第4章. 函数

理论上我们可以仅使用赋值语句、表达式、if语句和while语句解决任何可计算问题。然而仅使用这些语句编写出的Python脚本通常是冗长且低效的，因为对于那些需要重复执行的操作，每次都需要复制粘贴相同的代码。函数的引入极大地提高了开发效率，是过程化语言的核心技术。

4-1. 函数的定义和调用

（教程：4.7、4.8.2、4.8.3、4.9）
（语言参考手册：6.3.4、7.6、8.7）

编程语言中的所谓“函数”其实指的是对代码段的包装。这些代码段在程序执行过程中被多次执行，因此将它们写成函数，在需要执行它们的地方“调用（call）”这些函数，可以极大地缩短程序。而函数之所以得名“函数”，是因为它们都接受一个参数列表，调用完成后还会得到一个返回值，因此可以被视为将参数映射到返回值的黑箱，与数学中的函数具有相似性。

创建函数的基本方法是使用“函数定义（function definition）”语句，其语法为：

```
def funcname([parameter_list]):
    suite
```

执行该语句会导致在内存空间中创建一个函数对象，该对象默认将存在到脚本执行结束。funcname为引用该函数对象的标识符，也就是该函数的“函数名（function name）”，同时相应字符串也会被该函数对象的__name__特殊属性引用。parameter_list代表该函数的“形式参数列表（parameter list）”，而方括号表明它是可选的，也就是说函数也可以没有任何形式参数。而suite则代表调用该函数时被执行的代码块，亦即该函数的“函数体（function body）”。当形式参数列表非常长时，PEP 8建议应利用圆括号所支持的隐式行拼接将其写为多行。

一般而言，函数会通过其函数体内的return语句给出“返回值（returned value）”，该语句的语法为：

```
return [expr]
```

这表示对表达式expr求值，以得到的对象作为返回值结束这次函数调用。如果省略了expr，则以None作为返回值。如果函数体执行完时都没有遇到return语句，则默认也是以None作为返回值。PEP 8建议函数体中的return语句遵循如下规范：如果该函数体分为若干分支，那么这些分支要么全部不包含return语句，要么全部包含return语句，而不应让部分分支包含return语句，部分分支隐式返回None。

函数调用的语法为：

```
funcname([argument_list])
```

其中funcname是被调用函数的函数名；而argument_list是这次函数调用的“实际参数列表（argument list）”，必须与被调用函数的形式参数列表保持一致。函数调用可以被视为一种表达式，对其求值得到的就是被调用函数的返回值，因此可以嵌入其他运算符形成的表达式中使用。表2-2已经说明，函数调用的优先级相当高，与属性访问相同。同样，PEP 8建议当实际参数列表非常长时应利用圆括号所支持的隐式行拼接将其写为多行。

下面给出一个简单例子，即将上一章中number_sign_game_3.py的代码包装成一个函数，然后通过函数调用来执行：

```
import re


#定义函数number_sign_game()。    该函数没有形式参数。
def number_sign_game():
    print("这是一个判断整数符号的小游戏。\n")

    while True:
        s = input("请输入一个整数: ")

        if s == "quit":
            break

        if re.match(r"^[+\-]?[0-9]+$", s):
            n = int(s)
            if n < 0:
                print("这是一个负数。")
            elif n > 0:
                print("这是一个正数。")
            else:
                print("这是零。")
        else:
            print("您输入的并非整数。")

    print("你已经退出了游戏。\n")


#调用函数number_sign_game()。
number_sign_game()
```

请将上述代码保存为number_sign_game_4.py并验证。

如果一个函数在函数体内调用了自身，就被称为“递归函数（recursive function）”。递归函数在计算理论中非常重要，但在工程实践中却应尽量避开。

下面则是一个带参数且明确给出了返回值的函数的例子：

```
#这是一个通过递归实现阶乘的函数。
def factorial(n):
    if n > 1:
        return n * factorial(n-1)
    else:
        return 1
```

注意该函数在函数体内通过return语句调用了它自己，因此是一个递归函数。将上述代码保存为factorial1.py，然后通过如下命令行和语句验证该函数的作用：

```
$ python3 -i factorial1.py

>>> factorial(5)
120
>>> factorial(4)
24
>>> factorial(3)
6
>>> factorial(2)
2
>>> factorial(1)
1
>>> factorial(0)
0
>>>
```

递归函数能够以较精妙的方式解决复杂问题，且可以使代码变得简洁，但代价是它们被调用时会占用更多内存，因为每次函数调用都会在函数栈里创建一个帧。为了避免因递归而耗尽物理内存，Python解释器通常会限制递归的层数，例如CPython默认限制1000层。我们在编写程序时也应尽量避免使用递归，例如上面的函数完全可以这样实现：

```
#这是一个通过循环实现阶乘的函数。
def factorial(n):
    product = 1

    while n > 1:
        product = product*n
        n = n-1

    return product
```

将上述代码保存为factorial2.py，然后通过如下命令行和语句验证：

```
$ python3 -i factorial2.py

>>> factorial(5)
120
>>> factorial(4)
```

```
24
>>> factorial(3)
6
>>> factorial(2)
2
>>> factorial(1)
1
>>> factorial(0)
0
>>>
```

上述例子证明了前面介绍的函数的特点：它们可以被视为将参数映射到返回值的黑箱。调用函数的人没有必要知道一个函数是怎么实现的，即无需查看函数体的代码，只需要知道实际参数列表和返回值的映射关系，就足以正确使用该函数。

函数的黑箱特性为软件工程带来了“模块化（modularization）”的思想，即将一个复杂的项目分解为若干模块，每个模块又进一步划分为若干子模块，……直到最终模块的规模小到能够被一个人编写。而这些模块的本质就是若干函数的集合，每个人编写好函数后还需要提供这些函数的说明文档，以使其他人在不了解其实现细节的情况下也能使用这些函数。通过这种方式，成千上万人可以参与同一项目，并使项目有序进行。这是过程化语言隐含的项目管理方式。

我们已经知道函数名是引用相应函数对象的标识符。也可以让其他标识符引用该函数，进而成为该函数的“别名（aliases）”。下面的例子说明了这点：

```
>>> def f1():
...     return "Hello"
...
>>> f2 = f1
>>> f2()
'Hello'
>>> f1.__name__
'f1'
>>> f2.__name__
'f1'
>>>
```

前面已经提到，函数调用的实际参数列表必须与函数的形式参数列表相一致。而相对于其他编程语言，Python函数的形式参数列表的语法更加复杂，使实际参数与之相一致变得更加困难，因此调用函数时必须更小心。Python函数的形式参数列表的完整格式是：

```
pos1, pos2, ..., /, pk1, pk2, ..., *, kwd1, kwd2, ...
```

其中“/”和“*”不是参数，而是分隔符：

➤ “/”之前的参数（pos1，pos2，……）被视为“仅位置形式参数（position-only parameters）”，实际参数只能通过位置与它们对应。
➤ “*”之后的参数（kwd1，kwd2，……）被视为“仅关键字形式参数（keyword-only parameters）”，实际参数只能通过关键字与它们对应。
➤ “/”和“*”之间的参数（pk1，pk2，……）被视为“位置或关键字形式参数（position-or-keyword parameters）”，实际参数可以通过位置与它们对应，也可以通过关键字与它们对应。

在一个函数定义中，这三种形式参数并不需要都出现：如果没有仅位置形式参数，则可以省略“/”；如果没有仅关键字形式参数，则可以省略“*”；如果没有位置或关键字形式参数，则可以将“/”和“*”合并为“/*”。

前面例子中的函数定义都只有位置或关键字形式参数，因此同时省略了“/”和“*”。事实上，在factorial1.py和factorial2.py中定义的factorial()函数也可以通过如下方式调用：

```
>>> factorial(n=5)
120
>>>
```

因为形式参数n是一个位置或关键字形式参数，既可以通过位置0指定（“位置”即形式参数在形式参数列表中的索引），又可以通过关键字“n”指定（“关键字”即形式参数的名字）。

调用factorial()时只需要提供一个实际参数，与唯一的一个形式参数对应。当形式参数列表中有多个形式参数时，实际参数列表也必须包含多个实际参数，而保证两者一致的规则是：将所有基于位置对应的实际参数放在前面，所有基于关键字对应的实际参数放在后面，前者相互间的位置是固定的，而后者相互间的位置可以任意交换。

下面的例子integer_adder.py定义了三个功能相同的函数：add1()、add2()和add3()，它们分别只具有仅位置形式参数、仅关键字形式参数和位置或关键字形式参数：

```
#该函数只具有仅位置形式参数。
def add1(x, y, /):
    return x+y


#该函数只具有仅关键字形式参数。
def add2(*, x, y):
    return x+y


#该函数只具有位置或关键字形式参数。
def add3(x, y):
    return x+y
```

请通过下面的命令行和语句验证这三种参数的区别：

```
$ python3 -i integer_adder.py

>>> add1(1, 2)
3
>>> add1(x=1, y=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add1() got some positional-only arguments passed as keyword
arguments: 'x, y'
>>> add1(x=1, 2)
  File "<stdin>", line 1
    add1(x=1, 2)
                ^
SyntaxError: positional argument follows keyword argument
>>> add1(1, y=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add1() got some positional-only arguments passed as keyword
arguments: 'y'
>>>
>>>
>>> add2(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add2() takes 0 positional arguments but 2 were given
>>> add2(x=1, y=2)
3
>>> add2(x=1, 2)
  File "<stdin>", line 1
    add2(x=1, 2)
                ^
SyntaxError: positional argument follows keyword argument
>>> add2(1, y=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add2() takes 0 positional arguments but 1 positional argument
(and 1 keyword-only argument) were given
>>>
>>>
>>> add3(1, 2)
3
>>> add3(x=1, y=2)
3
>>> add3(x=1, 2)
  File "<stdin>", line 1
    add3(x=1, 2)
                ^
SyntaxError: positional argument follows keyword argument
>>> add3(1, y=2)
3
>>> add3(2, x=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add3() got multiple values for argument 'x'
>>>
```

这里应特别注意的是“add3(2, x=1)”，它先基于位置对应将2传递给x，然后基于关键字对应将3传递给x，因此产生了冲突。

下面给出一个同时具有三种类型形式参数的函数定义的例子：

```
#该函数包含如下参数：
# 仅位置参数op取如下字符串来指定运算：
#   'add': 加法
#   'sub': 减法
#   'mult': 乘法
#   'div': 除法
# 位置或关键字参数x和y为两个操作数，取浮点数。
# 仅关键字参数floordiv取布尔值，以表明是否进行整除。
def arithmetic_calculator(op, /, x, y, *, floordiv):
    #确保两个操作数都是浮点数。
    x = float(x)
    y = float(y)

    #根据指定的运算返回相应结果。   如果指定的运算不能识别，则返回NotImplemented。
    if op == 'add':
        return x + y
    elif op == 'sub':
        return x - y
    elif op == 'mult':
        return x * y
    elif op == 'div':
        if floordiv:
            return x // y
        else:
            return x / y
    else:
        return NotImplemented
```

请将上述代码保存到arithmetic_calculator.py中，然后通过下面的命令行和语句验证该函数的调用结果：

```
$ python3 -i arithmetic_calculator.py

>>> arithmetic_calculator('add', 14.0, 3.2, floordiv=False)
17.2
>>> arithmetic_calculator('sub', x=14.0, y=3.2, floordiv=False)
10.8
>>> arithmetic_calculator('mult', 14.0, 3.2, floordiv=False)
44.800000000000004
>>> arithmetic_calculator('div', 14.0, y=3.2, floordiv=False)
4.375
>>> arithmetic_calculator('div', 14.0, floordiv=True, y=3.2)
4.0
>>> arithmetic_calculator('power', 14.0, 3.2, floordiv=False)
NotImplemented
>>> arithmetic_calculator('concat', 'a', 'b', floordiv=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/float_calculator.py", line 11, in calculate
    x=float(x)
ValueError: could not convert string to float: 'a'
>>>
```

当然，我们也可以构造出仅包含两种类型形式参数的函数定义，但它们与同时具有三种类型形式参数的函数定义没有本质区别，故本书省略。

最后，PEP 8建议在函数定义语句前后都添加两个空行，而函数调用包含基于关键字对应的实际参数时，“=”两端不要添加空格。

4-2. 函数定义和调用的内部机制

（教程：4.7）
（语言参考手册：3.2、6.3.4）
（标准库：内置类型）

上一节介绍了函数定义和函数调用的语法，本节将讨论这些语法背后发生了什么。显然，本节将无法避免地涉及一些Python解释器的内部细节。

函数定义语句被执行时会在内存空间中创建一个函数对象和一个代码对象。（如果涉及闭包还会创建若干单元对象，这将在第6章讨论。）函数对象被作为函数名的标识符引用。代码对象是将函数体作为实际参数调用内置函数compile()得到的，并通过函数对象的__code__特殊属性引用。（compile()会在第6章被进一步讨论。）

函数调用被执行时会转化为对函数类型的__call__魔术属性的调用，而也就是说：

```
funcname([argument_list])
```

等价于

```
type(funcname).__call__(self, [argument_list])
```

其中self代表函数对象本身。第3章已经提到函数类型实现了魔术属性__call__，下面将讨论该魔术属性的执行细节。

函数类型的__call__每被调用一次，都会在内存空间中创建一个“帧对象（frame objects）”，该对象具有下列只读属性：

➤ f_back：引用前一个帧对象，以形成一个栈，即所谓的“函数栈（function stacks）”。位于函数栈的栈底的帧对象的f_back属性引用None。
➤ f_code：引用绑定到该帧对象的代码对象，与导致创建该帧对象的被调用函数对象的__code__属性相同。
➤ f_lasti：引用一个整数，解读为代码对象的一个索引，代表目前刚被执行完的指令。

➤ f_locals：引用实现本地变量的变量字典。
➤ f_globals：引用实现全局变量的变量字典。
➤ f_builtins：引用用于查找内置变量的内部对象。

以及下列可读可写属性：

➤ f_trace：引用绑定到在代码对象被执行期间产生的各类事件的函数，使得它们在相应事件产生时被自动调用，以支持调试（将在第16章讨论）。默认引用None。
➤ f_lineno：引用一个整数，为当前执行指令在源代码中对应行的行号。调试器可以通过修改该属性实现跳转（即指定接下来被执行的语句）。默认随f_lasti的变化而自动变化。

帧对象还有一个函数属性clear()，调用它会清空f_locals引用的变量字典。

以上就是关于帧对象的基本知识，本书不会进一步深入讨论。这里需要强调的是，帧对象并不代表某个函数，而是代表调用某个函数这一行为。换句话说，同一个函数被调用n次会产生n个独立的帧对象。帧对象是用来执行被调用函数的函数体的，该函数返回后帧对象就会被销毁。

第3章已经提到过，Python脚本的执行最终演变为对函数对象的调用。为了便于讨论，假设Python解释器是以单线程方式运行的。（事实上通过标准库中的threading模块可以实现多线程。）最简单的情况是调用一个函数，执行完，再调用下一个函数。在这种情况下任意时刻函数栈中都只存在一个帧对象，对应当前被执行的函数调用。

然而一个函数可以在其函数体内调用另一个函数，这会导致当前函数调用执行完成前就调用另一个函数，进而为该函数创建新的帧对象。易知，在新创建的帧对象被销毁前，原来就存在的帧对象不可能被销毁，这就使得函数栈中存在多个帧对象：位于栈顶的帧对象最先被销毁，位于栈底的帧对象最后被销毁。而帧对象之间则通过f_back属性相互连接成栈。

第3章也已经提到过，变量字典的内存开销比较大。而每个帧对象都有自己的实现本地变量的变量字典（实现全局变量的变量字典被所有帧对象共享），因此帧对象的内存开销也比较大。这意味着，如果被嵌套的函数调用导致函数栈过长，就有可能耗尽物理内存，导致Python解释器崩溃。而递归函数在函数体中调用了自身，很容易形成非常长的函数栈，这就是需要避免使用递归的原因。

接下来讨论函数调用过程中参数的传递，重点是解释实际参数和形式参数是如何保持一致的。

首先，形式参数只是一些占位符，当函数定义语句被执行后，不论是函数对象还是代码对象都没有为这些形式参数分配内存空间，而只是将形式参数作为一种说明信息储存下来。在函数被调用时，当帧对象被创建后，就会在帧对象的f_locals属性引用的变量字典中按照形式参数说明创建一个列表。我们不妨假设该列表名为“arguments”，那么arguments列表

中的成员就按照其顺序与形式参数列表中的形式参数一一对应，不论该形式参数是仅位置形式参数、仅关键字形式参数还是位置或关键字形式参数。

实际参数的传入本质上就是一系列赋值操作：首先将实际参数列表中所有基于位置对应的实际参数提取出来，依次赋值给arguments列表中的相应成员。然后，剩下的基于关键字对应的实际参数则会按照形式参数说明被赋值给arguments列表中的相应成员。（这就是基于关键字对应的实际参数可以任意排序的原因。）这一赋值过程所期待的结果是arguments列表中的每个成员都恰好被实际参数赋值一次，而在下列3种情况下则会抛出TypeError异常：

- arguments列表中的某个成员未被任何实际参数赋值。
- 某个实际参数在arguments列表中找不到对应的成员赋值。
- arguments列表中的某个成员被多个实际参数赋值。

当实际参数成功赋值后，在帧对象执行代码对象的过程中，对函数参数的访问都会转化为对arguments列表的访问，因此不再需要形式参数的名称。然而我们在编写函数体时依然需要用形式参数的名称来访问对应的实际参数，它们到arguments列表的变换是隐藏的。因此从效果上讲，形式参数列表可以被视为一个标识符列表，实际参数列表可以被视为一个值列表，而函数调用隐含了一次解包赋值（将在第12章讨论）。

最后，从上面的讨论可知，函数调用最终被转换为了对代码对象的执行。代码对象储存着将源代码进行伪编译得到的字节码。代码对象不可变，也不包含对可变对象的引用（哪怕是间接引用），仅具有下列只读属性：

- co_name：引用函数名。
- co_argcount：引用一个整数，为基于位置对应的形式参数（包括仅位置形式参数和位置或关键字形式参数）的数量。
- co_posonlyargcount：引用一个整数，为仅位置形式参数的数量。
- co_kwonlyargcount：引用一个整数，为仅关键字形式参数的数量。
- co_nlocals：引用一个整数，为本地变量（包括参数）的数量。
- co_varnames：引用一个字符串元组，包含本地变量（包括参数）的名称。
- co_cellvars：引用一个字符串元组，包含被该函数的闭包引用的本地变量的名称。
- co_freevars：引用一个字符串元组，包含自由变量的名称。
- co_code：引用一个二进制对象，包含字节码。
- co_consts：引用一个元组，包含所有字面值（其中第一个元素为文档字符串）。
- co_names：引用一个元组，包含所有标识符。
- co_filename：引用一个字符串，为相应源代码所在文件的文件名。
- co_firstlineno：引用一个整数，为相应函数首行在源文件中的行号。
- co_lnotab：引用一个字符串，编码了从字节码偏移量到源文件行号的映射。
- co_stacksize：引用一个整数，为执行该代码对象所需内存空间的大小。
- co_flags：引用一个整数，编码了一些旗标，至少包括：
  - 0x04：使用了“*name”形式的形式参数。
  - 0x08：使用了“**name”形式的形式参数。
  - 0x20：该函数是一个生成器函数。

➤ co_positions：引用一个函数，调用后会返回一个迭代器，对其迭代可以按顺序取得字节码中的每条指令对应源代码中的哪个位置。

代码对象的co_positions属性是Python 3.11添加的，主要用于支持在异常处理中显示额外的回溯信息。而获得这些回溯信息的代价是略微增大.pyc文件的大小，以及执行Python脚本时使用的内存略微增加。如果你想取消这一功能，可以在启动Python解释器时添加-X no_debug_ranges选项，或者设置PYTHONNODEBUGRANGES环境变量。

4-3. 默认实参值、任意实参列表和解包实参列表

（教程：4.8.1、4.8.3～4.8.5）
（语言参考手册：3.2、8.7、6.3.4）

一次成功的函数调用必须保证实际参数与形式参数一致，但这不意味着实际参数列表和形式参数列表的参数数量必然相同。本节介绍的三种技术都允许两个列表中参数的数量不同，但它们都通过各自的方式保证了实际参数与形式参数一致。

第一种技术被称为“默认实参值（default argument value）”。这一技术很好理解：在将实际参数传递给形式参数的过程中，如果某个被漏掉的形式参数设置了默认实参值，则使用该值而非抛出TypeError异常。设置默认实参值的方式也很简单，即在函数定义语句的形式参数列表中以赋值的形式将默认实参值与相应形式参数关联起来。

但要注意，对于所有基于位置对应的形式参数来说，如果其中某个被设置了默认实参值，那么它后面的所有形式参数都必须被设置默认实参值。而在函数调用中如果某个这样的形式参数没有被传入实际参数，则它后面的所有形式参数也都无法被传入实际参数。仅关键字形式参数不受这一限制。与函数调用类似，PEP 8推荐在设置默认实参值时“=”两端不添加空格。

下面用一个例子说明默认实参值的作用。第3章介绍了逻辑值检测和逻辑运算，我们在这里设计一个能够实现所有这些操作的函数boolean_calculator()，它将这些操作按照如下方式分类：
一元操作：逻辑值检测、not。
二元操作：and、or。
三元操作：... if ... else ...。
而二元操作和三元操作还需要支持选择返回对象还是返回布尔值。该函数的定义如下：

```
#该函数包含如下参数：
# 位置或关键字参数x、y和z为三个操作数，可以为任意对象。   对于一元操作，y和z取默认实
#   参值。   对于二元操作，z取默认实参值。
# 仅关键字参数boolean取布尔值，以表明对于二元操作和三元操作，是否将返回的对象转换为
#   布尔值，默认不转换。
# 仅关键字参数op取如下字符串来指定操作类型：
#   'bool'：一元操作逻辑值检测。
```

```
    #   'not': 一元操作逻辑非。
    #   'and': 二元操作逻辑与。
    #   'or': 二元操作逻辑或。
    #   'switch': 三元操作... if ... else ...。
    def boolean_calculator(
            x, y=NotImplemented,
            z=NotImplemented, *,
            boolean=False, op
            ):

        #处理一元逻辑操作。
        if op in ('bool', 'not'):
            #验证操作数是否为NotImplemented。
            if x is NotImplemented:
                return NotImplemented
            else:
                #逻辑值检测。
                if op == 'bool':
                    return bool(x)
                #逻辑非。
                else:
                    return not x

        #处理二元逻辑操作。
        elif op in ('and', 'or'):
            #验证操作数是否为NotImplemented。
            if x is NotImplemented or y is NotImplemented:
                return NotImplemented
            else:
                #逻辑与。如果boolean取True则返回布尔值。
                if op == 'and':
                    return bool(x and y) if boolean else (x and y)
                #逻辑或。如果boolean取True则返回布尔值。
                else:
                    return bool(x or y) if boolean else (x or y)

        #处理三元逻辑操作。
        elif op == 'switch':
            #验证操作数是否为NotImplemented。
            if (x is NotImplemented
                    or y is NotImplemented
                    or z is NotImplemented):
                return NotImplemented
            else:
                #... if ... else ...
                obj = x if y else z
                #如果boolean取True则返回布尔值。
                if boolean:
                    return bool(obj)
                else:
                    return obj

        #对于不能识别的操作，返回NotImplemented。
        else:
            return NotImplemented
```

请将上述代码保存到boolean_calculator.py中，然后通过下面的命令行和语句验证该函数的调用结果：

```
$ python3 -i boolean_calculator.py

>>> boolean_calculator(1, op="bool")
True
>>> boolean_calculator(1, op="not")
False
>>> boolean_calculator(0j, op="bool")
False
>>> boolean_calculator(0j, op="not")
True
>>> boolean_calculator(1.0, 0.0, op="and")
0.0
>>> boolean_calculator(1.0, 0.0, op="and", boolean=True)
False
>>> boolean_calculator(1.0, 0.0, op="or")
1.0
>>> boolean_calculator(1.0, 0.0, op="or", boolean=True)
True
>>> boolean_calculator("abc", None, "", op="switch")
''
>>> boolean_calculator("abc", None, "", op="switch", boolean=True)
False
>>> boolean_calculator("abc", object(), "", op="switch")
'abc'
>>> boolean_calculator("abc", object(), "", op="switch", boolean=True)
True
>>> boolean_calculator(NotImplemented, op="bool")
Not Implemented
>>> boolean_calculator(NotImplemented, op="not")
Not Implemented
>>> boolean_calculator(True, op="and")
Not Implemented
>>> boolean_calculator(True, op="or")
Not Implemented
>>> boolean_calculator(True, False, op="switch")
Not Implemented
>>> boolean_calculator(15, 64, op="add")
NotImplemented
>>>
```

从该例子可以看出，默认实参值主要有两个作用：

➤ 将某个参数最常见的传入值设置为默认实参值（在本例子中是给boolean传入False），使调用函数时有相当大概率不需要为该形式参数传入实际参数，进而精简函数调用。
➤ 当确定在某个情况下函数的具体执行过程不涉及某个参数时（在本例子中是进行一元操作时的y和z，以及进行二元操作时的z），为该参数设置默认实参值可以使得在该情况下调用函数时不需要为其赋值，进而精简函数调用。

绝大多数情况下我们都希望默认实参值是固定的。代表默认实参值的表达式是在执行函数定义语句时被求值的，得到的对象如果属于基于位置对应的形式参数则被存入函数对象的特殊属性__defaults__引用的元组中，如果属于仅关键字形式参数则被存入函数对象的特殊属性__kwdefaults__引用的元组中。这可以通过如下语句验证：

```
>>> boolean_calculator.__defaults__
(NotImplemented, Not Implemented)
>>> boolean_calculator.__kwdefaults__
{'boolean': False}
>>>
```

因此，如果对该表达式求值得到的对象是不可变的，且其直接或间接引用的对象也都是不可变的，就可以保证默认实参值是固定的；否则可能带来一些微妙的结果（具体请查看官方手册教程4.8.1中“重要警告”部分给出的例子）。

第二种技术被称为“任意实参列表（arbitrary argument list）”。如果所有可以基于位置对应的形式参数中的最后一个为“*name”形式，则意味着它可以接收基于位置对应的实际参数中的所有未被传入者，并将其包装成一个可以通过标识符“name”引用的元组（允许是空元组）。类似的，若所有仅关键字形式参数中的最后一个为“**name”形式，则意味着它可以接收基于关键字对应的实际参数中的所有未被传入者，并将其包装成一个可以通过标识符“name”引用的字典（允许是空字典）。需要注意的是，如果形式参数列表中存在“*name”形式的参数，则该参数后面的所有参数都被视为仅关键字形式参数，不需要额外的“*”分隔符。

下面用一个例子说明任意实参列表的用法。该例子定义了一个函数mixed_calculator()，将arithmetic_calculator()和boolean_calculator()的功能综合在一起：

```
#该函数包含如下参数：
# 位置或关键字参数op取如下字符串来指定操作类型：
#   'add': 加法。
#   'sub': 减法。
#   'mult': 乘法。
#   'div': 除法。
#   'bool': 逻辑值检测。
#   'not': 逻辑非。
#   'and': 逻辑与。
#   'or': 逻辑或。
#   'switch': ... if ... else ...。
# 位置或关键字参数operands接收所有操作数，当进行算术运算时操作数必须为浮点数，当进行
#  逻辑运算时操作数可以为任意对象。
# 仅关键字参数modifiers接收所有修饰符。可能的修饰符包括：
#   floordiv: 取布尔值，以表明是否进行整除，默认不进行。
#   boolean: 取布尔值，以表明对于二元和三元逻辑运算，是否将返回的对象转换为布尔值，
#  默认不转换。
def mixed_calculator(op, *operands, **modifiers):
    #处理算术运算。
    if op in ('add', 'sub', 'mult', 'div'):
        #确保正好有两个操作数。
        if len(operands) == 2:
            #确保两个操作数都是浮点数。
            x = float(operands[0])
            y = float(operands[1])

            #返回算术运算的结果。
            if op == 'add':
```

```
                return x + y
            elif op == 'sub':
                return x - y
            elif op == 'mult':
                return x * y
            else:
                #当给出了修饰符“floordiv=True”时进行整除。
                if 'floordiv' in modifiers and modifiers['floordiv']:
                    return x // y
                else:
                    return x / y
        else:
            return NotImplemented

    #处理一元逻辑操作。
    elif op in ('bool', 'not'):
        #确保正好有一个操作数。
        if len(operands) == 1:
            #验证操作数是否为NotImplemented。
            x = operands[0]
            if x is NotImplemented:
                return NotImplemented
            else:
                #逻辑值检测。
                if op == 'bool':
                    return bool(x)
                #逻辑非。
                else:
                    return not x
        else:
            return NotImplemented

    #处理二元逻辑操作。
    elif op in ('and', 'or'):
        #确保正好有两个操作数。
        if len(operands) == 2:
            #验证操作数是否为NotImplemented。
            x = operands[0]
            y = operands[1]
            if x is NotImplemented or y is NotImplemented:
               return NotImplemented
            else:
                #逻辑与。   如果给出了修饰符“boolean=True”则返回布尔值。
                if op == 'and':
                    if 'boolean' in modifiers and modifiers['boolean']:
                        return bool(x and y)
                    else:
                        return x and y
                #逻辑或。   如果给出了修饰符“boolean=True”则返回布尔值。
                else:
                    if 'boolean' in modifiers and modifiers['boolean']:
                       return bool(x or y)
                    else:
                        return x or y
        else:
            return NotImplemented

    #处理三元逻辑操作。
    elif op == 'switch':
        #确保正好有三个操作数。
        if len(operands) == 3:
            #验证操作数是否为NotImplemented。
            x = operands[0]
            y = operands[1]
            z = operands[2]
```

```
                if (x is NotImplemented
                        or y is NotImplemented
                        or z is NotImplemented):
                    return NotImplemented
                else:
                    #... if ... else ...
                    obj = x if y else z
                    #如果boolean取True则返回布尔值。
                    if 'boolean' in modifiers and modifiers['boolean']:
                        return bool(obj)
                    else:
                        return obj
            else:
                return NotImplemented

        #对于不能识别的操作，返回NotImplemented。
        else:
            return NotImplemented
```

请将上述代码保存到mixed_calculator.py中，然后通过下面的命令行和语句验证该函数的调用结果：

```
$ python3 -i mixed_calculator.py

>>> mixed_calculator('add', 14.0, 3.2)
17.2
>>> mixed_calculator('mult', 14.0, 3.2)
44.800000000000004
>>> mixed_calculator('div', 14.0, 3.2)
4.375
>>> mixed_calculator('div', 14.0, 3.2, floordiv=True)
4.0
>>> mixed_calculator('bool', 1)
True
>>> mixed_calculator('not', 1)
False
>>> mixed_calculator('bool', 0j)
False
>>> mixed_calculator('not', 0j)
True
>>> mixed_calculator('and', 1.0, 0.0)
0.0
>>> mixed_calculator('and', 1.0, 0.0, boolean=True)
False
>>> mixed_calculator('or', 1.0, 0.0)
1.0
>>> mixed_calculator('or', 1.0, 0.0, boolean=True)
True
>>> mixed_calculator('switch', "abc", None, "")
''
>>> mixed_calculator('switch', "abc", None, "", boolean=True)
False
>>> mixed_calculator('switch', "abc", object(), "")
'abc'
>>> mixed_calculator('switch', "abc", object(), "", boolean=True)
True
>>>
```

第三种技术被称为“解包实参列表（unpacking argument list）”。我们可以在实际参数列表中基于位置对应的部分插入任意多个“*expr”形式的实际参数，而对表达式expr求值得到的对象必须是除字符串和二进制序列之外的序列，该序列会替代“*expr”进而被视为多个基于位置的实际参数。类似的，我们可以在实际参数列表中基于关键字对应的部分插入任意多个“**expr”形式的实际参数，而对表达式expr求值得到的对象必须是字典，其中的键值对会被传入仅关键字形式参数。（字典中允许存在不对应任何形式参数的键值对。）

下面用一个例子说明解包实参列表的用法。该例子定义了函数mixed_accumulator()，既可以实现数值的累加又可以实现字符串的拼接，且在拼接字符串时可以指定前缀、后缀和间隔符号：

```
#该函数包含如下参数：
# 位置或关键字参数elements可以接收任意多个操作数，当进行数值累加时它们必须取数值，
#  当进行字符串拼接时它们必须取字符串。
# 仅关键字参数op取如下字符串来指定操作：
#   'sum'：累加。
#   'concat'：拼接。
# 仅关键字参数pre、suf和sep都用于拼接字符串，分别设置前缀、后缀和分隔符。
def mixed_accumulator(
        *elements, op='sum',
        pre='', suf='', sep=''):

    #确保指定的操作合法。
    if op not in ('sum', 'concat'):
        return NotImplemented

    #当操作数个数小于2时直接返回结果。
    length = len(elements)
    if length == 0:
        return None
    if length == 1:
        return elements[0]

    #分别处理累加和拼接。
    if op == 'sum':
        n = 0
        i = 0
        while i < length:
            n = n + float(elements[i])
            i = i + 1
        return n
    else:
        s = pre
        i = 0
        while i < length-1:
            s = s + str(elements[i]) + sep
            i = i + 1
        return s + str(elements[length-1]) + suf
```

该函数定义本身仅使用了默认实参值和任意实参列表，但并没有使用解包实参列表，因为后者是在函数调用时才能使用的。请将上述代码保存到mixed_accumulator.py中，然后通过下面的命令行和语句验证该函数的调用结果：

```
$ python3 -i mixed_accumulator.py

>>> mixed_accumulator(1, 2, 3, 4, 5)
15.0
>>> mixed_accumulator(*(1, 2, 3, 4, 5))
15.0
>>> mixed_accumulator(1, 2, *(3, 4), 5)
15.0
>>> mixed_accumulator(*(1, 2), 3, *(4, 5))
15.0
>>> mixed_accumulator(*(1, 2), *(3, 4, 5))
15.0
>>> mixed_accumulator(*[1, 2, 3, 4, 5])
15.0
>>> mixed_accumulator(*range(1,6))
15.0
>>> mixed_accumulator('a', 'b', 'c', op='concat')
'abc'
>>> mixed_accumulator('a', 'b', 'c', **{'op': 'concat'})
'abc'
>>> mixed_accumulator('a', 'b', 'c', op='concat', sep=' ', pre='English
Letters: ', suf=' ...')
'English Letters: a b c ...'
>>> mixed_accumulator('a', 'b', 'c', **{'op': 'concat'}, **{'sep': ' ',
'pre': 'English Letters: ', 'suf': ' ...'})
'English Letters: a b c ...'
>>> mixed_accumulator(*('a', 'b', 'c'), **{'op': 'concat'}, **{'sep': ' ',
'pre': 'English Letters: ', 'suf': ' ...'})
'English Letters: a b c ...'
>>>
```

4-4. 匿名函数

（教程：4.8.6）
（语言参考手册：6.14、8.7）

至此我们已经学会了用函数定义语句定义函数，以及用函数调用表达式调用函数。但需要注意，执行函数定义语句会创建一个标识符，即函数名。这意味着如果不显式通过del语句删除该标识符，函数对象的引用数就不会变成0，因此会一直存在到脚本执行完成。而在很多情况下，需要临时使用某个函数，使用完之后就将其销毁以节省内存。

当然，结合使用函数定义语句和del语句也可以达到这一目的，但对于那些函数体简单到可以用一个表达式表示的函数，更优雅的解决方案是通过lambda表达式创建“匿名函数（anonymous functions）”：此类函数不具有函数名，需要被赋值给其他标识符才能存在，当该标识符被销毁或引用其他对象时匿名函数也会随之被销毁。

lambda表达式的语法为：

```
lambda [parameter_list]: expr
```

其中parameter_list代表匿名函数的形式参数列表，如果省略则说明调用该匿名函数不需要任何实际参数。expr则是一个表达式，对其求值得到的对象将作为匿名函数的返回值。匿名函数的魔术属性__name__总是引用字符串“<lambda>”，它并不是一个标识符。（因为包含标点“<”和“>”而不符合标识符的命名规则。）换句话说，上述lambda表达式在逻辑上等价于：

```
def <lambda>([parameter_list]):
    expr
```

表2-2说明lambda表达式的优先级很低，仅高于“:=”。对lambda表达式求值得到的是一个没有任何引用的函数对象。下面的例子说明了lambda表达式的基本性质：

```
>>> inc = lambda n: n+1
>>> inc(0)
1
>>> inc(1)
2
>>> inc.__name__
'<lambda>'
>>> dec = lambda n: n-1
>>> dec(0)
-1
>>> dec(1)
0
>>> dec.__name__
'<lambda>'
>>> inc(dec(0))
0
>>> dec(inc(0))
0
>>> inc = None
>>> dec = None
>>>
```

该例子定义了两个简单的匿名函数，并分别通过标识符inc和dec引用。inc()的作用是让数值加1，dec()的作用是让数值减1，两者嵌套在一起正好相互抵消。请注意虽然这两个标识符引用了匿名函数，但不论“inc”还是“dec”都并非函数名。当执行到“inc = None”和“dec = None”时，这两个匿名函数的引用数将变为0，进而进入垃圾回收流程。

匿名函数的形式参数列表与函数定义中的形式参数列表格式完全相同。下面的例子证明匿名函数的形式参数列表也能识别分隔符“/”和“*”（内置函数eval()的作用是将一个字符串当成Python代码来执行）：

```
>>> arithmetic = lambda x, y, /, *, op: eval(str(x) + op + str(y))
>>> arithmetic(1, 2, op='+')
3
>>> arithmetic(1, 2, op='-')
-1
>>> arithmetic(1, 2, op='*')
2
>>> arithmetic(1, 2, op='/')
0.5
>>>
```

下面的例子证明匿名函数支持默认实参值：

```
>>> echo = lambda msg='no message': print(msg)
>>> echo("Hello")
Hello
>>> echo("World")
World
>>> echo()
no message
>>>
```

下面的例子证明匿名函数支持任意实参列表和解包实参列表（内置函数len()的作用是返回一个容器中元素的个数）：

```
>>> total_elements = lambda *ll, **dd: len(ll) + len(dd)
>>> total_elements(0, 1, 2)
3
>>> total_elements(x=1.0, y=2.0)
2
>>> total_elements(0, 1, 2, x=1.0, y=2.0)
5
>>> total_elements(*[0, 1, 2], **{'x': 1.0, 'y': 2.0})
5
>>> total_elements()
0
>>>
```

匿名函数最常见的应用场景是被传递给其他函数的形式参数，然后在后者的函数体内被调用。这种被当作实际参数使用的函数又称为“回调函数（callbacks）”，很多时候仅在它们被传入的函数的函数体内才有意义。最好保证回调函数在函数体外不能被访问（这涉及标识符的作用域，将在第6章详细讨论），因此需要使用匿名函数。

下面给出一个例子：

```
#该函数以冒泡排序算法对列表进行排序，其参数compare需被传入一个返回布尔值的函数，
# 而该函数自身的返回值是排好序的列表。
def bubble_sort(lt, compare=lambda a, b: a < b):
    #如果传入列表的成员数少于2，直接返回该列表。
    lt_len = len(lt)
    if lt_len < 2:
        return lt

    #创建一个新列表，然后依次取出原列表的成员，与新列表的已有成员基于回调函数进行
    # 比较，在回调函数第一次返回True的索引处插入该成员。
    nlt = [lt[0]]
    i = 1
    while i < lt_len:
        nlt_len = len(nlt)
        j = 0
        while j < nlt_len:
            if compare(lt[i], nlt[j]):
                nlt.insert(j, lt[i])
                break
            j = j + 1
        else:
            nlt.append(lt[i])
        i = i + 1

    return nlt
```

该例子定义了一个普适的排序函数bubble_sort()，通过lt参数接收一个成员可以是任意类型的列表，并返回一个将该列表中的成员按升序排列的列表。排序过程中需要对列表中的成员做比较，而这是通过调用由compare参数传入的回调函数实现的，该回调函数的形式参数必须接收两个任意类型的对象，返回值必须是布尔值。

在上面的函数定义中，compare参数的默认实参值是一个匿名函数，以它作为回调函数将遵循Python中数值和字符串的默认比较规则。在调用bubble_sort()时，还可以给compare参数传入别的函数以按照其他规则进行比较。请将上面的代码保存为bubble_sort.py，然后通过下面的命令行和语句验证匿名函数作为实际参数的用法（abs()是取绝对值的内置函数，将在第9章讨论）：

```
$ python3 -i bubble_sort.py

>>> bubble_sort([5, 6, -1, -9, 7])
[-9, -1, 5, 6, 7]
>>> bubble_sort([5, 6, -1, -9, 7], lambda x, y: abs(x) < abs(y))
[-1, 5, 6, 7, -9]
>>> bubble_sort(['cdef', 'bc', 'abe', 'z'])
['abe', 'bc', 'cdef', 'z']
>>> bubble_sort(['cdef', 'bc', 'abe', 'z'], lambda x, y: len(x) < len(y))
['z', 'bc', 'abe', 'cdef']
>>>
```

注意给compare参数传入一个通过函数定义语句定义的函数bubble_sort()也能正常工作，但如果像上面那样传入一个匿名函数，则该函数仅会在bubble_sort()被执行期间存在。

匿名函数另一个应用场景是作为其他函数的返回值。同样，可以返回一个通过函数定义语句定义的函数，但返回一个匿名函数将使代码更易读。

下面的例子将mixed_calculator()函数改造为calculator_factory()函数，使其返回一个执行指定操作的函数：

```
#该函数包含如下参数：
# 位置或关键字参数op取如下字符串来指定操作类型：
#   'add': 加法。
#   'sub': 减法。
#   'mult': 乘法。
#   'div': 除法。
#   'bool': 逻辑值检测。
#   'not': 逻辑非。
#   'and': 逻辑与。
#   'or': 逻辑或。
#   'switch': ... if ... else ...。
# 仅关键字参数modifiers接收所有修饰符。可能的修饰符包括：
#   floordiv: 取布尔值，以表明是否进行整除，默认不进行。
#   boolean: 取布尔值，以表明对于二元和三元逻辑运算，是否将返回的对象转换为布尔值，
#  默认不转换。
def calculator_factory(op, **modifiers):
    if op == 'add':
        return lambda x, y: float(x) + float(y)
    elif op == 'sub':
        return lambda x, y: float(x) - float(y)
    elif op == 'mult':
        return lambda x, y: float(x) * float(y)
    elif op == 'div':
        if 'floordiv' in modifiers and modifiers['floordiv']:
            return lambda x, y: float(x) // float(y)
        else:
            return lambda x, y: float(x) / float(y)
    elif op == 'bool':
        return lambda x: bool(x)
    elif op == 'not':
        return lambda x: not x
    elif op == 'and':
        if 'boolean' in modifiers and modifiers['boolean']:
            return lambda x, y: bool(x and y)
        else:
            return lambda x, y: x and y
    elif op == 'or':
        if 'boolean' in modifiers and modifiers['boolean']:
            return lambda x, y: bool(x or y)
        else:
            return lambda x, y: x or y
    elif op == 'switch':
        if 'boolean' in modifiers and modifiers['boolean']:
            return lambda x, y, z: bool(x if y else z)
        else:
            return lambda x, y, z: x if y else z
    else:
        return NotImplemented
```

请将上面的代码保存为calculator_factory.py，然后通过下面的命令行和语句验证匿名函数作为返回值的用法：

```
$ python3 -i calculator_factory.py

>>> f = calculator_factory('add')
>>> f(8, 9)
17.0
>>> f = calculator_factory('sub')
>>> f(8, 9)
-1.0
>>> f = calculator_factory('mult')
>>> f(8, 9)
72.0
>>> f = calculator_factory('div')
>>> f(8, 9)
0.8888888888888888
>>> f = calculator_factory('div', floordiv=True)
>>> f(8, 9)
0.0
>>> f = calculator_factory('bool')
>>> f('')
False
>>> f = calculator_factory('not')
>>> f('abc')
False
>>> f = calculator_factory('and')
>>> f('', 'abc')
''
>>> f = calculator_factory('and', boolean=True)
>>> f('', 'abc')
False
>>> f = calculator_factory('or')
>>> f('', 'abc')
'abc'
>>> f = calculator_factory('or', boolean=True)
>>> f('', 'abc')
True
>>> f = calculator_factory('switch')
>>> f('', object(), 'abc')
''
>>> f('', None, 'abc')
'abc'
>>> f = calculator_factory('switch', boolean=True)
>>> f('', object(), 'abc')
False
>>> f('', None, 'abc')
True
>>>
```

注意在这个例子中，如果calculator_factory()不返回匿名函数，而返回通过函数定义语句定义的函数，那么虽然语法上不会有错，但代码将变得冗长。

4-5. 函数装饰器

（语言参考手册：8.7）
（标准库：functools）

接下来我们将介绍Python最具特色的语法特性之一——“装饰器（decorators）”。其实装饰器能够装饰任意类型的可调用对象，但本章仅讨论它们对函数的影响。（第5章会将装饰器的概念扩展到其他类型的可调用对象。）

首先需要强调，装饰器并不是Python编程中必不可少的要素，而只是一种语法糖。

换句话说在用Python编程时完全可以不使用装饰器。但使用装饰器能让Python代码更简洁，且更有“Python范儿（pythonic）”。本节先从装饰器的本质讲起，然后给出创建和使用函数装饰器的标准方法。

概括地说，装饰器的作用是对一个可调用对象进行包装，进而给该对象添加额外的功能。装饰器自身也是一个可调用对象，仅具有一个用于传入被装饰对象的参数，返回值则是一个“包装器（wrappers）”。可以将被装饰对象称为“目标对象”，而由于本章只考虑装饰函数的情况，因此也称其为“目标函数”。

下面举一个模拟装饰器的例子：

```
#定义被用作装饰器的函数。
def mydecorator(wrapped):
    def wrapper(*args, **kwargs):
        print("I can do anything before calling the target!")
        result = wrapped(*args, **kwargs)
        print("I can do anything after calling the target!")
        return result
    return wrapper


#定义目标函数1。
def mytarget1():
    print("I am doing nothing.")
    return 0


#定义目标函数2。
def mytarget2(obj):
    print("I am doing the truth value testing.")
    return bool(obj)


#定义目标函数3。
def mytarget3(x, y):
    print("I am an adder.")
    return float(x) + float(y)


#定义目标函数4。
def mytarget4(*substr):
    print("I am an concatenator.")
    length = len(substr)
    if length == 0:
```

```
            return ''
        elif length == 1:
            return substr[0]
        else:
            i = 0
            s = ''
            while i < length:
                s = s + substr[i]
                i = i + 1
            return s


    #对目标函数1进行装饰。
    mytarget1 = mydecorator(mytarget1)

    #对目标函数2进行装饰。
    mytarget2 = mydecorator(mytarget2)

    #对目标函数3进行装饰。
    mytarget3 = mydecorator(mytarget3)

    #对目标函数4进行装饰。
    mytarget4 = mydecorator(mytarget4)
```

请将上面的代码保存为function_decorator1.py，然后通过下面的命令行和语句验证被装饰后的目标函数的行为：

```
$ python3 -i function_decorator1.py

>>> mytarget1()
I can do anything before calling the target!
I am doing nothing.
I can do anything after calling the target!
0
>>> mytarget2(None)。
I can do anything before calling the target!
I am doing the truth value testing.
I can do anything after calling the target!
False
>>> mytarget3(3, 4.5)
I can do anything before calling the target!
I am an adder.
I can do anything after calling the target!
7.5
>>> mytarget4('a', 'b', 'c', 'd')
I can do anything before calling the target!
I am an concatenator.
I can do anything after calling the target!
'abcd'
>>>
```

上面这个例子已经说明了一些自定义装饰器的技巧：

➤ 装饰器返回的包装器必须能够与目标函数接收相同的参数。

显然，最简单的解决方案是让包装器直接拷贝目标函数的形式参数列表，然后在调用目标函数时直接将形式参数列表改造为等价实际参数列表。

然而本例子中的装饰器为了能够装饰尽可能多的函数将包装器的形式参数列表设置为了“*args, **kwargs”——args用于接收所有基于位置对应的实际参数，kwargs用于接收所有基于关键字对应的实际参数。在调用目标函数时则将实际参数列表也设置为该形式，*args展开后为所有基于位置对应的实际参数，而**kwargs展开后为所有基于关键字对应的实际参数。

这一技巧结合使用了任意实参列表技术和解包实参列表技术，使目标函数可以具有任意形式的形式参数列表。

➤ 包装器的函数体内，在调用目标函数之前和之后都可以执行操作，而这些操作就是装饰器为目标函数添加的额外功能。

包装器的返回值必须是目标函数的返回值，但不一定要在目标函数返回的时候返回——可以用一个标识符先引用该返回值，然后再在合适的位置返回该标识符。

上面的例子揭示了装饰器的本质，但并没有用到Python专门为装饰器提供的语法，因此只是对装饰器的模拟。下面介绍标准的装饰器语法。

事实上，在定义了用作装饰器的可调用对象后，可以通过如下语法在定义目标函数的同时完成对该目标函数的装饰：

```
@decorator
def target(...):
      ...
```

这其实等价于：

```
def target(...):
      ...
target = decorator(target)
```

但标准的装饰器语法除了更简洁之外，在对一个目标函数进行多重装饰时将具有更好的可读性，例如：

```
@decorator1
@decorator2
...
@decoratorN
def target(...):
      ...
```

等价于

```
def target(...):
      ...
target = decoratorN(target)
...
target = decorator2(target)
target = decorator1(target)
```

显然，每进行一层装饰，目标函数就将获得一层额外的功能，但在调用目标函数时函数栈的长度就会增加一层，会对物理内存造成额外压力。所以除非确实有必要，否则不要随便给函数添加装饰器。

下面的例子用标准的装饰器语法对上面的例子进行了改写：

```
def mydecorator(wrapped):
    def wrapper(*args, **kwargs):
        print("I can do anything before calling the target!")
        result = wrapped(*args, **kwargs)
        print("I can do anything after calling the target!")
        return result
    return wrapper


#定义目标函数1，同时完成装饰。
@mydecorator
def mytarget1():
    print("I am doing nothing.")
    return 0


#定义目标函数2，同时完成装饰。
@mydecorator
def mytarget2(obj):
    print("I am doing the truth value testing.")
    return bool(obj)


#定义目标函数3，同时完成装饰。
@mydecorator
def mytarget3(x, y):
    print("I am an adder.")
    return float(x) + float(y)


#定义目标函数4，同时完成装饰。
```

```
@mydecorator
def mytarget4(*substr):
    print("I am an concatenator.")
    length = len(substr)
    if length == 0:
        return ''
    elif length == 1:
        return substr[0]
    else:
        i = 0
        s = ''
        while i < length:
            s = s + substr[i]
            i = i + 1
        return s
```

请将上面的代码保存为function_decorator2.py，然后用针对function_decorator1.py的语句验证，你会发现它们的效果完全相同。

但不论是function_decorator1.py中的代码，还是function_decorator2.py中的代码，都存在两个缺陷：目标函数被装饰之后，其实变成了装饰器返回的包装器，因此特殊属性__name__、__doc__、__annotations__、__module__和__qualname__引用的对象也将属于包装器的而非目标函数；目标函数在函数体内定义的标识符也无法在包装器的函数体内被访问。

请在执行完function_decorator1.py或function_decorator2.py并进入交互模式后通过如下语句验证这一事实：

```
>>> mytarget1.__name__
'wrapper'
>>> mytarget2.__name__
'wrapper'
>>> mytarget3.__name__
'wrapper'
>>> mytarget4.__name__
'wrapper'
>>>
```

弥补这两个缺陷的办法是将返回的包装器的上述特殊属性设置为与目标函数的相同，并更新包装器的变量字典以使它包括目标函数在函数体内定义的标识符。

然而在定义装饰器时手工实现这些设置是比较困难的，所以标准库中的functools模块为我们提供了完成这些设置的便捷工具。简单来说，我们只需要将function_decorator2.py中的装饰器函数定义修改为：

```
from functools import wraps


def mydecorator(wrapped):
    @wraps(wrapped)
    def wrapper(*args, **kwargs):
        print("I can do anything before calling the target!")
        result = wrapped(*args, **kwargs)
        print("I can do anything after calling the target!")
        return result
    return wrapper
```

这就是定义装饰器的标准方法。

请将修改后的代码保存为function_decorator3.py，然后通过下面的命令行和语句验证被装饰后的函数的特殊属性依然保持不变：

```
$ python3 -i function_decorator3.py

>>> mytarget1.__name__
'mytarget1'
>>> mytarget2.__name__
'mytarget2'
>>> mytarget3.__name__
'mytarget3'
>>> mytarget4.__name__
'mytarget4'
>>>
```

比较function_decorator3.py和function_decorator2.py可知，弥补缺陷的办法是在装饰器内部定义包装器的同时用functools.wraps()的返回值装饰它。“@wraps(wrapped)”语句不能被解读为以wraps()作为装饰器，因为装饰器语法“@xxx”中的“xxx”代表的是对一个可调用对象的引用，可以是标识符（例如“@mydecorate”），也可以是函数调用的返回值（例如“@wraps(wrapped)”）。

为了理解“@wraps(wrapped)”语句返回了一个什么样的可调用对象，下面将介绍functools定义的三个相互关联的函数：update_wrapper()、partial()和wraps()。

update_wrapper()的功能是完成包装器的特殊属性设置，并更新变量字典。该函数的语法为：

```
    functools.update_wrapper(wrapper, wrapped,
assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)
```

其中wrapper参数被传入包装器；wrapped参数被传入目标函数；assigned参数使我们可以指定拷贝目标函数的哪些特殊属性，但除非你是Python专家否则让其取默认值即可；updated参数则使我们可以控制是否更新包装器的变量字典，但同样除非你是Python专家否则应让其取默认值。该函数还会额外给包装器添加一个__wrapped__属性，并使该属性引用目标函数。

从逻辑上讲，update_wrapper()应该被作为包装器的装饰器来使用，但它有四个参数（其中有两个没有默认实参值），不符合装饰器的要求。partial()的功能即减少调用一个函数时必需给出的实际参数的数量，其语法为：

```
    functools.partial(func, /, *args, **keywords)
```

其中func参数被传入需减少参数的函数；args参数用于接收被减少的基于位置对应的参数；keywords参数用于接收被减少的基于关键字对应的参数。该函数会返回一个可调用的“partial对象”，该对象的调用语法为：

```
    partial_name(*fargs, **fkeywords)
```

而调用partial对象等价于如下调用：

```
    func(*args, *fargs, **keywords, **fkeywords)
```

wraps()的作用其实就是在函数体内通过partial()将update_wrapper()转变为只有1个参数的partial对象，而该partial对象符合装饰器的要求，可被作为装饰器使用。wraps()的语法为（注意“@”表明一个函数本身或其返回值通常被作为装饰器使用）：

```
    @functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS,
updated=WRAPPER_UPDATES)
```

该函数有三个参数，但除非你是Python专家否则应让assigned参数和updated参数取默认值；而wrapped参数应传入被装饰的目标函数。按上述语法调用wraps()其实等价于调用partial(update_wrapper, wrapped, assigned, updated)，也就是说wraps()的三个参数将分别传递给update_wrapper()的后三个同名参数，使得它只需要被传入wrapper参数就可以被执行。

至此我们已经介绍完了装饰器的核心知识。下面介绍一个更高级的技巧。按照上述标准方法定义的装饰器还有一点不足：包装器的形式参数列表必须与目标函数的形式参数列表相同，因此无法将包装器本身参数化，以使得它的行为可以通过参数控制。解决这一问题的办法是不直接定义装饰器本身，而是定义生成装饰器的函数（functools.wraps()就是一个这样的函数）。

下面先给出一个例子：

```
      from functools import wraps


      #定义一个生成装饰器的函数。
      def mydecorator_generator(
              before_action=lambda *args, **kwargs: print("I can do anything
before calling the target!"),
              after_action=lambda *args, **kwargs: print("I can do anything after
calling the target!")):
          def mydecorator(wrapped):
              @wraps(wrapped)
              def wrapper(*args, **kwargs):
                  before_action(*args, **kwargs)
                  result = wrapped(*args, **kwargs)
                  after_action(*args, **kwargs)
                  return result
              return wrapper
          return mydecorator


      #定义目标函数1，同时完成装饰。
      @mydecorator_generator()
      def mytarget1():
          print("I am doing nothing.")
          return 0


      #定义目标函数2，同时完成装饰。
      @mydecorator_generator(
              lambda *args, **kwargs: None,
              lambda *args, **kwargs: None)
      def mytarget2(obj):
          print("I am doing the truth value testing.")
          return bool(obj)


      #定义目标函数3，同时完成装饰。
      @mydecorator_generator(
              lambda *args, **kwargs: None,
              lambda *args, **kwargs: print("The expression calculated is: " +
str(float(args[0])) + "+" + str(float(args[1]))))
      def mytarget3(x, y):
          print("I am an adder.")
          return float(x) + float(y)


      #定义目标函数4，同时完成装饰。
      @mydecorator_generator(
```

```
            lambda *args, **kwargs: print("Having received " + str(len(args)) +
" strings!"),
            lambda *args, **kwargs: None)
    def mytarget4(*substr):
        print("I am an concatenator.")
        length = len(substr)
        if length == 0:
            return ''
        elif length == 1:
            return substr[0]
        else:
            i = 0
            s = ''
            while i < length:
                s = s + substr[i]
                i = i + 1
            return s
```

请将上面的代码保存为function_decorator4.py，然后通过下面的命令行和语句验证被装饰后的目标函数的行为：

```
    $ python3 -i function_decorator4.py

    >>> mytarget1()
    I can do anything before calling the target!
    I am doing nothing.
    I can do anything after calling the target!
    0
    >>> mytarget2(None)
    I am doing the truth value testing.
    False
    >>> mytarget3(3, 4.5)
    I am an adder.
    The expression calculated is: 3.0+4.5
    7.5
    >>> mytarget4('a', 'b', 'c', 'd')
    Having received 4 strings!
    I am an concatenator.
    'abcd'
    >>>
```

现在让我们分析上面的例子。生成装饰器的函数本身具有两个形式参数：

before_action：接收一个函数，使其在目标函数被调用前调用。

after_action：接收一个函数，使其在目标函数被调用后调用。

通过给这两个参数传入不同的函数，该函数返回的装饰器的行为可以千变万化。但返回的装饰器本身只有一个用于传入目标函数的参数，其内的包装器之所以能够访问外层函数的参数before_action和after_action，是因为它是在装饰器内部定义的，而装饰器又是在外层函数内部定义的，因此包装器是装饰器的闭包，而装饰器又是生成装饰器的函数的闭包。（闭包将在第6章讨论。）

由于传入before_action和after_action的函数的形式参数列表都是“*args, **kwargs”，所以能够接收任何实际参数，所以上面例子中的生成装饰器的函数已经具有相当大的通用性。事实上，生成装饰器的函数的形式参数可以任意设置，特别灵活，设计巧妙后能够以极短的代码实现极丰富的功能。

4-6. 为函数增加缓存功能

（标准库：functools）

functools模块除了提供定义装饰器的辅助函数之外，还预定义了一些很有用的装饰器，本节介绍其中的两个——cache()和lru_cache()。它们都为目标函数提供了缓存，该缓存是一个字典，键为实际参数列表，值为相应返回值。

当用它们装饰的目标函数被以某实际参数列表调用时，Python解释器将优先在它的缓存中查找匹配该实际参数列表的键值对：如果有则直接取得返回值，不需要执行目标函数；否则执行目标函数，得到返回值，同时将该实际参数列表到该返回值的映射添加到缓存中，以使得下次以相同实际参数列表调用目标函数时不再需要执行目标函数。

这样可以有效提高被频繁调用且实际参数列表有很大重叠概率的函数的执行效率。但要注意，实用这两个装饰器的前提是目标函数的实际参数列表中包含的对象都是可哈希的。（“可哈希”这一概念将在第11章详细讨论。）

---

cache()的语法为：

```
@functools.cache(user_function)
```

它的用法很简单，直接被当成装饰器使用即可。下面以计算斐波拉契序列为例说明cache()的作用（该例子用到了标准库的time模块中的time()函数，其功能是返回当前时间相对于1970年1月1日0时0分0秒的秒数）：

```
import time
import functools


#定义一个无缓存功能的斐波拉契序列生成函数。
def fibonacci(n):
    if n < 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)


#定义一个有缓存功能的斐波拉契序列生成函数。
@functools.cache
def fibonacci_cache(n):
    if n < 2:
        return 1
    else:
        return fibonacci_cache(n-1) + fibonacci_cache(n-2)


#定义一个统计函数执行时间的函数。
def process_time(func, *args, **kwargs):
```

```
        starttime = time.time()
        func(*args, **kwargs)
        endtime = time.time()
        return endtime - starttime
```

请将上面的代码保存为cache.py，然后通过下面的命令行和语句验证缓存对递归函数执行速度的提升（注意time()在Windows上精度不够，fibonacci_cache()的耗时会显示为0.0）：

```
$ python3 -i cache.py

>>> process_time(fibonacci, 40)
40.984959840774536
>>> process_time(fibonacci_cache, 40)
6.604194641113281e-05
>>> process_time(fibonacci, 40)
42.17100524902344
>>> process_time(fibonacci_cache, 40)
3.814697265625e-06
>>>
```

上述结果是在Macbook Pro上得到的：计算斐波拉契序列的第41个元素，不使用缓存需要约40秒；使用缓存第一次需要约0.07毫秒，第二次需要约0.004毫秒。这个巨大差异的背后逻辑是这样的：当计算斐波拉契序列的第41个元素时，不使用缓存的fibonacci()函数会被递归调用上亿次；而使用缓存的fibonacci_cache()函数第一次被调用时仅被递归调用41次，此时缓存中将有fibonacci_cache(0)、fibonacci_cache(1)、……、fibonacci_cache(40)的返回值，后续的递归调用全部被跳过，而再次调用fibonacci_cache(40)也变成直接从缓存中取得返回值。

cache()有一个缺点，即对缓存的大小没有限制，当目标函数的实际参数列表有很多种组合时将占用大量物理内存，甚至导致Python解释器崩溃。因此除非确定目标函数被调用的方式有限，否则强烈建议不要使用cache()，而应使用更加安全的lru_cache()。

---

lru_cache()与cache()的不同之处在于它会限制缓存的大小，并以“最近最少使用（Least Recently Used，LRU）”算法管理缓存内容。lru_cache()的语法为：

```
@functools.lru_cache(user_function)
@functools.lru_cache(maxsize=128, typed=False)
```

其中第一种语法直接将lru_cache()作为装饰器，限制缓存最多储存128个实际参数列表到返回值的映射，而那些类型不完全相同但可以通过“==”测试的对象（例如0、0.0和0j）将被视为相同的实际参数。第二种语法是将lru_cache()当成生成装饰器的函数使用，目的是让我们对缓存进行更多控制：maxsize参数用于指定缓存中最多存放多少个实际参数列表到返回值的映射（传入None表示无限制），而typed参数用于指定是否将类型不完全相同但可以通过“==”测试的对象视为相同的实际参数（默认是）。

当通过lru_cache()将目标函数修饰之后，得到的包装器将额外具有如下三个函数属性：

- cache_parameters()：返回一个字典，以显示缓存的maxsize和typed设置。
- cache_info()：返回一个命名元组以显示缓存的当前状态。
- cache_clear()：清空该缓存。

下面同样以计算斐波拉契序列为例子说明lru_cache()函数的用法：

```
import functools
import time


#定义一个能缓存128个映射的斐波拉契序列生成函数。
@functools.lru_cache
def fibonacci_lru(n):
    if n < 2:
        return 1
    else:
        return fibonacci_lru(n-1) + fibonacci_lru(n-2)


#定义一个能缓存10个映射的斐波拉契序列生成函数。
@functools.lru_cache(10)
def fibonacci_10(n):
    if n < 2:
        return 1
    else:
        return fibonacci_10(n-1) + fibonacci_10(n-2)


#定义一个统计函数执行时间的函数。
def process_time(func, *args, **kwargs):
    starttime = time.time()
    func(*args, **kwargs)
    endtime = time.time()
    return endtime - starttime
```

请将上面的代码保存为lru_cache.py，然后通过下面的命令行和语句验证lru_cache()的使用方法：

```
$ python3 -i lru_cache.py

>>> fibonacci_lru.cache_parameters()
{'maxsize': 128, 'typed': False}
>>> fibonacci_lru.cache_info()
CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)
>>> process_time(fibonacci_lru, 100)
0.0001990795135498047
>>> fibonacci_lru.cache_info()
CacheInfo(hits=98, misses=101, maxsize=128, currsize=101)
>>> process_time(fibonacci_lru, 50)
2.86102294921875e-06
>>> fibonacci_lru.cache_info()
```

```
CacheInfo(hits=99, misses=101, maxsize=128, currsize=101)
>>> fibonacci_lru.cache_clear()
>>> fibonacci_lru.cache_info()
CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)
>>> process_time(fibonacci_lru, 50)
5.5789947509765625e-05
>>> fibonacci_lru.cache_info()
CacheInfo(hits=48, misses=51, maxsize=128, currsize=51)
>>>
>>> fibonacci_10.cache_parameters()
{'maxsize': 10, 'typed': False}
>>> fibonacci_10.cache_info()
CacheInfo(hits=0, misses=0, maxsize=10, currsize=0)
>>> process_time(fibonacci_10, 100)
0.00011873245239257812
>>> fibonacci_10.cache_info()
CacheInfo(hits=98, misses=101, maxsize=10, currsize=10)
>>> process_time(fibonacci_10, 50)
5.7220458984375e-05
>>> fibonacci_10.cache_info()
CacheInfo(hits=146, misses=152, maxsize=10, currsize=10)
>>> fibonacci_10.cache_clear()
>>> process_time(fibonacci_10, 50)
5.3882598876953125e-05
>>> fibonacci_10.cache_info()
CacheInfo(hits=48, misses=51, maxsize=10, currsize=10)
>>>
```

上述结果说明了目标函数被lru_cache()装饰后获得的cache_parameters()、cache_info()和cache_clear()的作用，尤其是cache_info()使我们可以推导出某次调用导致目标函数被递归调用了多少次，缓存中储存的映射又命中了多少次。此外，由于fibonacci_10()只能缓存最后10个返回值，在例子中是fibonacci_10(91)、……、fibonacci_10(100)，所以两次调用fibonacci_10(50)的执行速度其实是一样的。

综上所述，装饰器cache()和lru_cache()的本质都是以空间换时间，特别是当maxsize参数传入None、typed参数传入False时lru_cache()就等价于cache()。这两个装饰器在大部分情况下都被应用于递归函数，因为只有这样才能保证缓存的映射有意义。