第 5 章
类和对象

学习目标

- 理解面向对象的基本概念。
- 理解 Python 的类和类型。
- 理解 Python 中的对象。
- 掌握类的定义及使用。
- 掌握类和对象的属性及方法。
- 掌握特殊属性和方法。
- 掌握伪私有属性和方法。
- 掌握静态方法、类的构造和初始化。
- 掌握简单继承、多重继承。

面向过程的程序设计方法需要编程人员直接定义每一个需要用到的变量，直接编写每一段需要的程序代码，编程人员直接操作所有的数据，实现所有的功能。这种程序设计方法难以保证程序的安全性和代码的可重用性，即难以有效保证程序的质量和开发效率。保证程序的安全性和代码的可重用性是面向对象程序设计方法的优势。

Python 程序的交互执行方式适合运行一些基本的语句或函数。程序或函数是对语句的封装，可以批量地执行源代码，既增强了程序的抽象能力，又支持了代码复用。更高层次的抽象和封装是面向对象的程序设计，不但可以封装代码，还可以封装操作的数据。

面向对象程序设计的核心是运用现实世界的概念抽象地思考问题，从而自然地解决问题。面向对象的程序设计使得软件开发更加灵活，能更好地支持代码复用和设计复用，适用于大软件的设计与开发。本章将介绍面向对象程序设计的基本特性，重点介绍类和对象的概念，以及类的封装、继承、多态等知识。

5.1 类和对象基本知识

5.1.1 面向对象的基本概念

面向对象的基本概念如下。

- 类和对象：描述对象的属性和方法的集合称为类，它定义了同一类对象共有的属性和方法；对象是类的实例，也称为实例对象。
- 方法：类中定义的函数用于描述对象的行为，也称为方法成员。

- 属性：类中在所有方法之外定义的变量(也称为类中的顶层变量)，用于描述对象的特点，也称为数据成员。
- 封装：类具有封装特性，其内部实现不应被外界知晓，只需要提供必要的接口供外部访问即可。
- 实例化：创建一个类的实例对象。
- 继承：当从一个基类(也称为父类或超类)派生出一个子类时，子类拥有基类的属性和方法，称为继承；子类可以定义自己的属性和方法。
- 重载(override)：在子类中定义和父类方法同名的方法称为子类对父类方法的重载，也称为方法重写。
- 多态：指不同类型对象的相同行为产生了不同的结果。

和其他面向对象的程序设计语言相比，Python 的面向对象机制更为简单。

5.1.2 Python 的类和类型

Python 的类使用 class 语句来定义，类通常包含一系列的赋值语句和函数定义。赋值语句定义类的属性，函数定义类的方法。在 Python 3 中，类是一种自定义类型。

Python 的所有类型(包括自定义类型)都是内置类型 type 的实例对象。例如，内置的 int、float、str 等都是 type 类型的实例对象。

type()函数可返回对象的类型，示例代码如下：

```
>>>type(int)
<class'type'>
>>>type(float)
<class'type'>
>>>type(str)
<class'type'>
>>>class test:                                    #定义一个空类
...    pass
...
>>>type(test)
<class'type'>
```

5.1.3 Python 中的对象

Python 中的一切数据都是对象，例如整数、小数、字符串、函数、模块等。

例如，下面的代码分别测试了字符串、整数、逻辑值和函数的类型。

```
>>>type('abc')
<class'str'>
>>>type(123)
<class'int'>
>>>type(True)
<class'bool'>
>>>def fun():
...    pass
>>>type(fun)
<class'function'>
```

Python 中的对象分为两种：类对象和实例对象。

类对象在执行 class 语句时创建。类对象是可调用的，类对象也称为类实例。调用类对象会创建一个类的实例对象。类对象只有一个，而类的实例对象可以有多个。

类对象和实例对象分别拥有自己的命名空间，并在各自的命名空间内使用对象的属性和方法。

1. 类对象

类对象具有下列主要特点。

- Python 在执行 class 语句时会创建一个类对象和一个变量（与类同名），变量引用类对象。与 def 类似，class 也是可执行语句。导入类模块时，会执行 class 语句。
- 类中的顶层赋值语句创建的变量是类的数据属性。类的数据属性用“对象名.属性名”的格式来访问。
- 类中的顶层 def 语句定义的函数是类的方法属性，用“对象名.方法名()”的格式来访问。
- 类的数据属性由类的所有实例对象共享。实例对象可读取类的数据属性值，但不能通过赋值语句修改类的数据属性值。

2. 实例对象

实例对象具有下列主要特点。

- 实例对象通过调用类对象来创建。
- 每个实例对象继承类对象的所有属性，并获得自己的命名空间。
- 实例对象拥有私有属性。当通过赋值语句为实例对象的属性赋值时，如果该属性不存在，则会创建属于实例对象的私有属性。

5.1.4　定义类

类定义的基本格式为

```
class 类名:
    赋值语句
    赋值语句
    …
    def 语句定义函数
    def 语句定义函数
    …
class testclass:
    data=100
    def setpdata(self,value):
        self.pdata=value
    def showpdata(self):
        print('self.pdata=',self.pdata)
```

5.1.5　使用类

使用类对象可访问类的属性、创建实例对象，示例代码如下：

```
>>>type(testclass)          #测试类对象的类型
<class'type'>
>>>testclass.data           #访问类对象的数据属性
100
>>>x=testclass()            #调用类对象创建第一个实例对象
>>>type(x)                  #查看实例对象的类型,交互环境中的默认模块名称为__main__
<class'__main__.testclass'>
>>>x.setpdata('abc')        #调用方法创建实例对象的数据属性 pdata
>>>x.showpdata()            #调用方法显示实例对象的数据属性 pdata 的值
self.pdata=abc
>>>y=testclass()            #调用类对象创建第二个实例对象
>>>y.setpdata(123)          #调用方法创建实例对象的数据属性 pdata
>>>y.showpdata()            #调用方法显示实例对象的数据属性 pdata 的值
self.pdata=123
```

5.2 类的方法

5.2.1 类和对象的属性

在 Python 中,实例对象拥有类对象的所有属性,可以用 dir()函数来查看对象的属性,示例代码如下:

```
>>>dir(testclass)                          #查看类对象的属性
…
>>>x=testclass()
>>>dir(x)                                  #查看实例对象的属性
…
```

1. 共享属性

类对象的数据属性是全局的,并可通过实例对象来引用。

testclass 类顶层的赋值语句“data=100”定义了类对象的属性 data,该属性可与所有实例对象共享,示例代码如下:

```
>>>x.data,y.data                           #访问共享属性
(100,100)
>>>testclass.data=200                     #通过类对象修改共享属性
>>>x.data,y.data                           #访问共享属性
(200,200)
```

需要注意的是,类对象的属性由所有实例对象共享,该属性的值只能通过类对象来修改。

试图通过实例对象对共享属性赋值时,实质是创建实例对象的私有属性,示例代码如下:

```
>>>testclass.data=200                      #修改共享属性值
>>>x.data,y.data,testclass.data            #此时访问的都是共享属性值
(200,200,200)
```

```
>>>x.data='def'                          #此时为 x 创建私有属性 data
>>>x.data,y.data,testclass.data          #x.data 访问的是 x 的私有属性 data
('def',200,200)
>>>testclass.data=300
>>>x.data,y.data,testclass.data
('def',300,300)
```

2. 属性的私有性

实例对象的私有属性指以“实例对象.属性名＝值”的格式赋值时创建的属性。

“私有”强调属性只属于当前实例对象，对其他实例对象而言是不可见的。

实例对象一开始只拥有继承自类对象的所有属性，没有私有属性。只有在给实例对象的属性赋值后，才会创建相应的私有属性，示例代码如下：

```
>>>x=testclass()                         #创建实例对象
>>>x.pdata                               #试图访问实例对象的属性会出错，属性不存在
Traceback(most recent call last):
File"<stdin>",line 1,in<module>
AttributeError:'testclass' object has no attribute 'pdata'
>>>x.setpdata(123)                       #调用方法为属性赋值
>>>x.pdata                               #赋值后，可以访问属性了
123
```

3. 属性的动态性

Python 总是在第一次给变量赋值时创建变量。

对于类对象或实例对象而言，当给不存在的属性赋值时，Python 会为其创建属性，示例代码如下：

```
>>>testclass.data2='abc'                 #赋值，为类对象添加属性
>>>x.data3=[1,2]                         #赋值，为实例对象添加属性
>>>testclass.data2,x.data2,x.data3       #访问属性
('abc','abc',[1,2])
>>>dir(testclass)                        #查看类对象属性列表
['__class__','__delattr__',…,'data','data2','setpdata','showpdata']
>>>dir(x)
['__class__','__delattr__',…,'data','data2','data3','pdata','setpdata',
'showpdata']
```

可以看到，赋值操作为对象添加了属性。而且，在为类对象添加了属性后，实例对象也自动拥有了该属性。

5.2.2　类和对象的方法

在通过实例对象访问方法时，Python 会创建一个特殊对象——绑定方法对象，也称为实例方法对象。

此时，当前实例对象会作为一个参数传递给实例方法对象。所以，在定义方法时，通常第一个参数的名称为 self。

使用 self 只是惯例，重要的是位置，完全可以用其他名称来代替 self。

通过类对象访问方法时，不会将类对象传递给方法，应按方法定义的形参个数提供参数，这和通过实例对象访问方法有所区别。例如：

```
>>>classtest:
...defadd(a,b):returna+b                    #定义方法,完成加法
...defadd2(self,a,b):returna+b              #定义方法,完成加法
>>>test.add(2,3)                           #5 通过类对象调用方法
>>>test.add2(2,3,4)                        #7 通过类对象调用方法,此时参数 self 的值为 2
>>>x=test()                                 #创建实例对象
>>>x.add(2,3)                               #出错,输出信息显示函数接收到 3 个参数
Traceback(most recent call last):
File"<stdin>",line 1,in<module>
TypeError:add() takes 2 positional arguments but 3 were given
>>>x.add2(2,3)                             #5 通过实例对象完成加法
>>>classtest:pass#
>>>test.__name__
'test'
>>>test.__module__
'__main__'
>>>print(test.__dict__)
{'__module__':'__main__','__dict__':…'test'objects>,'__doc__':None}
>>>test.__base__
<class'object'>
>>>print(test.__doc__)
None
>>>test.__class__
<class'type'>
```

5.2.3 特殊属性和方法

Python 会为类对象添加一系列特殊方法，这些特殊方法在执行特定操作时会被调用。可在定义类时定义这些方法，以取代默认方法，称为方法的重载。类对象常用的特殊方法如下。

- __eq__()：计算 x==y 时调用 x.__eq__(y)。
- __ge__()：计算 x>=y 时调用 x.__ge__(y)。
- __gt__()：计算 x>y 时调用 x.__gt__(y)。
- __le__()：计算 x<=y 时调用 x.__le__(y)。
- __lt__()：计算 x<y 时调用 x.__lt__(y)。
- __ne__()：计算 x!=y 时调用 x.__ne__(y)。
- __format__()：在内置函数 format()和 str.format()方法中格式化对象时调用，返回对象的格式化字符。
- __dir__()：执行 dir(x)时调用 x.__dir__()。
- __delattr__()：执行 del x.data 时调用 x.__delattr__(data)。
- __getattribute__()：访问对象属性时调用。例如，a=x.data 等同于 a=x.__getattribute__(data)。
- __setattr__()：为对象属性赋值时调用。例如，x.data=a 等同于 x.__setattr__(a)。

- __hash__()：调用内置函数 hash(x)时调用 x.__hash__()。
- __new__()：创建类的实例对象时调用。
- __init__()：类的初始化函数。例如，x=test()语句在创建 test 类的实例对象时，首先调用__new__()创建一个新的实例对象，然后调用__init__()执行初始化操作。完成初始化之后再返回实例对象，同时建立变量 x 对实例对象的引用。
- __repr__()：调用内置函数 repr(x)的同时调用 x.__repr__()，返回对象的字符串表示。
- __str__()：通过 str(x)、print(x)以及在 format()中格式化 x 时调用 x.__str__()，返回对象的字符串表示。

5.2.4　伪私有属性和方法

在 Python 中，可以以“类对象.属性名”或“实例对象.属性名”的格式在类的外部访问类的属性。在面向对象技术理论中，这种方式破坏了类的封装特性。Python 提供了一种折中的方法，即使用双下画线作为属性和方法名称的前缀，从而使这些属性和方法不能直接在类的外部使用。以双下画线作为名称前缀的属性和方法称为类的“伪私有”属性和方法，示例代码如下：

```
>>>class test:
...data=100
...__data2=200
...def add(a,b):
...   return a+b
...def __sub(a,b):
...   return a-b
>>>test.data                              #访问普通属性
100
>>>test.add(2,3)                         #访问普通方法
5
>>>test.__data2                          #访问"伪私有"属性，出错，属性不存在
>>>test.__sub(2,3)                       #"伪私有"方法，出错，方法不存在
```

Python 在处理“伪私有”属性和方法名称时，会加上“_类名”作为双下画线前缀的前缀。之所以称为“伪私有”，是指只要使用正确的名称，那么在类的外部也可以访问“伪私有”属性和方法，示例代码如下：

```
>>>test._test__data2                     #访问"伪私有"的属性
200
>>>test._test__sub(2,3)                  #访问"伪私有"的方法
-1
```

可使用 dir()函数查看类对象的“伪私有”属性和方法的真正名称，示例代码如下：

```
>>>dir(test)
['__class__','__delattr__',…,'_test__data2','_test__sub','add','data']
```

5.2.5 静态方法

可使用@staticmethod 将方法声明为静态方法。通过实例对象调用静态方法时，不会像普通方法一样将实例对象本身作为隐含的第一个参数传递给方法。通过类对象和实例对象调用静态方法的效果完全相同，示例代码如下：

```
>>>class test:
...@staticmethod                          #声明下面的 add()为静态方法
...def add(a,b):return a+b
>>>test.add(2,3)                         #通过类对象调用静态方法
5
>>>x=test()                              #创建实例对象
>>>x.add(3,5)                            #通过实例对象调用静态方法
```

5.3 对象初始化

5.3.1 类的构造和初始化

定义一个类，并生成初始化__init__对象函数和__new__对象函数。例如：

```
class A(object):
    def __init__(self, * args,**kwargs):
        print("init%s"%self.__class__)
    def __new__(cls, * args,**kwargs):
        print("new%s"%cls)
        return object.__new__(cls, * args,**kwargs)
a=A()
```

输出结果：

```
new<class'__main__.A'>
init<class'__main__.A'>
```

从结果可以看出，当实例化 A 类时，__new__方法首先被调用，然后是__init__方法。

一般来说，__init__和__new__函数都会有下面的形式：

```
def __init__(self, * args,**kwargs):
#fun c_suite
def __new__(cls, * args,**kwargs):
#fun c_suite
return obj
```

对于__new__和__init__，可以概括为：

__new__方法在 Python 中是真正的构造方法(创建并返回实例)，通过这个方法可以产生一个 cls 对应的实例对象，所以说__new__方法一定要有返回；

对于__init__方法，它是一个初始化的方法，self 代表由类产生出来的实例对象，__init__将

对这个对象进行相应的初始化操作。

前文已经介绍过了__init__的一些行为，包括继承情况中__init__的表现。下面重点介绍__new__方法。

5.3.2　__new__特性

__new__是在新式类中新出现的方法，它有以下行为特性。

- __new__方法是在类实例化对象时第一个调用的方法，返回实例对象。
- __new__方法始终都是类方法（第一个参数为 cls），即使没有被加上装饰器。
- 第一个参数 cls 是当前正在实例化的类，如果要得到当前类的实例，应当在当前类中的__new__方法语句中调用当前类的父类的__new__方法。

对于上面的第三点，如果当前类是直接继承自 object 的，则当前类的__new__方法返回的对象应该为：

```
def __new__(cls, * args,**kwargs):
#fun c_suite
return object.__new__(cls, * args,**kwargs)
```

1. 重写__new__

如果新式类中没有重写__new__方法，则 Python 默认调用该类的直接父类的__new__方法来构造该类的实例，如果该类的父类也没有重写__new__，那么将一直按照同样的规则追溯至 object 的__new__方法，因为 object 是所有新式类的基类。

而如果新式类中重写了__new__方法，则可以选择任意一个其他的新式类（必须是新式类，只有新式类有__new__，因为所有新式类都是从 object 派生的）的__new__方法来创建实例，包括这个新式类的所有前代类和后代类，只要它们不会造成递归死循环，示例代码如下：

```
class Fun(object):
    def __new__(cls, * args, **kwargs):
        obj = object.__new__(cls, * args, **kwargs)
        #这里的 object.__new__(cls, * args, **kwargs) 等价于 super(Fun, cls).__
#new__(cls, * args, **kwargs)
        #object.__new__(Fun, * args, **kwargs)
        #Ny.__new__(cls, * args, **kwargs)
        #person.__new__(cls, * args, **kwargs),即使 person 和 Fun 没有关系，也是允许
#的，因为 person 是从 object 派生的新式类
        #任何新式类不能调用自身的"__new__"来创建实例，因为这会造成死循环
    #所以要避免 return Fun.__new__(cls, * args, **kwargs)或 return cls.__new__
#(cls, * args, **kwargs)
        print("Call __new__for %s" %obj.__class__)
        return obj
class Ny(Fun):
    def __new__(cls, * args, **kwargs):
        obj = object.__new__(cls, * args, **kwargs)
        print("Call __new__for %s" %obj.__class__)
        return obj
class person(object):
    #person 没有__new__方法，会自动调用其父类的__new__方法来创建实例，即会自动调用
```

```
#object.__new__(cls)
    pass
class girl(object):
    def __new__(cls, *args, **kwargs):
        #可以选择用 Bar 来创建实例
        obj = object.__new__(Ny, *args, **kwargs)
        print("Call __new__for %s" %obj.__class__)
        return obj
fun = Fun()
ny = Ny()
girl = girl()
```

输出结果：

```
Call __new__for <class '__main__.Fun'>
Call __new__for <class '__main__.Ny'>
Call __new__for <class '__main__.Ny'>
```

2. __init__的调用

__new__决定是否要使用该类的__init__方法，因为__new__ 可以调用其他类的构造方法，或者直接返回其他类创建的对象来作为本类的实例。

通常来说，新式类开始实例化时，__new__方法会返回 cls(cls 指代当前类)的实例，然后调用该类的__init__方法作为初始化方法，该方法接收这个实例(self)作为自己的第一个参数，然后依次传入__new__方法中接收的位置参数和命名参数。

但是，如果__new__没有返回 cls(当前类)的实例，那么当前类的__init__方法是不会被调用的。看下面的例子：

```
class A(object):
    def __init__(self, *args, **kwargs):
        print("Call __init__from %s" %self.__class__)
    def __new__(cls, *args, **kwargs):
        obj = object.__new__(cls, *args, **kwargs)
        print("Call __new__for %s" %obj.__class__)
        return obj
class B(object):
    def __init__(self, *args, **kwargs):
        print("Call __init__from %s" %self.__class__)
    def __new__(cls, *args, **kwargs):
        obj = object.__new__(A, *args, **kwargs)
        print("Call __new__for %s" %obj.__class__)
        return obj
b = B()
print(type(b))
```

输出结果：

```
Call __new__for <class '__main__.A'>
<class '__main__.A'>
```