互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 4 章 Python 技巧

引言

感谢：David Ascher，ActiveState，*Learning Python* 合著者之一

编程语言就像自然语言一样。对于通晓多种语言的人来说，每种语言都有自己独特的一些属性和特征。俄语和法语以奔放著称，而英语则以精确性和活力著称：不像学院派的法语，英语总是为了满足使用上的需求而不停地增加新词，比如“carjacking”、“earwitness”、“snailmail”、“email”、“googlewhacking”和“blogging”。在计算机语言的世界中，Perl 以它随心所欲的自由性而闻名：TMTOWTDI（There's More Than One Way To Do It）是 Perl 程序员对它的最好的赞美。在 Perl 语言和 APL 社区中，简明扼要被认为是一个非常重要的优点。正如你在本书的各个章节的中读到的那样，作为对比，Python 程序员总是不断表达他们对清晰和优雅的推崇。一个非常有名的 Perl 高手曾经跟我说过，Python 确实漂亮得多，但是 Perl 更好玩。我不得不同意他，Python 对美学的追求无处不在（通过良好的设计和定义），而 Perl 则表达了不可救药的幽默感。

我之所以要在引言开头提到这些看上去与主题毫无关联的语言特性，是因为本章的内容将会涉及 Python 的设计美学和语言的活力。如果本书是关于 Perl 的，那么类似于本章这样关于语言窍门的内容，多半会让读者挠头、深思，然后发出一个“啊哈”的赞叹，说不定还会哈哈大笑，因为读者会慢慢领会到各个小把戏后面的天才思想。作为对比，在本章的大部分内容中，作者都在尝试展示语言的优雅特性，因为他认为这样美好的东西还没有引起足够的注意和赞美。就像我一样，我是温哥华的一个骄傲的居民，我会自告奋勇地带着旅游者，向他们展示这个美丽城市的所有的美好的地方，从公园到海滩，再到山峦，而一个 Python 用户则可能会找到他的朋友和同事，兴冲冲地说，“你一定得瞧瞧这个！”对于我和一些我了解的程序员来说，在 Python 中写程序有时候是一种喜悦的分享，而不是一种迫于竞争的追求。学习它的新特性，欣赏它的设计，它的优雅，它的洗练，完全是一种乐趣，但同时教授它的各种细节，讲解它的微妙用法，其实也是一种极大的快乐。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

还得提一提本章的历史：最初在为第 1 版决定各章内容时，我们认为应该有一系列的章节，每章内容都基于不同的目的——比如，蛋奶酥、果馅饼、炖小牛肘。那些章节都自然地分为各个不同的类别，比如甜食、开胃小食、肉食，或者没那么美好的，让人没食欲的文件，算法等。所以，我们选了一系列可能的分类，并在 Zope 的网站上征集内容，之后我完全地打开了思潮的闸门。

结果，很快我们就发现有一些内容无法归入任何预先设置的分类。这是可以解释的，拿烹饪来说吧。本章的内容，可以用制作掺油面糊来做比喻（如果你没有法式烹饪的背景，我要略作解释，这是一种面粉和油脂的混合物，一般用于制作酱汁），捏面团、加面、加入蛋清、在平底锅上翻面、烫煮，这些常见的技巧和知识，任何一个合格的厨师都知道，但是任何一本菜谱上都不会有。这些知识和技巧常常被用来准备菜肴，但却难于归档到任何一种菜肴的制作方法。如果你是一个新手厨师，想尝试某个很棒的菜谱，往往会铩羽而归，那是因为菜谱的作者已经假设你具备那些必要的知识了。作者们只会在类似 *Cooking for Divorced Middle-Aged Man* 的书中解释那些必要的知识（而且通常还配上插图）。但我们并不想把这些宝贵的内容从书中剔除，所以，新的一章就这么诞生了（抱歉，没有配上精美的插图）。

在本书第 1 版的本章的引言中，我是这么说的：

我相信本章中的内容是对时间最敏感的部分。这是因为这些技巧和窍门看上去和最近的语言特性有较高的关联度。

我为我的预言感到骄傲，我的说法完全正确。新版本的内容，完全关注当前的语言特点，最初的一些早期内容已经不再适用了。在本书第 1 版发行之后，Python 已经发布了两个重要的版本，Python 2.3 和 Python 2.4，这门语言的发展也使得初期的一些内容需要适应新特性和新的库函数，语言本身变得更简洁、紧凑和强大，同时有无数新出现的有趣的细节值得我们去发掘。

总之，本章约有一半内容（和本书的比例差不多）是全新的，另外一半则对第一版的内容进行了改编（大多是简化）。由于这些简化工作，以及把本书的重点放到仅有的两个语言版本（2.3 和 2.4）之上，而不再考虑和覆盖第 1 版涉及的很多旧的语言版本，本章比第 1 版的内容多出了超过 1/3，和本书的总的内容增幅差不多。

值得注意的是，很多在本章中经过改编的内容都涉及一些最基本的，没有改变的语言特性：赋值的语义、绑定、复制、引用；序列；字典。所有这些都是进行 Python 编程的最关键的要素，不过我也有点疑惑，不知道 Python 在以后几年的发展中会不会考虑改动这些关键特性。

4.1 对象拷贝

感谢：Anna Martelli Ravenscroft、Peter Cogolo

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

任务

你想拷贝某对象。不过，当你对一个对象赋值，将其作为参数传递，或者作为结果返回时，Python 通常会使用指向原对象的引用，并不是真正的拷贝。

解决方案

Python 标准库的 copy 模块提供了两个函数来创建拷贝。第一个常用的函数叫做 copy，它会返回一个具有同样的内容和属性的对象：

```
import copy
new_list = copy.copy(existing_list)
```

某些特殊的时候，你可能会需要对象中的属性和内容被分别地和递归地拷贝，可以使用 deepcopy：

```
import copy
new_list_of_dicts = copy.deepcopy(existing_list_of_dicts)
```

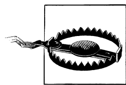
讨论

当给一个对象赋值（或者将其作为参数传递，或者作为结果返回时）时，Python（像 Java 一样）使用了一个指向原对象的引用，并不是真正的拷贝。其他一些语言则在每次赋值时都进行拷贝操作。Python 从来不为赋值操作进行“隐式”的拷贝：要得到一个拷贝，必须明确地要求，需要的是拷贝。

Python 的行为模式既简单又快速，而且很一致。如果需要拷贝但却不明确地要求，很有可能会遇到麻烦。举个例子：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b.append(5)
>>> print a, b
[1, 2, 3, 5] [1, 2, 3, 5]
```

这里，名字 a 和 b 都引用到同样的对象（列表 a），所以，无论我们通过哪个名字修改了对象的内容，之后，无论通过哪个名字来查看对象，修改结果都是一样的。这个过程中，并没有一个原始的，未被修改的拷贝。



警告

要想成为一个好的 Python 程序员，必须了解修改对象和将对象赋值给变量的区别，赋值使用的是引用。这两种操作相互之间并没有什么关联。类似于 a=[ ]这样的语句，是对名字 a 做了重新绑定，但却不会影响原先绑定到 a 的对象。因此，这里完全没有引用和拷贝的问题：只有当需要修改某些对象的时候，引用和拷贝才有可能成为问题。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果想修改一个对象，但又需要不改动原对象，必须做一个拷贝。如同前面提到的，Python 标准库的 copy 模块提供了两个函数来制作拷贝。一般情况下，可以使用 copy.copy，它完成的是对一个对象的浅拷贝——虽然生成了一个新对象，但是对象内部的属性和内容仍然引用原对象，这样的操作速度很快而且节省内存。

如果想从原对象真正地“复制”一个全新的对象，或者想修改的是对象内部的属性和内容，而不是对象本身，那么浅拷贝不能满足你的需求：

```
>>> list_of_lists = [ ['a'], [1, 2], ['z', 23] ]
>>> copy_lol = copy.copy(lists_of_lists)
>>> copy_lol[1].append('boo')
>>> print list_of_lists, copy_lol
[['a'], [1, 2, 'boo'], ['z', 23]] [['a'], [1, 2, 'boo'], ['z', 23]]
```

这里，名字 list_of_lists 和 copy_lol 指向了两个不同的对象（两个列表），所以我们可以分别修改它们而不互相影响。然而，list_of_lists 中的元素同样也是 copy_lol 的对应元素，所以无论通过哪个名字，一旦我们通过索引修改了元素，以后无论通过哪个名字访问其内容，我们会看到修改已经对两者同时生效了。

所以，如果需要拷贝一些容器对象，还必须递归地拷贝其内部引用的对象（包括所有的元素、属性、元素的元素、元素的属性等），使用 copy.deepcopy 这种深拷贝操作，会消耗相当的时间和内存，但如果深拷贝确实是需要的效果，你别无选择。而要实现深拷贝，copy.deepcopy 是唯一可用的方法。

对于普通的浅拷贝，你可能会需要另外一些方法实现同样的功能，当然，假设你知道想拷贝的对象的类型。对于列表 L，调用 list(L)；对于字典 d，调用 dict(d)；为了拷贝集合 s（Python 2.4 中已经引入了内建的类型 set），调用 set(s)。（由于 list、dict 和 2.4 中的 set 已经是内建的名字了，你无须做任何“准备工作”就可以直接使用。）你现在应该能够领会到这种通用的方法了：为了拷贝一个可被拷贝（copyable）的对象 o，该对象属于内建的 Python 类型 t，可以简单地调用 t(o)来创建拷贝。字典还提供了另一个浅拷贝方法：d.copy()，它做的事情和 dict(d)完全一样。不过在这两者之中，我建议你选择 dict(d)：这种方式可以和其他类型的拷贝方式统一起来，而且也短一些，少一个字符。

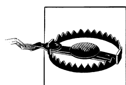
对于任意类型或者类的实例，无论是自己编写的类或者从库中引入的类，一般用 copy.copy 就行了。如果是自己编写的类，通常也不值得专门定义一个 copy 或者 clone 方法，如果想定制自己的类的浅拷贝方式，可以提供一个特殊的__copy__方法（关于实现这个方法的细节请参看 6.9 节），或者特殊的__getstate__和__setstate__方法（参考 7.4 节中关于这些特殊方法的细节，这些方法也有助于深拷贝和序列化——比如对实例进行 pickling 操作——的实现）。如果想实现自己的类的独有的深拷贝方式，需要提供一个特殊的__deepcopy__方法（参考 6.9 节）。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

注意，没有必要拷贝那些不可改变的对象（字符串、数字、元组等），因为完全不必担心会不经意改动它们。如果尝试进行拷贝操作，仍然会得到原对象，当然这也不会有太大害处，只不过浪费了一些时间和代码。举个例子：

```
>>> s = 'cat'
>>> t = copy.copy(s)
>>> s is t
True
```

is 操作符检查两个对象是否相同，而不是相等（is 检查对象是否相同；==操作符则检查两个对象是否相等）。对于不可改变的对象来说，检查是否相同几乎没有什么用处（这里我们只是为了展示对不可改变对象调用 copy.copy 是没意义的，但也是无害的）。对于可改变的对象，检查相同性有时却是至关重要的。举个例子，假设你不确定两个名字 a 和 b 是分别指向不同的对象还是引用同一个对象，可以用 a is b 这样一条简单快速的检查语句来找到答案。当需要确保原对象不被改变时，就可以考虑对原对象进行拷贝操作了。



警告

可以使用其他的方法来创建拷贝。给定一个列表 L，使用“整个对象的切片”L[:]以及列表推导[x for x in L]都可以完成对 L 的浅拷贝，而使用添加空列表，L+[]，以及用 L 乘以 1，L*1，诸如此类的方式都只是无谓的浪费时间和内存，直接调用 list(L)速度更快也更清晰。不过，由于某些历史原因以及应用的广泛性，你也应该熟悉 L[:]这种用法。所以，即使我不建议你在自己的代码中这么写，但你也可能在别人的代码中看到这种用法。

类似的，给定一个字典 d，也可以通过循环来创建一个浅拷贝 d1:

```
>>> d1 = { }
>>> for somekey in d:
...     d1[somekey] = d[somekey]
```

或者更简明的方式：d1 = { }; d1.update(d)。不过，这仍然是一种时间和代码上的浪费，除了增加一些迷惑性和降低速度，没有更多的东西。用 d1=dict(d)这样一句话就可以了，又快又轻便。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 copy 模块的内容。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

4.2 通过列表推导构建列表

感谢：Luther Blissett

任务

你想通过操作和处理一个序列（或其他的可迭代对象）中的元素来创建一个新的列表。

解决方案

假设你想通过给某个列表中的每个元素都加上 23 来构建一个新列表。用列表推导可以很直接地实现这个想法：

```
thenewlist = [x + 23 for x in theoldlist]
```

同样，假设需要用某列表中的所有大于 5 的元素来构成一个新列表。用列表推导代码可以写成这样：

```
thenewlist = [x for x in theoldlist if x > 5]
```

当你试图将这两种想法合二为一的时候，可以增加一个 if 子句，并对选定的项使用某些表达式，比如加上 23，这些都可以用一行概括成：

```
thenewlist = [x + 23 for x in theoldlist if x > 5]
```

讨论

优雅、清晰和务实，都是 Python 的核心价值观，列表推导说明了这三点是怎样和谐地统一起来的。事实上，当你直觉地考虑“改变某列表”而不是新建某列表时，列表推导常常是最好的办法。比如，假如需要将某列表 L 中的所有大于 100 的元素设置成 100，最好的方法是：

```
L[:] = [min(x,100) for x in L]
```

上面代码在给一个“整个列表的切片”赋值的同时，修改了该列表对位的数据，而不是试图对名字 L 重新绑定，比如写成 L = …。

当你只是需要执行一个循环的时候，不应该使用列表推导。如果需要循环，那就写相应的循环代码。关于循环列表的例子，参看 4.4 节。第 19 章还会有更多的有关 Python 的迭代的内容。

另外，如果 Python 有内建的操作或者类型能够以更直接的方式实现你的需求，你也不要使用列表推导。比如，为了复制一个列表，用 L1 = list(L)就好，不要这么用：

```
L1 = [x for x in L]
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

类似地，如果想对每个元素都调用一个函数，并使用函数的返回结果，应该用 L1=map(f, L)，而不是 L1 = [f(x) for x in L]。不过大多数情况下，这种列表推导用法也没问题。

在 Python 2.4 中，当序列过长，而你每次只需要获取一个元素的时候，应当考虑使用生成器表达式，而不是列表推导。生成器表达式的语法和列表推导一样，只不过生成器表达式是被()圆括号括起的，而不是方括号[]。比如，如果我们需要计算本节解决方案里的列表的某些元素之和，在 Python 2.3 中可以这么做：

```
total = sum([x + 23 for x in theoldlist if x > 5])
```

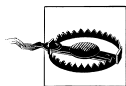
在 Python 2.4 中，代码可以写得更自然一点，方括号可以省略掉（也不用增加多余的圆括号——有调用内建的 sum 函数的圆括号就足够了）：

```
total = sum(x + 23 for x in theoldlist if x > 5)
```

除了更加简洁，这个方法还可以避免一次性将整个列表载入内存，从而在列表特别长的时候，在一定程度上提高处理速度。

更多资料

参考 *Reference Manual* 中 list display（列表推导的另一个名字）和 Python 2.4 中生成器表达式的内容；第 19 章；*Library Reference* 和 *Python in a Nutshell* 中 itertools 模块和内建的 map、filter 和 sum 函数；以及 Haskell，见 http://www.haskell.org。



警告

Python 从函数式语言 Haskell（http://www.haskell.org）借来了列表推导，当然在语法和关键字上有一些差异，而不是完全一样。如果你懂 Haskell，那么小心点，因为 Haskell 的列表推导正如这门语言的其余部分一样，使用惰性求值（lazy evaluation）的方式（也被称为正则序（normal order）或按需调用（call by need））。每一项都只在真正需要的时候才计算。Python，像很多语言一样，使用（包括列表推导和其他部分）主动求值（eager evaluation）方式（也被称为应用序（application order），按值调用（call by value），或者严格求值（strict evaluation））。也就是说，整个列表在列表推导执行的时候就完成了计算，而且结果被尽量长久地保存在内存中以供以后使用。如果你正在将一个 Haskell 程序改写成 Python 程序，而且这个 Haskell 程序用列表推导来表示无限序列或任何一个很长的序列。那么 Python 的列表推导可能不是很合适。不过 Python 2.4 中出现了生成器表达式，它的语义更加接近 Haskell 的惰性求值方式，同样是按需调用，所以它应该是更好的选择。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

4.3 若列表中某元素存在则返回之

感谢：Nestor Nissen、A. Bass

任务

你有一个列表 L，还有一个索引号 i，你希望当 i 是 L 的有效索引时获取 L[i]，若不是有效索引，则返回一个默认值 v。如果 L 是字典，可以使用 L.get(i, v)来满足需求，可是列表并没有 get 这个方法。

解决方案

很明显，我们得自己写个函数，在这里，最简单直接的方法就是最好的方法：

```
def list_get(L, i, v=None):
    if -len(L) <= i < len(L): return L[i]
    else: return v
```

讨论

解决方案中的函数根据 Python 的索引规则来检查 i 的有效性：有效索引只能在大于等于-len(L)和小于 len(L)这个区间中。但如果所有传递给 list_get 函数的参数 i 都是有效的索引，你可能会喜欢另外一种方式：

```
def list_get_egfp(L, i, v=None):
    try: return L[i]
    except IndexError: return v
```

但是，除非传递给此函数的索引绝大多数都是有效索引，否则这个函数（通过某些测试工具测量）将会比解决方案中的 list_get 函数慢 4 倍。因此，这个“获得原谅总是比获得许可容易（easier to get forgiveness than permission，EGFP）”的函数，虽然更具有 Python 的精神和风格，但在这种特殊的情况下，并不值得推荐。

我还试过几个看上去更漂亮、更复杂和更迷惑人的方法，不过，除了更加难于解释和理解之外，它们无一例外地比那个朴实无华的 list_get 函数慢。这里给出一个通用的准则：当你写 Python 程序时，应当倾向于清晰和可读性，而不是紧凑和精炼——选择简单，而不是精巧。只要你坚持这么做，你常常会发现你的代码跑得更快，而且也更强健，更易于维护。在真实世界中，对于 99.9%的应用而言，遵循这个原则要比获得一点速度提升重要的多。

更多资料

Language Reference 和 *Python in a Nutshell* 中关于列表索引的文档。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

4.4 循环访问序列中的元素和索引

感谢：Alex Martelli、Sami Hangaslammi

任务

需要循环访问一个序列，并且每一步都需要知道已经访问到的索引（因为需要重新绑定序列的入口），但 Python 提供的首选的循环方式完全不用依赖索引。

解决方案

内建函数 enumerate 正是为此而生。看例子：

```
for index, item in enumerate(sequence):
    if item > 23:
        sequence[index] = transform(item)
```

它看上去很干净易读，而且比那种通过索引访问元素的方式快：

```
for index in range(len(sequence)):
    if sequence[index] > 23:
        sequence[index] = transform(sequence[index])
```

讨论

循环遍历一个序列是很常见的需求，Python 强烈建议你用一种最直接的方式。事实上这也是最具有 Python 风格的访问序列中每个元素的方式：

```
for item in sequence:
    process(item)
```

而其他一些典型的比较底层的语言，不是用这种直接的循环方式，而是通过序列的索引，根据索引找到每一个对应的子项：

```
for index in range(len(sequence)):
    process(sequence[index])
```

直接的循环方式更加干净、更易读、更快，而且也更通用（因为根据定义，此法可以应用于任何可迭代对象，而根据索引访问的方式则只适用于序列，如列表）。

但是，有时候在循环中，你的确需要同时获得索引和索引对应的子项。一个常见的理由是，你想重新绑定列表的新入口，必须将 thelist[index]赋值为一个新的子项。为了支持这种需求，Python 提供了内建函数 enumerate，它接受任何可迭代的参数，并返回一个迭代器，迭代器产生的是一个（两个子项的元组）形如(index, item)的结果，一次一项。因此你的 for 子句的头部可以写成：

```
for index, item in enumerate(sequence):
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

这样，在 for 的主体中，索引和子项都是可以访问的。

为了帮助你记忆 enumerate 产生的结果，考虑惯用法 d=dict(enumerate(L))。实际上从某种意义上来讲，用此法获得的字典 d 是等价于列表 L 的，因为对于任意一个有效的非负索引 i，d[i] is L[i]都成立。

更多资料

Library Reference 和 *Python in a Nutshell* 中 enumerate 的相关内容；第 19 章。

4.5 在无须共享引用的条件下创建列表的列表

感谢：David Ascher

任务

你想创建一个多维度的列表，且同时避免隐式的引用共享。

解决方案

为了创建列表且避免隐式地共享引用，我们可使用列表推导。举个例子，创建一个 5×10 的全为 0 的阵列：

```
multilist = [[0 for col in range(5)] for row in range(10)]
```

讨论

当 Python 新手首次发现列表乘以一个整数得到的列表是原列表的多次重复时，他会感到非常兴奋，因为看上去这种处理极其优雅。举个例子：

```
>>> alist = [0] * 5
```

很明显，这是很棒的一种获得 1×5 维度的全为 0 的数组的方法。

问题是，一维的任务常常会根据需要扩展成二维的，所以，一个自然的想法是进行如下的处理：

```
>>> multi = [[0] * 5] * 3
>>> print multi
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

看上去它工作完全正常，但是新手们常常会发现自己被 bug 困扰。举个简单的例子，下面的代码片段很有说服力：

```
>>> multi[0][0] = 'oops!'
>>> print multi
[['oops!', 0, 0, 0, 0], ['oops!', 0, 0, 0, 0], ['oops!', 0, 0, 0, 0]]
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

绝大多数程序员都至少在这个问题上栽过一次跟头（见位于 http://www.python.org/doc/FAQ.html#4.50 的 FAQ 条目）。为了理解这个问题，可以将多维列表的创建分解成两个步骤：

```
>>> row = [0] * 5              # row 列表中的 5 个子项都引用 0
>>> multi = [row] * 3          #multi 列表中的 3 个子项都引用 row
```

分解过后的代码片段生成的 multi 列表完全等价于前面的更简洁的[[0]*5]*3 代码片段产生的结果，而且问题也完全一样：如果你给 multi[0][0]指定一个新值，会改变 multi[1][0]的值，multi[2][0]的值…，甚至改变了 row[0]的值！

代码中的注释是理解这个混淆的关键之处。将序列和数字相乘产生的新序列，将含有多个引用原始内容的子项，其数量等于乘数。在 row 的创建中，有无引用被复制完全不重要，因为被引用的是数字，而数字是不可改变的。换句话说，如果对象是不可改变的，则对象和对对象的引用实际上没什么区别。第二行，我们创建了一个新的列表，其中包含了 3 个对[row]的内容的引用，而其内容则是对一个列表的引用。因此，multi 包含了 3 个对列表的引用。这样，当 multi 的第一个子项的第一个子项被修改的时候，你会发现共享的列表的第一个子项被修改了。奇迹就是这么发生的。

而解决方案中所示的列表推导则避免了问题。使用列表推导，没有任何引用共享的问题——你事实上完成了一个真正的嵌套计算。如果确实领会了上面的讨论，你可能会意识到可以不用内层的列表推导，只需要外层的。换句话说，代码这样改一下是不是也可以？

```
multilist = [[0]*5 for row in range(10)]
```

答案是，可以，事实上在最内层使用列表乘法，而在其他层次使用列表推导，速度会快不少——要比示例快两倍。那么我为什么不推荐最后讨论的这种方法呢？答案是：对于这个例子，在 Python 2.3 中，速度的提升是从 57μs 降低到了 24μs，在 Python 2.4 中，从 49μs 降低到了 21μs，运行平台是一台又老又破的个人计算机（AMD Athlon 1.2 GHz CPU，运行 Linux）。优化一个列表的创建操作，从而节省几十μs 并不会对你的应用程序的性能产生什么大的影响：如果你真的需要优化，那就应该去找真正重要的地方，真正对应用程序的整体性能产生巨大影响的地方。所以，我更倾向于解决方案中给出的代码，它更加简单，因为它从内层到外层的列表创建都使用了相同的结构，看上去也更具有对称性，当然也更易读。

更多资料

Library Reference 和 *Python in a Nutshell* 中的内建函数 range 的文档。

4.6 展开一个嵌套的序列

感谢：Luther Blissett、Holger Krekel、Hemanth Sethuram、ParzAspen Aspen

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

任务

序列中的子项可能是序列，子序列的子项仍可能是序列，以此类推，则序列嵌套可以达到任意的深度。需要循环遍历一个序列，将其中所有的子序列展开成一个单一的、只具有基本子项的序列。（一个基本子项或者原子，可以是任何非序列的对象——或者说叶子，假如你认为嵌套序列是一棵树。）

解决方案

我们需要能够判断哪些我们正在处理的子项是需要被展开的，哪些是原子。为了获得通用性，我们使用了一个断定来作为参数，由它来判断子项是否是可以展开的。（断定[predicate]是一个函数，每当我们处理一个元素时就将其应用于该元素并返回一个布尔值：在这里，如果元素是一个需要展开的子序列就返回 True，否则返回 False。）我们假定每一个列表或者元组都是需要被展开的，而其他类型则不是。那么最简单的解决方法就是提供一个递归的生成器：

```
def list_or_tuple(x):
    return isinstance(x, (list, tuple))
def flatten(sequence, to_expand=list_or_tuple):
    for item in sequence:
        if to_expand(item):
            for subitem in flatten(item, to_expand):
                yield subitem
        else:
            yield item
```

讨论

展开一个嵌套的序列，或者等价地，按照顺序“遍历”一棵树的所有叶子，是在各种应用中很常见的任务。如果有一个嵌套的结构，元素都被组织成序列或者子序列，而且，基于某些理由，你并不关心结构本身，需要的只是一个接一个地处理所有的元素。举个例子：

```
for x in flatten([1, 2, [3, [  ], 4, [5, 6], 7, [8,], ], 9]):
    print x,
    输出 1 2 3 4 5 6 7 8 9。
```

这个任务唯一的问题是，怎样在尽量通用的尺度下，判断什么是需要展开的，什么是需要被当做原子的，这其实没有看上去那么容易。所以，我绕开直接的判断，把这个工作交给了一个可调用的断定参数，调用者可以将其传递给 flattern，如果调用者满足于 flatten 简单的默认行为方式，即只展开元组和列表。

在 flatten 所在的模块中，我们还需要提供另一个调用者可能需要用到的断定——它将

展开任何非字符串（无论是普通字符串还是 Unicode）的可迭代对象。字符串是可迭代的，但是绝大多数应用程序还是想把它们当成原子，而不是子序列。

至于判断对象是否可迭代，我们只需要对该对象调用内建的 iter 函数：若该对象是不可迭代的，此函数将抛出 TypeError 异常。为了判断对象是否是类字符串的，我们则简单地检查它是否是 basestring 的实例，当 obj 是 basestring 的任何子类的实例时，isinstance(obj, basestring)的返回值将是 True——这意味着任何类字符串类型。因此，这样的一个断定并不难写：

```
def nonstring_iterable(obj):
    try: iter(obj)
    except TypeError: return False
    else: return not isinstance(obj, basestring)
```

当具体的需求是展开任何可迭代的非字符串对象时，调用者可以选择调用 flatten(seq, nonstring_iterable)。无疑，不把 nonstring_iterable 断定作为 flatten 的默认选项是一个更好的选择：在简单的需求中，如我们前面展示的示例代码片段，使用 nonstring_iterable 会比使用 list_or_tuple 慢 3 倍以上。

我们也可以写一个非递归版本的 flatten。这种写法可以超越 Python 的递归层次的极限，一般不超过几千层。实现无递归遍历的要点是，采用一个明确的后进先出（LIFO）栈。在这个例子中，我们可以用迭代器的列表实现：

```
def flatten(sequence, to_expand=list_or_tuple):
    iterators = [ iter(sequence) ]
    while iterators:
        # 循环当前的最深嵌套（最后）的迭代器
        for item in iterators[-1]:
            if to_expand(item):
                # 找到了子序列，循环子序列的迭代器
                iterators.append(iter(item))
                break
            else:
                yield item
        else:
            # 最深嵌套的迭代器耗尽，回过头来循环它的父迭代器
            iterators.pop( )
```

其中 if 语句块的 if 子句会展开每一个我们需要展开的元素——即我们需要循环遍历的子序列；所以在该子句中，我们将那个子序列的迭代器压入栈的末尾，再通过 break 打断 for 的执行，回到外层的 while，外层 while 会针对我们刚刚压入的新的迭代器执行一个新的 for 语句。else 子句则用于处理那些不需要展开的元素，它直接产生元素本身。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果 for 循环未被打断，for 语句块所属的 else 字句将得以执行——换句话说，当 for 循环完全执行完毕，说明它已经遍历完当前的最新的迭代器。所以，在 else 子句中，我们移除了已经耗尽的嵌套最深（最近）的迭代器，之后外层的 while 循环继续执行，如果栈已经空了，则中止循环，如果栈中还有迭代器，则执行一个新的 for 循环来处理之——正好是上次执行中断的地方，本质上，迭代器的任务就是记忆迭代的状态。

flatten 的非递归实现产生的结果和前面的简单一些的递归版本的结果完全一致。如果你认为非递归实现会比递归方式快，那么你可能会失望：我采用了一系列的测试用例进行观察测量，发现非递归版本要比递归版本慢约 10%。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于序列类型和内建的 iter、isinstance 以及 basestring 的文档。

4.7 在行列表中完成对列的删除和排序

感谢：Jason Whitlark

任务

你有一个包含列表（行）的列表，现在你想获得另一个列表，该列表包含了同样的行，但是其中一些列被删除或者重新排序了。

解决方案

列表推导很适合这个任务。假设你有：

```
listOfRows = [ [1,2,3,4], [5,6,7,8], [9,10,11,12] ]
```

需要另一个有同样行的列表，但是其中第二列被删除了，第三和第四列则互换了位置。我们用一个简单的列表推导来完成任务：

```
newList = [ [row[0], row[3], row[2]] for row in listOfRows ]
```

还有一个方法同样具有可操作性，也许还更优雅一些，即采用一个辅助的序列（列表或元组），此序列的列索引位置正好是需要的顺序。这样，在外层对 listOfRows 进行循环操作的列表推导的内部，又加入了一个嵌套的对辅助序列进行循环操作的内层列表推导。

```
newList = [ [row[ci] for ci in (0, 3, 2)] for row in listofRows ]
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

讨论

我一般用列表的列表来代表二维的阵列。我把这些列表看做以二维阵列的“行”为元素的列表。我常常需要处理这种二维阵列的列，一般是重新排序，有时还会忽略列中的部分内容。尽管列表推导在别处大显神通，但粗略一看，它在这里似乎用处不大（至少我的第一感觉是这样）。

列表推导会产生一个新的列表，而不是修改现有的列表。当需要修改一个现有的列表时，最好的办法是将现有列表的内容赋值为一个列表推导。举个例子，假设要修改 listOfRows，对于本节的任务，可以写成：

```
listOfRows[:] = [ [row[0], row[3], row[2]] for row in listOfRows ]
```

正如本节解决方案的第二个例子所示，还可以考虑使用一个辅助的序列，其中包含的列索引按需要的顺序排列，这比第一个例子用硬编码指定顺序要好。你可能对嵌套的列表推导感到不放心，但事实上它比你想象的更简单安全。如果采用解决方案中的第二种方式，会同时获得更多的通用性，因为可以给辅助序列一个名字，并使用这个名字来对一些行列表进行重新排序，或者将其作为参数传递给一个函数，等等：

```
def pick_and_reorder_columns(listofRows, column_indexes):
    return [ [row[ci] for ci in column_indexes] for row in listofRows ]
columns = 0, 3, 2
newListOfPandas = pick_and_reorder_columns(oldListOfPandas, columns)
newListOfCats = pick_and_reorder_columns(oldListOfCats, columns)
```

上例所做的事情和本节前面的代码片段所做的事情完全一样，不过它操作两个独立的“旧的”列表，并分别获得两个对应的“新的”列表。追求最大程度的通用性是一种难以抵御的诱惑，但在这里，这个 pick_and_reorder_columns 所表现出的通用性似乎恰到好处。

最后一条，在前面用到的一些函数中，一些人喜欢用看上去更漂亮的方式来表示“内层”列表推导，而不采用简单直接的方式，比如：

```
[row[ci] for ci in column_indexes]
```

他们喜欢用内置的 map 函数以及 row 的特殊的__getitem__方法（常被用作被绑定方法）来完成索引子任务，因此他们这样编写代码：

```
map(row.__getitem__, column_indexes)
```

具体到某些不同的 Python 的版本，这种看上去漂亮但又有点让人困扰的方法可能速度会快一些。但我仍然认为体现简洁性的列表推导是最佳方式。

更多资料

Language Reference 和 *Python in a Nutshell* 中的列表推导的文档。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

4.8 二维阵列变换

感谢：Steve Holden、Raymond Hettinger、Attila Vàsàrhelyi、Chris Perkins

任务

需要变换一个列表的列表，将行换成列，列换成行。

解决方案

需要一个列表，其中的每一项都是同样长度的列表，像这样：

```
arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

列表推导提供了简单方便的方法以完成二维阵列的转换：

```
print [[r[col] for r in arr] for col in range(len(arr[0]))]
[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

另一个更快也更让人困惑的方法（输出是一样的）是利用内建函数 zip 实现的：

```
print map(list, zip(*arr))
```

讨论

本节展示了一种简洁而清晰的转换方式，还有一个更快速的备选方案。在需要简洁和清晰并存的时候，列表推导通常是很好的选择，而备选方案利用内建函数 zip 以另外一种方式达到目的，显得很晦涩难懂。

有时，你获得的数据的顺序是不正确的。举个例子，如果使用微软的 ActiveX Data Objects（ADO）数据库接口，由于 Python 和微软的首选实现语言（Visual Basic）在对数组元素排序上的差异，Getrows 方法返回的实际上是 Python 中的列。本节针对这种常见需求提出的两种解决方案，让你有机会在清晰和速度之间进行选择。

在列表推导的解决方案中，内层推导改变的是（从行中）选出的元素，外层推导则影响选择子（selector，即列）。由此实现转换。

而基于 zip 的解决方案，我们使用了*a 语法将 arr 中的每个元素（行），根据顺序，作为分隔开的参数传递给 zip。zip 返回的是元组的列表，其实已经完成了转换。通过 map 调用，我们可以对每个元组调用 list，以获得一个列表的列表。既然我们不能将 zip 的结果直接当做列表使用，我们可以通过使用 itertools.izip 来得到一点改进（因为 izip 并不会将结果当做列表载入内存，而是每次生成一个子项）：

```
import itertools
print map(list, itertools.izip(*arr))
```

不过，对这个例子而言，这一点速度提升也许并不能抵消它所带来的复杂性。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

***args 和**kwds 语法**

*args（*通常紧跟一个标识符，你会看到 a 或者 args 都是标识符）是 Python 用于接收或者传递任意基于位置的参数的语法。当你接收到一个用这种语法描述的参数时（比如你在函数的 def 语句中对函数签名使用了星号语法），Python 会将此标识符绑定到一个元组，该元组包含了所有基于位置的隐式地接收到的参数。当你用这种语法传递参数时，标识符可以被绑定到任何可迭代对象（事实上，它也可以是任何表达式，并不必须是一个标识符，只要这个表达式的结果是一个可迭代对象即可）。

**kwds（标识符可以是任意的，通常用 k 或者 kwds 表示）是 Python 用于接收或者传递任意命名的参数的语法。（Python 有时候会将命名参数称为关键字参数，它们其实并不是关键字——只是用它们来给关键字命名，比如 pass、for 或 yield，还有很多。不幸的是，这种让人疑惑的术语目前仍是这门语言及其文化的根深蒂固的一个组成部分。）当你接收到用这种语法描述的一个参数时（比如你在函数的 def 语句中对函数签名使用了双星号语法），Python 会将标识符绑定到一个字典，该字典包含了所有接收到的隐式的命名参数。当你用这种语法传递参数时，标识符只能被绑定到字典（其实它也可以是表达式，不一定是一个标识符，只要这个表达式的结果是一个字典即可）。

当你在定义或调用一个函数的时候，必须确保*a 和**k 在其他所有参数之后。如果这两者同时出现，要将**k 放在*a 之后。

如果要转换非常巨大的数字阵列，可以考虑 Numeric Python 和其他的第三方包。Numeric Python 支持一系列变换以及轴旋转，这些数学转换能把大多数人绕晕。

更多资料

Reference Manual 和 *Python in a Nutshell* 中 list displays（列表推导的另一种叫法）以及对于基于位置的参数*a 和命名参数**k 的传递；内建函数 zip 和 map；Numeric Python（http://www.pfdubois.com/numpy/）。

4.9 从字典中取值

感谢：Andy McKay

任务

你想从字典中取值，但是又不想由于你搜寻的键不存在而处理异常。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

解决方案

字典的 get 方法正是为取值而准备的。假设你有一个字典 d = {‘key’:’value’, }。为了得到 key 在 d 中对应的值，且不希望担心异常的问题，可以这样编写代码：

```
print d.get('key', 'not found')
```

如果想在取值之后将该条目删去，用 d.pop（执行 get 和 remove 操作）替换 d.get（只读取 d，从不修改 d 的值）即可。

讨论

为了在键不存在的时候取值且并不引发异常，用字典的简单的 get 方法即可。

如果试图通过索引的方式取值，比如 d[x]，而且 x 并不是字典 d 的键，你的举动会引发 KeyError 异常。这通常也没什么问题。如果期望获取字典中 x 对应的值，异常是最好的提醒你所犯的错误的方式（比如，可能需要调试你的程序）。

然而，有时候只是想尝试一下，因为你已经知道，x 可能不是 d 的键。这种情况下，不用引入 in 测试，如下：

```
if 'key' in d:
     print d['key']
else:
*args 和**kwds 语法 print 'not found'
```

或者使用 try/except 语句，如下：

```
try:
     print d['key']
except KeyError:
     print 'not found'
```

而应该使用 get 方法，就像“解决方案”所示的那样。如果调用 d.get(x)，不会有任何异常抛出：如果 x 是字典 d 中的键，你会得到 d[x]，如果不是，你只能得到 None（可以检查或者继续传递）。当 x 不是 d 的键的时候，如果 None 不是你期望的值，还可以调用 d.get(x, somethingelse)。这样，如果 x 不是 d 的键，得到的值是 somethingelse。

get 是一种简单而有用的机制，Python 的文档对此有很好的解释，奇怪的是有相当多的人并不清楚这一点。另一个类似的方法是 pop，与 get 很类似，只不过当键在字典中时，pop 会同时删除该条目。还有一条附加说明：get 和 pop 并不完全对应。如果 x 不是 d 的键，d.pop(x)会抛出 KeyError 异常；如果要想获得和 d.get(x)同样的效果，同时还具有删除条目的能力，调用 d.pop(x, None)即可。

更多资料

4.10 节；*Library Reference* 和 *Python in a Nutshell* 中关于映射类型的文档。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

4.10　给字典增加一个条目

感谢：Alex Martelli、Martin Miller、Matthew Shomphe

任务

给定一个字典 d，当 k 是字典的键时，你想直接使用 d[k]，若 k 不是 d 的键，则这个操作会给字典增加一个新条目 d[k]。

解决方案

字典的 setdefault 方法正是为此而设计的。假设我们正在创建一个由单词到页数的映射，字典将把每个单词映射到这个词出现过的页的页码构成的列表。这个应用中关键的代码段可能是这样的：

```
def addword(theIndex, word, pagenumber):
    theIndex.setdefault(word, [  ]).append(pagenumber)
```

这段代码等价于下面的更加详尽的版本：

```
def addword(theIndex, word, pagenumber):
    if word in theIndex:
        theIndex[word].append(pagenumber)
    else:
        theIndex[word] = [pagenumber]
```

以及：

```
def addword(theIndex, word, pagenumber):
    try:
        theIndex[word].append(pagenumber)
    except KeyError:
       theIndex[word] = [pagenumber]
```

使用 setdefault 方法能在相当大的程度上简化实现。

讨论

对于一个字典 d，d.setdefault(k, v)非常接近于 d.get(k, v)，后者在前面的 4.9 节曾介绍过。最本质的区别是，如果 k 不是字典的键，setdefault 方法会将 d[k]赋值为 v，即 d[k]=v（get 则仅仅返回 v，对 d 不会有任何影响）。因此，当需要类似于 get 的功能，但又需要同时提供这种特殊的效果时，请考虑 setdefault 方法。

如果字典中的值是列表，setdefault 方法尤其有用，4.15 节会提供更多的细节。最经典的 setdefault 应用大概会是这样：

```
somedict.setdefault(somekey, [  ]).append(somevalue)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

对于不可改变的值，setdefault 方法就没那么有用了。如果想对单词计数，最好的方法是使用 get，而不是 setdefault：

```
theIndex[word] = theIndex.get(word, 0) + 1
```

这是因为反正你也必须要将字典的 theIndex[word]的条目重新绑定（因为数字是不可改变的）。不过，针对我们这个单词到页码的例子，你一定不想由于下面这样的写法而造成性能上的损失：

```
def addword(theIndex, word, pagenumber):
    theIndex[word] = theIndex.get(word, [ ]) + [pagenumber]
```

让我们看看最后这个版本的 addword，每次调用它的时候都将创建 3 个新的列表：一个被作为第二个参数传递给 theIndex.get 的空列表，一个只含有一个元素 pagenumber 的列表，以及将前面两者拼接而得到的有 *N*+1 个元素的列表（*N* 是 word 之前出现的次数）。创建这么多列表很显然会降低整体性能。举个例子，在我的计算机上，我对出现在 1 000 页中的 4 个单词所进行的索引操作计时。用 addword 的第一个版本来作为参考，第二个版本（使用 try/except）快了约 10%，第三个版本（使用 setdefault）则慢了约 20%——这种性能差异基本上无关紧要。但第四个版本（使用 get）慢了 4 倍，这样的性能损失恐怕就不能视若无睹了。

更多资料

4.9 节；4.15 节；*Library Reference* 和 *Python in a Nutshell* 中字典的相关文档。

4.11 在无须过多援引的情况下创建字典

感谢：Brent Burley、Peter Cogolo

任务

当你逐渐习惯了 Python，会发现自己已经创建了大量的字典。当键是标识符时，可以用 dict 加上命名的参数来避免援引它们：

```
data = dict(red=1, green=2, blue=3)
```

这看上去比直接用字典形式的语法要整洁一些：

```
data = {'red': 1, 'green': 2, 'blue': 3}
```

讨论

一种创建字典的好方法是调用内建的 dict 类型，但有时用带有花括号和冒号的字典形式也不错。本节的代码说明，如果键都是文字的字符串并且在语法上对用户而言也是有效且适合作为标识符的，则通过调用 dict，无须援引字典的键。不能对像“12ba”和

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

“for”一样的字符串应用此法，因为“12ba”以数字开头，而 for 正好是 Python 关键字，不能作为标识符。

字典形式的语法是 Python 中唯一需要用到花括号的地方：如果不喜欢花括号，或者正好所用的键盘不太容易打出花括号（所有的意大利布局的键盘都这样），那么可以用 dict( )而不是{}来创建一个空字典。

调用 dict 还能给你带来一些其他额外的好处。dict(d)返回一个完全独立于原字典 d 的拷贝，就像 d.copy()一样。但当 d 不是一个字典，而是一个数对(key, value)的序列时，dict(d)仍然有效（如果 key 在序列中出现多次，那么只有最后一次出现的 key 会被计入）。一般创建字典的操作大概是这样：

```
d = dict(zip(the_keys, the_values))
```

the_key 是键的序列，the_values 则是键的对应值的序列。内建的 zip 函数创建并返回一个数对(key, value)构成的列表，内建类型 dict 接受这个列表作为参数并创建相应的字典。如果序列非常长，那么用 Python 标准库中的 itertoos 模块能有效地提高速度：

```
import itertools
d = dict(itertools.izip(the_keys, the_values))
```

内建函数 zip 会在内存中创建出包含所有数对的列表，而 itertools.izip 一次只生成一对数据。在我的计算机上，对于长度为 10 000 的序列，后面这种方式可以快两倍左右——Python 2.3 中是 18ms 对 45ms，Python 2.4 中则是 17ms 对 32ms。

还可以在 dict 调用中使用基于位置的参数和命名参数（如果命名的参数正好和基于位置的参数冲突，则前者生效）。举个例子，下面是一个字典的创建，其中用到了前面提到的 Python 关键字和另一个不宜做命名参数的键名：

```
d = dict({'12ba':49, 'for': 23}, rof=41, fro=97, orf=42)
```

如果想创建一个字典，其中每个键都对应相同的值，只需调用 dict.fromkeys (key_sequence, value)（如果你忽略了 value，它默认使用 None）即可。下面给个例子，用很清爽的方式初始化一个字典，并用它来统计不同的小写 ASCII 字母的出现次数：

```
import string
count_by_letter = dict.fromkeys(string.ascii_lowercase, 0)
```

更多资料

Library Reference 和 *Python in a Nutshell* 中内建的 dict 和 zip，以及模块 itertools 和 string 的文档。

4.12　将列表元素交替地作为键和值来创建字典

感谢：Richard Philips、Raymond Hettinger

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

任务

给定一个列表，需要交替地使用列表中的元素作为键和对应值来创建一个字典。

解决方案

内建的 dict 提供了很多创建字典的方法，但是并没有提供这种方式，所以我们得自己写一个函数来达到目的。一个方法是，对扩展的列表切片调用内建的 zip 函数：

```
def dictFromList(keysAndValues):
    return dict(zip(keysAndValues[::2], keysAndValues[1::2]))
```

一个更通用的适合任何序列或者可迭代参数的方式是，把从给定序列中获取多个数对（pair）的过程独立出来，变成一个单独的生成器。这方法不如 dictFromList 简洁，但是速度却更快，通用性也更好：

```
def pairwise(iterable):
    itnext = iter(iterable).next
    while True:
        yield itnext( ), itnext( )
def dictFromSequence(seq):
    return dict(pairwise(seq))
```

定义 pairwise 函数也使得我们可以用任意序列来更新任意一个已存在的字典，比如，mydict.update(pairwise(seq))。

讨论

本节介绍的两种“工厂函数”的实现方式本质上都是用同样的方法创建字典：都生成了一个(key, value)值对的序列，并将其作为参数传递给 dict。区别是它们具体怎么生成这个值对的序列。

dictFromList 使用内建函数 zip，并用 keysAndValue 列表的两个切片作为参数——两个切片分别根据奇偶索引搜集元素（一个的索引是 0、2、4…另一个则是 1、3、5…）。这个方法不错，但只在 keysAndValues 是支持扩展切片的类型或者类的实例时才有效，比如 list、tuple 或者 str。另外，它还会在内存中创建一些临时列表，如果 keysAndValues 是个很长的序列，这些列表的创建过程会降低一些性能。

而 dictFromSequence 则把创建值对序列的任务委托给了一个叫做 pairwise 的生成器。pairwise 的实现方式使得它能够使用任何可迭代对象——不仅仅是列表（或者元组、字符串等），甚至包括其他生成器的结果、文件、字典等。而且 pairwise 一次只生成一对，它从来不在内存中创建大列表，因此即使给定的序列很长，它的性能也不会有什么降低。

pairwise 的实现也很有趣。pairwise 的第一行语句将内建函数 iter 应用于传入的 iterable

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

参数，获得一个迭代器，然后再将一个本地变量 itnext 绑定到这个迭代器的 next 方法。这看上去有点奇怪，但这是 Python 中一种很好的通用的技巧：如果你有一个对象，你想对这个对象所做的事是在循环中不断调用它的一个方法，可以给它的这个被绑定的方法赋予一个本地的名字，然后就可以直接调用这个本地名字了，就像调用一个函数一样。对某些习惯其他语言的用户来说，像下面这样调用 next 方法可能会更舒适一些：

```
def pairwise_slow(iterable):
    it = iter(iterable)
    while True:
        yield it.next( ), it.next( )
```

不过，这个 pairwise_slow 变体并不比解决方案中的 pairwise 简单（“对不懂 Python 的人来说显得更熟悉”并不表示“更简单”），而且它慢了约 60%。重视简洁和清晰的确很重要，而且也是 Python 的核心价值观。完全不考虑任何性能问题也是一种主张，但在实践中这种观点不可能在任何语言中得到推荐。所以，既然我们都知道编写正确、清晰、简单代码的重要性，那么为什么不学习和遵循那些更加符合我们需要的 Python 用法呢？

更多资料

见 19.7 节中的关于用滑动窗口来循环可迭代对象的通用方法。参考 Python Reference Manual 中更多关于扩展切片的资料。

4.13 获取字典的一个子集

感谢：David Benjamin

任务

你有一个巨大的字典，字典中的一些键属于一个特定的集合，而你想创建一个包含这个键集合及其对应值的新字典。

解决方案

如果你不想改动原字典：

```
def sub_dict(somedict, somekeys, default=None):
    return dict([ (k, somedict.get(k, default)) for k in somekeys ])
```

如果你从原字典中删除那些符合条件的条目：

```
def sub_dict_remove(somedict, somekeys, default=None):
    return dict([ (k, somedict.pop(k, default)) for k in somekeys ])
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

下面是两个函数的使用和效果：

```
>>> d = {'a': 5, 'b': 6, 'c': 7}
>>> print sub_dict(d, 'ab'), d
{'a': 5, 'b': 6} {'a': 5, 'b': 6, 'c': 7}
>>> print sub_dict_remove(d, 'ab'), d
{'a': 5, 'b': 6} {'c': 7}
```

讨论

在 Python 中，我在很多地方都用到了字典——数据库的行、主键和复合键，用于模板解析的变量名字空间等。我常常需要基于另外一个已有的大字典创建一个新字典，此字典的键是大字典的键的一个子集。在大多数情况下，原字典应该保持不变；但有时，我也需要在完成了抽取之后删除在原字典中的子集。本节的解决方案对两种可能性都给出了答案。区别仅仅在于，如果需要原字典保持原样不变，使用 get 方法，如果需要删除子集，则使用 pop 方法。

如果 somekeys 中的某元素 k 并不是 somedict 的键，解决方案提供的函数会将 k 作为结果的键，并对应一个默认值（可以作为一个可选的参数传递给这两个函数，默认情况下是 None）。所以，最终结果也不一定是 somedict 的子集。不过我却发现这种行为方式对我的应用非常有帮助。

当你认为 somekeys 中的所有的元素都应当是 somedict 的键时，也许会希望在键“缺失”的时候获得一个异常，它可以提示和警告你程序中的 bug。记住，Tim Peters 在 *The Zen of Python* 中说过“错误不应该被静静地略过，除非有意为之”（在 Python 的交互式解释器的提示符下敲入 import this 并回车，你将看到精炼的 Python 设计原则）。所以，如果从你的应用的角度看，键不匹配是一个错误，那么会希望马上得到一个异常来提醒你错误的发生。如果这的确是你所希望的，可以对解决方案中的函数略作修改：

```
def sub_dict_strict(somedict, somekeys):
     return dict([ (k, somedict[k]) for k in somekeys ])
def sub_dict_remove_strict(somedict, somekeys):
     return dict([ (k, somedict.pop(k)) for k in somekeys ])
```

这些更加严格的变体版本甚至比原版本更简单——这充分说明了 Python 本来就喜欢在意外发生时抛出异常。

或者，你希望在键不匹配时直接将其忽略。这也只需要一点点修改：

```
def sub_dict_select(somedict, somekeys):
     return dict([ (k, somedict[k]) for k in somekeys if k in somedict])
def sub_dict_remove_select(somedict, somekeys):
     return dict([ (k, somedict.pop(k)) for k in somekeys if k in somedict])
```

列表推导中的 if 子句做完了我们期望的事，即在应用 k 之前先做鉴别工作。

在 Python 2.4 中可以用生成器表达式来替代列表推导，用它作为本节中的函数的参数。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

我们只需略微修改 dict 的调用，将 dict([…])改成 dict(…)（移除临近圆括号的方括号），就能享受进一步的简化和速度的提升。不过这些修改不适用 Python 2.3，因为它只支持列表推导而不支持生成器表达式。

更多资料

Library Reference 和 *Python in a Nutshell* 文档中关于 dict 的部分。

4.14 反转字典

感谢：Joel Lawhead、Ian Bollinger、Raymond Hettinger

任务

给定一个字典，此字典将不同的键映射到不同的值。而你想创建一个反转的字典，将各个值反映射到键。

解决方案

可以创建一个函数，此函数传递一个列表推导作为 dict 的参数以创建需要的字典。

```
def invert_dict(d):
    return dict([ (v, k) for k, v in d.iteritems( ) ])
```

对于比较大的字典，用 Python 标准库 itertools 模块提供的 izip 会更快一些：

```
from itertools import izip
def invert_dict_fast(d):
    return dict(izip(d.itervalues( ), d.iterkeys( )))
```

讨论

如果字典 d 中的值不是独一无二的，那么 d 无法被真正地反转，也就是不存在这样的字典，对于任意给定的键 k，满足 id[d[k]]==k。不过，本节展示的函数在这种情况下仍然能够创建一个“伪反转”字典 pd，对于任何属于字典 d 地值 v，d[pd[v]]==v。如果给你原始的字典 d，以及用本节函数获得的字典 x，可以很容易地检查 x 是 d 的反转字典还是伪反转字典：当且仅当 len(x)==len(d)时，x 才是 d 的真正的反转字典。这是因为，如果两个不同的键对应相同的值，对于解决方案给出的两个函数来说，两个键中的一个一定会消失，因而生成的伪反转字典的长度也会比原字典的长度短。在任何情况下，只有当 d 中的值是可哈希（hashable，意味着可以用它们做字典的键）的，前面展示的函数才能正常工作，否则，函数会抛出一个 TypeError 异常。

当我们编写 Python 程序时，我们通常会“无视小的优化”，正如 Donald Knuth 在 30 年前所说的“比起速度，我们更珍视清晰和正确性。”不过，了解更多让程序变快的知识

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

也没有害处：当我们为了简单和清晰而采用某种方法编写程序时，我们最好深入地考虑一下我们的决定，不要懵懵懂懂。

在这里，解决方案中的 invert_dict 函数可能会被认为更清晰，因为它清楚地表达了它在做的事。该函数取得了由 iteritems 方法生成的成对的键及其对应值 k 和 v，将它们包裹成(value, key)的顺序，并把最后生成的序列作为参数赋给 dict，这样 dict 就构建出了一个值成为键，而原先的键变成了对应值的新字典——正是我们需要的反转字典。

而解决方案中 invert_dict_fast 函数其实也没有那么复杂，它的操作更加抽象，它首先将所有的键和值分别转为两个独立的迭代器，再通过调用 Python 标准库 itertools 模块提供的 izip 将两个迭代器转化为一个迭代器，其中每个元素都是像(value, key)一样的一对值。如果你能够习惯于这种抽象层次，你将体会到更高层次的简洁和清晰。

由于这种高度的抽象性，以及不具化（materialize）整个列表（而是通过生成器和迭代器一次生成一项）的特性，invert_dict_fast 能够比 invert_dict 快很多。比如，在我的计算机上，反转 10 000 个条目的字典，invert_dict 耗时 63ms，而 invert_dict_fast 则仅用时 20ms。速度提升了 3 倍，颇为可观。当你处理大规模数据时，由于代码的高度抽象性而带来的性能提升将会变得更加明显。特别是当你使用 itertools 来替换循环和列表推导时，执行速度同样也能获得极大提升，因为你无须在内存中具化一些超大的列表。当你习惯了更高的抽象层次，性能的提升只是一个额外收益，除此之外，你在观念和创造性上也会有所进步。

更多资料

Library Reference 和 *Python in a Nutshell* 中的映射类型和 itertools 的文档；第 19 章。

4.15　字典的一键多值

感谢：Credit: Michael Chermside

任务

需要一个字典，能够将每个键映射到多个值上。

解决方案

正常情况下，字典是一对一映射的，但要实现一对多映射也不难，换句话说，即一个键对应多个值。你有两个可选方案，但具体要看你怎么看待键的多个对应值的重复。下面这种方法，使用 list 作为 dict 的值，允许重复：

```
d1 = { }
d1.setdefault(key, [ ]).append(value)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

另一种方案，使用子字典作为 dict 的值，自然而然地消灭了值重复的可能：

```
d2 = { }
d2.setdefault(key, { })[value] = 1
```

在 Python 2.4 中，这种无重复值的方法可等价地被修改为：

```
d3 = { }
d3.setdefault(key, set( )).add(value)
```

讨论

正常的字典简单地将一个键映射到一个值上。本节则展示了三个简单有效的方法来实现一个键对应多个值的功能，即将字典的值设为列表或字典，若在 Python 2.4 中，还有可能是集合。基于列表的方法的语义和其他两者差别不大，最重大的差别是它们对待值重复的态度。每种方式都依赖字典的 setdefault 方法（4.10 节有相关内容）来初始化字典的一个键所对应的条目，并在需要的时候返回上述条目。

除了给键增加对应值之外，还要做更多的事情。对于使用列表并允许重复的第一个方式，下面代码可取得键对应的值列表：

```
list_of_values = d1[key]
```

如果不介意当键的所有值都被移除后，仍留下一个空列表作为 d1 的值，可以用下面方法删除键的对应值：

```
d1[key].remove(value)
```

虽然有空列表，但要想检查一个键是否至少有一个值还是很容易的，使用一个总是返回列表（也可能是空列表）的函数就行了：

```
def get_values_if_any(d, key):
    return d.get(key, [ ])
```

比如，为了检查“freep”是否是字典 d1 的键“somekey”的对应值之一，可以这样编写代码：if 'freep' in get_values_if_any(d1, 'somekey')。

使用子字典且没有值重复的第二种方式的用法非常类似。为了获得键的对应值列表，具体做法是：

```
list_of_values = list(d2[key])
```

为了移除某键的一个值，可以像下面这样做，当然，当键的值都被删除之后，字典 d2 中仍会留下一个空字典：

```
del d2[key][value]
```

第三种方式，只适用于 Python 2.4 以上并使用了集合的方式，它的移除键值的操作如下：

```
d3[key].remove(value)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第二和第三种方式（无重复）的 get_value_if_any 函数：

```
def get_values_if_any(d, key):
     return list(d.get(key, ( )))
```

本节讨论了如何实现一个很基本的功能，但并没有提到如何以一种系统化的方式来应用它，你也许会考虑将这些代码封装成一个类。要达到这个目的，必须得通盘考虑你的设计。你能否接受某个值和某个键的对应关系出现多次？（用数学的语言可表述为，对于每个键而言，条目究竟是包还是集合？）如果是的话，则 remove 方法究竟是将总的对应次数减一，还是完全地删掉那些对应关系？这只是你面临各种各样决定的一个开始，不过，要想做出正确选择，必须基于应用的实际需求来考虑。

更多资料

4.10 节；*Library Reference* 和 *Python in a Nutshell* 关于映射类型的章节；18.8 节中对于 bag 类型的实现。

4.16 用字典分派方法和函数

感谢：Dick Wall

任务

需要根据某个控制变量的值执行不同的代码片段——在其他的语言中你可能会使用 case 语句。

解决方案

归功于面向对象编程的优雅的分派概念，case 语句的使用大多（但不是所有）都可以被替换成其他分派形式。在 Python 中，字典及函数是一等（first-class）对象这个事实（比如函数可以作为字典中的值被存储），使得“case 语句”的问题更容易被解决。比如，考虑下面的代码片段：

```
animals = [  ]
number_of_felines = 0
def deal_with_a_cat( ):
     global number_of_felines
     print "meow"
     animals.append('feline')
     number_of_felines += 1
def deal_with_a_dog( ):
     print "bark"
     animals.append('canine')
def deal_with_a_bear( ):
     print "watch out for the *HUG*!"
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

```
        animals.append('ursine')
    tokenDict = {
        "cat": deal_with_a_cat,
        "dog": deal_with_a_dog,
        "bear": deal_with_a_bear,
        }
    # 模拟，比如，从文件中读取的一些单词
    words = ["cat", "bear", "cat", "dog"]
    for word in words:
        # 查找每个单词对应的函数调用并调用之
        return tokenDict[word]( )
    nf = number_of_felines
    print 'we met %d feline%s' % (nf, 's'[nf==1:])
    print 'the animals we met were:', ' '.join(animals)
```

讨论

本节的要点是，构建一个字典，以字符串（或其他对象）为键，以被绑定的方法、函数或其他的可调用体作为值。在每一步的执行过程中，我们都先用字符串键来选择需要执行的可调用体。这个方法可以被当做一个通用的 case 语句用于各处。

这确实非常简单，我也经常使用这种技术。还可以使用被绑定的方法或者其他可调用体来替换本节示例中查找的函数。但当你使用未绑定的方法时，需要传递一个正确的对象作为第一个参数来调用它们。还可以将可调用体和它所需要的参数放在一个元组中，然后把元组当做字典的值存储起来，这样具有更强的通用性。

在别的语言中，我可能需要 case、switch 或者 select 语句，但在 Python 中，所有类似的地方我都用这个技术来实现同样的功能。

更多资料

Library Reference 中关于映射类型的章节；Reference Manual 中关于绑定和非绑定方法的介绍；*Python in a Nutshell* 中关于字典和可调用体的介绍。

4.17 字典的并集与交集

感谢：Tom Good、Andy McKay、Sami Hangaslammi、Robin Siebler

任务

给定两个字典，需要找到两个字典都包含的键（交集），或者同时属于两个字典的键（并集）。

解决方案

有时，尤其是在 Python 2.3 中，你会发现对字典的使用完全是对集合的一种具体化的

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

体现。在这个要求中，只需要考虑键，不用考虑键的对应值，一般可以通过调用 dict.fromkeys 来创建字典，像这样：

```
a = dict.fromkeys(xrange(1000))
b = dict.fromkeys(xrange(500, 1500))
```

最快计算出并集的方法是：

```
union = dict(a, **b)
```

而最快且最简洁地获得交集的方法是：

```
inter = dict.fromkeys([x for x in a if x in b])
```

如果字典 a 和 b 的条目的数目差异很大，那么在 for 子句中用较短的那个字典，在 if 子句中用较长的字典会有利于提升运算速度。在这种考虑之下，牺牲简洁性以获取性能似乎是值得的，交集计算可以被改为：

```
if len(a) < len(b):
    inter = dict.fromkeys([x for x in a if x not in b])
else:
    inter = dict.fromkeys([x for x in b if x not in a])
```

Python 也提供了直接代表集合的类型（标准库中的 sets 模块，在 Python 2.4 中已经成为了内建的部分）。可以把下面的代码片段用在模块的开头，这个代码片段确保了名字 set 被绑定到了适合的类型，这样在整个模块中，无论你用 Python 2.3 还是 2.4，都可以使用同样的代码：

```
try:
    set
except NameError:
    from sets import Set as set
```

这样做的好处是，可以到处使用 set 类型，同时还获得了清晰和简洁，以及速度的提升（在 Python 2.4 中）：

```
a = set(xrange(1000))
b = set(xrange(500, 1500))
union = a | b
inter = a & b
```

讨论

虽然 Python 2.3 的标准库模块 sets 已经提供了一个优雅的数据类型 set 来代表集合（带有可哈希（hashable）的元素），但由于历史原因，使用 dict 来代表集合仍然是很普遍的。基于这个目的，本节展示了如何用最快的方法来计算这种集合的交集和并集。本节的代码在我的计算机上，并集计算耗时 260µs，交集计算则耗时 690µs（Python 2.3；在 Python 2.4 中，这两个数字分别是 260µs 和 600µs），而其他的基于循环或者生成器表达式的方法会更慢。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

不过，最好还是用 set 类型而不是字典来代表集合。如同本节所示，使用 set 能让代码更加直接和易读。如果你不喜欢或操作符（|）和与操作符（&），可以使用等价的 a.union(b)和 a.intersection(b)。这样操作除了清晰，速度也有提升，特别是在 Pyton 2.4 中，计算并集需要 260μs，但计算交集只需要 210μs。即使是在 Python 2.3，其速度也是可以接受的：并集计算耗时 270μs，交集计算耗时 650μs，没有在 Python 2.4 快，但如果你仍然用字典来代表集合的话，速度其实是相当的。最后一点，一旦你引入 set 类型（无论是 Python 2.4 内建的，还是通过 Python 标准库 sets 模块引入的，接口是一样的），你将获得丰富的集合操作。举个例子，属于 a 或者 b 但却不属于 a 和 b 的交集的集合是 a^b，可以等价地被表示为 a.symmetric_difference(b)。

即使由于某些原因使用了 dict，也应当尽可能地用 set 来完成集合操作。举个例子，假设你有个字典 phones，将人名映射到电话号码，还有个字典 address，将人名映射到地址。最清楚简单地打印所有同时知道地址和电话号码的人名及其相关数据的方法是：

```
for name in set(phones) & set(addresses):
    print name, phones[name], addresses[name]
```

跟下面的方法比，这非常简洁，虽然清晰度可能还有争议：

```
for name in phones:
    if name in addresses:
        print name, phones[name], addresses[name]
```

另一个很好的可选方法是：

```
for name in set(phones).intersection(addresses):
    print name, phones[name], addresses[name]
```

如果使用 intersection 方法，而不是&交集操作，就不需要将两个字典都转化成 set，只需要其中一个。然后再对转化后的 set 调用 intersection，并传入另一个 dict 作为 intersection 方法的参数。

更多资料

Library Reference 和 *Python in a Nutshell* 的映射类型、sets 模块及 Python 2.4 中的内建 set 类型。

4.18　搜集命名的子项

感谢：Alex Martelli、Doug Hudgeon

任务

你想搜集一系列的子项，并命名这些子项，而且你认为用字典来实现有点不便。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

解决方案

任意一个类的实例都继承了一个被封装到内部的字典，它用这个字典来记录自己的状态。我们可以很容易地利用这个被封装的字典达到目的，只需要写一个内容几乎为空的类：

```
class Bunch(object):
    def __init__(self, **kwds):
        self.__dict__.update(kwds)
```

现在，为了将变量组织起来，创建一个 Bunch 实例：

```
point = Bunch(datum=y, squared=y*y, coord=x)
```

现在就可以访问并重新绑定那些刚被创建的命名属性了，也可以进行添加、移除某些属性之类操作。比如：

```
if point.squared > threshold:
    point.isok = True
```

讨论

我们常常需要搜集一些元素，然后给它们命名。这个需求用字典来实现完全没有问题，但是利用一个几乎什么都不做的小类明显更加方便美观。

如同解决方案的代码所示，创建一个小小的类，提供参数访问的语法，我们几乎不用写什么东西。字典也适合用来搜集一些子项，每个子项都有自己的名字（根据环境，字典中的子项的键可以被认为是该子项的名字），但如果所有的名字都是标识符，而且被当做变量使用，字典并不是最好的方案。在类 Bunch 的__init__方法中，通过**kwds 语法，可以接受任意的命名参数，并且用 kwds 参数来更新实例的空字典，这样，每个命名的参数都成为实例的一个属性。

与访问属性的语法相比，字典索引语法不是那么简洁和易读。比如，如果 point 是个字典，解决方案中的最后的那个代码片段就应该是这样：

```
if point['squared'] > threshold:
    point['isok'] = True
```

此外，还有另一个备选的实现方案，看上去也很吸引人：

```
class EvenSimplerBunch(object):
    def __init__(self, **kwds):
        self.__dict__ = kwds
```

将实例的字典重新绑定可能会让人感到不安，但比起调用字典的 update 方法，它不会造成什么不好的后果。所以，你可能会喜欢这个备选的 Bunch 实现在速度上的优势。不过，我从来没有在任何 Python 文档中看到对下面用法的担保：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

```
d = {'foo': 'bar'}
x = EvenSimplerBunch(**d)
```

最好使 x._ _dict_ _成为字典 d 的一个独立的拷贝，而不是共享一个引用。现在这个方法的确有效，在各个版本中都能工作，但除非语法文档规定了其语义，否则我们不能确信这种做法会永远有效。所以，如果选择了 EvenSimplerBunch 的实现，你可能会选择赋值一个拷贝（dict(kwds)或者 kwds.copy()），而不是 kwds 本身。而且，如果这样做，那一点速度优势也就消失了。总之，最好还是将原先的 Bunch 的实现方法作为首选。

另一个富诱惑的做法是直接让 Bunch 类继承 dict，并将属性访问的特殊方法设为该子项本身的属性方法，像下面这样：

```
class DictBunch(dict):
    __getattr__ = dict.__getitem__
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__
```

这个方法的一个问题是，根据定义，DictBunch 的一个实例 x 会拥有很多它实际上没有的属性，因为它获得了 dict 所有的属性（实际上是方法，但在这个环境中其实没什么区别）。所以，你通过 hasattr(x, someattr)来检查属性没有意义，但可以对先前实现的 Bunch 和 EvenSimplerBunch 这么做，而且，你还得事先排除 someattr 的值是一些通用语如“keys”、“pop”和“get”等的可能性。

Python 的关于属性和子项的区别是这门语言清晰和简洁的源泉。不幸的是，很多 Python 新手错误地以为将属性和子项混为一谈并没有什么问题，这大概是因为先前的 JavaScript 和其他语言的经验，在 JavaScript 中，属性和条目通常是可以混为一谈的。不过新手应该把概念厘清，不要继续稀里糊涂。

更多资料

Python Tutorial 关于类的章节；*Language Reference* 和 *Python in a Nutshell* 中对类的介绍；第 6 章关于 Python 面向对象编程的介绍；4.18 节对于**kwds 语法的介绍。

4.19 用一条语句完成赋值和测试

感谢：Alex Martelli、Martin Miller

任务

你正在将 C 或者 Perl 代码转换成 Python 代码，并试图尽量保留原有的结构，你现在需要一种表达方式，能够同时完成赋值和测试（如同其他语言中的 if((x=foo( ))或 while((x=foo( )))。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

解决方案

在 Python 中，不能这么写代码：if x=foo(): . . . 赋值是一个语句，不是一个表达式，而你只能在 if 和 while 中使用表达式作为条件。不过问题不大，只需要将代码修改得更 Python 化一点。举个例子，要对一个文件对象 f 逐行处理，C 风格的写法（在 Python 中这样的句法是错误的）应该是这样：

```
while (line=f.readline( )) != '':
     process(line)
```

而 Python 风格的写法（更易读、清爽及快速）：

```
for line in f:
     process(line)
```

有时，需要将 C、Perl 或其他语言编写的程序转换成 Python 代码，而且希望尽可能保留原有的结构。写一个简单的工具类会起到很大的作用：

```
class DataHolder(object):
     def __init__(self, value=None):
          self.value = value
     def set(self, value):
          self.value = value
          return value
     def get(self):
          return self.value
# 可选的，强烈不建议使用，但有时确实很方便:
import __builtin__
__builtin__.DataHolder = DataHolder
__builtin__.data = data = DataHolder( )
```

在 DataHolder 类和它的实例 data 的帮助下，原有的 C 风格结构得以保留：

```
while data.set(file.readline( )) != '':
      process(data.get( ))
```

讨论

在 Python 中赋值是语句，不是表达式。因此，在 if、elif 或 while 语句中，你无法将正在测试的东西赋值给其他名字或变量。这也没什么，调整你的程序结构，避免在测试的时候赋值即可（这样代码会显得更清晰）。具体地说，在任何时候如果你感到需要在一个 while 循环中同时完成赋值和测试，都应该认识到，你的循环结构可能需要被重构成一个生成器（或者其他的迭代器）。一旦以这种思路进行重构，你的循环就变成了简单而直接的 for 语句。解决方案给的例子，循环读取文本文件中的行，正是通过 Python 本身完成了重构，因为 file 对象就是一个迭代器，其中的元素是文件的行。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

不过，有时必须转换由 C、Perl 或其他语言编写的程序到 Python 代码，这些语言本身是支持赋值作为表达式的。当实现某个已经提供了参考实现的算法或者书上的算法等，并用 Python 编写其最初版本的时候，经常会遇到这种用赋值语句作表达式的情况。在这个条件下，让最初的 Python 实现代码贴近原有的结构是明智的。可以以后再对你的代码重构，使之更像 Python——清晰、快速等。不过首先，需要尽可能快地完成一个能工作的版本，而且需要你的代码接近原型以便进行错误和兼容性的检查。幸运的是，Python 具有足够的能力来满足你的需求。

Python 不让我们重新定义赋值的含义，不过我们可以写一个方法（或函数），把参数存在“某处”，同时也能返回那个参数用于测试。在这里，说到“某处”，我们会很自然地想到用一个对象的属性来代表这个“某处”，因此对象的方法是比函数更自然的选择。当然也可以直接去访问属性（这样，get 方法就变得多余了），不过，我感觉提供 data.set 和 data.get 会更匀称整齐一点。

data.set(whatever)比起 data.value=whatever 多了一点句法上的好处，即，这个加入的值可以是一个表达式。因此，它是一个很棒的方法，能够帮助我们完成忠实于原型的代码翻译。这样的 Python 代码和原型代码如 C 或 Perl 代码的唯一区别仅仅是句法上的微小差异——但总体结构一致，这是我们最关心的议题。

引入_ _builtin_ _并给它的属性赋值是一个小花招，其本质上是在运行时定义了一个新的内建对象。可以在你的应用程序开头玩这个花招，但马上所有的其他模块都自动地具有了访问这个新内建对象的能力，而且还无须引入模块。这不是好的 Python 解决问题的方式，相反，这是在挑战 Python 的良好品味的尺度，因为其他模块完全不应该为你的应用程序所造成的副作用负责。不过，既然本节的目的是提供一个快速且肮脏的花招来解决代码的初次翻译问题，之后将要进行的重构会改良整个代码，那么在这样特殊情况下，只要我们的花招不被用于正常的产品代码，这种手段还可以让人忍受。

另一方面，还有一个花招你是绝对不应该用的，即利用列表推导的一个漏洞：

```
while [line for line in [f.readline( )] if line!='']:
    process(line)
```

这个花招目前还能正常工作，这是因为 Python 2.3 和 2.4 都将列表推导的控制变量（这里是 line）“泄露”到周边的空间中。这是一个很容易混淆和不易读的手段，而且它也被废弃了，列表推导控制变量的泄露问题将在未来的 Python 版本中被修正，那时这个招数就彻底失灵了。

更多资料

Tutorial 的关于类的文档；*Library Reference* 和 *Python in a Nutshell* 中关于_ _builtin_ _模块的文档；*Language Reference* 和 *Python in a Nutshell* 中列表推导的文档。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

4.20 在 Python 中使用 printf

感谢：Tobias Klausmann、Andrea Cavalcanti

任务

你喜欢用 C 提供的 printf 函数将某些东西打印到程序的标准输出，但 Python 并不提供这样的函数。

解决方案

在 Python 中实现 printf 是很简单的事：

```
import sys
def printf(format, *args):
    sys.stdout.write(format % args)
```

讨论

Python 分开了输出（print 语句）和格式化（%操作符），如果你希望将这两个融合在一起，如同解决方案所示，它的实现是很简单的。无须担心空格和新行的自动插入，而只需要担心格式和参数的正确匹配问题。

举个例子，Python 的普遍做法是：

```
print 'Result tuple is: %r' % (result_tuple,),
```

它考虑得非常周到，但这两个逗号的意义不是很明确（一个在 result_tuple 之后，用于创建单元素元组，另一个则是避免了 print 插入默认的换行符），不过既然有了解决方案中提供的 printf 函数，就可以这样写：

```
printf('Result tuple is: %r', result_tuple)
```

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 sys 模块以及字符串格式化操作符%的文档；2.13 节提出的在 Python 中实现 C++风格的<<的方法。

4.21 以指定的概率获取元素

感谢：Kevin Parks、Peter Cogolo

任务

你想从一个列表中随机获取元素，就像 random.choice 所做的一样，但同时必须根据另

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

一个列表指定的各个不同元素的概率来获取元素，而不是用等同的概率撷取元素。

解决方案

Python 标准库中的 random 模块提供了生成和使用伪随机数的能力，但是它并没有提供这样特殊的功能，所以，我们必须得自己写一个函数：

```
import random
def random_pick(some_list, probabilities):
    x = random.uniform(0, 1)
    cumulative_probability = 0.0
    for item, item_probability in zip(some_list, probabilities):
        cumulative_probability += item_probability
        if x < cumulative_probability: break
    return item
```

讨论

Python 标准库中的 random 模块并没有提供根据权重做出选择的功能，这种功能在游戏、模拟和随机测试中是很常见的需求，所以，本节的目标是提供此功能的实现。解决方案使用了 random 模块的 uniform 函数获得了一个在 0.0 和 1.0 之间分布的伪随机数，之后同时循环元素及其概率，计算不断增加的累积概率，直到这个概率值大于伪随机数。

本节假设（但并未检查）概率序列 probabilities 具有和 some_list 一样的长度，其所有元素都在 0.0 和 1.0 之间，且相加之和为 1.0；如果违反了这个假设，仍能进行一些随机的撷取，但不能完全地遵循（不连贯）函数的参数所规定的行为。可能想在函数开头加上一些 assert 语句以确保参数的有效性：

```
assert len(some_list) == len(probabilities)
assert 0 <= min(probabilities) and max(probabilities) <= 1
assert abs(sum(probabilities)-1.0) < 1.0e-5
```

不过，这些检查会消耗一些时间，所以我通常都不这么做，在正式的解决方案中我也没有把它们纳入。

正如我前面提到的，这个任务要求每一项都有一个应对的概率，这些概率分布在 0 和 1 之间，且总和相加为 1。另一个有点类似的任务是根据一个非负整数的序列所定义的权重进行随机撷取——基于机会，而不是概率。对于这个问题，最好的解决方案是使用生成器，其内部结构和解决方案中的 random_pick 函数差异很大：

```
import random
def random_picks(sequence, relative_odds):
    table = [ z for x, y in zip(sequence, relative_odds) for z in [x]*y ]
    while True:
        yield random.choice(table)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

生成器首先准备一个 table，它的元素的数目是 sum(relative_odds)个，sequence 中的每个元素都可以在 table 中出现多次，出现的次数等于它在 relative_odds 序列中所对应的非负整数。一旦 table 被制作完毕，生成器的主体就可以变得又小又快，因为它只需要将随机撷取的工作委托给 random.choice。举个例子，关于这个 random_picks 的典型应用：

```
>>> x = random_picks('ciao', [1, 1, 3, 2])
>>> for two_chars in zip('boo', x): print ''.join(two_chars),
bc oa oa
>>> import itertools
>>> print ''.join(itertools.islice(x, 8))
icacaoco
```

更多资料

Library Reference 和 *Python in a Nutshell* 中的 random 模块。

4.22 在表达式中处理异常

感谢：Chris Perkins、Gregor Rayman、Scott David Daniels

任务

你想写一个表达式，所以你无法直接用 try/except 语句，但你仍需要处理表达式可能抛出的异常。

解决方案

为了抓住异常，try/except 是必不可少的，但 try/except 是一条语句，在表达式内部使用它的唯一方法是借助一个辅助函数：

```
def throws(t, f, *a, **k):
''' 如果 f(*a, **k)抛出一个异常且其类型是 t 的话则返回 True
 （或者，如果 t 是一个元组的话，类型是 t 中的某项）'''
    try:
        f(*a, **k)
    except t:
        return True
    else:
        return False
```

举个例子，假设你有一个文本文件，每行有一个数字，但文件也可能有多余的内容如空格行及注释行等。可以生成一个包含文件中的所有数字的列表，只需略去那些不是数字的行即可：

```
data = [float(line) for line in open(some_file)
                    if not throws(ValueError, float, line)]
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

讨论

你可能会喜欢将函数命名为 raises，但我个人更喜欢 throws，可能是出于对 C++的感情。不过不管什么名字，这个辅助函数都接受一个异常类型 t 作为第一个参数，接着是一个可调用体 f，然后是任意的基于位置的参数 a 和命名参数 k，两者都将被传递给 f。像 if not throws(ValueError, float(line))这样的写法是不行的。当你调用函数时，Python 在将控制权交给函数之前会对参数求值，如果参数的求值引发了异常，函数永远不会得到机会执行。这种情况，在很多人刚开始用 Python 标准库的 unittest.TestCase 类的 assertRaises 方法时屡有发生，我见过不止一次。

当 throws 函数执行时，它在 try/except 语句的 try 子句中调用 f，将那个任意的基于位置的参数和命名参数传递给 f。如果在 try 子句中对 f 的调用引发了异常，且异常的类型是 t（或者是列出的异常类型中的一种，如果 t 是一个异常类型的元组的话），则控制权交给了对应的 except 子句，在此例中，返回 true 作为 throws 的结果。如果 try 子句中没有异常发生，控制权会交给对应的 else 子句（如果有），它将返回 false 作为 throws 的结果。

注意，如果有什么非预期的异常（类型不在 t 的指定范围中），throws 函数并不会尝试截获异常，因而 throws 就将被终止，异常则交给它的调用者。这是一个有意为之的设计。在 except 子句中用太大的网去捕获异常并不是什么好主意，那常常意味着查错查得头昏脑胀。如果调用者真的想要 throws 截获所有的异常，它可以这样调用：throws(Exception, … 然后，就等着头疼吧。

throws 函数的问题是，实际上做了两次关键操作：一次是看它有没有抛出异常，先把结果抛诸脑后，另一次是获得结果。所以最好的结局是同时获得结果和被截获异常的提示。我刚开始是这么做的：

```
def throws(t, f, *a, **k):
    " 如果 f(*a, **k)抛出异常且异常类型为 t，返回(True, None)
    或者(False, x)，其中 x 是 f(*a, **k)的结果 "
    try:
        return False, f(*a, **k)
    except t:
        return True, None
```

不幸的是，这个版本不符合列表推导的要求，没有什么优雅的办法能够同时得到标志和结果。因此，我选择了一个不同的方法：一个在任何情况下都返回 list 的函数——如果有异常被捕获就返回空列表，否则就返回仅包含结果的列表。这个方法工作得很好，但是为了清晰起见，最好把函数名改一改：

```
def returns(t, f, *a, **k):
    " 正常情况下返回[f(*a, **k)]，若有异常返回[ ] "
    try:
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

```
        return [ f(*a, **k) ]
    except t:
        return [ ]
```

最后生成的列表推导变得更加优雅，比解决方案中的版本好多了，至少我这么认为。

```
data = [ x for line in open(some_file)
           for x in returns(ValueError, float, line) ]
```

更多资料

Python in a Nutshell 中关于截获和处理异常的文档；4.8 节对*args 和**kwds 语法的介绍。

4.23　确保名字已经在给定模块中被定义

感谢：Steven Cummings

任务

你想确保某个名字已经在一个给定的模块中定义过了（比如，你想确认已经存在一个内建的名字 set 了），如果该名字未被定义，你想执行一些代码来完成定义。

解决方案

这个任务的解决办法是我见到过的 exec 语句最好的用武之地。exec 使得我们可以执行一个字符串中的任意 Python 代码，这让我们可以写一个很简单的函数来达到目的：

```
import __builtin__
def ensureDefined(name, defining_code, target=__builtin__):
    if not hasattr(target, name):
        d = { }
        exec defining_code in d
        assert name in d, 'Code %r did not set name %r' % (
            defining_code, name)
        setattr(target, name, d[name])
```

讨论

如果你的代码要支持很多版本的 Python（或者支持第三方的包），那么你的很多模块可能得以这样的代码片段作为开头（这样可以确保 set 在 Python 2.4 或者 2.3 中都被正确地设置了，当然在 Python 2.4 中 set 其实是内建类型，而在 Python 2.3 中我们需要从标准库中导入 set）：

```
try:
    set
except NameError:
    from sets import Set as set
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

本节解决方案将逻辑直接封装起来，默认情况下即可工作于_ _builtin_ _模块，这是因为当你在老的 Python 版本中处理名字问题时，它是典型的需要用到的模块。用本节的解决方案，只需在程序初始化的时候运行以下代码一次，就可以确保名字 set 被正确地定义了：

```
ensureDefined('set', 'from sets import Set as set')
```

这个方法的最大优势是，你只需在初始化的时候，在程序中的某处调用 ensureDefined 即可，而无须在各个模块的开头写一堆 try/except 语句。另外，ensureDefined 使得代码可读性更好，因为它只做一件事，因此调用它的目的也一目了然，而 try/except 语句却应用面很广，需要花些时间才能看清楚和理解它们。最后一点，try/except 在类似于 pychecker 的检查工具中可能引发警告，而本节的做法却可以绕过这个问题（如果你没有用过 pychecker 或者其他类似的工具，应该赶紧去试试。http://pychecker.sourceforge.net/）

我们使用了一个辅助性的字典 d 作为 exec 语句的目标，而且也只转换被要求的名字，我们竭力避免对 target 造成的一些预期外的副作用。也可以使用一些非模块的对象（类，甚至类的实例）来作为 target，这样就无须给 target 加上一个叫做_ _builtins_ _的属性来引用 Python 内建类型的字典。如果想用得更随意一些，if 语句的主体部分还可以变成：

```
exec defining_code in vars(target)
```

你将不可避免地造成一些副影响，见 http://www.python.org/doc/current/ref/exec.html 的介绍。

很重要的一点是，一定要清楚 exec 能够执行给它的任何包含 Python 代码的有效的字符串。因此，必须确保在你调用函数 ensureDefined 时，传递给参数 defining_code 的值不是来自于一个不可信的来源，比如一个被恶意修改过的文本本件。

更多资料

Python Language Reference Manual 中关于 exec 的在线文档：http://www.python.org/doc/current/ref/exec.html