

# 第 1 章

## 装饰器

装饰器是一个用于封装函数或类的代码的工具。它显式地将封装器应用到函数或类上，从而使它们选择加入到装饰器的功能中。对于在函数运行前处理常见前置条件(例如确认授权)，或在函数运行后确保清理(例如输出清除或异常处理)装饰器都非常有用。对于处理已经被装饰的函数或类本身，装饰器也很有用。例如，装饰器可以将函数注册到信号系统，或者注册到 Web 应用程序的 URI 注册表中。

本章将概要介绍什么是装饰器，以及装饰器如何与 Python 的函数和类交互。本章还列举了几个 Python 标准类库中常见的装饰器。最后，本章提供了编写装饰器并将其附加到函数和类上的指南。

### 1.1 理解装饰器

究其核心而言，装饰器就是一个可以接受调用也可以返回调用的调用。装饰器无非就是一个函数(或调用，如有 `_call_method_` 方法的对象)，该函数接受被装饰的函数作为其位置参数。装饰器通过使用该参数来执行某些操作，然后返回原始参数或一些其他的调用(大概以这种方式与装饰器交互)。

由于函数在 Python 中是一级对象，因此它们能够像其他对象一样被传递到另一个函数。装饰器就是接受另一个函数作为参数，并用其完成一些操作的函数。

实际上这很容易理解。考虑下面一个非常简单的装饰器。它仅仅为被装饰的调用点字符串附加了一个字符串。

```
def decorated_by(func):  
    func.__doc__ += '\nDecorated by decorated_by.'  
    return func
```

现在，考虑下面这个普通的函数：

```
def add(x, y):  
    """Return the sum of x and y."""  
    return x + y
```

函数的 `docstring` 是在第一行指定的字符串。假如在 Python 的 Shell 中对该函数运行 `Help` 命令，就能看到该字符串。下面是将装饰器应用到 `add` 函数的示例：

```
def add(x, y):  
    """Return the sum of x and y."""  
    return x + y  
add = decorated_by(add)
```

以下是执行 `help` 命令得到的结果：

```
Help on function add in module __main__:  
  
add(x, y)  
    Return the sum of x and y.  
    Decorated by decorated_by.  
(END)
```

这里发生了什么？其实就是装饰器修改了 `add` 函数的 `_doc_` 属性，然后返回原来的函数对象。

## 1.2 装饰器语法

大多数时候开发人员使用装饰器来装饰函数，他们只对装饰过的最终函数感兴趣，而对于未装饰函数的引用最终就变得多余。

正因为如此(也是为了更整洁)，所以定义函数，给它赋一个特定的名称，然后立刻将装饰过的函数赋给相同的名称就不可取。

因此，Python 2.5 为装饰器引入了特殊的语法。装饰器的应用是通过在装饰器的名称前放置一个 `@` 字符，并在被装饰函数声明之上添加一行(不包含隐式装饰器的方法签名)来实现的。

下面来看一下如何优先将 `decorated_by` 装饰器应用到 `add` 方法：

```
@decorated_by  
def add(x, y):  
    """Return the sum of x and y."""  
    return x + y
```

再次注意，这里没有给 `@decorated_by` 提供方法签名。假设该装饰器有一个单独的位置参数，而这个参数是装饰过的方法(在某些情况下，会看到一个带有其他参数的方法签名，该内容将在本章后面讨论)。

该语法允许在声明函数的位置应用装饰器，从而代码更容易阅读并且可以立即意识到应用了装饰器。可读性很重要。

## 装饰器应用的顺序

何时应用装饰器？使用@语法时，在创建被装饰的可调用函数后，会立刻应用装饰器。因此以上两个示例中所展示的将 `decorated_by` 应用到 `add` 的方式几乎是一样的。首先创建 `add` 函数，然后立即使用 `decorated_by` 将其封装起来。

需要注意的重要一点是，对某个可调用函数，可以使用多个装饰器(就像可以多次封装函数调用一样)。

但请注意，如果通过@语法使用多个装饰器，就需要按照自底向上的顺序来应用它们。起初觉得这违反直觉，但是恰恰说明了 Python 解释器实际所做的工作。

考虑下面这个应用了两个装饰器的函数：

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

首先发生的是由解释器创建 `add` 函数，然后应用 `decorated_by` 装饰器。该装饰器返回了一个可调用函数(正如所有装饰器做的一样)，该函数被发送给 `also_decorated_by` 装饰器，`also_decorated_by` 也做了同样的事情，接下来结果被赋给 `add` 函数。

切记，装饰器 `decorated_by` 的应用程序与下面的代码在语法上是相同的：

```
add = decorated_by(add)
```

前面两个装饰器示例与下面的代码在语法上相同：

```
add = also_decorated_by(decorated_by(add))
```

在这两种情况下，读取代码时首先读到装饰器 `also_decorated_by`。但是，装饰器的应用是自底向上的，这与函数的解析(由内向外)是相同的。工作也采用同样的原则。

在传统的函数调用情况下，解释器一定首先解析内部函数调用，以便有合适的对象或值发送给外部调用。

```
add = also_decorated_by(decorated_by(add)) # First, get a return value for
# `decorated_by(add)`.
add = also_decorated_by(decorated_by(add)) # Send that return value to
# `also_decorated_by`.
```

有了装饰器后，通常首先创建 `add` 函数。

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

然后，调用装饰器@decorated\_by，并将其作为 add 函数的装饰方法。

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

@decorated\_by 函数返回自己的可调用函数(在本例中，是 add 的修改版本)。该返回值在最后步骤发送给@also\_decorated\_by。

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

应用装饰器时需要记住一件重要的事情，即装饰器的应用是自底向上的。很多时候，顺序非常重要。

### 1.3 在何处使用装饰器

Python 标准库中包括很多包含装饰器的模块，并且很多常用工具和框架利用它们实现常用功能。

例如，如果要使一个类上的方法不需要这个类的实例，可以使用@classmethod 或@staticmethod 装饰器，它们是标准库的一部分。mock 模块(用于单元测试，在 Python 3.3 以后被添加到标准库中)允许使用@mock.patch 或@mock.patch.object 作为装饰器。

一些常见工具也使用装饰器。Django(用于 Python 的常见 Web 框架)使用@login\_required 作为装饰器，允许开发人员指定用户必须登录才能查看一个特定页面，并且使用@permission\_required 应用更具体的权限限制。Flask(另一个常见的 Web 框架)使用@app.route 充当指定的 URI 与浏览器访问这些 URI 时所运行的函数之间的注册表。

Celery(常见的 Python 任务运行工具)使用复杂的@task 装饰器来标识函数是否为异步任务。该装饰器实际上返回 Task 类的实例，用来阐明如何使用装饰器制作一个方便的 API。

### 1.4 编写装饰器的理由

装饰器提供了一种绝妙的方式来告知，“在指定的位置，我想要这个指定的可重用的功能片段”。当装饰器编写得足够好时，它们是模块化且清晰明确的。

装饰器的模块化(可以很容易地从函数或类中使用和移除装饰器)使它们完美地避免重复前置和收尾代码。同样，因为装饰器与装饰函数自身交互，所以它们善于在其他地方注册函数。

另外，装饰器是显式的。它们在所有需要它们的被调用函数中即席使用。因此这对于可读性很有价值，从而使调试也变得更方便。被应用的位置以及被应用的内容都非常明显。

## 1.5 编写装饰器的时机

在 Python 的应用程序和模块中有几个很好的有关编写装饰器的用例。

### 1.5.1 附加功能

大概使用装饰器最常见的理由是想在执行被装饰方法之前或之后添加额外的功能。这可能包括检查身份、将函数结果记录到固定位置等用例。

### 1.5.2 数据的清理或添加

装饰器也可以清理传递给被装饰函数的参数的值，从而确保参数类型的一致性或使该值符合指定的模式。例如，装饰器可以确保发送给函数的值符合指定类型，或者满足一些其他的验证标准(稍后将介绍相关示例，`@requires_ints` 装饰器)。

装饰器也可以改变或清除从函数中返回的数据。如果想让函数返回一个原生的 Python 对象(如列表或字典)，但是最终在另一端接收到序列化格式的数据(如 JSON 或 YAML)，这将是很有价值的用例。

有些装饰器实际上为函数提供了额外的数据，通常这些数据是附加参数的形式。`@mock.patch` 装饰器就是这种示例，因为它(除了原有参数之外)为函数提供了一个作为附加位置参数创建的 mock 对象。

### 1.5.3 函数注册

很多时候，在其他位置注册函数很有用——例如，在任务运行器中注册一个任务，或者注册一个带有信号处理器的函数。任何由外部输入或路由机制决定函数运行的系统都可以使用函数注册。

## 1.6 编写装饰器

装饰器仅仅是这样的函数：(通常)接受被装饰的可调用函数作为唯一参数，并且返回一个可调用函数(如前面几个普通示例所示)。

一个很重要的注意事项是：当装饰器应用到装饰函数时(而不是调用装饰器时)，会执行装饰代码本身。理解这一点至关重要，通过接下来的几个示例，会对其有清晰的理解。

### 1.6.1 初始示例：函数注册表

考虑下面这个简单的函数注册表：

```
registry = []
def register(decorated):
    registry.append(decorated)
    return decorated
```

`register` 方法是一个简单的装饰器。它附加一个位置参数，该参数被装饰到注册表变量，然后返回未改变的装饰方法。任何接收 `register` 装饰器的方法都将把自身附加到 `registry`。

```
@register
def foo():
    return 3

@register
def bar():
    return 5
```

如果访问注册表，可以很容易遍历注册表并且在内部执行函数。

```
answers = []
for func in registry:
    answers.append(func())
```

`answers` 列表中此时包含了 `[3, 5]`。这是因为函数是按顺序执行的，所返回的值被附加到 `answers`。

有几个在函数注册中有意义的用例。例如，将“钩子”添加到代码中，以便能够在关键事件前后执行自定义功能。这里有一个 `registry` 类恰好能处理这种情况：

```
class Registry(object):
    def __init__(self):
        self._functions = []

    def register(self, decorated):
        self._functions.append(decorated)
        return decorated

    def run_all(self, *args, **kwargs):
        return_values = []
        for func in self._functions:
            return_values.append(func(*args, **kwargs))
        return return_values
```

在该类中值得注意的是 `register` 方法——即装饰器——仍然像以前一样工作。让绑定方法作为装饰器很好。它接收 `self` 作为第一个参数(就像其他绑定方法)，并且需要一个被装饰方法作为其额外的位置参数。

通过几个不同的注册表实例，可以拥有完全分离的注册表。甚至可以在多个注册表中注册同一个函数，如下所示：

```
a = Registry()
b = Registry()

@a.register
def foo(x=3):
    return x

@b.register
def bar(x=5):
    return x

@a.register
@b.register
def baz(x=7):
    return x
```

运行任意注册表的 `run_all` 方法中的代码将得出如下结果：

```
a.run_all()    # [3, 7]
b.run_all()    # [5, 7]
```

注意，`run_all` 方法能够接受参数，运行时将参数传入底层函数。

```
a.run_all(x=4) # [4, 4]
```

## 1.6.2 运行时封装代码

这种装饰器非常简单，因为被装饰函数是在未经修改的条件下传递的。但是，执行被装饰方法时，可能希望运行额外的功能。为此，可以返回一个添加合适功能且(通常)在执行过程中调用被装饰方法的可调用函数。

### 1. 一个简单的类型检查

下面是一个简单装饰器，确保函数接收的所有参数都是整型，否则报错：

```
def requires_ints(decorated):
    def inner(*args, **kwargs):
        # Get any values that may have been sent as keyword arguments.
        kwarg_values = [i for i in kwargs.values()]

        # Iterate over every value sent to the decorated method, and
        # ensure that each one is an integer; raise TypeError if not.
        for arg in list(args) + kwarg_values:
            if not isinstance(arg, int):
                raise TypeError('%s only accepts integers as arguments.' %
                                decorated.__name__)
```

```
# Run the decorated method, and return the result.  
return decorated(*args, **kwargs)  
return inner
```

这里发生了什么？

装饰器自身是 `requires_ints`。它接收一个参数：`decorated`，即被装饰的可调用函数。装饰器唯一做的事情就是返回一个新的可调用函数，即本地函数 `inner`。该函数替代了被装饰方法。

通过声明一个函数并使用 `requires_ints` 装饰它，可以看到实际结果：

```
@requires_ints  
def foo(x, y):  
    """Return the sum of x and y."""  
    return x + y
```

注意运行 `help(foo)` 会得到的内容：

```
Help on function inner in module __main__:  
  
inner(*args, **kwargs)  
(END)
```

将名称 `foo` 赋给 `inner` 函数，而不是赋给原来被定义的函数。如果运行 `foo(3.5)`，将利用传入的这两个参数运行 `inner` 函数。`inner` 函数执行类型检查，然后运行被装饰的方法，仅仅是由于 `inner` 函数使用返回的被封装方法(`*args, **kwargs`)调用它，返回值是 8。如果缺少这个调用，被装饰方法将被忽略。

## 2. 保存帮助信息

用一个装饰器去细究函数的文本字符串或者截取 `help` 的输出并不可行。因为装饰器是用于添加通用和可重用功能的工具，所以相对模糊的注释是有必要的。而且一般来讲，如果使用函数的人尝试对函数执行 `help`，那么他只希望了解函数的核心信息，而不是关于 `shell` 的信息。

该问题的解决方案实际上是装饰器。Python 实现一个名为 `@functools.wraps` 的装饰器，将一个函数中的重要内部元素复制到另一个函数。

下面是同一个 `@requires_ints` 装饰器，但是它使用 `@functools.wraps` 来添加：

```
import functools  
  
def requires_ints(decorated):  
    @functools.wraps(decorated)  
    def inner(*args, **kwargs):  
        # Get any values that may have been sent as keyword arguments.  
        kwarg_values = [i for i in kwargs.values()]  
  
        # Iterate over every value sent to the decorated method, and
```

```
# ensure that each one is an integer; raise TypeError if not.
for arg in args + kwarg_values:
    if not isinstance(i, int):
        raise TypeError('%s only accepts integers as arguments.' %
                        decorated.__name__)

# Run the decorated method, and return the result.
return decorated(*args, **kwargs)
return inner
```

装饰器本身几乎没有改变，除了在第二行增加了将@functools.wraps 装饰器应用到 inner 函数。现在还必须导入 functools(在标准类库中)。还需要注意一些额外的语法。这个装饰器实际上使用了参数(稍后详细介绍)。

现在将装饰器应用到同一个函数，如下所示：

```
@requires_ints
def foo(x, y):
    """Return the sum of x and y."""
    return x + y
```

接下来就可以看到现在运行 help(foo)会发生什么：

```
Help on function foo in module __main__:

foo(x, y)
    Return the sum of x and y.
(END)
```

可以看到 foo 函数的文本字符串及其方法签名，这就是查看 help 时读到的内容。但是从原理上讲，仍然应用了@requires\_ints 装饰器，并且 inner 函数仍然在运行。

根据所运行的 Python 版本的不同，针对 foo 运行 help 得到的结果略微不同，尤其是在有关函数签名方面。之前的部分代表 Python 3.4 的输出结果。但是在 Python 2 中，提供的函数签名仍将是 inner(因此，参数为\*args 与\*\*kwargsrather，而不是 x 与 y)。

### 3. 用户验证

该模式的一个常见用例(即在运行装饰方法之前执行某种正确性检查)是用户验证。考虑一个期望将用户作为其第一个参数的方法。

用户应该是这个 User 与 AnonymousUser 类的实例，如下所示：

```
class User(object):
    """A representation of a user in our application."""

    def __init__(self, username, email):
        self.username = username
        self.email = email

class AnonymousUser(User):
```

```
"""An anonymous user; a stand-in for an actual user that nonetheless
is not an actual user.
"""
def __init__(self):
    self.username = None
    self.email = None

def __nonzero__(self):
    return False
```

这里的装饰器是一个强大的工具，用于隔离用户验证的样板代码。`@requires_user` 装饰器能够非常容易地验证你得到 `User` 对象，因此就不是一个匿名用户。

```
import functools

def requires_user(func):
    @functools.wraps(func)
    def inner(user, *args, **kwargs):
        """Verify that the user is truthy; if so, run the decorated method,
        and if not, raise ValueError.
        """
        # Ensure that user is truthy, and of the correct type.
        # The "truthy" check will fail on anonymous users, since the
        # AnonymousUser subclass has a `__nonzero__` method that
        # returns False.
        if user and isinstance(user, User):
            return func(user, *args, **kwargs)
        else:
            raise ValueError('A valid user is required to run this.')
    return inner
```

装饰器应用于常用的模板需求——验证一个用户已经登录到应用程序。当以装饰器的方式实现该功能时，程序更容易复用且更加容易维护，函数的装饰器应用程序看上去也更加简洁与明显。

注意，该装饰器仅仅能够正确包装函数或静态方法，如果包装绑定到类的方法，将会失败。这是由于装饰器忽略了将 `self` 发送给绑定方法作为第一个参数的期望。

#### 4. 输出格式化

除了对输入到函数的参数进行检查，装饰器的另一用途是检查函数的输出。

当使用 Python 时，尽可能使用原生 Python 对象通常更加可行。但是你经常会希望得到序列化的输出格式(如 JSON 格式)。在所有相关函数的结尾手动将结果转换为 JSON 格式非常繁琐，也不是好主意。理想情况下，只有在必要时才应该使用 Python 结构，并且可能在序列化之前还需要应用其他样板代码(序列化或类似代码)。

装饰器为这一问题提供了完美且方便的解决方案。考虑下面这个接受 Python 输出结果并将其序列化为 JSON 格式的装饰器：

```
import functools
import json

def json_output(decorated):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    @functools.wraps(decorated)
    def inner(*args, **kwargs):
        result = decorated(*args, **kwargs)
        return json.dumps(result)
    return inner
```

将@json\_output 装饰器应用到一个简单的函数，如下所示：

```
@json_output
def do_nothing():
    return {'status': 'done'}
```

在 Python shell 中执行函数，会得到如下结果：

```
>>> do_nothing()
'{"status": "done"}'
```

注意，你将得到包含有效 JSON 的字符串而不是字典。

装饰器的优美之处在于它的简洁。将其应用到函数后，一个返回 Python 字典、列表或其他对象的函数立刻就会变为返回序列化 JSON 格式的版本。

你可能要问，“为什么它如此有价值？”毕竟，为添加装饰器增加的一行代码仅仅移除了一行代码——调用 json.dumps 的代码。但是请考虑在应用程序的需求扩展时，装饰器所起的作用。

例如，如果希望捕获特定异常并以指定格式的 JSON 输出，而不是让异常冒泡并输出回溯该怎么办？正因为有了装饰器，所以可以非常容易地添加该功能：

```
import functools
import json

class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message

def json_output(decorated):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
```

```
"""
@functools.wraps(decorated)
def inner(*args, **kwargs):
    try:
        result = decorated(*args, **kwargs)
    except JSONOutputError as ex:
        result = {
            'status': 'error',
            'message': str(ex),
        }
    return json.dumps(result)
return inner
```

通过将错误处理作为参数传递给@json\_output 装饰器，对于应用装饰器的任意函数都可以为其添加该功能。这就是装饰器的用武之地。对于代码的可移植性和重用性而言，它们是非常有用的工具。

现在，如果使用@json\_output 装饰的函数抛出 JSONOutputError，那么可以对该具体错误进行处理。函数示例如下：

```
@json_output
def error():
    raise JSONOutputError('This function is erratic.')
```

在 Python 解释器中运行 error 函数的结果如下：

```
>>> error()
'{"status": "error", "message": "This function is erratic."}'
```

注意，只有 JSONOutputError 异常类(及任意子类)可以接收此特殊处理。任何其他异常将被正常传递，并且生成一个回溯。考虑如下函数：

```
@json_output
def other_error():
    raise ValueError('The grass is always greener...')
```

当运行该函数时，将会得到期望的回溯，如下所示：

```
>>> other_error()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in inner
  File "<stdin>", line 3, in other_error
ValueError: The grass is always greener...
```

重用性和可维护性是装饰器价值的一部分。因为在整个应用程序中，装饰器被当成一个可重用的、普遍适用的概念(在本例中，是 JSON 序列化)。当存在适用该理念的新需求时，装饰器成为容纳该功能的容器。

从本质上讲，装饰器是避免重复的工具，它们的部分价值就是为代码的未来维护提供

钩子。

也可以不使用装饰器完成该功能。考虑下面这个需要登录用户的示例。编写一个实现该功能的函数并不难，只需要将该函数置于需要该功能的函数开头部分即可。该装饰器主要是一个语法糖，但语法糖也有价值。毕竟代码被读的次数远远多于被写的次数，而且很容易一眼就找到装饰器。

## 5. 日志管理

运行时封装代码的最后一个示例是通用的日志管理函数。考虑下面引发函数调用、计时然后将结果记录到日志的装饰器：

```
import functools
import logging
import time

def logged(method):
    """Cause the decorated method to be run and its results logged, along
    with some other diagnostic information.
    """
    @functools.wraps(method)
    def inner(*args, **kwargs):
        # Record our start time.
        start = time.time()

        # Run the decorated method.
        return_value = method(*args, **kwargs)

        # Record our completion time, and calculate the delta.
        end = time.time()
        delta = end - start

        # Log the method call and the result.
        logger = logging.getLogger('decorator.logged')
        logger.warn('Called method %s at %.02f; execution time %.02f '
                    'seconds; result %r.' %
                    (method.__name__, start, delta, return_value))

        # Return the method's original return value.
        return return_value
    return inner
```

当把装饰器应用到函数时，它会正常执行函数，但使用 Python 的 logging 模块在函数调用结束后会将相关信息记录到日志。现在，应用了装饰器的任意函数立刻就拥有了(最基本的)日志管理功能。

```
>>> import time
>>> @logged
... def sleep_and_return(return_value):
```

```
...     time.sleep(2)
...     return return_value
...
>>>
>>> sleep_and_return(42)
Called method sleep_and_return at 1424462194.70;
    execution time 2.00 seconds; result 42.
42
```

与前面的示例不同，该装饰器没有用显而易见的方式改变函数调用。不会出现将装饰器应用到一个函数后该函数返回不同结果这样的情况。在前面的示例中，如果检测不通过，则会引发异常或是修改返回结果；而本例中的装饰器并不明显，它默默完成后台工作，但在任何情况下都不会修改实际的返回结果。

## 6. 变量参数

值得注意的是，`@json_output` 和 `@logged decorators` 两个装饰器都提供了仅仅接受并以最小化检查、变量参数和关键字参数传递的 `inner` 函数。

这是一种重要的模式。它尤其重要的一种应用方式是很多装饰器被用于装饰普通函数以及类的方法。记住在 Python 中，在类中声明的方法接受一个额外的位置参数，按照惯例为 `self`。应用装饰器并不会改变这一点(这就是为什么前面示例中的 `quires_user` 装饰器无法应用于类中的绑定方法)。

例如，如果 `@json_result` 用来装饰一个类的方法，调用 `inner` 函数并且它会接收类的实例作为第一个参数。实际上，这没有问题。在本例中，该参数只是 `args[0]`，然后它被顺利地传递给被装饰的方法。

### 1.6.3 装饰器参数

至此，一直没变的是所枚举的所有装饰器本身看上去并没有任何参数。正如所讨论的，有一个隐式参数——即被装饰的方法。

但是，有时让装饰器自身带有一些需要的信息，从而使装饰器可以用恰当的方式装饰方法十分有用。一个传递给装饰器的参数与一个在调用时传递给函数的参数的区别正在于此。传递给装饰器的参数只被处理一次，即在函数声明并被装饰时处理。与之相反，传递给函数的参数在该函数被调用时处理。

在前面的示例中，已经介绍了一个参数被发送给重复使用 `@functools.wraps` 的装饰器。它接收一个参数——被封装的方法，该方法中的帮助和文档字符串以及类似的内容都将保存。

但是，装饰器有隐式的调用签名。它接收一个位置参数——被装饰的方法。那么，这又如何工作呢？

答案是这很复杂。回忆在运行时封装代码最基本的装饰器。这些装饰器在局部作用域声明一个内部方法后返回。这是由装饰器返回的可调用函数。该函数被赋值给函数名称。

接受参数的装饰器额外增加了一层封装。这是由于接受参数的装饰器并不是实际的装饰器，而是一个返回装饰器的函数，该函数接受一个参数(被装饰的方法)，然后装饰函数并返回一个可调用函数。

这听起来让人困惑。考虑下面的示例，其中对`@json_output`装饰器进行参数化，以允许输出结果缩进及按照键排序：

```
import functools
import json

class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message

def json_output(indent=None, sort_keys=False):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    def actual_decorator(decorated):
        @functools.wraps(decorated)
        def inner(*args, **kwargs):
            try:
                result = decorated(*args, **kwargs)
            except JSONOutputError as ex:
                result = {
                    'status': 'error',
                    'message': str(ex),
                }
            return json.dumps(result, indent=indent, sort_keys=sort_keys)
        return inner
    return actual_decorator
```

那么，这里发生了什么？为什么排序和缩进能够生效？

这是一个 `json_output` 函数，它接受两个参数(`indent` 和 `sort_keys`)，它返回另一个名为 `actual_decorator` 的函数，该函数(顾名思义)意在用作装饰器。这是一个经典的装饰器——一个接受单独可调用函数(`decorated`)作为参数并返回一个可调用函数(`inner`)的可调用函数。

注意，对 `inner` 函数做了微小的修改，从而可以接受 `indent` 与 `sort_keys` 参数。这些参数仿照 `json.dumps` 所接受的参数，因此调用 `json.dumps` 接受装饰器签名中提供给 `indent` 与 `sort_keys` 的值，并在倒数第三行代码中将它们提供给 `json.dump`。

`inner` 函数是最终使用 `indent` 和 `sort_keys` 参数的函数。这没有问题，因为 Python 的代码块的作用域规则允许这么做。即使在调用过程中为 `inner` 与 `sort_keys` 参数的赋值不同也

没有问题，因为 `inner` 是一个局部函数(每次使用装饰器都返回不同的副本)。

`json_output` 装饰器的应用如下所示：

```
@json_output(indent=4)
def do_nothing():
    return {'status': 'done'}
```

现在如果运行 `do_nothing` 函数，就会得到一个带有缩进和换行的 JSON 块，如下所示：

```
>>> do_nothing()
'\n  "status": "done"\n'
```

## 1. 为什么函数能被当成装饰器使用

如果 `json_output` 不是装饰器，而是返回装饰器的函数，那么应用它的方式看起来为什么就像是在应用装饰器？在此，Python 解释器是如何使它生效的呢？

在此要详细解释的是操作顺序。这里的关键是操作顺序。更具体地说，函数在装饰器应用语法(`@`)之前调用(`json_output(indent=4)`)。因此，函数调用的结果被应用到装饰器上。

首先，就是解释器发现对 `json_output` 函数的调用，然后解析该调用(注意，黑体字不包含`@`)：

```
@json_output(indent=4)
def do_nothing():
    return {'status': 'done'}
```

`json_output` 函数所做的就是定义另一个函数 `actual_decorator`，然后返回它。该函数的返回值被提供给`@`，如下所示：

```
@actual_decorator
def do_nothing():
    return {'status': 'done'}
```

现在，运行 `actual_decorator`。它声明了另一个局部函数 `inner`，然后返回它。如前所述，该函数被赋给名称 `do_nothing`，也就是被装饰方法的名称。调用 `do_nothing` 时，`inner` 函数被调用，执行被装饰的方法，然后输出带有合适缩进的 JSON 结果。

## 2. 调用签名很重要

当引入修改过的新函数 `json_output` 时，意识到实际上你引入的是一个不可向后兼容的变更，这非常重要。

为什么？因为现在期望一个额外的函数调用。即使想要这个旧的 `json_output` 函数行为，并且不需要任何可用参数的值，也仍然必须调用这个方法。

换言之，必须做如下操作：

```
@json_output()
def do_nothing():
    return {'status': 'done'}
```

注意圆括号，它们很重要，因为它们表明该函数已被调用(即使没有参数)，然后该结果被应用到@。

下面的代码与前面的代码不重复，也不等价：

```
@json_output
def do_nothing():
    return {'status': 'done'}
```

这里有两个问题。第一，代码在本质上是混乱的，因为如果习惯看到没有签名的装饰器，那么要求提供一个空的签名是违反直觉的。第二，如果旧的装饰器已经存在于应用程序中，则必须返回去编辑所有存在的调用。如果可能，应当尽量避免向后不兼容的修改。

在理想情况下，这个装饰器可以对 3 种不同类型的应用程序生效：

- @json\_output
- @json\_output()
- @json\_output(indent=4)

结果证明，通过基于装饰器所接受的参数修改装饰器的行为是可能的。记住，装饰器只是一个函数，具有其他函数的所有灵活性，它可以为了响应输入而完成所需要完成的工作。

考虑下面这个 json\_output 的更灵活的迭代：

```
import functools
import json

class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message

def json_output(decorated_=None, indent=None, sort_keys=False):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    # Did we get both a decorated method and keyword arguments?
    # That should not happen.
    if decorated_ and (indent or sort_keys):
        raise RuntimeError('Unexpected arguments.')

    # Define the actual decorator function.
    def actual_decorator(func):
        @functools.wraps(func)
        def inner(*args, **kwargs):
            try:
```

```
        result = func(*args, **kwargs)
    except JSONOutputError as ex:
        result = {
            'status': 'error',
            'message': str(ex),
        }
    return json.dumps(result, indent=indent, sort_keys=sort_keys)
return inner

# Return either the actual decorator, or the result of applying
# the actual decorator, depending on what arguments we got.
if decorated_:
    return actual_decorator(decorated_)
else:
    return actual_decorator
```

该函数尝试更加智能地判断当前它是否被作为装饰器使用。

首先，它确保不被以意料之外的方式调用。永远不要期望既可以接受被装饰的方法又可以接受关键字参数，因为装饰器在被调用过程中只能接受被调用方法作为唯一参数。

其次，它定义了 `actual_decorator` 方法，也就是(顾名思义)实际被返回或应用的装饰器。它定义的 `inner` 函数是由装饰器最终返回的函数。

最后，该装饰器会基于被调用的方式返回合适的结果：

- 如果设置了 `decorated_`，它将作为一个没有方法签名的纯装饰器被调用，它的职责是应用最终的装饰器并返回 `inner` 函数。再次注意，观察接受参数的装饰器实际上是如何生效的。首先，调用并解析 `actual_decorator(decorated_)` 函数，然后以 `inner` 作为唯一参数调用该函数的返回结果(结果必须是可调用函数，因为这是一个装饰器)。
- 如果没有设置 `decorated_`，那么这就是带有参数关键字的调用，并且函数必须返回一个实际的装饰器，该装饰器接受被装饰的方法并返回 `inner` 函数。该函数直接返回 `actual_decorator` 装饰器。然后 Python 解释器将它作为实际的装饰器应用(最终返回 `inner` 函数)。

为什么该技术很有价值？它允许维护已经应用的装饰器功能。这意味着不必更新每一处装饰器被应用的位置，但仍然可以获得在需要时添加参数的灵活性。

## 1.7 装饰类

记住，本质上来说装饰器是一个接受可调用函数的可调用函数，并返回一个可调用函数。这意味着装饰器可以被用于装饰类和函数(毕竟类本身也是可调用函数)。

装饰类有多种用途。它们十分有价值，因为正如函数装饰器那样，类装饰器可以与被装饰类的属性交互。类装饰器可以添加属性或将属性参数化，或是它可以修改一个类的 API，从而使类被声明的方式与实例被使用的方式不同。

你或许要问，“通过子类增加或修改一个类的属性是否合适”？通常答案是肯定的。但

是，在某些情况下另一种途径或许更加合适。例如，考虑一个会在应用程序中应用到多个类的通用功能，但该功能位于类层级的不同位置。

例如，考虑一个类的一个功能：每个实例都知道自身被实例化的时间，并按照创建时间排序。该功能对于很多不同的类都是通用的，实现方式是需要 3 个额外的属性——实例化的时间戳、`__gt__`方法和`__lt__`方法。

可以通过多种方式添加属性。下面是使用类装饰器的实现方式：

```
import functools
import time

def sortable_by_creation_time(cls):
    """Given a class, augment the class to have its instances be sortable
    by the timestamp at which they were instantiated.
    """
    # Augment the class' original `__init__` method to also store a
    # `_created` attribute on the instance, which corresponds to when it
    # was instantiated.
    original_init = cls.__init__

    @functools.wraps(original_init)
    def new_init(self, *args, **kwargs):
        original_init(self, *args, **kwargs)
        self._created = time.time()
    cls.__init__ = new_init

    # Add `__lt__` and `__gt__` methods that return True or False based on
    # the created values in question.
    cls.__lt__ = lambda self, other: self._created < other._created
    cls.__gt__ = lambda self, other: self._created > other._created

    # Done; return the class object.
    return cls
```

在该装饰器中，首先保存了类的原始方法`__init__`的副本。你无须担心该类是否已有`__init__`方法。由于`object`对象包含`__init__`方法，因此该属性一定会存在。接下来，创建一个将会被赋值给`__init__`的新方法，该方法首先调用原始方法，之后完成一些额外工作，也就是将实例化时间戳赋值给`self._created`属性。

值得注意的是，该模式与前面在执行时封装代码的示例非常相似——创建一个封装另一个函数的函数，该函数的主要职责是执行被封装的函数，但同时也增加了一小部分其他功能。

值得注意的是，如果应用`@sortable_by_creation_time`装饰器的类定义了自己的`__lt__`与`__gt__`方法，那么该装饰器将会重写这两个方法。

如果类没有识别出`__created`属性被用于排序，那么该属性值自身并没有什么用处。因此，装饰器还加入了`__lt__`与`__gt__`魔术方法。这使`and`操作符基于这些方法的结果返回`True`

或 `False`。这同时也影响了 `sorted` 函数和其他类似函数的行为。

这是必须的，以使任意类的实例可以根据实例化时间排序。该装饰器可以应用到任何类，包括那些与其父类不相关的类。

下面是一个简单类的示例，该类可以根据实例的创建时间排序：

```
>>> @sortable_by_creation_time
... class Sortable(object):
...     def __init__(self, identifier):
...         self.identifier = identifier
...     def __repr__(self):
...         return self.identifier
...
>>> first = Sortable('first')
>>> second = Sortable('second')
>>> third = Sortable('third')
>>>
>>> sortables = [second, first, third]
>>> sorted(sortables)
[first, second, third]
```

记住，装饰器仅可以用于解决问题，这并不意味着装饰器就是解决问题的最恰当方法。

例如，对于本例来说，可以使用 `mixin` 或一个仅仅定义 `_init_`、`_lt_` 与 `_gt_` 方法的小类来实现同样的功能。使用 `mixin` 的简单方式如下所示：

```
import time

class SortableByCreationTime(object):
    def __init__(self):
        self._created = time.time()

    def __lt__(self, other):
        return self._created < other._created

    def __gt__(self, other):
        return self._created > other._created
```

可以使用 Python 的多重继承将 `mixin` 应用到类：

```
class MyClass(MySuperclass, SortableByCreationTime):
    pass
```

该方法与使用装饰器的方法相比有不同的优缺点。一方面，它不会直接重写由类或父类定义的 `_lt_` 与 `_gt_` 方法(当随后阅读代码时，很难发现装饰器已经重载了这两个方法)。

另一方面，很容易陷入这样的情况——由 `SortableByCreateTime` 提供的 `_init_` 方法没有执行。如果 `MyClass` 或 `MySuperclass` 或是任何 `MySuperclass` 的父类定义了 `_init_` 方法，那么将会执行父类的 `_init_` 方法。将类继承顺序反过来并不能解决该问题；改变继承顺序并不

会影响执行父类 `_init_` 方法这一结果。

与之相反，装饰器能够正确处理 `_init_` 方法，仅仅通过重写被装饰类的 `_init_` 方法，从而添加额外操作或是完全不修改 `_init_` 方法。

那么哪一种方法是正确的方法？这要视情况而定。

## 1.8 类型转换

至此，本章的讨论仅考虑了装饰器期望装饰函数并返回函数，或装饰器期望装饰类并返回类的情况。

但是，并没有必须保持这种关系的理由。装饰器的唯一需求是一个可调用函数接受一个可调用函数并返回一个可调用函数。没有要求必须返回同种类型的可调用函数。

一个更高级的装饰器用例实际上不这样做。尤其是，装饰器装饰一个函数，但返回一个类，这很有价值。当增加大量样板代码时，可以允许开发人员对于简单情况使用简单函数，而对于复杂情况则允许继承应用程序 API 中的类，在这些情况下，装饰器是非常有用的工具。

对此，在 Python 生态系统的流行任务执行器中使用的装饰器是 `celery`。`celery` 包提供的 `@celery.task` 装饰器期望装饰一个函数，而该装饰器实际上会返回 `celery` 内部的 `Task` 类，而被装饰的函数在子类的 `run` 方法中被使用。

考虑下面一个使用类似方法的简单示例：

```
class Task(object):
    """A trivial task class. Task classes have a `run` method, which runs
    the task.
    """
    def run(self, *args, **kwargs):
        raise NotImplementedError('Subclasses must implement `run`.')

    def identify(self):
        return 'I am a task.'

def task(decorated):
    """Return a class that runs the given function if its run method is
    called.
    """
    class TaskSubclass(Task):
        def run(self, *args, **kwargs):
            return decorated(*args, **kwargs)
    return TaskSubclass
```

这里发生了什么？装饰器创建了 `Task` 的一个子类并返回该类。该类是一个可调用函数并调用一个类创建该类的实例，返回该类的 `__init__` 方法。

这么做的价值在于为大量的扩展提供一个钩子。基本的 `Task` 类可以比 `run` 方法定义更

多的内容。例如，`start` 方法或许可以异步执行任务。基本类或许也可以提供用于保存任务状态的方法。使用一个装饰器将一个函数替换为一个类，可以使开发人员只需要考虑所编写任务的实际内容，而装饰器会完成余下的工作。

可以通过接受一个该类的实例并执行它的 `identify` 方法来查看实际效果：

```
>>> @task
>>> def foo():
>>>     return 2 + 2
>>>
>>> f = foo()
>>> f.run()
4
>>> f.identify()
'I am a task.'
```

## 陷阱

这个特定方法会带来一些问题。尤其是，一旦任务函数被 `@task_class` 装饰器装饰时，它就会变为一个类。

考虑下面以这种方式被装饰的简单任务函数：

```
@task
def foo():
    return 2 + 2
```

现在，尝试在解释器中直接执行该函数：

```
>>> foo()
<__main__.TaskSubclass object at 0x10c3612d0>
```

这是一件糟糕的事情。该装饰器以这样一种方式修改函数：如果开发人员执行该函数，它并不会完成任何人期望的工作。通常很难接受函数被声明为 `foo`，并以复杂的 `foo().run()` 形式执行(而在本例中这样执行是必需的)。

解决该问题需要更多考虑装饰器与 `Task` 类的构建方式。考虑下面修订的版本：

```
class Task(object):
    """A trivial task class. Task classes have a `run` method, which runs
    the task.
    """
    def __call__(self, *args, **kwargs):
        return self.run(*args, **kwargs)

    def run(self, *args, **kwargs):
        raise NotImplementedError('Subclasses must implement `run`.')

    def identify(self):
        return 'I am a task.'
```

```
def task(decorated):  
    """Return a class that runs the given function if its run method is  
    called.  
    """  
    class TaskSubclass(Task):  
        def run(self, *args, **kwargs):  
            return decorated(*args, **kwargs)  
    return TaskSubclass()
```

在此有一些关键区别。第一个区别是用于调用基类 `Task` 而新增的 `_call_` 方法。第二个区别(用于补充第一点)是 `@task_class` 装饰器现在返回 `TaskSubclass` 类的实例而不是类本身。

能够接受该方案是由于对于装饰器来说唯一的要求是返回一个可调用函数，而 `Task` 新增的 `_call_` 方法意味着它的实例现在可以被调用。

为什么该模式很有价值？`Task` 类虽然非常简单，但是很容易看到如何将更多的功能添加进来，这对于管理和执行任务非常有用。

但是，该方法会在原始函数被直接调用时维持原始函数的功能，再次考虑被装饰的函数：

```
@task  
def foo():  
    return 2 + 2
```

现在，如果在解释器中执行该函数会返回什么？

```
>>> foo()  
4
```

这正是你所期望的，这使该类成为更好的类与装饰器设计。在底层装饰器返回一个 `TaskSubclass` 实例。当该实例在解释器中被调用时，它的 `_call_` 方法被调用，该方法会调用 `run` 函数，从而调用原始函数。

你会发现，使用 `identify` 方法仍然会获得返回的实例：

```
>>> foo.identify()  
'I am a task.'
```

现在有这样一个实例，直接调用它时，调用方法与原始函数并无不同。但是，该实例可以包含用于提供其他功能的其他方法和属性。

这个功能很强大，可以使开发人员编写一个可以容易且明显转换为类的函数，从而提供额外的函数调用方式或添加其他相关功能。这是一个有用的范例。

## 1.9 小结

对于编写可维护的、可读的 `Python` 代码而言，装饰器是极有价值的工具。装饰器的价值在于它是显式的且可重用的。装饰器提供了一种完美的方式来使用样板代码，一次编写

就可以将其用于多种不同情况。

这个有用的范例之所以能够生效，是因为 Python 的数据模型将函数与类作为一级对象提供，它们可以作为参数传递并可以像语言中的其他对象那样扩展。

另一方面，该模型也有缺点。尤其是装饰器的语法，虽然整洁且易于阅读，但使一个函数被封装到另一个函数这个事实变得模糊，从而给调试带来挑战。糟糕的装饰器或许会由于它们忽略了被封装可调用函数的本质而导致错误(例如，通过忽略绑定方法与被绑定函数之间的区别)。

此外，记住，与任何函数一样，解释器必须执行装饰器中的代码，这会对性能有影响。装饰器也不例外；留心使用装饰器的目的，这与留心编写其他代码的目的并无不同。

考虑使用装饰器作为一种封装不相关函数开头与结尾功能的方法。与之相似，装饰器是用于函数注册、发信号、某种情况下的类增强以及其他功能的强大工具。

第 2 章中讨论了上下文管理器，这是将需要在整个程序中一小段需要复用的功能以高效便捷的方式划分出来的方法。