

4. 更多控制流工具

除了刚介绍的 [while](#) 语句，Python 还用了一些别的。我们将在本章中遇到它们。

4.1. if 语句

最让人耳熟能详的语句应当是 [if](#) 语句：

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
更多
```

可有零个或多个 [elif](#) 部分，[else](#) 部分也是可选的。关键字 'elif' 是 'else if' 的缩写，用于避免过多的缩进。if ... elif ... elif ... 序列可以当作其它语言中 switch 或 case 语句的替代品。

如果是把一个值与多个常量进行比较，或者检查特定类型或属性，[match](#) 语句更有用。详见 [match 语句](#)。

4.2. for 语句

Python 的 [for](#) 语句与 C 或 Pascal 中的不同。Python 的 for 语句不迭代算术递增数值（如 Pascal），或是给予用户定义迭代步骤和结束条件的能力（如 C），而是在列表或字符串等任意序列的元素上迭代，按它们在序列中出现的顺序。例如（这不是有意要暗指什么）：

```
>>> # 度量一些字符串:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

很难正确地在迭代多项集的同时修改多项集的内容。更简单的方法是迭代多项集的副本或者创建新的多项集：

```
# 创建示例多项集
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}
```

```
# 策略：迭代一个副本
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# 策略：创建一个新多项集
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. [range\(\)](#) 函数

内置函数 [range\(\)](#) 用于生成等差数列：

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

生成的序列绝不会包括给定的终止值；`range(10)` 生成 10 个值——长度为 10 的序列的所有合法索引。`range` 可以不从 0 开始，且可以按给定的步长递增（即使是负数步长）：

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

要按索引迭代序列，可以组合使用 [range\(\)](#) 和 [len\(\)](#)：

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

不过大多数情况下 [enumerate\(\)](#) 函数很方便，详见 [循环的技巧](#)。

如果直接打印一个 `range` 会发生意想不到的事情：

```
>>> range(10)
range(0, 10)
```

[range\(\)](#) 返回的对象在很多方面和列表的行为一样，但其实它和列表不一样。该对象只有在被迭代时才一个一个地返回所期望的列表项，并没有真正生成过一个含有全部项的列表，从而节省了空间。

这种对象称为可迭代对象 [iterable](#)，适合作为需要获取一系列值的函数或程序构件的参数。[for](#) 语句就是这样的程序构件；以可迭代对象作为参数的函数例如 [sum\(\)](#)：

```
>>> sum(range(4)) # 0 + 1 + 2 + 3  
6
```

之后我们会看到更多返回可迭代对象，或以可迭代对象作为参数的函数。在 [数据结构](#) 这一章中，我们将讨论 [list\(\)](#) 的更多细节。

4.4. break 和 continue 语句

[break](#) 语句将跳出最近的一层 [for](#) 或 [while](#) 循环：

```
>>> for n in range(2, 10):  
...     for x in range(2, n):  
...         if n % x == 0:  
...             print(f"{n} equals {x} * {n//x}")  
...             break  
...  
4 equals 2 * 2  
6 equals 2 * 3  
8 equals 2 * 4  
9 equals 3 * 3
```

[continue](#) 语句将继续执行循环的下一次迭代：

```
>>> for num in range(2, 10):  
...     if num % 2 == 0:  
...         print(f"Found an even number {num}")  
...         continue  
...     print(f"Found an odd number {num}")  
...  
Found an even number 2  
Found an odd number 3  
Found an even number 4  
Found an odd number 5  
Found an even number 6  
Found an odd number 7  
Found an even number 8  
Found an odd number 9
```

4.5. 循环的 else 子句

在 [for](#) 或 [while](#) 循环中 [break](#) 语句可能对应一个 [else](#) 子句。如果循环在未执行 [break](#) 的情况下结束，[else](#) 子句将会执行。

在 [for](#) 循环中，[else](#) 子句会在循环结束其他最后一次迭代之后，即未执行 [break](#) 的情况下被执行。

在 `while` 循环中，它会在循环条件变为假值后执行。

在这两类循环中，当在循环被 `break` 终结时 `else` 子句 **不会** 被执行。当然，其他提前结束循环的方式，如 `return` 或是引发异常，也会跳过 `else` 子句的执行。

下面的搜索质数的 `for` 循环就是一个例子：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # 循环到底未找到一个因数
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(对，这是正确的代码。仔细看：其中 `else` 子句属于 `for` 循环，而 **不属于** `if` 语句。)

分析 `else` 子句的一种方式是想象它对应于循环内的 `if`。当循环执行时，它将运行一系列的 `if/if/if/else`。`if` 位于循环内部，会出现多次。当出现条件为真的情况时，将发生 `break`。如果条件一直不为真，则循环外的 `else` 子句将被执行。

当配合循环使用时，`else` 子句更像是 `try` 语句的 `else` 子句而不像 `if` 语句的相应子句：一个 `try` 语句的 `else` 子句会在未发生异常时运行，而一个循环的 `else` 子句会在未发生 `break` 时运行。有关 `try` 语句和异常的详情，请参阅 [异常的处理](#)。

4.6. pass 语句

`pass` 语句不执行任何动作。语法上需要一个语句，但程序毋需执行任何动作时，可以使用该语句。例如：

```
>>> while True:
...     pass # 无限等待键盘中断 (Ctrl+C)
...
```

这常用于创建一个最小的类：

```
>>> class MyEmptyClass:
...     pass
...
```

`pass` 还可用作函数或条件语句体的占位符，让你保持在更抽象的层次进行思考。`pass` 会被默默地忽略：

```
>>> def initlog(*args):
...     pass # 记得实现这个!
...
```

对于这后一种情况，许多人会使用省略符字面值 `...` 而不是 `pass`。这种用法在 Python 中没有特殊含义，也不是语言定义的一部分（这里你可以使用任意常量表达式），但 `...` 在传统上也被用作占位符号。参见 [省略符对象](#)。

4.7. match 语句

`match` 语句接受一个表达式并把它的值与一个或多个 `case` 块给出的一系列模式进行比较。这表面上像 C、Java 或 JavaScript（以及许多其他程序设计语言）中的 `switch` 语句，但其实它更像 Rust 或 Haskell 中的模式匹配。只有第一个匹配的模式会被执行，并且它还可以提取值的组成部分（序列的元素或对象的属性）赋给变量。如果没有匹配的 `case`，则不执行任何分支。

最简单的形式是将一个主语值与一个或多个字面值进行比较：

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

注意最后一个代码块：“变量名”`_` 被作为 [通配符](#) 并必定会匹配成功。

你可以用 `|`（“或”）将多个字面值组合到一个模式中：

```
case 401 | 403 | 404:
    return "Not allowed"
```

形如解包赋值的模式可被用于绑定变量：

```
# point 是一个 (x, y) 元组
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

请仔细学习此代码！第一个模式有两个字面值，可视为前述字面值模式的扩展。接下来的两个模式结合了一个字面值和一个变量，变量 [绑定了](#) 来自主语（`point`）的一个值。第四个模式捕获了两个

值，使其在概念上与解包赋值 `(x, y) = point` 类似。

如果用类组织数据，可以用“类名后接一个参数列表”这种很像构造器的形式，把属性捕获到变量里：

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

一些内置类（如 `dataclass`）为属性提供了一个顺序，此时，可以使用位置参数。自定义类可通过在类中设置特殊属性 `__match_args__`，为属性指定其在模式中对应的位置。若设为 `("x", "y")`，则以下模式相互等价（且都把属性 `y` 绑定到变量 `var`）：

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

建议这样来阅读一个模式——通过将其视为赋值语句等号左边的一种扩展形式，来理解各个变量被设为何值。`match` 语句只会为单一的名称（如上面的 `var`）赋值，而不会赋值给带点号的名称（如 `foo.bar`）、属性名（如上面的 `x=` 和 `y=`）和类名（是通过其后的 `"..."` 来识别的，如上面的 `Point`）。

模式可以任意嵌套。举例来说，如果我们有一个由 `Point` 组成的列表，且 `Point` 添加了 `__match_args__` 时，我们可以这样来匹配它：

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
```

```
case _:
    print("Something else")
```

我们可以为模式添加 `if` 作为守卫子句。如果守卫子句的值为假，那么 `match` 会继续尝试匹配下一个 `case` 块。注意是先将值捕获，再对守卫子句求值：

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

该语句的一些其它关键特性：

- 与解包赋值类似，元组和列表模式具有完全相同的含义并且实际上都能匹配任意序列，区别是它们不能匹配迭代器或字符串。
- 序列模式支持扩展解包：`[x, y, *rest]` 和 `(x, y, *rest)` 和相应的解包赋值做的事是一样的。接在 `*` 后的名称也可以为 `_`，所以 `(x, y, *_)` 匹配含至少两项的序列，而不必绑定剩余的项。
- 映射模式：`{ "bandwidth": b, "latency": l }` 从字典中捕获 `"bandwidth"` 和 `"latency"` 的值。额外的键会被忽略，这一点与序列模式不同。`**rest` 这样的解包也支持。（但 `**_` 将会是冗余的，故不允许使用。）
- 使用 `as` 关键字可以捕获子模式：

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

将把输入中的第二个元素捕获为 `p2`（只要输入是包含两个点的序列）

- 大多数字面值是按相等性比较的，但是单例对象 `True`、`False` 和 `None` 则是按 `id` 比较的。
- 模式可以使用具名常量。它们必须作为带点号的名称出现，以防止它们被解释为用于捕获的变量：

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green':"))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

更详细的说明和更多示例，可参阅以教程格式撰写的 [PEP 636](#)。

4.8. 定义函数

下列代码创建一个可以输出限定数值内的斐波那契数列函数：

```
>>> def fib(n):      # 打印小于 n 的斐波那契数列
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # 现在调用我们刚定义的函数:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

定义函数使用关键字 `def`，后跟函数名与括号内的形参列表。函数语句从下一行开始，并且必须缩进。

函数内的第一条语句是字符串时，该字符串就是文档字符串，也称为 *docstring*，详见 [文档字符串](#)。利用文档字符串可以自动生成在线文档或打印版文档，还可以让开发者在浏览代码时直接查阅文档；Python 开发者最好养成在代码中加入文档字符串的好习惯。

函数在执行时使用函数局部变量符号表，所有函数变量赋值都存在局部符号表中；引用变量时，首先，在局部符号表里查找变量，然后，是外层函数局部符号表，再是全局符号表，最后是内置名称符号表。因此，尽管可以引用全局变量和外层函数的变量，但最好不要在函数内直接赋值（除非是 `global` 语句定义的全局变量，或 `nonlocal` 语句定义的外层函数变量）。

在调用函数时会将实际参数（实参）引入到被调用函数的局部符号表中；因此，实参是使用 [按值调用](#) 来传递的（其中的 值始终是对象的 引用而不是对象的值）。[\[1\]](#) 当一个函数调用另外一个函数时，会为该调用创建一个新的局部符号表。

函数定义在当前符号表中把函数名与函数对象关联在一起。解释器把函数名指向的对象作为用户自定义函数。还可以使用其他名称指向同一个函数对象，并访问该函数：

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

如果你用过其他语言，你可能会认为 `fib` 不是函数而是一个过程，因为它没有返回值。事实上，即使没有 `return` 语句的函数也有返回值，尽管这个值可能相当无聊。这个值被称为 `None`（是一个内置名称）。通常解释器会屏蔽单独的返回值 `None`。如果你确有需要可以使用 `print()` 查看它：

```
>>> fib(0)
>>> print(fib(0))
None
```

编写不直接输出斐波那契数列运算结果，而是返回运算结果列表的函数也非常简单：

```
>>> def fib2(n): # 返回斐波那契数组直到 n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # 见下
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # 调用它
>>> f100                # 输出结果
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

本例也新引入了一些 Python 功能：

- `return` 语句返回函数的值。`return` 语句不带表达式参数时，返回 `None`。函数执行完毕退出也返回 `None`。
- 语句 `result.append(a)` 调用了列表对象 `result` 的 `append` 方法。方法是‘从属于’对象的函数，其名称为 `obj.methodname`，其中 `obj` 是某个对象（可以是一个表达式），`methodname` 是由对象的类型定义的方法名称。不同的类型定义了不同的方法。不同的类型的方法可以使用相同的名称而不会产生歧义。（使用 [类](#) 可以定义自己的对象类型和方法，参见 [类](#)。）在示例中显示的方法 `append()` 是由列表对象定义的；它会在列表的末尾添加一个新元素。在本例中它等同于 `result = result + [a]`，但效率更高。

4.9. 函数定义详解

函数定义支持可变数量的参数。这里列出三种可以组合使用的形式。

4.9.1. 默认值参数

为参数指定默认值是非常有用的方式。调用函数时，可以使用比定义时更少的参数，例如：

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
    print(reminder)
```

该函数可以用以下方式调用：

- 只给出必选实参：`ask_ok('Do you really want to quit?')`
- 给出一个可选实参：`ask_ok('OK to overwrite the file?', 2)`
- 给出所有实参：`ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

本例还使用了关键字 `in`，用于确认序列中是否包含某个值。

默认值在 定义作用域里的函数定义中求值，所以：

```
i = 5  
  
def f(arg=i):  
    print(arg)  
  
i = 6  
f()
```

上例输出的是 5。

重要警告： 默认值只计算一次。默认值为列表、字典或类实例等可变对象时，会产生与该规则不同的结果。例如，下面的函数会累积后续调用时传递的参数：

```
def f(a, L=[]):  
    L.append(a)  
    return L  
  
print(f(1))  
print(f(2))  
print(f(3))
```

输出结果如下：

```
[1]  
[1, 2]  
[1, 2, 3]
```

不想在后续调用之间共享默认值时，应以如下方式编写函数：

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

4.9.2. 关键字参数

`kwarg=value` 形式的 [关键字参数](#) 也可以用于调用函数。函数示例如下：

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):  
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")  
    print("-- Lovely plumage, the", type)  
    print("-- It's", state, "!")
```

该函数接受一个必选参数 (`voltage`) 和三个可选参数 (`state`, `action` 和 `type`)。该函数可用下列方式调用：

<code>parrot(1000)</code>	# 1 个位置参数
<code>parrot(voltage=1000)</code>	# 1 个关键字参数

```
parrot(voltage=1000000, action='V00000M')          # 2 个关键字参数
parrot(action='V00000M', voltage=1000000)          # 2 个关键字参数
parrot('a million', 'bereft of life', 'jump')      # 3 个位置参数
parrot('a thousand', state='pushing up the daisies') # 1 个位置参数, 1 个关键字参数
```

以下调用函数的方式都无效：

```
parrot()          # 缺失必需的参数
parrot(voltage=5.0, 'dead') # 关键字参数后存在非关键字参数
parrot(110, voltage=220) # 同一个参数重复的值
parrot(actor='John Cleese') # 未知的关键字参数
```

函数调用时，关键字参数必须跟在位置参数后面。所有传递的关键字参数都必须匹配一个函数接受的参数（比如，`actor` 不是函数 `parrot` 的有效参数），关键字参数的顺序并不重要。这也包括必选参数，（比如，`parrot(voltage=1000)` 也有效）。不能对同一个参数多次赋值，下面就是一个因此限制而失败的例子：

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

最后一个形参为 `**name` 形式时，接收一个字典（详见 [映射类型 --- dict](#)），该字典包含与函数中已定义形参对应之外的所有关键字参数。`**name` 形参可以与 `*name` 形参（下一小节介绍）组合使用（`*name` 必须在 `**name` 前面），`*name` 形参接收一个 [元组](#)，该元组包含形参列表之外的位置参数。例如，可以定义下面这样的函数：

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

该函数可以用如下方式调用：

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

输出结果如下：

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
```

shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

注意，关键字参数在输出结果中的顺序与调用函数时的顺序一致。

4.9.3. 特殊参数

默认情况下，参数可以按位置或显式关键字传递给 Python 函数。为了让代码易读、高效，最好限制参数的传递方式，这样，开发者只需查看函数定义，即可确定参数项是仅按位置、按位置或关键字，还是仅按关键字传递。

函数定义如下：

```
def f(pos1, pos2, /, pos_or_kwds, *, kwds1, kwds2):  
    -----  
    |           |           |  
    |       位置或关键字   |       |  
    |                   |       - 仅限关键字  
    |-----|-----|-----|  
    |       |           |  
    -- 仅限位置
```

`/` 和 `*` 是可选的。这些符号表明形参如何把参数值传递给函数：位置、位置或关键字、关键字。关键字形参也叫作命名形参。

4.9.3.1. 位置或关键字参数

函数定义中未使用 / 和 * 时，参数可以按位置或关键字传递给函数。

4.9.3.2. 仅位置参数

此处再介绍一些细节，特定形参可以标记为 **仅限位置**。仅限位置时，形参的顺序很重要，且这些形参不能用关键字传递。仅限位置形参应放在 / (正斜杠) 前。/ 用于在逻辑上分割仅限位置形参与其它形参。如果函数定义中没有 /，则表示没有仅限位置形参。

/ 后可以是 位置或关键字或 仅限关键字形参。

4.9.3.3. 仅限关键字参数

把形参标记为 **仅限关键字**，表明必须以关键字参数形式传递该形参，应在参数列表中第一个 **仅限关键字** 形参前添加 *。

4.9.3.4. 函数示例

请看下面的函数定义示例，注意 / 和 * 标记：

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
```

```
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

第一个函数定义 `standard_arg` 是最常见的形式，对调用方式没有任何限制，可以按位置也可以按关键字传递参数：

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

第二个函数 `pos_only_arg` 的函数定义中有 `/`，仅限使用位置形参：

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword arg
```

第三个函数 `kwd_only_arg` 如在函数定义中通过 `*` 所指明的那样只允许关键字参数。

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

最后一个函数在同一个函数定义中，使用了全部三种调用惯例：

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword
```

下面的函数定义中，`kwds` 把 `name` 当作键，因此，可能与位置参数 `name` 产生潜在冲突：

```
def foo(name, **kwds):
    return 'name' in kwds
```

调用该函数不可能返回 `True`，因为关键字 `'name'` 总与第一个形参绑定。例如：

```
>>> foo(1, {'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

加上 `/`（仅限位置参数）后，就可以了。此时，函数定义把 `name` 当作位置参数，`'name'` 也可以作为关键字参数的键：

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, {'name': 2})
True
```

换句话说，仅限位置形参的名称可以在 `**kwds` 中使用，而不产生歧义。

4.9.3.5. 小结

以下用例决定哪些形参可以用于函数定义：

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

说明：

- 使用仅限位置形参，可以让用户无法使用形参名。形参名没有实际意义时，强制调用函数的实参顺序时，或同时接收位置形参和关键字时，这种方式很有用。
- 当形参名有实际意义，且显式名称可以让函数定义更易理解时，阻止用户依赖传递实参的位置时，才使用关键字。
- 对于 API，使用仅限位置形参，可以防止未来修改形参名时造成破坏性的 API 变动。

4.9.4. 任意实参列表

调用函数时，使用任意数量的实参是最少见的选项。这些实参包含在元组中（详见 [元组和序列](#)）。在可变数量的实参之前，可能有若干个普通参数：

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

`variadic` 参数用于采集传递给函数的所有剩余参数，因此，它们通常在形参列表的末尾。`*args` 形参后的任何形式参数只能是仅限关键字参数，即只能用作关键字参数，不能用作位置参数：

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
```

```
>>> concat("earth", "mars", "venus", sep=".")  
'earth.mars.venus'
```

4.9.5. 解包实参列表

函数调用要求独立的位置参数，但实参在列表或元组里时，要执行相反的操作。例如，内置的 `range()` 函数要求独立的 `start` 和 `stop` 实参。如果这些参数不是独立的，则要在调用函数时，用 `*` 操作符把实参从列表或元组解包出来：

```
>>> list(range(3, 6))          # 附带两个参数的正常调用  
[3, 4, 5]  
>>> args = [3, 6]  
>>> list(range(*args))        # 附带从一个列表解包的参数的调用  
[3, 4, 5]
```

同样，字典可以用 `**` 操作符传递关键字参数：

```
>>> def parrot(voltage, state='a stiff', action='voom'):  
...     print("-- This parrot wouldn't", action, end=' ')  
...     print("if you put", voltage, "volts through it.", end=' ')  
...     print("E's", state, "!")  
...  
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}  
>>> parrot(**d)  
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin'
```

4.9.6. Lambda 表达式

`lambda` 关键字用于创建小巧的匿名函数。`lambda a, b: a+b` 函数返回两个参数的和。Lambda 函数可用于任何需要函数对象的地方。在语法上，匿名函数只能是单个表达式。在语义上，它只是常规函数定义的语法糖。与嵌套函数定义一样，`lambda` 函数可以引用包含作用域中的变量：

```
>>> def make_incremator(n):  
...     return lambda x: x + n  
...  
>>> f = make_incremator(42)  
>>> f(0)  
42  
>>> f(1)  
43
```

上例用 `lambda` 表达式返回一个函数。另一种用法是传入一个小函数作为参数。例如，`list.sort()` 接受排序键函数 `key`，它可以是一个 `lambda` 函数：

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]  
>>> pairs.sort(key=lambda pair: pair[1])  
>>> pairs  
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.9.7. 文档字符串

以下是文档字符串内容和格式的约定。

第一行应为对象用途的简短摘要。为保持简洁，不要在这里显式说明对象名或类型，因为可通过其他方式获取这些信息（除非该名称碰巧是描述函数操作的动词）。这一行应以大写字母开头，以句点结尾。

文档字符串为多行时，第二行应为空白行，在视觉上将摘要与其余描述分开。后面的行可包含若干段落，描述对象的调用约定、副作用等。

Python 解析器不会删除 Python 中多行字符串字面值的缩进，因此，文档处理工具应在必要时删除缩进。这项操作遵循以下约定：文档字符串第一行之后的第一个非空行决定了整个文档字符串的缩进量（第一行通常与字符串开头的引号相邻，其缩进在字符串中并不明显，因此，不能用第一行的缩进），然后，删除字符串中所有行开头处与此缩进“等价”的空白符。不能有比此缩进更少的行，但如果出现了缩进更少的行，应删除这些行的所有前导空白符。转化制表符后（通常为 8 个空格），应测试空白符的等效性。

下面是多行文档字符串的一个例子：

```
>>> def my_function():
...     """什么都不做，只是记录下来。
...
...     真的，它没有任何作用。
...
...     """
...
...     pass
...
>>> print(my_function.__doc__)
什么都不做，只是记录下来。

真的，它没有任何作用。.
```

4.9.8. 函数注解

[函数注解](#) 是可选的用户自定义函数类型的元数据完整信息（详见 [PEP 3107](#) 和 [PEP 484](#)）。

[标注](#) 是以字典形式存放在函数的 `__annotations__` 属性中的而对函数的其他部分没有影响。形参标注的定义方式是在形参名后加冒号，后面再跟一个将被求值为标注的值的表达式。返回值标注的定义方式是加符号 `->`，后面再跟一个表达式，它位于形参列表和表示 `def` 语句结束的冒号之间。

下面的示例具有加了标注的一个必需参数、一个可选参数以及返回值：

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10. 小插曲：编码风格

现在你将要写更长，更复杂的 Python 代码，是时候讨论一下 [代码风格](#) 了。大多数语言都能以不同的风格被编写（或更准确地说，被格式化）；有些比其他的更具有可读性。能让其他人轻松阅读你

的代码总是一个好主意，采用一种好的编码风格对此有很大帮助。

Python 项目大多都遵循 [PEP 8](#) 的风格指南；它推行的编码风格易于阅读、赏心悦目。Python 开发者均应抽时间悉心研读；以下是该提案中的核心要点：

- 缩进，用 4 个空格，不要用制表符。

4 个空格是小缩进（更深嵌套）和大缩进（更易阅读）之间的折中方案。制表符会引起混乱，最好别用。

- 换行，一行不超过 79 个字符。

这样换行的小屏阅读体验更好，还便于在大屏显示器上并排阅读多个代码文件。

- 用空行分隔函数和类，及函数内较大的代码块。

- 最好把注释放到单独一行。

- 使用文档字符串。

- 运算符前后、逗号后要用空格，但不要直接在括号内使用：`a = f(1, 2) + g(3, 4)`。

- 类和函数的命名要一致；按惯例，命名类用 `UpperCamelCase`，命名函数与方法用 `lowercase_with_underscores`。命名方法中第一个参数总是用 `self`（类和方法详见 [初探类](#)）。

- 编写用于国际多语环境的代码时，不要用生僻的编码。Python 默认的 UTF-8 或纯 ASCII 可以胜任各种情况。

- 同理，就算多语阅读、维护代码的可能再小，也不要在标识符中使用非 ASCII 字符。

备注

[1] 实际上，[对象引用调用](#)这种说法更好，因为，传递的是可变对象时，调用者能发现被调者做出的任何更改（插入列表的元素）。