第 5 章

对象与模块

使用计算机编程，主要是为了描述和解决现实世界中的现实问题。人类习惯于以对象思维认识现实世界，例如鸟类与一只家鸽、交通工具与一辆火车、狗类与一只小狗、平行四边形与一个矩形、汽车与一辆电动汽车。同类群体和个体反映了现实世界的不同事物之间有着复杂的联系。

面向对象编程(Object Oriented Programming)就是通过模拟现实世界中的个体和同类编写程序的一种思想，使用类来模拟同类群体，基于类来创建对象就是根据群体特征创建个体。面向对象的编程是针对大型软件设计而提出的，它不仅使软件开发更加灵活，而且提高了代码的复用率，利于大型软件的维护和开发。本章将介绍如何在 Python 中使用面向对象编程。

为什么程序员会说“人生苦短，我用 Python”？这主要因为 Python 中的模块。模块就是能实现特定功能的代码文件，包含一组相关函数，可以嵌入开发程序中，极大地提高了开发效率。Python 提供了丰富的模块，类似于一个丰富的武器库，也可以自己创建模块。本章将介绍如何在 Python 中使用模块、定义模块。

5.1 面向对象的程序设计

20 世纪 60 年代，为了应对软件危机，人们提出了面向对象(Object Oriented)的设计思想。从面向对象的概念提出到现在，面向对象已经发展成为一个比较成熟的编程思想，并且逐步成为软件开发领域的主流技术。

5.1.1 对象＝属性＋行为

对象的英文名是 Object，表示任意存在的事物，这是一个抽象概念。人类习惯于以对象思维认识事物，随处可见的事物就是一个对象，例如一个人、一只鸟、一只猫。

观察一个对象，可以将对象分成两部分：静态特征和动态行为。例如一个人的年龄、性别、身高、体重等都是这个人的静态特征。这些静态特征被称为属性，任何对象都有自己的属性，这些属性是客观存在的。动态行为是指一个对象的动作或行为，例如一个人可以完成

走路、跑步、说话、呼吸等动作，这些动作称为行为。

因此，对象＝属性＋行为。如果要在计算机上模拟现实的对象，就需要包括这两部分，即属性和行为。

5.1.2　类

类的英文名是 Class。在现实世界中，将具有相同属性和行为的一类实体称为类，例如鸟类和哺乳类。在编程中，类是指封装对象的属性和行为的载体。例如定义一个猫类(Cat)，可以定义猫类共有的静态属性：名字、年龄、颜色，也可以定义猫类共有的动态行为：趴下、打滚、舔毛、睡觉。

在 Python 中，使用数据来表示类的静态属性，使用函数来模拟类的动态行为，这些函数也称为方法。

5.1.3　面向对象程序设计的特点

封装、继承和多态是面向对象程序设计的三大特点。

1. 封装

封装就是将对象的静态属性和动态行为封装在一个载体中，这个载体就是类。封装在类中的静态属性主要是一些数据，封装在类中的动态行为主要是函数，也称为类的方法。使用类的程序员，不能看到类内部的实现细节，这就是封装思想。

采用封装思想保证了类内部数据结构的完整性，类的用户不能直接看到类内部的数据结构，只需调用类的方法或外部接口，这样避免了外部对内部数据的影响，提高了程序的可维护性。当然这涉及程序员的专业分工，有专门写类的程序开发者，有专门应用类的程序开发者。专业分工不仅能提高社会生产率，也能提高程序开发效率和代码的可维护性。

2. 继承

面向对象的程序设计提供了类的继承机制，允许程序员在保持原有类的基础上，创建更具体、更详细的类。以原有的类为基础产生新的类，也可以是新类继承了原有类的特征，其中原有的类称为父类，新的类称为子类。例如平行四边形和矩形，可以先创建一个平行四边形的类，具有对边平行且相等的特征，然后在平行四边形类的基础上创建矩形类，继承了父类的对边平行且相等的特征，而矩形类称为子类。

类的继承机制提高了代码的重用性和可扩充性。通过继承可以充分利用已有的分析、研究成果或解决方案。重用这些代码让程序员的开发工作不是无米之炊，而是站在了前人研究成果之上。当软件开发完成之后，如果对问题有了新的认识或问题发生了新的变化，则可以高效率地改造和扩充已有的软件。

3. 多态

多态是指以父类为基础创建多个子类，不同的子类调用相同名称的函数会产生不同的结果。例如先创建一个哺乳动物类，然后以哺乳动物类为基础创建狗类和猫类，狗类和猫类都继承了哺乳动物的特征，都可以发出声音，但声音是不同的。即子类在继承了父类的特征

时,也具备了自己的特征,并且产生了不同的结果,这就是多态。

5.2 类的定义和使用

在 Python 中,类表示封装了对象的属性和行为的载体。可以使用类来模拟现实世界中的事物,包括各类动物、植物、微生物、工业品。应用类创建对象,首先要定义类,然后使用类来创建实例,这样就能访问对象的属性和方法了。

5.2.1 定义一个简单的类

在 Python 中,使用关键字 class 来定义类,其语法格式如下:

```
class ClassName():
    '''类的注释信息'''
    statement          #类的内部代码块
```

其中,ClassName 表示要定义的类名,一般以大写字母开头,如果类名中有多个单词,则每个单词的首字母大写,这种命名法称为大驼峰式写法。当然,这只是惯例,也可以根据自己的习惯命名类。

'''类的注释信息'''是指定义该类的文档字符串,定义该字符串后,在创建类的某个对象时,输入类名和左侧的小括号"(",将显示该信息; statement 表示类的内部代码块,主要由类变量、方法、属性等定义语句组成,如果在定义类时没有想好具体的代码,则可以使用 pass 语句代替。

【实例 5-1】 创建一个最简单的类,代码如下:

```
# === 第 5 章 代码 5-1.py === #
class Cat():
    '''定义猫类'''
    pass
```

5.2.2 创建类的实例

定义完类之后,就可以创建类的实例了,即根据类来创建一个对象,其语法格式如下:

```
name1 = ClassName(parameters)
```

其中,name1 表示要创建对象的名称,只要符合 Python 的命名规则即可; ClassName 表示已经定义好的类; parameters 表示可选参数,创建一个类时,如果没有创建__init__()方法或 __init__()方法只有一个 self 参数,则 parameters 可省略不写。

【实例 5-2】 创建一个最简单的类,并使用该类创建一个对象,然后打印该对象,代码如下:

```
# === 第 5 章 代码 5 - 2.py === #
class Cat():
    '''定义猫类'''
    pass

cat1 = Cat()
print(cat1)
```

运行结果如图 5-1 所示。

图 5-1　代码 5-2.py 的运行结果

5.2.3　定义一个完整的类

在 Python 中可以使用关键字 class 来定义类。类中的函数称为方法。如果要使用类来实例化一个包含参数的对象，则必须使用__init__()方法，其语法格式如下：

```
class ClassName():
    '''类的注释信息'''
    def __init__(self,parameterlist):            #初始化方法
        statement1                               #方法内代码块 1

    def functionName(self,parameters):           #定义类的方法
        statement2                               #方法内代码块 2


instance1 = ClassName(parameters)                #使用类创建一个对象
instance1.functionName(parameter_value)          #调用对象的方法
```

其中，__init__()表示初始化方法，类似于 C++语言或 Java 语言的构造方法，每当创建一个类的新实例或对象时，Python 会自动执行该方法。__init__()方法必须包含一个 self 参数，而且这个参数必须是第 1 个参数，self 参数是一个指向实例本身的指针，用于访问类中的属性和方法，如果这个类是一栋房屋，则 self 参数就是这栋房屋的钥匙。在调用类的方法时，会自动传递 self 参数。如果__init__()方法只有一个参数，则在创建类的实例时不要指定实际的参数。parameterlist 表示初始化方法中除 self 以外的参数，各参数之间使用逗号分隔。parameters 是方法的参数。

functionName 表示类内部定义的方法名；instance1 表示类的实例名称，即使用类创建对象的名称；parameter_value 表示调用函数类的方法时，该方法的参数。

【实例 5-3】　创建一个猫类，并使用该类创建一个对象，然后调用该对象的方法，打印

该对象的属性,代码如下:

```
# === 第 5 章 代码 5-3.py === #
class Cat():
    '''猫类'''
    def __init__(self,name,age):                    # 初始化方法(构造方法)
            self.name = name
            self.age = age

    def run(self):                                   # 自定义方法
            print(self.name + "奔跑如飞.")

    def ask(self,state):                             # 自定义方法
            print(self.name + state)


cat1 = Cat('花猫',4)                                 # 创建类的实例
print('这只猫的名字是',cat1.name)                    # 调用对象的属性
print('这只猫的年龄是',cat1.age)                     # 调用对象的属性
cat1.run()                                           # 调用对象的方法
cat1.ask('你饿了?要吃东西?')                         # 调用对象的方法
```

运行结果如图 5-2 所示。

图 5-2 代码 5-3.py 的运行结果

注意:定义类内部的方法时,不同方法之间要空一行。定义完类后,当使用类创建对象时,代码之间要空两行。类就像一个工厂,对象就像一个工业品,运用类可以批量地创建对象。类与对象的编程思想模拟了现实世界的复杂关系。

5.2.4 类的数据成员

类的数据成员是指在类中定义的变量,也称为属性。代码 5-3.py 在对象中定义了属性,而不是在类中定义了属性。根据属性在类中定义的位置,可以分为类属性和实例属性。

1. 类属性

类属性是指将类的数据定义在类中,并且在方法体外的变量。类属性可以在类创建的所有实例之间共享值,即在所有实例化的对象中公用。

类属性可以通过类名或者实例名访问，而且可以动态地修改类属性，其语法格式如下：

```
class ClassName():
    '''类的注释信息'''
    name1 = value1                          #创建类属性
    name2 = value2                          #创建类属性
    def __init__(self):                     #初始化方法
        print(ClassName.name1)              #在初始化方法中调用类属性
        print(ClassName.name2)              #在初始化方法中调用类属性


instance1 = ClassName()                     #使用类创建一个对象
print(instance1.name1)                      #通过对象调用类属性
ClassName.name1 = value3                    #动态修改类属性
```

【实例 5-4】 创建一个猫类，在类中定义类属性。使用类创建实例，在实例中调用该属性，然后动态地修改类属性，代码如下：

```
# === 第 5 章 代码 5 - 4.py === #
class Cat():
    '''猫类'''
    name = '大脸猫'
    age = 1
    def __init__(self):                     #初始化方法(构造方法)
            print('名字是',Cat.name)
            print('年龄是',Cat.age)


cat1 = Cat()
Cat.name = '黑猫警长'                       #动态地修改类属性
cat2 = Cat()
```

运行结果如图 5-3 所示。

图 5-3　代码 5-4.py 的运行结果

2. 实例属性

实例属性是指定义在类的方法中的属性，只能作用于类创建的对象实例中，如果使用类名访问该属性，则会抛出异常，其语法格式如下：

```
class ClassName():
    '''类的注释信息'''
    def __init__(self):                          #初始化方法
            self.name1 = value1                  #定义实例属性
            self.name2 = value2                  #定义实例属性
            print(self.name1)
            print(self.name2)


instance1 = ClassName()                          #通过类创建对象实例 1
instance2 = ClassName()                          #通过类创建对象实例 2
instance2.name1 = value3                         #修改实例属性
```

【实例 5-5】 创建一个猫类,在类中定义实例属性。使用类创建对象,在对象中调用该属性,然后修改实例属性,代码如下:

```
# === 第 5 章 代码 5-5.py === #
class Cat():
    '''猫类'''
    def __init__(self):                          #初始化方法(构造方法)
            self.name = '大脸猫'
            self.age = 3
            print('名字是',self.name)
            print('年龄是',self.age)


cat1 = Cat()
cat2 = Cat()
cat2.name = '黑猫警长'
print('对象 2 的名字是',cat2.name)
```

运行结果如图 5-4 所示。

图 5-4 代码 5-5.py 的运行结果

5.2.5 访问限制

创建类时,可以定义该类的属性和方法;创建类后,在类的外部也可以调用属性和方法来操作数据,从而隐藏了类内部的代码逻辑。

为了保证类内部的某些属性或方法不被外部所访问，可以在属性名或方法名前面添加单下画线、双下画线；或首尾加下画线，从而限制访问权限，其中单下画线(_name)、双下画线(__name)、首尾双下画线(__name__)的作用如下：

(1) 以单下画线开头表示保护(protected)类型的属性或方法，只允许本身和子类进行访问，但不能使用“from module import *”语句导入的类。

(2) 以双下画线开头表示私有(private)类型的属性或方法，只允许类本身进行访问，而且不能通过类的对象实例进行访问，但是可以通过类的实例名._类名__xxx 的格式进行访问。

(3) 首尾双下画线表示定义特殊方法，一般是系统定义名字，例如初始化方法__init__()。

【实例 5-6】 创建一个猫类。在类中定义一个保护类型的属性，定义一个私有类型的属性。使用该类创建对象实例，并调用保护类型的属性和私有类型的属性，代码如下：

```
# === 第 5 章 代码 5 - 6.py === #
class Cat():
    '''猫类'''
    __name = '大脸猫'              #定义私有属性
    _age = 2                       #定义保护属性
    def __init__(self):            #初始化方法(构造方法)
            print('通过类本身访问私有属性:',Cat.__name)
            print('通过类本身访问保护属性:',Cat._age)


cat1 = Cat()
print('通过对象实例访问私有属性:',cat1._Cat__name)
print('通过对象实例访问保护属性:',cat1._age)
```

运行结果如图 5-5 所示。

图 5-5　代码 5-6.py 的运行结果

【实例 5-7】 创建一个圆类。在类中定义一个计算圆的面积的方法，定义一个计算圆的周长的方法。使用该类创建对象实例，并调用这两种方法，代码如下：

```
# === 第 5 章 代码 5 - 7.py === #
class Circle():
    '''圆类'''
    def __init__(self,radius):
```

```
            self.radius = radius

    def area(self):
            return 3.1415 * self.radius * self.radius

    def perimeter(self):
            return 2 * 3.1415 * self.radius


circle1 = Circle(10)
print('圆的面积是',circle1.area())
print('圆的周长是',circle1.perimeter())
```

运行结果如图 5-6 所示。

图 5-6　代码 5-7.py 的运行结果

注意：Python 中的列表、元组、字典、集合、字符串等数据类型，本质上都是类，封装了很多方法和属性，如同实例 5-7 的类。

前面学过列表、元组、字典、集合等数据类型，在 Python 中可以创建以对象为元素的列表。

【实例 5-8】 创建一个列表。列表的元素是对象，代码如下：

```
# === 第 5 章 代码 5 - 8.py === #
class Cat():
    color = '蓝色'
    name = '蓝猫'
    number = 0
    def __init__(self):
            Cat.number += 1
            print("\n 这是第" + str(Cat.number) + "只蓝猫.")


list1 = []
for i in range(5):
    list1.append(Cat())

print("一共有" + str(Cat.number) + "只蓝猫.")
```

运行结果如图 5-7 所示。

图 5-7　代码 5-8.py 的运行结果

5.3　继承与导入

编写类时，并不是每次都要从头到尾重新定义一个类。如果要编写的类和另一个已经存在的类之间存在一定的继承关系，则可以通过继承来达到代码重用的目的，从而提高开发效率。

当一个类继承另一个类时，这个类将自动获得另一个类的所有属性和方法。原有的类称为父类或基类，而新类称为子类或派生类。子类继承了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。

5.3.1　继承的基本语法

在 Python 中，可以在子类的定义语句的类名的右侧使用一对小括号将要继承的父类名字括起来，从而实现类的继承，其语法格式如下：


```
class ClassName(BasicClass):
    '''类的帮助信息'''
    statement              #类的代码块
```

其中，ClassName 是类名；BasicClass 是要继承的父类，可以有多个，类名之间用逗号隔开；'''类的帮助信息'''用于指定类的文档字符串，定义该字符串后，在创建类的某个对象时，输入类名和左侧的括号，将显示该信息；statement 表示类的代码块，主要由类的方法、属性等定义语句组成，如果没有构思好类的代码，则可以直接使用 pass 语句代替。

【实例 5-9】　创建一个水果类，然后以水果类为父类，创建桃类和苹果类，代码如下：

```
# === 第 5 章 代码 5 - 9.py === #
class Fruits():
```

```
    '''水果类'''
    taste = '酸涩'
    def ripe(self,taste):
            print('水果成熟前的味道是' + Fruits.taste + '的')
            print('水果成熟后的味道是' + taste + '的')


class Peach(Fruits):
    '''桃类'''
    taste = '甘甜'
    def __init__(self):
            print('我是桃子!')


class Apple(Fruits):
    '''苹果'''
    taste = '酸甜'
    def __init__(self):
            print('我是苹果!')


peach1 = Peach()
peach1.ripe(peach1.taste)
apple1 = Apple()
apple1.ripe(apple1.taste)
```

运行结果如图 5-8 所示。

图 5-8　代码 5-9.py 的运行结果

注意：在代码 5-9.py 文件中，桃类和苹果类都继承了水果类的方法 ripe()。

5.3.2　派生类中调用基类的__init__()方法

在派生类或子类中定义初始化方法__init__()时，一般不会自动调用基类或父类中的初始化方法__init__()。如果要调用基类的初始化方法__init__()，则需要在派生类中使用 super()函数调用基类的初始化方法__init__()，其语法格式如下：

```
super().__init__()
```

【实例 5-10】 创建一个水果类，然后以水果类为父类，创建桃类和苹果类，桃类和苹果类要调用基类的__init__()方法，代码如下：

```
# === 第 5 章 代码 5 - 10.py === #
class Fruits():
    '''水果类'''
    def __init__(self,taste = '既酸又涩'):
            Fruits.taste = taste

    def ripe(self,taste):
            print('水果成熟前的味道是' + Fruits.taste)
            print('水果成熟后的味道是' + taste + '的')


class Peach(Fruits):
    '''桃类'''
    taste = '甘甜'
    def __init__(self):
            print('我是桃子!')
            super().__init__()


class Apple(Fruits):
    '''苹果'''
    taste = '酸甜'
    def __init__(self):
            print('我是苹果!')
            super().__init__()


peach1 = Peach()
peach1.ripe(peach1.taste)
apple1 = Apple()
apple1.ripe(apple1.taste)
```

运行结果如图 5-9 所示。

图 5-9　代码 5-10.py 的运行结果

5.3.3 方法重写


8min

基类或父类的属性和方法都会被派生类或子类继承,当基类的某种方法不完全适用于派生类时,需要在派生类中重写这种方法。

【实例 5-11】 创建一个水果类,然后以水果类为父类,创建桃类,桃类要重写水果类的方法,代码如下:

```
# === 第 5 章 代码 5 - 11.py === #
class Fruits():
    '''水果类'''
    taste = '酸涩'
    def ripe(self,taste):
            print('水果成熟前的味道是' + Fruits.taste + '的')
            print('水果成熟后的味道是' + taste + '的')


class Peach(Fruits):
    '''桃类'''
    taste = '甘甜'
    def __init__(self):
            print('我是桃子!')

    def ripe(self,taste):
            print('成熟的桃子的味道是' + Peach.taste)
            print('成熟的桃子全身是宝,桃仁是一味中药材')


peach1 = Peach()
peach1.ripe(peach1.taste)
```

运行结果如图 5-10 所示。

图 5-10 代码 5-11.py 的运行结果

5.3.4 导入类


11min

如果不断地给类添加方法或属性,则文件会变得很长。为了遵循 Python 代码简洁的理念,Python 允许将类存储在一个文件中,然后在主程序中导入该文件中的类。

1. 导入单个类

创建一个代码文件 circle.py，将 Circle 类存储在这个代码文件中，其代码如下：

```
# === 第 5 章 代码 circle.py === #
class Circle():
    '''圆类'''
    def __init__(self,radius):
            self.radius = radius

    def area(self):
            return 3.1415 * self.radius * self.radius

    def perimeter(self):
            return 2 * 3.1415 * self.radius
```

在文件 circle.py 的同一目录下，创建一个文件 5-12.py。在该文件中使用 from circle import Circle 语句，即可导入 Circle 类，其代码如下：

```
# === 第 5 章 代码 5 - 12.py === #
from circle import Circle


circle1 = Circle(5)
print('圆的面积是',circle1.area())
print('圆的周长是',circle1.perimeter())
```

运行结果如图 5-11 所示。

图 5-11　代码 5-12.py 的运行结果

2. 在一个文件中导入多个类

在 Python 中，也可以在一个文件中存储多个类。例如将 Fruits 类、Peach 类、Apple 类都存储在 fruits.py 文件中，其代码如下：

```
# === 第 5 章 代码 fruits.py === #
class Fruits():
    '''水果类'''
    def __init__(self,taste = '既酸又涩'):
            Fruits.taste = taste

    def ripe(self,taste):
```

```
            print('水果成熟前的味道是' + Fruits.taste)
            print('水果成熟后的味道是' + taste + '的')


class Peach(Fruits):
    '''桃类'''
    taste = '甘甜'
    def __init__(self):
            print('我是桃子!')
            super().__init__()


class Apple(Fruits):
    '''苹果'''
    taste = '酸甜'
    def __init__(self):
            print('我是苹果!')
            super().__init__()
```

在文件 fruits.py 的同一目录下,创建一个文件 5-13.py。可以根据需要在 5-13.py 文件中导入任意需要的类,其代码如下:

```
# === 第 5 章 代码 5 - 13.py === #
from fruits import Peach,Apple


peach1 = Peach()
peach1.ripe(peach1.taste)
apple1 = Apple()
apple1.ripe(apple1.taste)
```

运行结果如图 5-12 所示。

图 5-12 代码 5-13.py 的运行结果

3. 导入整个文件

在 Python 中,可以导入整个文件,然后使用文件名和点号(.)表示要访问的类。以文件 fruits.py 为例,在该文件的同一目录下创建 5-14.py,其代码如下:

```
# === 第 5 章 代码 5 - 14.py === #
import fruits


peach1 = fruits.Peach()
peach1.ripe(peach1.taste)
apple1 = fruits.Apple()
apple1.ripe(apple1.taste)
```

运行结果如图 5-13 所示。

图 5-13　代码 5-14.py 的运行结果

4. 导入文件中的所有类

在 Python 中，可以导入整个文件中的所有类，然后使用 from file_name import * 语句，可以导入文件中的每个类。以文件 fruits.py 为例，在该文件的同一目录下创建 5-15.py，其代码如下：

```
# === 第 5 章 代码 5 - 15.py === #
from fruits import *              # 注意 import 和 * 之间无空格


peach1 = Peach()
peach1.ripe(peach1.taste)
apple1 = Apple()
apple1.ripe(apple1.taste)
fruit1 = Fruits()
fruit1.ripe('好吃极了')
```

运行结果如图 5-14 所示。

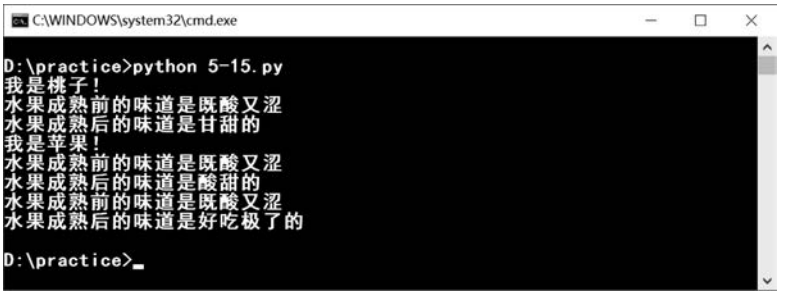

图 5-14　代码 5-15.py 的运行结果

5.4 模块

Python 中拥有数量巨大、功能强大的模块。Python 的标准库中包含大量模块,也称为标准模块,即只要安装 Python 程序,就已经拥有标准模块。Python 也提供了功能强大的第三方模块。可以在 Python 中安装第三方模块,也可以卸载第三方模块,还可以应用第三方模块。应用 Python 的模块,极大地提高了编程者的开发效率。

另外编程者也可以自定义模块,将自己认为经常用的功能集合在同一个模块中。

5.4.1 模块概述


3min

模块的英文单词是 Module,Module 在英文中也有组件、模件、单元、功能块的意思,可以将模块类比成一盒积木,利用这些"积木"可以拼接成需要的物品,也可以将模块类比成机械组件,利用这些组件可以组装成需要的机械装备。

在 Python 中,模块是指一个 Python 程序,其中封装了实现某些具体功能的函数,可以嵌入程序中。其实,平时写的 Python 代码,保存的每个.py 文件都是一个独立的模块。

在 Python 中可以使用语句 pip list 查看 Python 上安装的模块。在 Windows 命令行窗口中输入 pip list 语句,运行结果如图 5-15 所示。

5.4.2 安装、升级、卸载模块


6min

虽然 Python 标准库中包含了大量模块,但有时也需要使用第三方模块,因为第三方模块的功能很强大。

1. 安装模块

如何使用第三方模块?首先要在 Windows 命令行窗口中安装第三方模块,然后才能使用该模块。安装第三方模块的语法格式如下:

```
pip install 模块名[ == version]
```

其中,方括号表示可选参数,可有可无;version 表示版本号;如果省略"==version",则会安装最新版本的模块。

如果安装速度比较慢,则可以使用国内的软件镜像,其语法格式如下:

```
pip install -i url 模块名
```

或

```
pip install 模块名 -i url
```

其中,url 表示软件镜像的网络地址。例如在第 1 章第 2 节介绍过一款 Python 集成开发工具 Spyde。Spyde 也是 Python 的第三方模块。通过清华大学软件镜像安装 Spyder 模块的语句如下:

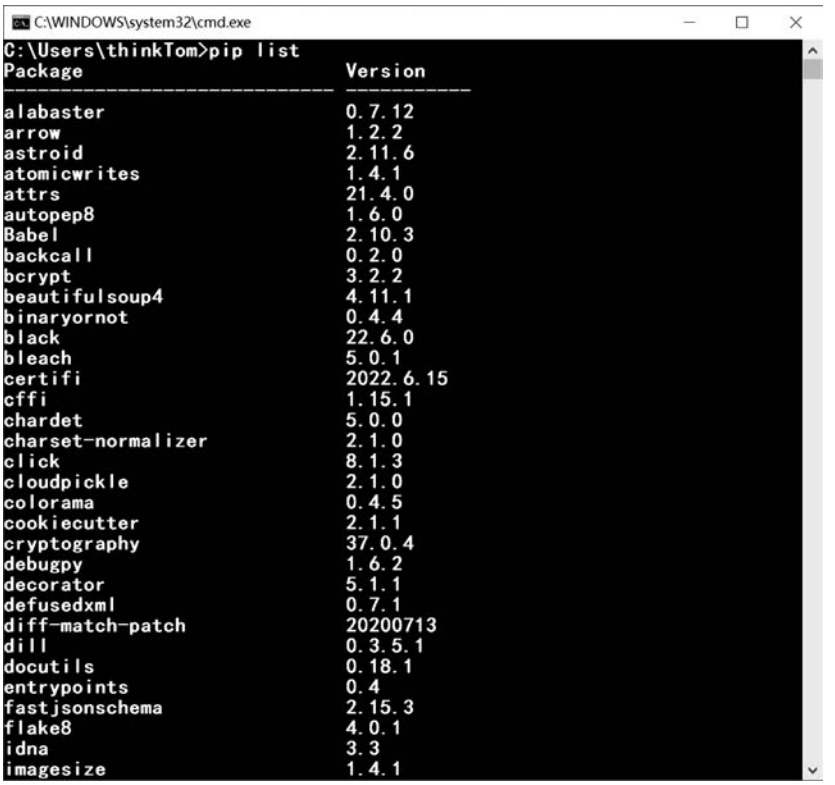

图 5-15　Python 中安装的模块

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple spyder
```

通过阿里云软件镜像安装 Spyder 模块的语句如下：

```
pip install -i http://mirrors.aliyun.com/pypi /simple spyder
```

注意：如果第三方模块的镜像网址发生了变更，则需要重新找出该模块的网络地址，否则在安装第三方模块时将会发生错误。

2. 升级模块

Python 中的第三方模块由专业的团队进行维护、更新、升级。如果要升级第三方模块，则要在 Windows 命令行窗口中输入升级模块语句，其语法格式如下：

```
pip install --upgrade 模块名[==version]
```

3. 卸载模块

对于不需要的第三方模块，Python 也提供了卸载该模块的方法。需要在 Windows 命令行窗口中输入卸载语句，其语法格式如下：

```
pip uninstall 模块名
```

例如卸载第三方模块 Spyder,卸载步骤如下:

(1) 在 Windows 命令行窗口中输入 pip uninstall spyder 语句,然后按 Enter 键,如图 5-16 所示。

图 5-16 卸载第三方模块 Spyder 过程(1)

(2) 在 Windows 命令行窗口中输入 Y,然后按 Enter 键,如图 5-17 所示。

图 5-17 卸载第三方模块 Spyder 过程(2)

(3) 如果在 Windows 命令行窗口中显示卸载成功的英文,则表示已经卸载了 Spyder 模块,如图 5-18 所示。

注意:不同的第三方模块的卸载过程与此类似,唯一不同的是需要输入不同次数的英文字母 Y 或 y。

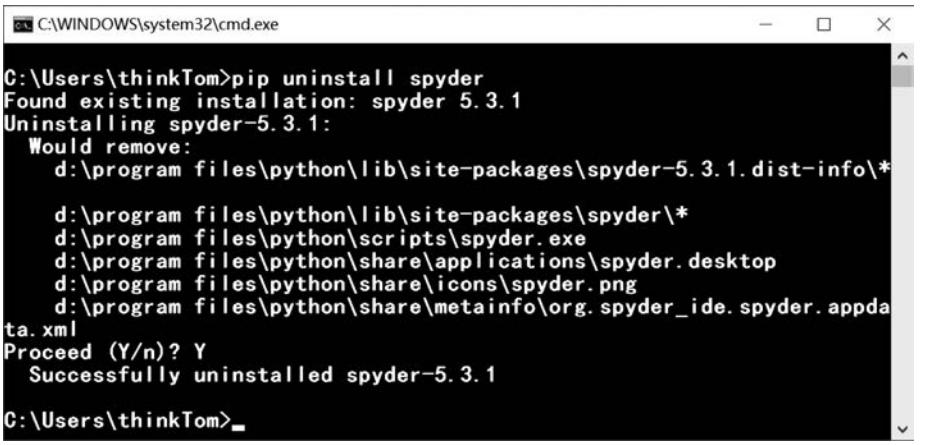

图 5-18　卸载第三方模块 Spyder 过程(3)

5.4.3　引入模块

Python 的模块既包括标准模块，也包括第三方模块。在 Python 中，无论是标准模块，还是第三方模块，引入模块的语句及方法是相同的。Python 提供了 4 种引入模块的方法。

1. 使用 import 语句导入整个模块

无论是标准模块、第三方模块，还是自定义模块，都可以使用该方法，其语法格式如下：

```
import 模块名
```

使用该语句引入模块后，如果要调用该模块中的变量、函数、类，则需要在变量名、函数名、类名前添加前缀“模块名.”。

Turtle 库是 Python 语言中的一个很流行的绘制图像的函数库。绘制图像的原理是：一个小乌龟在一个横轴为 x、纵轴为 y 的坐标系原点，从(0,0)位置开始，它根据一组函数指令的控制，在这个平面坐标系中移动，从而在它爬行的路径上绘制图形。

Turtle 库中的函数 forward()表示前进的步数(前进 1 步就是前进 1 像素)，函数 left()表示逆时针旋转，函数 pensize()表示设置画笔的宽度。

【实例 5-12】 使用 Python 的标准模块 Turtle 绘制一个等边三角形，代码如下：

```
# === 第 5 章 代码 5 - 16.py === #
import turtle

turtle.pensize(6)                    #设置画笔的宽度
for i in range(3):
    turtle.forward(200)              #前进 200 步
    turtle.left(120)                 #逆时针旋转 120°

input()                              #输入函数保持绘制画面
```

在代码文件的当前目录下,在 Windows 命令行窗口下输入 python 5-16.py,然后按 Enter 键即可运行该程序。运行结果如图 5-19 所示。

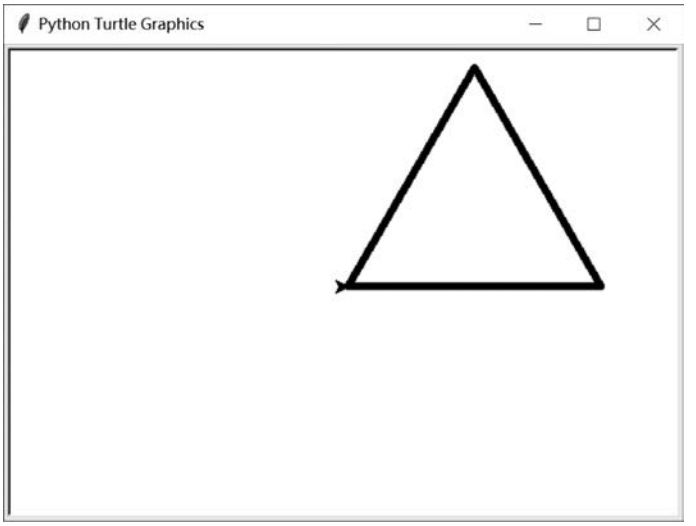

图 5-19 代码 5-16.py 的运行结果

注意:对于代码 5-16.py 运行而创建的图像,可以直接关闭该图像,也可以在 Windows 命令行窗口中输入任意一个字母或数字,然后按 Enter 键,即可关闭该画面。输入函数 input()的作用是保持该程序的运行状态,一旦该程序不再运行,绘制的图像也会被关闭。

2. 使用 import 语句重命名模块

无论是标准模块、第三方模块,还是自定义模块,都可以使用该方法,其语法格式如下:

```
import 模块名 as 新模块名
```

使用该语句引入模块后,如果要调用该模块中的变量、函数、类,则需要在变量名、函数名、类名前添加前缀"新模块名."。

【实例 5-13】 使用 Python 的标准模块 Turtle 绘制一个正方形,代码如下:

```
# === 第 5 章 代码 5-17.py === #
import turtle as tk

tk.pensize(6)                    # 设置画笔的宽度
for i in range(4):
    tk.forward(200)              # 前进 200 步
    tk.left(90)                  # 逆时针旋转 90°

input()                          # 输入函数保持绘制画面
```

在代码文件的当前目录下,在 Windows 命令行窗口下输入 python 5-17.py,然后按

Enter 键即可运行该程序。运行结果如图 5-20 所示。

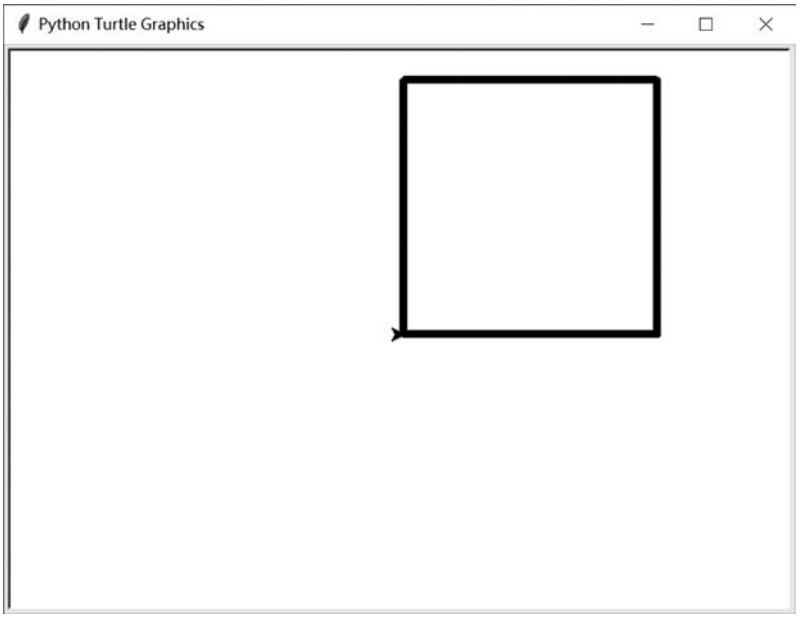

图 5-20　代码 5-17.py 的运行结果

在使用 import 语句导入模块时，每执行一条 import 语句都会创建一个新的命名空间(Namespace)，并且在当前的命名空间下执行与.py 文件相关的所有语句。如果两个命名空间发生冲突，则会引起编译混乱，因此需要在引入模块的具体变量名、函数名、类名前加上前缀“模块名.”或前缀“新模块名.”。类似于某年级 1 班有名同学叫小明，2 班有名同学也叫小明，如果 2 班的同学进入 1 班教室和 1 班的同学一起上课，则名字小明对应了两名同学，因此对 2 班的小明，要加上“2 班的”。

注意：命名空间是记录对象名字和对象之间对应关系的空间，可类比于某个教室中名字与同学之间的对应关系。在 Python 中，命名空间一般通过字典(dict)实现，其中 key 是标识符；value 是具体的对象。

3. 使用 from 语句导入模块中的所有成员

无论是标准模块、第三方模块，还是自定义模块，都可以使用该方法，其语法格式如下：

```
from 模块名 import *
```

使用该语句引入模块后，可以直接调用该模块中的变量、函数、类，而不需要加前缀。

【实例 5-14】 使用 Python 的标准模块 Turtle 绘制一个五角星，代码如下：

```
# === 第 5 章 代码 5 - 18.py === #
from turtle import *
```

```
pensize(6)                          #设置画笔的宽度
for i in range(5):
    forward(200)                    #前进 200 步
    left(144)                       #逆时针旋转 144°

input()                             #输入函数保持绘制画面
```

在代码文件的当前目录下,在 Windows 命令行窗口下输入 python 5-18.py,然后按 Enter 键即可运行该程序。运行结果如图 5-21 所示。

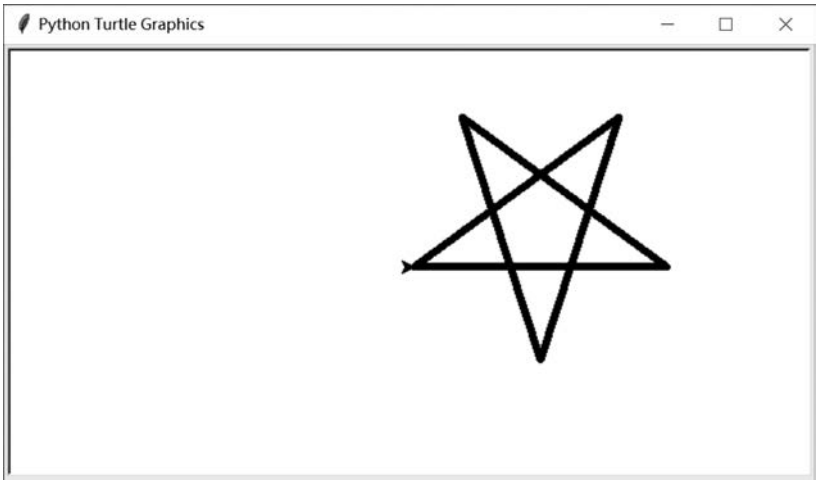

图 5-21 代码 5-18.py 的运行结果

4. 使用 from 语句导入模块中的部分成员

无论是标准模块、第三方模块,还是自定义模块,都可以使用该方法,其语法格式如下:

```
from 模块名 import member_name1,member_name2,...,member_namen
```

其中,member_name1、member_name2、……、member_namen 表示要导入的变量名、函数名、类名。

【实例 5-15】 使用 Python 的标准模块 Turtle 绘制一个正六边形,代码如下:

```
# === 第 5 章 代码 5-19.py === #
from turtle import pensize,forward,left

pensize(5)                          #设置画笔的宽度
for i in range(6):
    forward(130)                    #前进 130 步
    left(60)                        #逆时针旋转 60°

input()                             #输入函数保持绘制画面
```

在代码文件的当前目录下,在 Windows 命令行窗口下输入 python 5-19.py,然后按

Enter 键即可运行该程序。运行结果如图 5-22 所示。

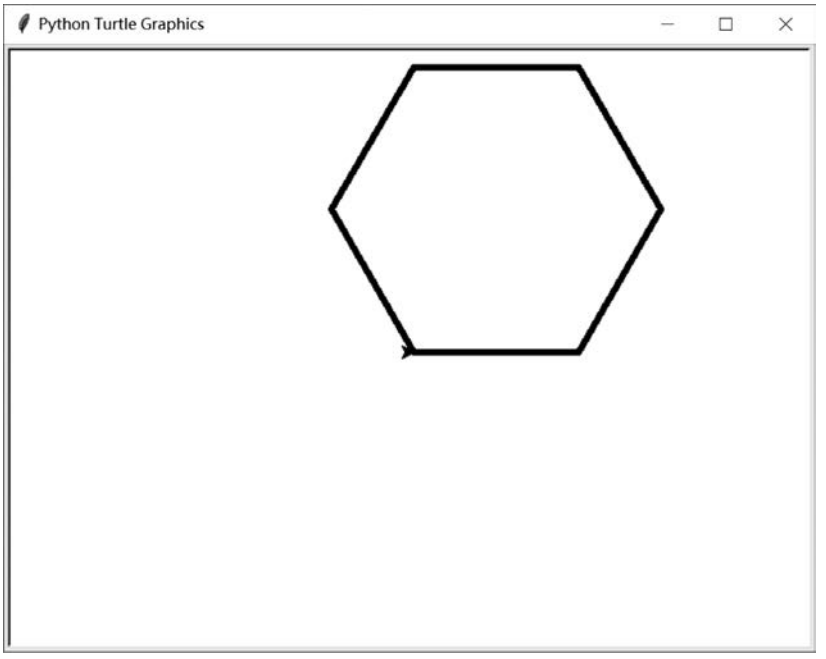

图 5-22　代码 5-19.py 的运行结果

注意：这 4 种方法都可以引入模块，如果需要引入多个模块而担心重名问题，则推荐使用前两种方法；如果需要引入一个模块，则推荐使用后两种方法。

5.4.4　创建主程序

如果读者阅读过别人写的 Python 代码，则会发现其中很多代码文件包含一行语句 if __name__==='__main__'，这行语句有什么作用？难道必须写这行语句？各位读者先不要着急寻找原因，先看一个例子。

【实例 5-16】 创建一个模块文件 rect.py，在该文件中创建一个全局变量 rectangle，然后创建一个矩形类，矩形类有两种方法：一种是计算周长，另一种是计算面积。在类定义体外，测试一下这个类，即创建一个对象，调用该对象的两种方法，代码如下：

```
# === 第 5 章 代码 rect.py === #
rectangle = '矩形'                          # 全局变量

class Rect():
    '''矩形类'''
    def __init__(self,width,height):   # 初始化方法
            self.width = width
```

```
            self.height = height

        def area(self):                          #计算面积
            return self.width * self.height

        def perimeter(self):                     #计算周长
            return 2 * (self.width + self.height)

    # ====== 类体外 ====== #
    print('现在是类体外')
    rect1 = Rect(10,8)
    print('该矩形的面积是',rect1.area())
    print('该矩形的周长是',rect1.perimeter())
```

运行结果如图 5-23 所示。

图 5-23　代码 rect.py 的运行结果

在模块文件 rect.py 的同级目录下,创建一个名称为 5-20.py 的文件,在该文件中导入 rect 模块,然后输出该模块中的全局变量 rectangle,代码如下:

```
# === 第 5 章 代码 5 - 20.py === #
import rect

print('全局变量的值为',rect.rectangle)
```

运行结果如图 5-24 所示。

图 5-24　代码 5-20.py 的运行结果

从代码 5-20.py 的运行结果可以看出,导入模块后,模块中原有的测试代码都被执行了,这个结果不是我们想要的。

在模块文件中，加入语句 if __name__==='__main__'，并且将该语句放在模块中的测试代码之前。模块 rect.py 的代码如下：

```
# === 第 5 章 修改后的代码 rect.py === #
rectangle = '矩形'                          #全局变量

class Rect():
    '''矩形类'''
    def __init__(self,width,height):   #初始化方法
            self.width = width
            self.height = height

    def area(self):                    #计算面积
            return self.width * self.height

    def perimeter(self):               #计算周长
            return 2 * (self.width + self.height)

# ====== 类体外 ====== #
if __name__ == '__main__':
    print('    现在是类体外    ')
    rect1 = Rect(10,8)
    print('该矩形的面积是',rect1.area())
    print('该矩形的周长是',rect1.perimeter())
```

运行结果如图 5-25 所示。

图 5-25　修改代码 rect.py 后的运行结果

从修改后模块文件 rect.py 的运行结果可以看出，加入语句 if __name__==='__main__'后，并不影响该模块中的测试代码的运行。

重新运行代码 5-20.py，运行结果如图 5-26 所示。

图 5-26　修改代码 rect.py 后，代码 5-20.py 的运行结果

从运行结果可以看出,这次没有运行模块文件中的测试代码,而是直接输出了全局变量。

在每个模块的定义时,都包含一个记录模块名称的变量__name__,Python 可以检查该变量,以确定模块文件中的程序在哪个文件中执行。如果一个模块不是被导入其他程序中,则模块程序可能在解释器的顶级模块中执行,Python 顶级模块的变量__name__的值是__main__。

5.4.5 自定义模块

在 Python 中,可以自定义模块。自定义模块有两个作用:一是可以将经常用的变量、函数、类保存在模块文件中,方便其他程序使用已经写好的代码,从而提高开发效率;二是规范代码,提高代码的可读性。

实现自定义模块分为两部分,一部分是创建模块,另一部分是导入模块。

【实例 5-17】 创建一个模块文件 bai.py,在该模块文件下创建一个函数 fun_bai(),这个函数可以计算并输出百钱买百鸡的结果,然后创建一个文件,导入该模块并调用模块中的函数。创建模块的代码如下:

```
# === 第 5 章 模块 bai.py === #
"""
@百钱买百鸡问题
"""
def fun_bai():
    for i in range(1,20):
            for j in range(1,33):
                    if 5 * i + 3 * j + (100 - i - j)/3 == 100:
                            print("公鸡、母鸡、小鸡的数目分别是:",i,j,100 - i - j)

if __name__ == '__main__':
    fun_bai()
```

引入模块的代码如下:

```
# === 第 5 章 代码 5 - 21.py === #
import bai

bai.fun_bai()
```

运行结果如图 5-27 所示。

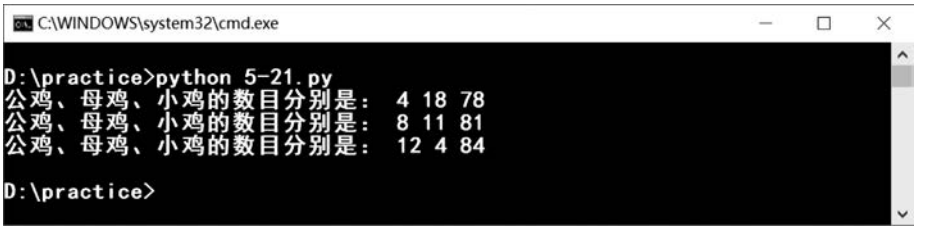

图 5-27 代码 5-21.py 的运行结果

注意：模块文件的本质就是 Python 程序文件，也就是后缀名是.py 的文件。如果 Python 程序文件被别的文件引入，则该程序文件也称为模块文件。

5.5 小结

本章介绍了面向对象程序设计的思想，在 Python 中如何定义类，如何应用类创建对象及类的继承机制，如何重写派生类的属性和方法，如何导入类的方法。

本章也介绍了 Python 中的模块，包括标准模块和第三方模块。如何安装第三方模块，如何引入模块，如何自定义模块，如何创建主程序。第 6 章将介绍 Python 中的异常处理。