# [python 面向对象（进阶篇）](https://www.cnblogs.com/wupeiqi/p/4766801.html)

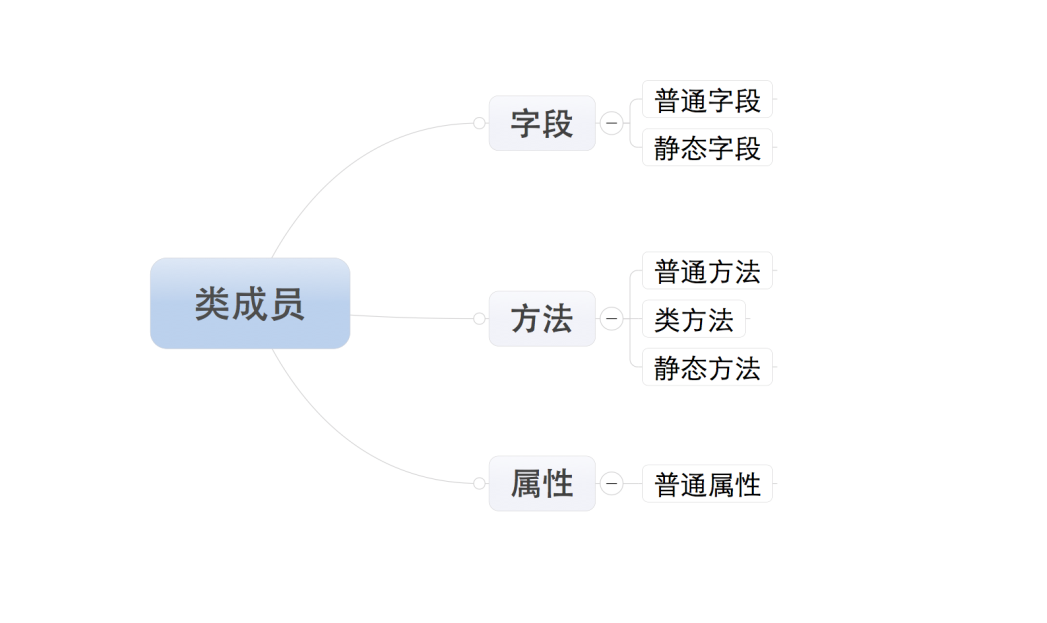
上一篇[《Python 面向对象（初级篇）》](http://www.cnblogs.com/wupeiqi/p/4493506.html" \t "_blank)文章介绍了面向对象基本知识：

* 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



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**

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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：



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；

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

[复制代码](javascript:void(0);)

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()

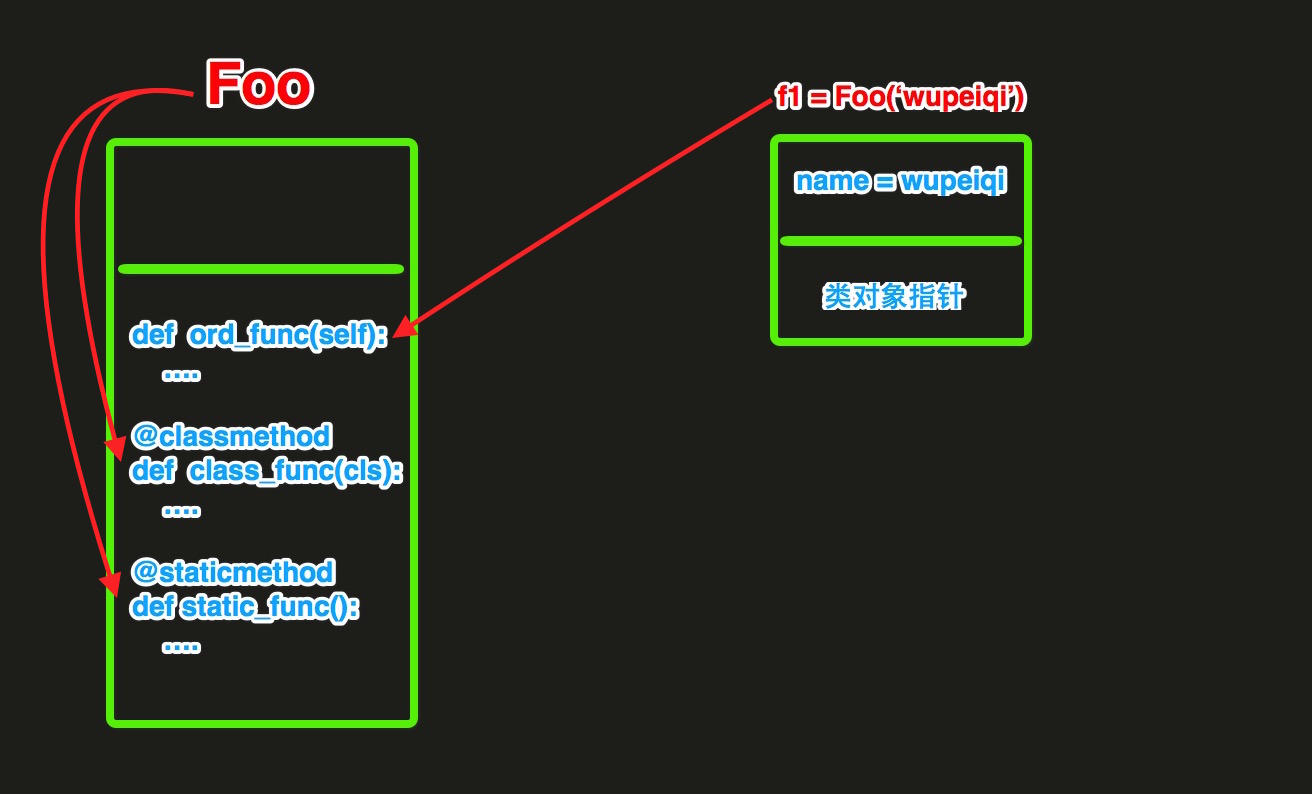
# 调用类方法

Foo.class\_func()

# 调用静态方法

Foo.static\_func()

[复制代码](javascript:void(0);)

****

**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**

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

[复制代码](javascript:void(0);)

# ############### 定义 ###############

class Foo:

def func(self):

pass

# 定义属性

@property

def prop(self):

pass

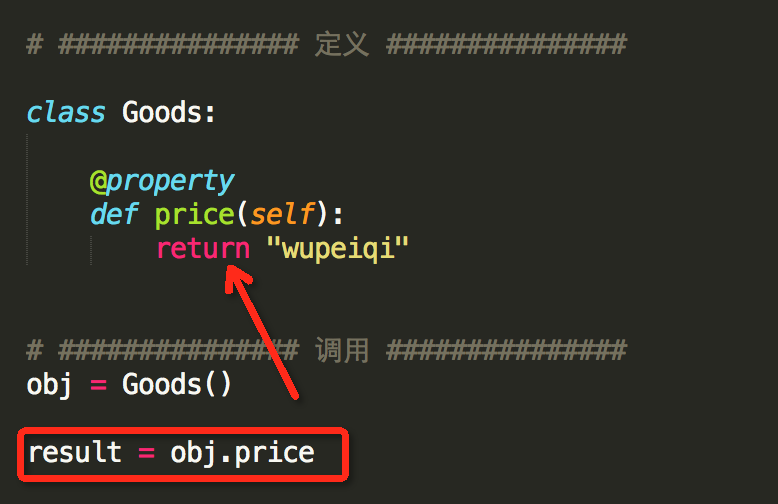
# ############### 调用 ###############

foo\_obj = Foo()

foo\_obj.func()

foo\_obj.prop #调用属性

[复制代码](javascript:void(0);)



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

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

[复制代码](javascript:void(0);)

# ############### 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

[复制代码](javascript:void(0);)

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）

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

[复制代码](javascript:void(0);)

# ############### 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

[复制代码](javascript:void(0);)

**新式类**，具有三种@property装饰器

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

[复制代码](javascript:void(0);)

# ############### 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

[复制代码](javascript:void(0);)

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

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

[复制代码](javascript:void(0);)

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.deleter

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

[复制代码](javascript:void(0);)

**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

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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

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

[复制代码](javascript:void(0);)

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.BAR.\_\_doc\_\_ # 自动获取第四个参数中设置的值：description...

[复制代码](javascript:void(0);)

 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

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

 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

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

[复制代码](javascript:void(0);)

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)

[复制代码](javascript:void(0);)

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；

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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。

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

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

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

**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

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

[复制代码](javascript:void(0);)

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

class C:

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

self.name = 'wupeiqi'

[复制代码](javascript:void(0);)

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

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。

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

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

**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。

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

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()()

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

[复制代码](javascript:void(0);)

class Foo:

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

pass

def \_\_call\_\_(self, \*args, \*\*kwargs):

print '\_\_call\_\_'

obj = Foo() # 执行 \_\_init\_\_

obj() # 执行 \_\_call\_\_

[复制代码](javascript:void(0);)

**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：



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

[复制代码](javascript:void(0);)

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'}

[复制代码](javascript:void(0);)

**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。

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

[复制代码](javascript:void(0);)

class Foo:

def \_\_str\_\_(self):

return 'wupeiqi'

obj = Foo()

print obj

# 输出：wupeiqi

[复制代码](javascript:void(0);)

**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\_\_

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

[复制代码](javascript:void(0);)

class Foo(object):

pass

obj = Foo()

for i in obj:

print i

# 报错：TypeError: 'Foo' object is not iterable

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

#!/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'

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

#!/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

[复制代码](javascript:void(0);)

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 |

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

[复制代码](javascript:void(0);)

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

obj = iter([11,22,33,44])

while True:

val = obj.next()

print val

[复制代码](javascript:void(0);)

**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。



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

[复制代码](javascript:void(0);)

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()

[复制代码](javascript:void(0);)

以上就是面向对象进阶篇的所有内容，欢迎拍砖...