

第十二章名称空间

一 类的实例化

二 名称空间

三 绑定方法

四 一切皆对象

本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务端进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 类的实例化

调用类 ==> 产生类的对象，该对象也可以成为类的一个实例，调用类的过程也称为类的实例化

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self)
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stul = DeepshareStudent('王小二', 18, 'male')
```

最后一行代码就是类的实例化，那么这行代码做了哪些事，我们再来回顾下

1. 先造一个空对象obj，和所有对象都一样
2. 连同这个对象和三个参数一块传递给init函数，
3. 执行DeepshareStudent.*init*(obj,'王二小',18,'male')

二 名称空间

我们可以使用类名加.*dict*方法，查看类的名称空间，那么对象的名称空间能不能查看呢？

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name # 把name这个属性放进对象的名称空间中
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self)
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小', 18, 'male')
print(stu1.__dict__)
```

在定义类的阶段产生类的名称空间，那么什么时候产生对象的名称空间呢？你要先告诉我什么时候产生对象，只有在调用类的时候才会产生对象，这个时候就会产生出对象的名称空间，有了名称空间就是把对象存好了，但是存不是目的，我们目的是取

```
print(stu1.__dict__['name'])
```

毫无疑问，这样肯定是可以的，但是我们还有更好的方法

```
print(stu1.name)
```

现在我们定义的类做一下修改

```
class DeepshareStudent:
    school = 'deepshare'
    name = 'aaaaaaaaaa'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self)
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小', 18, 'male')
```

现在有两个name，我再执行以下代码，打印的结果是什么呢

```
print(stu1.name)
```

很明显我用对象找他的独有的name属性 就应该从`init`中找name

但是，如果`init`函数中没有呢？

```
print(stu1.school)
```

这个对象stu1会先从他自己的名称空间中找school这个属性，但是他发现没有这个属性，那就后退一级，往类的名称空间中找，注意对象的名称空间和类的名称空间不是一个概念，接下来我们会说明这个问题

如果类的名称空间中，他还会往上找吗？

```
school = 'deepshare'
class DeepshareStudent:
    name = 'aaaaaaaaaa'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self)
```

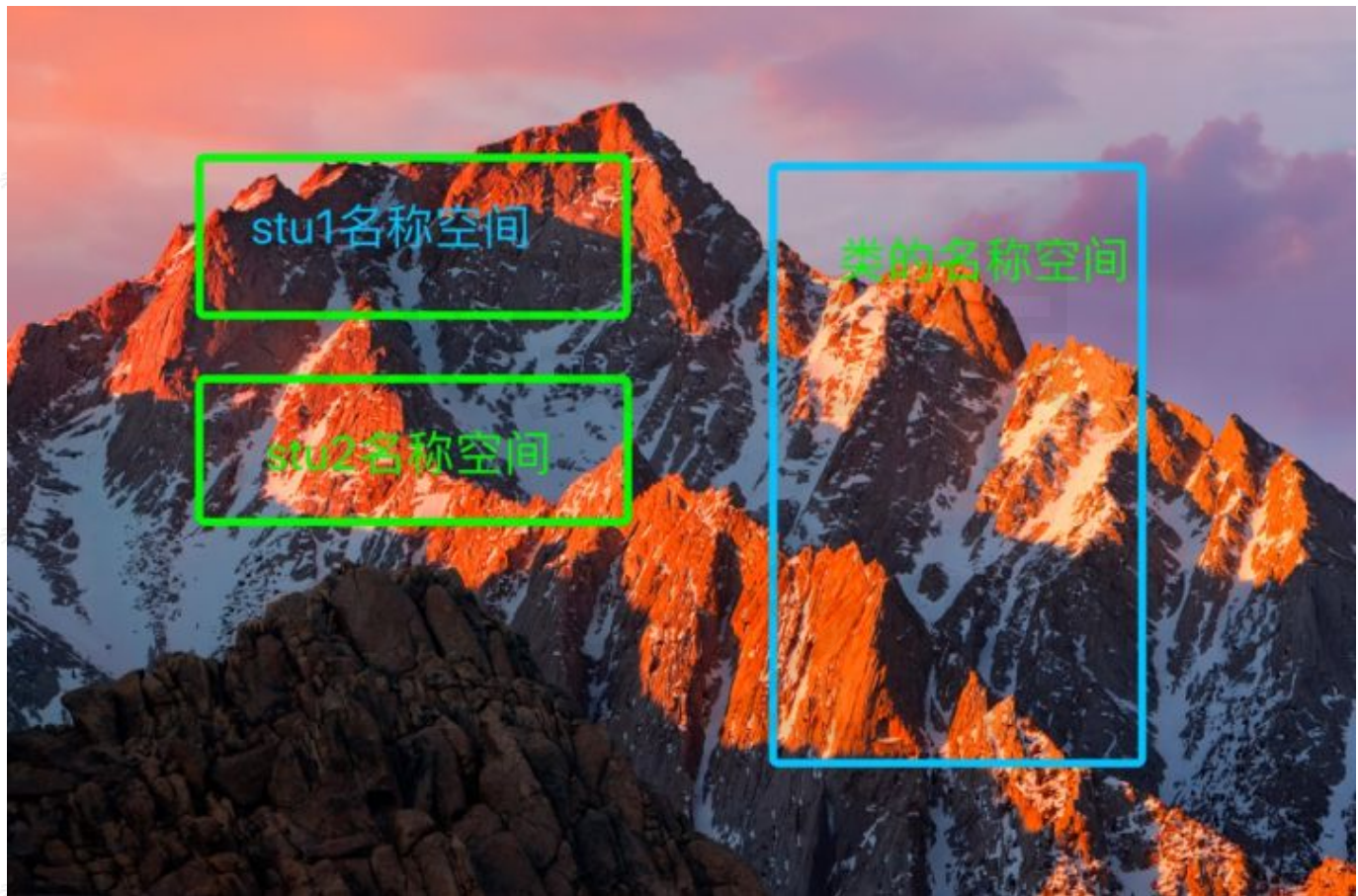
```
def eat(self):
    print('is eating')
def sleep(self):
    print('is sleeping')
stu1 = DeepshareStudent('王二小', 18, 'male')
print(stu1.school)
```

肯定是不可能的，写代码的时候Pycharm就会有错误的提示了，现在school是定义在全局，与对象stu1没有任何关系

我们再来看一下代码

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self)
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小', 18, 'male')
stu2 = DeepshareStudent('王三小', 18, 'male')
print(DeepshareStudent.__dict__)
print(id(DeepshareStudent.__dict__))
print(stu1.__dict__)
print(id(stu1.__dict__))
print(id(stu2.__dict__))
```

我现在造了两个对象stu1和stu2，打印结果可以说明，他们分别有自己的内存空间，类也有自己的内存空间，那么这之间有什么关系呢？



他们三部分是完全独立的，没有包含与被包含的关系，只不过用对象点属性做属性查找的时候先从对象的名称空间中查找，如果能够找到，就是使用对象的名称空间存的，如果找不到就取类的名称空间中找，再找不到就要报错了，错误提示：对象没有这个属性

三 绑定方法

前面我们研究是对象的特征（用变量表示的，接下来我们来研究对象的技能（用函数来表示的），不管是特征还是技能这些都是对象的属性，这是我们前面已经证明过的，那么接下来我们就来调用对象的属性

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self)
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
```

```

stu1 = DeepshareStudent('王二小', 18, 'male')
stu2 = DeepshareStudent('王三小', 18, 'male')
print(DeepshareStudent.learn)
print(stu1.learn)
print(stu2.learn)
print(id(DeepshareStudent.school))
print(id(stu1.school))
print(id(stu2.school))

```

输出

```

<function DeepshareStudent.learn at 0x1067bd488>
<bound method DeepshareStudent.learn of <__main__.DeepshareStudent object at 0x104215128>>
<bound method DeepshareStudent.learn of <__main__.DeepshareStudent object at 0x104215208>>
4550599216
4550599216
4550599216

```

从输出结果我们可以看到用类来调learn属性拿到的是一个普通方法，而用对象去调learn属性拿到的是

绑定方法，这个方法是绑定到类属性上的

我们再来理解一下，类是直接自己的名称空间中拿到learn属性，而对象在自己的名称空间中找learn属性没有找到，就要到类的名称空间中去，相当于是间接拿到了learn属性。类内部定义的函数自己能使用，但主要是给对象用的。再来看一下内存地址，上面三个函数属性的是完全不同的，而下面三个变量属性是完全一样的。这是因为类内部的变量是直接给对象使用，而类内部的函数是绑定给对象使用，这怎么理解呢

你们大家都是Deepshare的学生，都有一个相似的技能叫做学习，但是你学习能学到小明身上了，你学习并不能代表小明学习，虽然你们都具有学习的功能。这就叫绑定方法，大家用的是同一个功能，但是绑定给谁，就是谁在执行

那么这在程序中怎么体现呢？（其实我们前面的代码已经用过了，只是我还没讲）

```

class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name

```

```

        self.age = age
        self.gender = gender
    def learn(self):
        print('is learning')
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小',18,'male')
stu2 = DeepshareStudent('王三小',18,'male')
DeepshareStudent.learn('albert')
stu1.learn()

```

执行代码，我们会发现，类调用learn方法必须要传一个参数，而stu1调用learn方法不需要传参数，但是learn确实是需要一个参数的，那就说明他是自动传了一个参数，我们怎么验证呢

```

class DeepshareStudent:
    school = 'deepshare'
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('is learning',self) # 打印一下self就知道了
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小',18,'male')
stu2 = DeepshareStudent('王三小',18,'male')
DeepshareStudent.learn('albert')
stu1.learn()
print(stu1)

```

输出

```

is learning albert
is learning <__main__.DeepshareStudent object at 0x1014d1128>
<__main__.DeepshareStudent object at 0x1014d1128>

```

仔细看看是不是一样的

这就说明stu1调用learn方法本质原理就是把它自己传进来

```
stu1.learn() # DeepshareStudent.learn(stu1)
```

这就是绑定方法，类内部定义的函数，类可以使用，但是类来使用的时候就是一个普通函数，普通函数有几个参数就传几个参数

```
print(DeepshareStudent.learn)
DeepshareStudent.learn('albert')
```

但是类内部定义的函数其实是为了给对象用的，而且是绑定给对象用的，绑定给不同的对象，就是不同的绑定方法

```
print(stu1.learn)
print(stu2.learn)
```

绑定方法的特殊之处在于谁来调用，就会把谁当作第一个参数自动传入

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('is learning', self)
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小', 18, 'male')
stu2 = DeepshareStudent('王三小', 18, 'male')
stu1.learn()
stu2.learn()
```


接下来我们单独用stu1来说明

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    # self=stu1
    def learn(self):
        print('%s is learning' % self.name)  # self就是stu1, stu1有name方法
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')

stu1 = DeepshareStudent('王二小', 18, 'male')
stu2 = DeepshareStudent('王三小', 18, 'male')
stu1.learn()
stu2.learn()
```

输出

```
王二小 is learning
王三小 is learning
```

综上所述

类内部的变量是给所有对象共享，所有对象指向的都是同一个内存地址；类内部定义的函数其实是为了给对象用的，而且是绑定给对象用的，绑定给不同的对象，就是不同的绑定方法

类内部定义的函数必须要有self这个参数，但也可以有表的参数

```
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self.name)
    def choose(self, course):  # 在添加一个函数
```

```

        print('%s is choosing %s' %(self.name, course)) # 传一个course参数
def eat(self):
    print('is eating')
def sleep(self):
    print('is sleeping')
stu1 = DeepshareStudent('王二小',18,'male')
stu2 = DeepshareStudent('王三小',18,'male')
stu1.choose('Python') # 传一个实参
stu2.choose('AI')

```

输出

```

王二小 is choosing Python
王三小 is choosing AI

```

我们在使用Pycharm工具的时候，遇到括号里面需要传参，Pycharm会自动给我们提示参数的形参，当我们遇到self这个参数的，就当没有这参数，只需要传其他的参数就可以了，现在你理解了，因为self是会自动传参值的，而且是自动把它本身当作第一个参数传入。

四 一切皆对象

在Python中有一个一切皆对象的说法，面向对象讲到这里我们就可以解释了

```

class DeepshareStudent:
    school = 'deepshare'
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self.name)
    def choose(self,course):
        print('%s is choosing %s' %(self.name, course))
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小',18,'male')
stu2 = DeepshareStudent('王三小',18,'male')

```

```
print(stu1)
print(type(stu1))
print(id(stu1))
```

输出

```
<__main__.DeepshareStudent object at 0x106776390>
<class '__main__.DeepshareStudent'>
4403454864
```

一个等号表示赋值，stu1其实就是我们定义的一个变量，第一天我们就说过了，一个变量有id，type和值，他的值没什么好说的，就是一个Deepshare类的对象，id就是表示它的内存地址，这也没什么好说的，我们来看stu1的类型是什么，忽略`main`，stu1的类型就是Deepshare，stu1的类也是Deepshare，所以在 Python中

类与类型是一个概念

明确了这个概念之后，我们再来看下面代码

```
l1 = [1,2,3,4] # list([1,2,3,4])
print(type(l1))
```

输出

```
<class 'list'>
```


l1 写一个列表其本身就是# 后面的代码，我们可以看到 l1 的类型是list，那也就是说 l1的类就是list，所以list其实就是一个类，只不过它为了好看，没有首字母大写。那么我们之前所学的定义各种数据类型的变量（str，int，float，list，tuple，set等等）其实都是在调用类，来进行类的实例化，造出一个对象

```
l1 = [1,2,3,4] # list([1,2,3,4])
print(l1.append)
```

输出

```
<built-in method append of list object at 0x1077560c8>
```

在数据类型那一章我们就学过了一个列表有append方法，可以给列表添加元素，那么现在我们再来看，l1是一个对象，根据打印结果我们可以看到append其实就是一个方法，他虽然现在不叫绑定方法了，但他其实就是一个绑定方法。



```

17
18     def sleep(self):
19         print('is sleeping')
20
21     stu1 = DeepshareStudent('王二小', 18, 'male')
22     stu2 = DeepshareStudent('王三小', 18, 'male')
23
24     l1 = list([1, 2, 3, 4])
25     l1.append()

```

其实Pycharm做的挺好的，像self这种不需要传的参数他就会用灰色标记，真正要传的参数用黄色，绑定方法的特征是什么

```

l1 = [1, 2, 3, 4] # list([1, 2, 3, 4])
l1.append(5) # 其实就是在执行list.append(l1, 5)

```

既然是一样的，那我们就试一试

```

l1 = [1, 2, 3, 4]
list.append(l1, 5)
print(l1)

```

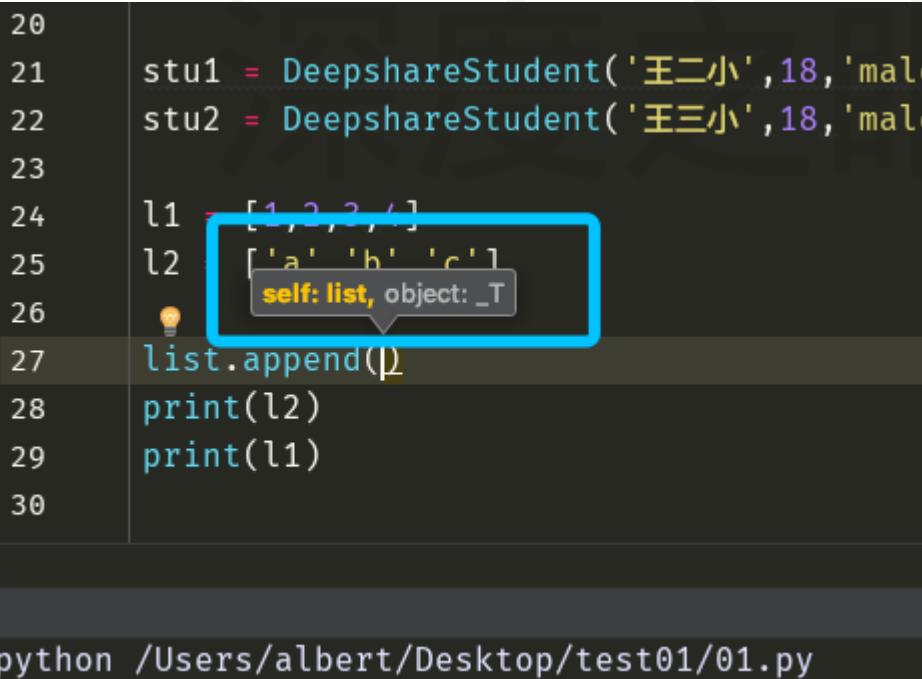
输出

```
[1, 2, 3, 4, 5]
```

绑定方法的精髓是什么，大家都是同一种功能，但问题是，我和你都具有吃饭的功能，我吃完了，吃到你肚子里去了吗？

```
l1 = [1,2,3,4]
l2 = ['a','b','c']
l2.append('d')
print(l2)
print(l1)
```

这就是绑定方法，l2 append的东西肯定不会添加到l1里面去，我们也可以换一种方式添加



```
20
21 stu1 = DeepshareStudent('王二小',18,'male')
22 stu2 = DeepshareStudent('王三小',18,'male')
23
24 l1 = [1,2,3,4]
25 l2 = ['a','b','c']
26
27 list.append(l2)
28 print(l2)
29 print(l1)
30
```

A tooltip is shown over the `list` argument in line 27, displaying `self: list, object: _T`. The command prompt at the bottom shows the file path: `bin/python /Users/albert/Desktop/test01/01.py`.

这个时候必须要传self参数

```
l1 = [1,2,3,4]
l2 = ['a','b','c']
list.append(l2,'d')
print(l2)
print(l1)
```

我们最后再来看一下我们自己写的类和Python内置的类

```

class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def learn(self):
        print('%s is learning' % self.name)
    def choose(self, course):
        print('%s is choosing %s' % (self.name, course))
    def eat(self):
        print('is eating')
    def sleep(self):
        print('is sleeping')
stu1 = DeepshareStudent('王二小', 18, 'male')
stu2 = DeepshareStudent('王三小', 18, 'male')
print(int)
print(str)
print(list)
print(tuple)
print(set)
print(dict)
print(DeepshareStudent)

```

输出

```

<class 'int'>
<class 'str'>
<class 'list'>
<class 'tuple'>
<class 'set'>
<class 'dict'>
<class '__main__.DeepshareStudent'>

```

都叫class，除去`main`，他们看起来是一样的，当然他们用法的原理也是一样的。

五 对象属性的增删改查

先在这里补充一点，我们初始化用的`init`函数，不是`int`函数

```
42
43 class Bar:
44     def __int__(self):
45         pass
46     def __init__(self):
47         pass
48
```

查看

```
class Bar:
    n = 1111
    def __init__(self, x):
        self.x = x
obj = Bar('abc')
print(obj.__dict__)
print(obj.x)
print(obj.n)
```

添加

```
class Bar:
    n = 1111
    def __init__(self, x):
        self.x = x
obj = Bar('abc')
print(obj.__dict__)
print(obj.x)
print(obj.n)
obj.abc = 'abc'
print(obj.abc)
```

修改

```
class Bar:
    n = 1111
    def __init__(self, x):
        self.x = x
obj = Bar('abc')
```

```
print(obj.__dict__)
print(obj.x)
print(obj.n)
obj.abc = 'abc'
print(obj.abc)
obj.abc = '123'
print(obj.abc)
```

删除

```
class Bar:
    n = 1111
    def __init__(self, x):
        self.x = x

obj = Bar('abc')
print(obj.__dict__)
print(obj.x)
print(obj.n)
obj.abc = 'abc'
print(obj.abc)
obj.abc = '123'
print(obj.abc)
del obj.abc
# print(obj.abc)
```

练习一：

实现这样一个功能，有一个对象，他有一个count属性，count属性是统计他所在的类产生了多少个对象，即print(obj.count)能打印出对象的个数

分析：

既然要统计有多少个对象，那么只要有一个对象就要执行一次类的调用，调用一次类，其实就要执行类内部的init函数，所以我们在init函数里面写上相应的逻辑就好了

```
class Foo:
    count = 0
    def __init__(self):
        self.count += 1

obj1 = Foo()
obj2 = Foo()
obj3 = Foo()
print(obj1.count)
```


是不是这样写呢？如果这样写就错了，我们可以执行程序，打印一下看看效果

```
1
```

这是什么原因呢？

`self`是对象自己，`obj1`，`obj2`和`obj3`是互不相通的，虽然每次`count`都会加一，但是其实每个对象都有自己的独一份的`count`，初始值永远是0，每次来都是加1，所以永远是1

我们应该对他们共享的那个属性来做修改，什么属性是共享的呢？

```
class Foo:
    count = 0
    def __init__(self):
        Foo.count += 1

obj1 = Foo()
obj2 = Foo()
obj3 = Foo()
print(obj1.count)
```

虽然这次我们没有在对象自己的名称空间中添加`count`属性，但是类里面有啊，对象在做属性查找的时候先从自己的名称空间中找，没有就去他所在的类里面找，在类里面，这个`count`变量属性是所有对象共用的。

练习二：

实现一个人狗大战的程序，人可以咬狗，狗也可以咬人，人和狗都有自己的生命值，被咬了之后会掉血，当生命值为0时，人或者狗就死了

按照步骤我们先总结出现实中的对象

```
现实中的人类
"""
    人1
    特征：
        名字='张二炮'
        攻击力=60
        生命值=100
    技能：
        咬
```

```

人2
    特征：
        名字='刘老三'
        攻击力=50
        生命值=100
    技能：
        咬

现实中的人类
    相同的特征 无
    相同的技能 咬
"""

现实中的狗类
"""
    狗1
        特征：
            名字='旺财'
            品种="京巴"
            攻击力=80
            生命值=50
        技能：
            咬

    狗2
        特征：
            名字='小黑'
            品种="藏獒"
            攻击力=200
            生命值=200
        技能：
            咬

现实中的狗类
    相同的特征 无
    相同的技能 咬
"""
    
```

注意：

两个人虽然有相同的特征的，都有名字，攻击力，和生命值，但是这是他们每个人独立的，不如章二炮被咬了之后，他的生命值下降，而刘老三没有被咬，还是原来的值，所以我们定义现实中的人类是没有相同的特征的，同理，现实中的狗类也是一样的。

代码实现

```

class People:
    def __init__(self, name, aggressivity, life_value=100):
        self.name = name
    
```

```

        self.aggresivity = aggressivity
        self.life_value = life_value

# 注意：咬肯定不能咬自己，一定是咬敌人，所以要传一个敌人参数
def bite(self, enemy): #self=p1    enemy=d1
    enemy.life_value-=self.aggresivity
    print("""
    人[%s] 咬了一口狗 [%s]
    狗掉血[%s]
    狗还剩血量[%s]
    """)
    print("%(self.name,enemy.name,self.aggresivity,enemy.life_value)"
          )

class Dog:
    def __init__(self, name, dog_type, aggressivity, life_value):
        self.name = name
        self.dog_type = dog_type
        self.aggresivity = aggressivity
        self.life_value = life_value
    def bite(self, enemy): #self = d1    enemy= p1
        enemy.life_value-=self.aggresivity
        print("""
        狗[%s] 咬了一口人 [%s]
        人掉血[%s]
        人还剩血量[%s]
        """)
        print("%(self.name,enemy.name,self.aggresivity,enemy.life_value)"
              )

p1 = People('张二炮', 60)
d1=Dog('小黑',"藏獒",200,200)
p1.bite(d1)
d1.bite(p1)

```

这段代码把程序的基本逻辑和功能实现了，但是还有一些明显的bug，大家如果有兴趣，可以自己完善。