序

你會不會做實驗呢？

當然，單就程式設計這件事來說，懂得做實驗是件很重要的事，看到一個功能特性，寫個範例實驗看看，不懂某個語法觀念，寫個片段實驗看看，雖然這不是在做化學實驗或物理實驗，不過有時也會想實驗看看，程式會不會就這麼炸了…

那麼，關掉電腦之後，你會不會做實驗呢？

願意從事程式設計的人，照理來說，應該也是個樂於做實驗的人，那麼現在就做個實驗吧！關掉電腦、離開桌子，想想，除了用電腦之外，還能在生活上做些什麼實驗？或者是說，嘗試看看其他的事物，然後看看會有什麼樣的結果。

有沒有對自己的人生做過實驗呢？

這跟電腦上做實驗不同，對人生做實驗需要耐心，沒有人能保證何時能有結果，有時人生中看似毫不相關、甚至是失敗的幾個實驗，卻在某個時間點達成莫名其妙的成果。

有沒有特意對未來的人生投以實驗呢？

你回想起過去曾經有過的幾次實驗，也許算不上實驗，只是過去隨波逐流的過程中，多少嘗試過做些努力，若過去無意識下，曾經對人生做過的實驗，促成了現在的你，那麼現在下意識地對人生做些實驗，未來的自己會是什麼樣子呢？

程式設計很強調 Get your hands dirty，別忘了，人生也需要 Get your hands dirty！

2016.06

chapter

4 流程語法與函式

學習目標

- 認識基本流程語法
- 使用 `for` Comprehension
- 認識函式與變數範圍
- 運用一級函式特性
- 使用 `yield` 建立產生器

4.1 流程語法

現實生活中待解決的事千奇百怪，在電腦發明之後，想要使用電腦解決的需求也是各式各樣：「如果」發生了…，就要…；「對於」…，就一直執行…；「如果」…，就「中斷」…。為了告訴電腦特定條件下該執行的動作，要使用各種條件式來定義程式執行的流程。

4.1.1 if 分支判斷

流程語法中最簡單也最常見的是 if 分支判斷，在 Python 中是這樣寫的：

basic_hello.py

```
import sys

name = 'Guest'
if len(sys.argv) > 1:
    name = sys.argv[1]
print('Hello, {}'.format(name))
```

在 Python 中，一個程式區塊是使用冒號「:」開頭，之後同一區塊範圍要有相同的縮排，不可混用不同空白數量，不可混用空白與 Tab，Python 的建議是使用四個空白作為縮排。

這個範例中，預設的名稱是'Guest'，如果執行時提供命令列引數，則 sys.argv 的長度就會大於 1（記得索引 0 會是.py 檔案名稱），len(sys.argv) > 1 的結果是 True，if 條件成立，因而將 name 設定為使用者提供的命令列引數。一個執行範例如下：

```
>python hello.py
Hello, Guest

>python hello.py Justin
Hello, Justin
```

if 可以搭配 else，在 if 條件不成立時，執行 else 中定義的程式碼，例如寫個判斷數字為奇數或偶數的範例：

basic is_odd.py

```
import sys

number = int(sys.argv[1])
if number % 2:
    print('{} 為奇數'.format(number))
else:
    print('{} 為偶數'.format(number))
```

若是偶數，那麼 number % 2 就會是 0，在 if 判斷式中就會被認定為 False，因此會執行 else 區塊內容。一個執行範例如下：

```
>python is_odd.py 10
10 為偶數

>python is_odd.py 9
9 為奇數
```

Python 的區塊定義方式，可以避免 C/C++、Java 這類 C-like 語言某些不明確的狀況。例如在 C-like 語言中，可能會出現這樣的程式碼：

```
if(condition1)
    if(condition2)
         doSometing();
else
    doOther();
```

乍看之下，else 似乎是與第一個 if 配對，但實際上，else 是與最近的 if 配對，也就是第二個 if。在 Python 中的區塊定義，就沒有這個問題：

```
if condition1:
    if condition2:
         do_something()
else:
    do_other()
```

以上例而言，else 必定是與第一個 if 配對，如果是下例，則 else 必定是與第二個 if 配對：

```
if condition1:
    if condition2:
         do_something()
    else:
         do_other()
```

如果有多重判斷，則可以使用 if..elif..else 結構。例如：

basic grade.py

```
score = int(input('輸入分數：'))
if score >= 90:
    print('得 A')
elif 90 > score >= 80:
    print('得 B')
elif 80 > score >= 70:
    print('得 C')
elif 70 > score >= 60:
    print('得 D')
else:
    print('不及格')
```

一個執行範例如下：

```
>python grade.py
輸入分數：88
得 B
```

在 Python 中有個 if..else 運算式語法。直接來看看怎麼用來改寫先前 is_odd.py 的程式碼：

basic is_odd2.py

```
import sys

number = int(sys.argv[1])
print('{} 為 {}'.format(number, '奇數' if number % 2 else '偶數'))
```

當 if 的條件式成立時，會傳回 if 前的數值，若不成立則傳回 else 後的數值，這個程式的執行結果，與 is_odd.py 是相同的。

4.1.2 while 迴圈

Python 提供 while 迴圈，可根據指定條件式來判斷是否執行迴圈本體，語法如下所示：

```
while 條件式:
    陳述句
else:
    陳述句
```

在條件式成立時，會執行 while 區塊，至於可與 while 搭配的 else，是其他語言幾乎沒有的特色，基本上不建議使用，原因稍後再來說明。先來看個很無聊的遊戲，看誰可以最久不碰到 5 這個數字：

basic lucky5.py

```
import random

number = 0
while number != 5: ◀── ❶如果不是 5 就執行迴圈
    number = random.randint(0, 9) ◀── ❷隨機產生 0 到 9 的數
    print(number)
    if number == 5:
        print('我碰到 5 了....Orz')
```

這個範例的 while 判斷式會判斷 number 是否為 5❶，若判斷為 True 就執行迴圈，random.randint()指定了隨機產生 0 到 9 的整數 ❷。一個參考的執行結果如下：

```
4
5
我碰到 5 了....Orz
```

至於可跟 while 搭配的 else，乍看會以為類似 if…else，誤認為若沒有執行 while 迴圈，就執行 else 的部份，然而實際上，若 while 迴圈正常執行結束，也會執行 else 的部份。

```
>>> while False:
...     print('while')
... else:
...     print('else')
...
else
>>> while num == 0:
...     print('while')
...     num = 1
... else:
...     print('else')
...
while
else
>>>
```

若不想讓 else 執行，必須是 while 中因為 break 而中斷迴圈，底下是求最大公因數的程式，程式碼經過特別安排，或許會比較好懂這個邏輯：

basic_gcd.py

```
print('輸入兩個數字...')

m = int(input('數字 1: '))
n = int(input('數字 2: '))

while n != 0:
   r = m % n
   m = n
   n = r
   if m == 1:
       print('互質')
       break     ◀── break 可用來中斷迴圈
else:
    print("最大公因數：", m)
```

在上面的範例中，如果求出的最大公因數是 1，顯示兩數互質並使用 break，在迴圈中若遇到了 break，迴圈就會中斷，此時就不會執行 else。一個執行結果如下：

```
>python gcd.py
輸入兩個數字...
數字 1: 20
數字 2: 16
最大公因數： 4

>python gcd.py
輸入兩個數字...
數字 1: 10
數字 2: 3
互質
```

在範例程式碼中，我特別使用粗體標示的部份，活像組成了一對 if…else，「if 某條件而執行 break 了，就不會執行 else」，或者反過來想「if 沒有執行 break，就執行 else」，這樣或許會比較能理解 while 與 else 的關係吧！

無論如何，這實在太難懂了，**建議別使用 while 與 else 的形式**，上頭的範例，改成以下寫法才容易理解：

basic gcd2.py

```
print('輸入兩個數字...')

m = int(input('數字 1: '))
n = int(input('數字 2: '))

while n != 0:
   r = m % n
   m = n
   n = r

if m == 1:
    print('互質')
else:
    print("最大公因數：", m)
```

4.1.3 for in 迭代

如果想要循序迭代某個序列，例如字串、list、tuple，則可以使用 for in 陳述句。例如，迭代使用者提供的命令列引數，轉為大寫後輸出。

basic uppers.py

```
import sys
for arg in sys.argv:
    print(arg.upper())
```

要被迭代的序列，是放在 in 之後，對於字串、list、tuple 等具索引特性的序列，for in 會依索引順序逐一取出元素，並指定給 in 之前的變數。一個執行結果如下：

```
>python uppers.py justin monica irene
UPPERS.PY
JUSTIN
MONICA
IRENE
```

如果在迭代的同時，需要同時提供索引資訊，那麼有幾個方式，例如使用 range()函式產生一個指定的數字範圍，使用 for in 進行迭代，再利用迭代出來的數字作為索引。例如：

```
>>> name = 'Justin'
>>> for i in range(len(name)):
...      print(i, name[i])
```

```
...
0 J
1 u
2 s
3 t
4 i
5 n
>>>
```

range()函式的形式是 range(start, stop[, step])，start 省略時，預設是 0，step 是步進值，省略時預設是 1，因此上例中，range(len(name)) 是產生 0 到 5 的數字。

你也可以使用 zip()函式，將兩個序列的各元素，像拉鏈般一對一配對（這就是為什麼它叫 zip 的原因，實際上 zip()可以接受多個序列），產生一個新的 list，當中每個元素都是個 tuple，包括了配對後的元素。

```
>>> list(zip([1, 2, 3], ['one', 'two', 'three']))
[(1, 'one'), (2, 'two'), (3, 'three')]
>>>
```

zip()函式會傳回一個 zip 物件，這個物件實際上還不包括真正配對後的元素，也就是具有惰性求值的特性（range()產生的 range 物件也是）。zip 物件可以使用 for in 迭代，因此若迭代時需要索引資訊，可以如下：

```
name = 'Justin'
for i, c in zip(range(len(name)), name):
    print(i, c)
```

在這邊還使用了 tuple 拆解的特性，將每一對 tuple 中的元素，拆解指定給 i 與 c 變數。

實際上，若真的要迭代時具有索引資訊，建議使用 enumerate()函式而不是 range()函式，enumerate()會傳回 enumerate 物件，一樣具有惰性求值特性，且可使用 for in 迭代，enumerate 可取得 tuple 元素，例如：

```
>>> name = 'Justin'
>>> list(enumerate(name))
[(0, 'J'), (1, 'u'), (2, 's'), (3, 't'), (4, 'i'), (5, 'n')]
>>>
```

因此，迭代時具有索引資訊，也可以使用以下方式：

```
name = 'Justin'
for i, c in enumerate(name):
    print(i, c)
```

預設的情況下，enumerate()會從 0 開始計數，如果想要從其他數字開始，可以在 enumerate()的第二個引數指定。例如從 1 開始：

```
name = 'Justin'
for i, c in enumerate(name, 1):
    print(i, c)
```

實際上，在之後章節你會看到，只要是實作了__iter__()方法的物件，都可以透過__iter__()方法傳回一個迭代器（Iterator），這個迭代器可以使用 for in 來迭代，像是之前的 range、zip、enumerate 物件就是如此。

set 也實作了__iter__()方法，因此可以進行迭代，不過因為 set 是無序的，你只能迭代出元素，但不一定是你想要的順序；至於想要迭代 dict 鍵值的話，可以使用它的 keys()、values()或 items()方法，它們各會傳回 dict_keys、dict_values、dict_items 物件，都實作了__iter__()方法，因此也可以使用 for in 迭代。舉例來說，來同時迭代 dict 的鍵值：

```
>>> passwds = {'Justin' : 123456, 'Monica' : 54321}
>>> for name, passwd in passwds.items():
...     print(name, passwd)
...
Justin 123456
Monica 54321
>>>
```

因為 dict_items 的元素是 tuple，各包括了一對鍵、值，同樣地，這邊使用了 tuple 拆解的特性，將 tuple 的鍵、值拆解給 name 與 passwd 變數。如果直接針對 dict 進行 for in 迭代，預設會進行鍵的迭代。

類似 while 可與 else 配對，for in 也有個與 else 配對的形式，若不想讓 else 執行，必須是 for in 中因為 break 而中斷迭代，不過**建議別使用 for in…else 的形式**，如果真的想看個應用，底下是個例子，可用來判斷指定的數字是否為質數：

basic is_prime.py

```
number = int(input('輸入數字：'))
half = number // 2
for num in range(2, half + 1):
    if number % num == 0:
        print(number, '不是質數')
        break    ◄── break 可用來中斷迭代
else:
    print(number, '是質數')
```

4.1.4 pass、break、continue

有時在某個區塊中，並不想做任何的事情，或者是稍後才會寫些什麼，對於還沒打算寫任何東西的區塊，可以放個 pass。例如：

```
if is_prime:
    print('找到質數')
else:
    pass
```

pass 就真的是 pass，什麼都不做，只是用來維持程式碼結構的完整性，雖然如此，未來也許會常常用到它，因為經常地，你會想要作些小測試，或者是先執行一下程式，看看其他已撰寫好的程式碼是否如期運作，這時 pass 就會派上用場。

至於 break，在先前談 while 與 for in 時已經知道它的功能了，分別可用來中斷 while 迴圈、for in 的迭代，在這邊再提一次，是為了與 continue 對照。在 while 迴圈中遇到 continue 的話，此次不執行後續的程式碼，直接進行下次迴圈，在 for in 迭代遇到 continue 的話，此次不執行後續的程式碼，直接進行下次迭代。

以下是利用 continue 的特性，實作出一個只顯示小寫字母的程式：

basic show_uppers.py

```
text = input('輸入一個字串：')
for letter in text:
    if letter.isupper():
        continue
    print(letter, end='')
```

這個範例在遇到大寫字母時，就會執行 continue，因此該次不會執行 print()。一個執行範例如下：

```
>python show_uppers.py
輸入一個字串：This is a Question!
his is a uestion!
```

4.1.5 for Comprehension

如果使用者輸入的命令列引數是數字，你想要將這些數字全部進行平方運算，該怎麼做呢？現在的你，也許會想出這樣的寫法：

```
import sys

squares = []
for arg in sys.argv[1:]:
    squares.append(int(arg) ** 2)
print(squares)
```

將一個 list 轉為另一個 list，是很常見的操作，Python 針對這類需求，提供了 for Comprehension 語法，你可以如下實現需求：

basic_square.py

```
import sys

squares = [int(arg) ** 2 for arg in sys.argv[1:]]
print(squares)
```

對於 for arg in sys.argv[1:]這部份，其作用是逐一迭代出命令列引數指定給 arg 變數，之後執行 for 左方的 int(arg) ** 2 運算，使用[]含括起來，表示每次迭代的運算結果，會被收集為一個 list。一個執行結果如下：

```
>python square.py 10 20 30
[100, 400, 900]
```

for Comprehension 也可以與條件式結合，這可以構成一個過濾的功能。例如想收集某個 list 中的奇數元素至另一 list，在不使用 for Comprehension 下，可以如下撰寫：

```
import sys

odds = []
for arg in sys.argv[1:]:
    if int(arg) % 2:
```

```
        odds.append(arg)
print(odds)
```

若使用 for Comprehension 的話，可以改寫為以下的程式碼：

basic odds.py

```
import sys

odds = [arg for arg in sys.argv[1:] if int(arg) % 2]
print(odds)
```

在這個例子中，只有在 if 條件式成立時，for 左邊的運算式才會被執行，並收集為最後結果 list 中的元素。一個執行結果如下：

```
>python odds.py 11 8 9 5 4 6 3 2
['11', '9', '5', '3']
```

如果要形成巢狀結構也是可行的，不過建議別太過火，不然可讀性會迅速降低。簡單地將矩陣為一維的 list 倒還不錯：

```
>>> matrix = [
...     [1, 2, 3],
...     [4, 5, 6],
...     [7, 8, 9]
... ]
>>> array = [element for row in matrix for element in row]
>>> array
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

另一個例子是，使用 for Comprehension 來取得兩個序列的排列組合：

```
>>> [letter1 + letter2 for letter1 in 'Justin' for letter2 in 'momor']
['Jm', 'Jo', 'Jm', 'Jo', 'Jr', 'um', 'uo', 'um', 'uo', 'ur', 'sm', 'so', 'sm',
'so', 'sr', 'tm', 'to', 'tm', 'to', 'tr', 'im', 'io', 'im', 'io', 'ir', 'nm',
'no
', 'nm', 'no', 'nr']
>>>
```

當你在 for Comprehension 兩旁放上[]，表示會產生 list，如果資料來源很長，或者資料來源本身是個有惰性求值特性的產生器時，直接產生 list 顯得沒有效率，這時可以在 for Comprehension 兩旁放上()，這樣的話就會建立一個 generator 物件，具有惰性求值特性。

舉個例子來說，Python 中有個 sum()函式，可以計算指定序列的數字加總值，像是若傳遞 sum([1, 2, 3])的話，結果會是 6。如果想計算 1 到 10000

的加總值呢？使用 sum([n for n in range(1, 10001)])是可以達到目的，不過，這會先產生具有 10000 個元素的 list，然後再交給 sum()函式運算，此時可以寫成 sum(n for n in range(1, 10001))，這樣的話就不會有產生 list 的負擔。

這邊其實也在說明，只要寫 n for n in range(1, 10001)就是個產生器運算式了，因此在傳給 sum()函式時，不必再寫成 sum((n for n in range(1, 10001)))，需要加上括號的情況，是在你需要直接參考一個產生器的時候，例如 g = (n for n in range(1, 10001))的情況。

for Comprehension 也可以用來建立 set，只要在 for Comprehension 兩旁放上{}。例如，建立一個 set，其中包括了來源字串中不重複的大寫字母。

```
>>> text = 'Your Right brain has nothing Left. Your Left brain has nothing
Right'
>>> {c for c in text if c.isupper()}
{'Y', 'R', 'L'}
>>>
```

若是想使用 for Comprehension 來建立 dict 實例，也是可行的。例如：

```
>>> names = ['Justin', 'Monica', 'Irene']
>>> passwds = [123456, 654321, 13579]
>>> {name : passwd for name, passwd in zip(names, passwds)}
{'Justin': 123456, 'Irene': 13579, 'Monica': 654321}
>>>
```

上面的 zip 函式，將 names 與 passwds 兩兩相扣在一起成為 tuple，每個 tuple 中的一對元素，會在 for Comprehension 中拆解指定給 name 與 passwd，最後 name 與 passwd 組成 dict 的每一對鍵值。

那麼，可以使用 for Comprehension 建立 tuple 嗎？可以的，不過不是在 for Comprehension 兩旁放上()，這樣的話就會建立一個 generator 物件，而不是 tuple，想要用 for Comprehension 建立 tuple 的話，可以將 for Comprehension 產生器運算式傳給 tuple()。例如：

```
>>> tuple(n for n in range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>>
```

4.2 定義函式

在學會了流程語法之後，你也開始能撰寫一些小程式，依不同的條件計算出不同的結果了，然而可能會發現有些流程你一用再用，老是複製、貼上、修改變數名稱，讓程式碼顯得笨拙而且不易維護，你可以將可重用的流程定義為函式，之後直接呼叫函式來重用這些流程。

4.2.1 使用 def 定義函式

當開始為了重用某個流程，而開始複製、貼上、修改變數名稱時，或者發現到兩個或多個程式片段極為類似，只有當中幾個計算用到的數值或變數不同時，就可以考慮將那些片段定義函式。例如你發現到程式中...

```
# 其他程式片段...
max1 = a if a > b else b
# 其他程式片段...
max2 = x if x > y else y
# 其他程式片段...
```

這時可以定義函式來封裝程式片段，將流程中引用不同數值或變數的部份設計為參數，例如：

```
def max(num1, num2):
    return num1 if num1 > num2 else num2
```

定義函式時要使用 def 關鍵字，max 是函式名稱，num1、num2 是參數名稱，如果要傳回值可以使用 return，如果函式執行完畢但沒有使用 return 傳回值，或者使用了 return 結束函式但沒有指定傳回值，預設就會傳回 None。

這麼一來，原先的程式片段就可以修改為：

```
max1 = max(a, b)
# 其他程式片段...
max2 = max(x, y)
# 其他程式片段...
```

函式是一種抽象，對流程的抽象，在定義了 max 函式之後，客戶端對求最大值的流程，被抽象為 max(x, y) 這樣的函式呼叫，求值流程實作被隱藏了起來。

函式也可以呼叫自身，這稱之為遞迴（Recursion），舉個例子來說，4.1.2 中的 gcd2.py 求最大公因數的流程片段，若定義為函式且用遞迴求解，可以寫成：

func_gcd.py

```
def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m % n)

print('輸入兩個數字...')

m = int(input('數字 1: '))
n = int(input('數字 2: '))

r = gcd(m, n)
if r == 1:
    print('互質')
else:
    print("最大公因數：", r)
```

提示 »» 有不少人覺得遞迴很複雜，其實只要一次只處理一個任務，而且每次遞迴只專注當次的子任務，遞迴其實反而清楚易懂，像這邊的 gcd() 函式，可清楚地看出輾轉相除法的定義。有興趣的話，也可以參考我在〈遞迴的美麗與哀愁〉中的一些想法：

openhome.cc/Gossip/Programmer/Recursive.html

在 Python 中，函式中還可以定義函式，稱為區域函式（Local function），可以使用區域函式將某函式中的演算組織為更小單元，例如，在選擇排序的實作時，每次會從未排序部份，選擇一個最小值放到已排序部份之後，在底下的範例中，尋找最小值的索引時，就以區域函式的方式實作：

func_sele_sort.py

```
import sys

def sele_sort(number):
    # 找出未排序中最小值
    def min_index(left, right):
        if right == len(number):
            return left
        elif number[right] < number[left]:
```

```
            return min(right, right + 1)
        else:
            return min(left, right + 1)

    for i in range(len(number)):
        selected = min_index(i, i + 1)
        if i != selected:
            number[i], number[selected] = number[selected], number[i]

number = [int(arg) for arg in sys.argv[1:]]
sele_sort(number)
print(number)
```

可以看到，區域函式的好處之一，就是可以直接存取包裹它的外部函式之參數或先前宣告之區域變數，如此可減少呼叫函式時引數的傳遞。一個執行結果如下：

```
>python sele_sort.py 1 3 2 5 9 7 6 8
[1, 2, 3, 5, 7, 6, 8, 9]
```

4.2.2 參數與引數

在 Python 中不支援函式重載（Overload）的實作，也就是在同一個名稱空間中，不能有相同的函式名稱。如果定義了兩個函式具有相同名稱，但擁有不同參數個數，則之後定義的函式會覆蓋先前定義的函式。例如：

```
>>> def sum(a, b):
...     return a + b
...
>>> def sum(a, b, c):
...     return a + b + c
...
>>> sum(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() missing 1 required positional argument: 'c'
>>>
```

在上面的例子中，因為後來定義的 sum()有三個參數，這覆蓋了先前定義的 sum()，若只指定兩個參數，就會引發 TypeError，實際上，第一次自行定義 sum()時，也覆蓋了標準程式庫內建的 sum()函式。

參數預設值

雖然不支援函式重載的實作，不過在 Python 中可以使用預設引數來有限度地模仿函式重載。例如：

```
def account(name, number, balance = 100):
    return {'name' : name, 'number' : number, 'balance' : balance}

# 顯示 {'name': 'Justin', 'balance': 100, 'number': '123-4567'}
print(account('Justin', '123-4567'))
# 顯示 {'name': 'Monica', 'balance': 1000, 'number': '765-4321'}
print(account('Monica', '765-4321', 1000))
```

使用參數預設值時，必須小心指定了可變動物件時的一個陷阱，Python 在執行到 def 時，就會依定義建立了相關的資源。來看看下面會有什麼問題？

```
>>> def prepend(elem, lt = []):
...     lt.insert(0, elem)
...     return lt
...
>>> prepend(10)
[10]
>>> prepend(10, [20, 30, 40])
[10, 20, 30, 40]
>>> prepend(20)
[20, 10]
>>>
```

在上例中，你的 lt 預設值設定為[]，由於 def 是個陳述，執行到 def 的函式定義時，就建立了[]，而這個 list 物件會一直存在，如果沒有指定 lt 時，使用的就會一直是一開始指定的 list 物件，也因此，隨著每次呼叫都不指定 lt 的值，你前置的目標 list，都是同一個 list。

想要避免這樣的問題，可以將 prepend()的 lt 參數預設值設為 None，並在函式中指定真正的預設值。例如：

```
>>> def prepend(elem, lt = None):
...     lt = lt if lt else []
...     lt.insert(0, elem)
...     return lt
...
>>> prepend(10)
[10]
>>> prepend(10, [20, 30, 40])
[10, 20, 30, 40]
>>> prepend(20)
[20]
>>>
```

在上面的 prepend()函式中，在 lt 為 None 時，使用[]建立新的 list 實例，這樣就不會有之前的問題。

關鍵字引數

事實上，在呼叫函式時，並不一定要依參數宣告順序來傳入引數，而可以指定參數名稱來設定其引數值，稱之為關鍵字引數。例如：

```
def account(name, number, balance):
    return {'name' : name, 'number' : number, 'balance' : balance}

# 顯示 {'name': 'Monica', 'balance': 1000, 'number': '765-4321'}
print(account(balance = 1000, name = 'Monica', number = '765-4321'))
```

*與**

如果有個函式擁有固定的參數，而你有個序列，像是 list、tuple，只要在傳入時加上*，則 list 或 tuple 中各元素就會自動拆解給各參數。例如：

```
def account(name, number, balance):
    return {'name' : name, 'number' : number, 'balance' : balance}

# 顯示 {'name': 'Justin', 'balance': 1000, 'number': '123-4567'}
print(account(*('Justin', '123-4567', 1000)))
```

像 sum()這種加總數字的函式，事先無法預期要傳入的引數個數，可以在定義函式的參數時使用*，表示該參數接受不定長度引數。例如：

```
def sum(*numbers):
    total = 0
    for number in numbers:
        total += number
    return total

print(sum(1, 2))         # 顯示 3
print(sum(1, 2, 3))      # 顯示 6
print(sum(1, 2, 3, 4))   # 顯示 10
```

傳入函式的引數，會被收集在一個 tuple 中，再設定給 numbers 參數，這適用於參數個數不固定，而且會循序迭代處理參數的場合。

如果有個 dict，打算依鍵名稱，指定給對應的參數名稱，可以在 dict 前加上**，這樣 dict 中各對鍵、值，就會自動拆解給各參數。例如：

```
def account(name, number, balance):
    return {'name' : name, 'number' : number, 'balance' : balance}

params = {'name' : 'Justin', 'number' : '123-4567', 'balance' : 1000}
# 顯示 {'name': 'Justin', 'balance': 1000, 'number': '123-4567'}
print(account(**params))
```

如果你的參數個數越來越多，而且每個參數名稱皆有其意義，像是 def ajax(url, method, contents, datatype, accept, headers, username, password)，這樣的函式定義不但醜陋，呼叫時也很麻煩，單純只搭配關鍵字引數或預設引數，也不見得能改善多少，將來若因為需求而必須增減參數，也會影響函式的呼叫者，勢必得逐一修改影響到的程式，造成未來程式擴充時的麻煩。

這個時候，可以試著使用**來定義參數，讓指定的關鍵字引數收集為一個 dict。例如：

```
def ajax(url, **user_settings):
    settings = {
        'method' : user_settings.get('method', 'GET'),
        'contents' : user_settings.get('contents', ''),
        'datatype' : user_settings.get('datatype', 'text/plain'),
        # 其他設定 ...
    }
    print('請求 {}'.format(url))
    print('設定 {}'.format(settings))

ajax('http://openhome.cc', method = 'POST', contents = 'book=python')
my_settings = {'method' : 'POST', 'contents' : 'book=python'}
ajax('http://openhome.cc', **my_settings)
```

像這樣定義函式就顯得優雅許多，呼叫函式時可使用關鍵字引數，在函式內部也可實現預設引數的效果，這樣的設計在未來程式擴充時比較有利，因為若需增減參數，只需修改函式的內部實作，函式的呼叫者不會受到影響。

在上面的函式定義中是假設，url 為每次呼叫時必須指定的參數，而其他參數可由使用者自行決定是否指定，如果已經有個 dict 想作為引數，也可以 ajax('http://openhome.cc', **my_settings)這樣使用**進行拆解。

可以在一個函式中，同時使用*與，如果想要設計一個函式接受任意引數，就可以加以運用。**例如：

```
>>> def some(*arg1, **arg2):
...     print(arg1)
...     print(arg2)
...
>>> some(1, 2, 3)
(1, 2, 3)
{}
>>> some(a = 1, b = 22, c = 3)
()
{'a': 1, 'c': 3, 'b': 22}
>>> some(2, a = 1, b = 22, c = 3)
(2,)
{'a': 1, 'c': 3, 'b': 22}
>>>
```

4.2.3 一級函式的運用

在 Python 中，函式不單只是個定義，還是個值，你定義的函式會產生一個函式物件，它是 function 的實例，既然函式是物件，也就可以指定給其他的變數。例如：

```
>>> def max(num1, num2):
...     return num1 if num1 > num2 else num2
...
>>> maximum = max
>>> maximum(10, 5)
10
>>> type(max)
<class 'function'>
>>>
```

上面在定義了 max()函式之後，透過 max 名稱將函式物件指定給 maximum 名稱，無論透過 max(10, 5)或者 maximum(10, 5)，結果都是呼叫了它們參考的函式物件。

函式跟數值、list、set、dict、tuple 等一樣，都被 Python 視為一級公民來對待，可以自由地在變數、函式呼叫時指定，因此具有這樣特性的函式，也被稱一級函式（First-class function），函式代表著某個可重用流程的封裝，當它可以作為值傳遞時，就表示你可以將某個可重用流程進行傳遞，這是個極具威力的功能。

filter_lt()函式

如果你有個 lt = ['Justin', 'caterpillar', 'openhome']，現在打算過濾出字串長度大於 6 的元素，一開始你可以寫出如下的程式碼：

```
lt = ['Justin', 'caterpillar', 'openhome']
result = []
for elem in lt:
    if len(elem) > 6:
        result.append(elem)
print(result)
```

你可能會多次進行這類的比較，因此定義出函式，以重用這個流程：

```
def len_greater_than_6(lt):
    result = []
    for elem in lt:
        if len(elem) > 6:
            result.append(elem)
    return result

lt = ['Justin', 'caterpillar', 'openhome']
print(len_greater_than_6(lt))
```

那麼，如果想要過濾長度小於 5 呢？在急著寫個 len_less_than_5()函式之前，先仔細想想，這類過濾某清單而後取得另一清單的流程，你寫過幾次呢？每次其實只有過濾的條件不同，其他流程都是相同的，如果將重複的流程提取出來，封裝為函式如何呢？

func_filter_demo.py

```
def filter_lt(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

def len_greater_than_6(elem):
    return len(elem) > 6

def len_less_than_5(elem):
    return len(elem) < 5

def has_i(elem):
    return 'i' in elem

lt = ['Justin', 'caterpillar', 'openhome']
```

```
print('大於 6：', filter_lt(len_greater_than_6, lt))
print('小於 5：', filter_lt(len_less_than_5, lt))
print('有個 i：', filter_lt(has_i, lt))
```

可以看到，將重複的流程提取出來後，你就可以呼叫函式，然後每次給予不同的函式來設定過濾條件。就目前來說，特別為 len(elem) > 6、len(elem) < 5、'i' in elem 使用 def 定義了 len_greater_than_6()、len_less_than_5()、has_i()，看起來有點小題大作，然後好處是，只要看 filter_lt(len_greater_than_6, lt)、filter_lt(len_less_than_5, lt)、filter_lt(has_i, lt)，就可以很清楚地知道程式碼的目的。這個範例的執行結果如下：

```
大於 6： ['caterpillar', 'openhome']
小於 5： []
有個 i： ['Justin', 'caterpillar']
```

當然，你可能覺得 len_greater_than_6()不夠通用，若真如此，也可以修改一下範例，讓它更通用些：

func_filter_demo2.py

```
def filter_lt(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

def len_greater_than(num):
    def len_greater_than_num(elem):
        return len(elem) > num
    return len_greater_than_num

lt = ['Justin', 'caterpillar', 'openhome']
print('大於 5：', filter_lt(len_greater_than(5), lt))
print('大於 7：', filter_lt(len_greater_than(7), lt))
```

這次在 len_greater_than() 中定義了一個區域函式 len_greater_than_num()，之後將區域函式傳回，傳回的函式接受一個參數 elem，而本身帶有呼叫 len_greater_than()時傳入的 num 參數值，因此，len_greater_than(5) 傳回的函式相當於進行 len(elem) > 5，而

len_greater_than(7) 傳回的函式相當於進行 len(elem) > 7，像這樣呼叫函式傳回（內部）另一個函式，也是函式作為一級公民的語言中常見的應用。

map_lt() 函式

類似地，如果你想將 lt 的元素全部轉為大寫後傳回新的清單，一開始可能會直接撰寫以下的流程：

```
lt = ['Justin', 'caterpillar', 'openhome']
result = []
for ele in lt:
    result.append(ele.upper())
print(result)
```

同樣地，將清單元素轉換為另一組清單，也是你寫過無數次的操作，何不將其中重複的流程抽取出來呢？

func_map_demo.py

```
def map_lt(mapper, lt):
    result = []
    for ele in lt:
        result.append(mapper(ele))
    return result

lt = ['Justin', 'caterpillar', 'openhome']
print(map_lt(str.upper, lt))
print(map_lt(len, lt))
```

可以看到，將重複的流程提取出來後，你就可以呼叫函式，然後每次給予不同的函式來設定對應轉換的方式。當轉換的函式早就定義好了，使用 map_lt 這樣的函式就很方便，就像這邊使用了 Python 標準程式庫中的 str.upper 與 len。這個範例的執行結果如下：

```
>python map_demo.py
['JUSTIN', 'CATERPILLAR', 'OPENHOME']
[6, 11, 8]
```

filter()、map()、sorted()函式

實際上，Python 就內建有 filter()、map()函式可以直接取用，在 Python 3，map()、filter() 傳回的實例並不是 list，分別是 map 與 filter 物件，都具有惰性求值的特性。底下來個簡單的示範：

func_filter_map_demo.py

```
def len_greater_than(num):
    def len_greater_than_num(elem):
        return len(elem) > num
    return len_greater_than_num

lt = ['Justin', 'caterpillar', 'openhome']
print(list(filter(len_greater_than(6), lt)))
print(list(map(len, lt)))
```

基本上，filter()、map()能做得到的，for Comprhension 基本上都做得到，大多情況下，for Comprhension 比較常見，不過有時透過適當的命名，使用 filter()、map()會有比較好的可讀性，像是 map(len, lt)就是一個例子。

再來看個一級函式傳遞的例子，到目前為止，經常會使用 list、tuple 等有序結構，有時會想將其中的元素進行排序，這時可以使用 sorted()函式，它可以針對你指定的方式進行排序。例如：

```
>>> sorted([2, 1, 3, 6, 5])
[1, 2, 3, 5, 6]
>>> sorted([2, 1, 3, 6, 5], reverse = True)
[6, 5, 3, 2, 1]
>>> sorted(('Justin', 'openhome', 'momor'), key = len)
['momor', 'Justin', 'openhome']
>>> sorted(('Justin', 'openhome', 'momor'), key = len, reverse = True)
['openhome', 'Justin', 'momor']
>>>
```

sorted()會傳回新的 list，其中包含了排序後的結果，key 參數可用來指定針對什麼特性來迭代，例如在指定 len()函式時，每個元素都會傳入 len()運算，得到的長度值再作為排序依據。

如果是可變動的 list，本身也有個 sort()方法，這個方法會直接在 list 本身排序，不像 sorted()方法會傳回新的 list。例如：

```
>>> lt = [2, 1, 3, 6, 5]
>>> lt.sort()
>>> lt
[1, 2, 3, 5, 6]
>>> lt.sort(reverse = True)
>>> lt
[6, 5, 3, 2, 1]
>>> names = ["Justin", "openhome", "momor"]
>>> names.sort(key = len)
>>> names
['momor', 'Justin', 'openhome']
>>>
```

Python 標準程式庫中，還有許多可接受函式值（或者傳回函式）的函式，本書之後的章節也有機會看到一些應用。

4.2.4 lambda 運算式

在之前的 filter_demo.py 中，大費周章地為 len(elem) > 6、len(elem) < 5、'i' in elem 使用 def 定義了 len_greater_than_6()、len_less_than_5()、has_i()，它們的函式本體其實都只很簡單，只有一句簡單的運算，對於這類情況，可以考慮使用 lambda 運算式。例如：

func_filter_demo3.py

```
def filter_lt(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

lt = ['Justin', 'caterpillar', 'openhome']
print('大於 6：', filter_lt(lambda elem: len(elem) > 6, lt))
print('小於 5：', filter_lt(lambda elem: len(elem) < 5, lt))
print('有個 i：', filter_lt(lambda elem: 'i' in elem, lt))
```

在 lambda 關鍵字之後定義的是參數，而冒號「:」之後定義的是函式本體，運算的結果會作為傳回值，不需要加上 return，像 lambda elem: len(elem) > 6 這樣的 lambda 運算式會建立 function 實例，也就是一個函式，有時臨時只是需要個小函式，使用 lambda 就很方便。

4.3 重點複習

在 Python 中，一個程式區塊是使用冒號「:」開頭，之後同一區塊範圍要有相同的縮排，不可混用不同空白數量，不可混用空白與 Tab，Python 的建議是使用四個空白作為縮排。

在 Python 中有個 if..else 運算式語法，當 if 的條件式成立時，會傳回 if 前的數值，若不成立則傳回 else 後的數值。Python 提供 while 迴圈，可根據指定條件式來判斷是否執行迴圈本體。如果想要循序迭代某個序列，例如字串、list、tuple，則可以使用 for in 陳述句。

range()函式的形式是 range(start, stop[, step])，start 省略時，預設是 0，step 是步進值，省略時預設是 1，因此上例中，range(len(name)) 是產生 0 到 5 的數字。可以使用 zip()函式，將兩個序列的各元素，像拉鏈般一對一配對，實際上 zip()可以接受多個序列。若真的要迭代時具有索引資訊，使用 enumerate()函式可能是最方便的。

只要是實作了__iter__()方法的物件，都可以使用 for in 來迭代，只要是實作了__iter__()方法的物件，都可以透過__iter__()方法傳回一個迭代器，這個迭代器可以使用 for in 來迭代。

set 也實作了__iter__()方法，因此可以進行迭代，想要迭代 dict 鍵值的話，可以使用它的 keys()、values()或 items()方法，它們各會傳回 dict_keys、dict_values、dict_items 物件，都實作了__iter__()方法，因此也可以使用 for in 迭代。

有時在某個區塊中，並不想做任何的事情，或者是稍後才會寫些什麼，對於還沒打算寫任何東西的區塊，可以放個 pass。

break 可分別可用來中斷 while 迴圈、for in 的迭代。在 while 迴圈中遇到 continue 的話，此次不執行後續的程式碼，直接進行下次迴圈，在 for in 迭代遇到 continue 的話，此次不執行後續的程式碼，直接進行下次迭代。

將一個 list 轉為另一個 list，是很常見的操作，Python 針對這類需求，提供了 for Comprehension 語法。for Comprehension 也可以與條件式結合，這可以構成一個過濾的功能。

在 for Comprehension 兩旁放上[]，表示會產生 list，如果資料來源很長，或者資料來源本身是個有惰性求值特性的產生器時，直接產生 list 顯得沒有效率，這時可以在 for Comprehension 兩旁放上()，這樣的話就會建立一個 generator 物件，具有惰性求值特性。

for Comprehension 也可以用來建立 set，只要在 for Comprehension 兩旁放上{}。若是想使用 for Comprehension 來建立 dict 實例，也是可行的。想要用 for Comprehension 建立 tuple 的話，可以將 for Comprehension 產生器運算式傳給 tuple()。

如果函式執行完畢但沒有使用 return 傳回值，或者使用了 return 結束函式但沒有指定傳回值，預設就會傳回 None。

在 Python 中，函式中還可以定義函式，稱為區域函式（Local function）。在 Python 中可以使用預設引數、關鍵字引數。

如果有個函式擁有固定的參數，而你有個序列，像是 list、tuple，只要在傳入時加上*，則 list 或 tuple 中各元素就會自動拆解給各參數。可以在定義函式的參數時使用*，表示該參數接受不定長度引數。

如果有個 dict，打算依鍵名稱，指定給對應的參數名稱，可以在 dict 前加上**，這樣 dict 中各對鍵、值，就會自動拆解給各參數。可以試著使用**來定義參數，讓指定的關鍵字引數收集為一個 dict。

可以在一個函式中，同時使用*與**，如果想要設計一個函式接受任意引數，就可以加以運用。

在 Python 中，函式不單只是個定義，還是個值，你定義的函式會產生一個函式物件，它是 function 的實例，既然函式是物件，也就可以指定給其他的變數。

有時會想將其中的元素進行排序，這時可以使用 sorted()函式。如果是可變動的 list，本身也有個 sort()方法，這個方法會直接在 list 本身排序。

函式本體很簡單，只有一句簡單運算的情況，可以考慮使用 lambda 運算式。

一個名稱在指定值時，就可以成為變數，並建立起自己的作用範圍，在取用一個變數時，會看看目前範圍中是否有指定的變數名稱，若無則向外尋找。

變數可以在內建（Builtin）、全域（Global）、外包函式（Endosing function）、區域函式（Local functon）中尋找或建立。Python 中的全域，實際上是以模組檔案為界。

dir()函式可用來查詢指定的物件上可取用的名稱。Python 中可以直接使用的函式，其名稱實際上是在 builtins 模組之中。在 Python 中有個 locals()函式，可用來查詢區域變數的名稱與值。Python 中還有個 global()，可以取得全域變數的名稱與值，當你在全域範圍呼叫 locals()時，取得結果與 global()是相同的。

如果對變數指定值時，希望是針對全域範圍的話，可以使用 global 宣告。在 Python 3 中新增了 nonlocal，可以指明變數並非區域變數，請直譯器依照區域函式、外包函式、全域、內建的順序來尋找變數，就算是指定運算時，也要求是這個順序。

可以在函式中使用 yield 來產生值，表面上看來，yield 有點像是 return，不過函式並不會因為 yield 而結束，只是將流程控制權讓給函式的呼叫者。yield 實際上是個運算式，除了呼叫產生器的__next__()方法，取得 yield 的右側指定值之外，還可以透過 send()方法指定值，令其成為 yield 運算結果。

課後練習

實作題

1. 在三位的整數中，153 可以滿足 $1^3 + 5^3 + 3^3 = 153$，這樣的數稱之為阿姆斯壯（Armstrong）數，試以程式找出所有三位數的阿姆斯壯數。

2. Fibonacci 為 1200 年代歐洲數學家，在他的著作中提過，若一隻兔子每月生一隻小兔子，一個月後小兔子也開始生產。起初只有一隻兔子，一個月後有兩隻兔子，二個月後有三隻兔子，三個月後有五隻兔子...，也就是每個月兔子總數會是 1、1 、2、3、5、8、13、21、34、55、89.....，這就是費氏數列，可用公式定義如下：

   ```
   fn = fn-1 + fn-2    if n > 1
   fn = n              if n = 0, 1
   ```

   請撰寫程式，可讓使用者輸入想計算的費式數個數，由程式全部顯示出來。例如：

   ```
   求幾個費式數？10
   0 1 1 2 3 5 8 13 21 34
   ```

3. 請撰寫一個簡單的洗牌程式，可在文字模式下顯示洗牌結果。例如：

   ```
   桃 9  心 10 梅 4  桃 J  磚 5  梅 10 梅 K  磚 9  梅 J  磚 2  磚 A  心 6  心 5
   桃 8  梅 2  磚 6  梅 3  梅 7  梅 A  心 4  心 J  心 8  心 Q  梅 6  磚 J  心 K
   桃 6  磚 8  心 7  桃 5  磚 K  磚 3  心 A  桃 7  梅 9  心 9  桃 3  磚 10 心 3
   桃 A  桃 4  桃 2  桃 10 桃 Q  磚 7  梅 8  心 2  梅 Q  梅 5  磚 Q  桃 K  磚 4
   ```

4. 試著使用 `for` Comprehension 來找出周長為 24，每個邊長都為整數且不超過 10 的直角三角形邊長。