第九章

用 Python 完成常见的任务

本章内容：

- 数据结构操作
- 文件操作
- 操作程序
- 与 Internet 相关的任务
- 较大的例子
- 练习

现在，我们已学习了 Python 的语法，它的基本的数据类型，和很多我们喜欢的 Python 的库函数。本章假设你至少理解了这门语言的所有基本成分，并且除了 Python 的优雅和“ 酷 ”的方面外，也了解了它实用的方面。我们将介绍 Python 程序员要面对的常见的任务。这些任务分为 —— 数据结构操作，文件操作等等。

数据结构操作

Python 的最大的特点之一是它把列表、元组和字典作为内置类型。它们非常灵活和容易使用，一旦你开始使用它们，你将发现你会不由自主地想到它们。

内嵌（inline）拷贝

由于 Python 引用的管理模式，语句 `a = b` 并没有对 `b` 引用的对象作拷贝，而只是对那个对象产生了新的引用。有时需要一个对象的新的拷贝，而不只是共享一个引用。怎样做到这一点依赖于对象的类型。拷贝列表和元组的最简单的方式有点奇怪。如果 `myList` 是一个列表，那么要对它做拷贝，你可以用：

```
newList =  myList[:]
```

你可以理解为“ 从开始到结尾的分片 ”，因为我们在第二章“ 类型和操作符 ”里学

到，一个分片开始的缺省索引是序列的开始（0），而缺省的结尾是序列的结尾。由于元组支持同样的分片操作，这个技术也适用于拷贝元组。而字典却不支持分片操作。为了拷贝字典 myDict，你可以用：

```
newDict={}
for key in myDict.keys():
    newDict[key] = myDict[key]
```

这个操作很常见，所以在 Python 1.5 里为字典对象增加了一个新方法来完成这个任务，就是 copy()方法。所以前面的代码可以替换为一句话：

```
newDict = myDict.copy()
```

另一个常见的字典操作现在也是标准的字典特性了。如果你有一个字典 oneDict，而想用另一个不同的字典 otherDict 的内容替换它，只需要用：

```
oneDict.update(otherDict)
```

这与下面的代码相同：

```
for key in otherDict.keys():
    oneDict[key] = otherDict[key]
```

如果在 update()操作前 oneDict 与 otherDict 共享一些键时，在 oneDict 中的键关联的旧值将被删除掉。这也许是你所想要的（通常是这样，这也是为什么选择这个操作并称之为 update()的原因）。如果这不是你期望的，那么要做的也许是抱怨（引发异常），如下：

```
def mergeWithoutOverlap(oneDict, otherDict):
    newDict = oneDict.copy()
    for key in otherDict.keys():
        if key in oneDict.keys():
            raise ValueError, "the two dictionaries are sharing keys!"
        newDict[key] = otherDict[key]
    return newDict
```

或者把二者的值结合为一个元组，例如：

```
def mergeWithOverlap(oneDict, otherDict):
    newDict = oneDict.copy()
```

```
    for key in otherDict.keys():
        if key in oneDict.keys():
            newDict[key] = oneDict[key], otherDict[key]
        else:
            newDict[key] = otherDict[key]
    return newDict
```

为了说明前面三个算法的不同，考虑下面两个字典：

```
phoneBook1 = {'michael': '555-1212', 'mark': '554-1121', 'emily': '556-0091'}
phoneBook2 = {'latoya': '555-1255', 'emily': '667-1234}
```

如果 phoneBook1可能是过时的，而 phoneBook2更新一些但不够完整，那么正确的用法可能是phoneBooke1.update(phoneBook2)。如果认为两个电话本不应该有重复的键时，使用 newBook = mergeWithoutOverlap(phoneBook1, phoneBook2)可以让你知道假设是否有错。最后一种，如果一个是家里的电话本而另一个是办公室的电话本，那么只要是后续的引用 newBook['emily']的代码能够处理newBook['emily']是元组('556-0091','667-1234')这一事实，就可以用：

```
newBook = mergeWithoutOverlap(phoneBook1, phoneBook2)
```

拷贝：copy 模块

回到拷贝主题上来：[:]和.copy()技巧适用于90%的情况。如果你正按照Python的精神，编写可以处理任何参数类型的函数，有时需要拷贝 X 而不管 X 是什么。这时就需要 copy 模块。它提供了两个函数，copy 和deepcopy。第一个就像序列的分片操作[:]或是字典的copy方法。第二个函数更微妙并且与深度嵌套结构有关（这正是 *deepcopy* 的意思）。例如用分片操作[:]完整地拷贝 listOne。这个技术产生了新的列表，如果原来的列表中的内容是不变的对象，如数字或字符串，这个拷贝就是“真正的”拷贝。然而假设 listOne 的第一项是一个字典（或任何其他容易变化的对象），那么 listOne 的拷贝的第一项只是对同一个字典的新的引用。所以如果你修改了那个字典，显然 listOne和它的拷贝都修改了。用一个例子可以看得更清楚些：

```
>>> import copy
>>> listOne = [{"name": "Willie", "city": "Providence, RI"}, 1, "tomato", 3.0]
```

```
>>> listTwo = listOne[:]   # or listTwo=copy.copy(listOne)
>>> listThree = copy.deepcopy(listOne)
>>> listOne.append("kid")
>>> listOne[0]["city"] = "San Francisco, CA"
>>> print listOne, listTwo, listThree
[{'name': 'Willie', 'city': 'San Francisco, CA'}, 1, 'tomato', 3.0, 'kid']
[{'name': 'Willie', 'city':'San Francisco, CA'}, 1, 'tomato', 3.0]
[{'name': 'Willie', 'city': 'Providence, RI'}, 1, 'tomato', 3.0]
```

正如你所见，直接修改 `listOne` 仅仅修改了 `listOne`。对 `listOne` 的第一项的修改影响到 `listTwo`，但没有影响 `listThree`。这就是浅度拷贝（`[:]`）和深度拷贝的区别。`copy` 模块的函数知道如何拷贝可以拷贝的内置函数（注 1），包括类和实例。

排序

在第二章你知道列表有一个排序方法，有时你想要遍历整个排好序的列表而不想影响列表的内容。或者你也许想列出一个排好序的元组，而元组是不可变的，不允许有排序的方法。唯一的解决办法是拷贝一个列表，然后派序列表。例如：

```
listCopy = list(myTuple)
listCopy.sort()
for item in listCopy:
    print item                                  # 或者做别的任何事情
```

这也是处理那些没有内在顺序的数据结构的办法，例如字典。字典非常快的一个原因是实现时保留了改变键的顺序的权利。这其实不是一个问题，因为你可以拷贝字典的键然后遍历它：

```
keys = myDict.keys()                            # 返回字典的未排序的键

keys.sort()
for key in keys:                                # 答应以键为序的健值对
    print key, myDict[key]
```

列表的 `sort` 方法使用的是 Python 的标准比较方案。但有时你需要别的方案。例

注 1： 有些对象是不可拷贝的，如模块、文件对象和套接字。记住，文件对象与磁盘上的文件是不同的。

如当你对一个单词列表排序时，大小写也许是没有意义的。而对字符串的标准比较中，所有大写字母都在小写之前，所以 'Baby' 小于 'apple' 而 'baby' 大于 'apple'。为了进行大小写无关排序，你需要定义一个有两个参数的函数，并且根据第一个参数是小于，等于或大于第二个参数，分别返回 -1，0，1。所以你可以这样写：

```
>>> def caseIndependentSort(something, other):
...     something, other  = string.lower(something), string.lower(other)
...     return cmp(something, other)
...
>>> testList = ['this', 'is', 'A', 'sorted', 'List']
>>> testList.sort()
>>> print testList
['A', 'List', 'is', 'sorted', 'this']
>>> testList.sort(caseIndependentSort)
>>> print testList
['A', 'is', 'List', 'sorted', 'this']
```

我们使用内置的函数 cmp 来完成比较工作。我们的排序函数只是把两项变成小写字母然后排序。也请注意小写转换是在比较函数局部范围内的，所以列表中的元素并没有因排序而修改。

随机：random 模块

怎样随机排列一个序列呢？比如一个文本行的列表。最简单的办法是使用 random 模块里的 choice 函数，它随机地返回序列的元素作为它的参数（注 2）。为了避免重复地得到同样的行，记住删除已经选择了的项。当操作一个列表对象时，使用 remove 方法：

```
while myList:                       # 当 myList 空时停止循环
    element = random.choice(myList)
    myList.remove(element)
    print element,
```

如果你需要随机处理一个非列表对象，通常最简单的办法是把它转换为一个列表，

注 2： random 模块提供了很多有用的函数，例如 random 函数，它返回一个介于 0 和 1 之间的随机浮点数。

然后对这个列表作随机处理，而不是对每种数据类型都采用新的办法。这似乎是一个浪费的办法，也许要产生一个很巨大的列表。然而一般来说，对你似乎很大的数据，对于计算机来说可能不那么大，感谢Python的引用系统。而且不用对每种数据类型采用不同的方法，所节约的时间是很多的。Python的设计初衷就是要节约时间 ;如果那意味着运行一个稍慢一点或者大一点的程序，那就让它这样吧。如果你正在处理大量的数据，也许值得优化。但只有当确实需要优化时才去优化，否则将是浪费时间。

定义新的数据结构

对于数据结构来说，不要重复发明轮子这一原则尤其重要。例如，Python的列表和字典也许不是你习惯于使用的，但如果这些数据结构可以满足要求，你应当避免设计自己的数据结构，它们使用的算法已经在各种情形下测试过，并且快而稳定。但有时这些算法的接口对某个特别的任务不方便。

例如，计算机科学的教科书中经常用其他数据结构术语如队列、堆栈来描述算法。为了使用这些算法，定义与这些数据结构有同样方法的数据结构也许是有意义的。（比如堆栈的pop和push，或者队列的enqueue和dequeue）。而且，重用内置的列表类型来实现堆栈也是有意义的。换句话说，你需要行为像堆栈但却是基于列表的结构。最简单的办法是用一个类来包裹一个列表。为了最低限度地实现一个类，你可以这样写：

```
class Stack:
    def __init__(self, data):
        self._data = list(data)
    def push(self, item):
        self._data.append(item)
    def pop(self):
        item = self._data[-1]
        del self._data[-1]
        return item
```

下面的语句不仅易写，易懂，而且易读，易用。

```
>>> thingsToDo = Stack(['write to mom', 'invite friend over', 'wash the
kid'])
```

```
>>> thingsToDo.push('do the dishes')
>>> print thingsToDo.pop()
do the dishes
>>> print thingsToDo.pop()
wash the kid
```

在上面的堆栈类中用了两个标准的Python命名习惯，第一个是类名由大写字母开始，以便与函数名区别开。另一个是_data属性以一个下划线开始，这介于公共属性（不以下划线开始）和私有属性之间（以两个下划线开始，参见第六章“类”）。而Python的保留字在开始和结尾都有两个下划线。这里的意思是：_data是一个属性，用户不应该直接访问，类的设计者希望这个“伪私有”属性只被类和子类的方法使用。

定义新的列表和字典：UserList和UserDict模块

前面展示的堆栈类作了恰当的工作。它采取了关于堆栈的最小定义，只支持两个操作：push和pop。然而，很快你就发现列表的特性确实好，比如可以用for...in...的方式访问所有的成员。这可以通过重用已有的代码来实现。在这里你应当应用UserList模块里定义的类UserList作为基类，堆栈由此派生而来。库里也包括UserDict模块，它是一个封装字典的类。一般来说，它们是用于特别子类的基类。

```
# 从UserList模块中导入UserList类
from UserList import UserList

# 继承UserList类
class Stack(UserList):
    push = UserList.append
    def pop(self):
        item = self[-1]                    # 使用__getitem__
        del self[-1]
        return item
```

这个堆栈是UserList的一个子类。UserList类通过定义特别的__getitem__和__delitem__方法实现了方括号的运算，所以前面代码里的pop能工作。你不必定义你自己的__init__方法，因为UserList已经定义了一个相当不错的。最后只是说明push方法等于UserList的append方法。现在我们可以用列表和堆栈两种方式来操作了。

```
>>> thingsToDo = Stack(['write to mom', 'invite friend over', 'wash the
kid'])
>>> print thingsToDo                    # 从UserList继承
['write to mom', 'invite friend over', 'wash the kid']
>>> thingsToDo.pop()
'wash the kid'
>>> thingsToDo.push('change the oil')
>>> for chore in thingsToDo:            # 我们也可以用"for ..in .."遍历内容
...     print chore                     # 因为有__getitem__
...
write to mom
invite friend over
change the oil
```

注意：当我们写这本书时，Guido van Rossom 宣布在 Python 1.5.2(以及后续版本里)，列表对象将增加一个 pop 方法，它也有一个可选参数来指定 pop 的索引（缺省是列表最后的一个成员）。

文件操作

脚本语言的设计目标之一是帮助人们快速而简单地做重复工作。Web 管理员，系统管理员和程序员的经常需要做的一件事是：从一个文件集合中选出一个子集，对这个子集做某种操作，并把结果写到一个或一组输出文件中（例如，在某个目录里的每个文件里，隔行查找以非 # 字符开头的行的最后一个词，并把它与文件名一起打印出来）。人们为这类任务已经设计了特定的工具，例如 *sed* 和 *awk*。我们发现 Python 能很简单地完成这个工作。

操作一个文本文件里的每一行

当解析一个包含文本的输入文件时，sys 模块是非常有用的。在它的属性中有三个文件对象，分别称为 sys.stdin、sys.stdout 和 sys.stderr。名字来源于三个流的概念：分别为标准输入、标准输出和标准错误。它们与命令行工具有关，print 语句使用标准输出。它是一个文件对象，具有以写模式打开的文件对象的所有输出方法，如 write 和 writelines。另一个常用的流是标准输入（stdin），它也是一个文件对象，不过它拥有的是输入方法，例如 read、readline 和 readlines。下面的脚本会算出通过“管道”输入的文件行数：

```
import sys
data = sys.stdin.readlines()
print "Counted", len(data), "lines."
```

在 Unix 上你可以做如下的测试：

```
% cat countlines.py | python countlines.py
Counted 3 lines.
```

在 Windows 或 DOS 上，你可以：

```
C:\> type countlines.py | python countlines.py
Counted 3 lines.
```

当实现简单的过滤操作时，`readlines`函数是有用的。这里有一些过滤操作的例子：

寻找所有以 # 开始的行

```
import sys
for line in sys.stdin.readlines():
    if line[0] == '#':
        print line,
```

注意`print`语句后的逗号是需要的，因为`line`字符串里已经有一个换行符。

取出一个文件的第四列（这里列是由空白符定义的）

```
import sys, string
for line in sys.stdin.readlines():
    words = string.split(line)
    if len(words) >= 4:
        print words[3]
```

我们通过`words`列表的长度判断是否确实至少有四个列，最后两行可以用`try/except`惯用法代替，这在 Python 里是常见的：

```
    try:
        print words[3]
    except IndexError:                    # 没有足够的列
        pass
```

取出文件的第四列，列由冒号分开，并用小写输出

```
import sys, string
for line in sys.stdin.readlines():
    words = string.split(line, ':')
```

```
    if len(words) >= 4:
        print string.lower(words[3])
```

打印头 10 行，最后 10 行，并隔行打印输出

```
import sys, string
lines = sys.stdin.readlines()
sys.stdout.writelines(lines[:10])          # 头10行
sys.stdout.writelines(lines[-10:])         # 最后10行
for lineIndex in range(0, len(lines), 2):  # 0, 2, 4, ......
    sys.stdout.write(lines[lineIndex])     # 0, 2, 4, ......行
```

计算单词“Python”在一个文件里出现的次数

```
import string
text = open(fname).read()
print string.count(text, 'Python')
```

把一个文件的列变换为一个列表的行

在这个比较复杂的例子里，任务是“转置”一个文件，设想你有这样一个文件：

```
Name:   Willie   Mark   Guido   Mary  Rachel   Ahmed
Level:    5       4      3       1     6        4
Tag#:    1234   4451   5515    5124   1881    5132
```

而你希望它变成这样：

```
Name:  Level:  Tag#:
Willie 5       1234
Mark   4       4451
...
```

你可以用下面的代码：

```
import sys, string
lines = sys.stdin.readlines()
wordlists = []
for line in lines:
    words = string.split(line)
    wordlists.append(words)
for row in range(len(wordlists[0])):
    for col in range(len(wordlists)):
        print wordlists[col][row] + '\t',
    print
```

当然你应当用更多的防卫性编程技巧来处理一些可能的情况,比如也许不是所有的行都有相同的单词数，也许丢失了数据等等。这些就作为练习留给读者。

选择数据块的大小

前面的所有例子都假设你能一次读入整个文件。然而有时候这是不可能的，比如在内存较小的计算机上处理大文件，或者处理不断地增加的文件（如日志文件）。对这种情况你可以用一个`while/readline`组合，一次读入文件的一小部分直到读完。对于不是基于行的文本文件，你必须一次读入一个字符：

```
# 逐字地读入
while 1:
    next = sys.stdin.read(1)          # 读入一个单字符串
    if not next:                      # 或者读到EOF时是空串
        break
    处理字符串 'next'
```

注意文件对象的`read()`方法在文件结尾时返回一个空串，并由此而跳出循环。然而更常见的是你将处理基于行的文本文件，并且一次处理一行：

```
# 逐行地读入
while 1:
    next = sys.stdin.readline()       # 读入一个单行字符串
    if not next:                      # 或者读到EOF时是空串
        break
    处理行 'next'
```

处理命令行上指定的一组文件

能够读`stdin`是一个重要的特征，它是Unix工具集的基础。可是一个输入并不总是足够的：很多任务需要操作一组文件。这通常是由Python解释器解析作为命令选项传给脚本的参数列表。例如，你键入：

```
% python myScript.py input1.txt input2.txt input3.txt output.txt
```

你也许打算让 *myScript.py* 处理前三个文件，并写一个名为 *output.txt* 的新文件。让我们看这样一个程序的开头是怎样的：

```
import sys
inputfilenames, outputfilename = sys.argv[1:-1], sys.argv[-1]
for inputfilename in inputfilenames:
    inputfile = open(inputfilename, "r")
    do_something_with_input(inputfile)
```

```
outputfile = open(outputfilename, "w")
write_results(outputfile)
```

第二行提取了 sys 模块的 argv 属性的部分。回忆一下那是命令行上的单词列表。列表以这个脚本的名字开始，所以在上面的例子中 sys.argv 的值是：

```
['myScript.py', 'input1.txt', 'input2.txt', 'input3.txt', 'output.txt']
```

脚本假设命令行有一个或多个输入文件以及一个输出文件组成，所以输入文件名的分片起始于 1（跳过脚本名本身，大多数情况下它都不是脚本的输入文件），而结束于最后一个单词前，最后一个单词是输出文件。脚本的其余部分应当是相当容易理解的。

注意前面的脚本实际上没有从文件里读数据，而是把文件对象传递给做实际工作的函数，该函数常常使用文件对象的 readlines()方法，返回文件的行的列表。do_something_with_input()函数的通用版本如下：

```
def do_something_with_input(inputfile):
    for line in inputfile.readlines()
        process(line)
```

处理一个或多个文件的每一行：fileinput 模块

上一个例子是用 sys.argv[1:]这个惯用组合来打开文件，这种情况十分常见，所以 Python 1.5 提供了一个新的模块来做这个工作。它的名字是 fileinput，它是这样工作的：

```
import fileinput
for line in fileinput.input():
    process(line)
```

fileinput.input()调用解析命令行参数，如果脚本没有参数就以 stdin 代替。它也提供了一组有用的函数帮助你了解正在处理的文件名和行号：

```
import fileinput, sys, string
# 从 sys.argv 里取第一个参数并赋值给 searchterm
searchterm, sys.argv[1:] = sys.argv[1], sys.argv[2:]
for line in fileinput.input():
   num_matches = string.count(line, searchterm)
```

```
    if num_matches:             # 大于零表示有匹配
        print "found '%s' %d times in %s on line %d."% (searchterm, num_matches,
            fileinput.filename(), fileinput.filelineno())
```

如果这个脚本称为 *mygrep.py*，它可以这样用：

```
% python mygrep.py in *.py
found 'in' 2 times in countlines.py on line 2.
found 'in' 2 times in countlines.py on line 3.
found 'in' 2 times in mygrep.py on line 1.
found 'in' 4 times in mygrep.py on line 4.
found 'in' 2 times in mygrep.py on line 5.
found 'in' 2 times in mygrep.py on line 7.
found 'in' 3 times in mygrep.py on line 8.
found 'in' 3 times in mygrep.py on line 12.
```

文件名和目录

我们已经讨论了如何读存在的文件，如果你还记得在第二章讨论过的open函数的话，你一定知道如何创建新的文件。然而有许多任务需要不同的文件操作，例如目录管理以及删除文件。你处理这些事务的最好帮手是os和os.path模块，我们在第八章“内置工具”里有介绍。

让我们来看一个典型的例子：你有很多文件，它们的名字都有一个空格，而你想用下划线代替空格。你所需要知道的一切就是 `os.curdir` 属性（当前目录），`os.listdir` 函数（返回指定目录的文件名列表），以及 `os.rename` 函数：

```
import os, string
if len(sys.argv) == 1:                      # 如果没有指定目录
    filenames = os.listdir(os.curdir)       # 就用当前目录
else:                                       # 否则用命令行指定的目录
    filenames = sys.argv[1:]
for filename in filenames:
    if '' in filename:
        newfilename = string.replace(filename, '', '_')
        print "Renaming", filename, "to", newfilename, "..."
        os.rename(filename, newfilename)
```

这个程序工作得很好，但它显示出一些以Unix为中心的倾向。那就是当你用通配符调用它时，例如：

```
python despacify.py *.txt
```

你会发现在 Unix 系统上它对所有以 *.txt* 结尾的文件都作了替换处理，而在 DOS 下它无法工作，因为 DOS 和 Windows 的 shell（命令解释器）并不把 **.txt* 转换为一个文件名列表，而是希望由应用程序来完成转换，* 的意思是匹配任意的字符。

匹配一组文件：glob 模块

glob 模块只输出一个函数，名字也叫 glob，它以文件名模式为参数并返回匹配这个模式的所有文件名（在当前工作目录）：

```
import sys, glob, operator
print sys.argv[1:]
sys.argv = reduce(operator.add, map(glob.glob, sys.argv))
print sys.argv[1:]
```

在 Unix 上测试表明 glob 函数没有做什么，因为 Unix 的 shell 已经完成了 glob 的工作，而在 DOS 上 Python 的 glob 函数得到了同样的结果：

```
/usr/python/book$ python showglob.py *.py
['countlines.py', 'mygrep.py', 'retest.py', 'showglob.py', 'testglob.py']
['countlines.py', 'mygrep.py', 'retest.py', 'showglob.py', 'testglob.py']

C:\python\book> python showglob.py *.py
['*.py']
['countlines.py', 'mygrep.py', 'retest.py', 'showglob.py', 'testglob.py']
```

这个脚本绝非无足轻重，因为它用到了两个概念上较难理解的函数：一个 map，接着是一个 reduce。map 在第四章“函数”里有介绍，而 reduce 现在对你是全新的（除非你有 LISP 类语言的背景）。map 以一个可调用对象（通常是一个函数）和一个序列为参数，依次地用序列中每一个成员为参数调用这个可调用对象，并返回由该对象（即函数）返回的值组成的列表。图 9-1 有一个 map 的示意图（注 3）。

注 3： map 还有更多的功能。例如，如果 None 是第一个参数，map 就把作为第二个参数的序列转换为一个列表。它一次可以操作多于一个序列。

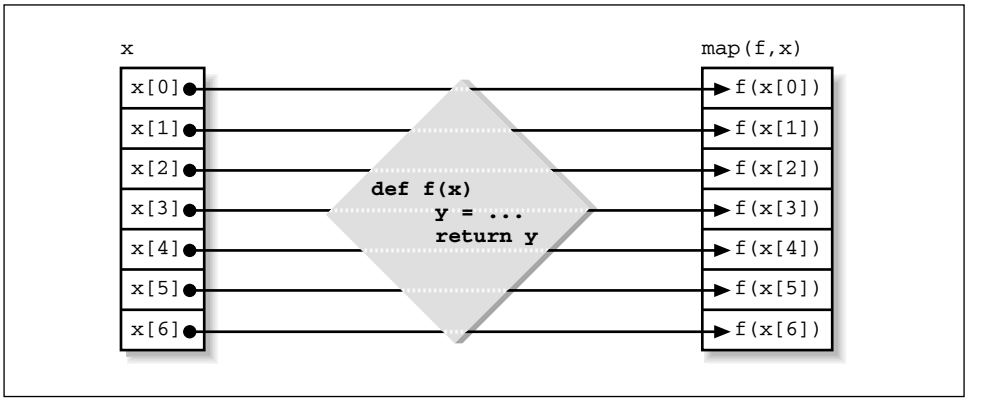

图 9-1　内置 map 函数行为的图形表示

map 在这里是需要的，因为你不知道命令行上输入了多少个参数（也许是 *.py, *.txt, *.doc）。所以，依次地用每个参数调用 glob.glob 函数，而每次调用 glob 就返回匹配文件名模式的文件名列表，map 操作就返回一个列表的列表，你需要把它转换为一个单个的列表 —— 把其中的所有列表组合在一起，也就是 list1 + list2 + … + lintN。这恰好是 reduce 函数派上用场的地方。

与 map 一样，reduce 的第一个参数是函数，并用它的第二个参数（必须是序列）的前两个成员作为参数调用该函数，然后再用返回的结果和序列的下一个成员作为参数再调用该函数（直到结束，请看图 9-2 reduce 的示意图）。但且慢：你需要对一组成员应用“ + ”，而“ + ”不像一个函数，所以你需要一个与“ + ”一样的工作函数，这里有一个：

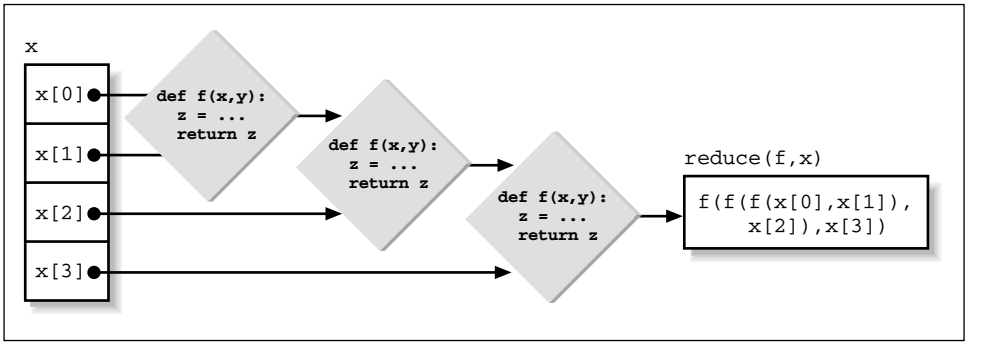

图 9-2　内置 reduce 函数行为的图形表示

```
define myAdd(something, other):
    return something + other
```

于是你可以用 reduce(myAdd, map(…))。这是可行的，但你也可以用 operator 模块定义的 add 函数，这也许更好些。operator 模块为每个 Python 里的操作都定义了函数（包括取属性和分片操作）。你应当用它而不是你自制的函数，这有两个原因。第一，它们已经由 Guido 编码、调试和测试过，他在写无故障代码方面有良好的记录；其次，它们实际上是 C 函数，而对 reduce（或 map，filter）使用 C 函数要比用 Python 函数快得多。当你一次只处理几百个文件时显然是没多大关系，然而如果是数千个文件，速度就很重要了，而现在你知道怎么做更快。

内置的 filter 函数也像 map 和 reduce 一样，以一个函数和序列作为参数，它返回该序列的子集，其中子集的成员能满足函数的条件。下面的例子是找出一个集合里的偶数：

```
>>> numbers = range(30)
>>> def even(x):
...     return x % 2 == 0
...
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29]
>>> print filter(even, numbers)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

或者如果你想找出一个文件中所有至少 10 个字符长的单词，可以这样：

```
import string
words = string.split(open('myfile.txt').read())    # 所有的单词

def at_least_ten(word):
    return len(word) >= 10

longwords = filter(at_least_ten, words)
```

图 9-3 展示了 filter 的工作。filter 有一个特别不错的特点，如果你以 None 为第一个参数，它将过滤掉序列中所有假值。所以为了找出一个文件中的所有非空行，可这样写：

```
lines = open('myfile.txt').readlines()
lines = filter(None, lines)        # 记住，空串是假
```

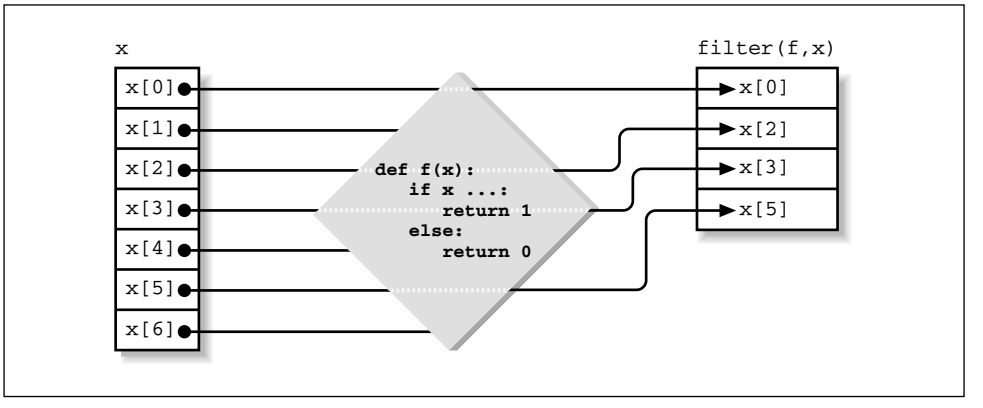

图 9-3 内置 filter 函数行为的图形表示

map、filter和reduce是三个功能很强的函数，它们是值得了解的。然而它们不是必需的。写一个同样功能的Python函数是很简单的，只是内置的版本更快，特别是当操作的是C语言写的函数时，比如operator模块里的函数。

使用临时文件

如果你曾经写过shell脚本，并且用中间文件来存放处理的中间结果，你很可能被目录里的垃圾困扰。你以20个*log_001.txt*，*log_002.txt*等开始，而你想要的只是名为*log_sum.txt*的最终文件，而且你机器里塞满了一系列的*log_001.tmp*，*log_001.tmp2*等临时文件，至少我们的生活是这样的。为了保持你的目录的秩序，用完临时文件就要立即清除掉。

为了帮助解决临时文件的管理问题，Python提供了一个不错的小模块tempfile，它发布了两个函数：mktemp()和TemporaryFile()。前者返回你机器的临时文件目录（如Unix里的*/tmp*和Windows的*c:\tmp*）中未使用的文件名，后者直接返回一个文件对象。例如：

```
# 读输入文件
inputFile = open('nput.txt', 'r')

import tempfile
# 创建临时文件
tempFile = tempfile.TemporaryFile()                    # 我们甚至不需要知道文件名
first_process(input = inputFile, output = tempFile)
```

```
# 创建输出文件
outputFile = open('output.txt', 'w')
second_process(input = tempFile, output = outputFile)
```

当需要立即操作文件对象时使用`tempfile.TemporaryFile()`很合适。它的一个好处是当删除它时它会自动删除对应的文件，保持磁盘的干净。临时文件的一个重要用法是用于`os.system`调用，而这需要文件名而不是文件对象。让我们看一个例子程序，它创建信件并邮寄给一组邮件地址（只用于 Unix）：

```
formletter = """Dear %s,\nI'm writing to you to suggest that ..."""   # 等等
myDatabase = [('Bill Clinton', 'bill@whitehouse.gov.us'),
              ('Bill Gates', 'bill@microsoft.com'),
              ('Bob', 'bob@subgenius.org')]
for name, email in myDatabase:
    specificLetter = formletter % name
    tempfilename = tempfile.mktemp()
    tempfile = open(tempfilename, 'w'
    tempfile.write(specificLetter)
    tempfile.close()
    os.system('/usr/bin/mail %(email)s -s "Urgent!"< %(tempfilename)s' % vars())
    os.remove(tempfilename)
```

`for`循环的第一行返回一个基于给定名字定制的信件，然后把它写入一个临时文件，并准备用`os.system`调用把文件邮寄给合适的邮件地址，最后清除掉临时文件。如果你忘记了`%`的用法，请回到第二章去复习它，它是值得学习的。`vars()`函数是一个内置函数，它返回对应于局部名字空间的变量的字典，字典的关键字是变量名，字典的值是变量的值。`vars()`非常适合于探索名字空间，也可以用一个对象为参数调用它（如一个模块，一个类或者一个实例），而它将返回该对象的名字空间。另外两个内置函数`locals()`和`globals()`函数分别返回局部和全局名字空间。在这三种情况下，修改返回的字典不保证对相应的名字空间有效，所以最好把它们看作只读的。例子中`vars()`函数创建了一个字典，并用于修改字符串，在字符串中的`%(…)s`部分的名字必须与程序里的变量名匹配。

扫描文件技术的更多细节

假设你运行了一个程序，把它的输出存在一个文本文件里，你需要载入它。程序创建一个文件，由一系列行组成，每一行包含由空白符分隔的一个值和一个键：

```
value key
value key
value key
等等...
```

在文件里一个键可在不同行出现，你也许想把一个键的所有值收集在一起，下面是一种可行的办法：

```
#!/usr/bin/env python
import sys, string
entries = {}
for line in open(sys.argv[1], 'r').readlines():
    left, right = string.split(line)
    try:
        entries[right].append(left)       # 扩展列表
    except KeyError:
        entries[right] = [left]           # 第一次遇上

for (right, lefts) in entries.items():
  print "%04d '%s'\titems => %s" % (len(lefts), right, lefts)
```

这个脚本利用了readlines方法逐行扫描文本文件，并调用内置的string.split函数把每一行切割成一个字符串表 —— 表中是值和键。为了保存一个键的所有出现，脚本里用了一个entries字典，循环中的try语句试图把一个键的新值增加到一个存在的成员中，如果没有对应键的成员就创建一个，注意这里的try可以用一个if代替：

```
if entries.has_key(right):            # 在字典里吗？
    entries[right].append(left)       # 把键的当前值加入列表中
else:
    entries[right] = [left]           # 初始化键的列表
```

测试一个字典是否有一个键，有时比用try捕捉异常要快，这依赖于符合测试的次数。这里是运行这个脚本的实际例子，文件名由命令行参数传递（sys.argv[1]）：

```
% cat data.txt
1       one
2       one
3       two
7       three
```

```
8       two
10      one
14      three
19      three
20      three
30      three

% python collector1.py data.txt
0003 'one'     items => ['1', '2', '10']
0005 'three'   items => ['7', '14', '19', '20', '30']
0002 'two'     items => ['3', '8']
```

你可以把扫描逻辑打包到一个函数里，函数返回`entries`字典，并且把循环打印逻辑用一个`if`测试打包在脚本底部，这回更有用：

```
#!/usr/bin/env python
import sys, string
def collect(file):
    entries = {}
    for line in file.readlines():
        left, right = string.split(line)
        try:
            entries[right].append(left)          # 扩展列表
        except KeyError:
            entries[right] = [left]              # 第一次遇上
    return entries

if __name__ == "__main__":                       # 当作为脚本运行时
    if len(sys.argv) == 1:
        result = collect(sys.stdin)              # 从stdin读如
    else:
        result = collect(open(sys.argv[1], 'r')) #从文件里读如
    for (right, lefts) in result.items():
        print "%04d '%s'\titems => %s" % (len(lefts), right, lefts)
```

这样程序更灵活一些。通过这个`if __name__=="__main__"`技巧，你仍然可以把它作为独立脚本来用，或者导入它定义的函数，并处理该函数返回的结果：

```
# 作为独立脚本运行
% collector2.py < data.txt
这里显示结果...

# 用在别的程序中（或交互式的使用）
from collector2 import collect
```

```
result = collect(open("spam.txt", "r"))
这里处理结果 ...
```

由于collect函数的参数是一个打开的文件，它也可以操作任何提供文件方法的对象。如果你只想扫描一个字符串，可以用一个实现了所需接口的类封装该字符串，并把该类的一个实例传给 collect 函数：

```
>>> from collector2 import collect
>>> from StringIO import StringIO
>>>
>>> str = StringIO("1 one\n2 one\n3 two")
>>> result = collect(str)              # 扫描封装的字符串
>>> print result                       # {'one':['1','2'],'two':['3']}
```

这个代码用了标准 Python 库里的 StringIO类，它封装了字符串并提供了所有的文件对象的方法，更多的细节请参考库参考。如果你想修改它的特征，你可以写一个不同的类或从StringIO继承一个子类。无论如何，collect函数愉快地从串 str 读入，而 str 是一个内存里的对象，不是一个磁盘文件。

这里行得通的主要原因是，collect函数避免了对它的file参数作任何类型的假设，只要对象输出了一个返回串表的readlines方法，collect 不关心它处理的对象的类型，这种运行期绑定（注 4）是 Python 对象系统的重要特征，这使得你可容易地写出与其他不同部件通信的程序。例如，考虑一个用标准文件接口读写卫星遥测数据的程序，你可以把它的输入输出流重定向到socket、图形界面窗口、Web 或者数据库，而不需要改变程序，甚至都不需要重新编译。

操作程序

调用别的程序的程序

Python 可以像一个 shell 语言一样用，Python 程序可以用运行时确定的参数调用别的工具。所以，如果你必须运行一个特定的程序（名为analyzeData），它运行

注 4： 运行时绑定的意思是，Python 在程序运行时才知道对象的接口。这是因为 Python 没有类型的声明，这导致了多态性概念的产生。在Python里一个对象操作的意义依赖于被操作的对象。

时需要不同的数据文件和参数，你可以用 os.system 调用，它的参数是一个命令字符串，如下：

```
for datafname in ['data.001', 'data.002', 'data.003']:
  for parameter1 in range(1, 10):
  os.system("analyzeData -in %(datafname)s -param1 %(paramter1)d" % vars())
```

如果 analyzeData 是一个 Python 程序的话，你最好不要激活一个子 shell，只需要用 import 语句导入，并在循环中调用一个函数就可以了。然而，不是每一个有用的程序都是 Python 程序。

在前面的例子中，analyzeData 的输出很可能是一个文件或标准输出。如果是标准输出的话，能够捕捉到它的输出是很不错的。popen() 函数就是做这件事的标准方法，我们将用一个实际的例子来展示它的用法。

当我们写这本书时，要求我们避免在源码列表中使用制表符（tab），而代之以空格字符。制表符可能受排版的影响，而 Python 的缩进格式是重要的，不正确的排版有可能破坏例子的使用。但由于旧有的习惯难以消除（至少我们中的一个喜欢用制表符），所以在交付排版前，我们需要一个工具找出任何可能进入我们代码的制表符。下面的脚本 *findtabs.py* 就是做这件事的：

```
#!/usr/bin/env python
# find files, search for tabs

import string, os
cmd = 'find . -name "*.py" -print'        # find 是一个标准 Unix 工具

for file in os.popen(cmd).readlines():     # 运行 find 命令
    num  = 1
    name = file[:-1]                       # 去掉 '\n'
    for line in open(name).readlines():    # 扫描文件
        pos = string.find(line, "\t")
        if  pos >= 0:
            print name, num, pos           # 报告找到的 tab
            print '....', line[:-1]        # [:-1]去掉最后的 \n
            print '....', ''*pos + '*', '\n'
        num = num+1
```

这个脚本用了两个嵌套的 for 循环，外面的循环用 os.poepn 运行一个 find 命令，返回在当前目录和子目录里所有的 Python 程序名。内部的循环逐行阅读当前文

件，用string.find函数寻找制表符。这个脚本的真正魔术就是它调用的内置工具：os.popen：

os.popen
: 以一个 shell 命令字符串作为参数，并返回一个链接到标准输入或标准输出的文件对象，如果你没有给出一个"r"或"w"参数的话，缺省是标准输出。通过读入这个文件对象，你可以像我们一样截取命令的输出 —— find的结果。在标准库里有一个名为find.py的模块提供了一个类似的函数，作为练习，你可以用它来重写 *findtabs.py*。

string.find
: 在一个字符串里从左到右地找一个子串，并返回的一个位置的索引，我们用它来找制表符，'\t'（转义的字符）。

当找到一个制表符时，脚本打印匹配的行，以及一个指示制表符位置的指针。注意字符串重复的用法：表达式''*pos把打印光标恰好移动到一个制表符的位置。在cmd这个单引号引用的串里使用了双引号，而不是反斜杠转义符号。下面是脚本的工作结果，捕获了非法的制表符：

```
C:\python\book-examples> python findtabs.py
./happyfingers.py 2 0
....   for i in range(10):
.... *

./happyfingers.py 3 0
....           print "oops..."
.... *

./happyfingers.py 5 5
.... print      "bad style"
....      *
```

关于移植性的说明：脚本里用的find命令是一个Unix命令，在其他平台里也许没有。os.popen在Windows版本的win32扩展里对应的是win32pipe.popen（注5）。如果你希望捕获shell命令输出的代码可移植，可使用下面的代码：

注5： 两个关于兼容性的重要注释：win32pipe模块也有一个popen2调用，与Unix的popen2调用相似，但它返回读写的管道的次序不同（参阅文档，可获得posix模块中poprn2的更多信息）。Mac机上没有popen的等价物，因为不存在管道。

```
import sys
if sys.platform == "win32":                # Windows
    try:
        import win32pipe
        popen = win32pipe.popen
    except ImportError:
        raise ImportError, "win32pipe模块找不到"
else:                                      # POSIX
    import os
    popen = os.popen
...随后可使用popen
```

`sys.platform`属性总是预设为一个标识所在平台的字符串。尽管 Python 语言是不依赖于平台的，但某些库也许是依赖的，检查`sys.platform`是一种标准的处理方式。注意这里嵌套的`import`语句，我们可看到，`import`是一个可执行语句，它给一个变量名赋值。

与 Internet 相关的任务

Internet是一个信息的宝库，但它的指数级的增长使它很难管理。而且大多数冲浪的工具都是不可编程的。用 Python 标准版的一些工具可以很简单地使很多 Web 相关的任务自动化。

自动下载一个 Web 页

如果你对找出一个地方数月来的天气情况感兴趣，你可以做一个程序自动地去收集信息。

这里是一个程序，它从 weather.com 网站寻找几个州和城市的天气：

```
import urllib, urlparse, string, time

def get_temperature(country, state, city):
    url = urlparse.urljoin('http://www.weather.com/weather/cities/',
                           string.lower(country)+'_' + \
                           string.lower(state) + '_' + \
                           string.replace(string.lower(city), '',
                                          '_') + '.html')
```

```
        data = urllib.urlopen(url).read()
        start = string.index(data, 'current temp: ') + len('current temp: ')
        stop = string.index(data, '&deg;F', start-1)
        temp = int(data[start:stop])
        localtime = time.asctime(time.localtime(time.time()))
        print ("On %(localtime)s, the temperature in %(city)s, " +\
               "%(state)s %(country)s is %(temp)s F.") % vars()

    get_temperature('FR', '', 'Paris')
    get_temperature('US', 'RI', 'Providence')
    get_temperature('US', 'CA', 'San Francisco')
```

它运行时的输出是：

```
~/book:> python get_temperature.py
On Wed Nov 25 16:22:25 1998, the temperature in Paris,  FR is 39 F.
On Wed Nov 25 16:22:30 1998, the temperature in Providence, RI US is 39 F.
On Wed Nov 25 16:22:35 1998, the temperature in San Francisco, CA US is 58 F.
```

这段代码有一个缺点，创建URL和提取温度的逻辑依赖于Web站点产生的HTML文件。如果某天该站点的图形设计员决定“ current temp: ”应该是大写，这个脚本就失效了。这只有等 Web 也采用更结构化的格式（如 XML）时，用程序解析 Web 页的问题才能得以解决（注 6）。

检查超链接的正确性和做 Web 镜像：Webchecker.py

维护一个Web站点的大麻烦之一是：随着站点里的超链接增加，一些链接失败的机会也增加了。好的Web站点维护会定期检查链接情况，Python标准版里有这样一个工具名为 *Webchecker.py*，位于 *tools/Webchecker* 目录。

在同一个目录里还有一个 *Websucker.py*，它能为远程的Web站点做本地拷贝。做的时候要小心，因为如果你不小心也许会把整个Web都下载到你的机器！在同一个目录里还有 *wsgui.py* 和 *Webgui.py* 这两个程序，它们分别是前面两个程序的基于Tkinter的前端程序。我们建议你看一看这些程序的源代码，学习如何用Python的标准工具建造复杂的 Web 管理系统。

注 6： XML 是一种标记结构化文本的语言，它强调了文档的结构，而不是图形特征。XML 处理是 Python 文本处理的全新领域，正在开发中。参见附录一“ Python 资源 ”。

在 *tools/scripts* 目录里，你将发现很多其他也许是有趣的中小规模的脚本，例如与 *Websucker.py* 相似的 ftp 版本 *ftpmirror.py*。

检查邮件

在当今的 Internet 上最重要的媒介很可能是电子邮件，它肯定是个人之间传递大多数信息的协议。Python 有几个处理邮件的库。你需要用哪一个依赖于你使用的邮件服务器类型。其中包括 `poplib` 模块（POP3 服务器）和 `imaplib` 模块（IMAP 服务器）。如果你需要与微软的 Exchange 服务器对话，你将需要一些 win32 版的工具（参见附录二中 win32 扩展的 Web 地址）。

这里有一个 `poplib` 的简单测试，它与运行 POP 协议的服务器对话：

```
>>> from poplib import *
>>> server = POP3('mailserver.spam.org')
>>> print server.getwelcome()
+OK QUALCOMM Pop server derived from UCB (version 2.1.4-R3) at spam starting.
>>> server.user('da')
'+OK Password required for da.'
>>> server.pass_('youllneverguess')
'+OK da has 153 message(s) (458167 octets).'
>>> header, msg, octets = server.retr(152)  # 取最新的信息
>>> import string
>>> print string.join(msg[:3], '\n')   # 看前三行
Return-Path: <jim@bigbad.com>
Received: from gator.bigbad.com by mailserver.spam.org (4.1/SMI-4.1)
        id AA29605; Wed, 25 Nov 98 15:59:24 PST
```

在实际的应用里你需要使用特殊的模块，比如用 `rfc822` 来解析邮件首部，也许要用 `mimetools` 和 `mimify` 模块从邮件信息体里取数据（比如处理附件）。

较大的例子

计算你的复利

有时我们希望把一些钱存在银行帐号里，银行也很欢迎并愿意付利息。一般你的银行是根据你的存款数付给你利息，而且他们会把增加的利息加到你的总数里，每年你的存款会增加一些。这里是一个简单计算每年增长的 Python 程序：

```
trace = 1  # 打印每一年吗?

def calc(principal, interest, years):
    for y in range(years):
        principal = principal * (1.00 + (interest / 100.0))
        if trace: print y+1, '=> %.2f' % principal
    return principal
```

这个函数逐年地累积你的存款数(你的初始存款加上利息),它假设你不会中途取走钱。现在假设我们有￥65000,利息率为5.5%,而我们想知道十年后会增加多少。我们导入并调用我们的利息计算函数,参数是初始存款、利息率和计划存入的年数:

```
% python
>>> from interest import calc
>>> calc(65000, 5.5, 10)
1 => 68575.00
2 => 72346.63
3 => 76325.69
4 => 80523.60
5 => 84952.40
6 => 89624.78
7 => 94554.15
8 => 99754.62
9 => 105241.13
10 => 111029.39
111029.389793
```

而我们最后获得￥111029。如果我们只想知道最后的结果,我们可以在调用前把`trace`变量设为0:

```
>>> import interest
>>> interest.trace = 0
>>> calc(65000, 5.5, 10)
111029.389793
```

自然地有很多方式计算复合利息。例如下面是另一个例子,它逐年地打印出赚取的利息和存款数:

```
def calc(principal, interest, years):
    interest = interest / 100.0
    for y in range(years):
```

```
        earnings  = principal * interest
        principal = principal + earnings
        if trace: print y+1, '(+%d)' % earnings, '=> %.2f' % principal
    return principal
```

我们得到同样的结果，但信息更多：

```
>>> interest.trace = 1
>>> calc(65000, 5.5, 10)
1 (+3575) => 68575.00
2 (+3771) => 72346.63
3 (+3979) => 76325.69
4 (+4197) => 80523.60
5 (+4428) => 84952.40
6 (+4672) => 89624.78
7 (+4929) => 94554.15
8 (+5200) => 99754.62
9 (+5486) => 105241.13
10 (+5788) => 111029.39
111029.389793
```

对这个脚本的最后注释是，它也许与银行计算的不完全一样。银行的程序是精确到分，我们的程序在打印结果时也精确到分（`%.2f` 的意思请参见第二章），但保留了计算机提供的计算结果的完整精度（见最后一行）。

自动拨号脚本

曾经有一个朋友在一个没有 Internet 连接的公司工作。但系统支持部门安装了一个拨号 modem，所以任何有 Internet 帐号并知道一点 Unix 的人就可以上网了。拨号时使用 Kermit 文件传送工具。

使用 modem 的一个缺点是，想拨出去的人不得不不断地尝试 10 个可用的 modem，直到找到一个空闲的。由于在 Unix 上可用文件名模式 */dev/modem** 访问 modem，用 */var/spool/locks/LCK*modem** 锁住 modem，所以只需要一个简单的 Python 脚本就可以自动的检查空闲的 modem。下面的程序 *dokermit.py* 使用了一个整数列表来跟踪锁住的 modem，`glob.glob` 用来做文件名扩展。当找到空闲的 modem 后用 `os.system` 运行一个 Kermit 命令：

```
#!/usr/bin/env python
```

```
# find a free modem to dial out on

import glob, os, string
LOCKS = "/var/spool/locks/"

locked = [0] * 10
for lockname in glob.glob(LOCKS + "LCK*modem*"): # 找锁住的 modem
    print "Found lock:", lockname
    locked[string.atoi(lockname[-1])] = 1        # 0..9 在名字末尾

print 'free: ',
for i in range(10):                              # 拨号报告
    if not locked[i]: print i,
print

for i in range(10):
    if not locked[i]:
        if raw_input("Try %d? " % i) == 'y':
            os.system("kermit -m hayes -l /dev/modem%d -b 19200 -S" % i)
            if raw_input("More? ") != 'y': break
```

按惯例, modem锁的文件名末尾是modem的号, 我们就用这个信息来构造Kermit命令里的modem 设备名。注意脚本里用了10个整数的列表来标识空闲的 modem（1表示锁住）。这个程序只能用于10个modem以内的情况, 如果有更多的modem, 你需要用更大的列表和循环，并解析文件名，而不只是看最后一个字符。

一个交互式的朋友名单

尽管大多数前面的例子都用列表作主要的数据结构, 但字典在很多方面更强大和更有趣。正是它的存在使得Python的层次较高，也就是说“容易用来对付复杂的问题”。对这个丰富的内置数据类型的一个重要的补充是一个扩展的标准库, 一个强大的模块 `cmd` —— 它提供了一个 `Cmd` 类，你可以继承它来生成一个简单的命令行解释器。下面的例子有点大，但并不太复杂，它很好地说明了字典的威力和对标准模块的重用。

我们的任务是记录名字和电话号码，并为使用者提供交互式操作界面，提供错误检查和友好的在线帮助。下面演示了交互的过程：

```
% python rolo.py
Monty's Friends: help
```

```
Documented commands (type help <topic>):
========================================
EOF             add             find            list            load
save

Undocumented commands:
======================
help
```

我们可获得特定命令的帮助：

```
Monty's Friends: help find              # 与 help_find()方法比较
Find an entry (specify a name)
```

我们可以简单的操作记录本里的记录：

```
Monty's Friends: add larry                  # 我们可以增加
Enter Phone Number for larry: 555-1216
Monty's Friends: add                        # 如果没有给出名字
Enter Name: tom                             # 程序会问
Enter Phone Number for tom: 555-1000
Monty's Friends: list
========================================
               larry : 555-1216
                 tom : 555-1000
========================================
Monty's Friends: find larry
The number for larry is 555-1216.
Monty's Friends: save myNames               # 保存我们的工作
Monty's Friends: ^D                         # 退出程序(Windows 上是 ^Z)
```

而最妙的是当我们重新启动程序时，我们可列出保存的数据：

```
% python rolo.py                    # 重启动
Monty's Friends: list               # 缺省地，没有任何数据
Monty's Friends: load myNames       # 重新装入数据
Monty's Friends: list
========================================
               larry : 555-1216
                 tom : 555-1000
========================================
```

大多数的交互式解释器功能是由 cmd 模块的 Cmd 类提供的，我们只需要作一些定制，我们需要设置 prompt 属性并增加一些以 do_ 和 help_ 开头的方法。do_ 方法

必须有一个参数，而 do_后面的部分就是命令的名字。一旦你调用了 cmdloop() 方法，一切都由 Cmd 类来完成了。读下面的程序 *rolo.py*，一次读一个方法并与前面的输出作比较：

```
#!/usr/bin/env python
# 一个交互式的电话本

import string, sys, pickle, cmd

class Rolodex(cmd.Cmd):

    def __init__(self):
        cmd.Cmd.__init__(self)              # 初始化基类
        self.prompt = "Monty's Friends: "   # 定制提示符 prompt
        self.people = {}                    # 一开始我们谁都不认识

    def help_add(self):
        print "Adds an entry (specify a name)"
    def do_add(self, name):
        if name == "": name = raw_input("Enter Name: ")
        phone = raw_input("Enter Phone Number for "+ name+": ")
        self.people[name] = phone           # 增加对应于姓名的电话号码

    def help_find(self):
        print "Find an entry (specify a name)"
    def do_find(self, name):
        if name == "": name = raw_input("Enter Name: ")
        if self.people.has_key(name):
            print "The number for %s is %s." % (name, self.people[name])
        else:
            print "We have no record for %s." % (name,)

    def help_list(self):
        print "Prints the contents of the directory"
    def do_list(self, line):
        names = self.people.keys()          # 键是姓名
        if names == []: return              # 如果没有姓名就退出
        names.sort()                        # 我们希望是字母排序
        print '='*41
        for name in names:
           print string.rjust(name, 20), ":"; string.ljust(self.people[name],
20)
        print '='*41
```

```
        def help_EOF(self):
            print "Quits the program"
        def do_EOF(self, line):
            sys.exit()
        def help_save(self):
            print "save the current state of affairs"
        def do_save(self, filename):
            if filename == "": filename = raw_input("Enter filename: ")
            saveFile = open(filename, 'w')
            pickle.dump(self.people, saveFile)

        def help_load(self):
            print "load a directory"
        def do_load(self, filename):
            if filename == "": filename = raw_input("Enter filename: ")
            saveFile = open(filename, 'r')
            self.people = pickle.load(saveFile)     # 注意这将覆盖
                                                    # 任何存在的人的记录

    if __name__ == '__main__':                  # 这样模块也可以被别的程序导入
        rolo = Rolodex()
        rolo.cmdloop()
```

people变量只是姓名和电话号码之间的的映射，add和find方法使用这个映射。命令是以do_开头的方法，而它们的帮助是对应的help_方法。最后，load和save用了pickle模块，我们将在第十章作更详细的解释。

这个例子演示了扩展Python已有模块的威力。cmd模块负责处理提示符、帮助功能和解析输入。pickle模块做了所有保存和装入的工作。我们需要写的就是与我们的任务相关的部分。通用的交互式解释器部分是免费的。

练习

本章里充满了我们建议你尝试的例子，但如果你真的想练习，这里有一些更具挑战性的习题：

1. 重定向标准输出stdout。修改脚本*mygrep.py*的输出到命令行上指定的最后一个文件，而不是屏幕。

请留意 Cmd 类是怎样工作的？

为了理解Cmd类是怎样工作的，请阅读已安装的Python标准库里的cmd模块。

Cmd 解释器的 onecmd()方法完成了我们感兴趣的工作的大部分，每次用户输入一行后就会调用它。这个方法从输入行中识别出第一个单词，然后它会查找 Cmd 的子类的实例中是否有对应的属性（如果命令是 "find tom"，它会查找do_find属性）。如果它找到了对应的属性，就会用命令行的参数（这里是"tom"）调用对应方法并返回结果。onecmd()方法曾经是这样的（你的版本也许有所增加）：

```
# Cmd 类的 onecmd 方法，见 Lib/cmd.py
def onecmd(self, line):            # 输入行例子 "find tom"
    line =   string.strip(line)    # 去掉多余的空白符
    if not line:                   # 如果没有输入
        line = self.lastcmd        # 重复上一次的命令
    else:
        self.lastcmd = line        # 为下一次保存命令
    i, n = 0, len(line)
                                   # 下一行作用是找出第一个单词尾
    while i < n and line[i] in self.identchars: i = i+1
                                   # 把输入分解为 command + arguments
    cmd, arg = line[:i], string.strip(line[i:])
    if cmd == ':                   # 如果输入不是以 A-z 开始
        return self.default(line)
    else:                          # cmd 是 'find', line 是 'tom'
        try:
            func = getattr(self, 'do_' + cmd)  # 寻找方法
        except AttributeError:
            return self.default(line)
        return func(arg)           # 用输入行的剩余部分调用方法
```

2. 写一个简单的shell。用cmd模块的Cmd类和第八章中操作文件和目录的函数写一个 shell，它能解释标准 Unix 命令（或者 DOS 命令）：ls（dir）列出当前目录，cd改变当前目录，mv（ren）移动/改名一个文件，cp（copy）拷贝一个文件。

3. 理解 map、reduce 和 filter。如果你是第一次遇到这类函数，它们有点难以理解，部分原因是它们的参数有函数，而且这样一个小名字却干了很多工

作。一个确保你理解它们工作原理的好方法是重写它们，在这个练习里，写三个函数（map2、reduce2、filter2），分别做与 map、reduce、filter 同样的事情：

— map2 有两个参数。第一个参数应当是一个函数，或者是 None。第二个参数应该是一个序列。如果第一参数是一个函数，那么将用序列里的成员作为参数调用它，并把结果值以一个列表返回。如果第一参数是 None，序列就被转换为一个列表并返回。

— reduce2 有两个参数。第一个必须是一个有两个参数的函数，第二个必须是一个序列。序列里的前两个成员作为调用该函数的参数，返回的结果又作为新一次调用的第一参数，序列的第三个成员作为第二参数再次调用函数，依此类推，直到遍历了整个序列。最后一次调用的返回值就作为 reduce2 的返回值。

— filter2 有两个参数。第一个可以是 None 或者是一个有两个参数的函数。第二个必须是一个序列。如果第一个是一个 None，filter2 就返回序列中为真值的成员。如果第一个是函数，filter2 就依次用序列里的成员调用该函数，而最后将返回那些在调用中结果为真值的成员。