[笔记][Learning Python][4. 函数和生成器]

Python

16. 函数基础

函数就是将一些语句集合在一起的组件，从而让它们能够不止一次地在程序中运行。
函数是 `Python` 为了达到代码重用最大化而提供的最基本的程序结构，它使我们能够更宏观地把握程序设计。

我们要学习：

- 函数调用语句
- 两种声明函数的方式 `def` 和 `lambda`
- 两种管理作用域的方式 `global` 和 `nonlocal`
- 两种传回返回值的方式 `return` 和 `yield`

Table 16-1. Function-related statements and expressions

Statement or expression	Examples
Call expressions	`myfunc('spam', 'eggs', meat=ham, *rest)`
def	`def printer(messge):` `  print('Hello ' + message)`
return	`def adder(a, b=1, *c):` `  return a + b + c[0]`

global	`x = 'old'`
	`def changer():`
	`  global x; x = 'new'`

nonlocal (3.X)	`def outer():`
	`  x = 'old'`
	`  def changer():`
	`  nonlocal x; x = 'new'`

yield	`def squares(x):` `  for i in range(x): yield i ** 2`
lambda	`funcs = [lambda x: x**2, lambda x: x**3]`

16.1 为何使用函数

函数有时候也被称为子程序或过程。
函数主要扮演了两个角色：

- 最大化代码重用和最小化代码冗余
- 过程的分解

一般来说，函数关注的是过程：关于如何做某事，而不是对哪些对象做这件事。

16.2 编写函数

- `def` 是可执行的代码。在 `Python` 运行 `def` 之前，函数并不存在。
  在 `if` 语句、 `while` 循环甚至其他的 `def` 中嵌套 `def` 语句是合法的。
- `def` 创建了一个对象并将其赋值给某一变量名。当 `Python` 运行到 `def` 语句时，将生成一个新的函数对象并将其赋值给这个函数名。和所有的赋值一样，函数名变成了一个函数对象的引用。
- `lambda` 创建一个对象并将其作为结果返回。
- `return` 将一个结果对象传回给调用者。
- `yield` 向调用者发回一个结果对象，但是会记住它离开的位置。
- `global` 声明了一个模块级的可被赋值的变量。
- `nonlocal` 声明了一个需要被赋值的外层函数变量。
  `nonlocal` 允许一个函数对一个在其外层的 `def` 语句作用域中已有的名称进行赋值。这就把外层函数用作了保留状态的 “仓库”。也就是说同一函数在不同次的调用之间可以不必借助全局变量而存储信息。
- 参数是通过赋值（对象引用）传递给函数的。
- 除非你显式指明形式参数与实际参数的对应，否则实际参数按位置赋值给形式参数。
- 参数、返回值与变量不需要被声明。

`def` 语句
`def` 语句会创建一个函数对象并将其赋值给一个变量名。
函数体：每次调用函数时 `Python` 所执行的语句。

`def` 语句执行于运行时。
由于是语句，所以可以嵌套在其他语句里面。

```
if test:
    def func(): # Define func this way
    ...
else:
    def func(): # Or else this way
    ...
...
func() # Call the version selected and built
```

通常，`def` 在运行到的时候才进行计算，而在 `def` 之中的代码也只在函数被调用后才会计算。

函数就是对象，还允许附加属性在其上。

```
def func(): ...      # Create function object
func()               # Call object
func.attr = value    # Attach attributes
```

16.3 第一个示例：定义和调用

函数 `times` 的作用取决于传递给它的值。

```
>>> def times(x, y):    # Create and assign function
...     return x * y     # Body executed when called
...
>>> x = times(3.14, 4)  # Save the result object
>>> x
12.56

>>> times('Ni', 4)      # Functions are "typeless"
'NiNiNiNi'
```

`*` 作为一个分派机制，将执行的控制权移交给被处理的对象。

> Really, * is just a dispatch mechanism that routes control to the objects being processed.

多态：一个操作的意义取决于备操作对象的类型。

> `the meaning of an operation depends on the objects being operated upon.`

比如函数可以自动地应用到所有类别的对象上，只要对象支持所预期的接口（也成为协议），函数就能处理它们。
也就是说，如果传给函数的对象支持预期的方法和表达式运算符，那么它们对函数的逻辑来说就是有着即插即用的兼容性。

在 `Python` 中，你的代码不应当关心特定的数据类型。
从宏观上来说，我们在 `Python` 中为对象接口编程，而不是数据类型。
这种多态性也被称为鸭子类型：你的代码不必在意一个对象是不是一只鸭子，只需要关心它能像鸭子那样叫。不管是不是鸭子，只要能鸭子叫就行，同时鸭子叫的实现留给对象自己来完成。

16.4 第二个示例：寻找序列的交集

```
def intersect(seq1, seq2):
    res = [] # Start empty
    for x in seq1: # Scan seq1
        if x in seq2: # Common item?
            res.append(x) # Add to end
    return res
```

单行列表推导式：

```
>>> [x for x in s1 if x in s2]
```

`intersect` 函数是多态的。
第一个参数支持 `for` 循环，第二个参数支持 `in` 成员测试即可。
如果传入了不支持这些接口的对象，`Python` 会自动检测出不匹配，并抛出一个异常。
这正是我们想要的，通过不编写类型测试，并且让 `Python` 来检测不匹配，我们既减少了自己手动编写的代码量，又增强了代码的灵活性。

注意如果第二个参数是单遍迭代器的话（比如文件对象），需要保证在读到末尾的时候回到开头。
或者需要通过提供 `__contains__`、要么提供 `__iter__` 或 `__getitem__` 方法来实现 `in` 运算符。

> `in` 运算符可以通过 `__iter__` 和 `__getitem__` 实现吗？

局部变量：仅在 `def` 内的函数中可见，并且仅在函数运行时存在。
所有在函数内部进行赋值的变量名都默认为局部变量。

16.5 本章小结

在 `Python` 中赋值意味着对象引用，内部真实含义就是指针。

16.6 本章习题

5. “检查传入函数的对象类型” 的做法有什么不妥？
   检查传入函数的对象类型，实质上破坏了函数的灵活性，并把函数限制在特定的类型上。没有这类检查时，函数可以处理更大范围的对象类型：任何支持函数所预期的接口的对象都将适用。
   接口一词是指函数所执行的一组方法和表达式运算符。

17. 作用域

`Python` 作用域：即变量定义和查找的地方。
如模块文件一样，作用域能阻止程序代码中的名称冲突；在一个程序单元中定义的名称不会干扰另一个单元内的名称。

代码中给名称赋值的这个位置，对于确定这个名称的含义很关键。
作用域提供了一种在函数调用之间保留状态的方式，在某些情况下也能成为类的替代品。

17.1 Python 作用域基础

命名空间：变量存在的地方。
`Python` 创建、改变或查找变量名都是在所谓的命名空间中进行的。
我们讨论在代码中查找变量名时，作用域 `scope` 这个术语指的就是命名空间。
也就是说，在源代码变量名被赋值的位置决定了这个变量名能被访问到的范围。
`Python` 中一切与变量名有关的事件（包括作用域的分类），都发生在赋值的时候。
由于变量没有最初声明，`Python` 将一个变量名被赋值的地点关联为（绑定给 `bind it to`）一个特定的命名空间。
换句话说，在代码中给一个变量赋值的地方决定了这个变量将存在于哪个命名空间，也就是它的可见范围。

函数还为程序增加了一个额外的命名空间层来最小化相同变量名之间的潜在冲突：在默认的情况下，一个函数内赋值的所有变量名斗鱼该函数的命名空间相关联。

- 在 `def` 内赋值的变量名只能够被 `def` 内的代码使用。你不能在函数的外部引用该变量名。
- 在 `def` 内赋值的变量名与在 `def` 外赋值的变量名并不冲突，即使是相同的变量名。

在任何情况下，一个变量的作用域（它可以被使用的地方）总是由它在代码中被赋值的位置决定的，而与函数调用完全无关。

变量可以在 `3` 个不同的地方被赋值，分别对应 `3` 种不同的作用域：

- 如果一个变量在 `def` 内赋值，它对于该函数而言是局部的
- 如果一个变量在一个外层的 `def` 中赋值，对于内层的函数来说，它是非局部的 `nonlocal`
- 如果一个变量在所有 `def` 外赋值，它对整个文件来说是全局的

我们称其为语义作用域 `lexical scoping`，因为变量的作用域完全由变量在程序文件中源代码的位置决定，而不是由函数调用决定。

> 更多的叫法是 *词法作用域* 和 *静态作用域*

作用域细节
函数提供了嵌套的命名空间（作用域），使其内部使用的变量名局部化，以便函数内部使用的变量

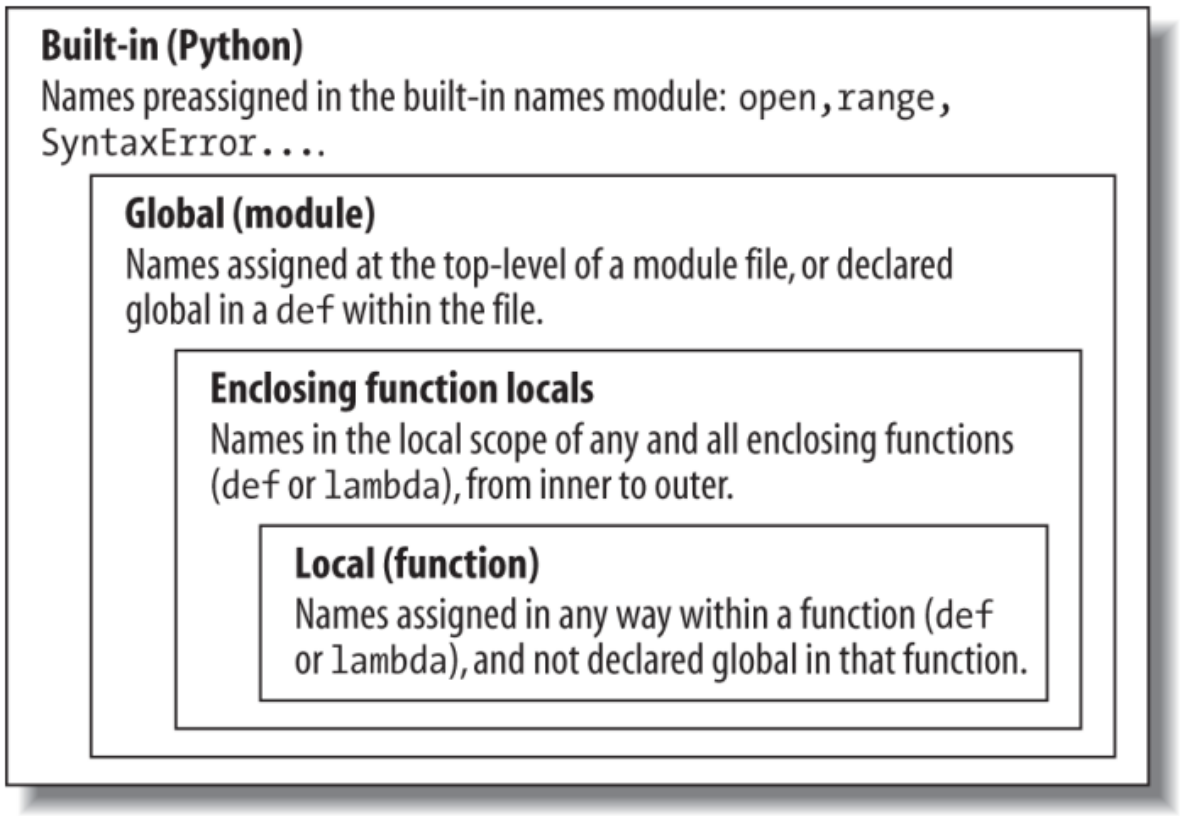
名不会与函数外（在一个模块或另一个函数中）的变量名产生冲突。

函数定义了局部作用域，而模块定义了具有如下属性的全局作用域：

- 每个模块都是一个全局作用域。
- 全局作用域的作用范围仅限于单个文件。
  在 `Python` 中是没有一个跨文件的单一且无所不包的全局作用域概念的。
  相反，变量名被划分到一个个模块中，并且你必须明确地导入一个模块文件才能使用这个文件中定义的变量名。
  当你在 `Python` 中听到 “全局” 时，你就应该联想到 “模块”。
- 函数定义内被赋值的变量名除非被声明为 `global` 或 `nonlocal`，否则均为局部变量。
- 所有其他的变量名都是外层函数的局部变量、全局变量或内置变量。
- 函数的每次调用都会创建一个新的局部作用域。
  局部作用域实际对应于一次函数调用。每一次被激活的调用都能拥有一套自己的函数局部变量副本。

一个函数内部任何类型的复制都会把一个名称划定为局部的。
这包括 `=` 语句、`import` 中的模块名、`def` 中的函数名、函数形式参数名等。
原位置改变对象并不会把变量划分为局部变量，实际上只有对变量名赋值才可以。

变量名解析：`LEGB` 规则



总结成下面的三条简单规则。
对于一条 `def` 语句（或者 `lambda`）：

- 在默认情况下，变量名赋值会创建或改变局部变量。
- 变量名引用至多可以在四种作用域内进行查找：首先是局部，其次是外层的函数（如果有的话），之后是全局，最后是内置。

- 使用 `global` 和 `nonlocal` 语句声明的名称将赋值的变量名分别映射到外围的模块和函数的作用域

所有在函数 `def` 语句（或者 `lambda`）内赋值的变量名默认均为局部变量。函数能够随意使用在外层函数内或全局作用域中的变量名，但是必须声明为非局部变量和全局变量来改变其属性。

这里讲的是简单变量名，而不是属性变量名 `object.spame`。

其他的 Python 作用域：预习
还有另外三种作用域：

- 推导语法中的临时循环变量
- `try` 处理语句中的异常引用变量
- `class` 语句中的局部作用域

前两个是特殊情形，第三个遵循 `LEGB` 原则。
推导变量：用来在推导表达式中指代当前的迭代项。这样的变量在所有的推导形式（生成器、列表、集合和字典）中对于表达式自身都是局部的。在 `2.X` 版本，它们对于生成器表达式、集合和字典推导是局部的，但对于将它们映射到表达式外部作用域的列表推导不是局部的。 `for` 循环在任何版本都不会在其语句块内局部化它们的变量。

异常变量：`except E as X` 中的 变量 `X`，对于 `except` 块是局部的，并且事实上当这个异常块退出时会被移除。因为 `3.X` 版本它们可能会推迟垃圾回收的内存恢复。在 `2.X` 中，这些变量在 `try` 语句之后一直存在。

`class` 块属于 `L` 级别。
同模块和导入一样，这些名称在 `class` 语句结束后也会转变为类对象的属性。

作用域实例
这种变量名隔离机制底层的基本原理就在于局部变量是作为临时的变量名，只有在函数运行时才需要它们。
事实上，当函数调用结束时，局部变量会从内存中移除，它们引用的对象如果没有在别的地方引用则会被垃圾回收掉。

内置作用域
实际上，内置作用域仅仅是一个名为 `builtins` 的内置模块，但是必须要导入 `builtins` 之后才能使用内置作用域，因为名称 `builtins` 本身并没有预先内置。

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
...many more names omitted...
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'su
m',
'super', 'tuple', 'type', 'vars', 'zip']
```

这个列表中的变量名组成了 `Python` 中的内置作用域。
前一半是内置的异常，后一半是内置函数。

特殊名称 `None`、`True` 和 `False` 也在这个列表当中，尽管 `3.X` 把它们当成保留字。
`Python` 会在 `LEGB` 查找中的最后自动查找这个模块，所以你能够使用这些变量名而不需要导入任何模块。
因此，有两种方式引用一个内置函数：利用 `LEGB` 法则，或者手动导入 `builtins` 模块。

```
>>> zip                     # The normal way
<class 'zip'>

>>> import builtins         # The hard way: for customizations
>>> builtins.zip
<class 'zip'>

>>> zip is builtins.zip  # Same object, different lookups
True
```

第二种实现方式有时在更高级的用法中是很有用的。

重定义内置名称：有好有坏
重定义一个内置名称不会让 `Python` 报错。
`PyChecker` 这样的工具可以。
不要重定义一个你需要使用的内置名称。

如果你偶然在交互式命令行重新定义了内置名称，可以：

- 重启会话
- `del name` 移除作用域中的重定义，从而恢复内置作用域中的原始名称

版本差异介绍
`3.X` 的 `builtins` 模块，在 `2.X` 中叫做 `__builtin__`。
在大多数全局作用域，包括交互式会话中，都预先设置了名称 `__builtins__`，来引用 `3.X` 中名为 `builtins` 模块和 `2.X` 中名为 `__builtin__` 的模块。所以你总是可以在不导入的情况下使用 `__builtins__`，但是不可以导入 `__builtins__`，因为它是一个预设变量，而不是一个模块的名称。

在 `3.X` 中，`builtins is __builtins__` 的运行结果在导入 `builtins` 后是 `True`。
在 `2.X` 中，`__bultin__ is __builtins__` 在导入 `__builtin__` 后是 `True`。
我们可以直接运行 `dir(__builtins__)` 来查看内置作用域，而不用导入。

打破 `Python 2.X` 的小宇宙
`2.X` 中由于 `True` 和 `False` 是内置作用域中的变量而不是保留字，所以用 `True = False` 可以给它们重新赋值。这样会影响该作用域的逻辑一致性。

`2.X` 中，还可以用 `__builtin__.True = False`，在整个 `Python` 进程中把 `True` 重置为 `False`。
这样行得通是因为一个程序中只有一个内置作用域模块，由所有它的客户端共享。

通过在内置作用域中重新赋值一个函数的名称，你为进程中的每一个模块都进行了定制化的重置。
可以使用 `PyChecker` 和 `PyLint` 第三方工具查看对内置名称的偶然性赋值的错误。
通常被叫做 `shadowing a built-in`

17.2 global 语句

`global` 语句和它的 `3.X` 近亲 `nonlocal` 语句是 `Python` 中唯一看起来有些像声明语句的语句。
但是，它们不是类型或大小的声明，而是命名空间的声明。
`global` 告诉 `Python` 函数计划生成一个或多个全局变量名，也就是说，存在于整个模块内部作用域（命名空间）的变量名。

总结：

- 全局变量是在外层模块文件的顶层被赋值的变量名。
- 全局变量如果是在函数内被赋值的话，必须经过声明。
- 全局变量名在函数的内部不经过声明也可以被引用。

`global` 语句包含了关键字 `global`，其后跟着一个或多个由逗号分开的变量名。

程序设计：最少化全局变量
`Python` 中的一种惯例：如果想做些“错误”的事情，就得多编写代码。

一些程序指定一个单一的模块文件来收集所有的全局变量。
在 `Python` 中使用多线程进行并行计算的程序实际上也是要依靠全局变量。
线程既可以在全局作用域中共享名称，也可以在进程的内存空间中共享对象。

程序设计：最小化跨文件的修改
每个模块都是自包含的命名空间（变量名的包），而且我们必须导入一个模块才能从另一个模块看到它内部的变量。
通过在文件的层次上隔离变量，它们避免了跨文件的名称冲突，这与局部变量避免跨函数的命名冲突的方式很像。
一个模块文件的全局作用域一旦被导入就成了这个模块对象的一个属性命名空间。
在文件间进行通信的最好办法就是通过调用函数，传递参数，然后得到其返回值。

```
# first.py
X = 99 # This code doesn't know about second.py

# second.py
import first
print(first.X) # OK: references a name in another file
first.X = 88 # But changing it can be too subtle and implicit
```

这种隐含的跨文件依赖性，在最好的情况下会导致代码不灵活，在最坏的情况下会引发错误。
改动成：

```
# first.py
X = 99

def setX(new): # Accessor make external changes explit
```

```
    global X # And can manage access in a single place
    X = new

# second.py
import first
first.setX(88) # Call the function instead of changing directly
```

第二个方式，在可读性和可维护性上有着天壤之别：
当别人仅阅读第一个模块文件时看到这个函数，会知道这是一个接口，并且知道这将改变变量 `X`。

> 属性访问器，`attribute accessor`。

其他访问全局变量的方式
由于全局变量构成了一个被导入的对象的属性，我们能通过使用导入外层模块并对其属性进行赋值来模拟 `global` 语句。
`sys.modules` 包含已载入的模块表。

> `sys.modules` 其实是一个字典。里面放的是 `模块名: 模块对象`。

```
# thismod.py

var = 99 # Global variable == module attribute
def local():
    var = 0 # Change local var

def glob1():
    global var # Declare global (normal)
    var += 1 # Change global var

def glob2():
    var = 0 # Change local var
    import thismod # Import myself
    thismod.var += 1 # Change global var

def glob3():
    var = 0 # Change local var
    import sys # Import system table
    glob = sys.modules['thismod'] # Get module object (or use __name__)
    glob.var += 1 # Change global var

def glob4():
    import sys
    glob = sys.modules[__name__]
    glob.var += 1

def test():
    print(var)
    local(); glob1(); glob2(); glob3();glob4()
```

```
    print(var)
```

在运行时，会给全局变量加 `4`。

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

17.3 作用域和嵌套函数

`E` 这一层是在 `Python 2.2` 中增加的，它包括了任意外层函数局部作用域的形式。
嵌套作用域有时也叫作静态嵌套作用域。
`Enclosing scopes are sometimes also called statically nested scopes.`
嵌套是一种代码写法上的表述：嵌套的作用域对应于程序源代码文本物理上和句法上的嵌套代码结构。

嵌套作用域的细节

- 一个引用在查找嵌套作用域的时候，会从内向外查找
- 一个赋值如果声明为非局部变量，会修改最近的嵌套函数的局部作用域中的变量。

嵌套作用域举例
可以说，`f2` 是一个临时函数，仅在 `f1` 函数内部执行的过程中存在，并且只对 `f1` 中的代码可见。

```
X = 99              # Global scope name: not used

def f1():
    X = 88          # Enclosing def local
    def f2():
        print(X)    # Reference made in nested def
    f2()

f1()                # Prints 88: enclosing def local
```

通过 `LEGB` 查找规则，`f2` 内的 `X` 自动映射为 `f1` 中的 `X`。
`the X in f2 is automatically mapped to the X in f1, by the LEGB lookup rule.`

这种嵌套作用域查找即便在外层的函数已经返回后也是有效的。更为普遍的使用模式：

```
def f1():
    X = 88
    def f2():
        print(X)      # Remembers X in enclosing def scope
    return f2         # Return f2 but don't call it

action = f1()         # Make, return function
action()              # Call it now: prints 88
```

对 `action` 名称的调用本质上运行了 `f1` 运行时我们命名为 `f2` 的函数。
这能行得通是因为 `Python` 中的函数与其他一切一样是对象，因此可以作为其他函数的返回值传递回来。
更为重要的是，`f2` 记住了 `f1` 的嵌套作用域中的 `X`，尽管 `f1` 已经不处于激活状态。

工厂函数：闭包
可以叫做闭包 `closure` 也可以叫做工厂函数 `factory function`。
前者把它描述成一种函数式编程技巧，后者将它认为是一种设计模式。
这里讨论的函数对象能够记忆外层作用域里的值，不管那些嵌套作用域是否还在内存中存在。
从结果上看，它们附加了内存包（又称为状态记忆）。
`In effect, they have attached packets of memory (a.k.a. state retention)`
内存包/状态记忆对于每个被创建的嵌套函数副本而言都是局部的。
从这一作用来看，它们经常提供了一种类的简单替代方式。

一个简单的函数工厂

> 函数工厂？
> 工厂函数（又名闭包）有时用于事件处理器程序，这些程序需要对运行时的情况做出即时响应。

```
>>> def maker(N):
        def action(X):      # Make and return action
            return X ** N   # action retains N from enclosing scope
    return action
```

这里定义了一个外层函数，用来简单地生成并返回一个嵌套的函数，却并不调用这个内嵌的函数。
如果调用外部函数，得到的是生成的内嵌函数的一个引用。

```
>>> f = maker(2)          # Pass 2 to argument N
>>> f
<function maker.<locals>.action at 0x0000000002A4A158>

>>> f(3)                  # Pass 3 to X, N remembers 2: 3 ** 2
9
>>> f(4)                  # 4 ** 2
16
```

虽然在调用 `action` 时 `maker` 已经返回了值并退出，但是内嵌的函数记住了整数 `2`。
实际上，在外层嵌套局部作用域内的 `N` 被作为执行的状态信息保留了下来，并附加到生成的 `action` 函数上。

嵌套作用域常常被 `lambda` 函数创建表达式利用。

```
>>> def maker(N):
        return lambda X: X ** N  # lambda functions retain state too
>>> h = maker(3)
>>> h(4)                         # 4 ** 3 again
64
```

闭包 vs 类：回合 1
类是一个更好的实现这种状态记忆的选择，因为它们用属性赋值来更加显式地创建它们的内存。
类也直接支持闭包函数所不支持的其他工具。
类可以实现很多行为。
所以，类在实现功能更完整的对象上往往表现的更好。

但是，当记忆状态是唯一的目标时，闭包函数经常提供一个轻量级的可行的替代方案。它们为每一次调用提供局部化存储空间，来存储一个单独的内嵌函数所需的数据。
`3.X` 的 `nonlocal` 还允许嵌套作用域状态改变。
而 `2.X` 的嵌套作用域是只读的。

当函数返回时，普通局部变量会消失，但是可以通过下面五种方式在不同的函数调用之间记忆状态（`retain state between calls`）

- 全局变量
- 类实例的属性
- 对作用域外层的引用
- 参数默认值
- 函数属性

> 函数属性是什么意思？
> 本章后面讲

当一个 `class` 嵌套在一个 `def` 中时，闭包也可以被创建：外层函数的局部名称的值会被类中的引用，或类中的一个方法函数所保存。外层 `def` 扮演了一个类似的角色：它成了一个类工厂，并为嵌套类提供状态记忆。

使用默认值参数来保存外层作用域的状态
较早版本的 `Python` 上面的代码会失败（`2.2` 以前），因为没有 `E` 层。
为了解决这一问题，程序员一般都会将默认参数的值传递给一个外层作用域内的对象。

```
def f1():
    x = 88
    def f2(x=x):  # Remember enclosing scope X with defaults
        print(x)
```

```
    f2()

f1()                # Prints 88
```

这种代码在所有版本通用。这种代码显式地赋值外层作用域状态并使其保存。
要求 `x` 在进入内嵌的 `def` 之前就已经完成其求值。
实际上，嵌套作用域查找规则之所以加入到 `Python` 中就是为了让默认值参数不再扮演这种角色。

扁平通常胜于嵌套，所以大部分代码还是不要嵌套 `def`。

在某一个函数内部调用一个之后才定义的函数是可行的，只要第二个函数定义的运行是在第一个函数调用前就行，在 `def` 内部的代码直到这个函数实际调用时才会被求值。

```
>>> def f1():
        x = 88      # Pass x along instead of nesting
        f2(x)       # Forward reference OK
>>> def f2(x):
        print(x)    # Flat is still often better than nested!
>>> f1()
88
```

嵌套作用域，默认值参数和 lambda
`lambda` 表达式在 `def` 中使用的越来越多。
`lambda` 是一个表达式，因此能用在 `def` 中不能使用的地方。比如字典或列表的字面量中。
像 `def` 一样，`lambda` 表达式也为其所创建的函数引入新的局部作用域。

```
def func():
    x = 4
    action = (lambda n: x ** n)  # x remembered from enclosing def
    return action

x = func()
print(x(2))                      # Prints 16, 4 ** 2
```

在没有引入嵌套作用域的时候，需要借助默认值参数从上层作用域传递值给 `lambda`。下面的代码适用于所有版本的 `Python`：

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n) # Pass x in manually
    return action
```

> 手动传入 `x`

由于 `lambda` 是表达式，所以它们非常适合嵌套在 `def` 中。
它们某些意义上成了后来在查找规则中新增外层函数作用域的最大初始受益者。

循环变量可能需要默认值参数，而不是作用域
如果在函数中定义的 `lambda` 或者 `def` 嵌套在一个循环之中，而这个内嵌函数又引用了一个外层作用域的变量，该变量被循环所改变，那么所有在这个循环中产生的函数会有相同的值——也就是在最后一次循环中完成时被引用变量的值。

```
>>> def makeActions():
        acts = []
        for i in range(5): # Tries to remember each i
            acts.append(lambda x: i ** x) # But all remember same last i!
        return acts

>>> acts = makeActions()
>>> acts[0]
<function makeActions.<locals>.<lambda> at 0x0000000002A4A400>
```

因为外层作用域中的变量在嵌套的函数被调用时才进行查找，所以它们实际上记住的是同样的值，也就是在最后一次循环迭代中循环变量的值。

```
>>> acts[0](2)  # All are 4 ** 2, 4=value of last i
16
>>> acts[1](2)  # This should be 1 ** 2 (1)
16
>>> acts[2](2)  # This should be 2 ** 2 (4)
16
>>> acts[4](2)  # Only this should be 4 ** 2 (16)
16
```

这是唯一一种还需要我们动用默认值参数来显式保持外层作用域中值的情况，而不是简单地使用外层作用域的引用。
因为默认值参数的求值是在嵌套函数创建时就发生的，而不是该函数之后被调用时发生的。

```
>>> def makeActions():
        acts = []
        for i in range(5): # Use defaults instead
            acts.append(lambda x, i=i: i ** x) # Remember current i
        return acts

>>> acts = makeActions()
>>> acts[0](2) # 0 ** 2
0
>>> acts[1](2) # 1 ** 2
1
>>> acts[2](2) # 2 ** 2
4
>>> acts[4](2) # 4 ** 2
```

```
16
```

如果对默认参数使用可变对象（比如 `def f(a=[])` ），因为默认参数是通过附加在函数上的 *单个* `single` 对象来实现的，可变类型的默认参数在调用之间记忆状态，而不是每次调用都重新初始化。这可以被认为是另外一种状态记忆的实现，也可以被认为是 `Python` 的怪现象。

17.4 Python 3.X 中的 nonlocal 语句

`nonlocal` 语句通过提供可改写的状态信息，让嵌套作用域闭包变的更加有用。
在声明 `nonlocal` 名称时，它必须已经存在于该外层函数的作用域中。
也就是说，它们只能现成存在于外层函数中，而不能由内嵌 `def` 中的第一次赋值来创建。

nonlocal 基础
它只在函数内部才有意义：

```
def func():
    nonlocal name1, name2, ... # OK here

>>> nonlocal X
SyntaxError: nonlocal declaration not allowed at module level
```

`nonlocal` 语句允许内嵌函数修改定义在语法上位于外层的函数的作用域中的一个或多个名称。
`nonlocal` 意味着：完全略过我的局部作用域，直接从外层作用域开始查找。
当执行到 `nonlocal` 语句的时候，`nonlocal` 中列出的名称必须在一个外层的 `def` 中被提前定义过，否则将引发一个错误。
`global` 意味着名称位于外层的模块中；`nonlocal` 意味着名称位于外层的 `def` 中。
`nonlocal` 甚至更严格，`nonlocal` 名称只能出现在外层的 `def` 中，而不能在模块的全局作用域中或 `def` 之外的内置作用域中。
`nonlocal` 语句的主要作用是能让外层作用域中的名称被修改，而不仅仅是被引用。

使用 `global` 和 `nonlocal` 都在某种程度上限制了查找规则：

- `global` 使得作用域查找从外围模块的作用域开始，并且允许对那里的名称赋值。如果不存在，继续在内置作用域中找。
- `nonlocal` 将作用域的查找限制为只在外层的 `def` 中，同时要求名称已经存在在那里，并允许对它们赋值。作用域查找不会继续进入到全局或内置作用域。

nonlocal 应用

```
>>> def tester(start):
        state = start
        def nested(label):
            print(label, state)
            state += 1 # Cannot change by default (never in 2.X)
        return nested
```

```
>>> F = tester(0)
>>> F('spam')
UnboundLocalError: local variable 'state' referenced before assignment
```

使用 nonlocal 进行修改
外层作用域中的 `state` 对象本质上被附加到了返回的 `nested` 函数对象中。
每次调用都产生一个新的、单独的 `state` 对象，以至于更新一个函数的 `state` 不会影响到其他的。

在闭包函数中，`nonlocal` 是基于调用的、多副本的数据。
`in a closure function, nonlocals are per-call, multiple copy data`

边界情况
首先，你不能通过赋值动态地在外层作用域中创建一个新的 `nonlocal` 名称。
事实上，`nonlocal` 名称在外层或内嵌函数被调用之前就已经在函数定义的时候被检查了：

```
>>> def tester(start):
        def nested(label):
            nonlocal state # Nonlocals must already exist in enclosing de
f!
            state = 0
            print(label, state)
        return nested

SyntaxError: no binding for nonlocal 'state' found

>>> def tester(start):
        def nested(label):
            global state # Globals don't have to exist yet when declared
            state = 0 # This creates the name in the module now
            print(label, state)
        return nested

>>> F = tester(0)
>>> F('abc')
abc 0
>>> state
0
```

其次，`nonlocal` 将作用域查找限制为只对外层的 `def`，不会再外围模块的全局作用域或所有 `def` 之外的内置作用域中查找，即便这些作用域中存在相同的名称。

```
>>> spam = 99
>>> def tester():
        def nested():
            nonlocal spam # Must be in a def, not the module!
            print('Current=', spam)
            spam += 1
```

```
        return nested

SyntaxError: no binding for nonlocal 'spam' found
```

17.5 为什么选 nonlocal？状态保持备选项

在很多程序中，状态信息是非常重要的。
虽然函数可以返回结果，但它们的局部变量一般不会保留，而我们有时需要让这些值在调用之间被保持。
此外按照不同的使用场景，其他一些应用程序又要求这些值不同。

nonlocal 变量的状态：仅适用于 Python 3.X

全局变量的状态：只有一份副本
在 `2.X` 和较早版本中实现 `nonlocal` 效果的一种常见方式，就是直接把状态移到全局作用域外（外围的模块）。
缺点：

- 两个函数都要使用 `global`
- 它只允许模块作用域中保存状态信息的单个共享副本

```
>>> def tester(start):
        global state # Move it out to the module to change it
        state = start # global allows changes in module scope
        def nested(label):
            global state
            print(label, state)
            state += 1
        return nested

>>> F = tester(0)
>>> F('spam') # Each call increments shared global state
spam 0
>>> F('eggs')
eggs 1

>>> G = tester(42) # Resets state's single copy in global scope
>>> G('toast')
toast 42

>>> G('bacon')
bacon 43

>>> F('ham') # But my counter has been overwritten!
ham 44
```

带状态的类：显式属性
`Python 2.X` 和较早版本中针对可变状态信息的另一种方式是使用带有属性的类。
从而让状态信息的访问比隐式作用域查找规则更明确。
作为附加优势，类的每个实例都会得到状态信息的一个新副本。
类中的 `self` 参数会自动接收调用主体。

> 之前听过的一句话：`self` 指的就是函数的调用者对象

当类调用时，名为 `__init__` 的函数会自动运行。

使用运算符重载把类对象用作可调用函数。
`__call__` 拦截了一个实例上的直接调用。

```
>>> class tester:
    def __init__(self, start):
        self.state = start
    def __call__(self, label): # Intercept direct instance calls
        print(label, self.state) # So .nested() not required
        self.state += 1

>>> H = tester(99)
>>> H('juice') # Invokes __call__
juice 99
```

类可以让状态信息更明显，通过利用显式属性赋值而不是隐式作用域查找。

函数属性的状态：Python 3.X 和 Python 3.X 的异同
函数属性：用户定义的直接附加给函数的名称。
`function attributes—user-defined names attached to functions directly.`
当你给外层工厂函数生成的内嵌函数附加用户定义的属性时，它们也可以作为基于调用、多副本和可写的状态。
与 `nonlocal` 作用域闭包和类属性一样。

关键是，函数属性与类一样是版本间可移植的，`2.X` 和 `3.X` 通用。
`2.1` 版本后就可用了。

此外，函数属性允许状态变量从内嵌函数的外部被访问，就像类的实例属性那样，而如果使用 `nonlocal`，状态变量只能在内嵌的 `def` 内部被直接看到。
通过函数名而不是简单的变量来访问状态，必须在内嵌的 `def` 之后初始化。

```
>>> def tester(start):
        def nested(label):
            print(label, nested.state) # nested is in enclosing scope
            nested.state += 1 # Change attr, not nested itself
        nested.state = start # Initial state after func defined
        return nested
```

```
>>> F = tester(0)
>>> F('spam') # F is a 'nested' with state attached
spam 0
>>> F('ham')
ham 1
>>> F.state # Can access state outside functions too
2
```

因为对于外层函数的每一次调用都产生一个新的内嵌函数对象，这一方案与 `nonlocal` 闭包和类一样，能支持基于调用的多副本可更改数据，这也是全局变量所不能提供的使用模式。

```
>>> G = tester(42) # G has own state, doesn't overwrite F's
>>> G('eggs')
eggs 42
>>> F('ham')
ham 2

>>> F.state # State is accessible and per-call
3
>>> G.state
43
>>> F is G # Different function objects
False
```

在原位置修改一个对象并不是给一个名称赋值，所以不用 `nonlocal`。

先抛开主观因素，函数属性的作用确实同 `3.X` 中较新的 `nonlocal` 相重叠，这使得后者从技术上来讲是多余的，可移植性也更差。

可变对象的状态：来自 `Python` 历史中的隐蔽幽灵？
在 `2.X` 和 `3.X` 中，不用 `nonlocal` 声明而改变外层作用域中的一个可变对象也是可能的。

```
def tester(start):
    def nested(label):
        print(label, state[0]) # Leverage in-place mutable change
        state[0] += 1 # Extra syntax, deep magic?
    state = [start]
    return nested
```

> 同理，也可以不使用 global 而修改全局变量的值，只要全局变量是可变的。
> 关键在于：在原位置修改一个对象并不是给一个名称赋值
> 下面是一个小例子

```
glob = [7]
```

```
def func():
    glob[0] += 1

func()

print(glob[0])
```

总结一下：
全局变量、非局部变量、带属性的类和函数属性都提供了可更改的状态记忆的备选项。全局变量只支持单副本的共享数据；非局部变量只能在 `3.X` 中被改变（ `nonlocal` ）；带属性的类和函数属性都提供了版本间可移植的解决方案，同时也允许从持有状态的可调用的对象自身的外部直接访问状态。

装饰器：一个天生就与多级状态记忆相关的工具。

请留意：定制 `open`
`makeopen.py`

```
import builtins
def makeopen(id):
    original = builtins.open
    def custom(*kargs, **pargs):
        print('Custom open call %r:' % id , kargs, pargs)
        return original(*kargs, **pargs)
    builtins.open = custom
```

```
>>> F = open('script2.py') # Call built-in open in builtins
>>> F.read()
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

>>> from makeopen import makeopen # Import open resetter function
>>> makeopen('spam') # Custom open calls built-in open

>>> F = open('script2.py') # Call custom open in builtins
Custom open call 'spam': ('script2.py',) {}
>>> F.read()
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
```

每一次定制过程都在自己的外层作用域中记住了上一个内置作用域版本。

```
>>> makeopen('eggs') # Nested customizers work too!
>>> F = open('script2.py') # Because each retains own state
Custom open call 'eggs': ('script2.py',) {}
Custom open call 'spam': ('script2.py',) {}
>>> F.read()
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
```

基于类的代码也可以实现：

```
import builtins

class makeopen: # See Part VI: call catches self()
    def __init__(self, id):
        self.id = id
        self.original = builtins.open
        builtins.open = self
    def __call__(self, *kargs, **pargs):
        print('Custom open call %r:' % self.id, kargs, pargs)
        return self.original(*kargs, **pargs)
```

17.6 本章小结

与函数相关的两个关键概念：

- 作用域
- 参数传递

17.7 本章习题

举出三种或更多在 `Python` 函数中保持状态信息的方式？

1. 全局变量
2. `3.X` 中的非局部变量 `nonlocal`
3. 默认值参数
4. 可变类型的对象
5. 函数属性
6. 带属性的类的实例

---

18. 参数

作用域：代码中定义和查找变量的位置。
参数传递：对象作为输入传递给函数的方式。

18.1 参数传递基础

- 参数的传递是通过自动将对象赋值给局部变量名来实现的。
  所以函数参数传递的是共享对象的引用。因为引用是以指针的形式实现的，所有的参数传递实际上都是通过指针传入的。作为参数被传递的对象从来不会自动复制。
- 在函数内部赋值参数名不会影响调用者。
  函数参数名和调用者作用域中的变量名是没有关联的。
- 改变函数的可变对象参数的值也许会对调用者有影响。
  原因在于函数可以原位置改变可变对象。
- 不可变对象做参数本质上传入了 “值”。
  虽然不可变对象传递的也是引用而不是赋值，但是你无法在原位置修改它，最终效果就像是创建了一份副本。
- 可变对象做参数本质上传入了 “指针”。
  有点像 `C` 语言用指针传递数组。

简单来说：就是把对象赋值给变量名。

参数和共享引用

```
>>> def changer(a, b): # Arguments assigned references to objects
        a = 2 # Changes local name's value only
        b[0] = 'spam' # Changes shared object in place

>>> X = 1
>>> L = [1, 2] # Caller:
>>> changer(X, L) # Pass immutable and mutable objects
>>> X, L # X is unchanged, L is different!
(1, ['spam', 2])
```

对函数内的一个参数名的赋值，不会影响到主调函数作用域中的变量。
而原位置修改，只有在被修改对象的生命周期比函数调用更长的时候，才会影响到调用者。
从函数调用的效果上来看，列表名 `L` 同时充当了函数的输入和输出。

模拟输出参数和多重结果

```
>>> def multiple(x, y):
        x = 2 # Changes local names only
        y = [3, 4]
        return x, y # Return multiple new values in a tuple

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L) # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

尽管 `Python` 不支持一些其他语言所谓的 “按引用调用” 的参数传递，我们一般能通过返回元组并将结果赋值给调用者中原有的参数变量名来模拟这种行为。

`2.X` 可以在传递给函数的参数中自动解包元组：

```
def f((a, (b, c))):
```

可以用于期望的结果相匹配的元组来调用：`f((1, (2, 3)))`，也可以传入事先创建好的对象 `f(T)`。
在 `3.X` 不支持这样的语法，作为替代可以这么写：

```
def f(T): (a, (b, c)) = T
```

在一条显式赋值语句中完成解包。

`2.X` 只支持元组形式的序列赋值，一般形式的序列比如列表会发生语法错误。
元组解包语法在 `3.X` 的 `lambda` 函数的参数列表中不被允许，但是在 `3.X` 中的 `for` 循环对象中仍然是自动化的。

18.2 特殊的参数匹配模式

参数在 `Python` 中总是通过赋值传入的。
在默认情况下，参数按照从左到右的配置进行匹配，而且你必须精确地传入和函数头部参数名一样多的参数。
当然，你也能通过形式参数名、提供默认的参数值以及对额外参数适用容器三种方法来指定匹配。

参数匹配基础
这些特殊模式是可选的，并且只能解决变量名与对象的匹配问题；匹配完成后在传递机制的底层依然是赋值。

匹配模式大纲：

- *位置次序：从左至右进行匹配*
- *关键字参数：通过参数名进行匹配*
- *默认值参数：为没有传入值的可选参数指定参数值*
- *可变长参数收集：收集任意多的基于位置或关键字的参数*
  这个特性常常被叫做 可变长参数，借鉴自 `C` 语言中的可变长度参数列表工具。
- *可变长参数解包：传入任意多的基于位置或关键字的参数*
- `keyword-only` *参数：必须按照名称传递的参数*
  `3.X` 才有的功能。

参数匹配语法

Table 18-1. Function argument-matching forms

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*other, name)`	Function	Arguments that must be passed by keyword only in calls (3.X)
`def func(*, name=value)`	Function	Arguments that must be passed by keyword only in calls (3.X)

这些特殊的匹配模式可以分为 函数调用 和 函数定义 两个阶段来理解。

更深入的细节

- 函数调用时的参数顺序：位置参数，任意关键字参数 和 `*iterable` 形式的组合，`**dict` 形式。
- 函数定义时的参数顺序：一般参数，默认参数，`*name` 或者是 `3.X` 中的 `*`，所有 `name` 或 `name=value` 的 `keyword-only` 参数，之后是 `**name` 形式。

在函数调用和函数定义中，`**kwargs` 必须在最后。

`Python` 内部大致是使用以下的步骤来在赋值前匹配参数的：

1. 通过位置分配非关键字参数
2. 通过匹配名称分配关键字参数
3. 将剩下的非关键字参数分配到 `*name` 元组中
4. 将剩下的关键字参数参数分配到 `**name` 字典中
5. 把默认值分配给在头部未得到匹配的参数
6. 检查确保每个参数只被传入了一个值，如果不是这样的话会报错
7. 当所有的匹配都完成了，`Python` 再把传入的对象赋值给参数名称

（这里没有考虑强制关键字参数）

在 `3.X` 中，函数头部中的参数名称还可以有一个注解值，其指定形式为 `name:value`，或在要给出默认值时，为 `name:value=default` 形式
函数自身也可以有一个注解值，以 `def f()->value` 的形式给出。 `Python` 将注解值附加到函数对象上。

关键字参数和默认值参数的示例

```
>>> def f(a, b, c): print(a, b, c)

>>> f(1, 2, 3)
1 2 3
```

```
>>> f(c=3, b=2, a=1)
1 2 3

>>> f(1, c=3, b=2) # a gets 1 by position, b and c passed by name
1 2 3
```

在使用了关键字后参数从左至右的关系就不在重要了，因为现在参数是通过名称匹配，而不是基于位置匹配。

关键字参数的作用：

- 关键字参数在调用中起到了数据标签的作用，它们让你的调用变得更自文档化一些。
- 关键字参数从本质上允许我们调用时跳过带默认值的参数。

默认值参数：允许我们让特定的参数变为可选的。

```
>>> def f(a, b=2, c=3): print(a, b, c) # a required, b and c optional

>>> f(1) # Use defaults
1 2 3
>>> f(a=1)
1 2 3

>>> f(1, 4) # Override defaults
1 4 3
>>> f(1, 4, 5)
1 4 5

>>> f(1, c=6) # Choose defaults
1 2 6
```

关键字参数从本质上允许我们跳过带默认值的参数。

不要把函数头部和函数调用时的 `name=value` 语法混为一谈。
在函数调用时，这意味着通过名称匹配的关键字参数；
而在函数头部时，它为一个可选的参数指定了默认值。
无论哪种情况，都不是一条赋值语句（尽管长的一样）。
它是在这两种指定上下文中特殊的语法，改变了默认的参数匹配机制。

混合使用关键字参数和默认值参数
`name=value` 形式在调用时和 `def` 中有两种不同的含义：在调用时代表关键字参数，而在函数头部代表默认值参数。

```
def func(spam, eggs, toast=0, ham=0): # First 2 required
    print((spam, eggs, toast, ham))

func(1, 2) # Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0) # Output: (1, 0, 0, 1)
func(spam=1, eggs=0) # Output: (1, 0, 0, 0)
```

```
func(toast=1, eggs=2, spam=3) # Output: (3, 2, 1, 0)
func(1, 2, 3, 4) # Output: (1, 2, 3, 4)
```

如果在调用时使用了关键字参数，那么关键字参数的顺序将不再影响其匹配。
必须提供的 `spam` 和 `egg`，可以自由选择使用位置或者关键字参数指定。

注意：可变类型的对象做默认值参数，比如 `def f(a=[])`，同一个可变类型的对象会在函数的所有调用中被反复地使用，即使它在函数中被原位置修改了。
为了每次都重置一个新的值，你需要把赋值语句转移到函数的函数体中。
可变类型的对象做默认值参数允许状态记忆。

可变长参数的实例
函数定义中：收集参数
`*` 在函数定义中把不能匹配的基于位置的参数收集到一个元组中。
`**` 把关键字参数收集到一个字典中。
混合使用：

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

这种代码比较少见，但是会出现在需要支持多重调用模式的函数中，例如 要实现向前兼容的函数。

函数调用中：解包参数
混合使用：

```
>>> def func(a, b, c, d): print(a, b, c, d)

>>> func(*(1, 2), **{'d': 4, 'c': 3}) # Same as func(1, 2, d=4, c=3)
1 2 3 4
>>> func(1, *(2, 3), **{'d': 4}) # Same as func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *(2,), **{'d': 4}) # Same as func(1, 2, c=3, d=4)
1 2 3 4
>>> func(1, *(2, 3), d=4) # Same as func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, *(2,), c=3, **{'d':4}) # Same as func(1, 2, c=3, d=4)
1 2 3 4
```

> 注意：这是我以前不知道的
> 只要保证前面是位置参数，后面是 `**dict`
> 中间可以是 `name=value` 和 `*iterable` 形式的组合

别混淆函数头部和函数调用时 `*` / `**` 的语法：

- 在函数头部，它意味着收集任意多的参数

- 在函数调用，它能解包任意多的参数
- 一个星号代表基于位置的参数
- 两个星号代表关键字参数

在调用时，`*args` 形式是一个迭代上下文，它可以接收任何可迭代对象。
比如 `func(*open('fname'))`
但是在赋值语句中的解包，总是创建列表而不是元组。

> 个人理解：赋值语句解包出来的参数需要进行修改，而函数调用时解包出来的参数不用进行修改。

泛化地使用函数
可变长参数的调用的强大之处在于：在编写一段脚本之前不需要知道一个函数调用需要多少参数。

```
if sometest:
    action, args = func1, (1,) # Call func1 with one arg in this case
else:
    action, args = func2, (1, 2, 3) # Call func2 with three args here
...etc...
action(*args) # Dispatch generically
```

函数本身是对象：函数可以用任意的变量名来引用和调用

每当你无法预计参数列表时，这种可变长参数调用语法都是很有用的。

```
def tracer(func, *pargs, **kargs): # Accept arbitrary arguments
    print('calling:', func.__name__)
    return func(*pargs, **kargs) # Pass along arbitrary arguments

def func(a, b, c, d):
    return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))
```

函数的 `__name__` 属性，是它的名称字符串。

过时的 `apply` 内置函数（`2.X`）
`3.X` 之前，`*args` 和 `**kwargs` 可变长参数的调用语法可以通过一个名为 `apply` 的内置函数来实现。

```
func(*pargs, **kargs) # Newer call syntax: func(*sequence, **dict)
apply(func, pargs, kargs) # Defunct built-in: apply(func, sequence, dict)
```

`2.X` 举例：

```
>>> apply(pow, (2, 100))
```

```
1267650600228229401496703205376L
>>> pow(*(2, 100))
1267650600228229401496703205376L
```

`2.0` 引入了 `apply`，`2.3` 被标为弃用，`3.0` 及以后被废除。

`3.X` 的 `keyword-only` 参数

`keyword-only` 参数：必须只按照关键字传入并且永远不会被基于位置来填充的参数。
从语法上讲，`keyword-only` 参数是出现在参数列表中 `*args` 之后的有名参数。
所有这些参数都必须在调用中使用关键字语法来传递。

```
>>> def kwonly(a, *b, c):
        print(a, b, c)

>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
```

还可以在参数列表中单独使用一个 `*` 字符，来表示一个函数不会接受可变长度的参数列表，但是仍然期待跟在 `*` 后面的所有参数都作为关键字参数传入。

下面的例子，`a` 可以按照位置或者名称传入，但是 `b` 和 `c` 必须按照关键字传入，且不允许其他额外的基于位置传入：

```
>>> def kwonly(a, *, b, c):
        print(a, b, c)
>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
>>> kwonly(1)
TypeError: kwonly() missing 2 required keyword-only arguments: 'b' and
'c'
```

可以对 `keyword-only` 参数使用默认值，只要让它们出现在函数头部 `*` 的后面。
实际上，带默认值的 `keyword-only` 参数都是可选的，但是没有默认值的 `keyword-only` 参数真正地变成了函数必须的 `keyword-only` 参数。

```
>>> def kwonly(a, *, b=1, c, d=2):
        print(a, b, c, d)
```

```
>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
```

> 注意函数头部定义 `def kwonly(a, *, b=1, c, d=2)`
> 这样写也是可以的！

在函数头部，`keyword-only` 参数必须在一个单独星号后面，或者可变长参数 `*args` 后面，而不能在 `**kwargs` 后面；
另外单独的两个星号会报错。

> 也就是说，强制关键字参数是出现在 `*args` 和 `**kwargs` 之间的参数。
> 如果没有 `*args`，用 `*` 代替。

在函数调用，`keyword-only` 参数必须在 `**args` 形式之前。`keyword-only` 参数可以编写在 `*args` 之前或之后，也可以在 `**args` 之内。

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # KW-only between * and **

>>> f(1, *(2, 3), **dict(x=4, y=5)) # Unpack args at call
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5), c=7) # Keywords before **args!
SyntaxError: invalid syntax

>>> f(1, *(2, 3), c=7, **dict(x=4, y=5)) # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, c=7, *(2, 3), **dict(x=4, y=5)) # After or before *
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5, c=7)) # Keyword-only in **
1 (2, 3) 7 {'y': 5, 'x': 4}
```

18.3 min 提神小例

`Python` 中的 `sort` 方法是用 `C` 语言编写的，使用了 `timsort` 算法，创始人是 `Tim Peters`，利用了序列中元素部分有序的性质来进行排序。

时间复杂度为 `O(nlog(n))`。

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args) # Or, in Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print(min1(3, 4, 1, 2))
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

让这些函数可以同时计算最大值和最小值：

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y # See also: lambda, eval
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

% python minmax.py
16
```

18.4 通用 set 函数

`inter2.py`

```
def intersect(*args):
    res = []
    for x in args[0]:                       # Scan first sequence
        if x in res: continue               # Skip duplicates
        for other in args[1:]:              # For all other args
            if x not in other: break        # Item in each one?
        else:                               # No: break out of loop
            res.append(x)                   # Yes: add items to end
    return res

def union(*args):
    res = []
    for seq in args:                        # For all args
        for x in seq:                       # For all nodes
            if not x in res:
                res.append(x)               # Add new items to result
    return res
```

编写了测试函数：

```
>>> def tester(func, items, trace=True):
        for i in range(len(items)):
            items = items[1:] + items[:1]
            if trace: print(items)
            print(sorted(func(*items)))

>>> tester(intersect, ('a', 'abcdefg', 'abdst', 'albmcnd'))
('abcdefg', 'abdst', 'albmcnd', 'a')
['a']
('abdst', 'albmcnd', 'a', 'abcdefg')
['a']
('albmcnd', 'a', 'abcdefg', 'abdst')
['a']
('a', 'abcdefg', 'abdst', 'albmcnd')
['a']

>>> tester(union, ('a', 'abcdefg', 'abdst', 'albmcnd'), False)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']

>>> tester(intersect, ('ba', 'abcdefg', 'abdst', 'albmcnd'), False)
['a', 'b']
['a', 'b']
['a', 'b']
['a', 'b']
```

测试函数中对参数顺序的打乱可以作为一种通用的工具，比如写成一个名为 `scramble` 的函数，`tester` 从而会被简化。

```
>>> def tester(func, items, trace=True):
        for args in scramble(items):
            ...use args...
```

18.5 模拟 Python 3.X print 函数

`2.X` 可以 `from __future__ import print_function`

在 `2.X` 中模拟 `3.X` 的 `print`，下面的代码对 `3.X` 也适用。
`print3.py`

```
#!python
"""
Emulate most of the 3.X print function for use in 2.X (and 3.X).
Call signature: print3(*args, sep=' ', end='\n', file=sys.stdout)
"""
import sys

def print3(*args, **kargs):
    sep = kargs.get('sep', ' ') # Keyword arg defaults
    end = kargs.get('end', '\n')
    file = kargs.get('file', sys.stdout)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)

print3(1, 2, 3, sep='??', end='.\n', file=sys.stderr)

"""
1??2??3.
"""
```

使用 keyword-only 参数
变体 `print3_alt1.py`

```
#!python3
"Use 3.X only keyword-only args"
import sys

def print3(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
```

```
        first = True
        for arg in args:
            output += ('' if first else sep) + str(arg)
            first = False
        file.write(output + end)
```

强制关键字参数的作用：自动验证配置参数。
这个函数不接收其他的关键字参数，如果提供，会发生异常。
而前一个函数会默默地忽略掉那些参数。

```
>>> print3(99, name='bob')
TypeError: print3() got an unexpected keyword argument 'name'
```

如果想要在最初的版本手动检测多余的关键字，可以用 `dict.pop()` 删除取回的传入项。

> 补充说明：
> D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
> If key is not found, d is returned if given, otherwise KeyError is raised

等价版本
`print2_alt2.py`

```
#!python
"Use 2.X/3.X keyword args deletion with defaults"
import sys

def print3(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('extra keywords: %s' % kargs)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

请留意：关键字参数
`tkinter` 实际上是 `Python` 中的标准 `GUI API`
该模块在 `2.X` 中的名称是 `Tkinter`
`GUI` 组件创建的时候有很多选项。
关键字参数的好处：不用根据位置列举出所有可能的选项，能让我们从中选择。

18.6 本章小结

18.7 本章习题

19. 函数的高级话题

`lambda` 在 `GUI` 中很常用。

19.1 函数设计概念

如何将任务分解为更有针对性的函数（内聚性）
函数将如何通信（耦合性）
`how to decompose a task into purposeful functions (known as cohesion), how your functions should communicate (called coupling)`

- 耦合性：在输入时使用参数，输出时使用 `return` 语句
- 耦合性：只在真正必要的情况下使用全局变量
- 耦合性：不要改变可变类型的参数，除非调用者希望这样做
  会导致调用者和被调用者之间的强耦合性，这种耦合性会导致函数过于局限和不友好。
- 内聚性：每一个函数都应该有一个单一的、统一的目标。
  每个函数中都应该做一件事：这件事可以用一个简单的说明句来总结。

  > 也就是说可以写在文档字符串里面了。

- 大小：每一个函数应该相对较小。
  一个过长或者有着深层嵌套的函数往往就是设计缺陷的征兆
- 耦合性：避免直接改变其他模块文件中的变量。
  在文件间改变变量会导致模块文件间的耦合性。会让模块难以理解和重用。
  你应该尽可能通过函数来访问，而不是直接使用赋值语句。

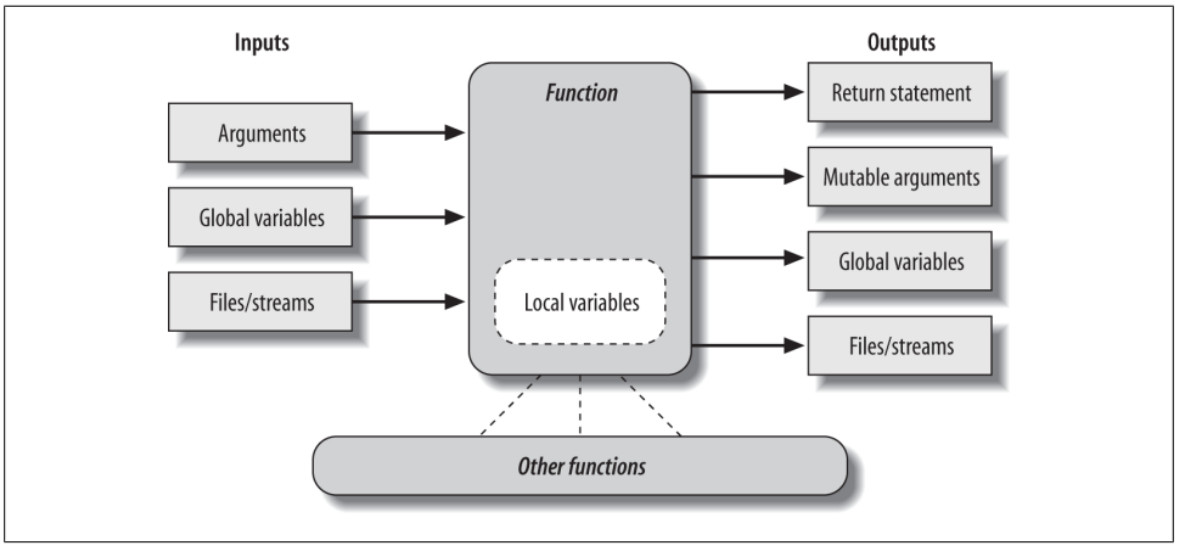


Figure 19-1. Function execution environment. Functions may obtain input and produce output in a variety of ways, though functions are usually easier to understand and maintain if you use arguments for input and return statements and anticipated mutable argument changes for output. In Python 3.X only, outputs may also take the form of declared nonlocal names that exist in an enclosing function scope.

优秀的函数设计者倾向于只使用参数作为输入，以及只使用 `return` 语句作为输出。

如果不使用类，全局变量通常是模块中函数保持调用中单副本状态的最直接的方式。

> 不使用类，指的应该是单例模式？

19.2 递归函数

递归函数：直接或间接地调用自身以进行循环的函数。
递归允许程序遍历拥有任意的、不可预知构型和深度的结构：

- 计划旅行路线
- 分析语言
- 使用爬虫在网页上爬链接

用递归求和

```
>>> def mysum(L):
        print(L) # Trace recursive levels
        if not L: # L shorter at each level
            return 0
        else:
            return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
```

```
[3, 4, 5]
[4, 5]
[5]
[]
15
```

编码替代方案

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:]) # Use ternary expression

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Any type, assum
e one

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest) # Use 3.X ext seq a
ssign
```

- 后两者如果传入空列表会失败，但是它们作用于支持 `+` 操作的任何对象类型，而不只是数字
- 后两者在输入单个字符串时也有效，因为字符串是单个字符的序列
- 第三种可以用于任何可迭代对象（比如打开的文件），而前两者不行，因为他们用到了索引。

间接递归

一个函数调用另一个函数，后者反过来调用其调用者。

```
>>> def mysum(L):
        if not L: return 0
        return nonempty(L)           # Call a function that calls me

>>> def nonempty(L):
        return L[0] + mysum(L[1:])   # Indirectly recursive

>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

循环语句 vs 递归

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
        sum += L[0]
        L = L[1:]

>>> sum
15
```

`for` 版本可以自动迭代：

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x

>>> sum
15
```

有了循环语句，我们不需要在调用栈上为每次迭代都保留一个局部作用域的副本，并避免一般的函数调用相关的开销。

处理任意结构
递归能够遍历任意形状的结构。

```
[1, [2, [3, 4], 5], 6, [7, 8]]  # Arbitrarily nested sublists
```

此时循环不好用：

```
# file sumtree.py
def sumtree(L):
    tot = 0
    for x in L: # For each item at this level
        if not isinstance(x, list):
            tot += x # Add numbers directly
        else:
            tot += sumtree(x) # Recur for sublists
    return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]] # Arbitrary nesting
print(sumtree(L))
```

递归 vs 队列和栈
在内部，`Python` 通过在每一次递归调用时把信息压入调用栈顶来实现递归，因此它记住了在什么地方返回以及在什么地方稍后继续。
不使用递归调用而实现递归风格的过程式编程一般来讲也是可能的，你可使用自己的显式的栈或队列追踪剩余步骤。

采用广度优先的方式遍历了列表中的各层。使用了先进先出的队列。

```
def sumtree(L): # Breadth-first, explicit queue
    tot = 0
    items = list(L) # Start with copy of top level
    while items:
        front = items.pop(0) # Fetch/delete front item
        if not isinstance(front, list):
            tot += front # Add numbers directly
```

```
            else:
                items.extend(front) # <== Append all in nested list
        return tot
```

为了进一步模拟递归调用版本的遍历，我们可以更改它来执行深度优先遍历，这需要在列表的开头简单地增加嵌套列表的内容，形成一个后进先出的栈：

```
def sumtree(L): # Depth-first, explicit stack
    tot = 0
    items = list(L) # Start with copy of top level
    while items:
        front = items.pop(0) # Fetch/delete front item
        if not isinstance(front, list):
            tot += front # Add numbers directly
        else:
            items[:0] = front # <== Prepend all in nested list
    return tot
```

循环、路径和限制

如果数据是一个有环图，这两种方案都会失败。
为了做得更好，递归调用版本可以简单地保存并传递一个已访问状态的集合、字典或列表，从而在运行时检查重复。

```
if state not in visited:
    visited.add(state) # x.add(state), x[state]=True, or x.append(state)
    ...proceed...
```

非递归替代版本：

```
visited.add(front)
...proceed...
items.extend([x for x in front if x not in visited])
```

如果需要记录每一次跟随的状态的完整路径，非递归版本更好用。

标准 `Python` 限制了运行时调用栈的深度来捕获无限递归的错误。
可以使用 `sys` 模块扩大上限。

```
>>> sys.getrecursionlimit() # 1000 calls deep default
1000
>>> sys.setrecursionlimit(10000) # Allow deeper nesting
>>> help(sys.setrecursionlimit) # Read more about it
```

最大值是依赖于平台的。

有时候需要意识到程序中无意隐含的递归的潜在性。
比如 `__setattr__`、`__getattribute__` 甚至于 `__repr__`，如果使用不正确，都会导致无限递归。

19.3 函数对象：属性和注解

`Python` 中的函数是个不折不扣的对象，其本身全部存储在内存块中。
它们可以在程序中自由地传递以及间接调用，也支持与调用几乎无关的操作，例如属性和注解。

函数间接调用："一等"对象
函数可以赋值给其他的名称，传递给其他函数，嵌入到数据结构中，从一个函数返回给另一个函数。同时，函数对象支持一个特殊操作，它们可以通过参数列表调用。

这通常被称为一等对象模型 `first-class object model`，在 `Python` 中是普遍存在的，也是函数式编程的一个必要部分。
因为函数式编程的主旨是建立在函数调用概念的基础上，所以函数必须被当做数据进行对待。
在 `def` 运行之后，函数名仅仅是一个对象的引用。

```
>>> def echo(message):      # Name echo assigned to function object
        print(message)
>>> echo('Direct call')     # Call object through original name
Direct call

>>> x = echo                # Now x references the function too
>>> x('Indirect call!')     # Call object through name by adding ()
Indirect call!

>>> def indirect(func, arg):
        func(arg)         # Call the passed-in object by adding ()
>>> indirect(echo, 'Argument call!') # Pass the function to another funct
ion
Argument call!

>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
func(arg)  # Call functions embedded in containers

Spam!
Ham!

>>> def make(label):  # Make a function but don't call it
        def echo(message):
            print(label + ':' + message)
        return echo

>>> F = make('Spam')  # Label in enclosing scope is retained
>>> F('Ham!')         # Call the function that make returned
```

```
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

函数自省

由于函数是对象，我们可以用常规的对象工具来处理函数。

调用表达式只是定义在函数对象上工作的一个操作。

`call expression is just one operation defined to work on function objects.`

通用的检查它们的属性：

```
>>> func.__name__
'func'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted: 34 total...
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

自省工具可以用来探索实现细节。

函数附加了代码对象 `code objects`，代码对象提供了诸如函数局部变量和参数等方面的细节。

```
>>> func.__code__
<code object func at 0x00000000021A6030, file "<stdin>", line 1>

>>> dir(func.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__',
...more omitted: 37 total...
'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lno
tab',
'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']

>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1
```

工具编写者可以利用这些信息来管理函数。

函数属性

`2.1` 之后可以向函数附加任意的用户定义的属性：

```
>>> func
<function func at 0x0000000000296A1E0>
>>> func.count = 0
>>> func.count += 1
```

```
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...and more: in 3.X all others have double underscores so your names
won't clash...
__str__', '__subclasshook__', 'count', 'handles']
```

`3.X` 中，所有函数的内部名称都有前置和末尾的双下划线。
`2.X` 中基本遵循同样的方案，但是也会赋值一些以 `func_X` 开始的名称。
如果你很小心，不以这种方式来命名属性，那么你就可以安全地使用函数的命名空间。就好像它是你自己的命名空间或作用域一样。

```
c:\code> py -3
>>> def f(): pass

>>> dir(f)
...run on your own to see...
>>> len(dir(f))
34
>>> [x for x in dir(f) if not x.startswith('__')]
[]

c:\code> py -2
>>> def f(): pass

>>> dir(f)
...run on your own to see...
>>> len(dir(f))
31
>>> [x for x in dir(f) if not x.startswith('__')]
['func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc',
'func_globals', 'func_name']
```

Python 3.X 中的函数注解
`3.X` 中可以给函数附加注解信息，即与函数的参数和结果相关的用户定义的任意数据。
它本身不做任何事情，完全是可选的，出现的时候只是直接附加到函数对象的 `__annotations__` 属性以供其他工具使用。
比如这些工具会在错误检测的时候使用注解。

函数注解编写在 `def` 头部行，作为与参数和返回值相关的任意表达式。

```
>>> def func(a, b, c):
    return a + b + c
>>> func(1, 2, 3)
6
```

```
>>> def func(a: 'spam', b: (1, 10), c: float) -> int:
return a + b + c
>>> func(1, 2, 3)
6

>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class
'int'>}
```

`Python` 将注解收集到字典中，并将它们赋值给函数对象的 `__annotations__` 属性。

```
>>> def func(a: 'spam', b, c: 99):
        return a + b + c

>>> func(1, 2, 3)
6
>>> func.__annotations__
{'c': 99, 'a': 'spam'}

>>> for arg in func.__annotations__:
    print(arg, '=>', func.__annotations__[arg])

c => 99
a => spam
```

三个注意点

首先，如果编写了注解的话，仍然可以对参数使用默认值。

`:注解` 要出现在 `=默认值` 之前。

```
>>> def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
        return a + b + c

>>> func(1, 2, 3)
6
>>> func() # 4 + 5 + 6 (all defaults)
15
>>> func(1, c=10) # 1 + 5 + 10 (keywords work normally)
16
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class
'int'>}
```

其次，空格都是可选的，看个人喜好。

```
>>> def func(a:'spam'=4, b:(1,10)=5, c:float=6)->int:
        return a + b + c
```

```
>>> func(1, 2) # 1 + 2 + 6
9
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class
'int'>}
```

最后，注解只在 `def` 中有效，`lambda` 表达式无效。

注解的用途
注解可以用作参数类型或值的特定限制，并且较大的 `API` 可能使用这一功能作为注册函数接口信息的方式。
注解还可以作为函数装饰器参数的一种替代方式。
注解是一种需要你发挥想象力来加以利用的工具。

19.4 匿名函数：lambda

`lambda` 是一个来自 `Lisp` 语言的名称，得名自 `lambda` 演算，而 `lambda` 演算是一种符号化逻辑。但在 `Python` 里面它只是一个定义匿名函数的关键字。

lambda 表达式基础
`lambda` 表达式的一般形式是关键字 `lambda` 后面跟上一个或多个参数，之后是一个冒号，再之后是一个表达式。

> 【个人见解】`0` 个参数也可以。

```
lambda argument1, argument2,... argumentN : expression using arguments
```

`lambda` 返回的函数对象用起来跟 `def` 创建并赋值后的函数对象完全一样。

注意

- `lambda` 是一个表达式，而不是语句。
  它可以出现在 `Python` 语法不允许 `def` 出现的地方。
  作为一个表达式，`lambda` 返回一个值（一个新的函数），可以选择性地被赋值给一个变量名。
- `lambda` 的主体是一个单独的表达式，而不是一个代码块。
  `lambda` 的主题简单得就好像放在 `def` 主体的 `return` 语句中的代码一样。
  因此 `lambda` 的功能通常比 `def` 功能要小。

默认参数

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

`lambda` 主体中的代码遵循作用域查找规则

```
>>> def knights():
        title = 'Sir'
        action = (lambda x: title + ' ' + x) # Title in enclosing def sco
pe
        return action # Return a function object

>>> act = knights()
>>> msg = act('robin') # 'robin' passed to x
>>> msg
'Sir robin'

>>> act # act: a function, not its result
<function knights.<locals>.<lambda> at 0x00000000029CA488>
```

`2.2` 之前需要把 `title` 作为默认参数传进去，因为根本没有 `E` 层级。

为什么使用 lambda
通常，`lambda` 起到了函数速写的作用，你总能用 `def` 来替代它们。
它们适用于在行内嵌入一小段可执行代码，会带来更简洁的代码结构。
比如作为回调处理器 `callback handler`。
也用来编写跳转表 `jump table`，就是可以按需执行相应动作的动作列表或字典。

```
L = [lambda x: x ** 2,  # Inline function definition
     lambda x: x ** 3,
     lambda x: x ** 4]  # A list of three callable functions

for f in L:
    print(f(2)) # Prints 4, 8, 16

print(L[0](3)) # Prints 9
```

等价的 `def` 语句需要占用临时性函数名称，可能同其他名称发生冲突。而且函数定义可能位于很远的地方。

> `lambda` 节约命名空间

```
def f1(x): return x ** 2
def f2(x): return x ** 3   # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]           # Reference by name

for f in L:
    print(f(2))                # Prints 4, 8, 16
```

```
print(L[0](3))              # Prints 9
```

多分支 switch 语句：尾声
可以用字典或者其他数据结构构建更多通用的动作表。

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
     'got': (lambda: 2 * 4),
     'one': (lambda: 2 ** 6)}[key]()
8
```

这样的字典与完整的 `if` 相比，是更加通用的多路分支工具。

> 注：上面的省略小括号也可以

可以运行 `eval(funcname)()` 来触发调用，从而不使用分发表字典。
但是这种方法会慢一些，必须编译和运行代码；而且有安全隐患。

`lambda` 也可以在函数调用参数列表里作为行内临时函数的定义，并且该函数不在程序中其他地方使用时很方便。

如何（不）让 Python 代码变得晦涩难懂
`3.X` 中可以在 `lambda` 里面直接编写 `print(X)`，因为它是一个调用表达式，而不是语句。
`2.X` 可以写 `sys.stdout.write(str(X)+'\n')`（当然 `3.X` 也能通用）
如果要嵌套选择逻辑，可以用三元表达式，或者 `and/or` 组合。
如果需要执行循环，可以嵌套 `map` 调用或者列表推导式。

```
>>> import sys
>>> showall = lambda x: list(map(sys.stdout.write, x))  # 3.X: must use l
ist
>>> t = showall(['spam\n', 'toast\n', 'eggs\n'])        # 3.X: can use pr
int
spam
toast
eggs
>>> showall = lambda x: [sys.stdout.write(line) for line in x]
>>> t = showall(('bright\n', 'side\n', 'of\n', 'life\n'))
bright
side
of
life
>>> showall = lambda x: [print(line, end='') for line in x]  # Same: 3.X
 only
>>> showall = lambda x: print(*x, sep='', end='')            # Same: 3.X
 only
```

使用表达式来模拟语句有一个限制：不能直接达到赋值语句的效果。
但是可以通过 `setattr`，命名空间的 `__dict__` 和能在原位置修改的可变对象的方法来替代。

作用域：lambda 也能嵌套
`lambda` 主要受益于嵌套函数作用域查找。
它还能获取任意外层 `lambda` 中的变量名。

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

> 注：写成 `(lambda x: (lambda y: x + y))(99)(4)` 也行

一般要避免写嵌套的 `lambda`。

请留意：lambda 回调

```
import sys
from tkinter import Button, mainloop # Tkinter in 2.X
x = Button(
        text='Press me',
        command=(lambda:sys.stdout.write('Spam\n'))) # 3.X: print()
x.pack()
mainloop() # This may be optional in console mode
```

回调处理器是通过传递一个用 `lambda` 所产生的函数作为 `command` 的关键字参数来注册的。

> 注册回调函数，这个说法有意思，要熟知

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.onPress("spam")))
    def onPress(self, message):
        ...use message...
```

`2.2` 以前，`self` 需要通过默认参数传递

除了匿名函数，那些带有 `__call__` 和绑定方法 `bound methods` 的类对象也经常扮演回调处理的角色。

19.5 函数式编程工具

函数式编程工具，能把函数作用于序列和其他可迭代对象：

- 在一个可迭代对象的各项上调用函数的工具 `map`
- 使用一个测试函数来过滤项的工具 `filter`
- 把函数作用在成对的项上来运行结果的工具 `reduce`

函数式编程兵器库还包括：

- 一等对象模型 `first-class object model`
- 嵌套作用域闭包
- 匿名函数 `lambda`
- 生成器和推导式语法
- 函数装饰器和类装饰器

在可迭代对象上映射函数：`map`
对每一个元素都进行一个操作，并把结果收集起来。

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
        updated.append(x + 10)  # Add 10 to each item

>>> updated
[11, 12, 13, 14]
```

`map` 内置函数

```
>>> def inc(x): return x + 10 # Function to be run

>>> list(map(inc, counters)) # Collect results
[11, 12, 13, 14]
```

`map` 在 `3.X` 中返回一个可迭代对象，而 `2.X` 返回一个列表。

`map` 和 `lambda` 经常配合使用，因为 `map` 接收一个函数并会应用这个函数。

自己实现的 `map`

```
>>> def mymap(func, seq):
        res = []
        for x in seq: res.append(func(x))
        return res
```

使用自己实现的 `map`

```
>>> list(map(inc, [1, 2, 3]))  # Built-in is an iterable
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])      # Ours builds a list (see generators)
[11, 12, 13]
```

`map` 的优点在于，它是内置函数，不用自己编写，还有一些性能上的优势。

`map` 高级用法：对于多个序列，`map` 将一个 `N` 参数的函数用于 `N` 序列。
`pow` 函数每次调用中都用到两个参数：传入 `map` 的每个序列中各取一个。

```
>>> pow(3, 4)                               # 3**4
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4]))  # 1**2, 2**3, 3**4
[1, 8, 81]
```

`map` 调用和列表推导式较为相似：

```
>>> list(map(inc, [1, 2, 3, 4]))
[11, 12, 13, 14]
>>> [inc(x) for x in [1, 2, 3, 4]] # Use () parens to generate items inst
ead
[11, 12, 13, 14]
```

某些情况下，比如映射一个内置函数时，`map` 比列表推导式要快，而且代码要少。
但是，`map` 需要额外的帮助函数或者 `lambda` 表达式。

选择可迭代对象中的元素：filter
`filter`：实现了基于一个测试函数选择可迭代对象的元素
`reduce`：向 “元素对” 应用函数

`3.X` 中 `filter` 和 `range` 需要 `list` 显示结果：

```
>>> list(range(-5, 5)) # An iterable in 3.X
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(filter((lambda x: x > 0), range(-5, 5))) # An iterable in 3.X
[1, 2, 3, 4]
```

`filter`：对于序列或可迭代对象中的元素，如果函数对该元素返回了 `True` 值，这个元素就会被加入到结果中。
与 `map` 一样，它也可以用 `for` 循环来等价，但是它是内置的，简明的，通常也更快。

> 个人理解：意思就是多用内置函数，少用 `for`

```
>>> res = []
>>> for x in range(-5, 5): # The statement equivalent
        if x > 0:
            res.append(x)

>>> res
[1, 2, 3, 4]
```

还有一点与 `map` 一样，`filter` 可以用列表推导式或者生成器表达式来模拟。
生成器表达式推迟结果的产生。

```
>>> [x for x in range(-5, 5) if x > 0] # Use () to generate items
[1, 2, 3, 4]
```

合并可迭代对象中的元素：reduce
`reduce` 在 `2.X` 中是一个内置函数，但在 `3.X` 中位于 `functools` 模块中。
它接收一个可迭代对象，返回一个单独的结果。

```
>>> from functools import reduce # Import in 3.X, not in 2.X
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
```

等价 `for` 循环：

```
>>> L = [1,2,3,4]
>>> res = L[0]
>>> for x in L[1:]:
        res = res + x
>>> res
10
```

自己实现 `reduce`：

```
>>> def myreduce(function, sequence):
    tally = sequence[0]
    for next in sequence[1:]:
        tally = function(tally, next)
    return tally

>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
```

> `tally` 积分表，计数器的意思。

`reduce` 还允许将一个可选的第三个参数放置与序列的各项之前，而且当序列为空时，充当默认的结果。

```
>>> reduce(lambda x, y: x + y, [], 666)
666
>>> reduce(lambda x, y: x + y, [1, 2], 666)
669
>>> reduce(lambda x, y: x + y, [1, 2], initial=666)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() takes no keyword arguments
```

> 第三个参数只能以位置参数的形式指定。
> 我就好奇这是怎么办到的，这样的行为好奇怪啊，居然不能通过关键字指定参数。

内置的 `operator` 模块对函数式工具来说，使用起来很方便：

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6]) # Function-based +
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

19.6 本章小结

19.7 本章习题

1. `lambda` 表达式和 `def` 语句有什么关系？
   `lambda` 和 `def` 都会创建函数对象，以便之后调用。
   因为 `lambda` 是表达式，它会返回一个函数对象，而不是将这个对象赋值给一个名称，同时它可以嵌入到 `def` 语法无法出现的地方。`lambda` 只允许单个隐式的返回值表达式。因为它不支持语句代码块。因此不适用于较大的函数。
2. 为什么要使用 `lambda` ？
   `lambda` 允许我们在行内嵌入小段可执行代码，推迟其执行，并以默认参数和外层作用域变量的形式为其提供状态。当这一小段代码在其他地方不会被用到时，很方便。
   它们通常出现在 `GUi` 这样基于回调的程序中，并且与 `map` 和 `filter` 这些期待一个处理函数的函数式工具密切相关。
   我们总是可以编写一段 `def` 来替代它。
3. 什么是函数注解，如何使用它们？
   函数注解在 `3.X` 中可用，并且是对函数参数及结果的语法上的修饰，它会被收集到一个字典

中并赋值给函数的 `__annotations__` 属性。
`Python` 在这些注解上没有放置语义含义，而是直接将其包装，以供其他工具潜在的使用。

4. 递归函数
   递归经常被使用显式栈或队列的代码来模拟和替代，后者能获得对遍历过程的更多控制。
5. 说出三种或三种以上函数同调用者通信结果的方式。
   函数可以通过 `return` 语句、改变传入的可变类型的参数以及设置全局变量来传回结果。
   还可以同诸如文件和套接字这样的系统设备结果通信。

---

20. 推导和生成

生成器函数和生成器表达式：是用户定义的按需生成结果的方法。

20.1 列表推导与函数式编程工具

`Python` 拥有一系列具有函数式本质的工具：闭包、生成器、 `lambda` 表达式、推导式、映射、装饰器、函数对象以及更多。

> 这里的映射指的就是 `map` 、 `filter` 、 `reduce` 等等把操作映射到元素的动作吧！

`map` 和 `filter` 借鉴自 `Lisp` 语言。作用是将操作映射到可迭代对象并收集最终的结果。
这是 `Python` 编程中一种相当常见的任务， `Python` 最终产生了一种新的表达式——列表推导。
列表推导最初收到了函数式编程语言 `Haskell` 的启发，发端于 `2.0` 版本。
列表推导把任意一个表达式而不是一个函数应用到一个可迭代对象中的元素。

列表推导 vs map
`ord` 返回一个单个字符的整数编码， `chr` 是逆过程。
如果你的字符恰好属于 `ASCII` 字符集的 `7` 位编码表示范围内的话，这一函数的返回值恰好为该字符的 `ASCII` 编码。
需求：收集整个字符串中所有字符的 `ASCII` 编码：

```
>>> res = []
>>> for x in 'spam':
        res.append(ord(x)) # Manual results collection

>>> res
[115, 112, 97, 109]
```

`map` 版本：

```
>>> res = list(map(ord, 'spam')) # Apply function to sequence (or other)
>>> res
[115, 112, 97, 109]
```

相较于 `map` 把一个函数映射到一个序列或可迭代对象的每一个元素，列表推导把一个表达式映射到一个序列或可迭代对象的每一个元素。
推导式版本：

```
>>> res = [ord(x) for x in 'spam'] # Apply expression to sequence (or oth
er)
>>> res
[115, 112, 97, 109]
```

从语法上说，列表推导包括在一对方括号中，这是为了提醒你它们构造了一个列表。

当我们希望对一个可迭代对象应用一个表达式而非函数的时候，列表推导更方便。

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> list(map((lambda x: x ** 2), range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

使用 filter 增加测试和循环嵌套

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> list(filter((lambda x: x % 2 == 0), range(5)))
[0, 2, 4]

>>> res = []
>>> for x in range(5):
        if x % 2 == 0:
            res.append(x)
>>> res
[0, 2, 4]
```

求平方之列表推导版本：

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

求平方之 `map` 配合 `filter` 版本：

```
>>> list( map((lambda x: x**2), filter((lambda x: x % 2 == 0),
range(10))) )
[0, 4, 16, 36, 64]
```

标准推导语法

`[ expression for target in iterable ]`

通用的列表推导结构如下所示：

```
[ expression for target1 in iterable1 if condition1
              for target2 in iterable2 if condition2 ...
              for targetN in iterableN if conditionN ]
```

每一个 `for` 分句都能带有一个 `if` 筛选器。

> 其实能带多个，`if cond1 if cond2`，默认 `cond1` 和 `cond2` 是 `and` 关系。

需求：组合 `0-4` 的偶数和 `0-4` 的奇数。
列表推导版本：

```
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 ==
 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

循环版本：

```
>>> res = []
>>> for x in range(5):
        if x % 2 == 0:
            for y in range(5):
                if y % 2 == 1:
                    res.append((x, y))
>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

`map` 和 `filter` 版本。

> 我不会写。。。

示例：列表推导与矩阵

使用 `Python` 编写矩阵（也称为多维数组）的一个基本的方法就是使用嵌套的列表结构。
使用列表推导选出第二列。

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

>>> [row[1] for row in M] # Column 2
[2, 5, 8]
```

```
>>> [M[row][1] for row in (0, 1, 2)] # Using offsets
[2, 5, 8]
```

选出对角线和副对角线：

```
>>> [M[i][i] for i in range(len(M))] # Diagonals
[1, 5, 9]
>>> [M[i][len(M)-1-i] for i in range(len(M))]
[3, 5, 7]
```

原位置修改矩阵：

```
>>> L = [[1, 2, 3], [4, 5, 6]]
>>> for i in range(len(L)):
        for j in range(len(L[i])): # Update in place
            L[i][j] += 10

>>> L
[[11, 12, 13], [14, 15, 16]]
```

列表推导原位置修改矩阵：

```
>>> [col + 10 for row in M for col in row] # Assign to M to retain new va
lue
[11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> [[col + 10 for col in row] for row in M]
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

用列表推导来操作多个矩阵中的值

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> N
[[2, 2, 2], [3, 3, 3], [4, 4, 4]]

>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]
[2, 4, 6, 12, 15, 18, 28, 32, 36]

>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

等效 `for` 循环：

```
res = []
```

```
for row in range(3):
    tmp = []
    for col in range(3):
        tmp.append(M[row][col] * N[row][col])
    res.append(tmp)
```

使用 `zip` 配对要相乘的元素：

```
[[col1 * col2 for (col1, col2) in zip(row1, row2)] for (row1, row2) in zi
p(M, N)]

res = []
for (row1, row2) in zip(M, N):
    tmp = []
    for (col1, col2) in zip(row1, row2):
        tmp.append(col1 * col2)
    res.append(tmp)
```

不要滥用列表推导：简单胜于复杂（KISS）
代码的可读性比它的精简性更重要。
如果你需要将代码转换成语句来理解它们，那么你一开始就应当使用语句。

另一方面：性能、简洁性、表现力
`map` 比 `for` 循环要快两倍，而列表推导往往比 `map` 调用还要稍快一些。
一般是由于 `map` 和列表推导在解释器中以 `C` 语言的速度来运行的。
因此比在 `Python` 虚拟机 `PVM` 中以步进运行的 `for` 循环代码要快得多。

列表推导赋予了代码迷人甚至是必要的简洁性，使代码在行数上打折，但其意思却保留下来。
`map` 和列表推导可以替代简单的迭代，尤其是你对程序的运行速度要求较高。
你应该尽量让 `map` 调用和列表推导保持简单，对于更复杂的任务，你应当换用完整的语句。

请留意：列表推导和 `map`
需求：去掉行末换行符

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']

>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']
>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']
>>> list(map((lambda line: line.rstrip()), open('myfile')))
['aaa', 'bbb', 'ccc']
```

> 我的写法更优秀：`list(map(str.rstrip, open('myfile')))`

列表推导可以用来提取列：

```
>>> listoftuple = [('bob', 35, 'mgr'), ('sue', 40, 'dev')]

>>> [age for (name, age, job) in listoftuple]
[35, 40]
>>> list(map((lambda row: row[1]), listoftuple))
[35, 40]
```

> 我的写法：`[age for _, age, _ in listoftuple]`

仅适用于 `2.X` 的：

```
# 2.X only
>>> list(map((lambda (name, age, job): age), listoftuple))
[35, 40]
```

> **注意**：在 `3.X` 中，在函数定义的参数列表中进行元组解包是不被允许的。
> 只能在 `for` 循环中进行元组解包。

除了运行函数和表达式之间的区别，`3.X` 中的 `map` 和列表推导的最大区别是：`map` 是一个可迭代对象，按需产生结果；为了同样达到内存节省与运行时间加快，列表推导必须编写为生成器表达式。

20.2 生成器函数与表达式

- 生成器函数：`2.3` 及以后可以使用。使用常规的 `def` 语句编写，但是使用 `yield` 语句一次返回一个结果，在每次产生结果之间挂起和恢复它们的状态。
- 生成器表达式：`2.4` 及以后可以使用：类似于列表推导，但是它们返回按需产生结果的一个对象，而不是创建一个结果列表。

`Python` 的生成器概念大量借鉴了其他的编程语言，尤其是 `Icon`。

生成器函数：yield vs return
生成器函数：传回一个值并随后从其挂起的地方继续。
生成器函数和常规函数一样，也是用 `def` 语句编写的。
但是，当创建时，它们被特殊地编译成一个支持迭代协议的对象。
并且在调用的时候它们不会返回一个结果，而是返回一个可以出现在任何迭代上下文中的结果生成器。

> 补充：生成器函数中的 `return`
> 参考来源：https://www.cnblogs.com/jessonluo/p/4732565.html

- 在一个生成器中，如果没有 `return` ，则默认执行到函数完毕时返回 `StopIteration` 。
- 如果遇到 `return` ,如果在执行过程中 `return` ，则直接抛出 `StopIteration` 终止迭代。
- 如果在 `return` 后返回一个值，那么这个值为 `StopIteration` 异常的说明，不是程序的返回值。
- 结论：生成器没有办法使用 `return` 来返回值。

```
def test():
    yield 1
    return 666

g = test()
print(next(g))
print(next(g))

"""
1
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(next(g))
StopIteration: 666
"""
```

状态挂起
生成器可以很方便地代替：

- 提前计算一系列值的方式
- 手动保存和在类中恢复状态的方式

生成器函数在挂起时保存的状态包含：

- 代码位置
- 整个局部作用域

因此当函数恢复时，它们的局部变量保持了信息并使其可用。

生成器函数和常规函数之间的不同在于：生成器函数产生 `yield` 一个值，而不是返回 `return` 一个值。
`yield` 语句会挂起该函数并向调用者传回一个值，但同时也保留了足够的状态使函数能从它离开的地方继续。
当继续时，函数在上一次 `yield` 运行后面开始执行。

与迭代协议集成
迭代器对象定义了一个 `__next__` 方法（ `2.X` 为 `next` ），它要么返回迭代的下一项，要么引发 `StopIteration` 异常来终止迭代。
为了支持迭代协议，函数必须包含一条 `yield` 语句，该函数将别编译为生成器：它们不再是普通的函数，而是作为通过特定的迭代协议方法来返回对象的函数。

当调用时，它们返回一个生成器对象，该对象支持用一个自动创建的名为 `__next__` 的方法接口，来开始或恢复执行。

生成器也可以有 `return` 语句，它会结束并退出函数，然后抛出 `StopIteration` 异常。

`next(x)` 内置函数等同于 `3.X` 中的 `x.__next__()` 和 `2.6` 及以后的 `x.next()`。在 `2.6` 之前，直接用 `x.next()`。

> 意思是 `2.6` 以前没有 `next` 内置函数嘛？

生成器函数的应用

```
>>> def gensquares(N):
        for i in range(N):
            yield i ** 2 # Resume here later
```

为了终止值的生成，可以使用一个无值的 `return` 或者让控制权自动脱离函数体。

运行生成器函数会得到一个生成器对象，该对象有 `__next__` 方法。
该方法可以：

- 开始这个函数
- 从它上次 `yield` 值后的地方恢复
- 到最后一个值后，引发 `StopIteration` 异常

如果被迭代的对象不支持迭代协议，那么 `for` 循环将使用索引协议作为替代。
`If the object to be iterated over does not support this protocol, for loops instead use the indexing protocol to iterate.`

生成器将它们自己返回给 `iter` 方法，因为它们直接支持 `next` 方法。

```
>>> y = gensquares(5) # Returns a generator which is its own iterator
>>> iter(y) is y # iter() is not required: a no-op here
True
>>> next(y) # Can run next()immediately
0
```

为什么要使用生成器函数？
`for` 循环版本

```
>>> def buildsquares(n):
        res = []
        for i in range(n): res.append(i ** 2)
        return res

>>> for x in buildsquares(5): print(x, end=' : ')
```

```
0 : 1 : 4 : 9 : 16 :
```

列表推导版本：

```
>>> for x in [n ** 2 for n in range(5)]:
        print(x, end=' : ')

0 : 1 : 4 : 9 : 16 :
```

`map` 版本：

```
>>> for x in map((lambda n: n ** 2), range(5)):
        print(x, end=' : ')

0 : 1 : 4 : 9 : 16 :
```

生成器函数可以用于任何迭代上下文，包括：`tuple` 调用、`enumerate` 和字典推导：

```
>>> def ups(line):
        for sub in line.split(','): # Substring generator
            yield sub.upper()

>>> tuple(ups('aaa,bbb,ccc')) # All iteration contexts
('AAA', 'BBB', 'CCC')

>>> {i: s for (i, s) in enumerate(ups('aaa,bbb,ccc'))}
{0: 'AAA', 1: 'BBB', 2: 'CCC'}
```

生成器表达式：一个能够将函数的灵活性与推导语法的间接性结合到一起的工具。

注解
生成器函数也是一种 “穷人的” 多线程机制。
它们通过把操作拆分到每一次的 `yield` 之间，从而将一个函数的执行交错地插入它的调用者的工作中。
然而生成器并不是线程：程序的运行仍旧在一个单线程的控制内，被显式地交给函数或从函数收走。
线程更加通用，但是生成器易于编写。
因为控制权是显式地在 `yield` 和 `next` 调用时传递的，生成器也同样不是回溯 `backtracking`，它与共行程序 `coroutines` 联系紧密。

> 翻译成协程更好，这是比线程更加轻量的概念。

扩展生成器函数协议：send vs next
`2.5` 生成器函数协议 `generator function protocol` 增加了 `send` 方法。

`send` 方法生成一系列结果的下一个元素，就像 `__next__` 方法一样，但是它也提供了一种调用者与生成器之间进行通信的方式，从而能够影响生成器的操作。

`yield` 现在是一个表达式的形式，而不是语句，尽管可以通过两种方式调用：

- `yield X`
- `A = (yield X)`

表达式必须包括在括号中，除非它是赋值语句右边的唯一一项。
`X = yield Y` 是对的，`X = (yield Y) + 42` 也是对的。

当使用这一额外协议时，可以通过调用 `G.send(value)` 发送一个值给生成器 `G`。
之后恢复生成器代码的执行，并且生成器中的 `yield` 表达式得到了发送给 `send` 函数的值。
如果调用了 `G.__next__()` 或者等价的 `next(G)`，`yield` 则返回 `None`。

```
>>> def gen():
        for i in range(10):
            X = yield i
            print(X)

>>> G = gen()
>>> next(G) # Must call next() first, to start generator
0
>>> G.send(77) # Advance, and send value to yield expression
77
1
>>> G.send(88)
88
2
>>> next(G) # next() and X.__next__() send None
None
3
```

> 一开始要 `G.send(None)` 才不会报错，或者 `next(G)`。

`2.5` 及以后的版本，支持

- `throw(type)` 方法
- `close` 方法：在生成器内部抛出 `GeneratorExit` 异常来彻底终止迭代

> 手动关闭生成器函数，后面的 `next` 调用会直接返回 `StopIteration` 异常。
> `G.throw(type)` 方法是干啥的还不知道

`3.3` 引入了 `yield` 的 `from` 分句，它允许生成器嵌套。

生成器表达式：当可迭代对象遇见推导语法
生成器表达式在圆括号里面，跟元组一样，圆括号通常是可选的。
从语法上来讲，生成器表达式就像一般的列表推导一样，而且也支持所有列表推导的语法。

从功能上来讲，列表推导式等同于 `list` 包含一个生成器表达式。
迭代上下文：`for` 循环，`sum`、`map` 和 `sorted` 等内置函数，列表推导。`any`、`all` 和 `list` 等内置函数。
生成器可以出现在任何可迭代上下文中。

`str.join` 和元组赋值，都为可迭代上下文。

```
>>> ''.join(x.upper() for x in 'aaa,bbb,ccc'.split(','))
'AAABBBCCC

>>> a, b, c = (x + '\n' for x in 'aaa,bbb,ccc'.split(','))
>>> a, c
('aaa\n', 'ccc\n')
```

如果生成器表达式包含在其他的括号之内，比如在函数调用之中，生成器自身的括号就不是必须的。

```
>>> sum(x ** 2 for x in range(4))                    # Parens optional
14
>>> sorted(x ** 2 for x in range(4))                 # Parens optional
[0, 1, 4, 9]
>>> sorted((x ** 2 for x in range(4)), reverse=True)  # Parens required
[9, 4, 1, 0]
```

> 注意：最后一个必须要加括号。如果不加括号会变成这样。
> 生成器表达式必须是唯一的参数，才可以省略括号。

```
>>> sorted(x ** 2 for x in range(4), reverse=True)
  File "<stdin>", line 1
SyntaxError: Generator expression must be parenthesized if not sole argum
ent
```

为什么要使用生成器表达式

- 节省内存
- 与生成器函数一样，把生成结果的过程拆分成更小的时间片。它们会一部分一部分地产生结果，而不是让调用者在一次调用中等待整个集合被创建出来。

生成器表达式在实际运行起来可能比列表推导稍慢一些，所以它们可能只对那些集合非常大的运算或者不能等待全部数据产生的应用来说是最优选择。

生成器表达式 vs map
生成器表达式通常等效于 `3.X` 版本的 `map` 调用，因为它们都是按需生成元素。
不过与列表推导一样，生成器表达式在所应用的操作不是函数调用的时候更易于编写。

```
>>> list(map(abs, (-1, -2, 3, 4))) # Map function on tuple
[1, 2, 3, 4]
>>> list(abs(x) for x in (-1, -2, 3, 4)) # Generator expression
[1, 2, 3, 4]
>>> list(map(lambda x: x * 2, (1, 2, 3, 4))) # Nonfunction case
[2, 4, 6, 8]
>>> list(x * 2 for x in (1, 2, 3, 4)) # Simpler as generator?
[2, 4, 6, 8]
```

注意，下面第一个示例的方括号毫无意义。

```
>>> line = 'aaa,bbb,ccc'
>>> ''.join([x.upper() for x in line.split(',')]) # Makes a pointless lis
t
'AAABBBCCC'

>>> ''.join(x.upper() for x in line.split(',')) # Generates results
'AAABBBCCC'
>>> ''.join(map(str.upper, line.split(','))) # Generates results
'AAABBBCCC'

>>> ''.join(x * 2 for x in line.split(',')) # Simpler as generator?
'aaaaaabbbbbbcccccc'
>>> ''.join(map(lambda x: x * 2, line.split(',')))
'aaaaaabbbbbbcccccc'
```

`map` 和生成器表达式可以任意嵌套：

```
>>> [x * 2 for x in [abs(x) for x in (-1, -2, 3, 4)]] # Nested comprehens
ions
[2, 4, 6, 8]

>>> list(map(lambda x: x * 2, map(abs, (-1, -2, 3, 4)))) # Nested maps
[2, 4, 6, 8]

>>> list(x * 2 for x in (abs(x) for x in (-1, -2, 3, 4))) # Nested genera
tors
[2, 4, 6, 8]
```

生成器嵌套可以任意混合，深度不限。

```
>>> import math
>>> list(map(math.sqrt, (x ** 2 for x in range(4))))  # Nested combinatio
ns
[0.0, 1.0, 2.0, 3.0]

>>> list(map(abs, map(abs, map(abs, (-1, 0, 1)))))  # Nesting gone bad?
```

```
[1, 0, 1]
>>> list(abs(x) for x in (abs(x) for x in (abs(x) for x in (-1, 0, 1))))
[1, 0, 1]
```

尽量使生成器表达式变的简单。
非嵌套方案：

```
>>> list(abs(x) * 2 for x in (-1, -2, 3, 4)) # Unnested equivalents
[2, 4, 6, 8]
>>> list(math.sqrt(x ** 2) for x in range(4)) # Flat is often better
[0.0, 1.0, 2.0, 3.0]
>>> list(abs(x) for x in (-1, 0, 1))
[1, 0, 1]
```

生成器表达式 vs filter
`filter` 在 `3.X` 中与使用了 `if` 分句的生成器表达式等价。

```
>>> line = 'aa bbb c'
>>> ''.join(x for x in line.split() if len(x) > 1) # Generator with 'if'
'aabbb'
>>> ''.join(filter(lambda x: len(x) > 1, line.split())) # Similar to filt
er
'aabbb'

>>> ''.join(x.upper() for x in line.split() if len(x) > 1)
'AABBB'
>>> ''.join(map(str.upper, filter(lambda x: len(x) > 1, line.split())))
'AABBB'
```

生成器表达式更简单些。
正如列表推导，一个生成器表达式总是存在等价的基于语句的形式。

```
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)
'AABBB'

>>> res = ''
>>> for x in line.split(): # Statement equivalent?
        if len(x) > 1: # This is also a join
            res += x.upper()

>>> res
'AABBB'
```

生成器函数 vs 生成器表达式

- 生成器函数
  一个使用了 `yield` 表达式的 `def` 语句是一个生成器函数。当被调用时，它返回一个新的生

成器对象，该对象能自动保存局部作用域和代码执行位置；一个自动创建的 `__iter__` 方法能够返回自身；一个自动创建的 `__next__` 方法用于启动函数或者从上次退出的地方继续执行，并且在结束产生结果的时候引发 `StopIteration` 异常。

- 生成器表达式
  一个包括在圆括号中的列表推导表达式被称为一个生成器表达式。当它运行时，会返回一个新的生成器对象，这个对象带有同样地被自动创建的方法接口和状态记忆功能。

生成器对象，既含有迭代协议所要求的方法，又能够存储生成器的局部变量和代码执行的位置。

生成器是单遍迭代对象
这与某些内置类型的行为不同，这些内置类型支持多个迭代器和多次迭代，并在激活的迭代器中传递并反映它们的原位置修改。

```
>>> L = [1, 2, 3, 4]
>>> I1, I2 = iter(L), iter(L)
>>> next(I1)
1
>>> next(I1)
2
>>> next(I2) # Lists support multiple iterators
1
>>> del L[2:] # Changes reflected in iterators
>>> next(I1)
StopIteration
```

通常，需要支持多次迭代的对象将返回一个辅助类的对象而非它们自身。

Python 3.3 的 yield from 扩展

```
>>> def both(N):
        for i in range(N): yield i
        for i in (x ** 2 for x in range(N)): yield i

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]

>>> def both(N):
        yield from range(N)
        yield from (x ** 2 for x in range(N))

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]
>>> ' : '.join(str(i) for i in both(5))
'0 : 1 : 2 : 3 : 4 : 0 : 1 : 4 : 9 : 16'
```

简单情况下，与 `yield` 的 `for` 循环形式等价。
`yield from generator` 允许委托给子生成器。

> 这里一带而过，讲的不详细
> 这两个资料以后要看看！
> https://www.cnblogs.com/wongbingming/p/9060989.html
> https://www.cnblogs.com/wongbingming/p/9085268.html

内置类型、工具和类中的值生成
字典也带有键的迭代器

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> next(x)
'c'
>>> next(x)
'b'
```

生成器和库工具：目录遍历器

```
>>> import os
>>> for (root, subs, files) in os.walk('.'): # Directory walk generator
        for name in files: # A Python 'find' operation
            if name.startswith('call'):
                print(root, name)

. callables.py
.\dualpkg callables.py
```

> 帮助文档中写的是 `for root, dirs, files in os.walk('路径'):`

位于 `C:\Python33\Lib` 目录下的 `os.py` 中的 `os.walk` 被编写为一个递归的生成器函数。使用了 `yield` 和 `3.3` 版本中作为 `for` 循环替代品的 `yield from`。

```
>>> G = os.walk(r'C:\code\pkg')
>>> iter(G) is G # Single-scan iterator: iter(G) optional
True
>>> I = iter(G)
>>> next(I)
('C:\\code\\pkg', ['__pycache__'], ['eggs.py', 'eggs.pyc', 'main.py', ...
etc...])
>>> next(I)
('C:\\code\\pkg\\__pycache__', [], ['eggs.cpython-33.pyc', ...etc...])
>>> next(I)
StopIteration
```

生成器函数和应用
函数调用时，`*` 可以解包可迭代对象，也就是说 `*` 操作遵循迭代协议。

```
>>> def f(a, b, c): print('%s, %s, and %s' % (a, b, c))

>>> f(0, 1, 2) # Normal positionals
0, 1, and 2
>>> f(*range(3)) # Unpack range values: iterable in 3.X
0, 1, and 2
>>> f(*(i for i in range(3))) # Unpack generator expression values
0, 1, and 2

>>> D = dict(a='Bob', b='dev', c=40.5); D
{'b': 'dev', 'c': 40.5, 'a': 'Bob'}
>>> f(a='Bob', b='dev', c=40.5) # Normal keywords
Bob, dev, and 40.5
>>> f(**D) # Unpack dict: key=value
Bob, dev, and 40.5
>>> f(*D) # Unpack keys iterator
b, c, and a
>>> f(*D.values()) # Unpack view iterator: iterable in 3.X
dev, 40.5, and Bob

>>> for x in 'spam': print(x.upper(), end=' ')
S P A M
>>> list(print(x.upper(), end=' ') for x in 'spam')
S P A M [None, None, None, None]
>>> print(*(x.upper() for x in 'spam'))
S P A M
```

预习：类中用户定义的可迭代对象

遵守迭代协议的类：定义了特殊方法 `__iter__`，它由内置的 `iter` 函数调用，并将返回一个对象，该对象有一个 `__next__` 方法，该方法由 `next` 内置函数调用。

另一方面，一个用户定义的可迭代类方法有时可以使用 `yield` 将它们自身变成迭代器。
使用 `__getitem__` 索引方法作为对迭代退而求其次的选项也是可以的，尽管不像 `__iter__` 和 `__next__` 那样灵活，但对于编写序列而言却有优势。

通过编写方法，类可以使迭代行为更加显式。

实例：生成乱序序列