# 课程介绍

* 1. 装饰器（掌握）
* 2. 闭包（掌握）

1. 装饰器
   1. 什么是装饰器
      1. 装饰器的功能要求

装饰器实际是软件设计模式的一种名称，他要求具备的功能是：

* + 是一个**可调用的对象**（a callable object）
  + 在**不改变原函数代码**的基础上，**增加新的功能**

“你以为穿个马甲，我就不认识你了”，这句话其实就是说的就是装饰器的意义，在不改变原有功能的基础上，在外面套一个东西，不仅不影响原有功能，还可以增加新的动作。

  

* + 1. 装饰器的使用场景
* 场景一：（运维和测试上）
* 在原有功能基础上，增加日志记录；
* 性能测试，比如查看某个功能的执行时间，寻找最优的解决方案；
* 场景二：（开发中）
* 判断功能调用时，参数的合法性；
* 调用功能前，身份权限的验证；
* 框架代码和用户代码的衔接，比如通用功能上预留自定义功能的组合；
  + 1. 注意事项
       1. 装饰器是一个设计模式的概念，他不仅针对python语言，而且针对所有软件的架构上。
       2. 装饰器不是一个什么新的事物，可以把他简单的理解为一个工厂，他可以加工一个原函数，装饰成一个新的方式。
  1. 如何在python中实现一个装饰器
     1. 在python中，函数名是什么
* 函数名就是一个可调用对象的别名

def fun():

print("in fun...")

相当于定义了一个名字空间：{'fun': <function>}

既然fun是一个名称，那就可以像普通变量一样，传递或重指向。

* 函数名可以赋值，也可以被重命名
  + 1. 一个最简单的装饰行为

饰器的一个特点是：在执行原函数的基础上，增加内容。

❓ 那么，如何执行原函数那？

✔ 把原函数的名字交给装饰函数，装饰函数就具备了执行老函数的能力。

# 旧函数，待装饰的函数

def fun():

print('我是老函数')

def factory(old\_fun):

print('='\*20)

old\_fun()

print('='\*20)

factory(fun)

* + 1. 完善装饰行为

❓ 上面的装饰行为还有哪些问题？

◇ 需要通过装饰工厂来调用原函数，原来调用这个功能的业务部门，将大面积修改代码；

❓ 那么如何改进这个问题那？

如何设计一个方式，在不改变原函数名称的情况下，调用时也执行装饰工厂那。

◇ old\_name = factory(old\_name)

* + 1. python的装饰器语法糖

old\_name = factory(old\_name)这句话就是生成装饰器的核心语句。为了避免每次都写这样的无脑代码，python提供了一个便捷的写法，我们称之为语法糖。

语法糖的写法：

· 找到要装饰的函数

· 在上面写上@，然后跟装饰器函数的名称

@factory

def fun():

print('in fun() ...')

* + 1. 总结

python的装饰器写法思路：

* + - 1. 定义一个中介函数，该函数命名为装饰行为，接收一个可调用对象

def get\_timer(fn):

pass

* + - 1. 在中介函数内部实现装饰行为函数

def get\_timer(fn):

def wrapper():

... ... # 装饰代码实现

* + - 1. 在中介函数最后返回装饰行为的函数名：

def get\_timer(fn):

def wrapper():

... ... # 装饰代码实现

**return wrapper**

**【注意】：返回的是函数名，而不是函数调用**

* + - 1. 在需要被装饰的函数前，定义@装饰器名称

@get\_timer

def fun():

pass

* 1. python装饰器的几种形式
     1. 无参数装饰器

Demo1: 增加函数什么时候运行的功能:

import time

def time\_fun(func):

def my\_time():

print(f"{func.\_\_name\_\_} running at {time.ctime()}")

func()

return my\_time

@time\_fun

def fun1():

print("+++++++++")

fun1()

time.sleep(2)

fun1()

* + 1. 被装饰函数有参数
* 装饰器的核心是内部函数的行为被重命名；
* 内部函数的形式要满足原函数形式；

import time

def time\_fun(func):

def my\_time(arg1): # 内部函数里预留接口

print(f"{func.\_\_name\_\_} running at {time.ctime()}")

func(arg1) # 原函数传递参数

return my\_time

@time\_fun

def fun1(arg1):

print("+++++{}++++".format(arg1))

fun1(10)

time.sleep(2)

fun1(10)

* + 1. 被装饰函数有不定参数和返回值

原函数有返回值，内部装饰函数也接收返回值进行返回。

import time

def time\_fun(func):

def my\_time(\*args, \*\*kwargs):

print(f"{func.\_\_name\_\_} running at {time.ctime()}")

func(\*args, \*\*kwargs)

return my\_time

@time\_fun

def fun1(arg1, arg2, name):

print("+++++{},{},{}++++".format(arg1, arg2, name))

fun1(10, 20, name='rocky')

time.sleep(2)

fun1(10, 20, name='jim')

* + 1. 改变装饰器行为

在装饰器中，我们也有一种需求，根据传入的参数不同，装饰的行为方式也会不同，那么如何定义这种装饰器那。

首先再理解装饰器的语法：

@time\_fun

def fun1(arg1, arg2, name):

print("+++++{},{},{}++++".format(arg1, arg2, name))

原材料名称= 装饰器名称(原材料名称)

@time\_fun(“itsource”)

def fun1(arg1, arg2, name):

print("+++++{},{},{}++++".format(arg1, arg2, name))

fun1 = time\_fun(“itsource”)(fun1)

这样来看，必须要保证time\_fun(“itsource”)返回的内容恰好是标准的装饰器就可以满足了。

import time

def time\_fun(flags):

def time\_arg(func):

def my\_time(\*args, \*\*kwargs):

print(f"{func.\_\_name\_\_} running at {time.ctime()}")

print("the flags is {}".format(flags))

return func(\*args, \*\*kwargs)

return my\_time

return time\_arg

@time\_fun("itsource")

def fun1(arg1, arg2, name):

print("+++++{},{},{}++++".format(arg1, arg2, name))

fun1(10, 20, name='rocky')

time.sleep(2)

fun1(10, 20, name='jim')

* + 1. 用类来定义装饰器（了解）

❓ 可执行对象究竟是什么？

✔ 具有\_\_call\_\_方法的对象空间，就称之为可执行对象，每定义一个函数，就相当于在空间中定义了\_\_call\_\_方法。

❓ 如何利用类来实现装饰器

@Test # fun = Test(fun)

def fun():

pass

* Test的\_\_init\_\_方法应该接收被装饰函数的名称
* 调用fun()实际就是调用了Test类的\_\_call\_\_方法

class Test():

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

self.old\_fun = fn

def \_\_call\_\_(self,):

... ... # 装饰内容

ret = self.old\_fun()

@Test

def fun(a, b, name):

pass

fun()

* 1. 多层装饰

在实际开发时，一个原函数，可以使用多个装饰函数进行修饰，这种行为就称之为多层装饰。

对于多层装饰，要理解他的执行过程。

def bold(fn):

def wrapper():

return f'<b>{fn()}</b>'

return wrapper

def italic(fn):

def wrapper():

return f'<i>{fn()}</i>'

return wrapper

@italic

@bold

def hello():

return 'Hello World'

print(hello())

【结果】：

<i><b>Hello World</b></i>

* 1. 总结
     1. 使用装饰器

提供了一个语法糖,

@装饰器函数名称,这个语法糖下面必须跟def的普通函数

@装饰器函数名称

def fun():

pass

* + 1. 如何设计一个装饰器
* 装饰器实际也是一个函数（可调用对象）；
* 接收一个老函数的名称；
* 返回新函数的名称；
* 在新函数里调用老函数；

1. 闭包
   1. 什么是闭包
      1. 闭包的定义

闭包，实际是一个特殊的函数，他的特点是：

* 一个函数A里定义了另外一个函数B；
* 函数B里使用了函数A作用域里的标识符；
* 函数B给外部使用；
  + 1. 闭包的原理
* \_\_closure\_\_属性：

普通函数：None

闭包： 元组，数量等于引用外部函数作用域的标识符

* cell\_contents：

闭包\_\_closure\_\_里保存的cell元素的属性

保存了引用外部函数作用域标识符

* 1. 闭包的定义方法
* 在外部函数A里定义一个内部函数B
* 内部函数B里使用了外部函数A的作用域变量
* 外部函数A返回了内部函数B的名字
  1. 注意事项

看下一个常见的面试题例子：

def count1():

fs = []

for i in range(1, 4):

def f():

return i \* i

fs.append(f)

return fs

f1, f2, f3 = count1()

print(f1())

print(f2())

print(f3())

分析为什么答案都是9，9，9；

* 1. 闭包的应用场景（了解）
     1. 延迟执行、惰性函数

由于使用的是代码段返回的方式，换句话，就是这个功能的执行是被延迟执行的。那这个闭包有时又称之为惰性函数。

比如我有一个非常耗时的计算操作，在系统初始化的时候，如果执行了，那么会影响系统的启动速度，这个时候，就可以使用闭包的惰性特性，来规避这样的问题。

def lazy\_sum(\*args):

def mysum():

count = 0

for i in args:

count += i

return count

return mysum

a = lazy\_sum(\*(range(10000001)))

print(a)

print(a())

* + 1. 保持一个状态稍后使用，装饰器的核心

在保持状态的过程中，闭包是如何和状态关联的那？是引用还是拷贝那？

分析下面的代码：

def lazy\_sum(arg):

def mysum():

count = 0

for i in arg:

count += i

return count

return mysum

buf = [1, 2, 3, 4]

a = lazy\_sum(buf)

buf.append(20)

print(a)

print(a())

1. 课程总结
   1. 重点
      * 1. 装饰器的语法糖写法及该写法代表的语义
        2. 读懂多层装饰器的语义
        3. 闭包的语法
   2. 难点
      * 1. 装饰器的写法，返回函数的代码实现；
        2. 带参数的装饰器写法；
        3. 闭包的使用场景；
   3. 如何掌握？
      * 1. 临摹课堂演示代码，先读懂课堂代码，翻译成中文意思。
        2. 先抄写代码，然后只看中文意思，写出对应代码。
2. 课后练习
   * + 1. 装饰器的代码抄写，注意写明注释
       2. 实现一个统计执行时间的装饰器，并写出测试用例。提示，可以统计计算100以内数字的平方和100000以内数字平方的差别。（选做）
       3. 闭包应用场景代码的抄写
       4. 实现一个直线的闭包，通过这个闭包可以得到直线的坐标。(选做)

提示，直线的坐标公式是y = kx + b。

1. 面试题
   * + 1. **这个代码执行的结果是 \_\_\_\_\_.**

[i\*\*i for i in xrange(3)]

* + - 1. **关于Python内存管理，下列说法错误的是（ ）。**

A. 变量不必事先声明 B. 变量无须先创建和赋值而直接使用

C. 变量无须指定类型 D. 可以使用del释放资源

* + - 1. **下面的代码运行结果是（ ）。**

a = 1

try:

a += 1

expect:

a += 1

else:

a += 1

finally:

a += 1

print a

* + - 1. **下面的打印结果是（ ）**

counter = 1

def doLotsOfStuff():

global counter

for i in (1, 2, 3):

counter += 1

doLotsOfStuff()

print(counter)

* + - 1. **一段代码定义如下，下列调用结果正确的是（ ）。**

def bar(multiple):

def foo(n):

return multiple \*\* n

return foo

A. bar(2)(3) == 8 B. bar(2)(3) == 6

C. bar(3)(2) == 8 D. bar(3)(2) == 6

6. 如何解释@mytime的含义？

答：这是python的一个装饰器语法糖，是给一个函数进行装饰的简略写法。他的含义是：

原函数名=mytime(原函数名)

7. 闭包的含义？

答： 一个函数A里定义了另外一个函数B；

函数B里使用了函数A作用域里的标识符；

函数B给外部使用；

函数B里定义了\_\_closure\_\_属性来保存A里的值

1. 扩展知识或课外阅读推荐（可选）
   1. 扩展知识
      1. functools.wraper

在上面的装饰器写法，已经可以满足工作中的需求了，不过他也有一个缺点，就是会丢失掉源函数的元信息，比如\_\_name\_\_,\_\_docstring\_\_等，那么如何解决那，可以利用python的functools模块下的wraper的装饰器。

"""

· 使用wraps装饰器，传入原函数信息，他将帮我们保存原函数的所有元信息

· 用这个装饰器修饰内部装饰函数

"""

import time

from functools import wraps

def my\_time(fn):

@wraps(fn)

def wrapper():

print(f"{fn.\_\_name\_\_} 运行在 {time.ctime()}")

fn()

return wrapper

@my\_time

def fun():

print('===')

fun()

print(fun.\_\_name\_\_)