

第十三章三大特性

一 继承

1. 继承的基本介绍
2. 如何寻找继承关系
3. 属性查找
4. 派生
5. 经典类与新式类
6. 多继承属性查找
7. `super()` 继承查找
8. 组合

二 封装

1. 封装基本介绍
2. 封装之Property方法

三 多态

1. 多态与多态性
2. 鸭子类型
3. 类的绑定方法
4. 非绑定方法

本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 继承

我们一直在说面向对象扩展性高，从来不知道他的扩展性高体现在哪些方面，接下来我们就来介绍他的三大特性，三大特性分别是继承，多态和封装，他的特性也是我们以后使用面向对象的一些技巧，第一个我

们来说的就是继承。

1. 继承的基本介绍

什么是继承

继承是一种新建类的方式，我以前定义的类都是class关键字加类名，在Python中支持多继承，就是一个儿子可以继承多个爹，这一点和其他的某些语言不同

从字面意思上理解就是你继承你爹的东西，他继承他爹的东西，继承就像一种遗传的关系，你爹有钱，你也有钱，要是你没钱，那可能你是你爹充话费送的，你爹有一套房子，你也可以跑进去住一下，当然如果你自己有房子，肯定会优先住你自己房子

继承新建的类称为子类或者派生类，被继承的类称为，父类，基类或者超类，子类会遗传父类的属性

假如你爹给你留了一个亿，你还有必要像其他同学一样苦逼呵呵的在这学Python吗，你就可以直接拿着这个钱来花，不用在自己挣钱，不用自己造了，可事实上你爹一毛都没留给你，所以你要靠自己。

为什么要用继承

所以使用继承就是为了减少代码冗余

怎么用继承

继承应该有多有儿子，下面我们来写代码

```
class ParentClass1:
    pass
class ParentClass2:
    pass
# 单继承，原来类名后面直接加冒号，现在我们可以先加括号，括号里面写他的父类
class SubClass1(ParentClass1):
    pass
# Python中支持多继承
class SubClass2(ParentClass1, ParentClass2):
    pass
# base是基础的意思，这个方法是查看他的父类
print(SubClass1.__bases__)
print(SubClass2.__bases__)
```

输出

```
(<class '__main__.ParentClass1'>,)
(<class '__main__.ParentClass1'>, <class '__main__.ParentClass2'>)
```

他的父类会以元组的形式现实出来

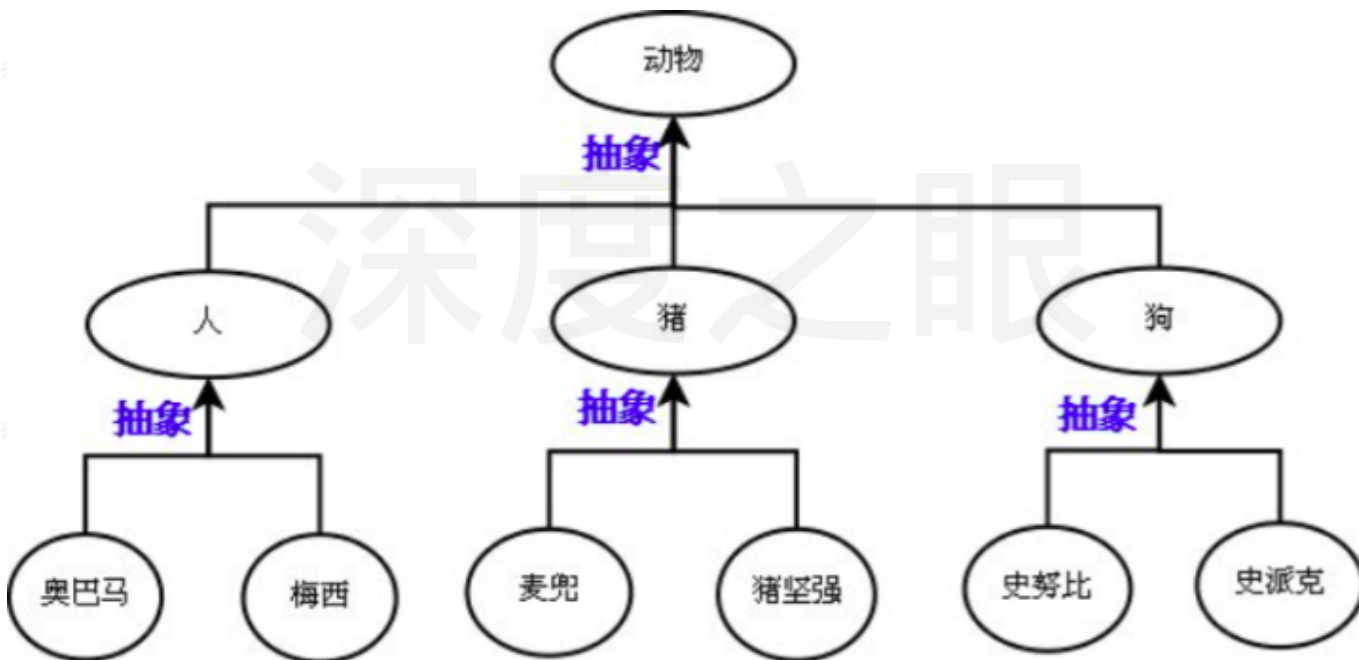
到目前位置我们大概知道了继承是怎么一回事，接下来我们先补充一个简单的小知识点，后面详细介绍

在Python2 中有经典类与新式类之分，
在Python3 中全都为新式类

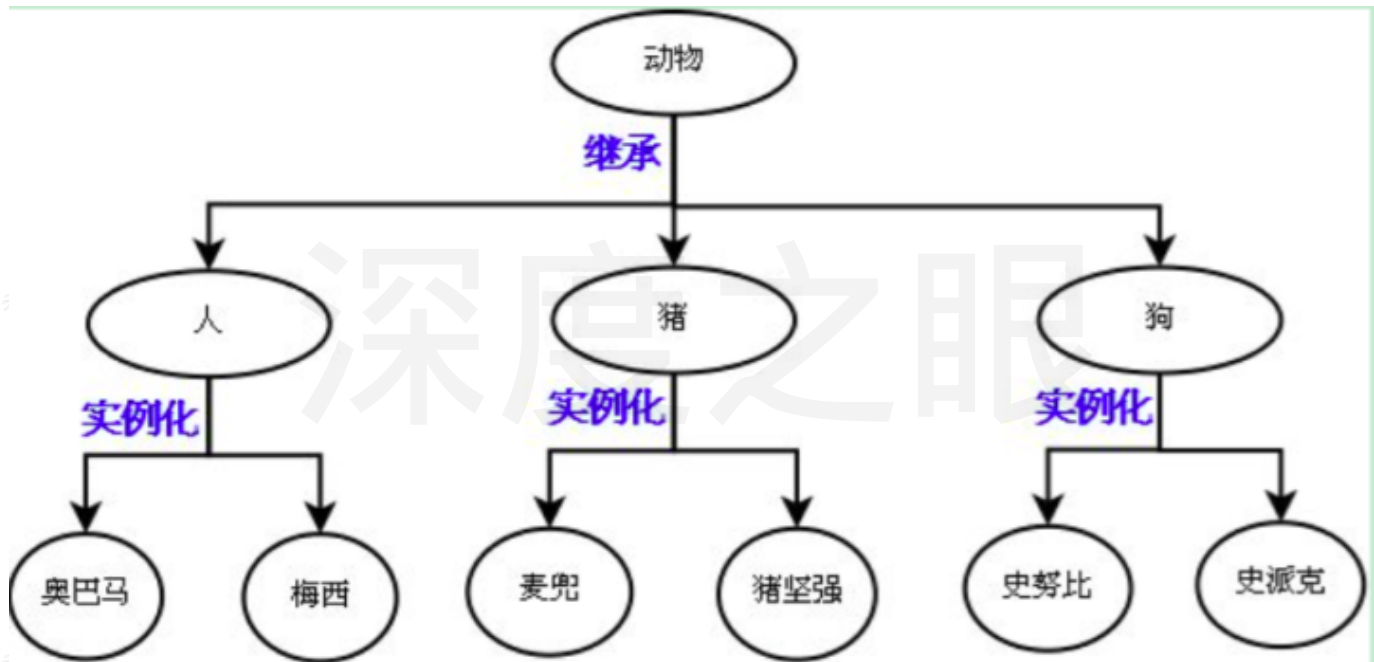
我们现在用的主流是Python3，而Python3 中全都是新式类，所以在Python3中没必要考虑这个，但是我们也需知道Python2中是怎么用的，具体怎么区分我们后面会详细讲到，现在大家先知道这个概念

2. 如何寻找继承关系

继承是类与类之间的关系，寻找这种继承关系需要先抽象，再具体。什么是抽象呢？就是抽取比较像的部分呀，我们刚学面向对象的时候就是抽取对象中相似的特征总结成了类，那么抽取类与类之间相似的地方就总结成了父类



这个图很明白，不同的人抽象成人类，人狗猪又可以抽象成动物类，抽象完了之后，继承关系就有了，人狗猪都是继承来自动物，他们分别实例化就能得到各自的对象



假如现在有这样一个需求，把老师和学生，学生可以自由选择课程学习，老师可以对学生的成绩做评分，按照原来的思路，我应该这样写代码

```

class DeepshareTeacher:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
class DeepshareStudent:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def choose(self):
        print('student %s is choosing course' % self.name)
    
```

代码写到这里你应可以发现了，这里面有重复的代码，为了避免代码冗余，我们应该使用继承的方式来优化我们的代码

```

class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
    
```

```

        self.name = name
        self.age = age
        self.gender = gender

class DeepshareTeacher(DeepsharePeople):
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)

class DeepshareStudent(DeepsharePeople):
    def choose(self):
        print('student %s is choosing course' % self.name)

teal = DeepshareTeacher('albert', 18, 'male')
stul = DeepshareStudent('张二狗', 18, 'female')

```

我们在执行代码实例化对象之前，先来看一下代码里面老师类和学生类里面都没有init这个实例化函数，但是程序没有报错说明他确实完成实例化了，怎么证明呢？

```
print(teal.name, teal.age, teal.gender)
```

输出

```
albert 18 male
```

这就说明了子类中没有init函数，他就会去父类中找，也就是说子类是继承了父类的属性，到此为止，你就会发现我们的代码已经精简了，但是他实现的功能并没有减少

3. 属性查找

有了继承关系以后，接下来我们再来说属性的查找，之前我们说过一个对象查找属性的时候，优先去自己的名称空间中查找，如果找不到就会去自己的类中去找，但是现在你会发现类中居然还有自己的父类，那你试想一下，如果一个对象他自己的名称空间中没有，他的类也没有，那接下来应该找谁？

没错，找他爹，如果他的父类没有并且再有父类呢？找他爷爷

```

class DeepsharePeople:
    school = 'deepshare'

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

```

```
class DeepshareTeacher(DeepsharePeople):
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
teal = DeepshareTeacher('albert', 18, 'male')
print(teal.__dict__)
print(teal.name)
print(DeepshareTeacher.__dict__)
print(DeepsharePeople.__dict__)
print(teal.school)
```

输出

```
{'name': 'albert', 'age': 18, 'gender': 'male'}
albert
{
  '__module__': '__main__',
  'modify_score': <function DeepshareTeacher.modify_score at 0x106db17b8>,
  '__doc__': None
}
{
  '__module__': '__main__',
  'school': 'deepshare',
  '__init__': <function DeepsharePeople.__init__ at 0x1092a2488>,
  '__dict__': <attribute '__dict__' of 'DeepsharePeople' objects>,
  '__weakref__': <attribute '__weakref__' of 'DeepsharePeople' objects>,
  '__doc__': None
}
deepshare
```

从第一个和第二个print中我们可以看出，对象自己的名称空间中有，会优先找自己的名称空间，从第三四五五个打印中我们可以发现，teacher对象自己名称空间中没有，会先从自己的类中找，发现自己的类中也没有，就会到他的父类中找，在他的父类中找到了，就会有结果

再来看一个例子

```
print(teal.modify_score)
```

输出

```
<bound method DeepshareTeacher.modify_score of <__main__.DeepshareTeacher
object at 0x10482cdd8>>
```

在对象自己的类里面直接就找到了，不再往下寻找了
接下来，我们再来看一个复杂点的例子

```
class Foo:
    def f1(self):
        print('Foo.f1')
    def f2(self):
        print('Foo.f2')
        self.f1()

class Bar(Foo):
    def f1(self):
        print('Bar.f1')

obj = Bar()
obj.f2()
```

输出结果，我先不贴出来，大家猜一下，再往后面看

分析：

obj是Bar的对象，他调f2方法，先从自己的名称空间中找f2属性，这肯定是没有的，因为obj我们没有给他定制自己的属性，来看一下吧

```
print(obj.__dict__)
```

输出

```
{}
```

那么接下来找自己的类的名称空间，自己类的名称空间没有，那就去父类Foo中找，找到了就会打印Foo.f2，这一点肯定没有问题，那么下面一行self.f1()应该打印哪个f1呢，是离他更近的Foo.f1,还是距离他较远的Bar.f1呢

这里其实与距离远近没有关系，很多同学容易误解，所以我这里就拿出做一个反响引导，当代码执行到self.f1()这一行的时候，我们先要搞清楚self是谁，他就是obj这个对象，按照属性查找顺序

对象自己的名称空间====>对象所在的类的名称空间====>对象的所在的类的父类的名称空间

4. 派生

刚刚我们讲完了继承，继承的好处是子类可以重用父类的代码，但问题是子类是不是就和父类一模一样就行了，如果这样那你写子类的目的是什么呢？子类有了以子类为准，所以在这里我们再讲一个叫做派生的概念

其实这个概念本章一开始我们就提及到了，假如你爹有一套房子，你可以进去住一下，但是如果你自己有房子，肯定会优先住自己的房子

派生就是在继承的基础上有自己新的东西，如果有自己新的东西，肯定就会以自己新的东西为准

派生：子类定义自己新的属性，如果与父类同名，以子类自己的为准

```
class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def f1(self):
        print('爹的f1')

class DeepshareTeacher(DeepsharePeople):
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def f1(self):
        print('儿子的f1')

tea1 = DeepshareTeacher('albert', 18, 'male')
tea1.f1()
```

输出

儿子的f1

我们再来看一个例子

```
class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
```



```

def f1(self):
    print('爹的f1')

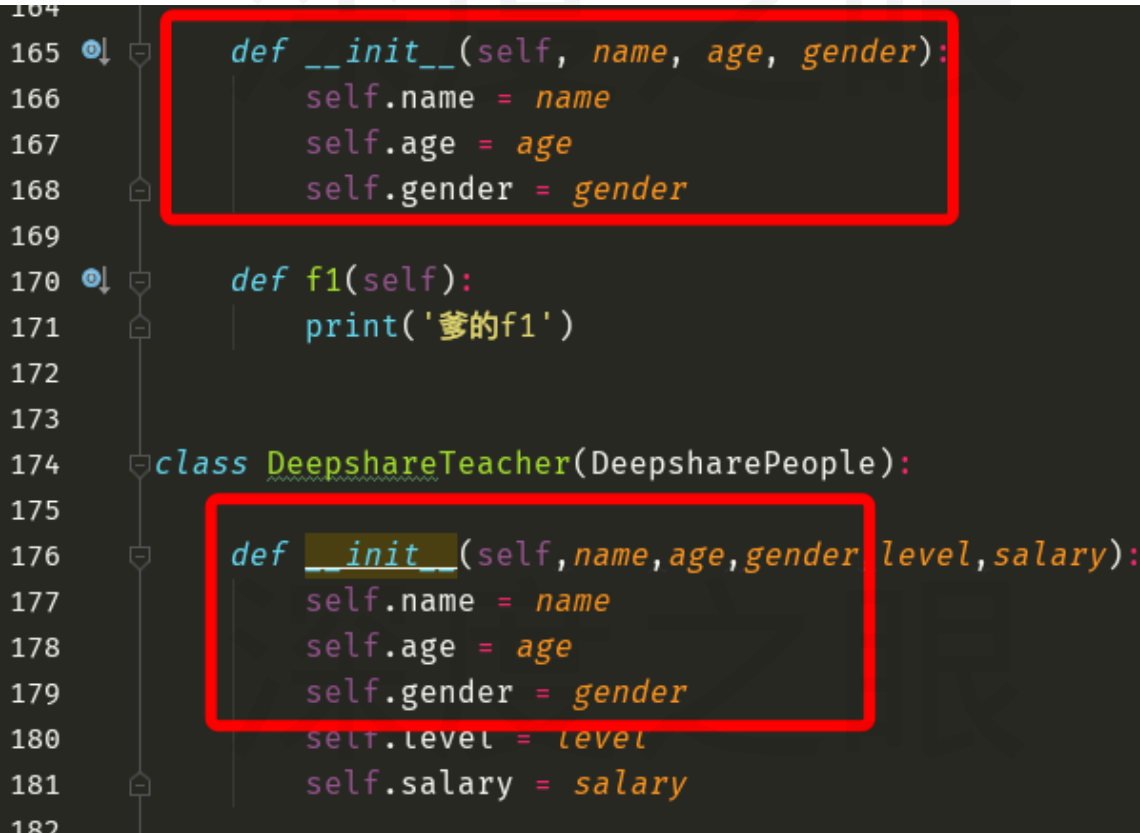
class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        self.name = name
        self.age = age
        self.gender = gender
        self.level = level
        self.salary = salary
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def f1(self):
        print('儿子的f1')

tea1 = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
print(tea1.name, tea1.age, tea1.gender, tea1.level, tea1.salary )

```

tea1实例化对象的时候其实就是去他的类中找init函数，如果他自己没有那肯定是去父类中找，但是现在他自己有了，就用自己的，这就是派生的应用，在子类定制自己的属性，从而覆盖父类。子类有一些独特的功能，单独定义，有一些大家共有的在父类还能使用。

从上面的派生中你有没有发现，我们虽然实现了功能，但是子类里面写了一些重复的代码，我们必须要把重复代码去掉，又要保证这些重复的代码还能用，这又应该怎么操作呢？



```

165 def __init__(self, name, age, gender):
166     self.name = name
167     self.age = age
168     self.gender = gender
169
170 def f1(self):
171     print('爹的f1')
172
173
174 class DeepshareTeacher(DeepsharePeople):
175
176     def __init__(self, name, age, gender, level, salary):
177         self.name = name
178         self.age = age
179         self.gender = gender
180         self.level = level
181         self.salary = salary
182

```

我们必须要想一个方案，在子类派生出来的方法中能够重用父类的一段功能

第一种想法

```
class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def f1(self):
        print('爹的f1')
class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        # self.name = name
        # self.age = age
        # self.gender = gender
        self.__init__(name, age, gender)
        self.level = level
        self.salary = salary
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def f1(self):
        print('儿子的f1')
teal = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
```

我们对象自己重复调用自己的init函数，这是一个递归，执行代码发现报错了，他说少两个参数，其实我们想要调用的并不是自己，而是父类的init，用对象调已经不行了，那我们就用他的父类来调
派生第一种方案（指名道姓的调用父类）：

```
class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def f1(self):
        print('爹的f1')
class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        # self.name = name
        # self.age = age
        # self.gender = gender
        # 用类来调用，没有自动传值，要把self写上
```

```

    DeepsharePeople.__init__(self, name, age, gender )
    self.level = level
    self.salary = salary
def modify_score(self):
    print('teacher %s is modifying score' % self.name)
def f1(self):
    print('儿子的f1')
teal = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
print(teal.name, teal.age, teal.gender, teal.level, teal.salary )

```

到目前为止，我们已经实现了子类重用父类的功能，但是这种方式 and 继承有关系吗，我们是否可以指名道姓的调用一个和我们当前类没有任何继承关系的一个类呢？当然可以

下面我们在父类中添加一些东西

```

class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def info(self):
        print("""
        =====个人信息=====
        姓名：%s
        年龄：%s
        性别：%s
        ""%(self.name, self.age, self.gender) )
class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        # self.name = name
        # self.age = age
        # self.gender = gender
        DeepsharePeople.__init__(self, name, age, gender )
        self.level = level
        self.salary = salary
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def f1(self):
        print('儿子的f1')
teal = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
teal.info()

```

teacher还有自己等级和薪资，那么我们来在子类中添加

```
class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def info(self):
        print("""
        =====个人信息=====
        姓名：%s
        年龄：%s
        性别：%s
        ""%(self.name,self.age,self.gender))

class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        # self.name = name
        # self.age = age
        # self.gender = gender
        DeepsharePeople.__init__(self, name, age, gender)
        self.level = level
        self.salary = salary
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def info(self):
        print("""
        =====个人信息=====
        姓名：%s
        年龄：%s
        性别：%s
        ""%(self.name,self.age,self.gender))
        print("""
        等级：%s
        薪资：%s
        ""%(self.level,self.salary))

tea1 = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
tea1.info()
```

依然有代码重复，如法炮制，他们的原理是一样的

```
class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def info(self):
        print("""
        =====个人信息=====
        姓名：%s
        年龄：%s
        性别：%s
        """)%(self.name, self.age, self.gender))

class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        # self.name = name
        # self.age = age
        # self.gender = gender
        DeepsharePeople.__init__(self, name, age, gender)
        self.level = level
        self.salary = salary
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def info(self):
        # 注意：用类去掉不会自动传self，需要我们手动写一下
        DeepsharePeople.info(self)
        print("""
        等级：%s
        薪资：%s
        """)%(self.level, self.salary))

tea1 = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
tea1.info()
```

派生第二种方案（用super()调用）：

super() 的返回值是一个特殊的对象，该对象专门用来调用父类中的属性

我们用super来修改我们的代码

```

class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def info(self):
        print("""
        =====个人信息=====
        姓名：%s
        年龄：%s
        性别：%s
        """)%(self.name, self.age, self.gender)

class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        # self.name = name
        # self.age = age
        # self.gender = gender
        # DeepsharePeople.__init__(self, name, age, gender)
        super().__init__(name, age, gender) # 这里调用的是父类的init, super()是一个对象
        self.level = level
        self.salary = salary
    def modify_score(self):
        print('teacher %s is modifying score' % self.name)
    def info(self):
        # DeepsharePeople.info(self)
        super().info() # 同理, super()返回值是一个对象, 不要self, 调用的info是父类的info

print("""
等级：%s
薪资：%s
""")%(self.level, self.level)

tea1 = DeepshareTeacher('albert', 18, 'male', '10 ', '3.1')
tea1.info()

```

输出

```

=====个人信息=====
    姓名：albert
    年龄：18
    性别：male

```

等级：10

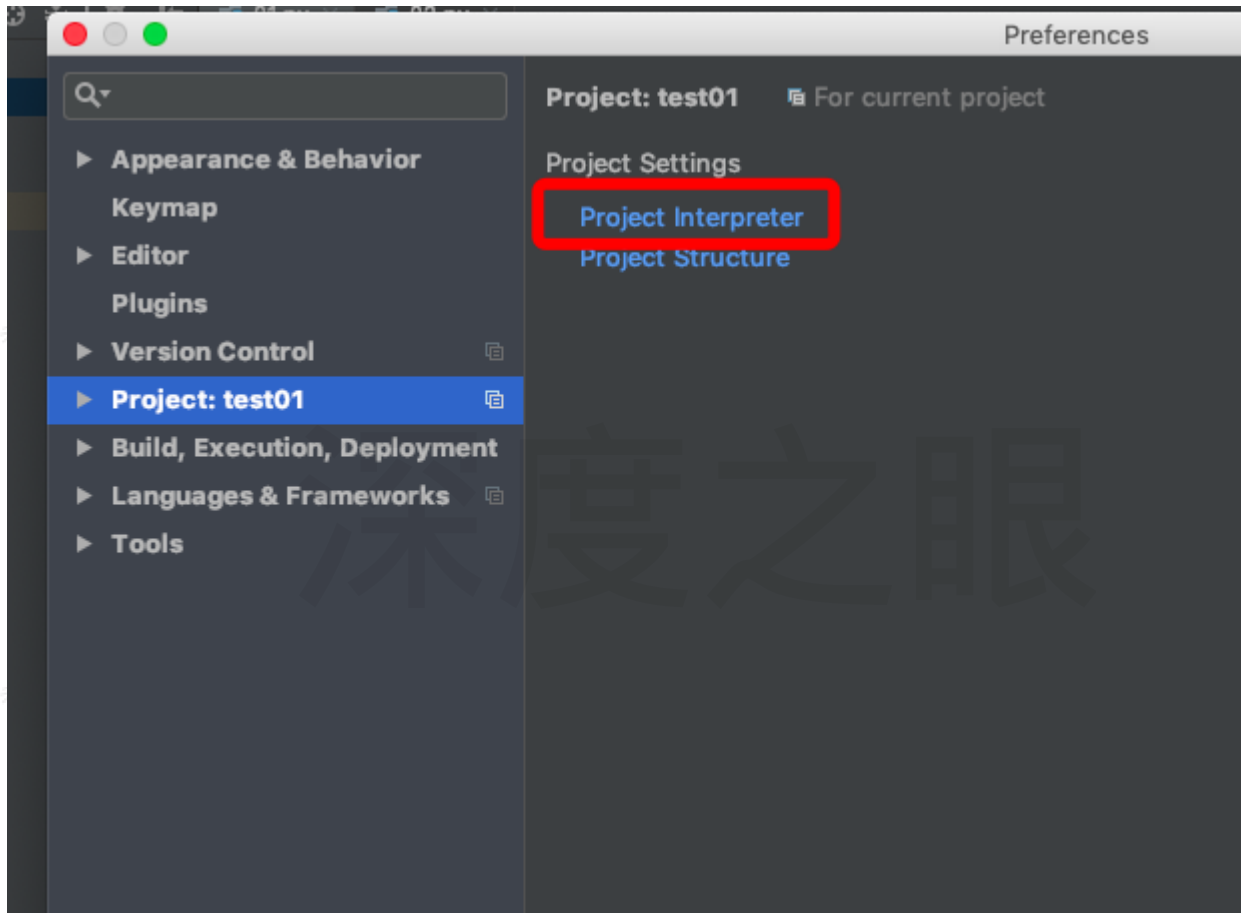
薪资：10

super的调用与第一种派生方式不同，他是严格依赖继承关系，这两种方式都可以使用，没有谁好谁坏，但是两种方式千万不要混合使用，那谁也救不了你啊

补充：

在Python2 中需要super(自己的类名，self)这么去用

示例：



这样切换解释器，然后代码这样写：

```

184         # self.gender = gender
185         # DeepsharePeople.__init__(self, name, age, gender)
186         super(DeepshareTeacher, self).__init__(name, age, gender)
187         self.level = level
188         self.salary = salary
189
190     def modify_score(self):
191         print('teacher %s is modifying score' % self.name)
192
193     def info(self):
194         # DeepsharePeople.info(self)
195         super(DeepshareTeacher, self).info()
196         print(
197             等级: %s

```

5. 经典类与新式类

一开始的时候我们就引出了这个概念，现在我们来理解一下

新式类：继承object的类，以及该类的子类孙子类，都是新式类

经典类：没有继承object的类，以及该类的子类，孙子类，都是经典类

我们先来看一些object是个什么东东

```
print(object)
```

输出

```
<class 'object'>
```

object也是一个类，他就是类的老祖宗，在Python3当中定制了大量的功能，都是自动触发的功能，以此来帮你来完善你编写的类的一些功能。Python3中的类都是新式类，那就必然继承了object，我们来看一下

```
class Foo:
    pass
```



```
print(Foo.__bases__)
```

输出

```
(<class 'object'>,)
```

这说明Python3中一个类没有指定的父类，默认就会继承object类，所以Python3中都是新式类
但如果是Python2 中呢（解释器已切换到Python2）

```
class Foo:  
    pass  
print(Foo.__bases__)
```

输出

```
()
```

这说明这个类没有继承object，那么他是一个经典类，但我要是让他继承呢？

```
class Foo(object):  
    pass  
print(Foo.__bases__)
```

输出

```
(<type 'object'>,)
```

这说明他是一个新式类，现在解释器切换回Python3，输出

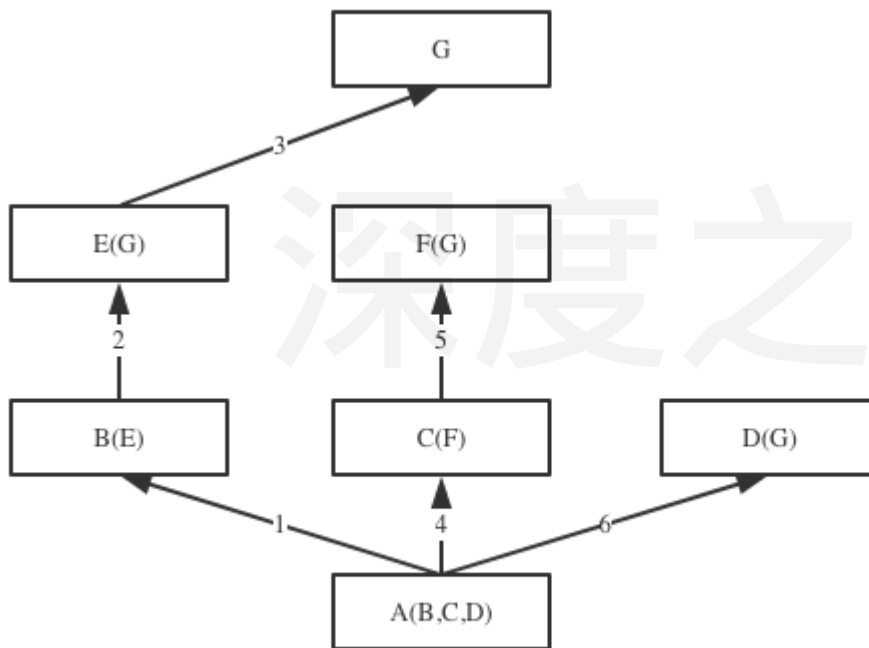
```
(<class 'object'>,)
```

很明显在Python3中，你加不加object都是一样的，这就说明了Python3中只有新式类，而Python2中既有经典类，又有新式类，但其实即使是使用Python2的程序员，经典类也是用的比较少了，所以你会看到一些老的程序员或者一些老得项目上，定义类的时候后面都会加一个object，这样有三种可能，一 项目是用Python2 写的，二 老程序员的习惯，因为他以前用Python2 都要加这个，三 为了是项目代码具有一定的兼容性，在Python2和Python3中都可以正常运行

6. 多继承属性查找

我们了解了经典类与新式类，接下来我们要清楚他们的区别在哪里，区别就是在属性查找上，我们知道Python支持多继承，我们之前研究的都是单继承，接下来我们就来研究多继承。

先说一下结论，请看下图，当继承关系为菱形结构，那么属性查找的方式有两种，分别是深度优先和广度优先



当类是经典类时，多继承情况下，在要查找属性不存在时，会按照深度优先的方式查找下去

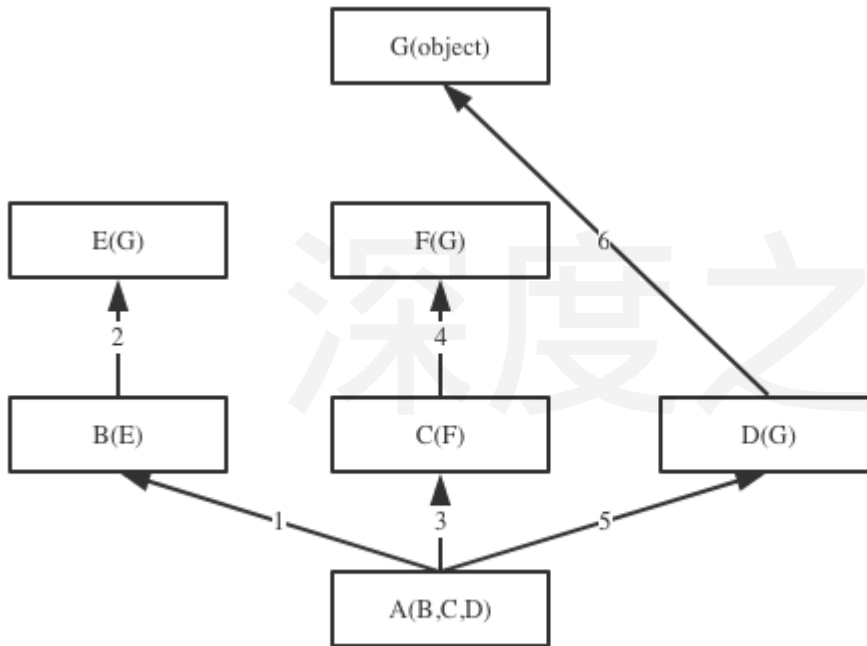
注意：

只有当继承关系组成一个菱形结构，也就是最后至少有两个类都继承了同一个类，这时才会区分经典类与新式类，也就是说如果EFD这三个类中有一个类或者没有类继承G类，那就不是菱形结构，他的继承关系属性查找就是从左到右， $B \Rightarrow E \Rightarrow (G \Rightarrow) C \Rightarrow F \Rightarrow D$ ，在这一点上新式类与经典类没有区别，

当继承关系是菱形结构时，深度优先就是一条道走到黑， $B \Rightarrow E \Rightarrow G \Rightarrow C \Rightarrow F \Rightarrow D$

广度优先就是先试探再迂回， $B \Rightarrow E \Rightarrow C \Rightarrow F \Rightarrow D \Rightarrow G$

这个是Python内部的实现机制，它使用的是C3算法去计算出来的



当类是新式类时，多继承情况下，在要查找属性不存在时，会按照广度优先的方式查找下去

接下来我们用代码来验证一下

```
class A(object):
    def test(self):
        print('from A')
class B(A):
    def test(self):
        print('from B')
class C(A):
    def test(self):
        print('from C')
class D(B):
    def test(self):
        print('from D')
class E(C):
    def test(self):
        print('from E')
class F(D,E):
    # def test(self):
    #     print('from F')
    pass
f1=F()
f1.test()
```

在这里我们就不花费大量的篇幅去验证了，大家可以自己去验证一下，另外补充一点，在新式类中，专门提供了一个属性查找方法mro

```
f1=F()
f1.test()
print(F.__mro__)
print(F.mro())
```

输出：

```
from D
(<class '__main__.F'>, <class '__main__.D'>, <class '__main__.B'>, <class
 '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
[<class '__main__.F'>, <class '__main__.D'>, <class '__main__.B'>, <class
 '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

他会以列表或者元组的形式保存查找顺序，因为这种数据类型都是有顺序的

7. super() 继承查找

接下来是super()对象继承查找的特殊之处，如果代码是这样的，那么自然是没有问题的：

```
class X:
    def test(self):
        print('x')
    def f1(self):
        print('XF')
class A(X):
    def test(self):
        print('1 打印一下，代码先经过这里')
        super().f1()
        print('3 再打印一下，代码最后经过这里')
class B:
    def f1(self):
        print('2 再经过这里from B')
class C(A, B):
    pass
c = C()
```

```
c.test()
print(C.mro())
```

输出：

```
# 1 打印一下，代码先经过这里
# XF
# 3 再打印一下，代码最后经过这里
# [<class '__main__.C'>, <class '__main__.A'>,
    <class '__main__.X'>, <class '__main__.B'>, <class 'object'>]
```

但是如果是`super()`对象参与进来的继承查找，`super()`所继承的类并没有该方法(object类或者上面代码的X类都可以)，这时mro列表的继承顺序会和我们理解的有冲突，继承顺序会严格按照mro列表从当前查找到的位置继续往后查找，mro方法认为，只要有与当前类平级的类中有他需要查找的方法，那就不管那么多了，认爹吧，所以：即使A与B之间本没有直接的继承关系，A也会把B当成爹，最后再回到当前位置，继续往下执行代码，就像是中了`super()`的毒一样，而不是按照我们原本的理解，寻找`super()`对象的test方法，找不到然后找它的父类object类或者X类，然后报错。

示例如下：

```
class X: # X类没有也是一样的
    def test(self):
        print('x')
    # def f1(self):
    #     print('XF')
class A(X):
    def test(self):
        print('1 打印一下，代码先经过这里')
        super().f1()
        print('3 再打印一下，代码最后经过这里')
class B:
    def f1(self):
        print('2 再经过这里from B')
class C(A, B):
    pass
c = C()
c.test()
print(C.mro())
```

输出

```
from B
[<class '__main__.C'>, <class '__main__.A'>,
    <class '__main__.B'>, <class '__main__.X'>, <class 'object'>]
```

```
<class '__main__.X'>, <class '__main__.B'>,
<class 'object'>]
```

在子类中重用父类的方法我们知道两种方式：一种是指名道姓的调用父类，另外一种是用super，这两种方式哪种好其实在Python社区的争论一直都有，各有优略，super其实查找一个属性并不够明确，而另外一种指名道姓的方式你立马就能看出来调用的是哪个类里面的东西，但这种方式却又和继承关系不大。所以，以后还是用哪一种都可以，但是别混着用。

8. 组合

严格来讲，其实组合与继承并没有直接的关系，但是组合和继承都是解决类与类之间代码冗余的方案，站在面向对象的角度考虑他们是同一类，所以我们这里放在一起讲。区别是继承描述的是类与类之间的从属关系，也就是什么是谁的关系，而组合描述的是类与类之间的交叉关系，也就是谁有什么的关系

一个类产生的对象，该对象拥有一个属性，这个属性的值是来自于另外一个类的对象

依然是我们前面的老师月学生的代码，我希望老师对象能够有出生日期这个属性，学生对象也有这个属性，这应该怎么操作呢？

```
class Date:
    def __init__(self, year, mon, day):
        self.year = year
        self.mon = mon
        self.day = day
    def tell_birth(self):
        print('出生年月日<%s-%s-%s>' % (self.year, self.mon, self.day))

class DeepsharePeople:
    school = 'deepshare'
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

class DeepshareTeacher(DeepsharePeople):
    def __init__(self, name, age, gender, level, salary):
        super().__init__(name, age, gender)
        self.level = level
        self.salary = salary
    def change_score(self):
        print('teacher %s is changing score' % self.name)
```

```

class DeepshareStudent(DeepsharePeople):
    def __init__(self, name, age, gender, course,):
        super().__init__(name, age, gender,)
        self.course=course
    def choose(self):
        print('student %s choose course' %self.name)
tea1=DeepshareTeacher('albert',18,'male',10,3.1)
date_obj=Date(2000,1,1)
date_obj.tell_birth()
tea1.birth=date_obj
print(tea1.birth)
tea1.birth.tell_birth()
tea1.change_score()
stu1=DeepshareStudent('张三',16,'male','AI')
stu1.birth=Date(2002,3,3)
stu1.birth.tell_birth()

```

到这里我们会发现解决代码重用问题会有两种方案，一种叫继承，一种叫组合，所以在我们本周的项目中会大量的使用组合，因为从软件设计的角度来考虑，使用组合其实会比继承更好，就刚才的场景来说继承也能实现，但是继承相当于是类与类之间做了一种强耦合，而组合实质上是一种解耦合，在工作中考虑到软件的可拓展性，代码的耦合性越高，越不利于拓展。当然继承依然是必不可少的，在后面的课程中，我们会带领大家去解读一些Python写的经典的源码，里面会大量的使用组合与继承，这一章节非常重要，所以我花费了大量的篇幅去讲解。

二 封装

1.封装基本介绍

什么是封装

先从字面意思来理解，封装就是封上口，装起来，让他跑不出来，你在外面不知道它里面有什么，但是如果把你一块装进去，你肯定就是知道了，不过你就别想出来了

装就是把属性存起来，封就是把这些属性隐藏起来

封装但从字面意思看等同于隐藏，但绝对不是单纯意义的隐藏

如何用封装

```
class Foo:
    x = 111
    def f1(self):
        print('Foo.f1')

obj = Foo()
print(obj.x)
```

以上这段代码，我想把x属性隐藏起来

```
class Foo:
    __x = 111  # 注意这里是__开头，没有__结尾，杠杠开头杠杠结尾的是在某种情况下自动触发的方法
    def f1(self):
        print('Foo.f1')

obj = Foo()
print(obj.x)
print(obj.__x)
print(obj.__x)
```

现在你不管怎么访问，都访问不到，这就已经完成了隐藏，同理，如下

```
class Foo:
    __x = 111  # 注意这里是__开头，没有__结尾，杠杠开头杠杠结尾的是在某种情况下自动触发的方法
    def __init__(self, y):  # 注意这不是封装
        self.__y = y
    def __f1(self):
        print('Foo.f1')

obj = Foo(222)
print(obj.x)
print(obj.__x)
print(obj.f1)
print(obj.__f1)
print(obj.y)
print(obj.__y)
```

x,y,f1属性都找不到了，如果我现在不告诉你，你能用所学知识找到他的属性吗？（据说十个人里有9个能找到）

为一个属性名加__开头，会在类定义阶段将属性名变形：_属性名 ==> _类名__属性名
复习：类定义阶段执行类体内的代码（类体内的函数只检测语法，不执行）

怎么证明呢？这也是上面的问题的答案

查看类的名称空间

修改调用方式

```
print(obj._Foo__x)
print(obj._Foo__f1)
print(obj._Foo__y)
```

输出

```
111
<bound method Foo.__f1 of <__main__.Foo object at 0x104e63f98>>
222
```

其实这种隐藏只是一种语法上的变形，并不是一定要限制你的访问，你自己要非要访问，那也可以，不过你自己做的隐藏，你自己非要在外部访问，俺不知道你咋想的
你要是能正常一点，肯定是应该在内部访问呀

```
class Foo:
    __x = 111 # 注意这里是__开头，没有__结尾，杠杠开头杠杠结尾的是在某种情况下自动触发的方法
    def __init__(self, y):
        self.__y = y
    def __f1(self):
        print('Foo.f1')
    def get__y(self): # 这个get__y属性肯定是能访问到的啊
        print(self.__y)

obj = Foo(222)
obj.get__y()
```

为什么get__y就能访问到__y属性的值呢？原因很简单，给你一次自己思考的机会，留作一个小问题，写到作业里

这种语法意思上的变形只在类定义阶段发生一次，类定义之后新增的【杠杠】开头的属性没有变形的效果，证明如下：

```
obj.__aaaaaaaaaaaaaa = 1
Foo.__bbbbbbbbbbbbbb = 2
print(Foo.__dict__)
print(obj.__dict__)
```

如果父类不想让子类覆盖自己的方法，可以在方法名前加【杠杠】开头，代码如下：

```
class Foo:
    def __f1(self): #_Foo__f1
        print('Foo.f1')
    def f2(self):
        print('Foo.f2')
        self.__f1() #obj._Foo__f1()
class Bar(Foo):
    def __f1(self): #_Bar__f1
        print("Bar.f1")
obj=Bar()
obj.f2()
"""
解释：因为__f1函数在两个类中发生了不同的形变
"""
```

为什么要用封装

试想一下电脑开机的过程，对用户而言只需要按一个按钮，但是对机器来说经过了很多的步骤，这些步骤根本没必要给你看（看了你也看不懂），只需要给你开一个接口，就是这个按钮，是个人都会按，按一下这个按钮就能控制开机

```
class People:
    def __init__(self,name,age):
        self.__name = name
        self.__age = age
    # 这就是封装的接口
    def tell_info(self):
        print(self.__name,self.__age)
```

```
p = People('albert',18)
# 给用户调这个接口
p.tell_info()
```

封装变量属性的目的：需要专门开辟接口给类外部的使用者使用，从而达到，我们可以在接口之上添加任意控制逻辑，从而严格控制访问者对属性的操作，封装不是不想给你用，而是不想给你随使用，从而达到控制的目的，如下：

```
class People:
    def __init__(self,name,age):
        self.__name = name
        self.__age = age
    # 这就是封装的接口
    def tell_info(self):
        u = input('user>>:').strip()
        p = input('pwd>>:').strip()
        if u == 'albert' and p == 'abc':
            print(self.__name,self.__age)
p = People('albert',18)
# 给用户调这个接口
p.tell_info()
```

接下来我们再来说封装函数属性，假如你去银行ATM机取钱的时候，只需要把银行卡插进去，输入密码，然后输入对应的取款金额就可以了，至于它内部经历了怎么样的过程，我们不需要考虑。

```
class ATM:

    def put_your_card(self):
        print('1 插卡')

    def input_password(self):
        print('2 输密码')

    def input_money(self):
        print('3 输入取款金额')

    def __sent_request(self):
        print('向银行系统发请求(不让你看)')

    def __dedcut_money(self):
        print('扣钱(不让你看)')
```

```
def __update_user_balance(self):
    print('更新用户余额（不让你看）')
```

```
def withdraw(self):
    self.__sent_request()
    self.__dedcut_money()
    self.__update_user_balance()
```

```
obj = ATM()
obj.put_your_card()
obj.input_password()
obj.input_money()
obj.withdraw()
```

封装函数属性的目的就是为了隔离复杂度

封装终极奥义：目的就是为了明确的区分内外，对外部是隐藏的，对内部是开放的

2. 封装之Property方法

BMI指数的计算，注意BMI并不是一个固定死的值，随着人的年龄增长，小孩的身高和体重都会发生相应的变化，成人的体重也可能会发生变化，所以你不能给他写到对象初始化的init函数中去，必须要单独编写一个功能来计算，也就是单独定义一个函数，但是他明显是一个属性，就是对象的名字身高一样，所以我们希望用对象点属性名拿到属性值

```
'''
BMI指数
成人的BMI数值：
过轻：低于18.5
正常：18.5-23.9
过重：24-27
肥胖：28-32
非常肥胖， 高于32

体质指数（BMI）=体重（kg）/身高^2（m）
EX：70kg /（1.75×1.75）=22.86
'''
```

如果没有Property方法

```
class People:
    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight
        self.height = height
    def bmi(self):
        return self.weight / (self.height * self.height)
albert = People('albert', 75, 1.80)
print(albert.bmi())
```

使用函数去调用，这样做的目的正常理解应该是触发一个功能，这个功能肯定是一个动作，这样并不规范，我们希望bmi后面不加括号，就像调对象的名字一样，来调用对象的bmi，不同是bmi是动态变化的，使用property修改

```
class People:
    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight
        self.height = height
    @property
    def bmi(self):
        return self.weight / (self.height * self.height)
albert = People('albert', 75, 1.80)
print(albert.name)
print(albert.bmi)
# print(albert.bmi())
```

其实property本质原理也是封装，不过现在是别人已经封装好了，我们以前讲过装饰器，就是在被装饰对象的正上方加一个@装饰器名，现在这个装饰器龟叔已经写好了，它相当与把一个对象的函数属性伪装成了变量属性，强大之处在于他可以动态变化，如下：

```
class People:
    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight
        self.height = height
    @property
```

```
def bmi(self):
    return self.weight / (self.height * self.height)

albert=People('albert',75,1.80)
print(albert.bmi)
albert.weight = 70
print(albert.bmi)
```

伪装属性的修改与删除

```
class People:
    def __init__(self,name):
        self.__name = name
    @property
    def name(self):
        print('访问'.center(50,'='))
        return self.__name

p = People('albert')
print(p.name)
# p.name = 'ALBERT'
del p.name
```

伪装的属性能够查看，但同时也有修改和删除的需求，现在我们并不能实现，因为他本质是一个函数，修改如下：

```
class People:
    def __init__(self,name):
        self.__name = name
    @property
    def name(self):
        print('访问'.center(50,'='))
        return self.__name
    @name.setter
    def name(self,x):
        print('修改'.center(50,'='))
        self.__name = x
    @name.deleter
    def name(self):
        print('删除'.center(50,'='))
        # print('就不让你删')
        # 可添加任意的认证机制，认证过了还是不给删，就是玩他
        del self.__name
```

```
p = People('albert')
print(p.name)
# p.name = 'ALBERT'
# print(p.name)
# del p.name
# print(p.name)
```

其他开发者在调用name这个属性的时候，访问修改和删除分别进入的是不同的函数，但是他们并不知道，这一切都在我们的掌控之中了

三 多态

1. 多肽与多态性

多态通俗讲就是用一种事物的多种形态，水的形态有液态水，水蒸气和冰，但只要他是水，那么一个水分子就是由两个氢原子和一个氧原子构成的，菜的程序员和牛逼的程序员都是程序员，都会写代码，但是写出来的代码能一样吗？这在程序里怎么实现呢？

或许你已经猜到了，这就是从属关系，用继承来表示

```
class Animal:
    def eat(self):
        print('eat')
        pass
    def drink(self):
        pass
    def run(self):
        pass
    def bark(self):
        pass

class Cat(Animal):
    pass

class Dog(Animal):
    pass

class Pig(Animal):
    pass

c = Cat()
d = Dog()
p = Pig()
c.eat()
```

```
d.eat()  
p.eat()
```

只要他是动物，我不需要管他是什么动物，直接就可以调用动物的方法，并不需要单独考虑每个动物怎么吃

多态性：可以在不用考虑的对象具体类型的前提下，而直接使用对象下的方法

尽管如此，每个动物依然可以有自己的差异，就是使用派生

```
class Animal:  
    def eat(self):  
        print('eat')  
        pass  
    def drink(self):  
        pass  
    def run(self):  
        pass  
    def bark(self):  
        pass  
class Cat(Animal):  
    def bark(self):  
        print('喵喵叫')  
class Dog(Animal):  
    def bark(self):  
        print('汪汪叫')  
class Pig(Animal):  
    def bark(self):  
        print('哼哼叫')  
  
c = Cat()  
d = Dog()  
p = Pig()  
c.bark()  
d.bark()  
p.bark()
```

每个动物都有了自己特征，同时调用的时候依然不用做任何改变，如果每个人都能这样写代码，那该有多好，但是就是有一些野生的程序员，他不听话，还觉得自己英语挺好


```

class Cat(Animal):
    def speak(self):
        print('喵喵叫')
class Dog(Animal):
    def talk(self):
        print('汪汪叫')
class Pig(Animal):
    def say(self):
        print('哼哼叫')

c = Cat()
d = Dog()
p = Pig()
c.speak()
d.talk()
p.say()

```

这样虽然也能实现，但是你要记住每个动物的叫的方法，开发效率很低，试想这样一个场景，假如你现在就是架构师或者项目经理，你从程序设计的角度考虑把上面的动物类吃喝跑叫的四个方法定义好，想把它做成一个规范，剩下的每个动物的低级的代码你肯定是不不会写的，交给下面的人去写，但是他就是不用你定义的方法，不遵守你定的规范，写成了这样，你怎么办，是不是很绝望？

```

import abc # abstract class
class Animal(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def eat(self):
        pass
    @abc.abstractmethod
    def drink(self):
        pass
    @abc.abstractmethod
    def run(self):
        pass
    @abc.abstractmethod
    def bark(self):
        pass
class Cat(Animal):
    def speak(self):
        print('喵喵叫')
class Dog(Animal):
    def talk(self):
        print('汪汪叫')
class Pig(Animal):

```

```
def say(self):
    print('哼哼叫')
# obj=Animal() # 抽象基类本身不能被实例化
# c = Cat()
# d = Dog()
# p = Pig()
```

这样他就不能实例化了，那调用他自己的狗屁方法更是别想了

```
class Animal(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def eat(self):
        pass
    @abc.abstractmethod
    def drink(self):
        pass
    @abc.abstractmethod
    def run(self):
        pass
    @abc.abstractmethod
    def bark(self):
        pass
# obj=Animal() # 抽象基类本身不能被实例化
class Cat(Animal):
    def eat(self):
        print('cat eat')
    def drink(self):
        print('cat drink')
    def run(self):
        print('cat run')
    def bark(self):
        print('喵喵喵')
class Dog(Animal):
    def eat(self):
        print('dog eat')
    def drink(self):
        print('dog drink')
    def run(self):
        print('dog run')
    def bark(self):
        print('汪汪汪')
class Pig(Animal):
    def eat(self):
```

```

        print('pig eat')
    def drink(self):
        print('pig drink')
    def run(self):
        print('pig run')
    def bark(self):
        print('哼哼哼')

c = Cat()
d = Dog()
p = Pig()
c.bark()
d.bark()
p.bark()

```

除非他像上面这样写代码，注意：他写的代码要满足两个条件才可以，1 函数名一样，2 四个函数缺一不可

接下来我们再来升级一下

```

# 后面追加
def BARK(animal):
    animal.bark()

BARK(c)
BARK(d)
BARK(p)

```

现在就定义成了统一的接口，不管谁来调用都是一样的，这种思想你之前其实早就学过，只是你不知道这叫多态性

Python中一切皆对象

```

s='hello'
l=[1,2,3]
t=(4,5,6)
s.__len__()
l.__len__()
t.__len__()
# 我们自己定义
# def LEN(obj):
#     return obj.__len__()
#
# print(LEN(s))
# print(LEN(l))

```

```
# print(LEN(t))
print(len(l))  # Python内部封装
print(len(s))
print(len(t))
```

· 鸭子类型

再来试想一下，你现在又成了架构师了，经过你的调教，你手下干活的这帮野生程序员都老实了，现在又有了新的项目，你就不用费劲多写代码来限制他们了

上面我们用的多肽其实是用继承把类与类强耦合到了一起，Python这门语言的语法风格并不会真正的限制你，Python所推崇的并不是硬性的限制。在Python中真正的体现多态性并不是用这种方式实现，我们使用多态性就是为了规范，如果大家都能遵守，我们可以定义两个没有关系的类，约定好了使用同样的方法，也就是内部的规范都一样

```
class Foo:
    def f1(self):
        print('from foo.f1')
    def f2(self):
        print('from foo.f2')
class Bar:
    def f1(self):
        print('from bar.f1')
    def f2(self):
        print('from bar.f2')
obj1 = Foo()
obj2 = Bar()
obj1.f1()
obj1.f2()
obj2.f1()
obj2.f2()
```

两个类完全接耦合，没有硬性的限制，在Python中就推崇这种形式，这叫鸭子类型

只要你走起路来像鸭子，看着像鸭子，那你就是鸭子。这个名字很特殊，我以前有幸和微软的人在一起工作过，他们也会使用这种方式，英文就是Duck Type，很土，但是很好用。

再来说一个统一规范的例子，Linux操作系统里有一个核心的这几理念，一切皆文件，硬盘是文件，进程是文件（后面会学进程），文件本身也是文件，所有的一切都是文件

```
class Disk:
    def read(self):
```

```
print('disk read')
def write(self):
    print('disk write')
class Txt:
    def read(self):
        print('txt read')
    def write(self):
        print('txt write')
class Process:
    def read(self):
        print('process read')
    def write(self):
        print('process write')
obj1 = Disk()
obj2 = Txt()
obj3 = Process()
obj1.read()
obj2.read()
obj3.read()
```

只要长得像，不是也是，用这种方式，既解开了耦合，又统一了标准

类的绑定方法

复习：

绑定方法：

在类内部定义的函数，默认就是给对象来用，而且是绑定给对象用的，称为对象的绑定方法

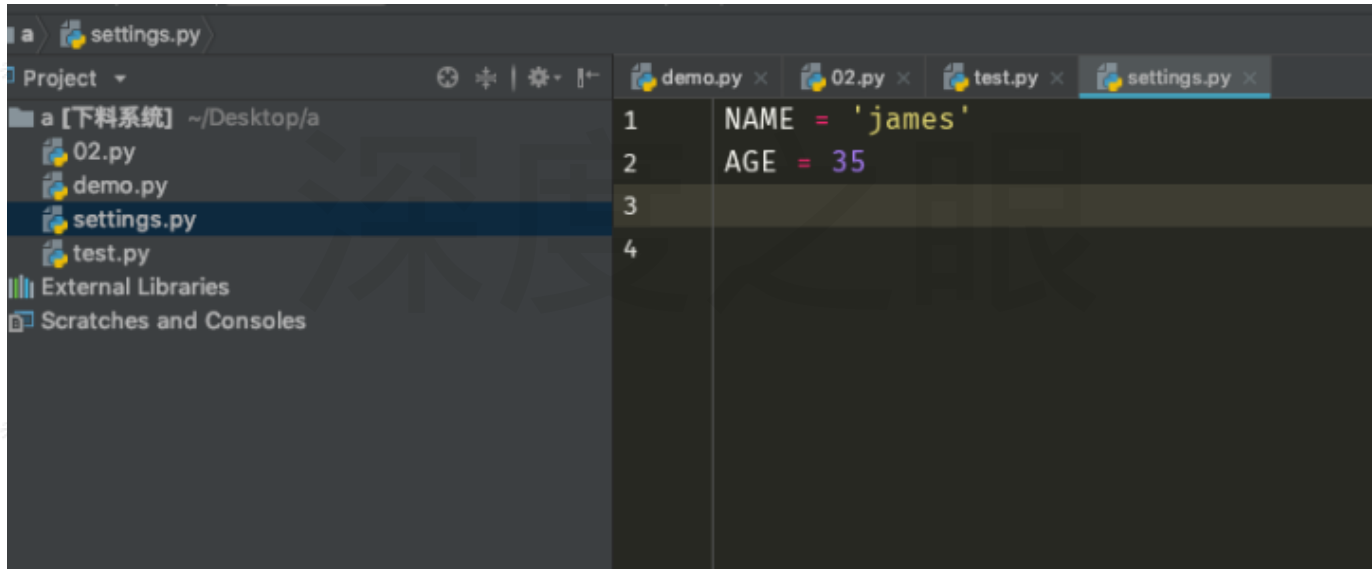
绑定对象的方法特殊之处：

应该由对象来调用，对象来调用，会自动将对象当作第一个参数传入

先来看一个熟悉的场景

```
class People:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def tell(self):
        print('%s %s' % (self.name, self.age))
p = People('albert', 18)
p.tell()
```

接下来我们不手动传值了，要从配置文件中读取



```
import settings
class People:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def tell(self):
        print('%s %s' % (self.name, self.age))
p = People(settings.NAME, settings.AGE)
p.tell()
```

如果我需要生成多个对象，每次都要从配置文件中读数据，我要不断重复的去看配置文件的东西，容易出错，我们来想一个更好的方法

```

251     import settings
252     class People:
253         def __init__(self, name, age):
254             self.name = name
255             self.age = age
256
257         def tell(self):
258             print('%s %s' %(self.name, self.age))
259
260         def read_from_conf(self):
261             return People(settings.NAME, settings.AGE)
262
263     p = People.read_from_conf()

```

现在用类来调一个方法，没有自动传值了，函数之所以需要一个参数是因为，他有一个Pycharm自动添加的self，而这个self，我们没有用到，那我们就把它去掉

```

251     import settings
252     class People:
253         def __init__(self, name, age):
254             self.name = name
255             self.age = age
256
257         def tell(self):
258             print('%s %s' %(self.name, self.age))
259
260         def read_from_conf():
261             return People(settings.NAME, settings.AGE)
262
263     p = People.read_from_conf()
264     p.tell()

```

这样没有问题了，但是我现在如果这个类改了名字，你现在就不能用了，所以你的代码261行写死了，所以我們还是需要传一个参数

深度之眼

```

251     import settings
252     class People:
253     def __init__(self, name, age):
254         self.name = name
255         self.age = age
256
257     def tell(self):
258         print('%s %s' %(self.name, self.age))
259
260     def read_from_conf(x): # 传一个x
261         return People(settings.NAME, settings.AGE)
262
263     p = People.read_from_conf(People) # 这需要一个类型
264     p.tell()

```

这样就写活了，但是很笨，代码263行是类来调用并且会把这个类当作第一参数传入，你想到了什么？和对象的绑定方法一样，那怎么解决呢？

```

251     import settings
252     class People:
253     def __init__(self, name, age):
254         self.name = name
255         self.age = age
256
257     def tell(self):
258         print('%s %s' %(self.name, self.age))
259
260     @classmethod
261     def read_from_conf(x): # 传一个x
262         return People(settings.NAME, settings.AGE)
263
264     p = People.read_from_conf() # 这不再需要传入
265     p.tell()

```

260行添加@classmethod方法后就会自动把类当作第一个参数传入，看以下两个截图来验证，这是不是类的绑定方法

深度之眼

作者：马一特

作者：马一特


```

260         @classmethod
261         def read_from_conf(x): # 传一个x
262             return People(settings.NAME, settings.AGE)
263
264     print(People.read_from_conf)

```

```

test x
/Users/albert/anaconda3/bin/python /Users/albert/Desktop/a/test.py
<bound method People.read_from_conf of <class '__main__.People'>>

```

再看下没有classmethod什么效果

```

260         # @classmethod
261         def read_from_conf(x): # 传一个x
262             return People(settings.NAME, settings.AGE)
263
264     print(People.read_from_conf)

```

People > read_from_conf()

```

test x
/Users/albert/anaconda3/bin/python /Users/albert/Desktop/a/test.py
<function People.read_from_conf at 0x109459730>

```

结论：

类的绑定方法：应该由类来调用，并会自动将类当作第一个参数传入

代码261行，参数叫x并不好，能让他表示类，class被用过了，就叫cls吧，你用Pytharm加上classmethod这个装饰器之后他就会给你自动添加了。

· 非绑定方法

依然是上面的代码，假如现在每一个用户都给他一个id号，添加这样一个功能，我们并不需要自动传值，也就是不需要任何的绑定方法，这应该怎么解决呢？

```

import settings
import hashlib
import time

```

```

class People:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def tell(self):
        print('%s %s' % (self.name, self.age))
    @classmethod
    def read_from_conf(cls):
        return People(settings.NAME, settings.AGE)
    def create_id(): # 这行不用传self, 我们手动去掉self, 定义成无参函数
        m = hashlib.md5()
        m.update(str(time.clock()).encode('utf-8'))
        return m.hexdigest()

p = People('albert', 18)
print(People.create_id())
print(People.create_id)
print(p.create_id)

```

输出:

```

/Users/albert/anaconda3/bin/python /Users/albert/Desktop/a/test.py
1462fb6e1d485e0d2c5be28f0f8eabee
<function People.create_id at 0x102744268>
<bound method People.create_id of <__main__.People object at 0x1009c6c18>>

```

从结果可知，虽然没有传参数，但是现在还是对象的绑定方法，但是绑定给对象并不合理，根本就不需要，同理，绑定给类也不合理，解决如下：

```

267     @staticmethod
268     def create_id():
269         m = hashlib.md5()
270         m.update(str(time.clock()).encode('utf-8'))
271         return m.hexdigest()
272
273
274     p = People('albert', 18)
275     print(People.create_id())
276     print(People.create_id)
277     print(p.create_id)
278

```

People > create_id()

```

/Users/albert/anaconda3/bin/python /Users/albert/Desktop/a/test.py
6ec7a50eaeda377599c70f004e0d75b3
<function People.create_id at 0x10d8e9268>
<function People.create_id at 0x10d8e9268>

```

Process finished with exit code 0

添加一个装饰器staticmethod全都变成普通方法，这叫非绑定方法，或者静态方法。

深度之眼

深度之眼

深度之眼