第九章

常見任務

本章主題：

＊資料結構
＊檔案實錄
＊駕馭工具程式
＊網際網路應用
＊實例操演
＊問題集

這本書已接近尾聲了。我們討論過 Python 的語法，而且基本的資料型態也帶讀者瀏覽一遍，另外，標準程式庫一些常常用得上的函式也做了簡單的交代。這一章寫作的根本信念是假定讀者已經對前面幾章的內容有了粗淺的理解，若有人向你問起 Python 究竟是什麼玩意兒，絕對不會有說不出話來窘境。Python 不僅品質優良，語法簡潔高雅，套上流行文化的術語，簡直可以說酷呆了。當然啦，身為 Python 導師的兩位作者並不期待讀者憑這一本薄薄的教學手冊就能掠盡 Python 的一切精神象徵，Python 的實務應用和完整的參考資料還是有待讀者努力的啃讀，才能創造屬於自己的一片天地。這一章的重點就是要介紹一些 Python 的軟體工程師最容易遇見的工作任務。我們以任務的種類做為分節的依據，先從資料結構說起，再來是檔案實錄，然後是軟體開發、網際網路應用以及軟體工程之始，最後，依然是讀者最愛的練習搖籃曲，問題集是也。

資料結構

Python 擁有的豪華配備中，最值得稱頌的莫過於內建的資料型態了。第二章介紹的串列、辭典和 tuple 等等，都是 Python 最有力的幫手。Python 支援的內建資料型態不僅具備高度的彈性，而且使用上也絕對有便利的優點，一旦熟悉之後，即使無法達到出神入化的境界，想必也能體會出愛不釋手的雀躍之情。

複製之謎

根據 Python 參考物件的技術策略可知，敘述 a = b 並不會使 a 得到 b 所參考的物件（也就是所謂的複製），而是另外再建立一個新的參考關係，使 a 的參考位址指向 b 所參考的物件。然而，有的時候實際複製物件的確是有必要的，共享物件並不能解決所有的問題，如前例的 a = b 就會使得 a 和 b 所儲存的參考位址統統指向 b 所參考的物件。複製的真實與否和物件的型態有密切的關係；複製串列和 tuple 最簡單的方法看起來就有點詭異。假設 myList 是一個串列，如果要做串列的複製，可以像這樣寫：

```
newlist = myList[:]
```

以人類的話來解讀就是“從頭到尾整個切下來複製給 newlist”。第二章談到切片運算時，曾提到切片內定的起始索引值就是序列的開頭（0），而結尾索引值就是序列的尾端；因此上例的切片運算就等於整個把 myList 複製給 newlist。由於 tuple 也支援切片運算，當然上例的複製語法也能適用 tuple。然而，對辭典而言，情況就不是如此。辭典並不支援切片運算，為了達到複製的目的，如複製 myDict，你可以這樣做：

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

辭典的複製可謂家常便飯，Python 1.5 版以後，辭典物件就多了一個新的 copy 成員函式，負責執行複製的工作。因此上面示範的辭典複製其實可以改寫成如下的敘述：

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

另一個常用的辭典運算如今也是辭典物件的標準配備了。如果現在有個辭典 oneDict，你想把 oneDict 的某些內容以另一個辭典 otherDict 的內容予以替換更新，就可以寫成 oneDict.update(otherDict)。如果雞婆一點要自己寫也行：

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

如果 oneDict 的 key 有一些和 otherDict 的重覆，那麼 oneDict key 對映的內容就會被 otherDict key 對映的內容給替換掉，而達到所謂的更新。例如：

```
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
```

不然的話，把兩個辭典相同 key 的兩個內容項目當成 tuple 合併起來也可以：

```
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
```

以上三種不同的演算法說明如下：

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

假設 phoneBook1 的資料很有可能過時了，而 phoneBook2 的資料比較新，但是比較不完整；對此而言，phoneBook1.update(phoneBook2) 也許合用。如果這兩個電話簿你以為都沒有重覆的 key，以 newBook = mergeWithoutOverlap(phoneBook1, phoneBook2) 來進行合併的工作，就可以知道你的先驗假設是否正確了。最後，如果一組電話簿是家人的電話，而另一組是公司同事的電話，也許 newBook = mergeWithOverlap(phoneBook1, phoneBook2) 正是你最想要的合併模式。但是，接下來所寫的程式碼必需能處理 newBook['emily'] 是 tuple（'556-0091','667-1234'）的事實。

複製之二：copy 模組

前面提到的 [:] 和 .copy 的技巧足夠你寫出 90% 的複製，只是格局未免太小。程式要能發揮無遠弗屆的功效，不論物件為何種資料型態，牽涉到複製時，函式都要能一一處理，copy 模組也因此因應而生。copy 模組提供兩個函式，copy 和 deepcopy。函式 copy 就像 [:] 一樣，也像辭典的 copy 成員函式。而函式 deepcopy 就比較令人迷惑，而且和巢狀資料結構頗有淵源（因此才有 deepcopy 之稱）。請測試下面的範例，以切片運算複製 listOne 串列，新生的串列是 listTwo，參考位址同樣指向 listOne 所參考的串列。串列的內容項目如果都是不可變更的物件，如數值和字串，這樣的切片運算得到的就是百分之百的複製。然而，假設 listOne 的第一個元素是一個可變更的物件，如辭典物件，如此複製 listOne 之後，listTwo 的第一個元素儲存的會是一個指向同一個辭典的參考位址。因此，如果後面的程式碼有做該辭典物件的更新動作，不論是 listOne，或是 ListTwo，都會受到更新的影響。換句話說，若是列印 listOne 或 listTwo 的第一個元素，列印出來的內容將會是先前更新過以後的辭典內容了。請看下列的實例說明，整個觀念會更清楚：

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

如你所見，listOne 的附加動作影響的只有 listOne 而已。但是，接著修改的辭典資料項（entry）則由 listOne 和 listTwo 共同參考，因此 listOne 和 listTwo 的辭典內容等於做了同步修正。可是，餘波卻未震到 listThree，這也正是淺層複製和深層複製的差別所在。copy 模組懂得該如何替可複製的內建型態物件做複製的工作，包括類別和實體【註】。

註　有些物件並不能做複製，如模組、檔案物件和 socket。記住一點，檔案物件和存放在磁碟?的檔案已經不能相提並論了。

排序和隨機

第二章讀者曾經和串列的排序成員函式打過招呼。有時候我們會想把串列排序好，再拿來做 for 迴圈的反覆運算，但是並不想真的把原有串列內容的排列順序弄亂；或者我們只是很單純的想把 tuple 的內容做排序（sort）然後列印出來而已。由於 tuple 是不可變更的物件，類似 sort 的運算，對可變更物件而言，自然是不合法的舉措。唯一的解決途徑就是把元素複製出來，變成一個串列，再對這個複製出來的串列進行排序，以此排序好的串列來做各種運算，如下所示：

```
listCopy = list(myTuple)
listCopy.sort()
for item in listCopy:
    print item                      # 或者別的任務
```

這種以複製物件的方式來處理資料結構對沒有次序關係的辭典而言也是很恰當的選擇。辭典之所有能如此有效率的運算，就在於實作的精神把變更 key 排列順序的權利保留給使用者，由使用者自行決定因應之道。這樣的設計機制並不會構成問題，你仍然可以複製辭典，以此複製的辭典進行 key 的排序，再由此排序過的 key 來領出對映的內容：

```
keys = myDict.keys()                # 傳回辭典 key 串列（未排序）
keys.sort()
for key in keys:                    # 印出 key 和數值
    print key, myDict[key]          # 以 key 進行排序
```

串列的 sort 成員函式用的是 Python 標準的比較方法。然而，有時候這種比較方法並不是真正所需的方法，排序所需的操作程序有可能是另類的型態。例如，對字詞組合而成的串列排序，字詞的大小寫就不那麼重要了。文字字串的標準比較法則卻認為大寫字母應該排在小寫字母之前，因此，“Baby”在“apple”之前，但“baby”在“apple”之後。為了避開大小寫的影響，我們得定義一個比較函式，帶有兩個引數，傳回 -1、0 或 1 的數值。如果第一個引數比第二個小，就傳回 -1；如果兩者相等，就傳回 0；如果第一個引數比第二個大，就傳回 1。由此可以定義出我們自己的排序函式，進行比較程序時對字母的大小寫能一視同仁：

```
>>> 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']
```

我們用到了 cmp 內建函式，負責把 'a' 小於 'b'，而 'b' 小於 'c' 的事實找出來。讀者可以發現，我們寫的排序函式不過是把所有的字母統統轉換成小寫字母，再依小寫字母的先後順序排序出結果。讀者可能嚇了一跳，怎麼如此簡單，卻又看起來如此的不起眼？事實正是如此，身為軟體工程師，應當效法物理之美，找出最簡單的路徑，以此路徑走完全程。

random 模組

序列做隨機運算（randomize）是怎麼回事？如果要從某個序列中隨機取出元素，最簡單的方式就是重覆的使用 random 模組的 choice 函式，choice 函式會從序列中隨機挑出一個元素傳回。我們只要把打算做隨機運算序列當做引數傳給 choice 函式，choice 就會自行替我們打理一切了【註】。為了避免 choice 函式隨機選取時選到了原先已選過的元素，請記得把已選過的元素從序列中移除。如果序列指的是串列，可以 remove 成員函式來移除元素：

```
while myList:                              # 如果 myList 為空時，就停止迴圈
    element = whrandom.choice(myList)
    myList.remove(element)
    print element,
```

註　random 模組提供了許多非常有用的函式，如 random 函式便是一例。random 函式會傳回介於 0 到 1 的浮點數數值。請閱讀程式庫參考手冊以明其細節。

如果要做隨機運算的不是串列物件，最好的辦法是把物件轉換成串列物件，以此串列版的物件來做隨機擇取，而不要另外針對每一種資料型態發展額外的應對策略。乍看之下也許是很浪費資源的計策，說不定轉換的物件是很龐大的物件。然而，平心而論，我們覺得大量的東西，對電腦而言可能是九牛一毛。以現有的工具來完成工作本來就是最節省成本的做法，不用傷腦筋去想各種資料型態的應對之道，難道這也有錯？Python 的設計初衷為的就是節省時間；如果你面對的果真是龐大的不得了的資料數量，也許對程式碼的演算法做最佳編碼的工作的確有其必要性，然而，除非問題真的出在程式碼沒有做最佳編碼，否則還是做懶人的好，不然，沒事就想著最佳編碼，只會白白浪費你的青春歲月罷了。

發展新資料結構

對資料結構而言，是否另行研發新式的資料結構更是讀者要謹慎評估的重點。例如，Python 的串列和辭典也許和你過去熟悉的串列和辭典型態有所偏差，但是如果 Python 提供的串列和辭典能夠滿足需求，就沒有道理另外開發新型的串列和辭典。Python 支援的內建資料結構都經過嚴格且廣泛的測試，不僅演算法優良快速，也搏得了穩定的好名聲。然而，有時候對某個特別的任務而言，這些資料結構的介面時常缺乏便利的優點。

例如，電腦科學的教科書講解演算法時常常會以佇列（queue）和堆疊這種資料結構來說明此種演算法。如果要使用這種演算法，最好是有這樣的資料結構，再搭配同樣的成員函式，如堆疊應該有 pop 和 push，而佇列應該有 enqueue 和 dequeue。然而，如果以內建型態來發展堆疊這種資料結構，應該也不是什麼不合理的要求吧。換句話說，我們當然可以發展出堆疊資料結構，只不過是架構在串列的基礎之上。最簡單的做法是用類別把串列包裹（wrap）起來，如下所示：

```
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
```

下面列出來的程式碼片段對讀者而言應該一點問題也沒有：

```
>>> 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
```

Stack 類別中用了兩個標準的 Python 命名慣例。其一是類別名稱的開頭以大寫字母起，以便於和函式做出區別。其二是 __data 屬性的開頭以底線字元起，以此和公眾屬性（開頭沒有底線字元）、私有屬性（開頭有兩個底線字元）以及 Python 保留的關鍵字（開頭和結尾都是底線字元）做出區分。_data 也是類別的屬性，只不過這個 _data 屬性只有類別的成員函式和其子類別的成員函式會用得上而已。

研製新串列和新辭典

前面介紹的 Stack 類別能夠做好模擬堆疊的工作。Stack 很恰巧的反映出堆疊的特性，特別是堆疊只有兩種運算，push 和 pop，Stack 類別都做到了。然而，很快的，你會發現到串列有些特性實在是不錯，例如，我們可以利用串列和 for 迴圈來做反覆運算。要讓先前開發的 Stack 類別擁有串列的特性，只要利用已寫好的程式碼就可以了。以此例而言，UserList 模組所定義的 UserList 類別正是 Stack 繼承串列屬性的最佳母類別。Python 程式庫當中也包含了一個 UserDict 模組，功能類似 UserList。一般而言，UserList 和 UserDict 都是拿來讓子類別繼承用的：

```
# 從 UserList 模組匯入 UserList 類別
from UserList import UserList

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

Stack 此時是 UserList 的一個子類別。類別 UserList 定義了 __getitem__ 和 __delitem__ 成員函式以實作出 [] 的操作運算，這也就是為什麼 pop 的程式碼能夠運作的原因。Stack 類別當中無需再定義 __init__ 了，因為 UserList 當中已經定義了一個相當完美的典範。最後，我們讓 push 成員函式的定義等於 UserList 的 append 成員函式。現在，Stack 類別不但能做堆疊該做的事，也可以做和串列相關的運算：

```
>>> thingsToDo = Stack(['write to mom', 'invite friend over', 'wash the kid'])
>>> print thingsToDo                    # 從 UserList 繼承
['write to mom', 'invite friend over', 'wash the kid']
>>> thingsToDo.pop()
'wash the kid'
>>> thingsToDo.push('change the oil')
>>> for chore in thingsToDo:            # 可以迴圈巡視內容
...     print chore                     # 因為 "for .. in .." 使用 __getitem__
...
write to mom
invite friend over
change the oil
```

注意

Python 的最新版 1.5.2 版中，Guido van Rossum 又替串列物件的成員函式添增了 pop。pop 成員函式有個可有可無的引數，用來指定索引值，以決定從何處做 pop 運算。

檔案實錄

研發命令稿語言的目的之一，就是為了協助操作者能很方便的快速處理一些具有重覆性質的任務。不論是網站管理大師、系統管理人或是軟體工程師，檔案的處理永遠都是最直截了當的日常任務。相信有經驗的你時常得鎖定某一群檔案，挑選出其中一部份進行處理，再把輸出的結果存放到某些個檔案?面。例如，也許你得把某個目錄底下所有的檔案都一一搜尋，把每個檔案文字列開頭不是 # 字元的最後一個字抓出來，再隨著該檔案的名稱一起列印到螢幕上。這種特別的任務時常會出現，自然有因應而生的工具支援，如 Unix/Linux 的 sed 和 awk，多半都是為此而生的命令。然而，讀者若以 Python 的眼光來看問題，會發現 Python 提供的工具，也能夠很輕易的完成這些日常任務。

檔案之列

sys 模組對於處理輸入檔案而言是絕佳的工具，例如，對檔案的文字進行語法分析，或者別類的處置。sys 模組有三種屬性值得再提一次，分別是 sys.stdin、sys.stdout 和 sys.stderr。屬性名稱的命名來自於三種資料流的觀念，分別對應標準輸入、標準輸出和標準錯誤，這幾種資料流的設備可用來結合命令列工具，產生管道間的資料流。標準輸出 stdout 正是 print 敘述最終的輸出地點。stdout 是一個檔案物件，擁有各種輸出成員函式，如 write 和 writelines 等等。另一個常用的資料流是標準輸入 stdin，stdin 也是檔案物件，只不過擁有的都是輸入方面的成員函式，如 read、readline 和 readlines 等等。例如，下列的命令稿會把輸入檔案的列數數算一遍：

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

對 Unix/Linux 而言，可以藉由管道來測試：

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

對 DOS/Windows 的系統而言，同樣可由管道來進行測試：

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

函式 readlines 適用於簡單的過濾運算（filter），以下是幾個例子：

把開頭為 # 字元的每一列都找出來：

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

注意到一點，line 字串已經包含了換行字元，因此 print 敘述的最後要接一個逗號，代表不換行。

把檔案第四個欄位的內容抽取出來（欄位以空白格分隔）：

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

我們測試 words 的長度以確定 words 是否足四字。若有，就把第四個項目印出來。最後面兩列也可以用 try/except 來替換，這種寫法很常見：

```
try:
    print words[3]
except IndexError:                    # 沒有足夠的字
    pass
```

把檔案第四個欄位的內容抽取出來，並全部改成小寫（欄位以分號分隔）：

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

印出隔列以及前十列、後十列：

```
import sys, string
lines = sys.stdin.readlines()
sys.stdout.writelines(lines[:10])             # 首十列
sys.stdout.writelines(lines[-10:])            # 末十列
for lineIndex in range(0, len(lines), 2):     # 取出 0, 2, 4, ...
    sys.stdout.write(lines[lineIndex])        # 取出索引列
```

累計檔案中“Python”出現的次數：

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

把檔案的欄位和列位互換：

這個例子比較複雜一點，任務的目的稱為“轉置”(tranpose) 檔案；數學所談的矩陣運算中，就有所謂的轉置矩陣，也就是把矩陣的列和行互相調換位置。假設有個檔案，儲存的資料格式和內容如下所示：

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

現在，我們打算把格式換成這樣：

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

下列的程式碼能夠滿足需求：

```
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
```

當然啦，以上列舉的程式碼都只是示範而已，讀者應該自行研發更具智慧的程式才行。以上例轉置檔案而言，說不定每一列所含有的字數不會統統都一樣，資料也許並不齊全。這種問題通常都隨任務的性質而異，我們把問題留給讀者練習。

一次讀一點

前面的範例都假定你能一次就把檔案讀完，也正是 readlines 的功用。然而，某些情況下，根本不可能這樣做，例如電腦的主記憶體容量有限，但是檔案的大小卻大於主記憶體的容量；或者，有些檔案的大小並非固定不變，而會隨時附加新的資料進來，例如記錄檔。對此而言，你可以使用 while/readline 的組合，讀檔案時一次只讀一點，直到檔案讀完檔案為止。如果不想以列為單位把檔案讀完，就只好以字元為單位，一次一字元的把檔案讀完：

```
# 一次讀取一字元
while 1:
    next = sys.stdin.read(1)          # 讀出一個字元的字串
    if not next:                      # 遇到 EOF 時就是空字串了
        break
    □ 處理 'next' 字元
```

注意一點，read() 讀到檔案結尾時傳回的是空字串，if 測試一遇到為假的 next，就跳離 while 迴圈的重覆循環。然而，通常我們會以列做為讀取檔案資料的單位，對此可以如下的寫法替換：

```
# 一次讀取一列
while 1:
    next = sys.stdin.readline()       # 讀出一列的字串
    if not next:                      # 遇到 EOF 時就是空字串了
        break
    □ 處理有 'next' 的列
```

命令列的活用

能夠讀取 stdin 可真是一項創舉，這可是 Unix/Linux 工具箱的基礎工程。然而，如果只有一個輸入來源恐怕還是不夠，許多任務之所以能夠完成，靠的不只是一個檔案而已。我們可以想辦法讓 Python 程式分析傳遞給命令稿當做命令列選項的引數，例如：

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

依常理判斷，大概可以推想得到 myScript.py 即將對前面三個輸入檔有所做為，然後把輸出結果寫到同一個檔案，名為 output.txt。想像一下需要這麼多引數的程式開頭會是怎樣的內容：

```
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)
```

第二列把 sys 模組的 argv 屬性內容抽取出來。argv 形成的資料結構是串列，讀者沒忘吧。argv 的第一項內容是此命令稿的名稱，對此例而言，sys.argv 的內容會是：

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

此命令稿假定命令列的引數列包含了一個（以上）的輸入檔案以及一個輸出檔。因此，取出輸入檔的切片運算從 1 開始，而結束於最後一項之前（-1）；最後一項就是輸出檔。程式其餘的部份應該很容易理解了，不過讀者得自行提供 do_something_with_input() 和 write_results() 函式的主體敘述，這個命令稿才有實質的意義。

上列命令稿並沒有從檔案中把資料讀出來，而是把檔案往下傳給函式，希望函式能做些什麼事。通常函式會以 readlines() 來讀取檔案物件，再傳回檔案一列一列的文字。一般而言，do_something_with_input() 大概會長得像這樣：

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

fileinput 模組

上例開啟 sys.argv[1:] 所含的多個檔案是非常常見的工作之一，Python 1.5 版以後引介了另一個新的模組 fileinput，目的就是幫我們處理上一小節所列舉的任務：

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

fileinput.input() 會對命令列的引數進行分析，如果沒有發現引數，就以 sys.stdin 替代。fileinput 提供了許多有用的函式，可以讓我們得知目前正在處理的檔案為何，進行到了那一列等等：

```
import fileinput, sys, string
# 取出 sys.argv 的第一個引數並指定給 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:                    # 只要計算值不是0，就表示比對到了
        print "found '%s' %d times in %s on line %d." % (searchterm, num_matches,
               fileinput.filename(), fileinput.filelineno())
```

假設此命令稿名為 mygrep.py，執行起來應該會像下面這樣：

```
% 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 6.
found 'in' 3 times in mygrep.py on line 8.
found 'in' 3 times in mygrep.py on line 12.
```

檔名和目錄

如何讀取檔案的資料已經詳加剖析了。讀者如果還記得，不妨回想一下第二章談 open 內建函式的情形，怎麼建新檔案，讀者應該不陌生。然而，有許多的工作必須配合不同的處理方式才行，如目錄和路徑的管理，或者移除檔案等等，這些都不是讀取檔案資料和建新檔的函式可以解決的。第八章所提的 os 和 os.path 模組就是你最好的幫手。

我們來看一個典型的例子。假設有很多檔案，每個檔案名稱都有一個空格，現在你想把空格替換成底線字元。對此任務而言，你需要的是 os.curdir 屬性，負責傳回當前的目錄；另一個是函式 os.lisdir，把指定目錄的檔案名稱傳回來；還有一個函式 os.rename，負責更換檔名：

```
import os, string
if len(sys.argv) == 1:                    # 如果沒有指定檔名
    filenames = os.listdir(os.curdir)     # 使用當前目錄
else:
    filenames = sys.argv[1:]              # 否則使用命令列指定的檔案
for filename in filenames:
    if ' ' in filename:
        newfilename = string.replace(filename, ' ', '_')
        print "Renaming", filename, "to", newfilename, "..."
        os.rename(filename, newfilename)
```

這支程式可以跑得很順，不過卻顯露出 Unix/Linux 為中心的地域偏狹。如果配合通用字元來使用，如下所示：

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

你會發現，對 Unix/Linux 而言，會把檔名結尾為 .txt，而且檔名中有空格的檔案全部更動名稱（把空格換成底線字元）。然而，對 DOS 的 shell 而言，這個程式沒有辦法發揮功效，因為 DOS/Windows 所用的 shell 命令並不會把 *.txt 轉換成檔名的列表，這部份要程式自行來完成。

glob 模組

glob 模組釋出一個函式，名為 glob。函式 glob 可以接收一個檔名的樣式，然後傳回符合該檔名樣式的所有檔名（當前目錄）：

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

此例若於 Unix/Linux 系統執行，會發現 Python 的 glob 模組似乎什麼也做不成，因為 Unix./Linux 的 shell 會在 Python 執行之前自行做好先前所說的轉換工作。然而，如果把程式搬到 Dos/Windows 系統來執行，就可以得到和先前相類似的答案：

```
/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']
```

這個命令稿可不是隨便寫寫的，就概念而言，有兩個重點需要提出來說明。reduce.map 的後面還另有一個 map，第四章談函式時就曾經做過介紹，不過 reduce 對讀者而言，應該就是全新的東西了，除非你有 LISP 語言的背景撐腰。map 函式接收可呼叫物件（通常是函式）和一個序列，把序列的每一個元素依序再當成引數來呼叫可呼叫物件，然後傳回串列結果。map 函式若從圖像的觀點來看，可以圖 9-1 來說明【註】。

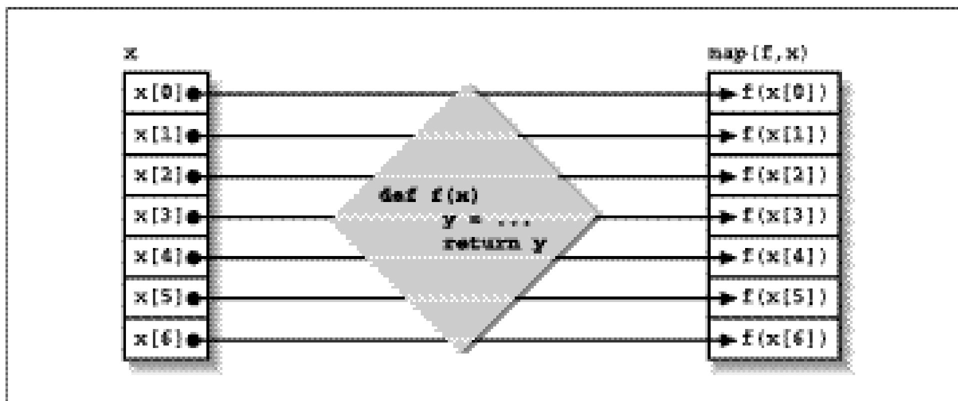

圖 9-1 map 函式的圖解說明

我們並不知到命令列會接收到多少個引數（有可能是 *.py *.txt *.doc），因此 map 的確有其必要。glob.glob 函式會依序接受 sys.argv 的每一個元素，傳回符合檔名樣式的檔案串列給 map，然後 map 再把整個結果以串列型式傳回。map 的運算結果等於是串列的串列，有需要把此疊架轉換成單一的串列，也就是 list1 + list2 + ... + listN。這也正是 reduce 函式存在的意義。

註　map 能夠做的不只這些，例如，如果傳給 map 的第一個引數是 None，map 會把第二個引數的序列轉換成串列。map 一次也可以處理多個序列。詳情請查閱參考手冊。

和 map 一樣，reduce 也是以一個函式當做其第一個引數，然後以第二引數的序列其前面兩個元素當做引數，傳給 reduce 接收的函式運算，再把結果傳回。接著，reduce 把先前的結果和序列中的下一個元素再當成引數傳遞給 reduce 接收的函式運算，如此反覆不斷，如圖 9-2 所示。但是，此時我們需要的是做加法運算，+ 好像不是函式吧（沒錯，不是）。因此，我們需要一個能做加法運算的函式。以下就是一個：

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

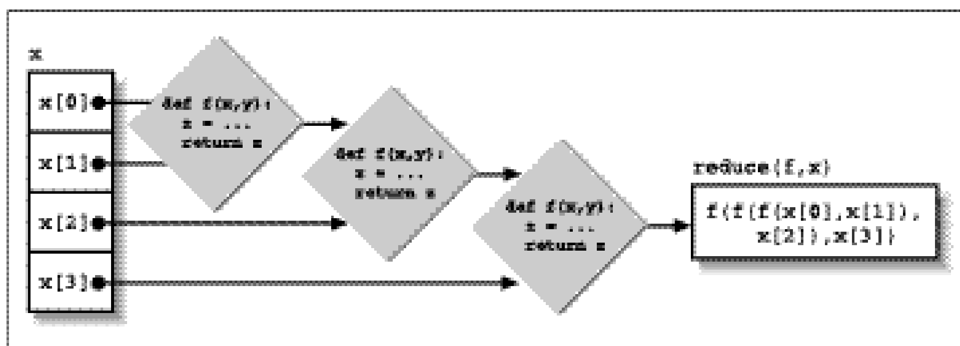

圖 9-2 reduce 函式的說明

你可以寫 reduce(myAdd, map(...))，程式跑起來也不錯，只是還有更好的做法。你可以使用 operator 模組定義的 add 函式，一樣是做加法運算。operator 模組替每一種語法運算都定義了相對的函式，包含切片運算和讀取屬性等等。有兩點好理由支持我們使用現有的函式，而不要親自去做雷同的函式。首先，內建好的程式庫函式通常有嚴格的品管測試，Python 的主持人 Guido 寫起零錯誤的程式碼，在電腦科學領域？可說是記錄優良的高手。其次，這些函式多半都是 C 函式的化身，所以用 C 函式（內建）來執行程式當然比起用 Python 函式（自己動手寫）還要來得有效率。當然，如果你處理的檔案只有幾百個，也許差不了多少，然而，如果是長時間得處理上千個檔案的話，加快執行速度就是必須重視的問題了，現在的你應該知道該怎麼提昇執行效率了。

filter 也是內建函式，和 map、reduce 一樣，也需要一個函式和序列當做引數。filter 接收的函式也會拿序列當做引數運算，傳回真假值，而 filter 函式最終的結果就是把具有真值的序列元素傳回來。例如，要找出某個集合？的所有偶數值，可以這樣寫：

```
>>> 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]
```

另外，也許你想把某個檔案中所有字元長度超過 10 的字找出來：

```
import string
words = string.split(open('myfile.txt').read())          # 取出所有的字

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

longwords = filter(at_least_ten, words)
```

圖 9-3 為 filter 函式的圖形說明。filter 函式還有一個很特別的特色，如果第一個引數是 None，filter 會把序列中為假的項目過濾掉。因此，如果要把 myfile.txt 中不為空列的每一列都找出來，你知道該怎麼做了吧：

```
lines = open('myfile.txt').readlines()
lines = filter(None, lines)                   # 記住，空字串為假
```

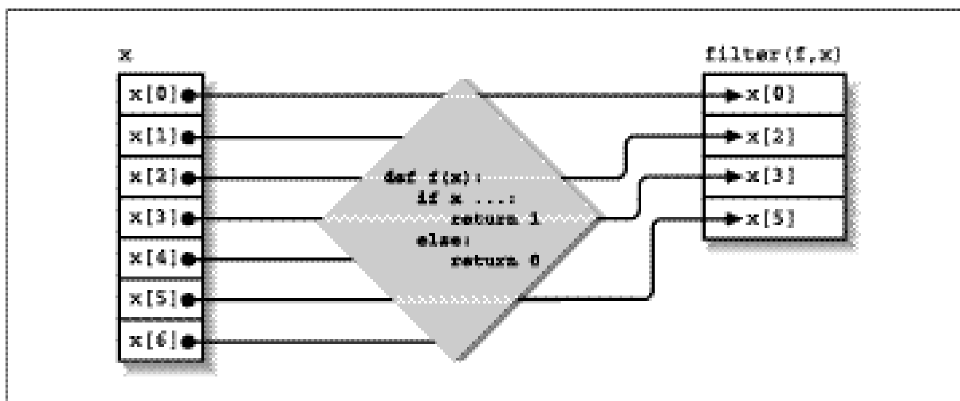

圖 9-3 filter 函式的說明

map、filter 和 reduce 是三個很有威力的函式，值得再三推薦。然而，並非任何場合都非得把視線集中在它們身上。如果要自行寫出類似功能的函式，以 Python 的語法而言，並非難事。只是內建函式不僅要好，而且快，實在是沒有理由花費那麼多時間重覆做一些人家已經做過的事吧。前人經驗有時必需模仿，才有成長的空間，但偶爾得全盤接收，把精力花在更具開創性的價值上。至於思與學該如何反覆搭配，就由你自己決定了。

暫存檔

如果你曾經寫過 shell 命令稿，而且命令稿需要一些檔案來儲存處理過程中的結果，也許就曾親身體會目錄的檔案混亂；也許開始是 log_001.tx、log_002.txt 等等，不過，最終想要的不過是一個檔案罷了，稱為 log_sum.txt。除此之外，可能整個處理過程中還會有一大疊的 log_001.tmp、log_002.tmp 等等。不管怎樣，這種問題對我們而言，是很惱人的瑣事。為了讓目錄的次序完整，一定要用某個特定的目錄來儲存暫存檔，完事之後，再把暫存檔刪除掉。

為了協助我們進行暫存檔的管理，Python 提供了一個很不錯的模組，名為 tempfile，釋出了兩個函式：mktemp() 和 TemporaryFile()。mktemp() 傳回暫存目錄中目前沒有在使用的檔案名稱（如 Unix/Linux 的 /tmp，Winodws 的 C:\TEMP 等等）；而 TemporaryFile() 則直接傳回新的檔案物件。例如：

```
# 讀取輸入檔
inputFile = open('input.txt', 'r')

import tempfile
# 建立暫存檔
tempFile = tempfile.TemporaryFile()                    # 甚至不用知道檔名
first_process(input = inputFile, output = tempFile)

# 建立最後的輸出檔
outputFile = open('output.txt', 'w')
second_process(input = tempFile, output = outputFile)
```

如果需要暫存檔，以 tempfile.TemporaryFile() 來管理是很不錯的做法；其中最值得稱讚的特點是當此檔案物件刪除時，對應於磁碟?的檔案也會跟著刪除，如此一來，自然不會造成目錄的檔案混亂。然而，如果配合 os.system 來使用暫存檔的話就得注意到一點：os.system 如果有用到 shell 命令，指的就不再是檔案物件，而只是檔名了。舉例來說，假設我們要用程式寫幾封信，再分別寄給不同的電傳郵件地址（Unix/Linux 適用）：

```
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 迴圈?的第一列程式碼會把 name 箝入 formletter ?，巧妙的依人名寫出制式化的信件。接著是把信件內容寫入暫存檔，再利用 os.system 把信件電送給合適的電傳郵件地址。最後是清理的步驟，我們把暫存檔移除掉。如果你忘記 % 的運算，請翻回第二章做點複習的功課，懂得愈多絕對沒錯。vars() 是內建函式，傳回一個辭典，內含區域名稱空間?所定義的變數；辭典的 key 就是變數名稱，而辭典的內容就是變數的內容。vars() 對探索名稱空間而言是相當便利的工具，vars() 也能接收物件做為引數，如模組、類別或實體；加了引數之後，vars() 傳回的就是該物件的名稱空間。另外是 locals() 和 globals() 函式，同樣也是內建函式家族的一份子，分別傳回區域和廣域的名稱空間。對此三種情況而言，修改傳回來的辭典會造成怎樣的影響實在無法評估，因此以唯讀的心態來觀看，當然是保證一帆風順的好榜樣。

掃描文字檔

假設你執行了一個程式，把輸出結果儲存到一個文字檔案，而這個文字檔案你得再次載入（load）處理。先前那個程式建立的文字檔案格式由一列一列的文字列組成，每一列都包含一個數值（value）和一個 key，兩者由空白格分開：

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

key 可以重覆出現，因此，你也許會想把每一個 key 對應的多個數值收編起來。換句話說，你有必要掃描檔案。以下是解決問題的方式：

```
#!/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)       # 延伸串列
    except KeyError:
        entries[right] = [left]           # 第一次出現

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

此命令稿以 readlines 一列一列的掃描文字檔，然後呼叫 string.split 把每一列切成子字串，也就是一個內含數值和 key 的串列。為了儲存重覆出現的 key，命令稿使用了一個 entries 辭典。迴圈中的 try 敘述會把新數值加到已存在的 key 資料項。如果 key 不存在，就由 except 建立新的資料項。請注意一點，try 敘述是可以用 if 來改寫的：

```
if entries.has_key(right):                  # 是否已在辭典中?
    entries[right].append(left)             # 增加至串列中
else:
    entries[right] = [left]                 # 設定 key 相對應的初值
```

測試辭典是否含有個 key 有時候比利用 try 敘述捕捉例外事件要還得快；重點就在於測試的結果有多少次為真。以下是此命令稿執行的實例。輸入檔名以命令列引數的形式傳遞（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']
```

我們可以把這個掃描程式以函式重寫，傳回 entries 辭典；在程式尾端在利用 if 測試把列印迴圈包裹（wrap）起來：

```
#!/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)                # 延伸串列
        except KeyError:
            entries[right] = [left]                    # 第一次出現
    return entries

if __name__ == "__main__":                             # 以命令稿執行時
    if len(sys.argv) == 1:
        result = collect(sys.stdin)                    # 從 stdin 讀取
    else:
        result = collect(open(sys.argv[1], 'r'))       # 從檔案讀取
    for (right, lefts) in result.items():
        print "%04d '%s'\titems => %s" % (len(lefts), right, lefts)
```

如此一來，整支程式就變得更有彈性了。利用 if __name__ == "__main__" 的技巧，就可以把這支程式檔成系統命令稿來執行，也可以在互動式環境把函式匯入，處理傳回的辭典等等：

```
# 以命令稿來執行
% collector2.py < data.txt
result displayed here...

# 用於其它元件中
from collector2 import collect
result = collect(open("spam.txt", "r"))
process result here...
```

collect 函式可以接受已開啟的檔案物件，也可以接受其它類型的物件，只要此種物件能夠提供檔案物件所有的成員函式。例如，如果你想從某個簡單的字串讀取文字，只要把該字串包裹在實作出此需求介面的類別，然後再把該類別的實體傳遞給 collect 函式即可：

```
>>> from collector2 import collect
>>> from StringIO import StringIO
>>>
>>> str = StringIO("1 one\n2 one\n3 two")
>>> result = collect(str)                  # 掃描包裹的字串
>>> print result                           # {'one':['1','2'],'two':['3']}
```

此程式碼使用 StringIO 類別，把字串包裹起來，以實體物件出現，而此實體物件擁有檔案物件所有的成員函式：詳細情形請參考 Python 的程式庫手冊有關 StringIO 的部份。如果有需要修改 StringIO 的行為，也可以利用繼承寫出 StringIO 子類別。無論如何，collect 順利地從字串 str 把文字讀取出來，而此 str 字串物件卻是儲存在記憶體中的物件，而非檔案。

繞了一大圈，就是希望 collect 函式的 file 參數不用預設立場，指定某些型態的物件才能運作。只要該物件能釋出 readlines 成員函式，傳回字串串列，collect 函式才不管現在處理的物件型態為何。整個問題的核心就在於介面而已。這種執行期間連結（runtime binding）【註】對 Python 的物件系統而言是很重要的特色，如此，我們可以很容易的寫一些屬於元件層級的程式，和其它元件程式進行溝通。例如，有支程式要利用標準的檔案介面讀取並寫入衛星傳輸訊號的資料。為了把物件導向正確的介面種類，我們可以把衛星傳輸的資料流導向到 socket、GUI、網頁介面或資料庫等等，而無需修改程式，甚至重新編譯。

註　Python 只有在程式執行時才有辦法得知是那一種物件實作了介面，也就是所謂的執行期間連結。這種行為導因於 Python 割棄型態宣告的語法，同時也是為了和同名異式的觀念相容而生。對 Python 而言，物件運算的意義會隨物件的型態而有不同的解釋（如索引參考、切片運算等等）。

駕馭工具程式

呼叫程式

Python 可以當成命令稿語言來使用，能夠呼叫別的程式，加上其所需的引數，而引數可於 Python 程式執行期間才求出。例如，如果你得執行某個特定的程式，暫時叫它作 analyzeData。analyzeData 需要各種不同的資料檔案和參數當作引數，在命令列的環境下執行。因此，你可以呼叫 os.system()，帶上一個字串，把預備執行的 analyzeData 及其所需引數寫成字串，好讓子 shell 去執行：

```
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 也是 Python 程式，最好不要透過子 shell 來執行；只要以 import 敘述把程式匯入，再於迴圈中以函式呼叫的方式來執行就可以了。不過，並不是每一種有用處的程式都是 Python 寫出來的。

前例 analyzeData 的輸出，不是存到檔案?，就是從標準輸出印出來。如果是從標準輸出跑出來，要捕捉輸出內容也不是什麼難事。popen() 函式可以說是這方面的大內高手。稍後我們會以實例說明 popen() 的應用。

當我們在編寫這本書時，有人希望編寫出來的程式碼不要含有跳格字元，全部都以空白字元替代。跳格字元的功效在文字排版來講的確有其優點，而且 Python 的縮排也有其語法上的意義，縮排縮錯了，程式碼的語意也很有可能走樣，然而，並非所有的文書處理程式都能處理跳格字元，過去用於網路上的傳輸，也時常發現莫名所以的問題，因此，跳格字元能不用，就最好不用。但是積習難改，作者之一就習慣以跳格字元來編寫程式碼。為了在交付印刷之前能夠把原始碼中的跳格字元全數換成空白字元，我們特別寫了一支程式，負責處理這種替換的工作。下列的 findtabs.py 就身負這個任務：

```
#!/usr/bin/env python
# 找出檔案，搜尋跳格

import string, os
cmd = 'find . -name "*.py" -print'          # find 是 Unix 標準工具

for file in os.popen(cmd).readlines():       # 執行 find 命令
    num = 1
    name = file[:-1] # strip '\n'
    for line in open(name).readlines():      # 掃描檔案
        pos = string.find(line, "\t")
        if pos >= 0:
            print name, num, pos             # 回報找到跳格字元
            print '....', line[:-1]          # [:-1] 把最後的 \n 抽掉
            print '....', ' '*pos + '*', '\n'
        num = num+1
```

上例用了兩個 for 迴圈。外層的迴圈使用 os.popen 以執行 find 命令，把當前目錄及其子目錄中可存取的 Python 原始碼檔案的串列傳回來。內層的迴圈把檔案中的每一列都讀出來，以 string.find 搜尋跳格字元。這支程式的神奇之處在於其所採用的內建工具：

os.popen

把 shell 命令當成字串傳入（此例是 cmd），然後傳回一個和檔案很類似的物件，連結到 shell 命令的標準輸入或標準輸出。如果沒有指明 "r" 或 "w" 模式的引數，預設的就是輸出。讀取這個類似檔案的物件，就可以攔截 shell 命令的輸出。此例就是在做這件事，也就是攔截 find 的結果。標準程式庫有一個模組，名為 find.py，就提供了一個函式，功能非常類似 popen 和 find 命令的搭配使用。讀者可自行改寫 findtabs.py。

string.find

發現搜尋中的字串時，就把索引值傳回去，搜尋的方向是從左至右。以此例而言，我們用 string.find 來搜尋跳格字元。

跳格字元找到時，命令稿會把符合比對的那一列列印出來，在配上跳格字元出現的位置說明。請注意到字串的重覆運算：' '*pos 會把當前的列印游標往右移到跳格字元所在的索引值位置。以下是命令稿執行的範例：

```
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"
....      *
```

find 命令是 Unix/Linux 的命令，對別種作業平台而言有可能找不到相對等的命令。os.popen 的功能來到微軟的世界之後，就成了 win32 延伸套件的 win32pipe.popen 了【註】。如果你想寫一些具移植能力的程式，最好在命令稿的前面加上下列的程式碼：

```
import sys
if sys.platform == "win32":       # Windows 平台
    try:
        import win32pipe
        popen = win32pipe.popen
    except ImportError:
        raise ImportError, "The win32pipe module could not be found"
else:                             # 否則就是 POSIX 相容的平台
    import os
    popen = os.popen
```

sys.platform 屬性會指出當前的作業系統為何。雖然 Python 語言並沒有作業平台不同的困擾，但是，Python 隸屬的程式庫卻有可能有這種問題，因此，檢查 sys.platform 就是處理平台差異問題的標準手段。注意到一點，import 敘述可以巢狀發展，因為 import 敘處述只是個可執行敘述，指定一個變數名稱而已。

註　win32pipe 模組也有一個 popen2 函式，和 Unix/Linux 版的 popen2 很相像，差別是 win32pipe 的 popen2 傳回管道的讀寫資料時順序和 Unix/Linux 的 popen2 不同。詳情請參考程式庫的細節。另外，popen 對 Mac 系統而言並無意義，因為 Mac 沒有管道這玩意兒。

網際網路應用

網際網路是資訊的寶山，然而呈指數型態的成長速度卻使得管理上出現了危機。再者，目前讓我們得以遊玩於網際間的工具很多都不是兩三句話就可以交代完全的。Python 的世界就替我們營造了簡化的天堂，很多網站相關的任務，若套用 Python 提供的工具，通常都會變得萬分的容易。

下載網頁

如果你想知道某個地區某個時段的天氣如何，最好的方式就是寫一個自動化的程式，把資訊抓下來收集到檔案？。

以下是一支範例程式，連上 weather.com 的網站，把好幾座城市和州的天氣找出來：

```
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')
```

執行時，輸出的結果看起來會像下面這樣：

```
~/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.
```

get_temperature.py 的用途十分有限，而且說不定得配合網站網頁的變更而做調整。如果那一天網頁設計師做了小小的更動，例如，把“current temp:”改成“Current temp:”，程式就完蛋了。這種問題只有當結構化的格式（如 XML）用於生產網頁時，問題才有可能解決吧【註】。

檢查連結和映射（mirroring）的正確性

管理網站最常遇到的問題就是網站的連結時常會失去連繫，尤其是連結的數量愈多，連結老舊的問題就會愈嚴重。因此，好的網站管理就得包含了定期的連結點檢查。Python 提供了一個工具可以做這件事，名為 webchecker.py，位在 Tools/webchecker 的目錄底下。

另一個相關的程式是 websucker.py，位在同一個目錄，以類似的做法建立遠端網站位址的複本，儲存於當地機器。請小心測試，那個地方寫錯了，很有可能整台機器的網站資料都會下載下來。另外，還有 usgui.py 和 webgui.py，也位在相同的目錄，兩者分別是 websucker 和 webchecker 的 Tkinter 前端程式。我們建議讀者不妨參看一下這幾支程式的原始碼，欣賞一下別人是怎麼利用 Python 語言寫出複雜精巧的網站管理程式。

在 Tools/Scripts 目錄底下，還可以發現許多有用的中小程式，讀者應該會很感興趣才對。例如，有一個相當於 websucker.py 的程式，名為 ftpmirror.py，主要用於 FTP 伺服器。

註　XML（eXtensible Markup Language）就是延伸標示語言，用於處理結構化的文字內容，強調的是文件的結構，而非圖像的色彩。XML 對 Python 的文字處理而言，可以說是全然不同的領域。附錄 A 指出了一些相關的討論群組和軟體。

檢查郵件

電子郵件可能是當今網際網路最重要的資訊傳播媒體了；對大眾而言，資訊的交換可以說都是透過電子郵件來完成的。Python 支援了幾個程式庫，用於處理郵件。至於該用那一個，這跟你所使用的郵件伺服器種類有關。例如，poplib 用來和 POP3 伺服器交談，而 imaplib 則用來和 IMAP 伺服器交談。如果你需要和微軟的 Exchange 伺服器交談，win32 延伸套件有提供相關的工具（請參考附錄 B）。

以下是 poplib 模組的測試範例，你可以用下列的範例和 POP 協定的郵件伺服器交談：

```
>>> 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)      # 取得最新的 msgs
>>> import string
>>> print string.join(msg[:3], '\n')            # 並查看前三列
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
```

對於真正的應用程式而言，你應該要使用一個特別的模組去分析郵件表頭，如 rfc822。另外，imnetools 和 mimify 可以用來把郵件主體的資料取出來（例如，附屬檔案等等）。

實例操演

計算利息

銀行戶頭?能夠出現一點多餘的錢，相信是大多數人的願望，也許時候還沒到，但作者們相信也許有一天夢想會成真，學生時代遺留下來的巨額貸款，能馬上付清該有多好啊！銀行也希望存款人多一些，存放的金額也高一些。一般而言，把錢存到銀行的戶頭，銀行得付你本金的利息才行。本金加利息是利上加利，存款額才會每年都有微幅的成長。結論是你得每年都做計畫，才能清楚的掌握銀行存款的成長情形。本節提供 interest.py 程式，教讀者如何以 Python 來寫這種功能的程式：

```
trace = 1     # 每年都印?

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
```

這個程式不過繞著你輸入的年數跑迴圈，累加每一年的本金金額。這個程式的假設之一是你不會把錢提出來。現在，假定我們有 65,000 元，要開一個戶頭，利率是 5.5%。我們想知道 10 年以後，本金能夠成長多少。首先把 interest.py 載入，呼叫計算本金的函式 calc，把最初的本金、利率和年數傳遞給 calc：

```
% 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
```

結果我們得到 111,029 元的答案。如果只想看最後的輸出結果，可以把 interest 的廣域變數 trace 設定成 0，然後再呼叫 calc 函式：

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

事實上，表現複利的方法有很多種。例如，下列變種的利息計算函式，把本金和利息都特別分了出來，印出來的結果也有利息和本金：

```
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
```

我們得到的結果和上一版的函式一樣，但是可以提供更多的資訊：

```
>>> 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
```

最後要提醒讀者，你開戶的銀行不見得會給你同樣的計算準則。

自動撥出命令稿

很久很久以前，有一個寫書的人在一家公司上班，但是，因為沒有辦法連上網際網路，結果整日愁眉苦臉。然而，如果公司的系統的確有安裝數據機，提供往外撥出去的連線功能，那麼任何人只要有網際網路的個人帳號和一點 Unix 的經驗，就可以連線到 shell 的帳號，做一點和網際網路相關的生意和工作。所謂的撥出指的就是使用 kermit 檔案傳輸公用程式。

對這種數據機而言，有一點最大的壞處就是想撥出的人都得不停的試撥，直到發現其中一個沒有人用的數據機才能連上去。由於 Unix 系統的數據機是有名稱而且可以定址的，例如，檔名的樣式就是 /dev/modem*；另外，數據機會經由 /var/spool/locsk/LCK*modem* 來上鎖。換句話說，一個簡單的 Python 命令稿就足以完成檢測空線中的數據機了。下列的程式使用了一組整數串列，負責追蹤上鎖的數據機，而 glob.glob 負責檔名的處理，最後，一旦發現空線的數據機，就呼叫 os.system 執行 kermit 命令：

```
#!/usr/bin/env python
# 找出閒置中的數據機用於撥出

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

locked = [0] *10
for lockname in glob.glob(LOCKS + "LCK*modem*"):          # 找出上鎖的數據機
   print "Found lock:", lockname
   locked[string.atoi(lockname[-1])] = 1                # 0 .. 9 位在名稱尾端

print 'free: ',
for i in range(10):                                # 回報，撥出
   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
```

上例的命令稿使用 10 個整數的串列當做旗號，把空出來的數據機標上記號，而以 1 代表上鎖。這支程式只適用 10 台以下的數據機環境，如果超過這個數量，就有必要用更大的整數串列來標示數據機，而且上鎖檔的檔名也有必要進行分析，不能只取最後一個字元作為識別之用。

互動式資料庫

前面舉的範例都以串列做為主要的資料結構。事實上，就多數情況而言，辭典才是真正主其事者。高階資料結構內建於 Python，正是 Python 足以笑傲群雄之處，任務再怎麼複雜，談笑間，煙飛虜滅。

Python 有個很不錯的模組，名為 cmd。cmd 提供了一個 Cmd 類別，讀者可依此 Cmd 類別自建子類別，要寫出簡單的命令列解譯程式也不是什麼難事。下列的範例是書中至今最具份量的程式，量雖多，但實作的精神只要把握住，任何疑點都能迎刃而解。

我們的任務是把名字和電話號碼記錄下來，並允許用戶以互動式的介面來處理資料庫的內容，此外，還要能提供錯誤檢查和援助說明的功能以增加親和力。下列的範例就是很具親和力的互動式程式：

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

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

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

我們可以得到某個特定命令的說明文字：

```
Monty's Friends: help find                  # 和 help.find() 成員函式比較
Find an entry (specify a name)
```

我們可以很方便的處理資料庫的內容：

```
Monty's Friends: add larry                          # 我們可以增加項目
Enter Phone Number for larry: 555-1216
Monty's Friends: add                                # 如果名稱沒有指定
Enter Name: tom                                     # 程式會要求輸入
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                       # 儲存
Monty's Friends: ^D                                 # 跳離
```

更好的是如果程式重新啟動，先前輸入的資料都可以再取出來：

```
% python rolo.py                                    # 重新啟動
Monty's Friends: list                               # 預設不做列印
Monty's Friends: load myNames                       # 載入
Monty's Friends: list
==================================
             larry : 555-1216
              tom: 555-1000
==================================
```

整個互動式解譯程式的機能多數都由 Cmd 類別提供。特別的地方是，prompt 屬性必需做設定，而且新增的成員函式要以 do_ 和 help_ 開頭。do_ 成員函式必需接收一個單一引數，而 do_ 之後的名稱才是命令的名稱。一旦呼叫 cmdloop() 成員函式，類別 Cmd 就會接手。請好好研究一下下列的程式碼，暫且命名為 rolo.py 吧。讀的時候，一次看一個成員函式，然後和先前列出來的輸出範例做比較：

```
#!/usr/bin/env python
# 互動式資料庫

import string, sys, pickle, cmd

class Rolodex(cmd.Cmd):

    def __init__(self):
        cmd.Cmd.__init__(self)                # 設定基底類別的初值
        self.prompt = "Monty's Friends: "     # 更改提示符號
        self.people = {}                      # 一開始沒有資料

    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             # 增加各人的電話

    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()            # key 是名稱
        if names == []: return                # 如果沒有名稱就跳離
        names.sort()                          # 我們要以字母順序排列
        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)      # 注意到這樣做
                                                 # 會覆蓋掉任何已存在的人的目錄
if __name__ == '__main__':                       # 這樣的話
    rolo = Rolodex()                             # 這個模組也可以讓別支程式匯入
    rolo.cmdloop()
```

所以囉！add 和 find 命令所用的 people 實體變數，其實不過是名字和電話號碼間簡單的對映（mapping）關係。命令就是 do_ 開頭的成員函式，而命令的文字說明都是由 help_ 開頭。最後，load 和 save 命令用到了 pickle 模組，第十章就會提到。

這個範例說明了利用 Python 現有的工具模組就可以成就如此多的偉業。cmd 模組負責提示符號、援助系統以及分析輸入資料。pickle 則負責載入和儲存資料。我們需要撰寫的部份，就只是和任務有關的部份而已。

學習焦點：Cmd 類別

為了瞭解 Cmd 類別如何運作，請詳讀 Python 程式庫手冊有關 cmd 模組的部份。

Cmd 類別以 onecmd() 成員函式完成了大多數我們感興趣的工作，一旦有人輸入一列文字，onecmd() 就會被呼叫。onecmd() 會負責把第一個字的涵義找出來，看是否合於某個命令。接著，onecmd() 會搜尋 Cmd 子類別的實體物件，看看是否有對應的屬性存在。例如，如果下達的命令列是 "find tom"，onecmd() 要找的就是 do_find。如果 onecmd() 能找得屬性，onecmd() 就會以命令列的引數（指 tom）來呼叫該命令（屬性），然後把結果傳回來。同樣的，對 do_help() 成員函式也是這麼一回事。onecmd() 的程式碼大概像下面這樣：

```
# cmd 類別的成員函式，請見 Lib/cmd.py
def onecmd(self, line):                      # line 是像 "find tom" 的東西
    line = string.strip(line)                # 弄掉空白格
    if not line:                             # 如果沒有東西留下
        line = self.lastcmd                  # 重新執行最後的命令
    else:
        self.lastcmd = line                  # 儲存起來下次再用
    i, n = 0, len(line)
                                             # 下一列找出第一個字的尾端
    while i < n and line[i] in self.identchars: i = i + 1
                                             # 把文字列分割成命令和引數
    cmd, arg = line[:i], string.strip(line[i:])
    if cmd == '':                            # 如果文字列的開頭不是 A-z 的話
        return self.default(line)
    else:                                    # cmd 是 'find', line 是 'tom'
        try:
          func = getattr(self, 'do_' + cmd)  # 搜尋成員函式
        except AttributeError:
          return self.default(line)
        return func(arg)                     # 以文字列剩餘的部分呼叫成員函式
```

問題集

這一章充滿了程式碼，我們建議讀者一定要親自輸入程式碼，試玩看看。然而，如果你真的那麼想寫作業，以下是幾個更具挑戰性的題目：

1. 輸出重導。修改 mygrep.py 命令稿，把輸出寫到命令列最後一個指定的檔案名稱，不要輸出到控制台上。

2. 寫 shell。請使用 cmd 模組的 Cmd 類別以及第八章所提的函式（處理檔案和目錄的部份），寫一個簡單的 shell 命令，能夠接收標準的 Unix 命令（或 DOS 命令）。如 ls（或 dir）可以把當前目錄的內容列示出來，cd 可以變更目錄位置，mv（或 ren）可以移動檔案（變更檔名），而 cp（或 copy）可以複製檔案。

3. 瞭解 map、reduce 和 filter。第一次碰上 map、reduce 和 filter 這種類型的函式，想必大多數人都會覺得自己的智商退步了不少。部份原因是因為這幾個函式的引數也是函式，部份原因是因為即使名稱這麼不起眼，做起事來總是讓人無法看清楚表？。要瞭解這幾個函式最好的辦法就是重寫這幾個函式；我們要讀者分別寫出 map2、reduce2 和 filter2，同樣做 map、reduce 和 filter 能做的事，至少本書提到的功能一樣也不可少：

   a. map2，有兩個引數。第一個引數理應是函式，能接收兩個引數；不然就是 None。第二個引數應該是一個序列。如果第一個引數是函式，函式呼叫的時候，傳遞給該函式的就是序列的每一個元素，呼叫一次傳遞一個，傳回的結果以串列型態表現。如果第一個引數是 None，就把序列轉換成串列，然後把串列傳回來。

   b. reduce2，有兩個引數。第一個引數一定是函式，能接收兩個引數；而第二個引數一定是序列。序列的前兩項元素會當成函式的兩個引數，然後傳回的結果會再度成為函式的第一個引數。因此，序列的第三個元素就順理成章的成為函式的第二個引數，如此循環，直至序列的所有元素都當成引數傳遞給函式為止。最後一個從函式傳回的內容就是傳回給 reduce2 的內容。

   c. filter2，有兩個引數。第一個引數可以是 None，也可以是函式（需得接收兩個引數）。filter2 的第二個引數一定是序列。如果第一個引數是 None，filter2 就傳回序列元素中測試為真的子序列。如果第一個引數是函式，filter2會隨序列元素而呼叫，但是只有經過引數函式測試為真的元素才會傳回。