第 5 章

搜索和排序

引言

感谢：Tim Peters，PythonLabs

在 1960 年代，计算机制造商们曾经估计，如果将所有的用户计入，他们制造的计算机有 25%的运行时间被用于排序。实际上，有很多计算机花了超过一半的计算时间在排序上。通过这样的评估结果，我们可以得出结论，可能（i）确实有很多非常重要的和排序相关的应用，或者（ii）很多人在进行一些不必要的排序计算，再或者（iii）低效的排序算法被广泛应用造成了计算时间的浪费。

——*Donald Knuth*

The Art of Computer Programming，*vol.3*，Sorting and Searching，第 3 页

在 Knuth 教授的巨著中，有关搜索和排序主题的部分是长达近 800 页的复杂的技术文献。在 Python 实践中，我们把它归纳为最重要的两条（我们已经读过了 Knuth 的书，所以不用去读了）：

- 当需要排序的时候，尽量设法使用内建 Python 列表的 sort 方法；
- 当需要搜索的时候，尽量设法使用内建的字典。

本章的很多内容将会展示这两条原则。最常见的主题是使用 decorate-sort-undecorate（DSU）模式，这是一种通用的方法，通过创建一个辅助的列表，我们可以将问题转化为列表的排序，从而可以利用默认的快速的 sort 方法。这个技术是在本章中最有用的部分。事实上，DSU 是如此常用，以至于 Python 2.4 导入了新的特性来使之更易于使用。因此很多节的解决方案在 Python 2.4 中变得简单多了，但本章也讨论了老版本中已经更新过的解决方案。

DSU 依赖 Python 的内建比较（built-in comparison）的一个不常见的特性：序列是按照条目的顺序（lexicographically）进行比较的。条目顺序（lexicographical order）是对列

表和元组的字符串比较（即字母顺序）规则的归纳。假设 s1 和 s2 是序列，内建函数 cmp (s1, s2)等价于下面的 Python 代码：

```
def lexcmp(s1, s2):
        # 找到最靠左的不相等的一对
        i = 0
        while i < len(s1) and i < len(s2):
             outcome = cmp(s1[i], s2[i])
             if outcome:
                  return outcome
             i += 1
        # 全部相等，其中一个序列已经消耗完所有元素
        return cmp(len(s1), len(s2))
```

这段代码试图找到第一个不相等的对应元素。如果不相等的一对元素被找到，则通过这对元素计算其结果。或者，如果一个序列正好是另一个序列的前半截，那么较短的那个序列被认为是较小的序列。最终，如果上述情况都没有发生，则这两个序列完全一样，被认为相等。下面是一些例子：

```
>>> cmp((1, 2, 3), (1, 2, 3))      # 相等
0
>>> cmp((1, 2, 3), (1, 2))         # 第一个大，因为第二个是第一个的前一部分
1
>>> cmp((1, 100), (2, 1))          # 第一个小，因为 1<2
-1
>>> cmp((1, 2), (1, 3))            # 第一个小，因为 1==1，然后 2<3
-1
```

如果想根据主键及二级主键对一个对象列表进行排序，可以简单地创建一个元组的列表，其中每个元组都遵循相同的顺序来存储主键、二级主键以及对象本身，这样我们就可以基于条目顺序进行比较。由于元组是按照顺序比较的，所以比较操作可以顺理成章地得到正确的结果。在比较元组时，主键被首先比较，当且仅当主键相等时，二级主键才会继续进行比较。

本章的 DSU 模式的很多例子展示了这种思想在各种需求中的应用。DSU 技术可以用于任意数目的主键。只要愿意，可以给元组增加足够多的键，当然它们的顺序得按照你希望进行的比较顺序进行排列。在 Python 2.4 中，用 sort 的新的 key=可选参数也可以得到同样效果，有几节内容将会展示这一用法。相比于手工构建一个辅助的元组列表，使用 sort 方法的 key=参数更加容易、更节省内存，而且速度也更快。

Python 2.4 还为排序提供了其他改进，包括一个方便的快捷方法：内建函数 sorted 可以将任何可迭代对象排序，并且不改动原对象，而是首先将其复制到一个新的列表。Python 2.3（没有那个新的可选的关键字参数，该参数既可用于内建的 sorted 函数也可用于 list.sort），可以编写如下代码来实现相同的功能：

```
def sorted_2_3(iterable):
    alist = list(iterable)
    alist.sort( )
    return alist
```

由于列表复制和列表排序都不是很轻松的操作，而内建的 sorted 需要执行这些操作，所以使用内建的 sorted 函数不会获得速度上的优势，它的优势就在于方便。有个预先实现的函数在手边，总比每次都要写四行程序来完成同样功能让人心情愉快一些，这是个实用与否的问题。另一方面，一些小函数用得非常广泛和频繁，对原有的内建对象和函数进行扩展变得非常必要。Python 2.4 增加的 sorted 和 reversed 函数在之前已经被要求了很多年。

自从本书第一版出版之后，Python 的排序的最大的变化是 Python 2.3 采用了新的排序实现。由此产生的明显区别是，很多常用操作的速度变快了，而且新的排序是稳定排序（即如果原列表中的两个被比较的元素相等，那么在完成排序之后它们的相对顺序不变）。这个新的实现是如此成功，以至于进一步提高的空间似乎都不大了，Guido 也被说服并宣布 Python 的 list.sort 方法将永远是稳定排序。这个始于 Python 2.4 的保证其实已经在 Python 2.3 中实现了。当然，排序的发展历史不断地提醒我们，更好的方法也许还没被发现。因此，我们也会概要介绍一下 Python 的排序的发展历史。

Python 排序的简短历史

在早期的 Python 发行版中，list.sort 使用平台 C 库提供的 qsort 例程。这个排序方法最终被替换的原因有好几个，但主要是由于 qsort 的质量在不同的计算机上差异很大。在对一个带有很多相等的值或者完全反序的列表的排序中，一些版本慢得让人无法容忍。一些版本甚至会引发 Core Dump，因为它们是不可重入的。用户定义的_ _cmp_ _函数也可能会调用 list.sort，所以一个 list.sort 会调用其他的 list.sort，这是比较操作的副作用。一些平台的 qsort 例程无法处理这种状况。一个用户定义的_ _cmp_ _函数也可能（假如它是恶意的或者疯了）在正在排序的时候修改列表，很多平台的 qsort 例程在这种情况发生时都会引发 Core Dump。

Python 因而发展了自己的快速排序算法。每个发行版都会重写此算法，因为总是能在实际应用中发现慢得无法接受的情况。快速排序真的是一种脆弱而精巧的算法。

在 Python 1.5.2 中，快速排序算法被替换成了抽样排序和折半插入排序的混合体，这个算法超过四年保持不变，直到 Python 2.3 问世。抽样排序可以被看做是快速排序的变种，它使用很大的样本空间来挑选划分元素（partitioning element），也被称为轴心（它对元素的一个大的随机子集递归地抽样排序并挑选它们的中值）。这个变种使得二次方时间复杂度的行为变得几乎不可能，同时也让平均的比较次数非常接近理论上的最小值。

不过，抽样排序是一个复杂的算法，对于小列表而言，它有比较大的管理上的开销。因此，小列表（以及抽样排序划分出来的小片结果）由独立的折半插入排序算法处

理——其实就是一种普通的插入排序，只不过它用二分搜索来决定新元素的归属。很多关于排序的文章说这不值得烦恼，因为他们假设比较两个元素的开销要比在内存中交换元素的开销小，但对于 Python 的排序来说，这个假设不成立。移动一个对象非常容易，因为复制的只不过是一个对象的引用。比较两个对象的开销则比较昂贵，这是因为要通过面向对象机制寻找合适的代码来比较两个对象，同时这部分代码每次都被强制调用。正因为这一点，二分搜索在 Python 的排序中是很成功的应用。

基于这个混合方法，一些特殊情况被特别照顾以利于提速。首先，已经排序或者反序的列表会被检查出来，并以线性的时间复杂度来处理。对于一些应用而言，这种类型的列表很常见。其次，如果一个序列已经大部分完成排序，只有少数位于末尾的元素仍然处于乱序状态，排序工作将由折半插入排序算法接手。这比完全交由抽样排序算法处理要快一些，尤其是一些特殊的应用需要不断地将列表排序，加入一个新元素，然后再排序。然后，抽样排序算法中的一些特别的代码会检查相等且相邻一段元素，并将这些片段标记为已完成部分。

最后，所有这些努力产生了一个极其优异的排序算法，无论是在实际的使用中体现出来的高效，还是针对一些常见的特殊例子所展现的梦幻般的速度，都充分证明了此算法的成功。这个算法包括了大约 500 行非常复杂的 C 代码，和 5.11 节展示的例子可以说有天壤之别。

抽样排序算法已经行之有年，我也曾说过要请能够写出更快的 Python 排序的人吃饭。不过迄今为止，我还是只能一个人去吃饭。我也仍然在继续关注一些文献资料，因为混合抽样排序算法的一些问题仍然让我耿耿于怀。

- 虽然还没有在实际中发现二次方时间复杂度行为的例子，但我知道这样的例子一定是可以设计出来的，因为要设计一个比平均速度慢两到三倍的用例是非常容易的。
- 针对某些极端偏序（partial order）的特例的优化在实际应用中非常有用，但真实数据中经常出现其他类型的偏序，这些类型的偏序也应当被特别处理。事实上，我已经开始相信真正的随机输入顺序在现实生活中根本不可能存在（当然要把用于测试排序算法的时间复杂度的例子排除掉）。
- 如果不增加内存的使用量，现在还没有什么可行的方法来使抽样排序成为稳定排序。
- 为了对一些特例进行优化而使代码变得极其复杂、晦涩及丑陋。

当前的排序

很明显归并排序有几个优点，它的最坏结果可保证为 nlogn 时间复杂度，而且很容易实现稳定排序。问题是在 Python 中的多次对归并算法的实现产生的只是更慢的结果（比

起抽样排序，归并排序过多地移动了数据）和更高的内存消耗。

很多文献资料——而且是越来越多的，开始关注适应性排序算法（adaptive sorting algorithm），这种算法试图探测不同的输入中的元素顺序。我写了很多这种算法的实现，但它们都比 Python 的抽样排序慢得多，除非例子是被专门设计的。这些算法的理论基础比较复杂，因此难于产生有效的可行算法。后来我读到了一篇文章，该文指出列表的合并自然地揭示了很多类型的偏序，只需简单地注意一下每个输入列表连续“赢”的频率。这个信息很简单但也很具有普遍性。当我意识到可以将它应用在一个自然的归并排序中，而且这种方法能够很好地应对我所知道和关心的各种特例时，我就痴迷地投入到了提高随机数据处理的速度以及减轻内存负担的工作中。

最后，Python 2.3 中的“适应性强的、自然的、稳定的”归并排序成为了一个很大的成功，但同时也是一个工程上的硬骨头——魔鬼藏在细节之中。该算法的实现包括了大约 1 200 行 C 程序，但不像混合抽样排序的代码，这些代码没有一行是为特例准备的，而且大概有一半是在实现一个技术上的技巧，以使得最坏情况下的内存负担可以减轻一半。我对这个算法感到骄傲，但是引言部分已经没有太多篇幅可供我解释细节了。如果你很好奇，我写了一个很长的技术描述文档，可以在 Python 的源码发行包中找到：主目录中——也就是你解压 Python 源码发行包的地方——（比如，Python-2.3.5 或者 Python-2.4）的 Objects/listsort.txt。在下面的列表中，我提供了 Python 2.3 的归并排序可以利用的偏序的例子，“排序完毕”意味着正向或者反向序。

- 输入序列已经是排序完毕状态。
- 输入序列接近于排序完毕状态，但是有一些随机元素处于尾部或中间，或者两处都有。
- 输入序列是两个或多个排序完毕列表的拼接。事实上，在 Python 中最快的归并多个排序完毕的列表的方法就是先将它们连接起来，然后执行 list.sort 即可。
- 输入序列有多个键对应相同的值。比如，对股票交易所的数据库中的美国公司排序，这些公司大多和 NYSE 或者 NASDAQ 联系起来。算法能够利用此特例的原因是：根据对“稳定”的定义，拥有相同的键的记录已经是排序完毕状态！算法能够自然地探测到这一点，无须特意寻找相等键的代码。
- 输入序列是排序完毕状态，但是不小心掉在地板上摔成很多块了；每一块都在随机的位置，而且其中的某些块内部也已经重新洗牌了。虽然这看上去是挺傻的例子，但它仍然能够被算法利用并提升处理性能，由此可见这种方法的通用性。

长话短说，Python 2.3 的提姆排序（timsort，嗯，它毕竟得有个短点的名字吧）是稳定的、健壮的，而且在实际应用中也快得像飞一样，尽可能的选择用它吧！

5.1 对字典排序

感谢：Alex Martelli

任务

你想对字典排序。这可能意味着需要先根据字典的键排序，然后再让对应值也处于同样的顺序。

解决方案

最简单的方法可以通过这样的描述来概括：先将键排序，然后由此选出对应值：

```
def sortedDictValues(adict):
    keys = adict.keys( )
    keys.sort( )
    return [adict[key] for key in keys]
```

讨论

排序的概念仅仅适用于那些有顺序的集——换句话说，一个序列。而一个映射，比如字典，是没有顺序的，因此它无法被排序。然而，“我怎么才能将一个字典排序”是 Python 邮件列表中一个很常见的问题，理论上这个问题没有什么意义。在绝大多数情况下，其实际目的是将字典中的键构成的序列排序。

至于实现部分，一些人总是考虑更复杂的方式，但其实解决方案中给出的最简单的方法也是最快的方法（对 Python 来说，这样的情况并不少见）。在 Python 2.3 中，在函数的最后的 return 语句中，将列表推导转换成对 map 的调用还可以获得一些速度的提升，大约 20%。比如：

```
return map(adict.get, keys)
```

解决方案中的代码在 Python 2.4 下已经比 Python 2.3 要快了，按照上面的方式进行改写也不会获得很大的速度提升。而使用其他方法，比如用 adict._ _getitem_ _来代替 adict.get，并不会提供任何性能的提升，反而会引起性能的些微下降，无论是在 Python 2.3 还是 2.4 中。

更多资料

5.4 节的方案，根据字典的对应值而不是键进行排序。

5.2 不区分大小写对字符串列表排序

感谢：Kevin Altis、Robin Thomas、Guido van Rossum、Martin V. Lewis、Dave Cross

任务

你想对一个字符串列表排序，并忽略掉大小写信息。举个例子，你想要小写的 a 排在大写的 B 前面。默认的情况下，字符串比较是大小写敏感的（比如所有的大写字符排在小写字符之前）。

解决方案

采用 decorate-sort-undecorate（DSU）用法既快又简单：

```
def case_insensitive_sort(string_list):
    auxiliary_list = [(x.lower( ), x) for x in string_list]  # decorate
    auxiliary_list.sort( )                                   # sort
    return [x[1] for x in auxiliary_list]                    # undecorate
```

Python 2.4 已经提供对 DSU 的原生支持了，因此（假设 string_list 的元素都是真正的普通字符串，而不是 Unicode 对象之类），可以用更简短更快的方式：

```
def case_insensitive_sort(string_list):
    return sorted(string_list, key=str.lower)
```

讨论

一个很明显的可选方案是编写一个比较函数，并将其传递给 sort 方法：

```
def case_insensitive_sort_1(string_list):
     def compare(a, b): return cmp(a.lower( ), b.lower( ))
     string_list.sort(compare)
```

不过，在每次比较中，lower 方法都会被调用两次，而对于长度为 n 的列表来说，比较的次数与 nlog(n)成正比。

DSU 方法则创建了一个辅助的列表，每个元素都是元组，元组的元素则来自原列表并被当做“键”处理。这个排序是基于键的，因为 Python 的元组的比较是根据条目顺序进行的（比如，它会首先比较元组的第一个元素）。要将一个长度为 n 的字符串列表排序，配合 DSU 的使用，lower 方法只需要被调用 n 次，因而在第一步，decorate 阶段，以及最后一步，undecorate 阶段节省了很多时间。

DSU 有时也被称为——但这种叫法不是很准确——Schwartzian 变换，这是对 Perl 的一个著名应用的一个不太准确的类比。（如果要说相似，DSU 更接近于 Guttman-Rosler 变换，见 http://www.sysarch.com/perl/sort_paper.html。）

DSU 是如此重要，因此 Python 2.4 提供了对它的直接支持。可以给列表的 sort 方法传递一个可选的命名参数 key，而且它可以被调用，作用于列表中的每个元素并获得用于排序的键。如果传递这样的一个参数，排序会在内部使用 DSU。因此，在 Python 2.4 中，string_list.sort(key = str.lower 实际上等价于 case_insensitive_sort 函数，只不过 sort 方法

会直接作用于原列表（且返回 None），而不是返回一个排序完毕的拷贝且不对原列表做任何修改。如果你希望 case_insensitive_sort 函数也能够直接作用于原列表，只需要将 return 语句修改为对列表本体的赋值：

```
string_list[:] = [x[1] for x in auxiliary_list]
```

反过来，在 Python 2.4 中，如果你希望获得一个排序完毕的拷贝，且让原列表保持不变，可以使用新的内建的 sorted 函数。比如，在 Python 2.4 中：

```
for s in sorted(string_list, key=str.lower): print s
```

上述代码打印列表中的每一个字符串，这些字符串根据大小写无关的规则进行排序，而且不会影响到 string_list 本身。

在 Python 2.4 的解决方案中，将 str.lower 作为 key 参数限制了你以特定的方式将字符串排序（不包括 Unicode 对象）。如果你知道你正在排序的是 Unicode 对象列表，可以使用 key = unicode.lower。如果你希望函数能够同时适用于普通字符串和 Unicode 对象，可以 import string 并使用 key = string.lower；另外，也可以使用 key = lambda s: s.lower( )。

如果需要对字符串列表进行大小写无关的排序，可能也需要用大小写无关的字符串作为键的字典和集合，需要列表对 index 和 count 方法表现出与大小写无关的行为方式，需要在各种搜索匹配的任务中忽略掉大小写，等等。如果这是你的需求，那么真正需要的是 str 的一个子类型，从而在比较和哈希的时候忽略大小写——相比于实现各种容器和函数来满足这些需求，这是解决这类问题的最好的方法。参考 1.24 节内容，可以看到如何实现这样一种子类型。

更多资料

Python Frequently Asked Questions，http://www.python.org/cgi-bin/faqw.py?req=show&file=faq04.051.htp；5.3 节；Python 2.4 的 *Library Reference* 中关于 sorted 内建函数，sort 和 sorted 的 key 参数；1.24 节。

5.3 根据对象的属性将对象列表排序

感谢：Yakov Markovitch、Nick Perkins

任务

需要根据各个对象的某个属性来完成对整个对象列表的排序。

解决方案

DSU 方法仍然一如既往地有效：

```
def sort_by_attr(seq, attr):
    intermed = [ (getattr(x, attr), i, x) for i, x in enumerate(seq) ]
```

```
        intermed.sort( )
        return [ x[-1] for x in intermed ]
    def sort_by_attr_inplace(lst, attr):
        lst[:] = sort_by_attr(lst, attr)
```

由于 Python 2.4 的对 DSU 的原生支持，代码可以写得更短、跑得更快：

```
    import operator
    def sort_by_attr(seq, attr):
        return sorted(seq, key=operator.attrgetter(attr))
    def sort_by_attr_inplace(lst, attr):
        lst.sort(key=operator.attrgetter(attr))
```

讨论

根据对象属性将对象排序的最佳方法仍然是 DSU，如同前面 5.2 节所介绍的。在 Python 2.3 和 2.4 中，DSU 不再像过去那样，是用来确保排序的稳定性的方法了（因为从 Python 2.3 开始，排序将一直保持稳定），但 DSU 的速度优势仍然如故。

排序，针对最普遍的用例，采用最好的算法，其时间复杂度是 O(nlogn)（如同常见的数学公式一样，这里 n 和 logn 之间是相乘的关系）。通过使用 Python 原生的比较操作（也是最快的），DSU 的速度大多来自于对 O(nlogn)部分的加速，O(nlogn)决定着长度为 n 的序列的排序时间。在预备的 decoration 阶段，即准备辅助的元组列表阶段，以及成功后的 undecoration 阶段，即在完成排序后的中间结果的列表的元组中取出需要的元素的阶段，时间复杂度仅仅是 O(n)。因此，如果 n 非常大，这两个阶段的一些低效的操作并不会造成很大的影响，在实际应用中，它们也确实影响甚微。

O( )记法

当我们需要思考性能问题时，采用的最有效的方法就是众所周知的大 O 分析法以及记法（O 表示的是“order”）。可以在 http://en.wikipedia.org/wiki/Big_O_notation 看到非常详细的解释，但这里我们只是给出了一个概要。

如果我们考虑对尺度为 N 的输入数据采用某个算法，则运行时间是可以描述的，例如针对足够大的 N（具有很大输入数据的应用通常会最关心性能问题），时间和 N 的函数成正比。对于这种表述，我们记录为相应的符号 O(N)（运行时间正比于 N：处理 2 倍的数据需要 2 倍的时间，10 倍的数据则需要 10 倍的时间，以此类推；也被称为线性时间复杂度），O(N squared)（运行时间正比于 N 的平方：处理 2 倍的数据，需要花费 4 倍的时间，10 倍的数据，则需要 100 倍的时间；这也被称为二次方时间复杂度），等等。另一个常见的情况是 O(NlogN)，它比 O(N squared)快但比 O(N)慢。

常数比率通常是被忽略的（至少在理论分析中是这样），因为它常常依赖于一些非算法的因素，如计算机的时钟频率，而不是算法本身。比如你买了一台计算机，它比你的老计算机快两倍，所有的事情处理起来都只需要一半时间，但这并不会改变不同算法之间的差异。

本节的方案是，给 intermed 列表的每一个元素所在的元组中加入了一个索引 i，位于对应的 x 之前（x 是 seq 的第 i 个元素）。这个举措保证了 seq 中任意两个子项都不会被直接用于比较，即使对同一个属性名 attr 它们都具有相同的值。在这种情况下，它们的索引仍然会保持不同，因此基于 Python 的根据条目顺序比较（lexicographical comparison）的规则，元组的最后一个元素（即 seq 的元素）无须被用于比较。避免对象的比较将极大地提高性能。举个例子，我们可以根据 real 属性对一个复数列表进行排序。如果直接比较两个复数，我们会引发一个异常，因为复数之间并没有定义顺序。但是正如我们前面提到的，这样的情况永远不会发生，因此排序将会正确地进行下去。

5.2 节曾经提到过，Python 2.4 直接支持 DSU。可以传递一个可选的关键字参数 key 给 sort，这样每个元素都可以用它来获取排序的键。标准库模块 operator 有两个新函数，attrgetter 和 itermgetter，它们被用来返回适用的可调用体。在 Python 2.4 中，针对这个问题的解决方案就变成了：

```
import operator
seq.sort(key=operator.attrgetter(attr))
```

这个片段执行的排序是直接应用于原列表的，因此速度快得惊人——在我的计算机上，比解决方案给出的第一个 Python 2.3 的函数快 3 倍。如果需要的是一个排序后的拷贝，而不想影响 seq，可以使用 Python 2.4 新的内建的 sorted 函数：

```
sorted_copy = sorted(seq, key=operator.attrgetter(attr))
```

不过它没有直接应用于原列表的排序快，这个代码片段比解决方案的第一个函数快 2.5 倍。另外，Python 2.4 保证了，如果传入了可选的 key 命名参数，列表的元素永远不会被直接比较，因此无须其他的安全保障。而且，排序的稳定性也是有保证的。

更多资料

5.2 节；Python 2.4 的 *Library Reference* 文档中有关 sorted 内建函数，operator 模块的 attrgetter 和 itermgetter 函数，以及 sort 和 sorted 的 key 参数。

5.4 根据对应值将键或索引排序

感谢：John Jensen、Fred Bremmer、Nick Coghlan

任务

需要统计不同元素出现的次数，并且根据它们的出现次数安排它们的顺序——比如，你想制作一个柱状图。

解决方案

柱状图，如果不考虑它在图形图像上的含义，实际上是基于各种不同元素（用 Python

的列表或字典很容易处理）出现的次数，根据对应值将键或索引排序。下面是 dict 的一个子类，它为了这种应用加入了两个方法：

```
class hist(dict):
      def add(self, item, increment=1):
            ''' 为 item 的条目增加计数 '''
            self[item] = increment + self.get(item, 0)
      def counts(self, reverse=False):
            ''' 返回根据对应值排序的键的列表 '''
            aux = [ (self[k], k) for k in self ]
            aux.sort( )
            if reverse: aux.reverse( )
            return [k for v, k in aux]
```

如果想将元素的统计结果放到一个列表中，做法也非常类似：

```
class hist1(list):
      def __init__(self, n):
            ''' 初始化列表，统计 n 个不同项的出现 '''
            list.__init__(self, n*[0])
      def add(self, item, increment=1):
            ''' 为 item 的条目增加计数 '''
            self[item] += increment
      def counts(self, reverse=False):
            ''' 返回根据对应值排序的索引的列表 '''
            aux = [ (v, k) for k, v in enumerate(self) ]
            aux.sort( )
            if reverse: aux.reverse( )
            return [k for v, k in aux]
```

讨论

hist 的 add 方法展示了 Python 用于统计任意（可哈希的）元素的常用方法，并使用 dict 来记录次数。在类 hist1 中，在一个普通的列表的基础上，我们采用了不同的方法，并在__init__中将所有的次数都设置成 0，因而 add 方法就变得更简单了。

counts 方法生成了一个键或者索引的列表，并且根据对应值进行了排序。这两个类针对的问题很类似，因此解决方式也几乎完全一样，都使用了前面 5.2 节和 5.3 节展示过的 DSU。如果我们想要在自己的程序中使用这两个类，由于它们的相似性，我们应该进行代码重构，从中间分离出共性并置入一个单独的辅助函数_sorted_keys：

```
def _sorted_keys(container, keys, reverse):
    ''' 返回 keys 的列表，根据 container 中的对应值排序 '''
    aux = [ (container[k], k) for k in keys ]
    aux.sort( )
    if reverse: aux.reverse( )
    return [k for v, k in aux]
```

然后实现各个类的 counts 方法，其实就是对_sorted_keys 函数进行一层很薄的封装：

```
class hist(dict):
      ...
      def counts(self, reverse=False):
           return _sorted_keys(self, self, reverse)
class hist1(list):
      ...
      def counts(self, reverse=False):
           return _sorted_keys(self, xrange(len(self)), reverse)
```

DSU 在 Python 2.4 中非常重要，前面 5.2 节和 5.3 节已经介绍过了，列表的 sort 方法和新的内建的 sorted 函数提供了一个快速的、原生的 DSU 实现。因此，在 Python 2.4 中，_sorted_keys 还可以变得更简单快速：

```
def _sorted_keys(container, keys, reverse):
     return sorted(keys, key=container.__getitem__, reverse=reverse)
```

被绑定的 container.__getitem__ 方法和 Python 2.3 中实现的获取索引的操作 container[k] 所做的事情完全一样，但是对于我们正在排序的序列而言，它是一个可调用体，可以应用于序列中的每个元素，即命名的键，因此我们可以将它传递给内建 sorted 函数的作为 key 关键字参数的值。Python 2.4 还提供了一个简单直接的方法来获取字典元素根据值排序后的列表：

```
from operator import itemgetter
def dict_items_sorted_by_value(d, reverse=False):
      return sorted(d.iteritems( ), key=itemgetter(1), reverse=reverse)
```

如果想排序一个元素为子容器的容器，Python 2.4 新出现的高级函数 operator.itermgetter 是一个很方便的提供 key 参数的方法，它可以针对每个子容器的特定元素建立键。这正是我们想要的，因为字典的条目实际上就是一个键和值构成的对（两元素的元组）的序列，所谓根据对应值排序，就是根据每个元组的第二个元素进行排序。

回到本节的主题，下面是本节解决方案中的 hist 类的一个使用示例：

```
sentence = ''' Hello there this is a test.  Hello there this was a test,
               but now it is not. '''
words = sentence.split( )
c = hist( )
for word in words: c.add(word)
print "Ascending count:"
print c.counts( )
print "Descending count:"
print c.counts(reverse=True)
```

上述代码片段产生了如下的输出：

```
Ascending count:
[(1, 'but'), (1, 'it'), (1, 'not.'), (1, 'now'), (1, 'test,'), (1, 'test.'),
(1, 'was'), (2, 'Hello'), (2, 'a'), (2, 'is'), (2, 'there'), (2, 'this')]
Descending count:
```

```
[(2, 'this'), (2, 'there'), (2, 'is'), (2, 'a'), (2, 'Hello'), (1, 'was'),
(1, 'test.'), (1, 'test,'), (1, 'now'), (1, 'not.'), (1, 'it'), (1, 'but')]
```

更多资料

Language Reference 的“特殊方法名字”一节以及 *Python in a Nutshell* 中 OOP 章节，特殊的_ _getitem_ _方法；Library Reference 中关于 Python 2.4 的内建 sorted 函数以及 sort 和 sorted 的 key=参数。

5.5 根据内嵌的数字将字符串排序

感谢：Sébastien Keim、Chui Tey、Alex Martelli

任务

你想将一个字符串列表进行排序，这些字符串都含有数字的子串（比如一系列邮寄地址）。举个例子，“foo2.txt”应该出现在“foo10.txt”之前。然而，Python 默认的字符串比较是基于字母顺序的，所以默认情况下，“foo10.txt”会在“foo2.txt”之前。

解决方案

需要先将每个字符串切割开，形成数字和非数字的序列，然后将将每个序列中的数字转化成一个数。这会产生一个数的列表，可以用来做排序时比较的键，可以对这个排序应用 DSU——写两个函数即可，做起来很快捷：

```
import re
re_digits = re.compile(r'(\d+)')
def embedded_numbers(s):
     pieces = re_digits.split(s)                # 切成数字与非数字
     pieces[1::2] = map(int, pieces[1::2])      # 将数字部分转成整数
     return pieces
def sort_strings_with_embedded_numbers(alist):
     aux = [ (embedded_numbers(s), s) for s in alist ]
     aux.sort( )
     return [ s for _ _, s in aux ]             # 惯例：_ _ 意味着“忽略”
```

在 Python 2.4 中，用相同的 embedded_number 函数，加上 DSU 的原生支持，代码变成：

```
def sort_strings_with_embedded_numbers(alist):
     return sorted(alist, key=embedded_numbers)
```

讨论

假设有一个未排序的文件名的列表，比如：

```
files = 'file3.txt file11.txt file7.txt file4.txt file15.txt'.split( )
```

如果只是排序并打印列表，比如在 Python 2.4 中用 print ' '.join(sorted(files))这样的代码，你的输出会是这样：file11.txt file15.txt file3.txt file4.txt file7.txt，因为默认情况下，字符串是根据字母顺序排序的（或者换句话说，排序顺序是由条目顺序指定的）。Python 猜不到你的真实意图其实是希望让它以另外的方式处理那些含有数字的子串，所以必须准确地告诉 Python 你想要什么，解决方案中的代码主要所做的工作其实就是这么一件事。

基于解决方案的代码，也能获得一个更好看的结果：

```
print ' '.join(sort_strings_with_embedded_numbers(files))
```

现在输出变成了 file3.txt file4.txt file7.txt file11.txt file15.txt，这应该正好就是需要的顺序。

这个实现基于 DSU。如果想在 Python 2.3 中达到同样目的，需要手工制作 DSU，但如果你的代码只需要在 Python 2.4 中运行，直接使用原生内建的 DSU 即可。我们传递了一个叫做 key 的参数（一个函数，该函数对每个元素都会调用一次以获取正确的比较键来用于排序）给新的内建函数 sorted。

本节解决方案中的 embedded_numbers 函数正是用来为每个元素获取正确的比较键的方法：一个非数字子串交替出现的列表，int 获取了每个数字子串。re_digits.split(s)给了我们一个交替出现的数字子串和非数字子串的列表（数字子串拥有偶数索引号），然后我们使用了内建的 map 和 int（采用了扩展切片的方式获得并设置了偶数索引号的元素）来将数字序列转化成整数。现在，对这个混合类型的列表进行的条目顺序比较就可以产生正确的结果了。

更多资料

Library Reference 和 *Python in a Nutshell* 文档中关于扩展切片以及 re 模块部分；Python 2.4 *Library Reference* 中的内建函数 sorted 以及 sort 和 sorted 的 key 参数；5.3 节和 5.2 节。

5.6 以随机顺序处理列表的元素

感谢：Iuri Wickert、Duncan Grisby、T. Warner、Steve Holden、Alex Martelli

任务

你想以随机的顺序处理一个很长的列表。

解决方案

一如既往的，在 Python 中最简单的方法常常是最好的。如果我们允许修改输入列表中

的元素的顺序，那么下面的函数就是最简单和最快的：

```
def process_all_in_random_order(data, process):
    # 首先，将整个列表置于随机顺序
    random.shuffle(data)
    # 然后，根据正常顺序访问
    for elem in data: process(elem)
```

如果我们需要保证输入列表不变，或者输入列表可能是其他可迭代对象而不是列表，可以在函数主体开头加上一条赋值语句 data = list(data)。

讨论

虽然过度关心速度常常是个错误，但我们也不能忽略不同算法的性能。假设我们必须以随机顺序处理一个不重复的长列表的元素。第一个想法可能会是这样：我们可以反复地、随机地挑出元素（通过 random.choice 函数），并将原列表中被挑选的元素删除，以避免重复挑选：

```
import random
def process_random_removing(data, process):
    while data:
        elem = random.choice(data)
        data.remove(elem)
        process(elem)
```

然而，这个函数慢得可怕，即使输入列表只有几百个元素。每个 data.remove 调用都会线性地搜索整个列表以获取要删除的元素。由于第 n 步的时间消耗是 O(n)，因此整个处理过程的消耗时间是 O(n2)，正比于列表长度的平方（而且要乘上一个很大的常数）。

对第一个想法的一点提高是将注意力集中在获取随机索引上，并使用列表的 pop 方法来同时获取和删除元素，这种更底层的方式避免了较大的消耗，尤其是在某些情况下，比如要被挑选的元素位于列表的最后，或者使用的压根不是列表，而是字典或集合。若我们面对的是字典或集合，一条思路是寄希望于使用 dict 的 popitem 方法（或者 sets.Set 或 Python 2.4 内建类型 set 的等价的 pop 方法），看上去这个函数好像被设计为随机选择一个元素并删除之，但是，小心上当。

dict.popitem 的文档指出，它返回并删除字典中的任意一个元素，但这和真正的随机元素还差得很远。看看这个：

```
>>> d=dict(enumerate('ciao'))
>>> while d: print d.popitem( )
```

你可能会很吃惊，在大多数的 Python 实现中，这个代码片段都将以看上去不太随机的方式打印 d 的元素，通常是(0, 'c') ，然后(1, 'i')，等等。一句话，如果需要 Python 中的伪随机行为，需要的是标准库的 randompopitem 模块。

如果你考虑使用字典而不是列表，那么你肯定在“Python 式思维”的路上又前进了一步，虽然字典并不会针对这个特定问题提供什么性能优势。但相比于选择正确的数据结构，更具有 Python 风格的方式是：总是利用标准库。Python 标准库是个庞大、丰富的库，塞满了各种有用的、强健的、快速的函数和类，可满足各种应用的需求。在这个前提下，最关键的一点是要意识到，想要以随机的顺序访问序列，最简单的方法是首先将序列转化成随机的顺序（也被称为对序列洗牌，是对扑克洗牌的类比），然后再线性的访问洗完牌的序列即可。random.shuffle 函数就可以执行洗牌操作，本节解决方案正是利用了这个函数。

实际性能总是需要测试，而不是猜测出来的，那也正是标准库模块 timeit 存在的原因。使用一个空的 process 函数和一个长度为 1 000 的列表作为 data，process_all_in_random_order 能比 process_random_removing 快大约 10 倍；对于长度为 2 000 的列表，这个比例变成了 20。如果提升仅仅是 25%，或者是一个常数因子 2，那么通常这个性能差异是可以忽略的，因为这不会对你的整体应用产生什么性能影响，但如果算法慢了 10 或 20 倍，情况就不同了。这种可怕的低效会成为整个程序的瓶颈。当我们谈到 O(n2)和 O(n)的行为对比时，问题的严重性根本无法忽视：对于这两种大 O 的行为，随着输入数据的增长，它们消耗时间的差异可以无限地递增下去。

更多资料

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

5.7 在增加元素时保持序列的顺序

感谢：John Nielsen

任务

你需要维护一个序列，这个序列不断地有新元素加入，但始终处于排序完毕的状态，这样你可以在任何需要的时候检查或者删除当前序列中最小的元素。

解决方案

假设有一个未排序的列表，比如：

```
the_list = [903, 10, 35, 69, 933, 485, 519, 379, 102, 402, 883, 1]
```

可以调用 the_list.sort( )将列表排序，然后用 result = the_list.pop(0)来获得和删除最小的元素。但是，每当加入一个元素（比如 the_list.append(0)），都需要再次调用 the_list.sort 来排序。

可以使用 Python 标准库的 heapq 模块：

```
import heapq
heapq.heapify(the_list)
```

现在列表并不一定完成了排序，但是它却满足堆的特性（若所有涉及的索引都是有效的，则 the_list[i]< = the_list[2*i + 1] 且 the_list[i]< = the_list[2*i+2]），所以，the_list[0]就是最小的元素。为了保持堆特性的有效性，我们使用 result = heapq.heappop(the_list)来获取并删除最小的元素，用 heapq.heappush(the_list, newitem)来加入新的元素。如果需要同时做这两件事：加入一个新元素并删除之前的最小的元素，可以用 result=heapq.heapreplace (the_list, newitem)。

讨论

当需要以一种有序的方式获取数据时（每次都选择你手中现有的最小元素），可以选择在获取数据时付出运行时代价，或者在加入数据时付出代价。一种方式是将数据放入列表并对列表排序。这样，可以很容易地让你的数据按照顺序从小到大排列。然而，你不得不每次在加入新数据时调用 sort，以确保每次在增加新元素之后仍能够获取最小的元素。Python 列表的 sort 实现采用了一种不太有名的自然的归并排序，它的排序开销已经被尽力地压缩了，但仍然难以让人接受：每次添加（和排序）及每次获取（以及删除，通过 pop）的时间，与当前列表中元素的数目成正比（O(N)，准确地说）。

另一种方案是使用一种叫做堆的组织数据的结构，这是一种简洁的二叉树，它能确保父节点总是比子节点小。在 Python 中维护一个堆的最好方式是使用列表，并用库模块 heapq 来管理此列表，如同本节解决方案所示的那样。这个列表无须完成排序，但你却能够确保每次你调用 heappop 从列表中获取元素时，总是得到当前最小的元素，然后所有节点会被调整，以确保堆特性仍然有效。每次通过 heappush 添加元素，或者通过 heappop 删除元素，它们所花费的时间都正比于当前列表长度的对数（O(logN)，准确地说）。只需要付出一点点代价（从总体来说，代价也非常小）。

举例来说，很适合使用堆方式的场合是这样的：假设有一个很长的队列，并且周期性地有新数据到达，你总是希望能够从队列中获取最重要的元素，而无须不断地重新排序或者在整个队列中搜索。这个概念叫做优先级队列，而堆正是最适合实现它的数据结构。注意，本质上，heapq 模块在每次调用 heappop 时向你提供最小的元素，因此需要安排你的元素的优先级值，以反映出元素的这个特点。举个例子，假设你每次收到数据都付出一个价钱，而任何时候最重要的元素都是队列中价钱最高的那个；另外，对于价钱相同的元素，先到达的要重要一些。下面是一个创建“优先级队列”的类，我们遵循上面提到的要求并使用了 heapq 模块的函数：

```
class prioq(object):
      def __init__(self):
            self.q = [  ]
            self.i = 0
      def push(self, item, cost):
```

```
            heapq.heappush(self.q, (-cost, self.i, item))
            self.i += 1
        def pop(self):
            return heapq.heappop(self.q)[-1]
```

这段代码的意图是将价钱设置为负，作为元组的第一个元素，并将整个元组压入堆中，这样更高的出价会产生更小的元组（基于 Python 的自然比较方式）；在价钱之后，我们放置了一个递增的索引，这样，当元素拥有相同的价钱时，先到达的元素将会处于更小的元组中。

在 Python 2.4 中，heapq 模块又被重新实现和进一步优化了，见 5.8 节中更多有关 heapq 的信息。

更多资料

Library Reference 和 *Python in a Nutshell* 中 heapq 模块的文档；Python 源码的 heapq.py 中包含了有关堆的一些非常有趣的讨论；5.8 节中更多的关于 heapq 的信息；19.14 节中使用 heapq 对完成排序的多个序列进行合并。

5.8 获取序列中最小的几个元素

感谢：Matteo Dell'Amico、Raymond Hettinger、George Yoshida、Daniel Harding

任务

你需要从一个给定的序列中获取一些最小的元素。可以将序列排序，然后使用 seq[:n]，但还有没有更好的办法呢？

解决方案

如果需要的元素数目 n 远小于序列的长度，也许还能做得更好。sort 是很快的，但它的时间复杂度仍然是 O(nlogn)，但如果 n 很小，我们获取前 n 个最小元素的时间是 O(n)。下面给出一个简单可行的生成器，在 Python 2.3 和 2.4 中都同样有效：

```
import heapq
def isorted(data):
    data = list(data)
    heapq.heapify(data)
    while data:
        yield heapq.heappop(data)
```

在 Python 2.4 中，如果事先知道 n，还有更简单和更快的方法从 data 中获取前 n 个最小的元素：

```
import heapq
def smallest(n, data):
    return heapq.nsmallest(n, data)
```

讨论

data 可能是任何有边界的可迭代对象，解决方案中的 isorted 函数通过调用 list 来确保它是序列。也可以删掉 data = list(data)这一行，假如下列条件满足的话：你知道 data 是一个序列，你并不关心生成器是否重新排列了 data 的元素，而且需要在获取的同时从 data 中删除元素。

- 如同 5.7 节所示，Python 标准库提供了 heapq 模块，它支持人们熟知的数据结构——堆。解决方案中的 isorted 先创建了一个堆（通过 heap.heapify），然后在每一步获取元素的时候（通过 heap.heappop），生成并删除堆中最小的元素。

在 Python 2.4 中，heapq 模块引入了两个新函数。heapq.nlargest (n, data) 返回的是一个长度为 n 的 data 中最大的元素的列表，heapq.nsmallest (n, data) 则返回一个包含前 n 个最小元素的列表。这些函数并不要求 data 满足堆的条件，它们甚至不要求 data 是一个列表——任何有边界的、元素是可以比较的可迭代对象都适用。解决方案中的函数 smallest 除了调用 heapq.smallest，其实什么也没干。

关于速度，我们总是应该进行实际测量，猜测不同代码片段的相对运行速度是毫无意义的行为。因此，当我们只是循环获取前几个（最小）元素的时候，isorted 的性能和 Python 2.4 内建的 sorted 函数的性能比起来究竟怎么样呢？为了进行测试，我写了一个 top10 函数，它可以调用这两种方法，然后我也为 Python 2.3 实现了一个 sorted 函数，因为在 Python 2.3 中此函数并未得到原生的支持：

```
try:
    sorted
except:
    def sorted(data):
          data = list(data)
          data.sort( )
          return data
import itertools
def top10(data, howtosort):
    return list(itertools.islice(howtosort(data), 10))
```

在我的计算机上，在 Python 2.4 中处理一个洗过牌的 1 000 个整数的列表，top10 调用 isorted 耗时 260μs，但采用内建的 sorted 则耗时 850μs。而 Python 2.3 甚至还要慢得多：isorted 耗时 12ms，sorted 耗时 2.7ms。换句话说，Python 2.3 的 sorted 比 Python 2.4 的 sorted 要慢 3 倍，但在 isorted 上则要慢 50 倍。需要记住这个重要的经验：当需要进行优化时，首先进行测量。不应该仅仅根据基本原理选择优化的方式，因为真实的性能数据变化多端，即使在两个兼容的发行版本之间也会有很大的差异。第二个经验是：如果真的很关心性能，那赶紧转移到 Python 2.4 吧。和 Python 2.3 相比，Python 2.4 经过了极大的优化和加速，特别是在搜索和排序的方面。

如果确定你的代码只需要支持 Python 2.4，那么，如同本节解决方案所建议的那样，应

当使用 heapq 的新函数 nsmallest。它速度快、使用简便，远胜于自己编写代码。举个例子，实现 Python 2.4 中的 top10，你只需要：

```
import heapq
def top10(data):
     return heapq.nsmallest(10, data)
```

比起前面展示的那个基于 isorted 的 top10，对同样的洗过牌的 1 000 个整数的列表进行处理，消耗的时间还可以削减一半。

更多资料

Library Reference 和 *Python in a Nutshell* 中 list 类型的 sort 方法，以及 heapq 和 timeit 模块；第 19 章中关于 Python 的迭代的内容；*Python in a Nutshell* 中有关优化的章节；Python 源码中的 heap.py 所包含的关于堆的有趣的讨论；5.7 节关于 heapq 的介绍。

5.9 在排序完毕的序列中寻找元素

感谢：Noah Spurrier

任务

你需要寻找序列中的一系列元素。

解决方案

如果列表 L 已经是排序完毕的状态，则 Python 标准库提供的 bisect 模块可以很容易地检查出元素 x 是否在 L 中：

```
import bisect
x_insert_point = bisect.bisect_right(L, x)
x_is_present = L[x_insert_point-1:x_insert_point] == [x]
```

讨论

对 Python 来说，在列表 L 中寻找一个元素 x 是很简单的任务：要检查元素是否存在，if x in L；要知道 x 的确切位置，L.index(x)。然而，L 的长度为 n，这些操作所花费的时间与 n 成正比，因为它们只是循环地检查每一个元素，看看是否与 x 相等。其实，如果 L 是排过序的，我们还可以做得更好。

在完成了排序的序列中寻找元素的最经典算法就是二分搜索，因为每一步它都将查找的范围减半——一般情况下它只需要$\log_2 n$步即可完成搜索。但当需要多次查找某些元素时，这个问题就很值得仔细探讨了，通过使用一些方法，可以为多次搜索尽量少付出一些开销。在调用L.sort( )之后，一旦你决定用二分搜索在L中查找x，应该马上想到Python标准库的bisect模块。

具体地说，我们需要 bisect.bisect_right 函数来保持原列表的排序状态，它将返回一个索引，指示出我们要插入的元素的位置，而且它也不会修改列表；如果列表中已经存在相等的元素了，bisect_right 将返回拥有相同值的元素右边邻接的索引。因此，在调用 bisect.bisect_right(L, x)获得了“插入点”之后，只需立刻检查插入点之前的位置，看看是否已经有一个等于 x 的元素存在了。

解决方案中计算 x_is_present 的方式可能不是那么直观。如果知道 L 不是空列表，我们还能写得更简单、更直观：

```
x_is_present = L[x_insert_point-1] == x
```

不过，这个更简单的方式在对空列表进行索引操作的时候会引发异常。当切片的边界无效时，切片操作比索引操作更加“不严谨”，因为它只是生成了一个空的切片，没有引发任何异常。一般来说，当 i 是 somelist 的有效索引时，somelist[i:i+1]是和[somelist[i]]一样的单元素列表，但当索引操作引发 IndexError 异常时，它却是一个空的列表[ ]。对 x_is_present 的计算充分利用了这种重要的特性，避免了处理异常，同时也能以统一的方式对 L 处理空和非空的情况。另一个可选的方式是：

```
x_is_present = L and L[x_insert_point-1] == x
```

这个方式利用了 and 的短路的行为模式来保护索引操作，避免了使用切片操作。

如果元素是可哈希的（意味着元素可以被用作 dict 的键），如 5.12 节的做法一样，使用一个辅助的 dict 也是一个可行的方法。不过，若元素是可比较的（comparable，若不可比较，排序是不可能实行的）且不可哈希的（因此字典无法将它们当键用），本节的这个基于已经排序的列表的方法就可能是唯一可行的方案了。

如果列表已经排序完毕，而且需要查找的元素数目不是非常多，那么大多数情况下，用 bisect 要比构建一个辅助字典快，因为在构建字典上的时间投资无法被大量查找分摊。尤其是在 Python 2.4 中，bisect 已经被高度优化，比在 Python 2.3 中的对应版本要快得多。比如，在我的计算机上，用 bisect.bisect_right 在含有 10 000 个元素的列表中查找一个元素，Python 2.4 要比 Python 2.3 快约 4 倍。

更多资料

Library Reference 和 *Python in a Nutshell* 中的 bisect 模块；5.12 节。

5.10 选取序列中最小的第 n 个元素

感谢：Raymond Hettinger、David Eppstein、Shane Holloway、Chris Perkins

任务

需要根据排名顺序从序列中获得第 n 个元素（比如，中间的元素，也被称为中值）。如

果序列是已经排序的状态，应该使用 seq[n]，但如果序列还未被排序，那么除了先对整个序列进行排序之外，还有没有更好的方法？

解决方案

如果序列很长，洗牌洗得很充分，而且元素之间的比较开销也大，那么也许还能找到更好的方式。排序的确很快，但不管怎样，它（一个长度为 n 的充分洗牌的序列）的时间复杂度仍然是 O(nlogn)，而时间复杂度为 O(n)的取得第 n 个最小元素的算法也的确是存在的。下面我们给出一个函数来实现此算法：

```
import random
def select(data, n):
    " 寻找第 n 个元素 (最小的元素是第 0 个). "
    # 创建一个新列表，处理小于 0 的索引，检查索引的有效性
    data = list(data)
    if n<0:
        n += len(data)
    if not 0 <= n < len(data):
        raise ValueError, "can't get rank %d out of %d" % (n, len(data))
    # 主循环，看上去类似于快速排序但不需要递归
    while True:
        pivot = random.choice(data)
        pcount = 0
        under, over = [ ], [ ]
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
        numunder = len(under)
        if n < numunder:
            data = under
        elif n < numunder + pcount:
            return pivot
        else:
            data = over
            n -= numunder + pcount
```

讨论

本节解决方案也可用于重复的元素。举个例子，列表[1, 1, 1, 2, 3]的中值是 1，因为它是将这 5 个元素按顺序排列之后的第 3 个。如果由于某些特别的原因，你不想考虑重复，而需要缩减这个列表，使得所有元素都是唯一的（比如，通过 18.1 节提供的方法），可以完成这一步骤之后再回到本节的问题。

输入参数 data 可以是任意有边界的可迭代对象。首先我们对它调用 list 以确保得到可迭代的对象，然后进入持续循环过程。在循环的每一步中，首先随机选出一个轴心元素；以轴心元素为基准，将列表切片成两个部分，一个部分“高于”轴心，一个部分“低于”轴心；然后继续在下一轮循环中对列表的这两个部分中的一个进行深入处理，因为我们可以判断第 n 个元素处于哪一个部分，所以另外一个部分就可以丢弃了。这个算法的思想很接近经典的快速排序算法（只不过快速排序无法丢弃任何部分，它必须用递归的方法，或者用一个栈来替换递归，以确保对每个部分都进行了处理）。

随机选择轴心使得这个算法对于任意顺序的数据都适用（但不同于原生的快速排序，某些顺序的数据将极大地影响它的速度）；这个实现花费大约$\log_2 N$时间用于调用 random.choice。另一个值得注意的是算法统计了选出轴心元素的次数，这是为了在一些特殊情况下仍能够保证较好的性能，比如data中可能含有大量的重复数据。

将局部变量列表 under 和 over 的被绑定方法 append 抽取出来，看起来没什么意义，而且还增加了一点小小的复杂性，但实际上，这是 Python 中一个非常重要的优化技术。为了保持编译器的简单、直接、可预期性以及健壮性，Python 不会将一些恒定的计算移出循环，它也不会“缓存”对方法的查询结果。如果你在内层循环调用 under.append 和 over.append，每一轮都会付出开销和代价。如果想把某些事情一次性做好，那么需要自己动手完成。当你考虑优化问题时，你总是应该对比优化前和优化后的效率，以确保优化真正起到了作用。根据我的测试，对于获取 range(10 000)的第 5 000 个元素这样的任务，去掉优化部分之后，性能下降了 50%。虽然增加一点小小的复杂性，但仍然是值得的，毕竟那是 50%的性能差异。

关于优化的一个自然的想法是，在循环中调用 cmp(elem, pivot)，而不是用一些独立的 elem < piovt 或 elem > pivot 来测试。不幸的是，cmp 不会提高速度；事实上它还有可能会降速，至少当 data 的元素是一些基本类型比如数字和字符串的时候，的确是这样。

那么，select 的性能和下面这个简单方法的性能相比如何呢？

```
def selsor(data, n):
    data = list(data)
    data.sort( )
    return data[n]
```

在我的计算机上，获取一个 3 001 个整数的充分洗牌的列表的中值，本节解决方案的代码耗时 16ms，而 selsor 耗时 13ms，再考虑到 sort 在序列部分有序的情况下速度会更快，元素是基本类型且比较操作执行得很快，而且列表长度也不大，所以使用 select 并没有什么优势。将长度增加到 30 001，这两个方法的性能变得非常接近，都是约 170ms。但当我将列表长度修改成 300 001，select 终于表现出了优势，它用了 2.2s 获得了中值，而 selsor 需要 2.5s。

但如果序列中元素的比较操作非常耗时，那么这两个方式刚刚表现出的大致平衡就被

彻底打破了，因为这两个方式的最关键的差异就是比较操作执行的次数——select 执行 O(n)次，而 selsor 执行 O(nlogn)次。举个例子，假如我们需要比较的是某个类的实例，其比较操作的开销相当大（模拟某些四维的坐标点，其前三维坐标通常总是相同的）：

```
class X(object):
     def __init__(self):
          self.a = self.b = self.c = 23.51
          self.d = random.random( )
     def _dats(self):
          return self.a, self.b, self.c, self.d
     def __cmp__(self, oth):
          return cmp(self._dats, oth._dats)
```

现在，即使只对 201 个实例求中值，select 就已经表现得比 selsor 快了。

换句话说，基于列表的 sort 方法的实现的确要简洁得多，实现 select 也确实需要多付出一点力气，但如果 n 足够大而且比较操作的开销也无法忽略的话，select 就体现出它的价值了。

更多资料

Library Reference 和 *Python in a Nutshell* 文档中关于 list 类型的 sort 方法，以及 random 模块。

5.11 三行代码的快速排序

感谢：Nathaniel Gray、Raymond Hettinger、Christophe Delord、Jeremy Zucker

任务

你想证明，Python 对函数式编程范式的支持比第一眼看上去的印象强多了。

解决方案

函数式编程语言非常漂亮，比如 Haskell 就是个例子，但 Python 的表现也不遑多让：

```
def qsort(L):
    if len(L) <= 1: return L
    return qsort([lt for lt in L[1:] if lt < L[0]]) + L[0:1] + \
           qsort([ge for ge in L[1:] if ge >= L[0]])
```

我个人认为，上述代码和 Haskell 的版本（http://www.haskell.org）比起来一点也不逊色：

```
qsort [  ] = [  ]
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
                  where
```

```
elts_lt_x = [y | y <- xs, y < x]
elts_greq_x = [y | y <- xs, y >= x]
```

下面给出一个 Python 版本的测试函数：

```
def qs_test(length):
    import random
    joe = range(length)
    random.shuffle(joe)
    qsJoe = qsort(joe)
    for i in range(len(qsJoe)):
        assert qsJoe[i] == i, 'qsort is broken at %d!' %i
```

讨论

相比于快速排序的原生实现，我们给出的实现能展现列表推导的强大表现力。但千万不要在真实的代码中使用这种方法。Python 列表有一个 sort 方法，速度要比我们的实现快得多，无疑应该是首选的方案：在 Python 2.4 中，新的内建函数 sorted 接受任意的有限序列并返回一个新的完成了排序的序列。这段代码的唯一用处是向朋友们炫耀，尤其是那些函数式编程的狂热分子，比如 Haskell 语言的爱好者。

我在 http://www.haskell.org/aboutHaskell.html 看到了 Haskell 的漂亮的快速排序之后，产生了写一个 Python 的对应版本的想法。在看到 Haskell 的函数优雅的表现能力之后，我意识到借助 Python 中的列表推导也能以同样方式实现。除了使用从 Haskell 偷学来的列表推导，我们也设法加入了很多 Python 风格的东西。

这两个实现都是以列表的第一个元素为轴心，因而对于一个普通的，已经完成排序的列表它们会有最差的性能表现 O(n)。绝不应该在工作代码中使用它，但既然本节的目标是显摆，那也无所谓了。

还可以写一个没有那么紧凑，但具有同样结构的版本，并使用命名的局部变量和函数来增加清晰度和可读性：

```
def qsort(L):
    if not L: return L
    pivot = L[0]
    def lt(x): return x<pivot
    def ge(x): return x>=pivot
    return qsort(filter(lt, L[1:]))+[pivot]+qsort(filter(ge, L[1:]))
```

一旦沿着这条思路走下去，就可以很容易地对原来的做法继续改进，比如使用一个随机的轴心元素来尽可能地避免最糟糕的情况，并对选用的轴心计数，来应对序列中有太多相等地元素的情况：

```
import random
def qsort(L):
    if not L: return L
    pivot = random.choice(L)
```

```
        def lt(x): return x<pivot
        def gt(x): return x>pivot
        return qsort(filter(lt, L))+[pivot]*L.count(pivot)+qsort(filter(gt,
L))
```

虽然代码得到了增强，但是看上去没那么有趣了，而且不利于炫耀。而真正的工作级别的排序代码是另一码事：不管我们多么喜欢这些看上去可爱的代码，它们在性能和稳固度方面永远也无法和 Python 内建的排序相提并论。

除了提高清晰度和健壮性，我们还可以把力气用在相反的方向上，即利用 Python 的 lambda 写出更紧凑和晦涩的东西：

```
q=lambda x:(lambda o=lambda s:[i for i in x if cmp(i,x[0])==s]:
            len(x)>1 and q(o(-1))+o(0)+q(o(1)) or x)( )
```

至少，它还足够漂亮（一行代码，当然由于长度的问题只好被截为两行），但你应该明白，真实的应用绝对不应该是这样的。我们用可读性较好的 def 语句来替换掉晦涩难懂的 lambda，得到的等价代码仍然不是很好读：

```
def q(x):
    def o(s): return [i for i in x if cmp(i,x[0])==s]
    return len(x)>1 and q(o(-1))+o(0)+q(o(1)) or x
```

但如果我们把那个过于精炼的 len(x)>1 and . . . or x 拆开，并用 if/else 语句代替，同时引入一些局部名字，清晰度似乎提高了不少：

```
def q(x):
     if len(x)>1:
         lt = [i for i in x if cmp(i,x[0]) == -1 ]
         eq = [i for i in x if cmp(i,x[0]) == 0 ]
         gt = [i for i in x if cmp(i,x[0]) == 1 ]
         return q(lt) + eq + q(gt)
     else:
         return x
```

幸亏在真实世界中，Python 用户并不喜欢写出那种盘绕的、层层叠叠的充满 lambda 的代码。事实上，很多（但也得承认不是所有）人都对 lambda 感到厌烦（可能是由于看到过多的对 lambda 的滥用），因此都尽量地使用可读性较好的 def 语句。因此，Python 并不要求用户具有能够从乱麻一般的代码中解码的能力，但也许别的语言是需要这种能力的。语言的任何一个特性都可能被程序员滥用，以显得更“聪明”。这样导致的结果是，一些 Python 用户（虽然是少数）甚至对列表推导（列表推导可以被揉进一堆事物，和简单的 for 循环相比，的确看上去没那么清晰）和 and/or 操作符的短路行为模式（因为利用它们可以写出非常晦涩精炼的代码，完全无法和 if 语句的清晰易读相比）也感到恐惧。

更多资料

Haskell 的主页，http://www.haskell.org。

5.12 检查序列的成员

感谢：Alex Martelli

任务

你需要对一个列表执行很频繁的成员资格检查。而 in 操作符的 O(n)时间复杂度对性能的影响很大，你也不能将序列转化为一个字典或者集合，因为你还需要保留原序列的元素顺序。

解决方案

假设需要给列表添加一个在该列表中不存在的元素。一个可行的方法是写这样一个函数：

```
def addUnique(baseList, otherList):
    auxDict = dict.fromkeys(baseList)
    for item in otherList:
        if item not in auxDict:
              baseList.append(item)
              auxDict[item] = None
```

如果你的代码只是在 Python 2.4 下运行，那么将辅助字典换成辅助集合效果是完全一样的。

讨论

下面给出一个简单（天真？）的方式，看上去相当不错：

```
def addUnique_simple(baseList, otherList):
      for item in otherList:
          if item not in baseList:
             baseList.append(item)
```

如果列表很短的话，这个方法倒也没问题。

但是，如果列表不是很短，这个简单的方法会非常慢。当你用 if item not in baseList 这样的代码进行检查时，Python 只会用一种方式执行 in 操作：对列表 baselist 的元素进行内部的循环遍历，如果找到一个元素等于 item 则返回 True，如果直到循环结束也没有发现相等的元素则返回 False。in 操作的平均执行时间是正比于 len(baseList)的。addUnique_simple 执行了 len(otherList)次 in 操作，因此它消耗的时间正比于这两个列表长度的乘积。

而解决方案给出的 addUnique 函数，首先创建了一个辅助的字典 auxDict，这一步的时间正比于 len(baseList)。然后在循环中检查 dict 的成员——这是造成巨大差异的一步，

因为检查一个元素是否处于一个 dict 中的时间大致是一个常数，而与 dict 中元素的数目没有关系。因此，那个 for 循环消耗的时间正比于 len(otherList)，这样，整个函数所需要的时间就正比于这两个列表的长度之和。

对于运行时间的分析还可以挖得更深一点，因为在 addUnique_simple 中 baseList 的长度并不是一个常量；每当找到一个不属于 baseList 的元素，baseList 的长度就会增加。但这样的分析结果不会与前面的简化版的结果有太大出入。我们可以准备一些用例进行测试。当每个列表中有 10 个整数且有 50%的重叠时，简化版比解决方案给出的函数慢 30%，这样的性能下降还可以忽略。若每个列表都有 100 个整数，而且仍然有 50%的重叠部分，简化版比解决方案的函数慢 12 倍——这种级别的减速效果就无法忽略了，而且当列表变得更长的时候，情况也变得更糟。

有时，将一个辅助的 dict 和序列一起使用并封装成一个对象能提高你的应用程序的性能。但在这个例子中，必须在序列被修改时不断地维护 dict，以保证它总是和序列当前所拥有的元素保持同步。这个维护任务并不是很简单，我们有很多方法来实现同步。下面给出一种“即时”的同步方式，当需要检查某元素，或者字典的内容可能已经无法和列表内容保持同步时，我们就重新构建一个辅助 dict。由于开销很小，下面的类优化了 index 方法和成员检查部分的代码：

```
class list_with_aux_dict(list):
     def __init__(self, iterable=( )):
          list.__init__(self, iterable)
          self._dict_ok = False
     def _rebuild_dict(self):
          self._dict = {  }
          for i, item in enumerate(self):
               if item not in self._dict:
                    self._dict[item] = i
          self._dict_ok = True
     def __contains__(self, item):
          if not self._dict_ok:
               self._rebuild_dict( )
          return item in self._dict
     def index(self, item):
          if not self._dict_ok:
               self._rebuild_dict( )
          try: return self._dict[item]
          except KeyError: raise ValueError
def _wrapMutatorMethod(methname):
     _method = getattr(list, methname)
     def wrapper(self, *args):
         # 重置字典的 OK 标志，然后委托给真正的方法
         self._dict_ok = False
         return _method(self, *args)
     #只适用于 Python 2.4:  wrapper.__name__ = _method.__name__
```

```
        setattr(list_with_aux_dict, methname, wrapper)
    for meth in 'setitem delitem setslice delslice iadd'.split( ):
         _wrapMutatorMethod('__%s__' % meth)
    for meth in 'append insert pop remove extend'.split( ):
         _wrapMutatorMethod(meth)
    del _wrapMethod              # 删除辅助函数，已经不再需要它了
```

list_with_aux_dict 扩展了 list，并将原 list 的所有方法仍然委托给它，除了__contains__和 index。所有能够修改 list 的方法都被封装进了一个闭包，该闭包负责重置一个标志，以确保辅助字典的有效性。Python 的 in 操作符调用__contains__方法。除非标志被设置，否则 list_with_aux_dict 的__contains__方法会重建辅助字典（标志被设置时，重建没有必要），而 index 方法仍然像原先一样工作。

上述 list_with_aux_dict 类并没有用帮助函数为列表的所有属性方法绑定和安装一个闭包，而是只取所需，我们也可以在 list_with_aux_dict 的主体中写出所有的 def 语句来替代 wrapper 方法。但是上述代码有个重要的优点是消除了冗余和重复（重复和啰嗦的代码让人生厌，而且容易滋生 bug）。Python 在自省和动态改变方面的能力给你提供了一个选择：可以创建一个 wrapper 方法，用一种聪明而简练的方式；或者，如果你想避免使用被人称为黑魔法的类对象的自省和动态改变，也可以写一堆重复啰嗦的代码。

list_with_aux_dict 的结构很适合通常的使用模式，即对序列的修改操作一般总是集中出现，然后接着又会有一段时间序列无须被修改，但需要检查元素的成员资格。如果参数 baseList 不是一个普通的列表，而是 list_with_aux_dict 的一个实例，早先展示的 addUnique_simple 函数也不会因此得到任何性能上的提升，因为这个函数会交替地进行成员资格检查和序列修改。因此，类 list_with_aux_dict 中过多的辅助字典的重建影响了函数的性能。（除非是针对某个特例，比如 otherList 中绝大多数元素都已经在 baseList 中出现过了，因此对序列的修改相比于对元素的检查，发生的次数要少得多。）

对这些成员资格的检查所做的优化有个重要的前提，即序列中的值必须是可哈希的（不然的话，它们不能被用来做字典的键或者集合的元素）。举个例子，元组的列表仍适用于本节的解决方案，但对于列表的列表，我们恐怕得另外想办法了。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于序列类型和映射类型的章节。

5.13 寻找子序列

感谢：David Eppstein、Alexander Semenov

任务

你需要在某大序列中查找子序列。

解决方案

如果序列是字符串（普通的或者 Unicode），Python 的字符串的 find 方法以及标准库的 re 模块是最好的工具。否则，应该使用 Knuth-Morris-Pratt 算法（KMP）：

```
def KnuthMorrisPratt(text, pattern):
    ''' 在序列 text 中找到 pattern 的子序列的起始位置
        每个参数都可以是任何可迭代对象
        在每次产生一个结果时，对 text 的读取正好到达（包括）
        对 pattern 的一个匹配 '''
    # 确保能对 pattern 进行索引操作，同时制作 pattern 的
    # 一个拷贝，以防在生成结果时意外地修改 pattern
    pattern = list(pattern)
    length = len(pattern)
    # 创建 KMP “偏移量表” 并命名为 shifts
    shifts = [1] * (length + 1)
    shift = 1
    for pos, pat in enumerate(pattern):
        while shift <= pos and pat != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift
    # 执行真正的搜索
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == length or matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == length: yield startPos
```

讨论

本节实现的 Knuth-Morris-Pratt 算法可被用于在一个大文本的连续序列中查找某种指定的模式。由于 KMP 是以顺序的方式访问文本，所以很自然地，我们也可以将其应用于包括文本在内的任意可迭代对象。在处理阶段，算法会创建一个关于偏移量的表，它消耗的时间正比于模式的长度，而每个文本标志则以恒定的时间处理。在所有关于文本处理的基础算法书中都可以看到 KMP 算法的解释和示例。（更多资料一栏中也提供了推荐读物。）

如果 text 和 pattern 都是 Python 字符串，可以通过使用 Python 的内建搜索方法得到一个更快的方案：

```
def finditer(text, pattern):
    pos = -1
    while True:
        pos = text.find(pattern, pos+1)
```

```
            if pos < 0: break
            yield pos
```

比如，使用一个长度为 4 的字母表（“ACGU”），在长度为 100000 的文本中查找长度为 8 的一个模式，在我的计算机上，借助 finditer 函数耗时为 4.3ms，但使用 KnuthMorrisPratt 函数执行同样任务则需要 540ms（在 Python 2.3 中；而在 Python 2.4 会快一些，约 480ms，但仍然比 finditer 慢了超过 100 倍）。所以请记住：本节的算法适用于在通用的序列中进行搜索，包括那些数据量大到无法放入内存的情况，如果只需要对字符串搜索，Python 内建的搜索方法具有完全压倒性的优势。

更多资料

关于基础算法的优秀书籍有很多；其中一本备受推崇的书是 Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest、Clifford Stein，合著的 Introduction to Algorithms（MIT Press），第二版。

5.14 给字典类型增加排名功能

感谢：Dmitry Vasiliev、Alex Martelli

任务

你需要用字典存储一些键和“分数”的映射关系。你经常需要以自然顺序（即以分数的升序）访问键和分数值，并能够根据那个顺序检查一个键的排名。对这个问题，用 dict 似乎不太合适。

解决方案

我们可以使用 dict 的子类，根据需要增加或者重写一些方法。在我们使用多继承、将 UserDict.DictMixin 放置在基类 dict、并仔细安排各种方法的委托或重写之前，我们可以设法获得一种美妙的平衡，既拥有极好的性能又避免了编写一些冗余代码。

我们可以在文档字符串中加入很多示例，还可以用标准库的 doctest 模块来提供单元测试的功能，这也能够确保我们在文档字符串中编写的例子的准确性：

```
#!/usr/bin/env python
''' 一个反映键到分数的映射的字典 '''
from bisect import bisect_left, insort_left
import UserDict
class Ratings(UserDict.DictMixin, dict):
    """ Ratings 类很像一个字典，但有一些额外特性：每个键
        的对应值都是该键的“分数”，所有键都根据它们的
        分数排名。对应值必须是可以比较的，同样，键则必须
        是可哈希的（即可以“绑”在分数上）
```

所有关于映射的行为都如同预期一样，比如：

```
>>> r = Ratings({"bob": 30, "john": 30})
>>> len(r)
2
>>> r.has_key("paul"), "paul" in r
 (False, False)
>>> r["john"] = 20
>>> r.update({"paul": 20, "tom": 10})
>>> len(r)
4
>>> r.has_key("paul"), "paul" in r
 (True, True)
>>> [r[key] for key in ["bob", "paul", "john", "tom"]]
 [30, 20, 20, 10]
>>> r.get("nobody"), r.get("nobody", 0)
 (None, 0)
```

除了映射的接口，我们还提供了和排名相关的方法。`r.rating(key)`返回了某个键的排名，其中排名为 0 的是最低的分数（如果两个键的分数相同，则直接比较它们两者，“打破僵局”，较小的键排名更低）：

```
>>> [r.rating(key) for key in ["bob", "paul", "john", "tom"]]
 [3, 2, 1, 0]
```

`getValueByRating(ranking)`和`getKeyByRating(ranking)`对于给定的排名索引，分别返回分数和键：

```
>>> [r.getValueByRating(rating) for rating in range(4)]
 [10, 20, 20, 30]
>>> [r.getKeyByRating(rating) for rating in range(4)]
 ['tom', 'john', 'paul', 'bob']
```

一个重要的特性是`keys( )`返回的键是以排名的升序排列的，而其他所有返回的相关的列表或迭代器都遵循这个顺序：

```
>>> r.keys( )
 ['tom', 'john', 'paul', 'bob']
>>> [key for key in r]
 ['tom', 'john', 'paul', 'bob']
>>> [key for key in r.iterkeys( )]
 ['tom', 'john', 'paul', 'bob']
>>> r.values( )
 [10, 20, 20, 30]
>>> [value for value in r.itervalues( )]
 [10, 20, 20, 30]
>>> r.items( )
 [('tom', 10), ('john', 20), ('paul', 20), ('bob', 30)]
>>> [item for item in r.iteritems( )]
 [('tom', 10), ('john', 20), ('paul', 20), ('bob', 30)]
```

实例可以被修改（添加、改变和删除键-分数对应关系）而且实例的每个方法都反映了实例的当前状态：

```
>>> r["tom"] = 100
>>> r.items( )
 [('john', 20), ('paul', 20), ('bob', 30), ('tom', 100)]
```

```
        >>> del r["paul"]
        >>> r.items( )
         [('john', 20), ('bob', 30), ('tom', 100)]
        >>> r["paul"] = 25
        >>> r.items( )
         [('john', 20), ('paul', 25), ('bob', 30), ('tom', 100)]
        >>> r.clear( )
        >>> r.items( )
         [ ]
        """
    ''' 这个实现小心翼翼地混合了继承和托管，因此在尽量减少
        冗余代码的前提下获得了不错的性能，当然，同时也保
        证了语义的正确性。所有未被实现的映射方法都通过
        继承来获得，大多来自 DictMixin，但关键的__getitem__
        来自 dict。'''
    def __init__(self, *args, **kwds):
        ''' 这个类就像 dict 一样被实例化 '''
        dict.__init__(self, *args, **kwds)
        # self._rating 是关键的辅助数据结构：一个所有（值，键）
        #的列表，并保有一种“自然的”排序状态
        self._rating = [ (v, k) for k, v in dict.iteritems(self) ]
        self._rating.sort( )
    def copy(self):
        ''' 提供一个完全相同但独立的拷贝 '''
        return Ratings(self)
    def __setitem__(self, k, v):
        ''' 除了把主要任务委托给 dict，我们还维护 self._rating '''
        if k in self:
            del self._rating[self.rating(k)]
        dict.__setitem__(self, k, v)
        insort_left(self._rating, (v, k))
    def __delitem__(self, k):
        ''' 除了把主要任务委托给 dict，我们还维护 self._rating '''
        del self._rating[self.rating(k)]
        dict.__delitem__(self, k)
    ''' 显式地将某些方法委托给 dict 的对应方法，以免继承了
        DictMixin 的较慢的（虽然功能正确）实现 '''
    __len__ = dict.__len__
    __contains__ = dict.__contains__
    has_key = __contains__
    ''' 在 self._rating 和 self.keys( )之间的关键的语义联系——DictMixin
        “免费”给了我们所有其他方法，虽然我们直接实现它们能够
        获得稍好一点的性能。'''
    def __iter__(self):
        for v, k in self._rating:
            yield k
    iterkeys = __iter__
    def keys(self):
        return list(self)
```

```
        ''' 三个和排名相关的方法 '''
        def rating(self, key):
             item = self[key], key
             i = bisect_left(self._rating, item)
             if item == self._rating[i]:
                  return i
             raise LookupError, "item not found in rating"
        def getValueByRating(self, rating):
             return self._rating[rating][0]
        def getKeyByRating(self, rating):
             return self._rating[rating][1]
   def _test( ):
        ''' 我们使用 doctest 来测试这个模块，模块名必须为
             rating.py，这样 docstring 中的示例才会有效 '''
        import doctest, rating
        doctest.testmod(rating)
   if __name__ == "__main__":
        _test( )
```

讨论

在很多方面，字典都是很自然地被应用于存储键（比如，竞赛中参与者的名字）和“分数”（比如参与者获得的分数，或者参与者在拍卖中的出价）的对应关系的数据结构。如果我们希望在这些应用中使用字典，我们可能会希望以自然的顺序访问——即键对应的“分数”的升序——我们也希望能够迅速获得基于当前分数的排名（比如，参与者现在排在第三位，排在第二位的参与者的分数，等等）。

为了达到这个目的，本节给 dict 的子类增加了一些它本身完全不具备的功能（rating 方法、getValueByRating、getKeyByRating），同时，最关键和巧妙的地方是，我们修改了 keys 方法和其他相关的方法，这样它们就能返回按照指定顺序排列的列表或者可迭代对象（比如按照分数的升序排列；对于两个有同样分数的键，我们继续比较键本身）。大多数的文档都放在类的文档字符串中——保留文档和示例是很重要的，可以用 Python 标准库的 doctest 模块来提供单元测试的功能，以确保给出的例子是正确的。

关于这个实现的有趣之处是，它很关心消除冗余（即那些重复和令人厌烦的代码，很可能滋生 bug），但同时没有损害性能。Ratings 类同时从 dict 和 DictMixin 继承，并把后者排在基类列表的第一位，因此，除非明确地覆盖了基类的方法，Ratings 的方法基本来自于 DictMixin，如果它提供了的话。

Raymond Hettinger 的 DictMixin 类最初是发布在 *Python Cookbook* 在线版本中的一个例子，后来被吸收到了 Python 2.3 的标准库中。DictMixin 提供了各种映射的方法，除了 __init__、copy、以及四个基本方法：__getitem__、__setitem__、__delitem__和 keys。如果需要的是一个映射类并且想要支持完整映射所具有的各种方法，可以从 DictMixin 派生子类，并且提供那些基本的方法（具体依赖于你的类的语义——比如，如果你的

类有不可修改的实例，你无须提供属性设置方法_ _setitem_ _和_ _delitem_ _）。还可以添加一些可选的方法以提升性能，覆盖 DictMixin 所提供的原有方法。整个 DictMixin 的架构可以被看做是一个经典的模板方法设计模式（Template Method Design Pattern），它用一种混合的变体提供了广泛的适用性。

在本节的类中，从基类继承了_ _getitem_ _（准确地说，是从内建的 dict 类型继承），出于性能上的考虑，我们把能委托的都委托给了 dict。我们必须自己实现基本的属性设置方法（_ _setitem_ _和_ _delitem_ _），因为除了委托给基类的方法，还需要维护一个数据结构 self._rating——这是一个列表，包含了许多（score, key）值对，此列表在标准库模块 bisect 的帮助下完成了排序。我们也重新实现了 keys（在这个步骤中，还重新实现了_ _iter_ _，即 iterkeys，很明显，借助_ _iter_ _可以更容易地实现 keys）来利用 self._rating 并按照我们需要的顺序返回键。最后，除了上面三个和排名有关的方法，我们又为_ _init_ _和 copy 添加了实现。

这个结果是一个很有趣的例子，它取得了简洁和清晰的平衡，并最大化地重用了 Python 标准库的众多功能。如果你在应用程序中使用这个模块，测试结果可能会显示，本节的类从 DictMixin 继承来的方法的性能不是太让人满意，毕竟 DictMixin 的实现是基于必要的通用性的考虑。如果它的性能不能满足你的要求，可以自己提供一个实现来获取最高性能。假设有个 Ratings 类的实例 r，你的应用程序需要对 r.iteritems ( )的结果进行大量的循环处理，可以给类的主体部分增加这个方法的实现以获得更好的性能：

```
def iteritems(self):
    for v, k in self._rating:
          yield k, v
```

更多资料

Library Reference 和 *Python in a Nutshell* 中 UserDict 模块下的 DictMixin 类，以及 bisect 模块。

5.15 根据姓的首字母将人名排序和分组

感谢：Brett Cannon、Amos Newcombe

任务

你想将一组人名写入一个地址簿，同时还希望地址簿能够根据姓的首字母进行分组，且按照字母顺序表排序。

解决方案

Python 2.4 的新 itertools.groupby 函数使得这个任务很简单：

```
    import itertools
    def groupnames(name_iterable):
        sorted_names = sorted(name_iterable, key=_sortkeyfunc)
        name_dict = {  }
        for key, group in itertools.groupby(sorted_names, _groupkeyfunc):
            name_dict[key] = tuple(group)
        return name_dict
    pieces_order = { 2: (-1, 0), 3: (-1, 0, 1) }
    def _sortkeyfunc(name):
        ''' name 是带有名和姓以及可选的中名或首字母的字符串,
            这些部分之间用空格隔开; 返回的字符串的顺序是
            姓-名-中名, 以满足排序的需要 '''
        name_parts = name.split( )
        return ' '.join([name_parts[n] for n in
pieces_order[len(name_parts)]])
    def _groupkeyfunc(name):
        ''' 返回的键(即姓的首字母)被用于分组 '''
        return name.split( )[-1][0]
```

讨论

本节解决方案中的 name_iterable 必须是一个可迭代对象，它的元素是遵循名-中名-姓格式的人名字符串，其中中名是可选的且各部分以空格隔开。对这个可迭代对象调用 groupnames 得到的结果是一个字典，它的键是姓的首字母，而对应的值则是完整的名、中名和姓的构成的元组。

不管是“名 姓”还是“名 中名 姓”的格式，辅助的_sortkeyfunc 函数都能将人名字符串切割开，并将各部分记录到一个列表中，其顺序是先姓后名，如果有中名，还要加上中名或首字母，最后将这个列表拼接成一个字符串并返回。根据任务的描述，这个字符串是用来排序的关键。Python 2.4 的内建函数 sorted 用这个函数（它将被应用到每个元素上用于获取排序的键）作为可选的名为 key 的参数。

辅助函数_groupkeyfunc 也接受同样格式的人名，并返回姓的首字母——根据问题的描述，这是我们用来将人名分组的关键。

方案中的主函数 groupnames 使用了两个辅助函数和 Python 2.4 的 sorted 和 itertools.groupby 来解决问题，创建并返回了我们要求的字典。

如果想在 Python 2.3 中完成这个任务，仍然可以使用这两个支持函数并重新编写 groupnames。由于 Python 2.3 的标准库中并没有提供 groupby 函数，先分组再分别对各个组排序会更方便一些：

```
    def groupnames(name_iterable):
        name_dict = {  }
        for name in name_iterable:
            key = _groupkeyfunc(name)
```

```
        name_dict.setdefault(key, [ ]).append(name)
    for k, v in name_dict.iteritems( ):
        aux = [(_sortkeyfunc(name), name) for name in v]
        aux.sort( )
        name_dict[k] = tuple([ n for _ _, n in aux ])
    return name_dict
```

更多资料

19.21 节；*Library Reference*（Python 2.4）关于 itertools 的文档。