第七章闭包函数

> 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 函数嵌套

1. 函数嵌套调用

函数嵌套调用指的是在函数内部又调用了其他的函数。

```
#  求三个数的最大值
def max2(x, y):
    if x > y:
        return x
    else:
        return y


def max3(x, y, z):
    res1 = max2(x, y)
    res2 = max2(res1, z)
    return res2


print(max3(11, 199, 2))
```

2. 函数嵌套定义

函数的嵌套定义指的是在函数内又定义了其他函数。

```
# 示例1
def func1():
    print('from func1')
```

```
    def func2():
        print('from func2')

    print(func2)
    func2()


func1()
# print(func2)


# 示例2
# 函数在定义阶段不执行函数体内的代码
def f1():
    print('f1')

    def f2():
        print('f2')

        def f3():
            print('f3')

        f3()

    f2()


f1()
```

二 名称空间

1. 名称空间说明

名称空间从字面意思理解是存放名字的地方，最开始我们在讲解变量的时候知道：定义一个变量就是开辟一块内存空间，这个内存空间存放的是变量的值，除了变量值之外，还有变量名，变量名与变量值的绑定关系这个数据要在内存中存储。变量名是名字，函数名也是名字，名称空间就是存放名字与值的绑定关系的地方。

2. 名称空间的分类

名称空间分为三类：

1. 内置名称空间：存放Python解释器自带的名字，在解释器启动时就生效，解释器关闭则失效。
2. 全局名称空间：文件级别的名字,在执行文件的时候生效，在文件结束或者在文件执行期间被删除则失效。有些全局名称空间可能一眼看上去不像是全局名称空间，我们只需要记住，只要不是内置名称空间和局部名称空间，那么就是全局名称空间。
3. 局部名称空间：存放函数内定义的名字(函数的参数以及函数内的名字都存放于局部名称空间)，在函数调用时临时生效，函数结束则失效。

```
# 内置
print(print)
print(len)

# 以下4个都是全局

x = 1   # 全局


def func():  # func是全局名称空间
    name = 'Albert'   # name是局部名称空间


# del func  # 我们不需要自己删除


if 10 > 3:
    y = 2   # 全局

while True:
    z = 5   # 全局
    break


# 局部
def func1(x):  # 实参1传给形参x就相当于在函数内被定义 x=1，故x为局部
    # x = 1
    y = 2   # 局部


func1(1)
```

内置名称空间与全局名称空间的结束的生命周期基本上是一致的，程序执行结束或者文件关闭(手动强制关闭写python代码的这个文件)，内置或全局名称空间生命周期结束。局部名称空间生命周期是从函数调用开始到函数结束，即函数生命周期终止。

> 加载顺序：内置名称空间–>全局名称空间–>局部名称空间
> 查找名字：局部名称空间–>全局名称空间–>内置名称空间

```
# 加载顺序很好理解，查找顺序是以当前位置为起始点

len = 0  # 我们是为了测试，自己写代码千万不能覆盖内置名称空间


def f1():
    # len = 1

    def f2():
        # len = 2
        print(len)

    # len = 3
    f2()


f1()
```

三 作用域

1. 全局作用域

全局作用域包含的是内置名称空间与全局名称空间的名字，它的特点是：

1. 在任何位置都能够访问的到
2. 该范围内的名字会伴随程序整个生命周期

如果在局部使用全局作用域的变量，是没有问题的，但是如果是在局部修改全局作用域的变量则不能直接修改，而要使用`global`关键字才可以修改。

```
global_count = 0


def global_check():
    print(global_count)  # 直接使用全局变量


def global_modify():
    global global_count  # 修改前需要先使用global
    global_count += 1
    print(global_count)


global_check()
global_modify()
```

2. 局部作用域

局部作用域包含的是局部名称空间的名字，它的特点是：

1. 只能在函数内使用
2. 调用函数时生效，调用结束失效

如果在局部使用的是嵌套在函数内部的局部变量，同理，可以直接使用，而修改需要使用`nonlocal`关键字。

```
def make_counter():
    count = 0

    def check_counter():
        print(count)

    check_counter()

    def modify_counter():
        nonlocal count
        count += 1
        print(count)

    modify_counter()
```

```
make_counter()
```

四 函数对象

1. 函数对象说明

函数在Python中是第一类对象，这句话可以通俗理解为函数也是一个对象，就像是int，字符串，列表和字典一样都是对象，等讲到了面向对象我们就会对这个概念有了进一步理解，现在就可以暂时理解为函数对象可以像int或者字符串一样使用。

```
# 1 函数可以被引用

# int示例
x = 1
y = x


# 函数示例
def bar():
    print('from bar')


f = bar
f()

# 2 可以当中参数传入

# int示例
x = 1


def func(a):
    print(a)


func(x)
```

```
# 函数示例
def bar():
    print('from bar')


def wrapper(func):
    func()


wrapper(bar)

# 3 可以当中函数的返回值

# int示例
x = 1


def foo():
    return x


res = foo()
print(res)

# 函数示例
def bar():
    print('from bar')


def foo(func):
    return func


f = foo(bar)
f()

# 4 可以当中容器类型的元素

# int示例
z = 1
l = [z, ]

print(l)

# 函数示例
```

```
def get():
    print('from get')


def put():
    print('from put')


l1 = [get, put]

l1[0]()
```

2. 函数对象应用(重点)

利用这一特性，可以优雅的取代原来的`if`多分支(`elif`这种多分支是我们写代码要尽可能避免的)。

```
def auth():
    print('登陆。。。。。')


def register():
    print('注册。。。。。')


def check():
    print('查看。。。。')


def transfer():
    print('转账。。。。')


def pay():
    print('支付。。。。')


func_dict = {
    '1': auth,
    '2': register,
    '3': check,
    '4': transfer,
    '5': pay
```

```
}


def interactive():
    while True:
        print("""
        1 登录
        2 注册
        3 查看
        4 转账
        5 支付
        """)
        choice = input('>>: ').strip()
        if choice in func_dict:
            func_dict[choice]()
        else:
            print('非法操作')


interactive()  # 是不是比多个if...elif...简洁明了多了？
```

五 闭包函数

1. 闭包函数说明

闭包函数就是定义在函数内部的函数，也就是函数的嵌套定义，根据字面意思理解，闭包函数有两个关键字`闭`和`包`分别是的封闭和包裹。需要注意的重点是：闭包函数的作用域关系在函数定义阶段就固定死了，与调用位置无关。

```
def outer():
    x = 1

    def inner():  # 在outer函数内部再定一个函数
        # x = 2
        print('from inner', x)

    return inner  # outer函数返回inner函数对象


f = outer()  # 现在的f是一个全局变量，同时是inner函数对象
```

```
print(f)
x = 3  # 这个x = 3并不能改变inner函数外层的x
f()


def foo():
    x = 4  # 这个x = 4 同样也不能改变
    f()  # 全局作用域在任意位置都可以调用


foo()
```

闭包函数可以用外层函数来调用内部的函数，打破了函数的层级限制，与此同时该函数包含对外部函数作用域中名字的引用。

```
def outer():
    name = 'Albert'

    def inner():
        print('my name is %s' % name)

    return inner


f = outer()
f()
```

2. 为函数体传值的两种方式

(1) 以参数的形式的传入

```
# 模块的导入后面章节会讲解，requests模块就是模拟浏览器向目标站点发请求
import requests


def get(url):
    response = requests.get(url)  # get方法获取请求返回对象
    print(response)
    if response.status_code == 200:  # 200是一个状态码，代表请求成功
```

```
        print(response.text)  # text方法是获取返回对象的内容


get('https://www.baidu.com')
```

(2) 以闭包函数的形式

闭包函数就是在函数外层再包裹一层作用域，由于这个作用域在外层函数内部，所以只作用在内层函数上。

```
import requests


def outer(url):  # 给外层函数传参就相当于 url='https://www.baidu.com'

    # url='https://www.baidu.com'  # 这个作用域就是作用在get函数上的
    def get():
        response = requests.get(url)
        if response.status_code == 200:
            print(response.text)

    return get


baidu = outer('https://www.baidu.com')
python = outer('https://www.python.org')

baidu()
print('================================>')
python()
```

简化与总结闭包函数用法

```
def outer(x):
    def foo():  # foo虽然没有直接传参，但是outer函数的作用域赋予了x动态变化
        print(x)

    return foo
```

```
f_10 = outer(10)

f_10()

f_100 = outer(100)
f_100()
```

六 装饰器

1. 装饰器说明

器指的工具(只要是工具，你就应该想到函数)，装饰指的是为被装饰对象添加新功能，需要注意的是：项目一旦上线之后，就应该遵循开发封闭的原则。开放封闭指的是对修改函数内的源代码和调用方式是封闭的，对功能的扩展是开放的。看起来有点矛盾，但这就是我们要做的。在这样的要求下，我们必须要找到一种解决方案，能够在不修改一个功能内源代码以及调用方式的前提下，为其添加新功能。这就用到了装饰器，它能够在不修改被装饰对象源代码与调用方式的前提下，为被装饰器对象添加新功能。

2. 无参装饰器

(1) 无参装饰器实现过程

无参装饰器指的是装饰器本身没有参数。

```
# 要求：为index函数添加一个统计时间的功能
import time  # 这是一个与时间相关的模块


def index():
    time.sleep(3)  # 睡3秒
    print('welcome to index page')


index()

# 版本一(只有index函数可以使用)
import time  # 这是一个与时间相关的模块
```

```
def index():
    time.sleep(3)  # 睡3秒
    print('welcome to index page')


start_time = time.time()  # 从1970年开始计时的时间戳
index()
end_time = time.time()
print('run time is %s' % (end_time - start_time))


# 版本二(两个函数都可以使用，但是有大量重复代码)
def index():
    time.sleep(3)
    print('welcome to index page')


def home(name):
    time.sleep(5)
    print('welcome %s to home page' % name)


start_time = time.time()
index()
stop_time = time.time()
print('run time is %s' % (stop_time - start_time))

start_time = time.time()
home('Albert')
stop_time = time.time()
print('run time is %s' % (stop_time - start_time))

# 版本三(修改了源函数的调用方式)
import time


def index():
    time.sleep(3)
    print('welcome to index page')


def home(name):
    time.sleep(5)
    print('welcome %s to home page' % name)
```

```
def wrapper(func):  # func=index
    start_time = time.time()
    func()  # index()
    stop_time = time.time()
    print('run time is %s' % (stop_time - start_time))


wrapper(index)

# 版本四(使用闭包函数，不修改源函数调用方式)
import time


def index():
    time.sleep(3)
    print('welcome to index page')


def outer(func):  # func=最原始的index
    # func=最原始的index
    def wrapper():
        start_time = time.time()
        func()
        stop_time = time.time()
        print(stop_time - start_time)

    return wrapper


# a = outer(index)  # outer函数结果可以赋值给任意变量
# b = outer(index)
# c = outer(index)

index = outer(index)  # 赋值给index覆盖原来的index, index = wrapper

index()  # wrapper()

# 版本五(解决原函数返回值无效)
import time


def index():
    time.sleep(1)
    print('welcome to index page')
```

```
    return 1  # 假如源函数有一个返回值


def outer(func):
    # func=最原始的home
    def wrapper():
        start_time = time.time()
        res = func()  # 调用最原始的index
        stop_time = time.time()
        print(stop_time - start_time)
        return res

    return wrapper


index = outer(index)  # 新的index=wrapper
res = index()  # 上一个版本返回值为None
print(res)

# 版本六(终极版，解决有参函数和无参函数通用的问题)

import time


def index():
    time.sleep(1)
    print('welcome to index page')
    return 1


def home(name):
    time.sleep(2)
    print('welcome %s to home page' % name)


def timer(func):  # 装饰器也是一个函数，我们给他一个好听的名字
    def wrapper(*args, **kwargs):  # wrapper函数有无参数由源函数决定
        start_time = time.time()
        res = func(*args, **kwargs)
        stop_time = time.time()
        print(stop_time - start_time)
        return res

    return wrapper
```

```
index = timer(index)  # 新的index=wrapper
home = timer(home)  # 新的home=wrapper

home(name='Albert')  # wrapper(name='Albert')
home('Albert')  # wrapper('Albert')
index()  # wrapper()


# 无参装饰器模板
def outer(func):
    def inner(*args, **kwargs):
        """
        这里写装饰器逻辑
        :param args: 任意位置参数
        :param kwargs: 任意关键参数
        :return: 一个函数对象
        """
        res = func(*args, **kwargs)
        return res

    return inner
```

(2) 装饰器语法糖

```
import time


# 装饰器也是一个函数，使用函数必先定义，所以装饰器放在最上方
def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        res = func(*args, **kwargs)
        stop_time = time.time()
        print(stop_time - start_time)
        return res

    return wrapper


@timer  # 在被装饰对象正上方单独一行添加，相当于执行index=timer(index)
def index():
    time.sleep(1)
```

```
    print('welcome to index page')
    return 1


# @timer  # home=timer(home) 当不需要装饰器的时候只需注释这一行即可
def home(name):
    time.sleep(2)
    print('welcome %s to home page' % name)


index()
home('Albert')
```

(3) 用户认证装饰器

```
import time

current_user = {
    'username': None,
}


def auth(func):
    def wrapper(*args, **kwargs):
        if current_user['username']:
            print('已经登陆过了')
            res = func(*args, **kwargs)
            return res

        name = input('用户名>>: ').strip()
        pwd = input('密码>>: ').strip()
        if name == 'Albert' and pwd == '1':
            print('登陆成功')
            current_user['username'] = name
            res = func(*args, **kwargs)
            return res
        else:
            print('用户名或密码错误')

    return wrapper


@auth
```

```
def index():
    time.sleep(1)
    print('welcome to index page')
    return 1


@auth
def home(name):
    time.sleep(2)
    print('welcome %s to home page' % name)


index()
home('Albert')
```

(3) 多个装饰器叠加

```
import time

current_user = {
    'username': None,
}


def auth(func):
    def wrapper(*args, **kwargs):
        if current_user['username']:
            print('已经登陆过了')
            res = func(*args, **kwargs)
            return res

        name = input('用户名>>: ').strip()
        pwd = input('密码>>: ').strip()
        if name == 'Albert' and pwd == '1':
            print('登陆成功')
            current_user['username'] = name
            res = func(*args, **kwargs)
            return res
        else:
            print('用户名或密码错误')

    return wrapper
```

```
def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        res = func(*args, **kwargs)
        stop_time = time.time()
        print(stop_time - start_time)
        return res

    return wrapper


"""
@auth
@timer  # 这样写的话timer只统计index的执行时间
"""


@timer  # timer 统计的是auth+index的执行时间
@auth
def index():
    time.sleep(1)
    print('welcome to index page')
    return 1


index()
```

3. 有参装饰器

有参装饰器就是装饰器本身需要一个参数，结合我们以前讲过的文件操作，其实文件就是存放数据的仓库，类似于数据库，数据库分很多种，常见的有MySQL，Oracle，PostgreSQL和DB2等等，在一些项目的需求中，不同的数据会分散存储在不同的数据库中，这时我们使用基于对象的数据模型(通俗讲就是使用编程语言来操作数据库)操作不同数据库就要执行不同的代码。

```
import time

current_user = {
    'username': None,
}
```

```python
def auth(engine):
    def user_auth(func):
        def wrapper(*args, **kwargs):
            if engine == 'file':
                print('基于文件的认证')
                if current_user['username']:
                    print('已经登陆过了')
                    res = func(*args, **kwargs)
                    return res

                name = input('用户名>>: ').strip()
                pwd = input('密码>>: ').strip()
                if name == 'Albert' and pwd == '1':
                    print('登陆成功')
                    current_user['username'] = name
                    res = func(*args, **kwargs)
                    return res
                else:
                    print('用户名或密码错误')
            elif engine == 'mysql':
                print('基于MyQL的认证')
            elif engine == 'ldap':
                print('基于LDAP的认证')
            elif engine == 'postgresql':
                print('基于PostgreSQL的认证')

        return wrapper

    return user_auth


@auth('file')  # auth装饰器本身是一个函数，在语法糖中也可以传参数
def index():
    time.sleep(1)
    print('welcome to index page')
    return 1


index()
```

4. 装饰器补充

```python
from functools import wraps


def deco(func):
    @wraps(func)  # 加在最内层函数正上方
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)

    return wrapper


@deco
def index():
    '''哈哈哈哈'''
    print(index.__doc__)


index()
```

装饰器使用的是闭包函数的原理，返回的是和原来函数同名字的函数地址，再加上()就能调用这个函数，所以给我们的感觉是原来的函数没有变化，却添加了新的功能，其实已经不是原来的函数了，你可以把以上代码的第三行注释掉，运行代码，打印结果为None，就是因为你运行的函数已经不是原来的函数了，所以这其实是一个伪装饰器，要想让装饰器真的是装饰器，调用别人写好的包，返回结果还是原函数，以上写法就是。