第5章

面向对象程序设计

CHAPTER 5

程序设计语言分为面向过程的程序设计语言和面向对象的程序设计语言。Python 既支持面向过程的程序设计，又支持面向对象的程序设计。

视频讲解

5.1 类与对象

“物以类聚,人以群分”,类是一组具有相同属性的事物的集合,如人类、鸟类、鱼类等。对象是类的具体事例。类是抽象的,而对象是具体的,如鲤鱼是鱼类的一个具体对象。类的本质是一组数据及其在这一组数据上的操作的集合。

5.1.1 类的定义

在 Python 中,类的定义如下。

```
class 类名:
    pass
```

其中,类名的命名符合变量的命名规则,不能是关键字,只能由字母、数字和下画线组成,且首字符只能是字母或下画线。

例 5-1 定义一个名称为 point 的类,用于表示二维坐标系下的点。point 类有横坐标和纵坐标两个属性,以及一个表示原点到 point 的距离的方法(In[1])。

```
In[1]:
  class point:
  #属性和方法
    def __init__(self,x1,y1):
      self.x = x1
      self.y = y1
    def distance_zero(self):
      z = (self.x * *2 + self.y * *2) * *0.5
      return z
  point1 = point(4,5)
  print(point1.x,point1.y)
  print(point1.distance_zero())
  print(point.distance_zero(point1)
```

在 point 类中,__init__是一种特殊的方法,在创建类的对象实例时由系统自动运行,不需要显式调用,类似于 C++语言中的构造函数。point 类的__init__方法有三个参数,其中 self 必不可少,且位于其他参数的前面,表示一个指向实例对象本身的引用,有了 self 后就可以让实例访问类中的属性和方法。self.x,self.y 表示类的属性:self 对象的 x 和 y 坐标。方法 distance_zero 用于计算 point 到原点的距离,该方法仅需要一个参数 self。

程序运行结果如下。

```
4 5
6.4031242374328485
6.4031242374328485
```

5.1.2 对象的创建

类是抽象的,而对象是具体的,类似 int32、float64 对于变量类型的规定,而对象则类似 10、10.5 等具体数字。下面将创建 point 类的一个对象并调用其方法。

例 5-2　对象的创建举例(In[2])。

```
In[2]:
  point1 = point(5,10)
  print(point1.x, point1.y)
  #使用对象调用类方法
  point1.distance_zero()
  #使用类调用类方法
  point.distance_zero(point1)
```

在上述定义类 point 的对象 point1 时,有两个参数(5,10)传递给了自动执行的__init__方法,用于创建对象 point1。在创建 point1 对象后,通过对象名.属性就可以直接访问它的属性 x 和 y。有两种调用类的方法,一种是通过对象.方法名(),另一种是使用类名.方法名(对象名),后者需以 self 作为参数。

程序运行结果如下。

```
5,10

Out[2]:
  11.180339887498949
```

5.2　类的封装

面向对象程序有三大思想,即封装、继承和多态。其中封装是将类的属性和方法包装在类中,在类的外部只能按一定的规则访问类属性和方法而不能随意访问,这样就能达到保护类属性和方法的目的。同时通过封装技术,用户不必关心类的内部实现细节,只需要了解调用类方法需要的参数,以及返回的参数,从而提高代码的安全性。

例 5-3　定义一个 circle,用于求出 circle 的周长和面积(In[3]～In[4])。

```
In[3]:
  import math
  class circle:
    def __init__(self,radius):
       self.radius = radius
    def area(self):
       return math.pi * self.radius * * 2
    def perimeter(self):
       return 2 * math.pi * self.radius
```

在上述 circle 类的定义中,需要引入 math 包,才能使用圆周率 math.pi。以下我们定义一个 circle 类的对象 circle1,用于计算圆的周长和面积。

```
In[4]:
  circle1 = circle(10)
  print('圆的面积为:%.2f' %circle1.area())
  print('圆的周长为:%.2f' %circle1.perimeter())
```

程序运行结果如下。

```
圆的面积为:314.16
圆的周长为:62.83
```

在默认情况下,Python 的属性和方法都是 public(公开)的,并没有像其他程序设计语言具有的 protected(保护)、private(私有)类型的属性和方法。Python 类中若要定义 private 类型的属性和方法,可以在属性或者方法前面加上两条下画线"__",则属性和方法定义为私有属性和方法。私有属性和方法只能通过公有方法来进行访问,不能通过类的对象直接访问,起到保护属性和方法的作用。另外,Python 类中若要定义 protected 类型的属性和方法,可以在变量或者方法前面加上一个下画线"_",Python 允许类的对象直接访问 protected 变量和方法。

例 5-4 类的保护变量和私有变量举例(In[5]~In[11])。

```
In[5]:
  class one:
    _x = 20
    __y = 30
    def __init__(self,z):
      self.z = z
    def second():
      print(one._x,one.__y)
    def first(self):
      print('可以通过公有方法访问保护变量%s'%self._x)
      print('可以通过公有方法访问私有变量%s'%self.__y)
      print('可以通过公有方法访问公有变量:%d'%self.z)
```

接着定义类 one 的一个对象 one1。

```
In[6]:
  one1 = one(5)
```

可以通过类名直接访问类的保护变量和保护方法。也可以通过对象名直接访问类的保护变量和保护方法。例如,输入以下程序段:

```
In[7]:
  print(one1._x)
  print(one._x)
```

程序运行结果如下。

```
20
20
```

以下程序段通过对象访问类的保护变量。

```
In[8]:
  print(one1.__y)
```

程序输出报错信息如下。

```
AttributeError:'one' object has no attribute '__y'
```

因此不能通过对象名访问类的保护变量,同样也不能通过类名访问类的保护变量。

```
In[9]:
  print(one.__y)
```

程序输出报错信息如下。

```
AttributeError:type object 'one' has no attribute '__y'
```

但可以通过类的公有方法访问类的公有属性、私有属性和保护属性，以及保护变量和公有变量。例如，在上述程序的基础上加入以下程序，即可使用对象的公有方法访问保护变量、私有变量和公有变量。

```
In[10]:
  one1.first()
```

程序运行结果如下。

```
可以通过公有方法访问保护变量:20
可以通过公有方法访问私有变量:30
可以通过公有方法访问公有变量:5
```

Python 类中的变量访问机制体现了类的封装性，让类和对象能有序访问私有变量、保护变量、公有变量。

可以通过类的公有方法访问类的私有变量和保护变量。

```
In[11]:
  one.second()
```

程序运行结果如下。

```
20 30
```

5.3　类的继承

类的继承可以提高代码的复用性，降低程序设计的复杂度，是面向对象程序设计中最重要的机制之一。Python 类的继承机制与其他高级程序设计语言的继承机制基本相同。我们首先定义一个 bus 类，包括生产厂家、型号、生产日期和行驶里程 4 个属性，其中行驶里程的默认值为 0。定义了 4 个修改属性的方法和 1 个所有属性的方法 print_all。

例 5-5　Python 类的继承(In[12]～In[18])。

```
In[12]:
  class bus:
    #给属性赋默认值
    mileage = 0
    def __init__(self,maker,model,year):
        self.maker = maker
        self.model = model
        self.year = year
    def modify_model(self,a):
        self.model = a
    def modify_maker(self,b):
        self.maker = b
    def modify_year(self,c):
        self.year = c
    def modify_mileage(self,d):
        self.mileage = d
    def print_all(self):
        print("生产厂家:%s,型号:%s,出厂年份:%d,行驶里程:%d" % (self.maker,self.
model,self.year,self.mileage))
```

现在定义一个 bus 类的对象 bus1,验证所定义类的功能。

```
In[13]:
  bus1 = bus('比亚迪','c6',2020)
```

通过对象和类名可以直接访问类的默认变量,示例程序如下。

```
In[14]:
  print(bus1.mileage)
  print(bus.mileage)
```

程序运行结果如下。

```
0
0
```

通过对象可以修改其默认参数,但类的默认参数不变,示例程序如下。

```
In[15]:
  #修改了对象的默认参数
  bus1.mileage = 1000
  print(bus1.mileage)
  print(bus.mileage)
```

程序运行结果如下。

```
1000
0
```

上述运行结果表明对象的默认参数修改了,而类的默认参数没有改变。

通过对象可以调用类的公有函数,示例程序如下。

```
In[16]:
  bus1.print_all()
```

程序运行结果如下。

```
生产厂家:比亚迪,型号:c6,出厂年份:2020,行驶里程:0
```

在上述程序段的基础上,继续修改程序,首先将 bus1 中的 model 属性由“c6”改为“c9”,并输出所有属性。

```
In[17]:
  bus1.modify_model('c9')
  bus1.print_all()
```

程序运行结果如下。

```
生产厂家:比亚迪,型号:c9,出厂年份:2020,行驶里程为:1000
```

通过类的公有方法可以修改行驶里程参数,如将 bus1 中的 mileage 属性由 0 修改为 10000,再输出所有属性。该方法的示例程序如下。

```
In[18]:
  bus1.modify_mileage(10000)
  bus1.print_all()
```

程序运行结果如下。

```
生产厂家:比亚迪,型号:c9,出厂年份:2020,行驶里程为:10000
```

在 bus 类的基础上，定义一个电动汽车类 electric_bus，增加一个属性 volume，用于表示电动汽车配置的动力电池容量。

例 5-6　Python 类的继承(In[19])。

```
In[19]:
  class electric_bus(bus):
      def __init__(self,maker,model,year,volume):
          super().__init__(maker,model,year)
          self.volume = volume
      def print_volume(self):
          print("电池容量:",self.volume)
      def modify_volume(self,a):
          self.volume = a
  #生成 elcetric_bus1 对象
  electric_bus1 = electric_bus('比亚迪','c10',2019,'250kWh')
  #修改 elcetric_bus1 的 volume 属性
  electric_bus1.modify_volume("350kWh")
  #调用父类的 modify_mileage 方法修改 mileage 属性
  electric_bus1.modify_mileage(10000)
  #调用父类的 print_all 方法输出相关属性
  electric_bus1.print_all()
  #调用子类的 print_volume 方法输出 volume 属性
  electric_bus1.print_volume()
```

上述程序段定义一个 electric_bus 类，继承了 bus 类，此时需在类名定义时把所继承的类放在新类名后面，并用圆括号括起来。electric_bus 类的__init__函数重载了父类的__init__方法，在__init__方法中，先调用父类 bus 的__init__方法，并把参数传给父类，注意这里不需要再传参数 self，最后给子类 electric_bus 特有的 volume 属性赋值，从而完成对象的初始化工作。通过类的继承机制，electric_bus 继承 bus 的所有属性和方法，减少了 electric_bus 类的设计工作量。

在 eletric_bus 类的定义中，增加了父类中没有的修改 volume 属性的方法 modify_volume 和打印输出 volume 属性的方法 print_volume。

程序运行结果如下。

```
生产厂家:比亚迪,型号:c10,出厂年份:2019,行驶里程:10000
电池容量: 350kWh
```

5.4　类的多态

类的多态是指类的方法具有同种形态，即使不知道一个对象属于哪个类，仍然可以通过这个对象调用一个方法，所执行的方法会在程序运行过程中根据所引用对象的类型，决定所调用的方法。例如，定义一个 person，具有 name、age 和 sex 三个属性，还有 print_info 和 fun 两个方法；接着定义一个 student 类和 teacher 类，两个类都继承 person 类。三个类都有一个 fun 函数，其中子类 student 和 teacher 重载了父类的同名方法，具体程序段如例 5-7 所示。

例 5-7 Python 类的多态举例(In[20]～In[22])。

```
In[20]:
  class person():
    def __init__(self,name,age,sex):
        self.name = name
        self.age = age
        self.sex = sex
    def print_info(self):
        print(f'姓名:{self.name},年龄:{self.age},性别:{self.sex}')
    def fun():
        print('我是一个人!')
  class student(person):
    def __init__(self,name,age,sex,stu_no):
        super().__init__(name,age,sex)
        self.stu_no = stu_no
    def fun():
        print('我是一个学生!')
    def print_info(self):
        print(f'姓名:{self.name},年龄:{self.age},性别:{self.sex},学号:{self.stu_no}')
  class teacher(person):
    def __init__(self,name,age,sex,tech_no):
        super().__init__(name,age,sex)
        self.tech_no = tech_no
    def print_info(self):
        print(f'姓名:{self.name},年龄:{self.age},性别:{self.sex},工号:{self.tech_no}')
    def fun():
        print('我是一个老师!')
```

这里定义一个方法 fun_all(obj),用于调用类中的方法 fun(),示例程序如下。

```
In[21]:
  def fun_all(obj):
    obj.fun()
```

最后让 obj 分别取 person、student、teacher,示例程序如下。

```
In[22]:
  fun_all(person)
  fun_all(student)
  fun_all(teacher)
```

程序运行结果如下。

```
我是一个人!
我是一个学生!
我是一个老师!
```

5.5 object 类

Python 中 objcet 类是所有类的祖先,任何类如果没有指定父类,都将默认为 objcet 类的子类,因此 Python 定义的所有类都有 object 类的属性。可以通过内置函数 dir 来查看对象的属性。以下首先定义一个 point 类。

例 5-8　object 类举例(In[23])。

```
In[23]:
  class point(object):
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def print_all(self):
        print(f'点的 x 坐标为:{self.x},点的 y 坐标为:{self.y}')
  point1 = point(5,10)
```

在上述程序中,首先定义一个 point 类,其父类为 object,具有两个属性 x、y 及一个方法 print_all,接着定义一个 point 类的对象 point1。在 point1 对象的属性中,除了比较熟悉的 __init__方法用于初始化对象外,下面再介绍几个重要的属性和方法。

1. __str__方法

__str__方法用于返回对对象的描述,可以用 print(__str__())输出对象的信息。

例 5-9　Python 类的__str__方法举例(In[24])。

```
In[24]:
  print(point1.__str__())
```

程序运行结果如下。

```
<__main__.point object at 0x00000000049C4910>
```

程序输出的是 point1 在内存中的地址。

现在重写 point 类的__str__方法,并重写生成 point1 对象。

例 5-10　Python 类方法__str__重载举例(In[25]~In[39])。

```
In[25]:
  class point(object):
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"点的 x 坐标为:{self.x},点的 y 坐标为:{self.y}"
    def print_all(self):
        print(f'点的 x 坐标为:{self.x},点的 y 坐标为:{self.y}')
  point1 = point(5,10)
  point1__str__()
```

程序运行结果如下。

```
'点的 x 坐标为:5,点的 y 坐标为:10'
```

由此可以看出,重写 point 类的__str__方法后不再输出对象的内存地址,而是通过运行重载后的__str__方法,得到最终的输出结果。

2. __dict__方法

当__dict__方法作用于类时,用于返回类的方法的字典;当__dict__方法作用于对象时,用于返回对象属性的字典。在上述程序的基础上,加入以下程序段:

```
In[26]:
  print(point.__dict__)
  print(point1.__dict__)
```

程序运行结果如下。

```
{'__module__': '__main__', '__init__': <function point.__init__ at 0x0000000004A0F9D0>,
'distance_zero':<function point.distance_zero at 0x0000000004A0FB80>, '__dict__':
<attribute '__dict__' of 'point' objects>, '__weakref__': <attribute '__weakref__' of 'point'
objects>, '__doc__': None}
{'x': 5, 'y': 10}。
```

3. __class__方法

__class__方法用于返回对象所属的类。在上述程序的基础上,加入以下程序段:

```
In[27]:
  print(point1.__class__)
```

程序运行结果如下。

```
<class '__main__.point'>
```

运行结果表明 point1 属于 point 类。

4. __bases__方法

__bases__方法用于从下到上返回类的所有父类,在上述程序的基础上,加入以下程序段:

```
In[28]:
  print(point.__bases__)
```

程序运行结果如下。

```
(<class 'object'>,)
```

运行结果表明 point1 仅属于 object 类。

5. __base__方法

__bases__方法用于返回类的直接父类,在上述程序的基础上,加入以下程序段:

```
In[29]:
  print(point.__base__)
```

程序运行结果如下。

```
(<class 'object'>)
```

运行结果表明 point1 的直接父类为 object。

涉及类的继承关系的还有__mro__和__subclass__方法,其中__mro__方法属性用于返回类的层次关系,__subclass__方法用于返回子类信息。

6. __new__方法

__new__方法和__init__方法都是类中的内建方法,这两个方法在实例化对象时会被自动调用。其中__new__方法的调用在__init__方法之前,__new__方法中有一个参数必须是

cls，__init__方法中有一个参数必须是 self。__new__方法的作用有两个：①为实例化对象分配一个空间；②返回这个对象的引用并传递给__init__方法的 self 参数。__init__方法的作用是对这个实例化对象再次加工。

例 5-11　Python 类方法__new__重载举例(In[30]～In[31])。

```
In[30]:
  class car(object):
    def __new__(cls, *args, **kwargs):
        print("创建一个车对象,并给该对象分配空间")
        instance = super().__new__(cls)
        return instance
    def __init__(self,name):
        self.name = name
        print(f"给对象{self.name}一个具体值")
```

接着定义 car 类的一个对象 car1。

```
In[31]:
  car1 = car('一汽奥迪')
```

程序运行结果如下。

```
创建一个车对象,并给该对象分配空间
给对象一汽奥迪一个具体值
```

7. __add__方法

如果要让两个对象相加，则必须重载__add__方法。下述程序定义一个 car 对象，然后重载__add__方法用于实现两个对象相加。

例 5-12　Python 类方法__add__重载举例(In[32])。

```
In[32]:
  class car:
    def __init__(self,model):
        self.model = model
    def __add__(self,self1):
        return self.model + self1.model
  car1 = car('一汽')
  car2 = car('长安')
  car3 = car1 + car2
  print(car3)
```

程序运行结果如下。

```
一汽长安
```

8. __len__方法

__len__方法一般用于计算列表的长度，如果要用于计算对象的长度，则必须重载__len__方法。以下将对上述程序进行修改，用__len__函数实现对对象长度的计算。

例 5-13　Python 类方法__len__重载举例(In[33])。

```
In[33]:
  class car:
```

```
    def __init__(self,model):
        self.model = model
    def __add__(self,self1):
        return self.model + self1.model
    def __len__(self):
      return len(self.model)
car1 = car('一汽')
car2 = car('长安')
car3 = car1 + car2
print(car3)
print(car3.__len__())
```

程序运行结果如下。

```
一汽长安
4
```

5.6 导入和使用模块

5.6.1 自定义模块的定义

在类生成后,就可以使用类所提供的方法了。下面的程序定义一个 bus 类和其子类 electric_bus。

例 5-14 Python 自定义模块举例(In[34]~In[39])。

```
In[34]:
  class bus:
    mileage = 0
    def __init__(self,maker,model,year):
        self.maker = maker
        self.model = model
        self.year = year
    def modify(self,a,b,c,d):
        self.maker = a
        self.model = b
        self.year = c
        self.mileage = d
    def print_mileage(self):
        print('%d' % self.mileage)
    def print_all(self):
        print("%s,%s,%d,%d" % (self.maker,self.model,self.year,self.mileage))
  class electric_bus(bus):
    def __init__(self,maker,model,year,volume):
        super().__init__(maker,model,year)
        self.volume = volume
    def modify(self,a,b,c,d,e):
        self.maker = a
        self.model = b
        self.year = c
        self.mileage = d
        self.volume = e
```

```
    def print_mileage(self):
        print('%d' % self.mileage)
    def print_all(self):
        print(("%s,%s,%d,%d,%s") %(self.maker,self.model,self.year,self.mileage,self.volume))
```

将以上程序以 txt 格式保存到 C:\anaconda3\Lib(此处假设 anaconda3 安装在 C 盘下),并将文件格式改为 Bus.py。这里要说明的是,最好能够先将以上代码复制至 Jupyter Notebook 调试正常后再复制至.txt 文件中,否则如果没有经过调试的程序有错误,则封装成的模块在导入使用时会报错。

1. 从模块中导入类

在 Jupyter 中导入 bus 类,并定义 bus 的一个 bus1 对象,通过 bus1 对象调用 print_all 方法,示例程序如下。

```
In[35]:
  from Bus import bus
  bus1 = bus('金龙','XMQ6129',2021)
  bus1.print_all()
```

程序运行结果如下。

```
金龙,XMQ6129,2021,0
```

2. 从模块中导入多个类

```
In[36]:
  from Bus import bus,electric_bus
  electric_bus1 = electric_bus('比亚迪','c6',2019,'300kWh')
  electric_bus1.print_all()
```

程序运行结果如下。

```
比亚迪,c6,2019,0,300kWh
```

3. 导入整个模块

```
In[37]:
  #仅导入整个模块,不显式导入 bus,electric_bus 类
  import Bus
  electric_bus1 = Bus.electric_bus('比亚迪','c6',2019,'300kWh')#前面加模块名 Bus
  electric_bus1.print_all()
```

程序运行结果如下。

```
比亚迪,c6,2019,0,300kWh
```

4. 从模块中导入所有类

```
In[38]:
  from Bus import *
  electric_bus1 = electric_bus('比亚迪','c6',2019,'300kWh')
  electric_bus1.print_all()
```

程序运行结果如下。

```
比亚迪,c6,2019,0,300kWh
```

5. 使用类的别名

```
In[39]:
  from Bus import electric_bus as e_b
  electric_bus1 = e_b('比亚迪','c6',2019,'300kWh')
  electric_bus1.print_all()
```

程序运行结果如下。

```
比亚迪,c6,2019,0,300kWh
```

5.6.2 导入第三方模块

1. 第三方模块的安装

Python 自带了很多第三方库可供使用,这些模块需要安装才能进行使用。第三方模块的安装指令如下。

```
pip intall 模块名
```

2. 第三方模块的使用

第三方模块的使用指令如下。

```
import 模块名
```

5.6.3 以主程序的方式运行

在模块的定义中都会自动定义一个模块名称的变量__name__,如果这个模块是主模块,则 Python 的解释器会在最高层的__main__模块中运行该模块,即此时模块的__name__变量值为__main__。但有时此模块不是按最高层模块运行的,而是提供给其他模块使用,则该模块中不需要其他模块执行的语句可以用下列语句屏蔽:

```
if __name__ = '__main__'
```

现在建立一个 py 文件(由于 Jupyter 环境下的文件名为.ipynb,因此不能直接在 Jupyter 中建立 py 文件,可以在写字板或者记事本中写程序并命名为以.py 作为文件扩展名),实现计算二维坐标上两个点的距离。

例 5-15 以主程序的方式运行举例(In[40]~In[41])。

```
In[40]:
  def distance(x1,y1,x2,y2):
    return ((x2 - x1) * * 2 + (y2 - y1) * * 2) * * 0.5
  if __name__ = = '__main__':
    distrance1 = distance(2,3,4,5)
    print(distrance1)
```

程序运行结果如下。

```
2.8284271247461903
```

将这段程序命名为 dist.py 并保存到 C:\anaconda3\Lib,再将 dist.py 作为模块导入程序,例如:

```
In[41]:
  from dist import *
  distrance2 = distance(5,4,3,8)
  print(distrance2)
```

程序运行结果如下。

```
4.47213595499958
```

可见程序不再输出 2.8284271247461903。

习题 5

本书提供在线测试习题,扫描下面的二维码,可以获取本章习题。

在线测试