Python 面向对象（进阶篇）

2017-08-12 Python开发者

（点击上方蓝字，快速关注我们）

> 来源：Mr.Seven
> www.cnblogs.com/wupeiqi/p/4766801.html
> 如有好文章投稿，请点击 → 这里了解详情

上一篇《Python 面向对象（初级篇）》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。

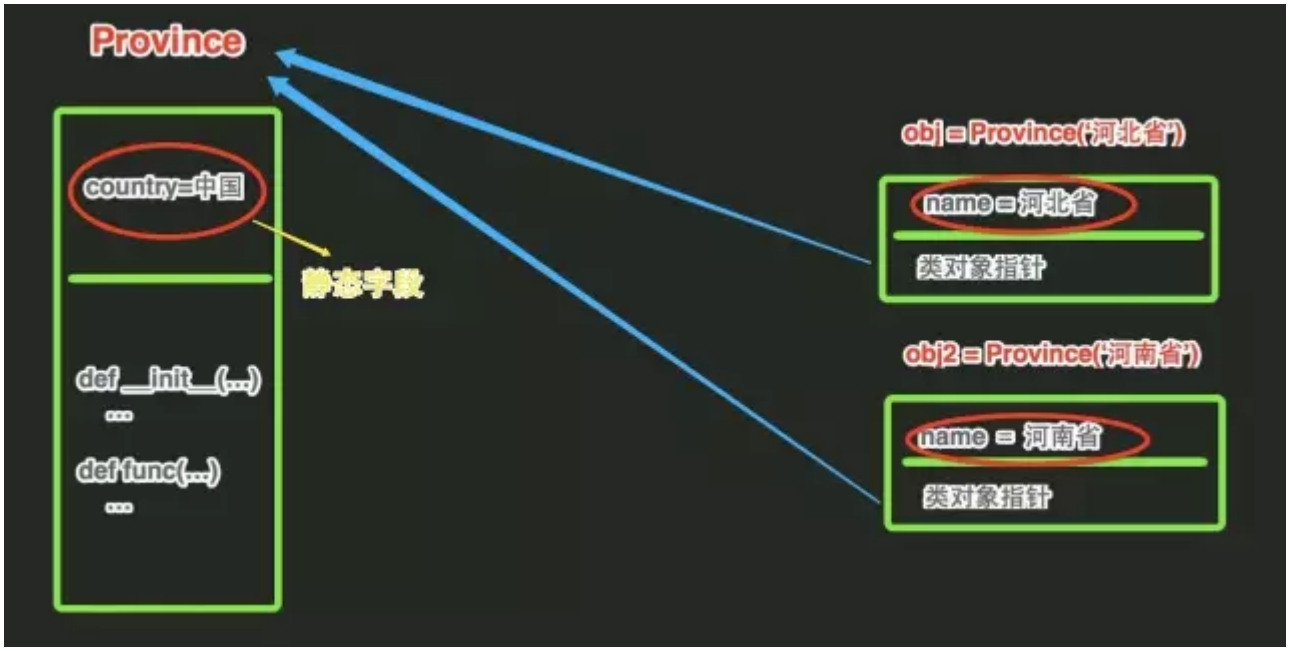
一、字段

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

```
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：



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

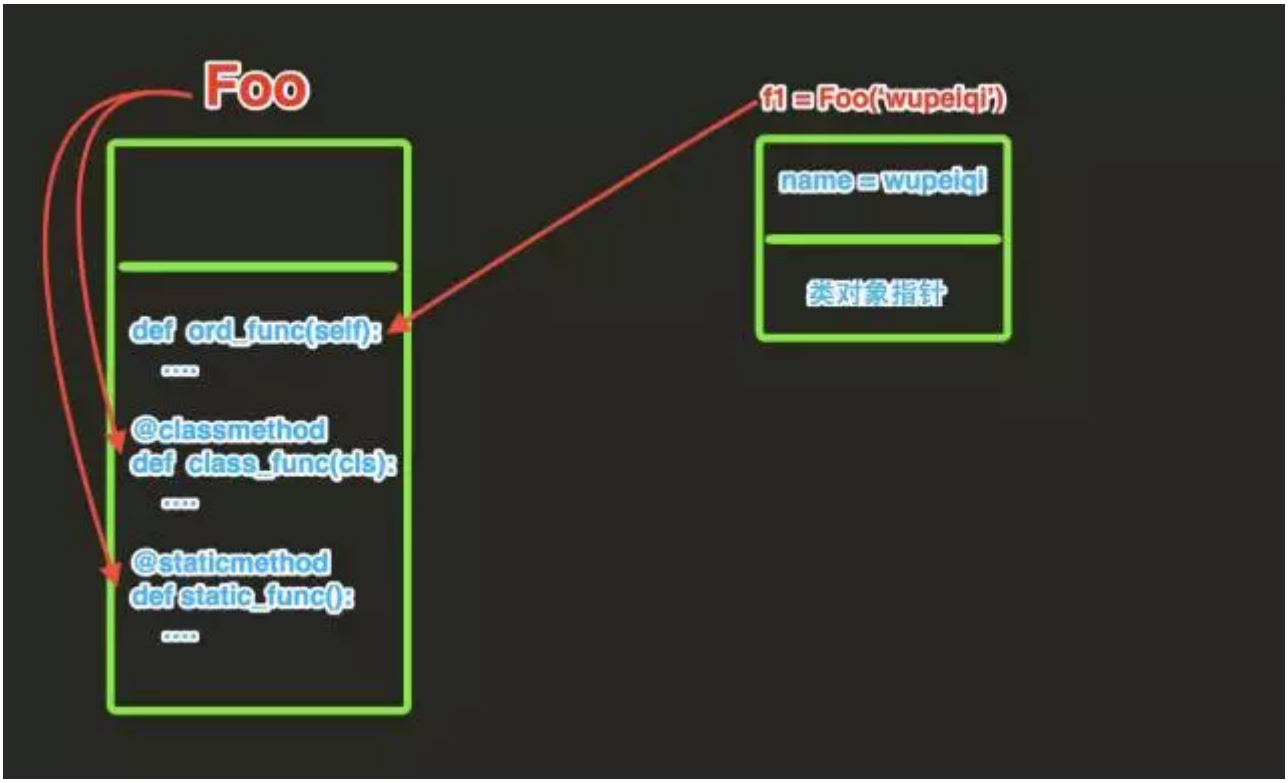
二、方法

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；

```
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()
```



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

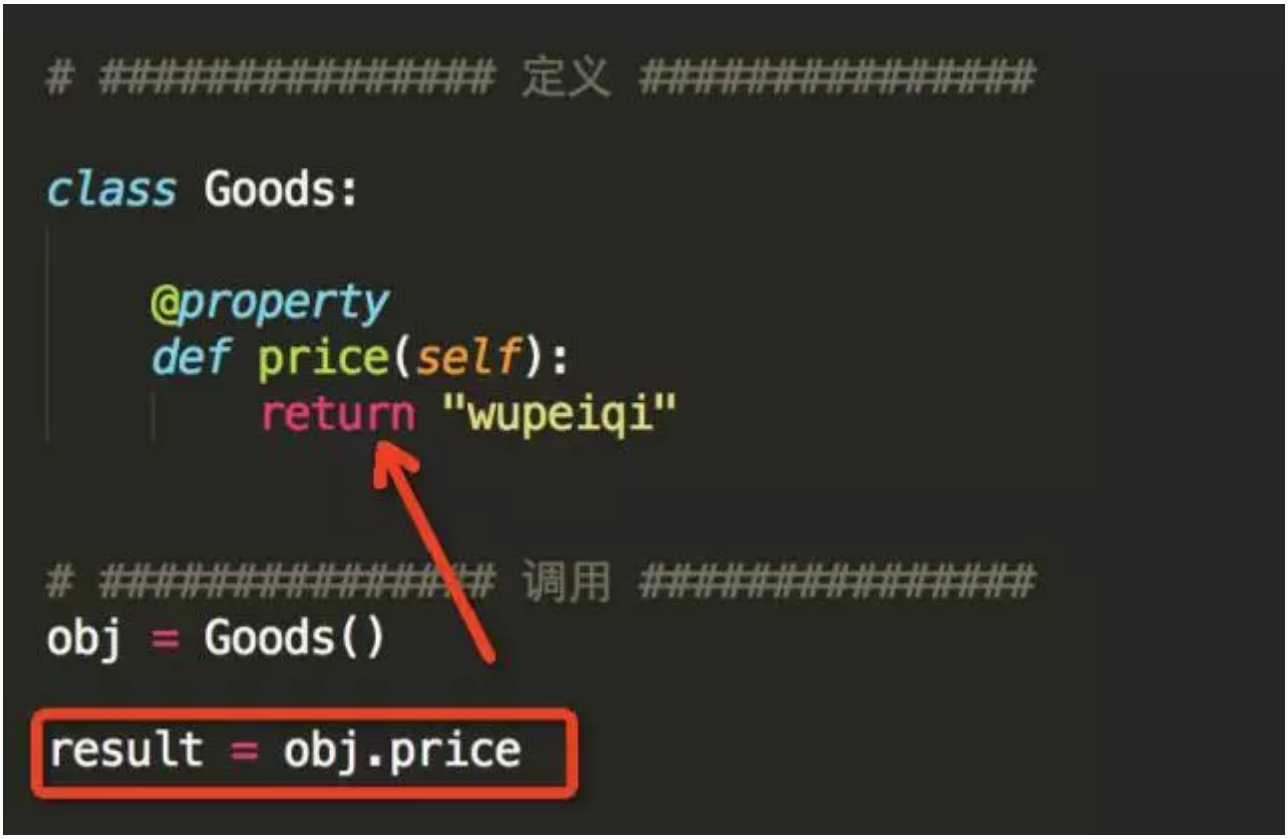
如果你已经了解Python类中的方法，那么属性就非常简单了，因为Python中的属性其实是普通方法的变种。

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

```
# ################ 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
```



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

```
# ############### 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）

```
# ################ 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

```
# ################ 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

```
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

```
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

```
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

```
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

```
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）

```
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；

```
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
```

```
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。

```
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
```

```
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

```
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

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

class C:
```

```
    def __init__(self):
        self.name = 'wupeiqi'


lib/aa.py
```

```
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。

```
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。

```
class Foo:


    def __del__(self):
        pass
```

5. __call__

对象后面加括号，触发执行。

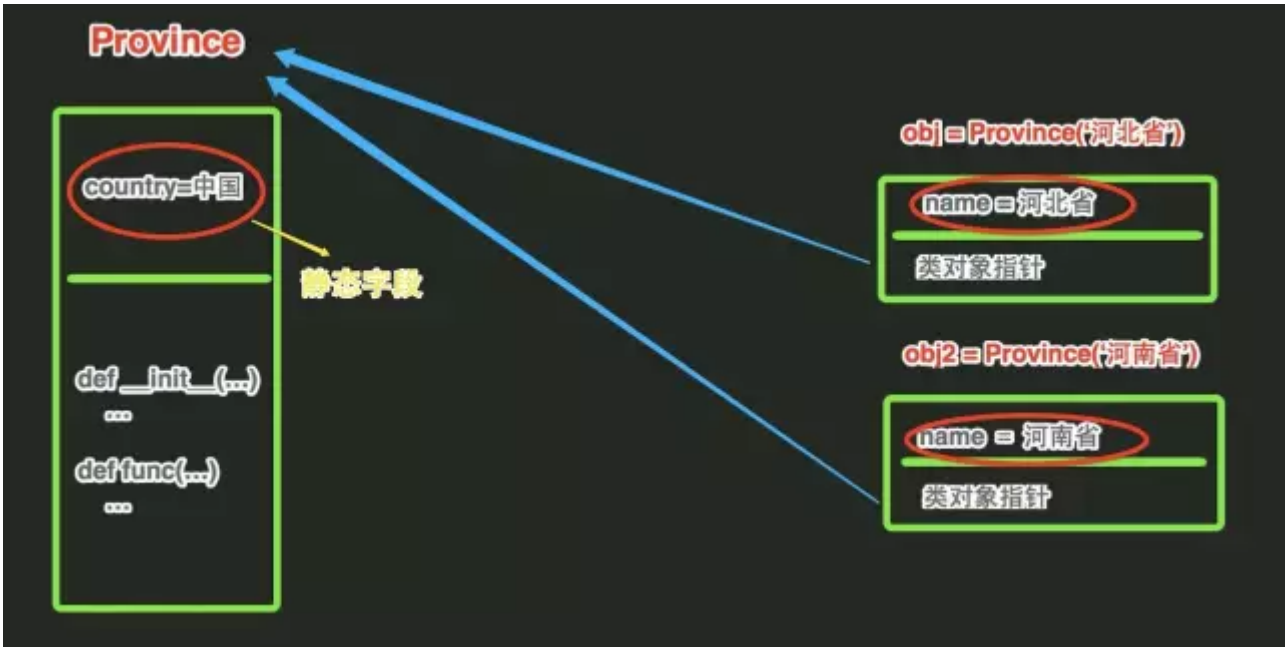
注：构造方法的执行是由创建对象触发的，即：对象 = 类名() ；而对于 __call__ 方法的执行是由对象后加括号触发的，即：对象() 或者 类()()

```
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：




```
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。

```
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

```
#!/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

```
#!/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__

```
class Foo(object):
    pass


obj = Foo()

for i in obj:
    print i

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

```
#!/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'
```

```
#!/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：

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

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

for i in obj:
    print i
```

```
#!/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：

```
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。

```
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

```
class Foo(object):


    def func(self):
        print 'hello wupeiqi'
```

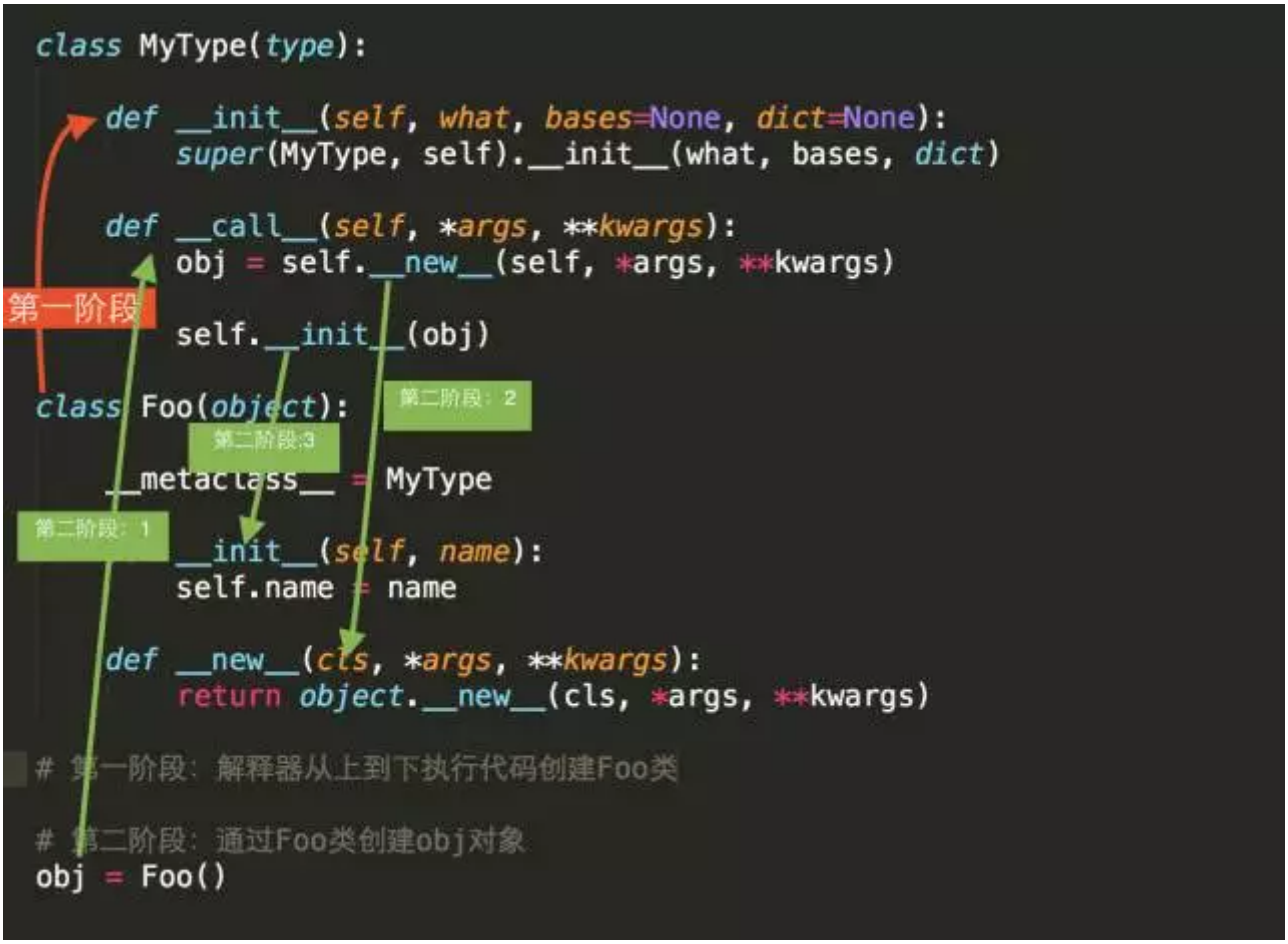
b).特殊方式（**type**类的构造函数）

```
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。




```
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…

阅读原文