第 5 章

面向对象基础

面向对象编程(Object Oriented Programming,OOP)是现代软件工程领域中的一项重要技术,能很好地支撑软件工程的 3 个主要目标:重用性、灵活性和扩展性。

Python 是完全支持面向对象的动态型语言,其完全支持面向对象的封装、继承、多态等特性。Python 中“一切皆对象”,即一切数据类型都被视为对象。本章将介绍面向对象的基础概念及其在 Python 中的具体体现和应用。

5.1 面向对象概念

在介绍面向对象编程概念之前,先介绍与之相对应的另一种程序设计思想:面向过程编程。面向过程编程的开发思想在早期的开发中被大量应用,所谓过程即步骤,在解决问题时通过各种变量存储需加工的数据,同时对解决问题的步骤进行分析,并将重复的步骤提取后封装在函数中,然后根据解决问题的步骤依次调用相关的函数来达到解决问题的目的,一切行为都以函数为基础。

而面向对象编程思想更接近人类的思维方式。在生活中我们经常接触不同特征的事物,如学生、老师等,这些事物之间也存在着各种各样的联系。使用面向对象思想解决问题时是在需解决的问题中抽取出多种不同类型的事物,在程序中使用类来抽象表达同类的事物,在类中描述出同类事物共同的数据和行为,通过类产生不同的对象来映射现实世界中的具体事物,并将事物数据及对数据操作的方法与对象绑定在一起,通过对象间的关系映射事物之间的联系,通过对象之间的互动达到解决问题的目的。

5.2 类和对象

类是具有相似特征和行为的事务的抽象表达,例如无论是小米手机还是华为手机都具有相似的特征(如外观尺寸、颜色、质量等)和行为(如打电话、拍照、上网等功能),可以抽象表达为手机类;对象则是现实世界中该类事物的具体个体,例如各位读者手中的一个个具体的手机个体。

在面向对象编程设计中首先将现实同类事物抽象映射为类,如上述的手机类,在类中描述同一类事物的共同特征和行为,这些特征(如手机的颜色、屏幕大小等)被称为事物的属性,由于手机都有打电话、拍照等功能行为,所以被称为方法,然后根据类来创建具体的一个个实例对象。

5.2.1 类定义和对象创建

1. 类定义

类的定义,语法格式如下:

```
class 类名[(基类)]:
    类成员定义
```

其中,class 为定义类的关键字,“类名”由用户自定义,须符合标识符的命名要求。一般行业规范要求类名采用大驼峰命名法,即类名中每个单词的首字母都大写,名称中不使用下画线,如 StudentAnswer。“基类”用来声明当前类的直接继承基类,可省略,省略表示直接继承 object 类,关于继承参见本章 5.3.2 节内容。

类成员包括各种属性和方法。在类中属性用变量形式表示,用来存储对象的特征数据;方法用函数形式表示,表达事物的某一项功能。

下面来看一简单的 Person 类的定义,代码如下:

```
class Person:                          #定义类 Person
    count = 1                          #属性: 数量
    def sayHello(self):                #实例方法: 打招呼的功能
        print("你好!")
```

以上代码定义了 Person 类,该类没有定义直接基类,类名后的括号省略不写。该类有数量 count 属性,有实例方法 sayHello(),能够输出“你好!”。

2. 创建对象

对象是类的实例。若 Person 类代表具有共同特征和行为的抽象人类,则对象就是根据 Person 类创建的某个具体的人。

创建对象的语法格式如下:

```
对象变量名 = 类名([参数列表])
```

下面根据 Person 类创建一实例对象,该过程称为类的实例化,代码如下:

```
p1 = Person()                                  #实例化对象并将引用赋值给 p1
print("p1 对象: ",p1)                          #显示对象 p1
print("p1 对象的 count 属性: ",p1.count)       #显示 p1 对象的 count 属性值
p1.sayHello()                                  #调用 p1 对象的 sayHello()方法
```

上述第 1 行代码通过 Person()方法创建一个 Person 的实例对象并将其引用赋值给 p1,第 2 行、第 3 行分别打印显示 p1 引用的对象和该对象的 count 属性值,第 4 行调用 p1

对象的 sayHello()方法。运行结果如图 5-1 所示。

```
p1对象： <__main__.Person object at 0x000002830E620988>
p1对象的count属性： 1
你好！
```

图 5-1　输出 p1 引用对象

第 1 行输出结果中的 0x000002830E620988 表示 p1 引用对象的内存地址，在程序运行期间，每个对象的内存地址都不一样，注意此处地址因个人机器环境的不同会有所不同。

在程序中通过以上语法格式可以继续创建多个 Person 类的实例对象，每个对象都会有 count 属性和 sayHello()方法。Person 类就像建造房屋时设计的图纸，有了这张图纸之后，就可以根据这张图纸创建多个相似的实体房屋对象。

5.2.2　属性


14min

1. 实例属性

假若设计一学生管理程序，在现实中，每名学生都有名字、年龄、分数等特征，这些特征对应的数据属于个体对象自身，这些特征在类中被称为实例属性，有时也简称为属性，对应的数据称为属性值。

定义实例属性的方式有两种：一是在类的初始化方法__init__()中定义实例属性，二是在使用实例对象时动态添加。

在类内部调用实例属性时需要加上 self 前缀，其调用语法格式如下：

```
self.属性名
```

在外部调用时由对象调用，其调用语法格式如下：

```
对象名.属性名
```

(1) 在初始化方法__init__()中添加实例属性。

【例 5-1】 创建一个 Student 类，并增加姓名和年龄两个实例属性，实例化 Student 类进行测试，代码如下：

```
#第 5 章/5 - 1.py
class Student:                                  #定义 Student 类
    def __init__(self, name, age):             #定义初始化方法
        self.name = name                        #定义实例属性 name
        self.age = age                          #定义实例属性 age
    def introduce(self):                        #实例方法
        print(f"我的名字是：{self.name},我今年{self.age}岁")
#主程序
stu = Student("郭靖", 22)                       #实例化对象并赋值给 stu
print("name 属性值：",stu.name)                 #使用对象名访问实例属性 name
print("age 属性值：",stu.age)                   #使用对象名访问实例属性 age
stu.introduce()                                 #使用对象调用 introduce()方法
huangrong = Student("黄蓉", 18)
huangrong.introduce()
```

上述代码运行的结果如图 5-2 所示。

```
name属性值： 郭靖
age属性值： 22
我的名字是：郭靖,我今年22岁
我的名字是：黄蓉,我今年18岁
```

图 5-2 例 5-1 实例属性测试的运行结果

案例代码 Student 类的__init__()方法用于完成实例对象的初始化工作,并在此方法中给该类对象添加 name 和 age 属性,其中的 self 关键字代表当前对象。

主体代码 stu=Student("郭靖",22)对 Student 类进行实例化,在实例化时自动调用__init__()方法,并传入实际参数值"郭靖"和 22,在__init__()方法中使用 self.name=name 给当前对象添加了实例属性 name 并进行初始化赋值操作,实例化完成后将实例对象引用赋值给 stu。

从 print("name 属性值：",stu.name)语句的执行结果可以看出上例代码中使用 stu.name 成功地访问了 stu 对象的实例属性 name 的值。

本案例代码中实例化了两个对象 stu 和 huangrong,通过两个对象调用 introduce()方法时的输出结果可以看出,两个对象各自访问了自己的 name 和 age 属性,即实例属性值属于实例对象本身,同一类的不同对象其属性值之间各自独立,互不影响。

在上述案例的基础上在主体代码后面添加一行代码,尝试由类 Student 调用实例属性 name,代码如下：

```
print(Student.name)      #使用类调用对象属性
```

重新运行程序,运行结果如图 5-3 所示。

```
Traceback (most recent call last):
  File "E:/pythonWork/pythonbookProject/ch5/5-1.py", line 17, in <module>
    print(Student.name) # 使用类调用对象属性
AttributeError: type object 'Student' has no attribute 'name'
```

图 5-3 使用类访问对象属性的运行结果

错误提示说明 Student 没有 name 属性,因为 name 属性是属于具体的实例对象而不属于类,只有实例对象才能调用实例属性。

注意：Python 中系统为每个类提供一个默认的__new__()方法,称为构造方法,主要负责对象的创建,该方法至少有一个参数 cls,代表要实例化的类,此参数在实例化时由 Python 解释器自动提供。__new__()方法有一个返回值,即实例化后的实例对象。

同时,系统为每个类提供了另一个默认方法__init__(),该方法通常用于进行对象的初始化工作,__init__()方法的第 1 个参数为 self 关键字,该关键字表示实例对象本身,该参数在方法被调用时不需要进行传参操作,会将当前调用方法的对象作为参数传递给 self,用户可根据需要重写__init__()方法完成属性添加及初始化工作。

当实例化对象时,系统首先自动调用__new__()方法构建对象,构建完成后自动调用__init__()方法,并对 self 参数进行赋值操作,自动传入__new__()方法返回的实例对象,完成实例对象的构造及初始化工作。由于最常使用__init__()方法给实例对象添加实例属性

及对属性进行数据初始化，因此也将__init__()方法称为构造方法。

(2) 在对象使用中添加实例属性。

Python 语言是动态语言，在对象使用过程中也可以动态地给对象添加成员。

【例 5-2】 创建一个 Car 类，并给该类动态地增加属性 name、speed 和成员方法 setSpeed()，代码如下：

```
#第 5 章/5 - 2.py
import types                          #导入 types 模块
class Car:
    price = 10000                     #定义公有类属性 price
    def __init__(self,color):
        self.color = color            #定义公有实例属性 color
    def setAddress(self):
        self.address = "成都"         #在其他方法中给实例对象添加属性,一般不提倡此方法
#主程序
car1 = Car("blue")
car2 = Car("red")
print(car1.color,car1.price,Car.price)
print(car2.color,car2.price,Car.price)
print("-" * 8)
car1.color = "yellow"                 #修改实例属性值
car1.name = "Geely"                   #动态地为实例对象添加属性 name
print(car1.color,car1.name,car1.price,Car.price)
print(car2.color,car2.price,Car.price)
print("-" * 8)
#定义函数
def setSpeed(self,speed):
    self.speed = speed
car1.setSpeed = types.MethodType(setSpeed,Car)                    #动态地为对象添加方法
car1.setSpeed(50)
print(car1.color,car1.speed,car1.name,car1.price,Car.price)
print(car2.color,car2.speed,car2.price,Car.price)
```

上述代码运行的结果如图 5-4 所示。

```
blue 10000 10000
red 10000 10000
--------
yellow Geely 10000 10000
red 10000 10000
--------
yellow 50 Geely 10000 10000
red 50 10000 10000
```

图 5-4　动态添加属性的运行结果

在案例代码中，在 Car 类的构造方法定义中只定义了实例属性 color，在类外部主程序中通过给对象名.属性名赋值的方式，给 car1 对象添加了一个 name 属性并赋初值为"Geely"。

另外在主程序中定义了一个函数 setSpeed，注意此函数的第 1 个参数为 self 关键字，在函数中通过给 self.属性名赋值的方式给当前对象添加了一个 speed 属性。在通过调用 setSpeed()方法传入实际参数值 50 后，从后两行代码的显示结果可以看出，对象都具有了 speed 属性。

继续添加代码,显示 car2 的 name 属性,代码如下:

```
print(car2.name)
```

重新运行代码程序,会在刚添加的代码行上报错误信息,错误提示如图 5-5 所示。

```
Traceback (most recent call last):
  File "E:/pythonWork/pythonbookProject/ch5/5-2.py", line 27, in <module>
    print(car2.name)
AttributeError: 'Car' object has no attribute 'name'
```

图 5-5 显示 car2 的 name 属性错误的提示图

错误提示 car2 对象没有 name 属性,说明动态添加的属性只属于当前对象所有,同类的其他对象没有此属性,无法进行访问。

注意:对象的实例属性也可以动态地删除,删除命令的格式为 del 对象名.实例属性。

2. 类属性

类属性定义在类的内部,所有方法的外部。类属性定义类的特征,由类的所有对象共有。在类内部用类名.类属性名访问类属性,在类外部通过类或类的实例对象访问类属性,但对类属性值的修改只能由类访问来修改。

【例 5-3】 定义 Student 类,并定义一类属性 number_of_schools 表示在校学生人数,代码如下:

```
#第 5 章/5 - 3.py
class Student:                                  #定义类
   numberOfSchool = 10000                       #定义类属性学校人数

   def __init__(self, name, age):              #构造方法
      self.name = name                          #定义实例属性 name
      self.age = age                            #定义实例属性 age
#主体程序
stu1 = Student("郭靖", 22)                      #实例化对象
stu2 = Student("黄蓉", 20)                      #实例化对象
#使用对象访问实例属性 name、age 和类属性 number_of_schools
print(stu1.name, stu1.age, stu1.numberOfSchool)
print(stu2.name, stu2.age, stu2.numberOfSchool)
print(Student.numberOfSchool)                   #使用类名访问类属性
print("-" * 8)
Student.numberOfSchool += 1                     #使用类对类属性进行修改
print(stu1.name, stu1.age, stu1.numberOfSchool)
Student.numberOfSchool += 1                     #使用类对类属性进行修改
print(stu2.name, stu2.age, stu2.numberOfSchool)
print(Student.numberOfSchool)                   #使用类名访问类属性
```

上述代码运行的结果如图 5-6 所示。

从显示结果可以看出，通过类名和实例对象都可以访问类属性。通过类 Student 对类属性 numberOfSchool 进行修改后，通过实例对象 stu1、stu2 和类 Student 对该属性的访问都为修改之后的数据，说明类的所有对象共有此类属性数据。

```
郭靖 22 10000
黄蓉 20 10000
10000
--------
郭靖 22 10001
黄蓉 20 10002
10002
```

图 5-6　例 5-3 的运行结果

【例 5-4】 测试是否可以由实例对象修改类属性值，代码如下：

```
#第 5 章/5 - 4.py
class Student:                                  #定义类 Student
   numberOfSchool = 10000                       #类属性
   def __init__(self, name, age):              #构造方法
      self.name = name                          #实例属性 name
      self.age = age                            #实例属性 age
#主体程序
stu1 = Student("郭靖", 22)                      #实例化对象
stu1.numberOfSchool += 1                        #尝试使用实例对象对类属性进行修改
#使用对象名访问实例属性和类属性
print(stu1.name, stu1.age, stu1.numberOfSchool, Student.numberOfSchool)
stu2 = Student("黄蓉", 20)                      #实例化对象
stu2.numberOfSchool += 1                        #尝试使用实例对象对类属性进行修改
print(stu2.name, stu2.age, stu2.numberOfSchool, Student.numberOfSchool)
print(Student.numberOfSchool)                   #使用类名访问类属性
```

上述代码运行的结果如图 5-7 所示。

```
郭靖 22 10001 10000
黄蓉 20 10001 10000
10000
```

图 5-7　例 5-4 的运行结果

从显示结果可以看出，通过实例对象试图改变类属性 numberOfSchool 的值并没有成功，通过类 Student 访问 numberOfSchool 的值依然是 10000。从 stu1.numberOfSchool 和 stu2.numberOfSchool 的显示结果都为 10001 可以看出，实例对象 stu1、stu2 各自独立有一个 numberOfSchool 值，其实 stu1.numberOfSchool＋＝1 的含义是为 stu1 对象增加一个新的实例属性 numberOfSchool，此属性覆盖了同名的类属性，此处访问 stu1.numberOfSchool 的值时显示的是实例属性 numberOfSchool 的值，所以对类属性的修改通常由类调用进行修改，不能由实例对象调用修改。

注意：*在编写应用程序时，对类属性进行修改时需要特别谨慎，因为所有的类实例对象都共享类属性，对类属性的修改将影响所有该类的实例；应尽量避免实例属性和类属性使用相同的名字，因为名称相同时，使用实例对象访问时实例属性将屏蔽掉类属性，但是当删除实例属性后，再使用相同的名称，访问的将是类属性。*

Python 内置了对属性进行访问操作的函数。

(1) getattr(object,attribute,default)：从指定的对象返回指定属性的值。object 是必需的，用于访问对象；attribute 是属性的名称；default 可选，当属性不存在时返回指定值。

(2) setattr(object,attribute,value)：指定对象的指定属性的值。前两个参数同上，value 参数是必需的，指定赋予属性的值。

(3) hasattr(object,attribute)：如果指定的对象拥有指定的属性，则函数返回 True，否则返回 False。

(4) delattr(object,attribute)：从指定对象中删除指定属性。

示例代码如下：

```
getattr(stu,'id')          # 返回 stu 对象 id 属性值,若无该属性,则会报异常
setattr(stu,'id',1001)     # 给 stu 对象添加 id 属性,其值为 1001
hasattr(stu,'name')        # 如果 stu 存在 name 属性,则返回值为 True,否则返回值为 False
delattr(stu,'name')        # 删除 stu 对象的 name 属性
```

本节介绍了两种属性：实例属性和类属性，这里对它们的特征进行归纳，具体见表 5-1。

表 5-1 实例属性和类属性的特征

属 性	定义位置	类和实例对象操作	访问方式
实例属性	__init__()方法(常用) 实例对象使用过程中 成员方法中(不建议用)	实例对象可以访问修改	self.属性名
类属性	类中	类和实例对象都可访问 类访问可以修改	类.属性名 对象.属性名

5.2.3 方法

1. 实例方法

实例方法是描述同一类对象的共同行为功能，如每名学生都有介绍自己个人信息的功能，这些共同的行为称为实例方法。

实例方法的定义和函数相同。实例方法在定义时第 1 个参数为 self 关键字，表示对象自身，在调用方法时，可以不用传入此参数值，系统会自动将调用此方法的实例对象作为 self 参数的值传入。

实例方法属于对象，通常通过实例对象来调用，调用命令的格式如下：

```
对象.方法名([参数列表])
```

【例 5-5】 定义 Student 类，添加一个实例方法 showinfo()，用于显示相应的属性，代码如下：

```
# 第 5 章/5-5.py

class Student:                              # 定义类 Student
    number_of_schools = 10000               # 类属性
    def __init__(self, id,name, age):       # 构造方法
        self.name = name                    # 实例属性 name
```

```
        self.age = age                    #实例属性 age
        self.__id = id                    #私有实例属性 id
        Student.number_of_schools += 1    #使用类名修改类属性
    def showInfo(self):                   #公开的实例方法
        print("我的名字是: {name},我今年{age}岁!"\
                        .format(name = self.name,age = self.age))
        print("我们学校共有{0}人.".format(Student.number_of_schools))
    def __showId(self):                   #私有实例方法
        print(f"我的学号是{self.__id}")
    def showAllInfo(self):                #公开的实例方法
        self.__showId()                   #访问内部私有方法
        print(f"姓名是: {self.name},年龄是: {self.age}")
#主程序
stu = Student(1001,"郭靖", 22)            #实例化对象
stu.showInfo()                            #通过实例对象访问实例方法
stu.showAllInfo()
#stu.__showId()                           #错误: 'Student' object has no attribute '__showId'
#print(stu.__id)                          #错误: 'Student' object has no attribute '__id'
```

上述代码运行的结果如图 5-8 所示。

```
我的名字是: 郭靖,我今年22岁!
我们学校共有10001人.
我的学号是1001
姓名是: 郭靖, 年龄是: 22
```

图 5-8　例 5-5 的运行结果

上例代码在 Student 类中定义了实例方法 showInfo()，该方法的功能是格式化显示实例对象自身的 name、age 实例属性值和类属性值。主程序中通过实例化对象 stu 调用实例方法 showInfo()。

实例方法是否可以用类名访问呢？在上例主体代码后加入如下测试代码，代码如下：

```
Student.showInfo()    #通过类名访问实例方法
```

运行代码程序，运行结果如图 5-9 所示。

```
Traceback (most recent call last):
  File "E:/pythonWork/pythonbookProject/ch5/5-5.py", line 26, in <module>
    Student.showInfo() # 通过类名访问实例方法
TypeError: showInfo() missing 1 required positional argument: 'self'
```

图 5-9　测试用类访问实例方法的运行结果

错误提示说明在调用方法 showInfo()时缺少参数 self 实参值，即缺少对象，因此不能用类直接调用对象方法。如果在调用时传入 self 参数呢？保持 Student 类定义不变，测试如下代码：

```
stu1 = Student(1002,"黄蓉", 18)          #实例化对象
Student.showInfo(stu1)                   #通过类名访问实例方法,传入对象参数
```

测试代码运行的结果如图 5-10 所示。

```
我的名字是：黄蓉,我今年18岁!
我们学校共有10002人.
```

图 5-10　测试用类访问实例方法的运行结果

从运行结果可以看出,类也可以调用实例方法,只是需要在调用时传入该类的实例对象,一般不建议使用此种方式调用实例方法。

注意：案例 5-5 代码中有两个下画线开头的属性或者方法是类的私有成员,私有成员在类的外部是不能直接访问的,需要通过特殊方式访问,一般在类的内部访问使用,参见案例中注释有错误的主体语句。没有下画线开头的属性或者方法称为公开的成员,在类的外部可以正常访问。更多相关内容参见本章 5.3.1 节封装性。

2. 类方法

类方法是在类中定义的使用装饰器@classmethod 修饰的方法。一般情况下,第 1 个参数命名为 cls,该参数在方法被调用时不用传入数据,系统会自动将调用该方法的类作为参数值传入。

类方法内不可以访问实例属性,但可以访问类属性,访问时用 cls.类属性或者类名.类属性方式访问。

类方法的调用方式有两种。

(1) 用类名调用,调用命令的格式如下：

```
类名.类方法名([参数列表])
```

(2) 用实例对象调用,调用的命令格式如下：

```
对象.类方法名([参数列表])
```

【例 5-6】 在 Student 类中定义一个类方法 class_showInfo(),代码如下：

```
#第 5 章/5 - 6.py
class Student:                                  #定义类
    number_of_schools = 10000                   #类属性
    def __init__(self, name, age):              #构造方法
        self.name = name                        #实例属性 name
        self.age = age                          #实例属性 age
        Student.number_of_schools += 1          #使用类名修改类属性
    def showInfo(self):                         #实例方法
        print(f"我的名字是: {self.name},今年{self.age}岁!")
        print(f"我们学校共有{Student.number_of_schools}人.")
    @classmethod
    def class_showInfo(cls):                    #类方法
        cls.number_of_schools += 1
        print("我们学校共有{0}人.".format(cls.number_of_schools))
#主体程序
stu = Student("郭靖", 22)                       #实例化对象
stu.showInfo()                                  #使用实例对象调用实例方法
print("-" * 8)
```

```
stu.class_showInfo()                    #使用实例对象调用类方法
Student.class_showInfo()                #使用类调用类方法
```

上述代码运行的结果如图 5-11 所示。

上述代码在 Student 类中定义了一个使用装饰器@classmethod 修饰的类方法 class_showInfo(),在该类方法中使用 cls. number_of_schools 方式访问类属性,分别用实例对象和类名调用该类方法,从运行结果看都能正确调用该类方法的功能。

```
我的名字是：郭靖,今年22岁!
我们学校共有10001人.
--------
我们学校共有10002人.
我们学校共有10003人.
```

图 5-11　例 5-6 的运行结果

注意：在 Python 中,cls 代表类本身,self 代表类的一个实例对象。

3. 静态方法

静态方法是在类中定义的使用装饰器@staticmethod 修饰的方法。该方法没有 self、cls 这样的特殊参数,只有普遍参数列表,与类没有很强的联系,更适合进行一些与类内数据关联性不强的操作,如在一些工具类中使用该方法,只需对方法传入的参数进行操作。

静态方法可以通过类名或实例对象调用。在静态方法中不能直接访问属于实例对象的实例属性或实例方法,只能访问属于类的类属性或类方法,访问类成员时需用类名访问。

【例 5-7】 在 Student 类中定义一个静态方法 static_showInfo(),代码如下：

```
#第 5 章/5 - 7.py
class Student:                                  #定义类
    numberOfSchool = 10000                      #定义类属性
    def __init__(self, name, age):             #构造方法
        self.name = name                        #定义实例属性 name
        self.age = age                          #定义实例属性 age
        Student.numberOfSchool += 1             #使用类名修改类属性
    def showInfo(self):                         #定义实例方法
        print(f"我的名字是: {self.name},今年{self.age}岁!")
        print(f"学校共有{Student.numberOfSchool}人.")
    @classmethod
    def class_showInfo(cls):                    #定义类方法
        print(f"学校共有{cls.numberOfSchool}人.")
    @staticmethod
    def static_showInfo(teststr):               #定义静态方法
        Student.numberOfSchool += 1             #使用类名访问类属性
        print(f"由{teststr}调用: 学校共有{Student.numberOfSchool}人.")
#主体程序
stu = Student("郭靖", 22)                       #实例化对象
stu.showInfo()
print("-" * 8)
stu.static_showInfo("实例对象")                 #使用实例对象调用静态方法
Student.static_showInfo("类")                   #使用类调用静态方法
```

上述代码运行的结果如图 5-12 所示。

```
我的名字是：郭靖,今年22岁!
学校共有10001人。
--------
由实例对象调用：学校共有10002人。
由类调用：学校共有10003人。
```

图 5-12　例 5-7 的运行结果

案例代码在 Student 类中定义了一个静态方法 static_showInfo(),该方法有一个普通参数 teststr,该方法内部使用类名.类属性访问类属性 numberOfSchool。在主体程序中分别使用实例对象 stu 和类名 Student 调用 static_showInfo()静态方法,从运行结果看均能成功调用。

本节介绍了 3 种方法：实例方法、类方法和静态方法,这里对它们的特征进行归纳,具体见表 5-2。

表 5-2　实例方法、类方法和静态方法的特征

方法	特 殊 参 数	装饰器	可被调用者	方法内部可访问的成员
实例方法	第 1 个参数为 self,表示实例对象本身	无	实例对象可以直接调用类,调用时需传入实例对象参数	实例属性、实例方法、类属性、类方法、静态方法
类方法	第 1 个参数为类,一般命名为 cls	@classmethod	类和实例对象都可调用	类属性、类方法、静态方法
静态方法	无	@staticmethod	类和实例对象都可调用	类属性、类方法、静态方法

5.3　面向对象三大特征

5.3.1　封装性


4min

封装是面向对象的一个重要原则,其包含着两层含义：

(1) 将与对象相关的数据和对数据操作的方法抽象表达在类中,即体现了封装的概念。

(2) 封装在类中的属性或方法在外部是可以随意访问的,如何保证类内部成员的安全性,即希望将类内部的某些细节隐藏在类里,只对外公开需要公开的类内部成员。例如不希望实例对象的某些属性数据被外界随意访问和修改,某些方法只是用于内部数据处理,不希望被外部随意调用,即将类内部的某些细节隐藏在类里,只对外公开需要公开的类内部成员,也达到了隔离复杂度的目的。

【例 5-8】 公开成员访问测试,代码如下：

```
#第 5 章/5 - 8.py
class Student:                                  #定义类
    def __init__(self, name, age):             #构造方法
        self.name = name                        #定义实例属性 name
        self.age = age                          #定义实例属性 age
    def innerfun(self):
        print(f"我的名字是: {self.name},今年{self.age}岁!")
#主体程序
stu = Student("郭靖", 22)                       #实例化对象
stu.innerfun()                                  #使用实例对象调用实例方法
print("-" * 8)
```

```
stu.age = 10000                    #在类外部随意修改实例属性 age 的值
stu.innerfun()                     #使用实例对象调用实例方法
```

上述代码运行的结果如图 5-13 所示。

```
我的名字是：郭靖,今年22岁!
--------
我的名字是：郭靖,今年10000岁!
```

图 5-13　例 5-8 的运行结果

上述代码在类 Student 中定义了两个实例属性 name 和 age，在类外部的主体程序中通过 Student 类的实例化对象 stu 将实例属性 age 的值修改为 10000，age 的值 10000 明显是一个不合理的数据，但依然传入了对象的内部，对于对象的内部数据来讲存在着一定的安全隐患。如何实现对类内部成员访问的可控性，提高数据的安全性？

Python 中一般的处理方式是把类内部的某些属性和方法隐藏起来，即将这些成员定义成私有的，采用的方式是在准备私有化的属性或方法名字的前面加两个下画线。

若需要对私有成员进行访问，则可添加一个公开方法，在方法中添加对属性数据或方法的访问代码，以便达到过滤控制的目的。

【例 5-9】 私有成员访问测试，代码如下：

```
#第 5 章/5 - 9.py
class Student:                          #定义类
    def __init__(self, name, age):     #构造方法
        self.__name = name              #定义私有实例属性 name
        self.__age = age                #定义私有实例属性 age
    def set_age(self, age):             #定义公开的实例方法,实现对实例属性 age 访问的控制
        if 18 < age <= 25:              #如果年龄值在 18～25 的合理数据,则更新属性值
            self.__age = age
        else:                           #如果年龄值是不合理数据,则将默认值设置为 20
            self.__age = 20
    def innerfun(self):
        print(f"公开方法: 名字是: {self.__name},今年{self.__age}岁!")
    def __privatefun(self):
        print(f"私有方法: 名字是: {self.__name},今年{self.__age}岁!")
#主体程序
stu1 = Student("郭靖", 22)              #实例化对象
stu1.innerfun()                         #使用实例对象调用实例方法
stu1.__age = 20                         #在外部修改私有实例属性 age 的值
stu1.innerfun()                         #使用实例对象调用实例方法
#stu1.__privatefun()                    #错误: 'Student' object has no attribute '__privatefun'
print("-" * 8)
stu2 = Student("黄蓉", 22)              #实例化对象并赋值给 stu1
stu2.innerfun()                         #使用实例对象调用实例方法
stu2.set_age(10000)                     #通过公开的方法实现对属性 age 值的修改控制
stu2.innerfun()                         #使用实例对象调用实例方法
#stu2.__privatefun()                    #错误: 'Student' object has no attribute '__privatefun'
```

上述代码运行的结果如图 5-14 所示。

```
公开方法：名字是：郭靖,今年22岁!
公开方法：名字是：郭靖,今年22岁!
--------
公开方法：名字是：黄蓉,今年22岁!
公开方法：名字是：黄蓉,今年20岁!
```

图 5-14　例 5-9 的运行结果

从运行结果可以看出，在将 name、age 实例属性设置为私有属性后，在类外部试图将 stu1 对象的 age 值修改为 20 时没有达到目的，stu1 对象的 age 属性值依然为 22，尝试访问私有方法时也会报错'Student' object has no attribute '__privatefun'。

如果希望实现在外部对私有属性 age 的值进行修改，同时又希望对这个修改操作具有一定的控制性，则可在上述代码中定义一个公开的方法 set_age()，以便实现对 age 属性值修改的控制，在此方法中对传入的 age 值做一判断控制，在合理范围内将 age 值赋值给相应私有实例属性 age，若不符合要求，则将私有实例属性 age 值设置为默认值 20。从 stu2 对象调用 set_age()方法后的数据结果可以看出，程序将 age 属性的值设置为默认值 20，不合理数据得到了一定的控制。

在业务设计时，除非有公开的必要，通常实例属性被设计为私有的，然后通过设置公开的方法实现对这些实例属性的访问和控制，具体需要根据实际业务场景加以实践。

注意：Python 语言为动态语言，不存在严格的私有成员，类中的私有成员在对象的外部可以通过“对象名._类名__成员名”进行访问。


14min

5.3.2　继承性

在现实世界中，事物与事物之间存在着多种关系，从属关系就是其中的一种，如交通工具可以分为车、飞机等，车又分为小轿车、公交车、卡车等，飞机又分为民航飞机、战斗机等，通过一张图来描述交通工具事物之间的层次关系，如图 5-15 所示。

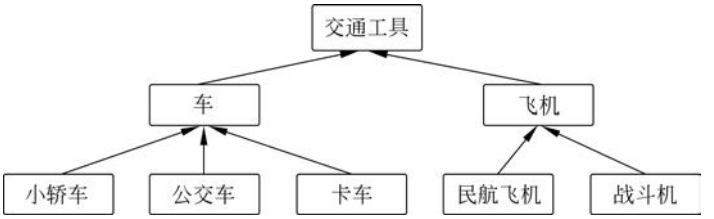

图 5-15　交通工具的层次关系

从图 5-15 可以看出，从交通工具到车再到小轿车，经历了 3 个层级，底层的具体交通工具具有高层交通工具的特征和行为，同时也可以增加高层交通工具不具备的特征和行为。如交通工具都具有品牌、最高时速等属性，具有加速、停止的行为，对于小轿车来讲，它在拥有交通工具的属性和行为的基础上，还可以增加自己特殊的属性和行为，如载客量和播放视频功能等。从高层级交通工具到低层级交通工具，事物越来越具体，从低层级交通工具到高层级交通工具，事物越来越抽象。这是继承在现实世界的直观体现。

在程序设计中，继承既可表达类与类之间的关系，也是一种创建新类的方式，是面向对象编程的另一个重要功能，它可以在原有类的基础上方便地创建新的类，新类既可以拥有原

有类的功能，同时又可以增加自己新的功能或者改写原有功能的实现方式，实现对原有类功能的扩展。继承实现了代码的重用，提高了开发的效率和扩展性，为后期代码的维护提供了便利。

1. 继承实现

Python 中一个类可以继承自一个类或者多个类，新建的类被称为子类或派生类，被继承的类被称为父类或基类、超类。若一个类只继承自一个基类，则称为单继承，若一个类继承自多个基类，则称为多继承。

Python 中继承关系的语法格式如下：

```
class 子类(基类 1,基类 2…,基类 n):
    子类成员
```

在继承关系中存在以下特点：

(1) 子类可以继承基类公开的成员，子类不能继承基类的私有属性和私有方法，也不能在子类中直接访问。

(2) 如果多个基类中具有同名成员，则子类在继承基类时默认按继承顺序表从左向右搜索，执行第 1 个搜索到的基类中的同名成员。若有需要，则子类中可以指定使用某个父类的成员以覆盖默认继承顺序。

【例 5-10】 单继承关系示例，代码如下：

```
#第 5 章/5 - 10.py
class Car():                                #定义父类
    def __init__(self, brand, speed):
        self.brand = brand                  #定义公开品牌属性
        self.speed = speed                  #定义公开速度属性
    def showInfo(self):                     #定义一个公开方法
        print(f"{self.brand}车正以{self.speed}km/h 的速度在行驶!")
    def __privateFun(self):                 #定义一个私有方法
        print("私有方法")

class Sedan(Car):                           #定义一个轿车类 Sedan 继承自 Car 类
    def childShowInfo(self):                #定义子类方法
        print(f"{self.brand}车正以{self.speed}km/h 的速度在行驶!")
#主体程序
car = Car("吉利", 100)
car.showInfo()
print("-" * 8)
sedan = Sedan("五菱宏光",80)
print("子类 brand 属性:",sedan.brand)      #子类继承基类公开属性
sedan.showInfo()                            #子类继承基类公开方法
sedan.childShowInfo()                       #子类自有的方法
#尝试调用私有方法出错: 'Sedan' object has no attribute '__privateFun'
#sedan.__privateFun()
#错误: 'Sedan' object has no attribute '__privateFun'
#sedan._Sedan.__privateFun()
```

运行程序,观察继承的特点。代码运行的结果如图 5-16 所示。

```
吉利车正以100km/h的速度在行驶!
--------
子类brand属性:  五菱宏光
五菱宏光车正以80km/h的速度在行驶!
五菱宏光车正以80km/h的速度在行驶!
```

图 5-16　例 5-10 的运行结果

案例代码中定义了类 Sedan 继承自 Car 类,在这个继承关系中,Car 类被称为父类,Sedan 类被称为子类。在定义 Car 类时没有列出其父类名字,默认继承自 object 类。

从运行结果中可以看出,子类 Sedan 由于继承自 Car 类,所以拥有了父类 Car 中公开的实例属性 brand 和 showInfo()方法,同时子类也扩展了自己的方法 childShowInfo(self)。

主体代码最后两句尝试用不同的方式访问私有方法,运行时会报错误'Sedan' object has no attribute '__privateFun',因为子类不能继承父类的私有成员。

【例 5-11】 多继承关系示例,创建类 C,并让它继承自基类 A、B,代码如下:

```
#第 5 章/5 - 11.py

class A(): #定义父类 A
    def afun(self):
        print("A 类方法")
    def show(self):
        print("A 类 show()方法")
class B(): #定义父类 B
    def bfun(self):
        print("B 类方法")
    def show(self):
        print("B 类 show()方法")
class C(A, B):                          #定义子类 C,继承基类 A 和 B
    pass
#主体程序
c = C()
c.afun()
c.bfun()
c.show()
```

上述代码运行的结果如图 5-17 所示。

```
A类方法
B类方法
A类show()方法
```

图 5-17　例 5-11 的运行结果

案例代码中子类 C 继承基类 A、B,类体中没有增加任何定义。运行结果显示,子类 C 继承了两个基类中的公开方法 afun()和 bfun(),即子类对象继承了多个基类中可以继承的方法和属性。

案例代码中基类 A 和基类 B 有一个同名方法 show(),那么子类继承时会继承哪个基类的方法呢?从案例代码的运行结果来看,此处子类 C 继承的是基类 A 中的 show()方法。将案例中子类 C 定义时的基类列表顺序调整一下,其他代码保持不变,调整后子类 C 的代码如下:

```
class C(B, A):                          #定义子类 C,继承基类 B 和 A
    pass
```

重新运行案例代码,运行结果如图 5-18 所示。

运行结果显示此处子类 C 继承的是基类 B 中的 show()方法。Python 中当多个基类都有一个同名方法时,子类在继承基类方法时默认按继承顺序表从左向右搜索,执行第 1 个搜索到的基类中的同名方法。

```
A类方法
B类方法
B类show()方法
```

图 5-18　例 5-11 修改继承顺序后的运行结果

【例 5-12】 指定继承成员示例,代码如下:

```
#第 5 章/5 - 12.py
class A():                              #定义基类 A
    value1 = "A value1"
    value2 = "A value2"
    def afun(self):
        print("A 类方法")
    def show(self):
        print("A 类 show()方法")
class B():                              #定义父类 B
    value1 = "B Value1"
    value2 = "B value2"
    def bfun(self):
        print("B 类方法")
    def show(self):
        print("B 类 show()方法")
class C(A, B):                          #定义子类 C,继承基类 B 和 A
    value2 = B.value2                   #显式指定继承某个基类的属性
    show = B.show                       #显式指定继承某个基类的方法
    def cfun(self):                     #C 类扩展功能,增加方法
        print("C 类独有的方法")
#主体程序
c = C()
c.afun()
c.bfun()
c.show()
print("c 的 value1 属性: ",c.value1)
print("c 的 value2 属性: ",c.value2)
c.cfun()                                #调用 C 类独有的方法
```

```
A类方法
B类方法
B类show()方法
c的value1属性:  A value1
c的value2属性:  B value2
C类独有的方法
```

图 5-19　例 5-12 的运行结果

上述代码运行的结果如图 5-19 所示。

案例代码中基类 A 和 B 中存在同名的属性 value1、value2 和同名的方法 show()。在子类 C 定义中使用 value2=B.value2 语句显式指定继承基类 B 中的 value2,使用 show=B.show 语句显式指定继承基类 B 中的 show()

方法。

运行结果显示,由于子类 C 中未显式指定 value1 属性的继承情况,所以按继承列表顺序继承自基类 A,由于 value2 显式指定继承基类 B,所以忽略继承顺序,按指定基类继承。同样地,show()方法也是如此。

通过继承关系,子类既具有父类的可继承功能,同时又可以扩展自身的功能,实现了代码的可扩展性。

2. 方法重写

子类可以继承基类能够继承的属性和方法。在某些情况下,子类需要对继承来的方法根据实际应用,对其进行修改,此时需要对子类中继承来的方法进行重写(Overriding)。

子类重写基类方法的方式是在子类中定义一个和父类需重写的方法的同名方法,要求参数也必须相同。

子类若重写基类的方法,子类所创建的实例对象在调用该方法时,优先调用子类重写后的方法,基类中的同名方法被覆盖隐藏。

【例 5-13】 方法重写示例,代码如下:

```
#第 5 章/5 - 13.py

class Animal():                          #定义父类 Animal
    def shout(self):
        print("动物叫")

class Dog(Animal):                       #定义子类 Dog,继承自 Animal
    def shout(self):                     #重写基类 shout()方法
        print("狗汪汪地叫")
#主体程序
animal = Animal()
dog = Dog()
animal.shout()                           #基类调用 shout()方法
print(" - " * 8)
dog.shout()                              #子类调用 shout()方法
```

上述代码运行的结果如图 5-20 所示。

```
动物叫
--------
狗汪汪地叫
```

图 5-20 例 5-13 的运行结果

例 5-13 代码中,父类 Animal 有一个动物叫的方法 shout(),子类 Dog 也属于动物,也有 shout()方法叫的功能,但狗叫的方式和动物叫的实现方式不同,所以在子类 Dog 中将自基类继承的 shout()方法进行重写,重新定义 shout()方法并重写实现代码。运行结果显示子类实例对象 c 在调用 shout()方法时,调用的是子类中重写的 shout()方法。

3. 魔术方法

Python 中所有的类都直接或间接继承自 object 类,object 类是所有类的直接或间接父类,在 object 类中有一类以两个下画线开头、两个下画线结尾的方法,如前面曾提过的

__new__()方法和__init__()等，这些方法被称为“魔术方法(Magic Methods)”，在 object 类的直接子类或间接子类中对这些方法进行重写，可以给这些子类增加特殊功能。

1) __str__()方法

此方法是用来描述对象信息的方法，当在进行项目开发时，为了便于记录日志信息，希望对象能输出自己的字符性描述信息，此时就可以在定义类时重写__str__()方法，如果没有重写该方法，则该方法会默认返回对象的类名、内存地址等信息，如本章 5.2.1 节中图 5-1 输出对象时输出的信息即为对象的类名和内存地址。

【例 5-14】 重写__str__()方法示例，代码如下：

```
#第 5 章/5 - 14.py
class Student:                                  #定义类
    def __init__(self, name, age):             #构造方法
        self.__name = name                      #定义私有实例属性 name
        self.__age = age                        #定义私有实例属性 age
    def __str__(self):              #重写__str__()方法,返回当前对象自定义的字符串描述信息
            return "我的名字是{0},今年{1}岁".format(self.__name, self.__age)
#主程序
stu = Student("郭靖", 22)                       #实例化对象并赋值给 stu
print(stu)                                      #输出对象时,自动调用__str__()方法
```

上述代码运行的结果如图 5-21 所示。

```
我的名字是郭靖，今年22岁
```

图 5-21　例 5-14 的运行结果

在上述代码中，在 Student 类中重写了__str__()方法，返回一个自定义字符串，当使用 print()方法输出对象时，自动调用__str__()方法，打印该方法返回的字符串。

2) __eq__()方法

定义一 Student 类，包含两个属性 name 和 age，代码如下：

```
class Student:                              #定义类 Student
    def __init__(self, name, age):         #定义初始化方法
        self.__name = name                  #定义私有实例属性 name
        self.__age = age                    #定义私有实例属性 age
```

创建两个 Student 类的实例对象，代码如下：

```
stu1 = Student("郭靖", 22)              #实例化对象并赋值给 stu1
stu2 = Student("郭靖", 22)              #实例化对象并赋值给 stu2
```

当使用比较运算符“==”比较 stu1 和 stu2 时，代码如下：

```
print(stu1 == stu2)                     #返回值为 False
```

判断两个对象是否相等,返回的结果为 False,因为 stu1 和 stu2 所引用的对象虽然名字和年龄相同,但在内存中属于两个不同的对象。

stu1 和 stu2 对象的名字和年龄都相同,如果希望在判断是否相等时返回的结果为 True,则需要对两个对象是否相等的含义进行重新定义,需要重写__eq__()方法。

【例 5-15】 重写__eq__()方法示例,代码如下:

```
#第 5 章/5 - 15.py
class Student:                                  #定义类 Student
    def __init__(self, name, age):             #构造方法
        self.__name = name                      #定义私有实例属性 name
        self.__age = age                        #定义私有实例属性 age
    #重写__eq__()方法,定义当前对象和 other 对象相等的规则
    def __eq__(self, other):
        #当当前对象和 other 对象名字相同时即认为这两个对象相等,返回值为 True
        if self.__name == other.__name:
            return True
        return False
#主程序
stu1 = Student("郭靖", 22)                      #实例化对象并赋值给 stu1
stu2 = Student("郭靖", 25)                      #实例化对象并赋值给 stu2
stu3 = Student("黄蓉", 22)                      #实例化对象并赋值给 stu3
print("stu1 和 stu2 是否相等: ",stu1 == stu2)   #输出对象用 == 运算符进行是否相等判别的结果
print("stu1 和 stu3 是否相等: ",stu1 == stu3)   #输出对象用 == 运算符进行是否相等判别的结果
```

上述代码运行的结果如图 5-22 所示。

```
stu1和stu2是否相等:  True
stu1和stu3是否相等:  False
```

图 5-22 例 5-15 的运行结果

从运行结果可以看出 stu1 和 stu2 对象符合__eq__()方法中对象相等的判断逻辑,即两个对象的 name 属性值相等,所以用“==”运算符判定二者是否相等时的结果为 True,而 stu1 和 stu3 不符合__eq__()方法中相等的判定规则,即两个对象的 name 属性值不相等,所以判定二者是否相等时的结果为 False。

3) 运算符重载

在 Python 中,也可以通过重写一些魔术方法实现将自定义的实例对象像内建对象一样进行运算符操作,并能够对自定义对象进行新的运算符规则定义,实现运算符的重载。以下列出了部分运算符重载的方法,具体见表 5-3。

表 5-3 运算符重载方法(部分)

运 算 符	对应的魔术方法	说 明
+	__add__(self,other)	对象加法运算
-	__sub__(self,other)	对象减法运算
*	__mul__(self,other)	对象乘法运算
/	__truediv__(self,other)	对象除法运算
//	__floordiv__(self,other)	对象地板除法
%	__mod__(self,other)	对象求余
**	__pow__(self,other)	对象次方

【例 5-16】 定义一 MyCalculator 类，并实现对其实例对象进行运算符“+”“-”“*”“/”运算重载定义，代码如下：

```
#第 5 章/5 - 16.py
class MyCalculator:                          #创建一个自定义的计算器类
    def __init__(self,data):                #构造方法
        self.data = data
    def __add__(self, other):                #重写__add__()方法,对 + 运算符进行重载
        return self.data + other.data
    def __sub__(self, other):                #重写__sub__()方法,对 - 运算符进行重载
        return self.data  -  other.data
    def __mul__(self, other):                #重写__mul__()方法,对 * 运算符进行重载
        return self.data * other.data
    def __truediv__(self, other):            #重写__truediv__()方法,对/运算符进行重载
        return self.data/other.data
#主程序
calculator1 = MyCalculator(20)               #定义一计算器 calculator1
calculator2 = MyCalculator(10)               #定义一计算器 calculator2
print(calculator1 + calculator2)             #输出两个 MyCalculator 对象的加法操作结果
print(calculator1 - calculator2)             #输出两个 MyCalculator 对象的减法操作结果
print(calculator1 * calculator2)             #输出两个 MyCalculator 对象的乘法操作结果
print(calculator1/calculator2)               #输出两个 MyCalculator 对象的除法操作结果
```

上述代码运行的结果如图 5-23 所示。

```
30
10
200
2.0
```

图 5-23　例 5-16 的运行结果

案例代码中，在 MyCalculator 类中重写了__add__()方法、sub__()方法、mul__()方法和__truediv__()方法，分别对应当对 MyCalculator 类的实例对象使用运算符“+”“-”“*”“/”进行运算时，对应的自定义运算规则，读者可根据结果一一分析验证。

4. super()函数

子类重写父类的方法后，若需要在子类中访问父类被覆盖隐藏的同名方法，则可以使用 super()函数。

【例 5-17】 super()函数的应用，代码如下：

```
#第 5 章/5 - 17.py

class Animal():                              #定义父类 Animal
    def shout(self):
        print("动物叫")

class Dog(Animal):                           #定义子类 Dog,继承自 Animal
    def shout(self):                         #重写基类 shout()方法
        print("狗汪汪地叫")
        super().shout()                      #使用 super()函数调用父类方法
```

```
#主体程序
animal = Animal()
dog = Dog()
animal.shout()                          #基类调用 shout()方法
print("-" * 8)
dog.shout()                             #子类调用 shout()方法
```

上述代码运行的结果如图 5-24 所示。

```
动物叫
--------
狗汪汪地叫
动物叫
```

图 5-24　例 5-17 的运行结果

运行结果显示,子类实例 dog 在调用 shout()方法时,使用 super()函数实现调用父类 shout()方法,使子类不仅具有父类 shout()方法的功能,同时根据需要增添自己的功能。

5.3.3　多态性

多态指同一类事物能呈现出多种形态,是面向对象编程的核心特征。

在第 2 章我们学习了列表、字符串等各种内置类型,实际上当定义一个类时也是定义一种新的数据类型。

【例 5-18】 对象类型判断示例,代码如下:

```
#第 5 章/5-18.py
class Animal():                         #定义父类
    def shout(self):
        print("动物叫")
class Dog(Animal):                      #定义子类
    def shout(self):                    #重写父类 shout()方法
        print("狗汪汪地叫")
#主体程序
animal = Animal()                       #animal 引用 Animal 类的实例对象
dog = Dog()                             #dog 引用 Dog 类的实例对象
#测试变量 animal 是否是 Animal 类型
print("animal 是否是 Animal 类型:",isinstance(animal,Animal))
print("dog 是否是 Dog 类型:",isinstance(dog,Dog))          #测试变量 dog 是否是 Dog 类型
print("-" * 8)
print("dog 是否是 Animal 类型:",isinstance(dog,Animal))    #测试 dog 是否是 Animal 类型
print("animal 是否是 Dog 类型:",isinstance(animal,Dog))    #测试 animal 是否是 Dog 类型
```

```
animal是否是Animal类型: True
dog是否是Dog类型: True
--------
dog是否是Animal类型: True
animal是否是Dog类型: False
```

图 5-25　例 5-18 的运行结果

上述代码运行的结果如图 5-25 所示。

isinstance()函数用于测试一个变量是否是某种类型。从运行结果可以看出 animal 变量是 Animal 类型,dog 变量是 Dog 类型。当测试变量 dog 是否是 Animal 类型时,

结果为 True，当测试变量 animal 是否是 Dog 类型时，结果为 False，即在继承关系中，一个子类实例对象的引用变量其数据类型也可以看作父类类型，但一个父类实例对象不能看作一个子类类型。

注意：isinstance()函数的语法：

```
isinstance(object, classinfo)
```

参数：

object 为实例对象。

classinfo 可以是直接或间接的类名、基本类型或者由它们组成的元组。

返回值：

如果对象的类型与参数二的类型(classinfo)相同，则返回 True，否则返回 False。计算时应考虑继承关系。

【例 5-19】 函数多态应用，代码如下：

```
#第 5 章/5 - 19.py
class Animal():                    #定义父类
    def shout(self):
        print("动物叫")
class Dog(Animal):                 #定义子类
    def shout(self):               #重写父类 shout()方法
        print("狗汪汪地叫")
#主体程序
def testfun(animal):               #测试函数
    animal.shout()                 #调用传入对象的 shout()方法
animal = Animal()
dog = Dog()
testfun(animal)                    #将 Animal 类型变量传入测试函数
print("-" * 8)
testfun(dog)                       #将 Dog 类型变量传入测试函数
```

上述代码运行的结果如图 5-26 所示。

```
动物叫
--------
狗汪汪地叫
```

图 5-26　例 5-19 的运行结果

运行结果显示，当给测试函数 testfun()传入 Animal 类型变量或者其子类(如 Dog 类型变量)时，该函数都能正常运行，并且传入变量类型不同，呈现结果也不同，如若传入的是 Animal 类型，则函数实现 Animal 类的 shout()方法，若传入的是 Dog 类型，则函数实现 Dog 类的 shout()方法，同一个函数 testfun()在不改变函数的定义及实现状态下，根据传入参数类型的不同执行结果会呈现出不同的形态。实际上，任何依赖 Animal 类型作为参数的函数或方法都可以在不做任何修改的情况下，在传入 Animal 类对象或其子类对象时，根据传入参数的不同呈现出不同的形态。

由于 Python 属于动态语言，实际上在传入参数时不一定必须传入 Animal 类对象或其

子类对象,只需保证传入的对象有一个 shout()方法,例如在上例中增加 Car 类的定义,在 Car 类中也有一个 shout()方法,其声明和 Animal 类中 shout()方法一致,代码如下:

```
class Car():
    def shout(self):
        print("汽车会发出声音")
```

Car 类和 Animal 类没有任何继承关系,但在 Car 类也有一 shout()方法。此时将 Car 类型实例对象传入函数 testfun()会如何呢? 修改主体测试代码,代码如下:

```
car = Car()
testfun(car)    #将 Car 类型变量传入测试函数
```

运行以上测试代码,运行结果如图 5-27 所示。

```
汽车会发出声音
```

图 5-27 例 5-19 测试 Car 类的运行结果

从运行结果可以看出,虽然 Car 类和 Animal 类没有任何继承关系,但在 Car 类也有一个 shout()方法,当使用 Car 类型变量传入函数 testfun()时,函数依然能正常运行且实现的是 Car 类的 shout()方法。

这是动态语言支持的"鸭子类型",相比与强类型静态语言(如 Java),Python 动态语言并不要求严格的继承关系,一个对象只要走起来像鸭子,叫起来像鸭子,那么它就可以被当作鸭子,即在函数或方法中传入参数时,不需要关注对象的类型,更需要关注对象的行为是否有同样的行为。

总之,多态的特征主要体现在 3 个方面:

(1) 重写方法的多态。在上面的案例中 Animal 类的子类都可以对父类 shout()方法重新定义,体现方法的多态性。

(2) 变量类型的多态性。在继承关系中,一个子类实例对象 dog 其数据类型可以看作 Dog 类型,也可以看作其父类类型 Animal 类,注意反过来不成立。

(3) 参数类型多态性。对于函数或方法的参数而言,不需要关注参数对象的类型,更需要关注参数对象的行为,只要有同类行为,如 Car 类和 Animal 类均有 shout()方法,就可以作为函数 testfun()的参数,实现参数的多态性。


10min

5.4 综合案例:编写程序模拟士兵突击任务

士兵突击原型钢七连的"不放弃,不抛弃!"的战斗精神,在今天的年轻士兵身上依然继续继承。我们的武器也在继承以往功能的基础上有了突飞猛进的发展。相信我们勇于继承前辈精神,再配上功能更强大的武器,一定能战胜一切来犯的敌人。

请结合本章所学内容,使用面向对象编程中的封装、继承、多态等概念模拟士兵突

击任务。

【要求】

(1) 封装设计一个枪类武器 Gun,主要描述枪类的属性(如名称和子弹数量)及枪支的功能(如装填子弹功能和射击功能)。

(2) 封装设计一个新式枪类武器 G95,继承自 Gun 类,添加新的夜视功能,并改进原有的射击功能。

(3) 封装设计一个士兵 Soldier 类,主要描述士兵的属性,如姓名及装配武器和冲锋射击功能。

(4) 编写主程序,初始化一名士兵,模拟装备上不同武器时冲锋射击的效果。

【目标】

(1) 通过该案例的训练,熟悉 Python 中面向对象的封装、继承、多态等概念及语法知识,掌握在实际应用中使用封装、继承等实现代码的重用和扩展,培养抽象表达能力、分析问题、解决问题能力及实际动手能力。

(2) 在编程的同时,深刻理解“不放弃,不抛弃!”的精神。

【步骤】

(1) 设计一个 Gun 类,代表枪类武器。

(2) 添加 model 和 buttet_count,代表枪的两个属性:枪的名称和子弹数量。

(3) 在 Gun 类的初始化方法__init__()中根据参数实现枪的名称和子弹数量的初始化工作。

(4) 编写 add_bullet()方法,模拟实现装填子弹功能。

(5) 编写 shoot()方法,模拟实现枪的射击功能。

(6) 重写__str__()方法,生成枪型号的字符描述。

(7) 编写程序,代码如下:

```
#第5章/Soldier.py
class Gun:
    def __init__(self,model,bullet_count):
        #枪的型号
        self.model = model
        #子弹数量
        self.bullet_count = bullet_count
    #模拟装填子弹功能
    def add_bullet(self,count):
        self.bullet_count += count;
    #模拟射击功能
    def shoot(self):
        #判断子弹数量
        if self.bullet_count <= 0:
            print("{0}没有子弹了".format(self.model))
            return;
```

```
        #发射子弹
        self.bullet_count -= 1
        #提示发射信息
        print("{0}射击哒哒哒... 还有子弹{1}发
".format(self.model,self.bullet_count))
    #生成枪型号的字符描述
    def __str__(self):
        return self.model
```

(8) 设计一个 G95 类,代表新式武器。该类继承自 Gun 类。

(9) 编写 add_night_vision()方法,模拟新式武器增加的夜视功能。

(10) 重写 shoot()方法,增强射击功能,在射击时自动使用夜视功能。

(11) 编写程序,代码如下:

```
#第 5 章/Soldier.py

class G95(Gun):
    #派生新功能
    def add_night_vision(self):
        print("{0}打开夜间瞄准装置".format(self.model))
    #重写新的 shoot()方法
    def shoot(self):
        self.add_night_vision()
        super().shoot()
```

(12) 编写一个 Soldier 类,代表士兵,并具有姓名 name 和武器 gun 两个实例属性。

(13) 在 Soldier 类的初始化方法__init__()中通过参数初始化士兵的名字,默认士兵没有配备武器。

(14) 编写方法 fire(),模拟实现冲锋射击。

(15) 编写方法 gunfix(self,gun),模拟实现装配武器。

(16) 编写程序,代码如下:

```
#第 5 章/Soldier.py
class Soldier:
    def __init__(self,name):
        #姓名
        self.__name = name
        #默认士兵没有配备武器
        self.__gun = None
    #模拟实现冲锋射击
    def fire(self):
        if self.__gun is None:
            print("{0}还没有枪".format(self.__name))
            self.__gun = Gun("56 式半自动步枪",10)
            print("{0}自动装配枪支: {1}".format(self.__name,self.__gun))
```

```
        else:
            print("{0}装配枪支: {1}".format(self.__name, self.__gun))
        #喊口号
        print("{0}冲啊...".format(self.__name))
        #发射子弹
        self.__gun.shoot()
    #装配武器
    def gunfix(self,gun):
        self.__gun = gun
```

(17) 编写主函数 main(),初始化一名士兵,模拟装备上不同的武器,并测试运行冲锋射击的效果。

(18) 编写程序,代码如下:

```
#第 5 章/Soldier.py

def main():
    #实例化士兵
    solider = Soldier("许三多")
    solider.fire()
    #分割线
    print("-" * 10)
    #给士兵配装新式武器
    solider.gunfix(G95("95 式自动步枪",30))
    solider.fire()
if __name__ == '__main__':
    main()
```

(19) 程序调试运行,通过运行结果可以看出当士兵配备上新的武器时,冲锋射击时武器的 shoot()方法调用的是子类重写的 shoot()方法,增加了子类扩展的武器功能。运行结果如图 5-28 所示。

```
许三多还没有枪
许三多自动装配枪支: 56式半自动步枪
许三多冲啊...
56式半自动步枪射击哒哒哒... 还有子弹9发
----------
许三多装配枪支: 95式自动步枪
许三多冲啊...
95式自动步枪打开夜间瞄准装置
95式自动步枪射击哒哒哒... 还有子弹29发
```

图 5-28　程序的运行结果

思考:从图 5-28 的运行结果可以看出新式武器射击时和老式武器一样,每次发射一发子弹,如何改写代码实现新式武器每次射击时发射五发子弹?如何增加新的武器功能?在上述代码的基础上进行改写,设计自己的武器系列和功能。

5.5　小结

本章主要介绍了面向对象编程的基础概念和思想,首先简要地介绍了面向对象编程概念,接下来重点介绍了类、属性、方法的基础概念和编写语法,通过案例介绍了面向对象的封

装性、继承性和多态性的体现及实现方式。最后通过一个综合案例对本章知识点进行巩固和实际应用。

本章的知识结构如图 5-29 所示。

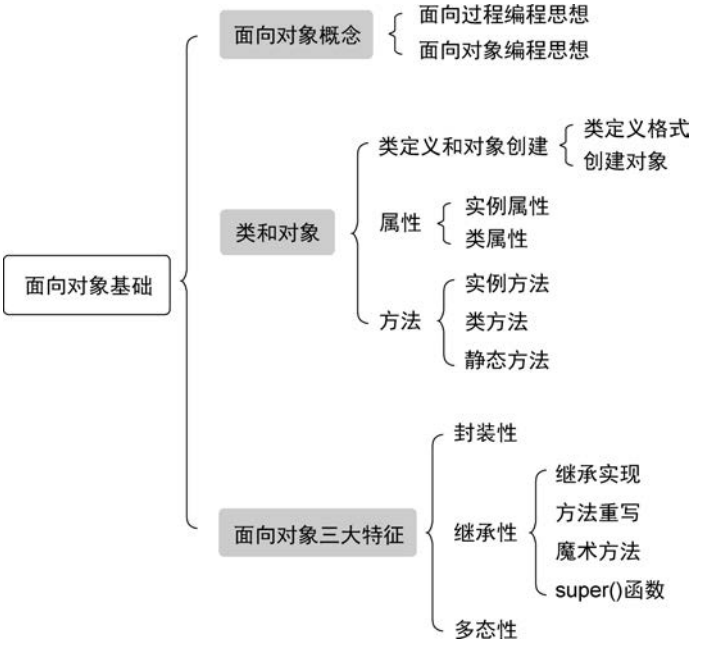

图 5-29　面向对象基础知识结构图

5.6　习题

1. 填空题

(1) 面向对象的程序设计具有 3 个基本特征：_____、_____和_____。

(2) 在 Python 中创建对象后，可以使用_____运算符来调用其成员。

(3) 在 Python 中，实例变量在类的内部通过_____访问，在外部通过对象实例访问。

(4) 创建一个类需要用关键字_____。

(5) 下列 Python 语句的运行结果为_____。

```
class Point():
    x = 100
    y = 100
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

```
point = Point(50,50)
print(point.x,point.y)
```

（6）下列 Python 语句的运行结果为_____。

```
class Account:
    def __init__(self, id, balance):
        self.id = id;
        self.balance = balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount

acc1 = Account('郭靖', 1000);
acc1.deposit(500)
acc1.withdraw(200);
print(acc1.balance)
```

（7）下列 Python 语句的运行结果为_____。

```
class Person():
    age = 20
class Student(Person):
    pass

print(Person.age, Student.age)
```

（8）下列程序的运行结果是_____。

```
class Car:
    __distance = 0
    def __init__(self,name):
        self.name = name
        Car.__distance = Car.__distance + 1
    def show(self):
        print('Car.__distance:',Car.__distance)

car1 = Car("Geely")
car2 = Car("极氪 001")
car3 = Car("领克 01")
car3.show()
```

（9）下列程序的运行结果是_____。

```
class A:
    @staticmethod
```

```
    def getSum(numbers):
        return sum(numbers)

a = A()
resultOne = a.getSum([1,2,3,4,5])
resultTwo = a.getSum([6,7,8,9,10])
print(resultOne,resultTwo)
```

(10) 下列程序的运行结果是_____。

```
class Cirle:
    def __init__(self):
        self.__radius = 1
    def setRadius(self,radius):
        if radius > 0:
            self.__radius = radius
    def getRadius(self):
        return self.__radius

class Geometry:
    def __init__(self):
        self.__circle = Cirle()
    def setCircle(self,c):
        c.setRadius(10)
        self.__circle = c
    def getCircle(self):
        return self.__circle

c = Cirle()
c.setRadius(100)
g = Geometry()
r = c.getRadius()
g.setCircle(c)
print(r,g.getCircle().getRadius(),c.getRadius())
```

2. 选择题

(1) Python 定义私有变量的方法为(　　)。

A. 使用__private 关键字　　B. 使用 public 关键字

C. 使用__xxx__定义变量名　　D. 使用__xxx 定义变量名

(2) 在面向对象思想中,继承是指(　　)。

A. 类之间共享属性和操作的机制　　B. 各对象之间的共同性质

C. 一组对象所具有的相似性质　　D. 一个对象具有另一个对象的性质

(3) 下列选项中,符合类的命名规范的是(　　)。

A. HolidayResort　　B. Holiday Resort

C. hoildayResort　　D. hoilidayresort

(4) 下列说法中不正确的是(　　)。

A. 类是对象的模板,对象是类的实例

B. 在 Python 中一切皆对象,类本身也是对象

C. 属性名如果以“__”开头,就变成了一个私有属性

D. 在 Python 中,一个子类只能有一个父类

(5) 下列 Python 语句的运行结果为(　　)。

```
class Swordsman:
    def __init__(self,name = "李寻欢"):
        self.name = name
    def show(self):
        print(self.name)

s = Swordsman("沈浪")
s.show()
```

A. 李寻欢　　B. 沈浪　　C. None　　D. 错误

(6) 下列 Python 语句的运行结果为(　　)。

```
class Person:
    def __init__(self,name = "傅红雪"):
        self.name = name

class Swordsman(Person):
    def __init__(self,s = "female"):
        self.sex = s
    def show(self):
        print(self.name,self.sex)

s = Swordsman("male")
s.show()
```

A. 傅红雪　　B. female　　C. 错误　　D. 傅红雪 male

(7) 下列 Python 语句的运行结果为(　　)。

```
class Person:
    def __init__(self,name = "傅红雪"):
        self.name = name

class Swordsman(Person):
    def __init__(self,s = "female"):
        super().__init__()
        self.sex = s
    def show(self):
        print(self.name,self.sex)
```

```
s = Swordsman("male")
s.show()
```

A. 傅红雪　　B. female　　C. 错误　　D. 傅红雪 male

(8) 下列 Python 语句的运行结果为(　　)。

```
class Animal:
    def get(self,n):
        return n

class Dog(Animal):
    def get(self,n):
        return n + super().get(n)

a = Animal()
m = a.get(10)
a = Dog()
n = a.get(10)
print(m,n)
```

A. 10 10　　B. 10 20　　C. 20 10　　D. 20 20

3. 编程题

(1) 定义一个人类 Person,该类中有两个私有属性,即姓名 name 和年龄 age。定义初始化方法,用来初始化属性数据。定义 showinfo()方法,以便将姓名和年龄打印出来。在 main()方法中创建人类的实例并显示个人信息。

(2) 定义一个交通工具 Vehicle 类,该类有 3 个属性,即名称 name、速度 speed 和容量 capacity,方法有加速 speedUp()、减速 speedDown()。在 main()方法中实例化一个交通工具类实例,在实例化时对 3 个属性值进行初始化并打印出来,另外调用加速和减速的方法对速度进行改变并显示改变后的状态值。

(3) 定义一个类 QuadraticEquation,求解一元二次方程 $ax^2+bx+c=0$ 的根,该类包括 a、b、c 共 3 个属性,分别表示方程的 3 个系数,方法 getRoot()用于返回方程的两个根。在 main()方法中提示用户输入 a、b、c 的值,打印显示方程的根。

(4) 定义一个食物类 Food,该类有一属性,即名称 name,定义 showinfo()方法,以便打印显示名称。定义一个水果类 Fruits,继承自 Food,并给水果类添加新的属性,即颜色 color,重写父类 showinfo()方法,以便显示水果的名称和颜色。在 main()方法中创建苹果对象、西瓜对象,给水果添加质量 weight 属性,并显示水果信息。

(5) 设计一个名为 Stock 的类,以此来表示一个公司的股票,包括以下内容:

① 股票代码、股票名称、前一天股票价格、当前股票价格共 4 个属性。

② 构造方法用于初始化股票代码、股票名称、前一天股票价格、当前股票价格等属性。

③ 用于返回股票名称的公开方法。

④ 用于返回股票代码的公开方法。

⑤ 获取或设置前一天股票价格的公开方法。

⑥ 获取或设置当前股票价格的公开方法。

⑦ 名为 getChangePercent()的方法，返回前日收市价至当前价格的变化百分比。

(6) 编写一个程序，创建一个 Stock 对象，它的代码是 00175，名称为吉利汽车，前一天价格为 10.72 港元，当前价格为 10.82 港元，输出显示该股票的信息及价格变化百分比。