

第三章 設計介面

在Python shell中寫些簡單的程式直接與直譯器互動，我們可以得到立即的結果。然而如果每次都要一行一行的輸入，就顯得沒什麼效率，也很難重複利用我們寫過的程式。我們要如何重複利用我們所寫的程式呢？

譬如我們現在要來寫一個程式，這個程式是要使用者輸入一個整數，然後在已經數列中搜尋一個整數，如果這個整數存在，程式印出「在...的位置」，而又如果這個整數不存在，程式則印出「不在這裡」。

我們希望藉由一種**介面**，使得透過介面與介面之間的聯繫來達到程式碼重複使用的目的。上述的例子中，我們會希望程式像是下面這樣的進行。

```
input_integer  
search  
print_result
```

先是輸入想要搜尋的整數數值，然後進行尋找。「`input_integer`」、「`search`」、「`print_result`」等各自屬於不同的介面，完成搜尋一個整數數值後，我們能夠再次依序透過介面搜尋另一個數值，而不必重寫所有的程式碼。

像這樣的介面是把程式分成一個一個的小單元，讓程式由小單元依次或是按照某個順序來執行，好處是我們只需要撰寫所需的程式碼，然後放入單元中，安排單元的執行順序。某些程式語言把這樣的單元稱為常式、程序或副程式等等，Python裡頭我們稱之為**函數**。

這裡所謂的函數，跟數學上的「函數」概念上十分相近，原文兩者都是function。數學透過函數求值，Python則是利用函數做一些事情，如執行計算，或是印出數值之類的動作，凡是陳述都可以放入函數的單元之中。

Note

Function的中文用詞函數與函式都有人用，函式的目的在於彰顯一種表達概念，諸如「程式」、「化學式」、「方程式」等結尾都用「式」。

函數是我們將要學習的介面其中之一，也是第一步，由自行定義函數，使程式分成一個個小單元運作，從而更易於控制程式的執行。第四章將學習第二步，**自訂資料型態**，讓我們可以自行定義在程式中的「物件」，使程式更符合我們的需求。

第五章則將學習第三步，自訂**模組**。所有的Python程式都是模組，這是說當我們寫好了一個Python程式，在其他的程式中就可以利用這個已經寫好的Python程式中所定義的函數與型態。我們也將在這一章中學習如何讓Python除了當作模組，還可以作為正常的執行程式。

其實我們已經見過一些函數，像是`license()`、`type()`、`help()`及`len()`都是函數的一種，有些需要參數，藉由參數得到跟參數相關的結果，這些之前我們所用的函數都是屬於Python廣大的**built-ins**。

我們先藉由複習提出一個觀念，程式語言的關鍵字，並且探討**built-ins**的概念。

關鍵字與**built-ins**

所謂的**關鍵字**就是程式語言保留給語法使用的字詞，具有其相關的語法功能，在IDLE中也都用淺橘色來表示。我們已經見過許多關鍵字，以下是Python完整的關鍵字列表。

| | | | | | | |
|--------|----------|---------|--------|--------|--------|-------|
| and | continue | except | global | lambda | raise | yield |
| as | def | exce | if | not | return | |
| assert | del | finally | import | or | try | |
| break | elif | for | in | pass | while | |
| class | else | from | is | print | with | |

除了已經學過的關鍵字，我們還會慢慢的學習其他的關鍵字，以能掌握Python語言更多的特性。另外，Python有大量的***built-ins***，***built-in***是嵌入或固定的意思，這是說Python本身就內建了許多有用的功能，我們接觸過的有兩大類，內建型態與內建函數。

諸如整數、浮點數、字串及串列都是內建型態，而license()、type()、help()及len()等都是內建函數，當然，***built-ins***不只有這兩類，內建型態與內建函數也絕非止於這幾種，但是當我們需要時才會做介紹。

舉例來說，1到100的總和可以簡單的用內建函數計算。range(101)建立一個串列，從0開始每次遞增1到100，然後利用sum(i)把串列內所有數字相加。

```
>>> i=range(101)
>>> i
[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, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
>>> sum(i)
5050
```

Note

range(x)所建立的串列是從0開始，然後一直到x-1為止。

其實不需要額外的變數，我們可以直接把range(101)當作sum()的參數。

```
>>> sum(range(101))
5050
```

這是個簡寫的方式。另外，利用內建函數還可以輕易的做到型態轉換。

```
>>> number="100"
>>> integer=int(number)
>>> integer
100
>>> floating=float(number)
>>> floating
100.0
>>> new_list=list(number)
>>> new_list
['1', '0', '0']
```

常用內建型態如整數、浮點數、字串、串列等，全都有相對應的內建函數可以互相轉換。

Note

關於Python中的***built-ins***，詳情可參考Python Library Reference的[Built-in Objects](#)及[Built-in Type](#)。

變數命名規則

我們已經見過Python的關鍵字以及大量的***built-ins***，現在重新來看看該怎麼為變數命名。

```
>>> while=0
SyntaxError: invalid syntax
```

凡是關鍵字都不可以用作變數名稱，因為關鍵字是專門為語法的保留用字，一旦關鍵字被用作變數名稱，直譯器會直接回傳這是錯誤的語法。另外，雖然***built-ins***的名稱在Python中也是保留用字，IDLE用紫色來表示，這些***built-ins***的名稱用作變數卻不會產生錯誤。

```
>>> sum=0
>>> sum
0
```

不過，雖然直譯器沒有回傳錯誤，我們還是應該盡量避免使用***built-ins***的名稱作為變數。

Python允許英文大小寫字母、底線及數字作為變數名稱，除了數字之外，英文大小寫字母、底線都可以用為變數開頭。要特別留心英文大小寫字母的不同，Snake與snake是不同的變數。

```
>>> Snake=1
>>> snake=1
>>> Snake==snake
True
>>> snake=2
>>> Snake==snake
False
```

Note

Python官方版並不支援中文做變數名稱，可以參考[周蟒](#)。

第一步：自訂函數

數學上有個有趣的**費伯納西數列**，假設第1個數為1，第2個數為1，第3個數等於第1個數與第2個數之和，第4個數等於第2個數與第3個數之和.....，第n個數為第n-2個數與第n-1個數之和.....

這個數列像是這樣。

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,

我們要如何計算費伯納西數列呢？因為每一個數都為前兩個數之和，第1、2個數都為1，所以我們假設有第0個數，而第0個數為0，這樣就形成了第二個數也為第0個數與第1個數之和。

```
fibonacci(0)=0
fibonacci(1)=1
fibonacci(2)=fibonacci(0)+fibonacci(1)
fibonacci(3)=fibonacci(1)+fibonacci(2)
fibonacci(4)=fibonacci(2)+fibonacci(3)
fibonacci(5)=fibonacci(3)+fibonacci(4)
...
fibonacci(m)=fibonacci(m-1)+fibonacci(m-2)
```

我們希望用一個函數把費伯納西數列算出來，就要先有個方式儲存數列中的數字，要用哪一種資料型態呢？要能儲存許多個數字，對了，串列是個不錯的選擇。

```
>>> f=[0, 1]
>>> f2=f[0]+f[1]
>>> f=f+[f2]
>>> f
[0, 1, 1]
>>> f3=f[1]+f[2]
>>> f=f+[f3]
>>> f
[0, 1, 1, 2]
>>> f4=f[2]+f[3]
>>> f=f+[f4]
>>> f
[0, 1, 1, 2, 3]
```

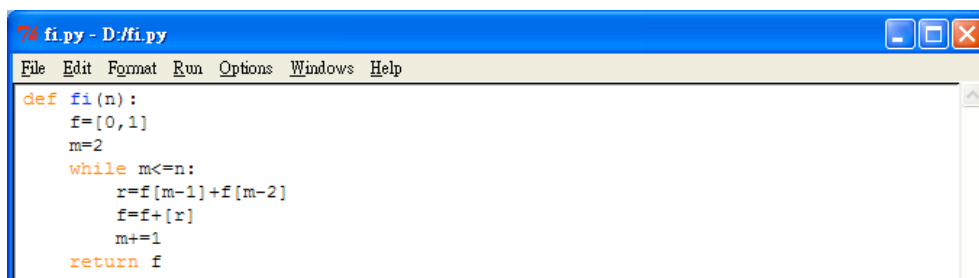
嗯，很麻煩，每算數列中的後一個數字就要寫兩行程式碼，若我們要算出十八個費伯納西數，顯然很沒有效率。因為每個計算動作都很相似，有一個很簡單的方式可以避免寫這麼多行程式碼，你想到了嗎？沒錯，就是利用迴圈。

```
>>> f=[0, 1]
>>> m=2
>>> while m<=18:
>>>     r=f[m-1]+f[m-2]
>>>     f=f+[r]
>>>     m=m+1
>>> f
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
```

在這裡有兩個初值設定，第一個是變數f，其型態為串列，我們利用f來儲存所算出的費伯納西數。接下來設定所計算費伯納西數的第一個，費伯納西數列中的第三個數，也就是fibonacci(2)，所以m設成2，然後就進入迴圈開始計算。

我們把r當作暫存變數，其為串列f中開始兩個數字的相加，因此r的型態為整數，所以當我們要把r再加回串列f時，r要加上中括弧變成[r]，這樣串列對串列才能相加。最後m遞增1，進行下一個費伯納西數的計算。

簡單多了，但我們希望的是函數，也就是透過一種介面會來的更方便。



```
fi.py - D:\fi.py
File Edit Format Run Options Windows Help
def fi(n):
    f=[0,1]
    m=2
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f
```

Note

這仍然是在IDLE之中，點擊【File】選單下的【New Window】，開啟新視窗來編輯程式。

函數定義所用的關鍵字為def，僅隨著是自訂的函數名稱，我們這裡用fi，要注意名稱後一定要有小括弧，小括弧裡的變數稱為**參數**，我們可藉由參數指定要算多少個費伯納西數。函數定義最後用了**return陳述**，這是用來回傳結果之用，因為我們結果儲存在變數f的串列中，所以我們在函數最後回傳變數f。

我們來看看怎麼樣利用這個函數。

```
>>> ===== RESTART =====
>>>
>>> fi(18)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
```

Note

執行新視窗所編輯的程式，要先存檔，記得要加上.py的副檔名，然後點擊【Run】選單下的【Run Module】。

Note

存檔時在Windows系統下可能會出現〔I/O Watning〕的視窗，這是提醒程式中有非ASCII的字元，我們暫時不用理會，按下〔Ok〕即可。

我們給予參數為18，fi函數並沒有讓我們失望，結果就算出18個費伯納西數。像是這樣的運作，就稱為**函數呼叫**，我們會一再的寫很多的函數，放入程式中然後需要時就呼叫。

區域變數

我們能不能在呼叫fi函數前，先建立空字串的變數f，然後透過呼叫fi函數，把計算結果儲存到變數f呢？

```
>>> ===== RESTART =====
>>>
>>> f=[]
>>> fi(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> f
[]
```

答案是不行的，因為函數內建立的變數為**區域變數**，效力僅限於該函數內，一旦離開了該函數，該變數的登錄就會被註銷。那有沒有方法可以繼續使用函數內建立的變數呢？有的，利用**global陳述**。

```
def fi(n):
    global f
    f=[0,1]
    m=2
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f
```

這是一種宣告的方式，告訴直譯器這個變數離開函數還會被用到。

```
>>> ===== RESTART =====
>>>
>>> fi(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> f
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

雖然這是繼續利用區域變數的方法，但是假如該區域變數並不是那麼重要，也就是說程式用到的地方不多，我們不建議使用global陳述。反倒是利用指派到新的變數，會使得變數名稱的使用更為靈活。

```
>>> ===== RESTART =====
>>>
>>> new_fi=fi(10)
>>> new_fi
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

註解

當函數寫的越來越多，我們可能會忘記這個函數在做什麼，那個函數又是做什麼用的，況且如果把程式碼分享給別人，要讀懂純粹的程式碼，有時候不是一件容易的事情。我們可以在程式碼中加入**註解**，凡是#後的文字Python直譯器都會忽略。

```
#計算費伯納西數列
def fi(n):
    f=[0,1] #儲存費伯納西數列
    m=2
    #計算的迴圈
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f
```

往後的程式中，我們都會適當的加入註解。

遞迴的方法

數學上提到費伯納西數列，常常會提到這是**遞迴**定義出來的，而在程式中，遞迴的方法就是設定初始條件後，然後不斷的讓函數呼叫函數本身，藉此計算出結果。

```
#用遞迴的方法計算費伯納西數
def fi_recursion(n):
    if n==1 or n==2:
        return 1
    else:
        return fi_recursion(n-1)+fi_recursion(n-2)
```

初始條件就是第一個及第二個為1，其餘均為前兩個費伯納西數相加的和。因此n等於與n等於2都讓函數傳回1，這就恰好是該費伯納西數的值，而n等於3以後都讓函數回傳fi_recursion(n-1)+fi_recursion(n-2)。

我們先來測試看看。

```
>>> ===== RESTART =====
>>>
>>> fi_recursion(2)
1
>>> fi_recursion(10)
55
>>> fi_recursion(18)
2584
```

結果正確無誤，第二個費伯納西數是2沒錯，第十個是55沒錯，第十八個的確也是2584，可是，這是為什麼函數中呼叫自己本身行得通呢？

應該說是當跑程式的時候，記憶體裡儲存了一張長長的表，譬如這時我們呼叫fi_recursion(2)，函數經過if陳述的條件判斷，很快的就知道fi_recursion(n-2)的值為2。當我們呼

叫fi_recursion(10)的時候，n等於10，既不等於1也不等於2，這時候函數便執行else陳述部份，便會進行fi_recursion(10-1)+fi_recursion(10-2)。

這時候fi_recursion(10)會先儲存在記憶體裡那一張長長的表之中，不知道就先放著。可是fi_recursion(9)與fi_recursion(8)也不知道，那也先放著，擱在儲存在記憶體裡的表。

於是接著會算fi_recursion(8)+fi_recursion(7)與fi_recursion(7)+fi_recursion(6)，但是這些函數的值仍然不知道，只好都先放著。一直拆解計算到fi_recursion(1)+fi_recursion(2)，終於這兩個值都有了，於是我們能夠先得到fi_recursion(3)=fi_recursion(1)+fi_recursion(2)=2，接著fi_recursion(4)也能被計算出來，然後全部從小的數往回加起來，這樣就算出結果了。

聽起來很麻煩不是嗎？遞迴是一種方法，卻不見得是最有效率的方法，因為計算第10個費伯納西數雖然只需要約0.001秒，第20個卻增加到0.008秒，第30個超過1秒，第40個就要超過兩分鐘。

我們之前所用的串列算費伯納西數列長度大約要超過4200個，時間才會超過1秒。嗯，差很多，不是嗎？顯然遞迴用在算數列不是很理想，然而遞迴的方法是一個程式設計的技巧，這種技巧我們將來還會用到。

Note

有關於這些時間測試是在2.4 GHz Intel的CPU上進行的，不同的CPU測試出的時間數據會有所不同，可是即使在相同電腦上，執行時間也可能會有所不同。所以我們應該著重於倍數的差別，遞迴的方法計算第20個約是第10個的時間的八倍，計算第30個已經是第20個的時間的125倍，而在相同時間的一秒，非遞迴算出的個數約是遞迴的140倍。

讓使用者輸入資料

如果我們要讓使用者輸入資料，Python中有兩個內建函數可以做到。

```
>>> x=input("請輸入運算式: ")
請輸入運算式: 1+1+1
>>> x
3
>>> y=raw_input("請輸入任何字元: ")
請輸入任何字元: 1+1+1
>>> y
'1+1+1'
```

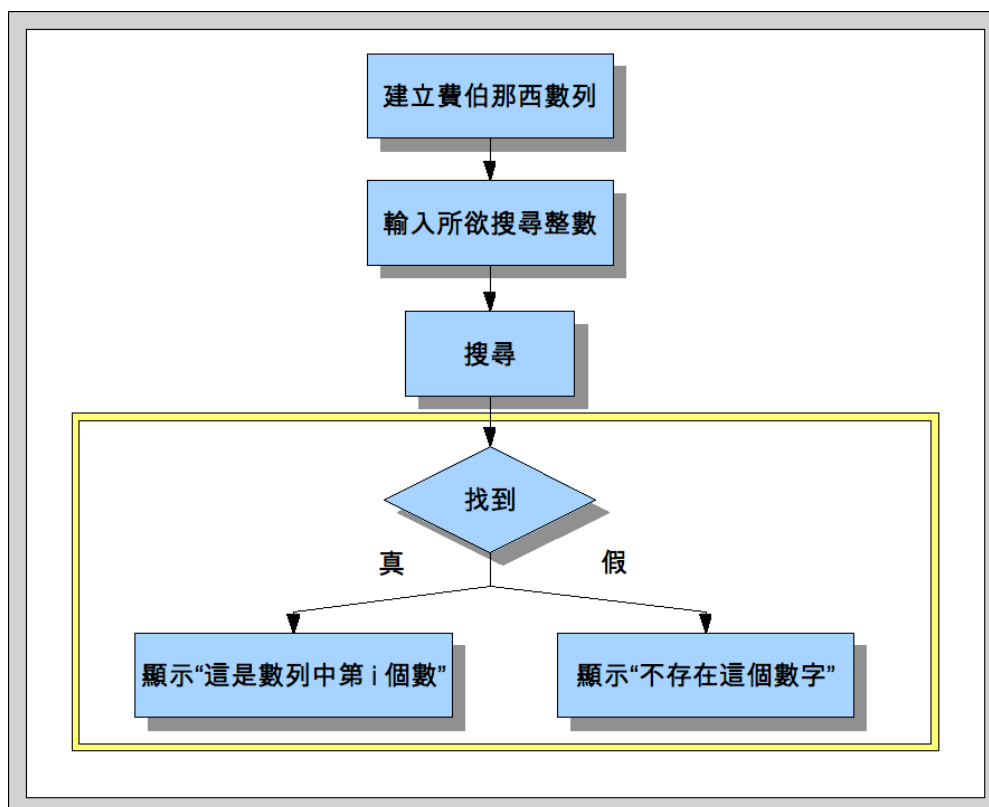
兩個函數的小括弧內所用的字串被稱為**提示字元**，這是說我們可以放入提示使用者輸入資料的任何文字。但是兩者在使用上要特別小心，input函數接受的是運算式，包括數字以及數字與運算子的組合，而raw_input函數會把使用者所輸入的內容都當成字串處理。

我們建議寫程式如果需要使用者輸入資料，可以大都用raw_input函數，因為這樣輸入的結果都一致為字串，之後如果某些資料需要個別處理，其實可以用型態轉換函數轉換成所需的型態。input函數雖然可以鍵入數字，然而如果不小心鍵入其他字元，會造成發生錯誤的情況。

線性搜尋

線性搜尋是種簡單的搜尋方式，逐步比對查找項目與資料是否相同，然後傳回找到與否。我們回到這一章開始所提過想寫的程式，使用者輸入整數，搜尋然後印出結果，我們已經建立過費伯納西數列，我們便以此寫線性搜尋費伯納西數列的程式。

我們將整個程式以流程圖來表示。



首先「建立費伯納西數列」，然後「輸入所欲搜尋整數」，這兩個工作我們都能直接套用寫過或內建的函數，這個部份先列出來。

```

#計算費伯納西數列
def fi(n):
    f=[0,1] #儲存費伯納西數列
    m=2
    #計算的迴圈
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f

fibonacci=fi(20) #建立20個費伯納西數到費伯納西數列
number=int(raw_input("請輸入整數： ")) #輸入要做搜尋的數字
  
```

我們利用變數fibonacci儲存所建立的費伯納西數列，共有二十個費伯納西數，然後用變數number儲存所要求輸入的整數，注意，這裡的raw_input作為int函數的參數，所以變數number的型態已經是整數。

之前我們都只在檔案寫函數的定義，因此載入Python shell並不會主動執行。而現在我們多寫了沒有縮排的兩行程式碼，這些沒有縮排的程式碼載入Python shell會直接執行，現在我們來測試看看。

```

>>> ===== RESTART =====
>>>
請輸入整數： 23
>>> print fibonacci
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
>>> print number
23
  
```


目前程式執行無誤，現在我們要接著寫搜尋及黃框的部份。線性搜尋，顧名思義是一個接著一個的找，用一個while迴圈即可，迴圈內做條件檢查，若是符合則回傳串列的索引值，若是走玩迴圈沒有找到符合的，就傳回**False**。寫成函數需要兩個參數，分別是所要找的數字與被搜尋的資料，當然，這兩個都以儲存在我們已經建立的變數之中。

```
#計算費伯納西數列
def fi(n):
    f=[0,1] #儲存費伯納西數列
    m=2
    #計算的迴圈
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f

#在數列中搜尋特定數字
def search(number, fibonacci):
    i=0
    #搜尋的迴圈
    while i<len(fibonacci):
        if number==fibonacci[i]:
            return i
        i+=1
    return False

fibonacci=fi(20) #建立20個費伯納西數到串列
number=int(raw_input("請輸入整數: ")) #輸入要做搜尋的數字
p=search(number, fibonacci) #搜尋結果
```

測試如下。

```
>>> ===== RESTART =====
>>>
請輸入整數: 23
>>> print p
False
```

的確，23不是費伯納西數，於是變數p所儲存的值為False。接下來我們要完成印出結果，在流程圖中是黃框部份，這裡用if...else...來寫即可。函數search回傳i或是False，若是i等於0，在if陳述的條件判斷等於False，事實上，費伯納西數列是從1開始，所以i等於0也是使用者輸入整數0，因而這樣的條件判斷符合我們的需求。

當然，這個if...else...的複合陳述仍要放進函數當中。

```
#計算費伯納西數列
def fi(n):
    f=[0,1] #儲存費伯納西數列
    m=2
    #計算的迴圈
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f

#在數列中搜尋特定數字
def search(number, fibonacci):
    i=0
    #搜尋的迴圈
    while i<len(fibonacci):
        if number==fibonacci[i]:
            return i
        i+=1
    return False

#印出結果
def result(p):
    if p:
        print "這是數列中第", p, "個數"
    else:
        print "不存在這個數字"

fibonacci=fi(20) #建立20個費伯納西數到串列
number=int(raw_input("請輸入整數: ")) #輸入要做搜尋的數字
p=search(number, fibonacci) #搜尋結果
result(p)
```

測試如下。

```
>>> ===== RESTART =====
>>>
請輸入整數： 21
這是數列中第 8 個數
>>> fibonacci=fi(20)
>>> number=int(raw_input("請輸入整數： "))
請輸入整數： 23
>>> p=search(number, fibonacci)
>>> result(p)
不存在這個數字
>>>
```

雖然第一次的執行會依序載入沒有縮排的程式碼，並且執行，然而如果我們要繼續測試其他的數字，卻要重新做三個指派，呼叫result函數，繼續測試第三個數字，又是重新做三個指派，呼叫result函數，如果繼續測試第四個、第五個……，這會變成一件很麻煩的事情。

那有沒有方法不要這麼麻煩呢？很簡單，我們把這四個動作放入同一個函數中就可以了。

```
#計算費伯納西數列
def fi(n):
    f=[0,1] #儲存費伯納西數列
    m=2
    #計算的迴圈
    while m<=n:
        r=f[m-1]+f[m-2]
        f=f+[r]
        m+=1
    return f

#在數列中搜尋特定數字
def search(number, fibonacci):
    i=0
    #搜尋的迴圈
    while i<len(fibonacci):
        if number==fibonacci[i]:
            return i
        i+=1
    return False

#印出結果
def result(p):
    if p:
        print "這是數列中第", p, "個數"
    else:
        print "不存在這個數字"

def main():
    fibonacci=fi(20) #建立20個費伯納西數到串列
    number=int(raw_input("請輸入整數： ")) #輸入要做搜尋的數字
    p=search(number, fibonacci) #搜尋結果
    result(p)

main()
```

main函數囊括程式執行的順序，因此我們最後呼叫一次，使得載入直譯器就會進行第一次的執行。

```
>>> ===== RESTART =====
>>>
請輸入整數： 21
這是數列中第 8 個數
>>> main()
請輸入整數： 23
不存在這個數字
```

繼續的測試就呼叫main函數，是不是簡單了許多呢？

中文資料處理

上一章我們提過算中文字數的程式，Big5編碼有點麻煩，要做連續兩個字元編碼的比較，況且我們很難熟記中文字的編碼情況。內建函數unicode可以將編碼轉換成Unicode編碼。

```
>>> s="有嘴說別人，無嘴說自己。"
>>> u=unicode(s,"big5")
>>> u
u'\u6709\u5634\u8aaa\u5225\u4eba\u5f0c\u7121\u5634\u8aaa\u81ea\u5df1\u3002'
>>> len(u)
12
```

Unicode編碼符合中文字的情況，因此我們若要計算中文字數不需要做連續兩個字元檢查。

```
#閩南俚語
s=unicode("有嘴說別人，無嘴說自己。","big5")

#計算「嘴」的字數
def words(s):
    count=0 #儲存字數

    #計算迴圈
    for i in s:
        if i==unicode("嘴","big5"):
            count=count+1

    #印出結果
    print "在「", s, "」中，嘴的字數有", count, "個"

words(s)
```

執行結果如下。

```
>>> ===== RESTART =====
>>>
在「有嘴說別人，無嘴說自己。」中，嘴的字數有 2 個
```

其實我們可以把程式寫的更為一般化，像是前面所寫的搜尋程式，我們可以讓使用者輸入中文，然後再輸入所要計算的字，最後印出結果。

```
#輸入中文句子
def sentence():
    return unicode(raw_input("請輸入中文句子："), "big5")

#輸入要找的字
def word():
    return unicode(raw_input("請輸入要找的中文字："), "big5")

#計算字數
def words(s,w):
    count=0

    #計算迴圈
    for i in s:
        if i==w:
            count=count+1
    return count

#印出結果
def result(s,c,w):
    print "在「", s, "」中，", w, "的字數有", c, "個"

def main(s):
    w=word()
    c=words(s,w)
    result(s,c,w)

s=sentence()
main(s)
```

為了有效重複使用輸入的句子，我們將sentence函數獨立出來，沒有放入main函數之中，而將sentence函數所得到的中文字串轉成Unicode編碼後，存入變數s之中，再以s作為main函數的參數。

我們以李白寫的詩句為例。

```
>>> ===== RESTART =====
>>>
請輸入中文句子：鳳凰臺上鳳凰遊，鳳去臺空江自流。
請輸入要找的中文字：鳳
在「 鳳凰臺上鳳凰遊，鳳去臺空江自流。 」中， 鳳 的字數有 3 個
>>> main(s)
請輸入要找的中文字：鳳
在「 鳳凰臺上鳳凰遊，鳳去臺空江自流。 」中， 鳳 的字數有 2 個
```

這比搜尋數列有趣多了，不是嗎？當我們往後學到檔案處理時，這邊寫過的函數還有許多用處。