# Python面向对象编程

* 面向对象是一种编程方式，此编程方式的实现是基于类和对象的使用。
* 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，实例用于调用被包装在类中的函数。
* 面向对象的三大特性：封装、多态和继承。

## 类的成员

类成员可以分为三大类：字段、方法和属性



注意：多个成员中，**只有普通字段的内容保存在对象中**，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。

### 字段

字段包括：普通字段和静态字段，本质的区别是在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（类属性）

class **Province**:

#静态字段

country = *"中国"*

def **\_\_init\_\_**(*self*, name):

#普通字段

*self*.name = name

obj = Province(*'山东省'*)

#直接访问普通字段

print obj.name #山东省

print obj.country #中国

#直接访问静态字段

print Province.country #中国

print Province.name

AttributeError: class Province has no attribute 'name'

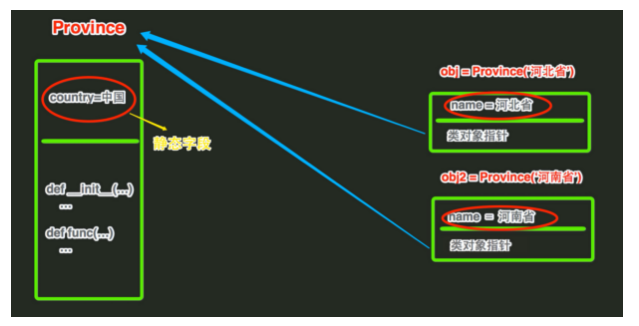
Province.country = *"韩国"*

print obj.country #韩国

obj.country = *"日本"*

print Province.country #韩国

print obj.country #日本



* **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。**

### 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赋值给cls。
* 静态方法：由类调用；无默认参数；

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

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 *"普通方法"*

*@classmethod*

def **claas\_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"*

f = Foo(*'func'*)

#调用普通方法

f.ord\_func() #普通方法

f.claas\_func() #类方法

f.static\_func() #静态方法

#Foo.ord\_func() #TypeError: unbound method ord\_func() must be called with Foo instance as first argument

Foo.ord\_func(f) #普通方法

#调用类方法

Foo.claas\_func() #类方法

#调用静态方法

Foo.static\_func() #静态方法

**使用staticmethod和classmethod内建函数：**

**class Aoo:**

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 *"普通方法"*

def **claas\_func**(cls):

*"""*

*定义类方法，至少有一个cls参数*

*"""*

print *'类方法'*

def **static\_func**():

*"""*

*定义静态方法，无默认参数*

*"""*

print *"静态方法"*

**claas\_func = classmethod(claas\_func) # 类方法**

**static\_func = staticmethod(static\_func) # 静态方法**

a = Aoo(*'func'*)

#调用普通方法

a.ord\_func() #普通方法

a.claas\_func() #类方法

a.static\_func() #静态方法

#Foo.ord\_func() #TypeError: unbound method ord\_func() must be called with Foo instance as first argument

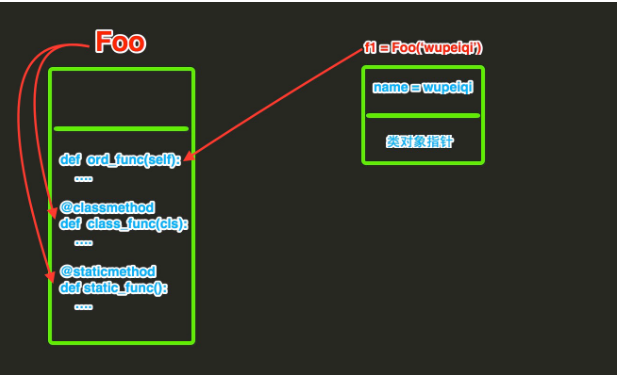
Aoo.ord\_func(a) #普通方法

#调用类方法

Aoo.claas\_func() #类方法

#调用静态方法

Aoo.static\_func() #静态方法



相同点：对于所有方法而言，均属于类中（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。

### 属性

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

p = Pager(1)

print p.start #0

p.start() #error

print p.end #10

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。

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。

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

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

静态字段方式，创建值为property对象的静态字段:

* 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...

### 类属性和实例属性

dir()返回的是对象的属性的一个名字列表，而\_\_dict\_\_返回的是一个字典，它的键是属性名，键值是相应的属性对象的数据值。



类属性是与类相关的数据值，和实例属性不同，类属性和实例无关。

类和实例都是名字空间。类是类属性的名字空间，实例则是实例属性的。

任何对实例属性的赋值都会创建一个实例属性（如果不存在的话）并且对其赋值。所以，给一个与类属性同名的实例属性赋值，会有效的“隐藏”类属性，一旦这个实例属性被删除，类属性又重见天日。

>>> class Foo:

x = 15

>>> foo = Foo()

>>> foo.x

15

>>> foo.x = 17

>>> foo.x

17

>>> foo.\_\_dict\_\_

{'x': 17}

>>>

>>> Foo.x

15

>>> Foo.x = 16

>>> foo.x

17

>>> del foo.x

>>> foo.x

16

>>>

**在类属性可变的情况下**：

>>> Foo.\_\_dict\_\_

{'x': 16, '\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

>>> dir(Foo)

['\_\_doc\_\_', '\_\_module\_\_', 'x']

>>>

>>> class Foo:

x = {2003:'poe2'}

>>> foo = Foo()

>>> foo.x

{2003: 'poe2'}

>>> foo.x[2004] = 'valid path'

>>> foo.x

{2003: 'poe2', 2004: 'valid path'}

>>> Foo.x

{2003: 'poe2', 2004: 'valid path'}

>>> del foo.x # 类属性没有被遮蔽，所以删除不了

Traceback (most recent call last):

File "<pyshell#24>", line 1, in <module>

del foo.x

AttributeError: Foo instance has no attribute 'x'

>>> dir(foo)

['\_\_doc\_\_', '\_\_module\_\_', 'x']

>>>

>>> foo.\_\_dict\_\_

{}

>>>

### 绑定方法和调用

方法仅仅是类内部定义的函数（这意味着**方法是类属性而不是实例属性**）。方法只有在其所属的类拥有实例时，才能被调用。当存在一个实例时，方法才被认为是绑定到那个实例了。没有实例时，方法就是未绑定的。

**self是什么？**

答：self变量用于在类实例方法中引用方法所绑定的实例。

## 多重继承

### 方法解释顺序（MRO）

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

class **P1**: # (object)

def **foo**(*self*):

print *"called P1-foo()"*

class **P2**: # (object)

def **foo**(*self*):

print *"called P2-called"*

def **bar**(*self*):

print *"called P2-bar()"*

class **C1**(P1, P2):

pass

class **C2**(P1, P2):

def **bar**(*self*):

print *'called C2-bar()'*

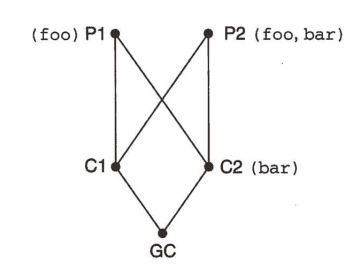
class **GC**(C1, C2):

pass

gc = GC()

gc.foo() # called P1-foo()

gc.bar() # called P2-bar()



1. 经典类

深度优先，从左至右

1. 新式类

广度优先

gc = GC()

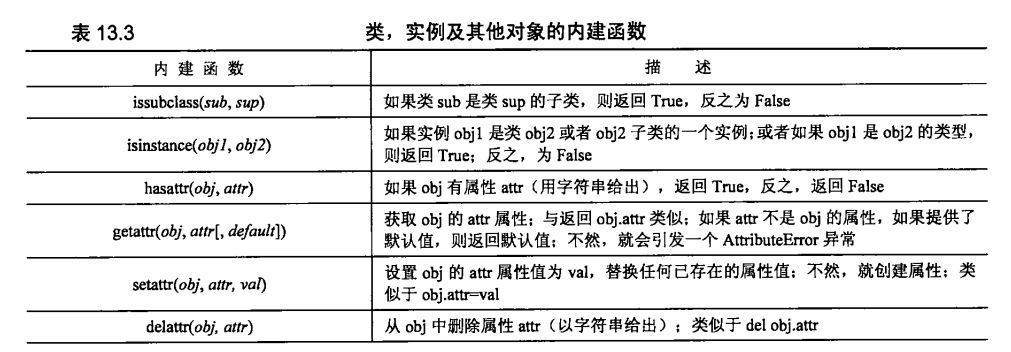
gc.foo() # called P1-foo()

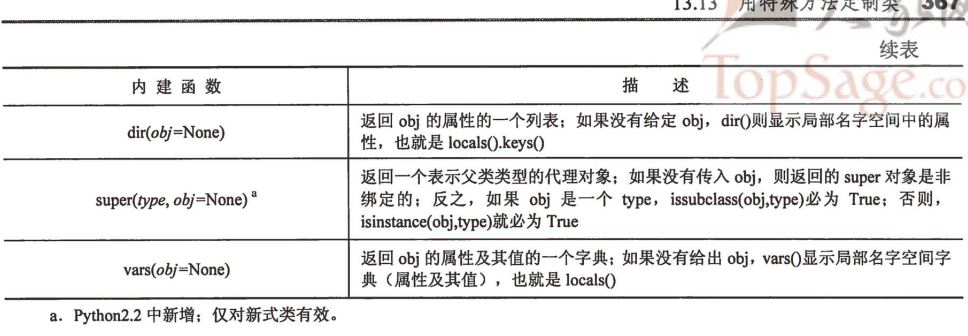
gc.bar() # called C2-bar()

print GC.\_\_mro\_\_ # (<class '\_\_main\_\_.GC'>, <class '\_\_main\_\_.C1'>, <class '\_\_main\_\_.C2'>, <class '\_\_main\_\_.P1'>, <class '\_\_main\_\_.P2'>, <type 'object'>)

新式类也有一个\_\_mro\_\_属性，告诉你查找的顺序是怎样的。

## 类、实例和其他对象的内建函数





## 私有化

### 双下划线

由双下划线开始的属性在运行时被“混淆”，所以直接访问时不允许的。实际上，会在名字前面加上下划线和类名。

### 单下划线

简单的模块级私有化，防止模块的属性用 “from mymoudle import \*”来加载。