

UNIT 7 pythonTM

Function and Data Structure

張傑帆 Chang, Jie-Fan



NTU CSIE

OUTLINE

- 函數定義
- 函數傳遞參數與回傳值
- 區域變數與全域變數



函數

- 包函許多程式碼的一行程式 (用來代表某種功能)
- 當程式碼太多且會重覆出現時，可以將部份程式碼抽離主程式，寫成一段函式，有需要用到時再去呼叫它
- 函數是經過組織且可重複使用的程式碼，是能用來實現單一或是相關聯的程式碼
- 巧妙的運用函數可以提高程式碼的重複利用率，避免一樣的事情卻需要重複寫好幾次的程式碼來執行。
- 這跟迴圈的概念有點像，都是在重複利用程式碼；不同的地方在於函數是在需要時才呼叫使用

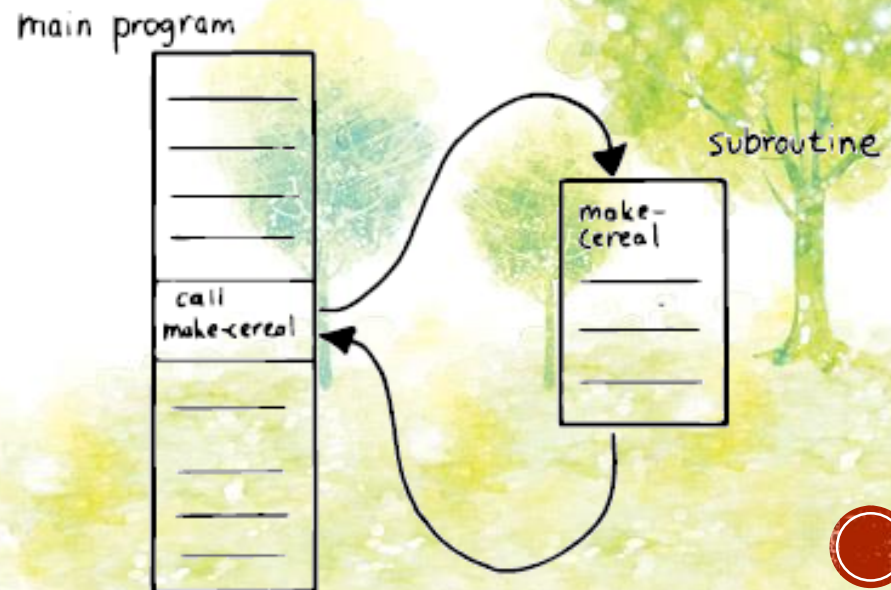


函數 (FUNCTION) 物件 (OBJECT)

- Python 已經內建許多好用的函數，例如 `input()`、`print()` 等等
- Python 內建函數是常用到的功能，如果想要客製化的功能，還是可以 **自己動手做**，執行一些工作，或是進行計算
- **定義函數** 使用關鍵字 (keyword) **def**，其後空一格接函數的識別字 (identifier) **名稱加小括弧()**，然後 **冒號**：
- 其從屬內容必定要縮排如：

def function_name():

縮排 **#do something**



定義函式 DEFINING FUNCTIONS

函式定義從“def”開始

函式名稱 傳入參數

```
def get_final_answer(filename):
```

```
    """說明文字"""
```

縮排

```
    line1
```

```
    line2
```

```
    return total_counter
```

冒號

縮排很重要

若是沒有縮排的話

會被認為是不屬於此函式的敘述

關鍵字‘return’代表要回傳給呼叫者的值

- 函式內的變數是獨立的，不會受外在變數的影響。
- 不需標頭檔，也不需宣告函式回傳值與參數型態



函式使用練習

Hello_func.py

```
1 def hello():  
2     print('hello',1)  
3     print('hello',2)  
4     print('hello',3)  
5  
6     print("before hello")  
7     hello()  
8     print("after hello")  
9     print(hello())  
10    print("last hello")  
11    print(hello)
```

>>>

before hello

hello 1

hello 2

hello 3

after hello

hello 1

hello 2

hello 3

None

last hello

<function hello at 0x02D7F108>

>>>

沒有回傳值的函數，呼叫 (call) 該函數會
自動回傳 **None** 物件 (object)



傳入參數與回傳值

- 函數的小括弧可以定義 **參數** (parameter)
是提供給函數計算的 **數值** (value) ，或是 **物件** (object)
- 函數 (function) 可以有 **回傳值** (return value)
回傳值可以是函數最後的 **計算結果** 。

Function.py

```
1 def F(x, y):  
2     ans = 2*x+y  
3     return ans  
4  
5  
6  
7  
8 num1 = eval(input())  
9 num2 = eval(input())  
10 result = F(num1, num2)  
11 print("F(%d, %d) = %d" % (num1, num2, result))
```

>>>
20
50
F(20, 50) = 90
>>>



- 注意先有函數定義，才可進行函數呼叫

Call_error.py

```
1 F(10, 20)
2
3 def F(x, y):
4     ans = 2*x+y
5     return ans
```

```
>>>
Traceback (most recent call last):
  File "D:/Dropbox/NTU/系統訓練班/Python
odule>
    F(10, 20)
NameError: name 'F' is not defined
>>>
```

- 因為 Python 直譯器從頭一行一行的解譯程式原始碼 (source code)，F() 既非 Python 的內建名稱，也還沒解譯到底下定義的部份，因此直譯器直接發生 **NameError**，終止程式的進行



課堂練習

- 用函數撰寫一程式，令其計算 $n*m$
- 包含了兩個傳入值 n, m
- 練習一
 - 不需回傳值，直接在函數內將相乘的結果印出
- 練習二
 - 有回傳值，將相乘的結果回傳給呼叫函式後印出



小練習

- 練習一：請試寫一函式可回傳 1 加到 n 的結果
- 練習二：請試寫另一函式可回傳 1 乘到 n ($n!$) 的結果
- 若呼叫需放參數的 **function** 時不放參數會發生什麼事？



函數 預設參數

- 函數的參數可以提供預設值 (default argument)
- 可以在參數列 (parameter list) 直接將參數指派數值

Def_arg.py

```
1 def PrintData(uid, name='No name', age=0):  
2     print('The user id:', uid)  
3     print('The username:', name)  
4     print('The user age:', age)  
5  
6 PrintData(1, 'John', 20)  
7 PrintData(2, 'May')  
8 PrintData(3)
```

>>>

The user id: 1

The username: John

The user age: 20

The user id: 2

The username: May

The user age: 0

The user id: 3

The username: No name

The user age: 0

>>>

- 如果只執行 PrintData() 呢？



函數回傳值

- 函數定義中也沒有限制回傳值的數量
- 可在 `return` 陳述中以逗號分隔回傳值

Multi_return.py

```
1 import random
2
3 def rand2int():
4     rand1 = random.randint(1,10)
5     rand2 = random.randint(1,10)
6     return rand1, rand2
7
8
9 a,b = rand2int()
10 print(a)
11 print(b)
12 print(rand2int())
```

```
>>>
4
9
(3, 1)
>>>
```



小練習

- 請試寫一函數可同時回傳 $n!$ 與 $\sum 1 \sim n$ 的結果
- 並改成即使呼叫時不加入引數也不會錯誤



函數範例

Call_error2.py

```
1 def F(x, y):  
2     ans = 2*x+y  
3     return ans  
4 print(F(10,20))  
5 print(x, y) #這行可以執行嗎？會發生什麼事？
```



- x, y 為 $F()$ 函數內的區域變數 (local variable)
- $F()$ 之外的地方無法存取 x, y 的值
- 直譯器在 $F()$ 函數的區塊內才認得變數 x, y
- 離開函數直譯器便不認識這個名稱。

```
Traceback (most recent call last):  
  File "D:/Dropbox/NTU/系統訓練班/Python/  
odule>  
    print(x,y)  
NameError: name 'x' is not defined  
>>>
```



小練習-進階版

- 排列組合應用
- 從**n**個相異物中不重覆取出**m**個之組合數

$${}_nC_m = \binom{n}{m} = \frac{{}_nP_m}{m!} = \frac{n!}{m!(n-m)!}$$

- 從**n**個相異物中不重覆取出**m**個之排列數

$${}_nP_m = \frac{n!}{(n-m)!} = n(n-1)(n-2) \dots (n-m+1)$$



- 呼叫 (call) 函數時
- 參數必須按照順序提供，若沒有按照順序，
- 就需要把參數名稱打出來，
- 沒有給預設值的參數必須給值

Def_arg.py

```
1 def PrintData(uid, name='No name', age=0):  
2     print('The user id:',uid)  
3     print('The username:',name)  
4     print('The user age:',age)  
5  
6 PrintData(age=20,name='John',uid=0)  
7 PrintData(name='May',uid=1)
```

>>>

The user id: 0

The username: John

The user age: 20

The user id: 1

The username: May

The user age: 0

>>>

- 如果執行 PrintData(name='Dany') 呢？



函數 不定個數參數

- 函數 (function) 可以有不定個數的參數 (parameter)，也就是可以在參數列 (parameter list) 提供任意長度的參數個數

Multi_arg.py

```
1 def function_name(*arguments, **keywords):  
2     #dosomething  
3     return something
```

- ***arguments** 就是參數識別字 (identifier) 前面加上一個星號，當成一組序對 (tuple)
- ****keywords** 參數識別字前面加上兩個星號，當成一組字典 (dictionary)



不定個數參數 範例

Multi_arg_tuple.py

```
1 def hello(*names):
2     for n in names:
3         print("Hello, %s."%n)
4     hello("Tom", "Peter", "Bob", "Rain")
```

```
>>>
Hello, Tom.
Hello, Peter.
Hello, Bob.
Hello, Rain.
```

```
Hello Bob, you are 33 years old
Hello John, you are 25 years old
Hello Tom, you are 20 years old
>>>
```

Multi_arg_dict.py

```
1 def hello(**names):
2     for n in names:
3         print("Hello",n,end=', ')
4         print("you are",names[n],"years old")
5     hello(John=25, Tom=20, Bob=33)
```

就像宣告變數一般，不得為數字



傳遞數值參數

- 傳遞的參數內容會被copy到函式用來接收參數的變數中
- 事實上是兩個不同的變數
- 所以函式中改變參數數值時，原來呼叫處的數值並不會改變

PassValue.py

```
1 def passValue(args):  
2     print(' before change:', args)  
3     args = 999  
4     print(' after change:', args)  
5  
6 num = 10  
7 print('before call:', num)  
8 passValue(num)  
9 print('after call:', num)
```

```
>>>  
before call: 10  
before change: 10  
after change: 999  
after call: 10  
>>>
```



傳遞容器參數

- 函式在做引數傳遞呼叫函式的引數是容器時
- 會將記憶體位址傳給被呼叫函式內的引數
- 函式中對容器所做的改變會影響原容器

PassList.py

```
1 def passList(lst):
2     print(' before sort:',lst)
3     lst.sort()
4     print(' after sort:',lst)
5
6 list1 = [44,63,7,22,5,1]
7 print('before call:',list1)
8 passList(list1)
9 print('after call:',list1)
```

```
>>>
before call: [44, 63, 7, 22, 5, 1]
before sort: [44, 63, 7, 22, 5, 1]
after sort: [1, 5, 7, 22, 44, 63]
after call: [1, 5, 7, 22, 44, 63]
>>>
```

pass by reference



fillCup()

pass by value



fillCup()

傳遞容器參數

- 把容器當參數來傳遞時，如果傳遞的只是容器的**某欄位**中的一個變數，那和傳普通變數沒有什麼分別

Ex:

```
list1 = [1,2,3,4,5]  
func(list1[0])
```

- 如果傳的是**整個容器**，傳遞的東西會是容器的位址

Ex:

```
list1 = [1,2,3,4,5]  
func(list1)
```



課堂練習

- 令使用者輸入一有**N**個數字的數列
- 以-1結束輸入
- 並存入**list**中(不包含-1)
- 再試寫一個函數傳入這個**N**個數字的**list**
- 並且回傳傳入**list/tuple**中**最大**的數
- 並印出此**list**



回家作業-矩陣相乘

- 宣告三個二維 **list** 來模擬數學矩陣相乘
- 矩陣**a**和矩陣**b**相乘
- 將所得乘積置入矩陣**c**
- 並將其包裝成一**function** 將 **c** 回傳回去

```
Matrix a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
Matrix b = [[2, 4, 6], [8, 10, 12], [14, 16, 18]]  
Matrix c = Matrix a x matrix b.  
The Result matrix c = [[60, 72, 84], [132, 162, 192], [204, 252, 300]]  
>>>
```



使用者輸入

■ 令使用者輸入資料產生二維矩陣

使用者輸入

```
row1 = input("請輸入2x3矩陣的第一列(各元素請用半形空白作區隔):")
```

```
row2 = input("請輸入2x3矩陣的第二列(各元素請用半形空白作區隔):")
```

分割

```
row1 = row1.split(" ")
```

```
row2 = row2.split(" ")
```

建立二維串列

```
matrix = [row1, row2]
```

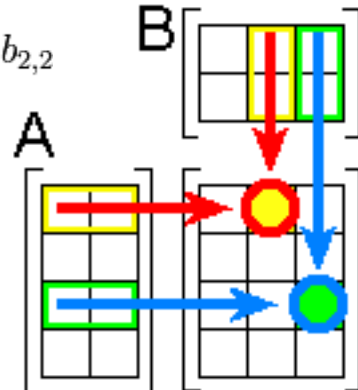


矩陣相乘的公式如下：

$$(AB)_{1,2} = \sum_{r=1}^2 a_{1,r}b_{r,2} = a_{1,1}b_{1,2} + a_{1,2}b_{2,2}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix} = \begin{bmatrix} 60 & 72 & 84 \\ 132 & 162 & 192 \\ 204 & 252 & 300 \end{bmatrix}$$

$$(AB)_{3,3} = \sum_{r=1}^2 a_{3,r}b_{r,3} = a_{3,1}b_{1,3} + a_{3,2}b_{2,3}$$



$$\begin{bmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \\ a[2][0] & a[2][1] & a[2][2] \end{bmatrix} \times \begin{bmatrix} b[0][0] & b[0][1] & b[0][2] \\ b[1][0] & b[1][1] & b[1][2] \\ b[2][0] & b[2][1] & b[2][2] \end{bmatrix} = \begin{bmatrix} c[0][0] & c[0][1] & c[0][2] \\ c[1][0] & c[1][1] & c[1][2] \\ c[2][0] & c[2][1] & c[2][2] \end{bmatrix}$$

$$c[i][j] += a[i][k] \times b[k][j]$$

$$c[0][0] = a[0][0] \times b[0][0] + a[0][1] \times b[1][0] + a[0][2] \times b[2][0]$$

$$c[0][1] = a[0][0] \times b[0][1] + a[0][1] \times b[1][1] + a[0][2] \times b[2][1]$$

$$c[0][2] = a[0][0] \times b[0][2] + a[0][1] \times b[1][2] + a[0][2] \times b[2][2]$$

$$c[1][0] = a[1][0] \times b[0][0] + a[1][1] \times b[1][0] + a[1][2] \times b[2][0]$$

$$c[1][1] = a[1][0] \times b[0][1] + a[1][1] \times b[1][1] + a[1][2] \times b[2][1]$$

$$c[1][2] = a[1][0] \times b[0][2] + a[1][1] \times b[1][2] + a[1][2] \times b[2][2]$$

$$c[2][0] = a[2][0] \times b[0][0] + a[2][1] \times b[1][0] + a[2][2] \times b[2][0]$$

$$c[2][1] = a[2][0] \times b[0][1] + a[2][1] \times b[1][1] + a[2][2] \times b[2][1]$$

$$c[2][2] = a[2][0] \times b[0][2] + a[2][1] \times b[1][2] + a[2][2] \times b[2][2]$$



函數 YIELD 產生器

- 函數 (function) 中若使用 **return**，函數會直接回傳數值 (value)，也隨之終止函數執行
- 若使用另一個關鍵字 (keyword) **yield**，可使函數產生數值，而不會結束函數執行，這樣的函數被稱為產生器函數 (generator function)

```
>>>  
<generator object gen at 0x02CB0698>  
In func 1  
In func 2  
In func 3  
In func 4  
In func 5  
In func 6  
In func 7  
In func 8  
In func 9  
In func 10  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
>>>
```

Yield.py

```
1 def gen(n):  
2     for i in range(1,n+1):  
3         print('In func',i)  
4         #return i  
5         yield i  
6 print(gen(10))  
7 print(list(gen(10)))
```



全域變數與區域變數

- 函數內的變數包括參數 (parameter) 都稱為區域變數 (local variable)，區域變數的使用範圍 (scope) 限於函數的區塊 (block) 內，一旦離開該區塊範圍便由記憶體中釋放掉，便無法繼續使用原本在區塊內的變數值，下次呼叫該函式時再重新配置記憶體給該函式使用。
- 另外提供一種變數可供多個函式共同使用，變數的有效時間一直到程式結束為止，我們將此類的變數稱為「全域變數」 (Global variable)。

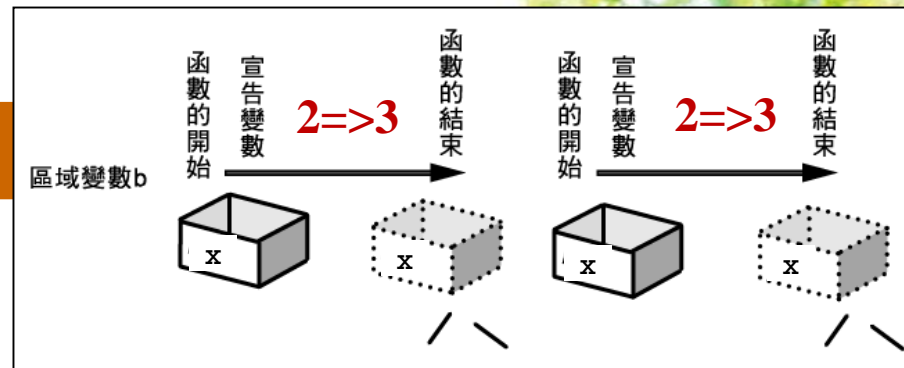


局部/區域變數 (LOCAL VARIABLE)

- 自動變數只在它所定義的**區塊內有效**。只要在變數所屬的區塊結構內執行，該變數的資料是有效而正確的。
- 當程式執行**離開了該區塊**，所有於區塊內定義的自動變數就**不存在了**。

Local.py

```
1 def func():
2     x = 2
3     x = x + 1
4     print('In func():', x)
5
6 x = 1
7 func()
8 func()
9 print(x)
```



```
>>>
In func(): 3
In func(): 3
1
>>>
```



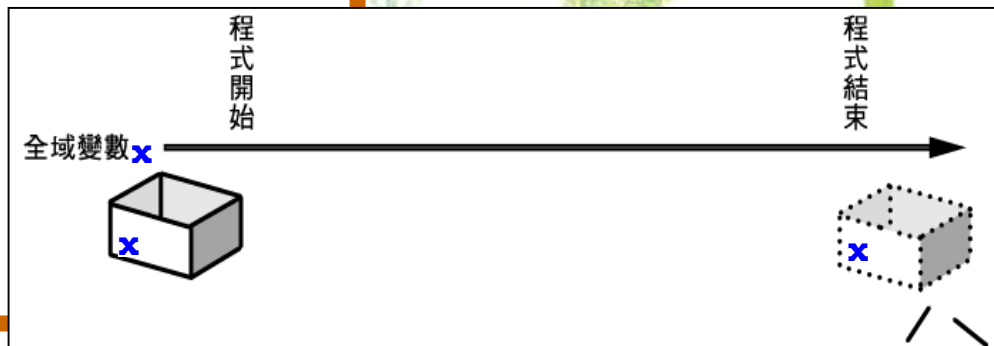
全域變數 (GLOBAL VARIABLE)

- 全域變數的**有效範圍**不是區域性，而是**整體**
- 變數定義在任何函數的外面，可被其他函數所共用
- 僅限取值，不得放在=號左邊，除非宣告成**global**

Global_v1.py

```
1 x = 10
2 def func1():
3     #x = x + 1 #這行不可執行
4     print(' In func1:', x)
5 def func2():
6     print(' In func1:', x)
7
8 print('1 x:', x)
9 func1()
10 func2()
11 print('2 x:', x)
```

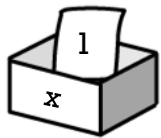
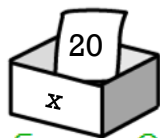
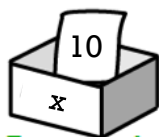
```
>>>
1 x: 10
  In func1: 10
  In func1: 10
2 x: 10
>>>
```



- 程式中區域變數與全域變數的**名稱相同**，當存取函式內的變數時會以**區域變數為優先**，使用時最好注意，建議全域變數**最好不要**和區域變數的**名稱重複**，以免參用時造成混淆。

Local2.py

```
1  #x = 1 #試試x的位置有何不同
2  def func1():
3      x=10
4      print("In func1:", x)
5  def func2():
6      x=20
7      print("In func2:", x)
8
9
10 x = 1
11 func1()
12 func2()
13 print(x)
```



區域變數的名稱**可以重複使用**

在**不同函數**內被宣告的**區域變數**就代表是**3個不同的變數**

```
>>>
In func1: 10
In func2: 20
1
>>>
```

全域變數

- global 修改區域變數

- Ex:

Global_v2.py

```
1 name = "John"
2 def change(n):
3     #global name #測試有無此行會有何不同
4     name = n
5     print ("change name", name)
6 print ("outside function", name)
7 change("Jack")
8 print ("outside function", name)
```

```
>>>
```

```
outside function John
change name Jack
outside function John
```

```
>>>
```

```
outside function John
change name Jack
outside function Jack
```



一般參數傳遞

```
def outer(a):  
    def inner(a):  
        a += 1  
        print("inner a =", a)  
    a += 1  
    inner(a)  
    print("outer a =", a)
```

```
a = 10  
print("global a =", a)  
outer(a)  
print("global a =", a)
```

```
>>>  
global a = 10  
inner a = 12  
outer a = 11  
global a = 10
```



全域 GLOBAL

```
def outer(a): #這時 inner() 裡的 a global 的 a
    def inner():
        global a
        a += 1
        print("inner a =", a)
    a += 1
    inner()
    print("outer a =", a)

a = 10
print("global a =", a)
outer(a)
print("global a =", a)
```

```
global a = 10|
inner a = 11
outer a = 11
global a = 11
>>>
```



NONLOCAL

def outer(a): #inner() 裡的 **a** 是上一層的 **a**，就是 **outer()** 的 **a**

def inner():

nonlocal a

a += 1

print("inner a =", a)

a += 1

inner()

print("outer a =", a)

a = 10

print("global a =", a)

outer(a)

print("global a =", a)

>>>

global a = 10

inner a = 12

outer a = 12

global a = 10

>>>

