装饰：处理其他的可调用对象（如函数，类）

装饰器本质是一种语法糖，并不是必须的。

装饰器本身需要返回一个可调用对象。

装饰器的作用：在不更改函数源代码的情况下，给函数增加额外的功能。例如插入日志，性能测试等。

函数装饰器：在函数定义的时候进行名称重绑定。

@decorator

def func(args): …

Python会自动将其翻译成：装饰器定义函数，会立刻执行。

参考：<https://blog.csdn.net/jyhhhhhhh/article/details/54627850>

def func(args): …

func = decorator(func)

可以看出，装饰器decorator接受了一个函数参数。

decorator可以是Python提供的装饰器，也可以是自定义的装饰器。

例：程序class\_detail/class\_detail16.py

class Spam:  
 number\_count = 0  
  
 def \_\_init\_\_(self):  
 Spam.number\_count += 1  
  
 # 函数装饰器定义静态方法  
 @staticmethod  
 def print\_number\_count():  
 print('Number of instances created: ', Spam.number\_count)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 a = Spam()  
 b = Spam()  
 c = Spam()  
 Spam.print\_number\_count()

输出为：

Number of instances created: 3

例：程序decorator\_test/decorator\_test1.py

def log(func):  
 def wrapper(\*args, \*\*kwargs):  
 print('call {0}(): '.format(func.\_\_name\_\_))  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
  
# Python解析为add = log(add)  
# log返回函数对象wrapper  
# add = wrapper  
# add(x, y)等同于wrapper(x, y)  
# 类似Python学习手册第17章的工厂函数  
@log  
def add(x, y):  
 return x + y  
  
  
class Sum:  
 @log  
 def method(self, x, y):  
 return x + y  
  
  
def log1(func):  
 print('call {0}(): '.format(func.\_\_name\_\_))  
 return func  
  
  
# 相当于sub = log1(sub)  
# log1被调用  
@log1  
def sub(x, y):  
 return x - y  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print(add(3, 5))  
 print()  
  
 sum1 = Sum()  
 print(sum1.method(4, 6))  
  
 print(sub(1, 3))

输出为：

call sub():

call add():

8

call method():

10

-2

从输出可以看出：装饰器定义函数后，会立刻执行，log函数和log1函数在被装饰的函数调用前就执行，所以log1输出了call sub():

装饰器的顺序：

例：程序decorator\_test/decorator\_test2.py

def decorator\_a(func):  
 print('Get in decorator\_a')  
  
 def inner\_a(\*args, \*\*kwargs):  
 print('Get in inner\_a')  
 return func(\*args, \*\*kwargs)  
 return inner\_a  
  
  
def decorator\_b(func):  
 print('Get in decorator\_b')  
  
 def inner\_b(\*args, \*\*kwargs):  
 print('Get in inner\_b')  
 return func(\*args, \*\*kwargs)  
 return inner\_b  
  
  
@decorator\_b  
@decorator\_a  
def func(x):  
 print('Get in func')  
 return x \* 2

输出为：

Get in decorator\_a

Get in decorator\_b

可以看出，即使不执行函数func的调用，仍然有输出，装饰器定义函数会立刻执行，装饰器函数decorator\_a和decorator\_b会执行输出。

添加测试代码：

if \_\_name\_\_ == '\_\_main\_\_':  
 print(func(2))

输出为：

Get in decorator\_a

Get in decorator\_b

Get in inner\_b

Get in inner\_a

Get in func

4

可以看出，先输出inner\_b，后输出inner\_a，原因是：

@decorator\_b  
@decorator\_a  
def func(x):

Python解释器解释成func = decorator\_b(decorator\_a(func))，执行func(2)时，inner\_b先被调用，然后是inner\_a，最后是原始的func。

装饰器可以是函数，也可以是类。

例：程序decorator\_test/decorator\_test3.py

class Decorator:  
 def \_\_init\_\_(self, func):  
 print('init Decorator')  
 self.func = func  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print('Decorator call')  
 return self.func(\*args, \*\*kwargs)  
  
  
@Decorator  
def func(x, y):  
 return x + y

@Decorator  
def func(x, y):

相当于func = Decorator(func)，会执行Decorator的构造函数

添加测试代码：

if \_\_name\_\_ == '\_\_main\_\_':  
 print(func(3, 5))

输出为：

init Decorator

Decorator call

8

装饰器定义类

例：程序decorator\_test/decorator\_test4.py

def decorator(cls):  
 class Wrapper:  
 def \_\_init\_\_(self, \*args):  
 print('init Wrapper')  
 self.wrapped = cls(\*args)  
  
 def \_\_getattr\_\_(self, name):  
 print('get attr in Wrapper')  
 return getattr(self.wrapped, name)  
 return Wrapper  
  
  
# 类装饰器  
@decorator  
class Spam:  
 def \_\_init\_\_(self, x, y):  
 self.attr = 'spam'  
 self.x = x  
 self.y = y  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 a = Spam(6, 7)  
 print(a.attr)

输出为：

init Wrapper

get attr in Wrapper

spam

通过装饰器构建一个单例模式

例：程序decorator\_test/decorator\_test5.py

class Singleton:  
 def \_\_init\_\_(self, aClass):  
 print('init Singleton')  
 self.aClass = aClass  
 self.instance = None  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 if self.instance is None:  
 self.instance = self.aClass(\*args, \*\*kwargs)  
 return self.instance  
  
  
@Singleton  
class Person:  
 def \_\_init\_\_(self, name, hours, rate):  
 print('init Person')  
 self.name = name  
 self.hours = hours  
 self.rate = rate  
  
 def pay(self):  
 return self.hours \* self.rate  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print()  
 print('Start Test')  
 bob = Person('Bob', 40, 10)  
 print(bob.name, bob.pay())  
 print()  
 sue = Person('Sue', 50, 20)  
 print(sue.name, sue.pay())

print('bob id', id(bob))  
print('sue id', id(sue))  
print('End test')

输出为：

init Singleton

Start Test

init Person

Bob 400

Bob 400

bob id 2349488723056

sue id 2349488723056

End test

从输出可以看出，尽管在代码中创建了两个Person，但是Person的构造函数只执行了一次，创建的两个Person对象都是同一个。

实现类的私有属性

简单的实现：不使用装饰器

例：程序decorator\_test/decorator\_test6.py

class PrivateExc(Exception):  
 pass  
  
  
class Privacy:  
 def \_\_setattr\_\_(self, key, value):  
 if key in self.privates:  
 raise PrivateExc(key, self)  
 else:  
 self.\_\_dict\_\_[key] = value  
  
  
class Test(Privacy):  
 privates = ['age', 'name'] # 类的属性  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 x = Test()  
 # x.name = 'Bob' # 抛出异常  
 # x.age = 20 # 抛出异常

带有参数的装饰器：

如果装饰器带有参数：

@decorator(a, b)

def func(args): …

Python会自动翻译成：

func = decorator(a, b)(func)

从上可以看出，装饰器decorator其实返回是实际的装饰器，这个实际的装饰器接收函数参数func，所以不同于不带参数的装饰器的两层def，一般带有参数的装饰器需要3层def

def decorator:

def actual\_decorator(func):

def f()

return f

return actual\_decorator

例：程序decorator\_test/ decorator\_test8.ipynb

## 带有参数的装饰器

def repeat\_func(n):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

print("before function run")

for \_ in range(n):

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

print("after function run")

return wrapper

return decorator

@repeat\_func(n=5)

def func():

print('function run')

func()

输出：

before function run

function run

function run

function run

function run

function run

after function run