如果你有一个函数，如下，

def funA(\*args\_funA):

dosomething

现在，

你需要往函数加更多的内容，并且不想改变原函数；

你需要对输入参数做一些额外的操作；

你需要有条件的对函数进行调用；

你需要对输出结果做进一步处理；

怎么办？

你原来的想保留的函数

扩展功能 Part A，可以理解为在调用function\_args之前处理；后面还可以有之后的处理

函数类型参数，如果你懂C++, 那它就是函数指针

这个是本来函数的参数，也就是function\_args函数的参数

def Decorator(function\_args):

def wrapper(\*args):

do other things here;

function\_args(\*args)

do other other things more

return wrapper

好了，现在你把你本来的函数传给Decorator(), 返回一个新函数，这个新函数就是你扩展之后的新工具啦！

还有一个好处，这个Decorator可以包装一系列的函数呢，类似一个Tempalte.

使用方法，如下：

Def old\_function():

Dosomething

Def decorator(function\_pointer):

Def wrapper(\*args):

More thing to do;

function\_pointer(\*args)

return wrapper

使用步骤：

Newfunction = decorator(old\_function) #这个decorator可以接受好多类型的函数哦

Newfunction(\*args)

基本原理就是这样，只不过还有点小小的有关python的语法问题，你如果print(Newfunction)，打印出来的是wrapper的信息，我们还想让它显示old\_function的信息，这样呢，就借助了python的一个库，具体做法如下

from functools import wraps

Def old\_function():

Dosomething

Def decorator(function\_pointer):

@wraps(function\_pointer)

Def wrapper(\*args):

More thing to do;

function\_pointer(\*args)

return wrapper

使用步骤：

Newfunction = decorator(old\_function) #这个decorator可以接受好多类型的函数哦

Newfunction(\*args)

继续看python的便利，每次装饰一个函数都通过

Newfunction = decorator(old\_function)

很麻烦吧，好了，python的语法来解决，Look！

from functools import wraps

@decorator

Def old\_function():

Dosomething

Def decorator(function\_pointer):

@wraps(function\_pointer)

Def wrapper(\*args):

More thing to do;

function\_pointer(\*args)

return wrapper

使用步骤：

old\_function(\*args), #目的达到啦！

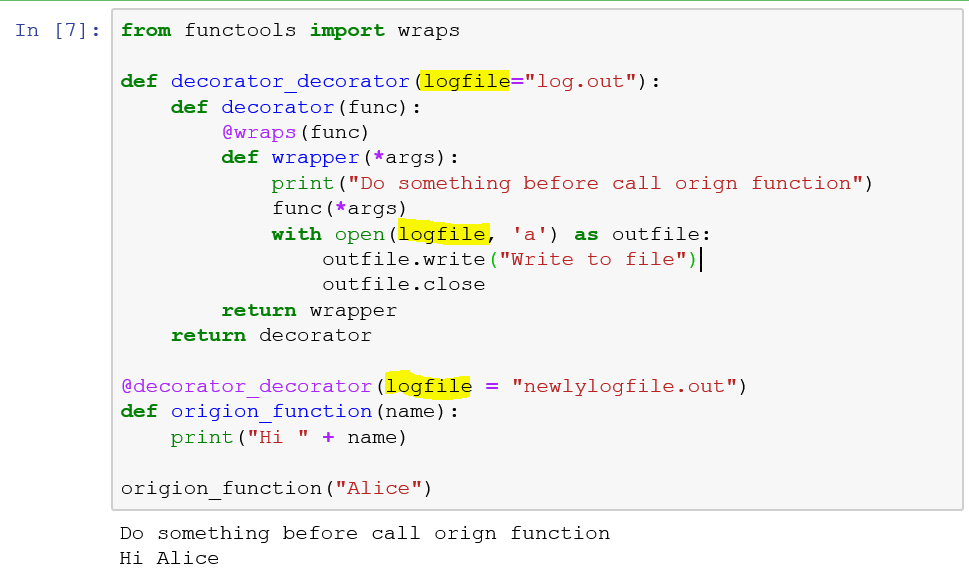
继续来研究！

如果要给你的wrapper传个参数怎么办？

通过上面的例子知道，wrapper的参数是你的old\_function的参数，而decorator的参数你的old\_function，如果你的wrapper里面的逻辑，要些参数怎么办？

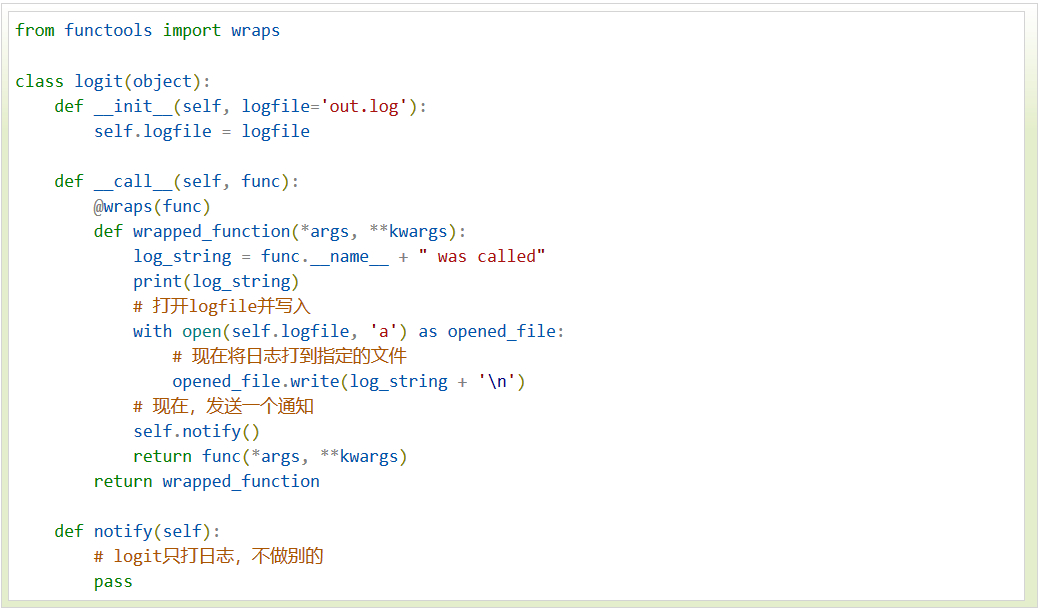
简单，再加一次wrapper!

下面的例子有点老套，就是打印日志到文件，文件名通过装饰器来实现。



最后一点，来谈谈装饰器类！

说白了就是重载了该类的\_\_call\_\_函数，而且还能继承进一步扩展，摘录2个网络上的例子。



用起来很简单。



关于继承，这里扩展了原来类的notify函数。



好了，基本的都全了。