11. 前方高能-装饰器初识

本节主要内容:

1. 函数名的运用, 第一类对象
2. 闭包
3. 装饰器初识

一. 函数名的运用.
函数名是一个变量, 但它是一个特殊的变量, 与括号配合可以执行函数的变量.

1. 函数名的内存地址

```
def func():
    print("呵呵")

print(func)

结果:
<function func at 0x1101e4ea0>
```

2. 函数名可以赋值给其他变量

```
def func():
    print("呵呵")

print(func)

a = func    # 把函数当成一个变量赋值给另一个变量
a()     # 函数调用 func()
```

3. 函数名可以当做容器类的元素

```
def func1():
    print("呵呵")

def func2():
    print("呵呵")

def func3():
    print("呵呵")

def func4():
    print("呵呵")
```

```
lst = [func1, func2, func3]
for i in lst:
    i()
```

4. 函数名可以当做函数的参数

```
def func():
    print("吃了么")


def func2(fn):
    print("我是func2")
    fn()    # 执行传递过来的fn
    print("我是func2")

func2(func)     # 把函数func当成参数传递给func2的参数fn.
```

5. 函数名可以作为函数的返回值

```
def func_1():
    print("这里是函数1")
    def func_2():
        print("这里是函数2")
    print("这里是函数1")
    return func_2

fn = func_1()   # 执行函数1.  函数1返回的是函数2, 这时fn指向的就是上面函数2
fn()    # 执行上面返回的函数
```

二. 闭包

什么是闭包? 闭包就是内层函数, 对外层函数(非全局)的变量的引用. 叫闭包

```
def func1():
    name = "alex"
    def func2():
        print(name)     # 闭包
    func2()
func1()
结果:
alex
```

我们可以使用__closure__来检测函数是否是闭包. 使用函数名.__closure__返回cell就是

闭包. 返回None就不是闭包

```
def func1():
    name = "alex"
    def func2():
        print(name)     # 闭包
    func2()
    print(func2.__closure__)    # (<cell at 0x10c2e20a8: str object at
0x10c3fc650>,)
func1()
```

问题, 如何在函数外边调用内部函数呢?

```
def outer():
    name = "alex"
    # 内部函数
    def inner():
        print(name)
    return inner

fn = outer()   # 访问外部函数, 获取到内部函数的函数地址
fn()    # 访问内部函数
```

那如果多层嵌套呢? 很简单, 只需要一层一层的往外层返回就行了

```
def func1():
    def func2():
        def func3():
            print("嘿嘿")
        return func3
    return func2

func1()()()
```

由它我们可以引出闭包的好处. 由于我们在外界可以访问内部函数. 那这个时候内部函数访问的时间和时机就不一定了, 因为在外部, 我可以选择在任意的时间去访问内部函数. 这个时候. 想一想. 我们之前说过, 如果一个函数执行完毕. 则这个函数中的变量以及局部命名空间中的内容都将会被销毁. 在闭包中. 如果变量被销毁了. 那内部函数将不能正常执行. 所以. python规定. 如果你在内部函数中访问了外层函数中的变量. 那么这个变量将不会消亡. 将会常驻在内存中. 也就是说. 使用闭包, 可以保证外层函数中的变量在内存中常驻. 这样做有什么好处呢? 非常大的好处. 我们来看一个关于爬虫的代码:

```
from urllib.request import urlopen

def but():
    content = urlopen("http://www.xiaohua100.cn/index.html").read()
    def get_content():
        return content
```

```
    return get_content

fn = but()  # 这个时候就开始加载校花100的内容
# 后面需要用到这里面的内容就不需要在执行非常耗时的网络连接操作了
content = fn()  # 获取内容
print(content)

content2 = fn()  # 重新获取内容
print(content2)
```

综上, 闭包的作用就是让一个变量能够常驻内存. 供后面的程序使用.

三. 装饰器初识

在说装饰器之前啊. 我们先说一个软件设计的原则: 开闭原则, 又被成为开放封闭原则, 你的代码对功能的扩展是开放的, 你的程序对修改源代码是封闭的. 这样的软件设计思路可以更好的维护和开发.

开放: 对功能扩展开放

封闭: 对修改代码封闭

接下来我们来看装饰器. 首先我们先模拟一下女娲造人.

```
def create_people():
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

create_people()
```

ok! 很简单. 但是现在问题来了. 上古时期啊. 天气很不稳定. 这时候呢大旱三年. 女娲再去造人啊就很困难了. 因为啥呢? 没水. 也就是说. 女娲想造人必须得先和泥. 浇点儿水才能造人.

```
def create_people():
    print("浇水")      # 添加了个浇水功能
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

create_people()
```

搞定. 但是, 我们来想想. 是不是违背了我们最开始的那个约定"开闭原则", 我们是添加了新的功能. 对添加功能开放. 但是修改了源代码啊. 这个就不好了. 因为开闭原则对修改是封闭的. 那怎么办. 我们可以这样做.

```
def create_people():
   # print("浇水")      # 添加了个浇水功能, 不符合开闭原则了
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

def warter():
    print("先浇水")
    create_people()  # 造人

# create_people()   # 这个就不行了.
```

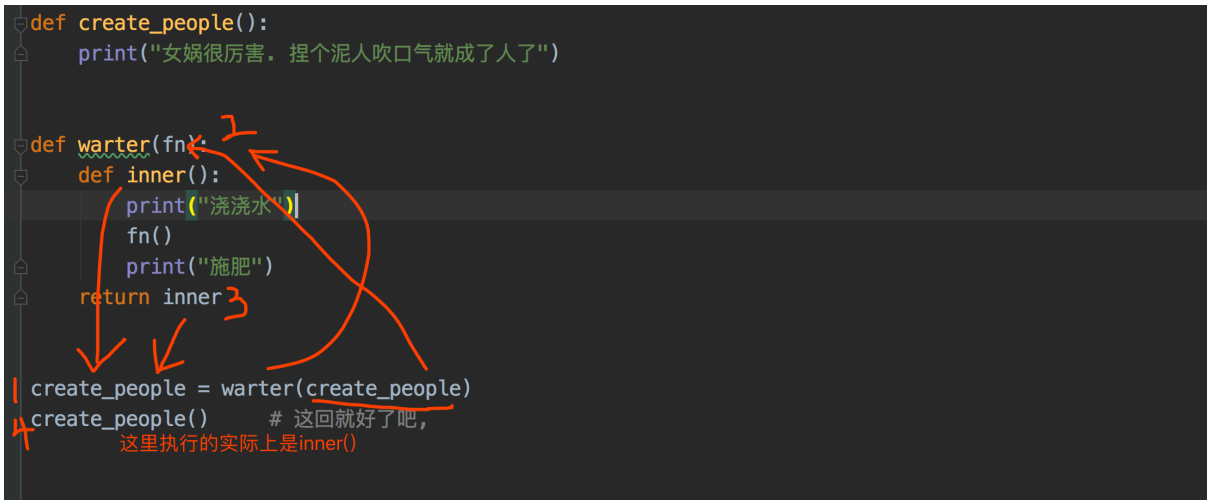
```
warter()        # 访问浇水就好了
```

现在问题又来了. 你这个函数写好了. 但是由于你添加了功能. 重新创建了个函数. 在这之前访问过这个函数的人就必须要修改代码来访问新的函数water() 这也要修改代码. 这个也不好. 依然违背开闭原则. 而且. 如果你这个函数被大量的人访问过. 你让他们所有人都去改. 那你就要倒霉了. 不干死你就见鬼了.

那怎么办才能既不修改原代码, 又能添加新功能呢? 这个时候我们就需要一个装饰器了. 装饰器的作用就是在不修改原有代码的基础上, 给函数扩展功能.

```
def create_people():
   # print("浇水")      # 添加了个浇水功能, 不符合开闭原则了
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

def warter(fn):
    def inner():
        print("浇浇水")
        fn()
        print("施肥")
    return inner

# # create_people()    # 这个就不行了.
# warter()        # 访问浇水就好了

func = warter(create_people)
func()  # 有人问了. 下游访问的不依然是func么, 不还是要改么?

create_people = warter(create_people)
create_people()     # 这回就好了吧,
```

说一下执行流程:



1. 首先访问warter(create_people).

2. 把你的目标函数传递给warter的形参fn. 那么后面如果执行了fn意味着执行了你的目标函数create_people

3. warter()执行就一句话. 返回inner函数. 这个时候. 程序认为warter()函数执行完. 那么前面的create_people函数名被重新覆盖成inner函数
4. 执行create_people函数. 实际上执行的是inner函数. 而inner中访问的恰恰使我们最开始传递进去的原始的create_people函数

结论: 我们使用warter函数把create_people给包装了一下. 在不修改create_people的前提下. 完成了对create_people函数的功能添加

这是一个装饰器的雏形. 接下来我们观察一下代码. 很不好理解. 所以呢. 我们可以使用语法糖来简化我们的代码

```
def warter(fn):
    def inner():
        print("浇浇水")
        fn()
        print("施肥")
    return inner


@warter     # 相当于 create_people = warter(create_people)
def create_people():
    print("女娲很厉害. 捏个泥人吹口气就成了人了")


create_people()

结果:
浇浇水
女娲很厉害. 捏个泥人吹口气就成了人了
施肥
```

我们发现, 代码运行的结果是一样的. 所谓的语法糖语法: @装饰器
类似的操作在我们生活中还有很多. 比方说. 约一约.

```
# 2--> 现在啊, 这个行情比较不好, 什么牛鬼蛇神都出来了.
# 那怎么办呢? 问问金老板. 金老板在前面给大家带路
# 这时候我们就需要在约之前啊. 先问问金老板了. 所以也要给函数添加一个功能, 这里依然可以使用装饰器
def wen_jin(fn):
    def inner():
        print("问问金老板, 行情怎么样, 质量好不好")
        fn()
        print("艹, 金老板骗我")
```

```
    return inner

@wen_jin
def yue():  # 1--> 约一约函数
    print("约一约")

yue()
```

ok, 接下来. 我们来看一下, 我约的话, 我想约个人. 比如约wusir, 这时, 我们要给函数添加一个参数

```
# 2--> 现在啊, 这个行情比较不好, 什么牛鬼蛇神都出来了.
# 那怎么办呢? 问问金老板. 金老板在前面给大家带路
# 这时候我们就需要在约之前啊. 先问问金老板了. 所以也要给函数添加一个功能, 这里依然可以使用装饰器
def wen_jin(fn):
    def inner():
        print("问问金老板, 行情怎么样, 质量好不好")
        fn()
        print("艹, 金老板骗我")
    return inner

@wen_jin
def yue(name):  # 1--> 约一约函数
    print("约一约", name)

yue("wusir")

结果:
Traceback (most recent call last):
  File "/Users/sylar/PycharmProjects/oldboy/fun_2.py", line 131, in <module>
    yue("wusir")
TypeError: inner() takes 0 positional arguments but 1 was given
```

程序报错. 分析原因: 我们在外面访问yue()的时候. 实际上访问的是inner函数. 而inner函数没有参数. 我们给了参数. 这肯定要报错的. 那么该怎么改呢? 给inner加上参数就好了

```
def wen_jin(fn):
    def inner(name):
        print("问问金老板, 行情怎么样, 质量好不好")
        fn(name)
        print("艹, 金老板骗我")
    return inner

@wen_jin
def yue(name):
    print("约一约", name)

yue("wusir")
```

这样就够了么？如果我的yue()改成两个参数呢？你是不是还要改inner. 对了. 用*args和**kwargs来搞定多个参数的问题

```
def wen_jin(fn):
    def inner(*args, **kwargs):  # 接收任意参数
        print("问问金老板，行情怎么样，质量好不好")
        fn(*args, **kwargs)    # 把接收到的内容打散再传递给目标函数
        print("艹，金老板骗我")
    return inner

@wen_jin
def yue(name):
    print("约一约", name)

yue("wusir")
```

搞定. 这时 wen_jin()函数就是一个可以处理带参数的函数的装饰器

光有参数还不够. 返回值呢?

```
def wen_jin(fn):
    def inner(*args, **kwargs):
        print("问问金老板，行情怎么样，质量好不好")
        ret = fn(*args, **kwargs)    # 执行目标函数. 获取目标函数的返回值
        print("艹，金老板骗我")
        return ret    # 把返回值返回给调用者
    return inner

@wen_jin
def yue(name):
    print("约一约", name)
    return "小萝莉"    # 函数正常返回

r = yue("wusir")    # 这里接收到的返回值是inner返回的. inner的返回值是目标函数的返回值
print(r)
```

返回值和参数我们都搞定了. 接下来给出装饰器的完整模型代码(必须记住)

```
# 装饰器: 对传递进来的函数进行包装. 可以在目标函数之前和之后添加任意的功能.
def wrapper(func):
    def inner(*args, **kwargs):
        '''在执行目标函数之前要执行的内容'''
        ret = func(*args, **kwargs)
        '''在执行目标函数之后要执行的内容'''
        return ret
    return inner

# @wrapper 相当于  target_func = wrapper(target_func) 语法糖
@wrapper
```

```
def target_func():
    print("我是目标函数")

# 调用目标函数
target_func()
```

请把以上面的代码写10遍, 并理解. 分析每一步的作用.