

函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

高阶函数

高阶函数英文叫Higher-order function。什么是高阶函数？我们以实际代码为例子，一步一步深入概念。

变量可以指向函数

以Python内置的求绝对值的函数 `abs()` 为例，调用该函数用以下代码：

```
>>> abs(-10)
10
```

但是，如果只写 `abs` 呢？

```
>>> abs
<built-in function abs>
```

可见，`abs(-10)` 是函数调用，而 `abs` 是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

```
>>> x = abs(-10)
>>> x
10
```

但是，如果把函数本身赋值给变量呢？

```
>>> f = abs
>>> f
<built-in function abs>
```

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

如果一个变量指向了一个函数，那么，可否通过该变量来调用这个函数？用代码验证一下：

```
>>> f = abs
>>> f(-10)
10
```

成功！说明变量 `f` 现在已经指向了 `abs` 函数本身。直接调用 `abs()` 函数和调用变量 `f()` 完全相同。

函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于 `abs()` 这个函数，完全可以把函数名 `abs` 看成变量，它指向一个可以计算绝对值的函数！

如果把 `abs` 指向其他对象，会有什么情况发生？

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把 `abs` 指向 10 后，就无法通过 `abs(-10)` 调用该函数了！因为 `abs` 这个变量已经不指向求绝对值函数而是指向一个整数 10！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复 `abs` 函数，请重启Python交互环境。

注：由于 `abs` 函数实际上是定义在 `import builtins` 模块中的，所以要让修改 `abs` 变量的指向在其它模块也生效，要用 `import builtins; builtins.abs = 10`。

传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
```

当我们调用 `add(-5, 6, abs)` 时，参数 `x`，`y` 和 `f` 分别接收 -5，6 和 `abs`，根据函数定义，我们可以推导计算过程为：

```
x = -5
y = 6
f = abs
f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
return 11
```

用代码验证一下：

```
# -*- coding: utf-8 -*-
def add(x, y, f):
    return f(x) + f(y)

print(add(-5, 6, abs))
```

编写高阶函数，就是让函数的参数能够接收别的函数。

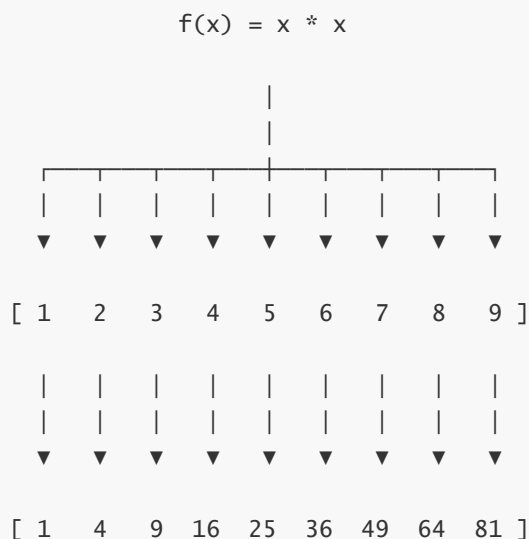
map/reduce

Python内建了 `map()` 和 `reduce()` 函数。

如果你读过Google的那篇大名鼎鼎的论文“[MapReduce: Simplified Data Processing on Large Clusters](#)”，你就能大概明白 `map/reduce` 的概念。

我们先看`map`。`map()` 函数接收两个参数，一个是函数，一个是 `Iterable`，`map` 将传入的函数依次作用到序列的每个元素，并把结果作为新的 `Iterator` 返回。

举例说明，比如我们有一个函数 $f(x)=x^2$ ，要把这个函数作用在一个list `[1, 2, 3, 4, 5, 6, 7, 8, 9]` 上，就可以用 `map()` 实现如下：



现在，我们用Python代码实现：

```
>>> def f(x):
...     return x * x
...
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> list(r)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`map()` 传入的第一个参数是 `f`，即函数对象本身。由于结果 `r` 是一个 `Iterator`，`Iterator` 是惰性序列，因此通过 `list()` 函数让它把整个序列都计算出来并返回一个 `list`。

你可能会想，不需要 `map()` 函数，写一个循环，也可以计算出结果：

```
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print(L)
```

的确可以，但是，从上面的循环代码，能一眼看明白“把 `f(x)` 作用在 `list` 的每一个元素并把结果生成一个新的 `list`”吗？

所以，`map()` 作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的 `f(x)=x2`，还可以计算任意复杂的函数，比如，把这个 `list` 所有数字转为字符串：

```
>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

只需要一行代码。

再看 `reduce` 的用法。`reduce` 把一个函数作用在一个序列 `[x1, x2, x3, ...]` 上，这个函数必须接收两个参数，`reduce` 把结果继续和序列的下一个元素做累积计算，其效果就是：

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比方说对一个序列求和，就可以用 `reduce` 实现：

```
>>> from functools import reduce
>>> def add(x, y):
...     return x + y
...
>>> reduce(add, [1, 3, 5, 7, 9])
25
```

当然求和运算可以直接用Python内建函数 `sum()`，没必要动用 `reduce`。

但是如果要把序列 `[1, 3, 5, 7, 9]` 变换成整数 `13579`，`reduce` 就可以派上用场：

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, [1, 3, 5, 7, 9])
13579
```

这个例子本身没多大用处，但是，如果考虑到字符串 `str` 也是一个序列，对上面的例子稍加改动，配合 `map()`，我们就可以写出把 `str` 转换为 `int` 的函数：

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> def char2num(s):
...     digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7':
7, '8': 8, '9': 9}
...     return digits[s]
...
>>> reduce(fn, map(char2num, '13579'))
13579
```

整理成一个 `str2int` 的函数就是：

```
from functools import reduce

DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8':
8, '9': 9}

def str2int(s):
    def fn(x, y):
        return x * 10 + y
    def char2num(s):
        return DIGITS[s]
    return reduce(fn, map(char2num, s))
```

还可以用 `lambda` 函数进一步简化成：

```
from functools import reduce

DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8':
8, '9': 9}

def char2num(s):
    return DIGITS[s]

def str2int(s):
    return reduce(lambda x, y: x * 10 + y, map(char2num, s))
```

也就是说，假设 Python 没有提供 `int()` 函数，你完全可以自己写一个把字符串转化为整数的函数，而且只需要几行代码！

`lambda` 函数的用法在后面介绍。

练习

利用 `map()` 函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：`['adam', 'LISA', 'barT']`，输出：`['Adam', 'Lisa', 'Bart']`：

```
# -*- coding: utf-8 -*-
# -*- coding: utf-8 -*-
from functools import reduce
def prod(L):
    pass
```

```
print('3 * 5 * 7 * 9 =', prod([3, 5, 7, 9]))
if prod([3, 5, 7, 9]) == 945:
    print('测试成功!')
else:
    print('测试失败!')
```

```
# -*- coding: utf-8 -*-
# -*- coding: utf-8 -*-
from functools import reduce

def str2float(s):
    pass
```

```
print('str2float(\'123.456\') =', str2float('123.456'))
if abs(str2float('123.456') - 123.456) < 0.00001:
    print('测试成功!')
else:
    print('测试失败!')
```

filter

Python内建的 `filter()` 函数用于过滤序列。

和 `map()` 类似，`filter()` 也接收一个函数和一个序列。和 `map()` 不同的是，`filter()` 把传入的函数依次作用于每个元素，然后根据返回值是 `True` 还是 `False` 决定保留还是丢弃该元素。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):
    return n % 2 == 1

list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
# 结果: [1, 5, 9, 15]
```

把一个序列中的空字符串删掉，可以这么写：

```
def not_empty(s):
    return s and s.strip()

list(filter(not_empty, ['A', '', 'B', None, 'C', ' ']))
# 结果: ['A', 'B', 'C']
```

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到 `filter()` 函数返回的是一个 `Iterator`，也就是一个惰性序列，所以要强迫 `filter()` 完成计算结果，需要用 `list()` 函数获得所有结果并返回list。

用filter求素数

计算素数的一个方法是[埃氏筛法](#)，它的算法理解起来非常简单：

首先，列出从 2 开始的所有自然数，构造一个序列：

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

取序列的第一个数 2，它一定是素数，然后用 2 把序列的 2 的倍数筛掉：

3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

取新序列的第一个数 3，它一定是素数，然后用 3 把序列的 3 的倍数筛掉：

5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

取新序列的第一个数 5，然后用 5 把序列的 5 的倍数筛掉：

7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

不断筛下去，就可以得到所有的素数。

用Python来实现这个算法，可以先构造一个从 3 开始的奇数序列：

```
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n
```

注意这是一个生成器，并且是一个无限序列。

然后定义一个筛选函数：

```
def _not_divisible(n):
    return lambda x: x % n > 0
```

最后，定义一个生成器，不断返回下一个素数：

```
def primes():
    yield 2
    it = _odd_iter() # 初始序列
    while True:
        n = next(it) # 返回序列的第一个数
        yield n
        it = filter(_not_divisible(n), it) # 构造新序列
```

这个生成器先返回第一个素数 2，然后，利用 filter() 不断产生筛选后的新的序列。

由于 primes() 也是一个无限序列，所以调用时需要设置一个退出循环的条件：

```
# 打印1000以内的素数：
for n in primes():
    if n < 1000:
        print(n)
    else:
        break
```

注意到 Iterator 是惰性计算的序列，所以我们可以用Python表示“全体自然数”，“全体素数”这样的序列，而代码非常简洁。

练习

回数是指从左向右读和从右向左读都是一样的数，例如 12321，909。请利用 filter() 筛选出回数：

sorted

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python内置的 `sorted()` 函数就可以对list进行排序：

```
>>> sorted([36, 5, -12, 9, -21])
[-21, -12, 5, 9, 36]
```

此外，`sorted()` 函数也是一个高阶函数，它还可以接收一个 `key` 函数来实现自定义的排序，例如按绝对值大小排序：

```
>>> sorted([36, 5, -12, 9, -21], key=abs)
[5, 9, -12, -21, 36]
```

`key`指定的函数将作用于list的每一个元素上，并根据`key`函数返回的结果进行排序。对比原始的list和经过 `key=abs` 处理过的list：

```
list = [36, 5, -12, 9, -21]

keys = [36, 5, 12, 9, 21]
```

然后 `sorted()` 函数按照`keys`进行排序，并按照对应关系返回list相应的元素：

```
keys排序结果 => [5, 9, 12, 21, 36]
               | |   |   |   |
最终结果      => [5, 9, -12, -21, 36]
```

我们再看一个字符串排序的例子：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'])
['Credit', 'Zoo', 'about', 'bob']
```

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于 `'z' < 'a'`，结果，大写字母 `z` 会排在小写字母 `a` 的前面。

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能用一个`key`函数把字符串映射为忽略大小写排序即可。忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给 `sorted` 传入`key`函数，即可实现忽略大小写的排序：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)
['about', 'bob', 'credit', 'zoo']
```

要进行反向排序，不必改动`key`函数，可以传入第三个参数 `reverse=True`：


```
>>> sorted(['bob', 'about', 'zoo', 'Credit'], key=str.lower, reverse=True)
['zoo', 'Credit', 'bob', 'about']
```

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

小结

- `sorted()` 也是一个高阶函数。用 `sorted()` 排序的关键在于实现一个映射函数。

练习

假设我们用一组tuple表示学生名字和成绩：

```
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
```

请用 `sorted()` 对上述列表分别按名字排序：

```
# -*- coding: utf-8 -*-

L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
def by_name(t):
    pass
```

```
L2 = sorted(L, key=by_name)
print(L2)
```

再按成绩从高到低排序：

```
# -*- coding: utf-8 -*-

L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
def by_score(t):
    pass
```

```
L2 = sorted(L, key=by_score)
print(L2)
```

返回函数

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数 `f` 时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum`，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()` 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `args`，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()` 才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用 `f1()`，`f2()` 和 `f3()` 结果应该是 1，4，9，但实际结果是：

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是 9！原因就在于返回的函数引用了变量 `i`，但它并非立刻执行。等到 3 个函数都返回时，它们所引用的变量 `i` 已经变成了 3，因此最终结果为 9。

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def count():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs
```

再看看结果：

```
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

缺点是代码较长，可利用 `lambda` 函数缩短代码。

练习

利用闭包返回一个计数器函数，每次调用它返回递增整数：

```
# -*- coding: utf-8 -*-
def createCounter():
    def counter():
        return 1
    return counter
```

```
# 测试:
counterA = createCounter()
print(counterA(), counterA(), counterA(), counterA(), counterA()) # 1 2 3 4 5
counterB = createCounter()
if [counterB(), counterB(), counterB(), counterB()] == [1, 2, 3, 4]:
    print('测试通过!')
else:
    print('测试失败!')
```

匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以 `map()` 函数为例，计算 $f(x)=x^2$ 时，除了定义一个 `f(x)` 的函数外，还可以直接传入匿名函数：

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

通过对比可以看出，匿名函数 `lambda x: x * x` 实际上就是：

```
def f(x):
    return x * x
```

关键字 `lambda` 表示匿名函数，冒号前面的 `x` 表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写 `return`，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x101c6ef28>
>>> f(5)
25
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):
    return lambda: x * x + y * y
```

练习

请用匿名函数改造下面的代码：

```
# -*- coding: utf-8 -*-
print(L)
```

装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

```
>>> def now():
...     print('2015-3-25')
...
>>> f = now
>>> f()
2015-3-25
```

函数对象有一个 `__name__` 属性，可以拿到函数的名字：

```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

现在，假设我们要增强 `now()` 函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 `now()` 函数的定义，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）。

本质上，decorator就是一个返回函数的高阶函数。所以，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
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个decorator，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的 `now` 变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)`，因此，`wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

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
```

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')`，返回的是 `decorator` 函数，再调用返回的函数，参数是 `now` 函数，返回值最终是 `wrapper` 函数。

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。

练习

请设计一个decorator，它可作用于任何函数上，并打印该函数的执行时间：

```
# -*- coding: utf-8 -*- import time, functools ``# 测试 @metric def fast(x, y):
time.sleep(0.0012) return x + y; @metric def slow(x, y, z): time.sleep(0.1234)
return x * y * z; f = fast(11, 22) s = slow(11, 22, 33) if f != 33: print('测试失
败!') elif s != 7986: print('测试失败!') Run
```

小结

- 在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。
- decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出 'begin call' 和 'end call' 的日志。

再思考一下能否写出一个 `@log` 的decorator，使它既支持：

```
@log
def f():
    pass
```

又支持：

```
@log('execute')
def f():
    pass
```

偏函数

Python的 `functools` 模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

`int()` 函数可以把字符串转换为整数，当仅传入字符串时，`int()` 函数默认按十进制转换：

```
>>> int('12345')
12345
```

但 `int()` 函数还提供额外的 `base` 参数，默认值为 10。如果传入 `base` 参数，就可以做N进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，于是，我们想到，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')
64
>>> int2('1010101')
85
```

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()`，可以直接使用下面的代码创建一个新的函数 `int2`：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 `int2` 函数，仅仅是把 `base` 参数重新设定默认值为 2，但也可以在函数调用时传入其他值：

```
>>> int2('1000000', base=10)
1000000
```

最后，创建偏函数时，实际上可以接收函数对象、`*args` 和 `**kw` 这3个参数，当传入：

```
int2 = functools.partial(int, base=2)
```

实际上固定了 `int()` 函数的关键字参数 `base`，也就是：

```
int2('10010')
```

相当于：


```
kw = { 'base': 2 }  
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把 10 作为 *args 的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)  
max(*args)
```

结果为 10。

小结

- 当函数的参数个数太多，需要简化时，使用 `functools.partial` 可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。