目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python ？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

21 给凡人添加超能力：入手装饰器

更新时间：2019-10-11 10:30:51

> 富贵必从勤苦得。
>
> ——杜甫

在学习装饰器前，我们先来了解两个函数概念。

函数中定义函数

在 Python 中，函数内部是可以嵌套地定义函数的。如：

```
def print_twice(word):
    def repeat(times):
        return word * times

    print(repeat(2))
```

```
>>> print_twice('go ')
gogo
```

内层函数只能在包裹它的外层函数中使用，而不能在外层函数外使用。比如上面的 `repeat()` 可以在 `print_twice()` 中使用，但是不能在 `print_twice()` 的外部使用。

另外，内层函数中可以使用外层函数的参数或其它变量。如上面的参数 `word` 。

函数返回函数

之前我们学习过，函数可以作为另一个函数的参数。类似的，函数的返回值也可以是一个函数。

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python ？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

```
def print_words(word):
    def repeat(times):
        return word * times

    return repeat
```

```
>>> f = print_words(‘go’)
>>> f
<function print_words..repeat at 0x10befe620>
```

我们调用 print_words() 并用变量 f 接收其返回值，f 是个函数，是 print_words 下的 repeat 函数。

既然 f 是个函数，自然可以被调用，这也就相当于调用 repeat()：

```
>>> f(2)
‘gogo’
```

扩展：我们直接调用 f（也就是 repeat()）时，repeat() 内部会使用变量 word，而这个变量时定义在外层函数 print_words() 中的，却会一直伴随 repeat() 而存在，这在 Python 中叫作闭包。

装饰器是什么

好了，回到正题，来看看什么是装饰器。我们在《类进阶》章节中介绍过类方法和静态方法的定义方式，还记得吗，定义它们时需要用到 @classmethod 和 @staticmethod，它们就是装饰器。写法为 @装饰器名称 。

装饰器用来增强一个现有函数的功能，并且不改变这个函数的调用方式。这种增强是非侵入式的，也就是说无需直接修改函数内部的代码，而是在函数的外部做文章。

举个例子，假设我们有这样一个函数：

```
def say_hello():
    print('Hello!')
```

```
>>> say_hello()
Hello!
```

这个函数非常简单，每次调用会输出「Hello!」，假如我们想在每次输出「Hello!」的同时附带上当前的时间，像这样：

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python ？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

```
[ 2019-09-14 16:38:10.942802 ]
Hello!
>>> say_hello()
[ 2019-09-14 16:42:58.409742 ]
Hello!
```

如果想具备上面的功能，但又不想修改 say_hello() 函数的内部实现，该怎么做？

这就是装饰器的典型使用场景了——非侵入的情况下让函数具备更多的功能。

假设我们已经有了一个能满足该需求的装饰器 @time ，只要像这样来装饰 say_hello() 即可：

```
@time
def say_hello():
    print('Hello!')
```

函数的调用方式依然不变：

```
>>> say_hello()
```

当然，虽然 Python 中内置有一些装饰器，如 @classmethod 、@staticmethod ，但并没 @time ，所以我们需要自己来定义它。

自定义装饰器

我们来自定义之前所说的装饰器 @time ，要求是使用它可以在函数调用时输出调用时间。

这里直接给出 @time 的实现：

```
import datetime    # 日期时间相关库，用于后续获取当前时间

def time(func):
    def wrapper(*args, **kw):
        print('[', datetime.datetime.now(), ']')
        return func(*args, **kw)
    return wrapper
```

我们暂且不关注具体的实现细节，先使用一下看看：

```
@time
def say_hello():
    print('Hello!')
```

```
>>> say_hello()
[ 2019-09-14 16:42:58.409742 ]
Hello!
>>> say_hello()
```

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

没有问题，效果和预期相同！那这是什么原理呢？

装饰器原理

其实，

```
@time
def say_hello():
    print('Hello!')
```

等效于：

```
def say_hello():
    print('Hello!')

say_hello = time(say_hello)
```

也就是说，我们用 `@time` 装饰 `say_hello()` 时，Python 会在背后做了这样一个操作（**重点**）：

```
say_hello = time(say_hello)
```

`@time` **（包括所有装饰器）本质上是个以函数作为参数，并返回函数的函数**。不妨回过头来观察下 `@time` 实现：

```
import datetime    # 日期时间相关库，用于后续获取当前时间

def time(func):
    def wrapper(*args, **kw):
        print('[', datetime.datetime.now(), ']')
        return func(*args, **kw)
    return wrapper
```

`say_hello = time(say_hello)` 这句代码将函数 `say_hello` 作为参数来调用 `time()`，`time()` 将其内部定义的函数返回了出来，并替换了函数 `say_hello`。结合装饰器实现来看，`say_hello()` 其实变成了 `time()` 中的 `wrapper()`。

```
>>> say_hello
<function time..wrapper at 0x10befea60>
```

那就来具体看下 `wrapper()`：

```
def wrapper(*args, **kw):
    print('[', datetime.datetime.now(), ']')
    return func(*args, **kw)
```

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python ？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

用于获取当前的时间。

最后一句 return func(*args, **kw) 比较关键，这里调用函数 func() 并将其结果返回出去。func() 是什么？它就是 say_hello() 。最初 say_hello 作为参数被传入 time() 中，其参数名便是 func 。

参数 *args 和 **kw 是什么？还记得我们在《函数进阶》中的内容吗， *args 可以接收一切非关键字参数，而 **kw 可以接收一切关键字参数，两个结合起来一起使用就可以接收一切参数了。用在这里的作用是，接收调用 say_hello() 时的所有参数，并悉数传给 func() 。

稍作梳理我们就能明白，装饰器之所以能够增强一个函数的功能，其实就是将被装饰函数用新函数替换，虽然还是同一个函数名，但函数内部实现已经变了。而这个新函数的内部在添加了一些功能的后，还会调用之前被装饰的函数。这样就相当于对被装饰的函数做了非侵入的扩展。

functools.wraps 装饰器

当一个函数不被装饰器装饰时，其函数名称就是自己。如：

```
>>> def say_hello():
…     print(‘Hello!’)
…
>>> say_hello
<function say_hello at 0x10efbb1e0>

>>> say_hello.__name__
‘say_hello’
```

在解释器中直接输入 say_hello ，显示其为 function say_hello 。使用 say_hello.__name__ ，可以直接获取到其函数名称，此处显示为 say_hello 。

如果我们用装饰器 @time 来修饰这个函数，那结果就不同了：

```
>>> @time
… def say_hello():
…     print(‘Hello!’)
…
>>> say_hello
<function time..wrapper at 0x10efbb048>

>>> say_hello.__name__
‘wrapper’
```

可以看到其名字信息被装饰器中的函数 wrapper 覆盖了。

是的，由于装饰器本质上是用一个新的函数来替换被装饰的函数，所以函数的元信息会被覆盖。

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python ？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

```
import datetime
import functools

def time(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('[', datetime.datetime.now(), ']')
        return func(*args, **kw)
    return wrapper
```

```
>>> say_hello
<function say_hello at 0x10ef5c378>

>>> say_hello.__name__
'say_hello'
```

可以看到使用 `@functools.wraps` 后，元信息恢复如初，不留痕迹。

带参数的装饰器

既然装饰器本质上是个函数，那这个函数能不能有参数呢？答案是可以有。

举个例子，刚才我们输出的时间格式是 `[ 2019-09-14 16:42:58.409742 ]`，如果我们想要自行指定这个格式，可以考虑用装饰器参数的形式来设置。

带时间格式的装饰器如下：

```
import datetime
import functools

def time(format):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print(datetime.datetime.now().strftime(format))
            return func(*args, **kw)
        return wrapper
    return decorator
```

可以看到，这回装饰器变成了三层函数嵌套的形式。是的，如果需要指定装饰器的参数，那么就需要在原来装饰器的基础上在再加一层函数。

`wrapper()` 中原本的 `print('[', datetime.datetime.now(), ']')` 被修改为 `print(datetime.datetime.now().strftime(format))`，其中的 `format` 便是装饰器的参数，也就是时间格式。

使用时，在装饰器 `@time` 后添加括号并写上参数：

```
@time('%Y/%m/%d %H:%M:%S')
def say_hello():
    print('Hello!')
```

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

> 2019/09/15 10:00:24
> Hello!

可以看到时间格式已经根据我们的设置而生效。

> 扩展：
>
> `'%Y/%m/%d %H:%M'` 是 `datetime` 包中用于指定时间格式的字符串，其中：
>
> - `%Y` 表示年
> - `%m` 表示月
> - `%d` 表示天
> - `%H` 表示小时
> - `%M` 表示分钟
> - `%S` 表示秒。

带参数的装饰器原理

带参数的装饰器的实现为什么要三层函数嵌套？看了下面的等效代码你就明白了！

```
@time('%Y/%m/%d %H:%M:%S')
def say_hello():
    print('Hello!')
```

等效于：

```
def say_hello():
    print('Hello!')

say_hello = time('%Y/%m/%d %H:%M:%S')(say_hello)
```

而不带参数的装饰器的等效代码是 `say_hello = time(say_hello)` 。对比可以看出，带参数的装饰器的等效代码多了一次函数调用，通过这种方式将装饰器参数传递到内部的两层函数中，这之后便回到了不带参数的装饰器的情形。

← 20 从小独栋升级为别墅区：函数式编程

22 Python 的小招数：其它常用语言特性 →

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目录

第 1 章 入门准备

01 开篇词：你为什么要学 Python ？

02 我会怎样带你学 Python？

03 让 Python 在你的电脑上安家落户

04 如何运行 Python 代码？

第 2 章 通用语言特性

05 数据的名字和种类—变量和类型

06 一串数据怎么存—列表和字符串

07 不只有一条路—分支和循环

08 将代码放进盒子—函数

09 知错能改—错误处理、异常机制

10 定制一个模子—类

11 更大的代码盒子—模块和包

12 练习—密码生成器

第 3 章 Python 进阶语言特性

13 这么多的数据结构（一）：列表、元祖、字符串

14 这么多的数据结构（二）：字典、集合

15 Python大法初体验：内置函数

16 深入理解下迭代器和生成器

17 生成器表达式和列表生成式

18 把盒子升级为豪宅：函数进阶

19 让你的模子更好用：类进阶

20 从小独栋升级为别墅区：函数式编

目前暂无任何讨论

千学不如一看，千看不如一练