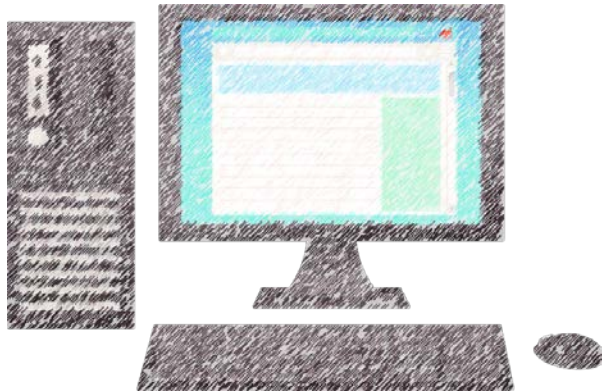


桃園市政府勞動局 112 年勞工學苑產業應用班
Python 程式設計：從零基礎入門到進階

第 8 單元

函式



林柏江老師

元智大學 電機工程學系 助理教授

pclin@saturn.yzu.edu.tw

預計課程進度

週次	日期	上午課程內容 (09:00 ~ 12:00)	下午課程內容 (13:00 ~ 16:00)
1	2023/07/23	01. 運算思維簡介	02. Python 快速上手
2	2023/07/30	03. Python 基礎	04. Python 基本資料結構
3	2023/08/06	05. 字串	06. 字典
4	2023/08/13	07. 流程控制	08. 函式
5	2023/08/20	09. 模組與作用域	10. Python 程式檔
6	2023/08/27	11. 檔案系統的使用與檔案讀寫	12. 例外處理
7	2023/09/03	13. 類別與物件導向程式設計	14. 初探資料結構與演算法
8	2023/09/10	15. 陣列	16. 鏈結串列
9	2023/09/17	17. 堆疊與佇列	18. 圖形結構
	2023/09/24, 2023/10/01, 2023/10/08 (共三周) 放假		
10	2023/10/15	19. 樹狀結構	20. 分治法
11	2023/10/22	21. 動態規劃	22. 貪婪演算法
12	2023/10/29	23. 回溯	24. 分支定界法

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

什麼是函式

- 函式 (function) 是一段定義好的程式碼，可以重複呼叫使用。
- Python 函式定義的基本語法：

```
def 函式名稱(參數1, 參數2, ...):  
    程式區塊
```

- 參數是選用的。
- 若不需要參數，小括號 () 裡面空著即可。
- 程式區塊的敘述必須縮排，且處於相同的縮排級別。

函式的範例

```
>>> def fact(n):  
...     """回傳參數 n 的階乘"""  
...     r = 1  
...     while n > 0:  
...         r = r * n  
...         n = n - 1  
...     return r  
...
```

文件字串，說明
此函式的用途。

回傳變數 r 的值。

函式的被呼叫端 (Callee) 與呼叫端 (Caller)

```
>>> def fact(n):  
...     """回傳參數 n 的階乘"""  
...     r = 1  
...     while n > 0:  
...         r = r * n  
...         n = n - 1  
...     return r  
...
```

函式定義 (被呼叫端)

```
>>> fact(4)  
24
```

呼叫者 (呼叫端)

函式的回傳值

- 函式可使用 `return` 敘述，把回傳值 (return value) 傳給呼叫者。
 - 即使函式有回傳值，程式設計可以選擇要 / 不要處理這個回傳值。
- 函式執行了 `return` 敘述後，無論此函式的程式區塊還有沒有其他敘述尚未執行，一律結束此函式，把控制權交給呼叫者。
- 如果函式的定義中沒有 `return` 敘述，則預設會回傳 `None`。

```
>>> fact(4)
```

```
24
```

```
>>> x = fact(4)
```

```
>>> x
```

```
24
```

← 可以只執行函式，而不處理回傳值。

← 也可以使用變數來接收函式的回傳值，以便做後續處理。

文件字串 (Documentation String，簡稱 Docstring)

- 函式定義裡面的第一行敘述，可使用一對的連續三個雙引號 `""" """` 來定義一個字串，用來說明這個函式的用途。
- 函式若有寫 docstring，便可透過 `help(函式名稱)` 或是 `函式名稱.__doc__` 來查看 docstring 的說明。有些工具可使用 docstring 自動產生說明文件。
- 有關於 docstring 的撰寫慣例，可參考 [PEP 257](#)。

```
>>> help(fact)
Help on function fact in module __main__:

fact(n)
    回傳參數 n 的階乘


>>> fact.__doc__
'回傳參數 n 的階乘'
```

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

參數 (Parameter) 與引數 (Argument)

參數 (定義函式時的形式變數，也稱作
形式參數，**formal parameter**)



```
>>> def fact(n):  
...     """回傳參數 n 的階乘"""  
...     r = 1  
...     while n > 0:  
...         r = r * n  
...         n = n - 1  
...     return r  
...
```

函式定義 (被呼叫端)

引數 (要指派給函式參數的值，也稱作
真正參數，**actual parameter**)



```
>>> fact(4)  
24
```

呼叫者 (呼叫端)

使用參數的位置來傳遞引數

- 在 Python 中，函式如果定義了多個參數，則呼叫該函式時，會依照順序，把參數值 (引數) 指派給各個參數。
- 這種依照位置傳遞的引數稱為位置引數 (positional argument)，其對應的參數稱為位置參數 (positional parameter)。

```
>>> def power(x, y):  
...     r = 1  
...     while y > 0:  
...         r = r * x  
...         y = y - 1  
...     return r  
...  
>>> power(3, 4)  
81
```

引數 3 傳給參數 x，
引數 4 傳給參數 y。

引數的數量與參數的數量必須一致

```
>>> def power(x, y):  
...     r = 1  
...     while y > 0:  
...         r = r * x  
...         y = y - 1  
...     return r  
...  
>>> power(3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: power() missing 1 required  
positional argument: 'y'
```

引數的數量與參數的數量不一致，出現參數值不足的 `TypeError` 例外。

參數的預設值 (預設引數)

- 函式定義中，可以為參數設定預設值 (default value)。
- 函式被呼叫時，若省略具有參數預設值的引數，該引數就會使用預設值，而不會出現參數值不足的 `TypeError` 例外。
- 定義參數預設值的語法：

```
def 函式名稱(參數1, 參數2=預設值2, 參數3=預設值3, ...):  
    程式區塊
```

- 注意：沒有預設值的參數一定要放在前面。

參數預設值的範例

```
>>> def power(x, y=2):  
...     r = 1  
...     while y > 0:  
...         r = r * x  
...         y = y - 1  
...     return r  
...  
>>> power(3, 4)  
81  
>>> power(3)  
9
```

沒有預設值的參數一定要放在前面

```
>>> def func(x=2, y):  
      File "<stdin>", line 1  
        def func(x=2, y):  
                ^  
SyntaxError: non-default argument follows  
default argument
```

沒有預設值的參數不能定義在後面，
會產生語法錯誤。

使用參數的名稱來傳遞引數

- 在 Python 中，呼叫函式時，可以直接指定參數名稱來把參數值 (引數) 指派給各個參數。
- 這種引數傳遞方式稱為關鍵字傳遞 (keyword passing) 或是指名傳遞 (named passing)。
- 這種有指定參數名稱的引數稱為關鍵字引數 (keyword argument)。

```
>>> def power(x, y=2):  
...     r = 1  
...     while y > 0:  
...         r = r * x  
...         y = y - 1  
...     return r  
...
```

```
>>> power(2, 3)  
8  
>>> power(3, 2)  
9  
>>> power(y=2, x=3)  
9
```

位置引數與關鍵字引數的呼叫順序

- 呼叫函式時，若混合使用位置引數以及關鍵字引數，其順序必須是位置引數在前，關鍵字引數在後。
- 若順序不對，會引發錯誤。

```
>>> power(y=3, 2)
File "<stdin>", line 1
    power(y=3, 2)
                ^
SyntaxError: positional argument follows
keyword argument
```

不可以指派多個引數給同一個變數

```
>>> power(2, x=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() got multiple values for
argument 'x'
```

位置引數與關鍵字引數同時傳遞給
同一個參數，會產生語法錯誤。

關鍵字引數搭配參數預設值

```
def list_file_info(size=False, create_date=False, mod_date=False, ...):  
    # 取得目前目錄下的檔案名稱...  
    if size:  
        # 取得檔案大小  
    if create_date:  
        # 取得檔案建立日期  
    if mod_date:  
        # 取得檔案修改日期  
  
    return fileinfostructure  
  
fileinfo = list_file_info(True, mod_date=True)
```

其他沒指定的參數
就用預設值。

位置引數 True
傳給參數 size。

關鍵字引數 True 傳給
參數 mod_date。

使用一個參數接收多出來的位置引數

函式定義時，若參數前面加上一個符號 *，
這個參數就可以用來接收多出來的引數。

```
def 某函式(x, y, *z):
```

函式主體

可用來接收 x 、 y 位置
之後的所有引數。

```
某函式(1, 2, 3, 4, 5)
```

傳給參數 x 。

傳給參數 y 。

打包為 tuple (3, 4, 5)
傳給參數 z 。

通常照慣例會使用 `*args` 來做為這種參數的名稱，不過
也可以使用其他名稱。

範例

傳入任意數量的數字，回傳其中的最大值。

```
>>> def maximum(*numbers):  
...     if len(numbers) == 0:  
...         return None  
...     else:  
...         maxnum = numbers[0]  
...         for n in numbers[1:]:  
...             if n > maxnum:  
...                 maxnum = n  
...         return maxnum  
... 
```

```
>>> maximum(3, 2, 8)  
8  
>>> maximum(1, 5, 9, -2, 2)  
9  
>>> maximum()
```

使用一個參數接收多出來的關鍵字引數

函式定義時，若參數前面加上符號 `**`，這個參數就可以用來接收多出來的關鍵字引數。

```
def 某函式(x, y, **z):  
    函式主體
```

可用來接收 `x`、`y` 位置之後的所有引數。

```
某函式(1, 2, a=3, b=4, c=5)
```

傳給參數 `x`。

傳給參數 `y`。

打包為字典 `{'a':3, 'b':4, 'c':5}`
傳給參數 `z`。

通常照慣例會使用 `**kwargs` 來做為這種參數的名稱，不過也可以使用其他名稱。

範例

```
>>> def example_fun(x, y, **other):  
...     print("x: {0}, y: {1}, 字典 'other' 內的鍵: {2}".format(x,  
...         y, list(other.keys())))  
...     other_total = 0  
...     for k in other.keys():  
...         other_total = other_total + other[k]  
...     print("字典 'other' 內值的總和為 {0}".format(other_total))
```

```
>>> example_fun(2, y="1", foo=3, bar=4)  
x: 2, y: 1, 字典 'other' 內的鍵: ['foo', 'bar']  
字典 'other' 內值的總和為 7
```


混用不同類型的參數定義與引數傳遞

- 定義 Python 函式時，各種參數的定義與傳遞方式可以同時使用。
- 但是定義時須注意各種參數的先後順序：

位置參數，預設值參數，`*args`，`**kwargs`

- 呼叫函式時，也要注意引數傳遞的順序：位置引數在前，關鍵字引數在後。
- 位置參數一定要有分配到引數，否則會導致錯誤。

範例：

混用不同類型的參數定義與引數傳遞

```
>>> def func(p1, p2, p3='three', *p4, **p5):  
...     print(p1, p2, p3, p4, p5)  
...
```

```
>>> func(1, 2, 3, 4, 5, x=1, y=2)  
1 2 3 (4, 5) {'x': 1, 'y': 2}
```

```
>>> func(1, 2, 3, 4, 5)  
1 2 3 (4, 5) {}
```

```
>>> func(1, 2, 3, 4, x=5)  
1 2 3 (4,) {'x': 5}
```

```
>>> func(1, 2, 3)  
1 2 3 () {}
```

```
>>> func(1, 2)  
1 2 three () {}
```

```
>>> func(1)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: func() missing 1 required  
positional argument: 'p2'
```

```
>>> func(1, 2, 3, p3=3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: func() got multiple values  
for argument 'p3'
```

```
>>> func(1, 2, p4=4)  
1 2 three () {'p4': 4}
```

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

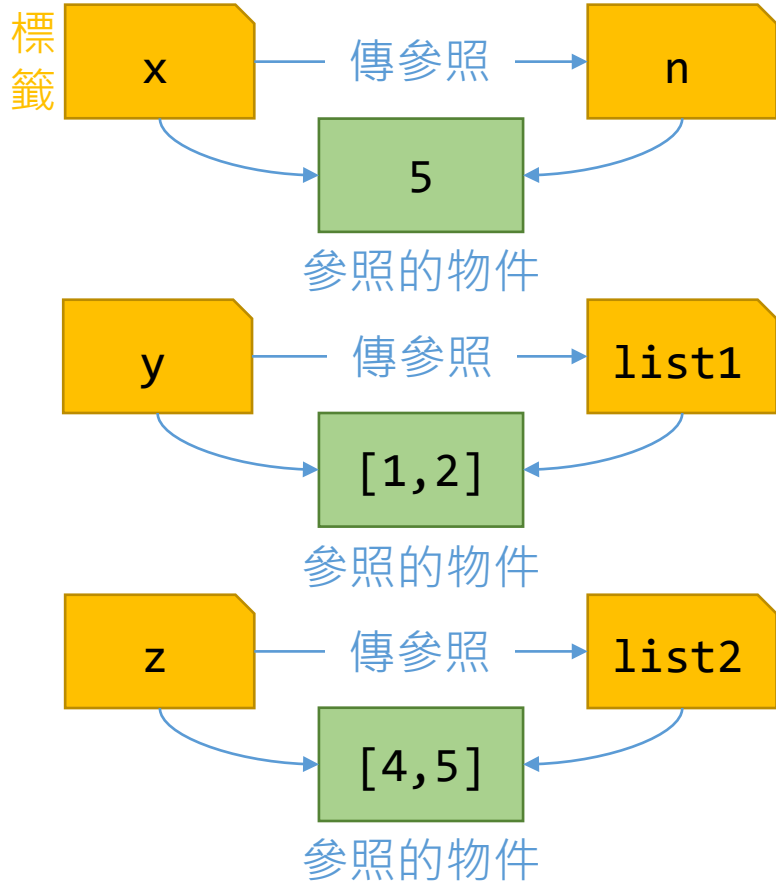
Python 函式的引數傳遞

- 許多程式語言 (例如 C 語言) 的函式呼叫預設是以傳值的方式來傳遞引數。
- Python 一律以物件參照的方式來傳遞引數。其引數傳入的是物件的參照。
- 參數就會參照到引數所參照的物件。
- 參數與引數所參照的是同一個物件。
- 如果傳入的引數是參照到可變物件 (例如 list、字典)，從函式內部修改物件內容，就會直接修改到原本的物件，函式外部也看得到該物件的改變。

```
>>> def f(n, list1, list2):  
...     list1.append(3)  
...     list2 = [4, 5, 6]  
...     n = n + 1  
...  
>>> x = 5  
>>> y = [1, 2]  
>>> z = [4, 5]  
>>> f(x, y, z)  
>>> x, y, z  
(5, [1, 2, 3], [4, 5])
```

Python 函式的引數傳遞 (續)

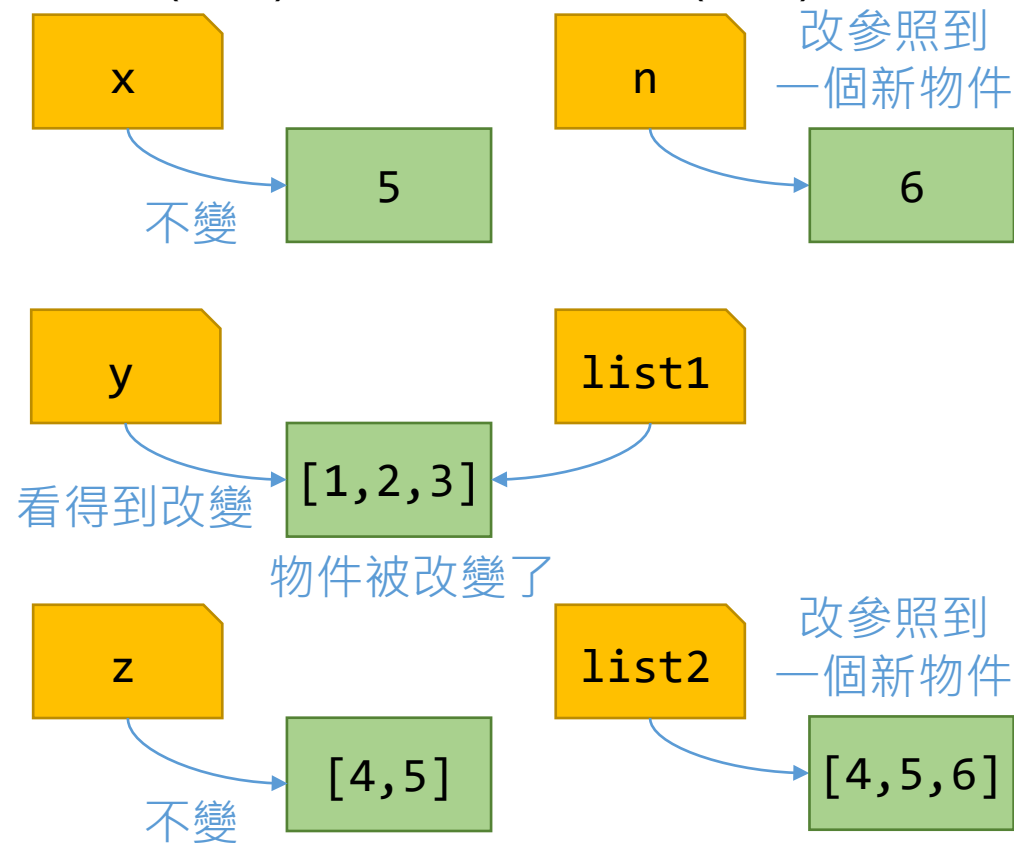
函式外部 (引數) 函式內部 (參數)



函式 `f()` 剛開始執行時的情形

```
>>> def f(n, list1, list2):  
...     list1.append(3)  
...     list2 = [4, 5, 6]  
...     n = n + 1  
...  
>>> x = 5  
>>> y = [1, 2]  
>>> z = [4, 5]  
>>> f(x, y, z)  
>>> x, y, z  
(5, [1, 2, 3], [4, 5])
```

函式外部 (引數) 函式內部 (參數)



函式 `f()` 執行到最後時的情形

傳入可變物件如何不影響函式外部

若想要把可變物件傳入函式，又不希望影響到函式外部的話，可以在函式內部建立一份可變物件的副本。

```
>>> def f(lst):  
...     lst = lst[:]  
...     lst.append(3)  
...  
>>> x = [1, 2]  
>>> f(x)  
>>> x  
[1, 2]
```

若可變物件是多層的 list 或 tuple 等物件，可用先前介紹過的 `deepcopy()`。

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. **local、nonlocal、global 變數**
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

區域變數 (Local Variable)

函式的參數以及函式裡建立的任何變數，都是函式內的區域變數。

```
>>> def fact(n):  
...     """回傳參數 n 的階乘"""  
...     r = 1  
...     while n > 0:  
...         r = r * n  
...         n = n - 1  
...     return r  
...
```

函式內部的變數 `r` 以及參數 `n` 屬於 `fact()` 函式的區域變數 (local variable)。不論函式內部對它們進行任何變動，對於函式外部的任何其他變數都沒有影響。

使用 `global` 宣告全域變數

- 若函式內部需要存取外部的變數，可以在函式內先透過 `global` 敘述明確地宣告該變數為全域變數。
- 之後，函式就可以存取函式外部的變數。

```
>>> def func():  
...     global a  
...     a = 1  
...     b = 2  
...
```

```
>>> a = "one"  
>>> b = "two"  
>>> func()  
>>> a  
1  
>>> b  
'two'
```

使用 `nonlocal` 宣告為上一層變數

- `nonlocal` 與 `global` 有點類似，都是用來宣告變數。
- 差別在於：
 - `global` 宣告的是最上層的變數。
 - `nonlocal` 宣告的是上一層的變數。
- 函式定義中可以包含另一個函式定義，形成多層的函式定義。
- 每一層函式定義會有它自己的區域變數，因此會有多層的區域變數。

使用 nonlocal 宣告為上一層變數 (續)

```
g_var = 0
nl_var = 0
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
def test():
    nl_var = 2
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
    def inner_test():
        global g_var
        nonlocal nl_var

        g_var = 1
        nl_var = 4

        print("in inner_test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
    inner_test()
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))

test()
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
```

```
top level-> g_var: 0 nl_var: 0
in test-> g_var: 0 nl_var: 2
in inner_test-> g_var: 1 nl_var: 4
in test-> g_var: 1 nl_var: 4
top level-> g_var: 1 nl_var: 0
```

對於函式外部的變數做讀取與修改的差別

- 如果要修改函式外部的變數，函式中必須明確地使用 `global` 或是 `nonlocal` 敘述。
- 如果只是要讀取函式外部的變數值，則不需要使用 `global` 或是 `nonlocal` 敘述。
 - 當 Python 在函式內找不到某個變數名稱，它便會嘗試逐層往上查找，所以不需要使用 `global` 或是 `nonlocal` 敘述，也可以讀取到函式外部的變數值。

變數的生命週期

- 區域變數會在函式結束時刪除。
- `nonlocal` 變數隨著上層函式結束而刪除。
- 全域變數會持續存在，直到整個程式執行完畢。
- 變數的生命週期跟命名空間 (namespace) 有關，下一個單元會來介紹。

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

可以把函式指派到變數

```
>>> def f_to_kelvin(degrees_f):  
...     return 273.15 + (degrees_f - 32) * 5 / 9  
...  
>>> def c_to_kelvin(degrees_c):  
...     return 273.15 + degrees_c  
...  
>>> abs_temperature = f_to_kelvin  
>>> abs_temperature(32)  
273.15  
>>> abs_temperature = c_to_kelvin  
>>> abs_temperature(0)  
273.15
```

也可以把函式放在 list、tuple 或字典

```
>>> def f_to_kelvin(degrees_f):  
...     return 273.15 + (degrees_f - 32) * 5 / 9  
...  
>>> def c_to_kelvin(degrees_c):  
...     return 273.15 + degrees_c  
...  
>>> t = {'FtoK': f_to_kelvin, 'CtoK': c_to_kelvin}  
>>> t['FtoK'](32)  
273.15  
>>> t['CtoK'](0)  
273.15
```


課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
- 6. lambda 匿名函式**
7. 產生器函式
8. 修飾器

lambda 匿名函式

- 對於比較簡短的函式，可以使用 lambda 來定義。
- lambda 可視為只能定義一行運算式的小函式。
- 因為 lambda 不需要命名，所以也稱為匿名函式 (anonymous function)。
- 語法如下：

```
lambda 參數1, 參數2, ...: 運算式
```

- lambda 不需要 return 敘述，運算式執行後的值會自動被回傳。

範例：lambda 匿名函式

```
>>> t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,  
...      'CtoK': lambda deg_c: 273.15 + deg_c}  
>>> t2['FtoK'](32)  
273.15
```

課程大綱

1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

產生器函式

- 產生器函式 (generator function) 是一種特殊的函式，用來自訂一個可走訪的物件。
- 定義產生器函式時，必須使用 `yield` 關鍵字來回傳每次走訪的值。
- 遇到以下情況時，產生器函式會停止：
 - 不再有 `yield` 回傳值
 - 遇到空的 `return` 敘述
 - 遇到函式的結尾
- 產生器函式的區域變數值會持續保存到下一次呼叫。
 - 普通函式的區域變數值會在跳出函式後消失。

範例：產生器函式

使用產生器函式來產生 0 ~ 3 的數字。

```
>>> def four():  
...     x = 0  
...     while x < 4:  
...         print("in generator, x =", x)  
...         yield x  
...         x += 1  
... 
```

```
>>> for i in four():  
...     print(i)  
...  
in generator, x = 0  
0  
in generator, x = 1  
1  
in generator, x = 2  
2  
in generator, x = 3  
3
```

範例：產生器函式 (續)

把產生器函式跟 `in` 搭配使用。

```
>>> def four():  
...     x = 0  
...     while x < 4:  
...         print("in generator, x =", x)  
...         yield x  
...         x += 1  
... 
```

```
>>> 2 in four()  
in generator, x = 0  
in generator, x = 1  
in generator, x = 2  
True  
>>> 5 in four()  
in generator, x = 0  
in generator, x = 1  
in generator, x = 2  
in generator, x = 3  
False
```

yield 與 yield from

- Python 3.3 開始，新增一個產生器函式關鍵字：`yield from`。
- `yield from` 可用來把多個產生器串聯在一起。
- `yield from` 的行為與 `yield` 相同，只是把產生器機制委託給另一個產生器。

範例：yield from

```
>>> def subgen(x):  
...     for i in range(x):  
...         yield i  
...  
>>> def gen(y):  
...     yield from subgen(y)  
...
```

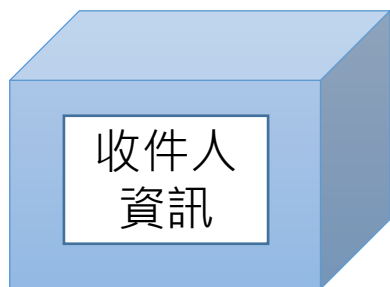
```
>>> for q in gen(6):  
...     print(q)  
...  
0  
1  
2  
3  
4  
5
```

課程大綱

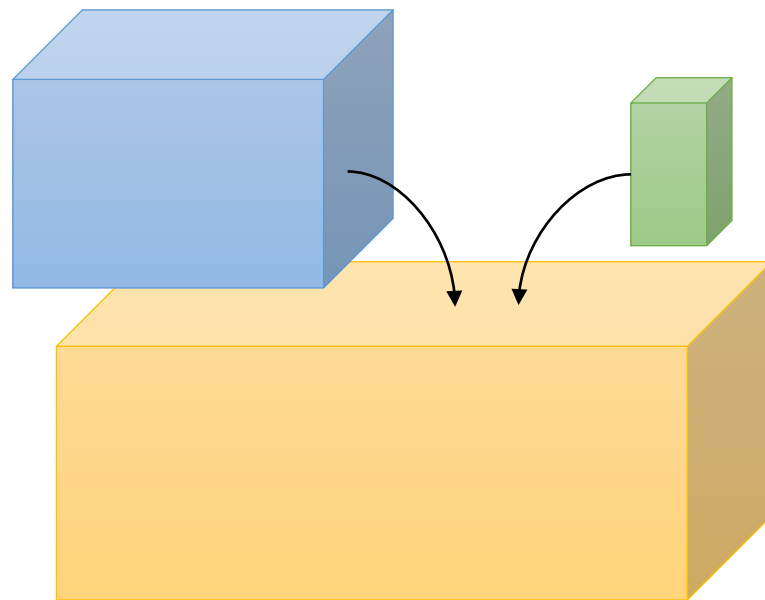
1. 基本函式定義
2. 函式的參數選項
3. 使用可變物件做為引數
4. local、nonlocal、global 變數
5. 把函式指派到變數
6. lambda 匿名函式
7. 產生器函式
8. 修飾器

修飾器 (Decorator)

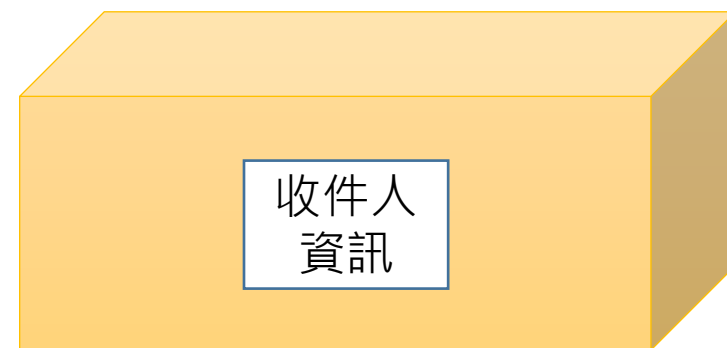
- 假設我們手上有一個函式，在不變更這個函式原始碼的前提下，我們想為這個函式增加額外的動作。Python 的修飾器可以達成這個需求。
- 類似的生活例子：我們有一箱物品要郵寄給家人，原本已經封箱、也已貼好收件人資訊標籤。突然想額外多加寄另一件物品。在不拆開原本箱子的前提下，可以這樣做：



原本的標籤先拿下來



原本箱子跟額外物品裝入新箱子



原本的標籤貼到新箱子₅₁

Python 的函式也是物件

- 在 Python 中，函式是第一級物件 (first-class object)，與數字、字串等相同，可進行任何物件操作，例如放在 list 中、傳遞為函式參數、當作函式回傳值等。
- 我們可以把函式指派給變數。
- 也可以把函式當作引數，傳遞給其他函式。
- 也可以把函式當作其他函式的回傳值。

範例：修飾器

```
>>> def decorate(func):
...     def wrapper_func(*args):
...         print("原函式執行前")
...         func(*args)
...         print("原函式已執行")
...     return wrapper_func
...
>>> def myfunction(parameter):
...     print(parameter)
...
>>> myfunction = decorate(myfunction)
>>> myfunction("hello")
原函式執行前
hello
原函式已執行
```

可以接收任何數量引數的包裝函式

原本的函式被包裝在這裡，前後可做額外的動作，例如 `print()`

回傳這個包裝函式

原本的函式定義與呼叫

把 `decorate` 函式回傳的物件指派給 `myfunction` 變數

範例：修飾器 (續)

```
myfunction = decorate(myfunction)
```

❶ 執行 `decorate` 函式，把 `myfunction` 函式包裝到 `wrapper_func` 函式內。

```
def decorate(func):  
    def wrapper_func(*args):  
        print("原函式執行前")  
        func(*args)  
        print("原函式已執行")  
    return wrapper_func
```

```
def myfunction(parameter):  
    print(parameter)
```

❷ 回傳 `wrapper_func` 函式物件。

```
def wrapper_func(*args):  
    print("原函式執行前")  
    def myfunction(parameter):  
        print(parameter)  
    print("原函式已執行")
```

❸ 把原本函式名稱 `myfunction` 改參照到 `wrapper_func` 函式。

← `myfunction`

@decorate 語法糖

- Python 提供了一個修飾器的語法糖 (syntactic sugar)，可用單行敘述 `@decorate` 來完成上述 `myfunction = decorate(myfunction)` 的動作。
 - 語法糖：某種語法並不是用來增加新功能，而是讓原有功能更容易使用，讓程式碼更簡潔、具有更好的可讀性，這種語法就會被稱為語法糖。

範例：@decorate 語法糖

在函式定義的前一行加入
這個敘述，取代原本後面
要寫的 `myfunction =
decorate(myfunction)`

```
>>> def decorate(func):
...     def wrapper_func(*args):
...         print("原函式執行前")
...         func(*args)
...         print("原函式已執行")
...     return wrapper_func
...
>>> def myfunction(parameter):
...     print(parameter)
...
>>> myfunction = decorate(myfunction)
>>> myfunction("hello")
原函式執行前
hello
原函式已執行
```

原本的寫法

```
>>> def decorate(func):
...     def wrapper_func(*args):
...         print("原函式執行前")
...         func(*args)
...         print("原函式已執行")
...     return wrapper_func
...
>>> @decorate
... def myfunction(parameter):
...     print(parameter)
...
>>> myfunction("hello")
原函式執行前
hello
原函式已執行
```



使用 @decorate 語法糖的寫法

重點整理

- 函式的定義方法：

```
def 函式名稱(參數1, 參數2, ...):  
    程式區塊
```

- 參數的種類與定義的順序：位置參數、預設值參數、`*args`、`**kwargs`。
- 呼叫函式時引數傳遞的順序：位置引數、關鍵字引數。
- 用可變物件做為引數時要小心。
- `global` 綁定到最上層的變數，`nonlocal` 綁定到上一層的變數。
- `lambda` 函式：不用命名的小函式。
- 產生器函式：走訪物件的產生器，用 `yield` 回傳走訪值。
- 修飾器：把函式包裝到新函式中。