**python3--面向对象的三大特性:封装，property，classmethod，staticmethod**

[用户2398817](https://cloud.tencent.com/developer/user/2398817)发表于[python3](https://cloud.tencent.com/developer/column/4155)订阅

65

python中的封装

隐藏对象的属性和实现细节，仅对外提供公共访问方式

好处：

1 将变化隔离

2 便于使用

3 提供复用性

4 提高安全性

封装原则

1 将不需要对外提供的内容都隐藏起来

2 把属性都隐藏，提供公共方法对其访问

私有变量和私有方法

在python中用双下划开头的方式将属性隐藏来(设置成私有的)

函数和属性装到了一个非全局的命名空间--封装

私有变量,错误示例

class A:

\_\_N = 'aaa' # 静态变量

print(A.\_\_N)

执行报错

AttributeError: type object 'A' has no attribute '\_\_N'

这个\_\_N就是类A的私有属性

定义一个私有的名字:就是在私有的名字前面加两条下划线\_\_N = 'aaa',所谓私有，就是不能在类的外面去引用它

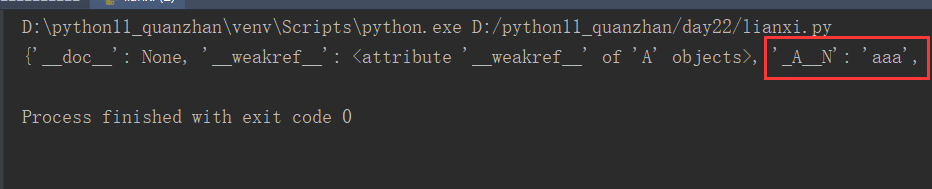
class A:

\_\_N = 'aaa' # 静态变量

def func(self):

print(A.\_\_N) # 在类的内部使用正常

print(A.\_\_dict\_\_)



可以看到类属性多了一个'\_A\_\_N':'aaa'的属性，其实这个\_A\_\_N 就是\_\_N,这是python解释器自动做的一个变形操作

类中所有双下划线开头的名称如\_\_x都会自动变形成:\_类名\_\_x的形式

例子

class A:

\_\_N = 0 # 类的数据属性就应该是共享的，但是语法上是可以把类的数据属性设置成私有的如\_\_N,会变形为\_A\_\_N

def \_\_init\_\_(self):

self.\_\_X = 10 # 变形为self.\_A\_\_X

def \_\_foo(self): # 变形为\_A\_\_foo

print('from A')

def bar(self):

self.\_\_foo() # 只有在类内部才可以通过\_\_foo的形式访问到

print(A.\_A\_\_N) # 是可以访问到的，即这种操作并不是严格意义上的限制外部访问，仅仅只是一种语法意义上的变形

a = A()

print(a.\_A\_\_X) # 是可以访问到的，即这种操作并不是严格意义上的限制外部访问，仅仅只是一种语法意义上的变形

a.bar()

执行结果

0

10

from A

一个私有的名字，在存储的过程中仍然会出现在A.\_\_dict\_\_中，所以我们仍然可以调用到，

python对其的名字进行了修改:\_类名\_\_名字

只不过在类的外部调用：需要\_类名\_\_名字去使用

在类的内部可以正常的使用名字

在类里面，只要你的代码遇到\_\_名字,就会被python解释器自动的转换成\_类名\_\_名字

虽然上面的方式可以访问到私有的，但是却不建议这么做，约定俗成

私有的属性

示例

class B:

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

self.\_\_name = name

def func(self):

print('in func: {}'.format(self.\_\_name)) # 外部通过调用func来获取类的私有属性

b = B('Sam')

b.func()

执行结果

in func: Sam

私有的方法

示例

class C:

def \_\_wahaha(self):

print('wahaha')

def ADCa(self): # 外部通过调用ADCa()方法，来执行类的私有方法\_\_wahaha()

self.\_\_wahaha()

c = C()

c.ADCa()

执行结果

wahaha

在类中，静态属性，方法，对象属性都可以变成私有的，只需要在这些名字之前加上\_\_

例3

class F:

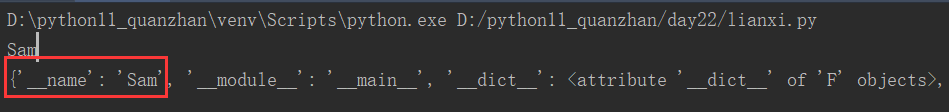
pass

F.\_\_name = 'Sam' # 误区，不是在创建私有属性

print(F.\_\_name)

print(F.\_\_dict\_\_)

执行结果



可以很明确的看到\_\_name并没有发生变形，变形只在类的内部发生，在类的外面创建的双下划线的属性，并不是私有属性

面试题：

1 下面代码执行的结果是什么？为什么？

class D:

def \_\_func(self):

print('in func')

class E(D):

def \_\_init\_\_(self):

self.\_\_func()

e = E()

解答：

class D:

def \_\_func(self): # 在类的内部遇到\_\_，python解释器会自动变形成\_D\_\_func

print('in func')

class E(D):

def \_\_init\_\_(self):

self.\_\_func()

# 在类的内部遇到\_\_,python解释器会自动变形成\_E\_\_func

# 实例化一个对象e,首先会找到E类中的\_\_init\_\_方法

# 遇到了\_\_，所以会去找self.\_E\_\_func()方法，子类E中没有找到

# 就去父类中找，而父类中的\_\_func变形成\_D\_\_func，而执行的是\_E\_\_func，所以会报错

e = E()

结果报错

AttributeError: 'E' object has no attribute '\_E\_\_func'

2 下面代码执行的结果是什么?为什么？

class D:

def \_\_init\_\_(self):

self.\_\_func()

def \_\_func(self):

print('in D')

class E(D):

def \_\_func(self):

print('in E')

e = E()

解答

class D: # 第一步加载类D

def \_\_init\_\_(self): # 第二步加载\_\_init\_\_ #第七步,由于子类没有init方法，所以找父类的

self.\_\_func() # 第八步执行self.\_\_func()，也就是\_D\_\_func

def \_\_func(self): # 第三步加载\_D\_\_func，双下划线变形 # 第九步，执行\_D\_\_func()

print('in D') # 第10步打印 'in D'

class E(D): # 第四步加载类E

def \_\_func(self): # 第五步加载\_E\_\_func,双下划线变形

print('in E')

e = E() # 第六步，实例化一个对象e ，

执行结果

in D

java中的对比

public  共有的   在类的内部可以使用，子类可以使用，外部可以使用     python类中所有的常规名字

protect 保护的   在类的内部可以使用，子类可以使用，外部不可以使用   python中没有

private 私有的   只能在类的内部使用，子类和外部都不可以使用         python中的\_\_名字

私有的用法

当一个方法不想被子类继承的时候

有些熟悉或者方法不希望从外部被调用，只想提供给内部的方法使用

示例1

描述一个房子，单价，面积，长宽高

class Room:

def \_\_init\_\_(self, name, price, length, width, height): # 名字,价格,长,宽,高

self.name = name

self.price = price

self.\_\_length = length # 私有属性长

self.\_\_width = width # 私有属性宽

self.\_\_height = height # 私有属性高

def area(self):

return self.\_\_length \* self.\_\_width

r = Room('张三', 240000, 11, 5, 2)

print('姓名:{},房子价格:{},面积:{}'.format(r.name, r.price, r.area()))

执行结果

姓名:张三,房子价格:240000,面积:55

示例2

用户输入账号名和密码，把密码转换为对应的ascii,最后打印出账号名和转换后的ascii值

class Person:

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

self.name = name

self.\_\_pwd = pwd

def \_\_showpwd(self):

s = []

for i in self.\_\_pwd:

s.append(str(ord(i)))

s2 = ''.join(s)

return s2

p = Person('zhangsan', '12345')

print('账号名为:{},加密后的密码为:{}'.format(p.name, p.\_Person\_\_showpwd()))

执行结果

账号名为:zhangsan,加密后的密码为:4950515253

property方法

将一个方法伪装成一个属性

1 并不会让你的代码有什么逻辑上的提高

2 只是从调用者的角度上换了一种方式，使之看起来更合理

示例

人体BMI指数

体质指数(BMI)=体重(kg) / 身高\*\*2(m)

写一个类，描述人体BMI指数

class Person:

def \_\_init\_\_(self, name, weight, height):

self.name = name

self.\_\_height = height

self.\_\_weight = weight

def cal\_BMI(self):

return self.\_\_weight / self.\_\_height \*\* 2

@property

def bmi(self):

return self.\_\_weight / self.\_\_height \*\* 2

P = Person('张三', 67, 1.71)

print(P.bmi)

执行结果

22.91303307000445

在一个类加载的过程中，会先加载这个类中的名字，包括被property装饰的

在实例化对象的时候，python解释器会先到类的空间里看看有没有这个被装饰的属性，如果有就不能在自己对象的空间中创建这个属性了

示例

计算圆形类的，面积，周长，将方法伪装成属性，方法中一般涉及的都是一些计算过程

from math import pi

class Circle:

def \_\_init\_\_(self, r):

self.r = r

@property

def area(self):

return self.r \*\* 2 \* pi

@property

def perimeter(self):

return 2 \* pi \* self.r

c = Circle(10)

print(c.area)

print(c.perimeter)

执行结果

314.1592653589793

62.83185307179586

例2

class Person:

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

self.\_\_name = name # 定义一个私有名字\_\_name

@property

def name(self):

return self.\_\_name

def set\_name(self, new\_name):

if type(new\_name) is str: # 判断name的数据类型是否为字符串

self.\_\_name = new\_name # 赋值

else:

print('你提供的姓名数据类型不合法')

p = Person('Sam')

print(p.name)

p.set\_name('Tom')

print(p.name)

执行结果

Sam

Tom

@func.setter --> func 对伪装的属性进行赋值的时候调用这个方法，一般情况下用来修改

示例：对方法伪装成的属性进行修改

class Person:

def \_\_init\_\_(self, n):

self.\_\_name = n # 私有的属性

@property

def name(self):

return self.\_\_name

@name.setter # 使用setter，一定要和property装饰的方法名相同，且要创建同名方法修改

def name(self, new\_name):

if type(new\_name) is str:

self.\_\_name = new\_name

else:

print('你提供的姓名数据类型不合法')

p = Person('sam')

print(p.name)

p.name = 'Jack' # 注意，修改使用等号，前面是老的名字，后面是新的名字

print(p.name)

执行结果

sam

Jack

@func.deleter --> func 在执行del 对象.func的时候调用这个方法 一般情况下用来做删除 基本不用

示例

class Person:

def \_\_init\_\_(self, n):

self.\_\_name = n

@property

def name(self):

return self.\_\_name

@name.deleter # 使用deleter也要确保和property装饰的方法同名，且要创建一个相同方法

def name(self):

del self.\_\_name

p = Person('Sam')

print(p.name)

del p.name # 删除名字

print(p.name) # 打印报错，因为名字已经被删除了

执行结果

AttributeError: 'Person' object has no attribute '\_Person\_\_name'

将一些需要随着一部分属性的变化而变化的值的计算过程 从方法 伪装成属性

将私有的属性保护起来，让修改的部分增加一些约束，来提高程序的稳定性和数据的安全性

示例：有个商品：原价，折扣，当我要查看价格的时候，肯定只看折后的价格

class Goods:

def \_\_init\_\_(self, name, origin\_price, discount):

self.name = name

self.\_\_price = origin\_price

self.\_\_discount = discount

@property

def price(self):

return self.\_\_price \* self.\_\_discount

@price.setter

def price(self, new\_price):

if type(new\_price) is int or type(new\_price) is float:

self.\_\_price = new\_price

apple = Goods('apple', 5, 0.8)

print(apple.price) # 打折后的价格

apple.price = 10 # 苹果价格上涨

print(apple.price)

执行结果

4.0

8.0

@classmethod

类方法 可以直接被类调用 不需要默认传对象参数 只需要传一个类参数就可以了

示例

class Goods:

\_\_discount = 0.8

def \_\_init\_\_(self, name, origin\_price):

self.name = name

self.\_\_price = origin\_price

@property

def price(self):

return self.\_\_price \* Goods.\_\_discount

@classmethod

def change\_discount(cls, new\_discount):

cls.\_\_discount = new\_discount

apple = Goods('apple', 5)

banana = Goods('banbana', 8)

print(apple.price)

print(banana.price)

# 折扣价格变了，不打折

# 不依赖对象的方法 就应该定义成类方法 类方法可以任意的操作类中的静态变量

Goods.change\_discount(1) # 更改全局静态变量

print(apple.price)

print(banana.price)

执行结果

4.0

6.4

5

8

@staticmethod

当一个方法要使用对象的属性时 就是用普通的方法

当一个方法要使用类中的静态属性时 就是用类方法

当一个方法要既不使用对象的属性也不使用类中的静态属性时，就可以使用staticmethod静态方法

示例

class Student:

def \_\_init\_\_(self): pass

@staticmethod

def login(): # login就是一个类中的静态方法 静态方法没有默认参数 就当成普通的函数使用即可

user = input('user:')

if user == 'root':

print('success')

else:

print('faild')

Student.login()

执行结果

user:root

success