装饰器

这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

--我们要定义一个能打印日志的decorator，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

观察上面的log，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

```
@log
def now():
    print('2015-3-25')
```

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

```
>>> now()
call now():
2015-3-25
```

--如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个3层嵌套的decorator用法如下：

```
@log('execute')
def now():
    print('2015-3-25')
```

执行结果如下：

```
>>> now()
execute now():
2015-3-25
```

--以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有__name__等属性，但你去看经过decorator装饰之后的函数，它们的__name__已经从原来的'now'变成了'wrapper'：

```
>>> now.__name__
'wrapper'
```

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的__name__等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写wrapper.__name__ = func.__name__这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的decorator：

```
import functools
```

```
def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

import functools是导入functools模块。模块的概念稍候讲解。

现在，只需记住在定义wrapper()的前面加上@functools.wraps(func)即可。