Table of Contents

Effective Python

59 Specific Ways to Write Better Python-->> 编写高质量Python代码的59个有效方法

是的，我打算翻译这本书，由于个人能力有限，有些不恰当的地方，大家见谅。

翻译工作正在

下面我简单的介绍一下自己，本科为软件工程，喜欢写博客，喜欢分享知识，CSDN博客专家，CSDN博乐成员。

对Python尤其的喜爱，由于对于繁重的无聊枯燥的事务感到厌烦，于是经常使用Python来开发一些自动化的工具，在开发的过程中也发现自己的代码不管是从效率还是可读性都有很大的问题，在提高的过程中，遇到了Effective Python这本书。

阅读一本好书，就像是跟一位智者在聊天,给人带来的影响是无价的，于是我想尽自己所能的来翻译这本，希望更多的人能从中受益，获得编程带来的快乐。

关于我：

GitHub账号： https://github.com/guoruibiao

CSDN 主页： http://blog.csdn.net/marksinoberg

个人主页： https://guoruibiao.github.io

邮箱： marksinoberg@gmail.com

QQ： 1064319632

如果您也对Effective Python的翻译工作感兴趣，很高兴您能与我取得联系。

Pythonic Thinking

编程语言中有这样的一个成语： `Pythonic` ，它是被它的使用者所定义的。这么多年以来，Python社区已经习惯于使用这个形容词性的单词 `Pythonic` 来描述符合特定的风格的代码。具体而言，不是刻板的或者必须按照解释器要求来执行的风格。在使用这门编程语言以及和其他人一起工作的时候它就已经出现了。Python 程序员更喜欢含义明确的代码；喜欢简单的代码而不是晦涩难懂的代码；并且将代码的可读性最大化。（更多细节，可以在Python的shell环境里面，输入 `import this` ，来欣赏Python的禅意）

一名对其他的编程语言很熟悉的程序员，很有可能使用C++，Java，或者别的他最了解的语言来写Python代码。 而新手程序员则可能正在辽阔的语法中畅游。对每个人来说，了解 `the Pythonic` 这个Python中最普通的理念是很有必要的。这些模式将影响到今后你写的每一个程序。

了解你正在使用的**Python**版本

贯穿本书，示例代码的语法依照的是Python3.4（2014年3月17日发布），同时也提供了以Python2.7（发布于2010年7月3日）为语法参照的例子来高亮二者的区别之处。我的大部分建议也都适应于流行的Python运行时环境： 如CPython，Jython,IronPython,PyPy,等等。

许多电脑会预先安装多版本标准的CPython。 但是，在命令行中默认的Python语义可能不是很清晰。例如我们通常默认将python作为Python2.7的别名，但是其也可以是更早的Python版本，例如Python2.6，Python2.5。为了明确你所使用的Python属于哪一个版本，可以使用 `--version` 来检测。

```
C:\Users\Administrator>python --version
Python 2.7.11
```

相较之下，Python3需要指定运行命令来检测其版本信息。

```
python3 --version
Python3.5.2
```

除此之外，你还可以使用Python的内置模块 `sys` 来检测正在使用过的Python版本。

```
import sys
print(sys.version_info)
print(sys.version)
```

```
# Ipython环境下测试如下
In [1]: import sys

In [2]: print sys.version_info
sys.version_info(major=2, minor=7, micro=11, releaselevel='final
', serial=0)

In [3]: print sys.version
2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:40:30) [MSC v.1500
 64 bit (AMD64)]
```

不管是Python2还是Python3，都被Python社区良好的维护着，在漏洞修复，安全优化和Python2~Python3的过渡之外，Python2已经不再更新了。许多有用的工具的存在正在使得Python3的推广更加的容易。

Python3正在不断的优化和添加新特性，而这些特性将不再添加到Python2。在作者写此书的时候，大部分的常用的开源库已经兼容了Python3。所以“我”强烈建议将Python3作为你下一个项目的开发首选。

备忘录

- 正在使用的两大活跃Python版本：Python2，Python3。
- Python有用多个运行时环境：CPython，Jython,IronPython,PyPy等等。
- 确保你的电脑上运行的Python版本正是你所希望的。
- 再下一个项目中使用Python3，因为其是目前Python社区的主流焦点。

遵循**PEP8** 风格编程风格

Python 增强了题案#8，类似于PEP8的一款格式化Python代码的风格。只要语法有效，你可以随心所欲的编写你的Python代码。但是使用一致的风格的话可以使你的代码更加的平易近人，易读。一个通用的代码风格对于一个巨大的项目中其他的Python程序员之间的合作，影响是很大的。 即使你是唯一一个将会阅读你源代码的人，使用这个风格，也会使你在之后的代码维护上受益。

`PEP8` 包含了很多如何编写Python代码的细节，随着Python语言的迭代其也会不断的更新。在线完整阅读。接下来本书将简单的介绍一点其中比较实用的规则。

空格

在Python中，空格是一个重要的语法。Python 程序员对于整洁空格代码带来的影响尤其的敏感。下面是普适的一些建议。

- 使用空格而不是tab键来缩进。
- 使用4个空格语法 作为每一层缩进的标准。
- 每行代码应尽量的少于79个字符。
- 一个长表达式被分成多行表示的时候要添加额外的4个空格来保持缩进的美观。
- 在一个文件中，函数和类之间应该用两个空白行分隔。
- 在类内部，方法之间应该使用赶一个空白行分隔。
- 不要将空格放在列表索引、函数调用或关键字参数赋值“两侧”，即不要用空格包围此三者。
- 在变量赋值前后 保证一次仅有一个空格被放置。

命名

`PEP 8` 推荐使用为不同的部分起一个独特的名字。这样在阅读源代码的时候就会很容易的辨别属于哪一个类型。

- 函数，变量以及属性应该使用小写，如果有多个单词推荐使用下划线进行分

隔。
- 被保护的属性应该使用前导下划线来声明。
- 私有的属性应该使用两个前导下划线来进行声明。
- 类以及异常信息 应该使用单词首字母大写形式，也就是我们经常使用的驼峰命名法。
- 模块级别的常量应该全部使用大写的形式。
- 类内部的实例方法的应该将 `self` 作为其第一个参数。且 `self` 也是对当前类对象的引用。
- 类方法应该使用 `cls` 来作为其第一个参数。且 `self` 引用自当前类。

表达式和语句

Python之禅中有这样的一句话。大致意思如下：完成一件事，有且仅有一种方式是最好的。 `PEP 8` 尝试在它的指导手册中为表达式和语句编写这样的一个优雅，简单的编程风格。

- 使用内联否定（如 `if a is not b` ） 而不是显示的表达式（如 `if not a is b` ）。
- 不要简单地通过变量的长度（ `if len(somelist) == 0` ）来判断空值。使用隐式的方式如来假设空值的情况（如 `if not somelist` 与 `False` 来进行比较）。
- 上面的第二条也适用于非空值（如 `[1]` ,或者'hi'）。对这些非空值而言 `if somelist` 默认包含隐式的 `True` 。
- 避免将 `if` 语句块， `for` , `while` , `except` 等包含多个语块的表达式写在一行内。为了使代码更易读，应该分割成多行。
- 总是把 `import` 语句写在 `Python` 文件的顶部。
- 当引用一个模块的时候使用绝对的模块名称，而不是与当前模块路径相关的名称。例如要想引入 `bar` 包下面的 `foo` 模块，应该使用 `from bar import foo` 而不是 `import foo` 。
- 如果非要相对的引用，应该使用明确的语法 `from . import foo` 。
- 按照以下规则引入模块：标准库，第三方库，你自己的库。每一个部分内部也应该按照字母顺序来引入。

Note: The Pylint tool 是一个深受欢迎的静态的Python源代码分析器。提供了 `PEP 8` 风格的自动指导以及常见的错误。

备忘录：

- 写代码的时候总是要遵循 `PEP 8` 的风格指导。
- 与较大的Python社区共享一个共同的风格，方便与他人合作。
- 使用一致的风格，方便你今后修改自己的代码。

了解字节，字符串以及unicode# 了解字节，字符串以及unicode之间的区别

在 `Python3` 中，有两种类型的字符代表序列： `bytes(字节)` 和 `str（字符串）` 。字节的实例包含8个原生的比特值，而字符串的实例则是用Unicode字符来堆砌的。

在 `Python2` 中，有两种类型的字符代表序列： `str(字符串)` 和 `unicode(Unicode字符)` 。与 `Python3` 相反，字符串实例代表着原生的8比特值序列，而 `unicode` 则由 `Unicode` 字符堆砌而成。

有许多方法来表示Unicode字符的二进制数据（原生的8比特值 序列）。最常用的编码方式为 `UTF-8` 编码。还有一点很重要，那就是 `Python3` 中的字符串实例和 `Python2` 中的 `unicode` 实例并没有相关联的二进制编码。所以要想将 `Unicode` 字符转换成二进制数据，就必须使用 `encode` 方法，反过来，要想把二进制数据转换成 `Unicode` 字符，就必须使用 `decode` 方法。

当你开始写 `Python` 程序的时候，在接口的最开始位置声明对 `Unicode` 的编码解码的细节很重要。在你的代码中，最核心的部分应使用 `Unicode` 字符类型（ `Python3` 中使用 `str` , `Python2` 中使用 `unicode` ）并且不应该考虑关于字符编码的任何其他方式。本文允许你使用自己喜欢的可替代性的文本编码方式（如 `Latin-1` , `Shift JIS` , `Big5` ），但是应该对你的文本输出编码严格的限定一下（理想的方式是使用 `UTF-8` 编码）。

由于字符类型的不同，导致了Python代码中出现了两种常见的情形的发生。

- 你想操作 `UTF-8` （或者其他的编码方式）编码的8比特值 序列。
- 你想操作没有特定编码的 `Unicode` 字符。 所以你通常会需要两个工具函数来对这两种情况的字符进行转换，以此来确保输入值符合代码所预期的字符类型。

在 `Python3` 中

- 你将需要一个方法，接收 `str` 或者 `bytes` ，总是来返回 `str` 类型的数据。如下：

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str,bytes):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    # str类型的数据
    return value
```

- 同理，我们需要另一个方法，来接收 `str` 或 `bytes` ，总是来返回 `bytes` 类型的数据。

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str,str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    # 字节类型的数据
    return value
```

在 `Python2` 中

- 需要一个方法，来接收str或者unicode类型的数据，总是来返回unicode类型的数据。

```
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str,str):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    # unicode类型的数据
    return value
```

- 同理，需要一个接收str或者unicode类型的数据，总是来返回str类型的数据。

```
def to_str(unicode_or_str):
    if isinstance(unicode_or_str,unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    # str类型的数据
    return value
```

两大陷阱

在 `Python` 中处理原生的8比特值序列以及 `Unicode` 字符的时候，有两大陷阱。

一个是在 `Python2` 中，当一个 `str` 数据仅仅包含7比特的 `ASCII` 码字符的时候， `unicode` 和 `str` 实例看起来是一致的。

- 可以使用‘+’运算符和合并str和unicode。
- 可以使用等价或者不等价运算符来比较str和unicode实例。
- 可以使用unicode来代换 像‘%s’这种字符串中的格式化占位符。

以上行为意味着，如果你的代码中仅仅处理原生的7比特序列，那么便可以不必在意是 `str` 类型的数据还是 `unicode` 类型的数据了。
在 `Python3` 中， `bytes` 和 `str` 实例是不可能等价的，即使是空的字符串也不可能等价。所以你必须谨慎地对正在处理的代码进行字符类别的区分处理。

另一个是在 `Python3` 中，涉及到文件处理的操作（使用内置的 `open` 函数）会默认的以 `UTF-8` 进行编码。而在 `Python2` 中默认采用二进制形式来编码。这也是导致意外事故发生的根源，特别是对于那些更习惯于使用 `Python2` 的程序员而言。

比方说，你想将几个随机的二进制数据写入到一个文件中。在 `Python2` 中，下面的这段代码可以正常的工作，但是在 `Python3` 中却会报错并退出。详细信息如下。

```
def open('/tmp/random.bin','w') as f:
    f.write(os.urandom(10))

>>>
TypeError: must be str,not bytes
```

导致这个异常发生的原因是在 `Python3` 中对于 `open` 函数又新增了一个名为 `encoding` 的参数。此参数默认为 `UTF-8` 。这使得其对于文件的读写操作预期的源为包含了 `Unicode` 字符串的 `str` 实例，而不是包含了二进制数据的字节文件。

为了使得上面的函数正常的工作，我们必须指明被操作的数据是以‘wb’模式打开，而不是简单的‘w’模式。这里，作者介绍了一个在 `Python2` 和 `Python3` 中都通用的方法，详细如下。

```
with open('/tmp/random.bin','wb) as f:
    f.write(os.urandom(10))
```

好了，写文件的问题算是解决了，但是不要忘了还有读文件的问题哦。同样的我们也只需要改变一下读文件的模式即可。即‘r’换成‘rb’。

备忘录：

- 在 `Python3` 中，字节包含的是8个比特值的序列， `str` 是包含 `Unicode` 的字符的串。字节和字符串实例不能同时出现在操作符‘>’ 或者 ‘+’中。
- 在 `Python2` 中， `str` 是包含8个比特值的序列， `unicode` 是包含 `Unicode` 字符的串，二者可以同时出现在只包含7个比特的 `ASCII` 码的运算中。
- 使用工具函数来确保程序输入的数据时程序预期的类型。
- 总是使用‘wb’和‘rb’模式来写文件和读文件。

编写有帮助的函数而不是复杂表达式

`Python` 简练的语法使得实现了复杂逻辑的单行表达式的书写变得容易。例如，你想从一个 `URL` 中解码出查询字符串，这里每一个查询参数代表着一个整形的值：

```
from urllib.parse import parse_qs
my_value = parse_qs('red=5&blue=0&green=',keep_blank_values=True
)
print(repr(my_values))
>>>
{'red': ['5'],'green': [''],'blue': ['0']}
```

查询字符串的一些参数可能含有多个值，有些可能仅有一个值，有些到目前为止可能是空值，还有些可能根本就没有这个参数。在不同的情况下，对结果集字典使用 `get` 方法来处理，将返回不同的对应值。

```
print('Red:',my_values.get('red'))
print('Green:',my_values.get('green'))
print('Opacity:',my_values.get('opacity'))
>>>
Red: ['5']
Green: ['']
Opacity: None
```

当一个参数未被提供或者其值为空的时候为其赋值默认值 `0` 是一个很好的处理方式。或许你更倾向于使用布尔表达式，因为默认为 `0` 从逻辑上看起来，貌似并不能保证所有的情况下都能够适用。

`Python` 的语法使得这个选择变得相当的容易。这的把戏就是空字符串，空列表，零值等都可以被隐式的转换为 `False` 值。因此，接下来的表达式中的子表达式为 `False` 的时候就会自动的被替换为 `or` 运算符后面的子表达式。

```
# 查询字符串为：'red=5&blue=0&green='
red = my_value.get('red', [''])[0] or 0
green = my_value.get('green', [''])[0] or 0
opacity = my_value.get('opacity', [''])[0] or 0
print('Red:%r' % red)
print('Green:%r' % green)
print('pacity:%r' % opacity)
>>>
Red: '5'
Green: 0
Opacity: 0
```

红色判例可以正常的取到值就是因为在 `my_values` 这个结果字典中含有这个值。而且是一个带有一个成员： `string '5'` 的列表 `list` 值。因此其可以被转换为 `True` ，因此 `or` 运算符后面的子表达式不会被执行，于是便不会返回 `0` 值。

绿色判例返回了 `0` 值的原因我们同样可以获悉，在 `my_values` 字典中，对应 `green` 这个 `key` 的 `value` 是一个带有一个空字符串成员的列表。经过隐式转换即为 `False` ，因此 `or` 运算符后面的子表达式得以执行，返回零值。

透明度的这个判例返回了 `0` 值，原因类似。在 `my_values` 字典中对应 `opacity` 这个 `key` 并没有值。所以按照 `get` 方法来运行的话就会返回其表达式的第二个子表达式。默认值就是一个带有一个空字符串成员的列表，于是结果和 `green` 判例自然的就保持一致了。

然而，这个表达式不仅难于阅读，还不能满足所有的情况。你还可能想确保返回的数据为数学的整形值，于是还需要手动的为刚才的表达式包装一个内置的函数，来将原本的字符串数据转换成整形的数据，如下：

```
red = int(my_values.get('red',[''])[0] or 0)
```

现在，你可能感觉代码更加难理解了。有这么多的视觉噪音干扰着我们的视线。这个方法也变得不在平易近人。一个新读者可能要花费好长时间来分隔语段来分析表达式到底做了什么。即使保持事事务精简很好，但是这并不是要把所有的逻辑都写在一行中。

`Python2.5` 新添了 `if/else` 条件语句和三目表达式 来使得代码更简洁，更清晰。

```
red = my_values.get('red',[''])
red = int(red[0]) if red[0] else 0
```

现在好多了吧。对于不太复杂的情况，`if/else` 条件表达式可以使得事情变得更加的美好。但是上面的这个例子并不能完全的用多行的 `if/else` 来替代。如果非要这样做的话，你就会看到事情变得更加的糟糕了。

```
green = my_values.get('gren',[''])
if green[0]:
    green = int(green[0])
else:
    green = 0
```

为了减少重复代码的出现，现将刚才的这段代码封装一下。

```
def get_first_int(values,key,default=0):
    found = values.get(key,[''])
    if found[0]:
        found = int(fount[0])
    else:
        found = default
    return found
```

现在调用这个函数比使用 `or` 或者使用了 `if/else` 的两行的那段代码变得简单多了。

```
green = get_first_int(my_values,'green')
```

当你的表达式开始变得复杂的时候，就是时候考虑一下分隔，重构了。更小的代码块，更多的工具函数。良好的可读性将会给你带来更多。所以，不要让简洁的 `Python` 语法成为你复杂脏乱代码的填充物。

备忘录：

- `Python` 简洁的语法使得编写复杂庞大的单行代码变得更加容易了。

- 把复杂的表达式重构到工具函数中，尤其是出现重复逻辑的地方更应如此。
- 布尔表达式（ `or` 或者 `in` ）的一个更好的替代方案就是使用 `if/else` ，来增强代码的可读性。

知道怎么去分片

`Python` 本身包含了如何将序列分片的语法。分片可以让你以最小的代价访问到子序列的某一项。最简单的使用方式是对 `Python` 内置的数据类型进行分片，如 `list` (列表), `str` (字符串)， `bytes` (字节数组)。最重要的是分片有很强的可实现性。对于实现了 `__getiten__` 和 `__setitem__` 这两个特殊方法的类都可以使用分片机制。（参考第28项：从集合，字符串继承来自定义容器）

简单分片

使用分片也很简单，基本形式是： `somelist[start:end]` ，当然，单独使用时 `start` 和 `end` 必须在集合的下标范围内。

```
a = ['a','b','c','d','e','f'.'g','h']
print('First Four:',a[:4])
print('Last Four:',a[-4:])
print('Middle Two:',a[3:-3])
>>>
First Four: ['a','b','c','d']
Last Four: ['e','f','g','h']
Middle Two: ['d','e']
```

当分片是从头开始的话，下标0可以省略，来减少视觉干扰。 `assert a[:5] == a[0:5]`

同样的，当分片恰好到达尾部的时候，最后一个下标也是可以省略的。 `assert a[5:] == a[5:len(a)]`

分片中使用负数对于从后往前的操作很方便。负数的绝对值就是相对于末尾数据的偏移量。所有的形式对于新手而言也都是清晰的。作者也非常建议使用这些变量。

```
a[:]      # ['a','b','c','d','e','f','g','h']
a[:5]      # ['a','b','c','d','e']
a[:-1]     # ['a','b','c','d','e','f','g']
a[4:]      # ['e','f','g','h']
a[-3:]     # ['f'.'g','h']
a[2:5]     # ['c','d','e']
a[2:-1]     # ['c','d','e','f','g']
a[-3:-1]    # [ 'f','g']
```

分片机制将自动的处理超出集合边界的下标的取值。这也使得建立一个最大长度的输入序列更加的容易。

```
first_twenty_items = a[:20]
last_twenty_items = a[-20:0]
```

对比起来，直接索引到与上边相同的下标会引发一个异常

```
In [1]: a = ['a','b','c','d','e','f','g','h']

In [2]: a[8]
---------------------------------------------------------------------
-----------
IndexError                                Traceback (most recent
 call last)
<ipython-input-2-7749ee2033a3> in <module>()
----> 1 a[8]

IndexError: list index out of range

In [3]: a[:8]
Out[3]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

>>>
IndexError: list index out of range
```

发生这个异常的原因就在于分片机制下数字是指相对于开始位置的 `offset` （偏移量），而单纯的使用下标的话还是需要遵守列表的访问规则的。

> 笔记：使用负数作为分片变量来获取一个较好的数据集的情况还是很少发生的。例如，这样的一个表达式：`somelist[-n:]` 也只是当 `n` 大于 `1` 的时候有效，当 `n` 为 `0` 的时候，`somelist[-0:]` 将返回当前列表的复制结果。

分片不改变原始数据

对一个列表进行分片，返回的仍然是一个列表。原来的列表仍然被维持不变，对返回的列表进行操作，不会影响原来的列表的值。

```
a = ['a','b','c','d','e','f','g','h']
b = a[4:]
print("Before:", b)
b[1] = 99
print("After:",b)
print("Original:",a)
>>>
Before: ['e','f','g','h']
After: ['e',99,'g','h']
Original: ['a','b','c','d','e','f','g','h']
```

赋值

当使用赋值语句的时候，分片将会替换原始列表中特定范围的值。而不像 `Tuple` (元组)赋值那样（如 `a, b = c[:2]` ），切片赋值的长度不必一定要与之相一致。分配片的赋值之前和之后的值将被保存。列表会相应的变大或者缩小来容纳新值。

```
print("Before:",a)
a[2:7] = [99,22,14]
print("After:",a)
>>>
Before: ['a','b','c','d','e','f','g','h']
After: ['a','b',99,22,14,'h']
```

如果省略开头索引和结尾索引，就会获得原始列表的拷贝版本。

```
b = a[:]
assert b ==a and b is not a
```

引用-变化-追随

当为列表赋值的时候省去开头和结尾下标的时候，将会用这个引用来替换整个列表的内容，而不是创建一个新的列表。同时，引用了这个列表的列表的相关内容，也会跟着发生变化。

```
b = a
print("Before:",a)
a[:] = [101,102,103]
assert a is b
print("After:",a)
>>>
Before: ['a','b','c','d','e','f','g','h']
After: [101,102,103]
```

备忘录：

- 避免冗余：列表首尾的下标 `0`，`len(somelist)` 不必再列出。
- 分片机制自动处理越界问题，但是最好在表达边界大小范围是更加的清晰。（如 `a[:20]` 或者 `a[-20:]`）
- 为一个列表赋值将会引起 应用该列表的其他列表相关的内容发生变化，即使他们的长度不相同。

在一个简单的分片中避免使用start,end,sreide等关键字

步幅

除了简单的分片（详见第5项：如何分片），`Python`还有针对步幅的特殊的语法，形如：`somelist[start:end:stride]`。这一语法对于每隔`n`个元素的分片，很有帮助。比如按奇偶来实现分片。

```
a = ['red','orange','yellow','green','blue','purple']
odds = a[::2]
evens = a[1::2]
print(odds)
print(evens)
>>>
['red','yellow','blue']
['orange','green','purple']
```

步幅bug

分片步幅的存在经常会引起一些意想不到的行为或者产生bug。例如，一个常用的把戏是利用步幅为-1来实现字符串的逆序。

```
# 当数据仅仅为ASCII码内数据时工作正常
x = b'mongoose'
y = x[::-1]
print(y)
>>>
b'esoognom'


# 出现Unicode字符的时候就会报错
w = '谢谢'
x = w.encode(utf-8')
y = a[::-1]
z = y.decode('utf-8')
>>>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x9d in posi
tion 0: invalid start byte.
```

负数步幅

当步幅小于-1的时候还会有效吗？思考一下下面的小例子吧。

```
a = ['a','b','c','d','e','f','g','h']
a[::2]        # ['a','c','e','g']
a[::-2]       # ['h','f','d','b']
```

不难得出下面的结论： 步幅为2代表着从下表为0开始，以2个元素为步幅抓取数据；步幅为-2则代表着从列表项的尾部开始，以每两个元素为步幅抓取数据。因此，步幅小于-1，对分片来说仍然是有效的。

按照刚才我们得出的结论，趁热打铁的来做点小测试吧。

```
a = ['a','b','c','d','e','f','g','h']
a[2::2]     # ['c','e','g']
a[-2::-2]    # ['g','e','c','a']
a[-2:2:-2]   # ['g','e'] 尤其注意这里，类似于坐标轴，分片范围是左闭右开，所以2的位置不可达
a[2:2:-2]    # []
```

上面的几个分页语法的小例子可以说是相当的让人迷惑的了。使用三个参数将使得其不易阅读。同时索引依赖于步幅的时候（尤其步幅是负数的情况下）也会变的不再明显。也就是说看起来会很混乱，所以为了解决我们遇到的问题，就得避免和 `start` 与 `end` 下标一起使用。如果非要使用步幅的话，使用正值的步幅并且省略索引下标。如果步幅必须要和索引下标一起出现，考虑使用一个赋值来调幅，一个来分片。

```
a = ['a','b','c','d','e','f','g','h']
b = a[::2]    # ['a','c','e','g']
c = b[1:-1]   # ['c','e']
```

先分片在调幅将会创建一个额外的数据副本。第一个操作应该尽可能的减小分片的大小。如果你的程序不能为两步内的计算时间或者内存占用“买单”的话，考虑使用内置模块中的迭代器 `itertools` 的 `islice` 方法（详见第46项：使用内置的算法和数据结构），这个方法默认不允许负数值的索引下标，以及步幅。

备忘录：

- 在分片中指定 `start` ， `end` , `stride` 会让人感到困惑，难于阅读。
- 尽可能的避免在分片中使用负数值。
- 避免在分片中同时使用 `start` ， `end` ， `stride` ；如果非要使用，考虑两次赋值（一个分片，一个调幅），或者使用内置模块 `itertoolsde` 的 `islice` 方法来进行处理。

使用列表表达式而不是map和filter

列表表达式 VS map

`Python` 从另一个导出一个列表提供了紧凑的语法。这些表达式被称为是“列表表达式”。例如：你想计算一下列表中每个数字的平方，你就可以使用列表表达式来循环的计算每一项的平方值。

```
a = [1,2,3,4,5,6,7,8,9,10]
squares = [x*x for x in a]
print(squares)
>>>
[1,4,9,16,25,36,49,64,81,100]
```

除非你申请一个单参数的函数，列表表达式比内置的 `map` 函数更加的简洁，清晰。因为 `map` 需要创建一个带有视觉干扰的 `lambda` 表达式，来方便计算。

```
squares = map(lambda x: x **2 ,a)
```

列表表达式 VS filter

和 `map` 不同，列表表达式可以让你更简单的过滤整个输入列表，从结果中删除相应的输出。例如：你只想计算可以被2整除的元素的平方。这里你就可以添加一个条件表达式来过滤出符合条件的元素，从而真正的参与到运算。

```
even_squares = [x**2 for x in a if x%2==0]
print(even_squares)
>>>
[4,16,36,64,100]
```

同样的，内置的过滤器 `filter` 配合 `map` 的使用同样可以达到相同的效果，但是可读性将变得很糟糕。

```
alt = map(lambda x: x**2, filter(lambda x: x%2==0,a))
assert even_squares== list(alt)
```

字典和集合有他们自己的一套列表表达式。这使得书写算法的时候导出数据结构更加的简单。

```
chile_rank = {'ghost':1,'habanero':2,'cayenne':3}
rank_dict = {rank:name for name,rank in child_rank.items()}
chile_len_set = {len(name) for name in rank_dict.values()}
print(rand_dict)
print(chile_len_set)
>>>
{1: 'ghost',2: 'habanero',3: 'cayenne'}
{8, 5, 7}
```

备忘录

- 列表表达式比内置的 `map` , `filter` 更加清晰，因为 `map` , `filter` 需要额外的 `lambda` 表达式的支持。
- 列表表达式允许你很容易的跳过某些输入值，而一个 `map` 没有 `filter` 帮助的话就不能完成这一个功能。
- 字典和集合也都支持列表表达式。

在列表表达式中避免使用超过两个的表达式

多层循环

排除基础的用法（详见第7项：使用列表表达式而不是 `map` 或者 `filter` ）,列表表达式也支持多层循环。例如：你想简化一个矩阵（也就是一个列表内是一些其他的列表）到一个大列表中。这里，作者通过内嵌两个表达式成功实现。表达式运行的顺序是由左至右。

```
matrix = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)

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

上面的例子简单，可读而且是条理清晰的多层循环。另一个关于多层循环的合理使用是对于如此两层输入列表的复制。例如你想计算一个两维矩阵的每一个位置上数的平方，虽然这个表达式看起来会因为空列表（[]）的存在略显杂乱，但是代码的可读性却得到了提高。

```
squared = [[ x**2 for x in row] for row in matrix]
print(squared)
>>>
[[1, 4, 9],[16, 25, 36],[49, 64, 81]]
```

如果这个表达式包含另外的一个循环体，此列表表达式将变得特别的长以至于你不得不将其分割成多行，来保持代码的可读性。

```
my_lists = [
    [[1, 2, 3],[4, 5, 6]],
    # ...
]
flat = [ x for sublist in my_lists
          for sublist2 in sublist
          for x in sublist2]

print(flat)
```

从这点来看，多行的列表表达式并不比替代方案少多少代码。这里，作者更加的建议使用正常的循环体语句。因为其比列表表达式更简洁好看一点,也更加易读，易懂。

```
flat = []
for sublist in my_lists:
    for sublist2 in sublist:
        flat.append(sublist2)
```

列表表达式同样支持if条件语句。多个条件语句出现在相同的循环水平中也是一个隐式 `&` 的表达,即同时成立才成立。例如：你只想获得列表中大于4且是偶数的值。那么下面的两个列表表达式是等价的。

```
a = [1,2,3,4,5,6,7,8,9,10]
b = [x for x in a if x> 4 if x%2 ==0]
c = [x for x in a if x > 4 and if x%2 ==0]
```

条件语句可以被很明确的添加在每一层循环的 `for` 表达式的后面，起到过滤的作用。例如：你想过滤出每行总和大于10且能被3正处的元素。虽然用列表表达式表示出这段代码很短，但是其可读性确实很糟糕。

```
matrix = [[ 1, 2, 3],[ 4, 5, 6],[ 7, 8, 9]]
filtered = [[x for x in row if x%3==0]
            for row in matrix if sum(row) >= 10 ]
print(filtered)
>>>
[[6],[9]]
```

可能你会感觉上面的例子有些费解，不过在实践中你将能看到更多列表表达式完美工作的情形。我强烈的建议你在遇到以下情形的时候避免使用列表表达式。那就是代码对别人来说难于理解；通过使用列表表达式来减少的代码行数不足以胜过其即将带来的麻烦时，请不要使用列表表达式。

拇指规则是指在一个列表表达式中避免使用超过两个的表达式。这些表达式可以是条件语句，循环语句，或者一个判断一个循环。只要事情变得比这种情况还要复杂，就不应该使用列表表达式了。 而是应该使用常规的语句来实现相同的业务逻辑。（详见第16项：考虑生成式而不是返回列表）

备忘录

- 列表表达式支持多层的循环和条件语句，以及每层循环内部的条件语句。
- 当列表表达式内部多余两个表达式的时候就会变得难于阅读，这种写法应该避免使用。

复杂地方考虑使用生成器表达式

列表生成式的缺点

关于列表理解（详见第7项：使用列表表达式而不是 `map` 和 `filter` ）方面的问题就在于其可能会给输入列表中的每一个只创建一个新的只包含一个元素的列表。这对于小的输入序列可能是很好用的，但是大的输入序列而言就很有可能导致你的程序崩溃。

例如：你想读取一个文件并且返回每行中的字符的数量。而是用列表表达式来处理这个问题的前提是内存中必须完全的包含这个文件的所有行。如果这个文件很大或者数据来自一个可能永远不会关闭的网络套接字数据流，此时列表表达式就会是一个大问题了。下面的这个例子就是指仅仅能处理小的输入序列的一个样例展示。

```
value = [len(x) for x in open('./my_file.txt','rb')]
print(values)
>>>
[100, 57, 15, 1, 12, 75, 5, 86, 89,11]
```

生成器表达式的好处

为了解决上面的这个问题， `Python` 提供了一个 `generator expression` （生成器表达式），其实就是列表表达式和生成器的一般化的体现。在程序运行的过程中，生成其表达式不实现整个输出序列，相反,生成其表达式仅仅是对从表达式中产生一个项目的迭代器进行计算，说白了就是每次仅仅处理一个迭代项，而不是整个序列。

生成器表达式通过使用类似于列表表达式的语法（在 `()` 之间而不是 `[]` 之间，仅此区别）来创建。这里，作者使用了一个和上例等价的生成器表达式。不同的是生成器表达式仅仅对当下的迭代项进行处理，并不做什么预先处理。

```
it = ( len(x) for x in open('/tmp/my_file.txt'))
print(it)
>>>
<generator object <genexpr> at 0x101b81480>
```

如果必须的话，返回的迭代器可以在生成器表达式中，一次被提前一步获得处理，用以产生下一个输出项。你的代码可以随心所欲的操作数据而无需顾虑内存使用率危机。

```
print(next(it))
print(next(it))
>>>
100
57
```

链式操作

生成器表达式的另一个强大的体现就在于其可以组合使用。这里，作者使用上面生成器表达式返回的迭代器来作为下一个生成器表达式的输出。

```
roots = ((x,x**0.5) for x in it)
```

每次推进这个迭代器，它就会推进内部的那个迭代器，通过循环，评估条件语句，输入输出等来产生一个类似于 多米诺骨牌 的效果。

```
print(next(roots))
>>>
(15,3.872983346207417)
```

在 `Python` 中，执行类似的链式的生成器是很快的，当你正寻找一种构建功能的方法来处理一个巨大的输入流的时候，生成器表达式将会是一个很不错的选择。唯一的缺点就是通过生成器表达式返回的迭代器是有状态的，所以你必须很小心的来使用（详见第17项：当迭代参数的时候要小心应对）。

备忘录

- 当遇到大输入事件的时候，使用列表表达式可能导致一些问题。
- 生成器表达式通过迭代的方式来处理每一个列表项，可以防止出现内存危机。
- 当生成器表达式 处于链式状态时，会执行的很迅速。

enumerate 比range更好用

迭代

迭代一个数值型的集合时，使用内置的 `range` 函数是一个很不错的选择，例如：

```
from random import *
random_bits = 0
for i in range(64):
    if randint(0,1):
        random_bits |= 1 << i
```

当你有一个待 迭代的数据结构（比方说字符串内容的集合）的时候，你就可以直接循环处理了,而不需要使用 `range` 函数。

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print("%s is delicious."%flavor)
```

很多时候，你想在迭代一个集合的同时获悉当前值在列表中的下标。例如：你想打印出你最喜爱的冰淇淋口味的范围，一种方式是使用 `range` 来完成。

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print("%d:%s"%(i+1, flavor))
```

与其他的迭代集合或者 `range` 范围的例子想必，这段代码看起来很是笨拙。你不得不先计算出列表的长度，还必须将这个数组索引化。所以，代码阅读起来就显得很困难。

枚举

其实针对于这种情况，`Python`提供了一个枚举函数。枚举函数可以使用懒模式来包装任何的迭代器。这对字段就代表了迭代器中当前项的下标以及值，这样一来，代码就显得很干净整洁了。

```
for i, flavor in enumerate(flavor_list):
    print("%d:%s"%(i+1,flavor))
>>>
1: vanilla
2: chocolate
3：pecan
4：strawberry
```

你也可以通过指定 索引开始的下标序号来简化代码，如下：

```
for i, flavor in enumerate(flavor_list,1):
    print("%d: %s"%(i,flavor))
>>>
1: vanilla
2: chocolate
3：pecan
4：strawberry
```

备忘录

- `enumerate`提供了简洁的语法，再循环迭代一个迭代器的同时既能获取下标，也能获取当前值。
- 在索引化一个序列的时候，应该避免使用`range`方式，而应该使用`enumerate`的方式。
- 可以添加第二个参数来指定 索引开始的序号，默认为`0`。

用拉链来并行处理迭代器

经常地，你会在 `Python` 中发现自己有许多相关的对象列表。列表表达式可以使得在获取源数据列表和通过赋值表达式（详见第7项：是用列表表达式而不是 `map` 或者 `filter` ）而衍生的列表处理上变的相当的容易。

```
names = ['Cecilia', 'Lise', 'Marie']
letters = [ len(n) for n in names]
```

这些衍生的数据项与其源数据项也是息息相关的，这与其索引相关。为了并行的迭代这两个列表，你可以遍历此源列表。

```
longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count
print(longest_name)
>>>
Cecilia
```

但是问题是，在循环过程中代码出现的阅读上的干扰忒多。 `names` 的下标以及 `letters` 使得代码难于阅读。并且循环中将数组索引化发生了两次。使用 `enumerate` （详见第10项：使用 `enumerate` 而不是 `range` ）函数可以稍微的改善这个状况，但是效果并不是特别的理想。

```
for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = name
        max_letters = count
```

为了使代码更加的干净，`Python`提供了一个内置的拉链式的函数。
在`Python3`中，`zip`通过懒模式生成器封装了两个或者更多的迭代器。`zip`生成器字段元祖包含了每一个迭代器的下一个元素的值，现在代码比刚才的使用的索引了两个列表的案例变的更加的清晰咯。

```
for name,count in zip(names,letters):
    if count > max_letters:
        longest_name = name
        max_letters = count
```

但是，关于内置的`zip`函数，有如下两个问题。

- 一是在`Python2`中`zip`并不是一个生成器，`zip`函数将完全的使用提供的迭代器，并且返回由它创建的所有元组而组成的一个集合。这很有可能由于自身使用了太多的内存空间而导致问题的发生，甚至崩溃。如果你想在`Python2`中处理较大的迭代器，那么你应该使用`itertools`模块中的`izip`函数(详见第46项：使用内置的算法和数据结构)。
- 另一个问题是如果输入序列的长度不一致的时候，`zip`表现的将会很奇怪。例如：你在上面的集合中添加了一个新的`name`值，但是忘记了更新`letter`的数量，此时运行代码的话就会出现意想不到的结果。

  ```
  names.append("Rosalind")
  for name, count in zip(names,letters):
    print(name)
  >>>
  Cecilia
  Lise
  Marie
  ```

而新添加的值 `Rosalind` 并没有出现在结果集中。这就是 `zip` 的工作方式，它将维持原来的字段元祖直到其中一个包装的迭代器走到了尽头。也就是默认会截断输出，剩下的数据将不予展示。如果你知道被处理的迭代器的长度一致（通常是被列表表达式衍生出来的）时，这个方法运行的将会很好。在很多其他情形下， `zip` 的截断行为是不好的，令人吃惊的。如果你不能保证你正处理的列表的长度一致，那么我建议你使用内置模块 `itertools` 的 `zip_longest` 函数来代替（在 `Python2` 中称为 `izip_longest` ）。

备忘录

- 内置的 `zip` 函数可以并行的对多个迭代器进行处理。
- 在 `Python3` 中，懒模式生成器获得的是元组；而在 `Python2` 中， `zip` 返回的是一个包含了其处理好的所有元祖的一个集合。
- 如果所处理的迭代器的长度不一致时， `zip` 会默认截断输出，使得长度为最先到达尾部的那个长度。
- 内置模块 `itertools` 中的 `zip_longest` 函数可以并行地处理多个迭代器，而可以无视长度不一致的问题。

在for 和while 循环体后避免使用else语句块

Python 的循环有一个其他编程语言中见不到的额外的特征：你可以在一个循环语句块之后立即的添加一个 else 语句块。

```
for i in range(3):
    print("Loop in %d!"% i )
else:
    print("Else Block!")
>>>
Loop in 0
Loop in 1
Loop in 2
Else Block!
```

让人惊讶的是，当循环体运行结束的那一刻 else 语句块就会立即得到执行。那为什么起名为 else 块而不是“ and ”语句块呢？在的 if/else 语句中， else 意味着“当 if 的那个条件不满足是，才执行这个语句块”。在 try/except 语句块中， except 则意味着相同的定义，“只有当 try 块内出现异常的时候才会执行 except 内部的代码”。

相似地，在 try/except/else 语句块中的 else 也遵循上面的那个模式（详见第13项：从 try/except/else/finally 语句块中受益）。因为这意味着“在如果前面的语句块没有失败的话就执行这个块”。 try/filally 仍然很直观，含义为“总是能够在前面的 try 块结束之后执行 finally 语句块”。

鉴于 Python 中所使用的 else ， except ， finally 块，一个新的编程语言可能会假定在 for/else 块中的 else 含义为：“如果循环体没有正常的结束就执行 else 语句”。 然而事实上，含义恰恰相反。再循环体中使用 break 语句可以真正地跳过 else 块。

```
for i in range(3):
    print('Loop %d' % i)
    if i == 1:
        break
else:
    print('else block')
>>>
Loop 0
Loop 1
```

另一个让人惊讶的地方就是：如果你便利一个空序列的话 `else` 语句块就会立即得到执行。

```
for x in []:
    print('Never runs')
else:
    print('else block')
>>>
else block
```

对于这个行为比较合理的解释就是：位于循环块之后的 `else` 语块会在通过循环查找一些东西时非常的有用。例如：你想查看两个数是否互为质数（公共的除数只能是 `1` ）。这里，我迭代了每一个可能的公约数来测试这些数据。尝试了每一个选择后，循环结束。当两个数互为质数的时候 `else` 语块就会得到执行，因为再循环体中并没有触发 `break` ，所以互为质数。

```
a = 4
b = 9
for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a % i == 0 and b % i == 0:
        print('not coprime')
        break
else:
    print('coprime')
>>>
Testing 2
Testing 3
Testing 4
coprime
```

在实际的使用中，你不可能像这样来书写代码，你可能会写一个工具函数来计算。常见的两种形式如下：

- 第一个是一旦发现正在寻找的条件就返回。如果遍历完整个循环则返回默认的输出。

```
def coprime(a, b):
  for i in range(2, min(a, b) + 1):
      if a % i == 0 and b % i == 0:
      return False
  return True
```

- 第二个是申请一个结果变量来表明你是否在循环中发现正在寻找的条件，一旦发现就通过 `break` 跳出循环来返回。

```
def coprime2(a, b):
  is_coprime = True
  for i in range(2, min(a, b) + 1):
      if a % i == 0 and b % i == 0:
          is_coprime = False
          break
  return is_coprime
```

面对如此逻辑的代码。使用以上两种方法都会给读者一个清晰简洁的阅读体验。你从中获得的就是一个良好的阅读体验，想象一下这样做是否能够胜出改善添加 `else` 语块而导致的今后代码上阅读难度就明白了。像循环这种简单的结构在 `Python` 中是不言而喻的了，你应该避免在循环块的后面使用 `else` 语句。

备忘录

- `Python` 有用特殊的语法能够让 `else` 语块在循环体结束的时候立刻得到执行。
- 循环体后的 `else` 语块只有在循环体没有触发 `break` 语句的时候才会执行。
- 避免在循环体的后面使用 `else` 语块，因为这样的表达不直观，而且容易误导读者。

在try/except/else/finally中受益

在 Python 中，有四种可能的情况你要在产生异常的时候采取行动。这些异常可以被 try，except，else，finally 块自动的捕获。在复合语句中，每一个块都服务着一个独特的目标，并且它们的组合语句也很有用（详见第 51 项：又如，“从 API 中定义一个根异常来隔离调用者”）。

finally 块

当你想使得异常传播的时候可以使用 try/finally，但是你还想在代码出现异常的时候仍然能够整洁的运行。一个通用的使用方式就是在 try/finally 块中可靠的关闭文件处理句柄（详见第43项：对另一个方法而言“为了可重复的使用 try/finally，考虑一下 contextlib 和 with 语句）。

```
# 此处可能引发IO异常
handle = open('/tmp/random_data.txt')
try:
    # 可能引发UnicodeDEcode异常
    data = handle.read()
finally:
    # 总是在try语句结束后关闭文件
    handle.close()
```

由于读文件而引发的异常总是会被抛到调用方代码处，而为保证 finally 块中的文件操作能被正确的关闭。你必须在 try 块之前调用 open 方法，因为这样如果文件不存在会引发 IOError，使得能跳过 finally 块而不会导致代码本身存在逻辑问题。

else 块

使用 try/except/else 块可以使得代码中对哪种一场要被处理，哪种异常要往上抛出的处理变得更加清晰。当 try 块内的代码没有引发异常的时候，else 块就会得到执行。else 块可以使得 try 块中的代码变得更加的简短，并且大大改善

代码的可读性。例如：你可能想从一个字符串中加载 JSON 字典数据，并且返回类似于键值对信息的结果。

```
def load_json_key(data, key):
    try:
        result_dict = json.loads(data)  # 可能引发ValueError异常
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]       # 可能引发KeyError
```

如果字符串中包含的数据不是合法的 JSON 串数据，解码函数 json.loads 将会引发一个 ValueError 异常，然后被 except 块捕获并且处理。如果解码工作可以正常的得到执行，那么在 else 块中寻找 key 对应的值的时候就很有可能引发 KeyError 异常。然后这个一场将会被上抛给调用该函数的代码，因为它们是 try 块之外的外部代码。 else 块的存在可以确保 try/except 块可以处理，分辨的异常有哪些。这使得异常要抛给谁，变得更加的清晰。

大综合

综合性的使用就是把 try/else/except/finally 都用上，写到一个复合语句中。例如：你想读取一个描述了工作信息的文件，然后处理，再更新到这个文件中。这里， try 块就经常的被用于读取文件和处理文件， except 块用于处理从 try 块中捕获到的预期的异常，而 else 块则习惯用于更新文件内部的信息并且允许相关的异常往上抛出，最后 finally 块用于关闭文件等打扫战场性质的工作。

```
UNDEFINED = object()
def divide_json(path):
    handle = open(path,'r+')    # 可能抛出IOError异常
    try:
        data = handle.read()    # 可能引发UnicodeDecodeError异常
        op = json.loads(data)   # 可能引发ValueError异常
        value = (
            op['numeerator'] /   # 这里是除号！
            op['denominator']   # 可能引发ZeroDivisionError异常
    except ZeroDivisionError as e:
        return UNDEFINED
    else:
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)
        handle.write(result)    # 可能引发IOError异常
        return value
    finally:
        handle.close()        # 肯定能够被执行成功
```

这样的结构是非常非常实用而且好用的，因为所有的块都能以直观的方法一起工作。例如：如果一个在 `else` 快中写文件数据的时候发生了，`finally` 块仍然会正确的关闭文件，减少了文件损坏的可能。

备忘录

- `try/finally` 组合语句可以使得你的代码变得很整洁而无视 `try` 块中是否发生异常。
- `else` 块可以最大限度的减少 `try` 块中的代码的长度，并且可以可视化地辨别 `try/except` 成功运行的部分。
- `else` 块经常会被用于在 `try` 块成功运行后添加额外的行为，但是要确保代码会在 `finally` 块之前得到运行。

函数

`Python` 程序员使用的第一个有组织性的工具便是 `function` （函数）了。和其他的编程语言一样，函数的存在可以让你把大段的代码分割成更小的，更简单的小块。这些小块不仅改善了代码的可读性，同时也使代码变得更加的平易近人。它们在今后代码的重用和重构上起到了巨大的作用。

`Python` 中的 `function` 拥有一些额外的特征，而正是这些特征的存在，程序员的生活才得以变得更加的轻松。一些可能和其他的编程语言中某些特征相似，但是一些在 `Python` 中却是独有的。这些额外的特征使得函数的功用性更加的明显。消除干扰，使调用者的意图更加的清晰，并且可以减少难于发现的微小的代码漏洞。

返回exceptions而不是None

编写一个功效函数，对 `Python` 程序员而言返回一个特殊的值为 `None` 时，要有一个提取的过程。有些时候这会显得很有意义。例如：你需要一个关于两个数除法的工具函数，当发生除零状况时返回 `None` 似乎显得很是自然而然，因为结果本身就是未定义的嘛。

```
def divide(a, b):
    try:
        return a/b
    except ZeroDivisionError:
        return None
```

因此，使用这个函数的代码就可以根据返回的结果来作进一步的解释处理了。

```
result = divide(x, y)
if result is None:
    print('Invalid inputs!')
```

当分子为 `0` 的时候会是什么样呢？这个函数将也会返回一个 `0` （当然咯，前提是分母非 `0` ）。然而当你在类似于 `if` 这样的条件语句进行评估的时候就有可能出现问题。因为你可能预期一个等价于 `False` 的任何可能的值来表明条件能否得到执行，而不是预期一个 `None` 值（详见第4项：编写功效函数而不是复杂的表达式）。

```
x, y = 0, 5
result = divide(x, y)
if not result:
    print("Invalid inputs")   # 很明显，这是错误的
```

当 `None` 拥有一个特殊的含义的时候， `Python` 中的这种代码就会是一个很常见的错误了。这也是为什么从一个函数中返回 `None` 容易使得代码出错的缘故。不过还好，我们有以下两种方法来减少此类事故的发生。

- 一是将返回值分割成一个二元组，元组中的第一个值代表本操作成功与否，而第二个参数就代表了对应于操作成功与否的真实的计算值了。

```
def divide(a, b):
  try:
      return True, a/b
  except ZeroDivisionError:
      return False, None
```

这样一来，调用了上面的 `divide` 函数的代码就需要解开返回的元组，这就要求调用方先考虑 `divide` 函数的执行状态，而不是单纯的仅仅盯着除法的计算结果。

```
success, result = divide(x, y)
if not success:
  print("Invalid inputs")
```

虽然这样做很方便，但是却容易出现问题，那就是调用方很容易忽略 `divide` 函数返回的元组中的第一个状态值（在 `Python` 中对于未使用的变量的惯例就是使用下划线变量名称，一个下划线即可）。再看 `divide` 函数，乍一看返回的结果也没什么问题，但是事实上这和直接返回 `None` 一样的糟糕。

```
_, result = divide(x, y)
if not result:
  print("Invalid inputs")
```

- 第二个方法也是更好一点的方法那就是永远不返回 `None` 值。相反，而是要触发，上抛给调用方一个异常，让调用方来处理。这里，我把一个 `ZeroDivisionError` 以上变成了一个 `ValueError` 异常来告知调用方，其输入值存在问题。

```
def divide(a, b):
  try:
      return a/b
  except ZeroDivisionError as e:
      return ValueError("Invalid inputs") from e
```

现在，如果由于输入值存在问题而引发的异常就可以被调用方直接的处理掉了（这个行为应该被文档化；详见第49项：为每一个函数，类以及模块写文档信息）。而且调用方也不再需要单独的为返回值的状态多写一段代码了。如果函数并没有触发一个异常，那就说明咱们的输入值是正确的,异常处理的结果也变的如此清晰了。

```
x, y = 5, 2
try:
    result = divide(x, y)
except ValueError:
    print("Invalid inputs")
else:
    print("Result is %.1f"% result)
 >>>
 Result is 2.5
```

备忘录

- 返回 `None` 的函数来作为特殊的含义是容易出错的，因为 `None` 和其他的变量（例如 `zero` ，空字符串）在条件表达式的判断情景下是等价的。
- 通过触发一个异常而不是直接的返回 `None` 是比较常用的一个方法。这样调用方就能够合理地按照函数中的说明文档来处理由此而引发的异常了。`

了解闭包中是怎样使用外围作用域变量

作用域问题

有没有遇到这样的一种情况，你想对一个列表中而定数字进行排序，但是有一组要优先进行。这个模式在实际的运用中也是很普遍的，比如你想渲染一个用户接口，或者在其他要处理的事情之前展示重要的消息或者异常事件，优先排序的重要性就体现出来了。

一个通用的方式是把一个工具函数当做一个关键参数传递给列表排序方法，此工具函数的返回值将作为每一个元素排序的权重依据，从而实现优先排序。这个工具函数也可以检测给定的元素是否在那个重要的组别内或者改变排序元素的顺序。

```
In [1]: values = [1,5,3,9,7,4,2,8,6]

In [2]: group = [7,9]

In [3]: def sort_priority(values, group):
   ...:     def helper(x):
   ...:         if x in group:
   ...:             return (0, x)
   ...:         return (1, x)
   ...:     values.sort(key=helper)
   ...:

In [4]: sort_priority(values,group)

In [5]: print values
[7, 9, 1, 2, 3, 4, 5, 6, 8]
```

关于此函数可以返回预期结果有如下三个原因：

- `Python` 支持闭包，即可以从它们被限定的范围引用变量的函数（通俗点讲就是函数可以引用声明域比自己大的变量）。这也是为什么 `helper` 函数可以访问 `set_priority` 函数中的 `group` 变量。

- 在 `Python` 中函数是头等对象，意味着你可以直接引用他们，把它们赋值给变量，作为参数传递给其他的函数，在表达式或者 `if` 条件语句中直接比较等等。这就是为什么排序算法可以接受一个闭包函数作为其参数。
- 在比较元组方面， `Python` 有一个特殊的规则。那就是先比较元组中下标为 `0` 的元素，然后依次递增。这也是为什么从闭包函数返回的值能够在两个不同的列表中的排序起到作用的原因。

如果拥有高优先级的元素无论被包含与否，用户接口处的代码，函数都可以正常工作，那将会是极好的。看起来添加这么一个行为似乎很是直截了当。然而这里仍然是后一个闭包函数来决定当前元素隶属于哪一个组别。那为什么不在高优先级元素被发现的时候使用闭包来翻转一下呢？届时函数就可以返回一个标志，来代表着其被闭包函数修改过了。这里，我尝试着模拟了一下。

```
def sort_priority2(values, group):
    found = False
    def helper(x):
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    values.sort(key=helper)
    return found
# begin to call
found = sort_priority2(values, group)
print("Found:",found)
print(values)
>>>
Found: False
[7, 9, 1, 2, 3, 4, 5, 6, 8]
```

排序的结果是正确的，但是很明显分组的那个标志是不正确的了。 `group` 中的元素无疑可以在 `values` 里面找到，但是函数却返回了 `False` ，为什么会发生这样的状况呢？

我们不妨这样来理解，当你在表达式中引用一个变量的时候， `Python` 的解释器将会横向的在作用域按如下顺序查找该值

- 当前函数的作用域。
- 任何其他的封闭域（比如其他的包含着的函数）。

- 包含该段代码的模块域（也称之为全局域）。
- 内置域（包含了像 `len`，`str` 等函数的域）。

如果在上面四种情况中没有一个域是包含一个定义过了的变量的话，`NameError` 异常就会被触发。

给不同的变量赋值的原理也是不同的。如果一个变量在当前作用域内已经被定义过了，仅仅赋予其新值即可；如果当前作用域内不存在该值，`Python` 就会将其当做变量先进行定义。此新定义的变量的作用域就是包含了这个赋值语句的函数块了。

```
def sort_priority2(numbers, group):
    found = False    # 作用域：sort_priority2
    def helper(x):
        if x in group:
            found = True    # 作用域： helper
            return (0, x)
        return (1, x)      # 一旦执行了return语句，found在helper的作用域就会由helper转至sort_priority2函数。相应的其值也会发生变化。
    numbers.sort(key=helper)
    return found
```

产生这个问题的原因就是有时函数调用，新值被定义而引起的作用域漏洞。但是这也是预期的结果。这可以很好的避免函数中的局部变量污染模块中的同名变量的值。另外函数内部的每一次赋值都会把垃圾放入到全局的模块域。当然，这不仅会扰乱视线，更加会由于全局变量的相互作用而引发一些不起眼的错误。

把数据放到外边

在 `Python3` 中，对于闭包而言有一个把数据放到外边的特殊的语法。`nonlocal` 语句习惯于用来表示一个特定变量名称的域的遍历发生在赋值之前。唯一的限制就是 `nonlocal` 不会向上遍历到模块域级别（这也是为了防止污染全局变量空间）。这里，我定义了一个使用了 `nonlocal` 关键字的函数。

```
def srt_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

当数据在闭包外将被赋值到另一个域时，`nonlocal` 语句使得这个过程变得很清晰。它也是对 `global` 语句的一个补充，可以明确的表明变量的赋值应该被直接放置到模块域中。

然而，像这样的反模式的全局便令，我发对使用在那些简单函数之外的其他的任何地方。`nonlocal` 引起的副作用是难以追踪的，而在那些包含着 `nonlocal` 语句和赋值语句交叉联系的大段代码的函数的内部则尤为明显。

当你感觉自己的 `nonlocal` 语句开始变的复杂的时候，我非常建议你重构一下代码，写成一个工具类。这里，我定义了一个实现了与上面的那个函数功能相一致的工具类。虽然有点长，但是代码却变得更加的清晰了（详见第23项：对于简单接口使用函数而不是类里面的 `__call__` 方法）。

```
class Sorter(object):
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter is True
```

Python2中的作用域

不幸的是，`Python2` 是不支持 `nonlocal` 关键字的。为了实现相似的功能，你需要广泛的借助于 `Python` 的作用与域规则。虽然这个方法并不是完美的，但是这是 `Python` 中比较常用的一种做法。

```
# Python2
def sort_priority(numbers, group):
    found = [False]
    def helper(x):
        if x in group:
            found[0] = True
            return (0, x)
        return (1, x)
    numbers.sort(sort=helper)
    return found[0]
```

就像上面解释的那样，`Python` 将会横向查找该变量所在的域来分析其当前值。技巧就是发现的值是一个易变的列表。这意味着一旦检索，闭包就可以修改 `found` 的状态值，并且把内部数据的改变发送到外部，这也就打破了闭包引发的局部变量作用域无法被改变的难题。其根本还是在于列表本身元素值可以被改变，这才是此函数可以正常工作的关键。

当 `found` 为一个 `dictionary` 类型的时候，也是可以正常工作的，原理与上文所言一致。此外，`found` 还可以是一个集合，一个你自定义的类等等。

备忘录

- 闭包函数可以从变量被定义的作用域内引用变量。
- 默认地，闭包不能通过赋值来影响其检索域。
- 在 `Python3` 中，可以使用 `nonlocal` 关键字来突破闭包的限制，进而在其检索域内改变其值。
- `Python2` 中没有 `nonlocal` 关键字，替代方案就是使用一个单元素（如列表，字典，集合等等）来实现与 `nonlocal` 一致的功能。
- 除了简单的函数，在其他任何地方都应该尽力的避免使用 `nonlocal` 关键字。

考虑使用生成器而不是返回列表

对于函数而言，处理一个结果集序列的最简单的方式就是返回一个元素的集合。例如：你想知道一个字符串中每个单词的索引下标。这里，我使用 `append` 方法将结果存储于一个列表中，并在函数的最后将这个结果集返回。

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index+1)
    return result
# 测试
address = "Four score ad sever years ago..."
result = index_words(address)
print(result[:3])
>>>
[0, 5, 11]
```

虽然结果可以正常的获取，但是这个函数中却存在两个问题。

- 一个是代码有点杂乱。没一个新结果被发现就会调用一次 `append` 方法。将追加元素之的工作变得散漫化了。并且代码中初始化 `result` 集合的时候占据了一行代码，返回结果集 `result` 的时候也使用了一行代码。而且代码块中大概有 `130` 个字符，但是只有约 `75` 个字符是有用的。所以整体看起来，代码的效率和简洁性都不是很高。

  针对上述情况，一个更好一点的解决办法就是使用 `generator` （生成器）。生成器就是使用了 `yield` 表达式的函数。当此函数被调用的时候 他不会真正的去执行，而是返回一个迭代器。每次调用这个内置的函数，迭代器就会把生成器推进到下一个 `yield` 表达式上。每个被迭代器经过的值就会被返回给调用者。这里我做了个关于生成其表达式的小例子，功能和前面的那段代码一致。

```
def index_words_iter(text):
  if text:
      yield 0
  for index, letter in enumerate(text):
      if letter ==" ":
          yield index + 1
```

很明显，由于淘汰了 `result` 集合内部元素之间的相互作用，代码变的更加容易了。结果集被传递到了 `yield` 表达式上面。被生成器调用而返回的迭代器可以轻松的被传递给内置的列表函数（详见第 `9` 项：为大段代码考虑使用生成器）而转换成我们预期的结果集。

```
result = list(index_words_iter(address))
```

- 第二个问题就是 `index_words` 函数需要列表中所有的结果元素是排好序的。对于大量的输入，这就很有可能引发内存危机甚至崩溃。相反，使用了生成器的那个版本就可以很轻松的适配大量数据的输入处理。

  这里，我定义了一个生成器的一次一行的文件流处理例子，并且使用 `yield` 来进行一次一个单词的输出。代码运行的内存被限制在了每行输入数据的最大长度。

```
def index_file(handle):
  offset = 0
  for line in handle:
      if line:
          yield = offset
      for letter in line:
          offset += 1
          if letter == " ":
              yield offset
# 测试运行此函数
with open('/tmp/address.txt','r')as f:
  it = index_file(f)
  results = islice(it, 0, 3)
  print(results)
>>>
[0, 5, 11]
```

是不是很完美啊，唯一的缺点就是此函数的调用方必须了解这个迭代器是状态相关的，并且不能被重用（详见第17项：迭代参数的时候记得保守一点点）。

备忘录

- 相较于返回一个列表的情况，替代方案中使用生成器可以使得代码变得更加的清晰。
- 生成器返回的迭代器，是在其生成器内部一个把值传递给了 `yield` 变量的集合。
- 生成器可以处理很大的输出序列就是因为它在处理的时候不会完全的包含所有的数据。

遍历参数的时候保守一点

常规遍历

当一个函数需要一个集合作为其参数的时候，多次遍历整个集合时很重要的。例如：你想分析一下美国德克萨斯州的旅游人数，想象一下每个城市的旅游人数的数据集该有多大！要求得每个城市中旅游人数在德克萨斯州旅游总人数的百分比，这将会是多么繁重的一项任务。

为了完成这个目标，你需要一个常规性的函数，貌似其接收的输入就是每年的旅游总人数。然后按旅游地分配到不同的成熟，最后计算出每个城市对德克萨斯州旅游业的贡献。

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
# 测试
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
>>>
[11.53846, 26.92307, 61.53846]
```

添加拓展

为了拓展这个函数，我还需要读取一个包含了德克萨斯州所有城市的文件。我定义了一个生成器来做这件事，因为如果我以后想计算全世界的旅游业状况的时候我可以很好的重用这段代码（详见第 16 项：使用生成器而不是返回列表）。

```
def read_visits(data_path):
    with open(data_path,'r') as f:
        for line in f:
            yield int(line)
#让人惊讶的是，调用了read_visits后，返回的结果为空（[]）。
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize(it)
print(percentages)
>>>
[]
```

造成上述结果的原因是 一个迭代器每次只处理它本身的数据。如果你遍历一个迭代器或者生成器本身已经引发了一个 `StopIteration` 的异常，你就不可能获得任何数据了。

```
it = read_visits('tmp/my_numbers.txt')
print(list(it))
print(list(it)) # 这里其实已经执行到头了
>>>
[15, 35, 80]
[]
```

当你感到疑惑的可能就是你无法在迭代器已经迭代结束的时候发现这个错误，而 `list` 够咱函数以及很多其他的 `Python` 标准库函数却在普通操作的过程中得到 `StopIteration` 异常。这些函数不能分辨到底是没有输入值还是迭代器已经到头了，所以才容易出错。明白了这点，就可以搞懂为什么上面的代码返回的是一个空列表了吧。为了解决这个问题，就需要从迭代器本身下手了，既然是在迭代的过程中导致的问题，那我们也就从这点下手。在未迭代之前先复制一份完整的副本，并保存到一个集合里面。然后就可以通过迭代这个集合来规避上面的那个问题了。代码如下：

```
def normalize_copy(numbers):
    numbers = list(numbers)  # 复制一份完整的副本，并转换成一个集合
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
# 测试
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
>>>
[11.53846, 26.92307, 61.53846]
```

看到 `list` 构造函数，聪明的你可能也会觉得，要是 `iterator` 非常的大怎么办，那样岂不是又会出现一开始的内存危机了吗？是的，确实是这样。解决这个问题的一个办法就是接受一个函数，这样每次调用这个函数的时候都会返回一个新的迭代器，这样也可以是问题得到解决。

```
def normalize_func(numbers):
    total = sum(get_iter()) # 一个新的迭代器
    result = []
    for value in get_iter(): # 又一个新的迭代器
        percent = 100 * value / total
        result.append(percent)
    return result
```

为了使用normalize_func函数，你可以将一个lambda表达式作为参数，对每次调用都返回一个迭代器。

```
percentages = normalize_func(lambda: read_visits(path))
```

自定义容器类

虽然代码可以正常的工作了，但是每次都要传递一个 `lambda` 表达式又显得很笨拙。一个更好的方式就是提供一个实现了 `iterator` 协议的 `container` 类。 `iterator` 协议是 `Python` 中对于循环和相关表达式内容类型的查找的解释。当 `Python` 遇到像 `for x in foo` 这样的表达式的时候，他就会调用 `iter(foo)` 。内置的 `iter` 函数然后会立刻调用 `foo.__iter__` 方法。而 `__iter__` 方法一定会返回一个迭代器对象（这个方法本身实现了 `__next__` 方法）。然后循环语句就可以重复的调用 `next` 这个内置于迭代器中的方法，直到结果集完全被遍历或者引发了一个 `StopIteration` 异常。

听起来貌似很复杂，但是事实上只要生成器实现了 `__iter__` 方法就足够了。这里，我定义了一个可迭代的容器类来读取旅游业数据。

```
class ReadVisitors(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield ine(line)
# 现在使用原来的测试函数也可以使得代码正确的运行了。
visits = ReadVisitors(path)
percentages = normalize(visits)
print(percentages)
>>>
[11.53846, 26.92307, 61.53846]
```

现在这段代码可以正常的工作的原因就在于 `noamalize` 调用了 `ReadVisitors` 的 `__iter__` 方法来重新分配了一个迭代器，从而避免了第二次迭代的时候迭代器已失效（因为在计算 `total` 的时候， `sum` 函数会一下子迭代完，整个迭代器，等下面的 `for` 循环再遍历的时候就会出现内容已经耗尽的情况发生）的状况。而现在不会了，现在我们会重新生成一个迭代器，在进行循环的时候也可以使得代码正常的工作。现在唯一的缺点就在于 这个方法会多次读取输入数据。

经过了上面的例子的洗礼，想必你已经明白了像 `ReadVisitors` 这样的容器是如何工作了的吧。当你的函数中某些参数不只是迭代器的时候你也可以仿照本例来完善一下。核心就在于：当一个迭代器被 `iter` 函数接收的时候会重新返回这个迭代

器本身。相反，每次当一个动器类型被传给 `iter` 函数的时候，一个新的迭代器就会返回一个新的迭代器对象(注意是一个全新的迭代器，待会会用这个条件来判断的)。因此，你可以测试一下这个行为，并触发一个 `TypeError` 来拒绝迭代器。

```
def normalize_defensive(numbers):
    if iter(numbers) is iter(numbers): # 是个迭代器，这样不好
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

如果你不想像 `normalize_copy` 函数那样复制整个输入迭代器，使用这个方式就变的很理想了，但是你仍然需要多次的迭代输入数据。这个函数可以符合预期的运行，因为 `ReadVisitors` 是一个容器类。当然了，符合 `iterator` 协议的容器类也都是可行的，这里就不再过多地叙述了。

```
visits = [15, 35, 80]
normalize_defensive(visits)
visits = ReadVIsitors(path)
normalize_defensive(visits)

# 但是如果输入值不是一个容器类的话，就会引发异常了
it = iter(visits)
normalize_defensive(it)
>>>
TypeError: Must supply a container
```

备忘录

- 多次遍历输入参数的时候应该多加小心。如果参数是迭代器的话你可能看到奇怪的现象或者缺少值现象的发生。
- `Python` 的 `iterator` 协议定义额容器和迭代器在 `iter` 和 `next` 下对于循环和相关表达式的关系。

- 只要实现了 `__iter__` 方法，你就可以很容易的定义一个可迭代的容器类。
- 通过连续调用两次 `iter` 方法，你就可以预先检测一个值是不是迭代器而不是容器。两次结果一致那就是迭代器，否则就是容器了。

减少位置参数上的干扰

接受可选择的位置参数（通常被称为星型参数的常规名称 `*args` ）可以使得一个函数调用起来更加的简洁，并能明显减少可视化的“噪音”。例如： 你想打印一些调试信息。通过一个固定的参数数字，你需要一个带有信息和一串值的集合的函数。

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s '%(message, values_str))

log('My numbers are', [1, 2])
log('Hi there', [])
>>>
My numbers are: 1, 2
Hi There
```

但是如果没有 `values` 要传递的时候必须跟着传递一个空集合，这样使得代码看起来不是很简洁。如果可以忽略第二个参数的话就会很好了。在 `Python` 中可以通过在参数前面添加 `*号` 来实现，这样就变成了可有可无的参数了。此时 `log` 函数的第一个参数是必不可少的,而其他的参数就是可选的了。函数体部分我们也不需要进行改变，唯一需要改变的就是调用这个函数的代码即可。

```
def log(message, *values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s '%(message, values_str))

log('My numbers are', [1, 2])
log('Hi there')
>>>
My numbers are: 1, 2
Hi There
```

如果你已经有一个集合了，并且向调用一个像 `log` 这样的可变参数函数。你就可以通过使用 `*` 操作符来实现。在 `Python` 中可以通过这样的方式把序列中的元素值当做位置参数来使用。

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)
>>>
Favorite colors: 7, 33, 99
```

位置参数的两个问题

虽然位置参数的使用使得我们的程序变得更加的灵活，但是同样有其自身的一些问题。

- 第一个是可变参数在被传递给函数的时候经常会被转变成元组。这意味着如果调用方对生成器使用了 `*` 操作符，程序将会遍历整个序列直至完毕。如果结果集元组包含很多数据的话，同样会因为内存危机而导致宕机的可能。

```
def my_generator():
    for i in range(10):
        yield i

def my__func(*agrs):
    print(*args)

it my_generator()
my_func(*it)
>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

当你明确的指导传进来的列表数目已知是很小的时候，接收 `*args` 参数的函数会很合适。如果函数可以指定变量名称进行传值的话，那将是完美的。彼时代码的简洁性和可读性都将得到大大的提高。

- 第二个问题就是带有 `*args` 参数的话，你的函数以后就不能再新增其他的位置参数了，这还涉及到今后的函数移植问题。如果你想在函数的前面部分添加一个位置参数的话，已存在的调用者就不能正常的处理了。

```
def log(sequence, message, *values):
    if not values:
        print('%s: %s' % (sequence, message))
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s：%s' %(sequence, message, values_str))

# 新的调用可以正常的工作
log(1, 'Favorites', 7, 33)
# 原来的函数调用方式不能正常的工作
log('Favorite numbers', 7, 33)
>>>
1： Favorites: 7, 33
Favorite numbers: 7: 33
```

这里出现的问题：由于 `sequence` 变量未给出而导致了第二个函数调用把 `7` 作为了 `message` 变量来用了。像这样的问题很难通过漏洞追踪来发现，因为其可以并没有引发异常信息。当你想拓展函数来接受 `*args` 这种变量的时候，为了避免这

种问题发生的可能性，你应该尽量的使用关键字变量（详见第21：使用关键字变量来增强简洁性）。

备忘录

- 通过使用 `*args` 定义语句，函数可以接收可变数量的位置参数。
- 你可以通过 `*` 操作符来将序列中的元素作为位置变量。
- 带有 `*` 操作符的生成器变量可能会引起程序的内存溢出，或者机器宕机。
- 为可以接受 `*args` 的函数添加新的位置参数可以产生难于发现的问题，应该谨慎使用。

使用关键字参数来提供可选行为

关键字参数

和其他的许多编程语言一样，调用 `Python` 中的函数可以根据参数的位置来指定传递的参数。

```
def remiander(number, divisor):
    return number % divisor

assert remainder(20, 7) == 6
```

`Python` 函数中的所有的位置参数也可以通过关键字来传值,具体的使用方式就是在函数调用的时候使用类似于赋值的语句来指定参数名称。只要函数中的必须传值的参数被赋予了值，其他的参数的位置可以是随意次序的。你可以像玩“连连看”游戏似得来关联关键字位置参数。说起来可能不是那么容易理解，看一下具体的代码就会很简单了。下面的几种形式其实是等价的：

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, numer=20)
```

两个注意

- 但是咧，位置参数应该先于关键字参数被指定。也就是说没有指定关键字的话，就会按顺序为函数传递参数值。

```
remainder(number=20, 7)
>>>
SyntaxError: non-keyword arg after keyword arg
```

- 每个参数只能被赋值一次

```
remainder(20, number=7)
>>>
TypeError: remiander() got multiple values for argument 'number'
```

三大好处

关键字参数的灵活性给我们的程序带来了三个好处。

- 第一个是代码可读性的提高，简单的使用 `remainder` (20, 7)对程序调用者而言很不明显，到底哪个数字代表除数，哪个代表被除数呢？但是在关键字参数的调用实例中，一切都是那么的一目了然，特明显。
- 第二个好处就是关键字参数可以在定义的时候初始化一个默认值。这就为你的程序提供了一个很好的拓展功能，尤其是大部分时间你只需要默认的行为的时候。这可以在一定程度上消除重复代码，减少代码干扰。 例如：你想计算液体流入一个桶中的速率。如果桶也有刻度的话，然后你就可以利用不同时间下的重量差来计算出流速了。

```
def flow_rate(weight_diff, time_diff):
     return weight_diff / time_diff

weight_diff = 0.5
time_diff = 3
rate = flow_date(weight_diff, time_diff)
print('%.3f kg per second' % flow)
```

在典型的案例中，每秒的千克流速是很有用的。其余时间，使用传感器测量大概的刻度就可以了，比如小时，天等等。你可以 在函数中添加一个 `period` 的参数来指定这一个行为。

```
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

但是问题也接踵而来了，每次调用此函数你你都必须指定period的大小，甚至是普通的情况（通常 `period` 是1）下也得这样。为了减少这种代码上的冗余，我们就可以使用带有默认值的关键字参数咯。

```
def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period

# 现在period参数也是可选的了，默认值为1
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

对于简单的默认值而言，这个函数可以很好的运行（当默认值很复杂的时候就会变得让人很为难，详见第20项：使用 `None` 和文档来指定动态的默认值参数）。

- 第三个好处就是：在前面的调用方式不变的情况下可以很好的拓展函数的参数。不用修改太多的代码，并且减少 `bug` 出现的机会。

  ```
  def flow_rate(weight_diff, time_diff, period=1, units_per_kg
  =1):
    return ((weight_diff / units_per_kg) / time_diff) * period
  ```

  参数 `units_per_kg` 的默认值为 `1` ，这样可以保证返回的重量仍然为千克。而对于之前的函数调用方而言，函数貌似并没有发生什么变化，照常调用即可，而新的函数调用方也可以指定新的关键字参数来查看新的行为了。这就是好的兼容项代码的最佳体现。

  ```
  pounds_per_hour = flow_rate(weight_diff, time_diff, period=3
  600, units_per_kg=2.2)
  ```

  大致来说，这个函数已经完美了。唯一遗憾的就是对于 `period` 和 `units_per_kg` 来说仍然需要项位置参数一样被指定。

  ```
  pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.
  2)
  ```

  提供可选的位置参数可能会让你感到疑惑，因为 `3600` 和 `2.2` 到底是为哪个参数赋值不是很明显。这里我的建议就是，尽量的使用关键字参数来为它们赋值，而不要走使用位置参数来赋值的老路。

> 使用可选参数做到向后兼容对于程序的移植性很重要。

备忘录

- 函数的参数值即可以通过位置被指定，也可以通过关键字来指定。
- 相较于使用位置参数赋值，使用关键字来赋值会让你的赋值语句逻辑变得更加的清晰。
- 带有默认参数的关键字参数函数可以很容易的添加新的行为，尤其适合向后兼容。
- 可选的关键字参数应该优于位置参数被考虑使用。

使用None和文档说明动态的指定默认参数

有些时候你需要使用一个非静态类型的数据作为关键字参数的默认值。例如：你想打印被日志事件的时间标记的日志信息。默认情况下，当函数被调用的时候你想让欣心中包含时间信息。你可能会尝试下面的方法，假设默认参数每次被调用都会被重新赋值。

```
def log(message, when=datatime.now()):
    print("%s:%s" %(when, message))

log('Hi there!')
sleep(0.1)
log('Hi again')
>>>
2014-11-15 21:10:10.371432: Hi there
2014-11-15 21:10:10.371432: Hi again
```

从上面打印的信息来看，时间戳是相同的，这是因为 `datetime.now` 仅仅被执行了一次，而且是在函数被定义的时候。默认参数的值仅仅在模块被引入的时候才会被赋值一次，而这通常发生在程序开始运行的时候。当模块已经加载完这段代码后， `datetime.now` 的默认值就不会被改变了。

在 `Python` 中，为了实现预期的结果，惯用的方式是提供一个值为 `None` 的默认值，并且在帮助文档中记录详细的行为和使用方法（详见第 `49` 项：为每一个函数，类，模块编写说明文档）。当代码发现一个值为 `None` 的参数的时候，就可以为其分配默认值了。

```
def log(message, when=None):
    """Log a message with a timestamp.

    Args:
        message: Message to print
        when: datetime of when the message occurred.
            Default to the present time.
    """
    when = datetime.now() if when is None else when
    print("%s: %s" %(when, message))

# 测试

log('Hi there!')
sleep(0.1)
log('Hi again!')
>>>
2014-11-15 21:10:10.472303: Hi there!
2014-11-15 21:10:10.473395: Hi again!
```

当参数容易发生改变的时候使用None作为默认参数值尤其重要。例如：你想加载一个被编码为 `JSON` 的数据值，如果解码的时候失败了，你想默认返回一个空字典，你就可以尝试下面的方法。

```
def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

但是这段代码出现的问题和上面的那个时间戳是一样的。默认的空字典被所有调用这个函数的函数共用，因为默认值只是在本模块被加载的时候执行了一次，或许这会导致让人吃惊的现象哦。

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

本想得到两个不同的单值字典，最后却得到了一致的双值字典结果。修改一个的话很明显也会改变另一个。罪魁祸首就是 `foo` 和 `bar` 都等价于默认参数，他们是同一个字典对象。

```
assert foo is bar
```

解决办法就是对关键字参数设置默认值 `None` 并且记录在该函数的说明文档中。

```
def decode(data, default=None):
    """Load JSON data from string.

    Args:
        data: JSON data to be decoded.
        default: Value to return if decoding fails.
            Defaults to an empty dictionary.
    """

    if default is None:
        default = {}
    try:
        return json.loads(data)
    except ValueError:
        return default

# 现在测试一下
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

备忘录

- 默认参数只会被赋值一次：在其所在模块被加载的过程中,这有可能导致一些奇怪的现象。
- 使用 `None` 作为关键字参数的默认值会有一个动态值。要在该函数的说明文档中详细的记录一下。

仅强调关键字参数

在 `Python` 中通过关键字来传递参数值是一个很强大的特征（详见第 `19` 项：使用关键字参数来提供可选行为操作）。关键字参数的灵活性可以帮助你书写更加简洁，易读的代码。

例如：你可能想用一个数除以另一个数，但是在特殊情况下，你需要非常的小心。你可能想忽略 `ZeroDivisionError` 异常，使用无穷值返回来替代。又有时，你想忽略 `OverflowError` 异常返回零来替代。

```
def safe_division(number, divisor, ignore_overflow, ignore_zero_
division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise

# 使用起来也是直截了当，调用的时候会忽略除数的float overflow，并返回0;或
者忽略除数为零，返回无限值
result = safe_division(1, 10**500, True, False)
print(result)
result = safe_division(1, 0, False, True)
print(result)
>>>
0.0
inf
```

容易使人疑惑的问题就是两个布尔类型的控制异常忽略的参数的位置。改变这一现状的一个方式就是使用关键字参数来改善代码的可读性。默认地，这个函数可能过于谨慎了，并且可能重新触发异常。

```
def safe_division_b(number, divisior, ignore_overflow=False, ign
ore_zero_division=False)
    # ......
```

这样的话调用方就可以指定关键字来决定到底要忽略哪一个操作，或者来覆盖默认的行为。

```
safe_division_b(1, 10**500, ignore_overflow=True)
safe_division_b(1, 10**500, ignore_zer0_division=True)
```

然而问题是，当关键字参数变成了可选的时候，就不能够强迫调用者使用关键字参数来简化代码了。不过还好，新定义的这个 `safe_division_b` 函数仍然可以通过位置参数的调用来工作。

```
safe_division_b(1, 10*500, True, False)
```

当函数变得像上面代码中展示的那样复杂的时候，最好是需要调用者对他们自己的意图非常的了解。如果都不知道自己要做什么，又怎么能做得好呢？
在 `Python3` 中，你可以只使用关键字参数来使你的代码变得更加的整洁。这些参数也仅仅只能通过关键字参数赋值的形式被使用，而不是通过位置参数赋值的方式。

```
def safe_division_c(number, division, *, ignore_overflow=False,
ignore_zero_division=False):
    # ......

# 1.然后现在如果还是使用位置参数赋值的方式的话，函数就不会正常地工作了。
safe_division_c(1, 10**500, True, False)
>>>
TypeError: safe_division_c() takes 2 position arguments but 4 wa
re given.

# 2. 现在使用关键字参数赋值的方式的话，就可以得到正常的预期结果了。
safe_division_c(1, 0, ignore_zero_division=True)
try:
    safe_division_c(1, 0)
except ZeroDivisionError:
    pass
```

Python2 中的只使用关键字参数

不幸的是，我们家的 `Python2` 中也是没有明确的像 `Python3` 中的 `keyword-only` 语法。不过伟大的程序员们总是可以想到替代方案嘛。那就是通过在参数列表中使用 `**` 操作符来帮助无效调用来触发 `TypeError` 。其实 `**` 操作符和 `*` 操作符起到的作用类似（详见第 `18` 项：使用可变位置参数减少代码干扰），除了接收可变数量的位置参数以外，它还可以接收任意数量的关键字参数。

```
# Python2
def print_args(*args, **kwargs):
    print 'Positional:', args
    print 'Keyword:', kwargs

print_args(1, 2, foo='bar', stuff='meep')
>>>
Positional: (1, 2)
Keyword: {'foo': 'bar', 'stuff': 'meep'}
```

为了实现在 Python2 中的 keyword-only 性质的 safe_division 函数，我们就需要使用 ** 操作符了。然后从 kwargs 字典中去除我们预期的关键字参数，使用 pop 方法的第二个参数来指定默认的值，这对关键字对应的值出现丢失的情况非常的适用。最终，确保 kwargs 列表中没有所需的关键字的遗留来防止调用方提供无效的参数。

```
# Python 2
def safe_division(number, divisor, **kwargs):
    ignore_overflow = kwargs.pop('ignore_overflow', False)
    ignore_zero_division = kwargs.pop('ignore_zero_division', Fa
lse)
    if kwargs:
        raise TypeError("Unexpected **kwargs: %r"%kwargs)
    # ...

# 测试
safe_division(1, 10)
safe_division(1, 0, ignore_zero_division=True)
safe_division(1, 10**500, ignore_overflow=True)
# 而想通过位置参数赋值，就不会正常的运行了
safe_division(1, 0, False, True)
>>>
TypeError：safe_division() takes 2 positional arguments but 4 we
re given.

# 而输入了非预期的关键字的时候，也会触发类型错误
safe_division(0, 0, unexpected=True)
>>>
TypeErroe: Unexpected **kwargs: {'unexpected': True}
```

备忘录

- 关键字参数使得函数调用的意图更加的清晰，明显。
- 适用 keyword-only 参数可以强迫函数调用者提供关键字来赋值，这样对于容易使人疑惑的函数参数很有效，尤其适用于接收多个布尔变量的情况。
- Python3 中有明确的 keyword-only 函数语法。
- Python2 中可以通过 **kwargs 模拟实现 keyword-only 函数语法,并且人

工的触发 `TypeError` 异常。

- `keyword-only` 在函数参数列表中的位置很重要，这点大家尤其应该明白！

类和继承

作为一门面向对象的编程语言，`Python` 支持面向对象的全部的特征，例如继承，多态和封装。使用 `Python` 完成一件事情经常会需要写一些新的类，并且定义他们相互作用的接口和架构。

`Python` 的类和继承使得你用对象来表示程序的意图非常的容易。它总是允许你改善和拓展里面的方法。对于经常改变的需求，这就提供了一个很大的灵活性。了解如何很好的使用它们可以帮助你编写易于维护的代码。

使用字典和元组来编写工具类

`Python` 中内置的字典类型对于一个对象整个生命期的动态内部状态维护来说，超级好用。至于动态，我的意思是你需要记录一些意想不到的组标识符的情况。例如：你想记录一下一些事先不知道名字的学生的成绩。你就可以定义一个类来把这些名字存储到一个字典中，而不是为每一个学生使用预定义的属性。

```
class SimpleGradebook(object):


    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grade[name].append(score)

    def average_grade(self, name):
        grades = self._grade[name]
        return sum(grades) / len(grades)

# 测试
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
# ...
print(book.average_grade('Isaac Newton'))
>>>
90.0
```

正是由于字典很容易被使用，以至于对字典过度的拓展会导致代码越来越脆弱。例如：你想拓展一下 `SimpleGradebook` 类来根据科目保存成绩的学生的集合,而不再是整体性的存储。你就可以通过修改 `_grade` 字典来匹配学生姓名，使用另一个字

典来包含成绩。而最里面的这个字典将匹配科目（ `keys`)和成绩(`values`)。说起来不是很容易理解，下面我们看下代码就明白了。

```
class BySubjectGradebook(object):


    def __init__(seld):
        self._grade = {}

    def add_student(self, name):
        """这看起来很是直截了当，但是貌似使得report_grade和average_gra
de变得更加复杂了，
        不过还好，这些都是可以被管理的。
        """
        self._grade[name] = {}

    def report_grade(self, name, subject, grade):
        by_subject = self._grade[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grade[name]
        total, count = 0, 0
        for grades in by_subject.values():
            total += sum(grades)
            count += len(grades)
        return total / count

# 测试
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
```

现在，想象一下你的需求又改变了。你还想根据班级内总体的成绩来追踪每个门类分数所占的比重，所以期中，期末考试相比于平时的测验而言更为重要。实现这个功能的一个方式是改变最内部的那个字典，而不是让其关联着科目（ `key`)和成绩（ `values`)。我们可以使用元组（ `tuple`)来作为成绩（ `values`)。

```
class WeightGradebook(object):
    """ 虽然改变report_grade方法看起来很简单，只是使用了一下元组。
    然而average_grade方法现在却由于循环嵌套循环而难于阅读。
    """

    #......

    def report_grade(self, name, subject, score, weight):
        by_subject = self._grade[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append((score, weight))

    def average_grade(self, name):
        by_subject = self._grade[name]
        for score_sum, score in by_subject.items():
            subject_avg, total_weight = 0, 0
            for score, weight, in score:
                #......
        return score_sum / score_count
```

这个类使用起来貌似也变的超级复杂了，并且每个位置参数代表了什么意思也不明不白的。

```
book.report_grade('Albert Einstein', 'Math', 80, 0.10)
```

当你看到这么复杂的代码时，就说明从单纯的使用字典和元组到使用多个类的分层架构的时候到了。

首先，你并不知道你需要提供对成绩权重的支持，所以这样复杂的可拓展的工具类是没有必要的。 `Python` 内置的字典和元组使得一层层的添加拓展变的很容易。然后就成就了上面那样复杂的一个类。但是这就是你恰恰应该避免的多级嵌套（俗话说就是字典内部又包含了字典）。这将使得你的代码变得难于阅读，难于维护。

一旦你意识到了这样做的复杂性，立刻将这个类分解成多个类是你最好的选择。这可以让你提供一个定义良好的接口，更好的封装数据。当然也允许你在接口和实现之间创建抽象层。这样，即使需求变化再多，你的代码也能灵活的应对。

重构成多个类

你可以从依赖树的底端开始，将其划分成多个类：一个单独的成绩类好像对于如此一个简单的信息权重忒大了。一个元组，使用元组似乎很合适，因为成绩是不会改变的饿了，这刚好符合元组的特性。这里，我使用一个元组（`score`, `weight`）来追踪列表中的成绩信息。

```
grades = []
grades.append((95, 0.45))
# ...
total = sum(score * weight  for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

看起来能满足我们的需求了，但是尴尬的是纯元组是位置相关的。当你想为成绩附加更多的信息，比如老师记录什么的，你就需要重写所有用到了这个二元元组来让其支持三元元组。这里我使用了`_`(下划线变量名称，`Python`对未用到的变量的别称)来捕获元组中的这第三个实体，然后仅仅是忽略它。

```
grades = []
grades.append((95, 0.45, 'Great Job'))
# ...
total = sum(score * weight  for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

这种元组拓展的模式越来越像深层的字典方式了，一旦你发现了这点，就是时候该考虑换个方法了。在`collections`模块中的`namedtuple`类型就是你所需要的了，它能够让你轻松的定义一个小型的，但是不变的数据类。

```
import collections
Grade = collections.namedtuple('Grade', ('score', 'weight')
```

这些类既可以通过位置参数被构造，也可以通过关键字参数来构造。字段也是可以根据命名的属性来访问的到。拥有命名属性使得其今后可以很容易地从 `namedtuple` 中移到你自己的类中，前提是如果你的类需要再次被改变，或者添加一些行为到这个简单的数据容器中的话。

下一步就是写一个代表了单独科目的包含了成绩的类。

```
class Subject(object):


    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade, in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

接下来你要写一个类来代表一个学生学了哪些科目。

```
class Stduent(object):


    def __init__(self):
        self._subjects = {}

    def subject(self, name):
        if name not in self._subjects:
            self._subjects[name] = Subject()
        return self._subjects[name]

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

最后，写个包含了所有根据学生名称的容器类就可以了。

```
class Gradebook(object):


    def __init__(self):
        self._student = {}

    def student(self, name):
        if name not in self._students:
            self._students[name] = Student()
        return self._students[name]
```

虽然从代码量上来看，这次的代码量基本上是前面那个的两三倍了，但是从代码的可拓展性和可维护性上来看，使用多个类来实现都比前面那个嵌套字典的例子强上好多。

```
book = Gradebook()
albert = book.student('Albert Einstein')
math = albert.subject('Math')
math.report_grade(80, 0.10)
# ...
print(albert.average_grade())
>>>
81.5
```

如果有需要，你还可以写向后兼容的方法来帮助原来的代码可用，更新到现在的分层架构实现上。

备忘录

- 避免字典中嵌套字典，或者长度较大的元组。
- 在一个整类（类似于前面第一个复杂类那样）之前考虑使用 `namedtuple` 制作轻量，不易发生变化的容器。
- 当内部的字典关系变得复杂的时候将代码重构到多个工具类中。

对于简单接口使用函数而不是类

Python 中的许多内置的 API 都允许你通过向函数传递参数来自定义行为。这些被 API 使用的 hooks 将会在它们运行的时候回调给你的代码。例如： list 类型的排序方法中有一个可选的 key 参数来决定排序过程中每个下标的值。这里，我使用一个 lambda 表达式作为这个键钩子，根据名字中字符的长度来为这个集合排序。

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=lambda x: len(x))
print(names)
>>>
['Plato', Socrates', 'Aristotle', 'Archimedes']
```

在其他的编程语言中，你可能期望一个抽象类作为这个 hooks 。但是在 Python 中，许多的 hooks 都是些无状态的有良好定义参数和返回值的函数。而对于 hooks 而言，使用函数是很理想的。因为更容易藐视，相对于类而言定义起来也更加的简单。函数可以作为钩子来工作是因为 Python 有 first-class 函数：在编程的时候函数，方法可以像其他的变量值一样被引用，或者被传递给其他的函数。

例如：你想定制 defaultdict 类的一些行为（详见第46项：使用内置的算法和数据结构）。这个数据结构允许你提供一个函数，这个函数会在字典中的缺省值被访问的时候调用。而且为字典中的这个缺省键来返回一个默认值。这里，我顶一个一个 hook 来打印每次缺省键的情况，并返回一个默认的值0.

```
def log_missing():
    print("Key added")
    return 0
```

给定一个初始化的字典和一组合理的增量，我可以导致 log_missing 函数执行并打印两次信息（这里是 red 和 orange ）。

```
current = {'green': 12, 'blue': 3}
incremetns = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9)
]
result = defaultdict(log_missing, current)
print("Before:", dict(result))
for key, amount in increments:
    result[key] += amount
print("After:", dict(result))
>>>
Before: {'green': 12, 'blue': 3}
Key added
Key added
After: {'orange': 9, 'green': 12, 'blue': 20, 'red': 5}
```

提供像上面的 `log_missing` 函数可以使得 `API` 更容易被构建和测试，因为它们从确定性的行为中分离了副作用。例如：你想转给 `defaultdict` 的默认值 `hook` 来为缺省的 `key` 计数。一个实现的方式就是使用状态闭包（详见第 `15` 项：了解闭包与变量作用于的联系）。这里，我定义了一个使用了这样的一个闭包的工具函数来作为默认值 `hook` 。

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count  # 状态闭包
        added_count  += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in crements:
    result[key] += amount

    return result, added_count
```

运行代码可以产生预期的结果 `2` ，即使 `defaultdict` 对缺省的 `hook` 状态保持情况一无所知。这也是接口接受简单函数而受益的有一个例子。可以通过在随后的闭包中隐藏状态来很容易地添加功能。

```
result, count = increment_with_report(current, increments)
assert count == 2
```

为状态 `hook` 定义一个闭包的问题就在于：相比于无状态函数而言其可阅读性变差了。另一个方法就是定义一个小类来封装你想追踪的状态信息。

```
class CountMissing(object):


    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0
```

在其他的编程语言中，你可能预期现在的 `defaultdict` 被修改来容纳 `CountMissing` 这个接口。但是在 `Python` 中，感谢 `first-class` 函数的存在，你可以直接引用 `CountMissing.missing` 方法，将其作为一个对象作为默认 `hook` 来传给 `defaultdict` 函数。而实现这样的一个函数来满足需求也是如此的简单。

```
counter = CountMissing()
result = defaultdict(counter.missing, current)
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

使用像这样的一个工具类来提供状态闭包相对于前面的 `increment_with_report` 函数更为清晰，简洁。然而，独立地看， `CountMissing` 类是干什么的还不是特别的明显。是谁够早

了 `CountMissing` 对象呢？谁调用了 `missing` 方法？类中的那些公共方法又被添加到功能区了呢？知道你看到了使用它的 `defaultdict` 函数，你才会明白到底是干什么的，而之前都是不知道的。

为了改善这个问题， `Python` 允许类来定义 `__call__` 这个特殊的方法。它允许一个对象像被函数一样来被调用。这样的一个实例也引起了 `callable` 这个内 `True` 的事实。

```
class BetterCountMissing(object):


    def __init__(self):
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0

counter = BetterCountMissing()
counter()
assert callable(counter)
# 这里我使用一个BetterCountMissing实例作为defaultdict函数的默认的hook值来追踪缺省值被添加的次数。
counter = BetterCountMissing()
result = defaultdict(counter, current)
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

现在，比单纯的使用 `CountMissing.missing` 的案例清晰多了吧。 `__call__` 方法表明本类将会在某处作为函数的参数来使用时也是可以的。
`BetterCountMissing` 类使得第一次看这段代码的读者更好的把握其行为。对于该类来追踪状态闭包的这个行为提供了强有力的提示。

最好的是，当你调用 `__call__` 方法的时候，defaultdict函数仍然对要发生什么不甚了解。 `defaultdict` 需要的仅仅是一个函数作为其默认值 `hook` 就足够了，不会管你传进来的到底是什么函数。 `Python` 提供了很多的方式来来满足你需要完成的依赖于简单函数接口的事务，这很方便。

备忘录

- 在 `Python` 中，不需要定义或实现什么类，对于简单接口组件而言，函数就足够了。
- `Python` 中引用函数和方法的原因就在于它们是 `first-class` ，可以直接的被运用在表达式中。
- 特殊方法 `__call__` 允许你像调用函数一样调用一个对象实例。
- 当你需要一个函数来维护状态信息的时候，考虑一个定义了 `__call__` 方法的状态闭包类哦（详见第 `15` 项：了解闭包是怎样与变量作用域的联系）。

使用@classmethod多态性构造对象

多态

在 `Python` 中，不仅对象支持多态，类也是支持多态的。这意味着什么呢？又有什么好处咧？

多态是一个对于分层良好的类树中，不同类之间相同名称的方法却实现了不同的功能的体现。这也使得许多提供了不同功能的类来来实现相同的接口或者抽象的基类（详见第 `28` 项：从 `ollections.abc` 继承来实现自定义的容器类型）。

例如：你想写一个 `MapReduce` 的实现类，并且使用一个普通类作为输入数据。这里我定义了一个这样一个带有 `read` 方法的类，但是为了下面的继承，它必须是一个超类。

```
class InputData(object):


    def read(self):
        raise NotImplementedError

# 现在实现一个继承了InputData的子类，并使用read方法来读取电脑上硬盘中的数据。
class PathInputData(InputData):


    def ____init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        return open(self.path).read()
```

你可以实现很多的像 `PathInputData` 这样的子类，每一个都可是实现标准接口中的 `read` 方法来返回他们各自处理过的数据。相比于从磁盘上 `read` 数据，其他的 `InputData` 子类可以从网络，解压缩的透明数据等等。你还可能使用一个相似

的 `MapREduce` 的抽象的接口，来以一个标准的方式来处理输入数据。

```
class Worker(object):


    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

# 现在可以定义一个子类来实现一个特殊的我想实现的MapReduce函数：一个简单的行数的计数器。
class LineCountWorker(Worker):


    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

    def reduce(self, other):
        self.result += other.result
```

看起来这个实现已经很不错了，但是我应经到达了极限了。什么将这些片段相互联系起来的呢？我有一系列合理的接口和抽象类，但是可重用性却有点差。那么什么对 `MapReduce` 的构建和策划负责呢？怎样才能更简洁，更有效率，这值得我们思考一番。

人工构建和连接这些对象的最简单的方式就是使用一些工具函数。这里，我列出了一个目录下的内容，并且为每一个它包含的文件构造了一个 `PathInputData` 实例。

```
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))

# 下一步就是创建LineCountWorker实例来使用generate_inputs返回的InputDa
ta实例。
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWork(input_data))
    return workers

# 我通过多线程的方式取出map中的元素之星这些Worker的实例（详见第37项：阻塞I
O的情况下使用线程，避免并行计算）。然后，我调用了reduce方法来重复的把临时计
算结果累加到最终结果上。
def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    for thread in threads: thread.join()

    first, reset in rest:
        first.reduce(worker)
    return first.result

# 每一步的调用看起来很繁琐，于是我封装了一个函数来交替的执行全部任务。
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

现在代码基本上已经算是编写完毕了，下面的任务就是测试我们的代码，看看到底符不符合我们的预期。

```
from tempfile import TemporaryDirectory

def write_test_files(tmpdir):
    # ...

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    result = mapreduce(tmpdir)

print("There are :", result, "lines")
>>>
There are : 4360 lines
```

虽然我们可以通过这段代码获取我们想要的结果，但是却存在一个巨大的问题，那就是 `mapreduce` 函数不够通用。如果你想编写另一个 `InputData` 或者 `Worker` 子类，你就不得不要重写 `generate_inputs` ， `create_workers` 以及 `mapreduce` 函数了。

解决这个问题我们需要一个通用的构造对象的方式。在别的编程语言中，你需要重载构造函数来解决这个问题，需要每一个 `InputData` 的子类提供一个特殊的构造方法，这样才能借助工具方法来更加通用的编排 `MapReduce` 类。然而问题就是 `Python` 中只允许使用 `__init__` 方法来构造，所以要兼容每一个 `nputData` 的子类是不可能的了。

@classmethod

下面终于到正题了，一个比较好的解决办法就是使用 `@classmethod` 多态性。和我在多态性的那个例子中对 `Input.read` 解释类似，除了这个是被附加到整个类上而不是它们的构造器对象上。这样说起来有点让人摸不着头脑，下面我就把这个想法附加到 `MapReduce` 类上吧。这里，我用一个通用的类方法拓展了 `InputData` 类，那就是借助于普通接口来创建 `InputData` 的实例。

```
class GenericInputData(object):


    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

我通过接受一个配置参数信息的集合的函数 `generate_inputs` 来创建 `InputData` 子类来实现，这里我使用 `config` 把为输入的文件转换成目录路径的集合。

```
class PathInputData(GenericInputData):


    # ...

    def read(self):
        return open(self.path).read()

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

相似的，我可以使用 `create_workers` 改善 `GenericWorker` 类。这里我使用 `input_class` 参数（必须为 `GenericInputData` 类的子类）来生成必须的输入。我使用 `cls` 方法来作为一个通用的构造器来构造了 `GenericWorker` 的实例。

```
class GenericWorker(object):


    # ...

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers
```

上面多态类中的 `input_class.generate_inputs` 方法的调用就是我尝试着讲解的了。其实也就是怎么可以不通过 `__init__` 方法来实现类似于重载的构造方法的体现。没有什么高深的地方，用得多了自然也就熟悉了。

在我的 `GenericWorker` 子类中，除了改变其父类的行为之外，也没什么了。

```
class LineCountWorker(GenericWorker):
    # ...

    # 最终，我只需要重写一下mapreduce方法就可以完成这个通用类的编写了

    def mapreduce(worker_class, input_class, config):
        workers = worker_class.create_workers(input_class, confi
g)
        return execute(workers)


# 测试部分的代码与之前的相比，除了参数树木上的不同，也没什么区别了。
with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
    result = mapreduce(LineCountWorker, PathInputData, config)
```

现在，你可以如你所愿地编写其他
的 `GenericInputData` 和 `GenericWorker` 类，而不需要修改代码了。

备忘录

- `Python` 的每个类只支持单个的构造方法，`__init__` 。
- 使用 `@classmethod` 可以为你的类定义可替代构造方法的方法。
- 类的多态为具体子类的组合提供了一种更加通用的方式。

使用super关键字初始化父类

元类和属性

并行与并发

内置模块

合作

产品