刚开始学习Python的类写法的时候觉得很是麻烦，为什么定义时需要而调用时又不需要，为什么不能内部简化从而减少我们敲击键盘的次数？你看完这篇文章后就会明白所有的疑问。

### self代表类的实例，而非类。

实例来说明

class Test:

def prt(self):

print(self)

print(self.\_\_class\_\_)

t = Test()

t.prt()

执行结果如下

<\_\_main\_\_.Test object at 0x000000000284E080>

<class '\_\_main\_\_.Test'>

从上面的例子中可以很明显的看出，self代表的是类的实例。而self.class则指向类。

### self不必非写成self

有很多童鞋是先学习别的语言然后学习Python的，所以总觉得self怪怪的，想写成this，可以吗？

当然可以，还是把上面的代码改写一下。

class Test:

def prt(this):

print(this)

print(this.\_\_class\_\_)

t = Test()

t.prt()

改成this后，运行结果完全一样。

当然，最好还是尊重约定俗成的习惯，使用self。

### self可以不写吗

在Python的解释器内部，当我们调用t.prt()时，实际上Python解释成Test.prt(t)，也就是说把self替换成类的实例。

有兴趣的童鞋可以把上面的t.prt()一行改写一下，运行后的实际结果完全相同。

实际上已经部分说明了self在定义时不可以省略，如果非要试一下，那么请看下面：

class Test:

def prt():

print(self)

t = Test()

t.prt()

运行时提醒错误如下：prt在定义时没有参数，但是我们运行时强行传了一个参数。

由于上面解释过了t.prt()等同于Test.prt(t)，所以程序提醒我们多传了一个参数t。

Traceback (most recent call last):

File "h.py", line 6, in <module>

 t.prt()

TypeError: prt() takes 0 positional arguments but 1 was given

当然，如果我们的定义和调用时均不传类实例是可以的，这就是类方法。

class Test:

def prt():

print(\_\_class\_\_)

Test.prt()

运行结果如下

<class '\_\_main\_\_.Test'>

****在继承时，传入的是哪个实例，就是那个传入的实例，而不是指定义了self的类的实例。****

先看代码

class Parent:

def pprt(self):

print(self)

class Child(Parent):

def cprt(self):

print(self)

c = Child()

c.cprt()

c.pprt()

p = Parent()

p.pprt()

运行结果如下

<\_\_main\_\_.Child object at 0x0000000002A47080>

<\_\_main\_\_.Child object at 0x0000000002A47080>

<\_\_main\_\_.Parent object at 0x0000000002A47240>

解释：

运行c.cprt()时应该没有理解问题，指的是Child类的实例。

但是在运行c.pprt()时，等同于Child.pprt(c)，所以self指的依然是Child类的实例，由于self中没有定义pprt()方法，所以沿着继承树往上找，发现在父类Parent中定义了pprt()方法，所以就会成功调用。

### 在描述符类中，self指的是描述符类的实例

不太容易理解，先看实例：

class Desc:

def \_\_get\_\_(self, ins, cls):

print('self in Desc: %s ' % self )

print(self, ins, cls)

class Test:

x = Desc()

def prt(self):

print('self in Test: %s' % self)

t = Test()

t.prt()

t.x

运行结果如下：

self in Test: <\_\_main\_\_.Test object at 0x0000000002A570B8>

self in Desc: <\_\_main\_\_.Desc object at 0x000000000283E208>

<\_\_main\_\_.Desc object at 0x000000000283E208> <\_\_main\_\_.Test object at 0x0000000002A570B8> <class '\_\_main\_\_.Test'>

大部分童鞋开始有疑问了，为什么在Desc类中定义的self不是应该是调用它的实例t吗？怎么变成了Desc类的实例了呢？

注意：此处需要睁大眼睛看清楚了，这里调用的是t.x，也就是说是Test类的实例t的属性x，由于实例t中并没有定义属性x，所以找到了类属性x，而该属性是描述符属性，为Desc类的实例而已，所以此处并没有顶用Test的任何方法。

那么我们如果直接通过类来调用属性x也可以得到相同的结果。

下面是把t.x改为Test.x运行的结果。

self in Test: <\_\_main\_\_.Test object at 0x00000000022570B8>

self in Desc: <\_\_main\_\_.Desc object at 0x000000000223E208>

<\_\_main\_\_.Desc object at 0x000000000223E208> None <class '\_\_main\_\_.Test'>

题外话：由于在很多时候描述符类中仍然需要知道调用该描述符的实例是谁，所以在描述符类中存在第二个参数ins，用来表示调用它的类实例，所以t.x时可以看到第三行中的运行结果中第二项为<main.Test object at 0x0000000002A570B8>。而采用Test.x进行调用时，由于没有实例，所以返回None。

### ****总结****

* self在定义时需要定义，但是在调用时会自动传入。
* self的名字并不是规定死的，但是最好还是按照约定是用self
* self总是指调用时的类的实例

**另一个的解释**

cls主要用在类方法定义，而self则是实例方法。

self, cls 不是关键字，完全可以使用自己写的任意变量代替实现一样的效果。(但我们还是要遵守约定俗成的方法)

 普通的实例方法，第一个参数需要是self，它表示一个具体的实例本身。

如果用了staticmethod，那么就可以无视这个self，而将这个方法当成一个普通的函数使用。  
而对于classmethod，它的第一个参数不是self，是cls，它表示这个类本身。  
>>> class A(object):

    def foo1(self):  
        print "Hello",self

    @staticmethod  
    def foo2():  
        print "hello"

    @classmethod  
    def foo3(cls):  
        print "hello",cls  
 >>> a = A()  
>>> a.foo1()          #最常见的调用方式，但与下面的方式相同  
Hello <\_\_main\_\_.A object at 0x9f6abec>  
>>> A.foo1(a)         #这里传入实例a，相当于普通方法的self  
Hello <\_\_main\_\_.A object at 0x9f6abec>  
>>> A.foo2()          #这里，由于静态方法没有参数，故可以不传东西  
hello  
>>> A.foo3()          #这里，由于是类方法，因此，它的第一个参数为类本身。  
hello <class '\_\_main\_\_.A'>  
>>> A                 #可以看到，直接输入A，与上面那种调用返回同样的信息。  
<class '\_\_main\_\_.A'>

---------------------------------------------------------------------------------

类的定义可以动态修改

class MyTest:

    myname = 'peter'  
    def sayhello(self):  
        print "say hello to %s" % self.myname  
if \_\_name\_\_ == "\_\_main\_\_":  
    MyTest.myname = 'hone'  
    MyTest.sayhello = lambda self, name: "I want say hello to %s" % name  
    MyTest.saygoodbye = lambda self,name: "I do not want say goodbye to %s" % name  
    print MyTest().sayhello(MyTest.myname)  
    print MyTest().saygoodbye(MyTest.myname)

这里修改了MyTest类中的变量和函数定义， 实例化的instance有了不同的行为特征。

-------------------------------------------------------------------------------------

1.Python是一门[动态语言](https://www.baidu.com/s?wd=%E5%8A%A8%E6%80%81%E8%AF%AD%E8%A8%80&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3rHRsuWf3mWIhmvD3PvmY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHTsPHcLn16dPWcLnWmkrHcvn0" \t "https://www.cnblogs.com/kex1n/p/_blank)，任何实体都可以动态地添加或删除属性。  
2.一个类定义了一个作用域。  
3.类实例也引入了一个作用域，这与相应类定义的作用域不同。  
4.在类实例中查找属性的时候，首先在实例自己的作用域中查找，如果没有找到，则再在类定义的作用域中查找。  
5.在对类实例属性进行赋值的时候，实际上会在类实例定义的作用域中添加一个属性（如果还不存在的话），并不会影响到相应类中定义的同名属性。  
class A:  
    cls\_i = 0  
    cls\_j = {}  
      
    def \_\_init\_\_(self):  
        self.instance\_i = 0  
        self.instance\_j = {}  
  
if \_\_name\_\_ == '\_\_main\_\_':  
    a = A()  
    print A.\_\_dict\_\_ #{'\_\_init\_\_': , '\_\_module\_\_': '\_\_main\_\_', 'cls\_i': 0, 'cls\_j': {}, '\_\_doc\_\_': None}  
    print a.\_\_dict\_\_ # {'instance\_j': {}, 'instance\_i': 0}

>>> a.cls\_i  
0  
>>>   
a.instance\_i  
0  
  
在查找cls\_i的时候，实例a的作用域中是没有它的，却在A的作用域中找到了它；在查找instance\_i的时候，直接可在a的作用域中找到它。  
  
  
如果我们企图通过实例a来修改cls\_i的值，那会怎样呢：  
>>> a.cls\_i = 1  
>>> a.\_\_dict\_\_  
{'instance\_j': {}, 'cls\_i': 1, 'instance\_i': 0}  
>>> A.\_\_dict\_\_  
{'\_\_init\_\_': , '\_\_module\_\_': '\_\_main\_\_', 'cls\_i': 0, 'cls\_j': {},   
'\_\_doc\_\_': None}  
我们可以看到，a的作用域中多了一个cls\_i属性，其值为1；同时，我们也注意到A作用域中的cls\_i属性的值仍然为0；在这里，我们其实是增加了一个实例属性，并没有修改到[类属性](https://www.baidu.com/s?wd=%E7%B1%BB%E5%B1%9E%E6%80%A7&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3rHRsuWf3mWIhmvD3PvmY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHTsPHcLn16dPWcLnWmkrHcvn0" \t "https://www.cnblogs.com/kex1n/p/_blank)。  
如果我们通过实例a操纵cls\_j中的数据（注意不是cls\_j本身），又会怎么样呢：  
>>> a.cls\_j['a'] = 'a'  
>>> a.\_\_dict\_\_  
{'instance\_j': {}, 'cls\_i': 1, 'instance\_i': 0}  
>>> A.\_\_dict\_\_  
{'\_\_init\_\_': , '\_\_module\_\_': '\_\_main\_\_', 'cls\_i': 0, 'cls\_j': {'a': 'a'}, '\_\_doc\_\_': None}  
  
我们可以看到a的作用域没有发生什么变化，但是A的作用域发生了一些变化，cls\_j中的数据发生了变化。  
  
实例的作用域发生变化，并不会影响到该类的其它实例，但是类的作用域发生变化，则会影响到该类的所有实例，包括在这之前创建的实例

python类里会出现这三个单词，self和cls都可以用别的单词代替，类的方法有三种，

一是通过def定义的 普通的一般的，需要至少传递一个参数，一般用self，这样的方法必须通过一个类的实例去访问，类似于c++中通过对象去访问；

二是在def前面加上@classmethod，这种类方法的一个特点就是可以通过类名去调用，但是也必须传递一个参数，一般用cls表示class，表示可以通过类直接调用；

三是在def前面加上@staticmethod，这种类方法是静态的类方法，类似于c++的静态函数，他的一个特点是参数可以为空，同样支持类名和对象两种调用方式；

代码：

[python] [view plain](http://blog.csdn.net/xluren/article/details/9168535" \o "view plain) [copy](http://blog.csdn.net/xluren/article/details/9168535" \o "copy) [print](http://blog.csdn.net/xluren/article/details/9168535" \o "print)[?](http://blog.csdn.net/xluren/article/details/9168535" \o "?)

class A:

    member = "this is a test."

    def \_\_init\_\_(self):

        pass

    @classmethod

    def Print1(cls):

        print "print 1: ", cls.member

    def Print2(self):

        print "print 2: ", self.member

    @classmethod

    def Print3(paraTest):

        print "print 3: ", paraTest.member

    @staticmethod

    def print4():

        print "hello"

a = A()

A.Print1()

a.Print1()

#A.Print2()

a.Print2()

A.Print3()

a.Print3()

A.print4()

# Python 静态方法和类方法的区别

# python staticmethod and classmethod

尽管 classmethod 和 staticmethod 非常相似，但在用法上依然有一些明显的区别。classmethod 必须有一个指向 ****类对象**** 的引用作为第一个参数，而 staticmethod 可以没有任何参数。

让我们看几个例子。

## 例子 – Boilerplate

class Date(object):

def \_\_init\_\_(self, day=0, month=0, year=0):

self.day = day

self.month = month

self.year = year

很明显，这个类的对象可以存储日期信息（不包括时区，假设他们都存储在 UTC）。

这里的 ****init**** 方法用于初始化对象的属性，它的第一个参数一定是 self，用于指向已经创建好的对象。

### Class Method

利用 classmethod 可以做一些很棒的东西。

比如我们可以支持从特定格式的日期字符串来创建对象，它的格式是 (‘dd-mm-yyyy’)。很明显，我们只能在其他地方而不是 ****init**** 方法里实现这个功能

大概步骤：

* 解析字符串，得到整数 day， month， year。
* 使用得到的信息初始化对象

代码如下

day, month, year = map(int, string\_date.split('-'))

date1 = Date(day, month, year)

理想的情况是 Date 类本身可以具备处理字符串时间的能力，解决了重用性问题，比如添加一个额外的方法。

C++ 可以方便的使用重载来解决这个问题，但是 python 不具备类似的特性。 所以接下来我们要使用 classmethod 来帮我们实现。

@classmethod

def from\_string(cls, date\_as\_string):

day, month, year = map(int, date\_as\_string.split('-'))

date1 = cls(day, month, year)

return date1

date2 = Date.from\_string('11-09-2012')

让我们在仔细的分析下上面的实现，看看它的好处。

我们在一个方法中实现了功能，因此它是可重用的。 这里的封装处理的不错（如果你发现还可以在代码的任意地方添加一个不属于 Date 的函数来实现类似的功能，那很显然上面的办法更符合 OOP 规范）。 ****cls**** 是一个保存了 ****class**** 的对象（所有的一切都是对象）。 更妙的是， Date 类的衍生类都会具有 from\_string 这个有用的方法。

### Static method

staticmethod 没有任何必选参数，而 classmethod 第一个参数永远是 cls， instancemethod 第一个参数永远是 self。

@staticmethod

def is\_date\_valid(date\_as\_string):

day, month, year = map(int, date\_as\_string.split('-'))

return day <= 31 and month <= 12 and year <= 3999

# usage:

is\_date = Date.is\_date\_valid('11-09-2012')

所以，从静态方法的使用中可以看出，我们不会访问到 class 本身 – 它基本上只是一个函数，在语法上就像一个方法一样，但是没有访问对象和它的内部（字段和其他方法），相反 classmethod 会访问 cls， instancemethod 会访问 self。

# Python 函数中，参数是传值，还是传引用？

在 C/C++ 中，传值和传引用是函数参数传递的两种方式，在Python中参数是如何传递的？回答这个问题前，不如先来看两段代码。

代码段1：

def foo(arg):

arg = 2

print(arg)

a = 1

foo(a) # 输出：2

print(a) # 输出：1

看了代码段1的同学可能会说参数是值传递。

代码段2：

def bar(args):

args.append(1)

b = []

print(b)#　输出：[]

print(id(b)) # 输出：4324106952

bar(b)

print(b) ＃　输出：[1]

print(id(b)) # 输出：4324106952

看了代码段2，这时可能又有人会说，参数是传引用，那么问题来了，参数传递到底是传值还是传引用或者两者都不是？为了把这个问题弄清楚，先了解 Python 中变量与对象之间的关系。

### 变量与对象

Python 中一切皆为对象，数字是对象，列表是对象，函数也是对象，任何东西都是对象。而变量是对象的一个引用（又称为名字或者标签），对象的操作都是通过引用来完成的。例如，[]是一个空列表对象，变量 a 是该对象的一个引用

Python



|  |  |
| --- | --- |
| 1  2 | a = []  a.append(1) |

在 Python 中，「变量」更准确叫法是「名字」，赋值操作 = 就是把一个名字****绑定****到一个对象上。就像给对象添加一个标签。

a = 1



整数 1 赋值给变量 a 就相当于是在整数1上绑定了一个 a 标签。

a = 2



整数 2 赋值给变量 a，相当于把原来整数 1 身上的 a 标签撕掉，贴到整数 2 身上。

b = a



把变量 a 赋值给另外一个变量 b，相当于在对象 2 上贴了 a，b 两个标签，通过这两个变量都可以对对象 2 进行操作。

变量本身没有类型信息，类型信息存储在对象中，这和C/C++中的变量有非常大的出入（C中的变量是一段内存区域）

### 函数参数

Python 函数中，****参数的传递本质上是一种赋值操作****，而赋值操作是一种名字到对象的绑定过程，清楚了赋值和参数传递的本质之后，现在再来分析前面两段代码。

def foo(arg):

arg = 2

print(arg)

a = 1

foo(a) # 输出：2

print(a) # 输出：1

IMG_259

在代码段1中，变量 a 绑定了 1，调用函数 foo(a) 时，相当于给参数 arg 赋值 arg=1，这时两个变量都绑定了 1。在函数里面 arg 重新赋值为 2 之后，相当于把 1 上的 arg 标签撕掉，贴到 2 身上，而 1 上的另外一个标签 a 一直存在。因此 print(a) 还是 1。

再来看一下代码段2

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def bar(args):      args.append(1)    b = []  print(b)*#　输出：[]*  print(id(b)) *# 输出：4324106952*  bar(b)  print(b) ＃　输出：[1]  print(id(b))  *# 输出：4324106952* |

IMG_260

执行 append 方法前 b 和 arg 都指向（绑定）同一个对象，执行 append 方法时，并没有重新赋值操作，也就没有新的绑定过程，append 方法只是对列表对象插入一个元素，对象还是那个对象，只是对象里面的内容变了。因为 b 和 arg 都是绑定在同一个对象上，执行 b.append 或者 arg.append 方法本质上都是对同一个对象进行操作，因此 b 的内容在调用函数后发生了变化（但id没有变，还是原来那个对象）

最后，回到问题本身，究竟是是传值还是传引用呢？说传值或者传引用都不准确。非要安一个确切的叫法的话，****叫传对象（call by object）****。如果作为面试官，非要考察候选人对 Python 函数参数传递掌握与否，与其讨论字面上的意思，还不如来点实际代码。

### show me the code

Python



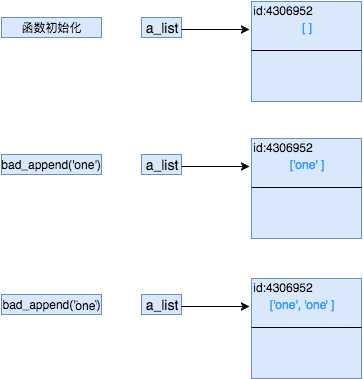
|  |  |
| --- | --- |
| 1  2  3 | def bad\_append(new\_item, a\_list=[]):      a\_list.append(new\_item)      return a\_list |

这段代码是初学者最容易犯的错误，用可变(mutable)对象作为参数的默认值。函数定义好之后，默认参数 a\_list 就会指向（绑定）到一个空列表对象，每次调用函数时，都是对同一个对象进行 append 操作。因此这样写就会有潜在的bug，同样的调用方式返回了不一样的结果。

Python



|  |  |
| --- | --- |
| 1  2  3  4 | >>> print bad\_append('one')  ['one']  >>> print bad\_append('one')  ['one', 'one'] |



而正确的方式是，把参数默认值指定为None

Python



|  |  |
| --- | --- |
| 1  2  3  4  5 | def good\_append(new\_item, a\_list=None):      if a\_list is None:          a\_list = []      a\_list.append(new\_item)      return a\_list |

IMG_262

参考：http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#other-languages-have-variables

# Python 面向对象（初级篇）

# 概述

* 面向过程：根据业务逻辑从上到下写垒代码
* 函数式：将某功能代码封装到函数中，日后便无需重复编写，仅调用函数即可
* 面向对象：对函数进行分类和封装，让开发“更快更好更强…”

面向过程编程最易被初学者接受，其往往用一长段代码来实现指定功能，开发过程中最常见的操作就是粘贴复制，即：将之前实现的代码块复制到现需功能处。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | while True：      if cpu利用率 > 90%:          #发送邮件提醒          连接邮箱服务器          发送邮件          关闭连接      if 硬盘使用空间 > 90%:          #发送邮件提醒          连接邮箱服务器          发送邮件          关闭连接      if 内存占用 > 80%:          #发送邮件提醒          连接邮箱服务器          发送邮件          关闭连接 |

随着时间的推移，开始使用了函数式编程，增强代码的重用性和可读性，就变成了这样：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def 发送邮件(内容)      #发送邮件提醒      连接邮箱服务器      发送邮件      关闭连接  while True：      if cpu利用率 > 90%:          发送邮件('CPU报警')      if 硬盘使用空间 > 90%:          发送邮件('硬盘报警')      if 内存占用 > 80%:          发送邮件('内存报警') |

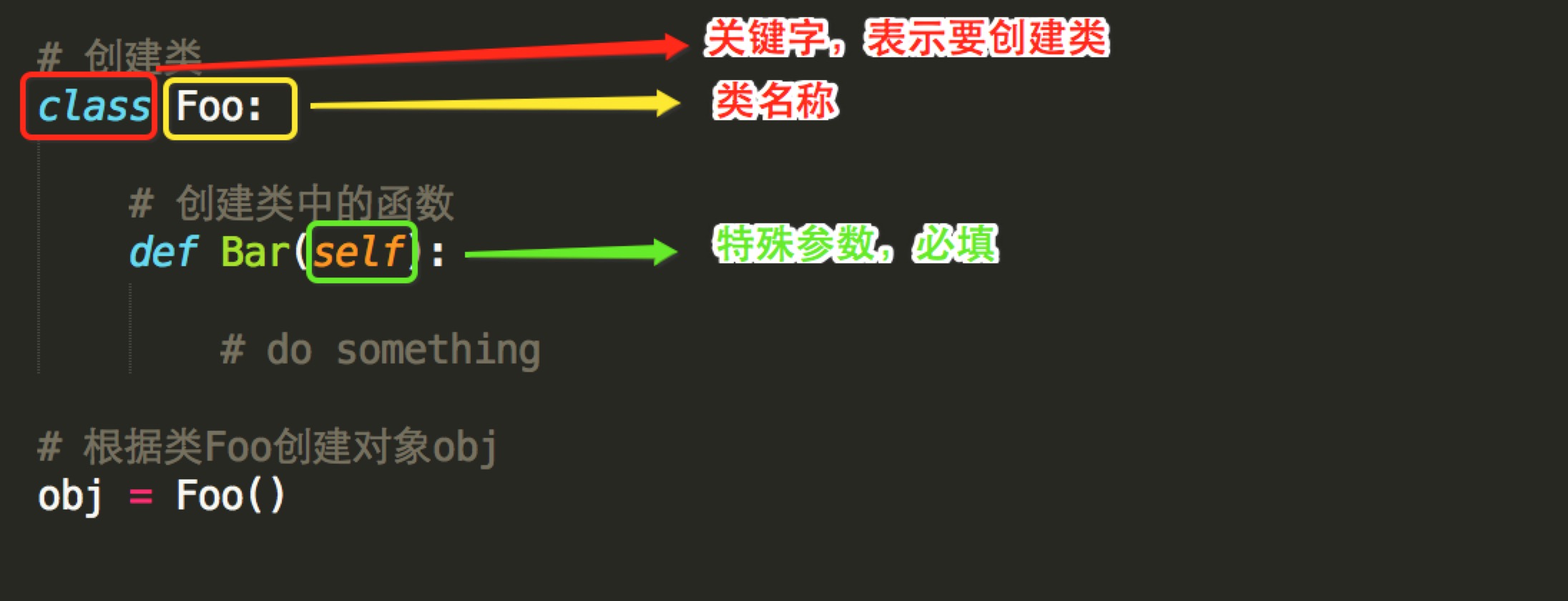
今天我们来学习一种新的编程方式：面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）。

注：Java和C#来说只支持面向对象编程，而python比较灵活即支持面向对象编程也支持函数式编程

# 创建类和对象

面向对象编程是一种编程方式，此编程方式的落地需要使用 “类” 和 “对象” 来实现，所以，面向对象编程其实就是对 “类” 和 “对象” 的使用。

* 类就是一个模板，模板里可以包含多个函数，函数里实现一些功能
* 对象则是根据模板创建的实例，通过实例对象可以执行类中的函数

[](http://jbcdn2.b0.upaiyun.com/2017/08/19281dc007521fcd520670e144c3aa68.jpg)

* class是关键字，表示类
* 创建对象，类名称后加括号即可

ps：类中的函数第一个参数必须是self（详细见：类的三大特性之封装）  
　　 类中定义的函数叫做 “方法”



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # 创建类  class Foo:        def Bar(self):          print 'Bar'      def Hello(self, name):          print 'i am %s' %name  # 根据类Foo创建对象obj  obj = Foo()  obj.Bar()            #执行Bar方法  obj.Hello('wupeiqi') #执行Hello方法 |

****诶，****你在这里是不是有疑问了？使用函数式编程和面向对象编程方式来执行一个“方法”时函数要比面向对象简便。

* 面向对象：【创建对象】【通过对象执行方法】
* 函数编程：【执行函数】

观察上述对比答案则是肯定的，然后并非绝对，场景的不同适合其的编程方式也不同。

总结：函数式的应用场景 –> 各个函数之间是独立且无共用的数据。

# 面向对象三大特性

面向对象的三大特性是指：封装、继承和多态。

****一、封装****

封装，顾名思义就是将内容封装到某个地方，以后再去调用被封装在某处的内容。

所以，在使用面向对象的封装特性时，需要：

* 将内容封装到某处
* 从某处调用被封装的内容

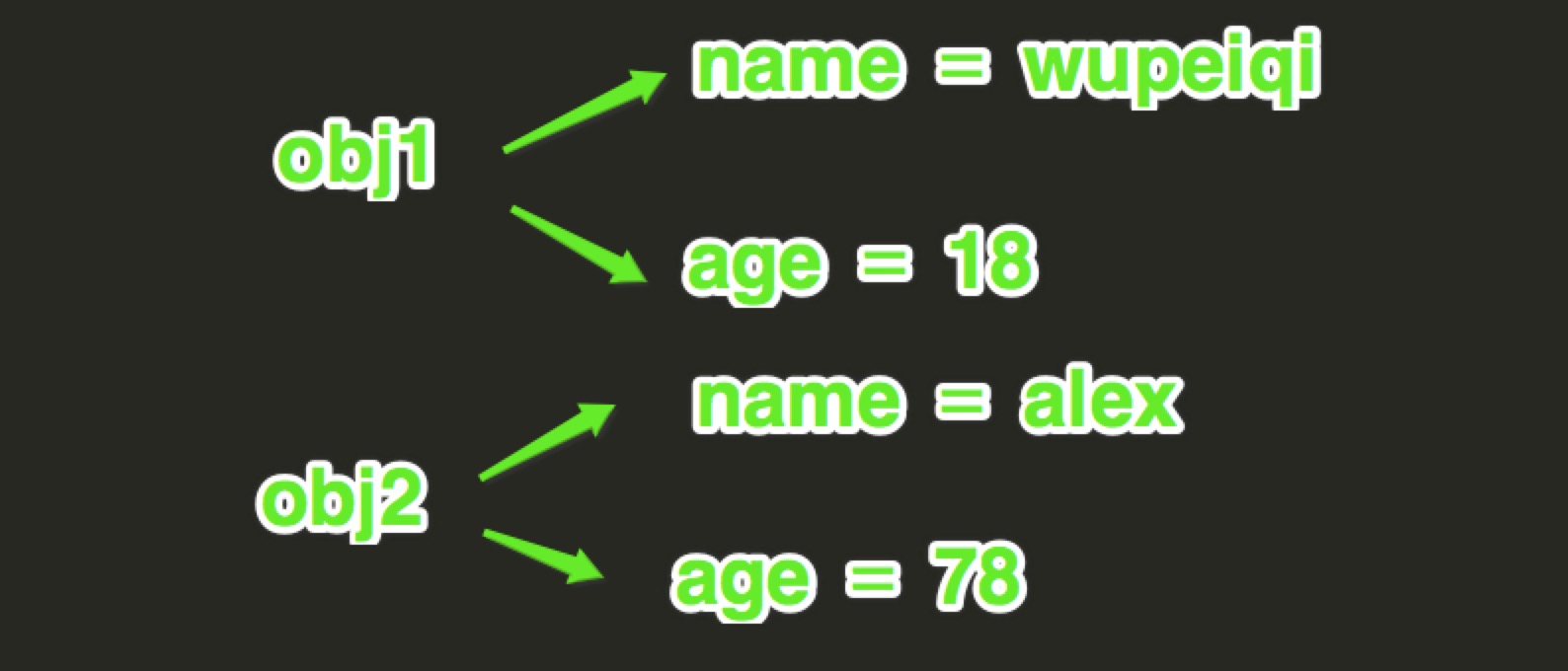
****第一步：将内容封装到某处****

[](http://jbcdn2.b0.upaiyun.com/2017/08/8c694e9ed3c95bea6bd7d8f66ef16a29.jpg)

self 是一个形式参数：

* 当执行 obj1 = Foo(‘wupeiqi’, 18 ) 时，self 等于 obj1
* 当执行 obj2 = Foo(‘alex’, 78 ) 时，self 等于 obj2

所以，内容其实被封装到了对象 obj1 和 obj2 中，每个对象中都有 name 和 age 属性，在内存里类似于下图来保存。

[](http://jbcdn2.b0.upaiyun.com/2017/08/bac98e25a8980b969654d9add13cc5ed.jpg)

****第二步：从某处调用被封装的内容****

调用被封装的内容时，有两种情况：

* 通过对象直接调用
* 通过self间接调用

1、通过对象直接调用被封装的内容

上图展示了对象 obj1 和 obj2 在内存中保存的方式，根据保存格式可以如此调用被封装的内容：对象.属性名



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Foo:      def \_\_init\_\_(self, name, age):          self.name = name          self.age = age  obj1 = Foo('wupeiqi', 18)  print obj1.name    # 直接调用obj1对象的name属性  print obj1.age     # 直接调用obj1对象的age属性  obj2 = Foo('alex', 73)  print obj2.name    # 直接调用obj2对象的name属性  print obj2.age     # 直接调用obj2对象的age属性 |

2、通过self间接调用被封装的内容

执行类中的方法时，需要通过self间接调用被封装的内容



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Foo:        def \_\_init\_\_(self, name, age):          self.name = name          self.age = age        def detail(self):          print self.name          print self.age    obj1 = Foo('wupeiqi', 18)  obj1.detail()  # Python默认会将obj1传给self参数，即：obj1.detail(obj1)，所以，此时方法内部的 self ＝ obj1，即：self.name 是 wupeiqi ；self.age 是 18    obj2 = Foo('alex', 73)  obj2.detail()  # Python默认会将obj2传给self参数，即：obj1.detail(obj2)，所以，此时方法内部的 self ＝ obj2，即：self.name 是 alex ； self.age 是 78 |

****综上所述，对于面向对象的封装来说，其实就是使用构造方法将内容封装到 对象 中，然后通过对象直接或者self间接获取被封装的内容。****

****练习一****：在终端输出如下信息

* 小明，10岁，男，上山去砍柴
* 小明，10岁，男，开车去东北
* 小明，10岁，男，最爱大保健
* 老李，90岁，男，上山去砍柴
* 老李，90岁，男，开车去东北
* 老李，90岁，男，最爱大保健
* 老张…



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | def kanchai(name, age, gender):      print "%s,%s岁,%s,上山去砍柴" %(name, age, gender)      def qudongbei(name, age, gender):      print "%s,%s岁,%s,开车去东北" %(name, age, gender)      def dabaojian(name, age, gender):      print "%s,%s岁,%s,最爱大保健" %(name, age, gender)      kanchai('小明', 10, '男')  qudongbei('小明', 10, '男')  dabaojian('小明', 10, '男')      kanchai('老李', 90, '男')  qudongbei('老李', 90, '男')  dabaojian('老李', 90, '男') |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Foo:        def \_\_init\_\_(self, name, age ,gender):          self.name = name          self.age = age          self.gender = gender        def kanchai(self):          print "%s,%s岁,%s,上山去砍柴" %(self.name, self.age, self.gender)        def qudongbei(self):          print "%s,%s岁,%s,开车去东北" %(self.name, self.age, self.gender)        def dabaojian(self):          print "%s,%s岁,%s,最爱大保健" %(self.name, self.age, self.gender)      xiaoming = Foo('小明', 10, '男')  xiaoming.kanchai()  xiaoming.qudongbei()  xiaoming.dabaojian()    laoli = Foo('老李', 90, '男')  laoli.kanchai()  laoli.qudongbei()  laoli.dabaojian() |

上述对比可以看出，如果使用函数式编程，需要在每次执行函数时传入相同的参数，如果参数多的话，又需要粘贴复制了… ；而对于面向对象只需要在创建对象时，将所有需要的参数封装到当前对象中，之后再次使用时，通过self间接去当前对象中取值即可。

****练习二****：游戏人生程序

1、创建三个游戏人物，分别是：

* 苍井井，女，18，初始战斗力1000
* 东尼木木，男，20，初始战斗力1800
* 波多多，女，19，初始战斗力2500

2、游戏场景，分别：

* 草丛战斗，消耗200战斗力
* 自我修炼，增长100战斗力
* 多人游戏，消耗500战斗力



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | # -\*- coding:utf-8 -\*-    # #####################  定义实现功能的类  #####################    class Person:        def \_\_init\_\_(self, na, gen, age, fig):          self.name = na          self.gender = gen          self.age = age          self.fight =fig        def grassland(self):          """注释：草丛战斗，消耗200战斗力"""            self.fight = self.fight - 200        def practice(self):          """注释：自我修炼，增长100战斗力"""            self.fight = self.fight + 200        def incest(self):          """注释：多人游戏，消耗500战斗力"""            self.fight = self.fight - 500        def detail(self):          """注释：当前对象的详细情况"""            temp = "姓名:%s ; 性别:%s ; 年龄:%s ; 战斗力:%s"  % (self.name, self.gender, self.age, self.fight)          print temp      # #####################  开始游戏  #####################    cang = Person('苍井井', '女', 18, 1000)    # 创建苍井井角色  dong = Person('东尼木木', '男', 20, 1800)  # 创建东尼木木角色  bo = Person('波多多', '女', 19, 2500)      # 创建波多多角色    cang.incest() #苍井空参加一次多人游戏  dong.practice()#东尼木木自我修炼了一次  bo.grassland() #波多多参加一次草丛战斗      #输出当前所有人的详细情况  cang.detail()  dong.detail()  bo.detail()      cang.incest() #苍井空又参加一次多人游戏  dong.incest() #东尼木木也参加了一个多人游戏  bo.practice() #波多多自我修炼了一次    #输出当前所有人的详细情况  cang.detail()  dong.detail()  bo.detail() |

****二、继承****

继承，面向对象中的继承和现实生活中的继承相同，即：子可以继承父的内容。

例如：

猫可以：喵喵叫、吃、喝、拉、撒

狗可以：汪汪叫、吃、喝、拉、撒

如果我们要分别为猫和狗创建一个类，那么就需要为 猫 和 狗 实现他们所有的功能，如下所示：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class 猫：        def 喵喵叫(self):          print '喵喵叫'        def 吃(self):          # do something        def 喝(self):          # do something        def 拉(self):          # do something        def 撒(self):          # do something    class 狗：        def 汪汪叫(self):          print '喵喵叫'        def 吃(self):          # do something        def 喝(self):          # do something        def 拉(self):          # do something        def 撒(self):          # do something |

上述代码不难看出，吃、喝、拉、撒是猫和狗都具有的功能，而我们却分别的猫和狗的类中编写了两次。如果使用 继承 的思想，如下实现：

动物：吃、喝、拉、撒

猫：喵喵叫（猫继承动物的功能）

狗：汪汪叫（狗继承动物的功能）



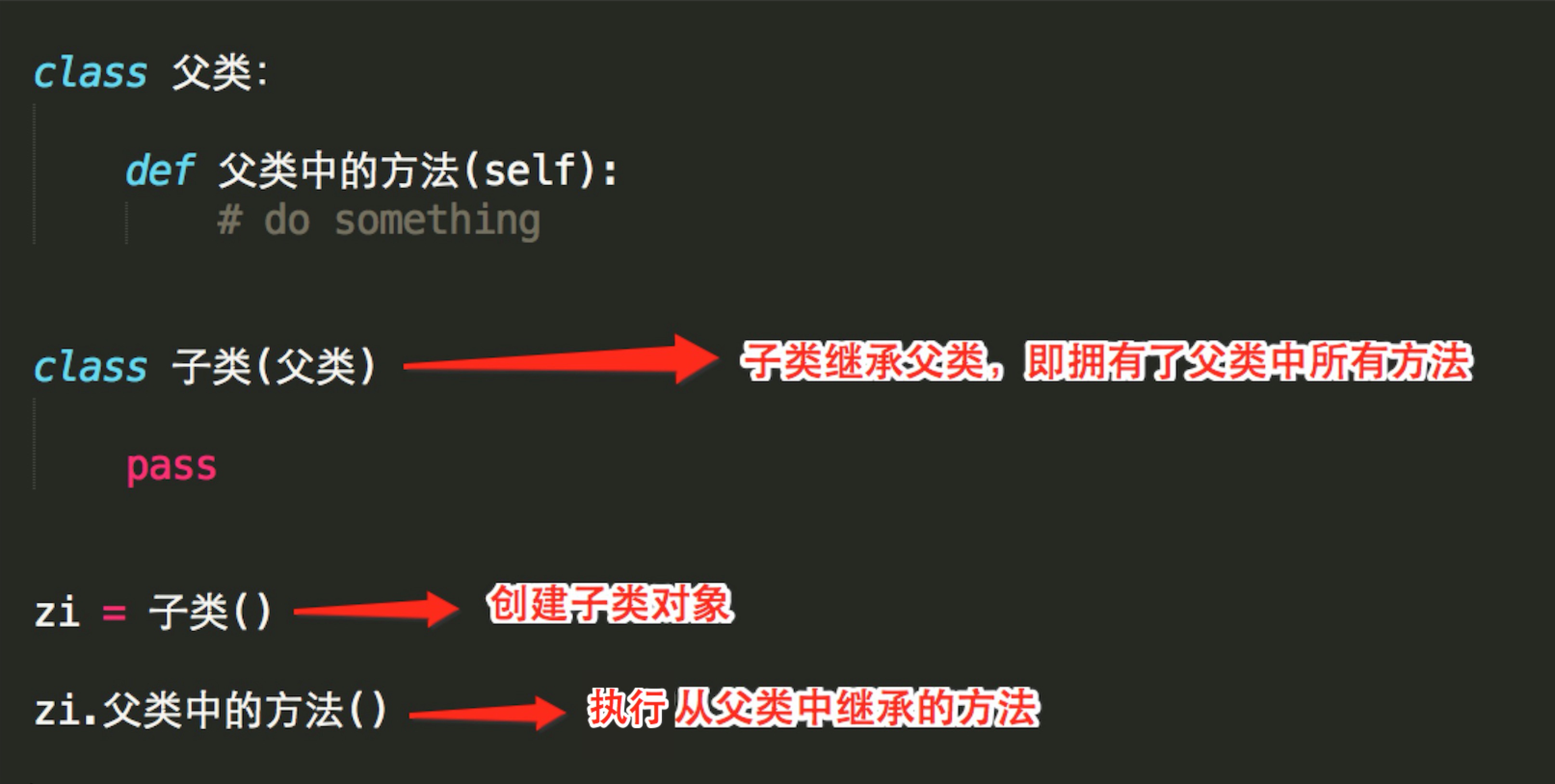
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class 动物:        def 吃(self):          # do something        def 喝(self):          # do something        def 拉(self):          # do something        def 撒(self):          # do something    # 在类后面括号中写入另外一个类名，表示当前类继承另外一个类  class 猫(动物)：        def 喵喵叫(self):          print '喵喵叫'    # 在类后面括号中写入另外一个类名，表示当前类继承另外一个类  class 狗(动物)：        def 汪汪叫(self):          print '喵喵叫' |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | class Animal:        def eat(self):          print "%s 吃 " %self.name        def drink(self):          print "%s 喝 " %self.name        def shit(self):          print "%s 拉 " %self.name        def pee(self):          print "%s 撒 " %self.name      class Cat(Animal):        def \_\_init\_\_(self, name):          self.name = name          self.breed ＝ '猫'        def cry(self):          print '喵喵叫'    class Dog(Animal):        def \_\_init\_\_(self, name):          self.name = name          self.breed ＝ '狗'        def cry(self):          print '汪汪叫'      # ######### 执行 #########    c1 = Cat('小白家的小黑猫')  c1.eat()    c2 = Cat('小黑的小白猫')  c2.drink()    d1 = Dog('胖子家的小瘦狗')  d1.eat() |

****所以，对于面向对象的继承来说，其实就是将多个类共有的方法提取到父类中，子类仅需继承父类而不必一一实现每个方法。****

注：除了子类和父类的称谓，你可能看到过 派生类 和 基类 ，他们与子类和父类只是叫法不同而已。

[](http://jbcdn2.b0.upaiyun.com/2017/08/b408d1eba72f6489af9bcd4e4eee142a.jpg)

学习了继承的写法之后，我们用代码来是上述阿猫阿狗的功能：



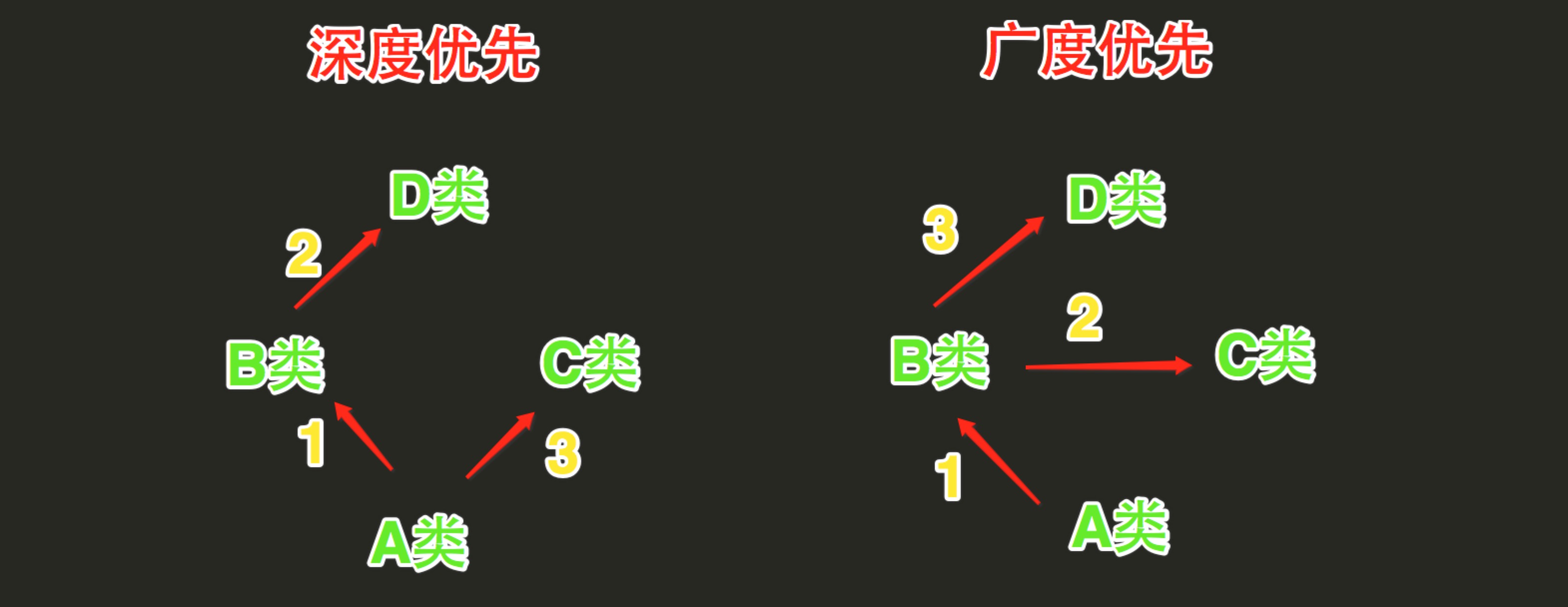
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | class Animal:        def eat(self):          print "%s 吃 " %self.name        def drink(self):          print "%s 喝 " %self.name        def shit(self):          print "%s 拉 " %self.name        def pee(self):          print "%s 撒 " %self.name      class Cat(Animal):        def \_\_init\_\_(self, name):          self.name = name          self.breed ＝ '猫'        def cry(self):          print '喵喵叫'    class Dog(Animal):        def \_\_init\_\_(self, name):          self.name = name          self.breed ＝ '狗'        def cry(self):          print '汪汪叫'      # ######### 执行 #########    c1 = Cat('小白家的小黑猫')  c1.eat()    c2 = Cat('小黑的小白猫')  c2.drink()    d1 = Dog('胖子家的小瘦狗')  d1.eat() |

****那么问题又来了，多继承呢？****

* 是否可以继承多个类
* 如果继承的多个类每个类中都定了相同的函数，那么那一个会被使用呢？

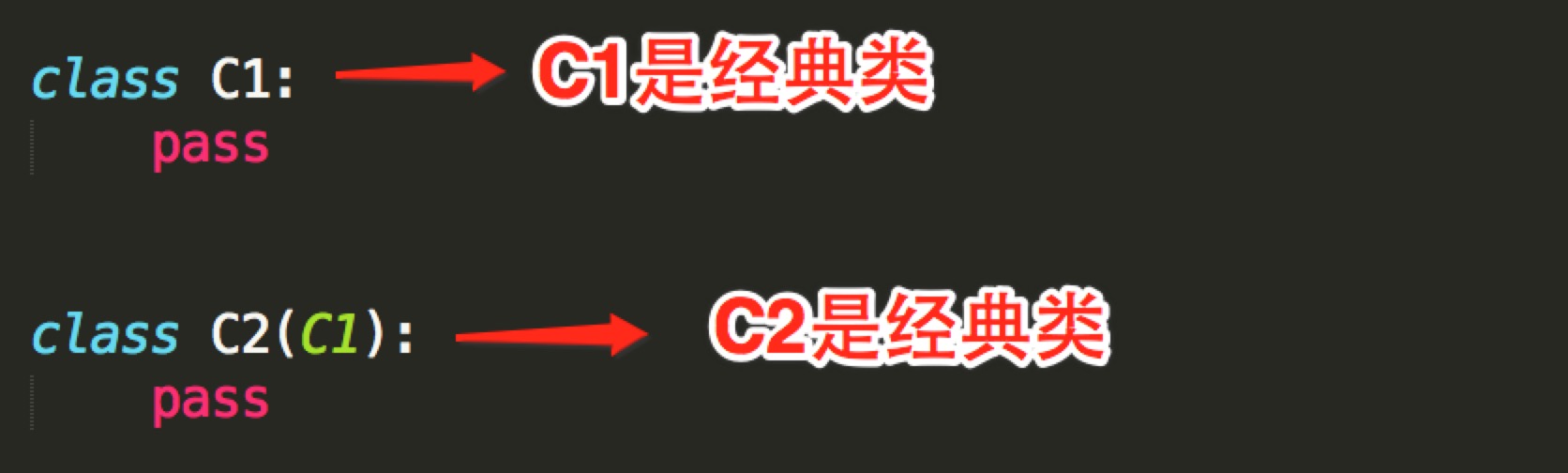
1、Python的类可以继承多个类，Java和C#中则只能继承一个类

2、Python的类如果继承了多个类，那么其寻找方法的方式有两种，分别是：****深度优先****和****广度优先****

[](http://jbcdn2.b0.upaiyun.com/2017/08/ebbcbd47ab37f53f10a29b06396bf15b.jpg)

* 当类是经典类时，多继承情况下，会按照深度优先方式查找
* 当类是新式类时，多继承情况下，会按照广度优先方式查找

经典类和新式类，从字面上可以看出一个老一个新，新的必然包含了跟多的功能，也是之后推荐的写法，从写法上区分的话，如果 ****当前类或者父类继承了object类****，那么该类便是新式类，否则便是经典类。

[](http://jbcdn2.b0.upaiyun.com/2017/08/46fe0781424b4d434ccb46892187a2d8.jpg) [](http://jbcdn2.b0.upaiyun.com/2017/08/1caa8bd2777d2591f5367a36a19739d9.jpg)



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class D:        def bar(self):          print 'D.bar'      class C(D):        def bar(self):          print 'C.bar'      class B(D):        def bar(self):          print 'B.bar'      class A(B, C):        def bar(self):          print 'A.bar'    a = A()  # 执行bar方法时  # 首先去A类中查找，如果A类中没有，则继续去B类中找，如果B类中么有，则继续去D类中找，如果D类中么有，则继续去C类中找，如果还是未找到，则报错  # 所以，查找顺序：A --> B --> D --> C  # 在上述查找bar方法的过程中，一旦找到，则寻找过程立即中断，便不会再继续找了  a.bar() |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class D(object):        def bar(self):          print 'D.bar'      class C(D):        def bar(self):          print 'C.bar'      class B(D):        def bar(self):          print 'B.bar'      class A(B, C):        def bar(self):          print 'A.bar'    a = A()  # 执行bar方法时  # 首先去A类中查找，如果A类中没有，则继续去B类中找，如果B类中么有，则继续去C类中找，如果C类中么有，则继续去D类中找，如果还是未找到，则报错  # 所以，查找顺序：A --> B --> C --> D  # 在上述查找bar方法的过程中，一旦找到，则寻找过程立即中断，便不会再继续找了  a.bar() |

经典类：首先去****A****类中查找，如果A类中没有，则继续去****B****类中找，如果B类中么有，则继续去****D****类中找，如果D类中么有，则继续去****C****类中找，如果还是未找到，则报错

新式类：首先去****A****类中查找，如果A类中没有，则继续去****B****类中找，如果B类中么有，则继续去****C****类中找，如果C类中么有，则继续去****D****类中找，如果还是未找到，则报错

注意：在上述查找过程中，一旦找到，则寻找过程立即中断，便不会再继续找了。

****三、多态****

Pyhon不支持Java和C#这一类强类型语言中多态的写法，但是原生多态，其Python崇尚“鸭子类型”。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class F1:      pass      class S1(F1):        def show(self):          print 'S1.show'      class S2(F1):        def show(self):          print 'S2.show'      # 由于在Java或C#中定义函数参数时，必须指定参数的类型  # 为了让Func函数既可以执行S1对象的show方法，又可以执行S2对象的show方法，所以，定义了一个S1和S2类的父类  # 而实际传入的参数是：S1对象和S2对象    def Func(F1 obj):      """Func函数需要接收一个F1类型或者F1子类的类型"""        print obj.show()    s1\_obj = S1()  Func(s1\_obj) # 在Func函数中传入S1类的对象 s1\_obj，执行 S1 的show方法，结果：S1.show    s2\_obj = S2()  Func(s2\_obj) # 在Func函数中传入Ss类的对象 ss\_obj，执行 Ss 的show方法，结果：S2.show |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class F1:      pass      class S1(F1):        def show(self):          print 'S1.show'      class S2(F1):        def show(self):          print 'S2.show'    def Func(obj):      print obj.show()    s1\_obj = S1()  Func(s1\_obj)    s2\_obj = S2()  Func(s2\_obj) |

# ****总结****

以上就是本节对于面向对象初级知识的介绍，总结如下：

* 面向对象是一种编程方式，此编程方式的实现是基于对 ****类**** 和 ****对象**** 的使用
* 类 是一个模板，模板中包装了多个“函数”供使用
* 对象，根据模板创建的实例（即：对象），实例用于调用被包装在类中的函数
* 面向对象三大特性：封装、继承和多态

****问答专区****

****问题一：什么样的代码才是面向对象？****

答：从简单来说，如果程序中的所有功能都是用 类 和 对象 来实现，那么就是面向对象编程了。

****问题二：函数式编程 和 面向对象 如何选择？分别在什么情况下使用？****

答：须知：对于 C# 和 Java 程序员来说不存在这个问题，因为该两门语言只支持面向对象编程（不支持函数式编程）。而对于 Python 和 PHP 等语言却同时支持两种编程方式，且函数式编程能完成的操作，面向对象都可以实现；而面向对象的能完成的操作，函数式编程不行（函数式编程无法实现面向对象的封装功能）。

所以，一般在Python开发中，****全部使用面向对象**** 或 ****面向对象和函数式混合使用****

面向对象的应用场景:

1. 多函数需使用共同的值，如：数据库的增、删、改、查操作都需要连接数据库字符串、主机名、用户名和密码



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class SqlHelper:        def \_\_init\_\_(self, host, user, pwd):            self.host = host          self.user = user          self.pwd = pwd        def 增(self):          # 使用主机名、用户名、密码（self.host 、self.user 、self.pwd）打开数据库连接          # do something          # 关闭数据库连接        def 删(self):          # 使用主机名、用户名、密码（self.host 、self.user 、self.pwd）打开数据库连接          # do something          # 关闭数据库连接        def 改(self):          # 使用主机名、用户名、密码（self.host 、self.user 、self.pwd）打开数据库连接          # do something          # 关闭数据库连接        def 查(self):      # 使用主机名、用户名、密码（self.host 、self.user 、self.pwd）打开数据库连接          # do something          # 关闭数据库连接# do something |

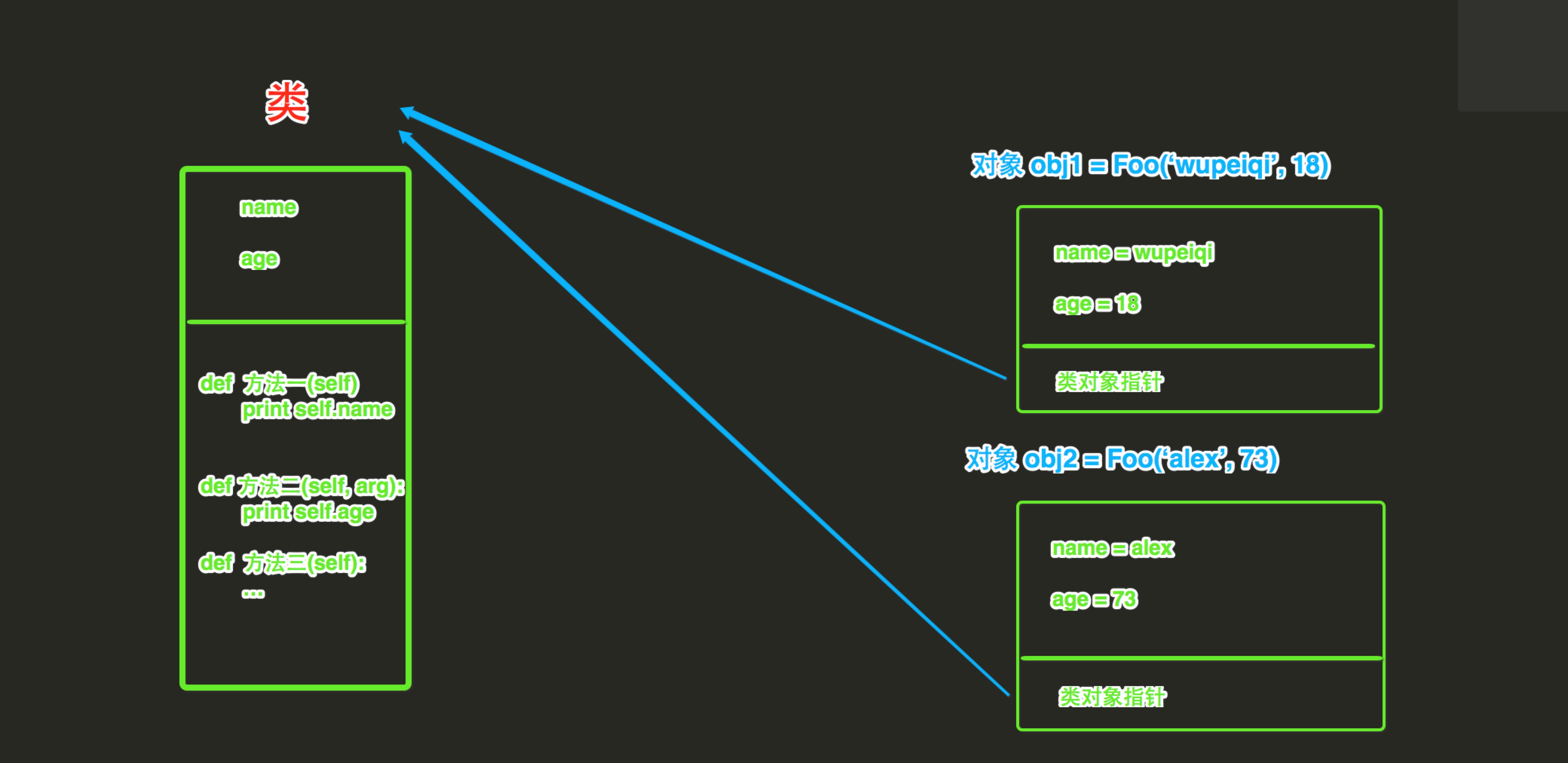
1. 需要创建多个事物，每个事物属性个数相同，但是值的需求  
   如：张三、李四、杨五，他们都有姓名、年龄、血型，但其都是不相同。即：属性个数相同，但值不相同



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person:        def \_\_init\_\_(self, name ,age ,blood\_type):            self.name = name          self.age = age          self.blood\_type = blood\_type          def detail(self):          temp = "i am %s, age %s , blood type %s " % (self.name, self.age, self.blood\_type)          print temp    zhangsan = Person('张三', 18, 'A')  lisi = Person('李四', 73, 'AB')  yangwu = Person('杨五', 84, 'A') |

****问题三：类和对象在内存中是如何保存？****

答：类以及类中的方法在内存中只有一份，而根据类创建的每一个对象都在内存中需要存一份，大致如下图：

[](http://jbcdn2.b0.upaiyun.com/2017/08/77ce69983f5f003910200d94d6f3216e.jpg)

如上图所示，根据类创建对象时，对象中除了封装 name 和 age 的值之外，还会保存一个****类对象指针****，该值指向当前对象的类。

当通过 obj1 执行 【方法一】 时，过程如下：

1. 根据当前对象中的 类对象指针 找到类中的方法
2. 将对象 obj1 当作参数传给 方法的第一个参数 self

# Python 面向对象（进阶篇）

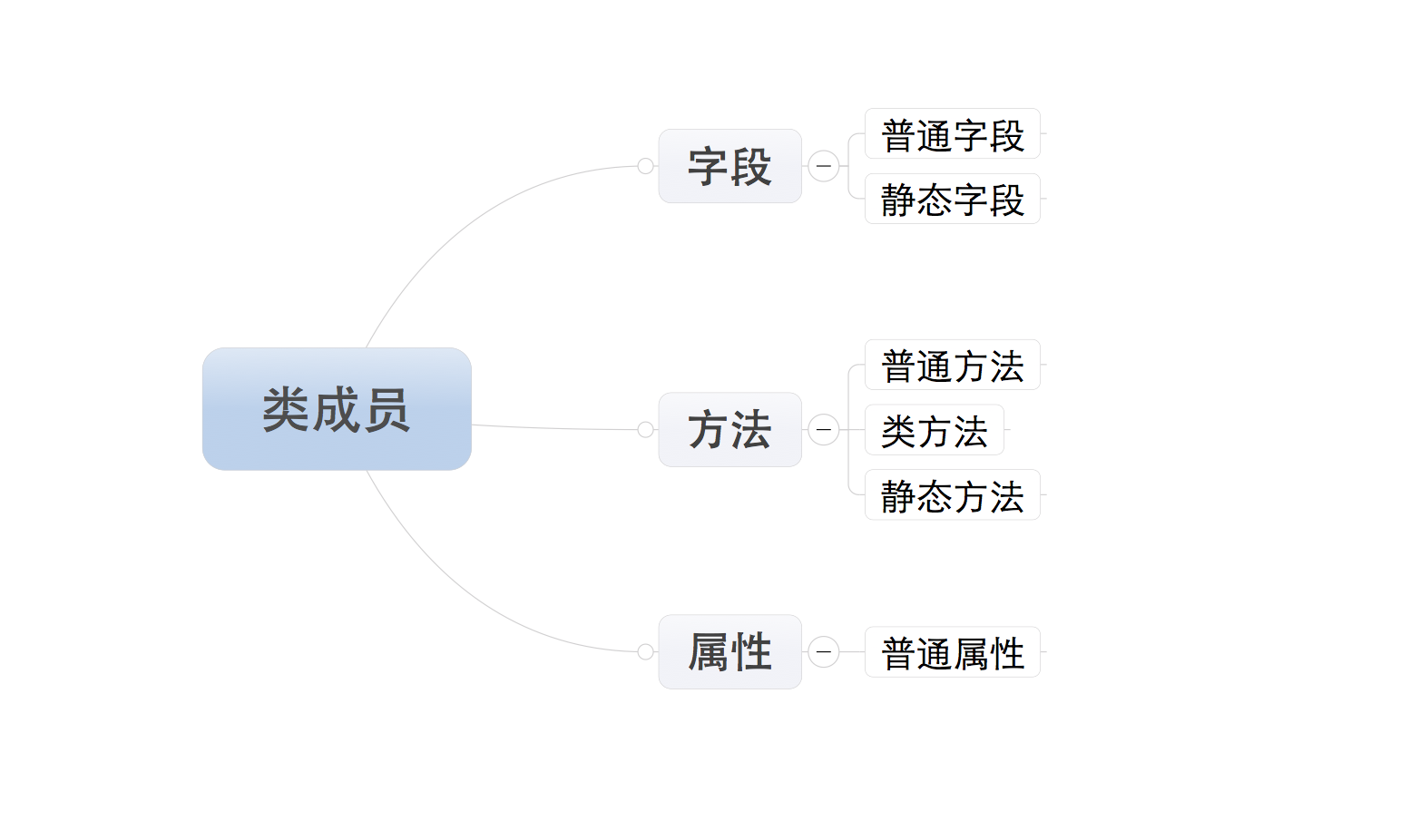
上一篇[《Python 面向对象（初级篇）》](http://python.jobbole.com/88214/" \t "http://python.jobbole.com/88230/_blank)文章介绍了面向对象基本知识：

* 面向对象是一种编程方式，此编程方式的实现是基于对 ****类**** 和 ****对象**** 的使用
* 类 是一个模板，模板中包装了多个“函数”供使用（可以讲多函数中公用的变量封装到对象中）
* 对象，根据模板创建的实例（即：对象），实例用于调用被包装在类中的函数
* 面向对象三大特性：封装、继承和多态

本篇将详细介绍Python 类的成员、成员修饰符、类的特殊成员。

# 类的成员

类的成员可以分为三大类：字段、方法和属性

[](http://jbcdn2.b0.upaiyun.com/2017/08/b30680eb1ad249a6837d3ebbd17382ec.png)

注：所有成员中，只有普通字段的内容保存对象中，即：根据此类创建了多少对象，在内存中就有多少个普通字段。而其他的成员，则都是保存在类中，即：无论对象的多少，在内存中只创建一份。

****一、字段****

字段包括：普通字段和静态字段，他们在定义和使用中有所区别，而最本质的区别是内存中保存的位置不同，

* 普通字段属于****对象****
* 静态字段属于****类****



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Province:        # 静态字段      country ＝ '中国'        def \_\_init\_\_(self, name):            # 普通字段          self.name = name      # 直接访问普通字段  obj = Province('河北省')  print obj.name    # 直接访问静态字段  Province.country |

由上述代码可以看出【普通字段需要通过对象来访问】【静态字段通过类访问】，在使用上可以看出普通字段和静态字段的归属是不同的。其在内容的存储方式类似如下图：

[](http://jbcdn2.b0.upaiyun.com/2017/08/905ec6b41b21f7d8c0108656ba158b74.jpg)

由上图可是：

* 静态字段在内存中只保存一份
* 普通字段在每个对象中都要保存一份

应用场景： 通过类创建对象时，如果每个对象都具有相同的字段，那么就使用静态字段

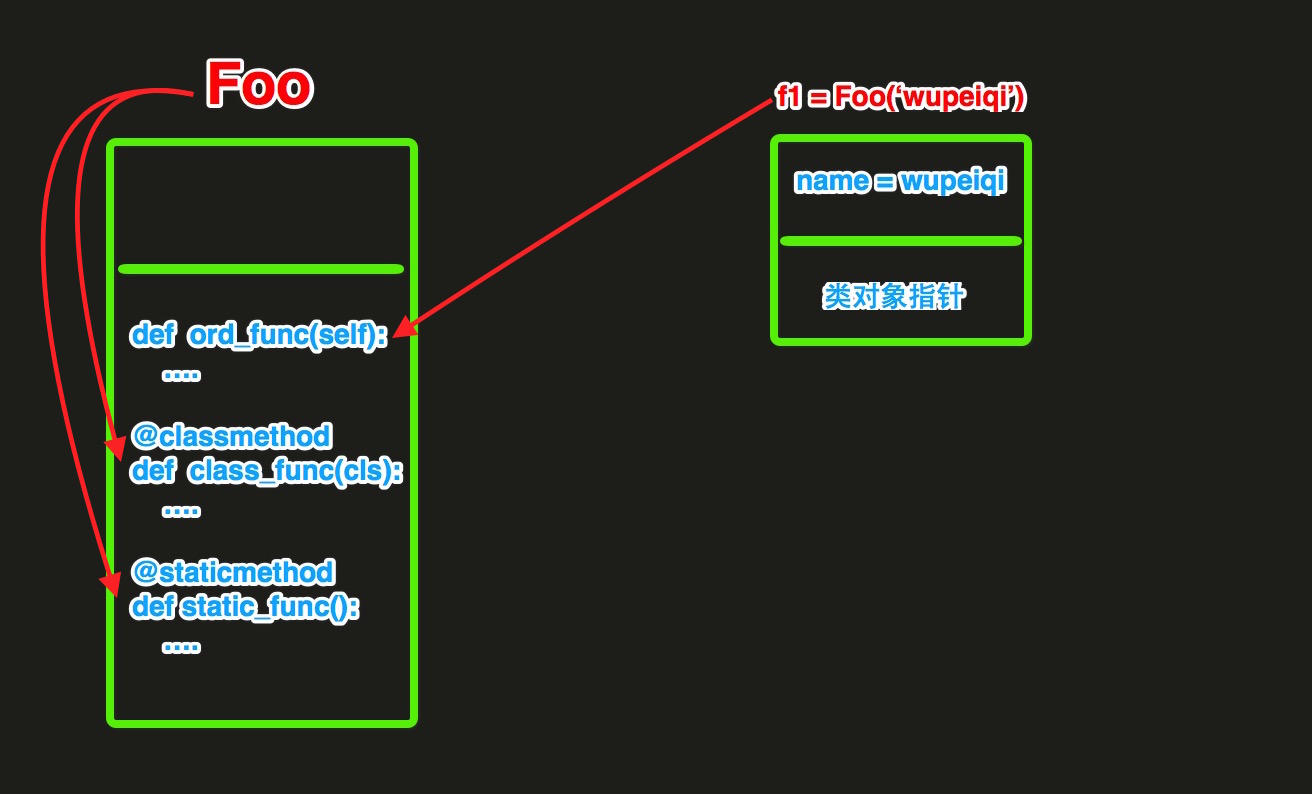
****二、方法****

方法包括：普通方法、静态方法和类方法，三种方法在****内存中都归属于类****，区别在于调用方式不同。

* 普通方法：由****对象****调用；至少一个****self****参数；执行普通方法时，自动将调用该方法的****对象****赋值给****self****；
* 类方法：由****类****调用； 至少一个****cls****参数；执行类方法时，自动将调用该方法的****类****复制给****cls****；
* 静态方法：由****类****调用；无默认参数；



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class Foo:        def \_\_init\_\_(self, name):          self.name = name        def ord\_func(self):          """ 定义普通方法，至少有一个self参数 """            # print self.name          print '普通方法'        @classmethod      def class\_func(cls):          """ 定义类方法，至少有一个cls参数 """            print '类方法'        @staticmethod      def static\_func():          """ 定义静态方法 ，无默认参数"""            print '静态方法'      # 调用普通方法  f = Foo()  f.ord\_func()    # 调用类方法  Foo.class\_func()    # 调用静态方法  Foo.static\_func() |

**[](http://jbcdn2.b0.upaiyun.com/2017/08/293d5a5b0462c3e2da015391d33240ba.jpg)**

****相同点：****对于所有的方法而言，均属于类（非对象）中，所以，在内存中也只保存一份。

****不同点：****方法调用者不同、调用方法时自动传入的参数不同。

****三、属性****

如果你已经了解Python类中的方法，那么属性就非常简单了，因为Python中的属性其实是****普通方法****的变种。

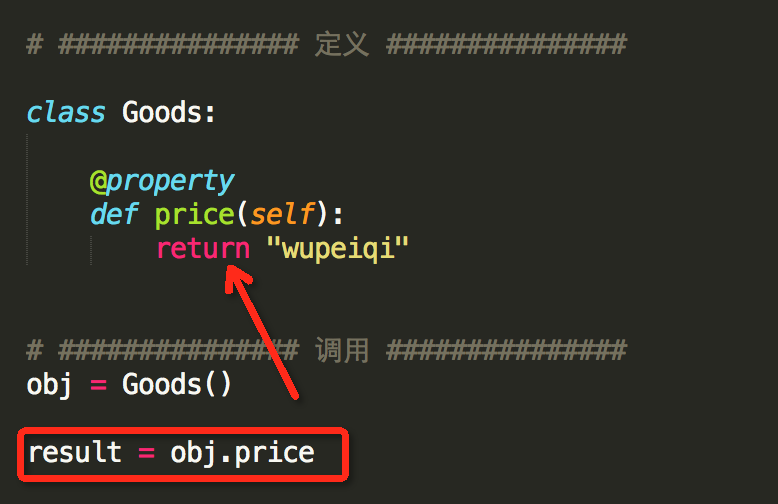
对于属性，有以下三个知识点：

* 属性的基本使用
* 属性的两种定义方式

****1、属性的基本使用****



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # ############### 定义 ###############  class Foo:        def func(self):          pass        # 定义属性      @property      def prop(self):          pass  # ############### 调用 ###############  foo\_obj = Foo()    foo\_obj.func()  foo\_obj.prop   #调用属性 |

[](http://jbcdn2.b0.upaiyun.com/2017/08/f0c0dd4fed3f5c28befc1a037650db60.png)

由属性的定义和调用要注意一下几点：

* 定义时，在普通方法的基础上添加 ****@property**** 装饰器；
* 定义时，属性****仅有一个****self参数
* 调用时，无需****括号****  
  方法：foo\_obj.func()  
  属性：foo\_obj.prop

注意：属性存在意义是：访问属性时可以制造出和访问字段完全相同的假象

属性由方法变种而来，如果Python中没有属性，方法完全可以代替其功能。

实例：对于主机列表页面，每次请求不可能把数据库中的所有内容都显示到页面上，而是通过分页的功能局部显示，所以在向数据库中请求数据时就要显示的指定获取从第m条到第n条的所有数据（即：limit m,n），这个分页的功能包括：

* 根据用户请求的当前页和总数据条数计算出 m 和 n
* 根据m 和 n 去数据库中请求数据



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | # ############### 定义 ###############  class Pager:        def \_\_init\_\_(self, current\_page):          # 用户当前请求的页码（第一页、第二页...）          self.current\_page = current\_page          # 每页默认显示10条数据          self.per\_items = 10          @property      def start(self):          val = (self.current\_page - 1) \* self.per\_items          return val        @property      def end(self):          val = self.current\_page \* self.per\_items          return val    # ############### 调用 ###############    p = Pager(1)  p.start 就是起始值，即：m  p.end   就是结束值，即：n |

从上述可见，Python的属性的功能是：属性内部进行一系列的逻辑计算，最终将计算结果返回。

****2、属性的两种定义方式****

属性的定义有两种方式：

* 装饰器 即：在方法上应用装饰器
* 静态字段 即：在类中定义值为property对象的静态字段

****装饰器方式：在类的普通方法上应用@property装饰器****

我们知道Python中的类有经典类和新式类，新式类的属性比经典类的属性丰富。（ 如果类继object，那么该类是新式类 ）  
****经典类****，具有一种@property装饰器（如上一步实例）



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # ############### 定义 ###############  class Goods:        @property      def price(self):          return "wupeiqi"  # ############### 调用 ###############  obj = Goods()  result = obj.price  # 自动执行 @property 修饰的 price 方法，并获取方法的返回值 |

****新式类****，具有三种@property装饰器



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | # ############### 定义 ###############  class Goods(object):        @property      def price(self):          print '@property'        @price.setter      def price(self, value):          print '@price.setter'        @price.deleter      def price(self):          print '@price.deleter'    # ############### 调用 ###############  obj = Goods()    obj.price          # 自动执行 @property 修饰的 price 方法，并获取方法的返回值    obj.price = 123    # 自动执行 @price.setter 修饰的 price 方法，并将  123 赋值给方法的参数    del obj.price      # 自动执行 @price.deleter 修饰的 price 方法 |

注：经典类中的属性只有一种访问方式，其对应被 @property 修饰的方法  
新式类中的属性有三种访问方式，并分别对应了三个被@property、@方法名.setter、@方法名.deleter修饰的方法

由于新式类中具有三种访问方式，我们可以根据他们几个属性的访问特点，分别将三个方法定义为对同一个属性：获取、修改、删除



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Goods(object):        def \_\_init\_\_(self):          # 原价          self.original\_price = 100          # 折扣          self.discount = 0.8        @property      def price(self):          # 实际价格 = 原价 \* 折扣          new\_price = self.original\_price \* self.discount          return new\_price        @price.setter      def price(self, value):          self.original\_price = value        @price.deltter      def price(self, value):          del self.original\_price    obj = Goods()  obj.price         # 获取商品价格  obj.price = 200   # 修改商品原价  del obj.price     # 删除商品原价 |

****静态字段方式，创建值为property对象的静态字段****

当使用静态字段的方式创建属性时，经典类和新式类无区别



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Foo:        def get\_bar(self):          return 'wupeiqi'        BAR = property(get\_bar)    obj = Foo()  reuslt = obj.BAR        # 自动调用get\_bar方法，并获取方法的返回值  print reuslt |

property的构造方法中有个四个参数

* 第一个参数是****方法名****，调用 对象.属性 时自动触发执行方法
* 第二个参数是****方法名****，调用 对象.属性 ＝ XXX 时自动触发执行方法
* 第三个参数是****方法名****，调用 del 对象.属性 时自动触发执行方法
* 第四个参数是****字符串****，调用 对象.属性.\_\_doc\_\_ ，此参数是该属性的描述信息



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class Foo：        def get\_bar(self):          return 'wupeiqi'        # \*必须两个参数      def set\_bar(self, value):          return return 'set value' + value        def del\_bar(self):          return 'wupeiqi'        BAR ＝ property(get\_bar, set\_bar, del\_bar, 'description...')    obj = Foo()    obj.BAR              # 自动调用第一个参数中定义的方法：get\_bar  obj.BAR = "alex"     # 自动调用第二个参数中定义的方法：set\_bar方法，并将“alex”当作参数传入  del Foo.BAR          # 自动调用第三个参数中定义的方法：del\_bar方法  obj.BAE.\_\_doc\_\_      # 自动获取第四个参数中设置的值：description... |

由于静态字段方式创建属性具有三种访问方式，我们可以根据他们几个属性的访问特点，分别将三个方法定义为对同一个属性：获取、修改、删除



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Goods(object):        def \_\_init\_\_(self):          # 原价          self.original\_price = 100          # 折扣          self.discount = 0.8        def get\_price(self):          # 实际价格 = 原价 \* 折扣          new\_price = self.original\_price \* self.discount          return new\_price        def set\_price(self, value):          self.original\_price = value        def del\_price(self, value):          del self.original\_price        PRICE = property(get\_price, set\_price, del\_price, '价格属性描述...')    obj = Goods()  obj.PRICE         # 获取商品价格  obj.PRICE = 200   # 修改商品原价  del obj.PRICE     # 删除商品原价 |

注意：Python WEB框架 Django 的视图中 request.POST 就是使用的静态字段的方式创建的属性



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | class WSGIRequest(http.HttpRequest):      def \_\_init\_\_(self, environ):          script\_name = get\_script\_name(environ)          path\_info = get\_path\_info(environ)          if not path\_info:              # Sometimes PATH\_INFO exists, but is empty (e.g. accessing              # the SCRIPT\_NAME URL without a trailing slash). We really need to              # operate as if they'd requested '/'. Not amazingly nice to force              # the path like this, but should be harmless.              path\_info = '/'          self.environ = environ          self.path\_info = path\_info          self.path = '%s/%s' % (script\_name.rstrip('/'), path\_info.lstrip('/'))          self.META = environ          self.META['PATH\_INFO'] = path\_info          self.META['SCRIPT\_NAME'] = script\_name          self.method = environ['REQUEST\_METHOD'].upper()          \_, content\_params = cgi.parse\_header(environ.get('CONTENT\_TYPE', ''))          if 'charset' in content\_params:              try:                  codecs.lookup(content\_params['charset'])              except LookupError:                  pass              else:                  self.encoding = content\_params['charset']          self.\_post\_parse\_error = False          try:              content\_length = int(environ.get('CONTENT\_LENGTH'))          except (ValueError, TypeError):              content\_length = 0          self.\_stream = LimitedStream(self.environ['wsgi.input'], content\_length)          self.\_read\_started = False          self.resolver\_match = None        def \_get\_scheme(self):          return self.environ.get('wsgi.url\_scheme')        def \_get\_request(self):          warnings.warn('`request.REQUEST` is deprecated, use `request.GET` or '                        '`request.POST` instead.', RemovedInDjango19Warning, 2)          if not hasattr(self, '\_request'):              self.\_request = datastructures.MergeDict(self.POST, self.GET)          return self.\_request        @cached\_property      def GET(self):          # The WSGI spec says 'QUERY\_STRING' may be absent.          raw\_query\_string = get\_bytes\_from\_wsgi(self.environ, 'QUERY\_STRING', '')          return http.QueryDict(raw\_query\_string, encoding=self.\_encoding)        # ############### 看这里看这里  ###############      def \_get\_post(self):          if not hasattr(self, '\_post'):              self.\_load\_post\_and\_files()          return self.\_post        # ############### 看这里看这里  ###############      def \_set\_post(self, post):          self.\_post = post        @cached\_property      def COOKIES(self):          raw\_cookie = get\_str\_from\_wsgi(self.environ, 'HTTP\_COOKIE', '')          return http.parse\_cookie(raw\_cookie)        def \_get\_files(self):          if not hasattr(self, '\_files'):              self.\_load\_post\_and\_files()          return self.\_files        # ############### 看这里看这里  ###############      POST = property(\_get\_post, \_set\_post)        FILES = property(\_get\_files)      REQUEST = property(\_get\_request) |

所以，定义属性共有两种方式，分别是【装饰器】和【静态字段】，而【装饰器】方式针对经典类和新式类又有所不同。

# 类成员的修饰符

类的所有成员在上一步骤中已经做了详细的介绍，对于每一个类的成员而言都有两种形式：

* 公有成员，在任何地方都能访问
* 私有成员，只有在类的内部才能方法

****私有成员和公有成员的定义不同****：私有成员命名时，前两个字符是下划线。（特殊成员除外，例如：\_\_init\_\_、\_\_call\_\_、\_\_dict\_\_等）



|  |  |
| --- | --- |
| 1  2  3  4  5 | class C:      def \_\_init\_\_(self):          self.name = '公有字段'          self.\_\_foo = "私有字段" |

****私有成员和公有成员的访问限制不同****：

静态字段

* 公有静态字段：类可以访问；类内部可以访问；派生类中可以访问
* 私有静态字段：仅类内部可以访问；



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class C:        name = "公有静态字段"        def func(self):          print C.name    class D(C):        def show(self):          print C.name      C.name         # 类访问    obj = C()  obj.func()     # 类内部可以访问    obj\_son = D()  obj\_son.show() # 派生类中可以访问 |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class C:        \_\_name = "公有静态字段"        def func(self):          print C.\_\_name    class D(C):        def show(self):          print C.\_\_name      C.\_\_name       # 类访问            ==> 错误    obj = C()  obj.func()     # 类内部可以访问     ==> 正确    obj\_son = D()  obj\_son.show() # 派生类中可以访问   ==> 错误 |

普通字段

* 公有普通字段：对象可以访问；类内部可以访问；派生类中可以访问
* 私有普通字段：仅类内部可以访问；

ps：如果想要强制访问私有字段，可以通过 【对象.\_类名\_\_私有字段明 】访问（如：obj.\_C\_\_foo），不建议强制访问私有成员。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class C:        def \_\_init\_\_(self):          self.foo = "公有字段"        def func(self):          print self.foo 　#　类内部访问    class D(C):        def show(self):          print self.foo　＃　派生类中访问    obj = C()    obj.foo     # 通过对象访问  obj.func()  # 类内部访问    obj\_son = D();  obj\_son.show()  # 派生类中访问 |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class C:        def \_\_init\_\_(self):          self.\_\_foo = "私有字段"        def func(self):          print self.foo 　#　类内部访问    class D(C):        def show(self):          print self.foo　＃　派生类中访问    obj = C()    obj.\_\_foo     # 通过对象访问    ==> 错误  obj.func()  # 类内部访问        ==> 正确    obj\_son = D();  obj\_son.show()  # 派生类中访问  ==> 错误 |

方法、属性的访问于上述方式相似，即：私有成员只能在类内部使用

ps：非要访问私有属性的话，可以通过 对象.\_类\_\_属性名

# 类的特殊成员

上文介绍了Python的类成员以及成员修饰符，从而了解到类中有字段、方法和属性三大类成员，并且成员名前如果有两个下划线，则表示该成员是私有成员，私有成员只能由类内部调用。无论人或事物往往都有不按套路出牌的情况，Python的类成员也是如此，存在着一些具有特殊含义的成员，详情如下：

****1. \_\_doc\_\_****

表示类的描述信息



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Foo:      """ 描述类信息，这是用于看片的神奇 """        def func(self):          pass    print Foo.\_\_doc\_\_  #输出：类的描述信息 |

****2. \_\_module\_\_ 和 \_\_class\_\_****

\_\_module\_\_ 表示当前操作的对象在那个模块

\_\_class\_\_ 表示当前操作的对象的类是什么



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-    class C:        def \_\_init\_\_(self):          self.name = 'wupeiqi'    lib/aa.py |



|  |  |
| --- | --- |
| 1  2  3  4  5 | from lib.aa import C    obj = C()  print obj.\_\_module\_\_  # 输出 lib.aa，即：输出模块  print obj.\_\_class\_\_      # 输出 lib.aa.C，即：输出类 |

****3. \_\_init\_\_****

构造方法，通过类创建对象时，自动触发执行。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Foo:        def \_\_init\_\_(self, name):          self.name = name          self.age = 18      obj = Foo('wupeiqi') # 自动执行类中的 \_\_init\_\_ 方法 |

****4. \_\_del\_\_****

析构方法，当对象在内存中被释放时，自动触发执行。

注：此方法一般无须定义，因为Python是一门高级语言，程序员在使用时无需关心内存的分配和释放，因为此工作都是交给Python解释器来执行，所以，析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。



|  |  |
| --- | --- |
| 1  2  3  4 | class Foo:        def \_\_del\_\_(self):          pass |

****5. \_\_call\_\_****

对象后面加括号，触发执行。

注：构造方法的执行是由创建对象触发的，即：对象 = 类名() ；而对于 \_\_call\_\_ 方法的执行是由对象后加括号触发的，即：对象() 或者 类()()



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Foo:        def \_\_init\_\_(self):          pass        def \_\_call\_\_(self, \*args, \*\*kwargs):            print '\_\_call\_\_'      obj = Foo() # 执行 \_\_init\_\_  obj()       # 执行 \_\_call\_\_ |

****6. \_\_dict\_\_****

类或对象中的所有成员

上文中我们知道：类的普通字段属于对象；类中的静态字段和方法等属于类，即：

[](http://jbcdn2.b0.upaiyun.com/2017/08/7db56d9c174505f6130cd93aaf441155.jpg)



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Province:        country = 'China'        def \_\_init\_\_(self, name, count):          self.name = name          self.count = count        def func(self, \*args, \*\*kwargs):          print 'func'    # 获取类的成员，即：静态字段、方法、  print Province.\_\_dict\_\_  # 输出：{'country': 'China', '\_\_module\_\_': '\_\_main\_\_', 'func': <function func at 0x10be30f50>, '\_\_init\_\_': <function \_\_init\_\_ at 0x10be30ed8>, '\_\_doc\_\_': None}    obj1 = Province('HeBei',10000)  print obj1.\_\_dict\_\_  # 获取 对象obj1 的成员  # 输出：{'count': 10000, 'name': 'HeBei'}    obj2 = Province('HeNan', 3888)  print obj2.\_\_dict\_\_  # 获取 对象obj1 的成员  # 输出：{'count': 3888, 'name': 'HeNan'} |

****7. \_\_str\_\_****

如果一个类中定义了\_\_str\_\_方法，那么在打印 对象 时，默认输出该方法的返回值。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Foo:        def \_\_str\_\_(self):          return 'wupeiqi'      obj = Foo()  print obj  # 输出：wupeiqi |

****8、\_\_getitem\_\_、\_\_setitem\_\_、\_\_delitem\_\_****

用于索引操作，如字典。以上分别表示获取、设置、删除数据



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-  class Foo(object):      def \_\_getitem\_\_(self, key):          print '\_\_getitem\_\_',key      def \_\_setitem\_\_(self, key, value):          print '\_\_setitem\_\_',key,value      def \_\_delitem\_\_(self, key):          print '\_\_delitem\_\_',key  obj = Foo()  result = obj['k1']      # 自动触发执行 \_\_getitem\_\_  obj['k2'] = 'wupeiqi'   # 自动触发执行 \_\_setitem\_\_  del obj['k1']           # 自动触发执行 \_\_delitem\_\_ |

****9、\_\_getslice\_\_、\_\_setslice\_\_、\_\_delslice\_\_****

该三个方法用于分片操作，如：列表



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-  class Foo(object):      def \_\_getslice\_\_(self, i, j):          print '\_\_getslice\_\_',i,j      def \_\_setslice\_\_(self, i, j, sequence):          print '\_\_setslice\_\_',i,j      def \_\_delslice\_\_(self, i, j):          print '\_\_delslice\_\_',i,j  obj = Foo()  obj[-1:1]                   # 自动触发执行 \_\_getslice\_\_  obj[0:1] = [11,22,33,44]    # 自动触发执行 \_\_setslice\_\_  del obj[0:2]                # 自动触发执行 \_\_delslice\_\_ |

****10. \_\_iter\_\_****

用于迭代器，之所以列表、字典、元组可以进行for循环，是因为类型内部定义了 \_\_iter\_\_



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Foo(object):      pass      obj = Foo()    for i in obj:      print i    # 报错：TypeError: 'Foo' object is not iterable |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-    class Foo(object):        def \_\_iter\_\_(self):          pass    obj = Foo()    for i in obj:      print i    # 报错：TypeError: iter() returned non-iterator of type 'NoneType' |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-    class Foo(object):        def \_\_init\_\_(self, sq):          self.sq = sq        def \_\_iter\_\_(self):          return iter(self.sq)    obj = Foo([11,22,33,44])    for i in obj:      print i |

以上步骤可以看出，for循环迭代的其实是 iter([11,22,33,44]) ，所以执行流程可以变更为：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-  obj = iter([11,22,33,44])  for i in obj:      print i |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #!/usr/bin/env python  # -\*- coding:utf-8 -\*-    obj = iter([11,22,33,44])    while True:      val = obj.next()      print val |

****11. \_\_new\_\_ 和 \_\_metaclass\_\_****

阅读以下代码：



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Foo(object):      def \_\_init\_\_(self):          pass  obj = Foo()   # obj是通过Foo类实例化的对象 |

上述代码中，obj 是通过 Foo 类实例化的对象，其实，不仅 obj 是一个对象，Foo类本身也是一个对象，因为在****Python中一切事物都是对象****。

如果按照一切事物都是对象的理论：obj对象是通过执行Foo类的构造方法创建，那么Foo类对象应该也是通过执行某个类的 构造方法 创建。



|  |  |
| --- | --- |
| 1  2 | print type(obj) # 输出：<class '\_\_main\_\_.Foo'>     表示，obj 对象由Foo类创建  print type(Foo) # 输出：<type 'type'>              表示，Foo类对象由 type 类创建 |

所以，****obj对象是Foo类的一个实例****，****Foo类对象是 type 类的一个实例****，即：Foo类对象 是通过type类的构造方法创建。

那么，创建类就可以有两种方式：

****a). 普通方式****



|  |  |
| --- | --- |
| 1  2  3  4 | class Foo(object):        def func(self):          print 'hello wupeiqi' |

****b).特殊方式（type类的构造函数）****



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def func(self):      print 'hello wupeiqi'    Foo = type('Foo',(object,), {'func': func})  #type第一个参数：类名  #type第二个参数：当前类的基类  #type第三个参数：类的成员 |

****＝＝》 类 是由 type 类实例化产生****

那么问题来了，类默认是由 type 类实例化产生，type类中如何实现的创建类？类又是如何创建对象？

答：类中有一个属性 \_\_metaclass\_\_，其用来表示该类由 谁 来实例化创建，所以，我们可以为 \_\_metaclass\_\_ 设置一个type类的派生类，从而查看 类 创建的过程。

[](http://jbcdn2.b0.upaiyun.com/2017/08/d7663a369606c8a343a7483f4fa96691.png)



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class MyType(type):        def \_\_init\_\_(self, what, bases=None, dict=None):          super(MyType, self).\_\_init\_\_(what, bases, dict)        def \_\_call\_\_(self, \*args, \*\*kwargs):          obj = self.\_\_new\_\_(self, \*args, \*\*kwargs)            self.\_\_init\_\_(obj)    class Foo(object):        \_\_metaclass\_\_ = MyType        def \_\_init\_\_(self, name):          self.name = name        def \_\_new\_\_(cls, \*args, \*\*kwargs):          return object.\_\_new\_\_(cls, \*args, \*\*kwargs)    # 第一阶段：解释器从上到下执行代码创建Foo类  # 第二阶段：通过Foo类创建obj对象  obj = Foo() |

以上就是面向对象进阶篇的所有内容，欢迎拍砖