楼+之Python实战第10期 (/courses/1190)

Python 高级特性

简介

Python 语言中有很多高级的用法，这些用法比较难理解，但熟悉后可以极大的提高编程的效率和代码的质量。我们选择其中最常用的几种在本次实验中学习。

知识点

- 高阶函数
- lambda 匿名函数
- 偏函数
- 切片
- 列表解析
- 字典解析
- 元组拆包
- 迭代器
- 生成器
- 装饰器

高阶函数

所谓高阶函数（Higher-order function），简言之就是可以接受函数作为参数的函数。

例如我们定义一个函数 `f` ，并设置一个参数，一个可以接受函数的参数：

```
def f(func):
    pass
```

那么，这个函数 `f` 就是高阶函数。

下面重点介绍 `sorted` 、 `filter` 、 `map / reduce` 这几种高阶函数。

sorted 函数

排序是编程中最常用的算法之一，Python 提供了 `sorted` 内置函数，可以对列表进行排序：

```
# 对字符串排序
>>> l = ['Python', 'hello',  'linux', 'Git', 'Shiyanlou']
>>> sorted(l)
['Git', 'Python', 'Shiyanlou', 'hello', 'linux']

# 对数字排序
>>> l = [1, 2, 3, -1, -2, -3, 0]
>>> sorted(l)
[-3, -2, -1, 0, 1, 2, 3]
```

`sorted` 函数就是高阶函数，它有一个名为 `key` 的参数，可以接受函数。下面就讲一下 `sorted` 函数的高阶用法，在此之前，先介绍一个很简单的内置函数 `abs` ，它的作用就是获得数字的绝对值：

```
>>> abs(-99)
99
>>> abs(99)
99
>>> abs(-3.14)
3.14
```

`sorted` 的高阶用法举例， `abs` 函数作为 `key` 参数的值，按照数字的绝对值进行排序：

```
>>> l = [1, 2, 3, -1, -2, -3, 0]
>>> sorted(l, key=abs)
[0, 1, -1, 2, -2, 3, -3]
```

思考题

备受玩家期待的篮球游戏大作《NBA 2K19》已于今年 9 月上线，下面的列表 `pp` 是该游戏中能力值排名前五的球员名字和对应的能力值，请使用 `sorted` 函数对该列表按球员名字进行排序：

```
>>> pp = [('Leborn James', 98), ('Kevin Durant', 97), ('James Harden', 96), ('Stephen Curry'
, 95), ('Anthony Davis', 94)]
```

希望得到如下结果：

```
[('Anthony Davis', 94), ('James Harden', 96), ('Kevin Durant', 97), ('Leborn James', 98), (
'Stephen Curry', 95)]
```

filter

和 `sorted` 函数一样， `filter` 也是一个 Python 内建的高阶函数，它的作用是对序列进行过滤。函数的高阶用法，会接受一个函数作为参数， `filter` 也是如此。

`filter` 函数的使用格式：

楼+之Python实战第10期 (/courses/1190)

```
filter(a, b)
# a 为函数，b 为被处理的数据列表
# a 会对 b 中的每个元素进行判断，结果为真则保留，否则舍弃
```

顺便介绍另一个十分简单的函数 `isinstance` ，它接受两个参数，作用是判断数据的数据类型，使用方法如下：

```
>>> isinstance(1, int)
True
>>> isinstance(2.2, float)
True
>>> isinstance('hello', int)
False
>>> isinstance('hello', float)
False
>>> isinstance('hello', str)
True
```

现在举例说明 `filter` 函数的用法。 我们要对一个列表进行过滤，保留其中的 `int` 类型的元素，舍弃其它数据类型的元素：

```
# 被处理的列表 b
>>> b = [9, 'Python', True, 3.14, 2018, -4, abs]
# 作为 filter 函数的第一个参数、用来决定元素去留的函数 a

>>> def a(i):
...     if isinstance(i, int):
...         return True
...     else:
...         return False
...
```

现在我们使用 filter 函数来得到想要的结果：

```
>>> filter(a, b)
<filter object at 0x7f00bd69b198>
```

有点奇怪，得到这样一个结果~ 请仔细看这行打印信息，它指的是一个 filter 对象，也就是说， `filter` 函数的返回值是一个 filter 对象，并不像前面讲的 `sorted` 函数，返回的结果就是列表。我们可以使用 `list` 方法将 `filter` 函数的返回值转换为列表，结果就是这样：

```
>>> list(filter(a, b))
[9, True, 2018, -4]
```

结果和我们期望的差不多，过滤掉了 float、字符串、函数，值得注意的是 `True` 这一项被保留下来，因为在判断数值的，`True` 和 1 是等值，`False` 和 0 是等值，所以它们也就属于 `int` 类型了：

```
>>> True == 1
True
>>> False == 0
True
```

思考题

仍然对上一个思考题中的 `pp` 列表进行处理，使用 `filter` 函数对其进行过滤，得到能力值为 96+（含 96）的球员的数据：

```
>>> pp = [('Leborn James', 98), ('Kevin Durant', 97), ('James Harden', 96), ('Stephen Curry'
, 95), ('Anthony Davis', 94)]
```

期望得到如下结果：

```
[('Leborn James', 98), ('Kevin Durant', 97), ('James Harden', 96)]
```

map / reduce

下面介绍一下这两个很酷很 pythonic 的函数。同样地，它们都是高阶函数。

map

`map` 函数接受两个参数，格式和 `filter` 一样：

```
map(a, b)
# a 是函数，用来处理 b 参数
# b 是可迭代对象
```

举例说明：

有这样一个列表 b：

```
>>> b = [1, -2, 3, -4, -5, 6, 7, 8, -9]
```

我们要得到它的每个元素的平方的值，并生成新的列表。先写 a 函数，用来处理每个元素：

```
>>> def a(i):
...     return i**2
...
```

现在，就可以使用 `map` 函数来获得想要的结果了：

```
>>> map(a, b)
<map object at 0x7f00bd69b160>
```

和 `filter` 函数类似， `map` 函数的返回值也是一个 map 对象，它是 map 类的实例，使用 `list` 方法将其转换为列表：

```
>>> list(map(a, b))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

reduce

`reduce` 函数的使用格式依然如此：

```
reduce(a, b)
# a 函数用来处理 b
# b 是可迭代对象
```

与 `map` 不同的是， `reduce` 的 a 函数对 b 序列做积累处理，我们来举例说明。 同样的 b 列表：

```
>>> b = [1, -2, 3, -4, -5, 6, 7, 8, -9]
```

现在我们要对 b 列表中的元素进行求和，即实现 `sum(b)` 的结果，定义 a 函数：

```
>>> def a(x, y):
...     return x + y
...
```

需要注意的一点： `reduce` 在 python2 中同 `map` 函数一样可以直接使用，在 python3 中需要从 `functools` 库中引入：

```
>>> from functools import reduce
```

现在可以使用 `reduce` 函数对 b 列表进行求和了，它的返回值是一个确定的数值，与 `sum(b)` 的结果一样：

```
>>> map(lambda x, y: x + y, a, b)
5
>>> sum(b)
5
```

思考题

使用 `map` 方法得到所有球员名字的全小写列表：

```
>>> pp = [('Leborn James', 98), ('Kevin Durant', 97), ('James Harden', 96), ('Stephen Curry'
, 95), ('Anthony Davis', 94)]
```

期望得到如下结果：

```
['leborn james', 'kevin durant', 'james harden', 'stephen curry', 'anthony davis']
```

lambda

匿名函数，顾名思义，这类函数没有函数名，这个特点的好处是避免自定义变量名冲突、减少代码量、使代码结构更加紧凑。缺点是不可重复使用。

Python 通过 `lambda` 提供了对匿名函数的支持，使用方法很简单，看下面的例子：

```
>>> double = lambda x: x * 2
>>> double(5)
10
```

上面的例子中 double 这个变量其实就是一个匿名函数，使用的时候直接 `double(x)` 就会执行函数。

例子中使用 `lambda` 定义了一个匿名函数。 `lambda` 返回值时不需要 `return` 。

`lambda` 函数通常用在需要传入一个函数作为参数，并且这个函数只在这一个地方使用的情况下，匿名函数一般会作为一个参数传递，冒号前面是参数，后面是返回值。它的好处是没有函数名，可以避免变量冲突，限制是只能有一个表达式。

还是举例说明：

```
b = [1, -2, 3, -4, -5, 6, 7, 8, -9]
```

对 b 列表中的元素进行求平方，然后得到新的列表，用 lambda 可以这样写：

```
>>> list(map(lambda i: i**2, b))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

楼+之Python实战第10期 (/courses/1190)

其中 `lambda i: i**2` 就是一个完整的匿名函数，冒号前面的 `i` 是参数，冒号后面的 `i**2` 是匿名函数的返回值，相当于具名函数的 `return` ，该匿名函数的作用等同于：

```
>>> def a(i):
...     return i**2
...
```

匿名函数也可以赋值给变量：

```
>>> b = [1, -2, 3, -4, -5, 6, 7, 8, -9]
>>> list(map(lambda i: i**2, b))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> a = lambda i: i**2
>>> list(map(a, b))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

匿名函数亦可作为函数的返回值。

举个例子，将下面代码写入 `test.py` 文件（这个例子没有实际意义，仅作为说明）：

```
from math import pi
def f(a):
    return lambda: a**2*pi

if __name__ == '__main__':
    print(f(2))
    print(f(2)())
```

在终端执行文件：

```
$ python3 test.py
<function f.<locals>.<lambda> at 0x7f79d7401c80>
12.566370614359172
```

- 拓展阅读 《知乎优质回答 - 关于 Python 中 Lambda 表达式的使用》(https://www.zhihu.com/question/20125256/answer/14058285)

思考题

1. 将前面几个高阶函数中作为参数的函数，用 lambda 实现
2. 使用 `lambda` 和 `map` 获取 `pp` 列表中球员的名字的全大写列表并使用 `sorted` 函数进行降序排序（一行代码实现）

```
>>> [('LeBron James', 98), ('Kevin Durant', 97), ('James Harden', 96), ('Stephen Curry'
, 95), ('Anthony Davis', 94)]
```


期望得到如下结果：

```
['STEPHEN CURRY', 'LEBORN JAMES', 'KEVIN DURANT', 'JAMES HARDEN', 'ANTHONY DAVIS']
```

练习题：Lambda 匿名函数的应用

很多初次接触 Lambda 的同学都有同样的迷惑，匿名函数该如何使用呢？本节我们通过一个简单的排序题目来学习 Lambda 表达式在函数调用过程中的一个应用场景。

创建一个代码文件 `/home/shiyanlou/lambdatest.py` ，完善下面的代码片段：

```
#!/usr/bin/env python3

list1 = [('Shi',100), ('Yan', 75), ('Lou', 200), ('Plus', 80)]

sortedlist = sorted(list1, key=TODO)

print(sortedlist)
```

列表 list1 为一个二元组的列表，我们需要使用每个二元组的第二个元素进行比较排序，从小到大排列。

sorted 为内置的排序函数，其中需要将列表作为参数传递给 sorted，key 参数表示指定排序的关键字，可以理解为告诉 sorted 函数该怎么排序。在这段代码中，我们需要用到 Lambda 表达式来处理，即提取每个二元组中的第二个元素来进行排序。

使用 Lambda 表达式修复上面程序的 TODO 部分，然后执行 `python3 /home/shiyanlou/lambdatest.py` ，输出的结果应该是：

```
[('Yan', 75), ('Plus', 80), ('Shi', 100), ('Lou', 200)]
```

解决这个问题还需要用到以下额外的知识点：

- Python3 sorted 官方文档 (https://docs.python.org/3/library/functions.html#sorted)

完成后点击 下一步 ，系统将自动检测完成结果。

偏函数

之前介绍到函数有五种参数，其中之一是默认参数，它可以降低函数调用的复杂度。本节要介绍的偏函数，可以达到同样的效果，且可以高效创建含有特定默认参数的函数。

例如我们创建一个函数，求数字的平方：

```
>>> def cal_sq(i):
...     return i**2
...
>>> cal_sq(2)
4
>>> cal_sq(3)
9
```

修改需求，创建一个可以求任意数的 m 次方的函数：

```
>>> def cal_power(i, m):
...     return i**m
...
>>> cal_power(2,4)
16
```

假设我们通常需要计算数值的平方，可以使用默认参数对 `cal_power` 函数进行优化：

```
>>> def cal_power(i, m=2):
...     return i**m
...
>>> cal_power(2,4)
16
>>> cal_power(2)
4
```

现在，我们需要一个函数，用在某个通常需要计算数值的 4 次方的场景中，就不必像上文那样再次重新编写函数了，此时就用到了 `partial` 函数，它可以高效创建有着特定的默认参数的函数。

与 `reduce` 一样，创建偏函数需要用到的 `partial` 函数也是来自 `functools` 这个模块：

```
>>> from functools import partial
>>> cal_power4 = partial(cal_power, m=4)
>>> cal_power4(3)
81
>>> cal_power4(3, m=2)
9
```

`partial` 函数接受两个参数，第一个参数为原函数名，第二个参数为原函数中的默认参数， `partial` 函数的返回值就是一个我们需要的新函数。

切片（slice)

切片用于获取一个序列（列表或元组）或者字符串的一部分，返回一个新的序列或者字符串，使用方法是中括号中指定一个列表的开始下标与结束下标，用冒号隔开，切片在先前的实验中讲解字符串的时候有介绍过，不只是字符串，列表或元组使用切片也非常常见。

以列表为例：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters[1:3]
['b', 'c']
```

下标可以是正数，也可以是负数，letters 下标的对应关系：

```
 0  1  2  3  4  5  6
 a  b  c  d  e  f  g
-7 -6 -5 -4 -3 -2 -1
```

所以也可以这样写：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters[1:-4]
['b', 'c']
```

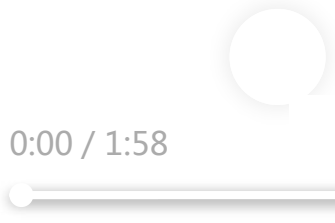
省略 start 则默认 start 为 0，省略 end 则默认截取到最后：

```
>>> letters[:4]
['a', 'b', 'c', 'd']
>>> letters[4:]
['e', 'f', 'g']
```

可以利用切片返回新列表的特性来复制一个列表：

```
>>> copy = letters[:]
>>> copy
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

切片操作视频：



列表解析（list comprehension）

楼+之Python实战第10期（/courses/1190）

列表解析（list comprehension），也称为列表推导，是从 Python 2.0 就被添加进来的新特性，提供了一种简单优雅的方式操作列表元素，看下面一个例子：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# 获取 numbers 中的所有偶数
>>> [x for x in numbers if x % 2 == 0]
[2, 4, 6, 8, 10]
# 对 numbers 的每个数求平方
>>> [x * x for x in numbers]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Python 中提供了一些高阶函数例如 `map` ，`filter` 以及匿名函数 `lambda` ，高阶函数的意思是可以把函数作为参数传入，并利用传入的函数对数据进行处理的函数。

上面例子中的操作，我们同样可以借助高阶函数完成：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> f = filter(lambda x: x % 2 == 0, numbers)
>>> m = map(lambda x: x * x, numbers)
```

对比这俩种实现，个人觉得使用列表解析更为简洁易读一点。另外，由于使用高阶函数增加了调用函数的开销，以至它的使用效率不如列表解析，这就难怪连 Python 的作者也推荐使用列表解析了。

列表解析操作视频：

组合前面的代码，我们来获取 numbers 列表中所有偶数的平方值，使用列表解析就更能凸显它的简洁与高效：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> f = filter(lambda x: x%2==0, numbers)  # 首先获取原列表中的所有偶数
>>> list(map(lambda x: x**2, f))  # 使用 map 方法获得所有偶数的平方，再用 list 方法将其转换为列表
[4, 16, 36, 64, 100]
>>> [x**2 for x in numbers if x%2==0]  # 列表解析的威力，直接使用 if 语句过滤掉所有奇数，然后计算各个元素的平方
[4, 16, 36, 64, 100]
>>>
```

字典解析（dict comprehension）

理解了列表解析，字典解析也是类似的，处理的对象是字典中的 key 和 value。直接看例子吧：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> {k:v*v for k, v in d.items()}
{'a': 1, 'b': 4, 'c': 9}
```

需要注意的是字典是不能被迭代的，需要使用字典的方法 `items()` 把字典变成一个可迭代对象。

元组拆包

Python 有个很强大的元组赋值特性，它允许等号左边的变量依次被元组内的元素赋值，这就是元组拆包，例如：

```
>>> t = ('hello', 'shiyanlou')
>>> a, b = t
>>> a
'hello'
>>> b
'shiyanlou'
```

元组拆包还有一个比较特殊的格式，就是用 `*` 达到拆包的作用，举例说明：

```
>>> t = ('Tom', 11)
>>> print('I\'m {}, I\'m {} years old.'.format(*t))
I'm Tom, I'm 11 years old.
```

`*t` 的结果就是将元组里的所有元素拆分出来。在前面的课程中，函数那一节讲到的函数参数，其中有一种可变参数，就是用的同样的原理。 当然了，列表、集合也可以使用这个方法，但它们用得极少。我们要需要根据实际需求来写出最符合应用场景的代码。

迭代器

如果学过设计模式中的迭代器模式，那么就很容易理解这个概念。要理解迭代器，首先需要明白迭代器和可迭代对象的区别。一个一个读取、操作对象称为迭代，Python 中，可迭代（Iterable）对象就是你能用 `for` 循环迭代它的元素，比如列表是可迭代的：

楼+之Python实战第10期 (/courses/1190)

```
>>> letters = ['a', 'b', 'c']
>>> for letter in letters:
...     print(letter)
...
a
b
c
>>>
```

而迭代器是指，你能用 `next` 函数不断的去获取它的下一个值，直到迭代器返回 `StopIteration` 异常。所有的可迭代对象都可以通过 `iter` 函数去获取它的迭代器，比如上面的 `letters` 是一个可迭代对象，那么这样去迭代它：

```
>>> letters = ['a', 'b', 'c']
>>> it = iter(letters)
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

所有的迭代器其实都实现了 `__iter__` 和 `__next__` 这俩个魔法方法， `iter` 与 `next` 函数实际上调用的是这两个魔法方法，上面的例子背后其实是这样的：

```
>>> letters = ['a', 'b', 'c']
>>> it = letters.__iter__()
>>> it.__next__()
'a'
>>> it.__next__()
'b'
>>> it.__next__()
'c'
>>> it.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

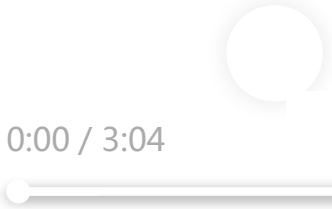
迭代器的另一种实现方式， `__iter__ + __next__` ：

```
>>> class Test:
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...     def __iter__(self):
...         return self
...     def __next__(self):
...         self.a += 1
...         if self.a > self.b:
...             raise StopIteration()
...         return self.a
...
>>> test = Test(0, 5)   # Test 类的实例就是迭代器
>>> next(test)
1
>>> next(test)
2
>>>
```

总结下：

能被 `for` 循环访问的都是可迭代对象，能被 `next` 函数获取下一个值的是迭代器。

迭代器示例视频：



练习题：修复错误代码

请将下面的代码写入到 `/home/shiyanlou/itertest.py` ：

```
#!/usr/bin/env python3

testlist = ['Linux', 'Java', 'Python', 'DevOps', 'Go']

it = iter(testlist)
print("Loop Start...")
while True:
    try:
        course = next(it)
        print(course)
    except OverflowError:
        print("Loop End")
```

运行这段代码发现结果存在错误，需要修复代码中的 BUG，预期正常的输出应该为：

楼+之Python实战第10期 (/courses/1190)

```
$ python3 /home/shiyanlou/itertest.py
Loop Start...
Linux
Java
Python
DevOps
Go
Loop End
```

修复之后点击 下一步 ，系统会自动评测代码。

生成器

上面已经介绍了可迭代对象和迭代器的概念，生成器首先它是一个迭代器，和迭代器一样，生成器只能被遍历迭代一次，因为每次迭代的元素不是像列表元素一样，已经在内存中，每迭代一次，生成一个元素。

生成器和迭代器的主要区别在于：

1、它们的创建方式不同

2、生成器有一些特殊方法是迭代器不具有的

我们常见常用的生成器和迭代器作用都差不多，只是创建方式有所不同，下面介绍创建生成器的两种方法。

方法一，使用生成器表达式创建一个生成器并迭代：

```
>>> g = (x**x for x in range(1, 4))
>>> g
<generator object <genexpr> at 0x10d1a5af0>
>>> for x in g:
...     print(x)
...
1
4
27
```

和列表解析有点像，只不过使用的是圆括号。不同于列表可以反复迭代，迭代完一次之后再迭代这个生成器，它不会打印元素，也不会报错。

使用生成器有什么好处呢？因为生成器不是把所有元素存在内存，而是动态生成的，所以当你要迭代的对象有非常多的元素时，使用生成器能为你节约很多内存，这是一个内存友好的特性。

方法二，使用 `yield` 编写生成器函数，函数的返回值就是生成器。

楼+之Python实战第10期 (/courses/1190)

`yield` 的使用方法和 `return` 类似。不同的是， `return` 可以返回有效的 Python 对象，而 `yield` 返回的是一个生成器，函数碰到 `return` 就直接返回了，而使用了 `yield` 的函数，到 `yield` 返回一个元素，当再次迭代生成器时，会从 `yield` 后面继续执行，直到遇到下一个 `yield` 或者函数结束退出。

下面是一个迭代斐波那契数列前 n 个元素的列子：

```
>>> def fib(n):
...     current = 0
...     a, b = 1, 1
...     while current < n:
...         yield a
...         a, b = b, a + b
...         current += 1
...
```

上面的函数使用了 `yield` 返回的是一个生成器。如果我们要迭代斐波那契数列的前 5 个元素，先调用函数返回的一个生成器：

```
>>> f5 = fib(5)
>>> f5
<generator object fib at 0x10d1a5888>
```

迭代：

```
>>> for x in f5:
...     print(x)
...
1
1
2
3
5
```

装饰器

装饰器可以为函数添加额外的功能而不影响函数的主体功能。在 Python 中，函数是第一等公民，也就是说，函数可以做为参数传递给另外一个函数，一个函数可以将另一函数作为返回值，这就是装饰器实现的基础。装饰器本质上是一个函数，它接受一个函数作为参数。看一个简单的例子，也是装饰器的经典运用场景，记录函数的调用日志：

```
>>> from datetime import datetime
>>> def log(func):
...     def decorator(*args, **kwargs):
...         print('Function ' + func.__name__ + ' has been called at ' + datetime.now().strf
time('%Y-%m-%d %H:%M:%S'))
...         return func(*args, **kwargs)
...     return decorator
...
>>> @log
... def add(x, y):
...     return x + y
...
>>> add(1, 2)
Function add has been called at 2017-08-29 13:11:48
3
```

@ 是 Python 提供的语法糖，语法糖指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。

上面的代码中 `*args` 和 `**kwargs` 都是 Python 中函数的可变参数。 `*args` 表示任何多个无名参数，是一个元组， `**kwargs` 表示关键字参数，是一个字典。这两个组合表示了函数的所有参数。如果同时使用时， `*args` 参数列要在 `**kwargs` 前。

它等价于进行了下面的操作：

```
>>> def add(x, y):
...     return x + y
...
>>> add = log(add)
>>> add(1, 2)
Function add has been called at 2017-08-29 13:16:02
3
```

也就是说，调用了 `log` 函数，把 `add` 函数作为参数，传了进去。 `log` 函数返回了另外一个函数 `decorator` ，在这个函数中，首先打印了日志信息，然后回调了传入的 `func` ，也就是 `add` 函数。

你可能已经发现了，执行完 `add = log(add)` ，或者说用 `@log` 装饰 `add` 后， `add` 其实已经不再是原来的 `add` 函数了，它已经变成了 `log` 函数返回的 `decorator` 函数：

```
>>> add.__name__
'decorator'
```

这也是装饰器带来的副作用，Python 提供了方法解决这个问题：

楼+之Python实战第10期 (/courses/1190)

```
>>> from functools import wraps
>>> def log(func):
...     @wraps(func)
...     def decorator(*args, **kwargs):
...         print('Function ' + func.__name__ + ' has been called at ' + datetime.now().strf
time('%Y-%m-%d %H:%M:%S'))
...         return func(*args, **kwargs)
...     return decorator
...
>>> @log
... def add(x, y):
...     return x + y
...
>>> add.__name__
'add'
```

装饰器的应用场景非常多，我们后续学习 Flask Web 开发的时候会大量使用装饰器来实现 Web 页面的路由等功能。

装饰器操作视频：

总结

本节实验中学习了 Python 编程语言的一些常用的高级用法，这些高级用法需要在项目实际开发的场景中才能够融会贯通，单纯从字面意思很难理解。实验中包括了下面知识点：

- lambda：匿名函数
- 切片：序列和字符串的子序列
- 列表解析：不用 for 也能操作列表元素
- 字典解析：不用 for 也能操作字典元素
- 迭代器：一个个读取元素的对象
- 生成器：只能被迭代一次的迭代器
- yield：返回生成器的 return
- 装饰器：对函数进行修饰

拓展阅读

- 《Python 官方文档（中文）- 迭代器》(http://www.pythondoc.com/pythontutorial3/classes.html#tut-iterators)

- 《Python 官方文档（中文）- 生成器》(http://www.pythondoc.com/pythontutorial3/classes.html#tut-generators)

楼+之Python实战第10期 (/courses/1190)

上一节：挑战：工资计算器读写数据文件 (/courses/1190/labs/8525/document)

下一节：[选学] Python 多进程与多线程 (/courses/1190/labs/8527/document)