

《交通大数据技术》



2024/6/14

交通大数据技术

崔志勇
交通科学与工程学院
2023年4月

目录导航



✓ 1 类和对象



1.1 类与对象的概念

1.2 Python中类的定义

1.3 对象：类的实例化

2 对类的进一步认识

2.1 关于初始化程序：__init__

2.2 关于参数：self

2.3 关于类的属性

3 类的继承

3.1 Python里类的继承

3.2 在子类中改写父类的方法

3.3 内置函数super()

4 Python中类的导入

4.1 类的导入

类和对象

1.1 类与对象的概念

例如，一说到球，人的头脑中马上就会把它想象成一种圆形的物体，它的大小由半径决定，它的外表可以有不同颜色；对球施以外力后，它会往远处滚动，或上下来回弹跳。

但是，我们能真正拿出一个叫“球”的物体吗？不可能，世上没有一个东西具体叫作“球”，只有诸如“乒乓球”“篮球”“排球”等，才是实实在在的“球”。

因此，“球”与“乒乓球”等不是一回事。

“球”，是对诸如“乒乓球”“篮球”“排球”等的一种**抽象**；而“乒乓球”“篮球”“排球”等，则是对“球”的一个个**具体化**。“球”是圆的，大小由半径决定，外表可以有不同颜色，施加外力后会滚动或弹跳，这些抽象的特征和行为，都是从“乒乓球”“篮球”“排球”等里面抽取出来的；而每一种具体的球，又都必须具有抽取出来的这些特征和行为。

于是，人们把抽象的“球”视为是世间的一种数据类型——“类”；把具体的“乒乓球”“篮球”“排球”等，视为这种类的“实例化”，或者说是该类的一个个“对象”。



类和对象

1.1 类与对象的概念

可以举出很多类和对象的例子。例如，“动物”是类，“狮子”“羚羊”“狗”等是动物类的对象。但从另一个角度讲，“狗”是一个类，“猎狗”“宠物狗”“藏獒”等又是狗这类的对象。“宠物狗”又可以再细化成一个类，“京巴狗”“约克夏”“哈士奇”等是宠物狗类的对象。人们就是以这种“分门别类”的办法，一点点深入，一点点细化，逐渐对世间万物进行了解、描述和研究。

从上面的讨论可以得出下面的两点共识。

- ◆ “类”是一种**抽象**的概念，它代表了现实世界中某些事物的**共有特征和行为**。
 - ◆ “对象”是对**类的实例化**，每个对象都会自动具有所属类的通用特征和行为。当然，根据需要，对象也可以**具有自己独特的个性特征与行为**。
-
- ◆ 大家想一想，编程过程中为什么需要抽象与实例化？

目录导航



1 类和对象

1.1 类与对象的概念



1.2 Python中类的定义

1.3 对象：类的实例化

2 对类的进一步认识

2.1 关于初始化程序：__init__

2.2 关于参数：self

2.3 关于类的属性

3 类的继承

3.1 Python里类的继承

3.2 在子类中改写父类的方法

3.3 内置函数super()

4 Python中类的导入

4.1 类的导入

1.2 Python中类的定义

Python中的类是借助**关键字class**来定义的。具体语法如下：

```
class <类名>():  
    <成员变量>  
    <成员函数>
```

- 它由两大部分组成，以**class**开头、“：”冒号结束的第1行被称为“类头”，缩进的<成员变量>与<成员函数>被称为“类体”。
- 其中，<类名>必须符合Python对变量取名所做的规定。进一步地，**为了便于区分，Python约定<类名>的第1个字母必须大写。**
- **<成员变量>**就是类的成员属性，简称“属性”。<成员变量>包含在描述该类成员（也就是对象）共有的抽象特征时涉及的各种变量。

类和对象

1.2 Python中类的定义

- **<成员函数>**即成员方法，简称“方法”。<成员函数>包含描述该类成员（也就是对象）所具有的抽象行为或动作的各种函数。
- 在一个“类”的定义里，<成员变量>和<成员函数>当然可以不止一个。

下面直接举出3个Python中类的定义的例子，只要了解了前面关于函数的那些知识，那么大致理解例子中所要描述的内容便不会有太大的难度。

例1 定义一个“狗”类：

```
class Dog(): #定义类Dog
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def sit(self):
        print(self.name.title() + 'is now sitting.')
    def r_over(self):
        print(self.name.title() + 'rolled over!')
```

1.2 Python中类的定义

该类的名字是Dog，它的首字母为大写，符合Python对类的定义所做的约定要求。在缩进类体里面，包含3个以def开头的函数。

- 第1个是名为__init__的特殊方法，它是“**初始化程序**”。在创建对象时，用来为代表属性的诸变量赋初值。例如，类Dog里有3个形参，即self、name、age，除了特殊的self后面会专门讲述它的作用外，name和age两个属性的初值都会由创建对象（即实例化）语句提供的实参传递过来。
- 第2个是名为sit(坐)的函数，它是描述狗会“坐”的这种行为的方法。该方法只有一个特殊参数self，功能是把狗名的第1个字母改为大写（由调用函数title()来实现），并输出“is now sitting”信息。如果在此能够插入一些动画，那么就应该出现一条狗蹲坐在那里、舌头吐在嘴外面呼呼喘气的情景。
- 第3个是名为r_over(奔跑)的函数，它是描述狗会“跑”的这种行为的方法。该方法只有一个特殊参数self，功能是把狗名的第1个字母改为大写（由调用函数title()来实现），并输出“rolled over!”信息。如果这里能够插入动画，那么就应该出现一条狗向前奔跑的情景。

类和对象

1.2 Python中类的定义

例2 定义一个“雇员”类：

```
class Employee():      #定义类Employee
    e_count = 0

    def __init__(self, name, salary):  #初始化程序
        self.name = name
        self.salary = salary
        Employee.e_count += 1

    def dis_employee(self):              #输出雇员信息
        print( 'Name : ', self.name, ', Salary: ', self.salary)
```

- 该类的名字是Employee，它的首字母为大写，符合Python的约定要求。在缩进的类体里，先是为一个变量e_count赋初值0，**由于它在类体的最前面，所以这个变量在整个类的定义里都有效**，也就是类里定义的函数都可以使用它。
- 该类里，第1个是名为**__init__的初始化程序**，我们后面会对它进行专门的讲述。除去特殊参数self外，它的两个属性name、salary的初始值，仍然是在创建对象（即实例化）时，靠创建语句通过实参传递过来。在这个方法里，还对变量e_count进行计数操作。
- 该类体里的第2个函数是dis_employee()，只有一个特殊参数self，功能是输出雇员的名单。

1.2 Python中类的定义

例3 定义一个关于“圆”类：

```
import math                #导入标准函数库math
class Circle():            #定义类Circle
    def __init__(self,radius=14):
        self.radius=radius
    def d_diameter(self):   #返回直径的方法
        return 2*self.radius
    def c_perimeter(self):  #返回圆周长的方法
        return 2*self.radius*math.pi
    def r_area(self):       #返回圆面积的方法
        return self.radius*self.radius*math.pi
```

该类的名字是Circle，它的首字母为大写，符合Python的约定要求。注意，在整个类定义的外面，有一条导入语句“import math”，表明该类里要用到有关数学计算的标准函数库。

缩进类体里有4个函数：

- 第1个仍然是名为__init__的初始化程序，它除了特殊参数self外，还有一个名为radius（半径）的默认参数，取默认值为14。
- 第2个方法名为d_diameter，只有一个特殊参数self，功能是返回圆的直径；
- 第3个方法名为c_perimeter，只有一个特殊参数self，功能是返回圆周长；
- 第4个方法名为r_area，只有一个特殊参数self，功能是返回圆面积。

类和对象

1.2 Python中类的定义

比较以上给出的3个类的定义，可以得到下面这样的一些对类的认识：

01

OPTION

当使用前面那些单一数据类型无法描述出世间的事物时，就应该考虑采用“类”这种数据类型；

02

OPTION

类名字的第1个字母，应该遵循大写的规则；

03

OPTION

类定义中，可以有一个名为 `__init__` 的初始化程序（它实际上也是一个方法），在它的里面聚集了抽象出来的属性；

04

OPTION

类中的各种方法里，即使是初始化程序，都有一个特殊的参数 `self`，它总是被放在方法形参表的第1个位置处，哪怕方法里没有任何别的参数，这个 `self` 都是不可或缺的。

综上所述，Python的类的定义，是把解决问题时需要用到的变量（即属性）和函数（即方法）组合在了定义中。通常，称这种组合为“封装”，前面的那些数据类型，都不可能实现这种把变量和方法封装在一起的效果。

目录导航



✓ 1 类和对象

1.1 类与对象的概念

1.2 Python中类的定义



1.3 对象：类的实例化

2 对类的进一步认识

2.1 关于初始化程序：__init__

2.2 关于参数：self

2.3 关于类的属性

3 类的继承

3.1 Python里类的继承

3.2 在子类中改写父类的方法

3.3 内置函数super()

4 Python中类的导入

4.1 类的导入

1.3 对象：类的实例化

定义了类，就可以进行类的实例化工作了。也就是说，可以从“抽象”转为“具体”，创建出一个个现实世界中的对象来。创建的对象可以通过“对象名.成员”的方式，访问类中所列的<成员变量>和<成员函数>。

例2 中有关雇员的实例化。例如在类Employee定义的基础上，编写程序主体如下：

<code>emp1 = Employee('Zara', 2000)</code>	<code>#创建一个名为emp1的对象</code>
<code>emp2 = Employee('Manni', 5000)</code>	<code>#创建一个名为emp2的对象</code>
<code>emp1.dis_employee()</code>	<code>#emp1调用方法dis_employee()</code>
<code>emp2.dis_employee()</code>	<code>#emp2调用方法dis_employee()</code>
<code>print ('Total Employee %d' % Employee.e_count)</code>	<code>#输出雇员数</code>

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

✓ 2 对类的进一步认识



2.1 关于初始化程序：__init__

- 2.2 关于参数：self
- 2.3 关于类的属性

3 类的继承

- 3.1 Python里类的继承
- 3.2 在子类中改写父类的方法
- 3.3 内置函数super()

4 Python中类的导入

- 4.1 类的导入

对类的进一步认识

2.1 关于初始化程序：__init__

下面罗列几个编写初始化程序时需要了解的问题。

1. 初始化程序的正确写法：两条下划线

初始化程序的名字，是以两条下划线开始，以两条下划线结束的。如果哪一边只有一条下划线，那么就会输出图6-3所示的出错信息，表明创建对象时，实参无法传递给类，导致对象得不到参数。这种错误提示得非常隐晦，让人难以发现到底是在哪里出了问题，其实只是因为丢失了一条小小的下划线“—”。

2. 初始化程序，不一定非要排在类定义的最前面

- 初始化程序的名字是固定的，只能写成“__init__”。因此，只要保证它出现在类定义语句块的里面就行，至于是块中的第1个方法（如例1~3那样），还是第几个方法，完全无关紧要；甚至如果不需要传递什么参数，在类的定义中不编写初始化程序也没有什么关系。
- Python在创建一个新的对象（即实例化）时，总是用这个名字去匹配类中所包含的方法名，找到这个名字，就执行并仅执行它一次，以完成初始化的工作；如果没有找到它，那就不做初始化工作。

对类的进一步认识

2.1 关于初始化程序：__init__

3. 初始化程序中name与self.name的不同含义

例如在例6-2的类定义中，初始化程序如下：

```
def __init__(self, name, salary):  
    self.name = name  
    self.salary = salary  
    Employee.e_count += 1
```

在它的里面除self外，还有两个位置参数：name、salary。一进入初始化程序，就执行：

```
self.name = name  
self.salary = salary
```

把接收到的实参（在name和salary里），赋予变量self.name和self.salary。因此，这两条语句右边的name和salary，是初始化程序接收到的传递过来的实参，而左边的self.name与self.salary是两个变量，由它们接收并存放这个新建对象的具体属性值。

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

✓ 2 对类的进一步认识

- 2.1 关于初始化程序：__init__
- 2.2 关于参数：self
- 2.3 关于类的属性

3 类的继承

- 3.1 Python里类的继承
- 3.2 在子类中改写父类的方法
- 3.3 内置函数super()

4 Python中类的导入

- 4.1 类的导入

对类的进一步认识

2.2 关于参数：self

例4 下面是一个用类编写的极为简单的图书查询的例子（为方便讲述，各部分程序分散列在下面）：

```
#定义一个字典catalog
catalog={'aa':3,'bb':0,'cc':5,'dd':2,'ee':4,'ff':1,'gg':8,'hh':3}
#定义类Book
class Book():
    global catalog ← 全局变量
    def __init__(self, name):
        self.name=name ← 类变量
    #下面是类中的方法loop(), 供创建的对象调用
    def loop(self, name): ← 类函数
        for key, value in catalog.items():
            if key==name :
                if value!=0:
                    print('There is the book!')
                elif value==0:
                    print('I\'m sorry that the book has been borrowed!')
            else:
                print('It\'s a pity that there is no book!')
        break
```

- 整个程序开始，先定义一个字典catalog，书名aa、bb、cc等是字典的键；现存书的数量是字典的值。由于字典catalog是在整个程序外定义的，进入类后才使用它，这样可能会引起不必要的误解，所以在类的里面用语句global catalog表明类里出现的catalog就是外面定义的全局变量catalog。
- 整个类Book里有3个函数。

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

✓ 2 对类的进一步认识

- 2.1 关于初始化程序：__init__
- 2.2 关于参数：self
- 2.3 关于类的属性

3 类的继承

- 3.1 Python里类的继承
- 3.2 在子类中改写父类的方法
- 3.3 内置函数super()

4 Python中类的导入

- 4.1 类的导入

对类的进一步认识

2.3 关于类的属性

1. 给类的属性指定默认值

借助上面的例4，在类Book的定义里，给出一个取默认值的属性“self.word='12345'”，并利用它作为口令的初始值。

```
catalog={'aa':3,'bb':0,'cc':5,'dd':2,'ee':4,'ff':1,'gg':8,'hh':3}
class Book():
    global catalog
    def __init__(self,name):
        self.name=name
        self.word='12345'    #设置属性word默认的初始值，它不出现在参数表里
```

2.3 关于类的属性

2. 对类的默认属性值的修改

每种类都有自己的属性和方法，属性是从类的实例化中抽象出来的特征，由变量来描述；方法是从类的实例化中抽象出来的行为，由函数来描述。因此，修改类的属性值，就是修改描述它的变量的值，这是非常容易做到的事情。

修改默认属性值最直接的办法，是在创建的实例里进行。例如上面原先设置的默认属性值是：

```
self.word='12345'
```

只要在实例化studentB的程序里，增加一条语句：

```
studentB.word='67890'
```

于是，现在创建对象studentB时的程序就改写为：

```
bnm=input('Enter book name:')  
studentB=Book(bnm)  
studentB.loop(bnm)  
studentB.word='67890'           #对默认属性值的修改
```

2.3 关于类的属性

3. 属性的增、删、改

定义一个类并创建了它的对象后，可以对定义的类进行添加新属性、修改原有属性、删除已有属性的操作。下面的程序中，定义了一个名为Car的类，它有一个形参col，有两个默认参数：price=100000和name="QQ"。

#定义类Car:

```
class Car:
    def __init__(self,col):
        self.color=col
        self.price=100000
        self.name='QQ'
```



#程序主体:

```
car1=Car('Red')
print(car1.name,car1.color,car1.price)
car1.price=110000
car1.color='Yellow'
car1.time='2024/08'
print(car1.name,car1.color,car1.price,car1.time)
del car1.color
print(car1.name,car1.color,car1.price,car1.time)
print('End')
```

对类的进一步认识

2.3 关于类的属性

程序主体里，创建了一个名为car1的对象，随之通过语句：

```
print(car1.name,car1.color,car1.price)
```

输出信息“QQ Red 100000”。接着有语句：

```
car1.price=110000  
car1.color='Yellow'  
car1.time='2016/08'
```

前两条是修改已有属性price和color，后一条语句是增加新属性time（出厂时间）。这样的操作，使得程序主体中的第2条输出语句输出了这辆QQ的新信息。最后，通过语句：

```
del car1.color
```

删除对象car1的属性color。第3条输出语句的执行，就会产生出错信息：

```
AttributeError: 'Car' object has no attribute 'color'
```

对类的进一步认识

2.3 关于类的属性

表示试图删除对象没有的属性，操作失败。图中记录了程序的整个执行过程。



```
D:\>python test2.py
QQ Red 100000
QQ Yellow 110000 2016/08
Traceback (most recent call last):
  File "test2.py", line 14, in <module>
    print(car1.name,car1.color,car1.price,car1.time)
AttributeError: 'Car' object has no attribute 'color'
```

4. 属性的保护

如上所述，在程序主体中可以对对象的属性进行增、删、改等操作。这种做法看似很方便，用起来也得心应手，但却违反了**类的封装原则：数据使用的安全性**。对象能够在类定义的外部随便访问其数据属性，就有可能修改它们，影响到类中提供的各种方法的正确运行，因为在类定义里给出的方法里面可能会用到属性变量。

对类的进一步认识

2.3 关于类的属性

类定义中的属性变量**没有了私密性**，就可能会带来各种想象不到的麻烦。考虑到这些问题，Python对类的属性提供了一种自我保护的简单办法。具体做法如下。

01

OPTION

在需要具有私密性的属性变量名前，加上双下划线。

02

OPTION

在类定义里增加供程序设计人员访问属性变量的接口。

仍以上面所定义类Car来加以说明。

```
class Car:
    def __init__(self,col):
        self.__color=col
        self.price=100000
        self.name='QQ'
    def pri(self):
        print(self.__color)
```

```
car1=Car('Red')
print(car1.name,car1.price)
car1.pri()
print('End')
```



对类的进一步认识

2.3 关于类的属性

假定要保护属性col的使用，不允许在程序主体内随意地访问它，那么可以在类定义中，**在接收col的形参名前增加双下划线**，即：

```
self.__color
```

另一方面，在类定义中写一个输出汽车颜色的方法：

```
def pri(self):  
    print(self.__color)
```

以便在程序主体内能够输出汽车的颜色。

先看一下在形参名前加上双下划线后的作用。这时整个程序是这样的：

```
class Car:  
    def __init__(self,col):  
        self.__color=col  
        self.price=100000  
        self.name='QQ'
```



```
car1=Car('Red')  
print(car1.name,car1.price,car1.color)  
print('End')
```

对类的进一步认识

程序编写如下：

```
class Car:
    def __init__(self,col):
        self.__color=col
        self.price=100000
        self.name='QQ'
    def print_color(self):
        print(self.__color)
```

```
car1=Car('Red')
print(car1.name,car1.price)
car1.print_color()
print('End')
```

#增加的新函数

#去除直接访问属性color的内容
#增加调用类函数pri()的语句



对类的进一步认识

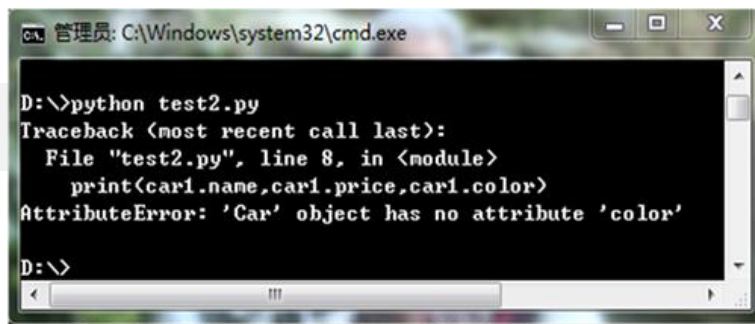
2.3 关于类的属性

运行该程序，结果如图所示，给出出错信息：

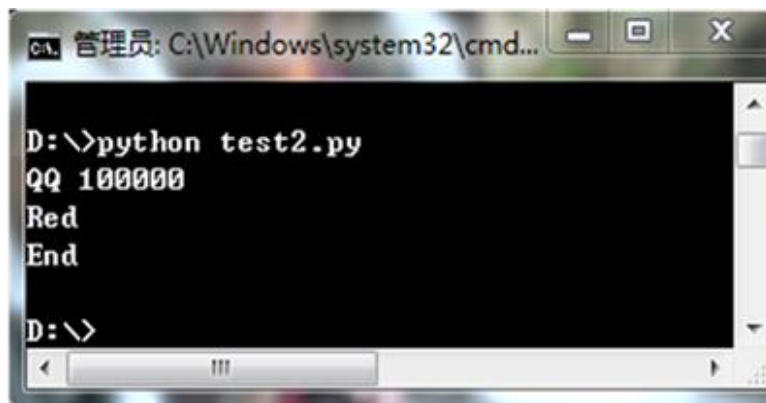
AttributeError: 'Car' object has no attribute 'color'

之所以会这样，是因为类定义里，在属性color变量名前加上了双下划线，Python对该变量进行了保护，不允许在类外通过“car1.color”直接访问该属性。

为了能够在类定义外访问属性color，可以在类定义里给出访问该变量的函数，例如 `print_color()`，然后在程序主体里，把语句“`print(car1.name,car1.price,car1.color)`”改写成语句“`print(car1.name,car1.price)`”，增加语句“`car1.print_color()`”。这样再运行，程序就正确了。



```
管理员: C:\Windows\system32\cmd.exe
D:\>python test2.py
Traceback (most recent call last):
  File "test2.py", line 8, in <module>
    print(car1.name,car1.price,car1.color)
AttributeError: 'Car' object has no attribute 'color'
D:\>
```



```
管理员: C:\Windows\system32\cmd...
D:\>python test2.py
QQ 100000
Red
End
D:\>
```

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

2 对类的进一步认识

- 2.1 关于初始化程序：__init__
- 2.2 关于参数：self
- 2.3 关于类的属性



3 类的继承



3.1 Python里类的继承

- 3.2 在子类中改写父类的方法
- 3.3 内置函数super()

4 Python中类的导入

- 4.1 类的导入

类的继承

3.1 Python里类的继承

- **“继承”，即承上启下。**一个类，就是把它所要描述事物的特征（即属性）和行为（即方法）包装（也就是封装）在了一起。
- 在继承关系中，已有的、设计好的类被称为**“父类”或“基类”**，新设计的类被称为**“子类”或“派生类”**。

只要遵守Python里“继承”的规则，在程序设计中完成继承是不困难的。例如，简单定义了一个名为People的类，代码如下：

```
class People():  
    def __init__(self,name,nationality):  
        self.name=name  
        self.nationality=nationality
```



```
def talk(self):  
    print('Communicate in language')  
  
def walk(self):  
    print('Walk upright with your legs!')
```

3.1 Python里类的继承

该类名为People，它有3个属性：self、name、nationality（国籍）。类中定义了两个方法，一是talk，调用它时输出信息“Communicate in language”（用语言交流）；二是walk，调用它时输出信息“Walk upright with your legs”（两腿直立行走）。

由它派生出来的类，都应该有这样的属性和方法，也就是说都会继承这些属性和方法。下面定义一个名为Ch_people（中国人）的类：

```
class Ch_people(People):    #定义一个子类，名为Ch_people
    pass                    #表示该类不做什么事情
```

- 在该类名字后面的括号里，填写了一个参数“People”，表明它是从类People派生出来的，要继承它的“衣钵”，也就是**继承它的所有属性和方法**。
- 该类的类体里只有一条语句“pass”，表示虽然定义了这个新类，但它什么事情也不做，它没有自己新的属性，也没有自己新的方法，纯粹就是一个“空”的类。

类的继承

3.1 Python里类的继承

定义了类People和类Ch_people后，编写如下程序：

```
#创建类People的一个对象peopA
peopA=People('Zong da hua','china')
print(peopA.name,peopA.nationality)
peopA.talk()
peopA.walk()
#创建类Ch_people的另一个对象peopB
peopB=Ch_people('Zong da hua','china')
print(peopB.name,peopB.nationality)
peopB.talk()
peopB.walk()
```

执行整个程序，结果如图所示。



```
管理员: C:\Windows\system32\cmd.exe

D:\>python test1.py
Zong da hua china
Communicate in language
Walk upright with your legs!
Zong da hua china
Communicate in language
Walk upright with your legs!
D:\>
```

这是对象peopA调用类方法的执行结果

这是对象peopB调用类方法的执行结果

3.1 Python里类的继承

- 上半部分是类People的对象peopA调用方法的结果；
- 下半部分是类Ch_people的对象peopB调用方法的结果。
- 由于类**Ch_people是由类People派生出来的**，它**继承了类People的一切**，自己又不多做任何事情，所以这两个对象的运行结果是完全一样的。

为了便于描述，常会使用如下的名称。

- **基类**：也称“**父类**”，表示这种类是可以被别的类继承的。
- **派生类**：也称“**子类**”，表示这种类是一个继承别的类的类。

例如，上面定义的类People，相对于类Ch_people来说，就是一个基类，即父类；而定义的类Ch_people，则是一个从类People派生出来的类，因此是一个子类。

总结，在Python里，要从一个类里派生出另一个类，也就是一个类要继承另外一个类，**只需将父类的名字放入该类定义的括号里即可**。由于在子类的定义里只有一条pass语句，所以它将会从它的父类那里继承所有的特征和行为。

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

2 对类的进一步认识

- 2.1 关于初始化程序：__init__
- 2.2 关于参数：self
- 2.3 关于类的属性

✓ 3 类的继承

3.1 Python里类的继承



3.2 在子类中改写父类的方法

3.3 内置函数super()

4 Python中类的导入

4.1 类的导入

3.2 在子类中改写父类的方法

在继承机制下，允许**子类改写父类**中已经出现过的方法。通常，这种改写被称为“覆盖”。

仍以类Ch_people这个由类People派生出来的类为例。类Ch_people当然是用语言来交流思想的，但类Ch_people是用特定的、自己民族的语言来交流思想的。因此，在定义类Ch_people时，可以将原先类People中的方法：

```
def talk(self):  
    print('Communicate in language')
```

加以**改写**（也就是“覆盖”），例如改写为：

```
def talk(self):  
    print('Exchange ideas in Chinese!')
```

这样，类Ch_people的定义就可以是：

```
class Ch_people(People):  
    def talk(self):  
        print('Exchange ideas in Chinese!')
```

3.2 在子类中改写父类的方法

即先取消类Ch_people定义中的pass语句，然后再改写方法talk()。

现在，整个程序变为：

```
class People():
    def __init__(self,name,nationality):
        self.name=name
        self.nationality=nationality

    def talk(self):
        print('Communicate in language')

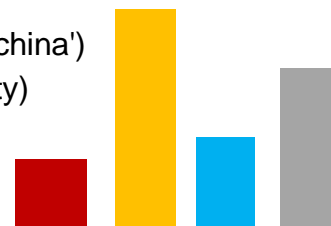
    def walk(self):
        print('Walk on two legs')
```



```
class Ch_people(People):
    def talk(self):    #改写父类中的同名方法talk()
        print('Exchange ideas in Chinese!')

peopA=People('Zong da hua','china')
print(peopA.name,peopA.nationality)
peopA.talk()
peopA.walk()

peopB=Ch_people('Zong da hua','china')
print(peopB.name,peopB.nationality)
peopB.talk()
peopB.walk()
```



类的继承

3.2 在子类中改写父类的方法

运行该程序，结果就有所不同了，如图所示。

这时在子类中，利用改写的初始化程序，就可以**使子类增添自己特有的属性**。例如，原先类People只有name和nationality两个属性，考虑到中国是一个多民族国家，希望在类Ch_people里，增加一个名为nation的属性。为此，可以在类Ch_people里，改写类People里的初始化程序，代码如下：

```
class Ch_people(People):  
    def __init__(self,name,nationality,nation):  
        self.name=name  
        self.nationality=nationality  
        self.nation=nation
```



```
管理员: C:\Windows\system32\cmd.exe  
  
D:\>python test8.py  
Zong da hua china  
Communicate in language  
Walk on two legs  
Zong da hua china  
Exchange ideas in Chinese! ← 把原先的方法talk()覆盖了  
Walk on two legs  
  
D:\>
```

3.2 在子类中改写父类的方法

当创建类Ch_people的对象时，Python就去执行该类如上的初始化程序，从创建处传递所需的实参给变量self.name、self.nationality、self.nation，完成初始化工作，以保证整个程序正常运行。

下面是整个程序的内容：

```
class People():
    def __init__(self,name,nationality):
        self.name=name
        self.nationality=nationality

    def talk(self):
        print('Communicate in language')

    def walk(self):
        print('Walk on two legs')
class Ch_people(People):
    def __init__(self,name,nationality,nation):
        self.name=name
        self.nationality=nationality
        self.nation=nation
```



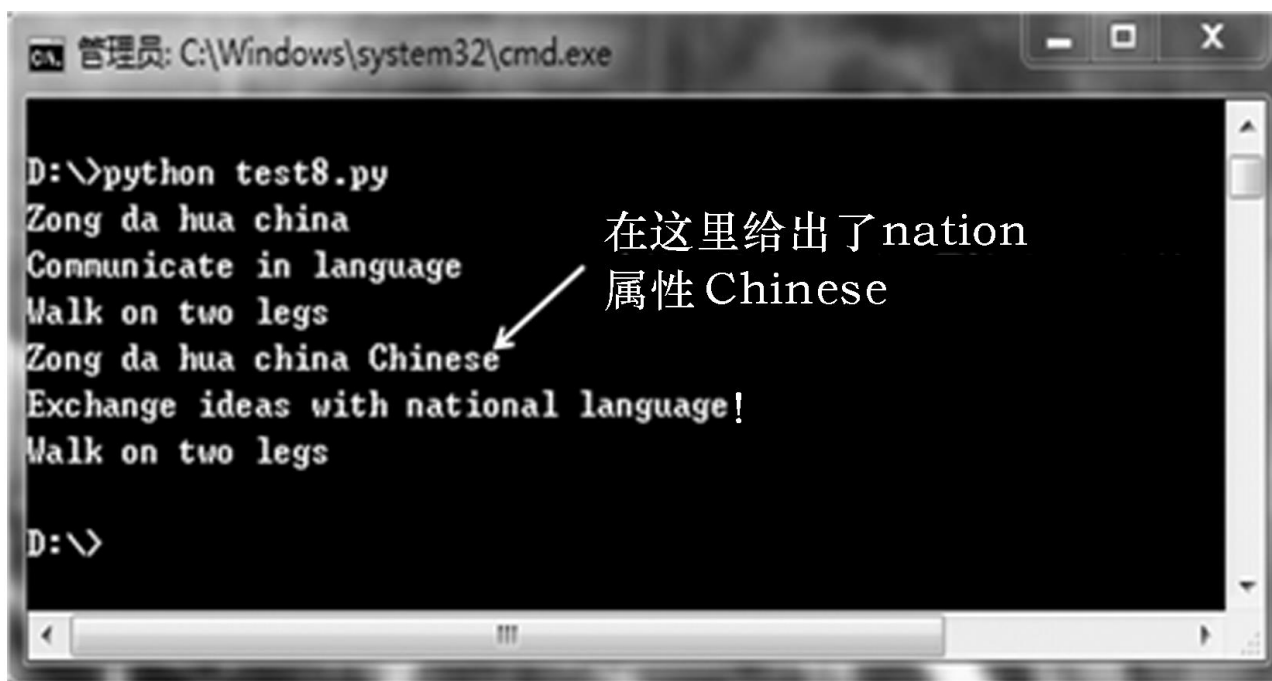
```
def talk(self):
    print('Exchange ideas with national language!')

peopA=People('Zong da hua','china')
print(peopA.name,peopA.nationality)
peopA.talk()
peopA.walk()

peopB=Ch_people('Zong da hua','china','Chinese')
print(peopB.name,peopB.nationality,peopB.nation)
peopB.talk()
peopB.walk()
```

3.2 在子类中改写父类的方法

运行该程序，结果就与图6-9又不一样了，如图



```
管理员: C:\Windows\system32\cmd.exe

D:\>python test8.py
Zong da hua china
Communicate in language
Walk on two legs
Zong da hua china Chinese
Exchange ideas with national language!
Walk on two legs

D:\>
```

在这里给出了nation
属性 Chinese

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

2 对类的进一步认识

- 2.1 关于初始化程序：__init__
- 2.2 关于参数：self
- 2.3 关于类的属性



3 类的继承

- 3.1 Python里类的继承
- 3.2 在子类中改写父类的方法
- 3.3 内置函数super()



4 Python中类的导入

- 4.1 类的导入

3.3 内置函数super()

实际应用中，肯定会有这样的情形发生：**子类改写了父类的某个方法，后面却还要用到已被改写的原先父类的那个方法。**Python的内置函数super()就可以解决这个问题，它建立起了子类与父类之间的联系，准确无误地在子类里找到父类，并实现对所需方法的调用。也就是说，**如果需要在子类中调用父类的方法，那么可以使用内置函数super()，或者通过“父类名.方法名()”的方式，来达到这一目的。**

例6-5 一个全部继承的例子：

```
class Parent():
    def speak(self):
        print('The voice of the father\'s voice!')
class Child(Parent):
    pass
dad=Parent()
son=Child()
dad.speak()
son.speak()
```



3.3 内置函数super()

例中先定义了一个名为Parent的类，它有一个方法speak()，功能是输出信息“The voice of the father's voice!”（父亲的说话声音）。接着定义一个名为Child的类，它无条件地继承了类Parent，因为它定义的括号里，有一个参数“Parent”，且类体里只有一条pass语句。

程序里共有4条语句：

- 第1条是dad=Parent()，它创建了一个名为dad的Parent对象；
- 第2条语句是son=Child()，它创建了一个名为son的Child对象；
- 第3条语句是对象dad调用方法speak()；
- 第4条语句是对象son调用方法speak()。

由于子类Child**全盘地继承了父类**，所以这两个对象调用方法speak()的结果，都是输出信息**“The voice of the father's voice!”**。

3.3 内置函数super()

例6 一个覆盖父类方法的例子。

把例5稍加修改，代码如下：

```
class Parent():
    def speak(self):
        print('The voice of the father\'s voice!')

class Child(Parent):
    def speak(self):
        print('The voice of the son\'s voice!')

dad=Parent()
son=Child()

dad.speak()
son.speak()
```



- ❑ 其他地方都没有动，只是子类虽然仍继承父类，但并没有全盘继承，而是**修改了父类的方法**speak()，即如果子类的对象再调用方法speak()，不是输出信息“The voice of the father's voice!”，而是输出信息**“The voice of the son's voice!”**。
- ❑ 程序的运行结果如图6-11中间所示：父类的方法speak()被子类改写的方法speak()覆盖了。

6.3.3 内置函数super()

例6-7 利用内置函数super()的例子

把例6-6稍加修改，代码如下：

```
class Parent():  
    def speak (self):  
        print('The voice of the father\'s voice!')  
  
class Child(Parent):  
    def speak(self):  
        print('The voice of the son\'s voice!')  
        super(Child,self).speak()  
  
dad=Parent()  
son=Child()  
  
dad.speak()  
son.speak()
```

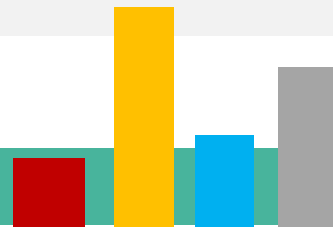


6.3.3 内置函数super()

- ❑ 其他地方都没有动，只是子类虽然仍继承了父类，并改写了父类的方法speak()，但它的改写又与例6-6不同，即**增加了一条内置函数super()调用方法speak()的语句**“**super(Child,self).speak()**”。
- ❑ 通常，**内置函数super()有两个参数，第2个是self，表示自己；第1个则是一个类名**，表示内置函数要调用的方法是这个类的父类的那个方法。例如在上述程序中，内置函数的第1个参数是Child，因此表明它要调用的那个方法speak()，是父类Child里的那个方法speak()，而不是自己覆盖的那个方法speak()。

按照这样对内置函数super()的理解，我们来看程序中最后两条语句的执行结果。dad是类Parent的实例化，son是类Child的实例化。于是，执行语句dad.speak()，就是输出信息：

The voice of the father's voice!



以下代码的输出是什么

- ☐ A Father's voice!
Son's voice!
Son's voice!
- ☒ B Father's voice!
Son's voice!
Father's voice!
- ☐ C Son's voice!
Father's voice!
Son's voice!

```
class Parent():
    def speak (self):
        print(Father's voice!)
class Child(Parent):
    def speak(self):
        print('Son's voice!')
        super(Child,self).speak()

dad=Parent()
son=Child()
dad.speak()
son.speak()
```



提交

6.3.3 内置函数super()

执行语句son.speak(), 就是要执行:

```
print('The voice of the son\'s voice!')  
super(Child,self).speak()
```

第1句好理解, 即输出信息:

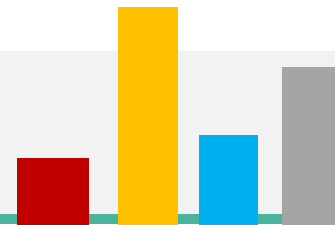
The voice of the son's voice!

第2句的意思是要去调用类Child的父类的方法speak()。于是就去执行类Parent的方法speak(), 即输出信息:

The voice of the father's voice!

于是, 该例的执行输出结果:

The voice of the father's voice!
The voice of the son's voice!
The voice of the father's voice!



3.3 内置函数super()

在Python中继承有以下几个特点。

01
OPTION

在继承中，父类的初始化程序__init__不会被自动调用，Python会先在其派生类的初始化程序中去寻找并调用，**只有当子类中没有初始化程序时，才以父类的初始化程序作为自己的初始化程序。**

02
OPTION

在子类对象里调用父类的方法时，需要以父类的类名作为前缀，并用"."分隔，还需要带上self参数变量。但在一般的类中调用方法时，并不需要带上self参数。

03
OPTION

Python总是首先查找对应类的方法，如果它不能在派生类中找到对应的方法，它才开始到父类中去逐个查找。即先在子类中查找调用的方法，找不到才去父类中查找。

通过上面对“继承”的讲述，可以总结如下，当类和类之间发生“继承”关系时，父类和子类就可能会出现下面的3种交互方式：

- 子类上的动作完全等同于父类上的动作；
- 子类上的动作完全覆盖了父类上的动作；
- 子类上的动作部分替代了父类上的动作。

目录导航



1 类和对象

- 1.1 类与对象的概念
- 1.2 Python中类的定义
- 1.3 对象：类的实例化

2 对类的进一步认识

- 2.1 关于初始化程序：__init__
- 2.2 关于参数：self
- 2.3 关于类的属性

3 类的继承

- 3.1 Python里类的继承
- 3.2 在子类中改写父类的方法
- 3.3 内置函数super()

✓ 4 Python中类的导入



4.1 类的导入

Python中类的导入

4.1 类的导入

- ❑ **模块就是一个个独立包装好的文件**，模块内可以包括**可执行的语句和定义好的函数**。当程序中打算用到整个模块中的内容，或用到它们中的一部分内容时，**应先将所需模块“import”（导入）程序中**，这样当前运行的程序才能够使用模块中的有关代码。
- ❑ 把这段话里的“函数”改成“类”，就成了类的导入的理由。

要让一个类成为可导入的，首先必须创建所谓的“导入模块”。

例如，编写如下的类和程序段：

```
#下面是类Mammal（哺乳动物）的定义
class Mammal():
    def __init__(self,name,age):
        self.name = name
        self.age = age
```



Python中类的导入

4.1 类的导入

```
def features(self): #输出哺乳动物的特征
    print('The name of the mammal is:'+self.name.title())
    print('The '+self.name.title()+ ' is '+str(self.age)+' years old')
    print('The '+self.name.title()+ ' has a head,body and limbs')
def lactation(self): #输出“哺乳期：为婴儿喂养自己的乳汁”
    print('Feed the baby\'s own milk for the young !')
def viviparous(self):      #输出“胎生：怀孕几个月后，生下自己的孩子”
    print('Give birth to a child after a few months of pregnancy!')

#下面是编写的程序体
whale=Mammal('whale',6)
whale.features()
whale.viviparous()
whale.lactation()
roo=Mammal('kangaroo',3)
roo.features()
roo.viviparous()
roo.lactation()
```



类和主体程序能否一拆为二？

4.1 类的导入

➤ 为了使其中的类成为可导入的，具体做法如下。

(1) 类Mammal的定义：

```
class Mammal():  
    def __init__(self,name,age):  
        self.name = name  
        self.age = age  
    def features(self):  
        print('The name of the mammal is:'+self.name.title())  
        print('The '+self.name.title()+ ' is '+str(self.age)+' years old')  
        print('The '+self.name.title()+ ' has a head,body and limbs')  
    def lactation(self):  
        print('Feed the baby\'s own milk for the young !')  
    def viviparous(self):  
        print('After a few months of birth to give birth to your own child!')
```

然后取名为**mammal**并把它存储在同级目录下（类文件位置可以根据需要调整）；



Python中类的导入

4.1 类的导入

(2) 将程序主体进行修改:

```
from mammal import Mammal
```

```
if __name__ == "__main__":
```

```
    whale =Mammal(' whale ',6)
```

```
    whale.features()
```

```
    whale.viviparous()
```

```
    whale.lactation()
```

```
    roo=Mammal('kangaroo',3)
```

```
    roo.features()
```

```
    roo.viviparous()
```

```
    roo.lactation()
```

#程序体开头增加导入语句

#程序体, 主程序入口 (C, C++需要main函数, Python不一定需要)



这里, 最主要的是增加了导入语句“**from mammal import Mammal**”。然后将该程序段取名为user1并存储在D盘。

(3) 在Python的“程序执行”模式下, 执行user1.py。

经过这样的3步, 就创建了一个名为mammal的导入模块。