EFFECTIVE 系列丛书

华章科技

从基本原则、惯用法、语法、库、设计模式、内部机制、开发工具和性能优化8方面深入探讨编写高质量Python代码的技巧、禁忌和最佳实践

Writing Solid Python Code

91 Suggestions to Improve Your Python Program

编写高质量代码

改善Python程序的91个建议

张颖 赖勇浩 著

机械工业出版社
China Machine Press

Effective 系列丛书

编写高质量代码

改善 Python 程序的 91 个建议

张　颖　赖勇浩　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

编写高质量代码：改善 Python 程序的 91 个建议 / 张颖，赖勇浩著 . —北京：机械工业出版社，2014.6
（Effective 系列丛书）

ISBN 978-7-111-46704-5

I. 编… II. ①张… ②赖… III. 软件工具 – 程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字（2014）第 100711 号

在通往“Python 技术殿堂”的路上，本书将为你编写健壮、优雅、高质量的 Python 代码提供切实帮助！内容全部由 Python 编码的最佳实践组成，从基本原则、惯用法、语法、库、设计模式、内部机制、开发工具和性能优化 8 个方面深入探讨了编写高质量 Python 代码的技巧与禁忌，一共总结出 91 条宝贵的建议。每条建议对应 Python 程序员可能会遇到的一个问题。本书不仅以建议的方式从正反两方面给出了被实践证明为十分优秀的解决方案或非常糟糕的解决方案，而且分析了问题产生的根源，会使人有一种醍醐灌顶的感觉，豁然开朗。

本书针对每个问题所选择的应用场景都非常典型，给出的建议也都与实践紧密结合。书中的每一条建议都可能在你的下一行代码、下一个应用或下一个项目中显露锋芒。建议你将本书搁置在手边，随时查阅，相信这么做一定能使你的学习和开发工作事半功倍。

编写高质量代码：
改善 Python 程序的 91 个建议

张　颖　等著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）

责任编辑：孙海亮　　责任校对：殷　虹

印　　刷：　　版　　次：2014 年 6 月第 1 版第 1 次印刷

开　　本：186mm×240mm　1/16　　印　　张：17

书　　号：ISBN 978-7-111-46704-5　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

Preface 前　言

为什么要写这本书

当这本书的写作接近尾声的时候，回过头来看看这一年多的写作历程，不由得心生感叹，这是一个痛并快乐着的过程。不必说牺牲了多少个周末，也不必计算多少个夜晚伏案写作，单是克服写作过程中因疲劳而迸发出来的彷徨、犹豫和动摇等情绪都觉得是件不容易的事情。但不管怎么说，这最终是个沉淀和收获的过程，写作的同时我也和读者们一样在进步。为什么要写这本书？可以说是机缘巧合。机械工业出版社的杨福川老师联系到我，说他们打算策划一本关于高质量 Python 编程方面的书籍，问我有没有兴趣加入。实话实说，最开始我是持否定态度的，一则因为业余时间实在有限，无法保证我“工作和生活要平衡”的理念；二则觉得自己水平有限，在学习 Python 的道路上我和千千万万读者一样，只是一个普通的“朝圣者”，我也有迷惑不解的时候，在没有修炼到大彻大悟之前拿什么来给人传道授业？是赖勇浩老师的加入给我注入了一针强心剂，他丰富的 Python 项目经验以及长期活跃于 Python 社区所积累下来的名望无形中给了我一份信心。杨老师的鼓励和支持也更加坚定了我的态度，经过反复考虑和调整自己的心态，最终我决定和赖老师一起完成这本书。因为我也经历过从零开始的 Python 学习过程，我也遇到过各种困惑，经历过不同的曲折，这些可能也正是每一个学习 Python 的人从最初到进阶这一过程中都会遇到的问题。抱着分享自己在学习和工作中所积累的一点微薄经验的心态，我开始了本书的写作之旅。这个过程也被我当作是对自己学过的知识的一种梳理。如果与此同时，还能够给读者带来一些启示和思索，那将是这本书所能带给我的最大收获了。

读者对象

- ❑ 有一定的 Python 基础，希望通过项目最佳实践来提升自己的相关 Python 人员。
- ❑ 希望进一步掌握 Python 相关内部机制的技术人员。

- ❑ 希望写出更高质量、更 Pythonic 代码的编程人员。
- ❑ 开设相关课程的大专院校师生。

如何阅读本书

首先需要注意的是，本书并不是入门级的语法介绍类的书籍，因此在阅读本书之前假定你已经掌握了最基础的 Python 语法。如果没有，也没有关系，你可以先找一本最简单的介绍 Python 语法的书籍看看，尝试写几个 Python 小程序之后再来阅读本书。

本书分为 8 章，主要从编程惯用法、基础语法、库、设计模式、内部机制、开发工具、性能剖析与优化等方面解读如何编写高质量的 Python 程序。每个章节的内容都以建议的形式呈现，这些建议或源于实际项目应用经验，或源于对 Python 本质的理解和探讨，或源于社区推荐的做法。它们能够帮助读者快速完成从入门到进阶的这个过程。

由于各个章节相对独立，因此无须花费整段的时间从头开始阅读。你可以在空闲的时候选取任意感兴趣的小节阅读。为了减轻读者负担，本书代码尽量保持完整，阅读过程中无须额外下载其他相关代码。

勘误和支持

由于作者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你在阅读过程中遇到任何问题或者发现任何错误，欢迎发送邮件至邮箱 highqualitypython@163com，我们会尽量一一解答直到你满意。期待能够得到你的真挚反馈。

致谢

首先要感谢机械工业出版社华章公司的杨福川老师，因为有了你的鼓励才使我有勇气开始这本书。还要感谢机械工业出版社的孙海亮编辑，在这一年多的时间中始终支持我的写作，是你的鼓励和帮助引导我顺利完成全部书稿。当然也要感谢我的搭档赖老师，和你合作是一件非常愉快的事情，也让我收获颇多。

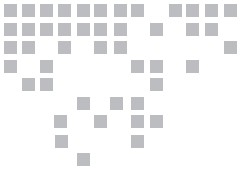
其次要感谢我的家人，是你们的宽容、支持和理解给了我完成本书的动力，也是你们无微不至的照顾让我不必为生活中的琐事烦心，从而能全身心地投入到写作中去。

最后，我想提前感谢一下本书的读者，谢谢你们能够选择阅读这本书，这将是作为作者的我们最大的荣幸。

谨以此书献给所有热爱 Python 的朋友们！

张　颖

Contents 目　录



Chapter 4 第 4 章

库

Python 具有丰富的库，掌握所有的库的用法和使用基本上是不可能的，更好的方法是掌握一些常见库的使用和注意事项，对于使用较少的库可以在具体应用时参考其文档以及使用范例。本章主要讨论一些常用库的使用和技巧。

建议 36：掌握字符串的基本用法

无名氏说：编程有两件事，一件是处理数值，另一件是处理字符串。要我说，对于商业应用编程来说，处理字符串的代码可能超过八成，所以掌握字符串的基本用法尤其重要。通过 Python 教程，读者已经掌握了基本的字符串字面量语法，比如 u、r 前缀等，但对于怎么更好地编写多行的字符串字面量，仍然有个小技巧值得向大家推介。

```
>>> s = ('SELECT * '
...      'FROM atable '
...      'WHERE afield="value"')
>>> s
'SELECT * FROM atable WHERE afield="value"'
```

这就是利用 Python 遇到未闭合的小括号时会自动将多行代码拼接为一行和把相邻的两个字符串字面量拼接在一起的特性做到的。相比使用 3 个连续的单（双）引号，这种方式不会把换行符和前导空格也当作字符串的一部分，则更加符合用户的思维习惯。

除了这个小技巧，也许你已经听说过 Python 中的字符串其实有 str 和 unicode 两种。是的，的确如此，虽然在 Python 3 中已经简化为一种，但如果你还在编写运行在 Python 2 上的程序，当需要判断变量是否为字符串时，需要注意了。判断一个变量 s 是不是字符串应使用

isinstance(s,basestring)，注意这里的参数是 basestring 而不是 str。

```
>>> a = "hi"
>>> isinstance(a,str)            ... ... ... ①
True
>>> b =u"Hi"
>>> isinstance(b,str)            ... ... ... ②
False
>>> isinstance(b,basestring)
True
>>> isinstance(b,unicode)
True
>>> isinstance(a,unicode)        ... ... ... ③
False
>>>
```

如标注①所示：isinstance(a,str) 用于判断一个字符串是不是普通字符串，也就是说其类型是否为 str ；因此当被判断的字符串为 Unicode 的时候，返回 False，如标注②所示。同样，标注③中 isinstance(a,unicode) 用来判断一个字符串是不是 Unicode。因此要正确判断一个变量是不是字符串，应该使用 isinstance(s,basestring)，因为 basestring 才是是 str 和 unicode 的基类，包含了普通字符串和 unicode 类型。

接下来正式开始学习字符串的基本用法。与其他书籍、手册不同，我们将通过性质判定、查找替换、分切与连接、变形、填空与删减等 5 个方面来学习。首先是性质判定，str 对象有以下几个方法：isalnum()、isalpha()、isdigit()、islower()、isupper()、isspace()、istitle()、startswith(prefix[, start[, end]])、endswith(suffix[,start[, end]])，前面几个 is*() 形式的函数很简单，顾名思义无非是判定是否数字、字母、大小写、空白符之类的，istitle() 作为东方人用得少些，它是判定字符串是否每个单词都有且只有第一个字母是大写的。

```
>>> assert 'Hello World!'.istitle() == True
>>> assert 'HEllo World!'.istitle() == False
```

相对于 is*() 这些“小儿科”来说，需要注意的是 *with() 函数族可以接受可选的 start、end 参数，善加利用，可以优化性能。另外，自 Python 2.5 版本起，*with() 函数族的 prefix 参数可以接受 tuple 类型的实参，当实参中的某个元素能够匹配时，即返回 True。

接下来是查找与替换，count(sub[, start[, end]])、find(sub[, start[, end]])、index(sub[, start[, end]])、rfind(sub[, start[,end]])、rindex(sub[, start[, end]]) 这些方法都接受 start、end 参数，善加利用，可以优化性能。其中 count() 能够查找子串 sub 在字符串中出现的次数，这个数值在调用 replace 方法的时候用得着。此外，需要注意 find() 和 index() 方法的不同：find() 函数族找不到时返回 –1，index() 函数族则抛出 ValueError 异常。但对于判定是否包含子串的判定并不推荐调用这些方法，而是推荐使用 in 和 not in 操作符。

```
>>> str = "Test if a string contains some special substrings"
>>> if str.find("some") != -1: # 使用 find 方法进行判断
```

```
...     print "Yes,it contains"
...
Yes,it contains
>>> if "some" in str: #使用 in 方法也可以判断
...     print "Yes,it contains using in"
...
Yes,it contains using in
```

replace(old, new[,count]) 用以替换字符串的某些子串，如果指定 count 参数的话，就最多替换 count 次，如果不指定，就全部替换（跟其他语言不太一样，要注意了）。

然后要掌握字符串的分切与连接，关于连接，会有一节专门进行讲述，在这里，专讲分切。partition(sep)、rpartition(sep)、splitlines([keepends])、split([sep [,maxsplit]])、rsplit([sep[,maxsplit]])，别看这些方法好像很多，其实只要弄清楚 partition() 和 split() 就可以了。*partition() 函数族是 2.5 版本新增的方法，它接受一个字符串参数，并返回一个 3 个元素的元组对象。如果 sep 没出现在母串中，返回值是 (sep, "","") ；否则，返回值的第一个元素是 sep 左端的部分，第二个元素是 sep 自身，第三个元素是 sep 右端的部分。而 split() 的参数 maxsplit 是分切的次数，即最大的分切次数，所以返回值最多有 maxsplit+1 个元素。但 split() 有不少小陷阱，需要注意，比如对于字符串 s、s.split() 和 s.split('') 的返回值是不相同的。

```
>>> ' hello   world!'.split()
['hello', 'world!']
>>> ' hello   world!'.split(' ')
['', '', 'hello', '', '', 'world!']
```

产生差异的原因在于：当忽略 sep 参数或 sep 参数为 None 时与明确给 sep 赋予字符串值时，split() 采用两种不同的算法。对于前者，split() 先去除字符串两端的空白符，然后以任意长度的空白符串作为界定符分切字符串（即连续的空白符串被当作单一的空白符看待）；对于后者则认为两个连续的 sep 之间存在一个空字符串。因此对于空字符串（或空白符串），它们的返回值也是不同的。

```
>>> ''.split()
[]
>>> ''.split(' ')
['']
```

掌握了 split()，可以说字符串最大的陷阱已经跨过去了。下面是关于变形的内容。lower()、upper()、capitalize()、swapcase()、title() 这些无非是大小写切换的小事，不过需要注意的是 titile() 的功能是将每一个单词的首字母大写，并将单词中的非首字母转换为小写（英文文章的标题通常是这种格式）。

```
>>> 'hello wORld!'.title()
'Hello World!'
```

因为 title() 函数并不去除字符串两端的空白符也不会把连续的空白符替换为一个空格，

所以不能把 title() 理解先以空白符分切字符串，然后调用 capitalize() 处理每个字词以使其首字母大写，再用空格将它们连接在一起。如果你有这样的需求，建议使用 string 模块中的 capwords(s) 函数，它能够去除两端的空白符，再将连续的空白符用一个空格代替。

```
>>> ' hello   world!'.title()
' Hello   World!'
>>> string.capwords(' hello   world!')
'Hello World!'
```

看，它们的结果是不相同的！最后，是删减与填充。删减在文本处理是很常用，我们常常得把字符串掐头去尾，就用得上它们。如果 strip([chars])、lstrip([chars])、rstrip([chars]) 中的 chars 参数没有指定，就是删除空白符，空白符由 string.whitespace 常量定义。填充则常用于字符串的输出，借助它们能够排出漂亮的版面。center(width[, fillchar])、ljust(width[, fillchar])、rjust(width[, fillchar])、zfill(width)、expandtabs([tabsize])，看，有了它们，居中、左对齐、右对齐什么的完全不在话下，这些方法中的 fillchar 参数是指用以填充的字符，默认是空格。而 zfill() 中的 z 是指 zero，所以顾名思义，zfill() 即是以字符 0 进行填充，在输出数值时比较常用。expandtabs() 的 tabsize 参数默认为 8，它的功能是把字符串中的制表符（tab）转换为适当数量的空格。

建议 37：按需选择 sort() 或者 sorted()

各种排序算法以及它们的时间复杂度分析是很多企业面试人员在面试时候经常会问到的问题，这也不难理解，在实际的应用过程中确实会遇到各种需要排序的情况，如按照字母表输出一个序列、对记录的多个字段排序等。还好，Python 中的排序相对简单，常用的函数有 sort() 和 sorted() 两种。这两种函数并不完全相同，各有各的用武之地。我们来具体分析一下。

1）相比于 sort()，sorted() 使用的范围更为广泛，两者的函数形式分别如下：

```
sorted(iterable[, cmp[, key[, reverse]]])
s.sort([cmp[, key[, reverse]]])
```

这两个方法有以下 3 个共同的参数：

- cmp 为用户定义的任何比较函数，函数的参数为两个可比较的元素（来自 iterable 或者 list），函数根据第一个参数与第二个参数的关系依次返回 -1、0 或者 +1（第一个参数小于第二个参数则返回负数）。该参数默认值为 None。
- key 是带一个参数的函数，用来为每个元素提取比较值，默认为 None（即直接比较每个元素）。
- reverse 表示排序结果是否反转。

```
>>> persons = [{'name': 'Jon', 'age': 32}, {'name': 'Alan', 'age': 50}, {'name':
 'Bob', 'age': 23}]
```

```
>>> sorted(persons, key=lambda x: (x['name'], -x['age']))
[{'age': 50, 'name': 'Alan'}, {'age': 23, 'name': 'Bob'}, {'age': 32, 'name': 'J
on'}]
```

从函数的定义形式可以看出，sorted() 作用于任意可迭代的对象，而 sort() 一般作用于列表。因此下面的例子中针对元组使用 sort() 方法会抛出 AttributeError，而使用 sorted() 函数则没有这个问题。

```
>>> a = (1,2,4,2,3)
>>> a.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
>>> sorted(a)
[1, 2, 2, 3, 4]
```

2）当排序对象为列表的时候两者适合的场景不同。sorted() 函数是在 Python2.4 版本中引入的，在这之前只有 sort() 函数。sorted() 函数会返回一个排序后的列表，原有列表保持不变；而 sort() 函数会直接修改原有列表，函数返回为 None。来看下面的例子：

```
>>> a=['1',1,'a',3,7,'n']
>>> sorted(a)
[1, 3, 7, '1', 'a', 'n']
>>> a
['1', 1, 'a', 3, 7, 'n']
>>> print a.sort()
None
>>> a
[1, 3, 7, '1', 'a', 'n']
>>>
```

因此如果实际应用过程中需要保留原有列表，使用 sorted() 函数较为适合，否则可以选择 sort() 函数，因为 sort() 函数不需要复制原有列表，消耗的内存较少，效率也较高。

3）无论是 sort() 还是 sorted() 函数，传入参数 key 比传入参数 cmp 效率要高。cmp 传入的函数在整个排序过程中会调用多次，函数开销较大；而 key 针对每个元素仅作一次处理，因此使用 key 比使用 cmp 效率要高。下面的测试例子显示使用 key 比 cmp 约快 50%。

```
>>> from timeit import Timer
>>> Timer(stmt="sorted(xs,key=lambda x:x[1])",setup="xs=range(100);xs=zip(xs,xs)
;").timeit(10000)
0.2900448249509081
>>>
>>> Timer(stmt="sorted(xs,cmp=lambda a,b: cmp(a[1],b[1]))",setup="xs=range(100);
xs=zip(xs,xs);").timeit(10000)
0.47374972749250155
>>>
```

4）sorted() 函数功能非常强大，使用它可以方便地针对不同的数据结构进行排序，从而

满足不同需求。来看下列例子。

- **对字典进行排序**：下面的例子中根据字典的值进行排序，即将 phonebook 对应的电话号码按照数字大小进行排序。

```
>>> phonebook = {'Linda': '7750', 'Bob': '9345', 'Carol': '5834'}
>>> from operator import itemgetter
>>> sorted_pb = sorted(phonebook.iteritems(),key=itemgetter(1))
>>> print sorted_pb
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
>>>
```

- **多维 list 排序**：实际情况下也会碰到需要对多个字段进行排序的情况，如根据学生的成绩、对应的等级依次排序。当然这在 DB 里面用 SQL 语句很容易做到，但使用多维列表联合 sorted() 函数也可以轻易达到类似的效果。

```
>>> from operator import itemgetter
>>> gameresult = [['Bob',95.00,'A'],['Alan',86.0,'C'],['Mandy',82.5,'A'],['Rob',
86,'E']] # 分别表示学生的姓名，成绩，等级
>>> sorted(gameresult , key=operator.itemgetter(2, 1))
[['Mandy', 82.5, 'A'], ['Bob', 95.0, 'A'], ['Alan', 86.0, 'C'], ['Rob', 86,
    'E'] # 当第二个字段成绩相同的时候按照等级从低到高排序
]
```

- **字典中混合 list 排序**：如果字典中的 key 或者值为列表，需要对列表中的某一个位置的元素排序也是可以做到的。下面的例子中针对字典 mydict 的 value 结构 [n,m] 中的 n 按照从小到大的顺序排列。

```
>>> mydict = { 'Li': ['M',7],
...            'Zhang': ['E',2],
...            'Wang': ['P',3],
...            'Du': ['C',2],
...            'Ma': ['C',9],
...            'Zhe': ['H',7] }
>>>
>>> from operator import itemgetter
>>> sorted(mydict.iteritems(), key=lambda (k,v): operator.itemgetter(1)(v))
[('Zhang', ['E', 2]), ('Du', ['C', 2]), ('Wang', ['P', 3]), ('Li', ['M', 7]), ('
Zhe', ['H', 7]), ('Ma', ['C', 9])]
```

- **List 中混合字典排序**：如果列表中的每一个元素为字典形式，需要针对字典的多个 key 值进行排序也不难实现。下面的例子是针对 list 中的字典元素按照 rating 和 name 进行排序的实现方法。

```
>>> gameresult = [{ "name":"Bob", "wins":10, "losses":3, "rating":75.00 },
...             { "name":"David", "wins":3, "losses":5, "rating":57.00 },
...             { "name":"Carol", "wins":4, "losses":5, "rating":57.00 },
...             { "name":"Patty", "wins":9, "losses":3, "rating": 71.48 }]
```

```
>>> from operator import itemgetter
>>> sorted(gameresult , key=operator.itemgetter("rating","name"))
[{'wins': 4, 'losses': 5, 'name': 'Carol', 'rating': 57.0}, {'wins': 3, 'losses'
: 5, 'name': 'David', 'rating': 57.0}, {'wins': 9, 'losses': 3, 'name': 'Patty',
 'rating': 71.48}, {'wins': 10, 'losses': 3, 'name': 'Bob', 'rating': 75.0}]
>>>
```

建议 38：使用 copy 模块深拷贝对象

在正式讨论本节内容之前我们先来了解一下浅拷贝和深拷贝的概念：

- 浅拷贝（shallow copy）：构造一个新的复合对象并将从原对象中发现的引用插入该对象中。浅拷贝的实现方式有多种，如工厂函数、切片操作、copy 模块中的 copy 操作等。
- 深拷贝（deep copy）：也构造一个新的复合对象，但是遇到引用会继续递归拷贝其所指向的具体内容，也就是说它会针对引用所指向的对象继续执行拷贝，因此产生的对象不受其他引用对象操作的影响。深拷贝的实现需要依赖 copy 模块的 deepcopy() 操作。

下面我们通过一段简单的程序来说明浅拷贝和深拷贝之间的区别。

```
import copy
class Pizza(object):
       def __init__(self,name,size,price):
                 self.name=name
                 self.size=size
                 self.price=price
       def getPizzaInfo(self):          ①获取 Pizza 相关信息
                 return self.name,self.size,self.price
       def showPizzaInfo(self):         ②显示 Pizza 信息
                 print "Pizza name:"+self.name
                 print "Pizza size:"+str(self.size)
                 print "Pizza price:"+str(self.price)
       def changeSize(self,size):
                 self.size=size
       def changePrice(self,price):
                 self.price=price

class Order(object):                    ③订单类
       def __init__(self,name):
                 self.customername=name
                 self.pizzaList=[]
                 self.pizzaList.append(Pizza("Mushroom",12,30))

       def ordermore(self,pizza):
                 self.pizzaList.append(pizza)
       def changeName(self,name):
                 self.customername=name

       def getorderdetail(self):
```

```
                print "customer name:"+self.customername
                for i in self.pizzaList:
                        i.showPizzaInfo()

        def getPizza(self,number):
                return self.pizzaList[number]

customer1=Order("zhang")
customer1.ordermore(Pizza("seafood",9,40))
customer1.ordermore(Pizza("fruit",12,35))
print "customer1 order infomation:"
customer1.getorderdetail()
print "--------------------------------"
```

程序描述的是客户在 Pizza 店里下了一个订单，并将具体的订单信息打印出来的场景。运行输出结果如下：

```
customer name:zhang
Pizza name:Mushroom
Pizza size:12
Pizza price:30
Pizza name:seafood
Pizza size:9
Pizza price:40
Pizza name:fruit
Pizza size:12
Pizza price:35
--------------------------------
```

假设现在客户 2 也想下一个跟客户 1 一样的订单，只是要将预定的水果披萨的尺寸和价格进行相应的修改。于是服务员拷贝了客户 1 的订单信息并做了一定的修改，代码如下：

```
customer2=copy.copy(customer1)
print "order 2 customer name:"+customer2.customername
customer2.changeName("li")
customer2.getPizza(2).changeSize(9)
customer2.getPizza(2).changePrice(30)
print "customer2 order infomation:"
customer2.getorderdetail()
print "-------------------------------------"
```

上面这段程序的输出也没有什么问题，完全满足了客户 2 的需求。输出结果如下所示：

```
order 2 customer name:zhang
customer2 order infomation:
customer name:li
Pizza name:Mushroom
Pizza size:12
Pizza price:30
Pizza name:seafood
Pizza size:9
```

```
Pizza price:40
Pizza name:fruit
Pizza size:9
Pizza price:30
-------------------------------------
```

在修改完客户 2 的订单信息之后，现在我们再来检查一下客户 1 的订单信息：

```
print "customer1 order information:"
customer1.getorderdetail()
```

你会发现客户 1 的订单内容除了客户姓名外，其他的居然和客户 2 的订单具体内容一样了。

```
-------------------------------------
customer1 order infomation:
customer name:zhang
Pizza name:Mushroom
Pizza size:12
Pizza price:30
Pizza name:seafood
Pizza size:9
Pizza price:40
Pizza name:fruit
Pizza size:9
Pizza price:30
```

这是怎么回事呢？客户 1 根本没要求修改订单的内容，这样的结果必定会直接影响到客户满意度。问题出现在哪里？这是我们本节要重点讨论的内容。我们先来分析客户 1 和客户 2 订单内容的关系图，如图 4-1 所示。

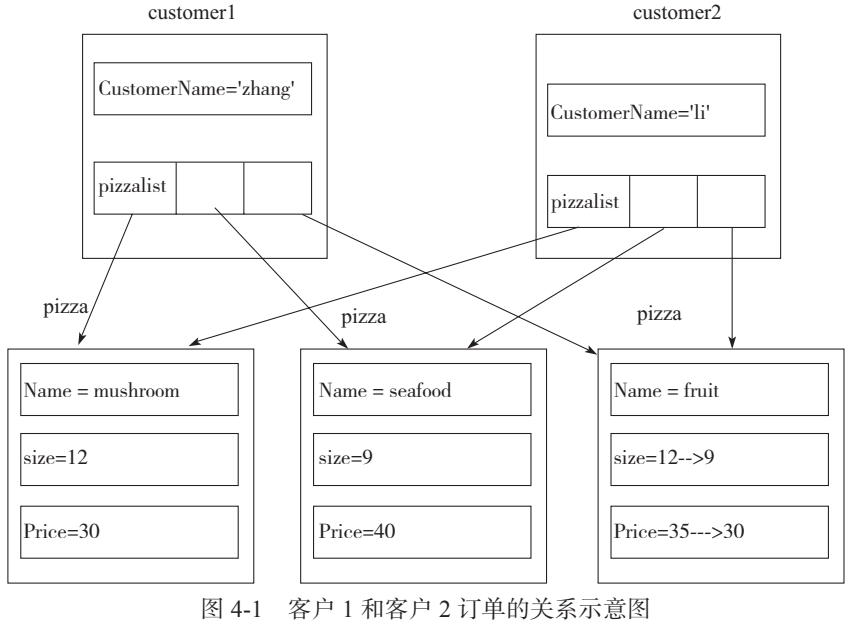


图 4-1　客户 1 和客户 2 订单的关系示意图

customer1 中的 pizzaList 是一个由 Pizza 对象组成的列表，其中存放的实际是对一个个具体 Pizza 对象的引用，在内存中就是一个具体的地址，可以通过查看 id 得到相关信息。

```
print id(customer1.pizzaList[0])    输出 14099440
print id(customer1.pizzaList[1])    输出 14101392
print id(customer1.pizzaList[2])    输出 14115344
print id(customer1.pizzaList)       输出 13914800
```

customer2 的订单通过 copy.copy(customer1) 获得，通过 id 函数查看 customer2 中 pizzaList 的具体 Pizza 对象你会发现它们和 customer1 中的输出是一样的（读者可以自行验证）。这是由于通过 copy.copy() 得到的 customer2 是 customer1 的一个浅拷贝，它仅仅拷贝了 pizzalist 里面对象的地址而不对对应地址所指向的具体内容（即具体的 pizza）进行拷贝，因此 customer2 中的 pizzaList 所指向的具体内容是和 customer1 中一样的，如图 4-1 所示。所以对 pizza fruit 的修改直接影响了 customer1 的订单内容。实际上在包含引用的数据结构中，浅拷贝并不能进行彻底的拷贝，当存在列表、字典等不可变对象的时候，它仅仅拷贝其引用地址。要解决上述问题需要用到深拷贝，深拷贝不仅拷贝引用也拷贝引用所指向的对象，因此深拷贝得到的对象和原对象是相互独立的。

上面的例子充分展示了浅拷贝和深拷贝之间的差异，在实际应用中要特别注意这两者之间的区别。实际上 Python copy 模块提供了与浅拷贝和深拷贝对应的两种方法的实现，通过名字便可以轻易进行区分，模块在拷贝出现异常的时候会抛出 copy.error。

```
copy.copy(x): Return a shallow copy of x.
copy.deepcopy(x): Return a deep copy of x.
exception copy.error: Raised for module specific errors.
```

实际上，上面的程序应该将 customer2=copy.copy(customer1) 改为 copy.deepcopy() 来实现（请读者自行验证）。关于对象拷贝读者可以查看网页 http://en.wikipedia.org/wiki/Object_copy 进行阅读扩展。

建议 39：使用 Counter 进行计数统计

计数统计相信大家都不陌生，简单地说就是统计某一项出现的次数。实际应用中很多需求都需要用到这个模型，如检测样本中某一值出现的次数、日志分析某一消息出现的频率、分析文件中相同字符串出现的概率等。这种类似的需求有很多种实现方法。我们逐一来看一下使用不同数据结构时的实现方式。

1）使用 dict。

```
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> count_frq = dict()
>>> for item in some_data:
...     if item in count_frq:
```

```
...            count_frq[item] +=1
...    else:
...            count_frq[item] = 1
...
>>> print count_frq
{'a': 3, 2: 1, 'b': 1, 4: 2, 5: 2, 7: 1, '2': 2, 'z': 1, 'd': 1}
```

2）使用 defaultdict。

```
>>> from collections import defaultdict
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> count_frq = defaultdict(int)
>>> for item in some_data:
...    count_frq[item] += 1
...
>>> print count_frq
defaultdict(<type 'int'>, {'a': 3, 2: 1, 'b': 1, 4: 2, 5: 2, 7: 1, '2': 2, 'z
    1, 'd': 1})
```

3）使用 set 和 list。

```
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> count_set = set(some_data)
>>> count_list = []
>>> for item in count_set:
...     count_list.append((item,some_data.count(item)))
...
>>> print count_list
[('a', 3), (2, 1), ('b', 1), (4, 2), (5, 2), (7, 1), ('2', 2), ('z', 1), ('d', 1)]
```

上面的方法都比较简单，但有没有更优雅、更 Pythonic 的解决方法呢？答案是使用 collections.Counter。

```
>>> from collections import Counter
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> print Counter(some_data)
Counter({'a': 3, 4: 2, 5: 2, '2': 2, 2: 1, 'b': 1, 7: 1, 'z': 1, 'd': 1})
```

Counter 类是自 Python2.7 起增加的，属于字典类的子类，是一个容器对象，主要用来统计散列对象，支持集合操作 +、–、&、|，其中 & 和 | 操作分别返回两个 Counter 对象各元素的最小值和最大值。它提供了 3 种不同的方式来初始化：

```
Counter("success")                      # 可迭代对象
Counter(s=3,c=2,e=1,u=1)                # 关键字参数
Counter({"s":3,"c":2,"u":1,"e":1})      # 字典
```

可以使用 elements() 方法来获取 Counter 中的 key 值。

```
>>> list(Counter(some_data).elements())
['a', 'a', 'a', 2, 'b', 4, 4, 5, 5, 7, '2', '2', 'z', 'd']
```

利用 most_common() 方法可以找出前 N 个出现频率最高的元素以及它们对应的次数。

```
>>> Counter(some_data).most_common(2)
[('a', 3), (4, 2)]
```

当访问不存在的元素时，默认返回为 0 而不是抛出 KeyError 异常。

```
>>> (Counter(some_data))['y']
0
```

update() 方法用于被统计对象元素的更新，原有 Counter 计数器对象与新增元素的统计计数值相加而不是直接替换它们。

subtract() 方法用于实现计数器对象中元素统计值相减，输入和输出的统计值允许为 0 或者负数。

```
>>> c = Counter("success")       #Counter({'s': 3, 'c': 2, 'e': 1, 'u': 1})
>>> c.update("successfully")     #'s': 3, 'c': 2, 'l': 2, 'u': 2, 'e': 1, 'f': 1, 'y': 1
>>> c                            #s 的值为变为 6，为上面 s 中对应值的和
Counter({'s': 6, 'c': 4, 'u': 3, 'e': 2, 'l': 2, 'f': 1, 'y': 1})
>>> c.subtract("successfully")
>>> c
Counter({'s': 3, 'c': 2, 'e': 1, 'u': 1, 'f': 0, 'l': 0, 'y': 0})
```

建议 40：深入掌握 ConfigParser

几乎所有的应用程序真正运行起来的时候，都会读取一个或几个配置文件。配置文件的意义在于用户不需要修改代码，就可以改变应用程序的行为，让它更好地为应用服务。比如 pylint 就带有一个参数 --rcfile 用以指定配置文件，实现对自定义代码风格的检测。常见的配置文件格式有 XML 和 ini 等，其中在 MS Windows 系统上，ini 文件格式用得尤其多，甚至操作系统的 API 也都提供了相关的接口函数来支持它。类似 ini 的文件格式，在 Linux 等操作系统中也是极常用的，比如 pylint 的配置文件就是这个格式。但凡这种常用的东西，Python 都有个标准库来支持它，也就是 ConfigParser。

ConfigParser 的基本用法通过手册可以掌握，但是仍然有几个知识点值得在这里跟大家说一下。首先就是 getboolean() 这个函数。getboolean() 根据一定的规则将配置项的值转换为布尔值，如以下的配置：

```
[section1]
option1=0
```

当调用 getboolean('section1', 'optioin1') 时，将返回 False。不过 getboolean() 的真值规则值得一说：除了 0 之外，no、false 和 off 都会被转义为 False，而对应的 1、yes、true 和 on 则都被转义为 True，其他值都会导致抛出 ValueError 异常。这样的设计非常贴心，使得我们能够在不同的场合使用 yes/no、true/false、on/off 等更切合自然语言语法的词汇，提升配置文

件的可维护性。

除了 getboolean() 之外，还需要注意的是配置项的查找规则。首先，在 ConfigParser 支持的配置文件格式里，有一个 [DEFAULT] 节，当读取的配置项在不在指定的节里时，ConfigParser 将会到 [DEFAULT] 节中查找。举个例子，有如下配置文件：

```
$ cat example.conf
[DEFAULT]
in_default = 'an option value in default'
 [section1]
```

简单地编写一段程序尝试通过 section1 获取 in_default 的值。

```
$ cat readini.py
import ConfigParser
conf = ConfigParser.ConfigParser()
conf.read('example.conf')
print conf.get('section1', 'in_default')
```

运行结果如下：

```
$ python readini.py
'an option value in default'
```

可见 ConfigParser 的行为跟上文描述是一致的。不过，除此之外，还有一些机制导致项目对配置项的查找更复杂，这就是 class ConfigParser 构造函数中的 defaults 形参以及其 get(section, option[, raw[, vars]]) 中的全名参数 vars。如果把这些机制全部用上，那么配置项值的查找规则如下：

1）如果找不到节名，就抛出 NoSectionError。

2）如果给定的配置项出现在 get() 方法的 vars 参数中，则返回 vars 参数中的值。

3）如果在指定的节中含有给定的配置项，则返回其值。

4）如果在 [DEFAULT] 中有指定的配置项，则返回其值。

5）如果在构造函数的 defaults 参数中有指定的配置项，则返回其值。

6）抛出 NoOptionError。

因为篇幅所限，这里就不提供示例了，大家可以自行构造相应的例子来验证。接下来要讲的是第三个特点。大家知道，在 Python 中字符串格式化可以使用以下语法：

```
>>> '%(protocol)s://%(server)s:%(port)s/' % { 'protocol':'http', 'server':'example.com',
   'port':1080}
'http://example.com:1080/'
```

其实 ConfigParser 支持类似的用法，所以在配置文件中可以使用。如，有如下配置选项：

```
$ cat format.conf
[DEFAULT]
conn_str = %(dbn)s://%(user)s:%(pw)s@%(host)s:%(port)s/%(db)s
```

```
dbn = mysql
user = root
host=localhost
port = 3306
 [db1]
user = aaa
pw=ppp
db=example
 [db2]
host=192.168.0.110
pw=www
db=example
```

这是一个很常见的 SQLAlchemy 应用程序的配置文件，通过这个配置文件能够获取不同的数据库配置相应的连接字符串，即 conn_str。如你所见，conn_str 定义在 [DEFAULT] 中，但当它通过不同的节名来获取格式化后的值时，根据不同配置，得到不同的值。先来看以下代码：

```
$ cat readformatini.py
import ConfigParser
conf = ConfigParser.ConfigParser()
conf.read('format.conf')
print conf.get('db1', 'conn_str')
print conf.get('db2', 'conn_str')
```

非常简单的代码，就是通过 ConfigParser 获取 db1 和 db2 两个节的配置。然后看如下输出：

```
$ python readformatini.py
mysql://aaa:ppp@localhost:3306/example
mysql://root:www@192.168.0.110:3306/example
```

可以看到，当通过不同的节名调用 get() 方法时，格式化 conn_str 的参数是不同的，这个规则跟上述查找配置项的规则相同。

建议 41：使用 argparse 处理命令行参数

尽管应用程序通常能够通过配置文件在不修改代码的情况下改变行为，但提供灵活易用的命令行参数依然非常有意义，比如：减轻用户的学习成本，通常命令行参数的用法只需要在应用程序名后面加 --help 参数就能获得，而配置文件的配置方法通常需要通读手册才能掌握；同一个运行环境中有多个配置文件存在，那么需要通过命令行参数指定当前使用哪一个配置文件，如 pylint 的 --rcfile 参数就是做这个事的。

为了做好命令行处理这件事，Pythonista 尝试好几个方案，标准库中留下的 getopt、optparse 和 argparse 就是证明。其中 getopt 是类似 UNIX 系统中 getopt() 这个 C 函数的实现，可以处理长短配置项和参数。如有命令行参数 -a -b -cfoo -d bar a1 a2，在处理之后的结果是

两个列表，其中一个是配置项列表 [('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]，每一个元素都由配置项名和其值（默认为空字符串）组成；另一个是参数列表 ['a1', 'a2']，每一个元素都是一个参数值。getopt 的问题在于两点，一个是长短配置项需要分开处理，二是对非法参数和必填参数的处理需要手动。如：

```
try:
    opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
except getopt.GetoptError as err:
    print str(err) # 此处输出类似 "option -a not recognized" 的出错信息
    usage()
    sys.exit(2)
output = None
verbose = False
for o, a in opts:
    if o == "-v":
        verbose = True
    elif o in ("-h", "--help"):
        usage()
        sys.exit()
    elif o in ("-o", "--output"):
        output = a
    else:
        assert False, "unhandled option"
```

从 for 循环处可以看到，这种处理非常原始和不便，而从 getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="]) 函数调用时的 "ho:v" 和 ["help", "output="] 两个实参可以看出，要编写和维护还是比较困难的，所以 optparse 就登场了。optparse 比 getopt 要更加方便、强劲，与 C 风格的 getopt 不同，它采用的是声明式风格，此外，它还能够自动生成应用程序的帮助信息。下面是一个例子：

```
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")
 (options, args) = parser.parse_args()
```

可以看到 add_option() 方法非常强大，同时支持长短配置项，还有默认值、帮助信息等，简单的几行代码，可以支持非常丰富的命令行接口。如，以下几个都是合法的应用程序调用：

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

除此之外，虽然没有声明帮助参数，但默认给加上了 -h 或 --help 支持，通过这两个参数调用应用程序，可以看到自动生成的帮助信息。

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

不过 optparse 虽然很好，但是后来出现的 argparse 在继承了它声明式风格的优点之外，又多了更丰富的功能，所以现阶段最好用的参数处理标准库是 argparse，使 optparse 成为了一个被弃用的库。

因为 argparse 自 optparse 脱胎而来，所以用法倒也大致相同，都是先生成一个 parser 实例，然后增加参数声明。如上文中 getopt 的那个例子，可以用其改造为如下形式：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-o', '--output')
parser.add_argument('-v', dest='verbose', action='store_true')
args = parser.parse_args()
```

可以看到，代码大大地减化了，代码更少，bug 更少。与 optparse 中的 add_option() 类似，add_argument() 方法用以增加一个参数声明。与 add_option() 相比，它有几个方面的改进，其中之一就是支持类型增多，而且语法更加直观。表现在 type 参数的值不再是一个字符串，而是一个可调用对象，比如在 add_option() 调用时是 type="int"，而在 add_argument() 调用时直接写 type=int 就可以了。除了支持常规的 int/float 等基本数值类型外，argparse 还支持文件类型，只要参数合法，程序就能够使用相应的文件描述符。如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```

另外，扩展类型也变得更加容易，任何可调用对象，比如函数，都可以作为 type 的实参。与 type 类似，choices 参数也支持更多的类型，而不是像 add_option 那样只有字符串。比如下面这句代码是合法的：

```
parser.add_argument('door', type=int, choices=range(1, 4))
```

此外，add_argument() 提供了对必填参数的支持，只要把 required 参数设置为 True 传递进去，当缺失这一参数时，argparse 就会自动退出程序，并提示用户。

如果仅仅是 add_argument() 比 add_option() 更加强大一点，并不足以让它把 optparse 踢出标准库，ArgumentParser 还支持参数分组。add_argument_group() 可以在输出帮助信息时更

加清晰，这在用法复杂的 CLI 应用程序中非常有帮助，比如 setuptools 配套的 setup.py 文件，如果运行 python setup.py help 可以看到它的参数是分组的。下面是一个简单的示例：

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo
group1:
  group1 description
  foo    foo help
group2:
  group2 description
  --bar BAR  bar help
```

如果仅仅是更加漂亮的帮助信息输出不够吸引你，那么 add_mutually_exclusive_group (required=False) 就非常实用：它确保组中的参数至少有一个或者只有一个（required=True）。

argparse 也支持子命令，比如 pip 就有 install/uninstall/freeze/list/show 等子命令，这些子命令又接受不同的参数，使用 ArgumentParser.add_subparsers() 就可以实现类似的功能。

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('--bar', type=int, help='bar help')
>>> parser.parse_args(['a', '--bar', '1'])
Namespace(bar=1)
```

看，就是这么简单！除了参数处理之外，当出现非法参数时，用户还需要做一些处理，处理完成后，一般是输出提示信息并退出应用程序。ArgumentParser 提供了两个方法函数，分别是 exit(status=0, message=None) 和 error(message)，可以省了 import sys 再调用 sys.exit() 的步骤。

虽然 argparse 已经非常好用，但是上进的 Pythonista 并没有止步，所以他们发明了 docopt，可以认为，它是比 argparse 更先进更易用的命令行参数处理器。它甚至不需要编写代码，只要编写类似 argparse 输出的帮助信息即可。这是因为它根据常见的帮助信息定义了一套领域特定语言（DSL），通过这个 DSL Parser 参数生成处理命令行参数的代码，从而实现对命令行参数的解释。因为 docopt 现在还不是标准库，所以在此不多介绍，有兴趣的读者可以自行去其官网（http://docopt.org/）学习。

建议 42：使用 pandas 处理大型 CSV 文件

CSV（Comma Separated Values）作为一种逗号分隔型值的纯文本格式文件，在实际应用中经常用到，如数据库数据的导入导出、数据分析中记录的存储等。因此很多语言都提供了对 CSV 文件处理的模块，Python 也不例外，其模块 csv 提供了一系列与 CSV 处理相关的 API。我们先来看一下其中几个常见的 API：

1）reader(csvfile[, dialect='excel'][, fmtparam])，主要用于 CSV 文件的读取，返回一个 reader 对象用于在 CSV 文件内容上进行行迭代。

参数 csvfile，需要是支持迭代（Iterator）的对象，通常对文件（file）对象或者列表（list）对象都是适用的，并且每次调用 next() 方法的返回值是字符串（string）；参数 dialect 的默认值为 excel，与 excel 兼容；fmtparam 是一系列参数列表，主要用于需要覆盖默认的 Dialect 设置的情形。当 dialect 设置为 excel 的时候，默认 Dialect 的值如下：

```
class excel(Dialect):
    delimiter = ','                 # 单个字符，用于分隔字段
    quotechar = '"'                 # 用于对特殊符号加引号，常见的引号为 "
    doublequote = True              # 用于控制 quotechar 符号出现的时候的表现形式
    skipinitialspace = False        # 设置为 true 的时候 delimiter 后面的空格将会忽略
    lineterminator = '\r\n'         # 行结束符
    quoting = QUOTE_MINIMAL         # 是否在字段前加引号，QUOTE_MINIMAL 表示仅当一个字段包
                                      含引号或者定义符号的时候才加引号
```

2）csv.writer(csvfile, dialect='excel', **fmtparams)，用于写入 CSV 文件。参数同上。来看一个使用例子。

```
with open('data.csv', 'wb') as csvfile:
    csvwriter = csv.writer(csvfile, dialect='excel',delimiter="|",quotechar='"',
        quoting=csv.QUOTE_MINIMAL)
    csvwriter .writerow(["1/3/09 14:44","'Product1'","1200''","Visa","Gouya"])
                                    # 写入行
输出形式为：1/3/09 14:44|'Product1'|1200''|Visa|Gouya
```

3）csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)，同 reader() 方法类似，不同的是将读入的信息映射到一个字典中去，其中字典的 key 由 fieldnames 指定，该值省略的话将使用 CSV 文件第一行的数据作为 key 值。如果读入行的字段的个数大于 filednames 中指定的个数，多余的字段名将会存放在 restkey 中，而 restval 主要用于当读取行的域的个数小于 fieldnames 的时候，它的值将会被用作剩下的 key 对应的值。

4）csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)，用于支持字典的写入。

```
import csv
#DictWriter
```

```
with open('test.csv', 'wb')  as csv_file:
# 设置列名称
        FIELDS = ['Transaction_date', 'Product', 'Price', 'Payment_Type']
        writer = csv.DictWriter(csv_file, fieldnames=FIELDS)
         # 写入列名称
        writer.writerow(dict(zip(FIELDS, FIELDS)))
        d = {'Transaction_date':'1/2/09 6:17','Product':'Product1','Price':'1200',
             'Payment_Type':'Mastercard'}
        # 写入一行  Writer.writerow(d)
with open('test.csv', 'rb') as csv_file:
        for d in csv.DictReader(csv_file):
              print d
        #output d is:{'Product': 'Product1', 'Transaction_date': '1/2/09 6:17',
            'Price': '1200', 'Payment_Type': 'Mastercard'}
```

csv 模块使用非常简单，基本可以满足大部分需求。但你有没有思考过这个问题：有些应用中需要解析和处理的 CSV 文件可能有上百 MB 甚至几个 GB，这种情况下 csv 模块是否能够应付呢？先来做个实验，临时创建一个 1GB 的 CSV 文件并将其加载到内存中，看看会有什么问题发生。

```
>>> f = open('large.csv',"wb")
>>> f.seek(1073741824-1)                          # 创建大文件的技巧
>>> f.write("\0")
>>> f.close()
>>> import os
>>> os.stat("large.csv").st_size                  # 输出文件的大小
1073741824L
>>> with open("large.csv","rb") as csvfile:
...     mycsv = csv.reader(csvfile,delimiter=';')
...     for row in mycsv:
...             print row
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
MemoryError                                       # 发生了内存异常
>>>
```

上面的例子中当企图读入这个 CSV 文件的时候抛出了 MemoryError 异常。这是为什么？因为 csv 模块对于大型 CSV 文件的处理无能为力。这种情况下就需要考虑其他解决方案了，pandas 模块便是较好的选择。

Pandas 即 Python Data Analysis Library，是为了解决数据分析而创建的第三方工具，它不仅提供了丰富的数据模型，而且支持多种文件格式处理，包括 CSV、HDF5、HTML 等，能够提供高效的大型数据处理。其支持的两种数据结构——Series 和 DataFrame——是数据处理的基础。下面先来介绍这两种数据结构。

- Series：它是一种类似数组的带索引的一维数据结构，支持的类型与 NumPy 兼容。如果不指定索引，默认为 0 到 N-1。通过 obj.values() 和 obj.index() 可以分别获取值和索

引。当给 Series 传递一个字典的时候，Series 的索引将根据字典中的键排序。如果传入字典的时候同时重新指定了 index 参数，当 index 与字典中的键不匹配的时候，会出现时数据丢失的情况，标记为 NaN。

在 pandas 中用函数 isnull() 和 notnull() 来检测数据是否丢失。

```
>>> obj1 = Series([1, 'a', (1,2), 3], index=['a', 'b', 'c', 'd'])
>>> obj1#value 和 index 一一匹配
a         1
b         a
c     (1, 2)
d         3
dtype: object
>>> obj2=Series({"Book":"Python","Author":"Dan","ISBN":"011334","Price":25},inde
x=['book','Author','ISBM','Price'])
>>> obj2.isnull()
book       True   # 指定的 index 与字典的键不匹配，发生数据丢失
Author    False
ISBM       True   # 指定的 index 与字典的键不匹配，发生数据丢失
Price     False
dtype: bool
>>>
```

- DataFrame：类似于电子表格，其数据为排好序的数据列的集合，每一列都可以是不同的数据类型，它类似于一个二维数据结构，支持行和列的索引。和 Series 一样，索引会自动分配并且能根据指定的列进行排序。使用最多的方式是通过一个长度相等的列表的字典来构建。构建一个 DataFrame 最常用的方式是用一个相等长度列表的字典或 NumPy 数组。DataFrame 也可以通过 columns 指定序列的顺序进行排序。

```
>>> data = {'OrderDate': ['1-6-10', '1-23-10', '2-9-10', '2-26-10', '3-15-10'],
...         'Region': ['East', 'Central', 'Central', 'West', 'East'],
...         'Rep': ['Jones', 'Kivell', 'Jardine', 'Gill', 'Sorvino']}
>>>
>>> DataFrame(data,columns=['OrderDate','Region','Rep'])# 通过字典构建，按照 cloumns 指定的顺序排序
  OrderDate   Region      Rep
0    1-6-10     East    Jones
1   1-23-10  Central   Kivell
2    2-9-10  Central  Jardine
3   2-26-10     West     Gill
4   3-15-10     East  Sorvino
```

Pandas 中处理 CSV 文件的函数主要为 read_csv() 和 to_csv() 这两个，其中 read_csv() 读取 CSV 文件的内容并返回 DataFrame，to_csv() 则是其逆过程。两个函数都支持多个参数，由于其参数众多且过于复杂，本节不对各个参数一一介绍，仅选取几个常见的情形结合具体例子介绍。下面举例说明，其中需要处理的 CSV 文件格式如图 4-2 所示。

```
OrderDate,Region,Rep,Item,Units,Unit Cost,Total
1-6-10,East,Jones,Pencil,95, 1.99 , 189.05
1-23-10,Central,Kivell,Binder,50, 19.99 , 999.50
2-9-10,Central,Jardine,Pencil,36, 4.99 , 179.64
2-26-10,Central,Gill,Pen,27, 19.99 , 539.73
3-15-10,West,Sorvino,Pencil,56, 2.99 , 167.44
4-1-10,East,Jones,Binder,60, 4.99 , 299.40
4-18-10,Central,Andrews,Pencil,75, 1.99 , 149.25
```

图 4-2　CSV 文件示例

1）**指定读取部分列和文件的行数**。具体的实现代码如下：

```
>>> df = pd.read_csv("SampleData.csv",nrows=5,usecols=['OrderDate','Item','Total
']])
>>> df
  OrderDate    Item   Total
0    1-6-10  Pencil  189.05
1   1-23-10  Binder  999.50
2    2-9-10  Pencil  179.64
3   2-26-10     Pen  539.73
4   3-15-10  Pencil  167.44
```

方法 read_csv() 的参数 nrows 指定读取文件的行数，usecols 指定所要读取的列的列名，如果没有列名，可直接使用索引 0、1、...、n–1。上述两个参数对大文件处理非常有用，可以避免读入整个文件而只选取所需要部分进行读取。

2）**设置 CSV 文件与 excel 兼容**。dialect 参数可以是 string 也可以是 csv.Dialect 的实例。如果将图 4-2 所示的文件格式改为使用“|”分隔符，则需要设置 dialect 相关的参数。error_bad_lines 设置为 False，当记录不符合要求的时候，如记录所包含的列数与文件列设置不相等时可以直接忽略这些列。下面的代码用于设置 CSV 文件与 excel 兼容，其中分隔符为“|”，而 error_bad_lines=False 会直接忽略不符合要求的记录。

```
>>> dia = csv.excel()
>>> dia.delimiter="|"                              # 设置分隔符
>>> pd.read_csv("SD.csv")
     OrderDate|Region|Rep|Item|Units|Unit Cost|Total
0          1-6-10|East|Jones|Pencil|95|1.99 |189.05
1  1-23-10|Central|Kivell|Binder|50|19.99 |999.50...
>>> pd.read_csv("SD.csv",dialect = dia,error_bad_lines=False)
Skipping line 3: expected 7 fields, saw 10       # 所有不符合格式要求的列将直接忽略

  OrderDate Region    Rep    Item  Units  Unit Cost   Total
0    1-6-10   East  Jones  Pencil     95       1.99  189.05
>>>
```

3）**对文件进行分块处理并返回一个可迭代的对象**。分块处理可以避免将所有的文件载入内存，仅在使用的时候读入所需内容。参数 chunksize 设置分块的文件行数，10 表示每一块包含 10 个记录。将参数 iterator 设置为 True 时，返回值为 TextFileReader，它是一个可迭代对

象。来看下面的例子，当 chunksize=10、iterator=True 时，每次输出为包含 10 个记录的块。

```
>>> reader = pd.read_table("SampleData.csv",chunksize=10,iterator=True)
>>> reader
<pandas.io.parsers.TextFileReader object at 0x0314BE70>
>>> iter(reader).next()   # 将 TextFileReader 转换为迭代器并调用 next 方法
     OrderDate,Region,Rep,Item,Units,Unit Cost,Total   # 每次读入 10 行
0        1-6-10,East,Jones,Pencil,95, 1.99 , 189.05
1  1-23-10,Central,Kivell,Binder,50, 19.99 , 999.50
2   2-9-10,Central,Jardine,Pencil,36, 4.99 , 179.64
3       2-26-10,Central,Gill,Pen,27, 19.99 , 539.73
4     3-15-10,West,Sorvino,Pencil,56, 2.99 , 167.44
5        4-1-10,East,Jones,Binder,60, 4.99 , 299.40
6  4-18-10,Central,Andrews,Pencil,75, 1.99 , 149.25
7   5-5-10,Central,Jardine,Pencil,90, 4.99 , 449.10
8     5-22-10,West,Thompson,Pencil,32, 1.99 , 63.68
9        6-8-10,East,Jones,Binder,60, 8.99 , 539.40
```

4）**当文件格式相似的时候，支持多个文件合并处理**。以下例子用于将 3 个格式相同的文件进行合并处理。

```
>>> filelst = os.listdir("test")
>>> print filelst                                    # 同时存在 3 个格式相同的文件
['s1.csv', 's2.csv', 's3.csv']
>>> os.chdir("test")
>>> dfs =[pd.read_csv(f) for f in filelst]
>>> total_df = pd.concat(dfs)                        # 将文件合并
>>> total_df
   OrderDate   Region       Rep     Item  Units  Unit Cost       Total
0     1-6-10     East     Jones   Pencil     95       1.99      189.05
1    1-23-10  Central    Kivell   Binder     50      19.99       999.5
```

了解完 pandas 后，读者可以自行实验一下使用 pandas 处理前面生成的 1GB 的文件，看看还会不会抛出 MemoryError 异常。

在处理 CSV 文件上，特别是大型 CSV 文件，pandas 不仅能够做到与 csv 模块兼容，更重要的是其 CSV 文件以 DataFrame 的格式返回，pandas 对这种数据结构提供了非常丰富的处理方法，同时 pandas 支持文件的分块和合并处理，非常灵活，由于其底层很多算法采用 Cython 实现运行速度较快。实际上 pandas 在专业的数据处理与分析领域，如金融等行业已经得到广泛的应用。

建议 43：一般情况使用 ElementTree 解析 XML

xml.dom.minidom 和 xml.sax 大概是 Python 中解析 XML 文件最广为人知的两个模块了，原因一是这两个模块自 Python 2.0 以来就成为 Python 的标准库；二是网上关于这两个模块的使用方面的资料最多。作为主要解析 XML 方法的两种实现，DOM 需要将整个 XML 文件加

载到内存中并解析为一棵树，虽然使用较为简单，但占用内存较多，性能方面不占优势，并且不够 Pythonic ；而 SAX 是基于事件驱动的，虽不需要全部装入 XML 文件，但其处理过程却较为复杂。实际上 Python 中对 XML 的处理还有更好的选择，ElementTree 便是其中一个，一般情况下使用 ElementTree 便已足够。它从 Python2.5 开始成为标准模块，cElementTree 是 ElementTree 的 Cython 实现，速度更快，消耗内存更少，性能上更占优势，在实际使用过程中应该尽量优先使用 cElementTree。由于两者使用方式上完全兼容本文将两者看做一个物件，除非说明不再刻意区分。ElementTree 在解析 XML 文件上具有以下特性：

- 使用简单。它将整个 XML 文件以树的形式展示，每一个元素的属性以字典的形式表示，非常方便处理。
- 内存上消耗明显低于 DOM 解析。由于 ElementTree 底层进行了一定的优化，并且它的 iterparse 解析工具支持 SAX 事件驱动，能够以迭代的形式返回 XML 部分数据结构，从而避免将整个 XML 文件加载到内存中，因此性能上更优化，相比于 SAX 使用起来更为简单明了。
- 支持 XPath 查询，非常方便获取任意节点的值。

这里需要说明的是，一般情况指的是：XML 文件大小适中，对性能要求并非非常严格。如果在实际过程中需要处理的 XML 文件大小在 GB 或近似 GB 级别，第三方模块 lxml 会获得较优的处理结果。关于 lxml 模块的介绍请参考本章后续小节或者参考文章“使用由 Python 编写的 lxml 实现高性能 XML 解析”，可通过链接 http://www.ibm.com/developerworks/cn/xml/x-hiperfparse/ 可以访问。

下面结合具体的实例来说明 elementtree 解析 XML 文件常用的方法。需要解析的 XML 实例如下：

```
<systems>
        <system platform="linux" name="linuxtest">
                  <purpose>automation test</purpose>
                  <system_type>virtual</system_type>
                  <ip_address/>
                  <commands_on_boot>
                            <command_details>
                                      <!-- Set root password. -->
                                      <command>echo root:mytestpwd | sudo /usr/
                                          sbin/chpasswd</command>
                                      <userid>root2</userid>
                                      <password>Passw0rd</password>
                            </command_details>
                            <command_details>
                                      <command>mkdir /TEST; chmod 777 /TEST</command>
                            </command_details>
                  </commands_on_boot>
        </system>
                  <system platform="aix" name="aixtest">
                  <purpose>manual test</purpose>
```

```
                <system_type>virtual</system_type>
                <ip_address/>
                <commands_on_boot>
                        <command_details>
                                <!-- Set root password. -->
                                <command>echo root:mytestpwd | sudo /usr/
                                    sbin/chpasswd</command>
                                <userid>root2</userid>
                                <password>Passw0rd</password>
                        </command_details>
                        <command_details>
                                <command>mkdir /TEST; chmod 777 /TEST</command>
                        </command_details>
                </commands_on_boot>
        </system>
</systems>
```

模块 ElementTree 主要存在两种类型 ElementTree 和 Element，它们支持的方法以及对应的使用示例如表 4-1 和表 4-2 所示。

表 4-1　ElementTree 主要的方法和使用示例

主要的方法、属性	方法说明以及示例

getroot()	返回 xml 文档的根节点
	>>> import xml.etree.ElementTree as ET
	>>> tree = ET.ElementTree(file ="test.xml")
	>>> root = tree.getroot()
	>>> print root
	<Element 'systems' at 0x26cbff0>
	>>> print root.tag
	systems

find(match)
findall(match)
findtext(match, default=None)
	同 Element 相关的方法类似，只是从跟节点开始搜索（见表 4-2 ）
	>>> for i in root.findall("system/purpose"):
	...　　print i.text
	...
	automation test
	manual test
	>>> print root.findtext("system/purpose")
	automation test
	>>> print root.find("system/purpose")
	<Element 'purpose' at 0x26d2170>

iter(tag=None)	从 xml 根结点开始，根据传入的元素的 tag 返回所有的元素集合的迭代器 >>> for i in tree.iter(tag = "command"): ...　　print i.text ... echo root:mytestpwd | sudo /usr/sbin/chpasswd mkdir /TEST; chmod 777 /TEST echo root:mytestpwd | sudo /usr/sbin/chpasswd mkdir /TEST; chmod 777 /TEST

(续)

主要的方法、属性	方法说明以及示例
iterfind(match)	根据传入的 tag 名称或者 path 以迭代器的形式返回所有的子元素 >>> for i in tree.iterfind("system/purpose"): ... print i.text ... automation test manual test

表 4-2　Element 主要的方法和使用示例

主要的方法、属性	方法说明以及示例
tag	字符串，用来表示元素所代表的名称 >>> print root[1].tag # 输出 system
text	表示元素所对应的具体值 >>> print root[1].text # 输出空串
attrib	用字典表示的元素的属性 >>> print root[1].attrib # 输出 {'platform': 'aix', 'name': 'aixtest'}
get(key, default=None)	根据元素属性字典的 key 值获取对应的值，如果找不到对应的属性，则返回 default >>> print root[1].attrib.get("platform") # 输出 aix
items()	将元素属性以 (名称, 值) 的形式返回 >>> print root[1].items() #[('platform', 'aix'), ('name', 'aixtest')]
keys()	返回元素属性的 key 值集合 >>> print root[1].keys() # 输出 ['platform', 'name']
find(match)	根据传入的 tag 名称或者 path 返回第一个对应的 element 对象，或者返回 None
findall(match)	根据传入的 tag 名称或者 path 以列表的形式返回所有符合条件的元素
findtext(match, default=None)	根据传入的 tag 名称或者 path 返回第一个对应的 element 对象对应的值，即 text 属性，如果找不到则返回 default 的设置
list(elem)	根据传入的元素的名称返回其所有的子节点 >>> for i in list(root.findall("system/system_type")): ... print i.text ... # 输出 virtual virtural

前面我们提到 elementree 的 iterparse 工具能够避免将整个 XML 文件加载到内存，从而解决当读入文件过大内存而消耗过多的问题。iterparse 返回一个可以迭代的由元组 (时间 , 元素) 组成的流对象，支持两个参数——source 和 events，其中 event 有 4 种选择——start、end、startns 和 endns（默认为 end），分别与 SAX 解析的 startElement、endElement、startElementNS 和 endElementNS 一一对应。

本节最后来看一下 iterparse 的使用示例：统计 userid 在整个 XML 出现的次数。

```
>>> count = 0
>>> for event,elem in ET.iterparse("test.xml"):# 对 iterparse 的返回值进行迭代
...     if event =='end':
...             if elem.tag =='userid':
...                     count+=1
...     elem.clear()
...
>>> print count
2
```

建议 44：理解模块 pickle 优劣

在实际应用中，序列化的场景很常见，如：在磁盘上保存当前程序的状态数据以便重启的时候能够重新加载；多用户或者分布式系统中数据结构的网络传输时，可以将数据序列化后发送给一个可信网络对端，接收者进行反序列化后便可以重新恢复相同的对象；session 和 cache 的存储等。序列化，简单地说就是把内存中的数据结构在不丢失其身份和类型信息的情况下转成对象的文本或二进制表示的过程。对象序列化后的形式经过反序列化过程应该能恢复为原有对象。Python 中有很多支持序列化的模块，如 pickle、json、marshal 和 shelve 等。最广为人知的为 pickle，我们来仔细分析一下这个模块。

pickle 估计是最通用的序列化模块了，它还有个 C 语言的实现 cPickle，相比 pickle 来说具有较好的性能，其速度大概是 pickle 的 1000 倍，因此在大多数应用程序中应该优先使用 cPickle（注：cPickle 除了不能被继承之外，它们两者的使用基本上区别不大，除有特殊情况，本节将不再做具体区分）。pickle 中最主要的两个函数对为 dump() 和 load()，分别用来进行对象的序列化和反序列化。

- pickle.dump(obj, file[, protocol])：序列化数据到一个文件描述符（一个打开的文件、套接字等）。参数 obj 表示需要序列化的对象，包括布尔、数字、字符串、字节数组、None、列表、元组、字典和集合等基本数据类型，此外 picike 还能够处理循环，递归引用对象、类、函数以及类的实例等。参数 file 支持 write() 方法的文件句柄，可以为真实的文件，也可以是 StringIO 对象等。protocol 为序列化使用的协议版本，0 表示 ASCII 协议，所序列化的对象使用可打印的 ASCII 码表示；1 表示老式的二进制协议；2 表示 2.3 版本引入的新二进制协议，比以前的更高效。其中协议 0 和 1 兼容老版本的 Python。protocol 默认值为 0。
- load(file)：表示把文件中的对象恢复为原来的对象，这个过程也被称为反序列化。

来看一下 load() 和 dump() 的示例。

```
>>> import cPickle as pickle
>>> my_data = {"name" : "Python", "type" : "Language", "version" : "2.7.5"}
>>> fp = open("picklefile.dat", "wb")     # 打开要写入的文件
>>> pickle.dump(my_data, fp)              # 使用 dump 进行序列化
>>> fp.close()
```

```
>>>
>>> fp = open("picklefile.dat", "rb")
>>> out = pickle.load(fp)                # 反序列化
>>> print(out)
{'version': '2.7.5', 'type': 'Language', 'name': 'Python'}
>>> fp.close()
```

pickle 之所以能成为通用的序列化模块，与其良好的特性是分不开的，总结为以下几点：

1）接口简单，容易使用。通过 dump() 和 load() 便可轻易实现序列化和反序列化。

2）pickle 的存储格式具有通用性，能够被不同平台的 Python 解析器共享，比如，Linux 下序列化的格式文件可以在 Windows 平台的 Python 解析器上进行反序列化，兼容性较好。

3）支持的数据类型广泛。如数字、布尔值、字符串，只包含可序列化对象的元组、字典、列表等，非嵌套的函数、类以及通过类的 __dict__ 或者 __getstate__() 可以返回序列化对象的实例等。

4）pickle 模块是可以扩展的。对于实例对象，pickle 在还原对象的时候一般是不调用 __init__() 函数的，如果要调用 __init__() 进行初始化，对于古典类可以在类定义中提供 __getinitargs__() 函数，并返回一个元组，当进行 unpickle 的时候，Python 就会自动调用 __init__()，并把 __getinitargs__() 中返回的元组作为参数传递给 __init__()，而对于新式类，可以提供 __getnewargs__() 来提供对象生成时候的参数，在 unpickle 的时候以 Class.__new__(Class, *arg) 的方式创建对象。对于不可序列化的对象，如 sockets、文件句柄、数据库连接等，也可以通过实现 pickle 协议来解决这些局限，主要是通过特殊方法 __getstate__() 和 __setstate__() 来返回实例在被 pickle 时的状态。来看以下示例：

```
import cPickle as pickle
class TextReader:
    def __init__(self, filename):
        self.filename = filename          # 文件名称
        self.file = open(filename)        # 打开文件的句柄
        self.postion = self.file.tell()   # 文件的位置
    def readline(self):
        line = self.file.readline()
        self.postion = self.file.tell()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.postion, line)
    def __getstate__(self):               # 记录文件被 pickle 时候的状态
        state = self.__dict__.copy()      # 获取被 pickle 时的字典信息
        del state['file']
        return state
    def __setstate__(self, state):        # 设置反序列化后的状态
         self.__dict__.update(state)
        file = open(self.filename)
        self.file = file
```

```
reader = TextReader("zen.txt")
print(reader.readline())
print(reader.readline())
s = pickle.dumps(reader)                    # 在 dumps 的时候会默认调用 __getstate__
new_reader = pickle.loads(s)                # 在 loads 的时候会默认调用 __setstate__
print(new_reader.readline())
```

5）能够自动维护对象间的引用，如果一个对象上存在多个引用，pickle 后不会改变对象间的引用，并且能够自动处理循环和递归引用。

```
>>> a = ['a','b']
>>> b = a                                   #b 引用对象 a
>>> b.append('c')
>>> p = pickle.dumps((a,b))
>>> a1,b1 = pickle.loads(p)
>>> a1
['a', 'b', 'c']
>>> b1
['a', 'b', 'c']
>>> a1.append('d')                          # 反序列化对 a1 对象的修改仍然会影响到 b1
>>> b1
['a', 'b', 'c', 'd']
```

但 pickle 使用也存在以下一些限制：

- **pickle 不能保证操作的原子性**。pickle 并不是原子操作，也就是说在一个 pickle 调用中如果发生异常，可能部分数据已经被保存，另外如果对象处于深递归状态，那么可能超出 Python 的最大递归深度。递归深度可以通过 sys.setrecursionlimit() 进行扩展。
- **pickle 存在安全性问题**。Python 的文档清晰地表明它不提供安全性保证，因此对于一个从不可信的数据源接收的数据不要轻易进行反序列化。由于 loads() 可以接收字符串作为参数，这意味着精心设计的字符串给入侵提供了一种可能。在 Pthon 解释器中输入代码 pickle.loads("cos\nsystem\n(S'dir'\ntR.") 便可查看当前目录下所有文件。如果将 dir 替换为其他更具有破坏性的命令将会带来安全隐患。如果要进一步提高安全性，用户可以通过继承类 pickle.Unpickler 并重写 find_class() 方法来实现。
- **pickle 协议是 Python 特定的，不同语言之间的兼容性难以保障**。用 Python 创建的 pickle 文件可能其他语言不能使用，如 Perl、PHP、Java 等。

建议 45：序列化的另一个不错的选择——JSON

JSON（JavaScript Object Notation）是一种轻量级数据交换格式，它基于 JavaScript 编程语言的一个子集，于 1999 年 12 月成为一个完全独立于语言的文本格式。由于其格式使用了其他许多流行编程的约定，如 C、C++、C # 、Java、JS、Python 等，加之其简单灵活、可

读性和互操作性较强、易于解析和使用等特点，逐渐变得流行起来，甚至有代替 XML 的趋势。关于 JSON 和 XML 之间的优劣，一直有很多争论，本书并不打算对这两者之间的是是非非做详尽的分析（笔者的观点是两者各有所长，在相当长的时间里还会共存共荣），这里关注的是 JSON 用于序列化方面的优势。在进行详细讨论之前，我们先来看看 Python 语言中对 JSON 的支持现状。

Python 中有一系列的模块提供对 JSON 格式的支持，如 simplejson、cjson、yajl、ujson，自 Python2.6 后又引入了标准库 JSON。简单来说 cjson 和 ujson 是用 C 来实现的，速度较快。据 cjson 的文档表述：其速率比纯 Python 实现的 json 模块大概要快 250 倍。yajl 是 Cpython 版本的 JSON 实现，而 simplejson 和标准库 JSON 本质来说无多大区别，实际上 Python2.6 中的 json 模块就是 simplejson 减去对 Python2.4、2.5 的支持以充分利用最新的兼容未来的功能。不过相对于 simplejson，标准库更新相对较慢，Python2.7.5 中 simplejson 对应的版本为 2.0.9，而最新的 simplejson 的版本为 3.3.0。在实际应用过程中将这两者结合较好的做法是采用如下 import 方法。

```
try: import simplejson as json
except ImportError: import json
```

本节仍采用标准库 JSON 来做一些探讨。Python 的标准库 JSON 提供的最常用的方法与 pickle 类似，dump/dumps 用来序列化，load/loads 用来反序列化。需要注意的 json 默认不支持非 ASCII-based 的编码，如 load 方法可能在处理中文字符时不能正常显示，则需要通过 encoding 参数指定对应的字符编码。在序列化方面，相比 pickle，JSON 具有以下优势：

1）使用简单，支持多种数据类型。JSON 文档的构成非常简单，仅存在以下两大数据结构。

- 名称 / 值对的集合。在各种语言中，它被实现为一个对象、记录、结构、字典、散列表、键列表或关联数组。
- 值的有序列表。在大多数语言中，它被实现为数组、向量、列表或序列。在 Python 中对应支持的数据类型包括字典、列表、字符串、整数、浮点数、True、False、None 等。JSON 中数据结构和 Python 中的转换并不是完全一一对应，存在一定的差异，读者可以自行查阅文档。Python 中一个 JSON 文档可以分解为如图 4-3 所示形式。

2）存储格式可读性更为友好，容易修改。相比于 pickle 来说，json 的格式更加接近程序员的思维，修改和阅读上要容易得多。dumps() 函数提供了一个参数 indent 使生成的 json 文件可读性更好，0 意味着“每个值单独一行”；大于 0 的数字意味着“每个值单独一行并且使用这个数字的空格来缩进嵌套的数据结构”。但需要注意的是，这个参数是以文件大小变大为代价的。如图 4-4 展示的是这两种格式之间的对比，其中 json.dumps() 使用了 indent 参数输出。

3）json 支持跨平台跨语言操作，能够轻易被其他语言解析，如 Python 中生成的 json 文件可以轻易使用 JavaScript 解析，互操作性更强，而 pickle 格式的文件只能在 Python 语言中支持。此外 json 原生的 JavaScript 支持，客户端浏览器不需要为此使用额外的解释器，特别适用于 Web 应用提供快速、紧凑、方便的序列化操作。此外，相比于 pickle，json 的存储格式更为紧凑，所占空间更小。

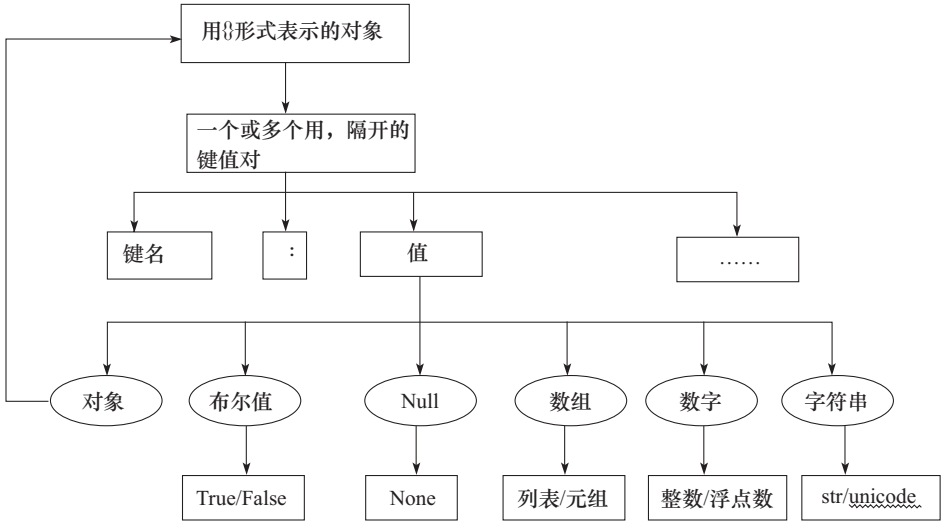


图 4-3　json 文档分解图

```
S'orderID'
p2
I12345
sS'shopperName'
p3
S'Tom'
p4
sS'orderCompleted'
p5
I01
sS'contents'
p6
(lp7
(dp8
S'productName'
p9
S'Soap'
p10
sS'quantity'
... ... ...
```

pickle格式

```
{
  "orderID": 12345,
  "shopperName": "Tom",
  "orderCompleted": true,
  "contents": [
    {
      "productName": "Soap",
      "quantity": 2,
      "productID": 34
    },
    {
      "productName": "SHOES",
      "quantity": 1,
      "productID": 56
    }
  ],
  "shopperEmail": "tom@gmail.com"
}
```

json格式

图 4-4　pickle 和 json 文件格式对比

4）具有较强的扩展性。json 模块还提供了编码（JSONEncoder）和解码类（JSONDecoder）以便用户对其默认不支持的序列化类型进行扩展。来看一个例子：

```
>>> d=datetime.datetime.now()
>>> d
datetime.datetime(2013, 9, 15, 8, 54, 59, 851000)
>>> json.dumps(d)
... ... ...
```

```
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: datetime.datetime(2013, 9, 15, 8, 54, 59, 851000) is not JSON seriali
zable
```

5）json 在序列化 datetime 的时候会抛出 TypeError 异常，这是因为 json 模块本身不支持 datetime 的序列化，因此需要对 json 本身的 JSONEncoder 进行扩展。有多种方法可以实现，下面的例子是其中实现之一。

```
import datetime
from time import mktime
try: import simplejson as json
except ImportError: import json

class DateTimeEncoder(json.JSONEncoder):        # 对 JSONEncoder 进行扩展
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
           return obj.strftime('%Y-%m-%d %H:%M:%S')
        elif isinstance(obj, date):
           return obj.strftime('%Y-%m-%d')
        return json.JSONEncoder.default(self, obj)

d=datetime.datetime.now()
print json.dumps(d, cls = DateTimeEncoder)      # 使用 cls 指定编码器的名称
```

最后需要提醒的是，Python 中标准模块 json 的性能比 pickle 与 cPickle 稍逊。如果对序列化性能要求非常高的场景，可以选择 cPickle 模块。图 4-5 显示的是这三者序列化时随着数据规模增加所消耗时间改变的图例。

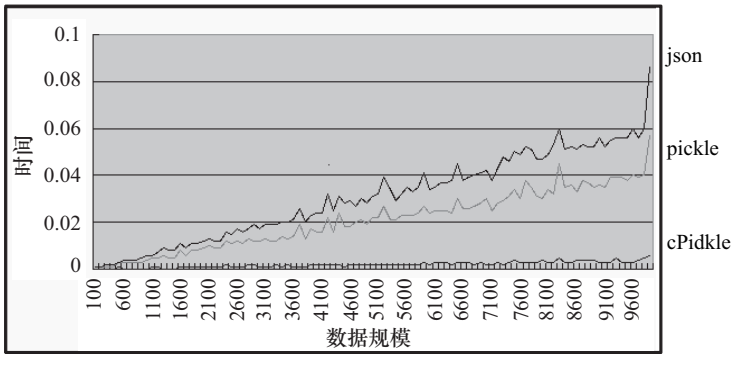


图 4-5　pickle、json、cPickle 序列化文件时性能比较

建议 46：使用 traceback 获取栈信息

当程序产生异常的时候，最需要面对异常的其实是开发人员，他们需要更多的异常提示的信息，以便调试程序中潜在的错误和问题。先来看一个简单的例子：

```
gList = ['a','b','c','d','e','f','g']
def f():
    gList[5]
    return g()
def g():
    return h()
def h():
    del gList[2]
    return i()
def i():
    gList.append('i')
    print gList[7]
if __name__ == '__main__':
    try:
        f()
    except IndexError as ex:
        print "Sorry,Exception occured,you accessed an element out of range"
        print ex
```

上述程序运行输出如下：

```
Sorry,Exception occured,you accessed an element out of range
list index out of range
```

信息提示有异常产生，对于用户这点还算是较为友好的，那么对于开发人员，他如何快速地知道错误发生在哪里呢？逐行检查代码吗？对于简单的程序，这也不失为一个办法，但在较为复杂的应用程序中这个方法就显得有点低效了。面对异常开发人员最希望看到的往往是异常发生时候的现场信息，traceback 模块可以满足这个需求，它会输出完整的栈信息。将上述代码异常部分修改如下：

```
except IndexError as ex:
    print "Sorry,Exception occured,you accessed an element out of range"
    print ex
    traceback.print_exc()
```

程序运行会输出异常发生时候完整的栈信息，包括调用顺序、异常发生的语句、错误类型等。

```
Sorry,Exception occured,you accessed an element out of rangelist index out of range
Traceback (most recent call last):
  File "trace.py", line 20, in <module>
    f()
  File "trace.py", line 5, in f
    return g()
  File "trace.py", line 8, in g
    return h()
  File "trace.py", line 12, in h
    return i()
  File "trace.py", line 16, in i
```

```
    print gList[7]
IndexError: list index out of range
```

traceback.print_exc() 方法打印出的信息包括 3 部分：错误类型（IndexError）、错误对应的值（list index out of range）以及具体的 trace 信息，包括文件名、具体的行号、函数名以及对应的源代码。Traceback 模块提供了一系列方法来获取和显示异常发生时候的 trace 相关信息，下面列举几个常用的方法：

1）traceback.print_exception(type, value, traceback[, limit[, file]])，根据 limit 的设置打印栈信息，file 为 None 的情况下定位到 sys.stderr，否则则写入文件；其中 type、value、traceback 这 3 个参数对应的值可以从 sys.exc_info() 中获取。

2）raceback.print_exc([limit[, file]])，为 print_exception() 函数的缩写，不需要传入 type、value、traceback 这 3 个参数。

3）traceback.format_exc([limit])，与 print_exc() 类似，区别在于返回形式为字符串。

4）traceback.extract_stack([file,[, limit]])，从当前栈帧中提取 trace 信息。

读者可以参看 Python 文档获取更多关于 traceback 所提供的抽取、格式化或者打印程序运行时候的栈跟踪信息的方法。本质上模块 trackback 获取异常相关的数据都是通过 sys.exc_info() 函数得到的。当有异常发生的时候，该函数以元组的形式返回 (type, value, traceback)，其中 type 为异常的类型，value 为异常本身，traceback 为异常发生时候的调用和堆栈信息，它是一个 traceback 对象，对象中包含出错的行数、位置等数据。上面的例子中也可以通过如下方式输出异常发生时候的详细信息：

```
tb_type, tb_val, exc_tb = sys.exc_info()
for filename, linenum, funcname, source in traceback.extract_tb(exc_tb):
    print "%-23s:%s '%s' in %s()" % (filename, linenum, source, funcname)
```

实际上除了 traceback 模块本身，inspect 模块也提供了获取 traceback 对象的接口，inspect.trace([context]) 可以返回当前帧对象以及异常发生时进行捕获的帧对象之间的所有栈帧记录列表，因此第一个记录代表当前调用对象，最后一个代表异常发生时候的对象。其中每一个列表元素都是一个由 6 个元素组成的元组：（frame 对象，文件名，当前行号，函数名，源代码列表，当前行在源代码列表中的位置）。本节开头的例子在异常部分使用 inspect.trace() 来获取异常发生时候的堆栈信息，其部分输出结果如下：

```
[(<frame object at 0x022CB480>,
  'testinspect.py',
  23,
  '<module>',
  ['\t    f()\n'],
  0),
... ...]
```

此外如果想进一步追踪函数调用的情况，还可以通过 inspect 模块的 inspect.stack() 函数

查看函数层级调用的栈相信信息。因此，当异常发生的时候，合理使用上述模块中的方法可以快速地定位程序中的问题所在。

建议 47：使用 logging 记录日志信息

仅仅将栈信息输出到控制台是远远不够的，更为常见的是使用日志保存程序运行过程中的相关信息，如运行时间、描述信息以及错误或者异常发生时候的特定上下文信息。Python 中自带的 logging 模块提供了日志功能，它将 logger 的 level 分为 5 个级别（如表 4-3 所示），可以通过 Logger.setLevel(lvl) 来设置，其中 DEBUG 为最低级别，CRITICAL 为最高级别，默认的级别为 WARNING。

表 4-3　日志级别

Level	使用情形
DEBUG	详细的信息，在追踪问题的时候使用
INFO	正常的信息
WARNING	一些不可预见的问题发生，或者将要发生，如磁盘空间低等，但不影响程序的运行
ERROR	由于某些严重的问题，程序中的一些功能受到影响
CRITICAL	严重的错误，或者程序本身不能够继续运行

logging lib 包含以下 4 个主要对象：

1）logger。logger 是程序信息输出的接口，它分散在不同的代码中，使得程序可以在运行的时候记录相应的信息，并根据设置的日志级别或 filter 来决定哪些信息需要输出，并将这些信息分发到其关联的 handler。常用的方法有 Logger.setLevel()、Logger.addHandler()、Logger.removeHandler()、Logger.addFilter()、Logger.debug()、Logger.info()、Logger.warning()、Logger.error()、etLogger() 等。

2）Handler。Handler 用来处理信息的输出，可以将信息输出到控制台、文件或者网络。可以通过 Logger.addHandler() 来给 logger 对象添加 handler，常用的 handler 有 StreamHandler 和 FileHandler 类。StreamHandler 发送错误信息到流，而 FileHandler 类用于向文件输出日志信息，这两个 handler 定义在 logging 的核心模块中。其他的 handler 定义在 logging.handles 模块中，如 HTTPHandler、SocketHandler。

3）Formatter。决定 log 信息的格式，格式使用类似于 %(< dictionary key >)s 的形式来定义，如 '%(asctime)s - %(levelname)s - %(message)s'，支持的 key 可以在 Python 自带的文档 LogRecord attributes 中查看。

4）Filter。用来决定哪些信息需要输出。可以被 handler 和 logger 使用，支持层次关系，比如，如果设置了 filter 名称为 A.B 的 logger，则该 logger 和其子 logger 的信息会被输出，如 A.B、A.B.C.

logging.basicConfig([**kwargs]) 提供对日志系统的基本配置，默认使用 StreamHandler 和 Formatter 并添加到 root logger，该方法自 Python2.4 开始可以接受字典参数，支持的字典参数如表 4-4 所示。

表 4-4　字典参数格式类型

格　　式	描　　述
filename	指定 FileHandler 的文件名，而不是默认的 StreamHandler
filemode	打开文件的模式，同 open 函数中的同名参数，默认为’a’
format	输出格式字符串
datefmt	日期格式
level	设置根 logger 的日志级别
stream	指定 StreamHandler。这个参数若与 filename 冲突，忽略 stream

我们通过修改上一节的例子来看如何结合 traceback 和 logging，记录程序运行过程中的异常。

```
import traceback
import sys
import logging
gList = ['a','b','c','d','e','f','g']
logging.basicConfig( # 配置日志的输出方式及格式
  level=logging.DEBUG,
  filename='log.txt',
  filemode='w',
  format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',
)
def f():
    gList[5]
    logging.info('[INFO]:calling method g() in f()')# 记录正常的信息
    return g()
def g():
    logging.info('[INFO]:calling method h() in g()')
    return h()
def h():
    logging.info('[INFO]:Delete element in gList in h()')
    del gList[2]
    logging.info('[INFO]:calling method i() in h()')
    return i()
def i():
    logging.info('[INFO]:Append element i to gList in i()')
    gList.append('i')
    print gList[7]
if __name__ == '__main__':
    logging.debug('Information during calling f():')
    try:
        f()
    except IndexError as ex:
```

```
print "Sorry,Exception occured,you accessed an element out of range"
#traceback.print_exc()
ty,tv,tb = sys.exc_info()
logging.error("[ERROR]:Sorry,Exception occured,you accessed an
    element out of range")# 记录异常错误信息
logging.critical('object info:%s' %ex)
logging.critical('Error Type:{0},Error Information:{1}'.format(ty, tv))
    # 记录异常的类型和对应的值
logging.critical(''.join(traceback.format_tb(tb)))# 记录具体的 trace 信息
sys.exit(1)
```

修改程序后在控制台上对用户仅显示错误提示信息“ Sorry,Exception occured,you accessed an element out of range”，而开发人员如果需要 debug 可以在日志文件中找到具体运行过程中的信息。

```
# 为了节省篇幅仅显示部分日志
2013-06-26 12:05:18,923 traceexample.py[line:41] CRITICAL object info:list
    index out of range
2013-06-26 12:05:18,923 traceexample.py[line:42] CRITICAL Error Type:<type
    'exceptions.IndexError'>,Error Information:list index out of range
2013-06-26 12:05:18,924 traceexample.py[line:43] CRITICAL   File "traceexample.py",
    line 35, in <module>
    f()
  File "traceexample.py", line 15, in f
    return g()
  File "traceexample.py", line 19, in g
    return h()
  File "traceexample.py", line 25, in h
    return i()
  File "traceexample.py", line 30, in i
    print gList[7]
```

上面的代码中控制运行输出到 console 上用的是 print()，但这种方法比较原始，logging 模块提供了能够同时控制输出到 console 和文件的方法。下面的例子中通过添加 StreamHandler 并设置日志级别为 logging.ERROR，可以在控制台上输出错误信息。

```
console = logging.StreamHandler()
console.setLevel(logging.ERROR)
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
console.setFormatter(formatter)
logging.getLogger('').addHandler(console)
```

为了使 Logging 使用更为简单可控，logging 支持 loggin.config 进行配置，支持 dictConfig 和 fileConfig 两种形式，其中 fileConfig 是基于 configparser() 函数进行解析，必须包含的内容为 [loggers]、[handlers] 和 [formatters]。具体例子示意如下：

```
[loggers]
keys=root
```

```
[logger_root]
level=DEBUG
handlers=hand01
[handlers]
keys=hand01

[handler_hand01]
class=StreamHandler
level=INFO
formatter=form01
args=(sys.stderr,)
[formatters]
keys=form01
[formatter_form01]
format=%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s
datefmt=%a, %d %b %Y %H:%M:%S
```

最后关于 logging 的使用，提以下几点建议：

1）尽量为 logging 取一个名字而不是采用默认，这样当在不同的模块中使用的时候，其他模块只需要使用以下代码就可以方便地使用同一个 logger，因为它本质上符合单例模式。

```
import logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger( __name__ )
```

2）为了方便地找出问题所在，logging 的名字建议以模块或者 class 来命名。Logging 名称遵循按“.”划分的继承规则，根是 root logger，logger a.b 的父 logger 对象为 a。

3）Logging 只是线程安全的，不支持多进程写入同一个日子文件，因此对于多个进程，需要配置不同的日志文件。

建议 48：使用 threading 模块编写多线程程序

GIL 的存在使得 Python 多线程编程暂时无法充分利用多处理器的优势，这种限制也许使很多人感到沮丧，但事实上这并不意味着我们需要放弃多线程。的确，对于只含纯 Python 的代码也许使用多线程并不能提高运行速率，但在以下几种情况，如等待外部资源返回，或者为了提高用户体验而建立反应灵活的用户界面，或者多用户应用程序中，多线程仍然是一个比较好的解决方案。Python 为多线程编程提供了两个非常简单明了的模块：thread 和 threading。那么，这两个模块在多线程处理上有什么区别呢？简单一点说：thread 模块提供了多线程底层支持模块，以低级原始的方式来处理和控制线程，使用起来较为复杂；而 threading 模块基于 thread 进行包装，将线程的操作对象化，在语言层面提供了丰富的特性。Python 多线程支持用两种方式来创建线程：一种是通过继承 Thread 类，重写它的 run() 方法（注意，不是 start() 方法）；另一种是创建一个 threading.Thread 对象，在它的初始化函数（__init__()）中将可调用对象作为参数传入。实际应用中，推荐优先使用 threading 模块而不

是 thread 模块，（除非有特殊需要）。下面来具体分析一下这么做的原因。

1）threading 模块对同步原语的支持更为完善和丰富。就线程的同步和互斥来说，thread 模块只提供了一种锁类型 thread.LockType，而 threading 模块中不仅有 Lock 指令锁、RLock 可重入指令锁，还支持条件变量 Condition、信号量 Semaphore、BoundedSemaphore 以及 Event 事件等。

2）threading 模块在主线程和子线程交互上更为友好，threading 中的 join() 方法能够阻塞当前上下文环境的线程，直到调用此方法的线程终止或到达指定的 timeout（可选参数）。利用该方法可以方便地控制主线程和子线程以及子线程之间的执行。来看一个简单示例：

```
import threading, time,sys
class test(threading.Thread):

    def __init__(self,name,delay):
        threading.Thread.__init__(self)
        self.name = name
        self.delay = delay

    def run(self):
        print "%s delay for %s" %(self.name,self.delay)
        time.sleep(self.delay)
        c = 0
        while True:
            print "This is thread %s on line %s" %(self.name,c)
            c = c + 1
            if c == 3:
                print "End of thread %s" % self.name
                break

t1 = test('Thread 1', 2)
t2 = test('Thread 2', 2)
t1.start()
print "Wait t1 to end"
t1.join()
t2.start()
print 'End of main'
```

上面的例子中，主线程 main 在 t1 上使用 join() 的方法，主线程会等待 t1 结束后才继续运行后面的语句，由于线程 t2 的启动在 join 语句之后，t2 一直等到 t1 退出后才会开始运行。输出结果如图 4-6 所示。

```
Thread 1 delay for 2Wait t1 to end

This is thread Thread 1 on line 0
This is thread Thread 1 on line 1
This is thread Thread 1 on line 2
End of thread Thread 1
Thread 2 delay for 2End of main

This is thread Thread 2 on line 0
This is thread Thread 2 on line 1
This is thread Thread 2 on line 2
End of thread Thread 2
```

图 4-6　多线程示例输出结果

3）thread 模块不支持守护线程。thread 模块中主线程退出的时候，所有的子线程不论是否还在工作，都会被强制结束，并且没有任何警告

也没有任何退出前的清理工作。来看一个例子：

```
from thread import start_new_thread
import time
def myfunc(a,delay):
        print "I will calculate square of  %s after delay for %s" %(a,delay)
        time.sleep(delay)
        print "calculate begins..."
        result = a*a
        print result
        return result

start_new_thread(myfunc,(2,5))# 同时启动两个线程
start_new_thread(myfunc,(6,8))
time.sleep(1)
```

运行程序，输出结果如下，你会发现子线程的结果还未返回就已经结束了。

```
I will calculate square of  2 after delay for 5I will calculate square of  6 after delya for 2
```

这是一种非常野蛮的主线程和子线程的交互方式。如果把主线程和子线程组成的线程组比作一个团队的话，那么主线程应该是这个团队的管理者，它了解每个线程所做的事情、所需的数据输入以及子线程结束时的输出，并把各个线程的输出组合形成有意义的结果。如果一个团队中管理者采取这种强硬的管理方式，相信很多下属都会苦不堪言，因为不仅没有被尊重的感觉，而且还有可能因为这种强势带来决策上的失误。实际上很多情况下我们可能希望主线程能够等待所有子线程都完成时才退出，这时应该使用 threading 模块，它支持守护线程，可以通过 setDaemon() 函数来设定线程的 daemon 属性。当 daemon 属性设置为 True 的时候表明主线程的退出可以不用等待子线程完成。默认情况下，daemon 标志为 False，所有的非守护线程结束后主线程才会结束。来看具体的例子，当 daemon 属性设置为 False，默认主线程会等待所有子线程结束才会退出。将 t2 的 daemon 属性改为 True 之后即使 t2 运行未结束主线程也会直接退出。

```
import threading
import time
def myfunc(a,delay):
       print "I will calculate square of  %s after delay for %s" %(a,delay)
       time.sleep(delay)
       print "calculate begins..."
       result = a*a
       print result
       return result

t1=threading.Thread(target=myfunc,args=(2,5))
t2=threading.Thread(target=myfunc,args=(6,8))
print t1.isDaemon()
print t2.isDaemon()
```

```
t2.setDaemon(True)                                    # 设置守护线程
t1.start()
t2.start()
```

4）Python3 中已经不存在 thread 模块。thread 模块在 Python3 中被命名为 _thread，这种更改主要是为了更进一步明确表示与 thread 模块相关的更多的是具体的实现细节，它更多展示的是操作系统层面的原始操作和处理。在一般的代码中不应该选择 thread 模块。

建议 49：使用 Queue 使多线程编程更安全

曾经有这么一个说法，程序中存在 3 种类型的 bug ：你的 bug、我的 bug 和多线程。这虽然是句调侃，但从某种程度上道出了一个事实：多线程编程不是件容易的事情。线程间的同步和互斥，线程间数据的共享等这些都是涉及线程安全要考虑的问题。纵然 Python 中提供了众多的同步和互斥机制，如 mutex、condition、event 等，但同步和互斥本身就不是一个容易的话题，稍有不慎就会陷入死锁状态或者威胁线程安全。我们来看一个经典的多线程同步问题：生产者消费者模型。如果用 Python 来实现，你会怎么写？大概思路是这样的：分别创建消费者和生产者线程，生产者往队列里面放产品，消费者从队列里面取出产品，创建一个线程锁以保证线程间操作的互斥性。当队列满的时候消费者进入等待状态，当队列空的时候生产者进入等待状态。我们来看一个具体的 Python 实现：

```
import Queue
import threading
import random

writelock = threading.Lock()                          # 创建锁对象用于控制输出
class Producer(threading.Thread):
       def __init__(self, q,con,name):
                 super(Producer, self).__init__()
                 self.q = q
                 self.name = name
                 self.con = con
                 print "Producer "+self.name+" Started"
       def run(self):
                 while 1:
                          global writelock
                          self.con.acquire()          # 获取锁对象
                          if self.q.full():           # 队列满
                                   with writelock:# 输出信息
                                             print 'Queue is full,producer wait!'
                                   self.con.wait()     # 等待资源
                          else:
                                   value = random.randint(0,10)
                                   with writelock:
                                             print self.name +" put value
```

```
                                                    "+self.name+":"+ str(value)+
                                                    "into queue"
                            self.q.put((self.name+":"+str(value)))   # 放入队列中
                            self.con.notify()                        # 通知消费者
                    self.con.release()                               # 释放锁对象

class Consumer(threading.Thread):                                    # 消费者
        def __init__(self, q,con,name):
                    super(Consumer, self).__init__()
                    self.q = q
                    self.con = con
                    self.name = name
                    print "Consumer "+self.name+" started\n "
        def run(self):
                    while 1:
                            global writelock
                            self.con.acquire()
                            if self.q.empty():                       # 队列为空
                                    with writelock:
                                              print 'queue is empty,consumer wait!'
                                    self.con.wait()                  # 等待资源
                            else:
                                    value = self.q.get()             # 获取一个元素
                                    with writelock:
                                              print self.name +"get value"+
                                                  value + " from queue"
                                    self.con.notify()                # 发送消息通知生产者
                            self.con.release()                       # 释放锁对象
if __name__ == "__main__":
        q = Queue.Queue(10)
        con = threading.Condition()                                  # 条件变量锁
        p = Producer(q,con,"P1")
        p.start()
        p1 = Producer(q,con,"P2")
        p1.start()
        c1 = Consumer(q,con,"C1")
        c1.start()
```

上面的程序实现有什么问题吗？回答这个问题之前，我们先来了解一下 Queue 模块的基本知识。Python 中的 Queue 模块提供了 3 种队列：

- Queue.Queue(maxsize)：先进先出，maxsize 为队列大小，其值为非正数的时候为无限循环队列。
- Queue.LifoQueue(maxsize)：后进先出，相当于栈。
- Queue.PriorityQueue(maxsize)：优先级队列。

这 3 种队列支持以下方法：

- Queue.qsize()：返回近似的队列大小。注意，这里之所以加“近似”二字，是因为当该值 >0 的时候并不保证并发执行的时候 get() 方法不被阻塞，同样，对于 put() 方

法有效。

- Queue.empty()：列队为空的时候返回 True，否则返回 False。
- Queue.full()：当设定了队列大小的情况下，如果队列满则返回 True，否则返回 False。
- Queue.put(item[, block[, timeout]])：往队列中添加元素 item，block 设置为 False 的时候，如果队列满则抛出 Full 异常。如果 block 设置为 True，timeout 为 None 的时候则会一直等待直到有空位置，否则会根据 timeout 的设定超时后抛出 Full 异常。
- Queue.put_nowait(item)：等价于 put(item, False).block 设置为 False 的时候，如果队列空则抛出 Empty 异常。如果 block 设置为 True、timeout 为 None 的时候则会一直等待直到有元素可用，否则会根据 timeout 的设定超时后抛出 Empty 异常。
- Queue.get([block[, timeout]])：从队列中删除元素并返回该元素的值。
- Queue.get_nowait()：等价于 get(False)。
- Queue.task_done()：发送信号表明入列任务已经完成，经常在消费者线程中用到。
- Queue.join()：阻塞直至队列中所有的元素处理完毕。

Queue 模块实现了多个生产者多个消费者的队列，当多线程之间需要信息安全的交换的时候特别有用，因此这个模块实现了所需要的锁原语，为 Python 多线程编程提供了有力的支持，它是线程安全的。需要注意的是 Queue 模块中的列队和 collections.deque 所表示的队列并不一样，前者主要用于不同线程之间的通信，它内部实现了线程的锁机制；而后者主要是数据结构上的概念，因此支持 in 方法。

再回过头来看看前面的例子，程序的实现有什么问题呢？答案很明显，作用于 queue 操作的条件变量完全是不需要的，因为 queue 本身能够保证线程安全，因此不需要额外的同步机制。那么，该如何修改呢？请读者自行思考。下面的多线程下载的例子也许有助于你完成上面程序的修改。

```
import os
import Queue
import threading
import urllib2
class DownloadThread(threading.Thread):
    def __init__(self, queue):
        threading.Thread.__init__(self)
        self.queue = queue
    def run(self):
        while True:
            url = self.queue.get()              # 从队列中取出一个 url 元素
            print self.name+"begin download"+url+"..."
            self.download_file(url)             # 进行文件下载
            self.queue.task_done()                              # 下载完毕发送信号
            print self.name+" download completed!!!"
    def download_file(self, url):               # 下载文件
        urlhandler = urllib2.urlopen(url)
```

```
            fname = os.path.basename(url)+".html"   # 文件名称
            with open(fname, "wb") as f:            # 打开文件
                while True:
                    chunk = urlhandler.read(1024)
                    if not chunk: break
                    f.write(chunk)
    if __name__ == "__main__":
        urls = ["http://wiki.python.org/moin/WebProgramming",
                "https://www.createspace.com/3611970",
                "http://wiki.python.org/moin/Documentation"
            ]
        queue = Queue.Queue()
        # create a thread pool and give them a queue
        for i in range(5):
            t = DownloadThread(queue)               # 启动 5 个线程同时进行下载
            t.setDaemon(True)
            t.start()

        # give the queue some data
        for url in urls:
            queue.put(url)

        # wait for the queue to finish
        queue.join()
```