续面向对象－继承之下对象继续聊

2017-01-16　哈哈小　菜猿观世界

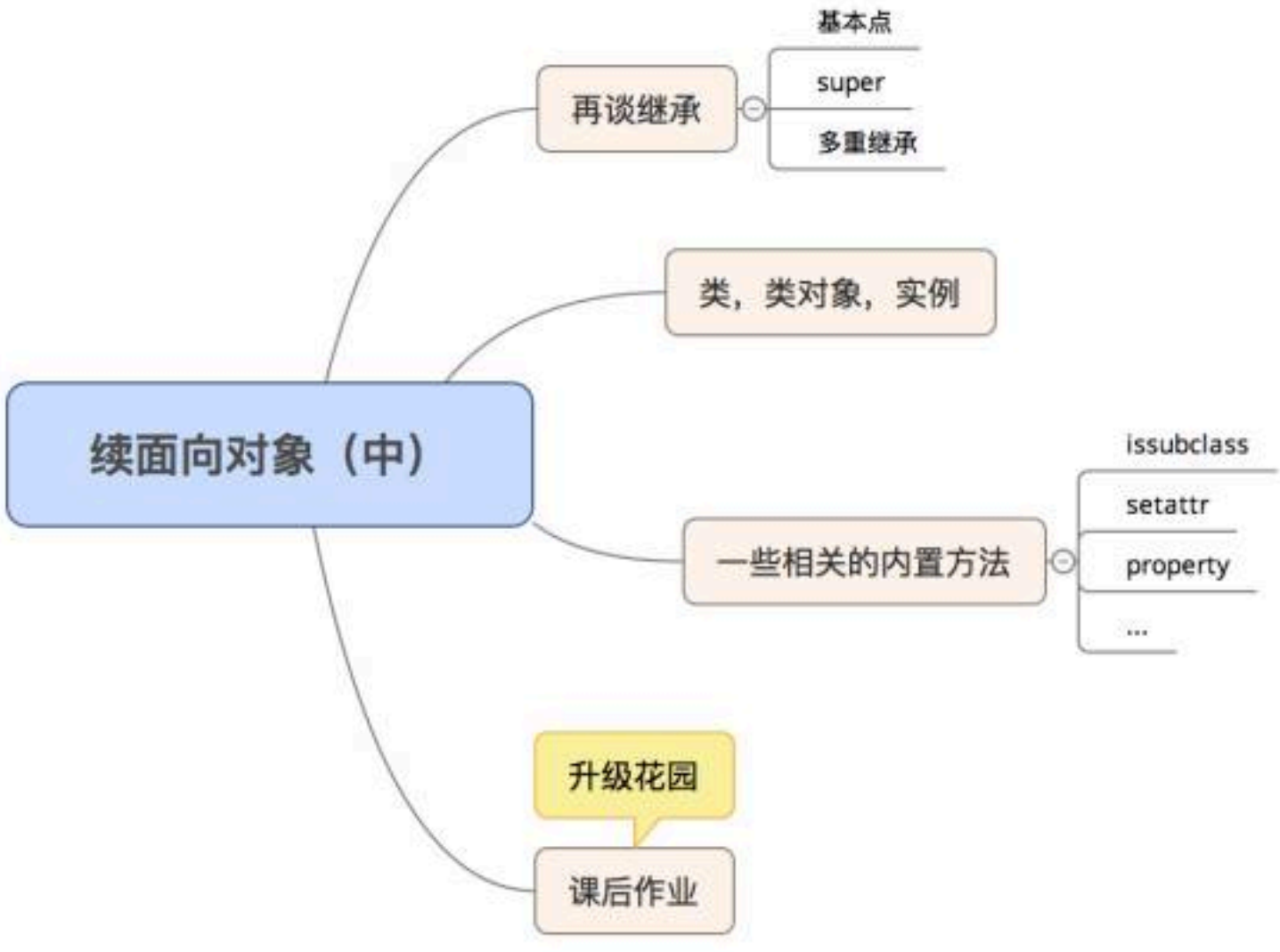
不断地放大，放大，出击，搏杀。乱箭齐飞，射向苍穹，惹怒众神，降暴雨雷霆，管他风雨无晴，只须踏步前行.....

呕偶，发生了什么事?

小事一件，哈哈～

哈哈小停笔一月备考，重新为大家献上好玩的python文章一篇。

通过本节的学习，你将了解一下内容：



一，再谈继承

1.基本点

“没墙的花园可以继承一般的花园，这里的一半的花园类叫做没墙花园类的基类或父类”

“多重继承用逗号隔开”

这是上篇讲面向对象三大特性时对继承的稍提一嘴的概括总结。

这篇的开头我们还是从继承讲起，不吝对他长篇大论，就因他的举足轻重。

记得王尔德在描述巨人的花园的首段有这样的描述：草丛中盛开着美丽的花朵 （Beautiful flowers grew in the grass）。

在这，我们发挥一下想象，想想这美丽的花朵都可能有哪些种类。

凤仙花(Impatiens)、紫茉莉(Mirabilis)、长春花(Vinca)、牵牛花(Ipomea)

这是哈哈小设想的结果。

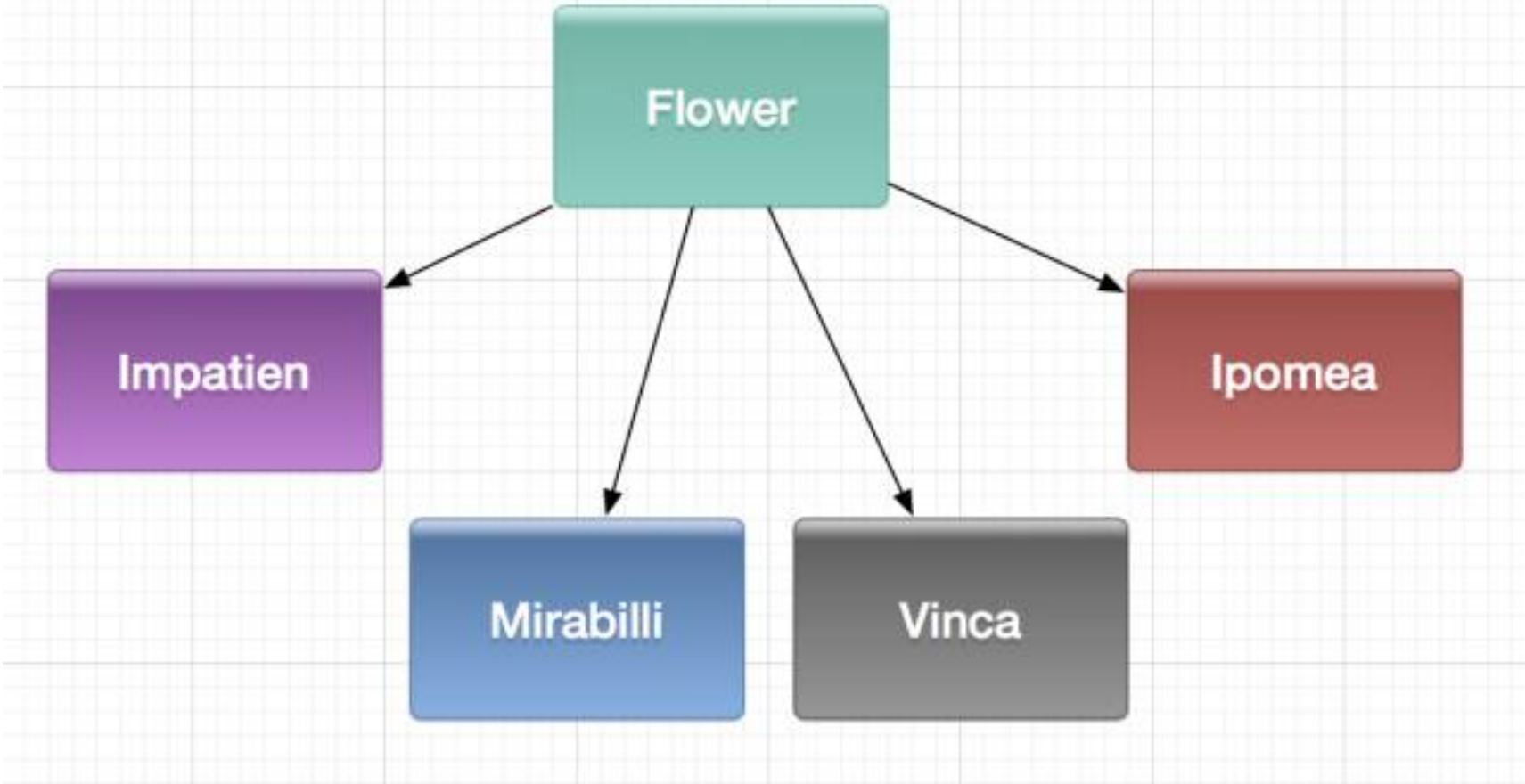
既然提这一茬，老司机应该都知道哈哈小接下来要干嘛了。

推进之前，这里先思考一个问题。

花的种类成千上万，这里只是列举四种。通过上一节的学习，我们都知道要对某件事物进行程序式的描述，可以通过分析其属性和行为从而对其类定义的方式。那么现在的问题是，如果我们要对这四种花去创建类，那是不是每个类的属性和行为都分析一遍，然后对照写成一个一个的类?

答案显然是否定的，我们稍加分析，就不难发现，寻找事物与事物之间的联系性，就很容易得出这样一个结论：

这四种花，毋宁说，都属于花。所以，他们有以下结构关系：



可以得出结论，通过对这四种相对具体的东西（具体的某种花）分析总结，可以抽象出一个更宽泛的概念（ general-izaiton）。

这个过程好比这样的：贯穿始终的学习过程，我们学习的大多就是前人对具体的系统总结，这总结常常可以说是某些理论。学习他们目的在于，当他们再次回归实际的时候，我们就能更好地用来指导行动了。

如果用程序来描述上一层级关系的话，继承就发挥了作用，见下：

```
class Flower: #start with upppercase
    """this is a class named Flower"""
    def __init__(self):
        self.color="红色"# 默认为红色
    def blossom(self):
        print("开花了 是",self.color)

class Impatien(Flower):
    pass

class Mirabili(Flower):
    pass

class Vinca(Flower):
    pass

class Ipomea(Flower):

    def __init__(self):
        self.shape="trumpet"
    def blossom(self):
        print("开花了 是",self.color)
    def getInfo(self):
        return "shape->"+self.shape+";color->"+self.color
```

代码对照图片理解，清晰明朗。

基类Flower中有一叫color的属性，给的默认颜色是红色，另外那四种花都是继承Flower的子类。牵牛花(Ipomea)中多了一个属性叫shape，多了一个方法，叫getInfo。

接下来跟着哈哈小来看一些例子，分析继承都对些子类做了些什么事。

```
>>> fl=Flower()
>>> fl.blossom()
开花了 是 红色
```

创建一个Flower类，再调用blossom方法，这个没毛病。

```
>>> im=Impatien()
>>> im.blossom()
开花了 是 红色
```

有发现，属于Flower子类的Impatient类的对象的blossom方法执行的结果跟它父类一样，看来是继承了父的方法。

再试一个

```
>>> im.color
'红色'
```

看来是继承了父的属性。现在可以推定，Mirabili和Vinca类也是一样的情况。

看看Ipomea怎么个情况?

```
>>> ip=Ipomea()
>>> ip.shape
'trumpet'  # 这个结果还挺靠谱的
>>> ip.blossom()
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>
ip.blossom()
File "/Users/wangcongcong/Desktop/flower1.py", line 22, in blossom
print("开花了 是",self.color)
AttributeError: 'Ipomea' object has no attribute 'color'
```

喔喔，不开心地发现报了个丑陋的错误。

仔细找找原因，推理得出，这个类有个__init__方法，它的父类也有，看来是把父类的重写覆盖了，所以程序才会说：'Ipomea' object has no attribute 'color'

在self.shape="trumpet"前面加这一行：Flower.__init__(self)后再运行

```
>>> ip=Ipomea()
>>> ip.blossom()
开花了 是 红色
>>>
```

验证成功。

2. super

上面解决Ipomea的问题用到的方法虽能可行但不雅，因为要是那天Flower吃饱了没事干，把自己变成了Hahahaflower，这时你要想程序正常运行，可得在Ipomea类中大作一顿修改了。

super对此自信十足，魔法棒一挥，就将Flower.__init__(self)被替换成了super().__init__()

```
>>> ip=Ipomea()
>>> ip.blossom()
开花了 是 红色
>>>
>>> ip=Ipomea()
>>> ip.getInfo()
'shape->trumpet;color->红色'
>>>
```

super靠谱，魔法显灵。

super的意思大概就是‘超’，‘上一级’，‘父’等。

super().__init__()这一句在表达这么一个意思：到我的父亲Flower， 告诉他，我需要他的__init__方法。

了解更多更深super的含义，可自行help(super)一下。

3.多重继承

“一个类可以继承多个父类（父类一，父类二），父类之间逗号隔开。就好比那位富二代也可以继承他叔或他舅的江山。”－来自上一节课程

花园里除了花，还有什么，也许还有一个池塘。现假设这池塘暂时有鱼和草，用多重继承来表述一下，是这样的：

```
class Fish:
    """this is a class named Fish"""
    def getFishInfo(self):
        print ('some fish')

class Grass:
    """this is a class named Grass"""
    def getGrassInfo(self):
        print ('some grass')


class Pool(Fish,Grass):
    pass
```

Pool类继承了Fish 和Grass类，继承了Fish和 Grass类的可见属性和可见方法。

```
>>> op.getGrassInfo()
some grass
>>> op.getFishInfo()
some fish
>>>
```

这么一通对继承的介绍，最后用两句话总结：

子类继承的是父类的可见的成员方法和属性

如果子类中定义与父类同名的属性或方法，则会自动覆盖父类对应的属性或方法（override）

二，类，类对象，实例对象

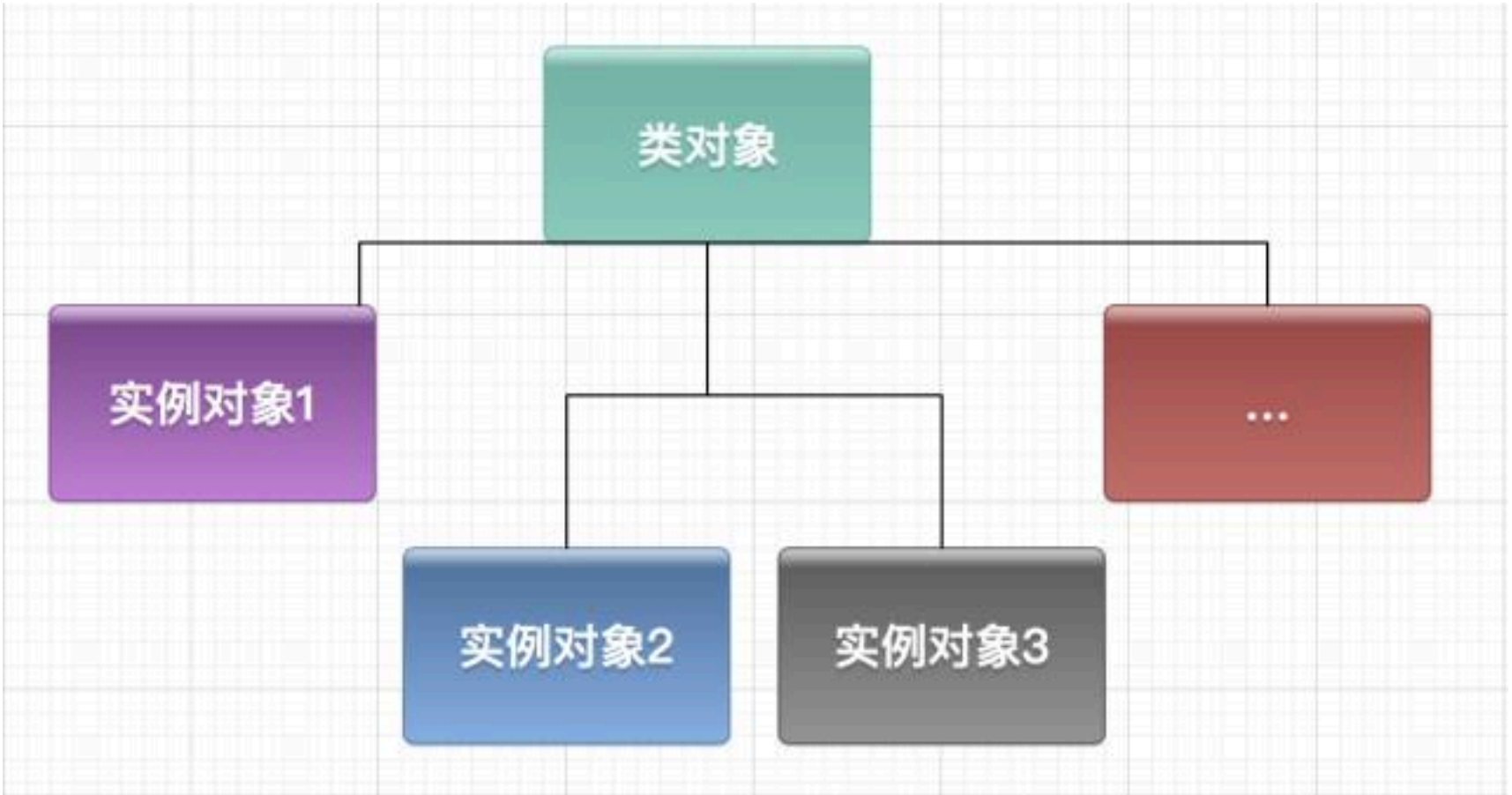
如果大家对前一篇类与对象的概念理解透彻，今天讲到的这三个概念就会省事很多。

类就不用多说了，静态的，用于创建出许许多多的实例对象。

类对象与实例对象不想的不同在于：前者是静态的，指向处于一块特定的内存空间（花园设计图纸的指引），指引地址直接使

用类名可得。后者是动态的，一个个对象指向处于一块块独立的不同的内存空间（依图纸建成的一座座处于不同地方的美丽的花园的指引），使用对象名得到对应的指引地址。

他们之间有以下约束关系：



废话少说，继续池塘的例子。

获取类对象的基本信息，可以使用一个内置方法 __dict__

```
>>> Pool.__dict__
mappingproxy({'size': 100, '__doc__': None, '__module__': '__main__'})
```

返回一个包含类对象基本信息的字典键值对，发现size属性也在其中

尝试调用

```
>>> Pool.size
100
>>>
```

完全没问题

创建两个实例对象pool1和pool2

```
>>> pool1=Pool()
```

```
>>> pool2=Pool()
>>> pool1.size=200
>>> pool2.size=300
>>> pool1.size
200
>>> pool2.size
300
>>> Pool.size
100
```

类对象的size值依然不变，而实例对象的size值做了对应的改变。

对照这个例子，从内存分配的角度去理解静态，动态，类对象，实例对象等概念，想必会轻松许多。

三， 一些好用的内置方法

类操作相关的一些内置方法

issubclass(class,classinfo)　classing可以是类对象组成的元组，只要class是其中任意一个候选类的子类，则返回true
isinstance(object,classinfo)　如果第一个参数不是对象，则永远为false
hasattr(object, name) 判断object对象是否含有叫name的属性
getattr(object, name[,default])得到object对象中叫name的属性，当不存在时，default可给提示
setattr(object, name, value)设置object对象中叫name的值为value的属性
delattr(object, name)删除object对象中叫name的属性
property(fget=None, fset=None, fdel=None, doc=None)

挑几个玩玩

> issubclass－判断一个类是否为另一个类的子类

池塘类拿来比方说

```
>>> issubclass(Pool,Grass)
True
>>> issubclass(Grass,Fish)
False
```

毫无疑问，Pool类是Grass的子类，所以返回True。Grass不是Fish的子类，所以结果你懂的。

再试一个

```
>>> issubclass(Pool,Pool)
True
```

也是True，这里要注意了，任何类视自身为其子类。

```
>>> issubclass(Pool,(Fish,Grass))
True
```

说好的可以传元组，所以这里没毛病

> isinstance()－判断一个对象是否为某个类或类元组的实例对象

试一下

```
>>> pool=Pool()
>>> isinstance(pool,Pool)
True
>>> isinstance(pool,(Pool,Fish))
True
>>>
```

是的，pool肯定是Pool类的实例。记住，类元组只要一项满足，就返回True

> setattr()－为某个实例对象设置属性

假设现在要为pool对象添加一个叫depth的属性，赋值为3

```
>>> setattr(pool,'depth',3)
>>> pool.depth
3
>>>
```

记住一点：在给name，第二个参数时，字符串要用分号包围

当然，你也可以用此方法对已存在的属性进行更改

```
>>> setattr(pool,'volum',350)
>>> pool.volum
350
>>>
```

> property() — 用起来会显得类的内聚力更强

看它所需参数，大概就可以猜出它的作用是，通过传入某个属性的删改查函数，从而更好地操作属性。

比方，现在要对pool的volum属性增加这三个函数并使用property方法

```
class Pool(Fish,Grass):
    def __init__(self):
        self.volum=300
    def getVolum(self):
        return self.volum
    def setVolum(self,v):
        self.volum=v
    def delVolum(self):
        del self.volum
    vo=property(getVolum,setVolum,delVolum,"I am a 'volum' property")
```

前三个参数，一见明朗，第四个参数就是用来说说话，没事唠唠的，不必太在意

再次运行，测试一下

```
>>> pool=Pool()
>>> pool.vo # 直接vo，实际就相当于调用了get方法
300
>>> pool.vo=350 # 实际就相当于调用了set方法
>>> pool.vo
350
>>> del pool.vo # 实际就相当于调用了del方法
>>> pool.vo # 删除之后，你还去访问 ，你这样忽略程序的感受，叫她情何以堪?
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>
pool.vo
File "/Users/wangcongcong/Desktop/flower1.py", line 42, in getVolum
return self.volum
```

```
AttributeError: 'Pool' object has no attribute 'volum'
>>>
```

通过上例的展示，也许有的同学会理直气壮地这样说：这property有个破用，直接调用那三个函数不就噢了，多此一举。

那哈哈小只能回答，你的理所当然就好比，你我的距离，就以为是咱两直线的距离，完全忽视了地球的感受。

哈哈，我们回归正题。

稍作思考，会发现：

一个类相当于一个封装📦，封装好后会对外界提供一些接口。

在这里，这个我们要改删查volum属性，仅仅通过使用vo这一个变量就可达到目的。otherwise，我们将要调用三个不同的函数，也就是说三个不同的接口。

这要是哪天吃饱了没事干，寻思着将这三个函数名改改，比方把delVolum 改成removeVolum，这还不得让外界那些调用了此借口的程序吃不消，也得跟着屁股做一番对应的修改?

专业一点说，这样的封装不符合开闭原则。

这样显然是不好的，再想想，如果用property生成vo，Pool类的这三个函数要再有变化，好像也影响不了外界调用者了。

勤思好学者，想了解这property到底用了什么魔法最终做成这样事的。

哈哈小和蔼地说：等着下回分晓吧。

四，课后作业

结合今天所学，升级成简单的巨人花园

要求：

有一花园类叫Garden

属性：

size=10000

花园类有一子类叫gaintGarden

方法：

__init__ :初始化name

def buildPool(self,pool): 传入池塘对象建设池塘

def getPool(self):返回池塘对象

def plantFlower(self,flower):传入花对象种植花

def getFlower(self):得到花对象

用property方法优化对池塘和花朵的访问

运行程序，尝试对所有出现的属性和方法进行访问和修改。

拓展题：

修改plantFlower和getFlower使得可以传入多种花的类型和任意花的数量

结束语：我们不做知识的搬运工，幸能成为思想的启迪者。

纠错：上篇出现的“分装”统统改为“封装”，手贱的叫人猝不及防，还请见谅。