9.1 在函数上添加包装器¶

问题¶

你想在函数上添加一个包装器，增加额外的操作处理(比如日志、计时等)。

解决方案¶

如果你想使用额外的代码包装一个函数，可以定义一个装饰器函数，例如：

```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

下面是使用装饰器的例子：

```
>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>
```

讨论¶

一个装饰器就是一个函数，它接受一个函数作为参数并返回一个新的函数。 当你像下面这样写：

```
@timethis
def countdown(n):
    pass
```

跟像下面这样写其实效果是一样的：

```
def countdown(n):
    pass
countdown = timethis(countdown)
```

顺便说一下，内置的装饰器比如 `@staticmethod`, `@classmethod`,`@property` 原理也是一样的。 例如，下面这两个代码片段是等价的：

```
class A:
    @classmethod
    def method(cls):
        pass
```

```
class B:
    # Equivalent definition of a class method
    def method(cls):
        pass
    method = classmethod(method)
```

在上面的 `wrapper()` 函数中， 装饰器内部定义了一个使用 `*args` 和 `**kwargs` 来接受任意参数的函数。 在这个函数里面调用了原始函数并将其结果返回，不过你还可以添加其他额外的代码(比如计时)。 然后这个新的函数包装器被作为结果返回来代替原始函数。

需要强调的是装饰器并不会修改原始函数的参数签名以及返回值。 使用 `*args` 和 `**kwargs` 目的就是确保任何参数都能适用。 而返回结果值基本都是调用原始函数 `func(*args, **kwargs)` 的返回结果，其中func就是原始函数。

刚开始学习装饰器的时候，会使用一些简单的例子来说明，比如上面演示的这个。 不过实际场景使用时，还是有一些细节问题要注意的。 比如上面使用 `@wraps(func)` 注解是很重要的， 它能保留原始函数的元数据(下一小节会讲到)，新手经常会忽略这个细节。 接下来的几个小节我们会更加深入的讲解装饰器函数的细节问题，如果你想构造你自己的装饰器函数，需要认真看一下。