# Python装饰器

## 什么是装饰器？

**装饰**是为函数和类指定管理代码的一种方式。**装饰器**本身的形式是处理其他（返回）**可调用对象的可调用对象。**

* 函数装饰器在函数定义的时候进行名称重绑定，提供一个逻辑层来管理函数和方法或随后对它们的调用。
* 类装饰器在类定义的时候进行名称重绑定，提供一个逻辑层来管理类，或管理随后调用它们所创建的示例。

装饰器提供了一种方法，在函数和类定义语句的末尾插入**自动运行代码**。

### 管理调用和实例

* 函数装饰器安装包装器对象，以在需要的时候拦截随后的函数调用并处理它们。
* 类装饰器安装包装器对象，以在需要的时候拦截随后的实例创建调用并处理它们。

### 管理函数和类

* 函数装饰器也可以用来管理函数对象，而不是随后对它们的调用。
* 类装饰器可以用来直接管理类对象，而不是实例创建调用。

### 为什么使用装饰器

* 装饰器有一种非常明确的语法，比那些辅助函数更容易使用和维护。
* 当主体函数或类定义的时候，装饰器应用一次；在对类或函数的每次调用的时候，不必添加额外的代码。
* 装饰器使得一个API的用户不太可能忘记根据API需求扩展一个函数或类。

## 基础知识

### 函数装饰器

函数装饰器自动将如下语法：

@decorator #装饰函数

def F(arg): #主体函数

...

F(99)

映射为这一对等的形式，其中装饰器是一个单参数的可调用对象，它返回与F具有相同数目的参数的一个可调用对象：

def F(arg):

...

F = decorator(F)

F(99)

**装饰器自身是一个返回可调用对象的可调用对象**。

### 类装饰器

@decorator #装饰类

class C:

...

x = C(99)

等同于

class C:

...

C = decorator(C)

x = C(99)

**类装饰器也是返回一个可调用对象的可调用对象。**

def **decorator**(cls):

class **Wrapper**:

def **\_\_init\_\_**(*self*, \*args):

*self*.warpped = cls(\*args)

def **\_\_getattr\_\_**(*self*, name):

return getattr(*self*.warpped, name)

return Wrapper

*@decorator*

class **C**:

def **\_\_init\_\_**(*self*, x, y):

*self*.attr = *'spam'*

这段代码处理多个被装饰的类（每个都产生一个新的D e c o r a t o r实例） ， 并且会拦截实例创建调用。

class **Decorator**:

def **\_\_init\_\_**(*self*, C):

*self*.C = C

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

*self*.warpped = *self*.C(\*args)

return *self*

def **\_\_getattr\_\_**(*self*, attrname):

return getattr(*self*.warpped, attrname)

*@Decorator*

class **A**:

def **\_\_init\_\_**(*self*, x,y):

*self*.new = *'hello'*

*X = A(1,2)*

*Y = A(1,2)*

*X与Y实际是同一对象*

### 装饰器嵌套

@A

@B

@C

def F(arg):

...

相当于：

F = A((B(C(F))))

例：

def **d1**(F):return lambda: *'X'*+ F()

def **d2**(F):return lambda: *'Y'* + F()

def **d3**(F):return lambda: *'Z'* + F()

*@d1*

*@d2*

*@d3*

def **func**():

return *'spam'*

print func()

### 装饰器参数

@decorator(A, B)

def F(arg):

...

F(99)

自动映射到其对等的形式：

def F(arg):

...

F = decorator(A, B)(F)

F(99)

装饰器参数在装饰之前就解析了，并且其通常保存状态信息供随后的调用使用。

装饰器参数往往意味着可调用对象的三个层级：接受装饰器参数的一个可调用对象，它返回一个可调用对象以作为装饰器，该装饰器返回一个可调用对象来处理最初的函数或类的调用。

例：

def **decorator**(A, B):

print A+B

def **actualDecorator**(F):

return F

return actualDecorator

*@decorator*(1,2)

def **func**(x):

print x

func(1)

## 编写函数装饰器

### 跟踪调用

class **Tracer**:

def **\_\_init\_\_**(*self*, func):

*self*.calls = 0

*self*.func = func

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

*self*.calls += 1

print *'call %s to %s'* % (*self*.calls, *self*.func.\_\_name\_\_)

*self*.func(\*args)

*@Tracer*

def **spam**(a, b, c):

print a+b+c

### 计时调用

import time

class **Timer**:

def **\_\_init\_\_**(*self*, func):

*self*.func = func

*self*.alltime = 0

def **\_\_call\_\_**(*self*, \*args, \*\*kwargs):

start = time.clock()

result = *self*.func(\*args, \*\*kwargs)

elapsed = time.clock() - start

*self*.alltime += elapsed

print *'%s:%.5f, %.5f'* % (*self*.func.\_\_name\_\_, elapsed, *self*.alltime)

*@Timer*

def **listcomp**(N):

return [x\*2 for x in range(N)]

*@Timer*

def **mapcall**(N):

return map((lambda x: x\*2), range(N))

result = listcomp(5)

listcomp(50000)

listcomp(500000)

listcomp(1000000)

listcomp(10000000)

print listcomp.alltime

### 添加装饰器参数

def **timer**(label=*''*, trace = True):

class **Timer**:

def **\_\_init\_\_**(*self*, func):

*self*.func = func

*self*.alltime = 0

def **\_\_call\_\_**(*self*, \*args, \*\*kwargs):

start = time.clock()

result = *self*.func(\*args, \*\*kwargs)

elapsed = time.clock() - start

*self*.alltime += elapsed

if trace:

print *'%s %s:%.5f, %.5f'* % (label, *self*.func.\_\_name\_\_,

elapsed, *self*.alltime)

return Timer

## 编写类装饰器

### 单体类

#-\*- coding:utf-8 -\*-

instances = {}

def **getInstance**(aClass, \*args):

if aClass not in instances:

instances[aClass] = aClass(\*args)

return instances[aClass]

def **singleton**(aClass):

def **onCall**(\*args):

return getInstance(aClass, \*args)

return onCall

*@singleton*

class **Person**:

def **\_\_init\_\_**(*self*, name, hours, rate):

*self*.name = name

*self*.hours = hours

*self*.rate = rate

def **pay**(*self*):

return *self*.hours \* *self*.rate

*@singleton*

class **Spam**:

def **\_\_init\_\_**(*self*, val):

*self*.attr = val

bob = Person(*'Bob'*, 50, 20)

print bob.name, bob.pay()

sue = Person(*'Sue'*, 50, 20)

print sue.name, sue.pay()

X = Spam(42)

Y = Spam(99)

print X.attr

print Y.attr

执行结果：

Bob 1000

Bob 1000

42

42