**一、认识闭包**

出于种种原因，我们有时候需要在函数外部得到函数内的局部变量。但是，由于[Python中作用域的搜索顺序](https://zhuanlan.zhihu.com/p/400388568)（"链式作用域"结构（chain scope）：子对象会一级一级地向上寻找所有父对象的变量），这一点通常是无法实现的。

def f1():

n=999;

print(n)

https://pic2.zhimg.com/80/v2-6712a8821efc23b7f1af0d19ff88e5d1_720w.webp

但是有一种方法除外，那就是在函数的内部，再定义一个函数。

def f1():

n=999

def f2():

print(n)

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。这就是开头说到的，Python语言特有的作用域搜索顺序。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗?

def f1():

n=999

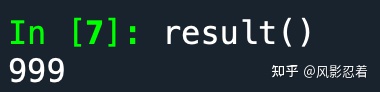
def f2():

print(n)

return f2

result = f1()

result()



**二、闭包的概念**

上一部分代码中的**f2函数**，就是闭包。

在上面的实例中，有一个外层函数的局部变量 n，有一个内层函数 f2，f2 里面可以访问到 n 变量，那这f2就是一个闭包。

下面再看一下维基百科的严谨定义：

在一些语言中，在函数中可以（嵌套）定义另一个函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。闭包可以用来在一个函数与一组“私有”变量之间创建关联关系。在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。

上面这段话实际上解释了闭包的一个定义和两个作用：

* 定义：闭包就是能够读取外部函数内的变量的函数。（前面已经讲解过）
* 作用1：闭包是将外层函数内的局部变量和外层函数的外部连接起来的一座桥梁。（下一部分讲解）
* 作用2：将外层函数的变量持久地保存在内存中。（下一部分讲解）

支持将函数当成对象使用的编程语言，一般都支持闭包。比如Python, JavaScript。

**三、闭包的用途**

闭包可以用在许多地方。

维基百科的定义中已经提到的它的两个用处：① 可以读取函数内部的变量，②让这些变量的值始终保持在内存中。

**（一）读取函数内部的变量**

在第一部分中，我们讲到，有时候会为了保证命名空间的干净而把一些变量隐藏到函数内部，作为局部变量。但是由于Python中作用域的搜索顺序，函数内的变量不会被函数外的代码读取到。

如果这时候想要函数外部的代码能够读取函数内部的变量，那么就可以使用闭包。

闭包存在的意义就是它夹带了外部变量（私货），如果它不夹带私货，它和普通的函数就没有任何区别。**同一个的函数**夹带了**不同的私货**，就实现了不同的功能。  
其实你也可以这么理解，闭包和面向接口编程的概念很像，可以把闭包理解成轻量级的接口封装。  
----

[@Wayne](https://www.zhihu.com/people/5bb8700e1883c346752afff210c72723)

这里再借用一下

[@Wayne](https://www.zhihu.com/people/5bb8700e1883c346752afff210c72723)

 的例子（[Wayne：用最简单的语言解释Python的闭包是什么？](https://zhuanlan.zhihu.com/p/59968665)）。

def tag(tag\_name):

def add\_tag(content):

return "<{0}>{1}</{0}>".format(tag\_name, content)

return add\_tag

content = 'Hello'

add\_tag = tag('a')

print add\_tag(content)

# <a>Hello</a>

add\_tag = tag('b')

print add\_tag(content)

# <b>Hello</b>

在这个例子里，我们想要一个给**content**加**tag**的功能，但是具体的**tag\_name**是什么样子的要根据实际需求来定，对外部调用的接口已经确定，就是**add\_tag(content)**。如果按照面向接口方式实现，我们会先把**add\_tag**写成接口，指定其参数和返回类型，然后分别去实现a和b的**add\_tag**。  
但是在闭包的概念中，add\_tag就是一个函数，它需要tag\_name和content两个参数，只不过tag\_name这个参数是打包带走的。所以一开始时就可以告诉我怎么打包，然后带走就行。  
----

[@Wayne](https://www.zhihu.com/people/5bb8700e1883c346752afff210c72723)

**（二）让函数内部的局部变量始终保持在内存中**

怎么来理解这句话呢？一般来说，函数内部的局部变量在这个函数运行完以后，就会被Python的垃圾回收机制从内存中清除掉。如果我们希望这个局部变量能够长久的保存在内存中，那么就可以用闭包来实现这个功能。

这里借用

[@千山飞雪](https://www.zhihu.com/people/d72af3d4fc31f9b68195e9caecc2d635)

 的例子（来自于：[千山飞雪：深入浅出python闭包](https://zhuanlan.zhihu.com/p/22229197)）。请看下面的代码。

以一个类似棋盘游戏的例子来说明。假设棋盘大小为50\*50，左上角为坐标系原点(0,0)，我需要一个函数，接收2个参数，分别为方向(direction)，步长(step)，该函数控制棋子的运动。 这里需要说明的是，每次运动的起点都是上次运动结束的终点。

def create(pos=[0,0]):

def go(direction, step):

new\_x = pos[0]+direction[0]\*step

new\_y = pos[1]+direction[1]\*step

pos[0] = new\_x

pos[1] = new\_y

return pos

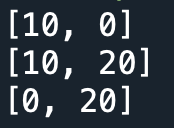
return go

player = create()

print(player([1,0],10))

print(player([0,1],20))

print(player([-1,0],10))



在这段代码中，player实际上就是闭包go函数的一个实例对象。

它一共运行了三次，第一次是沿X轴前进了10来到[10,0]，第二次是沿Y轴前进了20来到 [10, 20],，第三次是反方向沿X轴退了10来到[0, 20]。

这证明了，函数create中的局部变量pos一直保存在内存中，并没有在create调用后被自动清除。

为什么会这样呢？原因就在于create是go的父函数，而go被赋给了一个全局变量，这导致go始终在内存中，而go的存在依赖于create，因此create也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

这个时候，闭包使得**函数**的实例对象的内部变量，变得很像一个**类**的实例对象的属性，可以一直保存在内存中，并不断的对其进行运算。

**（三）总结**

* 局部变量无法共享和长久的保存，而全局变量可能造成变量污染，闭包既可以长久的保存变量又不会造成全局污染。
* 闭包使得函数内局部变量的值始终保持在内存中，不会在外层函数调用后被自动清除。
* 当外层函数返回了内层函数后，外层函数的局部变量还被内层函数引用
* 带参数的装饰器，那么一般都会生成闭包。
* 闭包在爬虫以及web应用中都有很广泛的应用。

**四、使用闭包的注意点**

**（一）内存消耗**

由于闭包会使得函数中的变量都被保存在内存中，会增加内存消耗，所以不能滥用闭包，否则会造成程序的性能问题，可能导致内存泄露。

解决方法是，在退出函数之前，将不使用的局部变量全部删除。

**（二）使用场景**

闭包的两个作用，“读取函数内部的变量”和“让函数内部的局部变量始终保持在内存中”，都可以被 Python 中现成的对象“类”很好地实现。我认为，“闭包”在 Python 中确实是一个必要性不大的概念。

那么为什么还要在 Python 中引入“闭包”这个概念呢？

首先，我觉得最重要的理由是，理解清楚这个概念，对于理解 Python 中的一大利器“装饰器”有很大的帮助。因为装饰器本身就是闭包的一个应用。

其次，当我们要实现的功能比较简单的时候，可以用闭包。例如：

* 当我们的代码中函数比较少的时候，可以使用闭包。（但是如果我们要实现很多功能，还是要使用类（OOP））
* 如果我们的对象中只有一个方法时，使用闭包是会比用类来实现更优雅。

这有点类似于，如果我们要实现比较简单的函数功能，通常使用 lambda 匿名函数比定义一个完整的function更加优雅，而且几乎不会损失可读性。类似的还有用列表解析式代替 for 循环。

**（三）闭包无法改变外部函数局部变量指向的内存地址**

这个是什么意思呢？我们来看下面的例子。

def outer\_fun():

x = 0

def inner\_fun():

x = 1

print('inner x:',x, 'at', id(x))

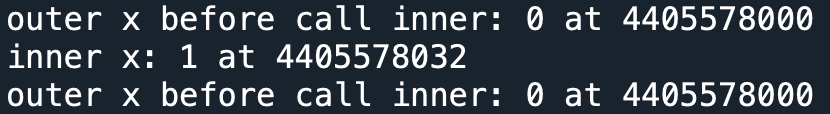
print('outer x before call inner:', x, 'at', id(x))

inner\_fun()

print('outer x before call inner:', x, 'at', id(x))

outer\_fun()

如果 innerFunc 可以修改 x 的的内存地址的话，那么 x 首先在outer\_fun中指向了一个储存着 0 的内存地址，后面又在 inner\_fun中，x 会指向新的储存着 1 的内存地址（由于int是不可变类型），但结果是：



在 innerFunc 中 x 的值发生了改变，但是原因是重新创建了一个变量 x，指向了一个新的内存地址。而在 outerFunc 中 x 的值以及内存地址并未发生变化。

造成这一结果的原因的根源，还是前面第一部分讲的[Python中作用域的搜索顺序](https://zhuanlan.zhihu.com/p/400388568)。在 inner\_fun 函数里面，有自己的命名空间，这个命名空间是独立于 outer\_fun 的命名空间的。它里面的x是一个局部名称（local names），在执行 “x=1” 命令的时候，是重新在 inner\_fun自己的命名空间里创建了一个新的变量 x ，而无法覆盖掉 outer\_fun 的命名空间的 x。

如果要让内层函数不仅可以访问，还要可以修改外层函数的变量，那么需要用到nonlocal声明，使得内层函数不要在自己的命名空间创建新的x，而是操作外层函数命名空间的x。

def outer\_fun():

x = 0

def inner\_fun():

nonlocal x # 注意这里

x = 1

print('inner x:',x, 'at', id(x))

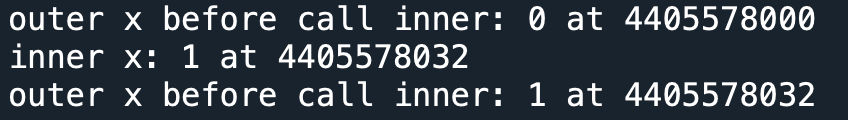
print('outer x before call inner:', x, 'at', id(x))

inner\_fun()

print('outer x before call inner:', x, 'at', id(x))

outer\_fun()

我们可以发现，此时 inner\_fun 改变了 outer\_fun 中的变量的内存地址。



同样地，在上文棋盘的例子中，外层函数的变量pos内的值虽然一直在改变，但是由于列表本身是可变类型的变量，虽然列表中的元素一直在变，但是列表本身的内存地址没有发生变化。

**（四）返回闭包时，返回函数不要引用任何循环变量，或者后续会发生变化的变量**

**在Python中，如果要返回一个函数，那么返回函数不要引用任何循环变量，或者后续会发生变化的变量。**

因为，返回的函数并没有立刻执行，而是直到调用了f()才执行。我们来看一个例子：

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都放在列表中，通过列表整体返回了。

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是：

>>> f1()

9

>>> f2()

9

>>> f3()

9

因为在向列表中添加 func 的时候，i 的值没有固定到f的实例对象中，而仅是将计算公式固定到了实例对象中。等到了调用f1()、f2()、f3()的时候才去取 i的值，这时候循环已经结束，i 的值是3，所以结果都是9。

**因此，返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。**

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变。

def count():

def f(j):

def g():

return j\*j

return g

fs = []

for i in range(1, 4):

fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()

return fs

再看看结果：

>>> f1, f2, f3 = count()

>>> f1()

1

>>> f2()

4

>>> f3()

9

**五、判断一个函数是否是闭包**

判断一个函数是不是闭包，可以查看它的**closure**属性。如果该函数是闭包，查看该属性将会返回一个cell对象组成的tuple。如果我们分别对每个cell对象查看其cell\_contents属性，返回的内容就是闭包引用的自由变量的值。

下面通过一个例子展示：

def add(x,y):

def f(z):

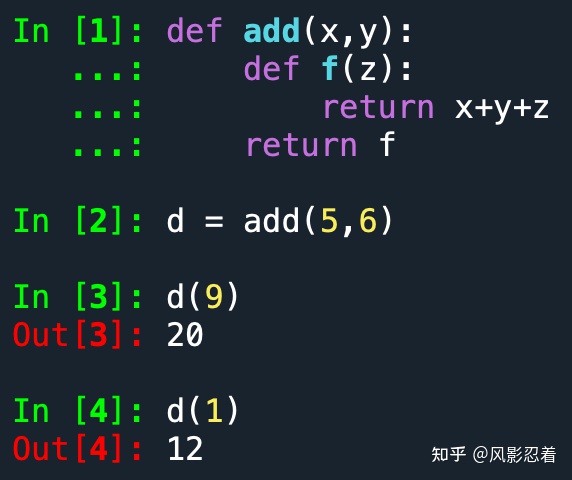
return x+y+z

return f

d = add(5,6)

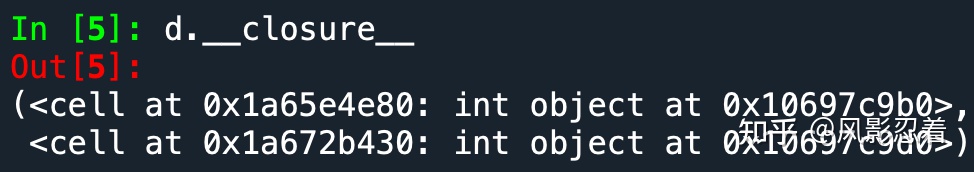
d(9)

d(1)



闭包的\_\_closure\_\_方法，可以展示出闭包储存了外部函数的两个变量，cell的内存地址是什么，在cell里面储存的对象类型是int，这个int储存的内存地址是什么。

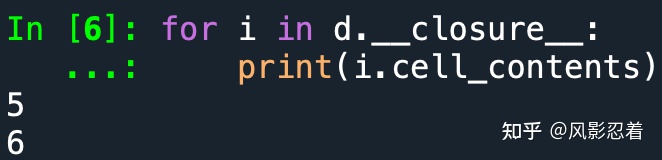
d.\_\_closure\_\_



闭包的\_\_closure\_\_方法，可以查看每个cell对象的内容。

for i in d.\_\_closure\_\_:

print(i.cell\_contents)



cell\_contents解释了局部变量在脱离函数后仍然可以在函数之外被访问的原因，因为变量被存储在cell\_contents中了。

如果你看到了这里，说明你是一个真正的 Python 深度的使用者和学习者。那么，不妨打开评论区看一看，里面有大量相同背景的人，发起了关于常见问题的全面讨论，说不定就能解决困惑你很久的问题。如果你的问题还没有被讨论到，也可以在评论区留言，考虑到本文下评论区热烈的讨论情况，说不定你的问题很快就能被解答：）