**2. 使用Python的直譯器   
2.1 如何啟動直譯器**

在Unix之類的作業系統上，如果有安裝的話，Python直譯器通常安裝在 /usr/ bin/python 這個目錄中。你可能需要先在Shell中設定尋找 /usr/ bin 目錄，這樣你才可以在Shell中打入以下的指令來啟動Python直譯器 python

**2.1.1 參數的傳遞**

如果interpreter認識sys的話(譯：可用“import sys”指令)，script的檔案名及附加傳入的參數都會被紀錄在 sys.argv 這個變數並並傳給script來使用。sys.argv 是一列的字串，長度至少為1，如果你什麼檔案或參數都沒傳，sys.argv[0]就是一個空字串。如果script的名字是 '-' (就是標準輸入的意思)的話， sys.argv[0]就會被設為 '-' 。當使用 **-c** *command* 的話，sys.argv[0] 就被設定為 '-c' 所有的在 **-c** *command*之後的option(例如 –i )都會被當成 sys.argv而被command所處理，所以就不是當作option一樣的來看待了。 **2.1.2 互動模式**

當指令是由tty終端機來傳入時，我們稱之為互動模式(*interactive mode*)。在此模式之下會出現主要的命令提示符號(*primary prompt*)來提示輸入下一個指令，這個primary prompt通常是 "> >> " 。如果是指令是延續上一行的話就會出現secondary prompt符號，這個 *secondary prompt* 就通常是 "... " 。一進入python的互動模式的話直譯器會出現一個歡迎信息，以及版本編號輯版權說明，接下來就是第一個prompt。如下所示：

python  
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06) [GCC 2.8.1] on sunos5  
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam  
>>>

當你輸入需要多行的結構時，直譯器就會自動會出現延續上一行的prompt符號，下面的例子是 if 敘述的情況：

>>> the\_world\_is\_flat = 1  
>>> if the\_world\_is\_flat:  
... print "Be careful not to fall off!"  
...   
Be careful not to fall off!

**2.2 直譯器及其周邊環境**

**2.2.1 程式錯誤處理**

當有錯誤產生時，直譯器就會在螢幕印出錯誤的信息以及stack trace的所有資料。在互動模式之下，印完資料之後會再印出prompt來。如果輸入是來自於檔案的話，在出現錯誤的情況下直譯器在印出stack trace之後程式會以nonzero exit 的狀態結束。(此處討論不包含已經由 try 敘述及 except 子句處理外的狀況(Exceptions))。有些的程式錯誤是沒有辦法挽救並且會造成nonzero exit的結束情況，這常常是內部的不一致或是某種running out of memory所造成。所有的錯誤信息都會被寫入至標準error stream之中，正常的程式執行的輸出則會寫入到標準(standard output)輸出之中。如果在primary或是secondary prompte之下打入中斷字元(通常是 Control-C 或是 DEL)，這會造成輸入的中斷並且會回到prompt之下。在指令執行之中打入中斷字元則會引起 KeyboardInterrupt 的exception，而這是可以在 try 敘述中處理的。

**2.2.2 執行Python腳本(script)**

在BSD之類的Unix 系統上，我們可以在script的最前面加入以下的敘述(類似shell script)，並改變檔案屬性為可執行：

#! /usr/bin/env python

如此script就會變成可執行檔，可以直接執行(假設Python的直譯器是在user的 $PATH) 變數之中)。"#!" 這兩個字必須在script檔案的最前面。值得一提的是 "#" 在Python之中也當作注解(comment)部分開始的符號。

**2.2.3 互動式啟動檔(startup file)**

當你使用互動模式的時候，如果可以在每次直譯器要啟動時先執行一些命令的話將是很有用的。要達成如此功能，你可以設定一個檔案名稱給環境變數$PYTHONSTARTUP，這個檔案可以包含你想要在啟動時執行的命令，類似 .profile 在Unix shell中的用法。

這個啟動檔(startup file)只有對在互動模式下有效，如果你用Python讀入script時就沒有用，在當/dev/tty 是命令的輸入來源也沒有用(其他的情況與互動模式相類似)。這個startup file所執行命令的命名空間是與其他互動模式下輸入的指令相同，所以在startup file內定義或是import的物件，在之後的互動模式指令中都是可以直接使用的。你也可以在這個startup file中改變 sys.ps1 及 sys.ps2，如此就可以改變你的primary prompt及secondary prompt。

如果你在你的startup file中想要使用另外的在目前目錄的startup file，你只需要在主要startup file (global start-up file)寫入 "if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')"。如果你想要在你的script之中使用startup file的話，你必須在你的script中寫入：

import os  
filename = os.environ.get('PYTHONSTARTUP')  
if filename and os.path.isfile(filename):  
 execfile(filename)

# 3. An Informal Introduction to Python

注釋(comment)在Python中是以" #" 之後的東西都是注釋(譯：跟Perl一樣)。注釋可以自成一行，也可以跟在空白字元或是程式碼的後面。但是如果" #"是在字串常數(string literal)之中的話，就不代表注釋的意義，而只是一個普通字元罷了。

底下是一些例子：

# this is the first comment  
SPAM = 1 # and this is the second comment  
 # ... and now a third!  
STRING = "# This is not a comment."

# 3.1 把Python當作計算機來用

現在我們來試一試一些簡單的Python指令吧。請先啟動Python的直譯器並且等待primary prompt( " >>> " )的出現。

## 3.1.1 數字

直譯器就好像一個計算機一樣：你可以打入一個表示式(expression)，然後直譯器會把這個expression的執行結果秀出來。Expression的語法都很簡單直接，一般的運算符號 +, -, \* 以及 / 的用法就跟其他的程式語言(像是Pascal或C)一樣。你也可以用括號 "( )" 來表示運算執行的先後次序。例子如下：

>>> 2+2  
4  
>>> # This is a comment  
... 2+2  
4  
>>> 2+2 # and a comment on the same line as code  
4  
>>> (50-5\*6)/4  
5  
>>> # Integer division returns the floor:  
... 7/3  
2  
>>> 7/-3  
-3

跟C語言一樣，等於符號 ("=") 其實是表示設定某個值給一個變數的意思。雖然設定 ("=") 運算本身是有結果值的，但是直譯器並不會輸出其結果來。

>>> width = 20  
>>> height = 5\*9  
>>> width \* height  
900

一個值是可以同時設給許多變數的：

>>> x = y = z = 0 # Zero x, y and z  
>>> x  
0  
>>> y  
0  
>>> z  
0

浮點數的運算在Python裡面也是支援的，如果整數與浮點數(帶小數點或e的數)進行運算的話，整數部分會先轉換(convert)成浮點數再進行運算。

>>> 4 \* 2.5 / 3.3  
3.0303030303  
>>> 7.0 / 2  
3.5

甚至連複數的運算也支援喔，只需要把虛數部分加上 "j" 或是 " J"在其後就可以了。如果實部不為零的話，複數的寫法就寫成 "(real+ imagj)"。或者我們也可以用函數的方式來表示複數為"complex(real, imag)"的形式。

>>> 1j \* 1J  
(-1+0j)  
>>> 1j \* complex(0,1)  
(-1+0j)  
>>> 3+1j\*3  
(3+3j)  
>>> (3+1j)\*3  
(9+3j)  
>>> (1+2j)/(1+1j)  
(1.5+0.5j)

複數的虛數部分及實數部分的值都是以浮點數(float point numbers)來表示的，如果 z 代表一個複數的話，你可以很輕易的用 z.real 以及 z.imag 得到一個複數的實數部分及虛數部分。

>>> a=1.5+0.5j  
>>> a.real  
1.5  
>>> a.imag  
0.5

複數沒有辦法直接用(float(), int() 或是long())轉換成浮點數或是整數。事實上，複數沒有直接對應的實數，你必須用abs(z) 來得到 z 的magnitude(以浮點數表示)，或是如上所述用z.real直接得到其實數部分。

>>> a=1.5+0.5j  
>>> float(a)  
Traceback (innermost last):  
 File "<stdin>", line 1, in ?  
TypeError: can't convert complex to float; use e.g. abs(z)  
>>> a.real  
1.5  
>>> abs(a)  
1.58113883008

在互動模式之下，最後一個印出來的expression的值會儲存在一個特殊變數 "\_ " 之中。這表示，當你用Python的直譯器來當作計算機用的時候，想要連續做運算其實是方便許多的。如下例：

>>> tax = 17.5 / 100  
>>> price = 3.50  
>>> price \* tax  
0.61249999999999993  
>>> price + \_  
4.1124999999999998  
>>> round(\_, 2)  
4.1100000000000003

對於使用者來說， "\_" 這個變數是一個唯讀的變數。你沒有辦法設定一個值給它，當你這樣做的時候，事實上你是重新創造一個同名的變數，但是跟之前系統內建的 "\_" 這個變數是一點關係也沒有的了。

## 3.1.2 字串

除了數字之外，Python也有能力處理字串(string)。字串在Python中有很多種表達方式，它可以放在雙括號””之中，也可以放在單括號’’裡面：

>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn\'t'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> '"Yes," he said.'  
'"Yes," he said.'  
>>> "\"Yes,\" he said."  
'"Yes," he said.'  
>>> '"Isn\'t," she said.'  
'"Isn\'t," she said.'

字串常數(string literals)是可以跨越多行的，其表示方法有很多。如果要換行的話可以用”\”符號來表示之。如下例：

hello = "This is a rather long string containing\n\  
several lines of text just as you would do in C.\n\  
 Note that whitespace at the beginning of the line is\  
 significant.\n"  
print hello

這個例子會印出以下的結果：

This is a rather long string containing  
several lines of text just as you would do in C.  
 Note that whitespace at the beginning of the line is significant.

你也可以用成對的三個雙引號(""")或單引號(''')來表示字串。在此情況下你所打入的ENTER就會直接被解讀為換行符號而不需要再用\n了。

print """  
Usage: thingy [OPTIONS]   
 -h Display this usage message  
 -H hostname Hostname to connect to  
"""

這個例子會印出以下的結果：

Usage: thingy [OPTIONS]   
 -h Display this usage message  
 -H hostname Hostname to connect to

如果你打入的expression是字串的運算，運算的結果的同樣的會由直譯器顯示出來，而且顯示的方式跟你直接打入字串常數(string literals)是一樣的：會在引號之中，所有有escape character “\”表示的字元都會依樣的顯示出來。如果字串本身包含有單引號，整個字串就會用雙引號括起來，要不然就會只用單引號來把整個字串括起來。(如果你使用print這個敘述(statement)來印出字串的話，螢幕的輸出就不會有引號出現，而且字串中的escape character (\”表示的特殊字元)都會顯示出其所代表的意義來。)字串可以用 + 這個運算元來相加 (連接起來)，或是用 \* 這個運算元來重複之。請看例子：

>>> word = 'Help' + 'A'  
>>> word  
'HelpA'  
>>> '<' + word\*5 + '>'  
'<HelpAHelpAHelpAHelpAHelpA>'

如果你把兩個字串常數放在一起，它們自動就會相加起來。所以，上面的例子的第一行也可以寫作 "word = 'Help' 'A'" 。不過這個方法只適用於兩個字串常數的相加，其他情況就不適合了。請看例子：

>>> import string  
>>> 'str' 'ing' # <- This is ok  
'string'  
>>> string.strip('str') + 'ing' # <- This is ok  
'string'  
>>> string.strip('str') 'ing' # <- This is invalid  
 File "<stdin>", line 1  
 string.strip('str') 'ing'  
 ^  
SyntaxError: invalid syntax

如同在C語言一樣，字串是有標記(subscript(index))的，第一個字元的標記(subscript(index))就是0。在Python中沒有另外一個字元character資料型態，一個字元就是一個長度為1的字串。就像是在Icon語言一樣，字串是可以用其subscript(index)來切出( *slice notation* )其中的一部份的，其語法為 ""。

>>> word[4]  
'A'  
>>> word[0:2]  
'He'  
>>> word[2:4]  
'lp'

與C不同的是，Python的字串是不可改變的(immutable)，如果你想要改變其中的一個字元或是一個部份(slice)，你會得到一個錯誤的信息：

>>> word[0] = 'x'  
Traceback (innermost last):  
 File "<stdin>", line 1, in ?  
TypeError: object doesn't support item assignment  
>>> word[:-1] = 'Splat'  
Traceback (innermost last):  
 File "<stdin>", line 1, in ?  
TypeError: object doesn't support slice assignment

但是你可以任意使用一個字串的一個字元或是一個部份(slice)來創造出另一個字串，這是完全可行的：

>>> 'x' + word[1:]  
'xelpA'  
>>> 'Splat' + word[-1:]  
'SplatA'

當你用字串切片(string slice)的語法時，可以使用其預定(default)的subscript(index)值，這是很方便的。第一個subscript(index)的預設值是0，第二個subscript(index)的預設值則是這個字串的整體長度。

>>> word[:2] # The first two characters  
'He'  
>>> word[2:] # All but the first two characters  
'lpA'

所以， s[:i] + s[i:] 會恰好等於 s 。你可以想一想為什麼：

>>> word[:2] + word[2:]  
'HelpA'  
>>> word[:3] + word[3:]  
'HelpA'

如果你用一些奇怪的index來切割字串，Python直譯器也都處理的很好：如果第二個index太大的話就自動代換為字串的長度，如果第二個index比第一個index還要小的話就自動傳回一個空字串。

>>> word[1:100]  
'elpA'  
>>> word[10:]  
''  
>>> word[2:1]  
''

字串的index甚至可以是負數，若是負數的話，就必須從字串的尾巴開始算起。如下例：

>>> word[-1] # The last character  
'A'  
>>> word[-2] # The last-but-one character  
'p'  
>>> word[-2:] # The last two characters  
'pA'  
>>> word[:-2] # All but the last two characters  
'Hel'

但是 -0 事實上是等於 0 ，所以不會從尾巴開始算起。

>>> word[-0] # (since -0 equals 0)  
'H'

如果負數index超過字串的範圍的話，就自動只會到最大可能的範圍，但是如果不是切割一部份的話就會造成錯誤的情形：

>>> word[-100:]  
'HelpA'  
>>> word[-10] # error  
Traceback (innermost last):  
 File "<stdin>", line 1  
IndexError: string index out of range

最好避免錯誤的方法是把index看成是指向字元及字元間位置的指標，字串的最開頭是0，字串的結尾處就是字串的長度。如下圖所示：

+---+---+---+---+---+   
 | H | e | l | p | A |  
 +---+---+---+---+---+   
 0 1 2 3 4 5   
-5 -4 -3 -2 -1

上圖的數字部分第一行代表的是正數的index，由0到字串的長度，第二行代表的是負數的index。字串的切片(slice)很容易就可以看出來，就是兩個index之間的所有字元組合成的字串囉。對於正數的index來說，如果兩個index都在範圍之內，字串的切片(slice)的長度就正好是其兩個index相減的結果。舉例來說 word[1:3] 的長度就正好是 2。

Python內建的 len() 函式可以幫助我們得到字串的長度值。

>>> s = 'supercalifragilisticexpialidocious'  
>>> len(s)  
34

## 3.1.4 列(List)

Python的list可以寫成一串由逗號分開的值(東西)，然後用角括號括起來便成。放在list裡的東西不需要是同一個資料型態

>>> a = ['spam', 'eggs', 100, 1234]  
>>> a  
['spam', 'eggs', 100, 1234]

跟字串的index用法相同，list的index也由0開始，同樣你可以用index來切割lists、組合兩個list等等：

>>> a[0]  
'spam'  
>>> a[3]  
1234  
>>> a[-2]  
100  
>>> a[1:-1]  
['eggs', 100]  
>>> a[:2] + ['bacon', 2\*2]  
['spam', 'eggs', 'bacon', 4]  
>>> 3\*a[:3] + ['Boe!']  
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']

與字串不相同的是，字串的個別字元是不可變動的( *immutable* )，但是list的個別成員是可以自由改變的。

>>> a  
['spam', 'eggs', 100, 1234]  
>>> a[2] = a[2] + 23  
>>> a  
['spam', 'eggs', 123, 1234]

你也可以設定一個值或是一個list給一個list的切割部分(slice)，但是這樣的結果會改變整個list的長度：

>>> # Replace some items:  
... a[0:2] = [1, 12]  
>>> a  
[1, 12, 123, 1234]  
>>> # Remove some:  
... a[0:2] = []  
>>> a  
[123, 1234]  
>>> # Insert some:  
... a[1:1] = ['bletch', 'xyzzy']  
>>> a  
[123, 'bletch', 'xyzzy', 1234]  
>>> a[:0] = a # Insert (a copy of) itself at the beginning  
>>> a  
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]

內建的 len() 函式仍然可用在list上面：

>>> len(a)  
8

一個list也可以是另一個list的成員(這叫作巢狀list, nested list)，參考下例：

>>> q = [2, 3]  
>>> p = [1, q, 4]  
>>> len(p)  
3  
>>> p[1]  
[2, 3]  
>>> p[1][0]  
2  
>>> p[1].append('xtra') # See section 5.1  
>>> p  
[1, [2, 3, 'xtra'], 4]  
>>> q  
[2, 3, 'xtra']

注意前一個例子， p[1] 以及 q 事實上指得是同一個物件。我們在之後還會再討論物件的語法(*object semantics*)。

# 3.2 邁向程式設計的第一步

當然Python能做比二加二更有用更複雜的事，例如說，我們可以寫一個程式來印出費氏數列(*the Fibonacci series*)來：

>>> # Fibonacci series:  
... # the sum of two elements defines the next  
... a, b = 0, 1  
>>> while b < 10:  
... print b  
... a, b = b, a+b  
...   
1  
1  
2  
3  
5  
8

這個範例告訴了我們很多新的事情：

* 程式的第一行是一個多重設定( *multiple assignment* )：兩個變數 a 以及 b 同時都設定了新的值0 與 1。程式的最後一行再次使用這個技巧，這次在設定符號(等號)的右邊我們使用了expression，所有在右邊的expression會先求得其值(evaluate)然後才進行設定(assign)的動作。對於在右邊的expression來說，其evaluate的次序則是由左至右的。
* 在 while 迴圈中，只要條件符合(在這裡是 b < 10 )，這個while 迴圈就會一直執行。與C相同的是，對Python而言只要是非零的整數都代表在決定true/false的情況下都代表true，0則代表false。我們也可以在迴圈的條件的地方放入字串或是一個list，只要這個字串或list的長度不是零就代表true，若是空字串或空的list就代表false。在這個例子裡，我們比較兩個值的大小。比較的運算元與C是完全相同的：< (小於), > (大於), == (等於), <= (小於或等於), >= (大於或等於) 以及 != (不等於)。
* 在迴圈中的執行部分是*縮排*的：縮排在Python中是表示一群敘述的方法(way of grouping statements)。Python沒有提供夠聰明的行排版機制，所以每個要縮排的行你都得打入空白鍵或是tab鍵來縮排。實際的工作環境中，你也許會有自己的文字編輯器，大部分的編輯器會自動幫你做縮排的工作。當在互動模式下輸入一個複合的statement時(一個由許多statements組合成的statement)，最後你需要再打入一個空白行(譯：按ENTER鍵)來告訴直譯器這個statement已經完成了(直譯器沒辦法猜你什麼時候完成這個statement)。如果你的statement是屬於同一群(block)的話，你縮排的距離就要是一樣的。
* print 這個敘述會印出一個expression的結果值，這點與我們之前所做的僅僅打入expression是不同的。不同之處在於對字串及多個的expression來說，用 print不會印出字串的引號，也會在多個expression之間印出空白來，這樣會讓結果好看一點。如下所示：

>>> i = 256\*256  
>>> print 'The value of i is', i  
The value of i is 65536

如果不想每次的輸出都換行的話可以在 print 敘述之後加上逗號，如下所示：

>>> a, b = 0, 1  
>>> while b < 1000:  
... print b,  
... a, b = b, a+b  
...   
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

如果最後一行還沒有完成的話，直譯器會在印出prompt之前印出新的一行。

# 4. 更多流程控制的工具

除了我們剛剛介紹的 while 敘述之外，Python也能夠使用大部分其他程式語言使用的流程控制形式。   
4.1 if 敘述

大概最為人所知的 statement 就是 if 敘述了，舉例如下：

>>> x = int(raw\_input("Please enter a number: "))  
>>> if x < 0:  
... x = 0  
... print 'Negative changed to zero'  
... elif x == 0:  
... print 'Zero'  
... elif x == 1:  
... print 'Single'  
... else:  
... print 'More'  
...

elif 的部份可以沒有也可以有很多個， else 部分可以有一個也可以沒有。 `elif' 這個關鍵字是`else if'的簡化，而且有減少過分縮排的效果。 用 if ... elif ... elif ... 這樣的寫法可以來取代在其他一些程式語言中常見的 *switch* 或是 *case* 的寫法。

# 4.2 for 敘述

在Python裡的for敘述的用法與在C裡的用法有所不同。不像是在C中讓使用者決定執行的進度(step)及結束執行的條件，Python的 for 敘述會將一個系列(sequence，像是list或是string)裡所有的成員走遍一次，執行的順序是依照成員在squence裡的順序。以下是一個例子：

>>> # Measure some strings:  
... a = ['cat', 'window', 'defenestrate']  
>>> for x in a:  
... print x, len(x)  
...   
cat 3  
window 6  
defenestrate 12

# 4.3 range() 函式

如果你真的需要一個迴圈執行一定數目的次數的話，你可以使用內建的range() 函式。這個函式會產生一個含有逐步增加數字的list。如下：

>>> range(10)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

在這個函式中的所傳入的參數是代表端點，而且這個端點不在產生的list之中。range(10) 正好產生10個數值，正好是這個list的index是由0到9。我們也可以讓這個產生的list從某個數值開始，或者規定其每次增加的數值為多少(增加值也可以是負數，這個增加值也叫做`step')。

>>> range(5, 10)  
[5, 6, 7, 8, 9]  
>>> range(0, 10, 3)  
[0, 3, 6, 9]  
>>> range(-10, -100, -30)  
[-10, -40, -70]

所以如果我們要循環一次一個sequence的index的話，我們可以用range()配合上len()一起使用：

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']  
>>> for i in range(len(a)):  
... print i, a[i]  
...   
0 Mary  
1 had  
2 a  
3 little  
4 lamb

# 4.4 break及continue 敘述，以及在迴圈中的 else 子句

同在C語言裡一樣，break敘述中斷最靠近的一個for或while 迴圈。continue敘述會中斷目前執行的迴圈，並且執行下一個循環。特別的是，Python的迴圈有一個else子句，這個子句之後的程式碼會在整個迴圈正常結束的時候執行，(對for)迴圈而言指的是list已經到底，對while迴圈而言指的是條件式變成false)。但若是在非正常結束(因為break 敘述)的情況下 else 子句的程式碼就不會執行。底下的例子是一個迴圈，用來找出所有的質數：

>>> for n in range(2, 10):  
... for x in range(2, n):  
... if n % x == 0:  
... print n, 'equals', x, '\*', n/x  
... break  
... else:  
... print n, 'is a prime number'  
...   
2 is a prime number  
3 is a prime number  
4 equals 2 \* 2  
5 is a prime number  
6 equals 2 \* 3  
7 is a prime number  
8 equals 2 \* 4  
9 equals 3 \* 3

# 4.5 pass 敘述

pass 敘述什麼也不做，通常是用在當你的程式的語法上需要有一個敘述，但是卻不需要做任何事的時候。例子如下：

>>> while 1:  
... pass # Busy-wait for keyboard interrupt  
...

# 4.6 定義函式

我們可以定義一個函式，在底下這個函式定義的例子，當我們給定想要印出的範圍，這個函式會印出一個費氏數列來：

>>> def fib(n): # write Fibonacci series up to n  
... "Print a Fibonacci series up to n"  
... a, b = 0, 1  
... while b < n:  
... print b,  
... a, b = b, a+b  
...   
>>> # Now call the function we just defined:  
... fib(2000)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

上例中這個def 關鍵字代表了一個函式的定義(function *definition*)，在def之後必須接著函式的名稱以及一個用括號括起來的一連串的參數。接下來一行之後的程式碼就是函式的主體部分，而且必須是縮排的。函式的程式碼部分的第一個statement可以是一個字串常數(string literal)，這個字串常數會被當作是函式的註解部分而叫做註解字串(documentation string或是 *docstring*)。

*執行*函式的時候會產生一個目前(local)的符號表(system table)，這個表是用來記錄函式中的所有local的變數的。更精確的來說，所有在函式中變數的設定值都會紀錄在這個system table中，所以當你要使用(reference)一個變數時，會先檢查local的system table，然後是整個程式(global)的system table，然後是內建的變數名稱。雖然global變數可以在函式使用(reference)，但是不能在函式之內直接的設定其值(除非是在一個global的statement中建立的)。

當函式被呼叫時，實際傳入的函式參數是會被紀錄在被呼叫函式的local system table裡，參數被傳入時是*以*物件的參考*傳入(call by reference)*，而非物件本身的*值*。當一個函式呼叫另一個函式時，就會呼叫而建立一個新的local system table。

當定義函式的時候，也就在目前所在的system table裡定義了這個函式的名稱。對直譯器來說，這個函式名稱的資料型態是一個使用者自訂的函式。這個函式的值名稱可以被設定給另一個名稱，然後這個新的名稱就可以被當作是一個函式名稱來使用。這個過程就是一個一般的重新命名的機制。

>>> fib  
<function object at 10042ed0>  
>>> f = fib  
>>> f(100)  
1 1 2 3 5 8 13 21 34 55 89

你也許認為 fib 不是一個函式(function)而是一個程序(procedure)。如同在C中一樣，在Python的procedure指的是沒有傳回值的函式(function)。就技術上而言，procedure也是有傳回值的，只是所傳回的是一個Python系統內鍵的值，叫做 None。通常如果只傳回None的話，直譯器不會印出這一個傳回值。但如果你真想看一看它的話，你可以這樣做：

>>> print fib(0)  
None

如果想讓你的函式傳回一個包含費氏數列的list，而不是只印出來的話，其實是很簡單的：

>>> def fib2(n): # return Fibonacci series up to n  
... "Return a list containing the Fibonacci series up to n"  
... result = []  
... a, b = 0, 1  
... while b < n:  
... result.append(b) # see below  
... a, b = b, a+b  
... return result  
...   
>>> f100 = fib2(100) # call it  
>>> f100 # write the result  
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

如同往例，這個例子事實上示範了一些新的Python的特點：

* return敘述使得函式傳回了一個值。如果單單return沒有其他的expression來表示傳回的值時，就表示這是一個從procedure傳回來的寫法(procedure到結束都沒有傳回值也是表示從procedure傳回來)。這種寫法表示傳回值是 None。
* result.append(b)這個敘述表示呼叫了result這個list物件的一個方法(*method*)。Method是一個特別”屬於”某個物件的函式，而且其名稱的形式是obj.methodname。在這個例子裡的append()方法是在list這個資料型態中定義的。這個方法會在list的最後面加入一個新的成員，在這個例子裡也可以寫作 " result = result + [b]" ，效果一樣，但是用方法來寫有效率多了。

# 4.7 定義函式(續)

在定義函式的時候我們可以加入不定數目的參數，加入參數的寫法有三種，是可以混和使用的。

## 4.7.1 預設內定參數值

最好用的一種寫法是，對其中的一個或多個參數給它一個特定的預設值。這樣子的話，當你在呼叫函式時，就可以不用傳入參數，或是傳入較少的參數了。請看下例：

def ask\_ok(prompt, retries=4, complaint='Yes or no, please!'):  
 while 1:  
 ok = raw\_input(prompt)  
 if ok in ('y', 'ye', 'yes'): return 1  
 if ok in ('n', 'no', 'nop', 'nope'): return 0  
 retries = retries - 1  
 if retries < 0: raise IOError, 'refusenik user'  
 print complaint

當你呼叫這個函式時可以用 ask\_ok('Do you really want to quit?') ，或者是 ask\_ok('OK to overwrite the file?', 2) 。

## 4.7.2 關鍵字參數

呼叫函式時也可以使用關鍵字參數，其形式是 " keyword = value" ，底下的這個函式：

def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):  
 print "-- This parrot wouldn't", action,  
 print "if you put", voltage, "Volts through it."  
 print "-- Lovely plumage, the", type  
 print "-- It's", state, "!"

用這些方式呼叫都是正確的：

parrot(1000)  
parrot(action = 'VOOOOOM', voltage = 1000000)  
parrot('a thousand', state = 'pushing up the daisies')  
parrot('a million', 'bereft of life', 'jump')

但是用這些方式都是不正確的：

parrot() # required argument missing  
parrot(voltage=5.0, 'dead') # non-keyword argument following keyword  
parrot(110, voltage=220) # duplicate value for argument  
parrot(actor='John Cleese') # unknown keyword

一般來說，一連串的參數的次序是先有非關鍵字參數(也可以沒有)然後才是關鍵字參數，關鍵字必須是函式定義時所用的參數名稱。這個定義時用的參數名稱有沒有預設值並不重要，但是一個傳入的參數只能有一個值(預設值不算)，如果你已經先用非關鍵字參數給了某個參數一個值，接下來你就不能再用關鍵字參數給它另外的值。底下的例子就違反了這個規則：

>>> def function(a):  
... pass  
...   
>>> function(0, a=0)  
Traceback (innermost last):  
 File "<stdin>", line 1, in ?  
TypeError: keyword parameter redefined

當一個函式定義時參數名稱是以 \*\* name 這種形式定義時，表示這個參數要接受的是一個dictionary(譯：字典，包含許多關鍵字以及值的對應)，這個 dictionary包含許多的關鍵字參數，但是這些關鍵字不能跟其他的參數名稱相同。另外參數也可以用 \*name 這種形式定義(下一小節會解釋)，這種方式定義的參數要接受的是一個 tuple(譯：不可更動的list)，這個tuple 可以接受不限數目的非關鍵字參數( \*name 必須要出現在 \*\*name 之前)。下面的例子就是一個函式定義的範例：

def cheeseshop(kind, \*arguments, \*\*keywords):  
 print "-- Do you have any", kind, '?'  
 print "-- I'm sorry, we're all out of", kind  
 for arg in arguments: print arg  
 print '-'\*40  
 for kw in keywords.keys(): print kw, ':', keywords[kw]

要呼叫這個函式，你可以這樣呼叫：

cheeseshop('Limburger', "It's very runny, sir.",  
 "It's really very, VERY runny, sir.",  
 client='John Cleese',  
 shopkeeper='Michael Palin',  
 sketch='Cheese Shop Sketch')

函式執行的結果如下：

-- Do you have any Limburger ?  
-- I'm sorry, we're all out of Limburger  
It's very runny, sir.  
It's really very, VERY runny, sir.  
----------------------------------------  
client : John Cleese  
shopkeeper : Michael Palin  
sketch : Cheese Shop Sketch

## 4.7.3 隨意的參數串

最後我們要介紹最不常見的形式，也就是定義一個函式可以接受任意數目的參數，這些傳入的參數會被放進一個tuple裡面去。在這一個任意數目的參數之前，可以定義沒有或是一個或是多個普通的參數：

def fprintf(file, format, \*args):  
 file.write(format % args)

# 5. 資料結構

這一章討論的內容有些你在之前已經看過，但我們會更深入的討論。另外，我們也會介紹一些新的東西。

# 5.1 列(Lists)(續)

列(list)這個資料型態有一些方法可以使用，底下我們就列出來一些常用的方法：

**append(x)** 在list的尾端加入一個成員，也可以用這個方法來寫 a[len(a):] = [x] 。

**extend(L)** 接受一個新的list的參數，然後把它加入到目前這個list的尾端，也可以寫作 a[len(a):] = L 。

**insert(i, x)** 在某個特定的位置加入一個成員。第一個參數是要加入的位置的index，所以 a.insert(0, x) 會加入在list的最前端，而 a.insert(len(a), x) 會在最後端加入，相等於 a.append(x) 。

**remove(x)** 拿掉第一個其值相等於 x. 的成員。如果整個list都沒有這個成員，那就會得到一個錯誤(error)。

**pop([i])** 從一個list中拿掉某個位置的成員，並且傳回這個被拿掉的成員。如果沒有傳入位置的index的話，a.pop()會傳回這個list的最一個成員，同樣的這個成為會被從這個list之中拿掉。

**index(x)** 傳回第一個其值相等於 x 的成員之位置(index)，如果整個list都沒有這個成員，那就會得到一個錯誤(error)。

**count(x)** 傳回在整個list裡面， x 出現了多少次。

**sort()** 針對list裡面的成員做排序。

**reverse()** 反轉整個list裡面成員的位置。

底下的這個例子使用了大部分的lsit的方法(method)：

>>> a = [66.6, 333, 333, 1, 1234.5]  
>>> print a.count(333), a.count(66.6), a.count('x')  
2 1 0  
>>> a.insert(2, -1)  
>>> a.append(333)  
>>> a  
[66.6, 333, -1, 333, 1, 1234.5, 333]  
>>> a.index(333)  
1  
>>> a.remove(333)  
>>> a  
[66.6, -1, 333, 1, 1234.5, 333]  
>>> a.reverse()  
>>> a  
[333, 1234.5, 1, 333, -1, 66.6]  
>>> a.sort()  
>>> a  
[-1, 1, 66.6, 333, 333, 1234.5]

## 5.1.1 把列(Lists)當作堆積(Stacks)使用

把列(list)當作堆積(stack)來使用是一件容易的事。Stacks的最後一個加入的成員是第一個被取出來的成員(後進先出”last-in, first-out''法則)。要在stack的最頂端加入一個成員可以使用append()，要從stack的最頂端取出一個成員可以用 pop() (不須加入參數)。例子如下：

>>> stack = [3, 4, 5]  
>>> stack.append(6)  
>>> stack.append(7)  
>>> stack  
[3, 4, 5, 6, 7]  
>>> stack.pop()  
7  
>>> stack  
[3, 4, 5, 6]  
>>> stack.pop()  
6  
>>> stack.pop()  
5  
>>> stack  
[3, 4]

## 5.1.2 把列(Lists)當作佇列(Queues)使用

你也可以很方便的拿list來當作佇列(queues)來使用。Queues的特點是第一個加入的成員會是第一個被取出的成員(先進先出``first-in, first-out''法則)。要在queue的後端加入一個成員可以使用append()，要從queue的最前端取出一個成員可以使用 use pop()，記得參數是0。例子如下：

>>> queue = ["Eric", "John", "Michael"]  
>>> queue.append("Terry") # Terry arrives  
>>> queue.append("Graham") # Graham arrives  
>>> queue.pop(0)  
'Eric'  
>>> queue.pop(0)  
'John'  
>>> queue  
['Michael', 'Terry', 'Graham']

# 5.2 del 敘述

del 敘述可以讓你輕鬆的去掉在list當中某一個位置(index)的成員。這個敘述也可以用切割(slice)的方法來去掉某一段的成員(在之前我們必須藉著設定某個slice為空list來達成同樣的效果)。請看下例：

>>> a  
[-1, 1, 66.6, 333, 333, 1234.5]  
>>> del a[0]  
>>> a  
[1, 66.6, 333, 333, 1234.5]  
>>> del a[2:4]  
>>> a  
[1, 66.6, 1234.5]

del 也可以用來去掉整個變數：

>>> del a

如果你在去掉之後還繼續使用這個變數名稱的話，就會得到一個錯誤(除非你之後再設定另外一個值給它)。我們稍後會繼續看到使用 del 的例子。

# 5.3 Tuples(固定有序列)及Sequences(有序列)

我們之前所討論的lists以及字串(strings)有很多的共通點，例如可以用index來定位置，可以切出其中的某一段(slicing)等等。事實上，list及字串都是 *sequence* 這個資料型態的特例。另外一種標準的sequence資料型態：固定有序列(*tuple*)。

一個tuple是由特定數目的值所組成，其成員與成員之間以逗號分開。舉例如下：

>>> t = 12345, 54321, 'hello!'  
>>> t[0]  
12345  
>>> t  
(12345, 54321, 'hello!')  
>>> # Tuples may be nested:  
... u = t, (1, 2, 3, 4, 5)  
>>> u  
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

如同在前面例子所見到的，tuples輸出的結果都會包含在括弧之中。所以，巢狀tuple(tuple之中有tuple)可以被清楚的區分出來。Tuple在輸入的時候可以有括弧也可以沒有，通常我們都會加上括弧(特別是用在在複雜的expression之中)。

Tuples有很多種用途，例如(x, y)座標，從資料庫取出的員工的資料庫記錄等等。Tuples跟字串一樣都是不可改變(immutable)的：我們不能單獨的設定一個tuple裡面的個別成員(雖然我們可以用切割及連結(concaatenation)來達到同樣的效果)。我們也可以做出可以改變成員的tuple來，例如list。空的tuple的表示方法是一對空的括弧，只有一個成員的tuple表示方法是在成員後面加上逗點(不能只是用括弧把一個成員括起來)。雖然有點奇怪，但是蠻有用的。請看例子：

>>> empty = ()  
>>> singleton = 'hello', # <-- note trailing comma  
>>> len(empty)  
0  
>>> len(singleton)  
1  
>>> singleton  
('hello',)

t = 12345, 54321, 'hello!' 這個敘述是一個tuple包裝(*tuple packing*)的例子：12345, 54321以及'hello!'這三個值都被包裝放在一個tuple裡面了。我們也可以使用相反的操作方式，例如：

>>> x, y, z = t

這個動作叫做打開sequence( *sequence unpacking* )。Sequence unpacking的動作需要在設定符號左邊有一串的變數，其數目應與右邊sequence的成員數目相同。值得注意的是，多重設定(a, b = 1, 2)其實只是tuple packing以及sequence unpacking的結合罷了！

有一個不太對稱的地方：packing的動作永遠結果是一個tuple，但是unpacking可以針對各種不同的sequence來做。

# 5.4 Dictionaries(字典)

另外一個在Python當中很好用的內建資料型態是字典(*dictionary*)。Dictionary有的時候在別的程式語言裡面也叫做連結記憶(“associative memories'')或者是連結陣列(“associative arrays'')。不同於sequence是由一連串的數字來做index，dictionary用一個特殊的不可改變的(immutable)鑰(*keys*來當作其index。字串及數字都不能被改變，所以都可以來當作dictionary的key。Tuple如果只含有字串，數目字，以及其他tuple的話也可以當作key。如果tuple裡面有包含任何可改變的(mutable)的物件的話(包括直接或間接)，就不能當作key來使用。List不能當作key，因為list的成員可以被改變(你可以用append()以及extend()之類的方法，或是切割(slicing) 或 index 來設定list的個別成員)。

我們把dictionary想像成一個沒有順序的 *key: value* 成對的組合。在dictionary裡面key的值必須是唯一不重複的。最簡單的dictionary就是一對空的中括弧：{}。在中括弧裡面放入由逗號分開的key:value對，就成了dictionary裡面的成員。這也是當dictionary被印到輸出時的標準格式。

我們可以對dictionary做一些事，包括加入一個帶有key的值、或者是用key來找一個特殊的值。我們也可以用del來刪去一對key:value的成員。如果你試圖存一對key:value但是這個key已經被使用了的話，原來的那一個value的值就會被蓋過。如果你想用一個不存在的key來找出某一個成員的話，你會得到一個error。

使用 keys()這一個dictionary的方法我們可以得到一個由所有的key值組成的list，其順序是隨機沒有次序的(如果你想要排序的話，只要針對這一個得到的list來呼叫其sort()方法就可以了)。要檢查某個key是不是存在的話，你可以使用 has\_key() 這一個method來做檢查。

底下是一個有關dictionary的小例子：

>>> tel = {'jack': 4098, 'sape': 4139}  
>>> tel['guido'] = 4127  
>>> tel  
{'sape': 4139, 'guido': 4127, 'jack': 4098}  
>>> tel['jack']  
4098  
>>> del tel['sape']  
>>> tel['irv'] = 4127  
>>> tel  
{'guido': 4127, 'irv': 4127, 'jack': 4098}  
>>> tel.keys()  
['guido', 'irv', 'jack']  
>>> tel.has\_key('guido')  
1

# 5.5 條件(續)

可以用in以及not in來檢查某個值是否出現(或沒出現)在某個sequence裡面。使用is以及is not來檢查兩個物件是否事實上指的是相同的一個物件(這只有跟像是list一樣可變的物件有關)。所有的比較運算的運算優先次序都是一樣的，都比所有的數字運算要來的低。

比較運算也可以用and及or等boolean運算來連結起來，其比較的結果(或其他boolean運算的結果)也可以用not來得到相反(negated)的結果。這些運算裡，not有最高的優先次序，or的優先次序最低，但是所有的優先次序都比比較運算來的低。所以A and not B or C 其實相等於 (A and (not B)) or C 。當然，最好適時的使用括弧來幫助你表達你真正想要的組合。

# 5.6 Sequences(有序列)及其他資料型態的比較

Sequence物件可以和其他的相同資料型態的sequence物件相比較，其比較方法是依照所謂的*lexicographical*順序(lexicographical ordering)。首先是兩個sequence的第一個成員互相比較，如果比較有大小之別的話就此決定其相對大小，若是相等的話就再比較下一個成員的大小，餘此類推直到sequence的結束。如果兩個要相比較的成員本身也是一個sequence的話，同樣的條件可以繼續遞迴的使用在這兩個sequence之上。如果這兩個sequence的所有成員都相等的話，我們就說這兩個成員是相等的。如果某一個sequence是另一個sequence的一部份的話，較短的那一個sequence就是較小的。字串的Lexicographical順序用的是個別字元的ASCII碼的順序。底下是一些同一資料型態的sequence的比較例子：

(1, 2, 3) < (1, 2, 4)  
[1, 2, 3] < [1, 2, 4]  
'ABC' < 'C' < 'Pascal' < 'Python'  
(1, 2, 3, 4) < (1, 2, 4)  
(1, 2) < (1, 2, -1)  
(1, 2, 3) == (1.0, 2.0, 3.0)  
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)

值得注意的是，我們也可以比較兩個不同資料型態的物件，而其結果是依其資料型態的名稱來決定的。所以所有的list都比字串還要來的小(因為list小於string)，所有的string也都比tuple還要小。至於數值的資料型態則是由其數值大小來決定其大小，所以0等於0.0的，其餘按此類推。

# 6. 模組

如果你離開Python直譯器然後又再打開Python直譯器的話，你會發現你剛才定義的一些東西(函式或變數)都不再存在了。所以說，如果你真的想寫一些比較大型的程式的話，你可能需要有一個文字編輯器來編輯一個檔案，然後再讓Python直譯器來將這個檔案當作輸入(input)來處理。這個過程就是寫腳本(*script*)的過程。如果你的程式繼續的越來越長的話，你也許會想要把你的程式分成幾個小的檔案，這樣比較方便來維護你的程式。你也許也會希望有一些方便的函式可以讓你自由的用在好幾個程式之中，你又不想要copy這些函式的定義在每個程式之中。

要達到以上的這些目的，Python有一個將定義放在檔案中的方法，你可以之後再在你的script或是互動模式的程式下使用這些存好的定義。這樣的檔案就叫做模組(*module*)。存在於module之中的定義可以用*imported*放入在其他的module或是主要的*main* module之中。(main module是一組你可以在script的最高一級(top level)部分使用，或是在互動模式中使用的變數)。

一個module就是一個包含有Python的定義及敘述的檔案，檔案的名稱就是module的名稱加上延伸檔名.py在後面。在一個module裡面，module的名字(是一個字串)會存在 \_\_name\_\_這個變數裡面並當作全域變數(global variable)使用。舉例來說，你可以用你喜歡的文字編輯器打入以下的內容，並將這個檔案存在目前的目錄，並取名為fibo.py ：

# Fibonacci numbers module  
  
def fib(n): # write Fibonacci series up to n  
 a, b = 0, 1  
 while b < n:  
 print b,  
 a, b = b, a+b  
  
def fib2(n): # return Fibonacci series up to n  
 result = []  
 a, b = 0, 1  
 while b < n:  
 result.append(b)  
 a, b = b, a+b  
 return result

現在你可以進入Python的直譯器裡面並且import你剛剛建立的module，其方法如下：

>>> import fibo

這個命令並不會使得所有的fibo裡面的函式名稱都寫入目前的符號表(symbol table)裡面，但是會把fibo這個module的名字寫在symbol table裡面。所以我們現在就可以使用module的名字來呼叫這些我們之前所定義的函式了：

>>> fibo.fib(1000)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987  
>>> fibo.fib2(100)  
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]  
>>> fibo.\_\_name\_\_  
'fibo'

如果你真的想要只用函式名稱的話，你可以把這些函式名稱設定到另一個local變數去(可以就是函式的名稱)：

>>> fib = fibo.fib  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

# 6.1 模組(續)

一個module裡面除了放函式的定義之外也可以放可執行的敘述(statement)。這些statement的功用在於初始化(initialize)這個module。這些statement也只有在module*第一次*被import的時候才會被執行。

每一個模組都有其自己的符號表(symbol table)，這個symbol table也就成為在module裡面所定義的函式的全域變數(global variables)。所以說，寫module的人就可以自由的使用這些global variable而不需要擔心會跟module的使用者的global variable有所衝突。從另一方面來說，如果你知道自己在做什麼的話，你也可以跟使用函式一樣的使用module裡面的global variable。其語法為 modname.itemname.

Module可以被import到其他的module裡面。習慣上我們會把所有的import的敘述都放在module(或者是script)的最開頭。這樣的話這個被import的module的名稱就會被放在目前這個module的global symbol table裡。有一個變形的方式可以直接import module裡面的變數或函式的名稱進入symbol table裡面。舉例如下：

>>> from fibo import fib, fib2  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

這樣做的話並不會使得module的名字被放在目前的symbol table裡面。(所以在上面的例子裡 fibo 是沒有被定義的)。

我們甚至可以一次將所有的在module裡面所定義的名稱都import進來：

>>> from fibo import \*  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

這一個寫法會import所有的定義的名稱，除了以底線 ( \_ ) 開頭的之外。

## 6.1.1 尋找模組的路徑

當你import一個叫做spam的module時，直譯器會先在目前的目錄尋找一個叫做spam.py的檔案，如果沒找到，會再依據定義在$PYTHONPATH (一個環境變數)裡面的所有路徑來找。$PYTHONPATH的語法與設定方法與$PATH是一樣的，也就是一連串的目錄路徑的名稱。如果你沒有設定$PYTHONPATH，或是在這些目錄當中也找不到的話，直譯器會繼續在一個安裝時預設的目錄來找，在Unix的機器上，通常這個目錄是.:/usr/local/lib/python。

事實上，module的搜尋路徑是依照存在sys.path這一個變數中的一個許多路徑名稱組成的list。這個變數當在Python直譯器啟動時，會從輸入的script(或目前的目錄)、$PYTHONPATH、以及安裝時設定的預設目錄來讀取所有的目錄。如果你知道自己在做什麼的話，你可以修改這個變數來改變直譯器尋找module的路徑。

## 6.1.2 “編譯過的”( “Compiled'') Python檔案

對於一些小的程式來說，如果使用很多標準的module，又想加速啟動的過程，你就可以用編譯過的Python檔案。比如你要找spam.py，如果在你找到這個檔案的目錄裡ey4又有一個叫做spam.pyc的檔案的話，這就表示spam這個module有一個已經二元編譯過的(``byte-compiled'')的版本可以使用。在 spam.pyc 裡面也會記錄用來創造它的spam.py上一次被修改的時間，如果.pyc裡面所儲存的時間與最新版本的.py的修改時間不符合的話，.pyc檔案就不會被使用。

一般來說，你不需要做任何事來創造一個spam.pyc檔案。當你成功的編譯一個 spam.py 檔時，自動的spam.pyc檔就會寫入在同一個目錄裡。如果這個過程裡有問題的話，系統不會當這是個錯誤情況(error)。相反的，如果寫入的檔案沒有完全成功的寫入的話，這個檔案只會被認為是不正確的而忽視其存在。spam.pyc檔案的內容是獨立於作業系統平台的，所以一個 Python module 的目錄是可以被在各種不同架構下的多台機器所共享的。

這裡有一些給專家們的秘訣：

* 當使用 **-O** 這個選項啟動Python直譯器時，直譯器產生會最佳化程式碼(optimized code)，並存在 .pyo 檔案裡。這個最佳化程式碼目前並沒有太多功能，它只是簡單的拿掉所有的 assert 敘述以及 SET\_LINENO 指令。當你使用 **-O** 這個選項時， *所有的* 二元碼(bytecode)都會被最佳化，所有的 .pyc 檔都會被忽略，所有的 .py 檔案都會被編譯成最佳化的二元碼。
* 讀.pyc以及.pyo檔案並不會比讀.py file，唯一的差距是在當被導入(load)時的速度有差別。
* 當你在命令列(command line)使用script的名稱來執行它的話，並不會造成二元碼被寫到.pyc或是.pyo。所以，你可以把這個script寫成一個module，然後再用一個小的啟動的script來import這個module。這樣可以減少啟動的時間。事實上，你也可以直接從命令列啟動.pyc 或是 .pyo 檔案。
* 你也可以把 spam.pyc (或是 spam.pyo ，如果你用了 **-O** 的話) 放在沒有 spam.py 的目錄裡。這樣子，當你給別人你的程式庫時，你可以給他們比較難用逆向工程(reverse engineer)破解的程式。
* 你可以用 compileall 這個module來將某個目錄裡面的所有module都便成.pyc檔案(或者是.pyo檔案，如果你用了 **-O** )。

# 6.2 標準模組

Python包含有一個標準模組的程式庫。有些標準模組已經內建在直譯器裡面，這些模組讓我們可以使用那些不在Python語言本身的一些功能，不管是為了效率或是要使用作業系統的資源(例如system call)。有些module是在設定時的選項，比如說。有一個module特別值得我們好好注意： sys 。這個module在每一個Python直譯器裡面都有，其中有兩個變數sys.ps1 以及 sys.ps2 是用來設定primary prompt 以及secondary prompt的：

>>> import sys  
>>> sys.ps1  
'>>> '  
>>> sys.ps2  
'... '  
>>> sys.ps1 = 'C> '  
C> print 'Yuck!'  
Yuck!  
C>

這兩個變數只有在當你在互動模式下啟動直譯器時才有定義。

sys.path這個變數是一個許多目錄路徑組成的list，裡面的目錄路徑就是直譯器尋找module的路徑。這個變數裡面的路徑都是從環境變數$PYTHONPATH裡面複製的，或者當$PYTHONPATH沒有設定時，就會使用預設值。你也可以用一般使用list的方法來修改之。例如：

>>> import sys  
>>> sys.path.append('/ufs/guido/lib/python')

# 6.3 dir() 函式

內建的dir()函式主要是用來找出某個module裡面所定義的所有名稱。其傳回值是一串經過排序了的字串list：

>>> import fibo, sys  
>>> dir(fibo)  
['\_\_name\_\_', 'fib', 'fib2']  
>>> dir(sys)  
['\_\_name\_\_', 'argv', 'builtin\_module\_names', 'copyright', 'exit',  
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',  
'stderr', 'stdin', 'stdout', 'version']

如果沒有傳入參數的話，dir()會列出所有你目前已經定義的名稱：

>>> a = [1, 2, 3, 4, 5]  
>>> import fibo, sys  
>>> fib = fibo.fib  
>>> dir()  
['\_\_name\_\_', 'a', 'fib', 'fibo', 'sys']

注意這裡的名稱是指所有類型的名稱：包括變數，函式，以及module等等。

dir()並沒有列出所有內建的函式及變數的名稱。如果你真想要列出來的話，它們都定義在 \_\_builtin\_\_ 這個標準module裡面：

>>> import \_\_builtin\_\_  
>>> dir(\_\_builtin\_\_)  
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',  
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',  
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',  
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',  
'ZeroDivisionError', '\_\_name\_\_', 'abs', 'apply', 'chr', 'cmp', 'coerce',  
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',  
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',  
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw\_input',  
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']

# 6.4 Packages(包裝)

Package是一種用點號表示模組名稱(“dotted module names'')的組織Python模組(module)命名空間的方法。舉例來說：如果module的名稱是A.B表示是在"A"這個package裡面的一個名稱為"B"的module。如同使用module使得其他寫module的不用擔心別人的global variable命名的問題，使用這種帶點號的module名稱也使得寫多個module的package的人不用擔心所用的module名稱會和別人有所重複。

現在假設你要設計一組的module(就是設計一個package)，這個package是用來標準化的處理聲音檔案以及聲音的資料的。由於聲音檔的格式有很多(通常是由其延伸檔名來辨別，例如.wav, .aiff, .au) 等格式)，你也許需要一個隨時會增加新module的package來處理新的聲音檔格式。由於你可能想對聲音資料做各種不同的處理(例如混音、加迴聲、加入平衡方程式，加入人工音響效果等等)，所以你還需要寫一些module來專門做這些處理。底下這個架構可能是你的package所需要的(用檔案階層系統來表示)：

Sound/ Top-level package  
 \_\_init\_\_.py Initialize the sound package  
 Formats/ Subpackage for file format conversions  
 \_\_init\_\_.py  
 wavread.py  
 wavwrite.py  
 aiffread.py  
 aiffwrite.py  
 auread.py  
 auwrite.py  
 ...  
 Effects/ Subpackage for sound effects  
 \_\_init\_\_.py  
 echo.py  
 surround.py  
 reverse.py  
 ...  
 Filters/ Subpackage for filters  
 \_\_init\_\_.py  
 equalizer.py  
 vocoder.py  
 karaoke.py  
 ...

為使Python能把這個目錄架構當作是一個package，上面的 \_\_init\_\_.py 這個檔是必須要的。這是為了要避免有些檔案目錄的名字是很普通的名字(例如 " string" )，這會讓直譯器誤認正確的module名稱而找不到在搜尋路徑中的module。在最簡單的例子裡，\_\_init\_\_.py可以是一個空的檔案。但是你也可以讓這個檔來做一些package初始化的動作，或者設定 \_\_all\_\_ 這個變數(稍後會再提)。

使用package的人可以從package裡使用(import)某一個module，例如：

import Sound.Effects.echo

上面的程式碼會導入(load) Sound.Effects.echo這個module。如果你要使用這個module，你必須使用完整的名稱，例如：

Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)

另外一個導入在package中的某個module的方法是：

from Sound.Effects import echo

同樣的，這會導入echo這個module。不同的是，當你使用這個module的時候你就不用在寫前面package的名稱了。請看以下使用這個module的例子：

echo.echofilter(input, output, delay=0.7, atten=4)

你也可以直接的import某一個在module裡面的函式或變數，如下例：

from Sound.Effects.echo import echofilter

同樣的，這會導入echo這個module，不同的是你現在可以直接的使用echofilter()這個函式了：

echofilter(input, output, delay=0.7, atten=4)

值得注意的是當你使用from package import item這樣的敘述時，你所import的東西可以是一個package中的module(或者是subpackage)，或者是在module裡面所定義的名稱，例如變數、類別或是函式等等。import敘述會先測試是否這個東西真的存在於這個package，如果沒有的話，會假設這是一個module然後試著導入(load)之。如果還失敗的話，就會引發一個ImportError的例外狀況(exception)。相反的是，當你使用import item.subitem.subsubitem這樣的敘述時，除了最後一個東西(item)以外，其餘的都必須是package。最後一個可以是module或是package，但是不能是一個定義在module裡面的類別、成員或函式。

## 6.4.1 從一個Package中Import \*

那如果使用者寫了from Sound.Effects import \*，會造成什麼結果呢？理想狀況下，我們可能會期望會搜尋整個package目錄，然後找出所有的module並且一一的import這些module。不幸的是，在Mac 以及 Windows 平台下，檔案的名稱大小寫並不統一。所以在這些平台之上，我們並無法保證ECHO.PY這個檔案應該被import成 echo, Echo 或 ECHO (例如，Windows 95 有一個惱人的特點，就是會自動把所有的檔案名稱第一個字元大寫)。DOS的 8+3 檔名限制對長的module名稱來說，也是另一個有趣的問題。

所以唯一的解決方法就是package的作者要提供一個明顯的index給用package的人。如果遵守這個習慣的話，當用package的人在import的時候使用 from Sound.Effects import \* 的話，就會去找這個package的 \_\_init\_\_.py 檔案裡面的 \_\_all\_\_ 這個list變數，這個list裡面就會包含所有應該被import進來的module名稱了。身為Package的作者有責任要保持 from package import \* 這個檔案的更新，但是如果package的作者確信沒有人會用 from Sound.Effects import \* 這種寫法的話，也可以不使用這個檔案。舉例來說 Sounds/Effects/\_\_init\_\_.py 這個檔案就可以有下面這樣的程式碼：

\_\_all\_\_ = ["echo", "surround", "reverse"]

這就表示 from Sound.Effects import \* 會從 Sound 這個package 裡面import 這三個module。

如果沒有定義 \_\_all\_\_ 的話，from Sound.Effects import \* 這個敘述就*不會*從Sound.Effects這個package裡面import所有的module進入目前的命名空間(namespace)。唯一能保證的是 Sound.Effects 這個package有被imported進來(可能會執行 \_\_init\_\_.py 裡面的初始化程式碼)，並且這個package裡面所定義的名稱會被import進來。Package裡所定義的名稱包含了在 \_\_init\_\_.py 裡面所定義的名稱(以及所import的module)。當然也包含了在之前用import引進來的module名稱，例如：

import Sound.Effects.echo  
import Sound.Effects.surround  
from Sound.Effects import \*

在這個例子裡，echo以及 surround 這兩個modules 都會被 import進來目前的命名空間(namespace)裡。這是因為當 from...import 這個敘述執行的時候，這兩個module都已經在這個package中有定義了(你也可以用 \_\_all\_\_ 來定義)。

值得注意的是使用import \* 這樣的寫法常常是不被鼓勵的，因為這通常會使得你的程式的可讀性降低。無論如何，在互動模式下這樣做的確會使你減少打太多字的機會，而且有些的module在設計的時候就故意只讓某些特別的名稱可以被使用。

記住，使用from Package import specific\_submodule沒有任何不對的地方。事實上，除非你的module的名字會和其他的名稱衝突，否則這是常被推薦使用的形式。

## 6.4.2 Package內的References(參考)

在package之中的module常常需要彼此互相使用。例如說，surround這個module就有可能會使用到echo這個module裡的東西。事實上，由於這是最常見的，所以import的時候總是會先找自己這的package裡面的module，然後再依照搜尋的路徑來尋找。因此surround這個module可以使用import echo或是from echo import echofilter就可以了。如果在自己所處的這個package裡面找不到這個要import的module的話，import指令就會在第一級(top-level)的module裡找所指定的名稱。

當一個subpackage是在另一個package裡的話(例如前面的Sound)，沒有其他捷徑可以讓你使用其他在同一個外圍的package裡面的subpackage裡的module，你必須使用完整的名稱來指稱你所要用的package。例如說，如果在 Sound.Filters.vocoder這個module裡面你想要使用在Sound.Effects這個package裡的echo這個module的話，你就要使用 from Sound.Effects import echo 這個敘述。

**8. 程式錯誤與例外(Exceptions)情形**

至此為止，我們都只有稍稍的提到錯誤訊息。但是如果你有試著執行上面的範例的話，你可能注意到，基本上錯誤的情況可以分成兩類：語法錯誤(*syntax errors*)以及例外情況(*exceptions*)。

**8.1 語法錯誤**

語法錯誤也叫做分析時的錯誤(parsing errors)，大概是一般在學Python時最常見到的直譯器所發出來的抱怨：

>>> while 1 print 'Hello world'  
 File "<stdin>", line 1  
 while 1 print 'Hello world'  
 ^  
SyntaxError: invalid syntax

Python分析器(parser)會在印出錯誤的行，並且用一個向上的箭號指出最早發現錯誤的地方，而這個錯誤是發生(至少是被發現)在這個箭號所指的單元(token)*之前*。在我們的例子裡面：錯誤發生在 print 這個關鍵字，因為前面應該有一個 ( " :" )。錯誤信息裡面也包含檔案名稱以及行數，所以你可以很快知道要到哪裡去找錯。

**8.2 例外(Exceptions)情形**

有的時候，甚至當你的語法完全正確時，當你執行程式時仍然會出錯。這種在程式執行階段發生的錯誤叫做例外情形 (*exceptions*)，並且會造成程式致命的終止(無法執行下去)。你待會就會知道在Python裡面要怎樣處理這樣的狀況，但是我們先來看這樣的狀況下會造成什麼錯誤信息：

>>> 10 \* (1/0)  
Traceback (innermost last):  
 File "<stdin>", line 1  
ZeroDivisionError: integer division or modulo  
>>> 4 + spam\*3  
Traceback (innermost last):  
 File "<stdin>", line 1  
NameError: spam  
>>> '2' + 2  
Traceback (innermost last):  
 File "<stdin>", line 1  
TypeError: illegal argument type for built-in operation

在這些錯誤信息的最後一行都是解釋到底發生了什麼事。例外情況(Exception)有很多種類型，類型的名稱也在錯誤信息之中，在上面的例子裡面，exception的類型分別是： ZeroDivisionError, NameError 以及 TypeError。對於內建的exception來說，這些印出來的字串都是這些內建的exception類型的真正類型名稱，但是對於使用者自己自定的exception類型就不一定了(雖然這是一個有用的約定俗成的習慣)。這些標準的exception名稱也正是他們內建的指稱(identifiers) (不是正式的關鍵字)。這一行其他部分則是有關這個exception類型的詳細解釋，其意義則是依照exception的類型而有不同。

在錯誤信息最後一行之前的部分則是顯示了這個exception發生時的狀況，也就是記憶體堆積(stack)的內容追朔(backtrace)。一般來說這個這個部分包含了stack backtrace的來源行數，但是這並不代表是在從標準輸入讀入時候的行數。

**8.3 例外(Exceptions)情形的處理**

我們可以寫一個程式來處理某些的exception。請看下面程式範例，我們要求使用者輸入一個有效的數字，直到所輸入的數字真正有效為止。但是使用者也可以中止這個程式(用Control-C 或者是任何作業系統支援的方式)。值得注意的是，使用者主動中止程式事實上是使用者引發一個KeyboardInterrupt的exception。

>>> while 1:  
... try:  
... x = int(raw\_input("Please enter a number: "))  
... break  
... except ValueError:  
... print "Oops! That was no valid number. Try again..."  
...

這個 try 敘述的作用如下：

* 首先，try之後的敘述(*try clause*，在 try 及 except 這兩個字之中所有的敘述)都會被執行。
* 如果沒有發生任何exception，except之後的敘述( *except clause* )會自動被忽略，整個 try 敘述就算是執行完畢。
* 如果當執行try之後的敘述時發生了exception，錯誤地方之後的敘述就不會被執行。然後如果這個exception的類型有某一個適合的 except 關鍵字之後的類型的話，就會執行這一個except之後的敘述，然後程式從整個 try 敘述之後的地方開始執行。
* 如果所發生的exception在except關鍵字之後找不到相對應的類型時，系統會將這個類型傳給外面一層的 try 敘述。如果外層的exception處理機制不存在的話，這就是一個沒有被處理的exception( *unhandled exception* )，然後整個程式會中斷，並出現上面出現的錯誤程式。

一個 try 敘述可以包含許多的except 部分來處理各種不同的exception，但是最多只有一個handler(譯：exception之後的敘述)會真正被執行。Handlers只處理在所對應的try部分發生的exception，其他的 try 部分發生的exception則不在處理範圍。一個except子句可以處理一個以上的exception，只要用list括弧把它們括起來。例如：

... except (RuntimeError, TypeError, NameError):  
... pass

最後的一個 except 可以不寫出exception 類型的名稱，這就當作是一個外卡(wildcard，譯：處理所有的exception)來使用。當使用時要特別的小心，因為如果你很有可能就把一個應該被注意的程式錯誤給隱藏起來了。你也可以在這個except子句裡面印出一個錯誤信息，然後把這個exception再丟(raise)出去(讓呼叫你程式的人來處理這個exception)。

import string, sys  
  
try:  
 f = open('myfile.txt')  
 s = f.readline()  
 i = int(string.strip(s))  
except IOError, (errno, strerror):  
 print "I/O error(%s): %s" % (errno, strerror)  
except ValueError:  
 print "Could not convert data to an integer."  
except:  
 print "Unexpected error:", sys.exc\_info()[0]  
 raise

這個 try ... except 的敘述有一個可有可無的else子句(*else clause*)可以使用，當這個子句存在時，必須是放在所有的except clauses的後面。這個子句裡的敘述是當try子句沒有發生任何exception時，一定要執行的敘述。請看例子：

for arg in sys.argv[1:]:  
 try:  
 f = open(arg, 'r')  
 except IOError:  
 print 'cannot open', arg  
 else:  
 print arg, 'has', len(f.readlines()), 'lines'  
 f.close()

使用else要比在try子句裡面加入多餘的程式碼來的好，因為這樣減少意外的處理到那些不是由try ... except 敘述中保護的程式碼所引發的exception。

當一個exception發生時，exception本身有一個連帶的值，也叫做這個exception的參數(*argument*)。至於這個參數是否存在以及其型態，則是由exception的類型所決定。對於有這個參數存在的exception類型來說，你可以在except clause的後面加入一個名稱(或是list)來接收這個參數的值。請看下例：

>>> try:  
... spam()  
... except NameError, x:  
... print 'name', x, 'undefined'  
...   
name spam undefined

如果一個exception有一個參數的話，當它在沒有被處理，當作錯誤信息印出來的時候，就會成為最後(詳細解釋(`detail'))的一部份。

Exception的處理者(handlers，exception clause)並不只處理在try clause當中所發生的exception，也會處理所有在try clause當中所(直接或間接)呼叫的函式所引發的exception。請看下例：

>>> def this\_fails():  
... x = 1/0  
...   
>>> try:  
... this\_fails()  
... except ZeroDivisionError, detail:  
... print 'Handling run-time error:', detail  
...   
Handling run-time error: integer division or modulo

**8.4 如何引發例外(Exceptions)**

使用raise敘述可以引發一個特定的exception，例如：

>>> raise NameError, 'HiThere'  
Traceback (innermost last):  
 File "<stdin>", line 1  
NameError: HiThere

raise 的第一個參數是想要引發的exception的類型，第二個參數(可有可無)則是指定這個exception的參數值。

**8.5 使用者自訂的例外(Exceptions)**

可以命名自己想要的excetion，其方法是指定一個字串給一個變數，或者是自己創造一個新的exception類別來。舉例說明：

>>> class MyError:  
... def \_\_init\_\_(self, value):  
... self.value = value  
... def \_\_str\_\_(self):  
... return `self.value`  
...   
>>> try:  
... raise MyError(2\*2)  
... except MyError, e:  
... print 'My exception occurred, value:', e.value  
...   
My exception occurred, value: 4  
>>> raise MyError, 1  
Traceback (innermost last):  
 File "<stdin>", line 1  
\_\_main\_\_.MyError: 1

許多標準的module都自己自訂其exception來報回(report)在他們自己所定義的函式裡面所發生的錯誤。

**8.6 定義善後的動作**

在try敘述的機制裡面有一個可有可無的子句(optional clause)，其功用是在定義不管什麼情況發生下，你都得要做的清除善後的工作。 舉例來說：

>>> try:  
... raise KeyboardInterrupt  
... finally:  
... print 'Goodbye, world!'  
...   
Goodbye, world!  
Traceback (innermost last):  
 File "<stdin>", line 2  
KeyboardInterrupt

這個*finally clause*不管你的程式在try裡面是否有任何的exception發生都會被執行。當exception發生時，程式會執行finally clause之後再引發這個exception。當程式的try部分因為break或return離開時，finally clause也一樣會在離開的時候(“on the way out'')被執行。一個 try 敘述機制應該要有一個或多個except clauses，或者是有一個 finally clause，但是不能兩個都有。

# 9. Class(類別)

對於class來說最重要的一些特性在Python裡面都完全的保留：類別的繼承可以繼承自個基礎類別(base classes)，一個子類別(derived class)可以override其所有基礎類別(base class)的任何方法(method)，一個method也可以呼叫一個基礎類別的同名方法，物件可以自由決定是否要讓某些資料是private的。

以C++的術語來說，Python所有的類別成員(包含其資料成員)都是*public*的，而且所有的函式成員(member functions)都是*virtual*的。也並沒有所謂的建構元(constructors)或是解構元(destructors)的存在。

# 9.1 術語的使用說明

由於缺乏普遍性的術語可以討論類別，我只好偶而C++的術語中借來用。物件這個字在Python裡面不必然指的是類別的一個特例(instance)，這是一個在物件導向讀者中常見的陷阱。並非所有在Python裡面的資料型態都是類別，像是整數及list這類的基本的內建型態就不是類別，甚至一些特別的資料型態像是file都不是類別。*所有的*Python的資料型態都或多或少都有一些基本相同的語意特性，我們可以把這個相同點叫做物件。

# 9.3 Class(類別)初探

類別(Classes)的觀念引進了許多新的語法，三種新的物件以及一些新的語言上的意義：   
9.3.1 定義Class(類別)的語法

最簡單的類別定義的形式看起來像是這樣的：

class ClassName:  
 <statement-1>  
 .  
 .  
 <statement-N>

類別的定義與函式的定義(都是用def 敘述)相同，都必須在要在它們有任何作用之前就定義好。(你也可以在 if 敘述或是一個函式裡面放入類別的定義)。在實務上，存在於類別定義內的敘述通常都是函式的定義，但是我們也可以放入其他的敘述。這樣的做法有時也很好用，我們之後會再會來看這個用法。類別定義內的函式定義通常都有一個特別的參數形式，這是為了method的特別呼叫習俗的。我們還是留到後面再來討論之。

當一個類別的定義正常的離開時(藉由定義的尾端)，一個類別物件(*class object* )就被創造出來了。這個類別物件基本上來說是只是一個包裝起來的東西，其內容是由這個類別定義所創造出來的namespace裡面的內容。

## 9.3.2 類別物件(Class Objects)

類別物件可以做兩件事情，一是attribute的指涉(references)，另一個是創造出一個特例來(instantiation)。

*Attribute references* 所使用的是在Python裡面標準的attribute reference的語法： obj.name 。有效的attribute的名稱指的是當類別物件被創造時，所有在類別的namespace裡面的名稱。所以，如果你的類別定義如同下面例子的話：

class MyClass:  
 "A simple example class"  
 i = 12345  
 def f(x):  
 return 'hello world'

你就可以使用 MyClass.i 以及 MyClass.f 這兩個有效的attribute references語法，它們分別會傳回一個整數以及一個method物件來。你也可以設定值給這些類別的attributes，如此你就可以改變 MyClass.i 的值了。 \_\_doc\_\_ 也是類別物件的一個有效的attribute，其傳回值是這個類別的注釋字串(docstring)，也就是："A simple example class"。

類別的特例化(Class *instantiation* )是使用函式的表示方法。看起來好像這個類別物件是一個沒有參數的函式，然後傳回來的就是這個類別的的一個特例(instance)。我們再以前面的類別為例子：

x = MyClass()

就會創造出一個新的類別的*instance*，然後我們再把這個物件設定給x這個local的變數。

類別的特例化(Class instantiation )這個動作(也就是”呼叫''一個類別物件)所創造出來的是一個空的物件。類別希望創造出來的物件有一個特定的初始狀態，所以你可以在類別裡面定義一個特別的method叫做\_\_init\_\_()，如同下例：

def \_\_init\_\_(self):  
 self.data = []

當你的類別有定義一個 \_\_init\_\_() method時，當你在特例化(instantiation)你的類別時，就會自動的引發\_\_init\_\_()執行，並且創造出一個類別的特例(instance)。所以，一個新的物件就可以截由底下的呼叫來創造出來：

x = MyClass()

當然， \_\_init\_\_()這個method可以有參數傳入，這樣可以增加使用時的彈性。如範例：

>>> class Complex:  
... def \_\_init\_\_(self, realpart, imagpart):  
... self.r = realpart  
... self.i = imagpart  
...   
>>> x = Complex(3.0,-4.5)  
>>> x.r, x.i  
(3.0, -4.5)

## 9.4 一些隨意的想法

習慣上我們把一個method的第一個參數叫做self。這只是一個習慣而已，self 這個名字對Python來說完全沒有什麼特殊的意義。(但是你要注意，如果你不用這一個習慣的話，對於某些讀你程式的Python程式設計師來說，也許你程式的可讀性就低了一點。而且可能有一些類似像*class browser*之類的程式是靠這個約定來分辨class的特性，所以你不遵守的話，你的類別它們可能就讀不懂)。

在method裡面呼叫其他的method，你所需要的只是用self 這個參數的method attribute就可以了。例如：

class Bag:  
 def \_\_init\_\_(self):  
 self.data = []  
 def add(self, x):  
 self.data.append(x)  
 def addtwice(self, x):  
 self.add(x)  
 self.add(x)

method跟一般的函式物件一樣可以使用全域名稱(global name)。Method的global scope所指的是類別的定義所存在的module，(注意：類別本身絕不會是一個global scope！)。 你大概很少有機會在method裡面會需要用到global scope，但是你還是可以使用global scope的，method可以使用在global scope之中所import進來的函式以及module，也可以使用在global scope裡面定義的函式及類別。

# 9.5 繼承(Inheritance)

一個程式語言如果沒有繼承的話就不需要擔心類別(“class'')這個字了。一個子類別(derived class)的定義看起來是這樣的：

class DerivedClassName(BaseClassName):  
 <statement-1>  
 .  
 .  
 <statement-N>

其中，基礎類別的名字BaseClassName這個字必須是在子類別所處的scope裡面有定義的。除了直接使用基礎類別的名字之外，你也可以使用一個expression。這在當你的基礎類別是定義在別的module裡的時候特別有用：

class DerivedClassName(modname.BaseClassName):

子類別定義的執行過程與基礎類別定義的執行過程是一樣的。當一個類別物件被創造出來時，基礎類別也同樣會存在記憶體中。這是為了要確保能夠找到正確的attribute的所在，如果你的子類別沒有定義某個attribute的話，就會自動去找基礎類別的定義。如果這個基礎類別也是某個類別的子類別的話，這個法則是一直延伸上去的。

子類別可以override基礎類別裡的method。因為method在呼叫自己物件的其他method的時候沒有特別的權限，當一個基礎類別的method呼叫原屬於該基礎類別的method的時候，有可能真正呼叫到的是一個在子類別裡面定義的override的method。(給C++的程式設計師們：所有在Python裡面的method都是virtual的。)