一样，这个方法的第一个形参为`self`；我们还在这个方法中包含了另外三个形参：`make`、`model`和`year`。方法`__init__()`接受这些形参的值，并将它们存储在根据这个类创建的实例的属性中。创建新的`Car`实例时，我们需要指定其制造商、型号和生产年份。

在❷处，我们定义了一个名为`get_descriptive_name()`的方法，它使用属性`year`、`make`和`model`创建一个对汽车进行描述的字符串，让我们无需分别打印每个属性的值。为在这个方法中访问属性的值，我们使用了`self.make`、`self.model`和`self.year`。在❸处，我们根据`Car`类创建了一个实例，并将其存储到变量`my_new_car`中。接下来，我们调用方法`get_descriptive_name()`，指出我们拥有的是一辆什么样的汽车：

```
2016 Audi A4
```

为让这个类更有趣，下面给它添加一个随时间变化的属性，它存储汽车的总里程。

9.2.2 给属性指定默认值

类中的每个属性都必须有初始值，哪怕这个值是0或空字符串。在有些情况下，如设置默认值时，在方法`__init__()`内指定这种初始值是可行的；如果你对某个属性这样做了，就无需包含为它提供初始值的形参。

下面来添加一个名为odometer_reading 的属性，其初始值总是为0。我们还添加了一个名为read_odometer() 的方法，用于读取汽车的里程表：

```
class Car():

    def __init__(self, make, model, year):
        """初始化描述汽车的属性"""
        self.make = make
        self.model = model
        self.year = year
❶       self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--

❷   def read_odometer(self):
        """打印一条指出汽车里程的消息"""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

现在，当Python调用方法__init__() 来创建新实例时，将像前一个示例一样以属性的方式存储制造商、型号和生产年份。接下来，Python将创建一个名为odometer_reading 的属性，并将其初始值设置为0（见❶）。在❷处，我们还定义了一个名为read_odometer() 的方法，它让你能够轻松地获悉汽车的里程。

一开始汽车的里程为0：

```
2016 Audi A4
This car has 0 miles on it.
```

出售时里程表读数为0的汽车并不多，因此我们需要一个修改该属性的值的途径。

9.2.3 修改属性的值

可以以三种不同的方式修改属性的值：直接通过实例进行修改；通过方法进行设置；通过方法进行递增（增加特定的值）。下面依次介绍这些方法。

1. 直接修改属性的值

要修改属性的值，最简单的方式是通过实例直接访问它。下面的代码直接将里程表读数设置为23：

```
class Car():
    --snip--

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

❶ my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

在❶处，我们使用句点表示法来直接访问并设置汽车的属性odometer_reading 。这行代码让Python在实例my_new_car 中找到属性odometer_reading ，并将该属性的值设置为23：

```
2016 Audi A4
This car has 23 miles on it.
```

有时候需要像这样直接访问属性，但其他时候需要编写对属性进行更新的方法。

2. 通过方法修改属性的值

如果有替你更新属性的方法，将大有裨益。这样，你就无需直接访问属性，而可将值传递给一个方法，由它在内部进行更新。

下面的示例演示了一个名为update_odometer() 的方法：

```
class Car():
    --snip--

❶   def update_odometer(self, mileage):
        """将里程表读数设置为指定的值"""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

❷ my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

对Car 类所做的唯一修改是在❶处添加了方法update_odometer() 。这个方法接受一个里程值，并将其存储到self.odometer_reading 中。在❷处，我们调用了update_odometer() ，并向它提供了实参23（该实参对应于方法定义中的形参mileage ）。它将里程表读数设置为23；而方法read_odometer() 打印该读数：

```
2016 Audi A4
This car has 23 miles on it.
```

可对方法update_odometer() 进行扩展，使其在修改里程表读数时做些额外的工作。下面来添加一些逻辑，禁止任何人将里程表读数往回调：

```
class Car():
    --snip--
```

```
    def update_odometer(self, mileage):
        """
        将里程表读数设置为指定的值
        禁止将里程表读数往回调
        """
❶       if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
❷           print("You can't roll back an odometer!")
```

现在，update_odometer() 在修改属性前检查指定的读数是否合理。如果新指定的里程（mileage ）大于或等于原来的里程（self.odometer_reading ），就将里程表读数改为新指定的里程（见❶）；否则就发出警告，指出不能将里程表往回拨（见❷）。

3. 通过方法对属性的值进行递增

有时候需要将属性值递增特定的量，而不是将其设置为全新的值。假设我们购买了一辆二手车，且从购买到登记期间增加了100英里的里程，下面的方法让我们能够传递这个增量，并相应地增加里程表读数：

```
  class Car():
      --snip--

      def update_odometer(self, mileage):
          --snip--

❶     def increment_odometer(self, miles):
          """将里程表读数增加指定的量"""
          self.odometer_reading += miles

❷ my_used_car = Car('subaru', 'outback', 2013)
  print(my_used_car.get_descriptive_name())

❸ my_used_car.update_odometer(23500)
  my_used_car.read_odometer()

❹ my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

在❶处，新增的方法increment_odometer() 接受一个单位为英里的数字，并将其加入到self.odometer_reading 中。在❷处，我们创建了一辆二手车——my_used_car 。在❸处，我们调用方法update_odometer() 并传入23500 ，将这辆二手车的里程表读数设置为23 500。在❹处，我们调用increment_odometer() 并传入100 ，以增加从购买到登记期间行驶的100英里：

```
2013 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

你可以轻松地修改这个方法，以禁止增量为负值，从而防止有人利用它来回拨里程表。

> **注意**　你可以使用类似于上面的方法来控制用户修改属性值（如里程表读数）的方式，但能够访问程序的人都可以通过直接访问属性来将里程表修改为任何值。要确保安全，除了进行类似于前面的基本检查外，还需特别注意细节。

动手试一试

9-4 就餐人数 ：在为完成练习9-1而编写的程序中，添加一个名为number_served 的属性，并将其默认值设置为0。根据这个类创建一个名为restaurant 的实例；打印有多少人在这家餐馆就餐过，然后修改这个值并再次打印它。

添加一个名为set_number_served() 的方法，它让你能够设置就餐人数。调用这个方法并向它传递一个值，然后再次打印这个值。

添加一个名为increment_number_served() 的方法，它让你能够将就餐人数递增。调用这个方法并向它传递一个这样的值：你认为这家餐馆每天可能接待的就餐人数。

9-5 尝试登录次数 ：在为完成练习9-3而编写的User 类中，添加一个名为login_attempts 的属性。编写一个名为increment_login_attempts() 的方法，它将属性login_attempts 的值加1。再编写一个名为reset_login_attempts() 的方法，它将属性login_attempts 的值重置为0。

根据User 类创建一个实例，再调用方法increment_login_attempts() 多次。打印属性login_attempts 的值，确认它被正确地递增；然后，调用方法reset_login_attempts() ，并再次打印属性login_attempts 的值，确认它被重置为0。

9.3　继承

编写类时，并非总是要从空白开始。如果你要编写的类是另一个现成类的特殊版本，可使用**继承** 。一个类**继承** 另一个类时，它将自动获得另一个类的所有属性和方法；原有的类称为**父类** ，而新类称为**子类** 。子类继承了其父类的所有属性和方法，同时还可以定义自己的属性和方法。

9.3.1　子类的方法__init__()

创建子类的实例时，Python首先需要完成的任务是给父类的所有属性赋值。为此，子类的方法__init__() 需要父类施以援手。

例如，下面来模拟电动汽车。电动汽车是一种特殊的汽车，因此我们可以在前面创建的Car 类的基础上创建新类ElectricCar ，这样我们就只需为电动汽车特有的属性和行为编写代码。

下面来创建一个简单的ElectricCar 类版本，它具备Car 类的所有功能：

electric_car.py

```
❶ class Car():
      """一次模拟汽车的简单尝试"""

      def __init__(self, make, model, year):
          self.make = make
          self.model = model
          self.year = year
          self.odometer_reading = 0

      def get_descriptive_name(self):
          long_name = str(self.year) + ' ' + self.make + ' ' + self.model
```

```
        return long_name.title()

    def read_odometer(self):
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

❷ class ElectricCar(Car):
    """电动汽车的独特之处"""

❸     def __init__(self, make, model, year):
        """初始化父类的属性"""
❹         super().__init__(make, model, year)


❺ my_tesla = ElectricCar('tesla', 'model s', 2016)
  print(my_tesla.get_descriptive_name())
```

首先是Car 类的代码（见❶）。创建子类时，父类必须包含在当前文件中，且位于子类前面。在❷处，我们定义了子类ElectricCar 。定义子类时，必须在括号内指定父类的名称。方法__init__() 接受创建Car 实例所需的信息（见❸）。

❹处的super() 是一个特殊函数，帮助Python将父类和子类关联起来。这行代码让Python调用ElectricCar 的父类的方法__init__() ，让ElectricCar 实例包含父类的所有属性。父类也称为**超类** （superclass），名称super因此而得名。

为测试继承是否能够正确地发挥作用，我们尝试创建一辆电动汽车，但提供的信息与创建普通汽车时相同。在❺处，我们创建ElectricCar 类的一个实例，并将其存储在变量my_tesla 中。这行代码调用ElectricCar 类中定义的方法__init__() ，后者让Python调用父类Car 中定义的方法__init__() 。我们提供了实参'tesla' 、'model s' 和2016 。

除方法__init__() 外，电动汽车没有其他特有的属性和方法。当前，我们只想确认电动汽车具备普通汽车的行为：

```
2016 Tesla Model S
```

ElectricCar 实例的行为与Car 实例一样，现在可以开始定义电动汽车特有的属性和方法了。

9.3.2 Python 2.7中的继承

在Python 2.7中，继承语法稍有不同，ElectricCar 类的定义类似于下面这样：

```
class Car(object):
    def __init__(self, make, model, year):
        --snip--

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(make, model, year)
        --snip--
```

函数super() 需要两个实参：子类名和对象self 。为帮助Python将父类和子类关联起来，这些实参必不可少。另外，在Python 2.7中使用继承时，务必在定义父类时在括号内指定object 。

9.3.3 给子类定义属性和方法

让一个类继承另一个类后，可添加区分子类和父类所需的新属性和方法。

下面来添加一个电动汽车特有的属性（电瓶），以及一个描述该属性的方法。我们将存储电瓶容量，并编写一个打印电瓶描述的方法：

```
  class Car():
      --snip--

  class ElectricCar(Car):
      """Represent aspects of a car, specific to electric vehicles."""

      def __init__(self, make, model, year):
          """
          电动汽车的独特之处
          初始化父类的属性，再初始化电动汽车特有的属性
          """
          super().__init__(make, model, year)
❶         self.battery_size = 70

❷     def describe_battery(self):
          """打印一条描述电瓶容量的消息"""
          print("This car has a " + str(self.battery_size) + "-kWh battery.")

  my_tesla = ElectricCar('tesla', 'model s', 2016)
  print(my_tesla.get_descriptive_name())
  my_tesla.describe_battery()
```

在❶处，我们添加了新属性self.battery_size ，并设置其初始值（如70 ）。根据ElectricCar 类创建的所有实例都将包含这个属性，但所有Car 实例都不包含它。在❷处，我们还添加了一个名为describe_battery() 的方法，它打印有关电瓶的信息。我们调用这个方法时，将看到一条电动汽车特有的描述：

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

对于ElectricCar 类的特殊化程度没有任何限制。模拟电动汽车时，你可以根据所需的准确程度添加任意数量的属性和方法。如果一个属性或方法是任何汽车都有的，而不是电动汽车特有的，就应将其加入到Car 类而不是ElectricCar 类中。这样，使用Car 类的人将获得相应的功能，而ElectricCar 类只包含处理电动汽车特有属性和行为的代码。

9.3.4 重写父类的方法

对于父类的方法，只要它不符合子类模拟的实物的行为，都可对其进行重写。为此，可在子类中定义一个这样的方法，即它与要重写的父类方法同名。这样，Python将不会考虑这个父类方法，而只关注你在子类中定义的相应方法。

假设Car 类有一个名为fill_gas_tank() 的方法，它对全电动汽车来说毫无意义，因此你可能想重写它。下面演示了一种重写方式：

```
def ElectricCar(Car):
    --snip--

    def fill_gas_tank():
        """电动汽车没有油箱"""
        print("This car doesn't need a gas tank!")
```

现在，如果有人对电动汽车调用方法fill_gas_tank() ，Python将忽略Car 类中的方法fill_gas_tank() ，转而运行上述代码。使用继承时，可让子类保留从父类那里继承而来的精华，并剔除不需要的糟粕。

9.3.5 将实例用作属性

使用代码模拟实物时，你可能会发现自己给类添加的细节越来越多：属性和方法清单以及文件都越来越长。在这种情况下，可能需要将类的一部分作为一个独立的类提取出来。你可以将大型类拆分成多个协同工作的小类。

例如，不断给ElectricCar 类添加细节时，我们可能会发现其中包含很多专门针对汽车电瓶的属性和方法。在这种情况下，我们可将这些属性和方法提取出来，放到另一个名为Battery 的类中，并将一个Battery 实例用作ElectricCar 类的一个属性：

```
  class Car():
      --snip--

❶ class Battery():
      """一次模拟电动汽车电瓶的简单尝试"""

❷     def __init__(self, battery_size=70):
          """初始化电瓶的属性"""
          self.battery_size = battery_size

❸     def describe_battery(self):
          """打印一条描述电瓶容量的消息"""
          print("This car has a " + str(self.battery_size) + "-kWh battery.")


  class ElectricCar(Car):
      """电动汽车的独特之处"""

      def __init__(self, make, model, year):
          """
          初始化父类的属性，再初始化电动汽车特有的属性
          """
          super().__init__(make, model, year)
❹         self.battery = Battery()


  my_tesla = ElectricCar('tesla', 'model s', 2016)

  print(my_tesla.get_descriptive_name())
  my_tesla.battery.describe_battery()
```

在❶处，我们定义了一个名为Battery 的新类，它没有继承任何类。❷处的方法__init__() 除self 外，还有另一个形参battery_size 。这个形参是可选的：如果没有给它提供值，电瓶容量将被设置为70。方法describe_battery() 也移到了这个类中（见❸）。

在ElectricCar 类中，我们添加了一个名为self.battery 的属性（见❹）。这行代码让Python创建一个新的Battery 实例（由于没有指定尺寸，因此为默认值70 ），并将该实例存储在属性self.battery 中。每当方法__init__() 被调用时，都将执行该操作；因此现在每个ElectricCar 实例都包含一个自动创建的Battery 实例。

我们创建一辆电动汽车，并将其存储在变量my_tesla 中。要描述电瓶时，需要使用电动汽车的属性battery ：

```
my_tesla.battery.describe_battery()
```

这行代码让Python在实例my_tesla 中查找属性battery ，并对存储在该属性中的Battery 实例调用方法describe_battery() 。

输出与我们前面看到的相同：

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

这看似做了很多额外的工作，但现在我们想多详细地描述电瓶都可以，且不会导致ElectricCar 类混乱不堪。下面再给Battery 类添加一个方法，它根据电瓶容量报告汽车的续航里程：

```
  class Car():
      --snip--

  class Battery():
      --snip--

❶     def get_range(self):
          """打印一条消息，指出电瓶的续航里程"""
          if self.battery_size == 70:
              range = 240
          elif self.battery_size == 85:
              range = 270

          message = "This car can go approximately " + str(range)
```

```
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):
    --snip--

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

❶处新增的方法`get_range()`做了一些简单的分析：如果电瓶的容量为70kWh，它就将续航里程设置为240英里；如果容量为85kWh，就将续航里程设置为270英里，然后报告这个值。为使用这个方法，我们也通过汽车的属性`battery`来调用它（见❷）。

输出指出了汽车的续航里程（这取决于电瓶的容量）：

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

9.3.6 模拟实物

模拟较复杂的物件（如电动汽车）时，需要解决一些有趣的问题。续航里程是电瓶的属性还是汽车的属性呢？如果我们只需描述一辆汽车，那么将方法`get_range()`放在`Battery`类中也许是合适的；但如果要描述一家汽车制造商的整个产品线，也许应该将方法`get_range()`移到`ElectricCar`类中。在这种情况下，`get_range()`依然根据电瓶容量来确定续航里程，但报告的是一款汽车的续航里程。我们也可以这样做：将方法`get_range()`还留在`Battery`类中，但向它传递一个参数，如`car_model`；在这种情况下，方法`get_range()`将根据电瓶容量和汽车型号报告续航里程。

这让你进入了程序员的另一个境界：解决上述问题时，你从较高的逻辑层面（而不是语法层面）考虑；你考虑的不是Python，而是如何使用代码来表示实物。到达这种境界后，你经常会发现，现实世界的建模方法并没有对错之分。有些方法的效率更高，但要找出效率最高的表示法，需要经过一定的实践。只要代码像你希望的那样运行，就说明你做得很好！即便你发现自己不得不多次尝试使用不同的方法来重写类，也不必气馁；要编写出高效、准确的代码，都得经过这样的过程。

动手试一试

9-6 冰淇淋小店：冰淇淋小店是一种特殊的餐馆。编写一个名为`IceCreamStand`的类，让它继承你为完成练习9-1或练习9-4而编写的`Restaurant`类。这两个版本的`Restaurant`类都可以，挑选你更喜欢的那个即可。添加一个名为`flavors`的属性，用于存储一个由各种口味的冰淇淋组成的列表。编写一个显示这些冰淇淋的方法。创建一个`IceCreamStand`实例，并调用这个方法。

9-7 管理员：管理员是一种特殊的用户。编写一个名为`Admin`的类，让它继承你为完成练习9-3或练习9-5而编写的`User`类。添加一个名为`privileges`的属性，用于存储一个由字符串（如`"can add post"`、`"can delete post"`、`"can ban user"`等）组成的列表。编写一个名为`show_privileges()`的方法，它显示管理员的权限。创建一个`Admin`实例，并调用这个方法。

9-8 权限：编写一个名为`Privileges`的类，它只有一个属性——`privileges`，其中存储了练习`9-7`所说的字符串列表。将方法`show_privileges()`移到这个类中。在`Admin`类中，将一个`Privileges`实例用作其属性。创建一个`Admin`实例，并使用方法`show_privileges()`来显示其权限。

9-9 电瓶升级：在本节最后一个electric_car.py版本中，给`Battery`类添加一个名为`upgrade_battery()`的方法。这个方法检查电瓶容量，如果它不是85，就将它设置为85。创建一辆电瓶容量为默认值的电动汽车，调用方法`get_range()`，然后对电瓶进行升级，并再次调用`get_range()`。你会看到这辆汽车的续航里程增加了。

9.4 导入类

随着你不断地给类添加功能，文件可能变得很长，即便你妥善地使用了继承亦如此。为遵循Python的总体理念，应让文件尽可能整洁。为在这方面提供帮助，Python允许你将类存储在模块中，然后在主程序中导入所需的模块。

9.4.1 导入单个类

下面来创建一个只包含`Car`类的模块。这让我们面临一个微妙的命名问题：在本章中，已经有一个名为car.py的文件，但这个模块也应命名为car.py，因为它包含表示汽车的代码。我们将这样解决这个命名问题：将`Car`类存储在一个名为car.py的模块中，该模块将覆盖前面使用的文件car.py。从现在开始，使用该模块的程序都必须使用更具体的文件名，如my_car.py。下面是模块car.py，其中只包含`Car`类的代码：

car.py

```
❶ """一个可用于表示汽车的类"""

class Car():
    """一次模拟汽车的简单尝试"""

    def __init__(self, make, model, year):
        """初始化描述汽车的属性"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """返回整洁的描述性名称"""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """打印一条消息，指出汽车的里程"""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """
        将里程表读数设置为指定的值
        拒绝将里程表往回拨
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """将里程表读数增加指定的量"""
        self.odometer_reading += miles
```

在❶处，我们包含了一个模块级文档字符串，对该模块的内容做了简要的描述。你应为自己创建的每个模块都编写文档字符串。

下面来创建另一个文件——my_car.py，在其中导入Car 类并创建其实例：

my_car.py

```
❶ from car import Car

  my_new_car = Car('audi', 'a4', 2016)
  print(my_new_car.get_descriptive_name())

  my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

❶处的import 语句让Python打开模块car ，并导入其中的Car 类。这样我们就可以使用Car 类了，就像它是在这个文件中定义的一样。输出与我们在前面看到的一样：

```
2016 Audi A4
This car has 23 miles on it.
```

导入类是一种有效的编程方式。如果在这个程序中包含了整个Car 类，它该有多长呀！通过将这个类移到一个模块中，并导入该模块，你依然可以使用其所有功能，但主程序文件变得整洁而易于阅读了。这还能让你将大部分逻辑存储在独立的文件中；确定类像你希望的那样工作后，你就可以不管这些文件，而专注于主程序的高级逻辑了。

9.4.2 在一个模块中存储多个类

虽然同一个模块中的类之间应存在某种相关性，但可根据需要在一个模块中存储任意数量的类。类Battery 和ElectricCar 都可帮助模拟汽车，因此下面将它们都加入模块car.py中：

car.py

```
"""一组用于表示燃油汽车和电动汽车的类"""

class Car():
    --snip--

class Battery():
    """一次模拟电动汽车电瓶的简单尝试"""

    def __init__(self, battery_size=60):
        """初始化电瓶的属性"""
        self.battery_size = battery_size

    def describe_battery(self):
        """打印一条描述电瓶容量的消息"""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")

    def get_range(self):
        """打印一条描述电瓶续航里程的消息"""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):
    """模拟电动汽车的独特之处"""

    def __init__(self, make, model, year):
        """
        初始化父类的属性，再初始化电动汽车特有的属性
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

现在，可以新建一个名为my_electric_car.py的文件，导入ElectricCar 类，并创建一辆电动汽车了：

my_electric_car.py

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

输出与我们前面看到的相同，但大部分逻辑都隐藏在一个模块中：

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

9.4.3 从一个模块中导入多个类

可根据需要在程序文件中导入任意数量的类。如果我们要在同一个程序中创建普通汽车和电动汽车，就需要将Car 和ElectricCar 类都导入：

my_cars.py

```
❶ from car import Car, ElectricCar
```

```
❷ my_beetle = Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = ElectricCar('tesla', 'roadster', 2016)
  print(my_tesla.get_descriptive_name())
```

在❶处从一个模块中导入多个类时，用逗号分隔了各个类。导入必要的类后，就可根据需要创建每个类的任意数量的实例。

在这个示例中，我们在❷处创建了一辆大众甲壳虫普通汽车，并在❸处创建了一辆特斯拉Roadster电动汽车：

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

9.4.4 导入整个模块

你还可以导入整个模块，再使用句点表示法访问需要的类。这种导入方法很简单，代码也易于阅读。由于创建类实例的代码都包含模块名，因此不会与当前文件使用的任何名称发生冲突。

下面的代码导入整个`car`模块，并创建一辆普通汽车和一辆电动汽车：

my_cars.py

```
❶ import car

❷ my_beetle = car.Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = car.ElectricCar('tesla', 'roadster', 2016)
  print(my_tesla.get_descriptive_name())
```

在❶处，我们导入了整个`car`模块。接下来，我们使用语法 *`module_name.class_name`* 访问需要的类。像前面一样，我们在❷处创建了一辆大众甲壳虫汽车，并在❸处创建了一辆特斯拉Roadster汽车。

9.4.5 导入模块中的所有类

要导入模块中的每个类，可使用下面的语法：

```
from module_name import *
```

不推荐使用这种导入方式，其原因有二。首先，如果只要看一下文件开头的`import`语句，就能清楚地知道程序使用了哪些类，将大有裨益；但这种导入方式没有明确地指出你使用了模块中的哪些类。这种导入方式还可能引发名称方面的困惑。如果你不小心导入了一个与程序文件中其他东西同名的类，将引发难以诊断的错误。这里之所以介绍这种导入方式，是因为虽然不推荐使用这种方式，但你可能会在别人编写的代码中见到它。

需要从一个模块中导入很多类时，最好导入整个模块，并使用 *`module_name.class_name`* 语法来访问类。这样做时，虽然文件开头并没有列出用到的所有类，但你清楚地知道在程序的哪些地方使用了导入的模块；你还避免了导入模块中的每个类可能引发的名称冲突。

9.4.6 在一个模块中导入另一个模块

有时候，需要将类分散到多个模块中，以免模块太大，或在同一个模块中存储不相关的类。将类存储在多个模块中时，你可能会发现一个模块中的类依赖于另一个模块中的类。在这种情况下，可在前一个模块中导入必要的类。

例如，下面将`Car`类存储在一个模块中，并将`ElectricCar`和`Battery`类存储在另一个模块中。我们将第二个模块命名为`electric_car.py`（这将覆盖前面创建的文件electric_car.py），并将`Battery`和`ElectricCar`类复制到这个模块中：

electric_car.py

```
  """一组可用于表示电动汽车的类"""

❶ from car import Car

  class Battery():
      --snip--

  class ElectricCar(Car):
      --snip--
```

`ElectricCar`类需要访问其父类`Car`，因此在❶处，我们直接将`Car`类导入该模块中。如果我们忘记了这行代码，Python将在我们试图创建`ElectricCar`实例时引发错误。我们还需要更新模块`car`，使其包含`Car`类：

car.py

```
"""一个可用于表示汽车的类"""

class Car():
    --snip--
```

现在可以分别从每个模块中导入类，以根据需要创建任何类型的汽车了：

my_cars.py

```
❶ from car import Car
  from electric_car import ElectricCar
```

```
my_beetle = Car('volkswagen', 'beetle', 2016)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2016)
print(my_tesla.get_descriptive_name())
```

在❶处，我们从模块car 中导入了Car 类，并从模块electric_car 中导入ElectricCar 类。接下来，我们创建了一辆普通汽车和一辆电动汽车。这两种汽车都得以正确地创建：

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

9.4.7 自定义工作流程

正如你看到的，在组织大型项目的代码方面，Python提供了很多选项。熟悉所有这些选项很重要，这样你才能确定哪种项目组织方式是最佳的，并能理解别人开发的项目。

一开始应让代码结构尽可能简单。先尽可能在一个文件中完成所有的工作，确定一切都能正确运行后，再将类移到独立的模块中。如果你喜欢模块和文件的交互方式，可在项目开始时就尝试将类存储到模块中。先找出让你能够编写出可行代码的方式，再尝试让代码更为组织有序。

动手试一试

9-10 导入Restaurant 类：将最新的Restaurant 类存储在一个模块中。在另一个文件中，导入Restaurant 类，创建一个Restaurant 实例，并调用Restaurant 的一个方法，以确认import 语句正确无误。

9-11 导入Admin 类：以为完成练习9-8而做的工作为基础，将User 、Privileges 和Admin 类存储在一个模块中，再创建一个文件，在其中创建一个Admin 实例并对其调用方法show_privileges() ，以确认一切都能正确地运行。

9-12 多个模块：将User 类存储在一个模块中，并将Privileges 和Admin 类存储在另一个模块中。再创建一个文件，在其中创建一个Admin 实例，并对其调用方法show_privileges() ，以确认一切都依然能够正确地运行。

9.5 Python标准库

Python**标准库** 是一组模块，安装的Python都包含它。你现在对类的工作原理已有大致的了解，可以开始使用其他程序员编写好的模块了。可使用标准库中的任何函数和类，为此只需在程序开头包含一条简单的import 语句。下面来看模块collections 中的一个类——OrderedDict 。

字典让你能够将信息关联起来，但它们不记录你添加键—值对的顺序。要创建字典并记录其中的键—值对的添加顺序，可使用模块collections 中的OrderedDict 类。OrderedDict 实例的行为几乎与字典相同，区别只在于记录了键—值对的添加顺序。

我们再来看一看第6章的favorite_languages.py示例，但这次将记录被调查者参与调查的顺序：

favorite_languages.py

```
❶ from collections import OrderedDict

❷ favorite_languages = OrderedDict()

❸ favorite_languages['jen'] = 'python'
  favorite_languages['sarah'] = 'c'
  favorite_languages['edward'] = 'ruby'
  favorite_languages['phil'] = 'python'

❹ for name, language in favorite_languages.items():
      print(name.title() + "'s favorite language is " +
          language.title() + ".")
```

我们首先从模块collections 中导入了OrderedDict 类（见❶）。在❷处，我们创建了OrderedDict 类的一个实例，并将其存储到favorite_languages 中。请注意，这里没有使用花括号，而是调用OrderedDict() 来创建一个空的有序字典，并将其存储在favorite_languages 中。接下来，我们以每次一对的方式添加名字—语言对（见❸）。在❹处，我们遍历favorite_languages ，但知道将以添加的顺序获取调查结果：

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

这是一个很不错的类，它兼具列表和字典的主要优点（在将信息关联起来的同时保留原来的顺序）。等你开始对关心的现实情形建模时，可能会发现有序字典正好能够满足需求。随着你对标准库的了解越来越深入，将熟悉大量可帮助你处理常见情形的模块。

注意 你还可以从其他地方下载外部模块。本书第二部分的每个项目都需要使用外部模块，届时你将看到很多这样的示例。

动手试一试

9-13 使用OrderedDict：在练习6-4中，你使用了一个标准字典来表示词汇表。请使用OrderedDict 类来重写这个程序，并确认输出的顺序与你在字典中添加键—值对的顺序一致。

9-14 骰子：模块random 包含以各种方式生成随机数的函数，其中的randint() 返回一个位于指定范围内的整数，例如，下面的代码返回一个1~6内的整数：

```
from random import randint
x = randint(1, 6)
```

请创建一个Die 类，它包含一个名为sides 的属性，该属性的默认值为6。编写一个名为roll_die() 的方法，它打印位于1和骰子面数之间的随机数。创建一个6面的骰子，再掷10次。 创建一个10面的骰子和一个20面的骰子，并将它们都掷10次。

9-15 Python Module of the Week：要了解Python标准库，一个很不错的资源是网站Python Module of the Week。请访问http://pymotw.com/ 并查看其中的目录，在其中找一