8. 面向对象

面向对象程序设计（Object Oriented Programming，OOP）的思想主要针对大型软件设计而提出，它使得软件设计更加灵活，能够很好地支持代码复用和设计复用，并且使得代码具有更好的可读性和可扩展性。Python 完全采用了面向对象设计的思想，是真正面向对象的高级动态编程语言，完全支持面向对象的基本功能。因此，掌握面向对象程序设计思想至关重要。

特性 -》属性

行为 -》方法

8.1. 面向对象程序设计入门

提到**面向对象**，自然会想到**面向过程。面向过程程序设计的核心是过程，过程即解决问题的步骤**，面向过程的设计就好比精心设计好一条流水线，需要考虑周全解决问题的每个步骤。

编写程序，模拟学生选课，每选一门课程，将课程名加入到学生的所选课程中，同时将课程的学分**累加**到学生的总学分中。

【问题分析】学生选课，首先需要定义一个学生和多门课程，然后再定义一个用于实现选课功能的函数，最后调用该函数。

```
stu = {'num':'201801','name':'Jack', 'credit': 0,'course':[]}    #定义一个学生
cours1 = {'num':'01','name':'Python','credit': 3}                #定义课程1
cours2 = {'num':'02','name':'C','credit': 4}                     #定义课程2
def choose(c):                                                   #定义实现选课功能的
函数
    stu['credit'] += c['credit']                                 #将课程的学分累加到
学生的总学分中
    stu['course'].append(c['name'])                              #将课程名加入到学生
的所选课程中
choose(cours1)                                                   #学生选课程1
choose(cours2)                                                   #学生选课程2
print(stu)                                                       #输出学生信息
```

【程序说明】这段代码中，如果新增加学生或者课程，虽然每个学生和每门课程都包括类似的信息，但都需要重新定义。

我们知道，**选课这个动作只能学生完成**，也就是说 choose() 函数只能由学生调用，但是程序中并没有这样的限制。假如不小心让课程调用了该函数，即将学生变量作为了该函数的参数，那么会发生什么呢？我们在上述代码的倒数第 2 行增加语句`choose(stu)`。

程序运行结果可以看到程序运行并没有出错，而是将学生的姓名添加到了所选课程中，并将学生的上一状态的总学分进行了累加。这显然是错误的。

此时最好的解决方法就是采用面向对象程序设计思路进行编程。使用面向对象思路实现上述问题时，可以将“学生”和“课程”分别看作两类对象，具体如下：

- 学生类：
    - 特征
        - 学号

        - 姓名
        - 总学分
        - 所选课程
    - 行为
        - 选课
- 课程类：
    - 特征
        - 课程编号
        - 课程名
        - 学分

有了这样的类后，我们可以很轻松地实例化多个学生和多门课程，执行选课操作时，也限制了只有学生能够进行选课操作。

> 总的来说，面向对象程序设计是一种解决代码**复用**的编程方法。
>
> 这种方法把软件系统中相似的**操作逻辑**、**数据**和**状态**以类的形式描述出来，以对象实例的形式在软件系统中复用，以达到**提高软件开发效率**的目的。

8.2. 类的定义与使用

在面向对象编程中，最重要的两个概念就是**类和对象（也称为实例）**。对象是某个具体存在的事物，例如，一个名叫“Jack”的学生就是一个对象。与对象相比，类是抽象的，它是对一群具有相同特征和行为的事物的统称。

例如，学校的学生，其特征包括学号、姓名、性别等，其行为包括选课、上课、考试等。

> 我们在前面的章节中已经用了很长时间的类和对象了。例如，字典类型的本质就是类，一说到字典，我们就知道是用“**{}**”表示的，由“键值”对这样的元素组成的，它还具有一些增、删、改、查的方法。但是我们并不知道字典里存储了哪些具体内容。所以说，字典这个类型就是类，而某一个具体赋值的字典就是对象。

8.2.1. 类的定义

面向对象程序设计思想是把事物的特征和行为包含在类中。其中，事物的特征作为类中的变量，事物的行为作为类的方法，而对象是类的一个实例。因此，要想创建一个对象，需要先定义一个类。定义类的基本语法格式如下：

```
class 类名:
    类体
```

> 特征 ---> 变量
>
> 行为 ---> 方法
>
> 对象 ---> 实例

Python 使用 **class** 关键字来定义类，class 关键字后是一个**空格**，然后是**类的名字**，再后是一个**冒号**，最后**换行并定义类的内部实现。**定义类时需要注意:

- （1）类名的**首字母**一般需要**大写**，如 Car。
- （2）类体一般包括变量的定义和方法的定义。
- （3）**类体**相对于 class 关键字**必须保持一定的空格缩进。**

例如，定义一个汽车类，包含价格特征和行驶行为，代码如下：

```
#定义类
class Car:
    price = 150000                    #定义价格变量
    def run(self):                    #定义行驶方法
        print('车在行驶中……')
```

【程序说明】上述代码中，使用 class 定义了一个名称为 Car 的类，类中有一个 price 变量和一个 run 方法。从代码中可以看出，方法和函数的格式是一样的，主要区别在于，**方法必须显式地声明一个 self 参数，而且位于参数列表的开头**。

8.2.2. 创建类的对象

程序想要完成具体的功能，仅有类是远远不够的，还需要根据类来创建实例对象。在Python中，创建对象的语法格式如下：

```
对象名 = 类名()
```

创建完对象后，可以使用它来访问类中的变量和方法，具体方法是：

```
对象名.类中的变量名
对象名.方法名([参数])
```

示例：

```
class Car:
    price = 150000
    def run(self):
        print("车在行驶中……")
car_1 = Car()
car_1.run()
print('车的价格是：', car_1.price)
```

8.2.3. self 参数

类的所有方法都必须至少有一个名为 self 的参数，并且必须是方法的第 1 个参数。

如果把类比作是制造汽车的图纸，那么由类实例化的对象才是真正可以开的汽车。根据一张图纸可以设计出成千上万的汽车，它们长得都差不多，但它们都有各自不同的属性，如颜色不同、内室不同等。所以 **self** 就相当于每辆车的编号，有了 self，就可以轻松投到对应的车了。

在 Python 中，由同一个类可以生成无数个对象，当一个对象的方法被调用时，对象会将自身的引用作为第一个参数传递给该方法，那么 Python 就知道需要操作哪个对象的方法了。

在类的方法中访问变量时，需要以 self 为前缀，但在外部通过对象名调用对象方法时不需要传递该参数。

self 的使用：

```
#定义类
class Car:
    def colour(self, col):                          #定义赋值颜色方法
        self.col = col
    def show(self):
        print('The color of the car is %s.'%self.col)
car_1 = Car()
car_1.colour('red')
car_2 = Car()
car_2.colour('white')
car_1.show()
car_2.show()
```

> 小提示
>
> Python中，类定义方法时将第一个参数命名为 self 只是一个习惯，而实际上名字是可以改变的。

```
class A:
    def show(my):
        print("hello!")

a = A()
a.show()
```

8.2.4. 构造方法

构造方法的固定名称为`__init__()`，当创建类的对象时，系统会自动调用构造方法，从而实现对对象进行初始化的操作。

使用构造方法：

```
#定义类
class Car:
#构造器方法
    def __init__(self):
        self.wheelNum = 4
        self.colour = '蓝色'
#方法
    def run(self):
        print('{}个轮子的{}车在行驶中……'.format(self.wheelNum, self.colour))
BMW = Car()
BMW.run()
```

【程序说明】在该程序中,第4~6 行实现了`__init__()`方法，给 Car 添加了 wheelNum 和 colour 属性并赋了初值，在 run() 方法中访问了 wheelNum 和 colour 的值。

上面例子中，无论创建多少个 Car 对象，wheelNum 和 colour 属性的初始值都是默认值，如果想要在实例化对象时，为不同对象传入不同的参数值，可以在构造方法中设置形参。

下面通过实例进行说明。

使用带参构造方法：

```
#定义类
class Car:
#构造器方法
    def __init__(self, wheelNum, colour):
        self.wheelNum = wheelNum
        self.colour = colour
#方法
    def run(self):
        print('{}个轮子的{}车在行驶中……'.format(self.wheelNum, self.colour))
BMW = Car(4,'红色')                          #创建对象
Audi = Car(4,'白色')                         #创建对象
BMW.run()                                   #调用方法
Audi.run()                                  #调用方法
```

【程序说明】在该程序中，定义了带参数的构造方法，定义了两个参数 wheelNum 和 colour，创建对象时，可为不同对象传入不同的参数值。

用面向对象程序设计思路改写《学生选课》。

【问题分析】根据分析，学生选课问题涉及到两类事物，一个是学生，其特征包括学号、姓名、总学分和所选课程，行为包括选课；另一个是课程，其特征包括课程编号、课程名和学分。因此，可设计 2 个类，其中学生类中可设计两个方法，包括用于初始化属性的构造方法和一个用于实现选课功能的方法；课程类中课设计一个用于初始化属性的构造方法。

```
#定义学生类
class Student:
    def __init__(self, num, name, credit, course):      #构造方法定义学生属性
        self.num = num
        self.name = name
        self.credit  = credit
        self.course = course
    def choose(self, c):                                #定义学生选课方法
        self.credit += c.credit
        self.course.append(c.name)
#定义课程类
class Course:
    def __init__(self, num, name, credit):              #构造方法定义课程属性
        self.num = num
        self.name = name
        self.credit  = credit
stu_1 = Student('201801','Jack',0,[])                       #创建学生1
stu_2 = Student('201802','Tom',3,['Math'])                  #创建学生2
cou_1 = Course('01','Python', 3)                            #创建课程1
cou_2 = Course('02','C', 4)                                 #创建课程2
stu_1.choose(cou_1)                                 #调用方法实现学生1选课程1
stu_2.choose(cou_2)                                 #调用方法实现学生2选课程2
#输出各学生信息
```

```
print('学号:',stu_1.num,'姓名:',stu_1.name,'总学分:',stu_1.credit,'所选课
程',stu_1.course)
print('学号:',stu_2.num,'姓名:',stu_2.name,'总学分:',stu_2.credit,'所选课
程',stu_2.course)
```

8.2.5. 析构方法

创建对象时，Python 解释器默认会调用构造方法：当需要删除一个对象来释放类所占的资源时，Python 解释器会调用另外一个方法，这个方法就是析构方法。析构方法的固定名称为`__del__()`，程序结束时会自动调用该方法，也可以使用 del 语句手动调用该方法删除对象。接下来，通过一个实例来演示如何使用析构方法释放资源。

通过析构方法释放资源：

```
class Animal():
    #构造方法
    def __init__(self):
        print('---构造方法被调用---')
    #析构方法
    def __del__(self):
        print('---析构方法被调用---')
#创建对象
dog = Animal()
print('---程序结束---')
```

```
class Animal():
    #构造方法
    def __init__(self):
        print('---构造方法被调用---')
    #析构方法
    def __del__(self):
        print('---析构方法被调用---')
#创建对象
dog = Animal()
del dog
print('---程序结束---')
```

【程序说明】以上两段代码的区别在手，代码二在程序结束前使用 del 语句手动调用析构方法删除对象，因此，先输出“---析构方法被调用---”。而代码一没有使用 del 语句，因此，在程序结束时才调用析构方法，后输出“---析构方法被调用---”。

8.3. 类成员和实例成员

在前面的例子中，定义类时，有的变量定义在构造函数中，有的变量定义在类中所有方法之外，那么它们有什么区别呢?

类中定义的变量又称为数据成员，或者叫广义上的属性。可以说数据成员有两种：一种是实例成员（实例属性），另一种是类成员（类属性）。

实例成员一般是指在构造函数 `__init__()` 中定义的，定义和使用时必须以 **self** 作为前缀；类成员是在类中所有方法之外定义的数据成员。两者的区别是：在主程序中（或类的外部），实例成员属于实例（即对象），只能通过对象名访问；而类成员属于类，可以通过类名或对象名访问。在类的方法中可以调用类本身的其他方法，也可以访问类成员以及实例成员。

注意：与很多面向对象程序设计语言不同，Python 允许动态地为类和对象增加成员，这是Python 动态类型特点的重要体现。

```
#定义类
class Car:
    price = 150000                          #类成员
    def __init__(self,colour):
        self.colour = colour                    #实例成员
car_1 = Car('红色')                           #创建对象
print(car_1.price,Car.price,car_1.colour)       #访问类成员和实例成员并输出
Car.name = 'Audi'                           #增加类成员
car_1.wheelNum = 4                          #增加实例成员
print(car_1.wheelNum, car_1.name, Car.name) #访问类成员和实例成员并输出
```

【程序说明】 Car 类中定义的 price 和动态为类增加的 name 都为类成员，因此，它们都属于类，可以通过类名或对象名访问。但构造方法中定义的 colour 和动态为对象 car_1 增加的 wheelNum 都为实例成员，因此，它们只能通过对象名访问。如果用类名进行访向会提示错误信息，例如，在程序的末尾增加一条语句 `print(Car.colour)`，程序运行出错，提示 Car 对象没有 colour 属性。

如果类中有相同名称的类成员和实例成员，那么程序又会如何访问呢？下面通过实例说明。

类中有相同名称的类成员和实例成员示例。

```
#定义类
class Car:
    price = 150000                          #类成员
    def __init__(self):
        self.price = 100000                     #实例成员
car_1 = Car()                               #创建对象
print(car_1.price,Car.price)            #访问类成员和实例成员并输出
```

【程序说明】从程序运行结果中可以看出，当类成员和实例成员的名字相同时，通过对象名访问成员（car_1.price）时获取的是实例成员的值，通过类名访问成员（Car.price）时获取的是类成员的值。

8.4. 封装

封装是面向对象的特征之一，是对象和类概念的主要特征。**封装，就是把客观事物封装成抽象的类，并规定类中的数据和方法只让可信的类或对象操作。**

封装可分为两个层面：

- （1）第一层面的封装，创建类和对象时，分别创建两者的名称，只能通过类名或者对象名加“.”的方式访问内部的成员和方法，前面介绍的例子其实都是这一层面的封装。
- （2）第二层面的封装，类中把某些成员和方法隐藏起来，或者定义为私有，只在类的内部使用，在类的外部无法访问，或者留下少量的接口（方法）供外部访问。本节重点介绍第二层面的封装。

在默认情况下，Python 中，**对象的数据成员和方法都是分开的**，可以直接同过点操作符“.”进行访问。为了实现更好的数据封装和保密性，可以将类中的数据成员和方法**设置成私有的。**

在 Python 中，私有化方法也比较简单，在准备私有化的数据成员或方法的名字前面加**两个下划线** `__` 即可。下面通过实例进行说明。

```
class A:                          #定义类
    def __init__(self):
        self.__X = 10             #定义私有变量并赋值为10
    def __foo(self):              #定义私有方法
        print('from A')
a = A()                           #创建对象
print(a.__X)                      #输出私有变量值
a.__foo()                         #调用私有方法
```

【程序说明】上述代码中定义了一个给私有属性赋值的构造方法，又定义了一个私有方法。运行程序后，出现错误信息，意思是"A"类中没有找到"X"属性。出现上述问题的原因是 `__X` 为私有属性，类的外部无访问类的私有属性。同理，如果程序执行到 `a.__foo()` 时，也会提示类似的错误信息。

对于这一层面的封装（隐藏），我们需要在类中定义一个方法（也称接口函数），在它内部访问被隐藏的属性和方法，然后外部可以通过接口函数进行访问。

修改上述代码，在类中增加一个方法（**接口函数**），实现通过调用该方法访问内部成员及内部方法。

```
class A:                          #定义类
    def __init__(self):
        self.__X = 10             #定义私有变量并赋值为10
    def __foo(self):              #定义私有方法
        print('from A')
    def bar(self):                #定义接口函数
        self.__foo()              #类内部访问私有方法
        return self.__X           #返回私有变量__X的值
a = A()                           #创建对象
b = a.bar()                       #调用接口函数，将返回值赋给b
print(b)                          #输出b的值
```

提示：Python 目前的私有机制其实是**伪私有**，实际上，在外部可以通过 `_类名__属性` 访问私有变量和方法。例如，将代码改成以下形式，即可正常运行程序。

```
class A:                          #定义类
    def __init__(self):
        self.__X = 10             #定义私有变量并赋值为10
    def __foo(self):              #定义私有方法
        print('from A')
a = A()                           #创建对象
print(a._A__X)                    #通过类名访问私有变量值
a._A__foo()                       #通过类名调用私有方法
```

8.5. 继承

在程序中，继承描述的是事物之间的从属关系，例如，学生和教师都属子人类，程序中就可以描述为学生和教师继承自人类。

设计一个新类时，如果可以继承一个已有的设计良好的类然后进行二次开发，可以大幅度减少开发工作量，并且可以很大程度地保证质量。在继承关系中，已有的、设计好的类称为**父类**或**基类**，新设计的类称为**子类**或**派生类**。继承可以分为**单继承**和**多继承**两大类。

8.5.1. 单继承

在 Python 中，当一个子类只有一个父类时称为单继承。子类定义如下：

```
class 子类名(父类名):
    类体
```

子类可以继承父类的所有公有方法，但不能继承其私有成员和私有方法。

```
#定义一个父类
class Person:
    name = '人'
    age = 30
    def speak(self):                                    #定义方法用于输出
        print ('%s 说: 我 %d 岁。' %(self.name,self.age))
#定义一个子类
class Stu(Person):
    def setName(self, newName):                         #定义方法用于修改名字
        self.name = newName
    def s_speak(self):                                  #定义方法用于输出
        print ('%s 说: 我 %d 岁。' %(self.name,self.age))
student = Stu()                                         #创建学生对象
print ('student的名字为:',student.name)                  #输出学生名字
print ('student的年龄为:',student.age)                     #输出学生年龄
student.s_speak()                                       #调用子类方法用于输出
student.setName('Jack')                                 #调用子类方法用于修改名字
student.speak()                                        #调用父类方法用于输出
```

【程序说明】上述代码中定义了一个 Person 类。该类中有一个 name 属性和一个 age 属性，还有一个 speak 方法；然后定义了一个继承自 Person 类的子类 Stu，其内部包含一个 setName 方法和一个 s_speak 方法。从程序的运行结果可以看出，**子类继承了父类的属性和方法。**

8.5.2. 多重继承

多继承指一个子类可以有多个父类，它继承了多个父类的特性。例如，沙发床是沙发和床的功能的组合。

多继承可以看作是对单继承的扩展，其语法格式如下：

```
class 子类名(父类名1，父类名2……)
```

示例

```
#定义沙发一个父类
class Sofa:
    def printA(self):
```

```
        print ('----这是沙发----')
#定义床一个父类
class Bed:
    def printB(self):
        print('----这是床----')
#定义一个子类，继承自Sofa和Bed
class Sofabed(Sofa,Bed):
    def printC(self):
        print('----这是沙发床----')
obj_C = Sofabed()                    #创建对象
obj_C.printA()                       #调用Sofa父类中的方法
obj_C.printB()                       #调用Bed父类中的方法
obj_C.printC()                       #调用自身的方法
```

【程序说明】上述代码中定义了一个 Sofa 类，该类有一个 printA 方法，然后定了一个 Bed 类，该类有一个 printB 方法。接着定义了一个继承自 Sofa 和 Bed 的子类 Sofabed，该类内部有一个 printC 方法。创建一个 Sofabed 类的对象 obj_C，分别调用 printA、printB 和 printC 方法，从程序输出结果可以看出，子类同时继承了多个父类方法。

> **提示：**在 Python 中，如果两个父类中有同名的方法，调用该同名方法时会调用先继承类中的方法。例如，如果 Sofa 和 Bed 类中有同名的方法，用 `class Sofabed(Sofa,Bed):` 语句定义子类时，子类会先继承 Sofa 类。

8.5.3. 重写父类方法与调用父类方法

在继承关系中，子类会自动继承父类中定义的方法，但如果父类中的方法功能不能满足需求，就可以在子类中重写父类的方法。**即子类中的方法会覆盖父类中同名的方法，这也称为重载。**

重写父类的方法示例。

```
#定义一个父类
class Person:
    def speak(self):                                    #定义方法用于输出
        print ('我是一个人类')
#定义一个子类
class Stu(Person):
    def speak(self):                                    #定义方法用于输出
        print('我是一个学生')
student = Stu()                                         #创建学生对象
student.speak()                                     #调用同名方法
```

【程序说明】从程序的输出结果可以看出，在调用 Stu 类对象的 speak 方法时，只调用了子类中重写的方法，不会再调用父类的 speak 方法。

如果需要再子类中调用父类的方法，可以使用内置函数 **super()** 或通过 “**父类名.方法名()**” 的方式来实现。

子类调用父类方法示例

```
#定义父类
class Person():
    def __init__(self, name, sex):
        self.name = name
        self.sex = sex
#定义子类
class Stu(Person):
    def __init__(self, name, sex, score):
        super().__init__(name, sex)              #调用父类中的__init__方法
        self.score = score
#创建对象实例
student = Stu('Jack','Male',90)
print("姓名:%s，性别:%s，成绩: %s"%(student.name, student.sex, student.score))
```

【程序说明】上述代码中首先定义了 Person 类，该类的 `__init__()` 方法中设置了 name 和 sex 属性。然后定义了继承自 Person 类的子类 Stu，在该类中重写了构造方法 `__init__()`，使用 `super()` 函数调用了父类的构造方法，并添加了自定义的属性 score，使 Stu 类即拥有自定义的属性 score，又有父类方法的属性 name 和 sex。其中，`super().__init__(name, sex)` 语句也可以用 `Person.__init__(self, name, sex)` 语句替换。

8.6. 多态

多态指的是一类事物有多种形态，如一个父类有多个子类，因而多态的概念依赖于继承。在面向对象方法中一般是这样描述多态性的：向不同的对象发送一条信息，不同的对象在接收时会产生不同的行为（即方法）。也就是说，每个对象可以用自己的方式去响应共同的消息（调用函数）。

多态实例

```
#定义父类
class Person:
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
    def who(self):                          #定义who方法
        print('I am a Person, my name is %s' % self.name)
#定义学生子类
class Student(Person):
    def __init__(self, name, gender, score):
        super().__init__(name, gender)
        self.score = score
    def who(self):                          #重写父类方法
        print('I am a Student, my name is %s' % self.name)
#定义教师子类
class Teacher(Person):
    def __init__(self, name, gender, course):
        super().__init__(name, gender)
        self.course = course
    def who(self):                          #重写父类方法
        print('I am a Teacher, my name is %s' % self.name)
#定义函数用于接收对象
def fun(x):
    x.who()                                 #调用who方法
```

```
#创建对象
p = Person('Jack', 'Male')
s = Student('Tom', 'Male', 88)
t = Teacher('Lily', 'Female', 'English')
#调用函数
fun(p)
fun(s)
fun(t)
```

【程序说明】上述代码中首先定义了 Person 类，该类中定义了一个 who 方法，然后定义了继承自 Person 类的两个子类 Student 和 Teacher，分别在这两个类中重写了 who 方法，接着定义了一个带参数的 fun 函数，在该函数中调用了 who 方法。最后分别创建了Person 类型的对象 p、Student 类型的对象 s 和 Teacher 类型的对象 t，并作为参数调用了 fun 函数。从程序运行结果可以看出，通过向函数中传入不同的对象，who 方法输出不同的结果。

8.7. 类方法和静态方法

前面介绍的所有实例方法（公有方法和私有方法）都必须至少有一个名为 self 的参数，且实例方法只能通过对象名进行调用。在 Python 中，还有两种方法——类方法和静态方法，它们都属于类的方法。

类成员 实例成员

类方法 实例方法

8.7.1. 类方法

类方法是类所拥有的方法，需要用修饰器“@classmethod”来标识其为类方法。对于类方法，第一个参数必须是类对象，一般以 **cls** 作为第一个参数（同 self 一样只是一个习惯），**能够通过对象名调用类方法，也可以通过类名调用类方法。**

类方法的使用

```
#定义类
class People:
    country = 'china'                   #定义类成员并赋值
#类方法，用classmethod来进行修饰
    @classmethod
    def getCountry(cls):
        return cls.country              #返回类成员的值
p = People()                            #创建对象
print(p.getCountry())                   #通过实例对象引用
print(People.getCountry())              #通过类对象引用
```

【程序说明】上述代码中定义了一个 People 类，首先在类中添加了类成员 country，然后在类方法 getCountry 中返回类成员的值。从运行结果可以看出，用**对象名**调用类方法和用**类名**调用类方法的效果是一样的。

> 提示
>
> 类方法可以访问类成员，但无法访问实例成员。

8.7.2. 静态方法

要在类中使用静态方法，需在类成员方法前加上“@staticmethod”标记符，以表示下面的成员方法是静态方法。使用静态方法的好处是，不需要实例化对象即可使用该方法。

静态方法可以不带任何参数，由于静态方法没有 self 参数，所以它无法访问类的实例成员；静态方法也没有 cls 参数，所以它也无法访问类成员。**静态方法既可以通过对象名调用，也可以通过类名调用。**

静态方法使用。

```
#定义类
class Test:
#静态方法，用@staticmethod进行修饰
    @staticmethod
    def s_print():
        print('----静态方法----')
Test.s_print()                  #通过类名调用
t = Test()                      #创建对象
t.s_print()                     #通过对象名调用
```

【程序说明】上述程序定义了 Test 类，在该类中定义了一个静态方法，然后创建Test 类的对象 t，分别通过类名和对象名调用静态方法，得到相同的输出结果。

> 小技巧
>
> 类的对象可以访问实例方法、类方法和静态方法，使用类可以访问类方法和静态方法。一般情况下，如果要修改实例成员的值，直接使用实例方法；如果要修改类成员的值，直接使用类方法；如果是辅助功能，如打印菜单，则可以考虑使用静态方法。

8.8. 典型案例

猫狗大战

编写程序，模拟猫狗大战，要求：

- （1）可创建多个猫和狗的对象，并初始化每只猫和狗（包括昵称、品种、攻击力、生命值等属性）。
- （2）猫可以攻击狗，狗的生命值会根据猫的攻击力而下降；同理狗可以攻击猫，猫的生命值会根据狗的攻击力而下降。
- （3）猫和狗可以通过吃来增加自身的生命值。
- （4）当生命值小于等于 0 时，表示已被对方杀死。

【问题分析】根据要求，可定义两个类——Cat 类和 Dog 类。Cat 类中包含一个构造方法用于初始化各个属性（包括昵称、品种、攻击力、生命值等），一个攻击狗的方法（攻击狗使得狗的生命值下降），一个吃的方法（调用一次可使自身的生命值增加），一个判断是否死亡的方法（如果生命值小于等于 0，表示已被对方杀死，否则输出当前的生命值）。用类似的方式定义 Dog 类。然后创建对象，开始战斗。

参考代码

```
#定义一个猫类
class Cat:
    role = 'cat'                            #猫的角色属性都是猫
#构造方法初始化猫
    def __init__(self, name, breed, aggressivity, life_value):
        self.name = name                    #每一只猫都有自己的昵称
        self.breed = breed                  #每一只猫都有自己的品种
        self.aggressivity = aggressivity    #每一只猫都有自己的攻击力
        self.life_value = life_value        #每一只猫都有自己的生命值
#定义猫攻击狗的方法
    def attack(self, dog):
        dog.life_value -= self.aggressivity #狗的生命值会根据猫的攻击力而下降
#定义增长生命值的方法
    def eat(self):
        self.life_value += 50
#定义判断是否死亡的方法
    def die(self):
        if self.life_value <= 0:            #如果生命值小于等于0表示已被对方杀死
            print(self.name,'已被杀死! ')
        else:
            print(self.name,'的生命值还有',self.life_value)
#定义一个狗类
class Dog:
    role = 'dog'                            #狗的角色属性都是狗
#构造方法初始化狗
    def __init__(self, name, breed, aggressivity, life_value):
        self.name = name                    #每一只狗都有自己的昵称
        self.breed = breed                  #每一只狗都有自己的品种
        self.aggressivity = aggressivity    #每一只狗都有自己的攻击力
        self.life_value = life_value        #每一只狗都有自己的生命值
#定义狗攻击猫的方法
    def bite(self,cat):
        cat.life_value -= self.aggressivity #猫的生命值会根据狗的攻击力而下降
#定义增长生命值的方法
    def eat(self):
        self.life_value += 30
#定义判断是否死亡的方法
    def die(self):
        if self.life_value <= 0:            #如果生命值小于等于0表示已被对方杀死
            print(self.name,'已被杀死! ')
        else:
            print(self.name,'的生命值还有',self.life_value)
#创建实例
cat_1 = Cat('Mily','波斯猫',30,1500)        #创造了一只实实在在的猫
dog_1 = Dog('Lucky','哈士奇',50,900)        #创造了一只实实在在的狗
cat_1.die()                             #输出猫的当前状态
dog_1.die()                             #输出狗的当前状态
print('------开始战斗-----')
cat_1.attack(dog_1)                         #猫攻击狗一次
dog_1.attack(cat_1)                           #狗攻击猫一次
cat_1.die()                             #输出猫的当前状态
dog_1.die()                             #输出狗的当前状态
for i in range(29):                         #循环实现，猫攻击狗29次
    cat_1.attack(dog_1)
dog_1.die()                             #输出狗的当前状态
cat_1.eat()                             #猫吃东西一次
cat_1.die()                             #输出猫的当前状态
```