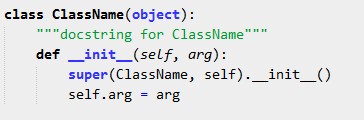
# [Python OOP(1):从基础开始](http://www.cnblogs.com/BeginMan/p/3510786.html)

#### 本文旨在Python复习和总结：

#### 1、如何创建类和实例？



# 创建类

class ClassName(object):

"""docstring for ClassName"""

def \_\_init\_\_(self, arg):

super(ClassName, self).\_\_init\_\_()

self.arg = arg

# 创建实例

instance = ClassName()

#### 2、经典类与新式类的区别？

#### 3、什么是方法？如何定义？如何使用？

方法是类的功能  
定义在类中  
通过实例调用

#### 4、self代表什么？用在哪些地方？

每个类方法都有一个self参数，代表**实例对象本身**，当实例调用方法时，由解释器悄悄地传递给方法，不用手动self进来。  
self在Python里不是关键字。self代表当前对象的地址。self能避免非限定调用造成的全局变量。  
在[wangkangluo1的Python为什么要self](http://www.cnblogs.com/wangkangluo1/archive/2011/09/23/2186479.html)中讲的很清楚：

创建了一个类MyClass，实例化MyClass得到了MyObject这个对象，然后调用这个对象的方法 MyObject.method(arg1,arg2) ，这个过程中，Python会自动转为Myclass.mehod(MyObject,arg1,arg2)  
这就是Python的self的原理了。即使你的类的方法不需要任何参数，但还是得给这个方法定义一个self参数。

#### 5、类对象的两种操作?

定义完类之后就产生了类对象，类对象支持的两种操作：**引用和实例**  
。引用：通过类对象去调用类中的属性或方法；实例：通过类对象实例化一个类对象的实体。

#### 6、什么是Python类属性和实例属性？

**属性就是另一个对象的数据或函数元素！**通过句点符号访问，如一些python类型如复数有数据属性，列表和字典拥有方法(函数属性)。还有可能是，访问一个属性时，这个属性又是对象，其又拥有自己的属性，这样就构成了属性链。如：

>>> import sys

>>> sys.stdout.write('beginman')

beginman

>>> myMoudel.myClass.\_\_doc\_\_

类属性与类相关，与实例无关，通常是类的数据属性，仅仅是类中定义的变量，通常称作静态变量或静态数据。在其他语言中相当于在变量前加上static。

>>> class C(object):

foo = 100 # 定义类属性

>>> print C.foo # 访问类属性

100

>>> C.foo = C.foo+100 # 更新类属性

>>> C.foo

200

由上所知，类属性仅仅与类(类也是对象，python中称类对象)相关，与实例半毛钱关系都没有。

>>> class C(object):

foo = 100 # 定义类属性

>>> print C.foo # 访问类属性

100

>>> C.foo = C.foo+100 # 更新类属性

>>> C.foo

200

>>> c=C() # 实例化一个对象c

>>> c.foo

200

>>> c.foo = 1000 # 实例试图修改类属性

>>> c.foo # 实例想看是否修改成功，于是就c.foo竟输出1000，于是实例就满以为自己成功了

1000

>>> C.foo # 类对象鄙夷的看了实例一样，说：“你就是老子生出来的，老子的东西你也能碰？？”

200

>>> del c.foo # 实例看完之后当场傻眼，心想mlgb,你牛B,我还是除掉自己负担沉重的改造吧

>>> c.foo # 实例除掉了自己负担沉重的改造后，老老实实地调用类对象给的sb玩意

200

>>> # 通过类对象与实例关于类属性争权大战后，我们知道了一点：

>>> #类属性仅仅与类(类也是对象，python中称类对象)相关，与实例半毛钱关系都没有。

>>> #如果类的实例没有同名变量也可以使用实例来访问。如果实例含有与类属性同名的属性，则用该实例访问属性时，访问的是实例中的属性。如果类的实例没有同名变量也可以使用实例来访问。如果实例含有与类属性同名的属性，则用该实例访问属性时，访问的是实例中的属性。

>>>

#### 7、类方法？

>>> class C(object):

foo = 100

def met(self):

print 'i am method for class.'

>>> C.met() # C心想方法也是我的一部分，那么方法也是类属性\*(这点很正确)，于是我调用你玩玩

Traceback (most recent call last):

File "<pyshell#31>", line 1, in <module>

C.met() # C心想方法也是我的一部分，那么方法也是类属性\*(这点很正确)，于是我调用你玩玩

TypeError: unbound method met() must be called with C instance as first argument (got nothing instead)

>>> # 出大事了，方法竟然不同意还反抗了

>>> # 心想这是为什么呢？突然想到了原来是那鸟人Python

>>> # 鸟人 Guido van Rossum 创造Python类给出的规定就是，你可以创类方法，但是不能调戏她。

>>> c=C() # 类心有不甘，于是创了实例小c来助纣为虐

>>> c.met() # 实例小c心想，mlgbd，上次想修改类属性就被你狠狠地鄙视了一番，现在又为难我，唉，命苦。无论如何试试吧

i am method for class.

>>> # 龌龊，类方法竟然听我实例的，于是实例就赶紧找Guido van Rossum问什么情况

>>> # Guido van Rossum就说”为了与OOP惯例保持一致，没有实例就不能调用方法，这种限制就是Python所描述的绑定概念，方法必须绑定在实例中才听话，不能把权利都交给了类，这是老子对你实例的恩惠啊。“

>>> # 实例听完泪牛满面，类对象发怒了，说：”今天，我就要把我身上的所有零件（属性）都掏出来看看，看都有谁不听话！“

>>> dir(C) # 于是类对象使用第一招 dir()内建

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'foo', 'met']

>>> print C.\_\_dict\_\_ # 第二招就是召遣他最听话的属性 \_\_dict\_\_ 来揪出

{'\_\_module\_\_': '\_\_main\_\_', 'met': <function met at 0x0000000002D33EB8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'C' objects>, 'foo': 100, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'C' objects>, '\_\_doc\_\_': None}

>>>

## 类对象出尽风头之后，该轮到实例了：

**实例的创建：**  
C++等编程应用，实例对象是New出来的，Python牛b，就跟人家不一样，偏以函数调用的形式实例化。

class CC(object):

#我是Python类默许的，没重写\_\_init\_\_,所以也没有什么特殊操作

pass

class C(object):

def \_\_init\_\_(self,name,phone,ID):

super(C,self).\_\_init\_\_()

self.name = name

self.phone = phone

self.id = ID

print 'Hi man 我重写了\_\_init\_\_,因为我需要更多的操作'

cc = CC() # 创建CC实例

c = C('BeginMan','110','12306') # 创建C实例

**重点：**：  
当类被调用，实例化的第一步就是创建实例对象，然后Python检查是否实现了\_\_init\_\_()方法，默认情况下如果没有覆盖\_\_init\_\_就不会施加特别的操作。任何特别的操作都需要重写\_\_init\_\_。  
接着就是传递参数，这依赖于你自己定义的\_\_init\_\_，它里面有多少个参数，在实例化的过程中就要传多少个参数，不管是否覆盖了\_\_init\_\_()，实例对象都要作为第一个参数传递进去。

**\_\_init\_\_与\_\_new\_\_，\_\_call\_\_的区别：**

\_\_new\_\_更像真正的构造器，创建对象时调用，返回当前对象的一个实例。但是实际中用的很少。  
\_\_init\_\_:初始化工作，创建对象时调用，对当前对象的实例进行初始化，**无返回值**。在Python中很常用的。  
\_\_call\_\_:让类的实例的行为表现的像函数一样，你可以调用他们，将一个函数当做一个参数传到另外一个函数中等等。很少用。  
优先级：\_\_new\_\_先与\_\_init\_\_

\_\_del\_\_析构器方法，见《py 核心》

#### 实例属性：

可在创建实例后的任意时间创建，也可以在"运行时"创建.\_\_init\_\_()是创建这些属性的关键点。

>>> c.\_\_dict\_\_ # 此时实例c还没属性

{}

>>> c.\_\_class\_\_ # 实例化的类

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

>>> c.foo=1

>>> c.name='CS'

>>> c.\_\_dict\_\_

{'foo': 1, 'name': 'CS'}

>>> dir(c)

[\*\*\*'foo', 'name'\*\*\*,'\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_']

#### python面向对象与其他语言的异同点?

大同小异  
[Differences between “Java OOP” and “Pythonic OOP”?](http://programmers.stackexchange.com/questions/62994/differences-between-java-oop-and-pythonic-oop)  
真tm的拗口，谁英文好能翻译下。

#### 什么是构造器，\_\_init\_\_表示什么? 有什么意义？

参考[面向对象的编程](http://sebug.net/paper/python/ch11s05.html)**[init](http://sebug.net/paper/python/ch11s05.html)**[方法](http://sebug.net/paper/python/ch11s05.html)

#### OOP常用术语

这在软件工程经常见到的，由于Python OOP实际运用的并不是太多，所以对于OOP特征的体现就不那么明显，C++/Java/C# OOP思想体现倒是挺深的，不过本人太菜，难以熟谙OOP的精髓。  
[面向对象程序设计中的常用术语总结](http://nyc1991.blog.51cto.com/6424159/1284314)  
其中对还包括**自省**。

#### 什么是Python自省(反射)？

[推荐阅读](http://www.cnblogs.com/huxi/archive/2011/01/02/1924317.html)

### 下期：Python OOP高级。

## [python super()](http://www.cnblogs.com/lovemo1314/archive/2011/05/03/2035005.html)

一、问题的发现与提出

　　在Python类的方法（method）中，要调用父类的某个方法，在Python 2.2以前，通常的写法如代码段1：

 代码段1：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

class A:

def \_\_init\_\_(self):

print "enter A"

print "leave A"

class B(A):

def \_\_init\_\_(self):

print "enter B"

A.\_\_init\_\_(self)

print "leave B"

>>> b = B()

enter B

enter A

leave A

leave B

[复制代码](javascript:void(0);)

即，使用非绑定的类方法（用类名来引用的方法），并在参数列表中，引入待绑定的对象（self），从而达到调用父类的目的。

　　这样做的缺点是，当一个子类的父类发生变化时（如类B的父类由A变为C时），必须遍历整个类定义，把所有的通过非绑定的方法的类名全部替换过来，例如代码段2，

 代码段2：

class B(C): # A --> C

def \_\_init\_\_(self):

print "enter B"

C.\_\_init\_\_(self) # A --> C

print "leave B"

　　如果代码简单，这样的改动或许还可以接受。但如果代码量庞大，这样的修改可能是灾难性的。

　　因此，自Python 2.2开始，Python添加了一个关键字super，来解决这个问题。下面是Python 2.3的官方文档说明：

 super(type[, object-or-type])

  Return the superclass of type. If the second argument is omitted the super object  
  returned is unbound. If the second argument is an object, isinstance(obj, type)   
  must be true. If the second argument is a type, issubclass(type2, type) must be   
  true. super() only works for new-style classes.

  A typical use for calling a cooperative superclass method is:

   class C(B):  
       def meth(self, arg):  
           super(C, self).meth(arg)

  New in version 2.2.

　　从说明来看，可以把类B改写如代码段3：

 代码段3：

[复制代码](javascript:void(0);)

class A(object): # A must be new-style class

def \_\_init\_\_(self):

print "enter A"

print "leave A"

class B(C): # A --> C

def \_\_init\_\_(self):

print "enter B"

super(B, self).\_\_init\_\_()

print "leave B"

[复制代码](javascript:void(0);)

　　尝试执行上面同样的代码，结果一致，但修改的代码只有一处，把代码的维护量降到最低，是一个不错的用法。因此在我们的开发过程中，super关键字被大量使用，而且一直表现良好。

　　在我们的印象中，对于super(B, self).\_\_init\_\_()是这样理解的：super(B, self)首先找到B的父类（就是类A），然后把类B的对象self转换为类A的对象（通过某种方式，一直没有考究是什么方式，惭愧），然后“被转换”的 类A对象调用自己的\_\_init\_\_函数。考虑到super中只有指明子类的机制，因此，在多继承的类定义中，通常我们保留使用类似代码段1的方法。

　　有一天某同事设计了一个相对复杂的类体系结构（我们先不要管这个类体系设计得是否合理，仅把这个例子作为一个题目来研究就好），代码如代码段4：

 代码段4：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

class A(object):

def \_\_init\_\_(self):

print "enter A"

print "leave A"

class B(object):

def \_\_init\_\_(self):

print "enter B"

print "leave B"

class C(A):

def \_\_init\_\_(self):

print "enter C"

super(C, self).\_\_init\_\_()

print "leave C"

class D(A):

def \_\_init\_\_(self):

print "enter D"

super(D, self).\_\_init\_\_()

print "leave D"

class E(B, C):

def \_\_init\_\_(self):

print "enter E"

B.\_\_init\_\_(self)

C.\_\_init\_\_(self)

print "leave E"

class F(E, D):

def \_\_init\_\_(self):

print "enter F"

E.\_\_init\_\_(self)

D.\_\_init\_\_(self)

print "leave F"

[复制代码](javascript:void(0);)

  f = F() result:

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

enter F

enter E

enter B

leave B

enter C

enter D

enter A

leave A

leave D

leave C

leave E

enter D

enter A

leave A

leave D

leave F

[复制代码](javascript:void(0);)

　　明显地，类A和类D的初始化函数被重复调用了2次，这并不是我们所期望的结果！我们所期望的结果是最多只有类A的初始化函数被调用2次——其实这是多继承的类体系必须面对的问题。我们把代码段4的类体系画出来，如下图：

    object  
   |       \  
   |        A  
   |      / |  
   B  C  D  
    \   /   |  
      E    |  
        \   |  
          F

　　按我们对super的理解，从图中可以看出，在调用类C的初始化函数时，应该是调用类A的初始化函数，但事实上却调用了类D的初始化函数。好一个诡异的问题！

　　也就是说，mro中记录了一个类的所有基类的类类型序列。查看mro的记录，发觉包含7个元素，7个类名分别为：

 F E B C D A object

　　从而说明了为什么在C.\_\_init\_\_中使用super(C, self).\_\_init\_\_()会调用类D的初始化函数了。 ???

　　我们把代码段4改写为：

 代码段9：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

class A(object):

def \_\_init\_\_(self):

print "enter A"

super(A, self).\_\_init\_\_() # new

print "leave A"

class B(object):

def \_\_init\_\_(self):

print "enter B"

super(B, self).\_\_init\_\_() # new

print "leave B"

class C(A):

def \_\_init\_\_(self):

print "enter C"

super(C, self).\_\_init\_\_()

print "leave C"

class D(A):

def \_\_init\_\_(self):

print "enter D"

super(D, self).\_\_init\_\_()

print "leave D"

class E(B, C):

def \_\_init\_\_(self):

print "enter E"

super(E, self).\_\_init\_\_() # change

print "leave E"

class F(E, D):

def \_\_init\_\_(self):

print "enter F"

super(F, self).\_\_init\_\_() # change

print "leave F"

[复制代码](javascript:void(0);)

f = F() result:

[复制代码](javascript:void(0);)

enter F

enter E

enter B

enter C

enter D

enter A

leave A

leave D

leave C

leave B

leave E

leave F

[复制代码](javascript:void(0);)

　　明显地，F的初始化不仅完成了所有的父类的调用，而且保证了每一个父类的初始化函数只调用一次。

　　再看类结构：

[复制代码](javascript:void(0);)

object

/ \

/ A

| / \

B-1 C-2 D-2

\ / /

E-1 /

\ /

F

[复制代码](javascript:void(0);)

E-1,D-2是F的父类，其中表示E类在前，即F（E，D）。

所以初始化顺序可以从类结构图来看出 ： F－>E->B -->C --> D --> A

由于C，D有同一个父类，因此会先初始化D再是A。

三、延续的讨论

　　我们再重新看上面的类体系图，如果把每一个类看作图的一个节点，每一个从子类到父类的直接继承关系看作一条有向边，那么该体系图将变为一个有向图。不能发现mro的顺序正好是该有向图的一个拓扑排序序列。

　　从而，我们得到了另一个结果——Python是如何去处理多继承。支持多继承的传统的面向对象程序语言（如C++）是通过虚拟继承的方式去实现多继承中父类的构造函数被多次调用的问题，而Python则通过mro的方式去处理。

　　但这给我们一个难题：对于提供类体系的编写者来说，他不知道使用者会怎么使用他的类体系，也就是说，不正确的后续类，可能会导致原有类体系的错误，而且这样的错误非常隐蔽的，也难于发现。

四、小结

　　1. super并不是一个函数，是一个类名，形如super(B, self)事实上调用了super类的初始化函数，  
       产生了一个super对象；  
　　2. super类的初始化函数并没有做什么特殊的操作，只是简单记录了类类型和具体实例；  
　　3. super(B, self).func的调用并不是用于调用当前类的父类的func函数；  
　　4. Python的多继承类是通过mro的方式来保证各个父类的函数被逐一调用，而且保证每个父类函数  
       只调用一次（如果每个类都使用super）；  
　　5. 混用super类和非绑定的函数是一个危险行为，这可能导致应该调用的父类函数没有调用或者一  
       个父类函数被调用多次。

[返回首页](http://blog.beginman.cn/)

### Python OOP 常见疑难点汇总(1).基础

BeginMan 发布于 4个月前    持续热度：145℃   类别：[python](http://blog.beginman.cn/blog/type/4/)   标签：[python](http://blog.beginman.cn/blog/tag/4/) [oop](http://blog.beginman.cn/blog/tag/26/)

Python面向对象常见疑难点汇总



### No1.公有私有属性

python oop无public 和private,对类的私有属性的处理通过\_\_

1. class P():
2. a = 100 # public
3. \_\_a = 200 # private
4. def func(self):
5. print 'Public function'
6. print self.a
7. print self.\_\_a
8. def \_\_pfunc(self):
9. print 'Private function'
10. print self.a
11. print self.\_\_a
12. p = P()
13. print p.a
14. print p.\_\_a #err
15. print p.func()
16. print p.\_\_pfunc() #err

### NO2.self的哲学

**self非关键字**，而是一个参数对象的代表，像this一样，代表实例对象本身。调用实例方法，会隐式传实递实例对象。

1. >>> class P():
2. ... def func(self):
3. ... print self
4. ...
5. ...
6. ...
7. >>> p = P() # 实例化
8. >>> p.func() # 调用实例方法，此时隐式传实递实例p对象
9. <\_\_main\_\_.P instance at 0xb69282cc>

### No3.新式类VS经典类

新式类：就是所有类都必须继承的类，如果什么都不想继承，就继承到object类。 经典类：没有父类

#### 1.递归问题

<http://zxn990.blog.51cto.com/4253193/1285599>

#### 2.调用父类构造器

<http://jianshu.io/p/2ZxRsn>

下面来自<http://stackoverflow.com/questions/54867/old-style-and-new-style-classes-in-python>

Up to Python 2.1, old-style classes were the only flavour available to the user. The concept of (old-style) class is unrelated(无关的) to the concept of type: if x is an instance of an old-style class, then x.**class** designates the class of x, but type(x) is always <type 'instance'>. This reflects the fact that all old-style instances, independently of their class, are implemented with a single built-in type, called instance.

New-style classes were introduced in Python 2.2 to unify classes and types. A new-style class neither more nor less than a user-defined type. If x is an instance of a new-style class, then type(x) is the same as x.**class**.

The major motivation for introducing new-style classes is to provide a unified object model with a full meta-model. It also has a number of immediate benefits, like the ability to subclass(亚纲) most built-in types, or the introduction of "descriptors(描述符号)", which enable computed properties.

For compatibility reasons, classes are still old-style by default. New-style classes are created by specifying another new-style class (i.e. a type) as a parent class, or the "top-level type" object if no other parent is needed. The behaviour of new-style classes differs from that of old-style classes in a number of important details in addition to what type returns. Some of these changes are fundamental to the new object model, like the way special methods are invoked. Others are "fixes" that could not be implemented before for compatibility concerns, like the method resolution order in case of multiple inheritance.

Python 3 only has new-style classes. No matter if you subclass from object or not, classes are new-style in Python 3. It is however recommended that you still subclass from object.

1. >>> p = P2() # 经典类
2. >>> p.\_\_class\_\_
3. <class \_\_main\_\_.P2 at 0xb68bd1dc>
4. >>> type(p)
5. <type 'instance'>
6. >>> pp = P() # 新式类
7. >>> pp.\_\_class\_\_
8. <class '\_\_main\_\_.P'>
9. >>> type(pp)
10. <class '\_\_main\_\_.P'>

### No4.类方法VS实例方法VS静态方法

类方法：classmethod修饰，第一个参数cls,类与实例共享。

实例方法：无修饰，第一个参数self,实例所有

静态方法：staticmethod修饰，无需参数，类与实例共享。无法访问类属性、实例属性，相当于一个相对独立的方法，跟类其实没什么关系，换个角度来讲，其实就是放在一个类的作用域里的函数而已。静态方法中引用类属性的话，必须通过类对象来引用。

1. class P(object):
2. tip = 'Persion tips' # 类属性，类及类成员共享
3. @classmethod
4. def clsFunc(cls):
5. """类方法共享"""
6. return 'Welcome:' + cls.tip
7. def insFunc(self):
8. """实例方法"""
9. return self.\_\_class\_\_
10. @staticmethod
11. def staFunc():
12. """静态方法"""
13. return P.tip
14. p = P()
15. print P.tip
16. print p.tip
17. print p.insFunc()
18. p.age = 20 # 实例属性无需在类中显示定义
19. print p.age
20. print p.clsFunc() # 类方法
21. print p.clsFunc()
22. print p.staFunc() # 静态方法实例与类共享
23. print p.staFunc()

### NO5.****init**** VS ****new****

首先看这段代码，代码参考[Python's use of](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init) **[new](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init)** [and](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init) **[init](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init)**[?](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init)

1. class P(object):
2. \_dict = dict()
3. def \_\_new\_\_(cls):
4. if 'key' in P.\_dict:
5. print "EXISTS"
6. #return P.\_dict['key'] # 返回实例对象
7. return P.\_dict # 返回不合法的对象
8. else:
9. print 'NEW'
10. return super(P, cls).\_\_new\_\_(cls) # 返回实例对象
11. def \_\_init\_\_(self):
12. print 'INIT..'
13. P.\_dict['key'] = self
14. print " "
15. p1 = P()
16. p2 = P()
17. p3 = P()
18. # 输出结果
19. NEW
20. INIT..
21. EXISTS
22. EXISTS
23. # 如果把上面的注释取消，变成：return P.\_dict['key'] 则输出结果如下：
24. NEW
25. INIT..
26. EXISTS
27. INIT..
28. EXISTS
29. INIT..

看出点端倪了么。。

\_\_new\_\_()：静态方法，cls作为第一个参数，控制创造一个新实例

\_\_init\_\_()：实例方法，self作为第一个参数，控制初始化一个实例

在Python OOP中，\_\_new\_\_()相比\_\_init\_\_()更像一个构造器，**\_\_new\_\_()是一个静态方法，必须返回一个合法的实例，这样解释器在调用init()之前把这个实例作为self传递过去**

\_\_init\_\_()实例初始化，它返回None,它的工作就是在实例被创建(**new**())后初始化它。

1. >>> class P(object):
2. ... def \_\_init\_\_(self):
3. ... print 'ok'
4. ... def \_\_new\_\_(cls):
5. ... print 'new'
6. ...
7. ...
8. ...
9. >>> p = P()
10. new
11. >>> class P(object):
12. ... def \_\_init\_\_(self):
13. ... print 'ok'
14. ...
15. ... def \_\_new\_\_(cls):
16. ... print 'new'
17. ... return super(P, cls).\_\_new\_\_(cls)
18. ...
19. ...
20. >>> p = P()
21. new
22. ok

### NO6.特殊的实例属性VS特殊的类属性

dir()相关知识移步这里<http://sebug.net/paper/python/ch08s06.html>.

特殊实例属性：\_\_dict\_\_,\_\_init\_\_,\_\_class\_\_,如：

1. >>> p.\_\_class\_\_
2. <class \_\_main\_\_.P at 0xb692d05c>
3. >>> p.\_\_doc\_\_
4. >>> p.\_\_init\_\_
5. <bound method P.\_\_init\_\_ of <\_\_main\_\_.P instance at 0xb6936bac>>
6. >>> p.\_\_module\_\_
7. '\_\_main\_\_'
8. >>> p.\_\_new\_\_
9. <bound method P.\_\_new\_\_ of <\_\_main\_\_.P instance at 0xb6936bac>>
10. >>> p.\_\_dict\_\_
11. {}
12. >>> p.age = 100
13. >>> p.\_\_dict\_\_
14. {'age': 100}
15. >>> dir(p)
16. ['\_\_doc\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', 'age']

特殊类属性：C.\_\_name\_\_:类名， C.\_\_bases\_\_ or(\_\_base\_\_)：父类，C.\_\_dict\_\_:类属性， C.\_\_module\_\_:定义类所在的model, C.\_\_class\_\_：实例C对应的类。

一般这些特殊的东西用的较少，这里展示下且当记忆。

### NO7.理解super()

super主要为了不显示的引用基类的东西，主要的优势体现在多重继承上如 class (A,B)等，使用super能够智能的帮助我们调用基类。

1. super并不是一个函数，是一个类名，形如super(B, self)事实上调用了super类的初始化函数，产生了一个super对象；
2. super类的初始化函数并没有做什么特殊的操作，只是简单记录了类类型和具体实例；
3. super(B, self).func的调用并不是用于调用当前类的父类的func函数；
4. Python的多继承类是通过mro的方式来保证各个父类的函数被逐一调用，而且保证每个父类函数只调用一次（如果每个类都使用super）；
5. 混用super类和非绑定的函数是一个危险行为，这可能导致应该调用的父类函数没有调用或者一个父类函数被调用多次。

关于Python super的解释参考这里**[python super()](http://www.cnblogs.com/lovemo1314/archive/2011/05/03/2035005.html)**.

这里有个误区就是:

1. super( BasicElement, self ).\_\_init\_\_()
2. super( BasicElement, self ).\_\_init\_\_(self)

实例如下：

1. >>> class A(object):
2. ... def \_\_init\_\_(self):
3. ... print 'AAAAAA'
4. ...
5. ...
6. ...
7. >>> class B(A):
8. ... def \_\_init\_\_(self):
9. ... super(B, self).\_\_init\_\_()
10. ...
11. ...
12. ...
13. >>> b = B()
14. AAAAAA
15. >>> class B(A):
16. ... def \_\_init\_\_(self):
17. ... super(B, self).\_\_init\_\_(self)
18. ...
19. ...
20. ...
21. >>> b = B()
22. Traceback (most recent call last):
23. File "<input>", line 1, in <module>
24. File "<input>", line 3, in \_\_init\_\_
25. TypeError: \_\_init\_\_() takes exactly 1 argument (2 given)
26. >>> class A(object):
27. ... def \_\_init\_\_(self, args):
28. ... print 'AAAAAAAAAAA', args
29. ...
30. ...
31. ...
32. >>> class B(A):
33. ... def \_\_init\_\_(self):
34. ... super(B, self).\_\_init\_\_('Man')
35. ...
36. ...
37. ...
38. >>> b = B()
39. AAAAAAAAAAA Man

**看出super(B, self).\_\_init\_\_("sss") 向基类传参。在Python3中，可以直接写super().init() 取代 super(B, self).init()**

作者：BeginMan

网址: <http://blog.beginman.cn/blog/46/>

在未标明转载的情况下，本文版权归作者，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

[上一篇：](http://blog.beginman.cn/blog/45/) [下一篇：](http://blog.beginman.cn/blog/47/)

最新最早最热

* [评论](javascript:void(0);)
* 还没有评论，沙发等你来抢

社交帐号登录:

* [微博](http://beginman.duoshuo.com/login/weibo/)
* [QQ](http://beginman.duoshuo.com/login/qq/)
* [人人](http://beginman.duoshuo.com/login/renren/)
* [豆瓣](http://beginman.duoshuo.com/login/douban/)
* [更多»](javascript:void(0))

[Kui](javascript:void(0);)

Top of Form



Bottom of Form

[BeginMan正在使用多说](http://duoshuo.com)