第二章 真假世界

日常生活常常會遭遇各式各樣的情況，我們也常常無法預期。真實世界往往是很複雜的，生物的多樣性告訴我們地球上的生物有非常多種，而且不斷的有新的生物出現，也不斷的有存在已久的生物突然消失。

如人的情感，有悲有喜，又有程度的不同，畢竟每個人所能領略的互有差異，也如藝術，喜歡不喜歡變成是對美感最直接的詮釋。但是很多時候我們仍是必須做出選擇，買了藍色墨水的筆，因為預算的關係，黑色墨水的筆大概就得捨棄。

選擇好像已經變成生活中經常性的事情，譬如去一個地方，走路去是一個選擇，騎腳踏車、坐公車或是利用其它種類的交通工具，這些都成為個別的選項。很多時候選項有很多個，但是有些時候我們寧可只要兩個相對的選項，「對」或是「錯」。

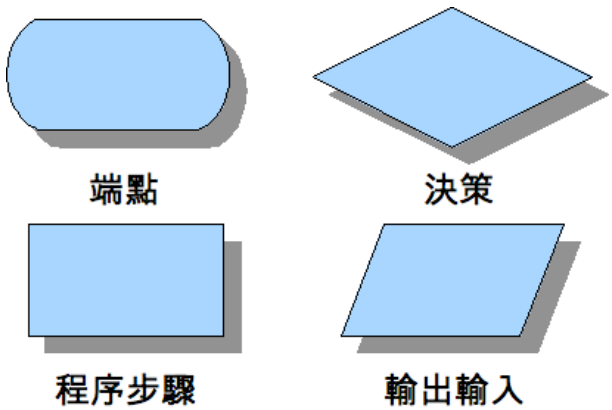
邏輯上利用**真假值**來分辨相對的概念，某種角度上來說，對就是「**真**」，而錯也就是「**假**」，他們的意思接近，同時在處理情況上，我們不講什麼是對的，倒是會刻意在乎這種情況是否為「真」。若是這種情況為真，接著做某某事，若是假，則是去做其他的事情。這樣一來，情況的處理變得相對簡單許多。

讓事情變得簡單，盡量簡單，很多時候直接的解決了問題。好比燈泡一般，燈泡要亮，可不只是讓電流通過燈絲而已，還要把燈泡內的空間抽成真空，為什麼呢？因為空氣裡有氧，氧會助燃，沒有抽成真空的結果，會使燈絲燃燒起來。

這就是個簡單的解決方法，當然還有其他的解決方法，然而成本效益的考量之下，簡單往往成為最好的辦法。其他如攝影，要讓照片看起來更具有張力，在構圖上就要更簡單，因為簡單的表現出主體，前景背景不至於雜亂，才能給人眼睛一亮，清楚明白的看出照片所表達的事物。

同樣利用電腦解決問題，我們也要洞察問題的本質，從而找出簡單的途徑，才能有效率的解決問題。為此程式語言提供**流程控制**的方法，讓我們可以將問題細部分割成一個一個的小單元，解決了一個單元，再解決下一個，接續著將所有小單元都解決完成，我們為了解決某一問題所寫的程式也就能夠順利運作。

流程圖是用來表現流程控制常用的方法，常用的符號如下圖。



「端點」代表開始或結束，「決策」表示對某個選擇做出決定，「程序步驟」通常是程式中單一個陳述，如指派陳述等，「輸出輸入」用來印出訊息，如print陳述，或是要求使用者輸入資訊。

我們首先會碰到處理選擇的方法，也就是流程圖符號中的決策部份---------**if陳述**。

如果的話

英文中，if的意思就是如果怎麼樣，那就怎麼樣，在程式語言中，if陳述的意義完全相同，句法如下。

```
>>> i=1
>>> if i%3==1:
        print "i被3除餘1"


i被3除餘1
```

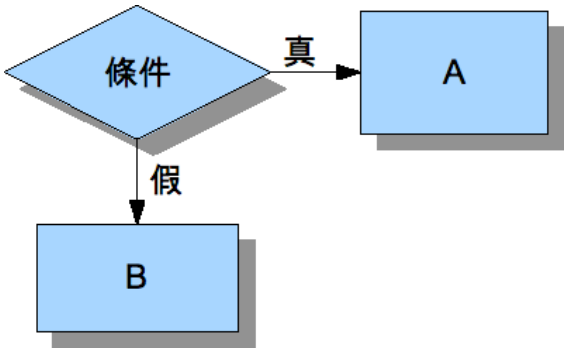
Python用單一個等號表示指派，而用連續兩個等號代表相等，因此「i%3==1」的意思就是判斷i是否被3除得到餘數為1，是的話直譯器會傳回True，也就是真。

```
>>> i%3==1
True
```

所以「if i%3==1:」等同於「if True:」，出現在if後面的「i%3==1」被稱為**條件**。如果條件為真，底下**縮排**四格的「print i」的陳述就會執行，在我們連續按下兩次Enter鍵，Python Shell就自動知道我們不會再輸入程式碼，於是執行結果，也就是印出i的數值。

縮排是Python分辨**程式區塊**的方法。所謂的程式區塊是程式中一個特定的區域，可能會被執行，也可能不會被執行，端看我們如何設計啟動程式區塊的條件，if陳述正是其中之一。

我們來看看if陳述的流程圖。



條件為「真」執行A步驟，條件為「假」則執行B步驟。剛才我們見到的是單一的if陳述，因此若是條件為假，就會跳過if陳述底下的程式區塊，直接執行沒有縮排的下一個陳述，在Python Shell之中，就是出現下一個提示符號（>>>）。

我們也可以在條件為假時讓程式去執行另一個程式區塊，這時候就要用到**else陳述**。

```
>>> i=i+1
>>> if i%3==1:
        print "i被3除餘1"
else:
        print "i被3除並非餘1"


i被3除並非餘1
```

將i+1指派到i，本來i儲存的值為1，這時候變成了2，所以3除以2餘2，並非餘1，條件為假於是執行else陳述的部份。

作者：張凱慶，Email：kaichingc@gmail.com

Note

相同的縮排量表示在某一特定等級的程式區塊內，這是說接續的陳述沒有縮排也就離開了該區塊，如果是另一個縮排量更多的陳述，那就是另一個的程式區塊了。 Python Shell 中的提示符號代表沒有縮排的等級，因此提示符號後接著相同等級陳述時，會變成與提示符號並排。

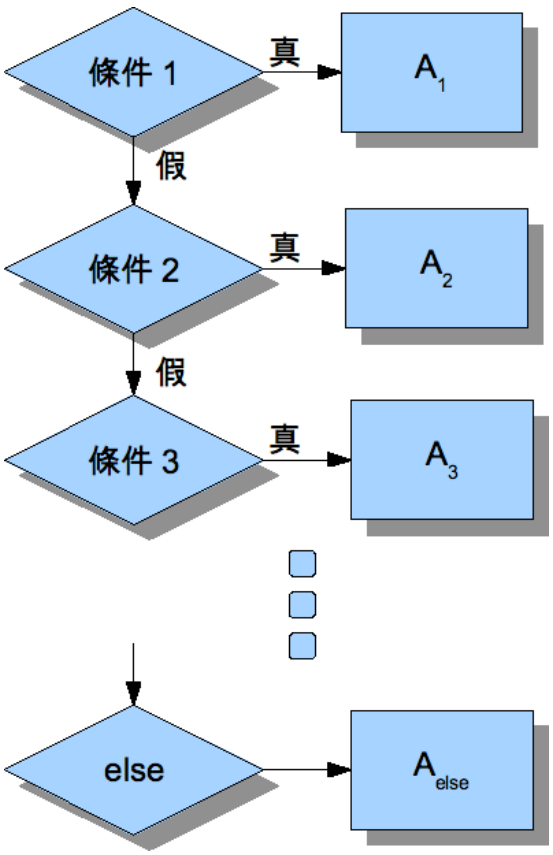
當然，很多時候條件不只是一個，當我們要檢查多過一個的條件時，我們可以利用**elif陳述**。

```
>>> i=i*0.6
>>> if i%3==0:
        print "i可被3整除"
elif i%3==1:
        print "i被3除餘1"
elif i%3==2:
        print "i被3除餘2"
else:
        print i, "不是整數"


1.2 不是整數
```

我們利用檢查i可否被3、2、1整除的條件判斷，最後的else則是代表以上皆非的情況。當然，最後不放else也是可以的，else陳述是避免連續的條件檢查找不到符合的條件，以致於全部跳過什麼都沒做。

if...elif...else的流程圖像是這樣。



連續兩個等號構成的運算子被稱為**比較運算子**，這樣的運算式就是比較前後兩者是否相等。其他的比較運算子還有不等、大於、小於、大於等於、小於等於。

運算種類	運算子
相等	=
不等	!=
大於	>
小於	<
大於等於	>=
小於等於	<=

迴圈

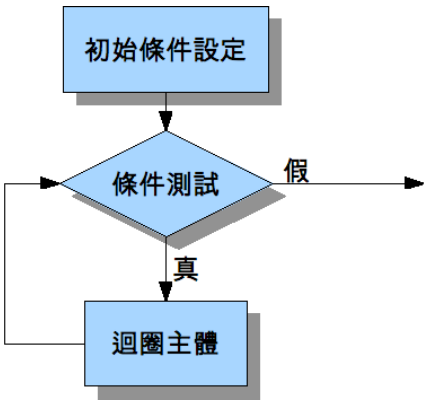
如果要讓電腦做些重複性的工作，譬如計算從1到100所有正整數的總和，這時候就需要**while陳述**了，我們先來看看一個簡單的例子。

```
>>> i=10
>>> while i>-1:
        print i
        i=i-1


10
9
8
7
6
5
4
3
2
1
0
```

這一段程式碼做了個簡單工作，有點像是倒數計時。首先替變數i設初值為10，然後是while陳述，其內要做一次條件檢查，結果若是真，接著執行區塊內的陳述，而若是假，就會跳出while陳述以及所屬的程式區塊。

像這樣的while陳述與其下的程式區塊，我們稱之為**迴圈**，這是因為好像不斷的繞圈子一樣，迴圈重複著做相同的事情。所以while陳述底下的程式區塊又被稱為**迴圈主體**，while陳述所構成的迴圈被稱為**while迴圈**。



典型的while迴圈利用變數做條件檢查，當然，可以自由為變數取名稱，不過常用的變數名稱與數學上相似，大都是i、j、k之類的。迴圈開始前替變數設定初值，條件檢查通常用作迴圈重複次數的設定，或是用某些臨界值來跳離迴圈，而迴圈主體內改變儲存在條件檢查的變數的數值，以作為下一次條件檢查之用。while迴圈的流程圖如右。

所以要如何計算從1到100所有正整數的總和呢？我們可以多用一個變數s來儲存加總的值。

```
>>> i=1
>>> s=0
>>> while i<=100:
        s=s+i
        i=i+1


>>> print s
5050
```

i等於101時就會跳出迴圈，在這之前，i從1變為2，再變為3...，每通過一次迴圈，i的值就被加進變數s之中。離開迴圈後，我們再用print陳述查看變數s之值，結果是5050。

$$\sum 1+2+3+...+98+99+100=\frac{(1+100)\times 100}{2}=5050$$

驗算符合結果，嗯，公式在手算時是一種簡便的方式，然而電腦的計算速度允許我們直接累加，而似乎絲毫不會讓我們感到緩慢。

使用while對於重複性工作的處理相當方便，不過還是有幾點要留心，首先要注意條件檢查的變數一定要設定初值，也就是在該變數進入while陳述的條件檢查之前，要先指派一個數值儲存到變數裡頭。

```
>>> while i<5:
        print i


Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    while i<5:
NameError: name 'i' is not defined
```

這裡發生了一個名稱錯誤，因為變數i從來沒出現過，直譯器不認識這個變數名稱，於是直譯器無法處理這個變數i。另外要避免**無窮迴圈**的發生。

```
>>> i=5
>>> while i>0:
        print i,


5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5
Traceback (most recent call last):
  File "<pyshell#12>", line 2, in <module>
    print i,
KeyboardInterrupt
```

由於i等於5，進入迴圈i大於0的條件測試成立，但是迴圈主體內沒有調整變數i的陳述，導致迴圈的條件測試永遠成立，於是程式不斷的印出數值i。這裡是用按〔ctrl〕+〔c〕終止目前的執行程式。

Note

print陳述若是以逗點做結束，這樣會抑制印出新行符號，新行符號也是跳脫序列的一種，其作用為印出新行，使游標移到下一行的開頭。

第二個複合資料型態：串列

Python的複合資料型態除了字串之外還有很多種，而**串列**好比是早期農村的牛，任勞任怨的替農夫耕田、拉車的做了許多粗活。字串是用兩個引號包起來的資料，串列則是用中括號將資料放在裡頭。

```
>>> my_list=["Taipei", "端午節", i, j, 33, 5.24, ["John", 1111]]
>>> type(my_list)
<type 'list'>
```

字串、整數、浮點數、變數甚至是串列本身都可以用串列來儲存，利用type(my_list）來查詢是哪種資料型態，我們可以看到是「list」。List的中文意思是表格、目錄，程式語言中這樣的資料型態通常是一長串列舉出的項目，因而我們稱list為「串列」。

Note

建立串列時可以包含變數，然而只會把變數所代表的數值納入。

習慣上，任何新的資料都會用一個變數來儲存，因為這麼一來可以直接透過變數來處理資料。串列與字串的操作有許多地方相似，兩者都可以利用索引值來存取個別的元素，或是切開存取出某一段資料。

```
>>> print my_list[1]
端午節
>>> new_list=my_list[4:6]
>>> print new_list
[33, 5.2400000000000002]
```

串列中可以儲存字串、串列等複合資料型態，其中的個別成份可以切開存取嗎？答案是肯定的，這時候就要用到兩層的索引值了。

```
>>> print my_list[6][1]
1111
>>> print my_list[1][0:4]
端午
>>> print my_list[6][0][3]
n
```

串列中串列裡的字串需要三層的索引值，像這樣複雜多層的結構，索引值依據層數而選取。加號、乘號都可運用在串列上。

```
>>> [3247]+new_list
[3247, 33, 5.2400000000000002]
>>> [11]*2+new_list
[11, 11, 33, 5.2400000000000002]
```

加號可加入新的項目，乘號則是可以加倍串列中項目的數量。同樣加號與乘號也可以在字串上運用。

```
>>> "Hello, "+my_list[6][0]+"!"*3
'Hello, John!!!'
```

作者：張凱慶，Email：kaichingc@gmail.com

Note

加號與乘號很容易聯想到數學上的關係，然而減號、除號等卻不容易，試想若是減號若是表示刪去串列中某個項目，而該項目原本就不存在，這樣運算有何意義嗎？

如果要刪除串列中某個項目，我們可以利用**del陳述**。

```
>>> del new_list[1]
>>> print new_list
[33]
```

然而，這卻不能施用在字串上。

```
>>> del my_list[0][0]

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    del my_list[0][0]
TypeError: 'str' object doesn't support item deletion
>>> print my_list[0]
Taipei
```

這是一個型態錯誤，因為字串被設計成不可變更其內容。為什麼呢？因為把某些型態設計成**不可變**的時候，使用這一類型態的資料時，有如數學某些常數一般，比如圓周率3.14159256...，我們用公式計算圓面積= πR^2時，大可用某些逼近的 π 代入計算，算出的結果都很接近，這是一種很自然而然的事情。假如 π 不能是常數，嗯，計算圓面積就變成是一件很麻煩的事情。

所以有些型態被設計成可變的，也就是可以更動其內的項目，有些則被設計成不可變的，這可以說是我們不希望內容被破壞。不可變的資料型態雖然我們不能更動內容，然而，當內容指派到變數時，我們卻可以利用重新指派來更改變數所儲存的內容。

```
>>> new_string=my_list[0]
>>> new_string
'Taipei'
>>> type(new_string)
<type 'str'>
>>> new_string=new_string+"是臺北的英文名稱"
>>> print new_string
Taipei是臺北的英文名稱
```

Note

事實上，雖然串列是可變的，另有個跟串列相似的資料型態，那是用小括弧圍起來的**序對**，這種資料型態則是不可變的。

另一種迴圈

Python提供了另一種迴圈，**for陳述**可以進入複合資料型態，然後尋訪每個項目。我們先來看看一個簡單的例子。

```
>>> bug="ant"
>>> for i in bug:
        print i


a
n
t
```

Note

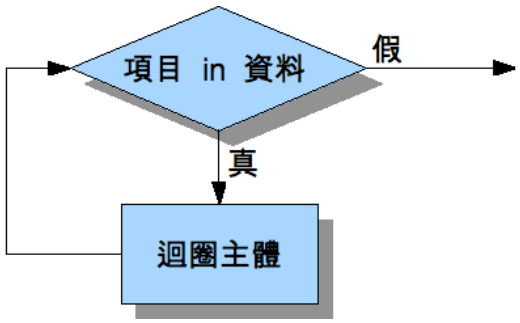
while迴圈裡用作條件檢查的變數i要先設初值，for迴圈卻不用，這是因為在for迴圈中資料的個別組成項目都會依序指派到變數i中，所以不需要先設初值。

變數bug之中儲存“ant”字串，而利用for陳述及**in運算子**將bug中所儲存的各個字元依次儲存到變數i之中，接下來一行一行的輪流印出“a”、“n”、“t”。in運算子用來檢查某個項目是否存在於資料之中，我們也可以單獨使用in運算子。

```
>>> "a" in bug
True
>>> "b" in bug
False
```

“a”在bug中，所以直譯器傳回True，也就是真，而bug裡不存在“b”，所以直譯器傳回False，也就是假，像這樣的運算子是屬於關係運算子的一種，可以檢查某個項目是否在資料內。

for迴圈的流程圖如下。



同樣的，for陳述也能施用在串列上。

```
>>> name=["Parker", "Helen", "Tony", "Alice"]
>>> for i in name:
        print "Hello,", i


Hello, Parker
Hello, Helen
Hello, Tony
Hello, Alice
```

嗯，除了說hello之外，我們可以在迴圈中做更多的事情。

```
>>> possessive=[]
>>> for i in name:
        i=i+"'s"
        possessive=possessive+[i]


>>> print possessive
["Parker's", "Helen's", "Tony's", "Alice's"]
```

我們把所有的名字改成所有格，然後用另一個串列來儲存結果，最後印出來檢查是否完成工作，的確，程式完成我們想要做的事情。

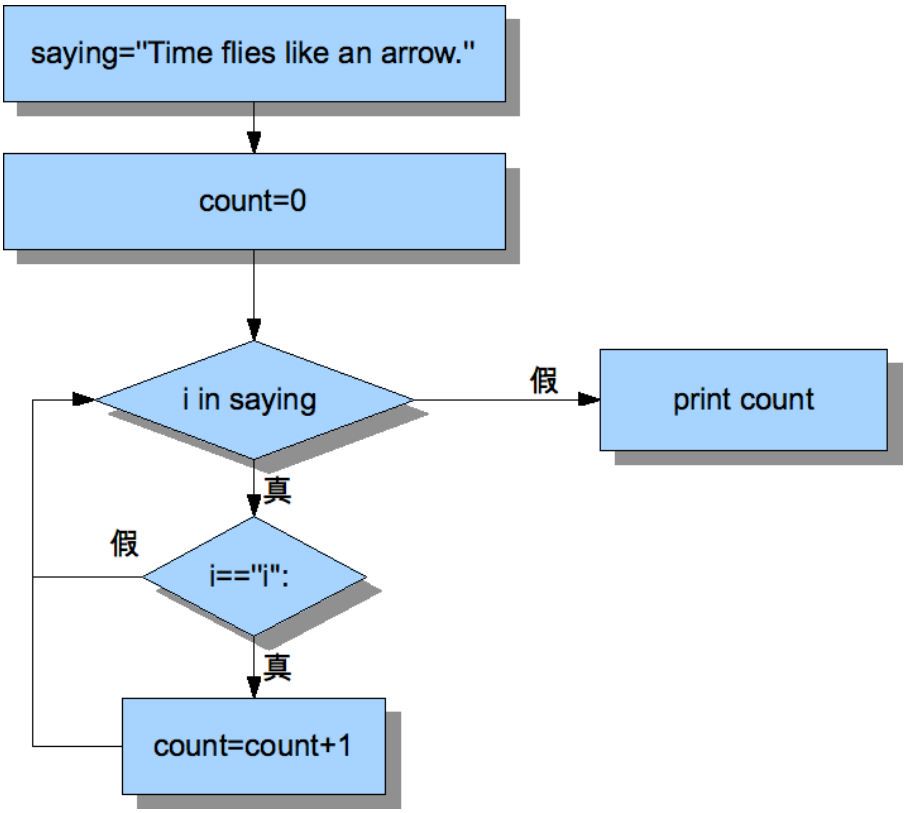
算字數

for迴圈有一個很方便的地方，就是能夠尋訪資料中的每個項目，因而用來統計英文句子中某個英文字母相當容易。譬如，有句英文是這樣說的：「Time flies like an arrow.」

```
>>> saying="Time flies like an arrow."
>>> count=0
>>> for i in saying:
        if i=="i":
                count=count+1


>>> print count
3
```

我們每發現一次i，就把變數count加1作為累計之用，最後印出count，的確總共有三個i。這一段程式若是用流程圖來表示，如下。



程式裡的一個動作，流程圖就用一個圖來表示，流程圖與程式碼之間也容易互相轉換，雖然寫出程式碼是一種直覺的工作，流程圖仍便於分析程式的所作所為。

有一個閩南俚語是這樣說的：「有嘴說別人，無嘴說自己。」我們來看看怎麼用程式來計算「嘴」出現的次數。

```
>>> slang="有嘴說別人，無嘴說自己。"
>>> count=0
>>> for i in slang:
        if i=="嘴":
                count=count+1


>>> print count
0
```

咦？怎麼會是0，程式寫法不是完全一樣嗎？問題在於for迴圈每次在字串中尋訪項目僅是一個字元，而每個中文字是由兩個字元所組成的。

```
>>> slang
'\xa6\xb3\xbcL\xbb\xa1\xa7O\xa4H\xa1A\xb5L\xbcL\xbb\xa1\xa6\xdb\xa4v\xa1C'
>>> "嘴"
'\xbcL'
```

我們可以看到「有」的編碼是“\xa6\xb3”，「嘴」的編碼是“\xbcL”，餘下類推。於是，每次i所被指派的內容為“\xa6”、“\xb3”、“\xbc”、“L”……，而我們在程式裡的判斷是否相等的內容卻是“\xbcL”，總是找不到相等的，所以變數count到尋訪結束仍是0。

那麼要怎麼解決這個問題呢？簡單的解法是一次檢查兩個項目內容，但是for迴圈的尋訪一次只能存取一個項目內容，也就是一個字元的單位。那for迴圈裡面再用一個for迴圈呢？嗯，不是個好方法。

因為字串可以用索引值存取其內容，for迴圈的尋訪過程也是從0開始依序前進，所以我們可以利用這個特性來設想解決的辦法。我們需要多兩個變數，j用來代表索引值，next用來代表for迴圈尋訪時的下一個字元。

```
>>> j=0
>>> count=0
>>> for item in slang:
        next=slang[j+1]
        if item=="\xbc":
                if next=="L":
                        count=count+1
        j+=1


Traceback (most recent call last):
  File "<pyshell#18>", line 2, in <module>
    next=slang[j+1]
IndexError: string index out of range
>>> print count
2
```

程式雖然可以運作，也成功算出正確的結果，但是發生了一個錯誤。這種錯誤稱為**執行錯誤**，也就是說在程式的執行期間發生的錯誤，這會導致程式的執行中斷。如果我們脫離Python Shell執行這個程式，執行錯誤會停止程式執行，同時我們無法存取出程式中變數的值。所以，雖然結果是對的，我們仍然要想辦法來修正這個錯誤。

這個錯誤是說字串的索引值超出範圍，為什麼呢？因為slang所儲存的共有十個中文字，兩個全形符號，所以有10+2乘以2等於24個字元，我們也可以藉由len()來查詢字串的項目個數。

```
>>> len(slang)
24
```

的確有24個字元，那麼索引值的範圍是從0到23。for迴圈尋訪的最後一次，變數item存取索引值為23的字元，這時候j的值也是23，而變數next卻是要去存取j+1的字元，j+1是24，實際上不存在索引值為24的字元，於是錯誤就發生了。

如果我們能夠在j等於索引值的最大值時就結束迴圈，其實就能夠修正這個錯誤。怎麼做呢？我們可以利用**break陳述**來中斷迴圈。

```
>>> j=0
>>> count=0
>>> for item in slang:
        next=slang[j+1]
        if item=="\xbc":
                if next=="L":
                        count=count+1
        j=j+1
        if j>=(len(slang)-1):
                break


>>> print count
2
```

Note

break陳述是用來中斷迴圈，另外有個**continue陳述**則會讓迴圈跳過該圈，而直接進行下一圈的運算。

我們還可以把程式碼寫的稍微簡短一點，利用**and運算子**將兩個字元的比較寫在同一行。

```
>>> j=0
>>> count=0
>>> for item in slang:
        next=slang[j+1]
        if item=="\xbc" and next=="L":
                count=count+1
        j=j+1
        if j>=(len(slang)-1):
                break


>>> print count
2
```

and是**邏輯運算**的一種，中文稱作「且」，必須是前後兩個運算元都為真，and邏輯運算的結果才會為真。

Note

還有其他兩種邏輯運算子，**or運算子**與**not運算子**。or中文稱作「或」，也是需要兩個運算元，而只要其中一個運算元為真，運算結果就為真。not中文稱作「否」，否定的意思，連接not只有一個運算元，若運算元為真，結果為假，反之亦然。

"Everything is an object."

「所有的一切都是物件。」這句話是什麼意思呢？所謂的“object”中文意思應該是那裡有個什麼，不知其名，姑且用“object”稱之，因此“object”中文意思接近的詞為「東西」，但是“object”又有對象、目標的意含。在程式語言的領域中，“object”被廣泛的應用。

那麼在程式語言裡頭，“object”的意思到底應該是什麼？中文相對用詞大家都已經很習慣用「物件」稱之，即有物品與事件的雙重含意，這也是說資料與資料的處理。在Python裡頭，所有的一切都是物件，物件被建立後必有型態與數值。

也許你聽說過**物件導向程式設計**。比如寫賽車遊戲好了， 這樣的設計方式，我們會希望所有的背景，譬如一棵樹會是一個物件，車子是一個物件，跑道也是物件，藍天白雲都是物件，好處是我們只需要把物件寫出來，不同的場景都能直接套用。

效益是減少許多重複的程式碼，因而更快的完成程式開發的工作。既然Python中所有的一切都是物件，Python本身就是強大的物件導向程式設計語言，因而我們不會刻意渲染物件的可貴，而會著重於物件本身的性質，也就是型態與數值。

我們已經見過了基本資料型態，整數與浮點數，複合資料型態，字串與串列，接下來我們在下兩章，分別會碰到**函數**與**自訂資料型態**，這些都是物件的一種，也都具有其相關的性質。

Note

關於Python中物件的詳細說明，可參考 Python Reference Manual 中的 Objects, values and types。