面向对象

语言的分类

面向机器

抽象成机器指令，机器容易理解

代表：汇编语言

面向过程

做一件事情，排出个步骤，第一步干什么，第二步干什么，如果出现情况A，做什么处理，如果出现了情况B，做

什么处理。

问题规模小，可以步骤化，按部就班处理。

代表：C语言

面向对象OOP

随着计算机需要解决的问题的规模扩大，情况越来越复杂。需要很多人、很多部门协作，面向过程编程不太适合

了。

代表：C++、Java、Python等

面向对象

什么是面向对象呢？

一种认识世界、分析世界的方法论。将万事万物抽象为各种对象。

类class

类是抽象的概念，是万事万物的抽象，是一类事物的共同特征的集合。

用计算机语言来描述类，就是属性和方法的集合。

对象instance、object

对象是类的具象，是一个实体。

对于我们每个人这个个体，都是抽象概念人类的不同的实体。

举例：

你吃鱼

你，就是对象；鱼，也是对象；吃就是动作

你是具体的人，是具体的对象。你属于人类，人类是个抽象的概念，是无数具体的人的个体的抽象。

鱼，也是具体的对象，就是你吃的这一条具体的鱼。这条鱼属于鱼类，鱼类是无数的鱼抽象出来的概念。

吃，是动作，也是操作，也是方法，这个吃是你的动作，也就是人类具有的方法。如果反过来，鱼吃人。吃就是鱼

类的动作了。

吃，这个动作，很多动物都具有的动作，人类和鱼类都属于动物类，而动物类是抽象的概念，是动物都有吃的动

作，但是吃法不同而已。

你驾驶车，这个车也是车类的具体的对象（实例），驾驶这个动作是鱼类不具有的，是人类具有的方法。

属性：它是对象状态的抽象，用数据结构来描述。

操作：它是对象行为的抽象，用操作名和实现该操作的方法来描述。

每个人都是人类的一个单独的实例，都有自己的名字、身高、体重等信息，这些信息是个人的属性，但是，这些信

息不能保存在人类中，因为它是抽象的概念，不能保留具体的值。

而人类的实例，是具体的人，他可以存储这些具体的属性，而且可以不同人有不同的属性。

哲学

一切皆对象

对象是数据和操作的封装

对象是独立的，但是对象之间可以相互作用

目前OOP是最接近人类认知的编程范式

面向对象3要素

1. 封装

组装：将数据和操作组装到一起。

隐藏数据：对外只暴露一些接口，通过接口访问对象。比如驾驶员使用汽车，不需要了解汽车的构造细

节，只需要知道使用什么部件怎么驾驶就行，踩了油门就能跑，可以不了解其中的机动原理。

2. 继承

多复用，继承来的就不用自己写了

多继承少修改，OCP（Open-closed Principle），使用继承来改变，来体现个性

3. 多态

面向对象编程最灵活的地方，动态绑定

人类就是封装；

人类继承自动物类，孩子继承父母特征。分为单一继承、多继承；

多态，继承自动物类的人类、猫类的操作”吃“不同。

Python的类

定义

class  ClassName:

语句块

1. 必须使用class关键字

2. 类名必须是用大驼峰命名

3. 类定义完成后，就产生了一个类对象，绑定到了标识符ClassName上

举例

class  MyClass:

"""A  example  class"""

x  =  'abc'  #  类属性

def  foo(self):  #  类属性foo，也是方法

return  'My  Class'

print(MyClass.x)

print(MyClass.foo)

print(MyClass.\_\_doc\_\_)

类对象及类属性

类对象，类的定义执行后会生成一个类对象

类的属性，类定义中的变量和类中定义的方法都是类的属性

类变量，上例中x是类MyClass的变量

MyClass中，x、foo都是类的属性，  \_\_doc\_\_  也是类的特殊属性

foo方法是类的属性，如同吃是人类的方法，但是每一个具体的人才能吃东西，也就是说吃是人的实例调用的方

法。

foo是方法method，本质上就是普通的函数对象function，它一般要求至少有一个参数。第一个形式参数可以是

self（self只是个惯用标识符，可以换名字），这个参数位置就留给了self。

self 指代当前实例本身

问题

上例中，类是谁？实例是谁？

实例化

a  =  MyClass()  #  实例化

使用上面的语法，在类对象名称后面加上一个括号，就调用类的实例化方法，完成实例化。

实例化就真正创建一个该类的对象（实例）。例如

tom  =  Person()

jerry  =  Person()

上面的tom、jerry都是Person类的实例，通过实例化生成了2个实例。

每次实例化后获得的实例，是不同的实例，即使是使用同样的参数实例化，也得到不一样的对象。

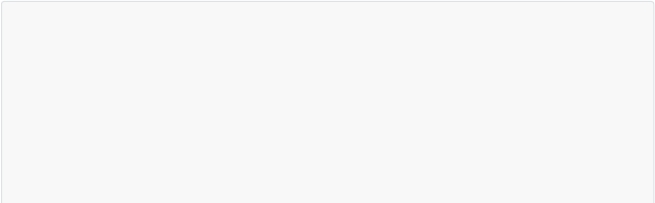
Python类实例化后，会自动调用  \_\_init\_\_  方法。这个方法第一个形式参数必须留给self，其它参数随意。

\_\_init\_\_方法

MyClass()实际上调用的是  \_\_init\_\_(self)  方法，可以不定义，如果没有定义会在实例化后隐式调用。

作用：对实例进行初始化

class  MyClass:



def  \_\_init\_\_(self):

print('init')

print(MyClass)  #  不会调用

print(MyClass())  #  调用\_\_init\_\_

a  =  MyClass()  #  调用\_\_init\_\_

初始化函数可以多个参数，请注意第一个位置必须是self，例如  \_\_init\_\_(self,  name,  age)

class  Person:

def  \_\_init\_\_(self,  name,  age):

self.name  =  name

self.age  =  age

def  showage(self):

print('{}  is  {}'.format(self.name,  self.age))

tom  =  Person('Tom',  20)  #  实例化，会调用\_\_init\_\_方法并为实例进行属性的初始化

jerry  =  Person('Je',  25)

print(tom.name,  jerry.age)

jerry.age  +=  1

print(jerry.age)

jerry.showage()

注意：  \_\_init\_\_()  方法不能有返回值，也就是只能是return None

实例对象instance

类实例化后一定会获得一个类的实例，就是实例对象。

上例中的tom、jerry就是Person类的实例。

\_\_init\_\_  方法的第一参数 self 就是指代某一个实例自身。

类实例化后，得到一个实例对象，调用方法时采用jerry.showage()的方式，实例对象会绑定到方法上。

但是该函数签名是showage(self)，少传一个实际参数self吗？

这个self就是jerry，jerry.showage()调用时，会把方法的调用者jerry实例作为第一参数self的实参传入。

self.name就是jerry对象的name，name是保存在了jerry对象上，而不是Person类上。所以，称为实例变量。

self

class  MyClass:

def  \_\_init\_\_(self):

print(1,  'self  in  init  =  {}'.format(id(self)))

def  showself(self):

print(2,  'self  in  showself()  =  {}'.format(id(self)))

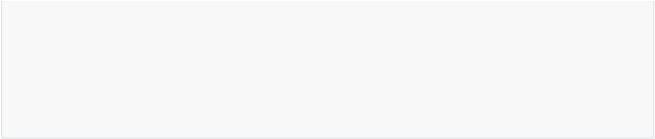
c  =  MyClass()  #  会调用\_\_init\_\_

print(3,  'c  =  {}'.format(id(c)))

print('-'  \*  30)

|  |  |
| --- | --- |
| 特殊属性 | 含义 |
| \_\_name\_\_ | 对象名 |
| \_\_class\_\_ | 对象的类型 |
| \_\_dict\_\_ | 对象的属性的字典 |
| \_\_qualname\_\_ | 类的限定名 |

c.showself()



#  打印结果为

1  self  in  init  =  2537636858512

3  c  =  2537636858512

------------------------------

2  self  in  showself()  =  2537636858512

上例说明，self就是调用者，就是c对应的实例对象。

self这个名字只是一个惯例，它可以修改，但是请不要修改，否则影响代码的可读性

看打印的结果，思考一下执行的顺序，为什么？

实例变量和类变量

class  Person:

age  =  3

def  \_\_init\_\_(self,  name):

self.name  =  name

#tom  =  Person('Tom',  20)  #

tom  =  Person('Tom')  #  实例化、初始化

jerry  =  Person('Jerry')

print(tom.name,  tom.age)

print(jerry.name,  jerry.age)

print(Person.age)

#print(Person.name)  #

Person.age  =  30

print(Person.age,  tom.age,  jerry.age)

#  运行结果

Tom  3

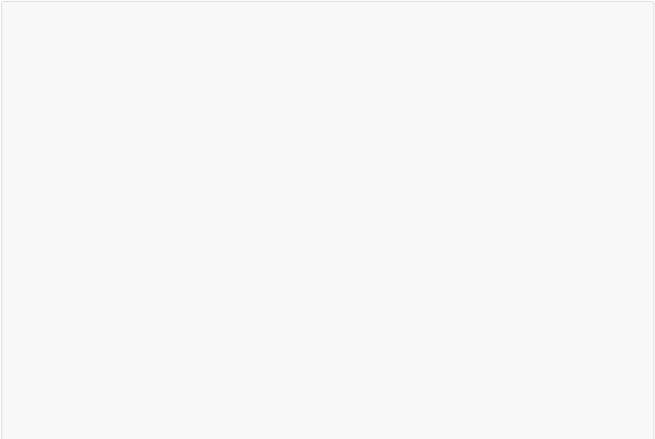
Jerry  3

3

30  30  30

实例变量是每一个实例自己的变量，是自己独有的；类变量是类的变量，是类的所有实例共享的属性和方法

注意：



Python中每一种对象都拥有不同的属性。函数、类都是对象，类的实例也是对象。

举例

class  Person:

age  =  3

def  \_\_init\_\_(self,  name):

self.name  =  name

print('----class----')

print(Person.\_\_class\_\_)

print(sorted(Person.\_\_dict\_\_.items()),  end='\n\n')  #  属性字典

tom  =  Person('Tom')

print('----instance  tom----')

print(tom.\_\_class\_\_)

print(sorted(tom.\_\_dict\_\_.items()),  end='\n\n')

print("----tom's  class----")

print(tom.\_\_class\_\_.\_\_name\_\_)

print(sorted(tom.\_\_class\_\_.\_\_dict\_\_.items()),  end='\n\n')

上例中，可以看到类属性保存在类的  \_\_dict\_\_  中，实例属性保存在实例的  \_\_dict\_\_  中，如果从实例访问类的属

性，也可以借助  \_\_class\_\_  找到所属的类，再通过类来访问类属性，例如  tom.\_\_class\_\_.age  。

有了上面知识，再看下面的代码

class  Person:

age  =  3

height  =  170

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.age  =  age

tom  =  Person('Tom')  #  实例化、初始化

jerry  =  Person('Jerry',  20)

Person.age  =  30

print(1,  Person.age,  tom.age,  jerry.age)  #  输出什么结果

print(2,  Person.height,  tom.height,  jerry.height)  #  输出什么结果

jerry.height  =  175

print(3,  Person.height,  tom.height,  jerry.height)  #  输出什么结果

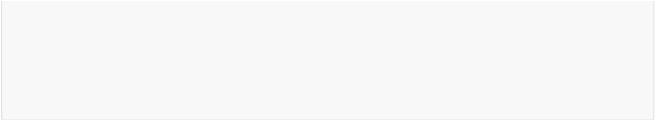
tom.height  +=  10

print(4,  Person.height,  tom.height,  jerry.height)  #  输出什么结果

Person.height  +=  15

print(5,  Person.height,  tom.height,  jerry.height)  #  输出什么结果

Person.weight  =  70



print(6,  Person.weight,  tom.weight,  jerry.weight)  #  输出什么结果

print(7,  tom.\_\_dict\_\_['height'])  #  可以吗

print(8,  tom.\_\_dict\_\_['weight'])  #  可以吗

总结

是类的，也是这个类所有实例的，其实例都可以访问到；

是实例的，就是这个实例自己的，通过类访问不到。

类变量是属于类的变量，这个类的所有实例可以共享这个变量。

对象（实例或类）可以动态的给自己增加一个属性（赋值即定义一个新属性）。

实例.\_\_dict\_\_[变量名]  和  实例.变量名  都可以访问到实例自己的属性。

实例的同名变量会隐藏掉类变量，或者说是覆盖了这个类变量。但是注意类变量还在那里，并没有真正被覆盖。

实例属性的查找顺序

指的是实例使用  .点号  来访问属性，会先找自己的  \_\_dict\_\_  ，如果没有，然后通过属性  \_\_class\_\_  找到自己的

类，再去类的  \_\_dict\_\_  中找

注意，如果实例使用  \_\_dict\_\_[变量名]  访问变量，将不会按照上面的查找顺序找变量了，这是指明使用字典的key

查找，不是属性查找。

一般来说，类变量可使用全大写来命名。

装饰一个类

回顾，什么是高阶函数？什么是装饰器函数？

思考，如何装饰一个类？

需求，为一个类通过装饰，增加一些类属性。例如能否给一个类增加一个NAME类属性并提供属性值

#  增加类变量

def  add\_name(name,  cls):

cls.NAME  =  name  #  动态增加类属性

#  改进成装饰器

def  add\_name(name):

def  wrapper(cls):

cls.NAME  =  name

return  cls

return  wrapper

@add\_name('Tom')

class  Person:

AGE  =  3

print(Person.NAME)

之所以能够装饰，本质上是为类对象动态的添加了一个属性，而Person这个标识符指向这个类对象。

类方法和静态方法

前面的例子中定义的  \_\_init\_\_  等方法，这些方法本身都是类的属性，第一个参数必须是self，而self必须指向一个

对象，也就是类实例化之后，由实例来调用这个方法。

普通函数

class  Person:

def  normal\_method():  #  可以吗？

print('normal')

#  如何调用

Person.normal\_method()  #  可以吗？

Person().normal\_method()  #  可以吗？

print(Person.\_\_dict\_\_)

Person.normal\_method()

可以放在类中定义，因为这个方法只是被Person这个名词空间管理的一个普通的方法，normal\_method是Person

的一个属性而已。

由于normal\_method在定义的时候没有指定self，所以不能完成实例对象的绑定，不能用

Person().normal\_method()调用。

注意：虽然语法是对的，但是，没有人这么用，也就是说禁止这么写

类方法

class  Person:

@classmethod

def  class\_method(cls):  #  cls是什么

print('class  =  {0.\_\_name\_\_}  ({0})'.format(cls))

cls.HEIGHT  =  170

Person.class\_method()

print(Person.\_\_dict\_\_)

类方法

1. 在类定义中，使用@classmethod装饰器修饰的方法

2. 必须至少有一个参数，且第一个参数留给了cls，cls指代调用者即类对象自身

3. cls这个标识符可以是任意合法名称，但是为了易读，请不要修改

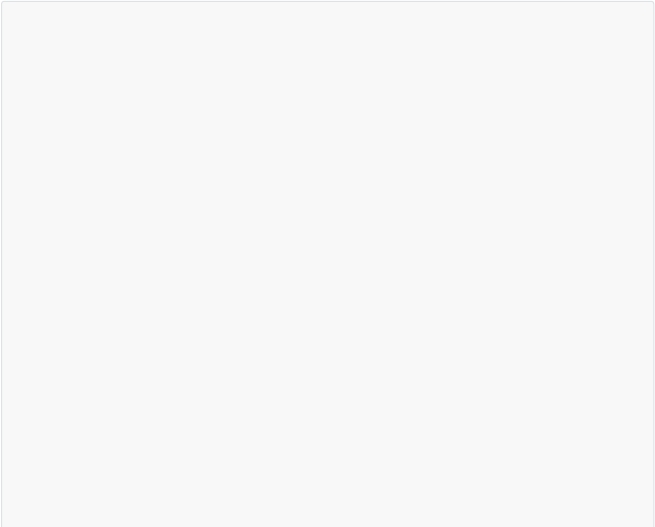
4. 通过cls可以直接操作类的属性

注意：无法通过cls操作类的实例。为什么？

类方法，类似于C++、Java中的静态方法

静态方法

class  Person:



@classmethod

def  class\_method(cls):  #  cls是什么

print('class  =  {0.\_\_name\_\_}  ({0})'.format(cls))

cls.HEIGHT  =  170

@staticmethod

def  static\_methd():

print(Person.HEIGHT)

Person.class\_method()

Person.static\_methd()

print(Person.\_\_dict\_\_)

静态方法

1. 在类定义中，使用@staticmethod装饰器修饰的方法

2. 调用时，不会隐式的传入参数

静态方法，只是表明这个方法属于这个名词空间。函数归在一起，方便组织管理。

方法的调用

类可以定义这么多种方法，究竟如何调用它们？

class  Person:

def  normal\_method():

print('normal')

def  method(self):

print("{}'s  method".format(self))

@classmethod

def  class\_method(cls):  #  cls是什么

print('class  =  {0.\_\_name\_\_}  ({0})'.format(cls))

cls.HEIGHT  =  170

@staticmethod

def  static\_methd():

print(Person.HEIGHT)

print('~~~~类访问')

print(1,  Person.normal\_method())  #  可以吗

print(2,  Person.method())  #  可以吗

print(3,  Person.class\_method())  #  可以吗

print(4,  Person.static\_methd())  #  可以吗

print(Person.\_\_dict\_\_)

print('~~~~实例访问')

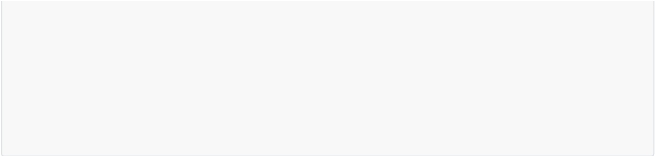
print('tom----')

tom  =  Person()

print(1,  tom.normal\_method())  #  可以吗

print(2,  tom.method())  #  可以吗

print(3,  tom.class\_method())  #  可以吗？



print(4,  tom.static\_methd())  #  可以吗

print('jerry----')

jerry  =  Person()

print(1,  jerry.normal\_method())  #  可以吗

print(2,  jerry.method())  #  可以吗

print(3,  jerry.class\_method())  #  可以吗？

print(4,  jerry.static\_methd())  #  可以吗

类几乎可以调用所有内部定义的方法，但是调用  普通的方法  时会报错，原因是第一参数必须是类的实例。

实例也几乎可以调用所有的方法，  普通的函数  的调用一般不可能出现，因为不允许这么定义。

总结：

类除了普通方法都可以调用，普通方法需要对象的实例作为第一参数。

实例可以调用所有类中定义的方法（包括类方法、静态方法），普通方法传入实例自身，静态方法和类方法需要找

到实例的类。

访问控制

私有（Private）属性

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.age  =  age

def  growup(self,  i=1):

if  i  >  0  and  i  <  150:  #  控制逻辑

self.age  +=  i

p1  =  Person('tom')

p1.growup(20)  #  正常的范围

p1.age  =  160  #  超过了范围，并绕过了控制逻辑

print(p1.age)

上例，本来是想通过方法控制属性，但是由于属性在外部可以访问，或者说可见，就可以直接绕过方法，直接修改

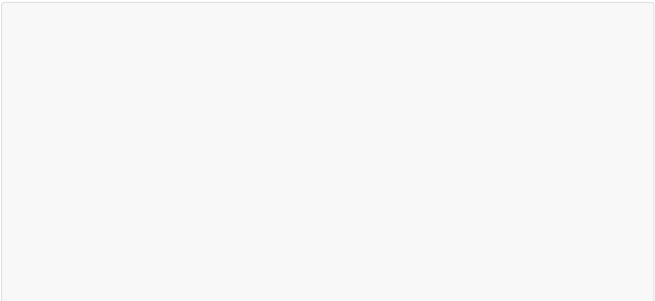
这个属性。

Python提供了私有属性可以解决这个问题。

私有属性

使用双下划线开头的属性名，就是私有属性

class  Person:



def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

def  growup(self,  i=1):

if  i  >  0  and  i  <  150:  #  控制逻辑

self.\_\_age  +=  i

p1  =  Person('tom')

p1.growup(20)  #  正常的范围

print(p1.\_\_age)  #  可以吗

通过实验可以看出，外部已经访问不到  \_\_age  了，age根本就没有定义，更是访问不到。

那么，如何访问这个私有变量  \_\_age  呢？

使用方法来访问

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

def  growup(self,  i=1):

if  i  >  0  and  i  <  150:  #  控制逻辑

self.\_\_age  +=  i

def  getage(self):

return  self.\_\_age

print(Person('tom').getage())

私有变量的本质

外部访问不到，能够动态增加一个  \_\_age  吗？

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

def  growup(self,  i=1):

if  i  >  0  and  i  <  150:  #  控制逻辑

self.\_\_age  +=  i

def  getage(self):

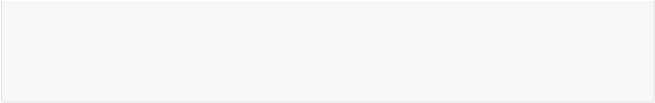
return  self.\_\_age

p1  =  Person('tom')

p1.growup(20)  #  正常的范围

#print(p1.\_\_age)  #  访问不到

p1.\_\_age  =  28



print(p1.\_\_age)

print(p1.getage())

#  为什么年龄不一样？\_\_age没有覆盖吗？

print(p1.\_\_dict\_\_)

秘密都在  \_\_dict\_\_  中，里面是{'\_\_age': 28, '\_Person\_\_age': 38, 'name': 'tom'}

私有变量的本质：

类定义的时候，如果声明一个实例变量的时候，使用双下划线，Python解释器会将其改名，转换名称为 \_类名\_\_变

量名 的名称，所以用原来的名字访问不到了。

知道了这个名字，能否直接修改呢？

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

def  growup(self,  i=1):

if  i  >  0  and  i  <  150:  #  控制逻辑

self.\_\_age  +=  i

def  getage(self):

return  self.\_\_age

p1  =  Person('tom')

p1.growup(20)  #  正常的范围

#print(p1.\_\_age)  #  访问不到

p1.\_\_age  =  28

print(p1.\_\_age)

print(p1.getage())

#  为什么年龄不一样？\_\_age没有覆盖吗？

print(p1.\_\_dict\_\_)

#  直接修改私有变量

p1.\_Person\_\_age  =  15

print(p1.getage())

print(p1.\_\_dict\_\_)

从上例可以看出，知道了私有变量的新名称，就可以直接从外部访问到，并可以修改它。

保护变量

在变量名前使用一个下划线，称为保护变量。

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_age  =  age

tom  =  Person('Tom')

print(tom.\_age)

print(tom.\_\_dict\_\_)

可以看出，这个\_age属性根本就没有改变名称，和普通的属性一样，解释器不做任何特殊处理。

这只是开发者共同的约定，看见这种变量，就如同私有变量，不要直接使用。

私有方法

参照保护变量、私有变量，使用单下划线、双下划线命名方法。

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_age  =  age

def  \_getname(self):

return  self.name

def  \_\_getage(self):

return  self.\_age

tom  =  Person('Tom')

print(tom.\_getname())  #  没改名

print(tom.\_\_getage())  #  无此属性

print(tom.\_\_dict\_\_)

print(tom.\_\_class\_\_.\_\_dict\_\_)

print(tom.\_Person\_\_getage())  #  改名了

私有方法的本质

单下划线的方法只是开发者之间的约定，解释器不做任何改变。

双下划线的方法，是私有方法，解释器会改名，改名策略和私有变量相同， \_类名\_\_方法名 。

方法变量都在类的  \_\_dict\_\_  中可以找到。

私有成员的总结

在Python中使用 \_单下划线 或者 \_\_ 双下划线来标识一个成员被保护或者被私有化隐藏起来。

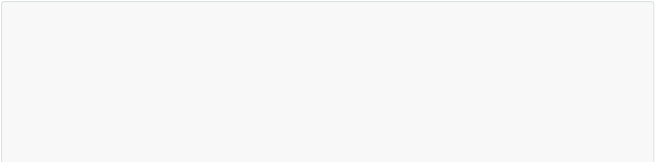
但是，不管使用什么样的访问控制，都不能真正的阻止用户修改类的成员。Python中没有绝对的安全的保护成员

或者私有成员。

因此，前导的下划线只是一种警告或者提醒，请遵守这个约定。除非真有必要，不要修改或者使用保护成员或者私

有成员，更不要修改它们。

补丁



可以通过修改或者替换类的成员。使用者调用的方式没有改变，但是，类提供的功能可能已经改变了。

猴子补丁（Monkey Patch）：

在运行时，对属性、方法、函数等进行动态替换。

其目的往往是为了通过替换、修改来增强、扩展原有代码的能力。

黑魔法，慎用。

#  test1.py

from  test2  import  Person

from  test3  import  get\_score

def  monkeypatch4Person():

Person.get\_score  =  get\_score

monkeypatch4Person()  #  打补丁

if  \_\_name\_\_  ==  "\_\_main\_\_":

print(Person().get\_score())

#  test2.py

class  Person:

def  get\_score(self):

#  connect  to  mysql

ret  =  {'English':78,  'Chinese':86,  'History':82}

return  ret

#  test3.py

def  get\_score(self):

return  dict(name=self.\_\_class\_\_.\_\_name\_\_,English=88,  Chinese=90,  History=85)

上例中，假设Person类get\_score方法是从数据库拿数据，但是测试的时候，不方便。

为了测试时方便，使用猴子补丁，替换了get\_score方法，返回模拟的数据。

属性装饰器

一般好的设计是：把实例的某些属性保护起来，不让外部直接访问，外部使用getter读取属性和setter方法设置属

性。

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

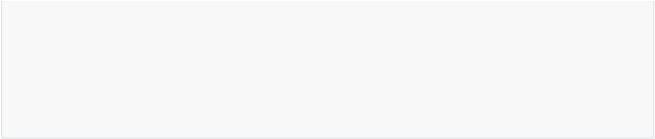
self.name  =  name

self.\_\_age  =  age

def  age(self):

return  self.\_\_age

def  set\_age(self,  age):



self.\_\_age  =  age

tom  =  Person('Tom')

print(tom.age())

tom.set\_age(20)

print(tom.age())

通过age和set\_age方法操作属性。

有没有简单的方式呢？

Python提供了属性property装饰器。

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

@property

def  age(self):

return  self.\_\_age

@age.setter

def  age(self,  age):

self.\_\_age  =  age

@age.deleter

def  age(self):

#  del  self.\_\_age

print('del')

tom  =  Person('Tom')

print(tom.age)

tom.age  =  20

print(tom.age)

del  tom.age

特别注意：使用property装饰器的时候这三个方法同名

property装饰器

后面跟的函数名就是以后的属性名。它就是getter。这个必须有，有了它至少是只读属性

setter装饰器

与属性名同名，且接收2个参数，第一个是self，第二个是将要赋值的值。有了它，属性可写

deleter装饰器

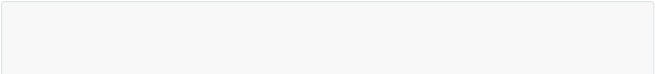
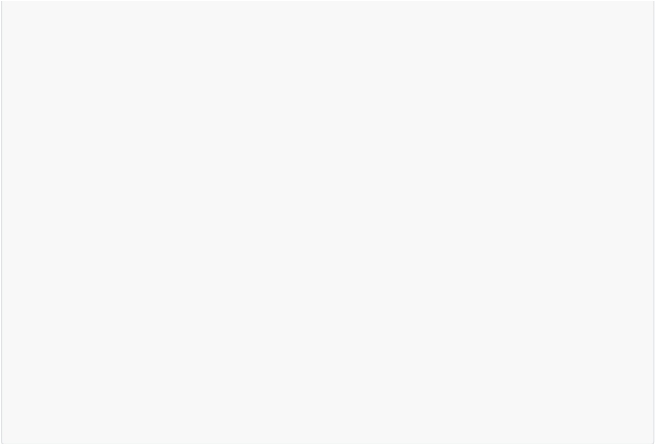
可以控制是否删除属性。很少用

property装饰器必须在前，setter、deleter装饰器在后。

property装饰器能通过简单的方式，把对方法的操作变成对属性的访问，并起到了一定隐藏效果

其它的写法

class  Person:



def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

def  getage(self):

return  self.\_\_age

def  setage(self,  age):

self.\_\_age  =  age

def  delage(self):

#  del  self.\_\_age

print('del')

age  =  property(getage,  setage,  delage,  'age  property')

tom  =  Person('Tom')

print(tom.age)

tom.age  =  20

print(tom.age)

del  tom.age

还可以如下

class  Person:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.\_\_age  =  age

age  =  property(lambda  self:self.\_\_age)

tom  =  Person('Tom')

print(tom.age)

对象的销毁

类中可以定义   \_\_del\_\_   方法，称为析构函数（方法）。

作用：销毁类的实例的时候调用，以释放占用的资源。其中就放些清理资源的代码，比如释放连接。

注意这个方法不能引起对象的真正销毁，只是对象销毁的时候会自动调用它。

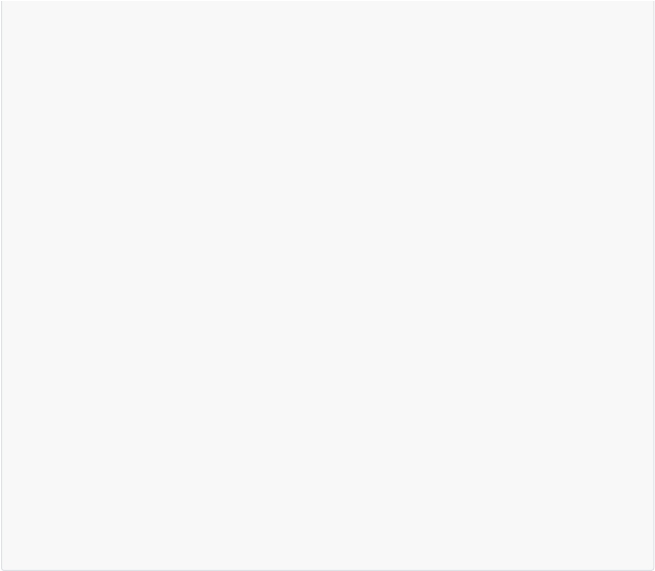
使用del语句删除实例，引用计数减1。当引用计数为0时，会自动调用  \_\_del\_\_   方法。

由于Python实现了垃圾回收机制，不能确定对象何时执行垃圾回收。

import  time

class  Person:

def  \_\_init\_\_(self,  name,  age=18):



self.name  =  name

self.\_\_age  =  age

def  \_\_del\_\_(self):

print('delete  {}'.format(self.name))

def  test():

tom  =  Person('tom')

tom.\_\_del\_\_()  #  手动调用

tom.\_\_del\_\_()

tom.\_\_del\_\_()

tom.\_\_del\_\_()

print('======start======')

tom2  =  tom

tom3  =  tom2

print(1,  'del')

del  tom

time.sleep(3)

print(2,  'del')

del  tom2

time.sleep(3)

print('~~~~~~~~~')

del  tom3  #  注释一下看看效果

time.sleep(3)

print('=======end')

test()

由于垃圾回收对象销毁时，才会真正清理对象，还会在回收对象之前自动调用  \_\_del\_\_   方法，除非你明确知道自

己的目的，建议不要手动调用这个方法。

方法重载(overload)

其他面向对象的高级语言中，会有重载的概念。

所谓重载，就是同一个方法名，但是参数数量、类型不一样，就是同一个方法的重载。

Python没有重载！

Python不需要重载！

Python中，方法（函数）定义中，形参非常灵活，不需要指定类型（就算指定了也只是一个说明而非约束），参

数个数也不固定（可变参数）。一个函数的定义可以实现很多种不同形式实参的调用。所以Python不需要方法的

重载。

或者说Python本身就实现了其它语言的重载。

封装

面向对象的三要素之一，封装Encapsulation

封装

将数据和操作组织到类中，即属性和方法

将数据隐藏起来，给使用者提供操作（方法）。使用者通过操作就可以获取或者修改数据。getter和setter。

通过访问控制，暴露适当的数据和操作给用户，该隐藏的隐藏起来，例如保护成员或私有成员。

练习

1、随机整数生成类

可以指定一批生成的个数，可以指定数值的范围，可以调整每批生成数字的个数

2、打印坐标

使用上题中的类，随机生成20个数字，两两配对形成二维坐标系的坐标，把这些坐标组织起来，并打印输出

3、车辆信息

记录车的品牌mark、颜色color、价格price、速度speed等特征，并实现车辆管理，能增加车辆、显示全部车辆的

信息功能

4、实现温度的处理

实现华氏温度和摄氏温度的转换。

℃  =  5  ×  (℉  -  32)  /  9

℉  =  9  ×  ℃  /  5  +  32

完成以上转换后，增加与开氏温度的转换，K  =  ℃  +  273.15

5、模拟购物车购物