8 闭包

> AI领域中的Python开发 — by 丁宁

SIGAI课程录制

- 上节课：函数式编程概述，一等函数，高阶函数，匿名函数
- 本节课：闭包，再谈变量作用域，详解`nonlocal`

本节课概述

- 装饰器的本质是一个闭包，而`@`仅仅是一个语法糖
- 闭包的基础是Python中的函数是一等对象
- 理解闭包需要知道Python如何识别变量所处的作用域
- 自定义变量所处的作用域有三种：`global nonlocal local`

再谈变量作用域

从内层函数的角度看，变量使用的两个维度

- 是否能访问：**LEGB**规则
- 是否能修改：需要声明才能修改

变量作用域识别三要素

- 出现位置：在哪里访问了
- 赋值位置：在哪里赋值了
- 声明类型：在哪里声明了

三种变量作用域

- 局部：`local`
- 全局：`global`
- 非全局：`nonlocal`

3种作用域；5次调用；3次赋值；0次声明

```
a = 1
print(a)

def func_enclosed():
    a = 2
    print(a)
```

```
    def func_local():
        a = 3
        print(a)

    func_local()
    print(a)

func_enclosed()
print(a)
```

```
1
2
3
2
1
```

总结：无声明的情况下，赋值即私有，若外部有相同变量名则将其遮挡

3种作用域；5次调用；1次赋值；0次声明

```
a = 1
print(a)

def func_enclosed():
    #a = 2
    print(a)

    def func_local():
        #a = 3
        print(a)

    func_local()
    print(a)

func_enclosed()
print(a)
```

```
1
1
1
1
1
```

总结：无赋值情况下，变量访问依据**LEGB**法则

3种作用域；5次调用；3次赋值；1次`global`声明

```
a = 1
print(a)

def func_enclosed():
    global a
    a = 2
    print(a)

    def func_local():
        a = 3
        print(a)

    func_local()
    print(a)

func_enclosed()
print(a)
```

```
1
2
3
2
2
```

总结：内层函数可以通过声明的方式直接修改外部变量

3种作用域；5次调用；3次赋值；1次`global`声明

```
a = 1
print(a)

def func_enclosed():
    a = 2
    print(a)

    def func_local():
        global a
        a = 3
        print(a)

    func_local()
```

```
    print(a)

func_enclosed()
print(a)
```

```
1
2
3
2
3
```

总结：位于最内层的函数，通过`global`声明，会越过中间层，直接修改全局变量

3种作用域；5次调用；3次赋值；2次`global`声明

```
a = 1
print(a)

def func_enclosed():
    global a
    a = 2
    print(a)

    def func_local():
        global a
        a = 3
        print(a)

    func_local()
    print(a)

func_enclosed()
print(a)
```

```
1
2
3
3
3
```

总结：`global`声明其实是一种绑定关系，意思是告诉解释器，不用新创建变量了，我用的是外面那个

3种作用域；5次调用；3次赋值；1次`nonlocal`声明

```
a = 1
print(a)

def func_enclosed():
    a = 2
    print(a)

    def func_local():
        nonlocal a
        a = 3
        print(a)

    func_local()
    print(a)

func_enclosed()
print(a)
```

```
1
2
3
3
1
```

总结：位于最内层的函数，如果仅想修改中间层变量，而不是全局变量，可使用`nonlocal`关键字

3种作用域；5次调用；3次赋值；1次`nonlocal`声明；1次`global`声明

```
a = 1
print(a)

def func_enclosed():
    global a
    a = 2
    print(a)

    def func_local():
        nonlocal a
        a = 3
        print(a)

    func_local()
    print(a)

func_enclosed()
```

```
print(a)
```

```
  File "test.py", line 10
    nonlocal a
    ^
SyntaxError: no binding for nonlocal 'a' found
```

总结：`nonlocal`只能绑定在中间层定义的变量，如果中间层变量被声明外全局变量，则会报错

金句：

- 无声明的情况下，赋值即私有，若外部有相同变量名则将其遮挡
- 想修改外部相同变量名，需要将外部变量声明
- 根据外部变量的作用域级别不同，使用`global或者nonlocal`

Python解释器如何识别变量的作用域？

- 先看出现了几层作用域
- 再看变量出现的位置
- 如果有对变量进行赋值操作，则再看是否声明为外部变量

$$global, nonlocal, local <= \text{变量出现的位置} + \text{变量是否被赋值} + \text{变量是否被明确声明}$$

为什么会有`nonlocal`关键字？

- `nonlocal`填补了`global与local`之间的空白
- `nonlocal`的出现其实是一种权衡利弊的结果：**私有之安全封装，全局之灵活共享**

> 而这也是**闭包**之所以出现的原因之一

什么是闭包？

- 定义：延伸了作用域的函数（能访问**定义体之外**定义的**非全局变量**）

从一个`avg`函数说起，这个函数是这样的：

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

第一个版本：用类实现

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

结果如下：

```
>>> from test import Averager
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

第二个版本：用闭包实现

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

认识闭包

- 闭包是一种函数，它会保留定义函数时存在的**外层非全局变量**的绑定

第三个版本：优化过的闭包

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

```
>>> from test import make_averager
>>> avg = make_averager()
>>> avg(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/dingning/Desktop/test.py", line 6, in averager
    count += 1
UnboundLocalError: local variable 'count' referenced before assignm
ent
```

怎么办？

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

```
>>> from test import make_averager
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
```

```
10.5
>>> avg(12)
11.0
```

思考：为什么刚才就可以修改？现在就不可以了呢？

- 还记得可变对象与不可变对象的区别吗？
- Python 2中没有`nonlocal`关键字，只能用可变对象来临时性的解决中间层变量修改的问题
- 而`nonlocal`是Python 3中引入的一个官方的解决方案，以弥补内层函数无法修改中间层不可变对象

闭包有什么用呢？

1. 共享变量的时候避免使用了不安全的全局变量
2. 允许将函数与某些数据关联起来，类似于简化版面向对象编程
3. 相同代码每次生成的闭包，其延伸的作用域都彼此独立（计数器，注册表）
4. 函数的一部分行为在编写时无法预知，需要动态实现，同时又想保持接口一致性
5. 较低的内存开销：类的生命周期远大于闭包
6. 实现装饰器

> 到此为止，我们已经具备了学习装饰器的全部准备工作，下节课，我们学习Python中的重头戏-装饰器