



Python编程案例教程

第8章 面向对象程序设计

本章导读

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程序设计的思想，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。

本章首先介绍面向对象程序设计基础，包括类的定义、实例的创建、方法的定义；然后介绍面向对象的三大特性——封装、继承和多态；最后通过典型案例，让读者进一步掌握面向对象程序设计的思路。



学习目标

- 理解面向对象程序设计思想
- 掌握定义类和创建类的实例的方法
- 掌握类中变量和方法的应用
- 掌握构造方法和析构方法的应用
- 理解类成员和实例成员的区别
- 掌握面向对象的三大特性（封装、继承和多态）及相关知识的应用
- 理解类方法和静态方法的概念

The background of the slide features an abstract design composed of overlapping, semi-transparent red polygons of various shades, creating a dynamic, layered effect on the left side of the frame.

8.1 面向对象程序设计入门

8.2 类的定义与使用

8.3 类成员和实例成员

8.4 封装

8.5 继承

8.6 多态

8.7 类方法和静态方法

8.8 典型案例—猫狗大战

The background of the slide is an abstract composition of various shades of red and pink. It features a complex pattern of overlapping triangles and polygons, creating a sense of depth and movement. The colors range from deep, dark reds to lighter, almost white pinks, with some areas showing a gradient effect. The overall style is modern and geometric.

8.1 面向对象程序设计入门

8.1 面向对象程序设计入门

例：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。

```
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
```

choose(stu)

程序运行效果

```
{'num': '201801', 'name': 'Jack', 'credit': 14, 'course': ['Python', 'C', 'Jack']}
[Finished in 0.2s]
```



Line 9, Column 12

Spaces: 4

Python

8.1 面向对象程序设计入门

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：其特征包括学号、姓名、总学分和所选课程，行为包括选课；

课程类：其特征包括课程编号、课程名和学分。

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

- ◆ 8.2.1 类的定义
- ◆ 8.2.2 创建类的对象
- ◆ 8.2.3 self参数
- ◆ 8.2.4 构造方法
- ◆ 8.2.5 析构方法

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。

定义类的基本语法格式如下：

```
class 类名:  
    类体
```

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.....')
```

8.2.2 创建类的对象

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])
```

例：为前面定义的Car类创建一个car_1对象，并访问类中的变量和方法。

```
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.....')
```

```
car_1 = Car()                    #创建一个对象，并用变量car_1保存它的引用
```

```
car_1.run()                      #调用run()方法
```

```
print('车的价格是：',car_1.price) #访问类中的变量
```

```
车在行驶中.....  
车的价格是：  
[Finished in
```

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。

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。

例：self的使用。

```
class Car:
    def colour(self,col):                #定义赋值颜色方法
        self.col=col                    #赋值
    def show(self):                      #定义显示颜色方法
        print('The color of the car is %s.'%self.col)    #输出颜色

car_1 = Car()                          #创建对象car_1
car_1.colour('red')                     #调用方法
car_2 = Car()                          #创建对象car_2
car_2.colour('white')                  #调用方法
car_1.show()                           #调用方法
car_2.show()                           #调用方法
```

注意

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。

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, 从而实现对象进行初始化的操作。

例: 使用构造方法。

#定义类

```
class Car:
```

#构造方法

```
    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()
```

#创建对象

```
BMW.run()
```

#调用方法

程序运行效果

```
4个轮子的蓝色车在行驶中.....  
[Finished in 0.2s]
```

Line 12, Column 1

8.2.4 构造方法

例：使用带参构造方法。

#定义类

```
class Car:
```

#构造方法

```
    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

程序运行效果

```
4个轮子的红色车在行驶中.....  
4个轮子的白色车在行驶中.....  
[Finished in 0.3s]
```

Line 13, Column 46

8.2.4 构造方法

例：用面向对象程序设计思路编写程序，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。

```
#定义学生类
class Stu:
    #构造方法定义学生属性
    def
__init__(self,num,name,credit,cours
e):
    self.num = num
    self.name = name
    self.credit = credit
    self.course = course
#定义学生选课方法
def choose(self,c):
    self.credit+=c.credit
    self.course.append(c.name)
```

```
#定义课程类
class Cou:
    #构造方法定义课程属性
    def
__init__(self,num,name,credit):
    self.num = num
    self.name = name
    self.credit = credit
```

8.2.4 构造方法

```
stu_1 = Stu('201801','Jack',0,[])      #创建学生1
stu_2 = Stu('201802','Tom',3,['Math']) #创建学生2
cou_1 = Cou('01','Python',3)           #创建课程1
cou_2 = Cou('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)
```

```
学号: 201801 姓名: Jack 总学分: 3 所选课程 ['Python']
学号: 201802 姓名: Tom 总学分: 7 所选课程 ['Math', 'C']
[Finished in 0.2s]
```



Line 26, Column 1

Tab Size: 4

Python

8.2.5 析构方法

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。

例：比较下面两个程序，分析输出结果。

```
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---')
```

```
---构造方法被调用---  
---程序结束---  
---析构方法被调用---
```

```
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---')
```

```
---构造方法被调用---  
---析构方法被调用---  
---程序结束---
```

The background of the slide is an abstract composition of various shades of red and pink. It features overlapping, semi-transparent geometric shapes, primarily triangles and polygons, which create a sense of depth and movement. The colors range from deep, dark reds to lighter, almost white pinks, with the most intense colors concentrated on the left side of the frame.

8.3 类成员和实例成员

8.3 类成员和实例成员

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。

提示

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。

8.3 类成员和实例成员

例：类成员和实例成员示例。

#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

print(Car.colour)

```
150000 150000 红色
4 Audi Audi
[Finished in 0.2s]
```

8.3 类成员和实例成员

例：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

```
100000 150000  
[Finished in 0.2s]
```

8.4 封装

8.4 封装

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。



8.4 封装

私有化方法：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。

例：私有化数据成员和方法。

注意

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。

```
class A:                                #定义类
    def __init__(self):                  #定义私有变量并赋值为10
        self.__X = 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
```

```
print(a._A__X)    #通过类名访问私有变量
a._A__foo()       #通过类名调用私有方法
```

程序运行效果

```
Traceback (most recent call last):
  File "E:\Python代码\第8章\8-10.py", line 7, in <module>
    print(a.__X)                                #输出私有变量值
AttributeError: 'A' object has no attribute '__X'
[Finished in 0.4s with exit code 1]
```

9 lines, 207 characters selected

Tab Size: 4

Pytl

```
10
from A
[Finished in 0.4s]
```

Line 8, Column 5

8.4 封装

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。

例：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):                  #定义私有变量并赋值为10
        self.__X = 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
```

程序运行效果

```
from A
10
[Finished in 0.4s]
```

12 lines, 316 characters selected

8.5 继 承

- ◆ 8.5.1 单继承
- ◆ 8.5.2 多继承
- ◆ 8.5.3 重写父类方法与调用父类方法

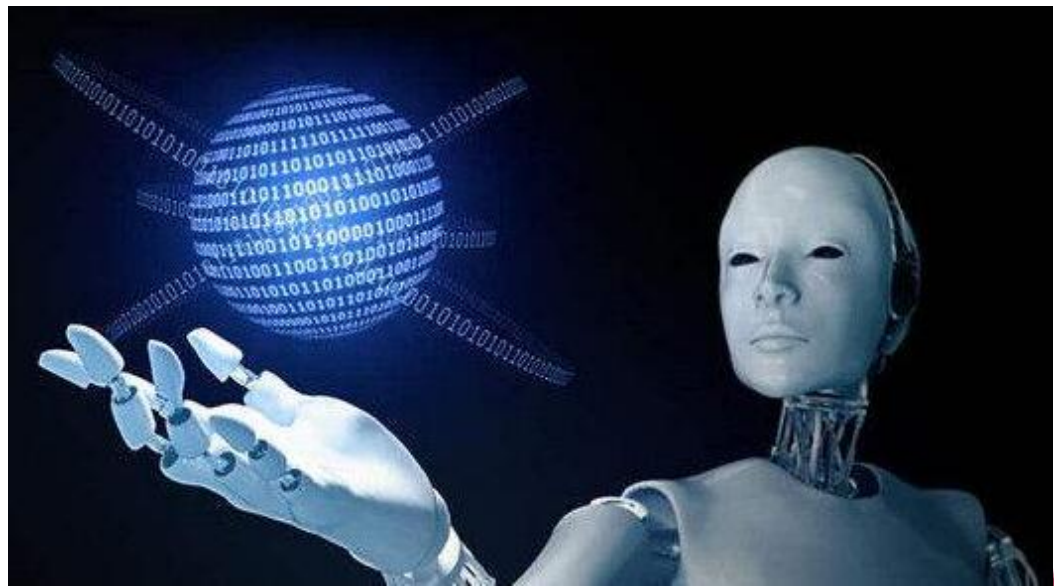
8.5.1 单继承

在程序中，继承描述的是事物之间的从属关系。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。

在Python中，当一个子类只有一个父类时称为单继承。子类的定义如下所示：

```
class 子类名(父类名):
```

子类可以继承父类的所有公有成员和公有方法，但不能继承其私有成员和私有方法。



8.5.1 单继承

例：单继承示例。

#定义一个父类

```
class Person:
```

```
    name = '人'
```

```
    age = 30
```

```
    __age=30
```

#定义方法用于输出

```
    def speak(self):
```

```
        print ('%s 说: 我 %d 岁。' %(self.name,self.age))
```

#定义一个子类

```
class Stu(Person):
```

#定义方法用于修改名字

```
    def setName(self, newName):
```

```
        self.name = newName
```

#定义方法用于输出

```
    def s_speak(self):
```

```
        print ('%s 说: 我 %d 岁。' %(self.name,self.age))
```

```
student = Stu()
```

```
print ('student的名字为:',student.name)
```

```
print ('student的年龄为:',student.age)
```

```
student.s_speak()
```

```
student.setName('Jack')
```

```
student.speak()
```

#创建学生对象

输出学生名字

程序运行效果

```
student的名字为: 人
```

```
Traceback (most recent call last):
```

```
  File "E:\Python代码\第8章\8-12.py", line 15, in <module>
```

```
    print ('student的年龄为:',student.__age)
```

```
AttributeError: 'Stu' object has no attribute '__age'
```

```
[Finished in 0.4s with exit code 1]
```

#输出学生年龄

Line 19, Column 1

Spaces: 4

Python

8.5.2 多继承

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。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...):
```

例：多继承示例。

#定义沙发父类

```
class Sofa:
    def printA(self):
        print ('----这是沙发----')
```

#定义床父类

```
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
```

程序运行效果

```
----这是沙发----
----这是床----
----这是沙发床----
[Finished in 0.4s]
```

Line 17, Column 1

8.5.2 多继承

注意

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

► 1. 重写父类方法

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**。

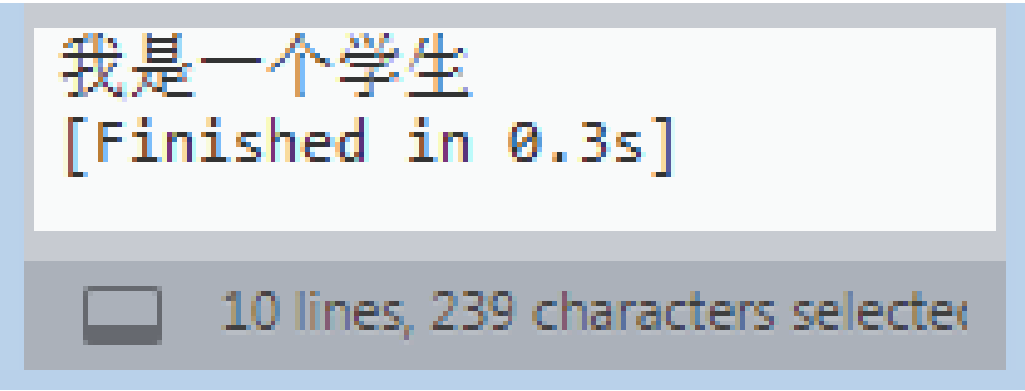
例：重写父类的方法示例。

```
#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
```

程序运行效果



```
我是一个学生
[Finished in 0.3s]
```

10 lines, 239 characters selected

8.5.3 重写父类方法与调用父类方法

► 2. 调用父类方法

如果需要在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。

例：子类调用父类的方法示例。

#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))
```

程序运行效果

```
姓名:Jack, 性别:Male, 成绩: 90  
[Finished in 0.5s]
```

13 lines, 364 characters selected

The background of the slide is an abstract composition of various shades of red. On the left side, there is a complex, overlapping pattern of triangles and polygons, creating a sense of depth and movement. This pattern transitions into a smoother, lighter red area on the right. The overall effect is modern and dynamic.

8.6 多 态

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。

在面向对象方法中一般是这样描述多态性的：向不同的对象发送同一条消息，不同的对象在接收时会产生不同的行为（即方法）。

例：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)

8.6 多态

#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)
```

程序运行效果

```
I am a Person, my name is Jack  
I am a Student, my name is Tom  
I am a Teacher, my name is Lily  
[Finished in 0.4s]
```



Line 33, Column 1

Spaces: 4

#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)
```

8.7 类方法和静态方法

◆ 8.7.1 类方法

◆ 8.7.2 静态方法

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。

例：类方法的使用。

#定义类

```
class People:
```

```
    country = 'china'
```

#类方法，用classmethod来进行修饰

```
    @classmethod
```

```
    def getCountry(cls):
```

```
        return cls.country
```

```
p = People()
```

```
print(p.getCountry())
```

```
print(People.getCountry())
```

#定义类成员并赋值

#返回类成员的值

#创建对象

#通过实例名调用

#通过类名调用

程序运行效果

```
china
china
[Finished in 0.4s]
```

Line 11, Column 1

Tab

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。

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。

8.7.2 静态方法

例：静态方法的使用。

#定义类

```
class Test:
```

#静态方法，用@staticmethod进行修饰

```
    @staticmethod
```

```
    def s_print():
```

```
        print('----静态方法----')
```

```
t = Test()
```

```
Test.s_print()
```

```
t.s_print()
```

#创建对象

#通过类名调用

#通过对象名调用

程序运行效果

```
----静态方法----  
----静态方法----  
[Finished in 0.3s]
```

9 lines, 188 characters s



8.8 典型案例

猫狗大战

【例】 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。

猫狗大战

#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)
```

猫狗大战

```
#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)
```

猫狗大战

#创建实例

```
cat_1 = Cat('Mily','波斯猫',30,1500)
dog_1 = Dog('Lucky','哈士奇',50,900)
cat_1.die()
dog_1.die()
print('-----开始战斗-----')
cat_1.attack(dog_1)
dog_1.bite(cat_1)
cat_1.die()
dog_1.die()
for i in range(29):
    cat_1.attack(dog_1)
dog_1.die()
cat_1.eat()
cat_1.die()
```

#创造了一只实实在在的猫
#创造了一只实实在在的狗
#输出猫的当前状态
#输出狗的当前状态

#猫攻击狗一次
#狗攻击猫一次
#输出猫的当前状态
#输出狗的当前状态
#循环实现，猫攻击狗29次

#输出狗的当前状态
猫吃东西一次
#输出猫的当前状态

程序运行效果

```
Mily 的生命值还有 1500
Lucky 的生命值还有 900
-----开始战斗-----
Mily 的生命值还有 1450
Lucky 的生命值还有 870
Lucky 已被杀死!
Mily 的生命值还有 1500
[Finished in 0.4s]
```

Line 61, Column 1

Spaces: 4

感谢您的观看

