

# Python

授課講師 蔡林甫

教材編寫 蔡林甫



緯育 *TibaMe*

提拔我的數位競爭力

<https://www.tibame.com/>

## 授課講師介紹



授課講師

蔡林甫

### 簡歷

資料科學家

- 程式效率優化
- 影像辨識導入 AOI 機台
- 工廠生產良率大數據化分析
- 大數據分析顧問

### 專長

負責專案使用：

Python、Linux、MongoDB、SQL、OpenCV、  
統計學、Machine Learning、Kafka... 等等

### 老師的話

化繁為簡，由淺入深

### 聯絡方式

tsailinfu420@gmail.com

# 課程大綱

- ◆ Module 1 : Python 程式的安裝
- ◆ Module 2 : Python 程式第一步
- ◆ Module 3 : Python 的基本知識
- ◆ Module 4 : 變數
- ◆ Module 5 : 運算式的基礎知識
- ◆ Module 6 : 輸入 – 由鍵盤輸入
- ◆ Module 7 : 條件式與流程控制
- ◆ Module 8 : 迴圈(Loop)
- ◆ Module 9 : 群集-串列(List)
- ◆ Module 10 : 群集-元組(Tuple)
- ◆ Module 11 : 群集-字典(Dictionary)
- ◆ Module 12 : 群集-集合(Set)

# 課程大綱

- ◆ Module 13 : 函數
- ◆ Module 14 : 類別
- ◆ Module 15 : 字串的操作
- ◆ Module 16 : 檔案
- ◆ Module 17 : 例外處理
- ◆ Module 18 : 系統處理
- ◆ Module 19 : 日期與時間
- ◆ Module 20 : 正規表示式
- ◆ Module 21 :  
Multiprocessing 與 Multithreading

## 學習本課程須知

### 先備知識

英文略懂、數學略懂、程式先後順序的邏輯。

程式中所使用的英文和數學不會太難，

特別是我們要優先以了解 Python 為主。

另外，邏輯就比較難說了，如果遇到比較複雜的狀況可以拿張紙來順一下邏輯的部分。

先備知識不用特別在意，就看之後遇到問題然後解決它。

### 學習目標

看比較遠的目標是大數據、AI的目標，

在那之前我們要先了解協助我們處理的工具，也就是 Python。

把學習好 Python 當成階段性目標，也就是先了解課程的內容。

## 學習本課程須知

### 學習方式

跟上進度，遇到問題即時反應。

每打完一個範例都要想一想：這個範例學到什麼。

課程中每個範例打完一遍後刪掉或放在別處，接著打第二遍打完後刪掉，  
然後打第三遍打完後刪掉。總之每個範例只少打三遍。

### 須完成 那些作業 或考試

會依照學習進度可能有相對應的作業，寫作業前請先將課程的範例練習三遍。

當然如果真的想不出來可以拿當下作業遇到的阻礙進行提問。

# Module 1 :

## 程式的安裝

1-1 程式安裝-前言

1-2 安裝 Python

1-3 安裝 Jupyter Notebook

1-4 安裝 PyCharm

- 我們要運用 Python 撰寫程式，需要先安裝 Python。
- 安裝完 Python 後為了方便撰寫程式，所以我們可以選擇用 PyCharm 或是 Jupyter notebook。
- 如果需要繪圖或是使用比較好看的表格的時候使用 Jupyter notebook 比較好，如果只是文字、數字...那麼 PyCharm 就好了。

- 雖然兩種撰寫平台未來都會接觸到，但如果是初學習程式語言時使用 PyCharm 會比較好，

原因是：

1. 程式有固定的格式，起初有軟體提醒你要注意那些格式。
2. 比較能掌握程式，因為 jupyter 會在你沒有要求的情況下自動幫你印出東西。
3. 使用 PyCharm 期間比較不會吃太多的記憶體，電腦比較不易卡卡的。

- 另外，學習程式的過程中，每一個練習就是基本功，希望上課練著打一遍，放學後再打兩到三遍。  
上課時統一用 PyCharm，  
放學後可以用 PyCharm 也可以用 Jupyter notebook 練習。  
如果放學後在使用 Jupyter notebook 有疑問，也可以提問。

- <https://www.python.org/downloads/>



<https://www.python.org> > [downloads](#) ▾ [翻譯這個網頁](#)

## Download Python

Download the latest version of Python. Download Python 3.9.7. Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other.

[Python Releases for Windows](#) · [Python Releases for macOS](#) · [Python 3.9.5 · 3.9.7](#)

版本可能會顯示更新的，但最新的不一定最好，用相對穩定的版本。

**Download the latest version for Windows**

[Download Python 3.10.0](#)

Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [macOS](#), [Other](#)

Want to help test development versions of Python? [Prereleases](#), [Docker images](#)

Looking for Python 2.7? See below for specific releases

- Python 3.9.7 - Aug. 30, 2021



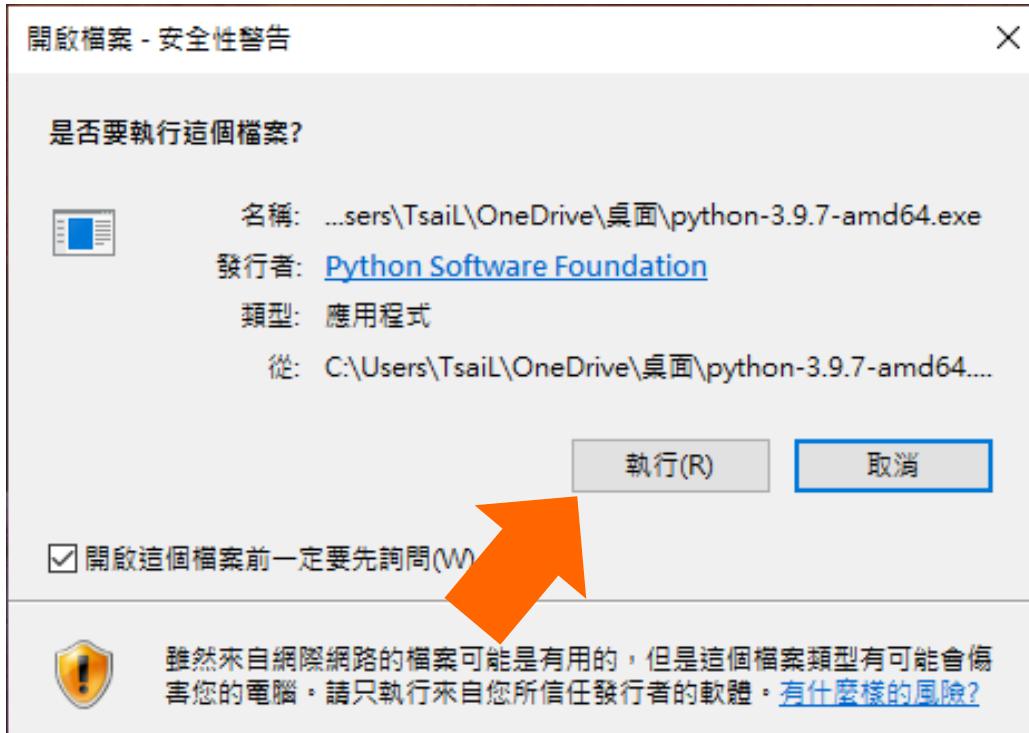
**Note that Python 3.9.7 cannot be used on Windows 7 or earlier.**

- Download Windows embeddable package (32-bit)

<u>Windows embeddable package (64-bit)</u>	Windows	
<u>Windows help file</u>	Windows	
<u>Windows installer (32-bit)</u>	Windows	Recommended
<u>Windows installer (64-bit)</u>	Windows	

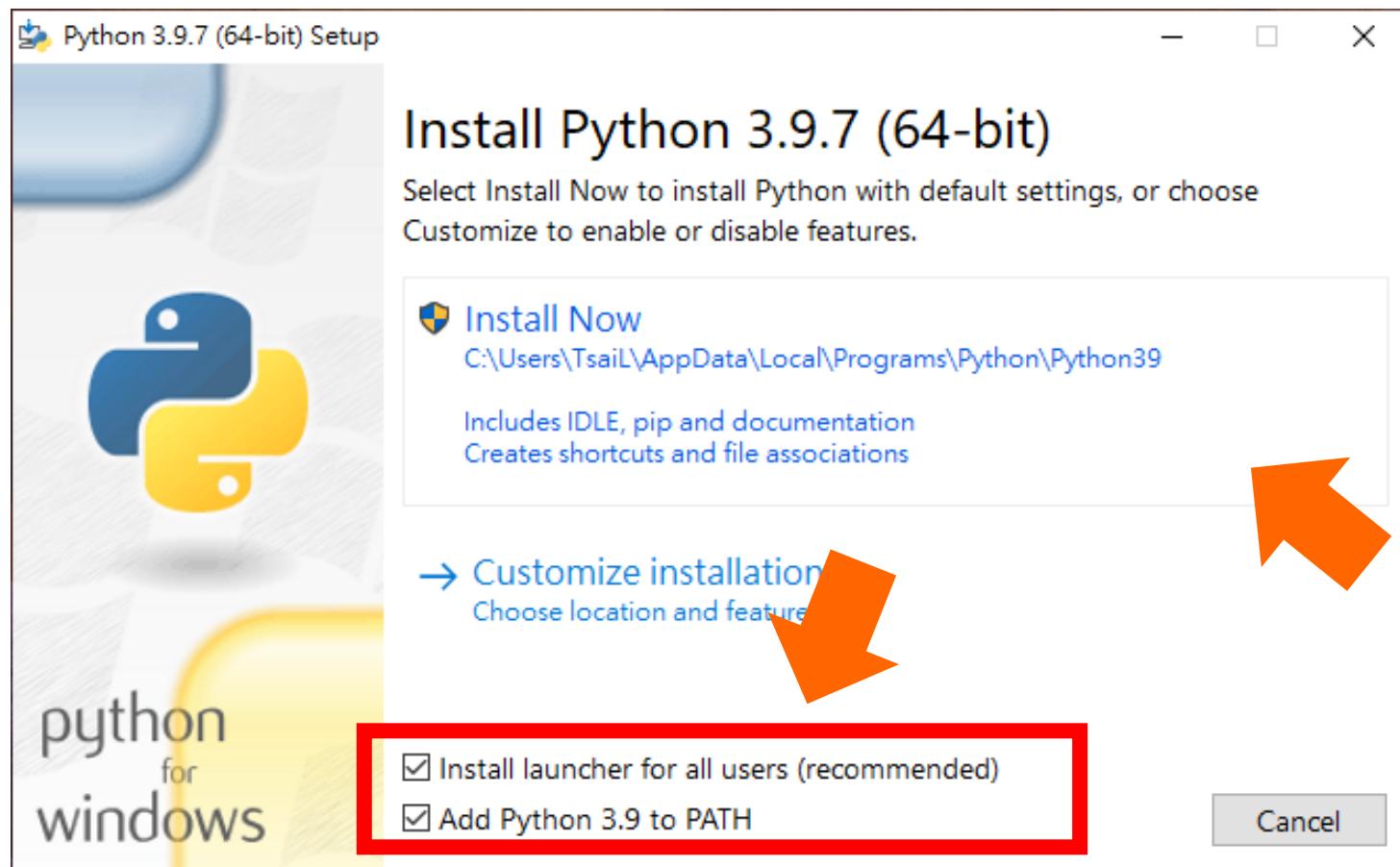


# 1-2 安裝 python



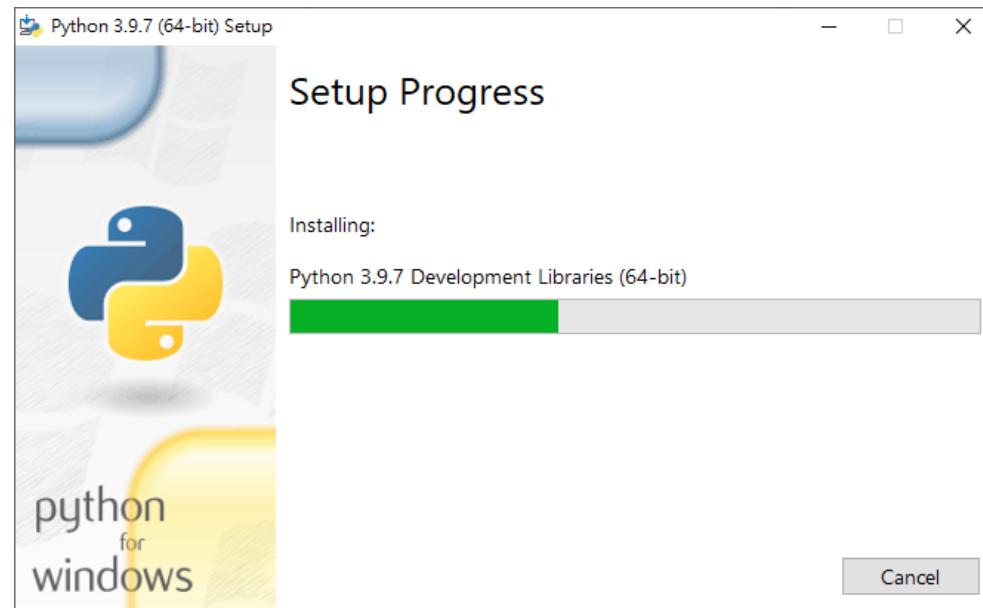
# 1-2 安裝 python

- 先按 Add Python 3.9 to PATH
- 再按 Install Now



# 1-2 安裝 python

- 等它安裝完



## 1-2 安裝 python

開啟命令提示字元：

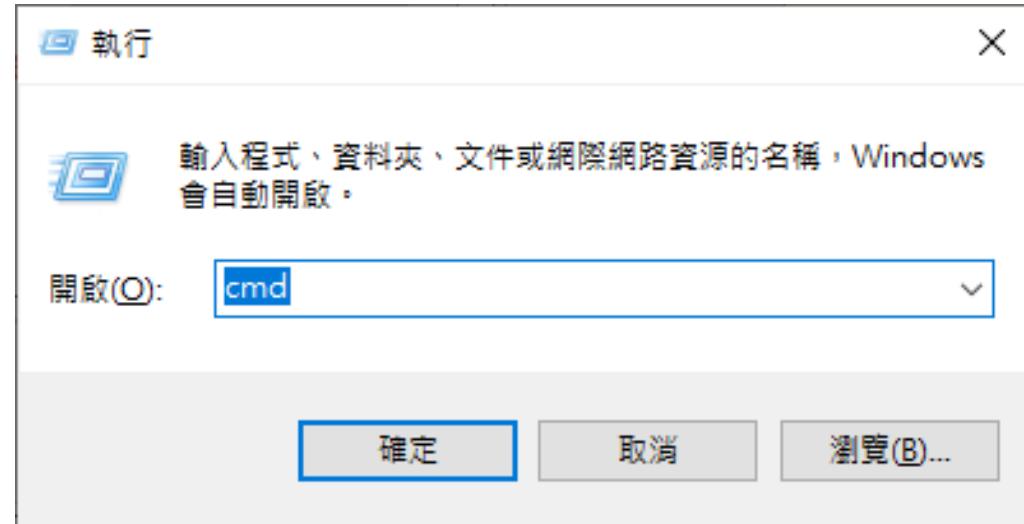
快捷鍵：win + R

後輸入 cmd

輸入：

python -V

大寫的 V 這裡表示  
version



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.19042.1165]
(c) Microsoft Corporation. 權利所有，並保留一切權利。
C:\Users\TsaiL>python -V
Python 3.9.7
```

輸入：

python -m pip -V 或 pip -V

確認是否也有的 pip 和確認版本。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.19042.1165]
(c) Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Users\TsaiL>python -V
Python 3.9.7

C:\Users\TsaiL>python -m pip --version
pip 21.2.3 from C:\Users\TsaiL\AppData\Local\Programs\Python\Python39\lib\site-packages\pip (python 3.9)

C:\Users\TsaiL>
```

備註：pip 是 Python 軟體包管理系統，可協助你安裝其他套件。

~到目前為止確認是否安裝成功，如果成功表示已安裝完 python 了，才可以接下去。~

- 如過之後電腦的 Python 需要安裝套件可以在 cmd 這裡安裝  
`pip install package_name`  
`pip install ipython`

 C:\WINDOWS\system32\cmd.exe

```
C:\Users\User>pip install ipython
```

`pip install jupyter`

 C:\WINDOWS\system32\cmd.exe

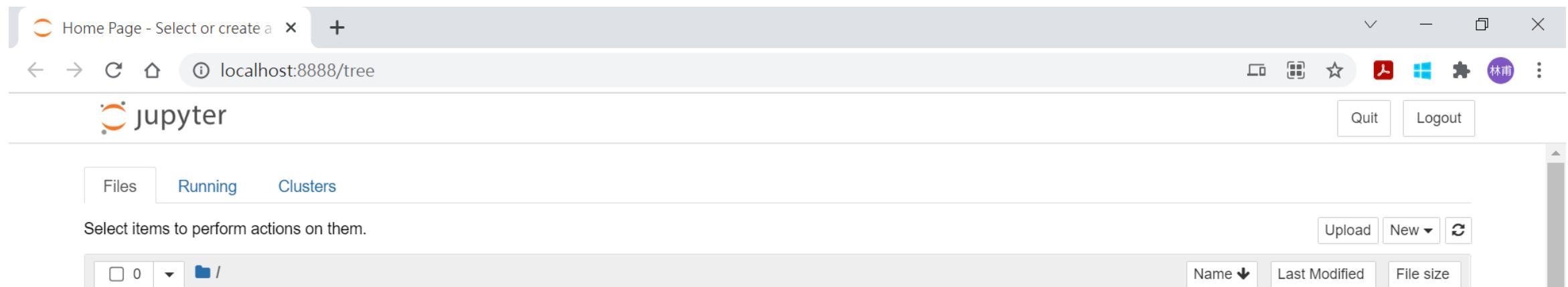
```
C:\Users\User>pip install jupyter
```

輸入 jupyter notebook

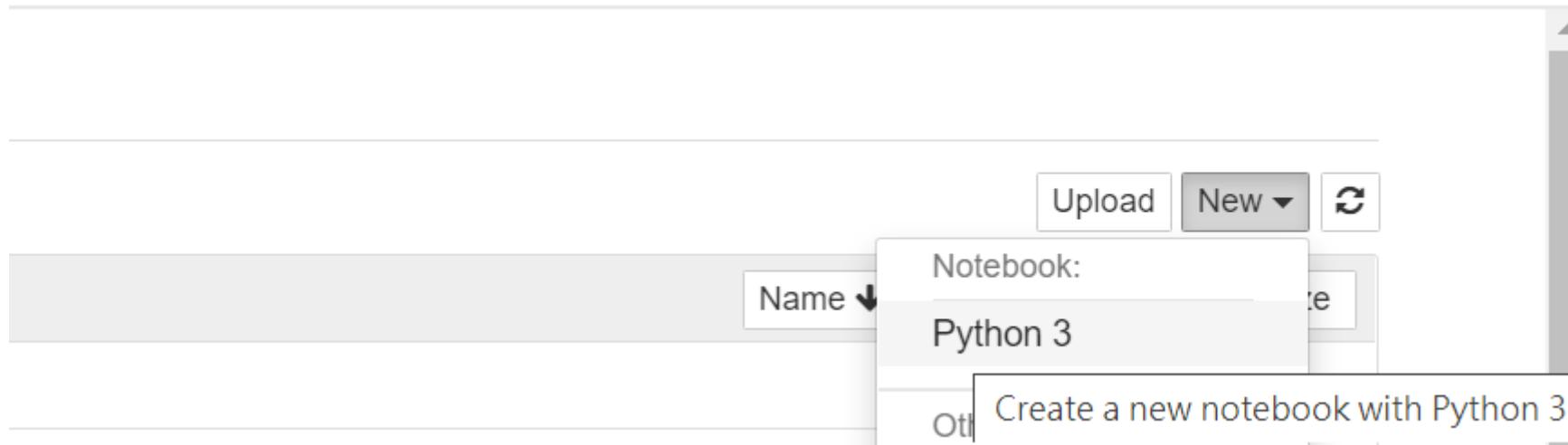
C:\WINDOWS\system32\cmd.exe

C:\Users\User>jupyter notebook

接下來如果你可以進入以下網頁，就算是 jupyter notebook 安裝成功

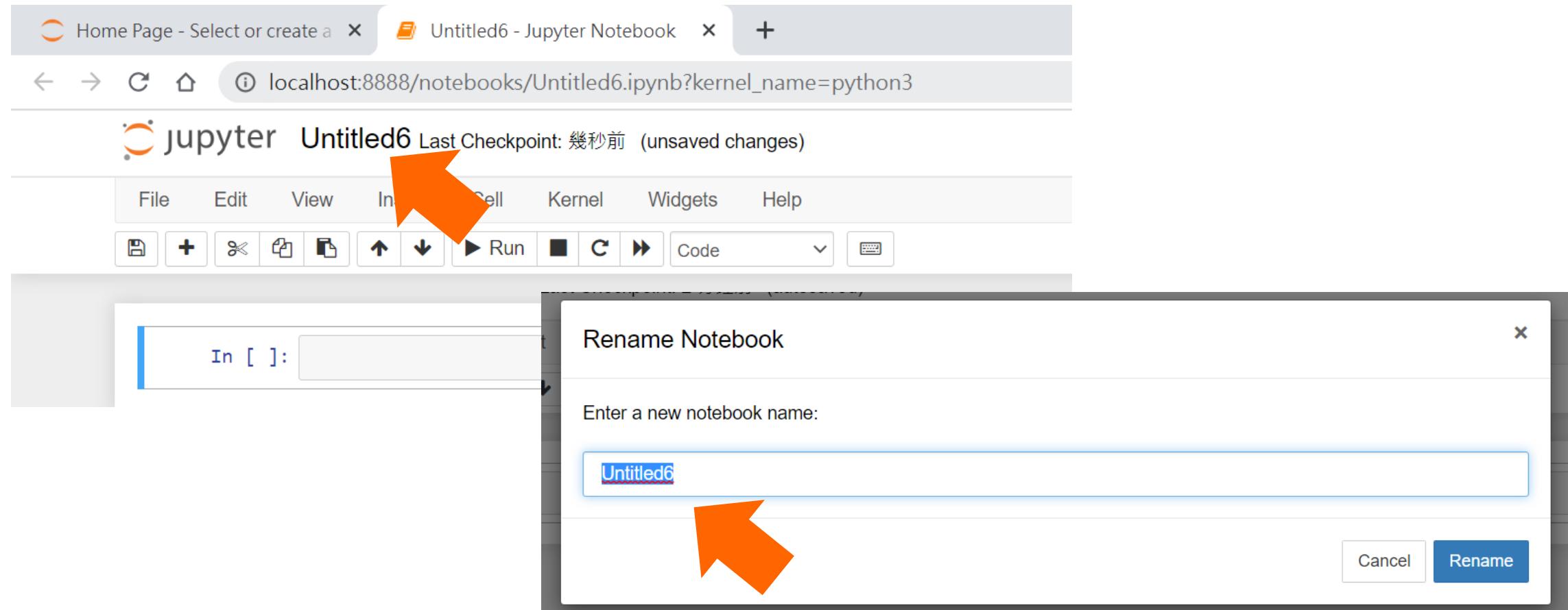


點選 New >> Python 3

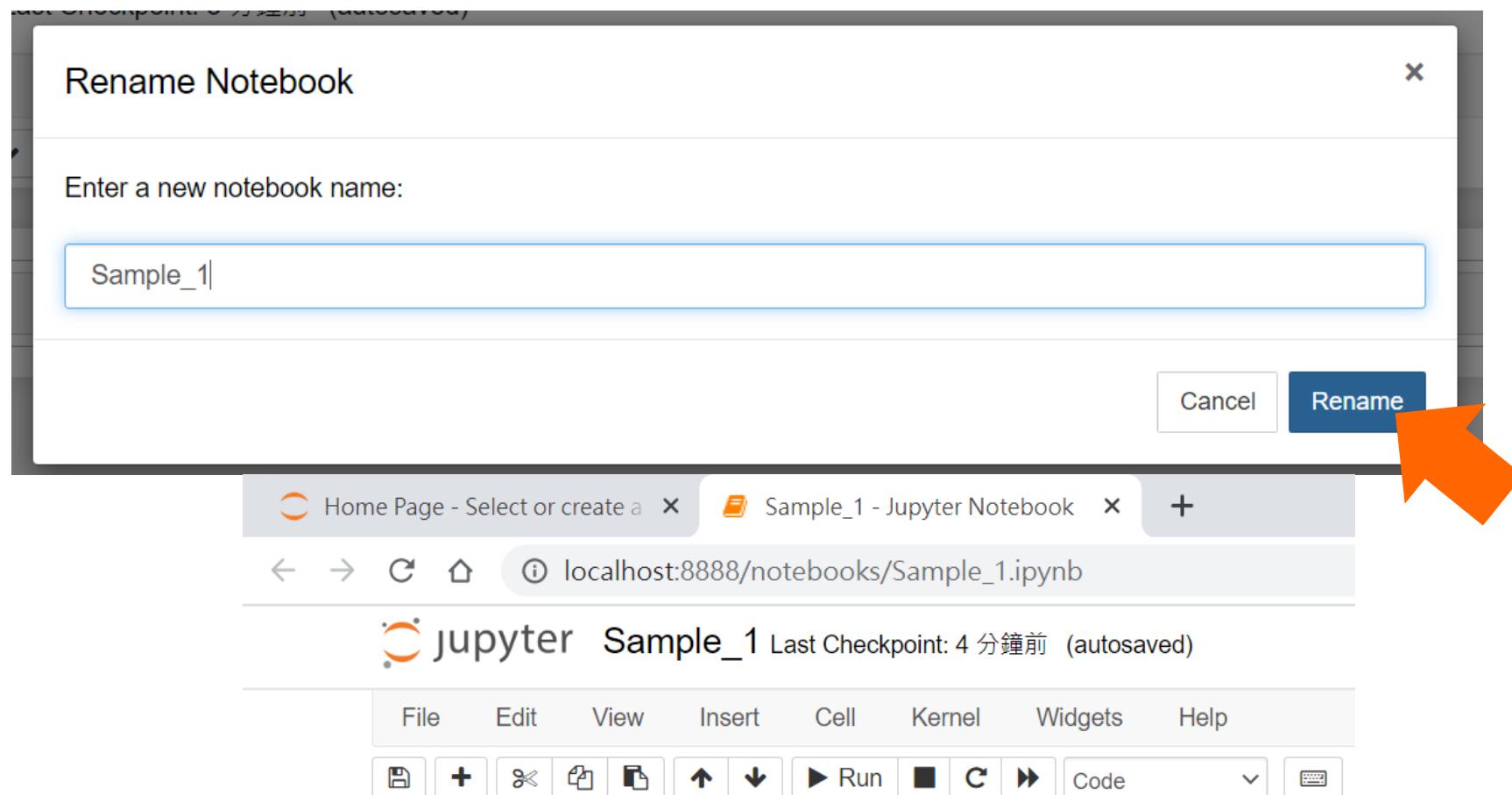


## 1-3 安裝 jupyter notebook

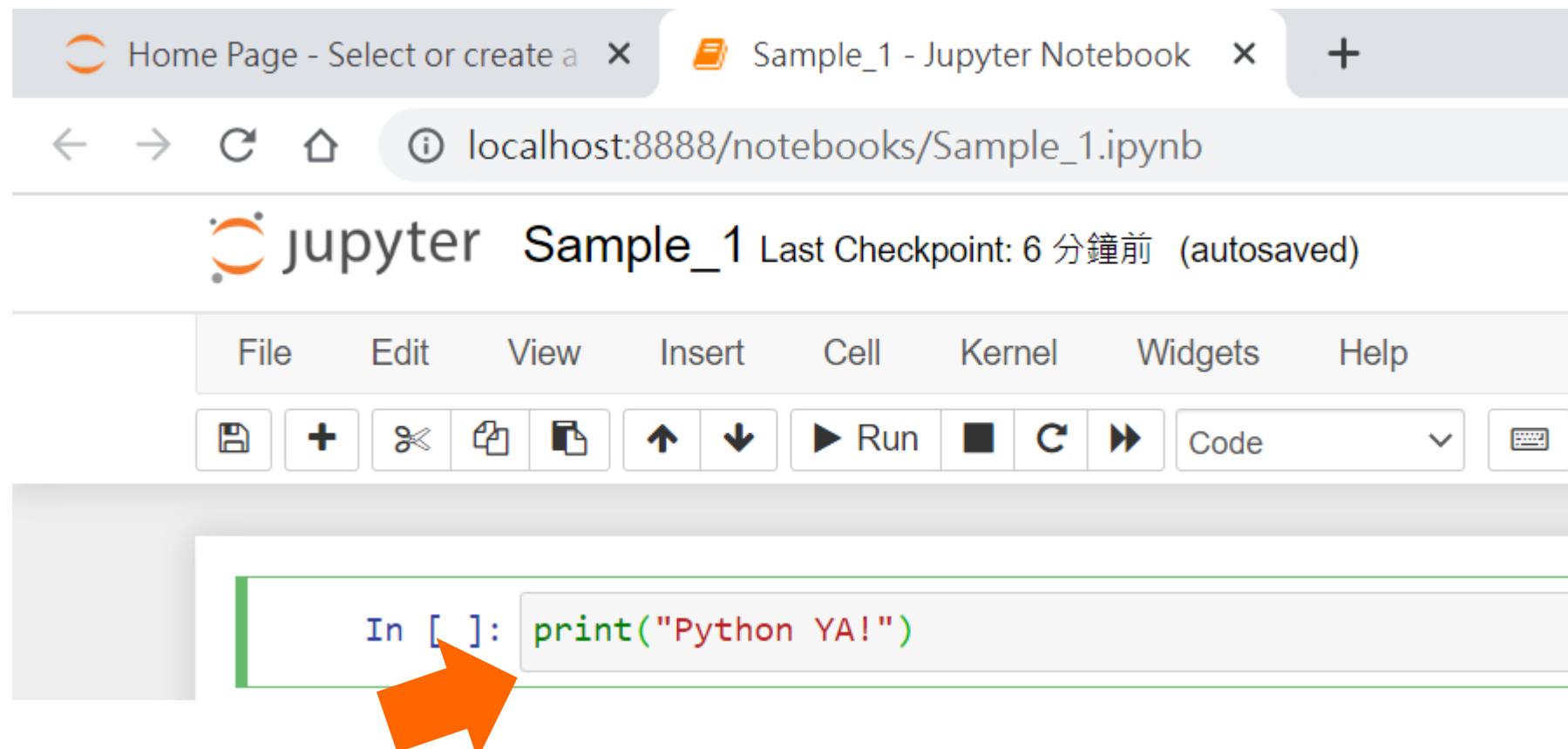
可以點選檔案名稱，這裡是 Untitled6，然後可以改檔名



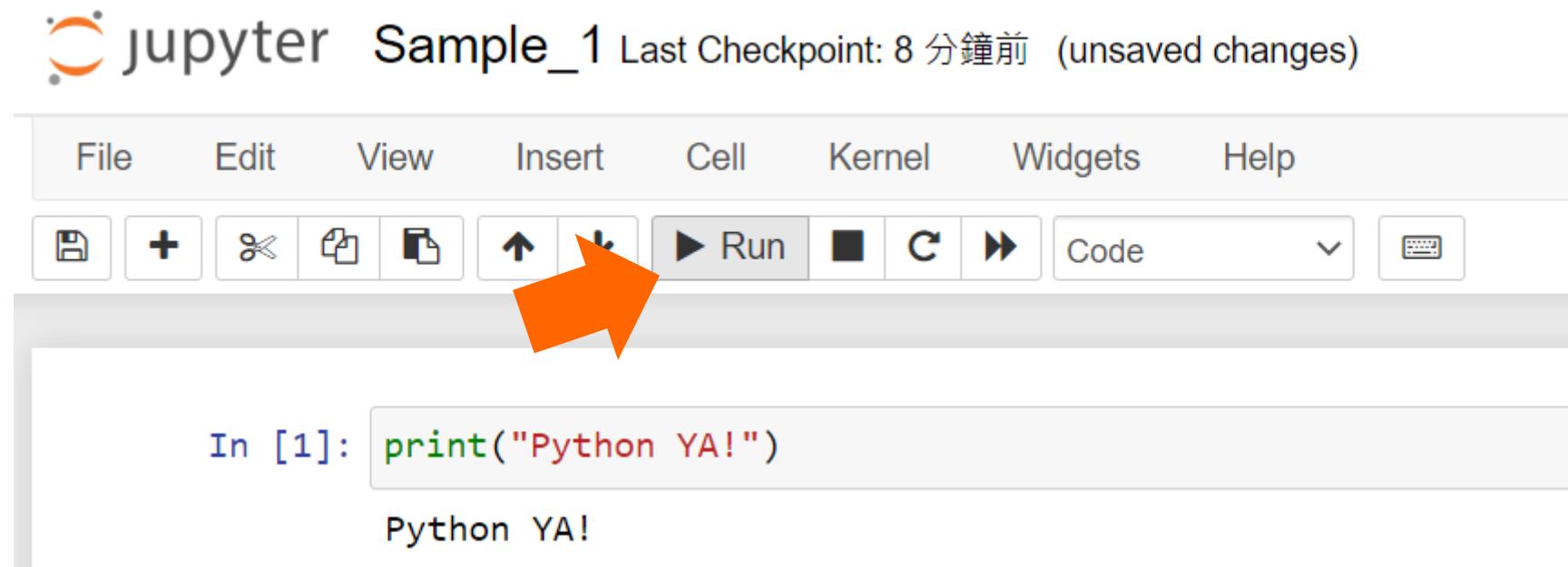
這裡我先命名 Sample\_1



接著可以在裡面寫程式，程式碼如下：



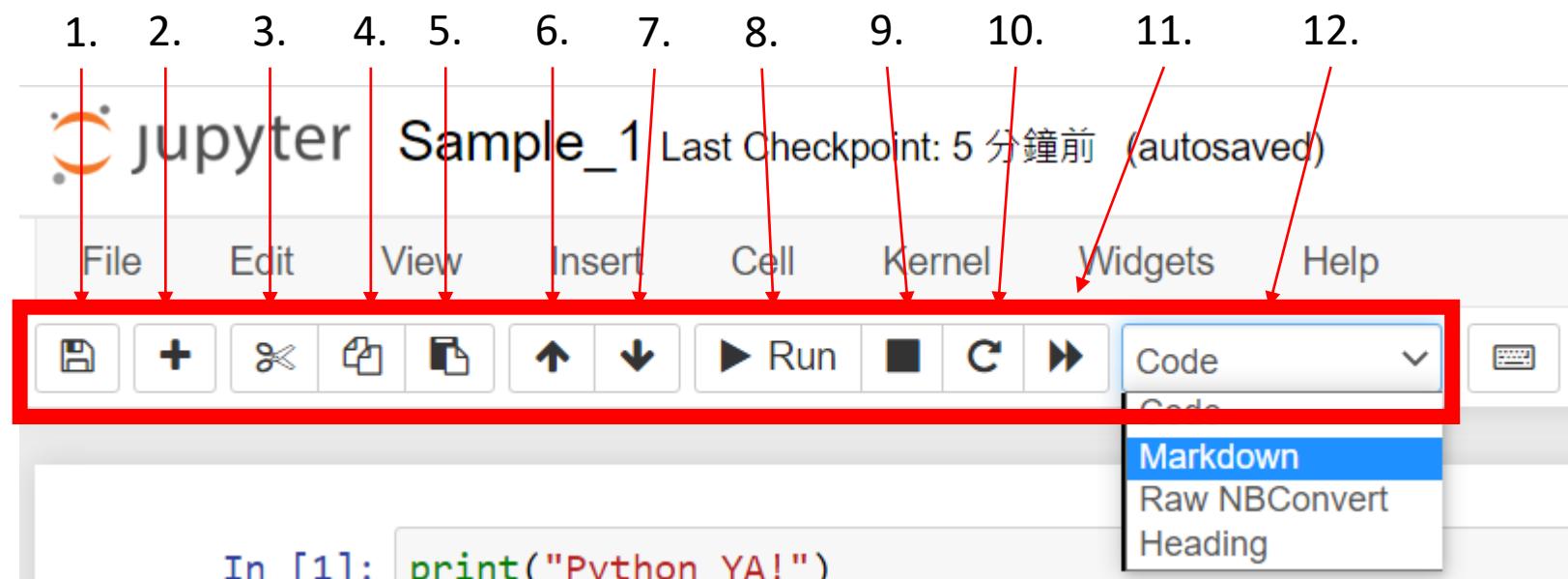
然後按 Run 就可執行：



# 1-3 安裝 jupyter notebook

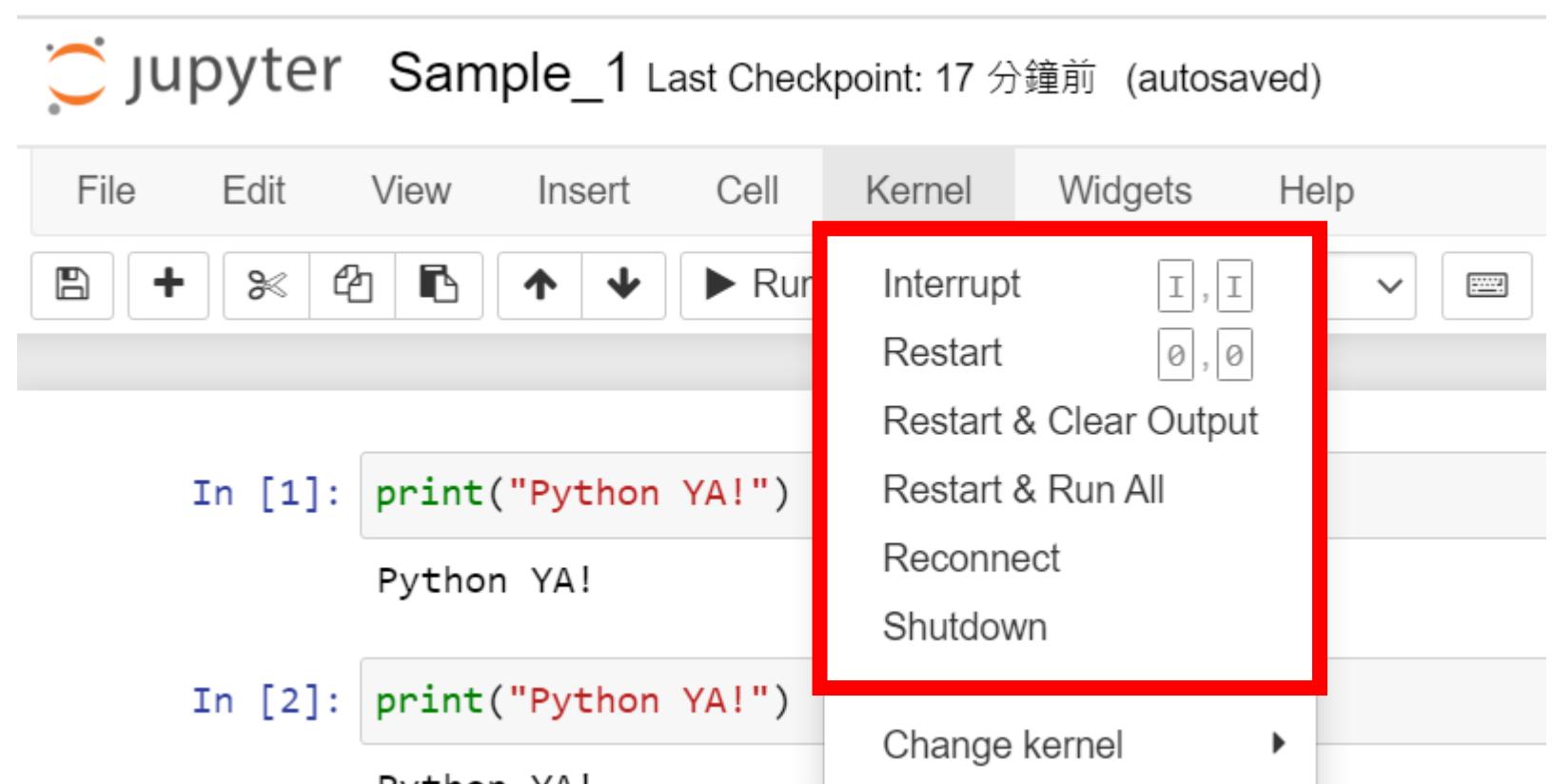
比較常用的是紅框內的：

1. 儲存檔案
2. 向下增加區塊
3. 把選定區塊刪掉
4. 複製區塊
5. 貼上區塊
6. 上移區塊
7. 下移區塊
8. RUN
9. 程式停止
10. 重新執行程式
11. 重啟並執行程式
12. Code or Markdown  
(Markdown 部分不會執行)



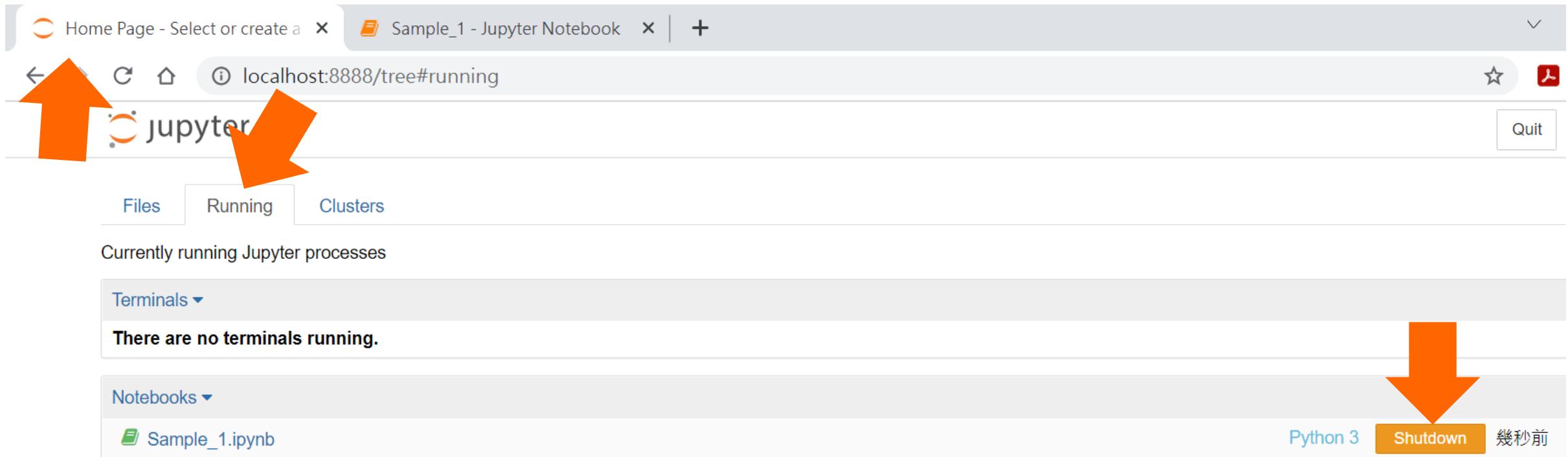
# 1-3 安裝 jupyter notebook

比較常用的是紅框內的：



比較常用的：

Home Page >> Running >> Shutdown



~ Jupyter notebook 就介紹到這，這門可主要用 PyCharm ~

- Python 程式已經安裝完了，其實也已經可以開始寫程式，不過如果有更方便我們寫程式的工具那就是 PyCharm 。

<https://www.jetbrains.com/pycharm/>



<https://www.jetbrains.com/pycharm/> ▾ 翻譯這個網頁

[PyCharm: the Python IDE for Professional Developers by ...](#)

PyCharm is the best IDE I've ever used. With PyCharm, you can access the command line, connect to a database, create a virtual environment, and manage your ...



# 1-4 安裝 PyCharm

PyCharm 2021.2.2



Version: 2021.2.2  
Build: 212.5284.44  
15 September 2021

[System requirements](#)

[Installation Instructions](#)

## Download PyCharm

[Windows](#)

[macOS](#)

[Linux](#)

### Professional

For both Scientific and Web Python development. With HTML, JS, and SQL support.

[Download](#)

[Free trial](#)

### Community

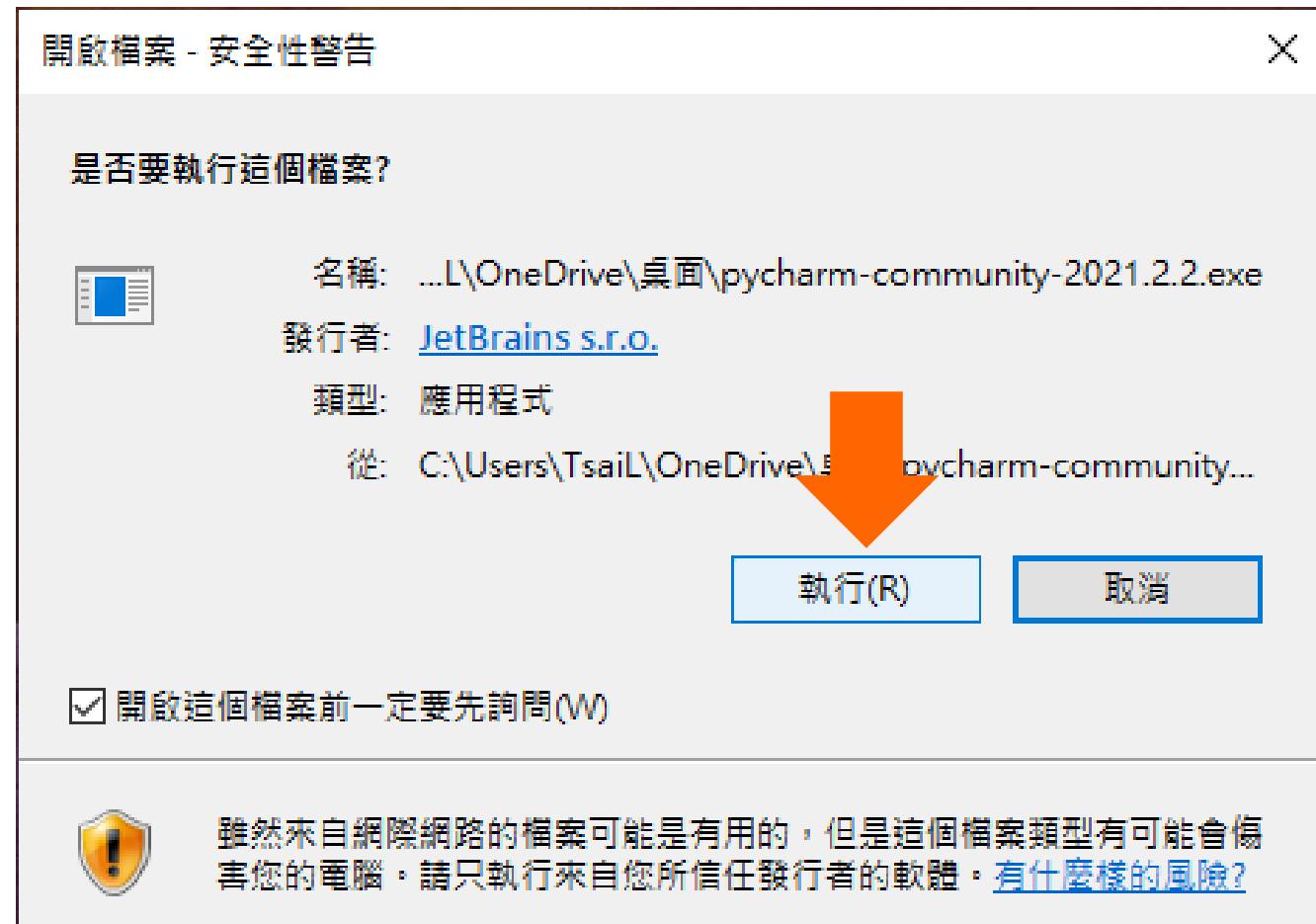
For pure Python development



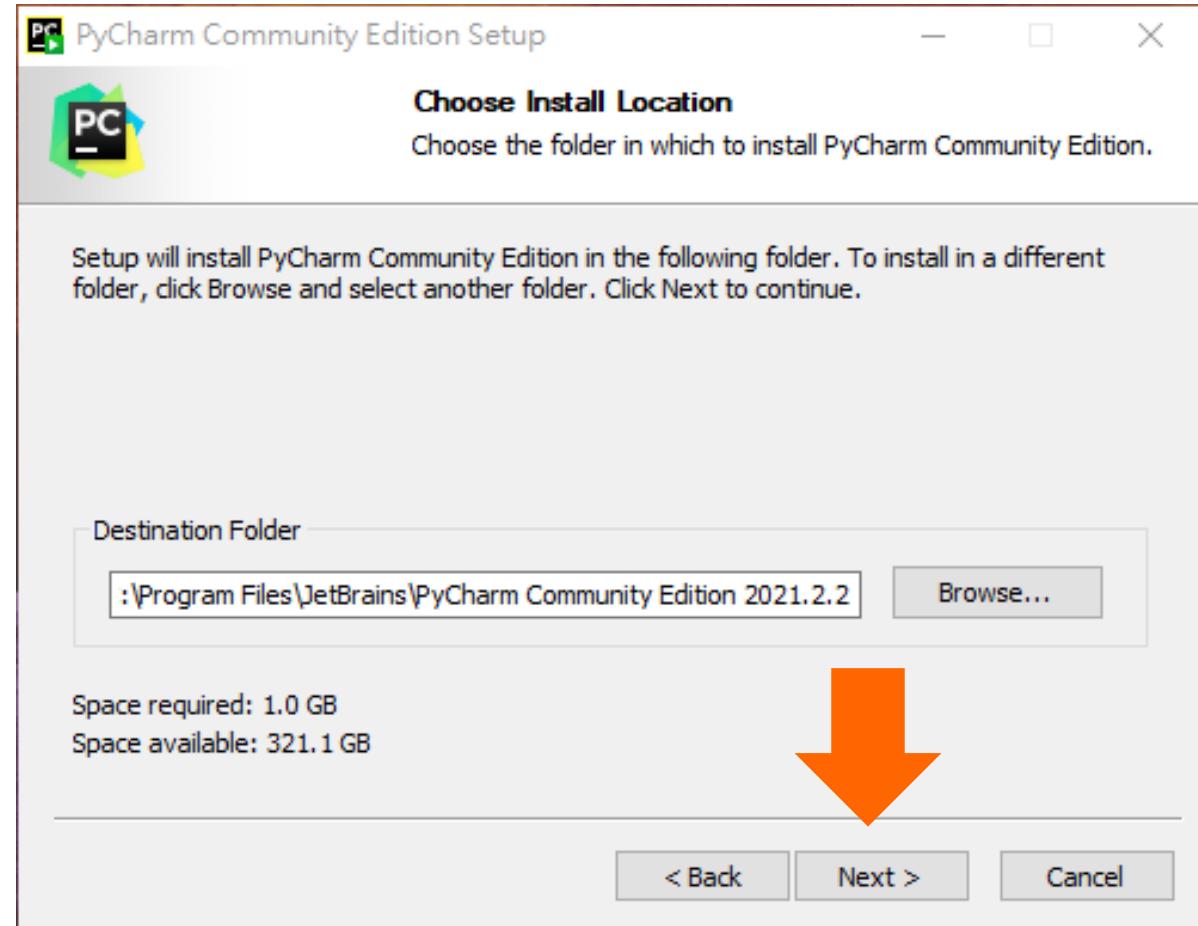
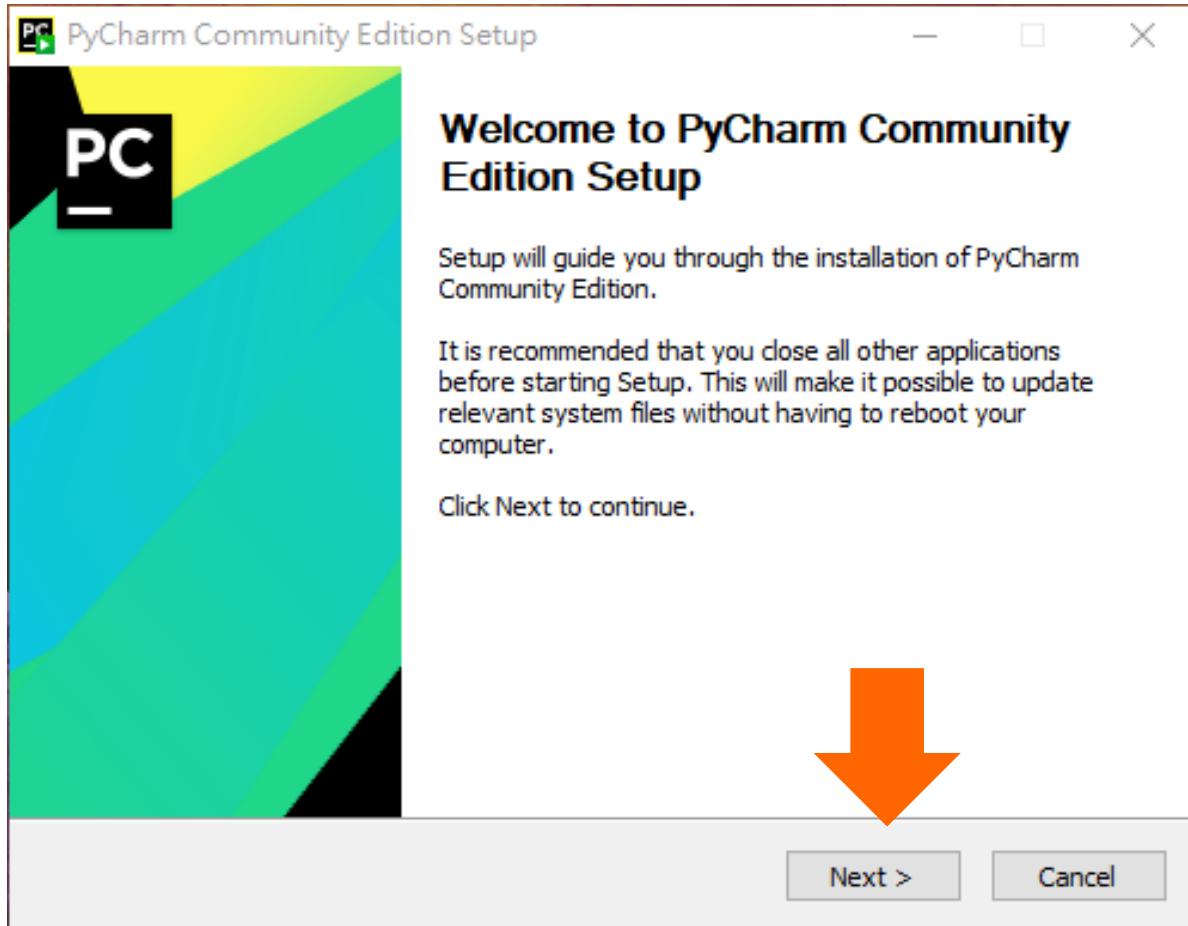
[Download](#)

Free, built on open-source

# 1-4 安裝 PyCharm

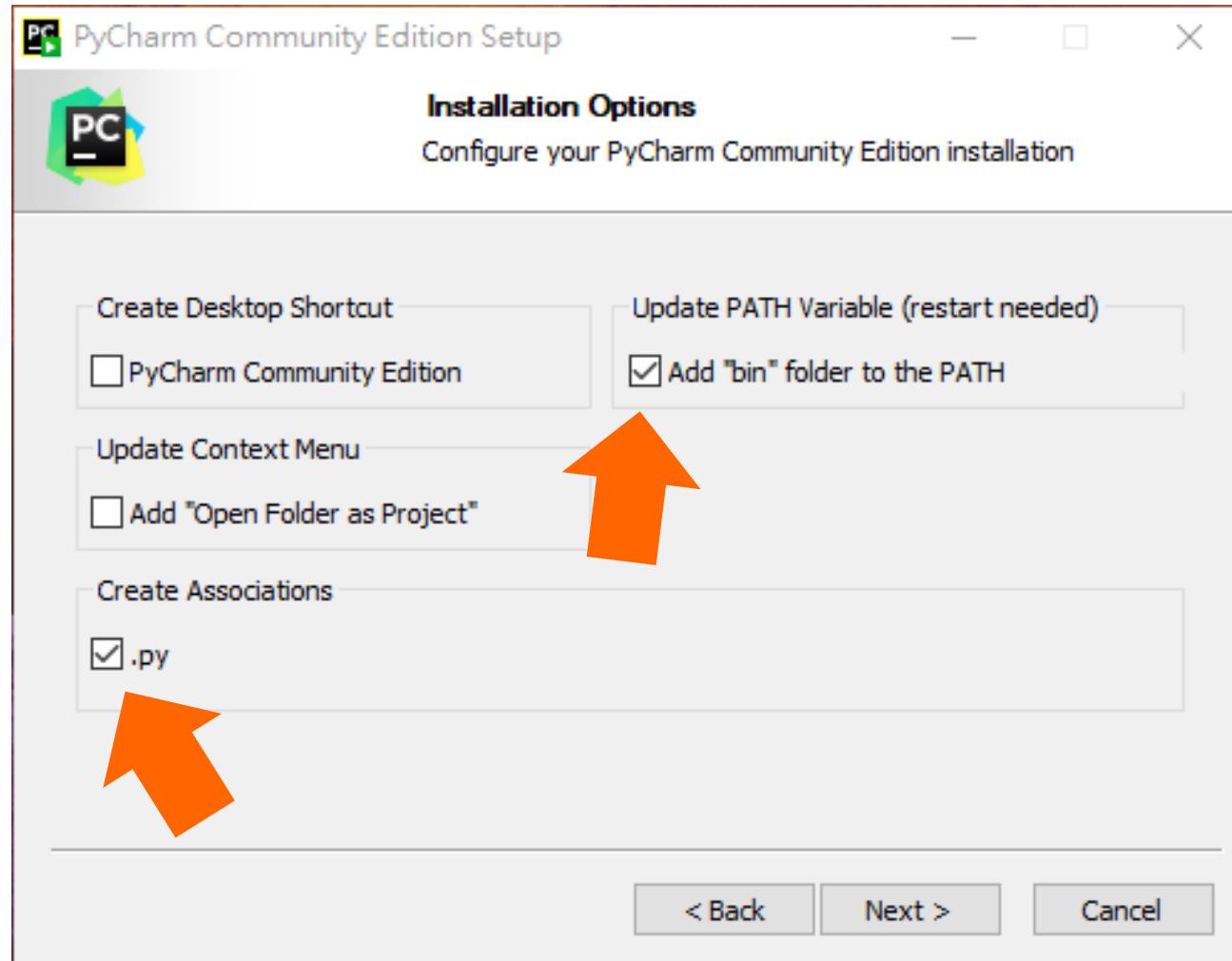


# 1-4 安裝 PyCharm



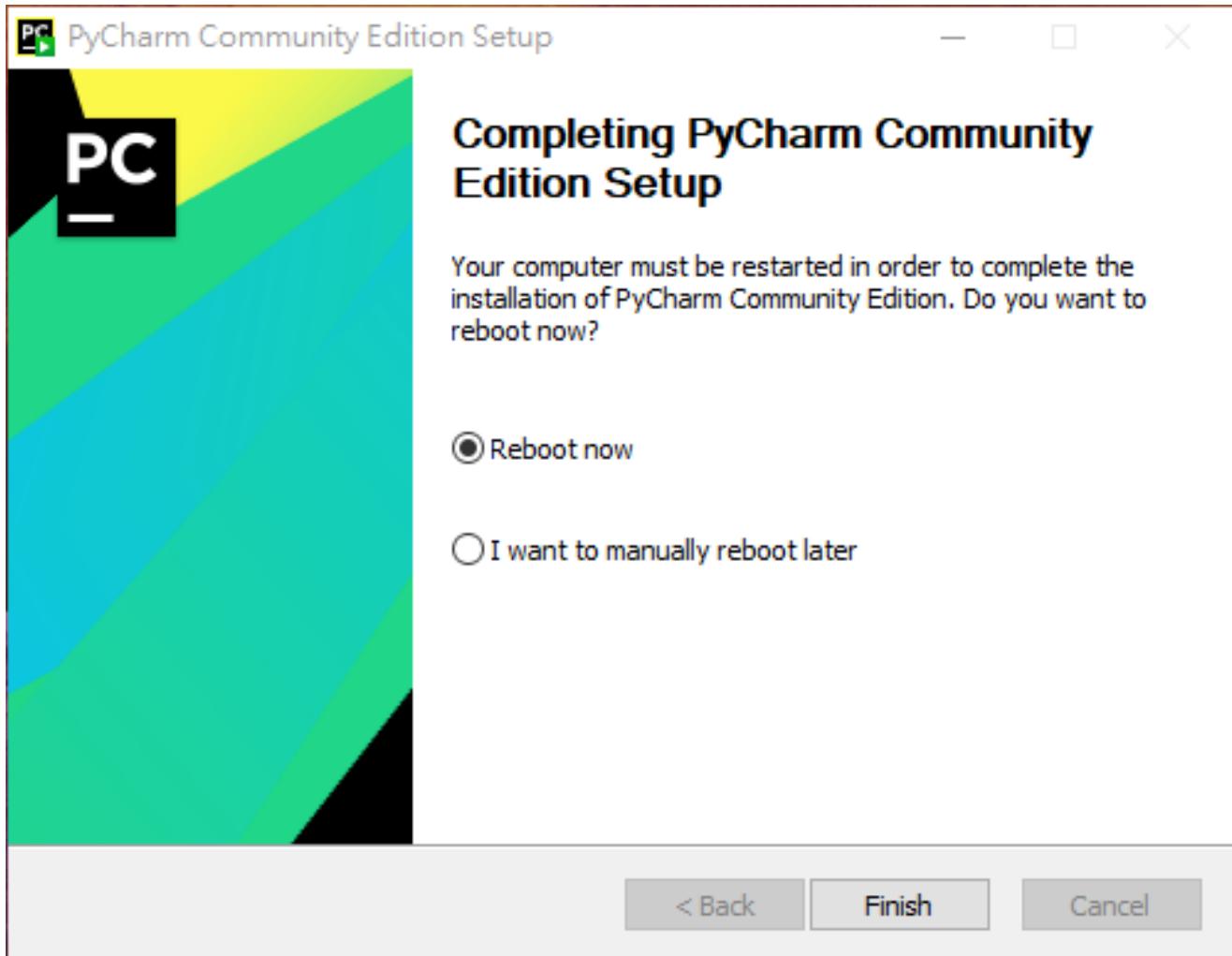
# 1-4 安裝 PyCharm

- 要打勾喔! 。

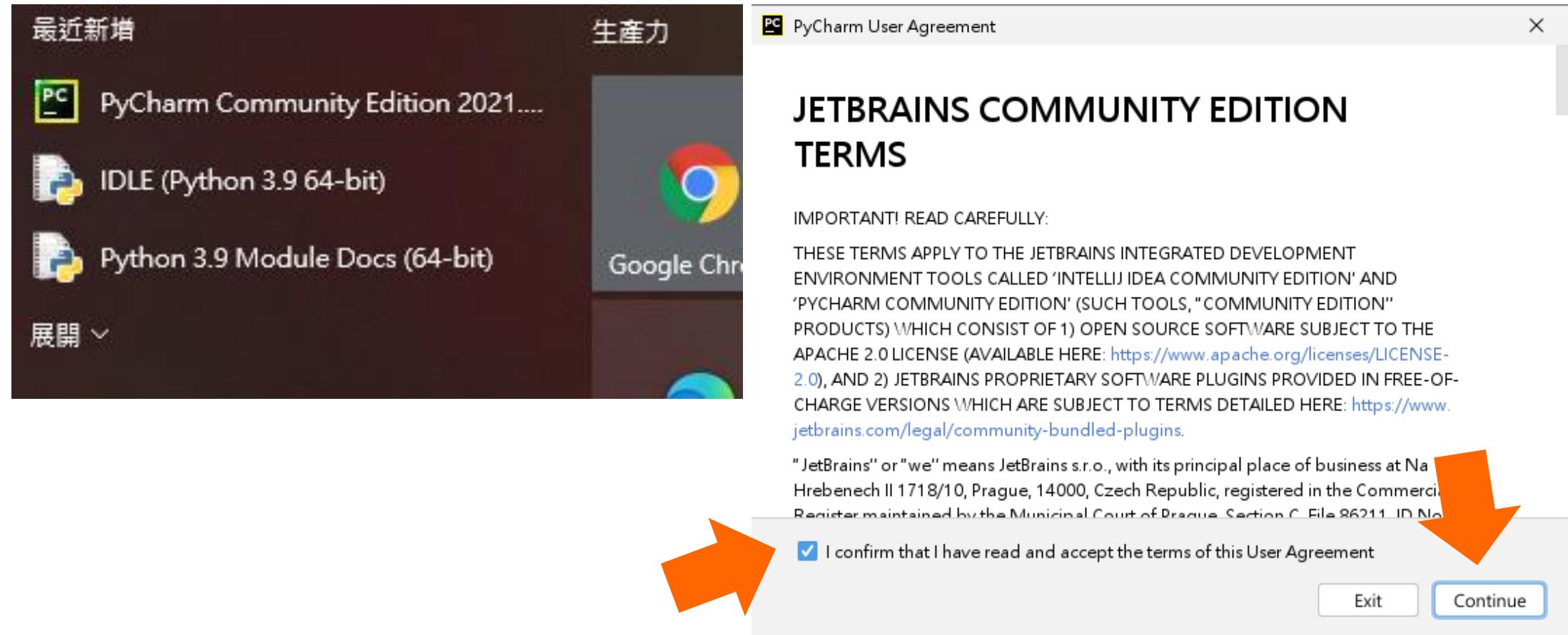


## 1-4 安裝 PyCharm

- 要重新啟動才會完成安裝。



## 1-4 安裝 PyCharm



這時可以自己創一個空的資料夾

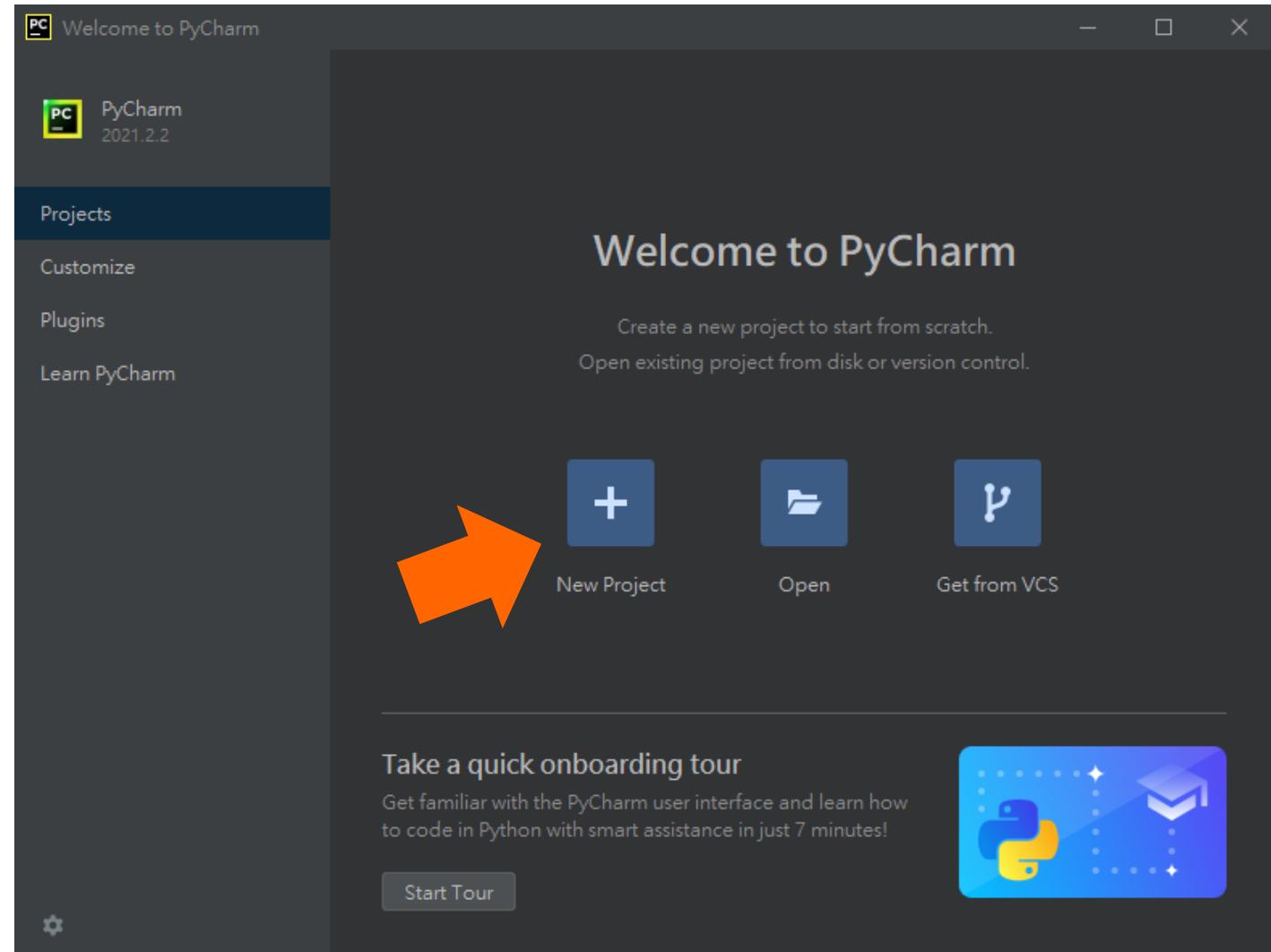
D:\Python\workspace



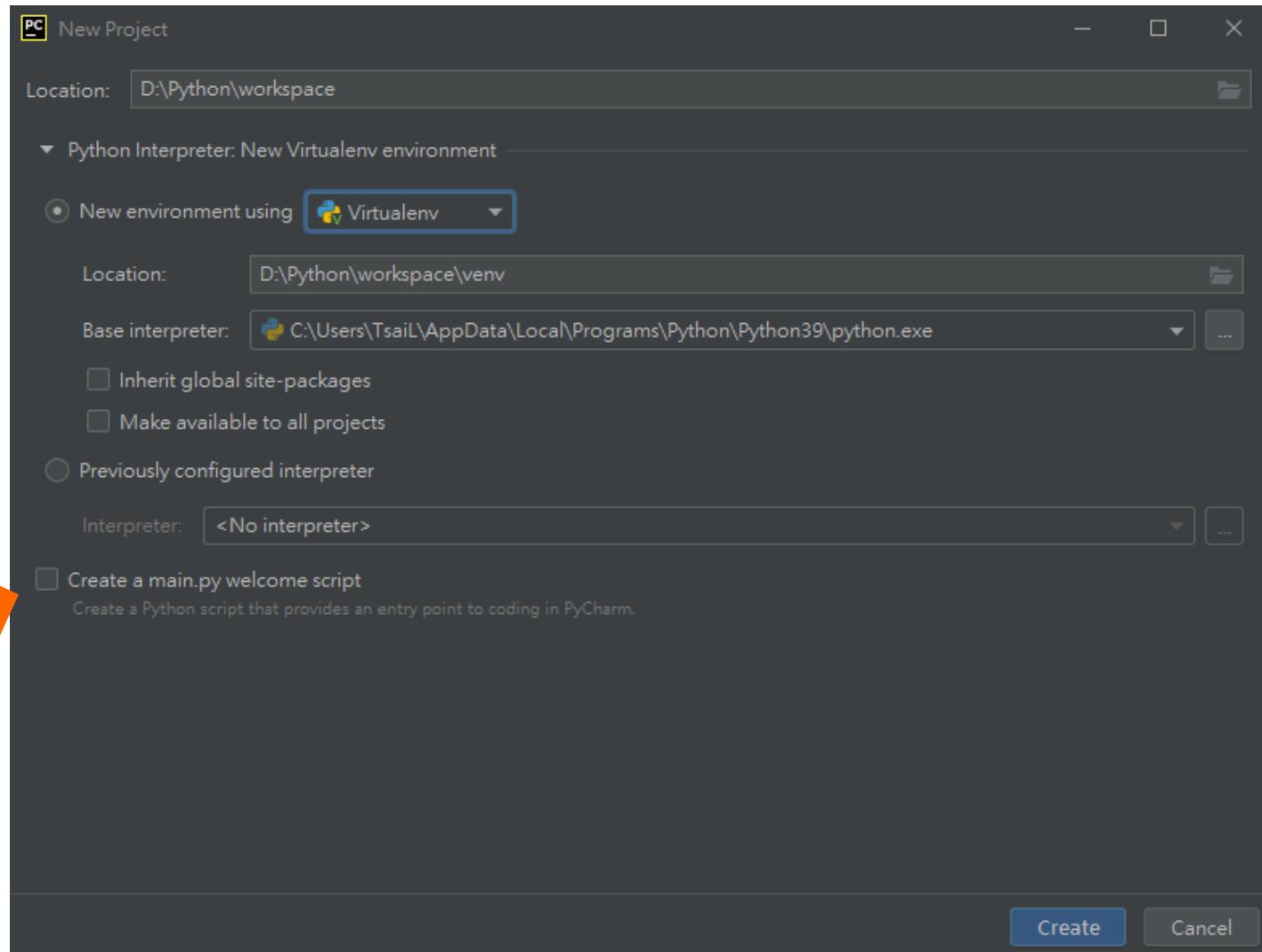
~程式碼請不要隨便放，請找好資料夾放~

# 1-4 安裝 PyCharm

New project



# 1-4 安裝 PyCharm



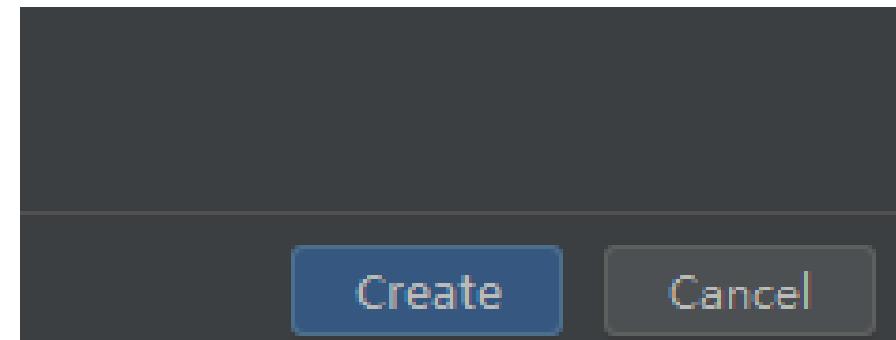
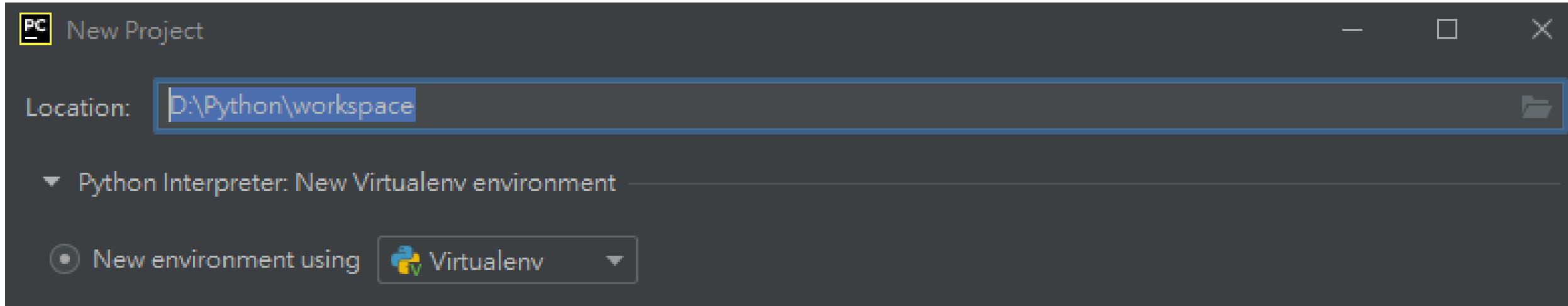
不要勾



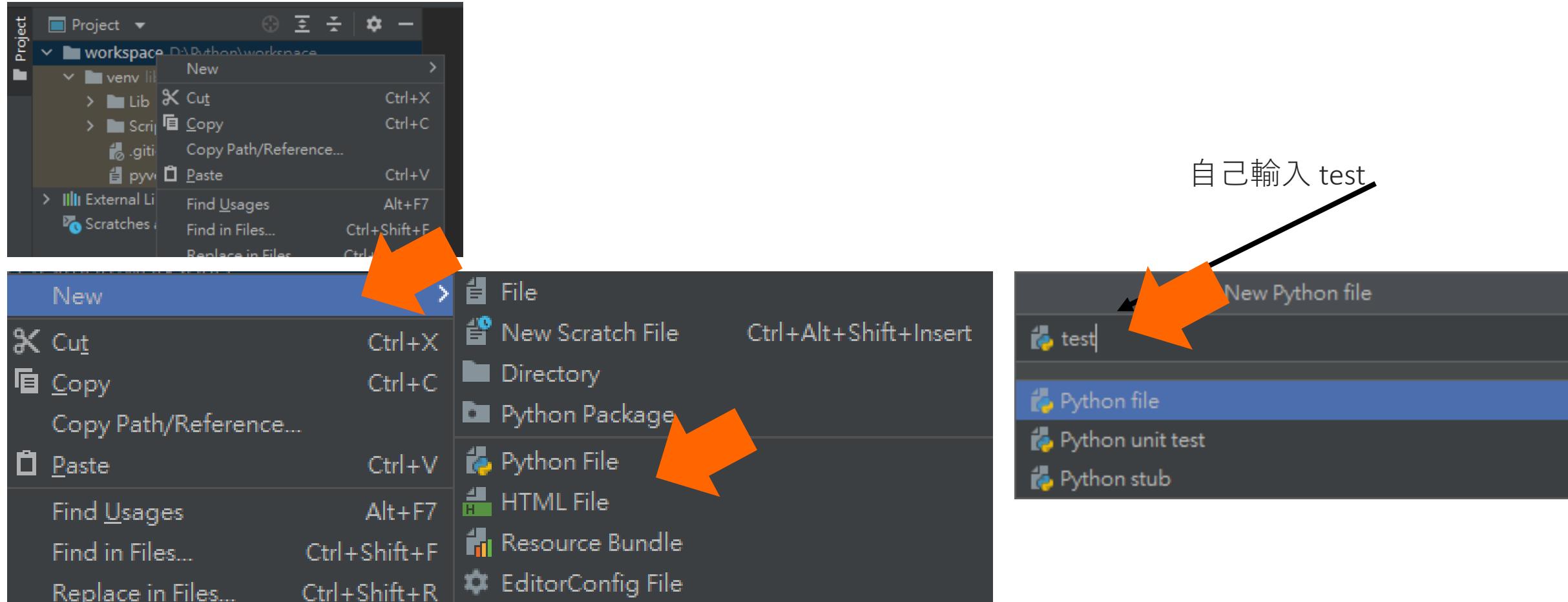
選擇資料夾



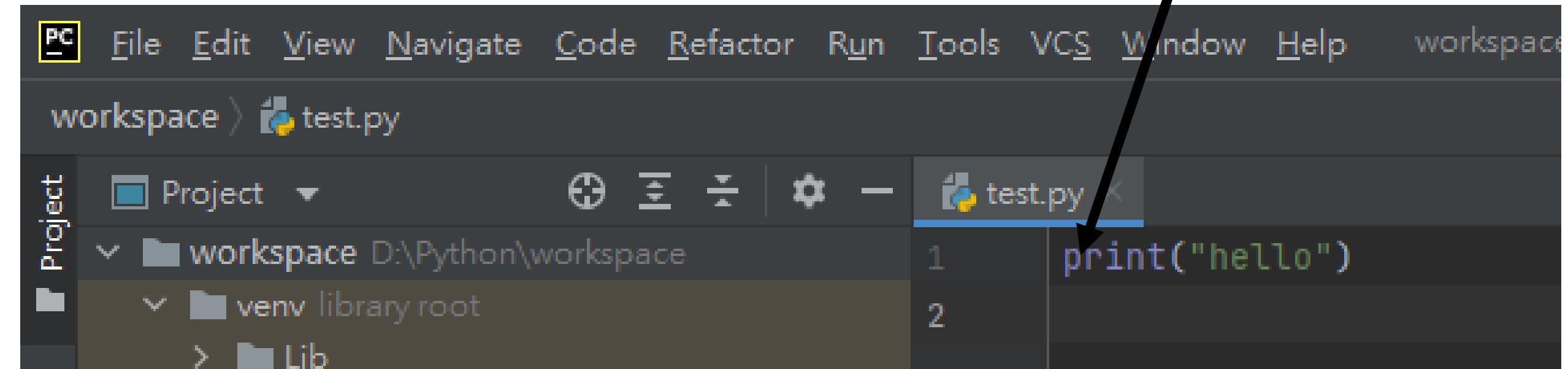
D:\Python\workspace



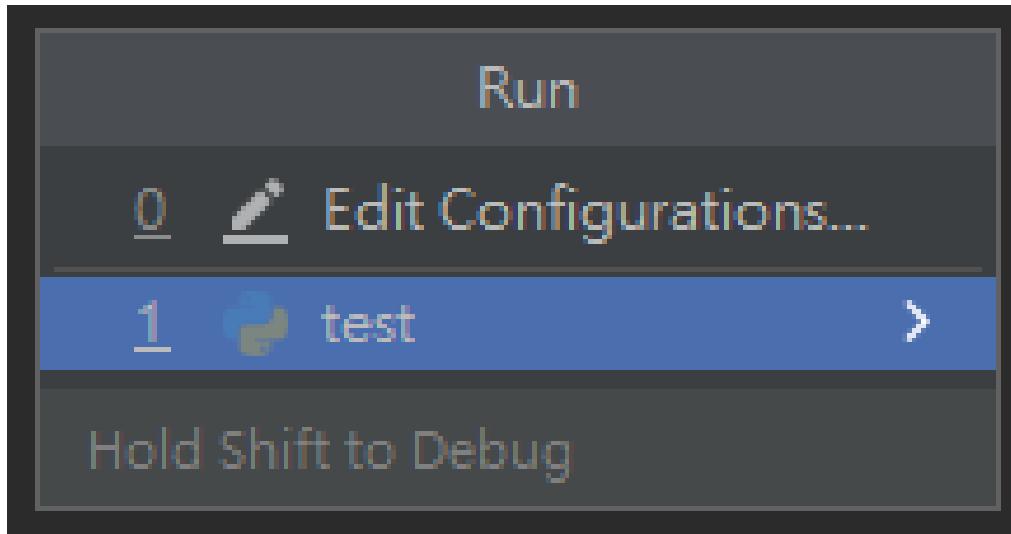
在 workspace 點滑鼠右鍵 >> New >> Python File



自己輸入  
print("Hello")



同時按 Alt + shift + f10



如果按 Alt + shift + f10 之後沒有  
下面的視窗，  
就請砍掉 workspace 資料夾，  
再完全重新 create

按 enter

```
D:\Python\workspace\venv\Scripts\python.exe D:/Python/workspace/test.py
hello

Process finished with exit code 0
```

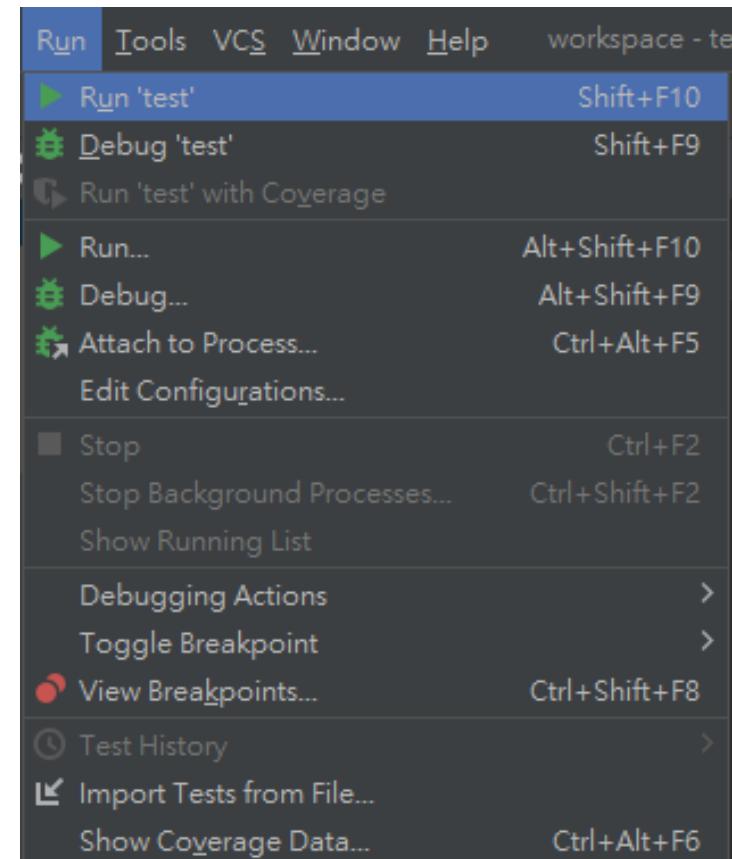
~到目前為止 Pycharm 安裝成功！~

到目前為止有兩個指令請一定要記一下：

1. **Alt + shift + f10** 選擇要 run 哪個檔，例如 test。

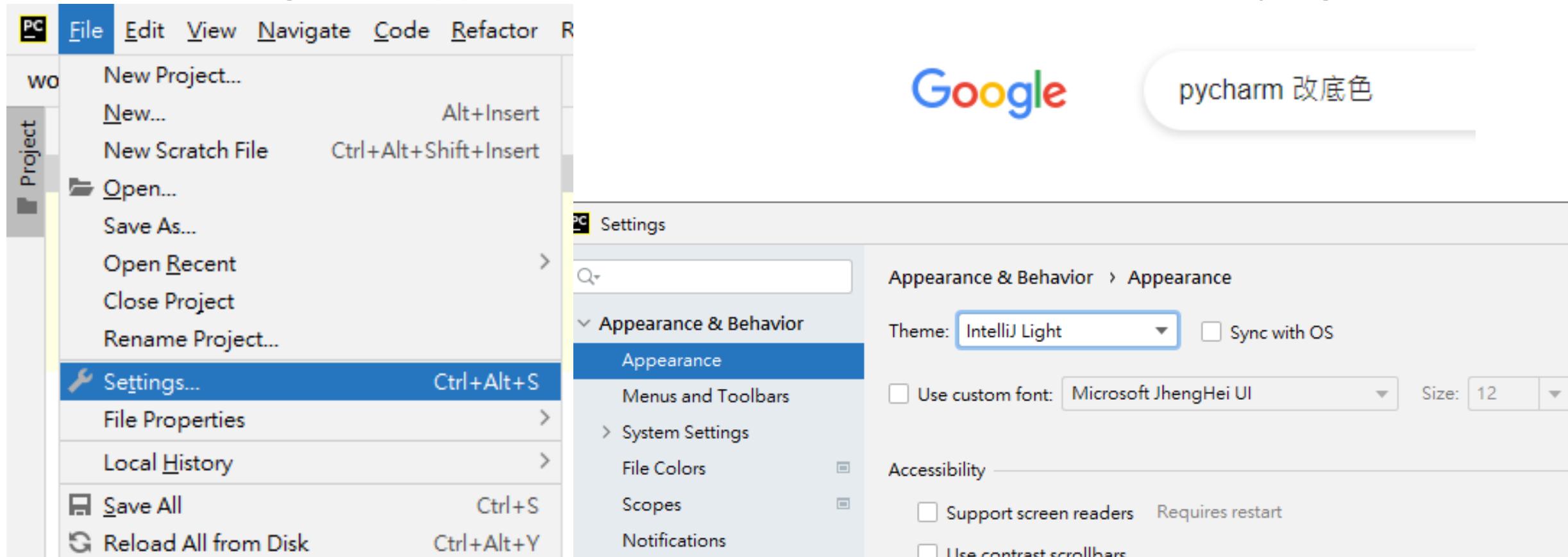
2. **Shift + f10** 上一步驟選好後，

接著按Shift + f10 就直接執行 test



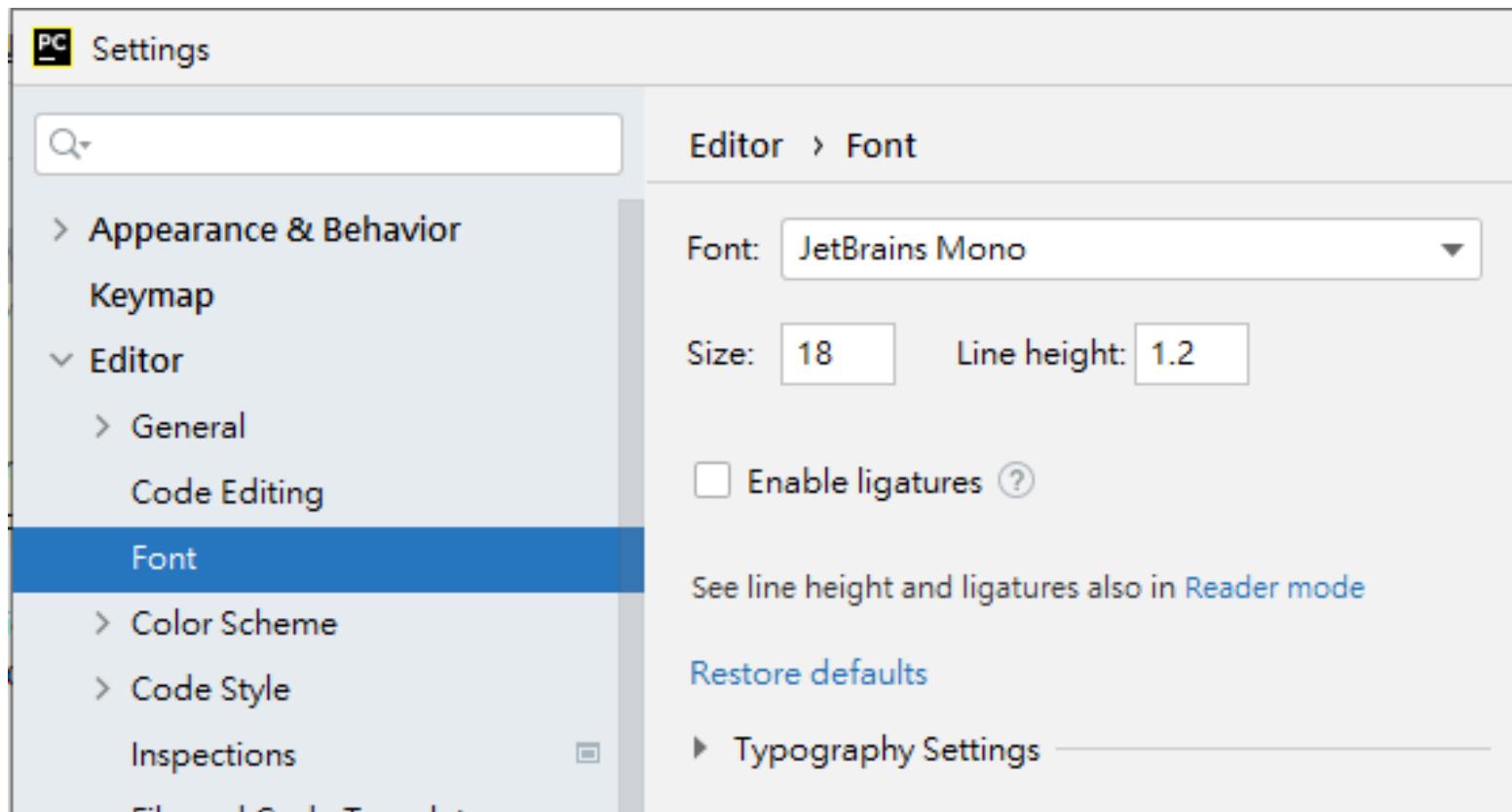
如果習慣看底色是白的可以更改：

File >> Settings >> Appearance & Behavior >> Theme >> IntelliJ Light



如果覺得字太小，可以改字體大小

File >> Settings >> Editor >> Font >> size

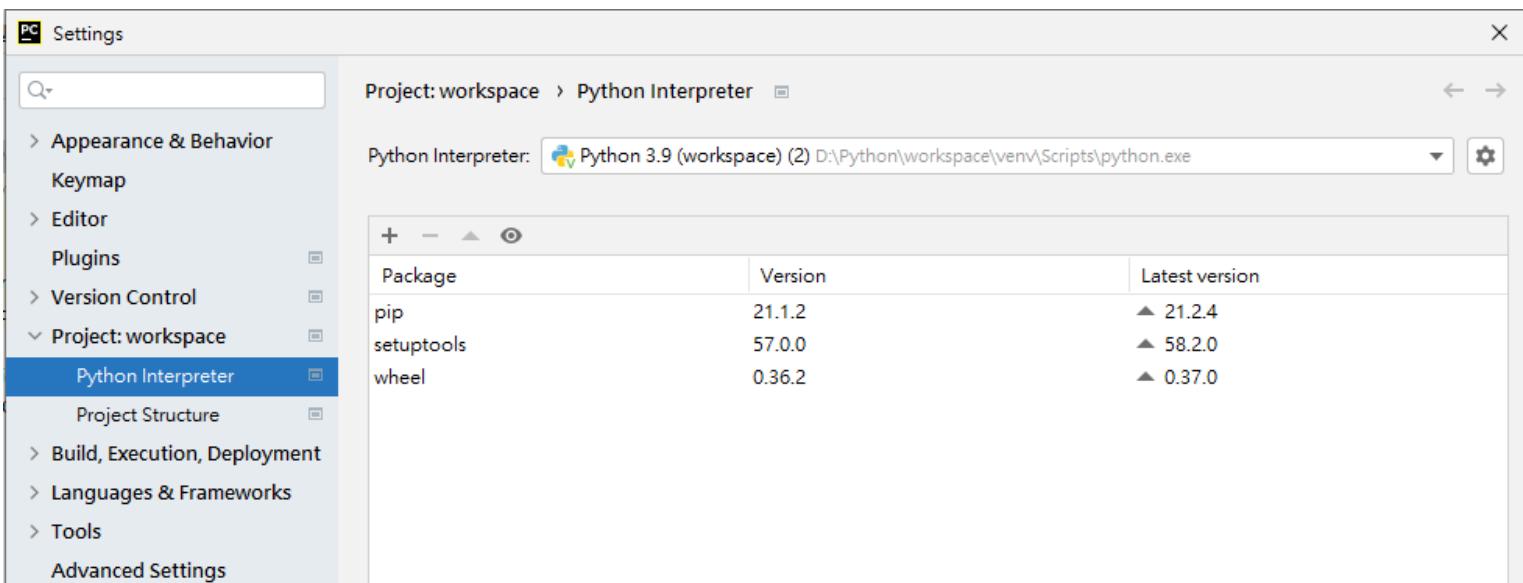


接著檢查

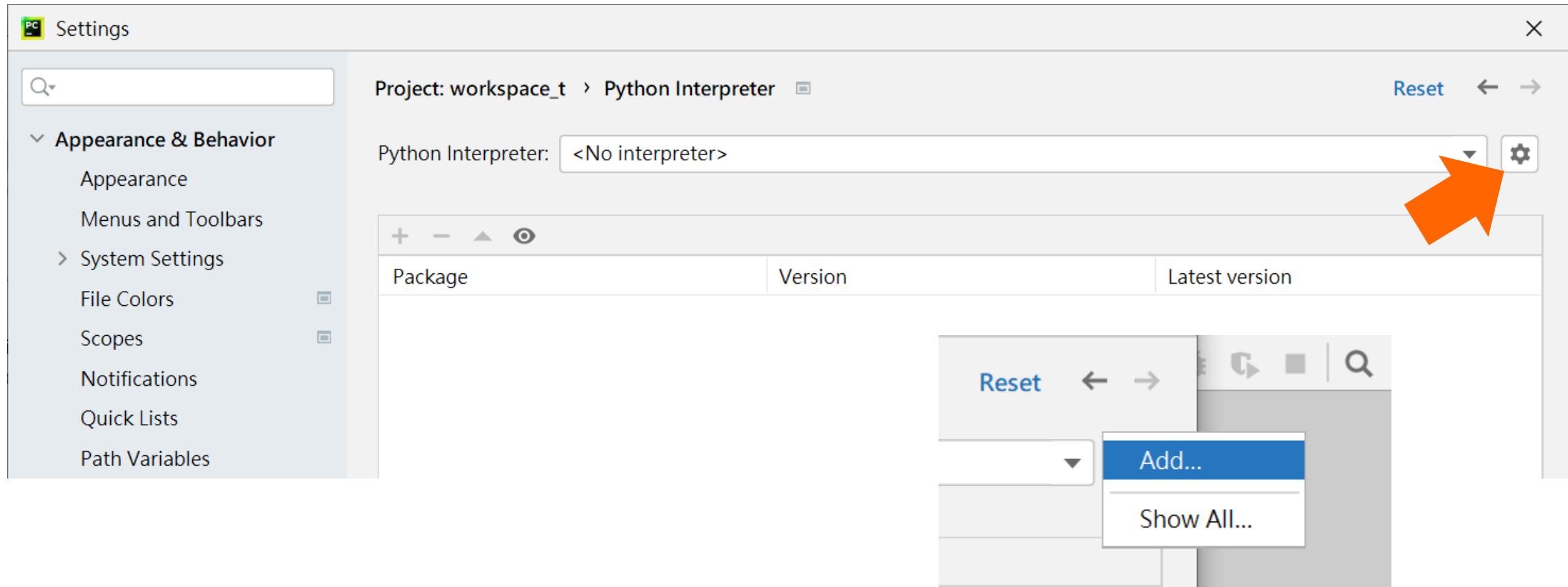
File >> Settings >> Project Interpreter

的內容，要有東西如下圖

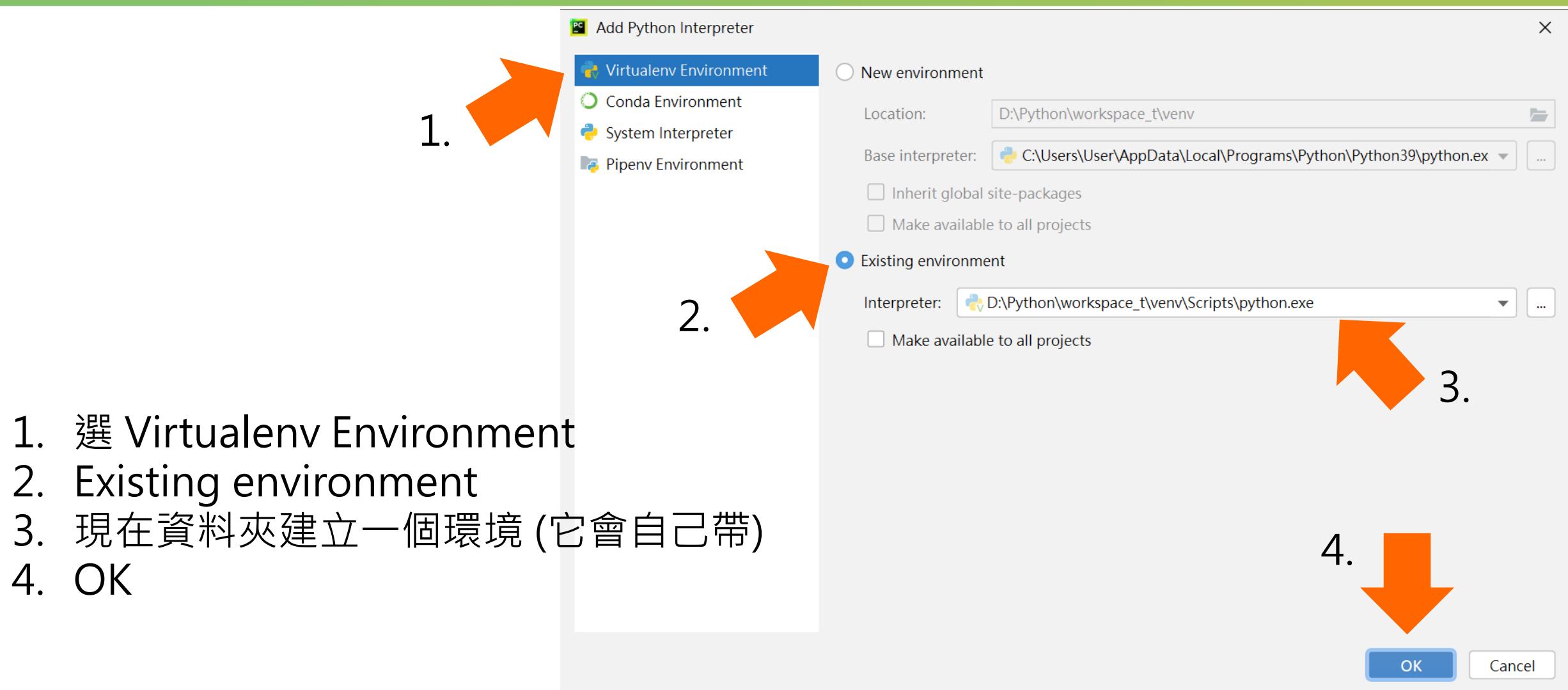
如果沒有的話那先按右邊的齒輪



先按右邊的齒輪 >> Add



# 1-4 安裝 PyCharm

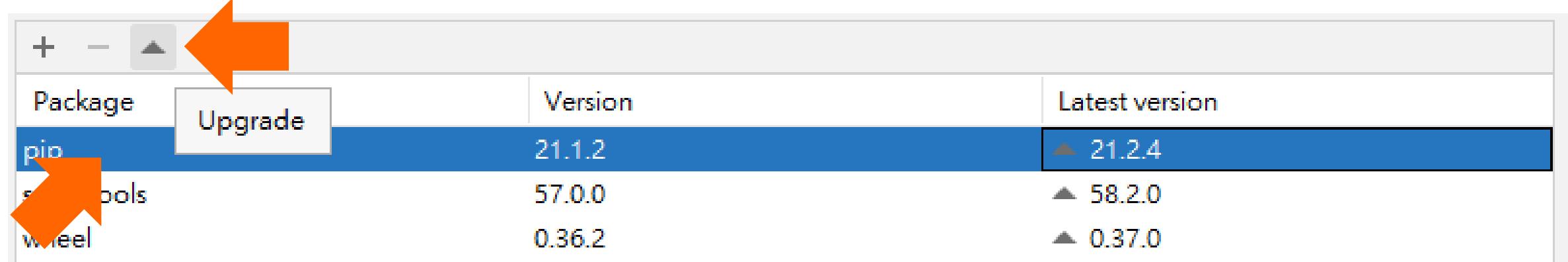


接著檢查

File >> Settings >> Project Interpreter

的內容，要有東西，如果有東西就順便把 pip upgrade 一下。

點 pip 然後按上三角形



# 1-4 安裝 PyCharm

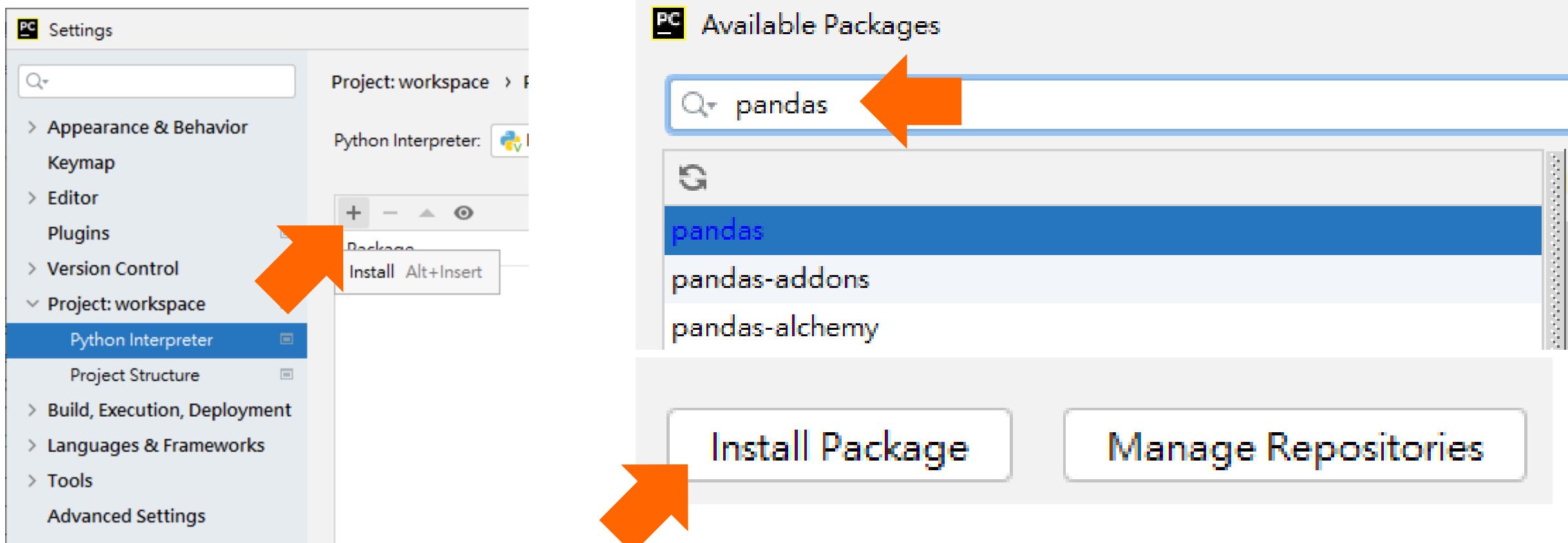
Package	Version	Latest version
pip	21.2.4	21.2.4
setuptools	57.0.0	▲ 58.2.0
wheel	0.36.2	▲ 0.37.0

Package pip successfully upgraded

如果 upgrade 成功，  
就會顯示 successful

如果失敗的話至少還有 21.2.2 可用

如果之後如果在 PyCharm 的工作環境要安裝套件：  
Python Interpreter >> + >> install package



# Module 2 :

## Python 程式第一步

2-1 程式的意義

2-2 機器碼 & 直譯器

2-3 互動模式 vs. 腳本模式

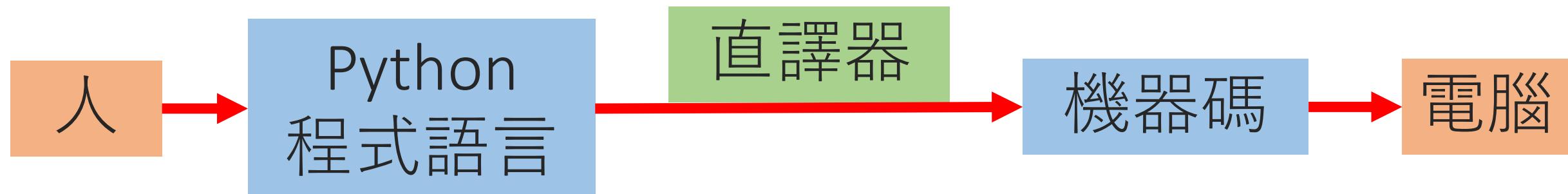
- 平時我們處理的工作可分為四大類：重複性動腦、非重複性動腦、重複性勞動與非重複性勞動。學習程式是為了協助我們處理「重複性動腦」的工作。
- 既然是重複性動腦，那麼我們想像一個情境。假設一位發號司令的長官，長官需要的部下是一個口令一個動作，一堆口令一堆動作，其中需要執行的事情需要把長官的想法有邏輯的寫在紙上，而看到這張紙的部下就能夠依照紙上的內容，來完成我們在紙上寫下的執行「命令」，但在執行的前提是部下要看得懂紙上寫的內容。以上其中的部下就是電腦，紙上的內容就是程式，而**程式不是隨便亂寫的，要有邏輯而且電腦要看的懂**。也就是需要依照程式的語法完成要傳達的邏輯。

## 2-1 程式的意義

- 在我們生活中會常遇到許多生活中常遇到的事物會有程式的協助。例如：紅綠燈、ATM、捷運進出站FRID系統、晶片控制...等。當然，不同的目的會使用不同的程式語言，有些適合用 C、JAVA、PLC...等。在大數據分析的適合語言是 Python。
- 因為 Python 有非常多的套件，能夠協助你處理 AI 大數據。
  - ~ 這一門課會讓你了解 Python，讓你先學程式的基本功，之後遇到套件的時候你就能比較好應對~
  - ~ 當然接下去的其他門課也會教那一門課所需的套件應用~

學習 Python 就是讓我們和電腦溝通的方式。

首先我們會將想做的事用Python這個程式語言寫成程式，接著按下執行鍵，之後它會經過「直譯器」然後轉為「機器碼」，所謂的「機器碼」是由「0」，「1」組成，而電腦看得懂「機器碼」。直譯器扮演的角色就像翻譯人員。



## 2-3 互動模式 v.s. 腳本模式

互動模式：一個口令一個動作。

C:\WINDOWS\system32\cmd.exe - python

```
C:\>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>>
```

腳本模式：寫在一個文件裡，如果是用Python所寫那麼副檔名用py，接著用python 執行它。

C:\WINDOWS\system32\cmd.exe

```
C:\>python ./hello.py
Hello
```

~我們在 PyCharm 寫程式是腳本模式~  
~jupyter notebook 同時有互動模式和腳本模式~

# Module 3 :

## Python 的基本知識

3-1 輸出到螢幕上

3-2 註解(comment)

3-3 由上而下，依序執行

3-4 字串(string)

3-5 數值

3-6 2 進位、8進位、16進位

3-7 跳脫字元

3-8 原始字串

3-9 print() 的結尾

語法：  
`print()`

```
1 # print 是印出來顯示在螢幕上的語法
2 print("Hello") # 註解也可以寫在這裡
3 print("你好")
```

輸出結果

Hello

你好

符號、英數字是以半形的方式輸入。  
開頭要對齊，不要隨意在每列的開頭前輸入空白。  
(之後的章節會介紹何謂「縮排」)

```
1     # print 是印出來顯示在螢幕上的語法
2     print("Hello")  # 註解也可以寫在這裡
3     print("你好")
```

註解：「#」字號後面的就是註解

註解的目的：

註解是給人看的，不是給電腦看的，經註解的文字不會轉為機器碼，所以電腦是看不到註解的內容。

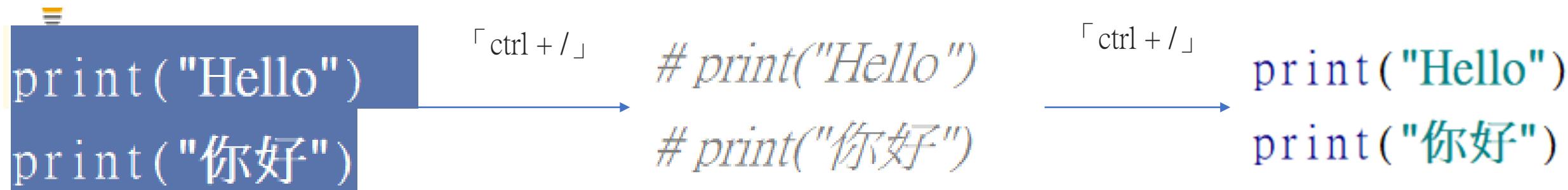
而寫註解的目的是為了說明一段程式的功能、目的及意義，以方便管理程式。或是有一段程式你暫時不想讓電腦看到它時可以先用將它註解起來。

## 3-2 註解

註解有單行也有多行：

單行註解範例如上，從「#」符號開始到當行最後的文字。

多行註解的方式是用選擇要註解的範圍用快捷鍵「ctrl + /」將多行同時註解。  
(如果要解除註解也可用「ctrl + /」)



### 3-3 由上而下，依序執行

# print 是印出來顯示到螢幕上的語法  
print("Hello") # 註解也可以寫在這  
print("你好")



Python 程式是由上而下，依序的執行

並列的文字在 Python 中稱為字串，而其中的文字可以是英文或中文。

字串的表示方法會用雙引號或單引號「"」、「'」括起來表示。  
如果用「" " "」、「' ' '」括起來可用來表示換行的字串。  
另外如過數字用雙引號或單引號括起來也算是字串。

"Hello"

'Hello'

"""Hello

YH! YH! YH!

'''

1	print(type(123))	查看資料類型可用 type()函數
2	print(type("123"))	

輸出結果

```
<class 'int'>
<class 'str'>
```

備註：如果沒有用 print 語法，那麼就不會印出到螢幕上。  
如果輸入的完後的顏色變成灰色(如下)，先不用太在意。(之後的章節會介紹變數)

*Hello'*

數值有表示的方式，與不同的類型

整數(integer)：-22、0、1、2、3、50。

整數可以是負值、零和正值，也就是沒有小數點的算是正數。

浮點數(floating)：-22.3、-22.0、0.0、2.1。有小數點的為浮點數。

虛數(imaginary)：如果有需要表示複數中的虛數時，在虛數數字尾端加上j。

## 3-5 數值

查看資料類型可用 type()函數

```
1     print(6)
2     print(3.8)
3     print(4+5j)
```

輸出結果

```
6
3.8
(4+5j)
```

```
1     print(type(6))
2     print(type(3.8))
3     print(type(4+5j))
```

輸出結果

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

強制轉換時可要求轉換為其他數值類型，  
如果是整數需轉為浮點數的數學意義上不變，  
但如果浮點數轉為整數小數點後會直接消失，  
這要注意。

```
1     print(int(3.8))  
2     print(float(6))
```

輸出結果

3

6.0

整數中除了 10 進位之外還有 2 進位、8 進位、16 進位。

2 進位(使用數字 0~1 表示) , 數值前面加 0b ;

8 進位(使用數字 0~7 表示) , 數值前面加 0o ;

16 進位(使用數字 0~9、A~F 表示) , 數值前面加 0x 。

```
print("10 進位的 10 是", 10)
print("2 進位的 10 是", 0b10)
print("8 進位的 10 是", 0o10)
print("16 進位的 10 是", 0x10)
```

輸出結果

10 進位的 10 是 10  
2 進位的 10 是 2  
8 進位的 10 是 8  
16 進位的 10 是 16

注意：

如果使用字串需要用「" "」表示，  
如果使用數值則不需要用「" "」表示。

可以使用強制轉換 `int()`、`bin()`、`oct()`和`hex()`，  
分別轉換為十進位、二進位、八進位和十六進位。

```
1     print("十進位的 10 轉為二進位:", bin(10))
2     print("十進位的 10 轉為八進位:", oct(10))
3     print("十進位的 10 轉為十六進位:", hex(10))
4     print("八進位的 0o12 轉為十六進位:", hex(0o12))
5     print("八進位的 0o12 轉為二進位:", bin(0o12))
6     print("八進位的 0o12 轉為十進位:", int(0o12))
```

輸出結果

十進位的 10 轉為二進位: 0b1010

十進位的 10 轉為八進位: 0o12

十進位的 10 轉為十六進位: 0xa

八進位的 0o12 轉為十六進位: 0xa

八進位的 0o12 轉為二進位: 0b1010

八進位的 0o12 轉為十進位: 10

在大數據分析領域下通常獲得的資料會是人看的，  
例如：金融數據、網頁上的資訊、溫度、濕度、  
臭氧濃度…等等。

另外如果是工廠的生產機台的內部數據，  
為了節省儲存空間會以八進位或十六進位表示。

## 3-7 跳脫字元

有些特殊文字無法使用單一符號表示，透過「\」結合後可達到特定目的。

跳脫字元	說明
\n	換行
\t	tab
\b	backspace，倒退一格
\r	return
\'	顯示單引號
\"	顯示雙引號
\\	顯示反斜線

```
1 print("1.\t", "Hello\nPython\n\tgood")
2 print("-----")
3 print("2.\t", "Hello\tPython\tgood")
4 print("-----")
5 print("3.\t", "Hello\rPython\rgood")
6 print("-----")
7 print("4.\t", "Hello\bPython\bgood")
8 print("-----")
9 print("5.\t", "Hello\vPython\vgood")
10 print("-----")
11 print("6.\t", "Hello\"Python\"good")
12 print("-----")
13 print("7.\t", "Hello\\Python\\good")
```

## 輸出結果

1. Hello  
Python  
good  
-----
2. Hello Python good  
-----  
good  
-----
4. HellPythogood  
-----
5. Hello'Python'good  
-----
6. Hello"Python"good  
-----
7. Hello\Python\good

在字串前面加上 r 或 R，那麼字串中的「\」不會被當成跳脫字元。

```
1     print(r"顯示:\")  
2     print(r"顯示:\\")
```

輸出結果

顯示:\'

顯示:\\

## 3-9 print() 的結尾

Python 程式中的 print 最後有預設的「\n」換行。如果不想換行時可加入語法如下

語法：

Print(, end= “●” )，將顯示結尾是●

```
print("Hello", end="")  print("Hello", end=", ")  print("Hello", end="\t")
```

```
print("Ya", end="")    print("Ya", end=", ")    print("Ya", end=".")
```

```
print("Oa", end="")    print("Oa", end=".")     print("Oa", end=".")
```

輸出結果

HelloYaOa

輸出結果

Hello, Ya, Oa.

輸出結果

Hello, Ya, Oa.

~每個範例都打三遍後再做作業~

請撰寫一個程式，輸出結果印在螢幕上，結果如下：

輸出結果

```
coffee  
"一杯 100 元"  
12345  
1 2 3 4 5
```

# Module 4 :

## 變數

4-1 變數的相關知識

4-2 變數命名限制

4-3 指定變數的值

4-4 使用變數與變更變數的值

4-5 變數儲存字串

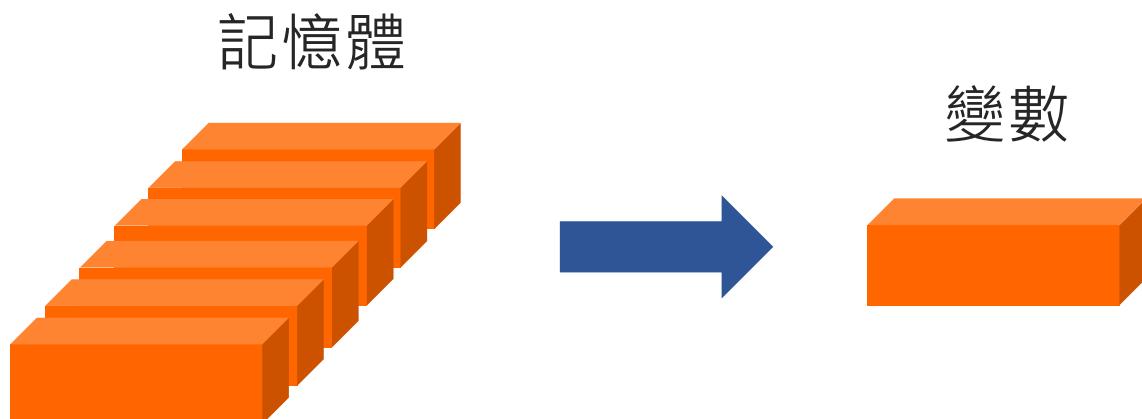
4-6 type() 語法

## 4-1 變數的相關知識

電腦中有記憶體，而記憶體是用來存各種不同的值，以提供運算的需求。

Python 中的「變數」會在記憶體上要一個空間用來記憶資料，並將記憶的資料取用、運算。

示意圖如下，我們可以想像記憶體空間就像盒子，藉由變數找一個空間存放接下來會使用的資料。



變數的命名需要遵守以下規則：

- 開頭不可以是數字。
- 英文大小寫有區分。Sale 和 sale 兩者是不同的變數。
- 不可以使用 Python 程式語言的關鍵字。
- 名稱中不可以包含特殊符號。
- 可以用英文字母、數字、下底線(\_)來命名。

命名的習慣最好是讓人易於了解使用變數的目的，以方便閱讀程式。

Python 的關鍵字如右：

FALSE	class	finally	is	return
None	continue	for	lambda	try
TRUE	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

## 4-3 指定變數的值

取好變數名稱後，就可以把資料或是運算式(表示式)指定給變數，讓變數去記憶資料。  
**指定運算子「=」**

語法：

變數名稱 = 表示式

```
1 coffee_price = 100  
2 print("咖啡價格:", coffee_price, "元/杯")
```

輸出結果

咖啡價格： 100 元/杯

備註：

學過數學的人都知道，等號(=)的左邊和右邊是一樣的，

但在 Python 請不要這樣理解，這樣理解會讓你發生錯亂。

在 Python 中請稱他為「**指定運算子(=)**」，如上例：把 23 這個數字指定給 coffee\_price 這個變數。



## 4-4 使用變數與變更變數的值

可以用新的值指定給變數，讓變數變更為不同的值。  
而變數可以變更不同的資料，所以稱為「變」數。

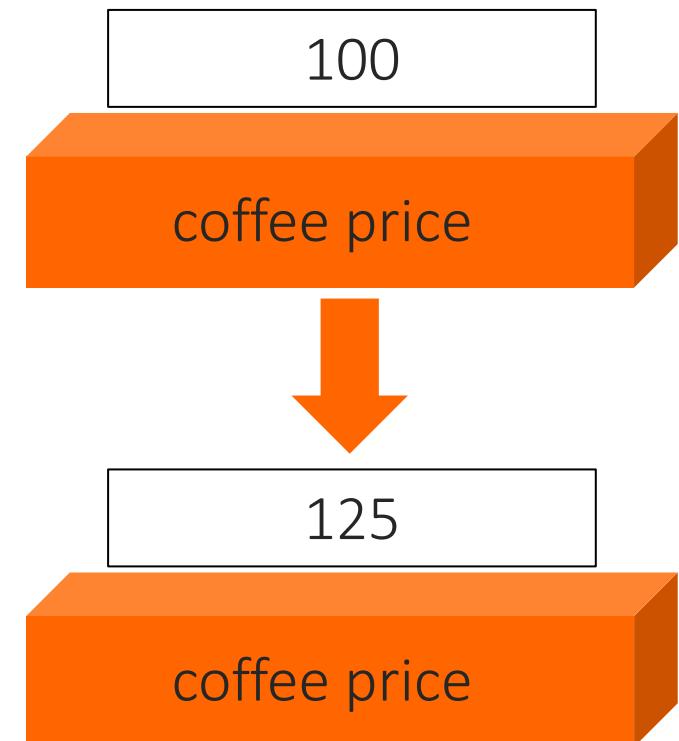
```
1 coffee_price = 100
2 print("咖啡價格: ", coffee_price, "元/杯")
3 print("變更價格")
4 coffee_price = 125
5 print("咖啡價格: ", coffee_price, "元/杯")
```

輸出結果

咖啡價格: 100 元/杯

變更價格

咖啡價格: 125 元/杯



變數可以存放數值，當然也包含字串。

```
1 product_name = "coffee"  
2 product_price = 100  
3 print(product_name, "價格:", product_price, "元/杯")
```

輸出結果

coffee 價格: 100 元/杯

變數可以被許多種不同類型的資料所指定，當程式龐大而你搞混變數中的資料型態時，除了印出資料本身以外，也可以用 type() 語法查看變數的類型。

語法：

type(變數名稱)

```
1 product_name = "coffee"
2 product_price = 100
3 # print(product_name, "價格: ", product_price, "元/杯")
4
5 print("product_name 的資料型態為: ", type(product_name))
6 print("product_price 的資料型態為: ", type(product_price))
```

輸出結果 product\_name 的資料型態為: <class 'str'>
product\_price 的資料型態為: <class 'int'>

# Module 5 :

## 運算式的基礎知識

5-1 運算式的相關知識

5-2 輸出運算式的值

5-3 運算子的類型

5-3-1 算數運算子

5-3-2 比較運算子

5-3-3 邏輯運算子

5-3-4 身份運算子

5-3-5 位元運算子

5-3-6 行列運算子

5-4 指定運算子

5-5 運算子的優先順序

5-6 操作字串的運算子

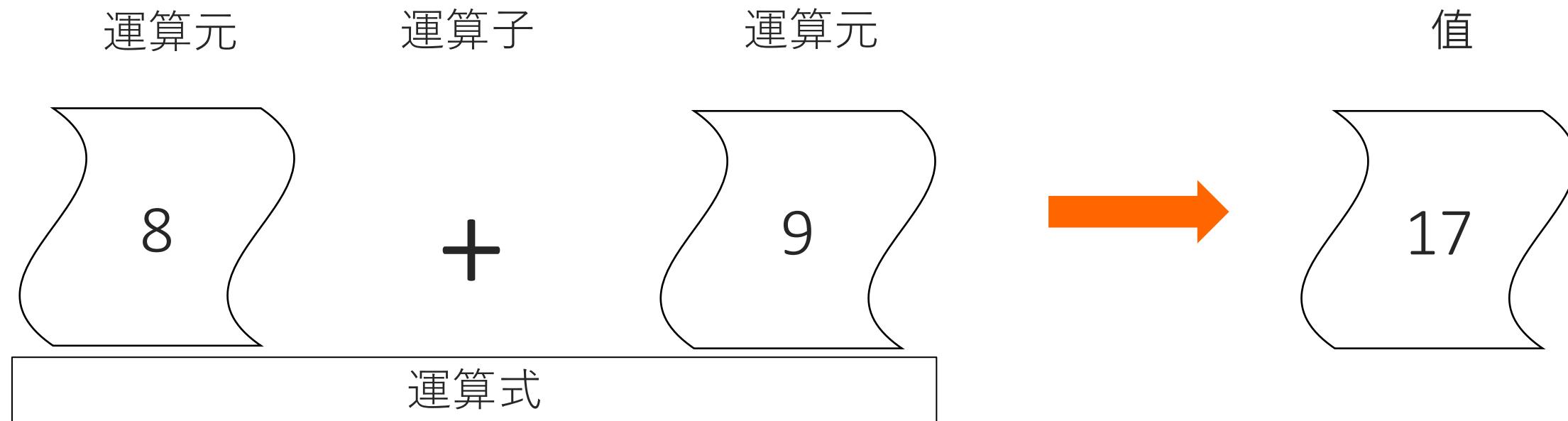
## 5-1 運算式的相關知識

程式中常常會做「計算」，那麼接著我們就來了解關於計算的相關語法。

首先了解運算式、運算子和運算元之間的關係：

運算式(expression)可以理解成「 $8 + 9$ 」這樣常見的算式，

其中的「8」和「9」稱為運算元(operand)，其中的「+」稱為運算子(operator)。



```
1     print("8 + 9 等於:", 8 + 9)
```

輸出結果

8 + 9 等於: 17

---

```
1     coffee_price = 100
2     sale_number = 1024
3     sales_figures = coffee_price * sale_number
4     print("咖啡價格: ", coffee_price, "元/杯")
5     print("銷售數量: ", sale_number, "杯")
6     print("銷售額: ", sales_figures, "元")
7
8     sales_figures = sales_figures * 0.9
9     print("打折後銷售額: ", sales_figures, "元")
```

輸出結果

咖啡價格: 100 元/杯

銷售數量: 1024 杯

銷售額: 102400 元

打折後銷售額: 92160.0 元

## 5-3 運算子的類型

符號	名稱	符號	名稱
+	加法運算子	=	指定運算子
-	減法運算子	>	大於關係運算子
*	乘法運算子	>=	大於等於關係運算子
/	除法運算子	<	小於關係運算子
//	商數運算子	<=	小於等於關係運算子
%	餘數運算子	==	相等運算子
**	次方運算子	!=	不相等運算子
@	行列運算子	and	邏輯AND運算子
+	正號(一元運算子)	or	邏輯OR運算子
-	負號(一元運算子)	not	邏輯NOT運算子
~	位元補數運算子	if else	條件運算子
	位元OR運算子	in	in 成員運算子
&	位元AND運算子	not in	not in 成員運算子
^	位元XOR運算子	is	is 身份運算子
<<	位元左移運算子	is not	is not 身份運算子
>>	位元右移運算子	lambda	lambda 運算子

## 5-3-1 算數運算子

1	num_1 = 15	輸出結果
2	num_2 = 2	num_1 + num_2 等於: 17
3		num_1 - num_2 等於: 13
4	print("num_1 + num_2 等於:", num_1 + num_2)	num_1 * num_2 等於: 30
5	print("num_1 - num_2 等於:", num_1 - num_2)	num_1 / num_2 等於: 7.5
6	print("num_1 * num_2 等於:", num_1 * num_2)	num_1 // num_2 等於: 7
7	print("num_1 / num_2 等於:", num_1 / num_2)	num_1 % num_2 等於: 1
8	print("num_1 // num_2 等於:", num_1 // num_2)	num_1 ** num_2 等於: 225
9	print("num_1 % num_2 等於:", num_1 % num_2)	
10	print("num_1 ** num_2 等於:", num_1 ** num_2)	

何謂商數運算子和餘數運算子：

「 $\text{num\_1} \div \text{num\_2} = \bullet \dots \dots \text{餘 } \blacktriangle$ 」，

商數運算子求得的值是 $\bullet$ ，餘數運算子求得的值是 $\blacktriangle$ ，「 $15 \div 2 = 7 \dots \dots \text{餘 } 1$ 」。

## 5-3-2 比較運算子

1	num_1 = 15	輸出結果
2	num_2 = 2	num_1 == num_2 等於: False
3		num_1 != num_2 等於: True
4	print("num_1 == num_2 等於:", num_1 == num_2)	num_1 > num_2 等於: True
5	print("num_1 != num_2 等於:", num_1 != num_2)	num_1 >= num_2 等於: True
6	print("num_1 > num_2 等於:", num_1 > num_2)	num_1 < num_2 等於: False
7	print("num_1 >= num_2 等於:", num_1 >= num_2)	num_1 <= num_2 等於: False
8	print("num_1 < num_2 等於:", num_1 < num_2)	
9	print("num_1 <= num_2 等於:", num_1 <= num_2)	

## 5-3-3 邏輯運算子

邏輯運算子是用來執行「布林值(Boolean)」True、False 的判斷。

運算子	表達式	說明
and	a and b	a、b 兩者都為 True，結果才為 True
or	a or b	a、b 其中只要一個為 True，結果就為 True
not	not b	反向，b 如果是 False，結果成為 True

```

1     T = True
2     F = False
3
4     print("T and F: ", T and F)    T and F: False
5     print("T or F: ", T or F)      T or F: True
6     print("not T: ", not T)        not T: False
7     print("not F: ", not F)        not F: True

```

輸出結果

True 和 False 是用來表示真、假的值，稱之為「布林值(Boolean)」。在 Python 之中，除了真假以外的值，也可以當做布林值來處理。

	True	False
數值	0 以外的值	0
字串	空值以外的值	空值(None)

`is` 、`not is` 身分運算子是判斷左邊和右邊是否相同，而他判斷的方式是以記憶體位置來判定。一般來說要判定 `True` 、`False` 和 `None`，會使用「`is` 、`not is` 運算子」。

```
1  a = 1000
2  b = 1000
3  c = True
4  d = False
5
6  print("1.\t", id(a))
7  print("2.\t", id(b))
8  print("3.\t", a is b)
9  print("4.\t", a is not b)
10 print("5.\t", c is True)
11 print("6.\t", d is not False)
```

輸出結果

1. 2561204670192
2. 2561204670192
3. True
4. False
5. True
6. False

備註：`id()`

是可以讓你知道變數在記憶體上所在的「記憶體位址」，本身的數值沒有運算意義。  
不同電腦、時機所顯示的不見的相同。

## 5-3-5 位元運算子

位元運算子是用二進位進行的位元操作。(較接近電腦，人類較不易解讀)。

運算程序是先將運算元轉換成二進位，接著再做二進位布林運算。

位元運算子有「&」(AND)、「|」(OR) 和「^」(XOR)，

「&」(AND，且)，運算時，兩者皆為 1 結果才會是 1。

「|」(OR，或)，運算時，只要一個是 1 結果就會是 1。

「^」(XOR，互斥)，運算時，兩者不同結果就會是 1，若相同結果就會是 0。

A	B	A&B	A B	A^B
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

## 5-3-5 位元運算子

A = 1

B = 1

print("A & B:", A & B)

print("A | B:", A | B)

print("A ^ B:", A ^ B)

輸出結果

A & B : 1

A | B : 1

A ^ B : 0

A = 1

B = 0

print("A & B:", A & B)

print("A | B:", A | B)

print("A ^ B:", A ^ B)

輸出結果

A & B : 0

A | B : 1

A ^ B : 1

A = 0

B = 1

print("A & B:", A & B)

print("A | B:", A | B)

print("A ^ B:", A ^ B)

輸出結果

A & B : 0

A | B : 1

A ^ B : 1

## 5-3-5 位元運算子

「~」位元補位運算子，是用二進位進行的位元操作。

將運算元進行「二補數」運算，接著運用「~」位元補位運算子，將 1 變成 0，0 變成 1。

```
A = -7
```

```
print(~A:, ~A)
```

輸出結果

```
~A : 6
```

```
A = 7
```

```
print(~A:, ~A)
```

輸出結果

```
~A : -8
```

二補數	十進位
0111	7
0110	6
...	...
0010	2
0001	1
0000	0
1111	-1
1110	-2
...	...
1001	-7
1000	-8

十進位的 -7 轉換為二進位的二補數為 1001，接著經過「~」位元補位運算子後成為 0110。

二補數是一種用二進位表示有符號數的方法，也是一種將數字的正負號變號的方式，常在電腦科學中使用。

二補數以有符號位元的二進位數定義。

## 5-3-5 位元運算子

「`>>`」位元左移運算子，將左側「運算元」依右側設定的數量向右移  
(「最高有效位」正值補 0、負值補 1，最低有效位丟棄)

「`>>`」位元右移運算子，將左側「運算元」依右側設定的數量向左移  
(「最低有效位」補 0)

`A = 3`                                 輸出結果

`print("A >> 1:", A >> 1)`    `A >> 1 : 1`

`print("A << 1:", A << 1)`    `A << 1 : 6`

十進位的 3 轉為二補數的二進位為 0011 往右移 1 個位元後成為 0001，也就成為十進位的 1。  
十進位的 3 轉為二補數的二進位為 0011 往左移 1 個位元後成為 0110，也就成為十進位的 6。

行列運算子@ 可以用來做矩陣的相乘

```
import numpy as np
```

```
z = np.array([[1, 2, 3], [4, 5, 6]])
```

```
w = np.array([[1, 2], [3, 4], [5, 6]])
```

```
print(z @ w)
```

匯入 numpy 套件，  
匯入之後用 as 幫它取一個別名，  
別名這裡取 np。

另外，numpy 使用於多維串列(list) 和矩陣運算。

輸出結果

```
[[22 28]
 [49 64]]
```

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix} \times \begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{matrix} = \begin{matrix} 22 & 28 \\ 49 & 64 \end{matrix}$$

## 5-4 指定運算子

指定運算子的功能是將邊的資料指定給左邊的變數，另外指定運算子並不只有先前介紹的「=」，還有 = 與其它運算子的組合。

符號	說明
<code>+=</code>	進行加法運算後，指定給變數
<code>-=</code>	進行減法運算後，指定給變數
<code>*=</code>	進行乘法運算後，指定給變數
<code>/=</code>	進行除法運算後，指定給變數
<code>//=</code>	進行求商數運算後，指定給變數
<code>%=</code>	進行求餘數運算後，指定給變數
<code>**=</code>	進行次方運算後，指定給變數
<code>@=</code>	進行行列運算後，指定給變數
<code>&amp;=</code>	進行邏輯AND運算後，指定給變數
<code> =</code>	進行邏輯OR運算後，指定給變數
<code>^=</code>	進行邏輯XOR運算後，指定給變數
<code>&lt;&lt;=</code>	位元左移後，指定給變數
<code>&gt;&gt;=</code>	位元右移後，指定給變數

`A = A ■ B`，可改寫成 `A ■= B`。

```
1     total = 2
2
3     num_1 = 3
4
5     total += num_1
print("total 等於:", total)
```

輸出結果      total 等於: 5

注意：`+=` 之間不可以有空白

平常做四則運算時，會先計算括號內的算式，接著先乘除後加減，如果遇到相同優先順序的運算子則由左至右開始運算，在 Python 中也是用這樣的方式運作。

但要注意的是「=」指定運算子是把「=」右邊的算完再指定給「=」左邊的變數，所以等號右邊的要先算。

```
1     a = 3
2     # 先乘除後加減
3     # 等號右邊先算完, 在指定給等號左邊
4     b = a + 5 * 8
5     print("result: ", b)
```

輸出結果

result : 43

```
1     a = 3
2     # 若有括號, 則先算括號的內容
3     b = (a + 5) * 8
4     print("result: ", b)
```

輸出結果

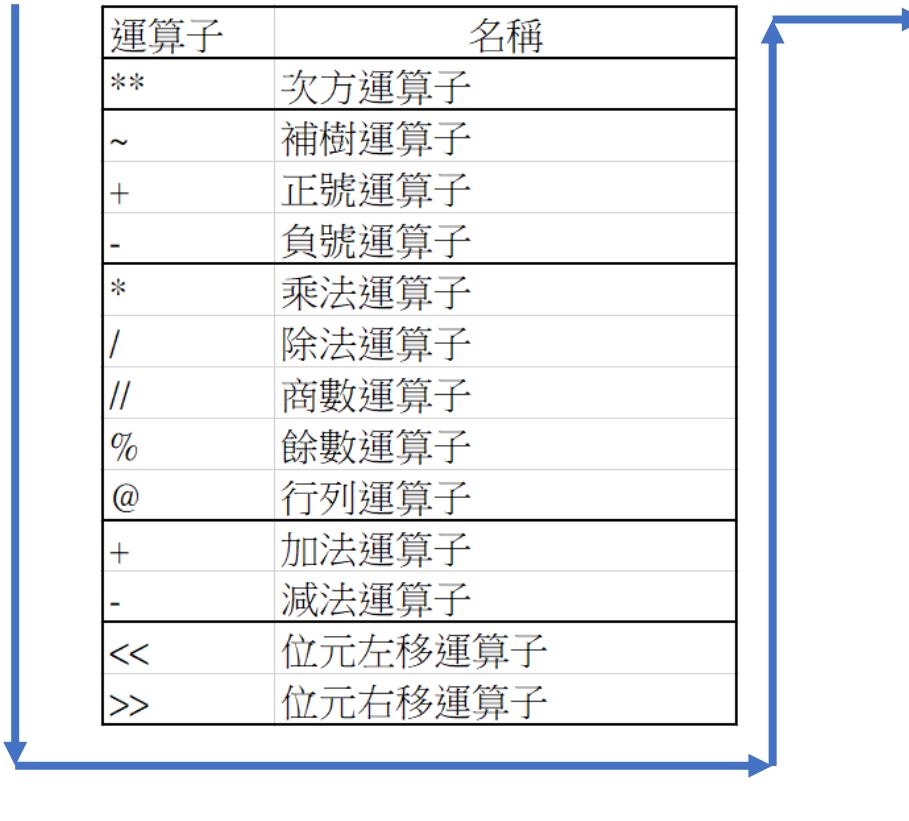
result: 64

## 5-5 運算子的優先順序

另外除了四則運算以外，Python 還有提供許多的運算子，  
接下來提供運算子的優先順序列表。(無粗框線表示優先順續相同)

運算子	名稱
**	次方運算子
~	補樹運算子
+	正號運算子
-	負號運算子
*	乘法運算子
/	除法運算子
//	商數運算子
%	餘數運算子
@	行列運算子
+	加法運算子
-	減法運算子
<<	位元左移運算子
>>	位元右移運算子

運算子	名稱
&	位元 AND 運算子
^	位元 XOR 運算子
	位元 OR 運算子
>	大於運算子
>=	大於等於運算子
<	小於運算子
<=	小於等於運算子
==	等於運算子
!=	不等於運算子
in、not in	成員內、非成員內運算子
is、not is	相等、不相等運算子
not	邏輯 NOT 運算子
and	邏輯 AND 運算子
or	邏輯 OR 運算子
if else	條件判斷運算子
lambda	lambda



## 5-6 操作字串的運算子

並非所有運算子都能操作字串，只有「+」(加號)、「\*」(乘號)。

用「+」加法運算子操作字串與字串連接，

用「\*」乘法運算子操作字串的重複次數。

注意：連接時要同一類型的資料連接。

1	place = "Taipei"	輸出結果
2	weather = "晴時多雲偶陣雨"	
3	print(place + weather)	Taipei晴時多雲偶陣雨
4		紫外線指數: *****
5	start = "*"	
6	print("紫外線指數:", start * 10)	Taipei102
7		
8	number = 102	
9	print(place + str(number))	

學會看 error code :

程式是由上往下依序執行，所以它會告訴你  
第幾行、哪一行 code 以及甚麼原因導致 error code 。

```
1     place = "Taipei"  
2     number = 102  
3     print(place + number)
```

輸出結果

Traceback (most recent call last):

```
  File "D:/Python/workspace/Sample_5_9.py", line 3, in <module>  
    print(place + number)
```

TypeError: must be str, not int

# Module 6 :

## 輸入-由鍵盤輸入

6-1 由鍵盤輸入字串

6-2 由鍵盤輸入數值

6-3 輸入數值須注意事項

6-4 由鍵盤輸入數值後運算

由鍵盤輸入是使用 `input()`，所輸入的值皆會被視為字串。

語法：

變數名稱 = `input("螢幕上顯示的資訊")`

```
1     str_a = input("Please enter a word:")
2     print("What you entered is:", str_a)
```

輸出結果

Please enter a word:*Hello*

What you entered is: Hello

Please enter a word:*你好!*

What you entered is: 你好!

ㄉ一ˇ 按 enter 才會顯示

## 6-2 由鍵盤輸入數值

由於 `input()` 所輸入的值皆為字串，所以如果要輸入數值時需要做「強制轉換」。

```
1  data = input("Please enter a number:")
2  print("轉換前資料型態", type(data))
3  data = int(data) # 強制轉換
4  print("轉換後資料型態", type(data))
5  print("What number you entered is:", data)
```

輸出結果

```
Please enter a number:33
轉換前資料型態 <class 'str'>
轉換後資料型態 <class 'int'>
What number you entered is: 33
```

```
1  data = int(input("Please enter a number:"))
2  print("What number you entered is:", data)
```

輸出結果

```
Please enter a number:45
What number you entered is: 45
```

人類看到的數字，在 python 中可以用字串表示也可以用數值表示，所以在 Python 中的數字可以是字串、整數或浮點數。但如果需要數值的運算時一定要是整數或浮點數。而文字只能是字串，文字轉為整數、浮點數沒有意義。所以強制轉換可以讓整數轉為字串、浮點數。

1	data = 1234	輸出結果
2	print("原本的資料型態:", type(data))	原本的資料型態: <class 'int'>
3	print("原本的 data", data)	原本的 data 1234
4	print("-" * 20)	-----
5	data = str(data)	轉換後的資料型態: <class 'str'>
6	print("轉換後的資料型態:", type(data))	轉換後的 data 1234
7	print("轉換後的 data", data)	

## 6-3 輸入數值須注意事項

整數轉為浮點數，多了小數點，而值沒變

```
1  data = 1234
2  print("原本的資料型態:", type(data))
3  print("原本的 data", data)
4  print("-" * 20)
5  data = float(data)
6  print("轉換後的資料型態:", type(data))
7  print("轉換後的 data", data)
```

輸出結果

原本的資料型態: <class 'int'>

原本的 data 1234

-----

轉換後的資料型態: <class 'float'>

轉換後的 data 1234.0

## 6-3 輸入數值須注意事項

浮點數轉為字串，資料看起來沒變，但資料型態改變。

```
1     data = 12.34
2     print("原本的資料型態:", type(data))
3     print("原本的 data", data)
4     print("-" * 20)
5     data = str(data)
6     print("轉換後的資料型態:", type(data))
7     print("轉換後的 data", data)
```

輸出結果

原本的資料型態: <class 'float'>

原本的 data 12.34

-----

轉換後的資料型態: <class 'str'>

轉換後的 data 12.34

## 6-3 輸入數值須注意事項

浮點數轉為整數，這裡要特別注意：資料小數點後直接捨去，資料型態改變。

```
1 data = 12.34
2 print("原本的資料型態:", type(data))
3 print("原本的 data", data)
4 print("-" * 20)
5 data = int(data)
6 print("轉換後的資料型態:", type(data))
7 print("轉換後的 data", data)
```

### 輸出結果

原本的資料型態: <class 'float'>

原本的 data 12.34

-----

轉換後的資料型態: <class 'int'>

轉換後的 data 12

## 6-3 輸入數值須注意事項

人類看是整數，電腦看是字串，轉換為整數。

人類看是浮點數，電腦看是字串，轉換為浮點數。

```
1     data = "1234"  
2     print("原本的資料型態:", type(data))  
3     print("原本的 data", data)  
4     print("-" * 20)  
5     data = int(data)  
6     print("轉換後的資料型態:", type(data))  
7     print("轉換後的 data", data)
```

輸出結果

原本的資料型態: <class 'str'>

原本的 data 1234

-----

轉換後的資料型態: <class 'int'>

轉換後的 data 1234

```
1     data = "1234"  
2     print("原本的資料型態:", type(data))  
3     print("原本的 data", data)  
4     print("-" * 20)  
5     data = float(data)  
6     print("轉換後的資料型態:", type(data))  
7     print("轉換後的 data", data)
```

輸出結果

原本的資料型態: <class 'str'>

原本的 data 1234

-----

轉換後的資料型態: <class 'float'>

轉換後的 data 1234.0

從鍵盤輸入數值後可以用來計算，範例如下：

```
1 # Data input from keyboard
2 product_name = input("Please enter product name: ")
3 product_price = int(input("Please enter product price: "))
4 sale_number = int(input("Please enter count: "))
5
6 # Calculate
7 total_price = product_price * sale_number
8
9 # print to screen
10 print(product_name, "\t價格:", product_price, "元/單位")
11 print("銷售數量:", sale_number, "單位")
12 print("總金額:", total_price, "元")
```

輸出結果

Please enter product name: coffee  
Please enter product price: 100  
Please enter count: 400  
coffee 價格: 100 元/單位  
銷售數量: 400 單位  
總金額: 40000 元

這裡提供一個錯誤的範例，未來請小心，範例如下：

```
1 # Data input from keyboard
2 num_1 = input("Please enter first number:")
3 num_2 = input("Please enter second number ")
4
5 # Calculate
6 total_number = num_1 + num_2
7
8 # print to screen
9 print("總數為:", total_number)
```

輸出結果

Please enter first number: 100  
Please enter second number 5  
總數為: 1005

備註：  
字串 + 字串，會變成字串了連接。

~每個範例都打三遍後再做作業~

請撰寫一個程式，需用鍵盤輸入身高與體重，  
輸入完之後程式會計算 BMI，最後印在螢幕上，

BMI值計算公式:  $BMI = \frac{\text{體重(公斤)}}{\text{身高}^2(\text{公尺}^2)}$

輸出結果

請輸入身高(cm): 174

請輸入體重(kg): 60

您的身高為: 174.0 (cm) 體重為: 60.0 (kg)

BMI 值為: 19.817677368212443

# Module 7 :

## 條件式與流程控制

7-1 條件式與流程控制的概念

7-2 if 敘述

7-2-1 條件式的敘述

7-2-2 pass 的用法

7-3 if...else

7-3-1 條件運算子 if else

7-4 if...elif...else

7-5 流程控制的巢狀結構

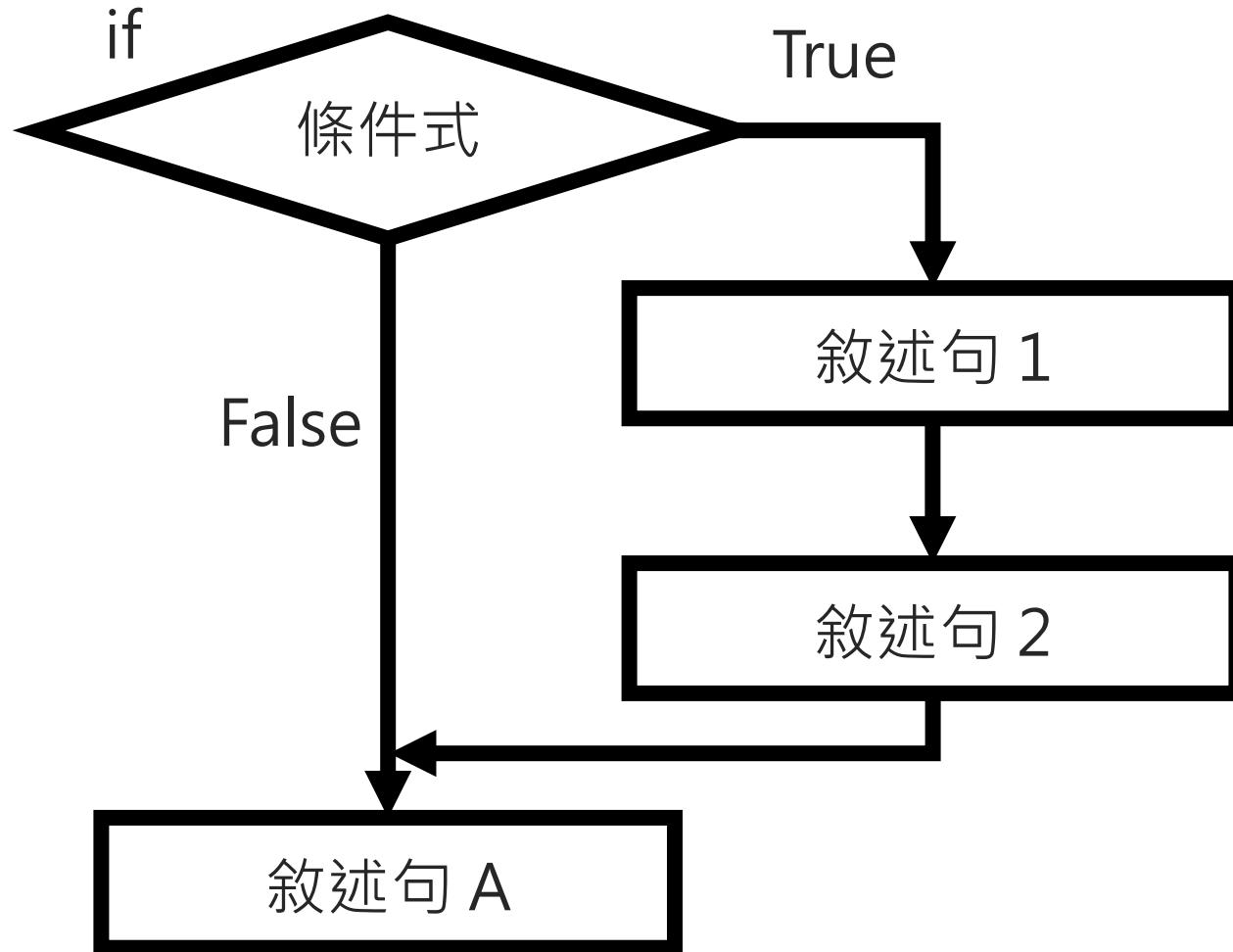
7-6 使用邏輯運算子敘述條件式

我們生活中會遇到許多根據不同的情況，做出相對應處理，  
例如：

1. 如果(if) 行車時速超過60的(條件式)成立的情況下(:) 則執行開超速罰單(這個動作)。
2. 如果(if) 成績大於等於60分的(條件式)成立的情況下(:) 則顯示及格，  
若以上(條件式)不成立則接著做另外(else)的處理，  
成績小於60分的(條件式)成立的情況下(:) 則顯示 fail。
3. 在ATM操作下，如果(if) 按「1」的(條件式)成立的情況下(:) 則執行提領 1000，  
另外如果else if (elif)，按「2」的(條件式)成立的情況下(:) 則執行提領 3000，  
另外如果else if (elif)，按「3」的(條件式)成立的情況下(:) 則執行提領 5000，  
(依此類推)  
另外(else) (:), 則執行取消。

在 Python 中的流程控制：

1. if 敘述。
2. if ... else 敘述。
3. If ... elif ... else 敘述。(其中的 elif 可以多個)



語法：

if 條件式：

敘述句 1

敘述句 2

敘述句 A

注意：

if 條件式和敘述句不是對齊的，  
敘述句 1、敘述句 2 前有「縮排」。

備註：

到目前為止，你應該會有幾個疑問，

1. 條件式？
2. True、False？
3. 敘述句 1、敘述句 2 和敘述句 A 的差別？
4. 縮排？

```
1     speed = 88
2
3     if speed >= 60:
4         print("你超速了")
5         print("您的車速為: ", speed, "Km/hr")
6
7     print("program end")
```

輸出結果

你超速了  
您的車速為: 88 Km/hr  
program end

```
1     speed = 40
2
3     if speed >= 60:
4         print("你超速了")
5         print("您的車速為: ", speed, "Km/hr")
6
7     print("program end")
```

輸出結果  
program end

左邊的範例是 if 條件式判斷為 True，所以會執行縮排內的程式碼。

左邊的範例是 if 條件式判斷為 False，所以不會執行縮排內的程式碼。

程式中的教學測試句不在 if 的縮排內，所以不論 if 條件式判斷為何，皆會執行。

你可以想像「縮排」是要讓程式知道哪一段程式是在誰的管轄範圍內。

另外，「縮排」一般是會用 4 個空白字元，雖然不用 4 個空白字元程式也會執行。

## 7-2-1 條件式的敘述

運算子	條件式為 True 的情況
>	左邊的值大於右邊的值
>=	左邊的值大於或等於右邊的值
<	左邊的值小於右邊的值
<=	左邊的值小於或等於右邊的值
==	左邊的值等於右邊的值
!=	左邊的值不等於右邊的值
in	左邊存在於右邊的值
not in	左邊不存在於右邊的值
is	左邊的物件等於右邊的物件
not is	左邊的物件不等於右邊的物件

```
a = 6
print("條件式 a > 3 的結果為：", a > 3)
print("條件式 a < 3 的結果為：", a < 3)
print("條件式 a != 3 的結果為：", a != 3)
print("條件式 a == 3 的結果為：", a == 3)
print("條件式 a >= 3 的結果為：", a >= 3)
print("條件式 a <= 3 的結果為：", a <= 3)
```

輸出結果 條件式  $a > 3$  的結果為 : True  
 條件式  $a < 3$  的結果為 : False  
 條件式  $a \neq 3$  的結果為 : True  
 條件式  $a == 3$  的結果為 : False  
 條件式  $a \geq 3$  的結果為 : True  
 條件式  $a \leq 3$  的結果為 : False

## 7-2-2 pass 的用法

如果你寫 if 條件，但還執行內容是空的，那麼至少要在縮排後寫 pass。

```
1     speed = 88
2
3     if speed >= 60:
4         pass # 還沒想好要做哪些事
5
6     print("program end")
```

輸出結果      program end

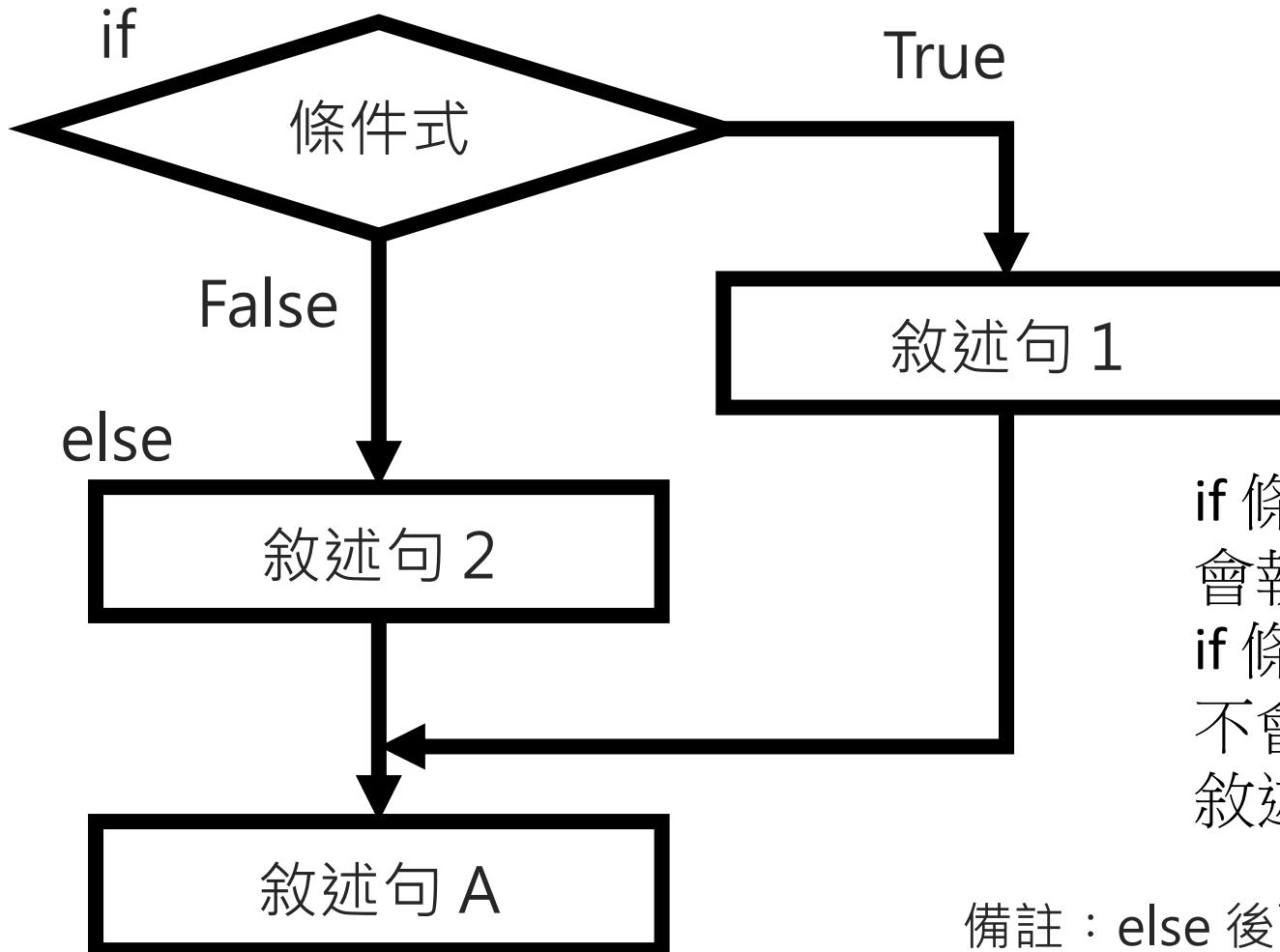
還沒確切想好條件式成立時要做哪些動作，  
可以先寫好流程控制的架構，  
然後再慢慢寫內容，內容寫好後再把 pass 拿掉。

```
1     speed = 88
2
3     if speed >= 60:
4
5
6     print("program end")
```

輸出結果

```
File "D:\Python\workspace\Sample_7\Sample_7_2_1.py", line 6
    print("program end")
    ^
IndentationError: expected an indented block
```

## 7-3 if...else



語法：  
if 條件式：  
    敘述句 1  
else:  
    敘述句 2  
    敘述句 A

if 條件式是 True 的情況下，  
會執行敘述句 1，但敘述句 2 不會執行。  
if 條件式是 False 的情況下，  
不會執行敘述句 1，敘述句 2 會執行。  
敘述句 A 都會執行。

備註：else 後面不需加條件式。

## 7-3 if...else

```
1 score = float(input("Please enter your score:"))
2
3 if score >= 60:
4     print("Pass")
5     print("Your score is :", score)
6 else:
7     print("Fail")
8     print("Your score is :", score)
9 print("program end")
```

輸出結果

Please enter your score:*88*

Pass

Your score is : 88.0

program end

Please enter your score:*55*

Fail

Your score is : 55.0

program end

語法：

True 情況下的結果 if 條件式 else False 情況下的結果

```
1 score = float(input("Please enter your score:"))

2

3 result = "Pass" if score >= 60 else "Fail"
4 print(result)
5 print("Your score is :", score)
6 print("program end")
```

輸出結果

Please enter your score:*88*

Pass

Your score is : 88.0

program end

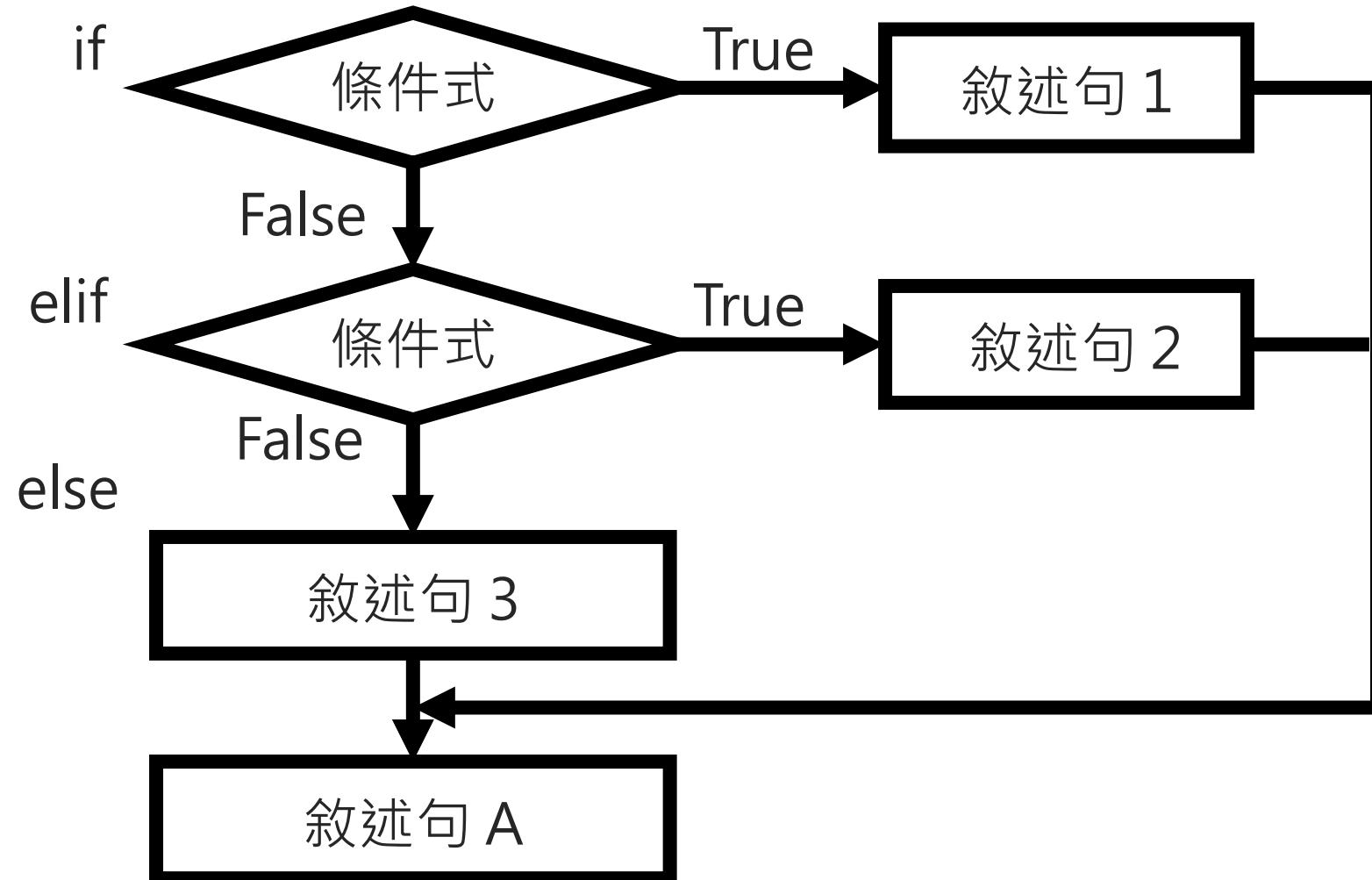
## 7-3-1 條件運算子 if else

```
1 score_f = float(input("Please enter score:"))
2 score = int(score_f) if score_f == int(score_f) else score_f
3 result = "Pass" if score >= 60 else "Fail"
4 print(result)
5 print("Your score is :", score)
6 print("program end")
```

## 輸出結果

Please enter score:*60.5*  
Pass  
Your score is : 60.5  
program end

Please enter score:*60*  
Pass  
Your score is : 60  
program end



語法：

if 條件式 :

敘述句 1

elif 條件式:

敘述句 2

else:

敘述句 3

敘述句 A

備註：

elif 可以多個。

## 7-4 if...elif...else

```
1 operator_number = input("Please enter operator number: (1~3):")  
2  
3 if operator_number == "1":  
4     # withdraw 1000 process  
5     print("提領 1000 元")  
6 elif operator_number == "2":  
7     # withdraw 3000 process  
8     print("提領 3000 元")  
9 elif operator_number == "3":  
10    # withdraw 5000 process  
11    print("提領 5000 元")  
12 else:  
13     print("Please enter (1~3), no other number")  
14     print("Process end")
```

輸出結果

Please enter operator number: (1~3):1  
提領 1000 元  
Process end

Please enter operator number: (1~3):3  
提領 5000 元  
Process end

Please enter operator number: (1~3):4  
Please enter (1~3), no other number  
Process end

## 7-4 if...elif...else

```
1 score = float(input("Please enter score:")) 輸出結果
2
3 if score >= 60:
4     print("成績合格")
5 elif score >= 60:
6     print("成績不合格")
7 elif score == 0.0:
8     print("0.0")
9 else:
10    print("要再更多!更多!練習!")
11 print("Process end")
```

Please enter score:*65*  
成績合格  
Process end

Please enter score:*0*  
0.0  
Process end

## 7-5 流程控制的巢狀結構

流程控制的巢狀結構，簡單來說就是：流程控制內部還有流程控制。

```
1 score_chi = float(input("Please enter chinese score:"))
2 score_eng = float(input("Please enter english score:"))
3 score_chi = int(score_chi) if score_chi == int(score_chi) else score_chi
4 score_eng = int(score_eng) if score_eng == int(score_eng) else score_eng
5 print("chinese score:", score_chi, "english score:", score_eng)
6 if score_chi >= 60:
7     if score_eng >= 60:
8         print("中文和英文都合格")
9     else:
10        print("中文合格, 英文不合格")
11 else:
12     if score_eng >= 60:
13         print("英文合格, 中文請加油")
14     else:
15         print("都不合格")
```

輸出結果

Please enter chinese score:**70**  
Please enter english score:**68.5**  
chinese score: 70 english score: 68.5  
中文和英文都合格

Please enter chinese score:**59.9**  
Please enter english score:**59.9**  
chinese score: 59.9 english score: 59.9  
都不合格

有些時候我們遇到的條件式未必只有一條，像是：

1. 條件式 A 和條件式 B 要同時成立(True)的情況下，才算是真的成立(True)。  
「邏輯 AND 運算子」
2. 條件式 A 或條件式 B 只要其中一項成立(True)，就算是成立(True)。  
「邏輯 OR 運算子」
3. 如果條件式 A 是成立(True)，但我反而不要讓他成立(True)。  
「邏輯 NOT 運算子」

~使用邏輯運算子搭配敘述條件式可以取代流程控制的巢狀結構~

# 7-6 使用邏輯運算子敘述條件式

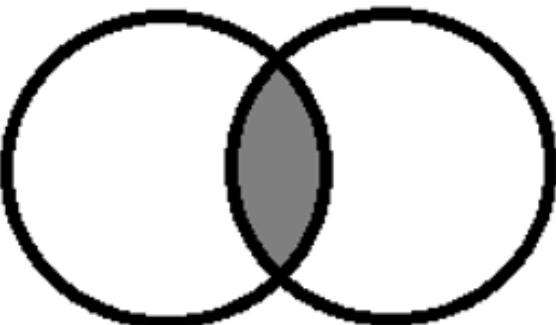
條件式 A	邏輯運算子	條件式 B	總體結果
True	and	True	True
True		False	False
False		True	False
False		False	False

條件式 A	邏輯運算子	條件式 B	總體結果
True	or	True	True
True		False	True
False		True	True
False		False	False

邏輯運算子	條件式	總體結果
not	True	False
	False	True

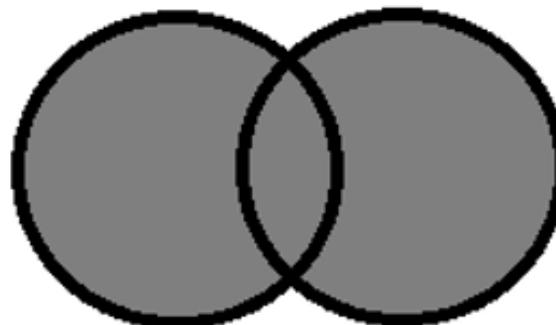
同時滿足

條件式 A 和條件式 B 的部分



只要滿足

條件式 A 或條件式 B 的部分



原本滿足條件式，

但我其實需要不滿足的地方。



## 7-6 使用邏輯運算子敘述條件式

```
1 score_chi = float(input("Please enter chinese score:"))
2 score_eng = float(input("Please enter english score:"))
3 print("chinese score:", score_chi, "english score:", score_eng)
4
5 if score_chi >= 60 and score_eng >= 60:
6     print("中文和英文都合格")
7 elif (score_chi >= 60) and (score_eng < 60):
8     print("中文合格, 英文不合格")
9 elif (score_chi < 60) and (score_eng >= 60):
10    print("中文不合格, 英文合格")
11 elif not (score_chi >= 60 and score_eng >= 60):
12    print("都不合格!")
13 else:
14    print("檢查為符合條件原因")
```

輸出結果

Please enter chinese score: 100  
Please enter english score: 100  
chinese score: 100.0 english score: 100.0  
中文和英文都合格

## 作業實作

~每個範例都打三遍後再做作業~

還記得上一個作業是關於計算 BMI 吧!接續上一個作業，  
請撰寫一個程式把 BMI 算完後將得到的結果加入流程控制。  
條件式參考如右：

體重過瘦  $BMI < 18.5$   
體重標準  $18.5 \leq BMI < 24$   
體重過重  $24 \leq BMI < 27$   
輕度肥胖  $27 \leq BMI < 30$   
中度肥胖  $30 \leq BMI < 35$   
重度肥胖  $BMI \geq 35$

### 輸出結果

請輸入身高(cm): 174

請輸入體重(kg): 60

您的身高為: 174.0 (cm) 體重為: 60.0 (kg)

BMI 值為: 19.817677368212443

\*\*\*\*\*體重標準\*\*\*\*\*

請輸入身高(cm): 174

請輸入體重(kg): 100

您的身高為: 174.0 (cm) 體重為: 100.0 (kg)

BMI 值為: 33.029462280354075

\*\*\*\*\*中度肥胖\*\*\*\*\*

# Module 8 :

## 迴圈

8-1 for 迴圈

8-1-1 for 迴圈與流程控制

8-2 while 迴圈

8-3 for 、while 差異

8-4 迴圈的巢狀結構

8-5 迴圈中的變更處理流程

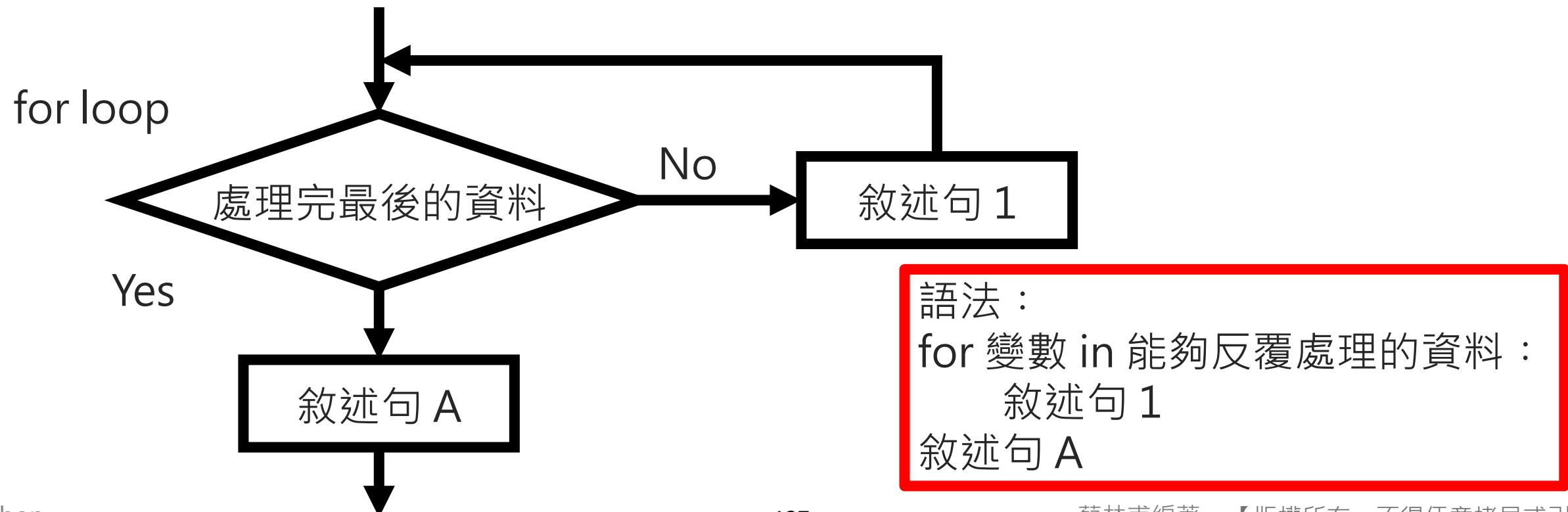
8-5-1 break

8-5-2 continue

平常我們遇到的事情有那些事需要反覆處理的？

比如說：去便利商店結帳時刷條碼的機器正式反覆輸入商品、價格和數量的處理。或是驗鈔機上每一張鈔票需要檢驗真偽鈔，並同時計算鈔票數量。

迴圈的目的就是將一樣的事情或是很類似的事情寫成程式。



```
1     for i in range(6):
2         print("正在執行:", i)
3
4     print("program end")
```

### 輸出結果

正在執行: 0

首先看到 range(6) 的部分，

正在執行: 1

range(6) 會提供給電腦由 0 ~ 5 的數字，最高的數字是 5 也就是  $(6 - 1)$ 。

接著數字會依需給變數 i，變數 i 可將數值帶入迴圈內。

正在執行: 2

所謂的迴圈內是指 for 之下有縮排的部分。

正在執行: 3

等到 for 迴圈所指定數量的程式做完，才會接著前往下一段程式。

正在執行: 4

正在執行: 5

program end

```
1  for i in range(1, 6):
2      print("正在執行:", i)
3
4  print("program end")
```

## 輸出結果

正在執行: 1

正在執行: 2

正在執行: 3

正在執行: 4

正在執行: 5

program end

如果你最初的數字不想從 0 開始，你也可以設定最初開始的數字。  
這時最高的數字還是 5 也就是 (6 - 1)。

## 8-1 for 迴圈

```
1     for i in range(1, 6+1):
2         print("正在執行:", i)
3
4     print("program end")
```

輸出結果

正在執行: 1

正在執行: 2

正在執行: 3

正在執行: 4

正在執行: 5

正在執行: 6

program end

如果你希望看到最高的數字是你想要的數字，而不是要做「減 1」，  
那就自己幫它加個 1。

## 8-1 for 迴圈

如果你需要有間格的數字，而不是固定加 1，你也可以幫它設定間隔。

```
1 for i in range(1, 6+1, 1):  
2     print("正在執行:", i)  
3  
4 print("program end")
```

輸出結果

正在執行: 1  
正在執行: 2  
正在執行: 3  
正在執行: 4  
正在執行: 5  
正在執行: 6  
program end

```
1 for i in range(1, 6+1, 2):  
2     print("正在執行:", i)  
3  
4 print("program end")
```

輸出結果

正在執行: 1  
正在執行: 3  
正在執行: 5  
program end

```
1 for i in range(1, 6+1, 3):  
2     print("正在執行:", i)  
3  
4 print("program end")
```

輸出結果

正在執行: 1  
正在執行: 4  
program end

## 8-1 for 迴圈

如過你想做反向運輸的話，設開始數值、結束數值以及間格用負的。

```
1 for i in range(6, 0, -1):  
2     print("正在執行:", i)  
3  
4 print("program end")
```

輸出結果

正在執行: 6  
正在執行: 5  
正在執行: 4  
正在執行: 3  
正在執行: 2  
正在執行: 1  
program end

```
1 for i in range(6, 0, -2):  
2     print("正在執行:", i)  
3  
4 print("program end")
```

輸出結果

正在執行: 6  
正在執行: 4  
正在執行: 2  
program end

注意：反向運輸的最低值這時是 1 (0+1)。

```
1     j = 0
2     for i in range(0, 10+1, 1):
3         j += i
4
5     print("Total j: ", j)
```

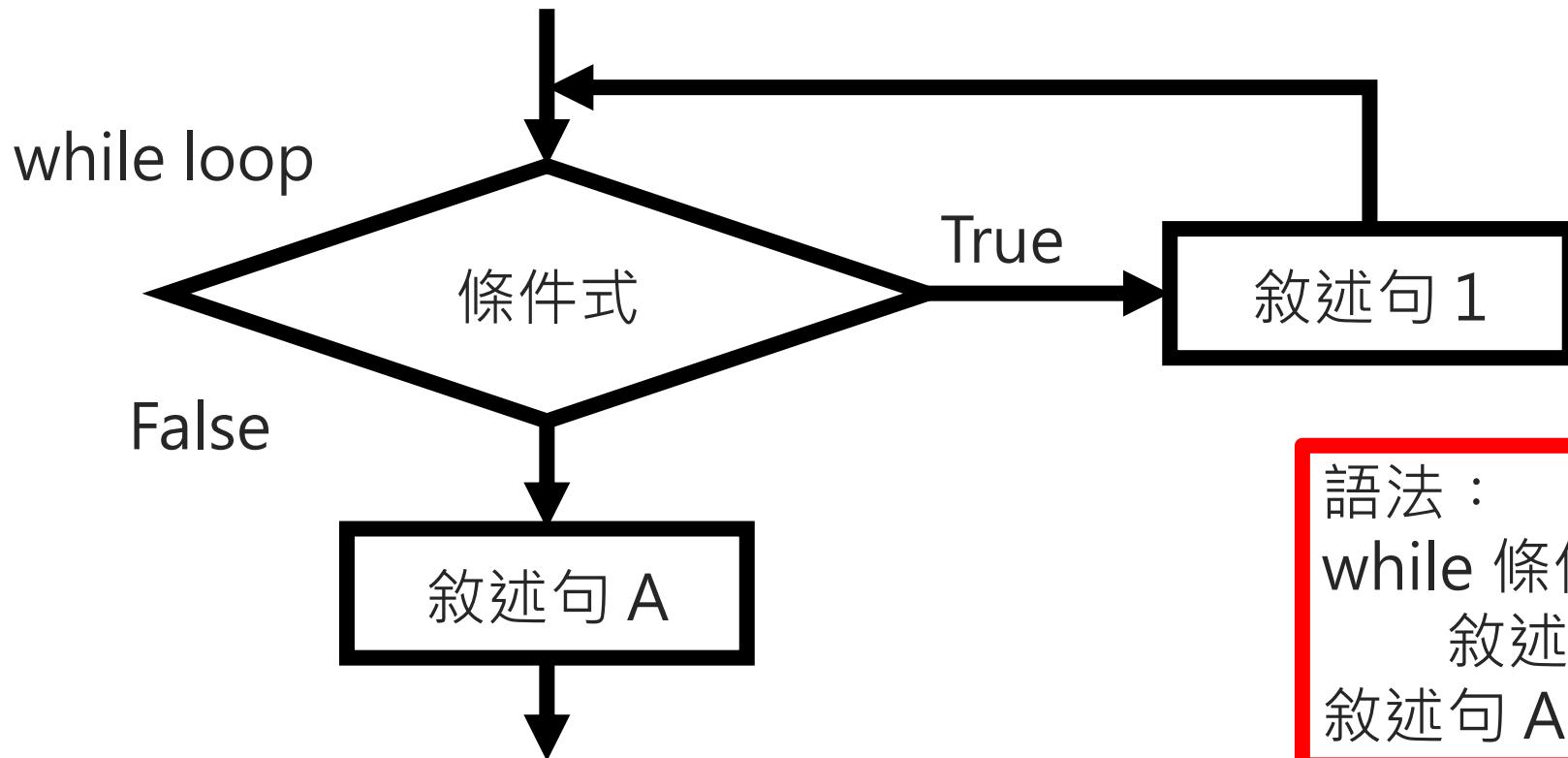
輸出結果      Total j: 55

目前只是先學著熟悉 for 迴圈的簡單語法。  
接下來的需到越多東西，就能讓 for 迴圈提供更高的性能。

迴圈內部可以寫流程控制，當然流程控制內也可以有迴圈。  
但一定要注意縮排的位置。

```
1  for i in range(1, 6, 1):      輸出結果      1 is odd
2      if (i % 2) == 0:           2 is even
3          print(i, "is even")
4      else:                     3 is odd
5          print(i, "is odd")    4 is even
6      print("program end")     5 is odd
7
```

在學習 while 迴圈之前，必須先注意，它有可能成為無窮迴圈。



語法：  
while 條件式：  
    敘述句 1  
    敘述句 A

## 8-2 while 迴圈

```
1     i = 1
2
3     while i <= 6:
4         print("正在執行第", i, "次迴圈")
5         i += 1
6
7     print("program end")
```

輸出結果

正在執行第 1 次迴圈

正在執行第 2 次迴圈

正在執行第 3 次迴圈

正在執行第 4 次迴圈

正在執行第 5 次迴圈

正在執行第 6 次迴圈

program end

while 後面的條件式為「True」的情況下，會執行迴圈內的程式。  
直到條件式為「False」的情況下，才會停止 while 迴圈。

如果條件式一直是「True」的情況下，就會形成無窮迴圈。

## 8-3 for、while 差異

在 Python 程式語言中，就只有 for 和 while 兩種迴圈，  
為了判斷何時適合用 for，何時適合用 while 迴圈，  
所以你必須了解兩者的最大差別。

如果你可以知道何時開始、何時結束、中間會做幾次迴圈，那就用 for 迴圈。  
如果你無法知道要做幾次，那就用 while 迴圈。

迴圈的巢狀結構簡單來說就是迴圈之中還有迴圈。

```
1  for i in range(1, 3+1, 1):
2      for j in range(1, 3+1, 1):
3          print("外圈 i 執行第", i, "圈\t內圈 j 執行第", j, "圈")
4          print("-" * 10)
5  print("program end")
```

外圈的 for 迴圈是以變數 i 控制執行，  
經過縮排內是以變數 j 控制執行。

要注意的是：

外圈第 1 圈，內圈執行 1、2、3 圈，  
接著才執行外圈第 2 圈。

### 輸出結果

外圈 i 執行第 1 圈	內圈 j 執行第 1 圈
外圈 i 執行第 1 圈	內圈 j 執行第 2 圈
外圈 i 執行第 1 圈	內圈 j 執行第 3 圈
-----	
外圈 i 執行第 2 圈	內圈 j 執行第 1 圈
外圈 i 執行第 2 圈	內圈 j 執行第 2 圈
外圈 i 執行第 2 圈	內圈 j 執行第 3 圈
-----	
外圈 i 執行第 3 圈	內圈 j 執行第 1 圈
外圈 i 執行第 3 圈	內圈 j 執行第 2 圈
外圈 i 執行第 3 圈	內圈 j 執行第 3 圈
-----	
program end	

## 8-4 迴圈的巢狀結構

```
1     i = 1
2     while i <= 3:
3         j = 1
4         while j <= 3:
5             print("外圈 i 執行第", i, "圈\n內圈 j 執行第", j, "圈")
6             j += 1
7             print("-" * 10)
8         i += 1
9     print("program end")
```

## 輸出結果

外圈 i 執行第 1 圈 內圈 j 執行第 1 圈

外圈 i 執行第 1 圈 內圈 j 執行第 2 圈

外圈 i 執行第 1 圈 內圈 j 執行第 3 圈

-----

外圈 i 執行第 2 圈 內圈 j 執行第 1 圈

外圈 i 執行第 2 圈 內圈 j 執行第 2 圈

外圈 i 執行第 2 圈 內圈 j 執行第 3 圈

-----

外圈 i 執行第 3 圈 內圈 j 執行第 1 圈

外圈 i 執行第 3 圈 內圈 j 執行第 2 圈

外圈 i 執行第 3 圈 內圈 j 執行第 3 圈

-----

program end

```
1     P = False
2
3     for i in range(0, 5, 1):
4         for j in range(0, 9, 1):
5             if P is False:
6                 print("*", end="")
7                 P = True
8             else:
9                 print("-", end="")
10            P = False
11
12        print()
```

輸出結果

\*-\*-\*-\*-\*

-\*-\*-\*-\*-\*

\*-\*-\*-\*-\*

-\*-\*-\*-\*

\*-\*-\*-\*-\*

for 迴圈的巢狀結構教過了，  
if ... else 教過了，  
is 身份運算子教過了，  
請試著自己解析程式的運作過程。

目前為止，所學的迴圈是從初始值依序做一直做到結束，  
如果想要變更處理流程有兩種方式：

1. 跳出迴圈，停止迴圈的運作，去做之後的程式。使用 break 敘述。
2. 停止迴圈的這一輪，從下一輪開始繼續迴圈。使用 continue 敘述。

語法：  
`break`

語法：  
`continue`

## 8-5-1 break

```
1 stop_condition = int(input("請輸入做到第幾圈時要跳出迴圈: (1 ~ 100):"))
2
3 for i in range(1, 100+1, 1):
4     print("目前做到第", i, "圈", end="\t")
5     if i == stop_condition:
6         break
7     print("Do something process.")
8
9     print()
10    print("----- program end -----")
```

輸出結果

請輸入做到第幾圈時要跳出迴圈: (1 ~ 100):**5**  
目前做到第 1 圈 Do something process.  
目前做到第 2 圈 Do something process.  
目前做到第 3 圈 Do something process.  
目前做到第 4 圈 Do something process.  
目前做到第 5 圈  
----- program end -----

```
1     stop_condition = int(input("請輸入做到第幾圈時要跳出迴圈: (1 ~ 8):"))
2
3     for i in range(1, 8+1, 1):
4         print("目前做到第", i, "圈", end="\t")           輸出結果
5         if i == stop_condition:
6             print()
7             continue
8         print("Do something process.")
9
10        print()
11        print("----- program end -----")
```

請輸入做到第幾圈時要跳出迴圈: (1 ~ 8):5  
目前做到第 1 圈 Do something process.  
目前做到第 2 圈 Do something process.  
目前做到第 3 圈 Do something process.  
目前做到第 4 圈 Do something process.  
目前做到第 5 圈  
目前做到第 6 圈 Do something process.  
目前做到第 7 圈 Do something process.  
目前做到第 8 圈 Do something process.

----- program end -----

## 作業實作

~每個範例都打三遍後再做作業~

請撰寫一個程式運用巢狀迴圈印出九九乘法表。

## 輸出結果

1 * 1 = 1	2 * 1 = 2	3 * 1 = 3	4 * 1 = 4	5 * 1 = 5	6 * 1 = 6	7 * 1 = 7	8 * 1 = 8	9 * 1 = 9
1 * 2 = 2	2 * 2 = 4	3 * 2 = 6	4 * 2 = 8	5 * 2 = 10	6 * 2 = 12	7 * 2 = 14	8 * 2 = 16	9 * 2 = 18
1 * 3 = 3	2 * 3 = 6	3 * 3 = 9	4 * 3 = 12	5 * 3 = 15	6 * 3 = 18	7 * 3 = 21	8 * 3 = 24	9 * 3 = 27
1 * 4 = 4	2 * 4 = 8	3 * 4 = 12	4 * 4 = 16	5 * 4 = 20	6 * 4 = 24	7 * 4 = 28	8 * 4 = 32	9 * 4 = 36
1 * 5 = 5	2 * 5 = 10	3 * 5 = 15	4 * 5 = 20	5 * 5 = 25	6 * 5 = 30	7 * 5 = 35	8 * 5 = 40	9 * 5 = 45
1 * 6 = 6	2 * 6 = 12	3 * 6 = 18	4 * 6 = 24	5 * 6 = 30	6 * 6 = 36	7 * 6 = 42	8 * 6 = 48	9 * 6 = 54
1 * 7 = 7	2 * 7 = 14	3 * 7 = 21	4 * 7 = 28	5 * 7 = 35	6 * 7 = 42	7 * 7 = 49	8 * 7 = 56	9 * 7 = 63
1 * 8 = 8	2 * 8 = 16	3 * 8 = 24	4 * 8 = 32	5 * 8 = 40	6 * 8 = 48	7 * 8 = 56	8 * 8 = 64	9 * 8 = 72
1 * 9 = 9	2 * 9 = 18	3 * 9 = 27	4 * 9 = 36	5 * 9 = 45	6 * 9 = 54	7 * 9 = 63	8 * 9 = 72	9 * 9 = 81

## 作業實作

~每個範例都打三遍後再做作業~

還記得上一個作業是關於計算 BMI 並把結果分不同的流程控制，但它做一遍就結束了。

如果我們想讓他做很多遍，但又不知道要做幾遍，所以適合用哪個迴圈去撰寫呢？

請撰寫一個程式，輸出結果如下：

請輸入身高(cm): 174

請輸入體重(kg): 60

您的身高為: 174.0 (cm) 體重為: 60.0 (kg)

BMI 值為: 19.817677368212443

\*\*\*\*\*體重標準\*\*\*\*\*

請輸入執行代碼:(1. 繼續, 2. 停止):

程式的最後會問你

1. 繼續
2. 停止

如果繼續那就會接著執行迴圈，執行完後還會再問你一遍。

如果選擇停止，就會跳出迴圈。

也就是說繼續和停止哪個會用到 break，又是誰用到 continue!?

如果輸入錯誤那程式就一直跟你耗，那又要再進一步用什麼迴圈跟你耗!?

## 作業實作

### ~每個範例都打三遍後再做作業~

請輸入身高(cm): 174

請輸入體重(kg): 60

您的身高為: 174.0 (cm) 體重為: 60.0 (kg)

BMI 值為: 19.817677368212443

\*\*\*\*\*體重標準\*\*\*\*\*

請輸入執行代碼:(1. 繼續, 2. 停止): a

輸入錯誤請重新輸入

請輸入執行代碼:(1. 繼續, 2. 停止): b

輸入錯誤請重新輸入

請輸入執行代碼:(1. 繼續, 2. 停止): 1

-----  
請輸入身高(cm): 180

請輸入體重(kg): 75

您的身高為: 180.0 (cm) 體重為: 75.0 (kg)

BMI 值為: 23.148148148148145

\*\*\*\*\*體重標準\*\*\*\*\*

請輸入執行代碼:(1. 繼續, 2. 停止):

如果輸入錯誤，那麼它就會一直說輸入錯誤請重新輸入，也就是它會再進入一個迴圈，知道輸入正確才跳出迴圈，回到外部迴圈繼續執行。

直到你選擇 1 或 2，  
一樣輸入 1. 繼續 2. 停止。

~這題的流程雖然比較複雜，  
但所用到的程式都有教過~

# Module 9 :

## 群集-串列(List)

9-1 群集 (collection)的介紹

9-2 串列的基本知識

9-2-1 建立串列

9-2-2 建立空串列

9-2-3 取得串列中元素的值

9-2-4 顯示串列長度

9-2-5 串列搭配成員運算子(in、not in)

9-2-6 從迴圈中操作串列

9-2-7 迴圈的特殊方法

9-3 串列的操作

9-3-1 變更串列的值

9-3-2 新增、插入串列的值

9-3-3 刪除串列的值

9-3-4 取出後刪除串列的值

9-3-5 取得串列中的值出現次數與索引值

# Module 9 :

## 群集-串列

9-4 串列的注意事項-串列的指定

9-4-1 建立新串列

9-5 串列的連接與切片

9-5-1 串列的連接

9-5-2 串列的切片

9-5-3 逆向排列的串列

9-5-4 認識與使用疊代器

9-6 串列的打包與解包

9-6-1 串列的打包

9-6-2 串列的解包

9-6-3 解包後的指定

9-7 串列推導式

9-8 串列的統計與排序

9-8-1 串列的統計

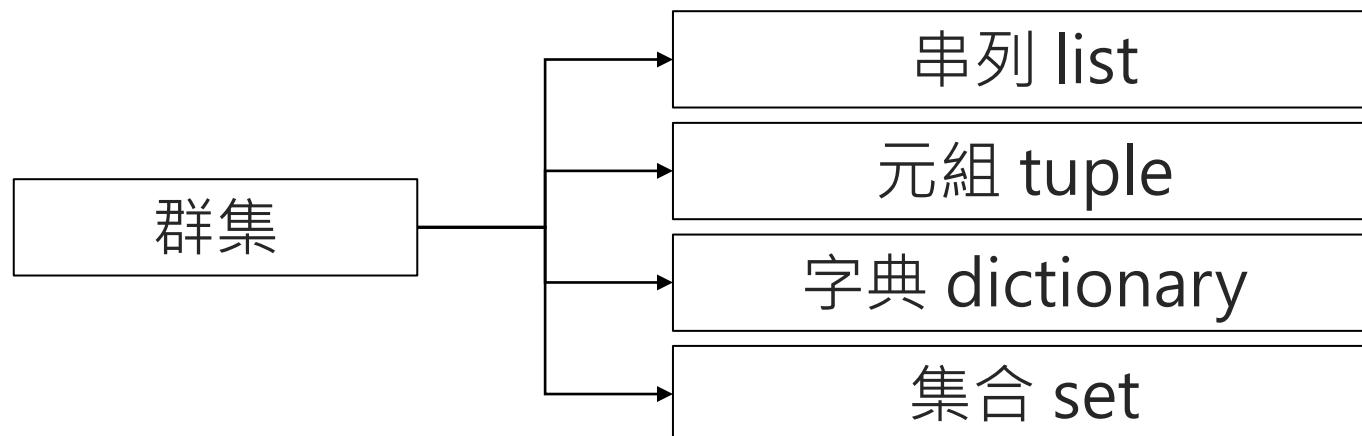
9-8-2 串列的排序

9-9 多維串列

撰寫程式的目的是協助我們彙整資料，  
在 Python 中彙整資料的方式稱為群集(collection)，  
而群集又分為四種：

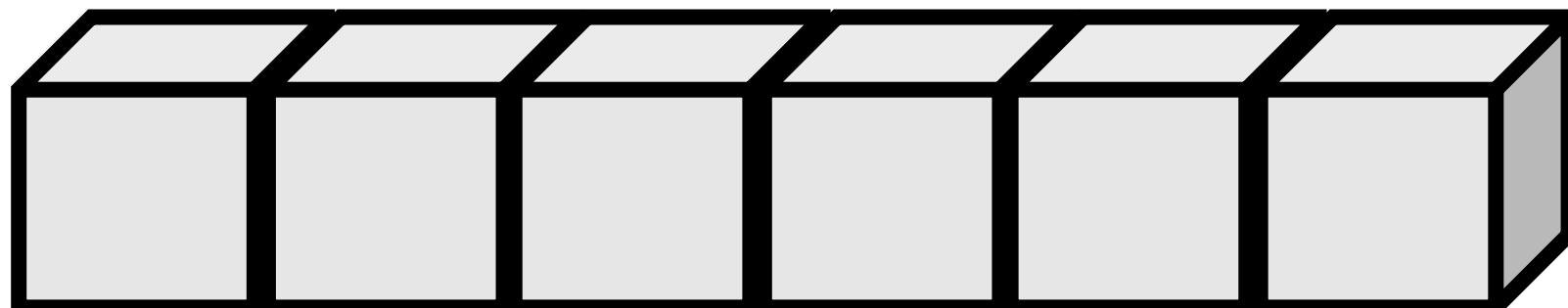
1. 串列(list)
2. 元組(tuple)
3. 字典(dictionary)
4. 集合(set)

這四種的存取方式各有差異，熟悉各種方式存取的特性，能夠讓你更簡便的處理資料。



我們可以先想像串列(list)就像一串盒子，每一個盒子中可以存放資料。盒子有從 0 開始的編號，我們可以選擇編號進而取得盒子內的資料。盒子的編號又稱為位置、索引。盒子內的資料又稱為元素(element)。我們可以也可以修改盒子內的資料，新增盒子的數量以新增資料，刪除盒子的數量以刪除資料。

換句話說，串列(list)可以做新增、修改、插入和刪除，這種特性又稱為可變的(mutable)。



編號

0

1

2

3

4

5

編號

-6

-5

-4

-3

-2

-1

語法：

串列名稱 = [值1, 值2, ...]

建立串列時，使用中括號「[]」，其中每一個盒子的資料是以逗號「，」隔開，接著將它指定給串列名稱，串列名稱也就是變數。

```
1 score = [98, 99, 87, 78, 56]  
2 print("score: ", score)
```

輸出結果        score: [98, 99, 87, 78, 56]

```
1 drink = ["coffee", "tea", "juice", "water", "milk"]  
2 print("drink: ", drink)
```

輸出結果        drink: ['coffee', 'tea', 'juice', 'water', 'milk']

如果串列有點長，也可以換行寫。

```
1      score = [98, 99, 87, 78, 56,  
2                  97, 99, 87, 78, 56]  
3      print("score: ", score)
```

輸出結果 score: [98, 99, 87, 78, 56, 97, 99, 87, 78, 56]

如果需要先建立空的串列，有兩種方法：

1. 直接只寫 []
2. list()

```
1 score = []
2 print("score: ", score)
3 print("type(score): ", type(score))
```

輸出結果 score: []
type(score): <class 'list'>

```
1 score = list()
2 print("score: ", score)
3 print("type(score): ", type(score))
```

輸出結果 score: []
type(score): <class 'list'>

要取得串列的資料，只要在串列名稱後加入中括號[]，並寫下盒子的編號，就會取得所指定編號內的資料。

1	score = [98, 99, 87, 78, 56]	輸出結果	score[0]: 98
2			score[1]: 99
3	print("score[0]: ", score[0])		score[3]: 78
4	print("score[1]: ", score[1])		score[4]: 56
5	print("score[3]: ", score[3])		score[-1]: 56
6	print("score[4]: ", score[4])		score[-3]: 87
7	print("score[-1]: ", score[-1])		score[-5]: 98
8	print("score[-3]: ", score[-3])		
9	print("score[-5]: ", score[-5])		

如果想要知道串列的長度，可以使用內建函數 `len()`。

語法：

`len(串列名稱)`

```
1 score = [98, 99, 87, 78, 56]
2 drink = ["coffee", "tea", "juice", "water", "milk"]
3 print("length of list score: ", len(score))
4 print("length of list drink: ", len(drink))
```

輸出結果

length of list score: 5

length of list drink: 5

# 緯 育 TibaMe 9-2-5 串列搭配成員運算子(in、not in)

之前在學習運算子時，並未介紹如何使用，  
如今介紹了串列，那麼再來介紹成員運算子的用法會比較好了解。  
成員運算子可以檢查串列中是否存在成員，範例如下：

1	score = [98, 99, 87, 78, 56]	輸出結果	1.	False
2			2.	True
3	print("1.\"", 90 in score)		3.	True
4	print("2.\"", 99 in score)		4.	True
5	print("3.\"", 78 in score)		5.	False
6	print("4.\"", 90 not in score)		6.	False
7	print("5.\"", 99 not in score)			
8	print("6.\"", 78 not in score)			

串列可以搭配迴圈使用。它會依序處理串列的資料。

```
1     score = [98, 99, 87, 78, 56]  
2  
3     for i in score:  
4         print("分數:", i)
```

輸出結果 分數: 98  
          分數: 99  
          分數: 87  
          分數: 78  
          分數: 56

串列可以搭配迴圈使用，也可以再搭配流程控制。

```
1     score = [98, 99, 87, 76, 56]
2     for i in score:
3         if i > 60:
4             print(i, "分數合格")
5         else:
6             print(i, "分數不合格")
```

輸出結果      98 分數合格  
                  99 分數合格  
                  87 分數合格  
                  76 分數合格  
                  56 分數不合格

現在我們對於串列有初步的知識，  
也學過搭配for loop 和 while loop，和結合流程控制 if... else，  
另外這裡插播一個在 python 中的特殊用法：

1. for / else
2. while / else

語法：

for 變數 in 能夠反覆處理的資料：

if 條件式:

    敘述句 1

    break

else:

    敘述句 2

    敘述句 A

for 迴圈的執行過程中，  
如果沒有觸發 break 就會執行 else  
的敘述。

如過有觸發 break，  
就不會執行 else 的敘述。

敘述句 A 都會執行

```
1     num = [31, 21, 54, 61, 62]
2     for i in num:
3         if i > 100:
4             print("number is", i)
5             break
6     else:
7         print("No element bigger than 100")
```

輸出結果

No element bigger than 100

```
1     num = [31, 21, 54, 110, 61, 62]
2     for i in num:
3         if i > 100:
4             print("number is", i)
5             break
6     else:
7         print("No element bigger than 100")
```

輸出結果

number is 110

## 9-2-7 迴圈的特殊方法

語法：

While 條件式：

if 條件式：

敘述句 1

break

else:

敘述句 2

敘述句 A

while 迴圈的執行過程中，  
如果沒有觸發 break 就會執行 else 的敘述。

如過有觸發 break，  
就不會執行 else 的敘述。

敘述句 A 都會執行

```
1     num = [31, 21, 54, 61, 62]
2     i = 0
3     while i < len(num):
4         if num[ i ] > 100:
5             print("number is", num[ i ])
6             break
7         i += 1
8     else:
9         print("No element bigger than 100")
```

輸出結果      No element bigger than 100

```
1     num = [31, 21, 54, 110, 61, 62]
2     i = 0
3     while i < len(num):
4         if num[ i ] > 100:
5             print("number is", num[ i ])
6             break
7         i += 1
8     else:
9         print("No element bigger than 100")
```

輸出結果      number is 110

變更串列的元素：

選擇要變更的編號以及要變更的值。

```
1 score = [98, 99, 87, 78, 56]
2
3 print("變更前的 score 資料: ", score)
4 score[1] = 100
5 score[4] = 59.9
6 print("變更後的 score 資料: ", score)
```

輸出結果 變更前的 score 資料: [98, 99, 87, 78, 56]

變更後的 score 資料: [98, 100, 87, 78, 59.9]

要在串列中加入新的元素有兩種方法：

1. `append()`，可以在串列的尾端加入新的值。
2. `insert()`，可以在指定的位置(索引)插入值。被插隊的值會順勢往後排。

語法：

串列名稱.append(值)

```
1 score = [98, 99, 87, 78, 56]
2 print("新增前的 score 資料: ", score)
3 score.append(100)
4 print("新增後的 score 資料: ", score)
```

輸出結果

新增前的 score 資料: [98, 99, 87, 78, 56]

新增後的 score 資料: [98, 99, 87, 78, 56, 100]

語法：

串列名稱.insert(位置, 值)

```
1 score = [98, 99, 87, 78, 56]
2 print("插入前的 score 資料: ", score)
3 score.insert(2, 100)
4 print("插入後的 score 資料: ", score)
```

輸出結果

插入前的 score 資料: [98, 99, 87, 78, 56]

插入後的 score 資料: [98, 99, 100, 87, 78, 56]

## 9-3-3 刪除串列的值

刪除串列中的元素有兩種：

1. `del`：以索引值為刪除依據。
2. `remove()`：刪除指定的值，但只有最左邊的被刪除

語法：

`del 串列名稱[索引值]`

```
1 score = [98, 99, 87, 78, 56]
2 print("刪除前的 score 資料: ", score)
3 del score[2]
4 print("刪除後的 score 資料: ", score)
```

輸出結果

刪除前的 score 資料: [98, 99, 87, 78, 56]  
刪除後的 score 資料: [98, 99, 78, 56]

語法：

`串列名稱.remove[值]`

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 print("刪除前的 score 資料: ", score)
3 score.remove(78)
4 print("刪除後的 score 資料: ", score)
```

輸出結果 注意：只有最左邊的被刪除

刪除前的 score 資料: [98, 99, 87, 78, 56, 78, 78]  
刪除後的 score 資料: [98, 99, 87, 56, 78, 78]

那如果要全刪除的話怎麼做呢？請依照先前所學，寫寫看。

## 作業實作

~每個範例都打三遍後再做作業~

請撰寫一個程式，程式中有建立一個串列資料 [98, 99, 87, 78, 56, 78, 78]，接著會讓你有可以選擇輸入要刪除的值的地方，輸入完後會把你輸入的值刪光光。

執行結果如下：

```
score 資料: [98, 99, 87, 78, 56, 78, 78]
```

```
請輸入要刪除的值: 78
```

```
刪除後的 score 資料: [98, 99, 87, 56]
```

## 9-3-3 刪除串列的值

如果用remove()刪除值的方式刪除如果串列中本身沒有那個值會出現的狀況如下：

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 print("刪除前的 score 資料: ", score)
3 score.remove(100)
4 print("刪除後的 score 資料: ", score)
```

輸出結果

```
刪除前的 score 資料: [98, 99, 87, 78, 56, 78, 78]
Traceback (most recent call last):
  File "D:/Python/workspace/Sample_9_15.py", line 3, in <module>
    score.remove(100)
ValueError: list.remove(x): x not in list
```

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 print("刪除前的 score 資料: ", score)
3 remove_value = 100
4 if remove_value in score:
5     score.remove(remove_value)
6 print("刪除後的 score 資料: ", score)
```

輸出結果

```
刪除前的 score 資料: [98, 99, 87, 78, 56, 78, 78]
刪除後的 score 資料: [98, 99, 87, 78, 56, 78, 78]
```

之後學習 try...except 會教如何處理例外狀況。

## 9-3-4 取出後刪除串列的值

可以將串列的值取出使用，同時被取出的值在串列中會刪除。  
預設由最後一個值開始取用，也可以指定索引值

語法：

串列名稱.pop(index)

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 print("原本的", score)
3 print("取出的值", score.pop())
4 print("後來的", score)
```

輸出結果

```
原本的 [98, 99, 87, 78, 56, 78, 78]
取出的值 78
後來的 [98, 99, 87, 78, 56, 78]
```

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 print("原本的", score)
3 print("取出的值", score.pop(2))
4 print("後來的", score)
```

輸出結果

```
原本的 [98, 99, 87, 78, 56, 78, 78]
取出的值 87
後來的 [98, 99, 78, 56, 78, 78]
```

# 緯育 TibaMe 9-3-5 取得串列中的值出現次數與索引值

可以使用

1. count(元素值)：查看該元素值共有幾個。
2. index(元素值)：查看該元素值第一次出現的 index。

語法：

串列名稱.count(value)

語法：

串列名稱.index(value)

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 print("出現次數: ", score.count(78))
3 print("出現索引值: ", score.index(78))
4
5 print("出現次數: ", score.count(56))
6 print("出現索引值: ", score.index(56))
```

輸出結果

出現次數: 3

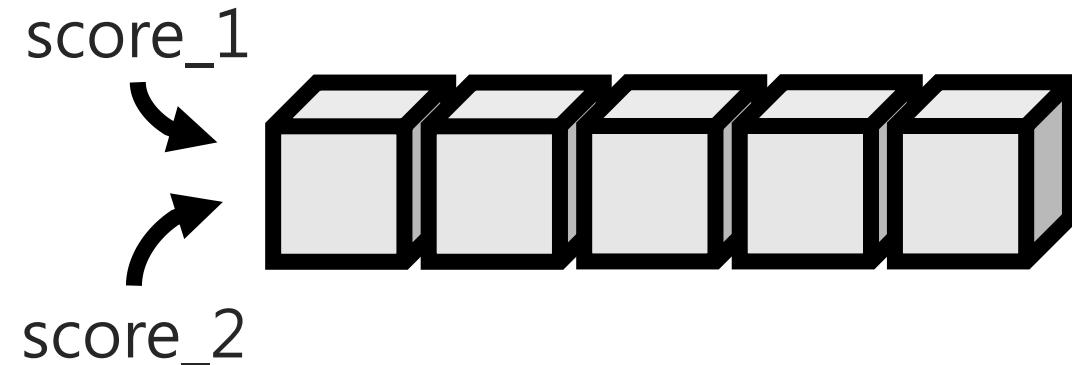
出現索引值: 3

出現次數: 1

出現索引值: 4

如果把串列指定給另一個串列，這時兩個串列會一起使用同一個資料，所以當修改其中一個串列的資料時，另一個串列也會同時修改。

```
1 score_1 = [98, 99, 87, 78, 56]
2 score_2 = score_1
3 print("原本的 score_1", score_1)
4 print("原本的 score_2", score_2)
5 score_2[4] = 59.9
6 print("修改後的 score_1", score_1)    輸出結果
7 print("修改後的 score_2", score_2)
8 print("-" * 30)
9 print("id of score_1", id(score_1))
10 print("id of score_2", id(score_2))
```



```
原本的 score_1 [98, 99, 87, 78, 56]
原本的 score_2 [98, 99, 87, 78, 56]
修改後的 score_1 [98, 99, 87, 78, 59.9]
修改後的 score_2 [98, 99, 87, 78, 59.9]
-----
id of score_1 2190236925512
id of score_2 2190236925512
```

~ `id()` 用來查看該變數在記憶體的位置。~

如果要讓兩個串列使用不同的資料，就需要建立新的串列，建立新的串列有兩種方式：

1. list()
2. copy()

```
1 score_1 = [98, 99, 87, 78, 56]
2 score_2 = list(score_1)
3 print("原本的 score_1", score_1)
4 print("原本的 score_2", score_2)
5 score_2[4] = 59.9
6 print("修改後的 score_1", score_1)
7 print("修改後的 score_2", score_2)
8 print("-" * 30)
9 print("id of score_1", id(score_1))
10 print("id of score_2", id(score_2))
```

輸出結果

原本的 score\_1 [98, 99, 87, 78, 56]

原本的 score\_2 [98, 99, 87, 78, 56]

修改後的 score\_1 [98, 99, 87, 78, 56]

修改後的 score\_2 [98, 99, 87, 78, 59.9]

-----  
id of score\_1 1992634613448

id of score\_2 1992634713672

~如此以來記憶體的位置也會不同。~

## 9-4-1 建立新串列

```
1 score_1 = [98, 99, 87, 78, 56]    輸出結果
2 score_2 = score_1.copy()          原本的 score_1 [98, 99, 87, 78, 56]
3 print("原本的 score_1", score_1)   原本的 score_2 [98, 99, 87, 78, 56]
4 print("原本的 score_2", score_2)   修改後的 score_1 [98, 99, 87, 78, 56]
5 score_2[4] = 59.9                修改後的 score_2 [98, 99, 87, 78, 59.9]
6 print("修改後的 score_1", score_1) -----
7 print("修改後的 score_2", score_2) id of score_1 1531019538248
8 print("-" * 30)                  id of score_2 1531018695432
9 print("id of score_1", id(score_1))
10 print("id of score_2", id(score_2))
```

~如此以來記憶體的位置也會不同。~

## 9-5-1 串列的連接

如果要將兩個串列連接起來成為新的串列，有兩種方式：

1. 用「+」相加的方式將串列連接。
2. 用 `extend()`，用擴大的方式去擴大要被擴大的串列，讓串列連接。

語法：

串列1 + 串列2

```
1     number_1 = [98, 99, 87, 78, 56]
2     number_2 = [4, 9, 16]
3     number_new = number_1 + number_2
4     print("number_new: ", number_new)
```

輸出結果

number\_new: [98, 99, 87, 78, 56, 4, 9, 16]

語法：

串列1.`extend(串列2)`

```
1     number_1 = [98, 99, 87, 78, 56]
2     number_2 = [4, 9, 16]
3     number_1.extend(number_2)
4     print("number_1: ", number_1)
```

輸出結果

number\_1: [98, 99, 87, 78, 56, 4, 9, 16]

運用指定運算子「`+ =`」，連接。

```
1     number_1 = [98, 99, 87, 78, 56]
2     number_2 = [4, 9, 16]
3     number_1 += number_2
4     print("number_1:", number_1)
```

輸出結果

```
number_1: [98, 99, 87, 78, 56, 4, 9, 16]
```

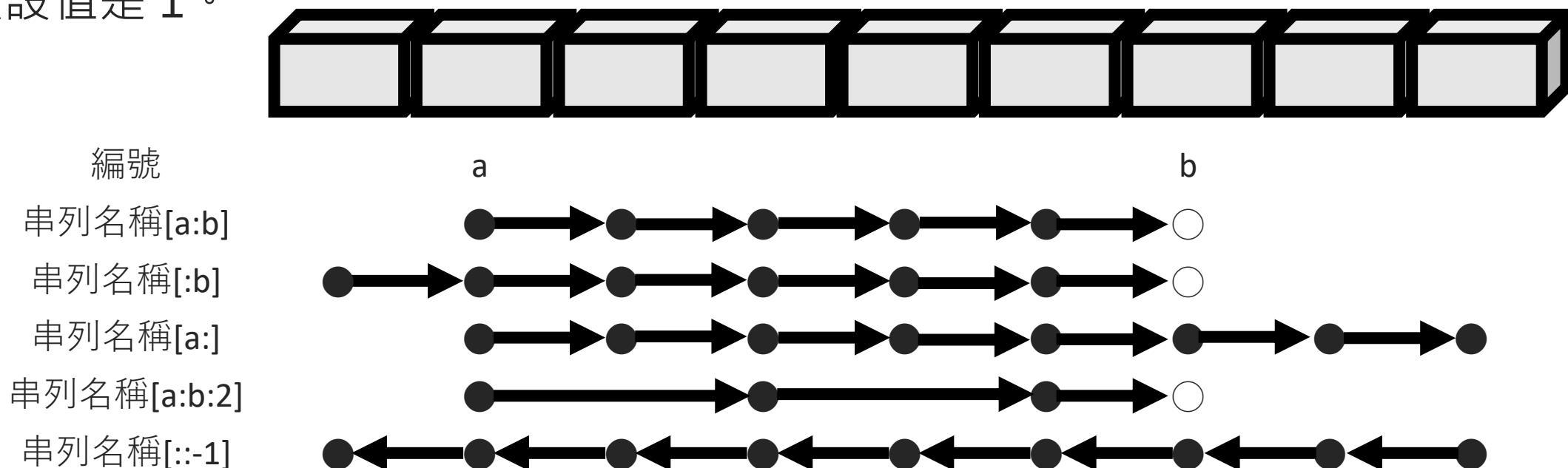
## 9-5-2 串列的切片

串列的切片，是所謂的藉由選定串列的索引值範圍，以取得該範圍的串列。

語法：

串列名稱[起始索引:結束索引:間隔]

起始索引的預設值是取第一個，結束索引的預設值是取道最後一個，  
間隔的預設值是 1。



## 9-5-2 串列的切片

```
1 number = [98, 99, 87, 78, 56, 4, 9, 16] 輸出結果
2 number_1 = number[3:7]
3 print("1.\\t", number_1)
4 number_2 = number[:7]
5 print("2.\\t", number_2)
6 number_3 = number[3:]
7 print("3.\\t", number_3)
8 number_4 = number[3:7:2]
9 print("4.\\t", number[3:7:2])
10 number_5 = number[::-1]
11 print("5.\\t", number[::-1])
```

1. [78, 56, 4, 9]
2. [98, 99, 87, 78, 56, 4, 9]
3. [78, 56, 4, 9, 16]
4. [78, 4]
5. [16, 9, 4, 56, 78, 87, 99, 98]

也可以將切片的範圍改值。

```
1     number = [98, 99, 87, 78, 56, 4, 9, 16]
2     print("原本的串列: ", number)
3     number[3:7] = [100, 59.9, 49.9, 99.8]
4     print("改值後的串列", number)
```

輸出結果

原本的串列: [98, 99, 87, 78, 56, 4, 9, 16]

改值後的串列 [98, 99, 87, 100, 59.9, 49.9, 99.8, 16]

也可以將切片的範圍刪除。

```
1     number = [98, 99, 87, 78, 56, 4, 9, 16]
2     print("原本的串列: ", number)
3     del number[3:7]
4     print("改值後的串列", number)
```

輸出結果

原本的串列: [98, 99, 87, 78, 56, 4, 9, 16]

改後的串列 [98, 99, 87, 16]

要讓串列做逆向排列有三種方式，雖然結果都是逆向排列，但使用上有所差異。

1. 使用切片方式，原本的資料不會改變。變數名稱`[::-1]`。
2. 使用物件方法「變數名稱`.reverse()`」，原本的資料會改變。變數名稱`.reverse()`。
3. 使用內建函數「`reversed(變數名稱)`」，原本的資料不會改變，會得到疊代器(iterator)。`reversed(變數名稱)`。

```
1 number = [4, 9, 16, 25, 36, 49]
2 print("原本的串列: ", number)
3 rev_number = number[::-1]
4 print("逆向後的 number 資料: ", number)
5 print("逆向後的 rev_number 資料: ", rev_number)
```

輸出結果

原本的串列: [4, 9, 16, 25, 36, 49]

逆向後的 number 資料: [4, 9, 16, 25, 36, 49]

逆向後的 rev\_number 資料: [49, 36, 25, 16, 9, 4]

## 9-5-3 逆向排列的串列

```
1     number = [4, 9, 16, 25, 36]
2     print("原本的 number 資料: ", number)
3     number.reverse()
4     print("逆向後的 number 資料: ", number)
```

語法：  
變數名稱.reverse()

輸出結果

原本的 number 資料: [4, 9, 16, 25, 36]  
逆向後的 number 資料: [36, 25, 16, 9, 4]

## 9-5-3 逆向排列的串列

```
1  number = [4, 9, 16, 25, 36]
2  print("原本的串列: ", number)
3  rev_number = reversed(number)
4  print("逆向後的 number 資料: ", number)
5  print("逆向後的 rev_number 資料: ", rev_number)
6  for i in rev_number:
7      print(i)
```

輸出結果

原本的串列: [4, 9, 16, 25, 36]  
逆向後的 number 資料: [4, 9, 16, 25, 36]  
逆向後的 rev\_number 資料: <list\_reverseiterator object at 0x000001CFBE032828>  
36  
25  
16  
9  
4

語法：  
`reversed(變數名稱)`

疊代器是利用 for 敘述等，依照順序重複執行的一種機制。

疊代器是反覆處理的機制，而不是反覆處理所得到的結果。

例如在使用 reversed() 的時候，  
所以它可以用for的方式，逆向的只取出值。  
而無法直接給你處理的結果，逆向後的串列。

如果需要疊代器可以使用內建函數 `iter()`，  
將串列等可重複執行的可疊代結構做成疊代器。  
另外除了 `for` 敘述，`next(疊代器名稱)` 也可以依序取出值。

```
1     number = [4, 9, 16, 25, 36]
2     ite_number = iter(number)
3     print("ite_number: ", ite_number)
4     print(next(ite_number))
5     print(next(ite_number))
6     print(next(ite_number))
```

輸出結果      ite\_number: <list\_iterator object at 0x000001F231B62828>
4
9
16

## 9-6-1 串列的打包

如果需要將多個串列合併(打包)，可以使用內建函數 `zip()`。

語法：

`zip(串列 A, 串列 B, ...)`

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2 product_price = [100, 90, 80, 70]
3
4 for i in zip(product_name, product_price):
5     print(i)
```

輸出結果

```
('coffee', 100)
('tea', 90)
('juice', 80)
('milk', 70)
```

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2 product_price = [100, 90, 80]
3
4 for i in zip(product_name, product_price):
5     print(i)
```

輸出結果

```
('coffee', 100)
('tea', 90)
('juice', 80)
```

注意：

資料打包的長度，往後算多的部分不會被打包。

```
1 product_name = ["coffee", "tea", "juice", "milk"] 輸出結果 ('coffee', 100)
2 product_price = [100, 90, 80, 70] <class 'tuple'>
3
4 for i in zip(product_name, product_price): ('tea', 90)
5     print(i) <class 'tuple'>
6     print(type(i)) ('juice', 80)
7
8     print(i) <class 'tuple'>
9     print(type(i)) ('milk', 70)
10
11    print(i) <class 'tuple'>
```

~打包後的資料型態是 tuple，下一章會介紹 tuple。~

## 9-6-1 串列的打包

如果需要將索引值打包，可以使用內建函數 `enumerate()`。

語法：

`enumerate(串列)`

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2
3 for i in enumerate(product_name):
4     print(i)
```

輸出結果

```
(0, 'coffee')
(1, 'tea')
(2, 'juice')
(3, 'milk')
```

```
1 score = [98, 99, 87, 78, 56, 78, 78]
2 look_for = 78
3 for i in enumerate(score):
4     if i[1] == look_for:
5         print("index of", look_for, "is:", i[0])
```

輸出結果

```
index of 78 is: 3
index of 78 is: 5
index of 78 is: 6
```

如果能夠將串列的元素打包，那麼解包的方式如下：

語法：

for 變數A, 變數B in 串列名稱:

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2 product_price = [100, 90, 80, 70]
3
4 for name, price in zip(product_name, product_price):
5     print("商品名稱:", name, "價格:", price)
```

輸出結果

商品名稱: coffee 價格: 100  
商品名稱: tea 價格: 90  
商品名稱: juice 價格: 80  
商品名稱: milk 價格: 70

解包也可以同時指定給變數。

語法：

變數A, 變數B = 串列名稱

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2 name_1, name_2, name_3, name_4 = product_name
3 print(name_1)
4 print(name_2)
5 print(name_3)
6 print(name_4)
```

輸出結果

coffee

tea

juice

milk

注意：串列的長度和指定的變數數量要相等。

## 9-6-3 解包後的指定

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2 product_price = [100, 90, 80, 70]
3 print(zip(product_name, product_price))
4 product_1, product_2, product_3, product_4 = zip(product_name, product_price)
5 print(product_1)
6 print(product_2)
7 print(product_3)
8 print(product_4)
```

輸出結果

```
<zip object at 0x00000209E66634C8>
('coffee', 100)
('tea', 90)
('juice', 80)
('milk', 70)
```

使用串列推導式從串列中取得新的串列。

語法：

[表達式 for 變數 in 串列 if 條件]

```
1 product_price = [100, 90, 80, 70]
2 print("product_price: ", product_price)
3 new_product_price = [i * 0.9 for i in product_price if 70 < i <= 90]
4 print("new_product_price: ", new_product_price)
```

輸出結果

```
product_price: [100, 90, 80, 70]
new_product_price: [81.0, 72.0]
```

語法：

[表達式 for 變數 in 串列]

```
1 product_name = ["coffee", "tea", "juice", "milk"]
2 product_price = [100, 90, 80, 70]
3 print("產品名稱: ", product_name)
4 print("產品價格: ", product_price)
5 new_product_price = [i * 0.5 for i in product_price]
6 print("----全面商品五折優惠----")
7 print("打折後價格", new_product_price)
```

輸出結果

產品名稱: ['coffee', 'tea', 'juice', 'milk']  
產品價格: [100, 90, 80, 70]  
----全面商品五折優惠----  
打折後價格 [50.0, 45.0, 40.0, 35.0]

語法：

[表達式A if 條件 else 表達式B for 變數 in 串列]

```
1 product_name = ["coffee", "tea", "juice", "milk", "water"]
2 product_price = [100, 90, 80, 70, 10]
3 print("產品名稱:", product_name)
4 print("產品價格:", product_price)
5 new_product_price = [int(i * 0.5) if i <= 70 else int(i * 0.9) for i in product_price]
6 print("----全面商品九折優惠----部分商品五折優惠----")
7 print("打折後價格", new_product_price)
```

輸出結果

產品名稱: ['coffee', 'tea', 'juice', 'milk', 'water']

產品價格: [100, 90, 80, 70, 10]

-----全面商品九折優惠-----部分商品五折優惠-----

打折後價格 [90, 81, 72, 35, 5]

```
1 product_name = ["coffee", "tea", "juice", "milk", "water"]
2 product_price = [100, 90, 80, 70, 10]
3 print("產品名稱: ", product_name)
4 print("產品價格: ", product_price)
5 new_product_price = [int(i * 0.95) if i > 80 else
6                     int(i * 0.8) if 20 < i <= 80 else
7                     int(i * 0.1)
8                     for i in product_price]
9 print("----全面商品大優惠----優惠最大是 1 折----")
10 print("打折後價格", new_product_price)
```

輸出結果

產品名稱: ['coffee', 'tea', 'juice', 'milk', 'water']  
產品價格: [100, 90, 80, 70, 10]  
----全面商品大優惠----優惠最大是 1 折----  
打折後價格 [95, 85, 64, 56, 1]

串列推倒式沒有 elif，  
但能在 else 的敘述內  
再做一次 if...else。

## 9-8-1 串列的統計

內建函數	說明
max()	取出最大值
min()	取出最小值
sum()	算出總和
sorted()	將資料排序

```
1     list_n = [31, 21, 54, 61, 62, 55, 18, 63, 64, 72, 74]
2     print("資料最初值: ", list_n)
3     print("資料最大值: ", max(list_n))
4     print("資料最小值: ", min(list_n))
5     print("資料的總和: ", sum(list_n))
6     print("資料的排序: ", sorted(list_n))
7     print("資料反向排序: ", sorted(list_n, reverse=True))
```

輸出結果

資料最初值: [31, 21, 54, 61, 62, 55, 18, 63, 64, 72, 74]  
資料最大值: 74  
資料最小值: 18  
資料的總和: 575  
資料的排序: [18, 21, 31, 54, 55, 61, 62, 63, 64, 72, 74]  
資料反向排序: [74, 72, 64, 63, 62, 61, 55, 54, 31, 21, 18]

串列的排序方式有兩種：

1. sorted(串列名稱)
2. 串列名稱.sort()

sorted() 不會改變原本串列值，sort() 會改變原本串列值。

兩者的預設是升冪排列 reverse = False，若要降冪排列則改成 reverse = True。

```
1     list_n = [31, 21, 54, 61, 62, 55, 18, 63]
2     print("資料最初值: ", list_n)
3     list_n.sort()
4     print("排序後的資料 (生冪): ", list_n)
5     list_n.sort(reverse=True)
6     print("排序後的資料 (降冪): ", list_n)
```

輸出結果

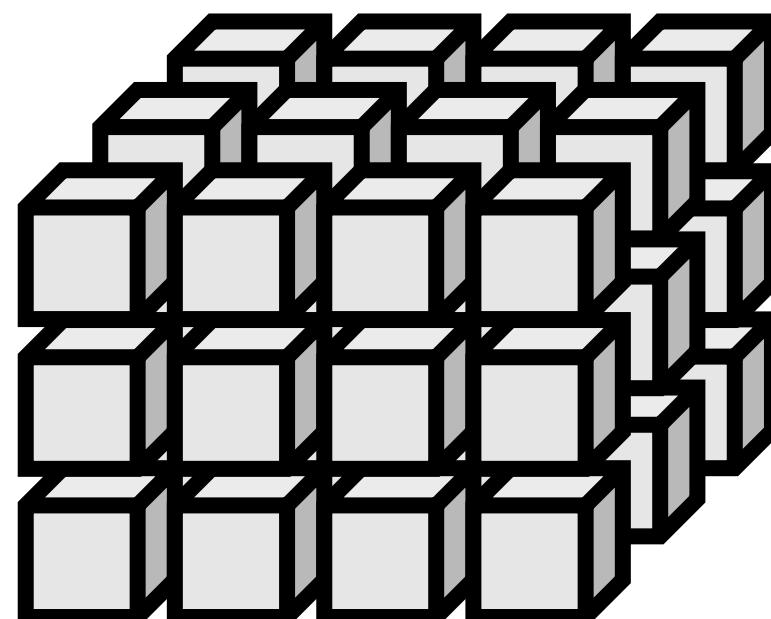
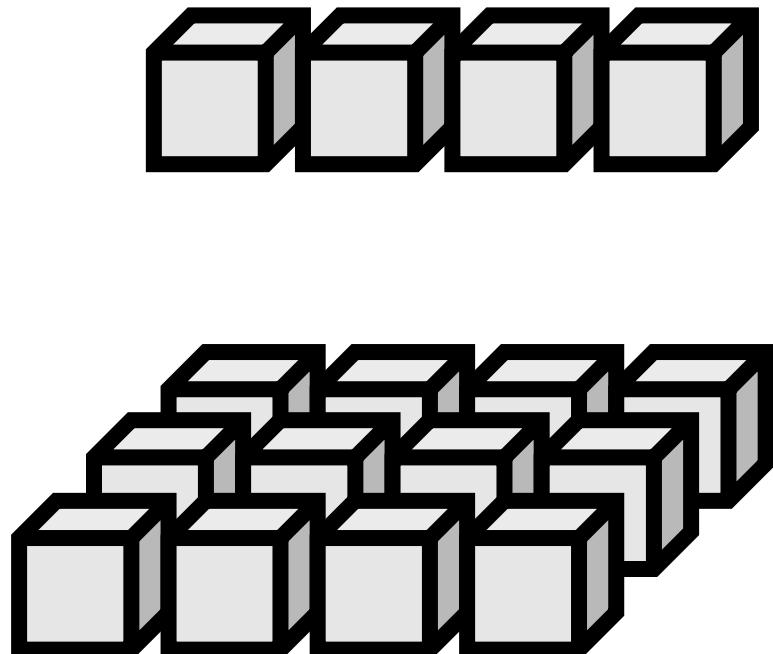
資料最初值: [31, 21, 54, 61, 62, 55, 18, 63]  
排序後的資料 (生冪): [18, 21, 31, 54, 55, 61, 62, 63]  
排序後的資料 (降冪): [63, 62, 61, 55, 54, 31, 21, 18]

## 9-9 多維串列

何謂 1 維串列、2 維串列、3 維串列和多維串列。

前述我們把串列當成可以放資料的盒子，同樣我們先用盒子比喻。

通常同一類型的資料會放在同一維。



回想一下只有一維串列時串列怎麼建立的。

現在看看二維串列，

第一點：串列最外層是中括號「[]」。

第二點：串列的盒子有編號(索引值)。

第三點：可以依找索引值取的資料。

第四點：看到內層括號。

```
1     data = [[ "黃豆", 1289.88, 1280.50],  
2             ["小麥", 700.25, 697.10],  
3             ["玉米", 524.40, 520.38]]  
4     print(data)
```

輸出結果

```
[['黃豆', 1289.88, 1280.5], ['小麥', 700.25, 697.1], ['玉米', 524.4, 520.38]]
```

## 運用索引值取資料

```
1  data = [[ "黃豆", 1289.88, 1280.50],  
2      ["小麥", 700.25, 697.10],  
3      ["玉米", 524.40, 520.38]]  
4  print("原本資料", data)  
5  
6  print(data[0])  
7  print(data[2])  
8  print(data[2][0])  
9  print(data[2][1])
```

## 輸出結果

```
原本資料 [[ '黃豆', 1289.88, 1280.5], ['小麥', 700.25, 697.1], ['玉米', 524.4, 520.38]]  
['黃豆', 1289.88, 1280.5]  
['玉米', 524.4, 520.38]  
玉米  
524.4
```

```
1     data = [["黃豆", 1289.88, 1280.50],  
2             ["小麥", 700.25, 697.10],  
3             ["玉米", 524.40, 520.38]]  
4  
5     for i in data:  
6         print("各筆資料:", i)  
7         print("商品", i[0], "最高價", i[1], "最低價", i[2])
```

輸出結果

各筆資料: ['黃豆', 1289.88, 1280.5]

商品 黃豆 最高價 1289.88 最低價 1280.5

各筆資料: ['小麥', 700.25, 697.1]

商品 小麥 最高價 700.25 最低價 697.1

各筆資料: ['玉米', 524.4, 520.38]

商品 玉米 最高價 524.4 最低價 520.38

# Module 10 :

## 群集-元組(Tuple)

10-1 元組的基本知識

10-1-1 建立元組

10-1-2 取得元組中元素的值

10-1-3 顯示元組長度

10-1-4 從迴圈中操作元組

10-1-5 取得元組中的值出現次數與索引值

10-1-6 元組搭配成員運算子(in、not in)

10-2 元組的連接

10-3 元組的切片

10-4 元組的打包

10-4-1 元組的解包

10-5 元組的統計與排序

10-6 元組不可新增、插入、修改和刪除

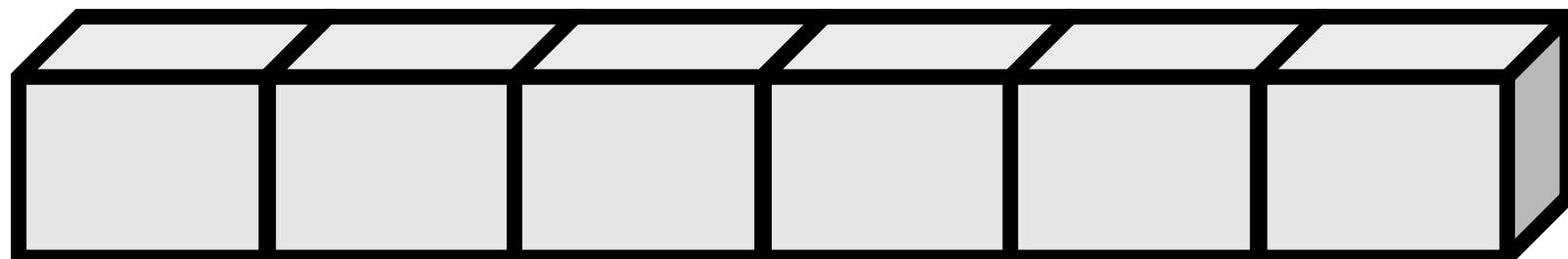
10-6-1 元組和串列之間的差異

10-6-2 元組和串列之間的轉換

## 10-1 元組的基本知識

介紹完串列後接下來將介紹元組(tuple)，元組和串列很像，在學習時可以相互比對。元組和串列不一樣的地方是，元組沒有新增、插入、修改和刪除的功能，而這種特性又稱為不可變的(inmutable)。所以如果需要建立不想修改的資料時可以使用元組。

我們可以先想像元組(tuple)就像一串盒子，每一個盒子中可以存放資料。盒子有從 0 開始的編號，我們可以選擇編號而取得盒子內的資料。盒子的編號又稱為位置、索引。



語法：

元組名稱 = (值1, 值2, ...)

建立元組時，使用小括號「()」，其中每一個盒子的資料是以逗號「，」隔開，接著將它指定給元組名稱，元組名稱也就是變數。

```
1     num = (31, 21, 54, 61, 62)
2     print("num: ", num)
3     print("type(num): ", type(num))
```

輸出結果

```
num: (31, 21, 54, 61, 62)
type(num): <class 'tuple'>
```

```
1     name = ("coffee", "tea", "juice", "milk")
2     print("name: ", name)
3     print("type(name): ", type(name))
```

輸出結果

```
name: ('coffee', 'tea', 'juice', 'milk')
type(name): <class 'tuple'>
```

如果建立的元組只有一個值，最後必須加逗號(,)

```
1     num_1 = (31,)  
2     num_2 = (31)  
3     print("num_1: ", num_1)  
4     print("num_2: ", num_2)  
5     print("type(num_1): ", type(num_1))  
6     print("type(num_2): ", type(num_2))
```

語法：  
元組名稱 = (值1,)

輸出結果

```
num_1: (31,)  
num_2: 31  
type(num_1): <class 'tuple'>  
type(num_2): <class 'int'>
```

如果建立的空元組：

語法：

元組名稱 = ()

語法：

元組名稱 = tuple()

```
1 num_1 = ()  
2 num_2 = tuple()  
3 print("num_1: ", num_1)  
4 print("num_2: ", num_2)  
5 print("type(num_1): ", type(num_1))  
6 print("type(num_2): ", type(num_2))
```

輸出結果

```
num_1: ()  
num_2: ()  
type(num_1): <class 'tuple'>  
type(num_2): <class 'tuple'>
```

取得元組內的值，方式和串列一樣。

注意：

元組取值的時候是也是用中括號「[]」

1	num = (31, 21, 54, 61, 62)	輸出結果	
2			num[0]: 31
3	print("num[0]: ", num[0])		num[2]: 54
4	print("num[2]: ", num[2])		num[3]: 61
5	print("num[3]: ", num[3])		-----
6	print("-" * 10)		num[-1]: 62
7	print("num[-1]: ", num[-1])		num[-3]: 54
8	print("num[-3]: ", num[-3])		num[-5]: 31
9	print("num[-5]: ", num[-5])		

如果想要知道元組的長度，可以使用內建函數 `len()`。

```
1     num = (31, 21, 54, 61, 62)
2     print("length of tuple num: ", len(num))
```

輸出結果

length of tuple num: 5

語法：

`len(元組名稱)`

元組可以搭配迴圈使用。它會依序處理元組的資料。

```
1     num = (31, 21, 54, 61, 62)
2
3     for i in num:
4         print(i)
```

輸出結果

31

21

54

61

62

# 緯 育 TibaMe10-1-5 取得元組中的值出現次數與索引值

可以使用

1. count(元素值) : 查看該元素值共有幾個。
2. index(元素值) : 查看該元素值第一次出現的 index。

語法 :

元組名稱.count(value)

語法 :

元組名稱.index(value)

1	num = (31, 21, 54, 67, 45, 21, 21)	輸出結果
2	print("出現次數:", num.count(21))	出現次數: 3
3	print("出現索引值:", num.index(21))	出現索引值: 1
4		出現次數: 1
5	print("出現次數:", num.count(54))	出現索引值: 2
6	print("出現索引值:", num.index(54))	

# 緯 育 TibaMe 10-1-6 元組搭配成員運算子(in、not in)

in、not in 運算子可以搭配 list、tuple，檢查元素值是否存在。

```
1 num = (31, 21, 54, 61, 62, 55, 18, 63)
2 check_a = 55
3 check_b = 44
4 ans_a_in = check_a in num
5 ans_a_ni = check_a not in num
6 ans_b_in = check_b in num
7 ans_b_ni = check_b not in num
8 print("a_in:", ans_a_in)
9 print("a_ni:", ans_a_ni)
10 print("b_in:", ans_b_in)
11 print("b_ni:", ans_b_ni)
```

輸出結果

```
a_in: True
a_ni: False
b_in: False
b_ni: True
```

```
1 num = [31, 21, 54, 61, 62, 55, 18, 63]
2 check_a = 55
3 check_b = 44
4 ans_a_in = check_a in num
5 ans_a_ni = check_a not in num
6 ans_b_in = check_b in num
7 ans_b_ni = check_b not in num
8 print("a_in:", ans_a_in)
9 print("a_ni:", ans_a_ni)
10 print("b_in:", ans_b_in)
11 print("b_ni:", ans_b_ni)
```

```
a_in: True
a_ni: False
b_in: False
b_ni: True
```

Tuple 和 list 一樣，可以用「+」連接兩個 tuple

```
1 num_1 = (31, 21, 54, 61, 62)
2 num_2 = (55, 18, 63, 72, 74)
3 conn = num_1 + num_2
4 print(conn)
```

輸出結果 (31, 21, 54, 61, 62, 55, 18, 63, 72, 74)

```
1 num_1 = (31, 21, 54, 61, 62)
2 num_2 = (55, 18, 63, 72, 74)
3 num_1 += num_2
4 print(num_1)
```

輸出結果 (31, 21, 54, 61, 62, 55, 18, 63, 72, 74)

Tuple 和 list 一樣，可以使用切片，但僅限於取得資料。

```
1 num = (31, 21, 54, 61, 62, 55, 18, 63)
2 print("1.\"", num[2:7])
3 print("2.\"", num[:7])
4 print("3.\"", num[2:])
5 print("4.\"", num[2:7:2])
6 print("5.\"", num[::-1])
```

輸出結果 1. (54, 61, 62, 55, 18)

2. (31, 21, 54, 61, 62, 55, 18)

3. (54, 61, 62, 55, 18, 63)

4. (54, 62, 18)

5. (63, 18, 55, 62, 61, 54, 21, 31)

Tuple 和 list 一樣，可以使用 zip() 將兩 tuple 打包，  
也可以使用 enumerate() 將索引值打包。

```
1 product_name = ("coffee", "tea", "juice", "milk") 輸出結果 ('coffee', 100)
2 product_price = (100, 90, 80, 70)                      ('tea', 90)
3
4 for i in zip(product_name, product_price):           ('juice', 80)
5     print(i)                                         ('milk', 70)
```

```
1 product_name = ("coffee", "tea", "juice", "milk") 輸出結果 (0, 'coffee')
2
3 for i in enumerate(product_name):                   (1, 'tea')
4     print(i)                                         (2, 'juice')
5
6                                         (3, 'milk')
```

## 10-4-1 元組的解包

```
1 product_name = ("coffee", "tea", "juice", "milk")      輸出結果
2 product_price = (100, 90, 80, 70)
3
4 for i, j in zip(product_name, product_price):
5     print("商品: ", i, "價格: ", j)
```

```
1 product_name = ("coffee", "tea", "juice", "milk")      輸出結果
2 product_price = (100, 90, 80, 70)
3
4 n_p_1, n_p_2, n_p_3, n_p_4, = zip(product_name, product_price)
5 print("1.\t", n_p_1)
6 print("2.\t", n_p_2)
7 print("3.\t", n_p_3)
8 print("4.\t", n_p_4)
```

1. ('coffee', 100)
2. ('tea', 90)
3. ('juice', 80)
4. ('milk', 70)

## 10-5 元組的統計與排序

```
1 tuple_n = (31, 21, 54, 61, 62, 55, 18, 63, 64, 72, 74)
2 print("資料最初值: ", tuple_n)
3 print("資料最大值: ", max(tuple_n))
4 print("資料最小值: ", min(tuple_n))
5 print("資料的總和: ", sum(tuple_n))
6 print("資料的排序: ", sorted(tuple_n))
7 print("資料反向排序: ", sorted(tuple_n, reverse=True))
8
9 print("type(sorted(tuple_n))", type(sorted(tuple_n)))
```

輸出結果

資料最初值: (31, 21, 54, 61, 62, 55, 18, 63, 64, 72, 74)  
資料最大值: 74  
資料最小值: 18  
資料的總和: 575  
資料的排序: [18, 21, 31, 54, 55, 61, 62, 63, 64, 72, 74]  
資料反向排序: [74, 72, 64, 63, 62, 61, 55, 54, 31, 21, 18]  
type(sorted(tuple\_n)) <class 'list'>

內建函數	說明
max()	取出最大值
min()	取出最小值
sum()	算出總和
sorted()	將資料排序

Tuple 排序後的結果會是 list 為什麼!?

# 緯 育 TibaMe 10-6 元組不可新增、插入、修改和刪除

```
1     tuple_n = (31, 21, 54, 61)
2     print("tuple_n[2]: ", tuple_n[2])
3     tuple_n[2] = 40
```

輸出結果

Traceback (most recent call last):

```
File "D:/Python/workspace/Sample_10_18.py", line 3, in <module>
    tuple_n[2] = 40
```

TypeError: 'tuple' object does not support item assignment

介紹了 tuple 和 list ，兩者之間幾乎沒有差異，但是為何需要 tuple ？

1. 將資料存入 tuple ，會比較安全，  
開發大型程式時不會因為程式設計的疏忽，  
導致把原本不想變動的資料更改。
2. 由於 tuple 無法修改，所以設計比 list 簡單，所以執行速度會較優。

也因為 tuple 為了保護資料的內容，所以會改動資料的函數皆不可使用，  
所以 tuple 無法使用 append() 、 insert() 、 delete() 、 pop() 、 remove()... 等等。

除非把 tuple 轉為 list ，才可使用。

使用 `list()`、`tuple()` 做轉換：

1        number_1 = (31, 21, 54, 61, 62)	輸出結果
2        print("初始資料 number_1: ", number_1)	
3        print("初始資料 number_1 型態: ", type(number_1))	
4        print("-" * 30)	
5        number_2 = list(number_1)	
6        print("資料 number_2: ", number_2)	初始資料 number_1: (31, 21, 54, 61, 62)
7        print("資料 number_2 型態: ", type(number_2))	初始資料 number_1 型態: <class 'tuple'>
8        print("-" * 30)	-----
9        number_3 = tuple(number_2)	-----
10      print("資料 number_3: ", number_3)	資料 number_2: [31, 21, 54, 61, 62]
11      print("資料 number_3 型態: ", type(number_3))	資料 number_2 型態: <class 'list'>

1        number_1 = (31, 21, 54, 61, 62)	輸出結果
2        print("初始資料 number_1: ", number_1)	
3        print("初始資料 number_1 型態: ", type(number_1))	
4        print("-" * 30)	
5        number_2 = list(number_1)	
6        print("資料 number_2: ", number_2)	初始資料 number_1: (31, 21, 54, 61, 62)
7        print("資料 number_2 型態: ", type(number_2))	初始資料 number_1 型態: <class 'tuple'>
8        print("-" * 30)	-----
9        number_3 = tuple(number_2)	-----
10      print("資料 number_3: ", number_3)	資料 number_2: [31, 21, 54, 61, 62]
11      print("資料 number_3 型態: ", type(number_3))	資料 number_2 型態: <class 'list'>

# Module 11 :

## 群集-字典(Dictionary)

11-1 字典的基本知識

11-1-1 建立字典

11-1-2 取得字典中元素的值

11-1-3 取得字典的長度

11-1-4 取得字典的相關訊息

11-1-5 從迴圈中操作字典

11-1-6 字典搭配成員運算子(in、not in)

11-1-7 查詢資料為 0 或空值

11-2 字典的操作

11-2-1 修改字典的值

11-2-2 新增字典

11-2-3 更新字典

11-2-4 刪除與清除字典

11-2-5 取出後刪除字典

11-2-6 複製字典

11-3 檢查兩字典是否相等

11-4 字典推導式

處理資料的方法包含串列(list)、元組(tuple)、字典(dictionary)和集合(set)，接下來介紹字典(dictionary)。

字典(dictionary)並不像串列(list)、元組(tuple)，

字典(dictionary)是沒有索引值的。

字典是用鍵值(key)來存放資料，而鍵值(key)是可以自訂名稱，

所以鍵值(key)是沒有順序的，

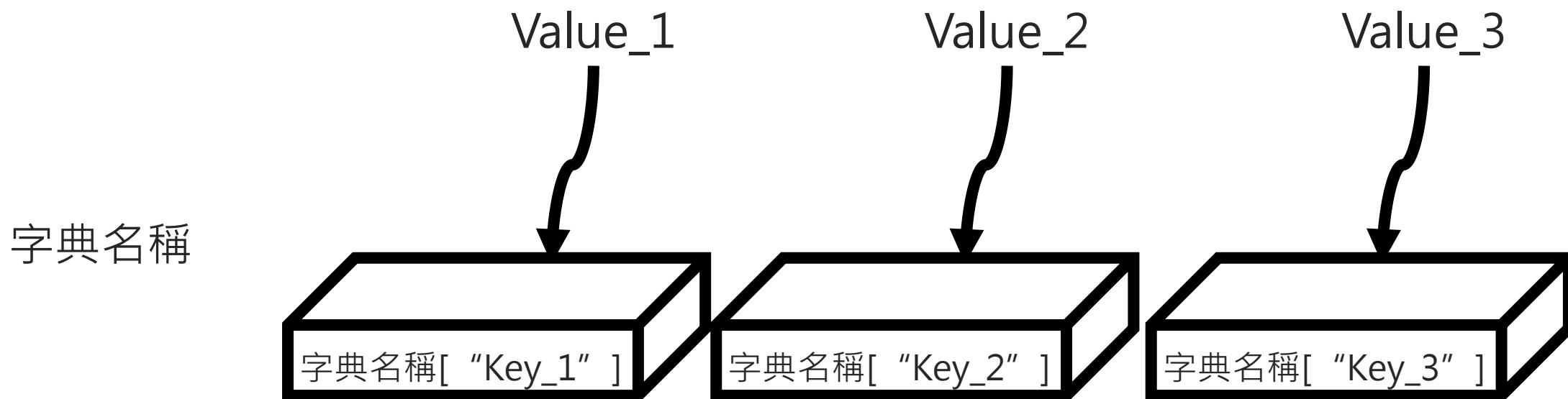
而鍵值(key)內存放的資料又稱為元素值(value)。

字典(dictionary)是你去搜尋什麼鍵值(key)就會得的相對應的元素值(value)，

所以鍵值(key)不可重複，但是元素值(value)可以重複。

字典可以做新增、修改、刪除和更新，而因為鍵值(key)是沒有順序所以沒有插入。

如果我們想像盛裝資料的盒子，盒子的名稱就是字典名稱[ “鍵值”]，裡面的資料就是元素值(value)。



語法：

字典名稱 = {"key 1": value 1, "key 2": value 2, ...}

字典(dictionary)是以大括號「{}」表示，  
每一個鍵值(key)與所對應的元素值(value)是以冒號「:」，  
項目之間是以逗號「，」隔開。

```
1 product = { "coffee": 100, "tea": 90, "juice": 80, "milk": 70}
2 print(product)
3 print(type(product))
```

輸出結果

```
{'coffee': 100, 'tea': 90, 'juice': 80, 'milk': 70}
<class 'dict'>
```

注意：

Value 可以重複。

Key 不可以重複，重複會出錯而且還不會告訴你 error。

```
1 product = {"coffee": 100, "tea": 100, "juice": 80, "juice": 70}
2 print(product)
3 print(type(product))
```

輸出結果

```
{'coffee': 100, 'tea': 100, 'juice': 70}
<class 'dict'>
```

字典(dictionary) 的 value 可以是字串。

```
1     product = {"coffee": "Africa", "tea": "Taiwan", "milk": "New Zealand"}  
2     print(product)
```

輸出結果 { 'coffee': 'Africa', 'tea': 'Taiwan', 'milk': 'New Zealand' }

字典(dictionary) 的 key 可以是數字，原則一樣：「key 不可重複」。

```
1     product = {5: "Africa", 6: "Taiwan", 8: "New Zealand"}  
2     print(product)
```

輸出結果 {5: 'Africa', 6: 'Taiwan', 8: 'New Zealand'}

字典(dictionary) 的 value 也可以放 tuple、list。

如果建立的空字典

語法：

字典名稱 = {}

語法：

字典名稱 = dict()

```
1     dict_a = {}  
2     print("dict_a:", dict_a)  
3     print("type(dict_a): ", type(dict_a))  
4     print("-" * 20)  
5     dict_b = dict()  
6     print("dict_b:", dict_b)  
7     print("type(dict_b): ", type(dict_b))
```

輸出結果

```
dict_a: {}  
type(dict_a): <class 'dict'>  
-----  
dict_b: {}  
type(dict_b): <class 'dict'>
```

語法：

字典名稱[鍵值]

```
1     product = {"coffee": 100, "tea": 90, "juice": 80, "milk": 70}
2     place = {5: "Africa", 6: "Taiwan", 8: "New Zealand"}
3     print("1.\"", product["juice"])
4     print("2.\"", place[6])
```

輸出結果

1. 80
2. Taiwan

如果想要知道字典的長度，可以使用內建函數 `len()`。

```
1     product = {"coffee": 100, "tea": 90, "juice": 80, "milk": 70}
2     place = {5: "Africa", 6: "Taiwan", 8: "New Zealand"}
3     print("1.len(product): ", len(product))
4     print("2.len(place): ", len(place))
```

輸出結果

1. `len(product): 4`
2. `len(place): 3`

如果需要單獨去取得字典內的資訊還有其他的方法，可以取得字典的鍵值(key)或元素值(value)，

1. 逐一取得鍵值，字典名稱.keys()
2. 逐一取得元素值，字典名稱.values()
3. 逐一取得(鍵值, 元素值)的元組，字典名稱.items()

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 product_k = product.keys()
3 product_v = product.values()
4 product_i = product.items()
5 print("1. At", product_k)
6 print("2. At", product_v)
7 print("3. At", product_i)
8 print("-" * 10)
9 for i in product_k:
10     print(i, end="\t")
11 print("\n" + "-" * 10)
12 for i in product_v:
13     print(i, end="\t")
14 print("\n" + "-" * 10)
15 for i in product_i:
16     print(i, end="\t")
```

## 輸出結果

```
1. dict_keys(['coffee', 'tea', 'juice'])
2. dict_values([100, 90, 80])
3. dict_items([('coffee', 100), ('tea', 90), ('juice', 80)])
-----
coffee    tea    juice
-----
100    90    80
-----
('coffee', 100)  ('tea', 90)  ('juice', 80)
```

可以使用 for 迴圈取得字典的鍵值，在依照鍵值取得元素值。

```
1     product = {"coffee": 100, "tea": 90, "juice": 80} 輸出結果 coffee
2
3     for i in product:
4         print(i)
5         print(product[i])
6         print("-" * 10)
```

coffee	100
-----	
tea	90
-----	
juice	80
-----	

# 緯 育 TibaMe 11-1-6 字典搭配成員運算子(in、not in)

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 ans_a = "coffee" in product
3 ans_b = "coffee" not in product
4 ans_c = "water" in product
5 ans_d = "water" not in product
6 print("1.\"", ans_a)
7 print("2.\"", ans_b)
8 print("3.\"", ans_c)
9 print("4.\"", ans_d)
```

輸出結果

1.	True
2.	False
3.	False
4.	True

# 緯 育 TibaMe 11-1-6 字典搭配成員運算子(in、not in)

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2
3 print("飲品名稱 : coffee  tea  juice")
4 order = input("請輸入您想喝的飲品:")
5
6 if order in product:
7     print("您點的飲品為:", order, "價格:", product[order])
8 else:
9     print("很抱歉! 菜單上沒有提供")
```

## 輸出結果

飲品名稱 : coffee      tea      juice

請輸入您想喝的飲品: *tea*

您點的飲品為: tea 價格: 90

飲品名稱 : coffee      tea      juice

請輸入您想喝的飲品: *milk*

很抱歉! 菜單上沒有提供

可使用內建函數查詢 any()、all()。

查詢串列(list)、元組(tuple)和字典(dictionary)中是否有 0 或空值

1. any()：只要有一個值不為 0 或空值，就會是 True
2. all()：所有值皆不為 0 或空值，就會是 True

```
1 product = {"coffee": 100, "tea": 90, "juice": 80} 輸出結果
2 print("1.\"", any(product))
3 print("2.\"", all(product))                                1.  True
4 product = {"coffee": 100, "tea": 90, "": 80}               2.  True
5 print("3.\"", any(product))                               3.  True
6 print("4.\"", all(product))                                4.  False
7 product = {"": 80}                                         5.  False
8 print("5.\"", any(product))                               6.  False
9 print("6.\"", all(product))
```

```
1     number = [31, 21, 54, 61, 62]
2     print("1.\t", any(number))
3     print("2.\t", all(number))
4     number = [31, 21, 54, 61, 0]
5     print("3.\t", any(number))
6     print("4.\t", all(number))
7     number = [0]
8     print("5.\t", any(number))
9     print("6.\t", all(number))
```

輸出結果

1. True
2. True
3. True
4. False
5. False
6. False

```
1     number = (31, 21, 54, 61, 62)
2     print("1.\t", any(number))
3     print("2.\t", all(number))
4     number = (31, 21, 54, 61, 0)
5     print("3.\t", any(number))
6     print("4.\t", all(number))
7     number = (0,)
8     print("5.\t", any(number))
9     print("6.\t", all(number))
```

字典可以做新增、修改和刪除。

字典的新增和修改的語法是一樣的。

語法：

字典名稱[鍵值] = 元素值

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 print("原本資料: ", product)
3 product["coffee"] = 99
4 print("變更後的資料: ", product)
```

輸出結果

原本資料: {'coffee': 100, 'tea': 90, 'juice': 80}

變更後的資料: {'coffee': 99, 'tea': 90, 'juice': 80}

新增字典，就直接將值(value) 指定給它還不存在的鍵值(key)。

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 print("原本資料: ", product)
3 product["milk"] = 70
4 print("新增後的資料: ", product)
```

輸出結果

原本資料: {'coffee': 100, 'tea': 90, 'juice': 80}

新增後的資料: {'coffee': 100, 'tea': 90, 'juice': 80, 'milk': 70}

新增的時候要將想存入的值(value)指定不存在的鍵值(key)，  
修改的時候要將想存入的值(value)指定存在的鍵值(key)。

語法：

字典名稱.update(新增的字典名稱)

備註：

product\_a 更新後會增加，product\_b 資料不變，如果有相同的 key 則 value 會更新。

```
1     product_a = {"coffee": 100, "tea": 90, "juice": 80}
2     product_b = {"milk": 70, "water": 10, "coffee": 99}
3     print("更新前資料: ", product_a)
4     product_a.update(product_b)
5     print("更新後資料: ", product_a)
```

輸出結果 更新前資料: {'coffee': 100, 'tea': 90, 'juice': 80}

更新後資料: {'coffee': 99, 'tea': 90, 'juice': 80, 'milk': 70, 'water': 10}

備註：

串列是用「+」來做資料的連接。

字典沒有「+」做資料的連接，而是使用 update()。

字典的刪除語法。

語法：

**del** 字典名稱[鍵值]

```
1     product = {"coffee": 100, "tea": 90, "juice": 80}
2     print("原本資料: ", product)
3     del product["coffee"]
4     print("刪除後資料: ", product)
```

輸出結果

原本資料: {'coffee': 100, 'tea': 90, 'juice': 80}

刪除後資料: {'tea': 90, 'juice': 80}

字典的清除語法。

語法：

字典名稱.clear()

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 print("原本資料: ", product)
3 product.clear()
4 print("清除後資料: ", product)
```

輸出結果

原本資料: {'coffee': 100, 'tea': 90, 'juice': 80}

刪除後資料: {}

字典和串列一樣可以將值取出使用，同時被取出的資料會刪除。而這種做法又分兩種：

1. `pop(鍵值)`：需要選定鍵值，依照鍵值做取出後刪除。
2. `popitem()`：不需要選定鍵值，由最後一項開始做取出後刪除。

語法：

字典名稱.`pop(鍵值)`

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 print("原本資料: ", product)
3 data = product.pop("coffee")
4 print("data: ", data)
5 print("pop後資料: ", product)
```

輸出結果

```
原本資料: {'coffee': 100, 'tea': 90, 'juice': 80}
data: 100
pop後資料: {'tea': 90, 'juice': 80}
```

語法：

字典名稱.popitem()

```
1 product = {"coffee": 100, "tea": 90, "juice": 80}
2 print("原本資料: ", product)
3 data = product.popitem()
4 print("data: ", data)
5 print("pop後資料: ", product)
```

輸出結果

原本資料: { 'coffee': 100, 'tea': 90, 'juice': 80}

data: ('juice', 80)

pop後資料: { 'coffee': 100, 'tea': 90}

複製新的字典有兩種方式：

1. dict()
2. copy()

```
1 product_a = {"coffee": 100, "tea": 90, "juice": 80}
2 product_b = product_a.copy()
3 product_b["coffee"] = 200
4 print("product_a: ", product_a)
5 print("product_b: ", product_b)
```

輸出結果

```
product_a: {'coffee': 100, 'tea': 90, 'juice': 80}
product_b: {'coffee': 200, 'tea': 90, 'juice': 80}
```

複製新的字典有兩種方式：

1. dict()
2. copy()

```
1 product_a = {"coffee": 100, "tea": 90, "juice": 80}
2 product_b = dict(product_a) ← 只改這裡
3 product_b["coffee"] = 200
4 print("product_a: ", product_a)
5 print("product_b: ", product_b)
```

輸出結果

```
product_a: {'coffee': 100, 'tea': 90, 'juice': 80}
product_b: {'coffee': 200, 'tea': 90, 'juice': 80}
```

判斷兩個字典是否一樣，可以使用「相等運算子( $==$ )」與「不相等運算子( $!=$ )」來檢視，因為字典內的資料沒有順序，所以判斷時也無關順序。

```
1 product_a = {"coffee": 100, "tea": 90, "juice": 80}          輸出結果
2 product_b = {"juice": 80, "coffee": 100, "tea": 90}
3 product_c = {"tea": 90, "juice": 70, "coffee": 100}
4 ans_1 = product_a == product_b
5 ans_2 = product_a != product_b
6 ans_3 = product_a == product_c
7 ans_4 = product_a != product_c
8 print("1.\t", ans_1)
9 print("2.\t", ans_2)
10 print("3.\t", ans_3)
11 print("4.\t", ans_4)
```

- 1. True
- 2. False
- 3. False
- 4. True

還記得串列推導式吧! 字典也有推導式

```
1 product = {"coffee": 100, "tea": 90, "juice": 80, "milk": 70, "water": 10}
2
3     print("產品名稱: ", product)
4
5     new_product_price = {i: int(product[i] * 0.95) if product[i] > 80 else
6                           int(product[i] * 0.8) if 20 < product[i] <= 80 else
7                           int(product[i] * 0.1))
8         for i in product}
9
10    print("----全面商品大優惠----優惠最大是 1 折----")
11    print("打折後價格", new_product_price)
```

輸出結果 產品名稱: {'coffee': 100, 'tea': 90, 'juice': 80, 'milk': 70, 'water': 10}  
----全面商品大優惠----優惠最大是 1 折----

打折後價格 {'coffee': 95, 'tea': 85, 'juice': 64, 'milk': 56, 'water': 1}

# Module 12 :

## 群集-集合(Set)

12-1 集合的基本知識

12-1-1 建立集合

12-1-2 凍結集合

12-1-3 顯示集合長度

12-1-4 從迴圈中操作集合

12-1-5 集合搭配成員運算子(in、not in)

12-1-6 檢查兩集合是否相等

12-2 集合的操作

12-2-2 新增集合

12-2-3 更新集合

12-2-4 刪除集合

12-3 在集合中進行集合運算

12-4 超集合與子集合的關係

處理資料的方法包含串列(list)、元組(tuple)、字典(dictionary)和集合(set)，接下來介紹集合(set)。

集合(set) 中的元素值是沒有順序，而且元素值不可重複。

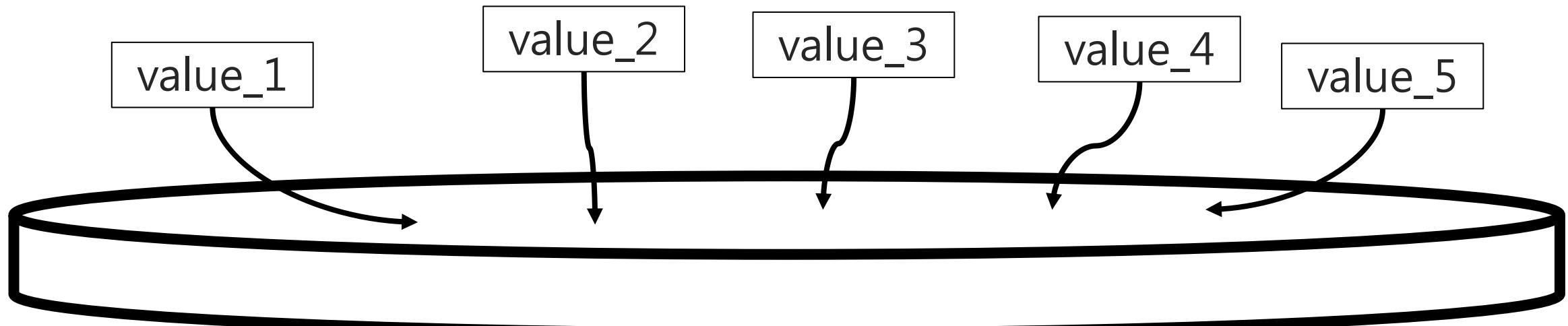
集合(set) 可以做新增、刪除和更新。

集合(set) 的方便在於可以操作集合運算：

聯集(union)、交集(intersection)、差集(difference)和

對稱差集(symmetric difference)。

如果你把集合想像成裝資料的盒子，那麼裡面的資料是沒有順序且不可重複的。



語法：

集合名稱 = {元素值1, 元素值2, ...}

建立集合時，使用大括號「{}」，其中每一個盒子的資料是以逗號「，」隔開。

```
1 name = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"}  
2 print(name)  
3 print(type(name))  
4  
5 print("-" * 30)  
6 company_id = {1001, 1002, 1003, 2003, 2033}  
7 print(company_id)  
8 print(type(company_id))
```

輸出結果

```
{'富富', '星星', '厚厚', '嘉嘉', '喬喬'}  
<class 'set'>  
-----  
{1001, 1002, 1003, 2033, 2003}  
<class 'set'>
```

建立集合也可以使用 `set()` 將串列(list)或元組(tuple)轉為集合(set)。  
同時幫你把重複的值去掉，保留一個。

```
1     name = ["嘉嘉", "喬喬", "厚厚", "富富", "星星", "厚厚", "富富"]
2     print("原本的資料: ", name)
3     print("原本的資料型態: ", type(name))
4
5     name_s = set(name)
6     print("後來的資料: ", name_s)
7     print("後來的資料型態: ", type(name_s))
```

語法：  
集合名稱 = `set([值1, 值2])`

輸出結果

原本的資料: ['嘉嘉', '喬喬', '厚厚', '富富', '星星', '厚厚', '富富']  
原本的資料型態: <class 'list'>  
後來的資料: {'嘉嘉', '喬喬', '厚厚', '富富', '星星'}  
後來的資料型態: <class 'set'>

語法：

集合名稱 = set((值1, 值2))

```
1     name = ("嘉嘉", "喬喬", "厚厚", "富富", "星星", "厚厚", "富富")
2     print("原本的資料: ", name)
3     print("原本的資料型態: ", type(name))
4
5     name_s = set(name)
6     print("後來的資料: ", name_s)
7     print("後來的資料型態: ", type(name_s))
```

輸出結果    原本的資料： ('嘉嘉', '喬喬', '厚厚', '富富', '星星', '厚厚', '富富')  
                原本的資料型態： <class 'tuple'>  
                後來的資料： {'喬喬', '富富', '厚厚', '星星', '嘉嘉'}  
                後來的資料型態： <class 'set'>

如果建立的空集合一定要用set()才能建立，  
如果用 {} 會是字典

語法：

集合名稱 = set()

1	data_a = set()	輸出結果
2	print(data_a)	set()
3	print(type(data_a))	<class 'set'>
4	print("-" * 20)	-----
5	data_b = {}	{}
6	print(data_b)	<class 'dict'>
7	print(type(data_b))	

集合(set)是可以變更的，  
但如果是使用frozenset() (凍結集合)則可以建立不可變更集合。

```
1     name_a = frozenset(("嘉嘉", "喬喬", "厚厚", "富富", "星星"))
2     name_b = frozenset(["嘉嘉", "喬喬", "厚厚", "富富", "星星"])
3     name_c = frozenset({"嘉嘉", "喬喬", "厚厚", "富富", "星星"})
4     print("1.\t", name_a)
5     print("2.\t", name_b)
6     print("3.\t", name_c)
7     print("4.\t", type(name_a))
8     print("5.\t", type(name_b))
9     print("6.\t", type(name_c))
```

輸出結果

集合的執行結果順序不一樣是正常的。

1. frozenset({'喬喬', '富富', '厚厚', '星星', '嘉嘉'})
2. frozenset({'喬喬', '富富', '厚厚', '星星', '嘉嘉'})
3. frozenset({'厚厚', '喬喬', '富富', '星星', '嘉嘉'})
4. <class 'frozenset'>
5. <class 'frozenset'>
6. <class 'frozenset'>

如果想要知道集合的長度，可以使用內建函數 `len()`。

```
1     name = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"}  
2     print("length of name: ", len(name))
```

輸出結果

```
length of name: 5
```

```
1     name = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"}    輸出結果    好久不見! 星星  
2     for i in name:  
3         print("好久不見!" + i)
```

```
1     name = frozenset({"嘉嘉", "喬喬", "厚厚", "富富", "星星"})  
2     for i in name:  
3         print("好久不見!" + i)
```

好久不見! 富富  
好久不見! 嘉嘉  
好久不見! 厚厚  
好久不見! 喬喬

輸出結果  
好久不見! 厚厚  
好久不見! 嘉嘉  
好久不見! 喬喬  
好久不見! 富富  
好久不見! 星星

# 緯 育 TibaMe 12-1-5 集合搭配成員運算子(in、not in)

in、not in 運算子也可以搭配集合(set)使用，查詢資料中的元素值是否存在。

1	name = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"}	
2	ans_a = "幕幕" in name	
3	ans_b = "幕幕" not in name	
4	ans_c = "厚厚" in name	輸出結果
5	ans_d = "厚厚" not in name	
6	print("1.\t", ans_a)	1. False
7	print("2.\t", ans_b)	2. True
8	print("3.\t", ans_c)	3. True
9	print("4.\t", ans_d)	4. False

和字典一樣，判斷兩個集合是否一樣，可以使用「相等運算子( $==$ )」與「不相等運算子( $!=$ )」來檢視，同樣因為集合內的資料沒有順序，所以判斷時也無關順序。

```
1     name_a = {"嘉嘉", "喬喬", "厚厚", "富富", "星星", "望望"}  
2     name_b = {"嘉嘉", "星星", "喬喬", "富富", "厚厚", "望望"}  
3     name_c = {"嘉嘉", "喬喬", "姿姿", "富富", "星星", "望望"}  
4     ans_a = name_a == name_b  
5     ans_b = name_a != name_b                                輸出結果  
6     ans_c = name_a == name_c  
7     ans_d = name_a != name_c  
8     print("1.\t", ans_a)                                     1.  True  
9     print("2.\t", ans_b)                                     2.  False  
10    print("3.\t", ans_c)                                     3.  False  
11    print("4.\t", ans_d)                                     4.  True
```

集合的元素可以操作新增。

集合的新增用：add()

語法：  
集合名稱.add(元素值)

```
1 name_a = {"嘉嘉", "喬喬", "厚厚", "富富"}  
2 print("原本的資料: ", name_a)  
3 name_a.add("姿姿")  
4 print("新增後的資料: ", name_a)
```

輸出結果

原本的資料: {'嘉嘉', '富富', '厚厚', '喬喬'}

新增後的資料: {'嘉嘉', '富富', '厚厚', '姿姿', '喬喬'}

集合(set)和字典(dictionary)可以操作更新，更新後新的資料會加入原本資料。

```
1 name_a = {"嘉嘉", "喬喬", "厚厚", "富富"}  
2 name_b = {"明明", "居居", "忠忠", "喬喬"}  
3 print("原本的資料 name_a : ", name_a)  
4 print("原本的資料 name_b : ", name_b)  
5 name_a.update(name_b)  
6 print("更新後的資料 name_a : ", name_a)  
7 print("更新後的資料 name_b : ", name_b)
```

語法：

集合名稱.update(新增的集合名稱)

輸出結果 原本的資料 name\_a : {'厚厚', '喬喬', '嘉嘉', '富富'}  
原本的資料 name\_b : {'忠忠', '居居', '喬喬', '明明'}  
更新後的資料 name\_a : {'居居', '厚厚', '嘉嘉', '富富', '忠忠', '喬喬', '明明'}  
更新後的資料 name\_b : {'忠忠', '居居', '喬喬', '明明'}

集合的元素可以操作刪除。  
集合的刪除用：remove()

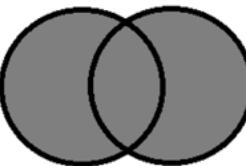
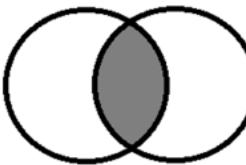
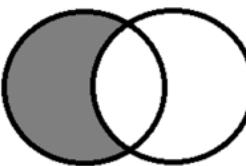
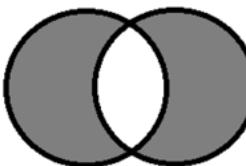
語法：  
集合名稱.remove(元素值)

```
1     name_a = {"嘉嘉", "喬喬", "厚厚", "富富"}  
2     print("原本的資料: ", name_a)  
3  
4     name_a.remove("富富")  
5     print("刪除後的資料: ", name_a)
```

輸出結果

```
原本的資料: {'厚厚', '喬喬', '嘉嘉', '富富'}  
刪除後的資料: {'厚厚', '喬喬', '嘉嘉'}
```

集合可以做集合運算(set operation)。集合運算是集合的最大特點。

名稱	文字意思	圖片意思	運算子	方法
聯集	取兩集合的所有元素			集合.union(集合或其他可疊代物件)
交集	取同時出現兩集合的元素		&	集合.intersection(集合或其他可疊代物件)
差集	取本身有的元素 但扣除掉另外集合出現的元素		-	集合.difference(集合或其他可疊代物件)
對稱差集	取兩個集合中沒有同時出現的元素		^	集合.symmetric_difference(集合或其他可疊代物件)

```
1 name_a = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"}  
2 name_b = {"嘉嘉", "喬喬", "柏柏", "貴貴", "華華"}  
3  
4 print("聯集: ", name_a | name_b)  
5 print("交集: ", name_a & name_b)  
6 print("差集: ", name_a - name_b)  
7 print("對稱差集: ", name_a ^ name_b)
```

輸出結果 聯集: {'富富', '嘉嘉', '柏柏', '星星', '喬喬', '華華', '貴貴', '厚厚'}  
交集: {'嘉嘉', '喬喬'}  
差集: {'厚厚', '星星', '富富'}  
對稱差集: {'厚厚', '華華', '富富', '貴貴', '柏柏', '星星'}

```
1 name_a = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"} # set
2 name_b = {"嘉嘉", "喬喬", "柏柏", "貴貴", "華華"} # set
3 name_c = ("嘉嘉", "喬喬", "柏柏", "貴貴", "華華") # tuple
4 name_d = ["嘉嘉", "喬喬", "柏柏", "貴貴", "華華"] # list
5 name_e = {"嘉嘉": "0912345678", "喬喬": "09",
6         "柏柏": "09", "貴貴": "09", "華華": "09"} # dictionary
7
8 print("聯集: ", name_a.union(name_b))
9 print("交集: ", name_a.intersection(name_c))
10 print("差集: ", name_a.difference(name_d))
11 print("對稱差集: ", name_a.symmetric_difference(name_e))
```

輸出結果

```
聯集: {'厚厚', '柏柏', '星星', '嘉嘉', '華華', '富富', '貴貴', '喬喬'}
交集: {'嘉嘉', '喬喬'}
差集: {'厚厚', '星星', '富富'}
對稱差集: {'華華', '厚厚', '柏柏', '貴貴', '富富', '星星'}
```

附帶一提，判斷是否為可疊代時，可以用 `instance()` 函數協助，不過要先 `from collections import Iterable`。

```
1  from collections import Iterable
2
3  name_a = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"} # set
4  name_b = {"嘉嘉", "喬喬", "柏柏", "貴貴", "華華"} # set
5  name_c = ("嘉嘉", "喬喬", "柏柏", "貴貴", "華華") # tuple
6  name_d = ["嘉嘉", "喬喬", "柏柏", "貴貴", "華華"] # list
7  name_e = {"嘉嘉": "0912345678", "喬喬": "09",
8
9
10
11
12
     "柏柏": "09", "貴貴": "09", "華華": "09"} # dictionary
```

輸出結果

Is set iterable:	True
Is tuple iterable:	True
Is list iterable:	True
Is dictionary iterable:	True

有兩個集合，我們暫且稱它為 A 集合和 B 集合。

如果 B 集合的所有元素都有在 A 集合中出現，

那麼 A 集合就是 B 集合的超集合(superset)，而 B 集合就是 A 集合的子集合。

A\_集合是否為B\_集合的超集合  
語法：

A\_集合.issuperset(B\_集合)

A\_集合是否為B\_集合的子集合  
語法：

A\_集合.issubset(B\_集合)

1	name_a = {"嘉嘉", "喬喬", "厚厚", "富富", "星星"} # set	
2	name_b = {"嘉嘉", "喬喬"} # set	
3		
4	ans_a = name_a.issubset(name_b)	
5	ans_b = name_a.issuperset(name_b)	
6	ans_c = name_b.issubset(name_a)	輸出結果
7	ans_d = name_b.issuperset(name_a)	
8	print("1.\t", ans_a)	1. False
9	print("2.\t", ans_b)	2. True
10	print("3.\t", ans_c)	3. True
11	print("4.\t", ans_d)	4. False

學習了 list、tuple、dictionary 和 set 之後，  
你是對 Python 越來越熟悉還是有點暈頭轉向，  
大數據分析所要面對的資料是各式各樣的。

\*\*\*\*\*

所以請你每個範例都打三遍的同時  
整理

list、tuple、dictionary、set

之間的共同性與差異性，以及各自的特色。

\*\*\*\*\*

在介紹接下來的課程前我們回想看一下  
從最開始學習 Python 到現在，我們學過了：

1. 安裝程式。
2. 顯示在螢幕上、註解、跳脫字元...等等。
3. 字串、整數、浮點數、二進位、八進位、十六進位、布林值...等等。
4. 變數。
5. 各種運算子、條件式。
6. 從鍵盤輸入資料。
7. 條件判斷(if、if...else...、if...elif...else)。
8. 迴圈(for、while loop)、迴圈的巢狀結構、變更迴圈的處理流程(break、continue)。
9. 多筆資料的彙整：串列(list)、元組(tuple)、字典(dictionary)和集合(set)。

我把以上的課程內容歸納為學習程式的第一階段，  
請好好的多練習其中的範例。

如果我們運用以上的知識去撰寫程式，  
那麼一旦要處理的流程越多、越大，程式會動輒上千行，  
就會很不方便去閱讀它那麼更別說是去修改他。

比如說，  
在操作ATM的時候，會有請你輸入密碼的流程，  
接著問你要選擇進入哪一個流程，  
如果要現金提領，它還會有計算餘額...等等的各種流程。

所以如果程式中有某些流程是專門處理特定功能，  
那麼就會把那個特定功能寫成函數。

# Module 13 :

## 函數

13-1 函數的相關知識

13-2 函數的定義與呼叫

13-2-1 函數的定義

13-2-2 函數的呼叫

13-2-3 程式的執行流程

13-2-4 函數可以多次被呼叫

13-2-5 函數可以寫在別處

13-3 函數的引數列表

13-3-1 用引數列表傳遞訊息

13-3-2 呼叫傳遞的引數

13-3-3 在函數中使用多個引數列表

13-3-4 定義引數列表預設值

13-3-5 使用可變長度引數列表

13-4 傳回值的基本知識

13-4-1 沒有傳回值的函數

13-4-2 傳回一個傳回值

13-4-3 傳回多個傳回值

# Module 13 :

## 函數

13-5 變數的使用範圍

13-5-1 變數的總類

13-5-2 了解變數的可使用範圍

13-5-3 使用 `global`

13-5-4 區域變數間的名稱重複

13-5-5 全域變數與區域變數的名稱重複

13-5-6 函數的巢狀結構與使用 `nonlocal`

13-6 變數的生命週期

13-7 與函數相關的其他話題

13-7-1 將函數指定給變數

13-7-2 將函數指定給串列

13-7-3 用 `lambda` 敘述簡單的函數

13-7-4 遞迴

13-7-5 用裝飾器在函數中新增功能

13-7-6 定義生成器

把程式寫成函數的優點：

1. 方便閱讀程式，也就更容易找到要修改的範圍。
2. 一個函數可以被多個程式呼叫，不用每個程式中都有一個區塊都寫一樣的流程。
3. 幫你釋放不必要的記憶體空間，讓程式更有效率的執行。

函數可以寫在同一個「.py」，定義函式的位置要寫在使用函數之上；

函數也可以寫在不同的「.py」，在使用函數前要先「import」函數。

函數之中還可以有函數，如果函數中還有些流程，還是可以再把那些流程寫成函數。

以上先說明關於函數相關知識，接下來我們從最基本的開始學起：

1. 建立函數(定義函數)
2. 使用函數(呼叫函數)

在呼叫函數前，要先定義函數。

語法：

```
def 函數名稱(引數列表):  
    敘述句  
    ...  
    return 傳回值
```

```
1     def function_sample():  
2         print("Do something")
```

輸出結果

定義完函數後，執行結果是空白。  
因為我們還沒有呼叫它。

備註：

1. def 是 define (定義) 的縮些。
2. 函數的名稱和定義變數名稱的規則是一樣的。
3. 接著記住是小括號「()」，不是中括號也不是大括號。
4. 要加冒號「:」。
5. 引數列表可以沒有、一個或多個。接著會教。
6. 如果沒有引數列表，小括號「()」不可省略。
7. 在這函數的管轄範圍內的程式要縮排
8. 傳回值可以沒有、一個或多個。接著會教。
9. 如果沒有傳回值那麼 return 可以省略。
10. 定義函數的引述列表又稱為「參數」。

定義函數後可以呼叫它。

語法：

函數名稱(引數列表)

備註：

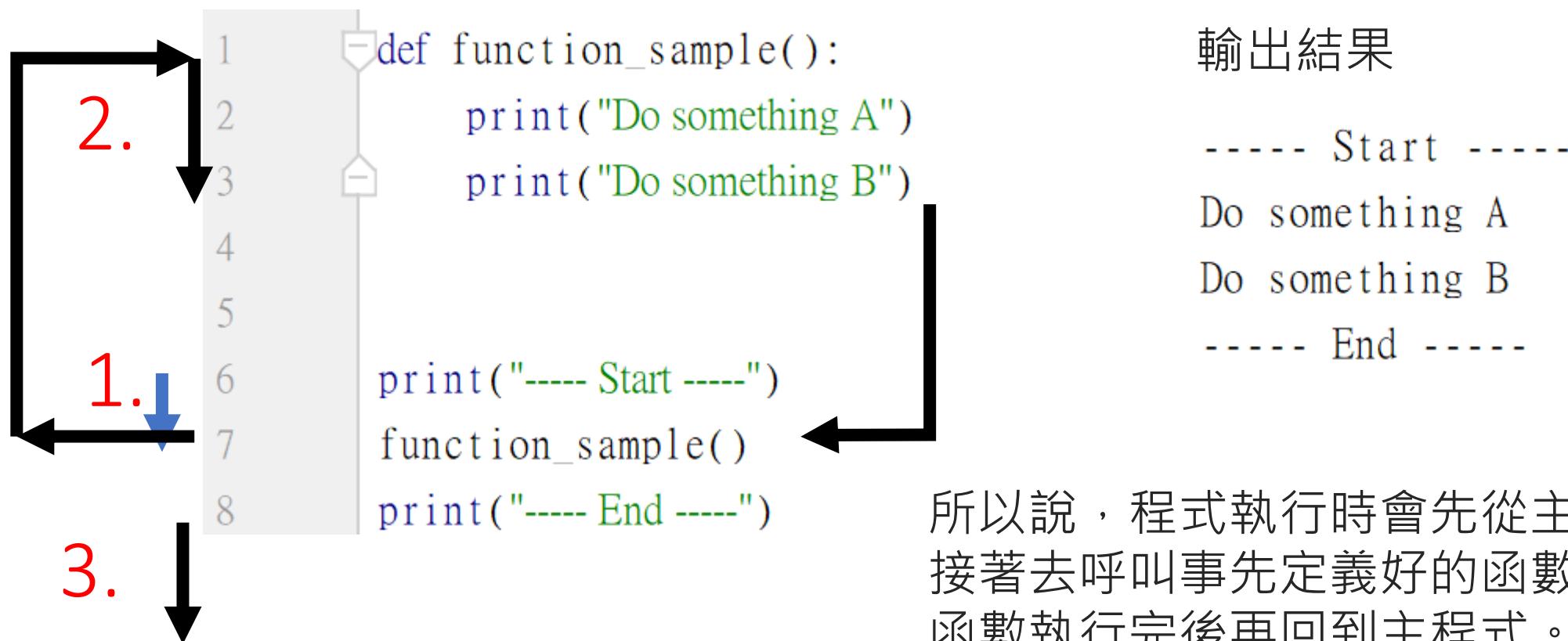
1. 函數名稱是已經定義好的函數名稱。
2. 引數列表可以沒有、一個或多個。接著會教。
3. 呼叫函數的引數列表又稱為「引數」

```
1     def function_sample():
2         print("Do something")
3
4
5     function_sample()
```

輸出結果

Do something

我們撰寫程式的函數，會先定義函式接著再去呼叫(執行)它。那麼程式執行的流程又是如何。



```
1      def function_sample():
2          print("Do something A")
3          print("Do something B")
4
5
6          print("---- Start ----")
7          function_sample()
8          print("-" * 20)
9          function_sample()
10         print("---- End ----")
```

自己嘗試解釋左邊程式執行的流程

輸出結果

----- Start -----

Do something A

Do something B

-----

Do something A

Do something B

----- End -----

The screenshot shows a code editor with two tabs open:

- Sample\_13\_5.py**: Contains the following code:

```
1 def function_sample():
2     print("Do something A")
3     print("Do something B")
```
- Sample\_13\_5\_1.py**: Contains the following code:

```
1 from Sample_13_5 import function_sample
2
3     print("---- Start ----")
4     function_sample()
5     print("-" * 20)
6     function_sample()
7     print("---- End ----")
```

輸出結果

----- Start -----  
Do something A  
Do something B  
-----  
Do something A  
Do something B  
----- End -----

定義函數的「引數列表」稱之為「參數(parameter)」，  
呼叫函數的「引數列表」稱之為「引數(argument)」。

主程式執行的過程中去呼叫函數，有些函數的處理需要主程式提供資料，  
那麼「引數」就是主程式呼叫函數時同時傳給函數的資料，  
定義函數的「參數」就是接收資料的變數。

```
1 def withdraw_money(variable):
2     print("現金提領: ", variable, "元整")
3
4
5 money = 3000
6 withdraw_money(money)
```

輸出結果 現金提領: 3000 元整

這裡我刻意把呼叫函數的「引數」和  
定義函數的「參數」用不同的變數名稱。

如果用一樣的變數名稱也可以，如果用一樣的反而容易搞混。  
因為引數和參數之間不是用變數名稱做資料的傳遞，而是位置順序。

當然也可以把值，直接寫入引數。

```
1 def withdraw_money(variable):
2     print("現金提領: ", variable, "元整")
3
4
5 withdraw_money(3000)
6
7 money = 5000
8 withdraw_money(money)
```

自己嘗試解釋左邊程式執行的流程

輸出結果

現金提領: 3000 元整

現金提領: 5000 元整

引數是用來傳遞資料的，  
所以這些資料也可以是串列(list)、元組(tuple)、字典(dictionary)和集合(set)。

```
1 def withdraw_money(variable):  
2     print("現金提領: ", variable, "元整")  
3  
4  
5     data = [1000, 3000, 5000, 7000]  
6     # data = (1000, 3000, 5000, 7000)  
7     # data = {1000, 3000, 5000, 7000}  
8     # data = {"嘉嘉": 1000, "喬喬": 3000, "富富": 5000}  
9     withdraw_money(data)
```

輸出結果 現金提領: [1000, 3000, 5000, 7000] 元整

引數列表可以有多個，引數之間和參數之間以逗號「，」隔開。

```
1 def withdraw_money(bal, wit):  
2     if bal >= wit:  
3         bal -= wit  
4         print("現金提領: ", wit, "元整")  
5         print("顯示餘額: ", bal, "元整")  
6     else:  
7         print("----- 餘額不足 -----")  
8  
9  
10    withdraw = 2000  
11    balance = 10000  
12    withdraw_money(balance, withdraw)
```

輸出結果  
現金提領: 2000 元整  
顯示餘額: 8000 元整

以下是一個不好的範例：

```
1 def withdraw_money(withdraw, name):  
2     print(name, "現金提領: ", withdraw, "元整")  
3  
4  
5     withdraw = 2000  
6     name = "Philips"  
7     withdraw_money(name, withdraw)
```

輸出結果

2000 現金提領: Philips 元整

「引數」和「參數」之間不是以變數名稱來傳遞，  
而是以位置順序來傳遞，所以要小心。

到目前為止已經學會「引數列表」從沒有、一個到多個，應該對於「引數」和「參數」有基本的認識了。接下來更進一步地介紹有預設值的情形。

函數有預設值的情況的規則：

1. 預設值是放在定義函數的「參數」，不是放在呼叫函數的「引數」中。
2. 定義函數中，沒有預設值的「參數」放在左邊，有預設值的「參數」放在右邊。
3. 定義函數中，沒有預設值的參數是用位置順序取得呼叫函數的「引數」。
4. 定義函數中，有預設值的「參數」可以運用位置順序取得呼叫函數的「引數」，也可以運用「參數名稱=值」的方式指定給參數。

而這種作法又稱為「指定關鍵字引數」。

預先說明完規則後，接下來運用範例協助了解。

開始看以下範例：

習慣規則1. 的範例

```
1     def fee(parking_fee=10, cleaning_fee=5):
2         print("停車費: ", parking_fee)
3         print("清潔費: ", cleaning_fee)
4         print("Total: ", parking_fee + cleaning_fee)
5
6
7     fee()
```

輸出結果

停車費: 10

清潔費: 5

Total: 15

如果參數都有預設值時，  
那麼如果不帶入引數，那麼參數就會使用預設值。

## 習慣規則1. 的範例

```
1  def fee(parking_fee=10, cleaning_fee=5, service_fee=5):      自己嘗試解釋左邊程式執行的流程
2      print("停車費: ", parking_fee)
3      print("清潔費: ", cleaning_fee)
4      print("服務費: ", service_fee)
5      print("Total: ", parking_fee + cleaning_fee + service_fee)
6
7
8  fee()
```

輸出結果

停車費: 10  
清潔費: 5  
服務費: 5  
Total: 20

## 習慣規則1. 2. 3. 的範例

```
1 def fee(a, b, c, parking_fee=10, cleaning_fee=5, service_fee=5):
2     print("fee a: ", a, "\tfee b: ", b, "\tfee c: ", c)
3     print("停車費: ", parking_fee, end="\t")
4     print("清潔費: ", cleaning_fee, end="\t")
5     print("服務費: ", service_fee)
6     print("Total: ", a + b + c + parking_fee + cleaning_fee + service_fee)
7
8
9 fee(10, 20, 30)
```

## 輸出結果

fee a: 10 fee b: 20 fee c: 30

停車費: 10 清潔費: 5 服務費: 5

Total: 80

## 習慣規則1. 2. 3. 的範例

```
1     def fee(a, b, c, parking_fee=10, cleaning_fee=5, service_fee=5):
2         print("fee a: ", a, "\tfee b: ", b, "\tfee c: ", c)
3         print("停車費: ", parking_fee, end="\t")
4         print("清潔費: ", cleaning_fee, end="\t")
5         print("服務費: ", service_fee)
6         print("Total: ", a + b + c + parking_fee + cleaning_fee + service_fee)
7
8
9     fee(10, 20, 30, 40) ← 只改這裡
```

## 輸出結果

```
fee a: 10  fee b: 20  fee c: 30
停車費: 40 清潔費: 5  服務費: 5
Total: 110
```

習慣規則1. 2. 3. 4. 的範例

```
1 def fee(a, b, c, parking_fee=10, cleaning_fee=5, service_fee=5):
2     print("fee a: ", a, "\tfee b: ", b, "\tfee c: ", c)
3     print("停車費: ", parking_fee, end="\t")
4     print("清潔費: ", cleaning_fee, end="\t")
5     print("服務費: ", service_fee)
6     print("Total: ", a + b + c + parking_fee + cleaning_fee + service_fee)
7
8
9 fee(10, 20, 30, cleaning_fee=50) ← 只改這裡
```

輸出結果    fee a: 10    fee b: 20    fee c: 30  
             停車費: 10    清潔費: 50    服務費: 5  
             Total: 125

習慣規則1. 2. 3. 4. 的範例

```
1 def fee(a, b, c, parking_fee=10, cleaning_fee=5, service_fee=5):
2     print("fee a: ", a, "\tfee b: ", b, "\tfee c: ", c)
3     print("停車費: ", parking_fee, end="\t")
4     print("清潔費: ", cleaning_fee, end="\t")
5     print("服務費: ", service_fee)
6     print("Total: ", a + b + c + parking_fee + cleaning_fee + service_fee)
7
8
9 fee(10, 20, 30, cleaning_fee=50, parking_fee=60) ← 只改這裡
```

輸出結果

fee a: 10 fee b: 20 fee c: 30

停車費: 60 清潔費: 50 服務費: 5

Total: 175

目前介紹的「引數列表」都是固定長度的，接下介紹「引數列表」是可變長度的情況。因為定義時不用先決定引數個數的長度，因此稱為可變長度引數(variadic argument)。我們直接範例中解釋語法：

1. 呼叫函數的資料是以逗號「隔開」的不固定長度的引數，可以隨意增減資料。
2. 定義函數的「參數」要加上「\*」。

```
1 def function_sample(*args):  
2     print("初讀進來時的資料: ", args)          備註：  
3     print("初讀進來時的資料型態: ", type(args)) 定義函數讀進來的資料是元組(tuple)，  
4                                         如有需求可以使用 list() 或 set()。  
5  
6     function_sample(31, 21, 54, 61)
```

輸出結果

初讀進來時的資料: (31, 21, 54, 61)  
初讀進來時的資料型態: <class 'tuple'>

注意：

呼叫函數內的引數列表不是元組，只是可變長度引數讀進定義函數的資料會是元組。

可變長度除了讀進來是元組(tuple)，或是可轉為串列(list)和集合(set)之外，另外有一種語法可以讀進來是字典(dictionary)，我們直接範例中解釋語法：

1. 呼叫函數的資料是以逗號「隔開」的不固定長度的引數，可以隨意增減資料。
2. 定義函數的「參數」要加上「\*\*」。

```
1 def function_sample(**args):  
2     print("初讀進來時的資料: ", args)  
3     print("初讀進來時的資料型態: ", type(args))  
4  
5  
6 function_sample(coffee=100, tea=90, juice=80, milk=70)
```

輸出結果 初讀進來時的資料: {'coffee': 100, 'tea': 90, 'juice': 80, 'milk': 70}  
初讀進來時的資料型態: <class 'dict'>

還記得最初學習定義函數的語法時有「傳回值」把！接下來介紹何謂傳回值。

定義函數的語法：

```
def 函數名稱(引數列表):  
    敘述句  
  
    ...  
    return 傳回值
```

為了比較好理解何謂傳回值，我們先想像一個故事：

老闆叫某員工做事情，員工確實有做事情，  
但最後沒有將做事情的結果報告給老闆，這樣的狀況就是沒有傳回值。  
反之，如果員工有將結果報告給老闆，這樣就是有回傳值。

如果把老闆改為主程式，員工改為定義函數，  
回傳值就是定義函數回傳給主程式要的資訊。

在程式語言中，回傳值可以沒有、一個或多個傳回值。

沒有回傳值的範例在介紹「引數列表」之都是使用沒有傳回值的，所以沒有傳回值的情況「return」也可省略。

```
1 def withdraw_money(variable):  
2     print("現金提領: ", variable, "元整")  
3     return  
4  
5 money = 3000  
6 withdraw_money(money)
```

輸出結果

現金提領: 3000 元整

```
1 def withdraw_money(variable):  
2     print("現金提領: ", variable, "元整")  
3  
4 money = 3000  
5 withdraw_money(money)
```

輸出結果

現金提領: 3000 元整

我們直接從範例來介紹只有一個回傳值的情況。

```
1 def turnover(price, number):  
2     tot = price * number  
3     return tot  
4  
5  
6 product_name = "coffee"  
7 product_price = 100  
8 sales_number = 44  
9 total = turnover(product_price, sales_number)  
10 print("產品:", product_name, "營業額:", total, "元")
```

輸出結果

產品: coffee 營業額: 4400 元

```
1 def turnover(price, number):  
2     return price * number
```

另外，簡單的運算可以直接寫在 `return` 後，傳回。

首先還記得程式從哪裡開始吧！  
接著

1. 呼叫函數
2. 帶入「引數」
3. 進入定義函數
4. 接收「參數」
5. 執行運算
6. `return` 傳回值

接著注意看喔！

`return` 傳回值 `tot` 會指定給「變數」 `total`，  
「變數」 `total` 就回到主程式。

當傳回值有多個的情況，`return` 後的每筆資料依序以逗號「，」隔開，傳回主程式的接收變數 `total`、`after_discount` 也是依序以逗號「，」隔開。

```
1 def turnover(price, number, discount):
2     tot = price * number
3     dis_tot = tot * discount
4     return tot, dis_tot
5
6
7 product_name = "coffee"
8 product_price = 100
9 sales_number = 44
10 product_discount = 0.9
11 total, after_discount = turnover(product_price, sales_number, product_discount)
12 print("產品:", product_name, "原價共:", total, "元", "打折後共:", after_discount, "元")
```

輸出結果

產品: coffee 原價共: 4400 元 打折後共: 3960.0 元

關於函數，我們已經學習如何定義函數、呼叫函數、引數列表、回傳值...等等。接下來我們要介紹關於函數內的變數。

主程式和函數中有都有變數，兩者間的變數的差別是：全域變數與區域變數。

全域變數：在定義函數外部的變數。

區域變數：在定義函數內部的變數。函數的參數也是區域變數。

另外還有一個變數稱為「nonlocal」，

他不是全域變數，而他只能在特定的定義函數內的區域執行。

用之前寫過的範例來看，

全域變數包含 product\_name、product\_price、product\_number、  
product\_discount、total、after\_discount

區域變數包含：price、number、discount、tot、dis\_tot

```
def turnover(price, number, discount):  
    tot = price * number  
    dis_tot = tot * discount  
    return tot, dis_tot
```

```
product_name = "coffee"  
product_price = 100  
sales_number = 44  
product_discount = 0.9  
total, after_discount = turnover(product_price, sales_number, product_discount)  
print("產品:", product_name, "原價共:", total, "元", "打折後共:", after_discount, "元")
```

全域變數可以在函數內部和函數外部的範圍使用。

區域變數只能在該函數區域下使用，不可在主程式和其他函數下使用。

```
1      def function_b():
2          var_b = 50
3          print("2.\tvariable_a:", var_a)
4          print("3.\tvariable_b:", var_b)
5
6
7      var_a = 100
8      print("1.\tvariable_a:", var_a)
9      function_b()
10     # print("4.\tvariable_b:", var_b)
```

輸出結果

1. variable\_a: 100
2. variable\_a: 100
3. variable\_b: 50

看左邊的範例，

全域變數是 var\_a，它可以在函數內部和函數外部使用。

區域變數是 var\_b，它只能在該函數區域下使用。

可以將註解打開並看看執行結果。

Traceback (most recent call last):

File "D:/Python/workspace/Sample\_13\_25.py", line 10,

print("4.\tvariable\_b:", var\_b)

NameError: name 'var\_b' is not defined

區域變數只能生活在該區域，無法在其他區域使用，  
所以 var\_b 無法在 function\_c() 使用， var\_c 也無法在 function\_b() 使用。  
可以將註解打開並看看執行結果。

```
1  def function_b():
2      var_b = 50
3      print("4.\tvariable_a:", var_a)
4      print("5.\tvariable_b:", var_b)
5      #print("6.\tvariable_c:", var_c)
6
7
8  def function_c():
9      var_c = 25
10     print("7.\tvariable_a:", var_a)
11     #print("8.\tvariable_b:", var_b)
12     print("9.\tvariable_c:", var_c)
13
14
15     var_a = 100
16     print("1.\tvariable_a:", var_a)
17     function_b()
18     #print("2.\tvariable_b:", var_b)
19     function_c()
20     #print("3.\tvariable_c:", var_c)
```

輸出結果

1. variable\_a: 100
4. variable\_a: 100
5. variable\_b: 50
7. variable\_a: 100
9. variable\_c: 25

了解全域變數和區域變數的使用範圍後，  
如果有需要把區域變數的使用範圍擴大為和全域變數一樣，  
那就是把區域變數變為全域變數。  
可以利用 **global** 來指定變數名稱，讓指定的變數變為全域變數。

```
1 def function_b():
2     global var_b
3     var_b = 50
4     print("2.\tvariable_a:", var_a) 1. variable_a: 100
5     print("3.\tvariable_b:", var_b) 2. variable_a: 100
6
7
8     var_a = 100
9     print("1.\tvariable_a:", var_a)
10    function_b()
11    print("4.\tvariable_b:", var_b)
```

輸出結果

把區域變數 var\_b 變為全域變數後。  
可使用範圍就和全域變數一樣。

如果在不同的定義函數內的區域變數，就算是相同的變數名稱，也是不同的變數。

```
1 def function_1():
2     product_name = "coffee"
3     print("1. 產品名稱: ", product_name)
4
5
6 def function_2():
7     product_name = "tea"
8     print("2. 產品名稱: ", product_name)
9
10
11 function_1()
12 function_2()
```

輸出結果

1. 產品名稱: coffee
2. 產品名稱: tea

在 `function_1` 內的區域變數 `product_name` 可使用範圍就只有在 `function_1` 內。  
同樣的，在 `function_2` 內的區域變數 `product_name` 可使用範圍就只有在 `function_2` 內。

# 緯 育 TibaMe 13-5-5 全域變數與區域變數的名稱重複

全域變數和區域變數的名稱相同時，如果進入函數時，  
變數是以區域變數掌握，出了函數後變數名稱會回到全域變數。

```
1 def function_1():
2     product_name = "coffee"
3     print("2. 產品名稱: ", product_name)
4
5
6     product_name = "tea"
7     print("1. 產品名稱: ", product_name)
8     function_1()
9     print("3. 產品名稱: ", product_name)
```

輸出結果

1. 產品名稱: tea
2. 產品名稱: coffee
3. 產品名稱: tea

# 緯 育 TibaMe 13-5-5 全域變數與區域變數的名稱重複

```
1 def function_1():
2     product_price = 50
3     print("2. 產品價格: ", product_price)          輸出結果
4
5
6     product_price = 100
7     print("1. 產品價格: ", product_price)
8     function_1()
9     product_price -= 10 # product_price = product_price - 10
10    print("3. 產品價格: ", product_price)
```

1. 產品價格: 100
2. 產品價格: 50
3. 產品價格: 90

# 緯 育 TibaMe 13-5-6 函數的巢狀結構與使用 nonlocal

```
1 def function_2():
2     product_name = "coffee"
3     print("3.\t產品名稱: ", product_name)
4
5
6 def function_1():
7     product_price = 50
8     function_2()
9     print("2.\t產品價格: ", product_price)
10
11 product_price = 100
12 print("1.\t產品價格: ", product_price)
13 function_1()
```

介紹 nonlocal 之前先看一下這個範例

如果有一個定義函數內會呼叫函數，那麼他會先做完裡面的函數後再繼續往下做。

輸出結果

1. 產品價格: 100
2. 產品價格: 50
3. 產品名稱: coffee

# 緯 育 TibaMe 13-5-6 函數的巢狀結構與使用 nonlocal

延續上一個範例，也可以把定義函數寫進定義函數中。

```
1 def function_1():
2     product_price = 50
3     def function_2():
4         product_name = "coffee"
5         print("3. 產品名稱: ", product_name)
6         function_2()
7         print("2. 產品價格: ", product_price)
8
9     product_price = 100
10    print("1. 產品價格: ", product_price)
11    function_1()
```

輸出結果

1. 產品價格: 100
2. 產品價格: 50
3. 產品名稱: coffee

# 緯 育 TibaMe 13-5-6 函數的巢狀結構與使用 nonlocal

```
1 def function_1():
2     product_price = 50
3     def function_2():
4         product_name = "coffee"
5         print("3.\t產品名稱: ", product_name)
6         function_2()
7         print("4.\t產品名稱: ", product_name) ← 只改這裡
8         print("2.\t產品價格: ", product_price)
9
10    product_price = 100
11    print("1.\t產品價格: ", product_price)
12    function_1()
```

延續上一個範例，  
product\_name 是 function\_2 的區域變數，  
所以出了 function\_2 後 product\_name 就  
無法使用。

輸出結果

```
File "D:\Python\workspace\Sample_13\Sample_13_31_2.py", line 7,
      print("4.\t產品名稱: ", product_name)
NameError: name 'product_name' is not defined
```

# 緯 育 TibaMe 13-5-6 函數的巢狀結構與使用 nonlocal

```
1 def function_1():
2     product_price = 50
3     product_name = "string"    ← 改這裡
4
5     def function_2():
6         nonlocal product_name    ← 改這裡
7         product_name = "coffee"
8         print("3.\t產品名稱: ", product_name)
9
10    function_2()
11    print("4.\t產品名稱: ", product_name)
12    print("2.\t產品價格: ", product_price)
13
14    product_price = 100
15    print("1.\t產品價格: ", product_price)
16
17    function_1()
```

如果把變數變為 **nonlocal**，  
哪麼出了function\_2 還可以用。

輸出結果

1. 產品價格: 100
2. 產品價格: 50
3. 產品名稱: coffee
4. 產品名稱: coffee

# 緯 育 TibaMe 13-5-6 函數的巢狀結構與使用 nonlocal

```
1 def function_1():
2     product_price = 50
3     product_name = "string"
4     def function_2():
5         nonlocal product_name
6         product_name = "coffee"
7         print("3.\t產品名稱: ", product_name)
8     function_2()
9     print("4.\t產品名稱: ", product_name)
10    print("2.\t產品價格: ", product_price)
11
12    product_price = 100
13    product_name = "ABCD"   ← 改這裡
14    print("1.\t產品價格: ", product_price)
15    function_1()
16    print("5.\t產品名稱: ", product_name)
```

如果再往外一層來看，  
出了 function\_1 後  
product\_name 變數 coffee 又沒有了。

輸出結果

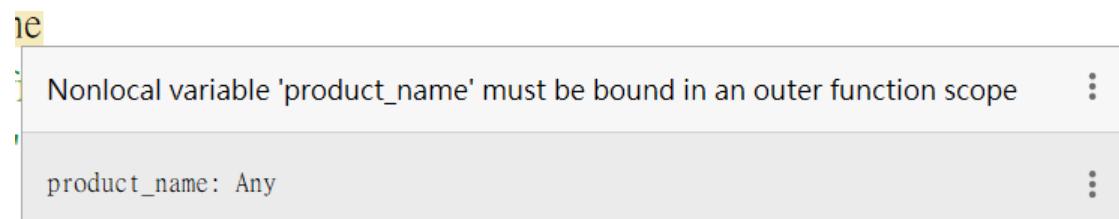
1. 產品價格: 100
3. 產品名稱: coffee
4. 產品名稱: coffee
2. 產品價格: 50
5. 產品名稱: ABCD

← 改這裡

# 緯 育 TibaMe 13-5-6 函數的巢狀結構與使用 nonlocal

```
1 def function_2():
2     nonlocal product_name
3     product_name = "coffee"
4     print("3. 產品名稱:", product_name)
5
6 def function_1():
7     product_price = 50
8     product_name = "string"
9     function_2()
10    print("4. 產品名稱:", product_name)
11    print("2. 產品價格:", product_price)
12
13    product_price = 100
14    print("1. 產品價格:", product_price)
15    function_1()
```

現在我把 function\_2 往 function\_1 外搬，  
把滑鼠移到黃色上面會給我以下提示，  
也就是不可以把 nonlocal 變數移到外層 function\_1 外面，  
要在他的下面。



輸出結果

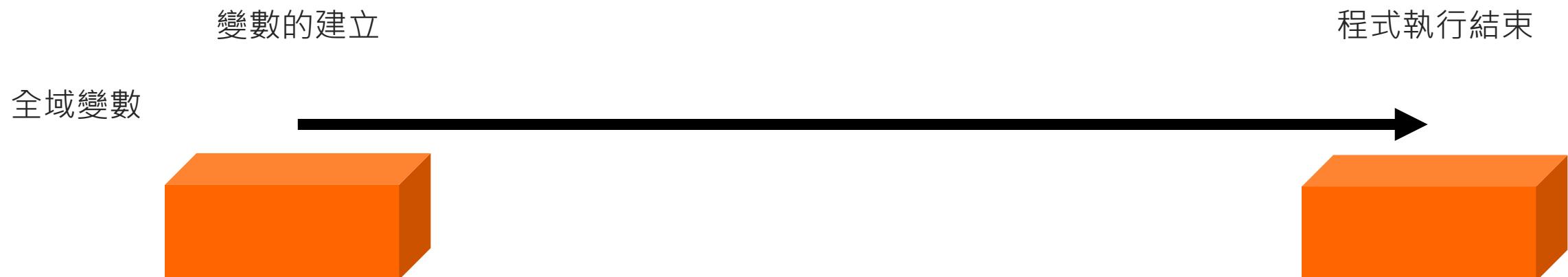
```
File "D:\Python\workspace\Sample_13\Sample_13_31_4.py", line 2
    nonlocal product_name
          ^
SyntaxError: no binding for nonlocal 'product_name' found
```

所以 nonlocal 可以在巢狀函數的  
外層定義函數範圍下使用，  
出了外層定義函數就無法使用。

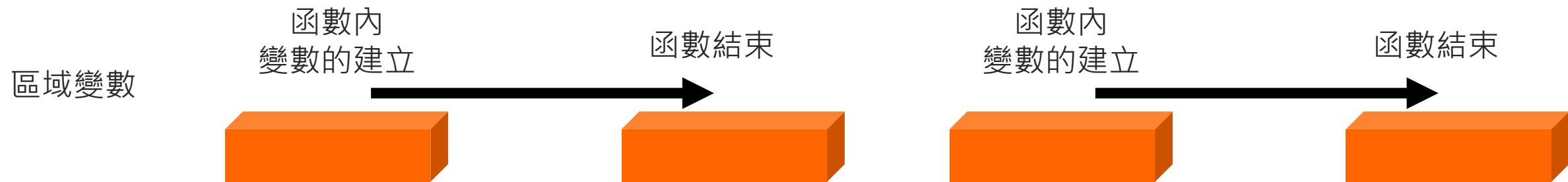
還記得剛學習「變數」以及「函數」的時候曾經說過：

1. 「變數」會在記憶體上要一個空間用來記憶資料，並將記憶的資料取用、運算。
  2. 程式寫成函數的優點，幫你釋放不必要的記憶體空間，讓程式更有效率的執行。
- 另外我們剛學過全域變數和區域變數的可使用範圍，接著談談變數的生命週期。

全域變數的生命週期的開始是從有指定值給變數的時候，一直到程式執行完畢才結束，期間會一直占用記憶體。



區域變數的生命週期的開始是從在函數內有指定值給變數的時候，一直到當下函數執行完畢時將記憶體空間釋放。



同一個目的的程式，執行的速度是勝負的關鍵。  
在寫程式時一定要考慮如何最小程序度的占用記憶體以及適時歸還記憶體空間。

從開始學習函數到這裡以前，  
已經對於函數的相關知識做了介紹。  
接下來談談與函數相關的話題。  
(每個話題彼此獨立)。

之前學過將函數的傳回值指定給變數，那麼如果將函數本人指定給變數的情況又是如何？

```
1 def turnover(price, number, discount):  
2     tot = price * number  
3     dis_tot = tot * discount  
4     return tot, dis_tot  
5  
6  
7     product_name = "coffee"  
8     product_price = 100  
9     sales_number = 44  
10    product_discount = 0.9  
11    t_o = turnover # 將函數指定給變數  
12    print("t_o 的資料型態", type(t_o))  
13    total, after_discount = t_o(product_price, sales_number, product_discount)  
14    print("產品:", product_name, "原價共:", total, "元", "打折後共:", after_discount, "元")
```

輸出結果

t\_o 的資料型態 <class 'function'>  
產品: coffee 原價共: 4400 元 打折後共: 3960.0 元

可以把定義函數指定給變數，接著變數後可以帶入「引數」，進入函數執行，接著回傳傳回值。

```
1 def withdraw_1000():
2     print("提領 1000 元程序")
3 def withdraw_3000():
4     print("提領 3000 元程序")
5 def withdraw_5000():
6     print("提領 5000 元程序")
7
8 withdraw = [withdraw_1000, withdraw_3000, withdraw_5000]
9 print("1. 提領 1000元\n2. 提領 3000元\n3. 提領 5000元\n")
10 # 串列從 0 開始, 所以輸入編號 -1, 如此一來輸入 1 就會取得串列 0 的索引值
11 ans = int(input("請輸入操作編號 (1~3): ")) - 1
12
13 if 0 <= ans <= 2:
14     withdraw[ans]()
15 else:
16     print("輸入錯誤")
```

可以將函數指定給串列，  
成為串列的元素值，  
並享有串列有編號的特性。

### 輸出結果

1. 提領 1000元
2. 提領 3000元
3. 提領 5000元

清輸入操作編號 (1~3): 2  
提領 3000 元程序

還記得在學習運算子時有說過之後會教的 lambda 運算子(lambda operator)吧！

在執行簡單的運算時，可以不決定函數名稱就進行定義，  
這樣的情況可以使用 lambda 運算子，以定義最簡函數。  
另外函數沒有名稱又稱之為無名函數(匿名函數 anonymous function)。

語法：

(lambda 參數 : 表達式)(引數)

```
1     ans = (lambda x: x*0.8)(100)  
2     print("ans:", ans)
```

輸出結果

ans: 80.0

語法：

lambda 參數：表達式

```
1     discount = (lambda x: x*0.8)
2     print(discount(100))
3     print(type(discount))
```

輸出結果      80.0  
                  <class 'function'>

還記得剛剛學過將函數定給變數!?  
左邊是將無名函數指定給變數 discount。

```
1     product_name = "coffee"
2     (lambda x: print(x, "極品"))(product_name)
```

輸出結果      coffee 極品

lambda 可以搭配內建函數 map() 使用。

內建函數 map() 是在引數中指定了「函數」與「能反覆處理的資料」的組合。

```
1 num = [100, 90, 80, 70]
```

語法：  
map(函數, 可疊代物件)

```
3 ans = map(lambda x: x*2, num)
4 print(ans)
5 print(type(ans))
6 print(list(ans))
```

輸出結果

```
<map object at 0x000001F9E6303748>
<class 'map'>
[200, 180, 160, 140]
```

```
1 num = [100, 90, 80, 70]
```

輸出結果 200

```
2
3 for i in map(lambda x: x*2, num):
4     print(i)
```

180

160

140

內建函數 map() 中的函數可以是匿名函數也可以是一般函數。

```
1 def discount(number):
2     print("-" * 10)
3     return number * 0.9
4
5
6 num = [100, 90, 80, 70]
7
8 for i in map(discount, num):print(i)
9
10
11 print(map(discount, num))
12 print(type(map(discount, num)))
```

輸出結果

-----	90.0
-----	81.0
-----	72.0
-----	63.0
<map object at 0x000002B1008D3748>	
<class 'map'>	

map() 能夠指定串列、疊代器等可疊代物件。

```
1 def discount(number):  
2     print("-" * 10)          輸出結果  
3     return number * 0.9  
4  
5  
6     num = [1000, 900, 800]  
7     ite = iter(num)  
8  
9     for i in map(discount, ite):  
10        print(i)
```

-----  
900.0  
-----  
810.0  
-----  
720.0

lambda 可以搭配內建函數 reduce() 使用。

內建函數 reduce() 是在引數中指定了「函數」與「能反覆處理的資料」的組合。

而 reduce() 函數在於資料元素逐一和下一個元素處理，並回傳最後的結果。

語法：

reduce(函數, 可疊代物件)

```
1 from functools import reduce  
2  
3 num = [100, 90, 80]  
4 ans = reduce(lambda x, y: x+y, num)  
5 print("ans: ", ans)
```

輸出結果

ans: 270

```
1 from functools import reduce  
2  
3 num = [100, 90, 80]  
4 ans = reduce(lambda x, y: x*y, num)  
5 print("ans: ", ans)
```

輸出結果

ans: 720000

而 filter() 函數在於能夠協助我們篩選資料。

```
1 num = [100, 90, 80, 70, 60, 50]
2 ans = filter(lambda x: 60 <= x <= 90, num)
3 print("ans: ", ans)
4 print(list(ans))
```

輸出結果      ans: <filter object at 0x000001DF95393908>  
                  [90, 80, 70, 60]

還記得之前用過的 instance() !?

```
1 from collections import Iterable
2 num = [100, 90, 80, 70, 60, 50]
3 ans = filter(lambda x: 60 <= x <= 90, num)
4 print("Is Iterable: ", isinstance(ans, Iterable))
```

輸出結果

Is Iterable: True

假如有一個函數，  
這個函數做出來的資料要在進去同一函數裡繼續做下去，  
一直到一個終止條件達到為止，  
那麼這樣的做法稱為遞迴。

例如計算階層：  
 $5! = 5 * 4 * 3 * 2 * 1$   
他的終止條件是做到 1，  
像類似這樣的運算就很適合做遞迴。

遞迴的方式在程式上較不易理解，所以我們逐步拆解他。

在介紹遞迴前，  
我們先來看看，  
如果一函數中有一個函數，  
之中又有一個函數，  
那麼他的程式會怎麼跑!?

```
1 def fun_3():
2     return 1
3 def fun_2():
4     print("fun_3 start")
5     b = fun_3()
6     print("fun_3 end")
7     return b
8 def fun_1():
9     print("fun_2 start")
10    c = fun_2()
11    print("fun_2 end")
12    return c
13    print("fun_1 start")
14    ans = fun_1()
15    print("fun_1 end")
16    print("ans: ", ans)
```

輸出結果

```
fun_1 start
fun_2 start
fun_3 start
fun_3 end
fun_2 end
fun_1 end
ans: 1
```

~這個範例還不是遞迴，  
只是先幫你順一下邏輯。~

```
1 def fun_3():
2     return 1
3 def fun_2():
4     print("fun_3 start")
5     b = fun_3()
6     print("fun_3 end")
7     print("b:", b, "b * (b + 1):", b * (b + 1)) ←
8     return b * (b + 1) ←
9 def fun_1():
10    print("fun_2 start")
11    c = fun_2()
12    print("fun_2 end")
13    print("c:", c, "c * (c + 1):", c * (c + 1)) ←
14    return c * (c + 1) ←
15    print("fun_1 start")
16    ans = fun_1()
17    print("fun_1 end")
18    print("ans", ans)
```

只改這些

輸出結果

fun\_1 start  
fun\_2 start  
fun\_3 start  
fun\_3 end  
b: 1 b \* (b + 1): 2  
fun\_2 end  
c: 2 c \* (c + 1): 6  
fun\_1 end  
ans 6

~這個範例還不是遞迴，  
只是先幫你順一下邏輯。~

```
1 def fun(n):  
2     if n == 1:  
3         return n  
4     elif n > 1:  
5         return n * fun(n-1)   
6  
7  
8     ans = fun(3)  
9     print("ans: ", ans)
```

輸出結果

ans: 6

為了避免輸入的是負值，所以加入  $n > 1$  的條件式。

```
1 def fun(n):  
2     if n == 1:  
3         return n  
4     elif n > 1:  
5         return n * fun(n-1)   
6  
7  
8     ans = fun(10)  
9     print("ans: ", ans)
```

輸出結果

ans: 3628800

fact(10) =

**3,628,800**

## 作業實作

作業：

請自行找出下列數列的規律性並撰寫函數內容，  
輸出結果如下下方。

1、1、2、3、5、8、13、21、34、55、89.....

```
1  def fun(n):...  
8  
9  
10 data = int(input("請問要做到費氏數列第幾項? "))  
11 for i in range(data):  
12     print(fun(i), end=",")
```

輸出結果

請問要做到費氏數列第幾項? 11

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

在定義函數時，有時會在函數的區塊內再定義函數。

另外，也可以把函數當成是傳回值，來定義要 return 的函數。

或是讓函數來接收「值」，當作引數。

而裝飾器是一個函數，他會接收一個函數做修改，接著再回傳函數。

```
def decorate(function):
    def wapper():
        ...
        return ...
    return wrapper
```

例如，左邊的定義 decorate() 函數中，包含了定義 wapper() 函數的敘述。另外 decorate() 函數將內側的 wapper() 函數當成傳回值返回。

引數列表部分是由 function() 函數來接收。

```
@decorate
def message():
```

上述的函數，可以用「@」來指定，以新增其他函數的功能，這個稱為裝飾器(decorator)。

■ 裝飾器的語法為：

@修飾其他物件並用來新增功能的函數名稱

def 被新增功能的函數名稱：

...

如果我們更進一步來看

```
1  def decorate(func):    輸出結果
2      print(func)
3
4
5      @decorate
6
7      def message_name():
8          pass
```

如果我們更進一步來看

2. 定義  
decorator 函數

```
1 def decorate(func):  
2     print("What is func", func)  
3     def wrapper():  
4         pass  
5     return wrapper  
6  
7  
8     @decorate  
9     def message_name():  
10        pass
```

1. 裝飾器

3. 參數的 function  
本身是個函數

輸出結果

What is func <function message\_name at 0x000002A73DFA9048>

實際上新增功能的是在裝飾器函數的內側定義 wrapper() 函數。  
wrapper() 函数接收到的引數做處理。

```
1  def decorate(func):
2      print("What is func", func)
3      def wrapper(data):
4          print("In wrap:", data)
5          data_m = "-----" + data + "-----\t好喝"
6          return func(data_m)
7      return wrapper
8
9
10 @decorate
11 def message_name(par):
12     print(par)
13
14
15 message_name("coffee")
```

### 輸出結果

```
What is func <function message_name at 0x000001CA35B29048>
In wrap: coffee
-----coffee----- 好喝
```

裝飾器對於 message\_name()函數的內容進行變更，  
使用 wapper() 函数來對他新增功能。

裝飾器可以在不修改函數的情況下新增函數功能。

生成器：python 程式的函數使用形式，可以定義出和疊代器一樣的效果，用於反覆回傳元素。這樣的函數稱為生成器。

生成器回根據 `yield` 敘述，反覆的回傳個元素的內容。回傳的方法與傳回值相似，但不用 `return` 。

```
1 def function(x):  
2     while True:  
3         yield x  
4         x = x + 1  
5  
6  
7     n = function(0)  
8     print("n:", n)
```

如左邊，定義一個生成器取名為 `function` 。

輸出結果

`n: <generator object function at 0x00000187CC28C258>`

如果要取值的話，可以用 `next()` 函數。

在呼叫 `next()` 時才會去計算並取得 `yield` 敘述所指的元素。

```
1 def function(x):  
2     while True:  
3         yield x  
4         x = x + 1
```

輸出結果

```
7 n = function(0)          0  
8 #print("n: ", n)  
9 print(next(n))          1  
10 print(next(n))          2  
11 print(next(n))          3  
12 print(next(n))
```

定義生成器後，可以取得疊代器。

## 作業實作

~每個範例都打三遍後再做作業~

還記得有 BMI 這個作業吧!，請將 BMI 的方法寫成函數，  
過程不拘，但一定要寫成函數，  
左邊的程式碼單純參考，右邊是一定要輸出的結果!

```
1  def calculate_bmi(h, w):...
5
6
7  height = float(input("請輸入身高(cm): "))
8  weight = float(input("請輸入體重(kg): "))
9
10
11 BMI = calculate_bmi(height, weight)
12 print("您的身高為:", height, "體重為:", weight)
13 print("BMI值為", BMI)
```

BMI值計算公式:  $BMI = \frac{\text{體重(公斤)}}{\text{身高}^2(\text{公尺}^2)}$

## 輸出結果

請輸入身高(cm): 174  
請輸入體重(kg): 60  
您的身高為: 174.0 體重為: 60.0  
BMI值為 19.817677368212443

## 作業實作

~每個範例都打三遍後再做作業~

接續上一個練習，把 BMI 評斷的標準寫成函數。

下的程式碼單純參考，右邊是一定要輸出的結果！

體重過瘦  $BMI < 18.5$

體重標準  $18.5 \leq BMI < 24$

體重過重  $24 \leq BMI < 27$

輕度肥胖  $27 \leq BMI < 30$

中度肥胖  $30 \leq BMI < 35$

重度肥胖  $BMI \geq 35$

輸出結果

請輸入身高(cm): 174

請輸入體重(kg): 60

您的身高為: 174.0 體重為: 60.0

BMI 值為 19.817677368212443

體重標準

~每個範例都打三遍後再做作業~

```
1  def calculate_bmi(h, w):  
2      h = h / 100  
3      bmi = w / (h ** 2)  
4      return bmi  
5  
6  
7  def judge_bmi(bmi):...  
8  
9  
10  
11  
12  height = float(input("請輸入身高(cm): "))  
13  weight = float(input("請輸入體重(kg): "))
```

```
26  BMI = calculate_bmi(height, weight)  
27  figure = judge_bmi(BMI)  
28  print("您的身高為:", height, "體重為:", weight)  
29  print("BMI值為", BMI)  
30  print(figure)
```

# Module 14 :

## 類別

14-1 類別的相關知識

14-1-1 定義類別

14-1-2 建立實體物件

14-1-3 使用資料屬性及方法

14-1-4 建立多個實體物件

14-2 建構子的相關知識

14-3 類別變數及類別方法

14-3-1 與類別相關的

資料屬性及類別方法

14-3-2 類別資料屬性的相關知識

14-3-3 類別方法的相關知識

14-3-4 利用類別變數及類別方法

14-4 物件導向的相關知識

14-5 封裝

14-5-1 限制屬性存取-屬性私有化

14-5-2 撰寫 get

14-5-3 撰寫 set

14-5-4 可以指定 setter 與 getter

14-5-6 限制方法存取-方法私有化

# Module 14 :

## 類別

14-6 繼承

14-6-1 繼承的相關知識

14-6-2 衍生類別

14-6-3 多重繼承

14-6-4 衍生類別再衍生

14-6-5 覆寫的相關知識

14-6-6 利用衍生類別

14-7 多型

14-8 與類別相關的主題

14-9 模組

14-9-1 分割檔案

14-9-2 建立模組

14-9-3 匯入模組

14-9-4 使用匯入模組的  
檔案及類別

14-10 模組的應用

14-10-1 匯入模組的命名

14-10-2 用名稱直接匯入

14-10-3 全部匯入

14-10-4 用套件來分類

14-11 利用標準函數庫的模組

隨著程式寫得愈多、愈大也愈複雜，就愈需要更有效率的建立程式。  
而類別(class)主要就是用來整合「資料」和「處理」。

之前我們學過變數，使用變數來儲存各種不同的「資料」，  
也學過串列、元組、字典和集合來使用各種不同的「資料」和「處理」。  
也學過函數，可以整合可重複使用的「處理」。  
而類別主要就是用來整合之前學過的「資料」和「處理」。

在更進一步來說：

類別(class)是以「物件」的概念，並且整合「資料」和「處理」。

那麼什麼又是「物件」!?

「物件」是類別(class)的實例(instance)。

好！我們先從舉例來慢慢說明，

~ 關於「物件」會有近一步的介紹，我們先從程式來了解類別(class)。 ~

類別的敘述中整合了變數、函數等等。

語法：

```
class 類別名稱：  
    def 方法名稱(self, 引數列表):  
        self.變數名稱 = ...  
        return 敘述
```

1. 宣告(定義)類別的建立名稱

2. 表示處理的方法(函數)

3. 表示資料的變數(資料屬性)

類別名稱的命名規則需要和命名變數的規則一樣，但開頭字母習慣大寫。

方法名稱：定義第一個參數為「self」，這個函數稱為方法(method)。

資料屬性：附加「self.」，的變數名稱，這個變數稱之為資料屬性(data attribute)。

```
1 class Drink:  
2     def get_product_name(self, name):  
3         self.name = name  
4         return self.name
```

name 外部變數指定給「self.變數」讓它成為自帶變數，  
表示該物件有了儲存資料的變數。

這裡的 name 是外部帶進來的

而在python 用「self」是用來表示實體物件而慣用的引數名稱。  
(是一種約定俗成的名稱)

```
1 class Drink: ← 1.宣告(定義)類別的建立名稱
2     def get_name(self):
3         return self.name
4
5     def get_price(self):
6         return self.price
```

備註：  
目前我們的 name 和 price 是有黃底，  
原因是我們還沒教到「建構子」，  
目前先了解定義類別及方法。  
~教完建構子就比較完整~

3. 資料屬性

1. 宣告(定義)類別的建立名稱

2. 方法的定義

備註：

目前我們的 name 和 price 是有黃底，  
原因是我們還沒教到「建構子」，  
目前先了解定義類別及方法。  
~教完建構子就比較完整~

要使用定義的類別(class)時，會建立實體物件(instance)。

換句話說，從類別(class)建立的個別物件為實體物件(instance)。

實體物件(instance)就好像依照 Drink 類別來製作每一種飲品的實體一樣。

語法：

實體物件名稱 = 類別名稱()

```
1 class Drink:  
2     def get_name(self):  
3         return self.name  
4  
5     def get_price(self):  
6         return self.price  
7  
8 drink_1 = Drink()
```

變數 drink\_1 即為 Drink 類別建立的實體物件。  
如此一來 drink\_1 這個實體物件可以利用Drink 類別的資料屬性與方法。

接下來，  
實體物件(instance)要怎麼使用類別(class)的方法。

實體物件(instance) 使用類別(class)的方法，會透過句點「.」來設定。

```
1 class Drink:  
2     def get_name(self):  
3         return self.name  
4  
5     def get_price(self):  
6         return self.price  
7  
8 drink_1 = Drink()  
9 drink_1.name = "coffee"  
10 drink_1.get_name()
```

語法：

實體物件名稱.資料屬性名稱  
實體物件名稱.方法名稱(引數列表)

1. 將值指定給資料屬性

2. 呼叫方法

## 14-1-3 使用資料屬性及方法

```
1 class Drink: ← 1. 定義類別(class)
2     def get_name(self):
3         return self.name
4
5     def get_price(self):
6         return self.price
7
8
9 drink_1 = Drink() ← 4. 建立實體物件
10 drink_1.name = "coffee" } ← 5. 將值指定給資料屬性
11 drink_1.price = 100
12
13 n_1 = drink_1.get_name() } ← 6. 呼叫方法
14 p_1 = drink_1.get_price()
15 print(n_1, "價格為", p_1, "元整")
```

輸出結果

coffee 價格為 100 元整

實體物件可以建立多個，  
因此多個「商品也能做好管理」。

```
1 class Drink:  
2     def get_name(self):  
3         return self.name  
4  
5     def get_price(self):  
6         return self.price  
7  
8
```

輸出結果

coffee 價格為 100 元整  
tea 價格為 90 元整

```
9     drink_1 = Drink()  
10    drink_1.name = "coffee"  
11    drink_1.price = 100  
12    n_1 = drink_1.get_name()  
13    p_1 = drink_1.get_price()  
14    print(n_1, "價格為", p_1, "元整")  
15  
16    drink_2 = Drink()  
17    drink_2.name = "tea"  
18    drink_2.price = 90  
19    n_2 = drink_2.get_name()  
20    p_2 = drink_2.get_price()  
21    print(n_2, "價格為", p_2, "元整")
```

另外，如果類別(class)越來越多，或是實體物件越來越多的時候，之前學過的串列、元組、字典和集合，也可以協助我們管理更多的實體物件。

```
1 class Drink:  
2     def get_name(self):  
3         return self.name  
4  
5     def get_price(self):  
6         return self.price  
7  
8  
9     Drink = [Drink(), Drink()] # 放不同的class  
10
```

輸出結果

coffee 價格為 100 元整  
tea 價格為 90 元整

```
11 drink_1 = Drink[0]  
12 drink_1.name = "coffee"  
13 drink_1.price = 100  
14 n_1 = drink_1.get_name()  
15 p_1 = drink_1.get_price()  
16 print(n_1, "價格為", p_1, "元整")  
17  
18 drink_2 = Drink[1]  
19 drink_2.name = "tea"  
20 drink_2.price = 90  
21 n_2 = drink_2.get_name()  
22 p_2 = drink_2.get_price()  
23 print(n_2, "價格為", p_2, "元整")
```

建立實體物件(instance)時，有時會想在一開始時就執行各種處理。  
像是在一開始時就將初始值指定給實體物件(instance)，  
或是在指定實體物件(instance)時同時確認是否正確初始化。

而建構子就是在建立實體物件(instance)時一定要先執行的方法。

建構子語法：

```
def __init__(self, 引數列表):  
    ...
```

備註：init 前後各有 2 個半形的下底線組成。

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.price = price  
5  
6     def get_name(self):  
7         return self.name  
8  
9     def get_price(self):  
10        return self.price  
11  
12 drink_1 = Drink("coffee", 100)  
13 n_1 = drink_1.get_name()  
14 p_1 = drink_1.get_price()  
15 print(n_1, "價格為", p_1, "元整")
```

輸出結果

coffee 價格為 100 元整

第 13. 行

drink\_1 是由 Drink 這個類別建立的實體物件也就是實例，而 Drink 這個類別的屬性有兩個資料分別為 self.name 和 self.price，在這個實例 drink\_1 的屬性也就是 coffee 和 100。

接著實體物件使用物件的方法，也就是第 14, 15 行。

```
1 class Drink:  
2     def __init__(self, name, price, discount):  
3         self.name = name  
4         self.price = price  
5         self.discount = discount  
6  
7     def get_name(self):  
8         return self.name  
9  
10    def get_price(self):  
11        return self.price, self.price * self.discount  
12  
13  
14 drink_1 = Drink("coffee", 100, 0.9)  
15 n_1 = drink_1.get_name()  
16 p_1, p_d_1 = drink_1.get_price()  
17 print(n_1, "原價為", p_1, "元整\n特價後", p_d_1, "元整")
```

輸出結果

coffee 原價為 100 元整 特價後 90.0 元整

# 緯 育 TibaMe 14-3-1 與類別相關的資料屬性及類別方法

到目前為止，所學習的類別(class)都是與個別實體物件(instance)相關聯。相對的，有時候需要對於全體實體物件(instance)相關聯，這時候如果有全體類別共有的資料屬性或方法就比較方便。因此將學習與類別相關的資料屬性及類別方法。

首先，我們先來了解實體變數(instance variable)與類別變數(class variable)的差別：從之前了範例中，存在於每個實體物件(instance)之中，前面會附加「self」的資料屬性，它稱為實體變數(instance variable)。另外，在類別中所定義的資料屬性，在類別中只會存在一個值，它稱為類別變數(class variable)。

```
1 class Drink:  
2     count = 0 ← 類別變數，  
3  
4     def __init__(self, name, price, discount):...  
5  
6     def get_name(self):  
7         return self.name ← 實體變數，  
8  
9     def get_price(self):...  
10
```

屬於類別的值的資料屬性  
每個實體物件各自有的值的資料屬性

類別方法(class method)，與類別相關聯的方法。

類別方法的定義要在類別定義之中，@classmethod 敘述的下方。

```
1  class Drink:  
2      count = 0  
3  
4      def __init__(self, name, price, discount):...  
5  
6      def get_name(self):...  
7  
8      def get_price(self):...  
9  
10     @classmethod  
11     def get_count(cls):  
12         return cls.count  
13  
14  
15  
16  
17
```

「cls」是為了要接收類別名稱的引數。  
這裡使用代表類別名稱的引數cls，  
來返回類別的資料屬性。

類別方法

類別變數與類別方法的實際操作範例如下：

```
1 class Drink:  
2     count = 0  
3  
4     def __init__(self, name, price, discount):  
5         Drink.count = Drink.count + 1  
6         self.name = name  
7         self.price = price  
8         self.discount = discount  
9  
10    def get_name(self):...  
11  
12    def get_price(self):...  
13  
14  
15    @classmethod  
16    def get_count(cls):  
17        return cls.count  
18
```

有加這行

```
21     drink_1 = Drink("coffee", 100, 0.9)  
22     n_1 = drink_1.get_name()  
23     p_1, p_d_1 = drink_1.get_price()  
24     print(n_1, "原價為", p_1, "元整\t特價後", p_d_1, "元整")  
25  
26     drink_2 = Drink("tea", 80, 0.8)  
27     n_2 = drink_2.get_name()  
28     p_2, p_d_2 = drink_2.get_price()  
29     print(n_2, "原價為", p_2, "元整\t特價後", p_d_2, "元整")  
30  
31     print("合計銷售:", Drink.get_count(), "杯")
```

輸出結果

coffee 原價為 100 元整 特價後 90.0 元整  
tea 原價為 80 元整 特價後 64.0 元整  
合計銷售： 2 杯

```
1 class Drink:  
2     count = 0  
3     total_price = 0  
4  
5     def __init__(self, name, price, discount):  
6         Drink.count = Drink.count + 1  
7         Drink.total_price = Drink.total_price + price * discount  
8         self.name = name  
9         self.price = price  
10        self.discount = discount  
11  
12    def get_name(self):...  
14  
15    def get_price(self):...  
17  
18    @classmethod  
19    def get_count(cls):  
20        return cls.count
```



含下頁

```
22     @classmethod  
23     def get_total(cls):  
24         return cls.total_price  
25  
26  
27     drink_1 = Drink("coffee", 100, 0.9)  
28     n_1 = drink_1.get_name()  
29     p_1, p_d_1 = drink_1.get_price()  
30     print(n_1, "原價為", p_1, "元整\n特價後", p_d_1, "元整")  
31  
32     drink_2 = Drink("tea", 80, 0.8)  
33     n_2 = drink_2.get_name()  
34     p_2, p_d_2 = drink_2.get_price()  
35     print(n_2, "原價為", p_2, "元整\n特價後", p_d_2, "元整")  
36  
37     print("合計銷售:", Drink.get_count(), "杯")  
38     print("營業額:", Drink.get_total(), "元")
```

## 輸出結果

coffee 原價為 100 元整 特價後 90.0 元整  
tea 原價為 80 元整 特價後 64.0 元整  
合計銷售: 2 杯  
營業額: 154.0 元

我們對於類別有了初步的認識，那麼來談談何謂物件導向的相關介紹。

如果程式語言是可以使用「物件」的概念來做程式設計，那麼這樣的程式就稱為「物件導向程式語言」，Python 就屬於其中。

如果是以「物件」的概念來做程式設計，那麼這樣就稱為「物件導向程式設計」。

這裡做幾個比喻來了解。

假如我要把車子當作比喻，

那麼車子有分很多種也就是可以有不同的類別(class)，

有卡車、休旅車、轎車、跑車、摩托車...等等，

接著我把卡車當成其中一種類別並將她做成實體物件(當然其他的車也可以當物件)，

描述卡車這個物件的屬性，有車牌、載重、廠牌、年分、型號...等等的資訊，

描述卡車這個物件的方法，載貨。

假如我用商品當作比喻，

那麼商品有分很多種，有麵包、水果...等等，

接著我把麵包當成物件，

描述麵包這個物件的屬性，重量、生產時間、價格...等等，

描述麵包這個物件的方法，計算賞味期限、計算打折...等等。

接著再進一步來說，什麼是「物件」。

物件可以用「屬性」與「方法」來描述，

而「屬性」也就是資料，「方法」也就是他的作用。

所以在程式中「屬性」就是變數，「方法」就是函數。

而類別 (class) 就是一群有共同特徵的「物件」。

那麼「物件導向程式語言」有什麼特性：

1. 封裝
2. 繼承
3. 多型

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.price = price  
5  
6     def get_price(self):  
7         return self.price  
8  
9  
10    drink_1 = Drink("coffee", 100)  
11    print("1.\t", drink_1.price)  
12    drink_1.price = 10  
13    print("2.\t", drink_1.price)  
14    print("3.\t", drink_1.get_price())
```

到目前為止，我們教過的範例範例程式如下，也就是建立類別、建構子、建立實體物件...等等。

接著我們發現一個情況，就是容易讀到類別內的資料，也容易寫入資料。

### 輸出結果

1. 100
2. 10
3. 10

接著如想要限制資料屬性的存取，我們可以在想要限制存取的屬性名稱前面加上兩個半形下底線「\_」來命名。

而這種方式是做名詞上的修改。

但是，在初始化時資料有進去，但要取的時候又取不出來。

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.__price = price  
5         print("In class price:", self.__price)  
6  
7 drink_1 = Drink("coffee", 100)  
8 print(drink_1.price)
```

輸出結果

```
Traceback (most recent call last):  
In class price: 100  
  File "D:/Python/workspace/Sample_14_14.py", line 9, in  
    print(drink_1.price)  
AttributeError: 'Drink' object has no attribute 'price'
```

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.__price = price  
5         print("In class price:", self.__price)  
6  
7 drink_1 = Drink("coffee", 100)  
8 print(drink_1.__price)
```

輸出結果

```
File "D:/Python/workspace/Sample_14_15.py", line 9, in <  
    print(drink_1.__price)  
AttributeError: 'Drink' object has no attribute '__price'
```

那麼就寫一個能讓我先看得到資料屬性的方法。

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.__price = price  
5         print("In class price:", self.__price)  
6  
7     def get_price(self):  
8         print("---- get price ----")  
9         return self.__price  
10  
11 drink_1 = Drink("coffee", 100)  
12 print("drink_1.get_price(): ", drink_1.get_price())
```

輸出結果

In class price: 100  
---- get price ----  
drink\_1.get\_price(): 100

```
1  class Drink:  
2      def __init__(self, name, price):  
3          self.name = name  
4          self.__price = price  
5          print("In class price:", self.__price)  
6  
7      def get_price(self):  
8          print("---- get price ----")  
9          return self.__price  
10  
11  
12  drink_1 = Drink("coffee", 100)  
13  print("drink_1.get_price(): ", drink_1.get_price())  
14  drink_1.__price = 200  
15  drink_1.price = 200  
16  print("1.\t", drink_1.__price)  
17  print("2.\t", drink_1.price)  
18  print("3.\tdrink_1.get_price(): ", drink_1.get_price())
```

剛剛讀的時候可以，那麼現在我想要寫入，所以我就做了異想天開的寫入方式。

輸出結果

```
In class price: 100  
----- get price -----  
drink_1.get_price(): 100
```

```
1.   200  
2.   200  
----- get price -----  
3.   drink_1.get_price(): 100
```

我們發現最後讀資料屬性時沒有改變。

```
1 class Drink:  
2     def __init__(self, name, price):...  
3  
4     def get_price(self):  
5         print("---- get price ----")  
6         return self.__price  
7  
8     def set_price(self, price):  
9         print("---- set price ----")  
10        self.__price = price  
11  
12 drink_1 = Drink("coffee", 100)  
13 print("1.\tdrink_1.get_price(): ", drink_1.get_price())  
14 drink_1.set_price(200)  
15 print("2.\tdrink_1.get_price(): ", drink_1.get_price())
```

所以我就需要做一個寫入的方法

輸出結果

In class price: 100

----- get price -----

1. drink\_1.get\_price(): 100

----- set price -----

----- get price -----

2. drink\_1.get\_price(): 200

如此一來，  
我們就能夠比較安全的讀取與寫入。  
比較不會輕易的不小心改到資料屬性，  
比較需要思考的去改它。

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.__price = price  
5         print("In class price:", self.__price)  
6  
7     @property  
8     def price_manage(self):  
9         print("---- get price ----")  
10        return self.__price  
11  
12    @price_manage.setter  
13    def price_manage(self, price):  
14        print("---- set price ----")  
15        self.__price = price
```

```
18     drink_1 = Drink("coffee", 100)  
19     print("1.\tdrink_1.get_price(): ", drink_1.price_manage)  
20     drink_1.price_manage = 200  
21     print("2.\tdrink_1.get_price(): ", drink_1.price_manage)
```

首先我們需要內建函數 `property()`，  
接著寫能夠讓我取得資料屬性的方法。

輸出結果 In class price: 100

----- get price -----

1. drink\_1.get\_price(): 100

----- set price -----

----- get price -----

2. drink\_1.get\_price(): 200

其中的 `price_manage` 是自己的命名。  
(如果要寫成 `price` 的話也是可, 但就像是繞一大圈)

在類別中可以讓函數(方法)，僅限於類在別中使用，類別外部無法使用。  
要定義函數(方法)的私有化僅須在函數名稱前加入兩個下底線。

語法：

```
def __函數名稱(self):  
    ...
```

```
1 class Drink:  
2     def __init__(self, name, price, discount):  
3         self.name = name  
4         self.price = price  
5         self.discount = discount  
6  
7     def __calculate_discount(self):  
8         return self.price * self.discount  
9  
10    def get_name(self):  
11        return self.name  
12  
13    def get_price(self):  
14        return self.price, self.__calculate_discount()
```

```
17     drink_1 = Drink("coffee", 100, 0.9)
18     n_1 = drink_1.get_name()
19     p_1, p_d_1 = drink_1.get_price()
20     print(n_1, "原價為", p_1, "元整\n特價後", p_d_1, "元整")
21     t_1 = drink_1.__calculate_discount()
```

輸出結果

```
Traceback (most recent call last):
  File "D:\Python\workspace\Sample_14\Sample_14_17_1.py", line 21, in <module>
    t_1 = drink_1.__calculate_discount()
AttributeError: 'Drink' object has no attribute '__calculate_discount'
```

把 21 註  
解：

```
17     drink_1 = Drink("coffee", 100, 0.9)
18     n_1 = drink_1.get_name()
19     p_1, p_d_1 = drink_1.get_price()
20     print(n_1, "原價為", p_1, "元整\n特價後", p_d_1, "元整")
21     #t_1 = drink_1.__calculate_discount()
```

輸出結果

coffee 原價為 100 元整 特價後 90.0 元整

隨著程式越多，

早先寫的許多程式用類別(class)做出了實例(instance)可以用得很好，  
但新開發的程式想用原本的類別，但又有新的方法想要加入，  
或是讓原本的方法的功能提升。

這時會發現一個問題，

就是如我改動類別那麼之前寫過的程式中的實例可能都有所變動。

所以我們可以將原本的類別當成基礎類別(base class)，

想要提升原本方法或是新增方法的做成衍生類別(derived class)。

如此一來，就可以不用更動原本的類別，

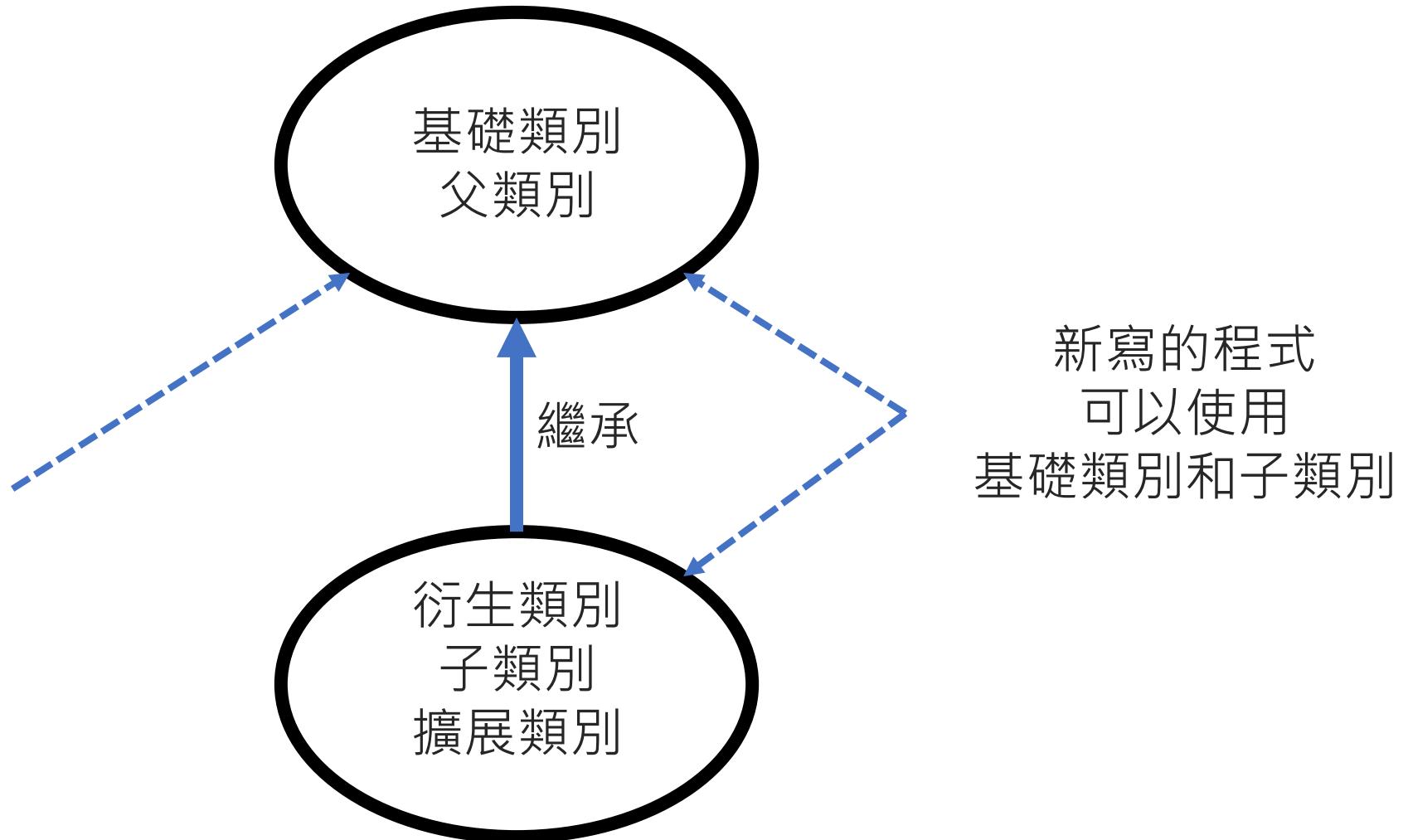
早先寫的許多程式用基礎類別，新寫的程式可以用衍生類別。

另外，基礎類別(base class)又稱為父類別(parent class)，  
衍生類別(derived class) 又稱為子類別(child class)，

還有，  
衍生類別是擴展出來的，所以也被稱為擴展類別(extends)。  
衍生類別是繼承基礎類別的，所以這樣的方式又稱為繼承(inherit)。

## 14-6-1 繼承的相關知識

原本寫的許多程式  
可以使用  
基礎類別



衍生類別要繼承基礎類別的語法

衍生語法：

```
class 衍生類別名稱(基礎類別名稱):  
    ...
```

部分的程式碼如左邊：

基礎類別的部分之前說過，  
衍生類別的部分，

1. 定義衍生類別名稱(這裡用 OriginDrink)
2. 繼承的基礎類別源自於(Drink)
3. 定義衍生類別的建構子
4. 部分衍生類別的資料屬性來自於基礎類別  
這時要使用「super().基礎類別的資料屬性」。  
除了 super() 以外，還可以用基礎類別的類別名稱。

```
1  class Drink:...  
16  
17  
18  class OriginDrink(Drink):  
19      def __init__(self, name, price, country, variety):  
20          super().__init__(name, price)  
21          # Drink.__init__(self, name, price)  
22          self.country = country  
23          self.variety = variety
```

```
1  class Drink:  
2      def __init__(self, name, price):  
3          self.name = name  
4          self.__price = price  
5  
6      def get_name(self):  
7          return self.name  
8  
9      @property  
10     def price_manage(self):  
11         return self.__price  
12  
13     @price_manage.setter  
14     def price_manage(self, price):  
15         self.__price = price
```

完整程式碼如下：

```
18     class OriginDrink(Drink):  
19         def __init__(self, name, price, country, variety):  
20             super().__init__(name, price)  
21             # Drink.__init__(self, name, price)  
22             self.country = country  
23             self.variety = variety  
24  
25         def get_country(self):  
26             return "產地：" + self.country  
27  
28         def get_variety(self):  
29             return "品種：" + self.variety
```



```
32     drink = OriginDrink("coffee", 100, "Blue Mountain", "Arabica")
33     d_n = drink.get_name()
34     d_p = drink.price_manage
35     d_c = drink.get_country()
36     d_v = drink.get_variety()
37     print(d_v, d_c, d_n, "價格:", d_p)
```

輸出結果

品種: Arabica 產地: Blue Mountain coffee 價格: 100

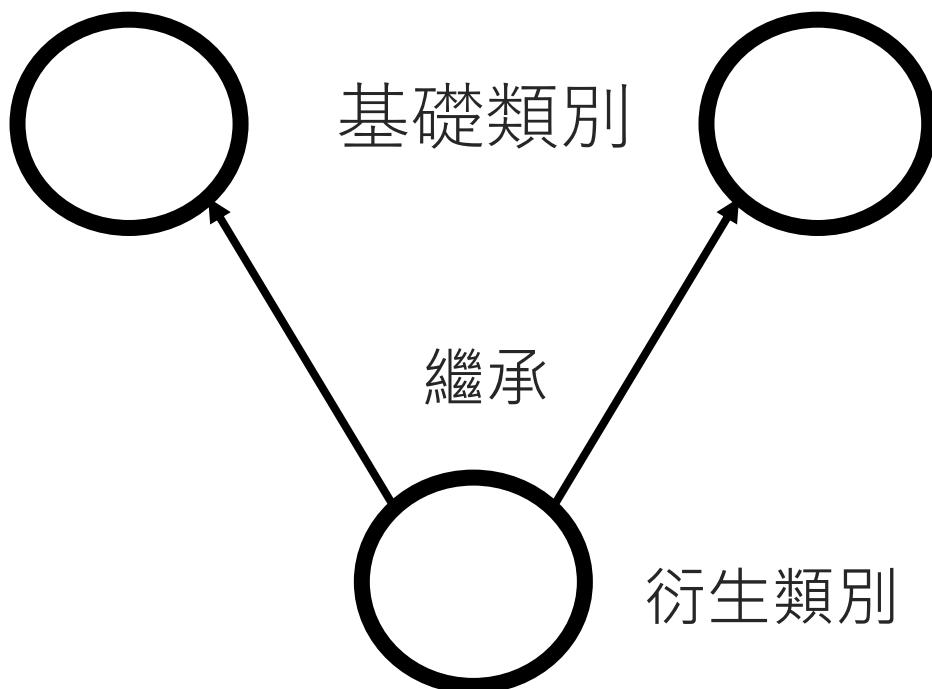
如此以來新的程式寫新的功能，  
也可以沿用基礎類別的方法。

繼承也可以同時承接兩個基礎類別

衍生語法：

```
class 衍生類別名稱(基礎類別名稱 1, 基礎類別名稱 2):
```

...



```
1  class Drink:  
2      def __init__(self, name, price):  
3          self.name = name  
4          self.__price = price  
5  
6      def get_name(self):  
7          return self.name  
8  
9      @property  
10     def price_manage(self):  
11         return self.__price  
12  
13     @price_manage.setter  
14     def price_manage(self, price):  
15         self.__price = price
```

範例程式如下：

```
18  class Flavor:  
19      def __init__(self, smell):  
20          self.smell = smell  
21  
22      def get_smell(self):  
23          return "味道: " + self.smell
```

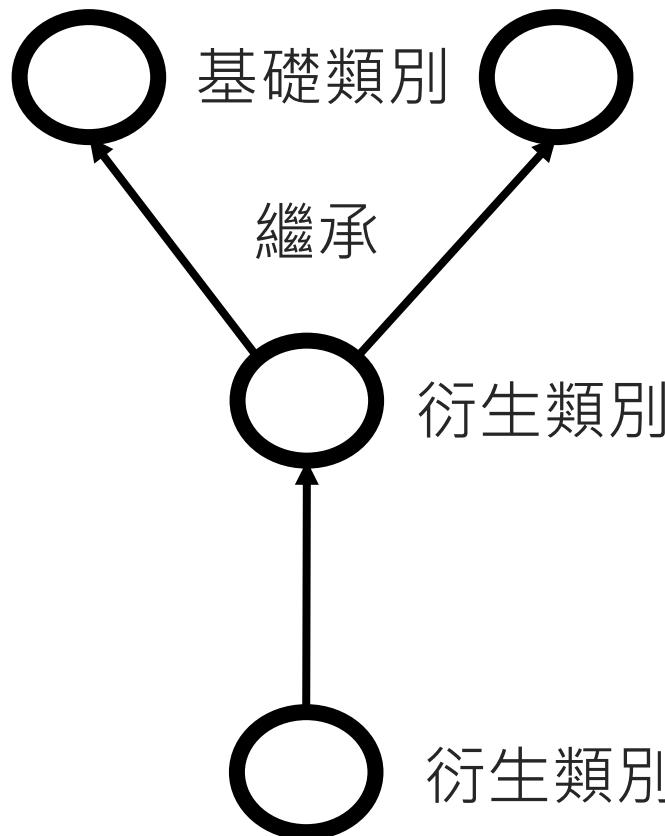
↓ 含下頁

```
26 class OriginDrink(Drink, Flavor):
27     def __init__(self, name, price, country, variety, smell):
28         super().__init__(name, price)
29         # Drink.__init__(self, name, price)
30         Flavor.__init__(self, smell)
31         self.country = country
32         self.variety = variety
33
34     def get_country(self):
35         return "產地: " + self.country
36
37     def get_variety(self):
38         return "品種: " + self.variety
39
40
41     drink = OriginDrink("coffee", 100, "Blue Mountain", "Arabica", "good")
42     d_n = drink.get_name()
43     d_p = drink.price_manage
44     d_c = drink.get_country()
45     d_v = drink.get_variety()
46     d_s = drink.get_smell()
47     print(d_v, d_c, d_n, "價格: ", d_p, d_s)
```

輸出結果

品種: Arabica 產地: Blue Mountain coffee 價格: 100 味道: good

衍生類別可以再衍生類別：



```
1  class Drink:...
16
17
18  class Flavor:...
24
25
26  class OriginDrink(Drink, Flavor):...
39
40
41  class Company(OriginDrink):
42      def __init__(self, company_name, name, price, country, variety, smell):
43          super().__init__(name, price, country, variety, smell)
44          self.company_name = company_name
45
46      def get_company(self):
47          return "銷售公司：" + self.company_name
```

↓ 含下頁

```
50     drink = Company("一號店", "coffee", 100, "Blue Mountain", "Arabica", "good")
51     d_n = drink.get_name()
52     d_p = drink.price_manage
53     d_c = drink.get_country()
54     d_v = drink.get_variety()
55     d_s = drink.get_smell()
56     d_cp = drink.get_company()
57     print(d_cp, d_v, d_c, d_n, "價格:", d_p, d_s)
```

輸出結果

銷售公司：一號店 品種：Arabica 產地：Blue Mountain coffee 價格：100 味道：good

使用衍生類別的目的是，  
在不更動基礎類別的情況下去新增、加強方法，  
前述的範例是新增，也就是說我刻意寫不同的方法。  
如果我要加強原來的方法，  
這時可以用和基礎類別一樣的方法名稱，  
而這又稱為覆寫。

```
1 class Origin:  
2     def __init__(self, country, variety, company_name):  
3         self.country = country  
4         self.variety = variety  
5         self.company_name = company_name  
6  
7     def get_country(self):  
8         return "產地: " + self.country  
9  
10    def get_variety(self):  
11        return "品種: " + self.variety  
12  
13    def get_company(self):  
14        return "銷售公司: " + self.company_name
```

基礎類別

↓ 含下頁

```
17 class Company(Origin):                                衍生類別  
18     def __init__(self, country, variety, company_name):  
19         super().__init__(country, variety, company_name)  
20  
21     def get_company(self): ← 覆寫，  
22         return "銷售分店: " + self.company_name  
23  
24  
25 product = Company("Blue Mountain", "Arabica", "一號店") 建立實體物件  
26 p_cp = product.get_company()                         如此一來可以用  
27 p_ct = product.get_country()                        加強的衍生類別  
28 p_v = product.get_variety()  
29 print(p_cp, p_ct, p_v)                            呼叫方法  
                                              輸出結果
```

銷售分店: 一號店 產地: Blue Mountain 品種: Arabica

學完衍生類別、繼承、覆寫，  
請試著自己解釋程式碼！

```
1 class ProductOriginal:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.price = price  
5  
6     def get_name(self):  
7         return "產品名稱: " + self.name  
8  
9     def get_price(self):  
10        return {"售價": self.price}
```

```
13 class ProductDiscount(ProductOriginal):  
14     def __init__(self, name, price, discount):  
15         super().__init__(name, price)  
16         self.discount = discount  
17  
18     def get_discount_price(self):  
19         return {"特價": self.price * self.discount}
```



```
22     product = ProductOriginal("coffee", 100)
23     p_gn = product.get_name()
24     p_gp = product.get_price()
25     print(p_gn, p_gp)
26     print("-" * 20)
27     product_2 = ProductDiscount("coffee", 100, 0.8)
28     p_gn_2 = product_2.get_name()
29     p_gp_2 = product_2.get_discount_price()
30     print(p_gn_2, p_gp_2)
```

輸出結果

產品名稱: coffee {'售價': 100}

-----

產品名稱: coffee {'特價': 80.0}

有不同的類別，但是有相同的方法，使用時彼此間互相獨立不會干擾。也正因如此，所以可以提供一個介面統合相同方法的物件。

```
1  class DrinkA:  
2      def size(self):  
3          return "大杯"  
4  
5  
6  class DrinkB:  
7      def size(self):  
8          return "特大杯"  
9  
10  
11 a = DrinkA()  
12 print("1.\t", a.size())  
13  
14 b = DrinkB()  
15 print("2.\t", b.size())
```

輸出結果

1. 大杯
2. 特大杯

```
1 class DrinkA:  
2     def size(self):  
3         return "大杯"  
4  
5 class DrinkB:  
6     def size(self):  
7         return "特大杯"  
8  
9 def cup(par):  
10    return par.size()  
11  
12 a = DrinkA()  
13 print("1.\t", cup(a))  
14 b = DrinkB()  
15 print("2.\t", cup(b))
```

輸出結果

1. 大杯
2. 特大杯

## 14-8 特殊的定義方法

在類別內的定義可以是我們的定義，另外還有既有的定義方法：

方法	對應的函數	說明
<code>_str_</code>	<code>str()</code>	傳回字串
<code>_format_</code>	<code>format()</code>	傳回格式化字串
<code>_int_</code>	<code>int()</code>	傳回整數
<code>_float_</code>	<code>float()</code>	傳回浮點數

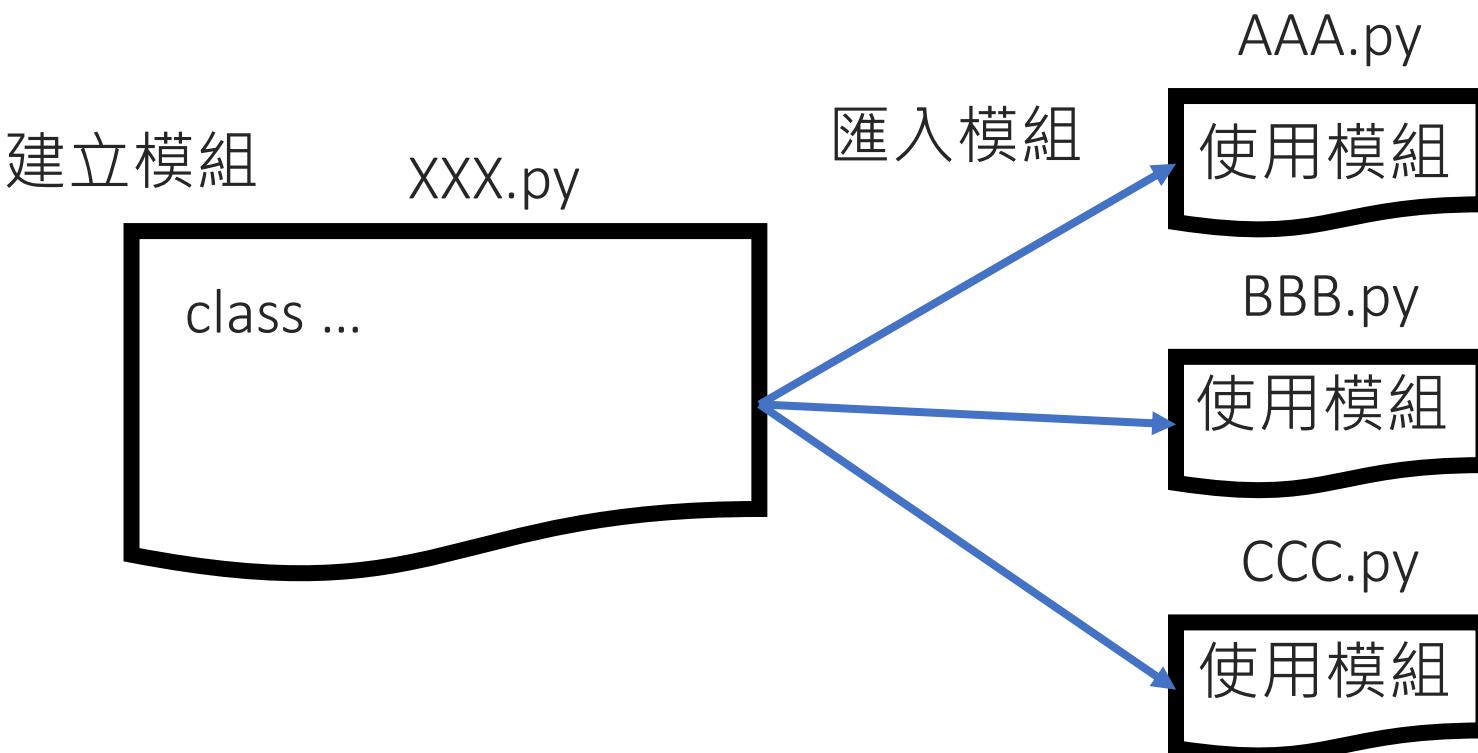
方法	對應的函數	說明
<code>_add_(self, other)</code>	<code>+</code>	加法
<code>_sub_(self, other)</code>	<code>-</code>	減法
<code>_mul_(self, other)</code>	<code>*</code>	乘法
<code>_truediv_(self, other)</code>	<code>/</code>	除法
<code>_mod_(self, other)</code>	<code>%</code>	取餘數

```
1 class Drink:  
2     def __init__(self, name, price):  
3         self.name = name  
4         self.price = price  
5  
6     def __str__(self):  
7         dr = self.name + "好喝"  
8         return dr  
9  
10    def __int__(self):  
11        price = self.price  
12        return price  
13  
14    def __add__(self, other):  
15        price = self.price + other.price  
16        return price
```

```
19 drink = Drink("coffee", 100)  
20 print(str(drink))  
21 print(int(drink))  
22 drink_2 = Drink("tea", 90)  
23 print(str(drink_2))  
24 print(int(drink_2))  
25 print("總額:", drink + drink_2)
```

輸出結果	
coffee	好喝
100	
tea	好喝
90	
總額:	190

到目前為止，我們學過許多和函數與類別的相關內容。  
也提到過，寫成函數與類別可以讓多個程式可以使用函數與類別，  
接下來我們就來談談它是如何運作、執行的。



這裡建立一個.py檔案，檔案名稱可自由命名(這裡我命名為 Sample\_14\_file\_name.py)，範例程式如下。

```
Sample_14_file_name.py x
1  class Origin:
2      def __init__(self, country, variety, company_name):
3          self.country = country
4          self.variety = variety
5          self.company_name = company_name
6
7      def get_country(self):
8          return "產地: " + self.country
9
10     def get_variety(self):
11         return "品種: " + self.variety
12
13     def get_company(self):
14         return "銷售公司: " + self.company_name
15
16
17 class Company(Origin):
18     def __init__(self, country, variety, company_name):
19         super().__init__(country, variety, company_name)
20
21     def get_company(self):
22         return "銷售分店: " + self.company_name
```

另外我有別的檔案，是需要執行的程式，而這程式需要之前寫好的模組。  
要使用其他模組要先做匯入，

匯入語法：

import 模組名稱

1. 匯入模組

```
1 import Sample_14_file_name
2
3 product = Sample_14_file_name.Company("Blue Mountain", "Arabica", "一號店")
4 p_cp = product.get_company()
5 p_ct = product.get_country()
6 p_v = product.get_variety()
7 print(p_cp, p_ct, p_v)
```

輸出結果

銷售分店：一號店 產地：Blue Mountain 品種：Arabica

匯入模組後，其函式與類別需加入模組名稱使用。

使用匯入的函數和類別語法：

模組名稱.模組內的函數名稱或類別名稱

```
1 import Sample_14_file_name  
2  
3 product = Sample_14_file_name.Company("Blue Mountain", "Arabica", "一號店")  
4 p_cp = product.get_company()  
5 p_ct = product.get_country()  
6 p_v = product.get_variety()  
7 print(p_cp, p_ct, p_v)
```

2. 使用模組，  
加入模組名稱

輸出結果

銷售分店：一號店 產地：Blue Mountain 品種：Arabica

匯入模組後模組的名稱有時會很長，使用匯入時會都要多一串模組名稱。  
所以有一種方法可以幫他取個別名。

命名語法：

import 模組名稱 as 自己取的別名

```
1 import Sample_14_file_name as Sfn
2
3 product = Sfn.Company("Blue Mountain", "Arabica", "一號店")
4 p_cp = product.get_company()
5 p_ct = product.get_country()
6 p_v = product.get_variety()
7 print(p_cp, p_ct, p_v)
```

輸出結果

銷售分店：一號店 產地：Blue Mountain 品種：Arabica

還有一種方法是匯入要使用的函數或類別名稱

命名語法：

from 模組名稱 import 函數名稱或類別名稱

```
1   from Sample_14_file_name import Company, Origin  
2  
3   product = Company("Blue Mountain", "Arabica", "一號店")  
4   p_cp = product.get_company()  
5   p_ct = product.get_country()  
6   p_v = product.get_variety()  
7   print(p_cp, p_ct, p_v)  
8   product_o = Origin("Blue Mountain", "Arabica", "一號店")  
9   o_cp = product_o.get_company()  
10  o_ct = product_o.get_country()  
11  o_v = product_o.get_variety()  
12  print(o_cp, o_ct, o_v)
```

如此一來在使用上就不用加上模組名稱

輸出結果

銷售分店：一號店 產地：Blue Mountain 品種：Arabica  
銷售公司：一號店 產地：Blue Mountain 品種：Arabica

從上一個範例中，如果需要使用多的類別或函數，那就也必須匯入多個，所以有可以全部匯入的方式。

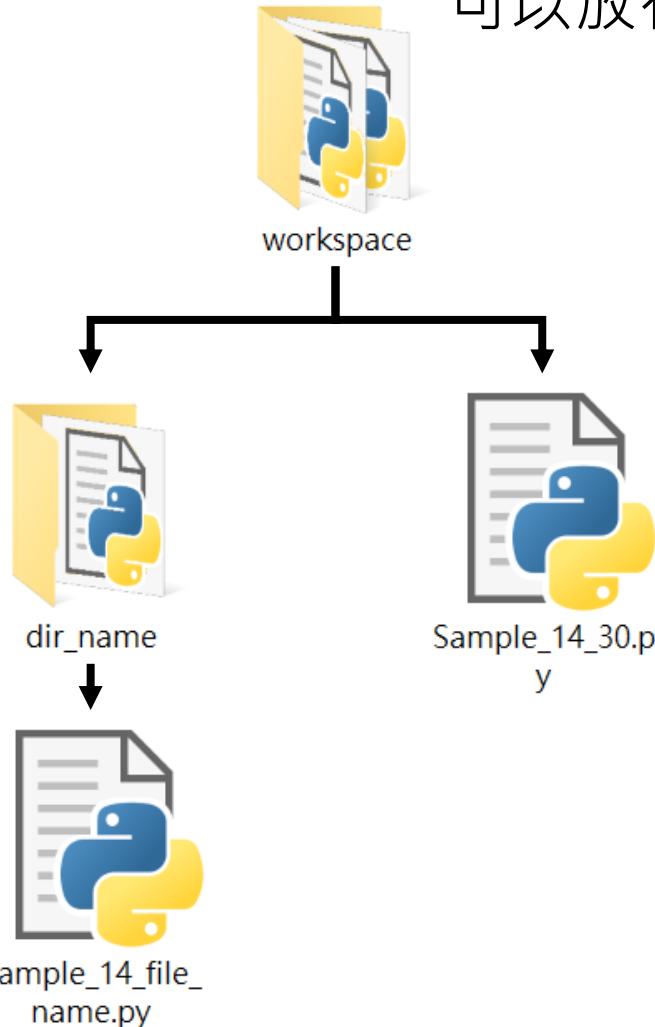
```
1 from Sample_14_file_name import *      ↗ 只改這裡  
2  
3 product = Company("Blue Mountain", "Arabica", "一號店")  
4 p_cp = product.get_company()  
5 p_ct = product.get_country()  
6 p_v = product.get_variety()  
7 print(p_cp, p_ct, p_v)  
8 product_o = Origin("Blue Mountain", "Arabica", "一號店")  
9 o_cp = product_o.get_company()  
10 o_ct = product_o.get_country()  
11 o_v = product_o.get_variety()  
12 print(o_cp, o_ct, o_v)
```

命名語法：

from 模組名稱 import \*

輸出結果

銷售分店：一號店 產地：Blue Mountain 品種：Arabica  
銷售公司：一號店 產地：Blue Mountain 品種：Arabica



可以放在特定的資料夾下

加入套件名稱

```
Sample_14_30.py
```

```
from dir_name.Sample_14_file_name import *

product = Company("Blue Mountain", "Arabica", "一號店")
p_cp = product.get_company()
p_ct = product.get_country()
p_v = product.get_variety()
print(p_cp, p_ct, p_v)

product_o = Origin("Blue Mountain", "Arabica", "一號店")
o_cp = product_o.get_company()
o_ct = product_o.get_country()
o_v = product_o.get_variety()
print(o_cp, o_ct, o_v)
```

The screenshot shows a code editor window titled "Sample\_14\_30.py". The code imports modules from a specific directory ("dir\_name") and uses them to create objects ("Company" and "Origin") with their respective attributes and methods. The code is annotated with arrows: one arrow points from the "dir\_name" folder in the workspace diagram to the "dir\_name" in the import statement, and another arrow points from the "Sample\_14\_file\_name" file in the workspace diagram to the "Sample\_14\_file\_name" in the import statement.

輸出結果

銷售分店: 一號店 產地: Blue Mountain 品種: Arabica  
銷售公司: 一號店 產地: Blue Mountain 品種: Arabica

Python 有許多的模組可以使用，  
其可用的模組之多，多到沒有一一列出來的必要，  
而且有許多模組可以從網路上取得。

以及未來在學習大數據分析時會使用到很多種的模組。

所以接下來的課程有些會教一些常用的模組。  
一方面能讓我們更方便的撰寫程式，  
同時也為我們進入數據分析前的鋪路。

模組之多，但不需要一一去學，未來想用 python 寫某些功能程式，  
只需要運用關鍵字去收尋網路提供的建議，再針對特定的模組從做中學。

# Module 15： 字串的操作

15-1 字串的基本操作

15-2 格式化字串

15-3 認識轉換字串的方法

15-4 字串的搜尋、取代的方法

先前我們有簡單的提到過字串，而沒有說過字串能怎麼操作，而程式可以處理各種來源的資料，所以我們可以利用操作字串的方式，協助我們更方便的取得我們要處理字串的某一部分。

在 Python 中，字串(string)和串列(list)、元組(tuple)一樣都是有序列的。所以有些串列有的操作，字串也有。

利用索引值取得字串的資料	輸出結果	
1	str_a = "coffee"	1. c
2	print("1.\t", str_a[0])	2. o
3	print("2.\t", str_a[1])	3. f
4	print("3.\t", str_a[3])	4. e
5	print("4.\t", str_a[-1])	

利用切片取得字串的資料

```
1 str_a = "coffee"
2 print("1.\\t", str_a[:4])
3 print("2.\\t", str_a[2:4])
4 print("3.\\t", str_a[::-1])
```

輸出結果

1. coff
2. ff
3. eefffoc

還記得串列中教過索引值和切片吧!

利用for 迴圈取得字串的資料

```
1     str_a = "coffee"  
2  
3     for i in str_a:  
4         print(i)
```

輸出結果

c  
o  
f  
f  
e  
e

利用 len() 取得字串的長度

```
1     str_a = "coffee"  
2  
3     print(len(str_a))
```

輸出結果

6

字串無法將內部的值進行變更

```
1     str_a = "coffee"
2     print(str_a[3])
3     str_a[3] = "g"
```

輸出結果

Traceback (most recent call last):

```
  File "D:/Python/workspace/Sample_15_5.py", line 3, in <module>
    str_a[3] = "g"
```

TypeError: 'str' object does not support item assignment

如果將「+」使用在字串上，字串的連接。

```
1     str_a = "Pyt"
2     str_b = "hon"
3     str_c = str_a + str_b
4     print(str_c)
```

輸出結果

Python

可以用format() 的方式，協助我們執行設計好要的字串格式。

執行範例如下，用{}來指定，索引值或 key。

```
1     str_a = "產品:{0}, 評價:{1}".format("coffee", "*" * 5)
2     print(str_a)
```

輸出結果                    產品:coffee, 評價:\*\*\*\*\*

```
1     str_a = "產品:{key_1}, 評價:{key_2}".format(key_1="tea", key_2="good")
2     print(str_a)
```

輸出結果                    產品:tea, 評價:good

格式化字串有些主要的格式列表如下：

格式符號	說明	格式符號	說明
數值	數字位數	%	顯示 %
空白	正數前面加上空白	b	2 進位
+	加上正符號	o	8 進位
-	加上負符號	d	10 進位
,	顯示千分位	x	16 進位
<	向左對齊	f	固定小數點位數
>	向右對齊	e	指數標記
^	置中對齊		

如下列範例中使用了：

置中對齊、向左對齊、向右對齊(其中預留 5 個空位可填入)，數字位數小數點位數寫到第 10 位。

```
1     number_of_voters = 121
2     number_of_agree = 70
3     number_of_disagree = 20
4     vote_rate = (number_of_agree + number_of_disagree) / number_of_voters * 100
5
6     str_ans = "投票人數:{0:^5}, 同意人數:{1:<5}, 不同意人數:{2:>5}, 投票率:{3:.10f}\"\n"
7         .format(number_of_voters, number_of_agree, number_of_disagree, vote_rate)
8     print(str_ans)
```

輸出結果 投票人數: 121 , 同意人數: 70 , 不同意人數: 20, 投票率: 74.3801652893

如下列範例中使用了：

千分位

```
1     number_of_voters = 1234567890
2     str_ans = "投票人數:{0:,}".format(number_of_voters)
3     print(str_ans)
```

輸出結果

投票人數:1,234,567,890

如下列範例中使用了：  
各進位表示法與指數表示

```
1     number = 1234567890
2     str_ans = "{0}的 2 進位: {0:b},\n" \
3                 "{0}的 8 進位: {0:o},\n" \
4                 "{0}的 10 進位: {0:d},\n" \
5                 "{0}的 16 進位: {0:x},\n" \
6                 "{0}的指數標記: {0:e},\n".format(number)
7     print(str_ans)
```

輸出結果

1234567890的 2 進位: 1001001100101100000001011010010,  
1234567890的 8 進位: 11145401322,  
1234567890的 10 進位: 1234567890,  
1234567890的 16 進位: 499602d2,  
1234567890的指數標記: 1.234568e+09,

也可以搭配字典使用

```
1     number_of_voters = 121
2     number_of_agree = 70
3     number_of_disagree = 20
4     vote_rate = (number_of_agree + number_of_disagree) / number_of_voters * 100
5
6     info = {"number_of_voters": number_of_voters,
7               "number_of_agree": number_of_agree,
8               "vote_rate": vote_rate}
9
10    str_ans = "投票人數:{number_of_voters}, "
11        "同意人數:{number_of_agree }, "
12        "投票率:{vote_rate:.2f}%".format(**info)
13
14    print(str_ans)
```

輸出結果

投票人數:121, 同意人數:70, 投票率:74.38%

也可以搭配串列使用

```
1     number_of_voters = 121
2     number_of_agree = 70
3     number_of_disagree = 20
4     vote_rate = (number_of_agree + number_of_disagree) / number_of_voters * 100
5
6     info = [number_of_voters, number_of_agree, vote_rate]
7
8     str_ans = "投票人數:{0[0]}, 同意人數:{0[1]}, 投票率:{0[2]:.2f}%".format(info)
9     print(str_ans)
```

輸出結果 投票人數:121, 同意人數:70, 投票率:74.38%

~也可以搭配元組使用~

格式化字串也可以用在最前面加上「f」或「F」。

```
1     number_of_voters = 121
2     number_of_agree = 70
3     number_of_disagree = 20
4     vote_rate = (number_of_agree + number_of_disagree) / number_of_voters * 100
5
6     info = [number_of_voters, number_of_agree, vote_rate]
7
8     str_ans_1 = f"投票人數:{number_of_voters:^5}, 同意人數:{number_of_agree}, 投票率:{vote_rate:.2f}%" 
9     print(str_ans_1)
10    str_ans_2 = F"投票人數:{number_of_voters:^5}, 同意人數:{number_of_agree}, 投票率:{vote_rate:.2f}%" 
11    print(str_ans_2)
```

輸出結果

投票人數: 121 , 同意人數:70, 投票率:74.38%  
投票人數: 121 , 同意人數:70, 投票率:74.38%

字串有一些轉換的方法，列表如下：

方法	方法說明
字串.upper()	取得字串轉成大寫的字串
字串.lower()	取得字串轉成小寫的字串
字串.swapcase()	取得字串大寫轉小寫，小寫轉大寫
字串.capitalize()	取得字串首字大寫，其餘小寫
字串.title()	取得每個單字字首轉成大寫
字串.center(寬度, 文字)	預設總體所占寬度，字串置中，可選擇指定單個字符的文字
字串.ljust(寬度, 文字)	預設總體所占寬度，字串靠左，可選擇指定單個字符的文字
字串.rjust(寬度, 文字)	預設總體所占寬度，字串靠右，可選擇指定單個字符的文字
字串.strip(文字)	刪除空白或指定文字
字串.lstrip(文字)	刪除字串最左邊的空白或指定文字
字串.rstrip(文字)	刪除字串最右邊的空白或指定文字
字串.split(sep=None, maxsplit=-1)	將字串分成每個單字組成的列表，也可依照特定符號和次數分割
字串.splitline(有無換行符號)	依照 \n 作分割符號，並可選擇是否保留 \n
字串.join(要連接的元素序列)	連接字串並決定分隔符號
字串.format(要填入的字串)	以指定的方式填入字串

字串可以操作大小寫的轉換：

1. 字串.upper()：將字串轉為全大寫
2. 字串.lower()：將字串轉為全小寫

```
1 str_a = "coffee"
2 str_b = "Tea"
3 str_c = "MILK"
4
5 print("1.{0:^8} 取得大寫後 {1}".format(str_a, str_a.upper()))
6 print("2.{0:^8} 取得大寫後 {1}".format(str_b, str_b.upper()))
7 print("3.{0:^8} 取得小寫後 {1}".format(str_b, str_b.lower()))
8 print("4.{0:^8} 取得小寫後 {1}".format(str_c, str_c.lower()))
```

輸出結果

1.	coffee	取得大寫後 COFFEE
2.	Tea	取得大寫後 TEA
3.	Tea	取得小寫後 tea
4.	MILK	取得小寫後 milk

字串可以操作大小寫的轉換：

3. 字串.capitalize()：取得字串首字大寫，其餘小寫
4. 字串.swapcase()：取得字串大寫轉小寫，小寫轉大寫

```
1 str_a = "coffee"
2 str_b = "Tea"
3 str_c = "MILK"

5 print("1.{0:^8} 首字大寫其餘小寫 {1}".format(str_a, str_a.capitalize()))
6 print("2.{0:^8} 首字大寫其餘小寫 {1}".format(str_b, str_b.capitalize()))
7 print("3.{0:^8} 首字大寫其餘小寫 {1}".format(str_c, str_c.capitalize()))
8 print("4.{0:^8} 大小寫互換 {1}".format(str_a, str_a.swapcase()))
9 print("5.{0:^8} 大小寫互換 {1}".format(str_b, str_b.swapcase()))
10 print("6.{0:^8} 大小寫互換 {1}".format(str_c, str_c.swapcase()))
```

### 輸出結果

1.	coffee	首字大寫其餘小寫	Coffee
2.	Tea	首字大寫其餘小寫	Tea
3.	MILK	首字大寫其餘小寫	Milk
4.	coffee	大小寫互換	COFFEE
5.	Tea	大小寫互換	tEA
6.	MILK	大小寫互換	milk

## 5. 字串.title()：取得每個單字字首轉成大寫

```
1 str_d = "MILK is good drink"
2
3 print("1. {0}每個單字字首轉成大寫 {1}".format(str_d, str_d.title()))
4 print("2. {0}取得大寫後 {1}".format(str_d, str_d.upper()))
5 print("3. {0}取得小寫後 {1}".format(str_d, str_d.lower()))
6 print("4. {0}首字大寫其餘小寫 {1}".format(str_d, str_d.capitalize()))
7 print("5. {0}大小寫互換 {1}".format(str_d, str_d.swapcase()))
```

- 輸出結果
1. MILK is good drink 每個單字字首轉成大寫 Milk Is Good Drink
  2. MILK is good drink 取得大寫後 MILK IS GOOD DRINK
  3. MILK is good drink 取得小寫後 milk is good drink
  4. MILK is good drink 首字大寫其餘小寫 Milk is good drink
  5. MILK is good drink 大小寫互換 milk IS GOOD DRINK

也可以預設總體所占寬度，  
字串於其中可靠左、靠右或置中，再選擇是否空白或指定單個字符的文字。

6. 字串.center(寬度, 文字)
7. 字串.ljust(寬度, 文字)
8. 字串.rjust(寬度, 文字)

#### 輸出結果

1. coffee
2. -----coffee-----
3. coffee
4. coffee\*\*\*\*\*
5. coffee
6. \*\*\*\*\*coffee
7. Coffee
8. -----COFFEE-----

```
1 str_a = "coffee"
2
3 print("1.\t", str_a.center(20))
4 print("2.\t", str_a.center(20, "-"))
5 print("3.\t", str_a.ljust(20))
6 print("4.\t", str_a.ljust(20, "*"))
7 print("5.\t", str_a.rjust(20))
8 print("6.\t", str_a.rjust(20, "*"))
9
10 print("7.\t", str_a.capitalize().center(20))
11 print("8.\t", str_a.swapcase().center(20, "-"))
```

```
1 str_a = "-----coffee-----"
2 str_b = "      coffee      "
3 str_c = "1232coffee3232"
4
5 print("0.\t", str_a)
6 print("1.\t", str_a.strip("-"))
7 print("2.\t", str_a.lstrip("-"))
8 print("3.\t", str_a.rstrip("-"))
9 print("4.\t", str_b)
10 print("5.\t", str_b.strip())
11 print("6.\t", str_b.lstrip())
12 print("7.\t", str_b.rstrip())
13 print("8.\t", str_c)
14 print("9.\t", str_c.strip("123"))
15 print("10.\t", str_c.lstrip("123"))
16 print("11.\t", str_c.rstrip("123"))
```

輸出結果

也可以刪除空白或指定文字。

6. 字串.strip(文字)
7. 字串.lstrip(文字)
8. 字串.rstrip(文字)

0. -----coffee-----
1. coffee
2. coffee-----
3. -----coffee
4. coffee
5. coffee
6. coffee
7. coffee
8. 1232coffee3232
9. coffee
10. coffee3232
11. 1232coffee

也可以將字串分成每個單字組成的列表，  
包含空格、\n 和 \t 等，也可依照特定符號分割。

### 9. 字串.split(sep=None, maxsplit=-1)

```
1     str_a = "It's fun to stay at the Y.M.C.A."
2     str_b = "Python\nIs\tFun"
3     str_c = "0912-343-545"
4
5     print("1.\t", str_a.split())
6     print("2.\t", str_b.split())
7     print("3.\t", str_c.split("-"))
8     print("4.\t", str_a.split(" ", 2))
9     print("5.\t", str_c.split("-", 1))
```

#### 輸出結果

1. ["It's", 'fun', 'to', 'stay', 'at', 'the', 'Y.M.C.A. ']
2. ['Python', 'Is', 'Fun']
3. ['0912', '343', '545']
4. ["It's", 'fun', 'to stay at the Y.M.C.A. ']
5. ['0912', '343-545']

依照 \n 作分割符號，並可選擇是否保留 \n

### 10. 字串.splitlines(有無換行符號)

```
1     str_a = "It's fun to stay at the Y.M.C.A."  
2     str_b = "It's fun\n to stay at the\n Y.M.C.A."  
3  
4     print("1.\t", str_a.splitlines())  
5     print("2.\t", str_b.splitlines())  
6     print("3.\t", str_b.splitlines(True))
```

輸出結果

1. ["It's fun to stay at the Y.M.C.A."]
2. ["It's fun", ' to stay at the', ' Y.M.C.A. ']
3. ["It's fun\n", ' to stay at the\n', ' Y.M.C.A. ']

可以結合串列的字串，同時選擇加入分隔的符號。

### 11. 字串.join(要連接的元素序列)

```
1     list_c = ['0912', '343', '545']
2     str_a = ""
3     str_b = "-"
4     print("1.\t", str_a.join(list_c))
5     print("2.\t", str_b.join(list_c))
```

輸出結果

1. 0912343545
2. 0912-343-545

字串有一些搜尋、取代和計算次數的方法，列表如下：

方法	說明
被找的字串.find(要找的字串, 開始位置, 結束位置)	搜尋部分字串位置
被找的字串.rfind(要找的字串, 開始位置, 結束位置)	以逆向方式搜尋部分字串位置
被找的字串.index(要找的字串, 開始位置, 結束位置)	和 .find() 相似，但找不到時會出現錯誤
被取代的字串.replace(old string, new string, 次數)	將字串中的 old 取代成 new, 並可選擇取代次數
被找的字串.count(要找的字串, 開始位置, 結束位置)	計算特定字串出現的次數
被找的字串.startswith(要找的字串, 開始位置, 結束位置)	判斷開頭字串是否符合要找的字串
被找的字串.endswith(要找的字串, 開始位置, 結束位置)	判斷結尾字串是否符合要找的字串

字串的搜尋方法：

1. 搜尋字串 `find()`：被找的字串.`find(要找的字串, 開始位置, 結束位置)`
2. 以逆向方式搜尋字串 `rfind()`：被找的字串.`rfind(要找的字串, 開始位置, 結束位置)`

其中開始的預設值是 0，結束的預設值是最後也就是 -1，  
如果有找到字串則會回傳找到的位置，若沒找到則會回傳 -1。

1	str_a = "It's fun to stay at the Y.M.C.A."		
2	str_f = "to"		
3	str_e = "ABC"		
4		輸出結果	
5	print("1.\t", str_a.find(str_f))	1.	9
6	print("2.\t", str_a.find(str_f, 5))	2.	9
7	print("3.\t", str_a.find(str_f, 5, 15))	3.	9
8	print("4.\t", str_a.find(str_f, 10))	4.	-1
9	print("5.\t", str_a.find(str_e))	5.	-1

```
1     str_a = "This is an apple"
2     str_f = "a"
3     str_e = "ABC"
4
5     print("1.\t", str_a.rfind(str_f))
6     print("2.\t", str_a.find(str_f))
7
8     print("3.\t", str_a.rfind(str_f, 2, 15))
9     print("4.\t", str_a.rfind(str_f, 2, 10))
10    print("5.\t", str_a.rfind(str_e))
```

輸出結果

1.	11
2.	8
3.	11
4.	8
5.	-1

字串的搜尋方法：

3. 搜尋字串 `index()`：被找的字串. `index(要找的字串, 開始位置, 結束位置)`

使用方式和 `find()` 相同，差別在於：

如果找不到會顯示錯誤。

```
1     str_a = "This is an apple"
2
3     str_f = "a"
4
5     print("1.\t", str_a.index(str_f))
6     print("2.\t", str_a.index(str_f, 5, 7))
7     #print("3.\t", str_a.index(str_f, 10))
8     #print("4.\t", str_a.index(str_e))
```

輸出結果

```
1.    8
Traceback (most recent call last):
  File "D:/Python/workspace/Sample_15_25.py", line 6,
        print("2.\t", str_a.index(str_f, 5, 7))
ValueError: substring not found
```

字串的取代方法：

1. 取代字串 `replace()`：被取代的字串. `index(old string, new string, 取代次數)`，將字串中的 `old` 取代成 `new`, 並可選擇取代次數。取代次數預設全部取代。

```
1 str_a = "ABC.txt.txt.txt.txt"  
2 str_b = ".txt"  
3 str_c = ".py"  
4  
5 print("1.\t", str_a.replace(str_b, str_c))          輸出結果  
6 print("2.\t", str_a.replace(str_b, str_c, 1))  
7 print("3.\t", str_a.replace(str_b, str_c, 3))  
1.    ABC.py.py.py.py  
2.    ABC.py.txt.txt.txt  
3.    ABC.py.py.txt
```

可以計算特定字串出現的次數：

字串.count(特定字串, 開始位置, 結束位置)。

其中開始的預設值是 0，結束的預設值是最後也就是 -1，  
如果有找到字串則會回傳次數，若沒找到則會回傳 0。

```
1 str_a = "ABC.txt.txt.txt.txt"  
2 str_b = ".txt"  
3  
4 print("1.\t", str_a.count(str_b))  
5 print("2.\t", str_a.count(str_b, 5))  
6 print("3.\t", str_a.count(str_b, 5, 15))
```

輸出結果  
1. 4  
2. 3  
3. 2

字串.startwith(特定字串, 開始位置, 結束位置)。

搜尋字串為字串開頭時回傳 True , 否則回傳 False 。

字串.endswith(特定字串, 開始位置, 結束位置)。

搜尋字串為字串結尾時回傳 True , 否則回傳 False 。

1	str_a = "This is an apple"	輸出結果
2	str_b = "This"	
3	str_c = "is"	
4		
5	print("1.\"", str_a.startswith(str_b))	
6	print("2.\"", str_a.startswith(str_c))	
7	print("3.\"", str_a.startswith(str_b, 5))	
8	print("4.\"", str_a.startswith(str_c, 5))	

```
1     str_a = "This is an apple"  
2     str_b = "apple"  
3     str_c = "is"  
4  
5     print("1.\t", str_a.endswith(str_b))  
6     print("2.\t", str_a.endswith(str_c))  
7     print("3.\t", str_a.endswith(str_b, 5))  
8     print("4.\t", str_a.endswith(str_c, 0, 4))
```

輸出結果

1. True
2. False
3. True
4. True

# Module 16 :

## 檔案

16-1 檔案的相關知識

16-2 文字檔案

16-2-2 寫入文字檔案

16-2-3 讀取文字檔案

16-2-4 新增文字檔案

16-2-5 新建立文字檔案

16-2-6 可寫後讀文字檔案

16-2-7 可讀後寫文字檔案

16-2-8 先讀後寫文字檔案

16-2-9 先寫後讀文字檔案

16-2-10 可選擇讀取多少元組序列

作業實作

16-2-11 用 with 開啟檔案

# Module 16 :

## 檔案

16-3 csv 檔案

16-3-1 csv 相關知識

16-3-2 寫入 csv 檔案

16-3-3 讀取 csv 檔案

16-4 JSON 檔案

16-4-1 JSON 相關知識

16-4-2 寫入 JSON 檔案

16-4-3 讀取 JSON 檔案

作業實作

程式通常會面對的是資料，而資料又是如何進入程式!?以及資料要怎麼輸出程式!?之前只有學過從鍵盤輸入，接著我們會學從檔案輸入。

關於資料輸出的部分，之前只有學顯示在螢幕上，一旦程式關閉就看不到了，所以說我們需要學習如何使用檔案。

檔案的寫入有寫入的方法，讀取有讀取的方法，  
模式與讀取方法列表如下：

開啟模式	說明	方法名稱	說明
w	寫入模式	檔案.write(字串)	將字串寫入檔案
r	讀取模式	檔案.writelines(字串)	將多數列字串寫入檔案
a	新增模式	檔案.readline()	讀取 1 列字串後回傳字串
x	新建立模式	檔案.readlines()	讀取多數列後回傳串列
w+	更新寫入	檔案.read(大小)	讀取字串，可選擇要讀多少元組序列。 預設全讀
r+	更新讀取	檔案.seek(位置)	移到讀寫位置
a+	更新新增	檔案.tell()	取得現在讀寫位置
wb	在 2 進位模式中寫入	檔案.close()	關閉檔案
rb	在 2 進位模式中讀取		

～在學習讀寫檔案前，我絕對要提醒，它有既定的語法，要用語法讀寫。～

首先我們先來做個小練習。

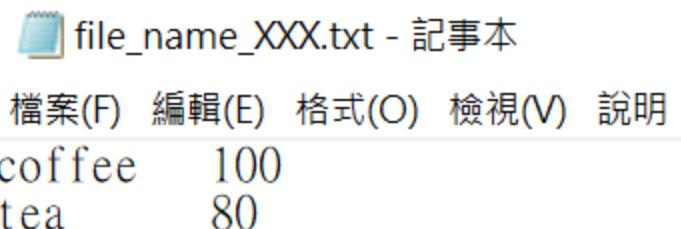
```
1     file_a = open("file_name_XXX.txt", "w")
2
3     file_a.write("coffee\t 100\n")
4     file_a.write("tea\t 00\n")
5
6     file_a.close()
```

如左邊範例：

1. 開啟檔案是運用 `open()` 的函數執行。
2. 開啟一個檔案名稱為 `file_name_XXX.txt` (自創名稱)的文字檔(`.txt`)，並且要以 “`w`” 寫入模式。
3. 使用 `write()` 將字串寫入資料
4. 寫完資料後請關閉檔案。否則可能會發生問題。

輸出結果

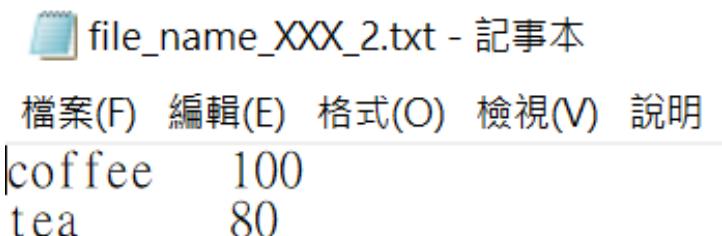
輸出結果請到  
同一層資料夾下找



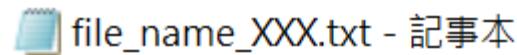
```
1 file_a = open("file_name_XXX.txt", "w")  
2  
3 data = ["coffee\t 100\n", "tea\t 80\n"]  
4 file_a.writelines(data)  
5  
6 file_a.close()
```

使用 writelines()  
可以將多數列字串寫入檔案。

輸出結果



1. 在同一資料夾下準備好檔案



2. 以閱讀模式開啟檔案

3. `readlines()` :

讀取多行文字，並回傳串列。

4. 運用迴圈依序顯示在螢幕上

5. 關閉檔案。

```
1     file_a = open("file_name_XXX.txt", "r")
2
3     data = file_a.readlines()
4     print("data: ", data)
5     for i in data:
6         print(i, end="")
7
8     file_a.close()
```

輸出結果 data: ['coffee\t 100\n', 'tea\t 80\n']  
coffee 100  
tea 80

```
1     file_a = open("file_name_XXX.txt", "r")  
2  
3     data = file_a.readlines()  
4  
5     for i in data:  
6         j = i.split()  
7         print("飲品: {0:^8}價格為: {1}".format(j[0], j[1]))  
8  
9     file_a.close()
```

請自行解釋左邊範例

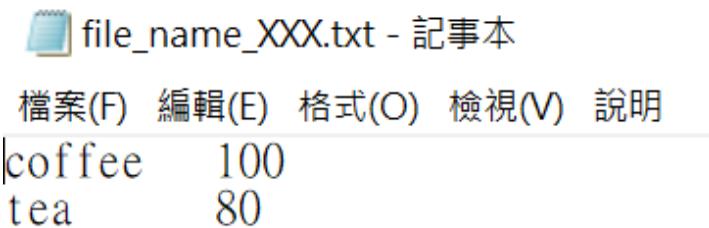
輸出結果 飲品: coffee 價格為: 100  
飲品: tea 價格為: 80

```
1     file_a = open("file_name_XXX.txt", "r")  
2  
3     data = file_a.readline()  
4     print(data)                                使用 readline() ·  
5  
6     file_a.close()                            可以將一列字串後回傳字串。
```

輸出結果

coffee 100

## 準備檔案



file\_name\_XXX.txt - 記事本

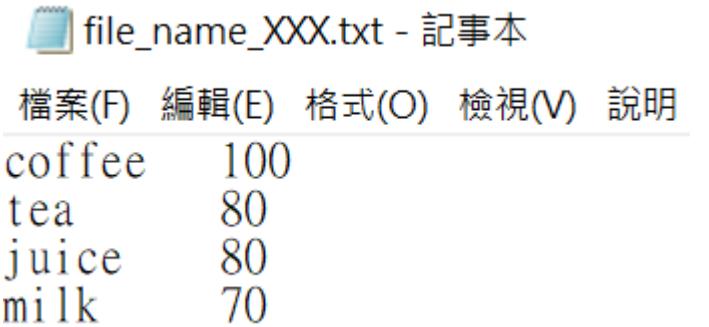
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

coffee 100  
tea 80

```
1 file_a = open("file_name_XXX.txt", "a")  
2  
3 data = ["juice\t 80\n", "milk\t 70\n"]  
4 file_a.writelines(data)  
5  
6 file_a.close()
```

a: append  
以附加的模式新增檔案資料

## 輸出結果



file\_name\_XXX.txt - 記事本

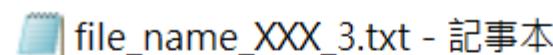
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

coffee 100  
tea 80  
juice 80  
milk 70

模式「x」新建立模式，如果尚未存在檔案，則會幫你建立。  
如果已經有檔案了，那麼就會給你錯誤訊息。  
相較於之前的「w」寫入模式，是直接弄全新的。

```
1     file_a = open("file_name_XXX_3.txt", "x")
2
3     data = ["juice\t 80\n", "milk\t 70\n"]
4     file_a.writelines(data)
5
6     file_a.close()
```

輸出結果



檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

juice 80  
milk 70

Traceback (most recent call last):

File "D:/Python/workspace/Sample\_16\_7.py", line 1, in <module  
 file\_a = open("file\_name\_XXX\_3.txt", "x")  
FileExistsError: [Errno 17] File exists: 'file\_name\_XXX\_3.txt'

模式「w+」為可寫入可讀取模式。

seek(位置) 可移動讀寫位置。

```
1 file_a = open("file_name_XXX_4.txt", "w+")
2
3 data = ["juice\t 80\n", "milk\t 70\n"]
4 file_a.writelines(data)
5 file_a.seek(0) # 把讀寫位置移到開始位置
6 lines = file_a.readlines()
7 for i in lines:
8     j = i.split()
9     print("飲品: {0:^8}價格為: {1}".format(j[0], j[1]))
10
11 file_a.close()
```

輸出結果



檔案(F) 編輯(E) 格式(O) 檢視(V) 說明		
juice	80	
milk	70	

飲品: juice 價格為: 80  
飲品: milk 價格為: 70

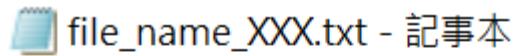
`tell()` 可以取得目前讀寫位置。

```
1  file_a = open("file_name_XXX_4.txt", "w+")
2
3  print("1.\t目前讀寫位置: ", file_a.tell())
4  data = ["juice\t 80\n", "milk\t 70\n"]
5  file_a.writelines(data)
6  print("2.\t目前讀寫位置: ", file_a.tell())
7  file_a.seek(0) # 把讀寫位置移到開始位置
8  print("3.\t目前讀寫位置: ", file_a.tell())
9  lines = file_a.readlines()
10 for i in lines:
11     j = i.split()
12     print("飲品: {0:^8}價格為: {1}".format(j[0], j[1]))
13
14 print("4.\t目前讀寫位置: ", file_a.tell())
15 file_a.close()
```

輸出結果

1. 目前讀寫位置: 0
2. 目前讀寫位置: 21
3. 目前讀寫位置: 0  
飲品: juice 價格為: 80  
飲品: milk 價格為: 70
4. 目前讀寫位置: 21

準備檔案



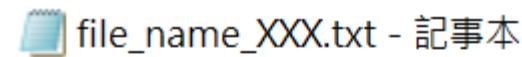
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

coffee	100
tea	80
juice	80
milk	70

```
1     file_a = open("file_name_XXX.txt", "r+")
2
3     lines = file_a.readlines()
4
5     for i in lines:
6         print(i, end="")
7
8     data = ["water\t 60\n", "milktea\t 110\n"]
9
10    file_a.writelines(data)
11
12    file_a.close()
```

輸出結果

coffee	100
tea	80
juice	80
milk	70



檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

coffee	100
tea	80
juice	80
milk	70
water	60
milktea	110

## 準備檔案

file_name_XXX.txt - 記事本	
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明	
coffee	100
tea	80
juice	80
milk	70
water	60
milktea	110

## 輸出結果

1. 目前讀寫位置: 68  
2. 目前讀寫位置: 0

-----  
coffee 100  
tea 80  
juice 80  
milk 70  
water 60  
milktea 110

3. 目前讀寫位置: 68

file_name_XXX.txt	
檔案(F) 編輯(E) 格式(O)	
coffee	100
tea	80
juice	80
milk	70
water	60
milktea	110
cake	60

```
1 file_a = open("file_name_XXX.txt", "a+")
2 print("1.\t目前讀寫位置: ", file_a.tell())
3 file_a.seek(0) # 把讀寫位置移到開始位置
4 print("2.\t目前讀寫位置: ", file_a.tell())
5 print("-" * 20)
6 lines = file_a.readlines()
7 for i in lines:
8     print(i, end="")
9
10 print("3.\t目前讀寫位置: ", file_a.tell())
11 file_a.write("cake\t60\n")
12 file_a.close()
```

## 準備檔案



coffee	100
tea	80
juice	80
milk	70
water	60
milktea	110
cake	60

1. 目前讀寫位置: 78  
2. 目前讀寫位置: 90

---

coffee	100
tea	80
juice	80
milk	70
water	60
milktea	110
cake	60
oolong	50

## 輸出結果

```
1 file_a = open("file_name_XXX.txt", "a+")
2 print("1.\t目前讀寫位置: ", file_a.tell())
3 file_a.write("oolong\t 50\n")
4 print("2.\t目前讀寫位置: ", file_a.tell())
5 file_a.seek(0) # 把讀寫位置移到開始位置
6 print("-" * 20)
7
8 lines = file_a.readlines()
9 for i in lines:
10     print(i, end="")
11
12 file_a.close()
```

準備檔案

	檔案(F)	編輯(E)	格式(O)
coffee	100		
tea	80		
juice	80		
milk	70		
water	60		
milktea	110		
cake	60		
oolong	50		

```
1     file_a = open("file_name_XXX.txt", "r")
2     data = file_a.read()
3     print(data)
4     file_a.close()
```

輸出結果

coffee 100

tea 80

juice 80

milk 70

water 60

milktea 110

cake 60

oolong 50

```
1     file_a = open("file_name_XXX.txt", "r")
2     data = file_a.read(20)
3     print(data)
4     file_a.close()
```

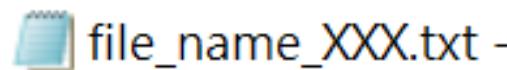
輸出結果

coffee 100

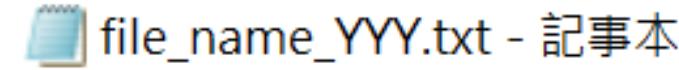
tea 80

## 作業實作

接下來做個練習，有一個檔案其內容如左圖，接著這個檔案的資料會輸入進入你的程式，程式執行後，輸出後的結果內容要是不同的檔案，其中最右邊是打完九折後的價格。



檔案(F) 編輯(E) 格式(C)		
coffee	100	
tea	80	
juice	80	
milk	70	
water	60	
milktea	110	
cake	60	
oolong	50	



檔案(F) 編輯(E) 格式(O) 檢視			
coffee	100	90.0	
tea	80	72.0	
juice	80	72.0	
milk	70	63.0	
water	60	54.0	
milktea	110	99.0	
cake	60	54.0	
oolong	50	45.0	

上述的範例都是一般開啟檔案的方式，

另外有一種開啟方式是用 with 開啟，用 with 開啟的好處是可以不用寫 close() 程式會在縮排結束後幫你關檔案。

語法：

with 指定的檔案與處理 as 檔案變數  
... # 要縮排

```
1 with open("file_name_ZZZ.txt", "w") as file_a: 1
2     file_a.write("apple\t 100\n") 2
3     file_a.write("banana\t 110\n") 3
```

輸出結果

file\_name\_ZZZ.txt - 記事  
檔案(F) 編輯(E) 格式(O) 檔  
apple 100  
banana 110

```
1 with open("file_name_ZZZ.txt", "w") as file_a: 1
2     pass 2
3     file_a.write("apple\t 100\n") 3
4     file_a.write("banana\t 110\n") 4
```

Traceback (most recent call last):  
File "D:/Python/workspace/Sample\_16\_17.py", line 3,  
 file\_a.write("apple\t 100\n")  
ValueError: I/O operation on closed file.

CSV (Comma Separated Value)，資料是以逗號為分隔的檔案格式。

許多資料來源會提供 csv 格式的檔案。

在使用前會先匯入 csv 模組

語法：

```
import csv
```

方法名稱	說明
write(檔案)	取得 writer
reader(檔案)	取得 reader
標的物變數.writerow(序列)	以一列的方式寫入 csv
標的物變數.writerows(序列)	以多列的方式寫入 csv

如下範例是寫入一列的方式：

```
1 import csv  
2  
3 file_a = open("file_name.csv", "w", newline="")  
4  
5 write_f = csv.writer(file_a)  
6 write_f.writerow(["產品名稱", "產品價格"])  
7  
8 file_a.close()
```

備註：

Python 在寫入 csv 時會自動有換行符，  
如果加上 newline= ""，  
則換行符不會發生。

輸出結果

	A	B	C
1	產品名稱	產品價格	
2			

如下範例是寫入多列的方式：

準備檔案

	A	B	C
1	產品名稱	產品價格	
2			

```
1 import csv  
2  
3 file_a = open("file_name.csv", "a", newline="")  
4  
5 write_f = csv.writer(file_a)  
6 write_f.writerow(["coffee", 100], ["tea", 90])  
7  
8 file_a.close()
```

輸出結果

	A	B	
1	產品名稱	產品價格	
2	coffee		100
3	tea		90
4			

準備檔案

	A	B	
1	產品名稱	產品價格	
2	coffee	100	
3	tea	90	
4			

```
1 import csv  
2  
3 file_a = open("file_name.csv", "a", newline="")  
4  
5 write_f = csv.writer(file_a)  
6 data = {"juice": 70, "milk": 60}  
7 for i in data:  
8     write_f.writerow([i, data[i]])  
9  
10 file_a.close()
```

輸出結果

	A	B	
1	產品名稱	產品價格	
2	coffee	100	
3	tea	90	
4	juice	70	
5	milk	60	

準備檔案

	A	B
1	產品名稱	產品價格
2	coffee	100
3	tea	90
4	juice	70
5	milk	60

輸出結果

```
read_f: <_csv.reader obj
['產品名稱', '產品價格']
['coffee', '100']
['tea', '90']
['juice', '70']
['milk', '60']
```

```
1 import csv
2
3 file_a = open("file_name.csv", "r", newline="")
4
5 read_f = csv.reader(file_a)
6 print("read_f: ", read_f)
7 for i in read_f:
8     print(i)
9
10 file_a.close()
```

Json (JavaScript Object Notation) 格式可運用在網際網路以及 NoSql 資料庫。  
Json 格式，是由之前學過的串列(list)字典(dictionary)組成的。

這個格式的最外層可以是字典或串列，  
接著字典的 value 可以是數字、字串、串列、字典等等，  
如果是串列，那麼串列內的元素值也可以是字典。  
簡單來說就是字典內有串列，串列內有字典。  
取值的方式也就和之前學過的一樣。還記得吧!

讀寫 JSON 檔案的模組如右：

函數	說明
load(檔案)	讀取 JSON 檔案
dump(檔案)	寫入 JSON 檔案

Json 格式的內容資料可以自己設計，只要符合串列和字典的原則就行。

```
1 data = {"drink": [ {"hot": {"coffee": 100, "tea": 90}},  
2                         {"juice": {"apple": 95, "banana": 85}}],  
3             "table": ["", "A01", "A02", "A03", "A04", "A05"]           輸出結果  
4         }  
5  
6     print(data["drink"][1]["juice"]["apple"])  
7     print(data["table"][3])  
95  
A03
```

寫入時要使用 json 模組的 dump()函數。

```
1 import json
2
3 data = {"drink": [{"hot": {"coffee": 100, "tea": 90}},
4                   {"juice": {"apple": 95, "banana": 85}}],
5         "table": ["", "A01", "A02", "A03", "A04", "A05"]
6     }
7
8 file_a = open("file_name_JJJ.json", "w")
9 json.dump(data, file_a)
10 file_a.close()
```

輸出結果

```
1 {
2   "drink": [
3     {
4       "hot": {
5         "coffee": 100,
6         "tea": 90
7       }
8     },
9     {
10       "juice": {
11         "apple": 95,
12         "banana": 85
13       }
14     }
15   ],
16   "table": [
17     "",
18     "A01",
19     "A02",
20     "A03",
21     "A04",
22     "A05"
23   ]
24 }
```

## 16-4-2 寫入 JSON 檔案

備註：可下載安裝 notepad ++。並用 notepad ++ 開啟 json 檔。



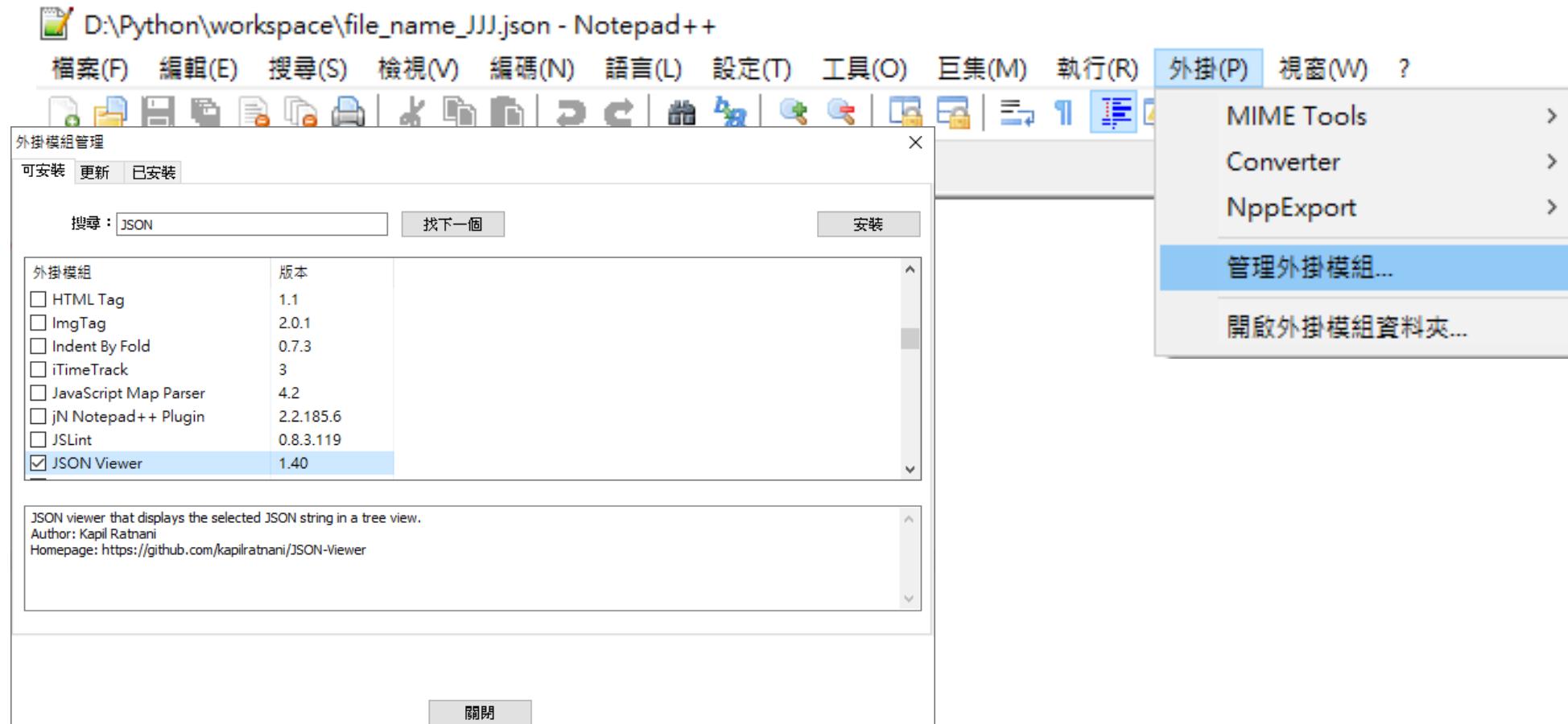
<https://notepad-plus-plus.org> › downloads

Downloads | Notepad++

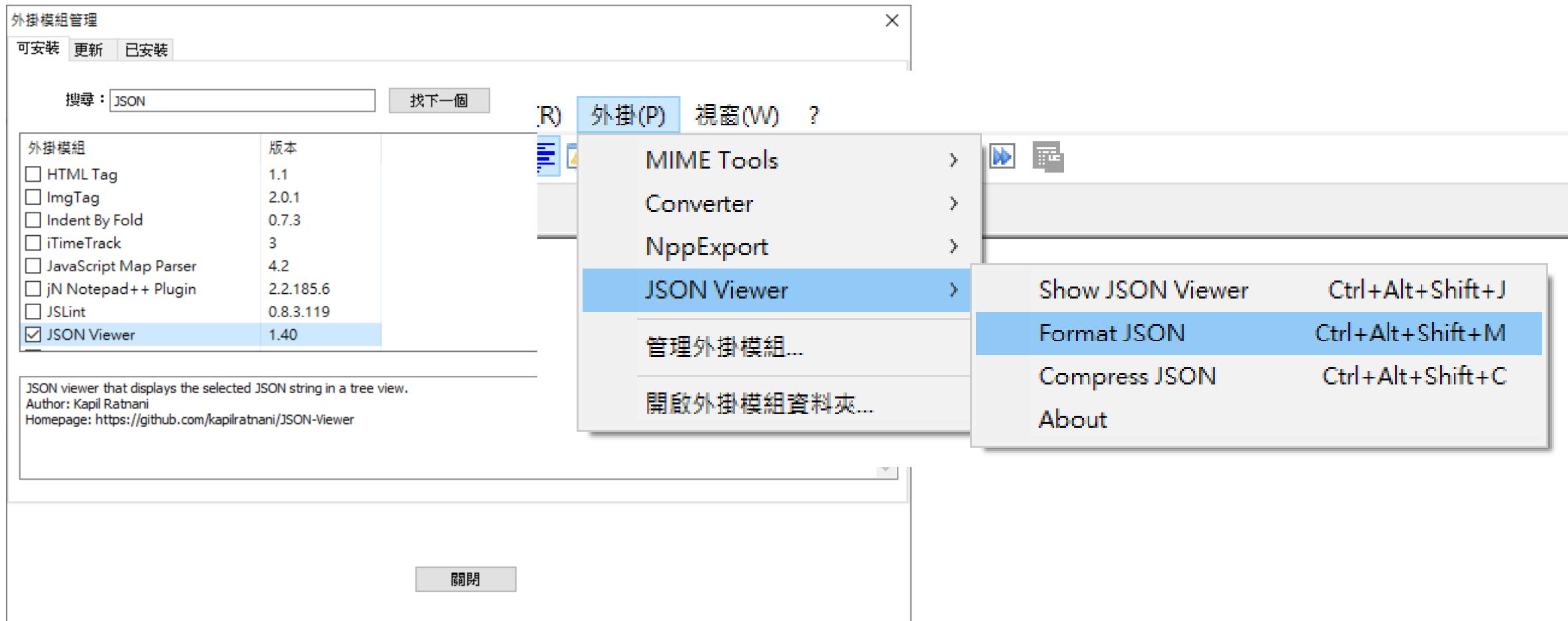
版本不拘，直接 downloads



要看 json 格式排版：外掛 >> 管理外掛模組>>可安裝>>JSON Viewer



安裝完後 外掛 >> JSON Viewer >> Format JSON



讀取時要使用 json 模組的 load() 函數。

```
1 import json  
2  
3 file_a = open("file_name_JJJ.json", "r")  
4 data = json.load(file_a)  
5 print("data: ", data)  
6 file_a.close()
```

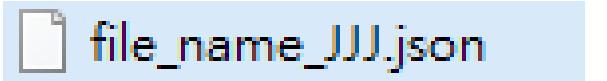
輸出結果

```
data: {'drink': [{ 'hot': { 'coffee': 100, 'tea': 90 } }, { 'jui
```

## 作業實作

接下來做個練習，首先有個資料如下左圖，其中的coffee: 100，請撰寫一個程式讓他的結果變為coffee: 90，json 檔案名稱不變。

```
1  {
2      "drink": [
3          {
4              "hot": {
5                  "coffee": 100,
6                  "tea": 90
7              }
8          },
9          {
10             "juice": {
11                 "apple": 95,
12                 "banana": 85
13             }
14         }
15     ],
16     "table": [
17         "",
18         "A01",
19         "A02",
20         "A03",
21         "A04",
22         "A05"
23     ]
24 }
```



```
1  {
2      "drink": [
3          {
4              "hot": {
5                  "coffee": 90,
6                  "tea": 90
7              }
8          },
9          {
10             "juice": {
11                 "apple": 95,
12                 "banana": 85
13             }
14         }
15     ],
16     "table": [
17         "",
18         "A01",
19         "A02",
20         "A03",
21         "A04",
22         "A05"
23     ]
24 }
```

# Module 17 :

## 例外處理

17-1 例外處理的相關知識

17-2 例外處理的敘述

17-2-1 認識內建的例外類型

17-2-2 例外處理的應用

17-3 定義例外類別

程式撰寫後，我們當然都希望他能順利執行，但如過出現例外的狀況，又應該如何在程式做特殊的處理。

舉一個例子：

假如我要讀取一個檔案，但其實這個檔案並不存在，所以跳出了 error code。但使用者不會去看 error code，他只要執行然後有結果而已。或是他只想知道怎麼做能順利執行。

```
1     file_a = open("RRR.txt", "r")
2     print(file_a)
```

輸出結果

```
Traceback (most recent call last):
  File "D:/Python/workspace/Sample_17_1.py", line 1, in <module>
    file_a = open("RRR.txt", "r")
  FileNotFoundError: [Errno 2] No such file or directory: 'RRR.txt'
```

我們先從語法來了解如何做例外處理的：

例外處理語法：

try:

    可能會發生例外的敘述

except 該例外的類別名稱:

    有發生例外時的處理敘述

else:

    沒有發生例外狀況的處理敘述

finally:

    最後一定會執行的敘述

首先，

程式會執行 try 的縮排內容，  
如果正常執行接著會去執行 else 的內容，  
接著執行 finally 的內容。

另外，

如果程式執行 try 的縮排內容時發生例外，  
那麼程式會執行 except 的內容，  
接著執行 finally 的內容。

那麼 except 後的「該例外的類別名稱」  
又是什麼？

```
1     file_a = open("RRR.txt", "r")
2     print(file_a)
Traceback (most recent call last):
  File "D:/Python/workspace/Sample_17_1.py", line 1, in <module>
    file_a = open("RRR.txt", "r")
FileNotFoundException: [Errno 2] No such file or directory: 'RRR.txt'
```

1. 這就是發生這個錯誤了例外類別

```
1     try:
2         file_a = open("RRR.txt", "r")
3     except FileNotFoundError:
4         print("Please check your file.")
```

2. 接著就可以把這個例外類別寫進來

主要的內建例外以及其衍生類別。

主要內建例外	衍生類別	說明
BaseException		所有例外的基礎類別
ModuleNotFoundError		找不到模組的錯誤
ImportError		匯入模組的錯誤
NameError		找不到名稱的錯誤
OSError		系統相關錯誤
	FileNotFoundException	檔案不存在的錯誤
	FileExistsError	檔案存在的錯誤
ArithmeticError		算數錯誤
	ZeroDivisionError	0 作為除數的錯誤
RuntimeError		未分類的錯誤

~列表還有很多，但也不用去背它。~

也可以把例外處裡的資訊印在螢幕上：

使用「`except` 例外類別 `as` 指定變數」

```
1     try:  
2         file_a = open("RRR.txt", "r")  
3     except FileNotFoundError as err_name:  
4         print(err_name)  
5         print("Please check your file.")
```

輸出結果

```
[Errno 2] No such file or directory: 'RRR.txt'  
Please check your file.
```

```
Traceback (most recent call last):  
  File "D:/Python/workspace/Sample_17_1.py", line 1, in <module>  
    file_a = open("RRR.txt", "r")  
FileNotFoundError: [Errno 2] No such file or directory: 'RRR.txt'
```

到目前為止，是否覺得左邊的結果相較於使用著來說比較知道是什麼問題。

右邊的結果對使用者或是不懂程式的人可能會覺得程式 line 1 有問題... (有 bug 拉 !~~~~~)

```
1 import json
2
3     try:
4         file_a = open("file_name_JJJ.json", "r")
5     except FileNotFoundError as er_name:
6         print(er_name)
7         print("Please check your file.")
8     else:                                輸出結果
9         data = json.load(file_a)
10        print(data["drink"][0])           { 'hot': { 'coffee': 90, 'tea': 90}}
11        file_a.close()                  ----- Program end -----
12
13    finally:
14        print("----- Program end -----")
```

除此之外，**else** 和 **finally** 的部分可以省略：  
如此一來程式正常時就會處理 **try** 的內容敘述，  
發生例外時就執行 **except**的內容敘述。

```
1 import json
2
3 try:
4     file_a = open("file_name_JJJ.json", "r")
5     data = json.load(file_a)
6     print(data["drink"][0])
7     file_a.close()
8 except FileNotFoundError as er_name:
9     print(er_name)
10    print("Please check your file.")
```

輸出結果

{ 'hot': { 'coffee': 90, 'tea': 90}}

```
1 import json
2
3 try:
4     file_a = open("file_name_JJ.json", "r")
5     data = json.load(file_a)
6     print(data["drink"][0])
7     file_a.close()
8 except FileNotFoundError as er_name:
9     print(er_name)
10    print("Please check your file.")
```

只改這裡  


輸出結果 [Errno 2] No such file or directory: 'file\_name\_JJ.json'  
Please check your file.

try...except 內還  
是可以有 try...except。  
except 可以寫不只一個。

```
1 import json
2
3
4 def read_json(file_name):
5     file_a = open(file_name, "r")
6     da = json.load(file_a)
7     file_a.close()
8     return da[ "table" ]
9
```

```
11     try:
12         data = read_json("file_name_JJJ.json")
13         print(data[1:])
14     try:
15         table_num = int(input("Please enter table number (1~5): "))
16         print(data[table_num])
17     except IndexError as err_name:
18         print("Please enter table number (1~5)")
19         print("No other number.")
20         print(err_name)
21     except ValueError as err_name:
22         print("Please enter number")
23         print("Please enter table number (1~5)")
24         print(err_name)
25
26     except FileNotFoundError as er_name:
27         print(er_name)
28         print("Please check your file.")
```



含下頁

## 輸出結果

```
[ 'A01', 'A02', 'A03', 'A04', 'A05' ]  
Please enter table number (1~5): 7  
Please enter table number (1~5)  
A03  
No other number.  
list index out of range  
  
[ 'A01', 'A02', 'A03', 'A04', 'A05' ]  
Please enter table number (1~5): f  
Please enter number  
Please enter table number (1~5)  
invalid literal for int() with base 10: 'f'
```

`except`可以把多個類型寫在一起：

```
1 import json
2
3 def read_json(file_name):...
4
5     try:
6         data = read_json("file_name_JJJ.json")
7         print(data[1:])
8
9     try:
10         table_num = int(input("Please enter table number (1~5): "))
11         print(data[table_num])
12
13     except (IndexError, ValueError) as err_name:
14         print("Please enter table number (1~5)")
15         print("No other number. & No word.")
16         print(err_name)
17
18     except FileNotFoundError as er_name:
19         print(er_name)
20         print("Please check your file.")
```

輸出結果

```
[ 'A01', 'A02', 'A03', 'A04', 'A05']
Please enter table number (1~5): 5
A05
[ 'A01', 'A02', 'A03', 'A04', 'A05']
Please enter table number (1~5): a
Please enter table number (1~5)
No other number. & No word.
invalid literal for int() with base 10: 'a'
[ 'A01', 'A02', 'A03', 'A04', 'A05']
Please enter table number (1~5): 8
Please enter table number (1~5)
No other number. & No word.
list index out of range
```

也可以自己定義例外類別：

其中 BaseException 是所有例外類別的基礎類別，還記得「繼承」把！

所以衍生類別是 My\_Name\_Is\_Exception。

另外，引發自己定義的例外是用 raise 敘述。

```
1 import json
2
3
4 def read_json(file_name):
5     file_a = open(file_name, "r")
6     da = json.load(file_a)
7     file_a.close()
8     return da["table"]
9
```

1. 自定義一個例外類別，這裡用 MyNameIsException。
2. BaseException，是所有例外類別的基礎類別。

```
11  class MyNameIsException(BaseException):  
12      def __init__(self, num, message):  
13          self.num = num  
14          self.message = message  
15  
16  o↑ 16      def __str__(self):  
17          return self.num + "\t" + self.message  
18
```



```
20     try:
21         data = read_json("file_name_JJJ.json")
22         print(data[1:])
23         try:
24             table_num = input("Please enter table number (1~5): ")
25             if table_num not in ["1", "2", "3", "4", "5"]:
26                 raise MyNameIsException(table_num, "is not 0~5")
27                 print(data[int(table_num)])
28             except MyNameIsException as er_name:
29                 print(er_name)
30             except FileNotFoundError as er_name:
31                 print(er_name)
32                 print("Please check your file.")
```

輸出結果

```
[ 'A01', 'A02', 'A03', 'A04', 'A05' ]
```

```
Please enter table number (1~5): 1
```

```
A01
```

```
[ 'A01', 'A02', 'A03', 'A04', 'A05' ]
```

```
Please enter table number (1~5): 6
```

```
6    is not 0~5
```

```
[ 'A01', 'A02', 'A03', 'A04', 'A05' ]
```

```
Please enter table number (1~5): a
```

```
a    is not 0~5
```

# Module 18 :

## 系統處理

18-1 執行系統相關的處理

18-2 OS 模組

18-3 取的指定路徑的資訊

Python 可以使用 os 模組，讓我們從程式操作 os，這樣一來不用程式跑到一半，然後在手動去操作 os。在使用前需要先匯入 os 模組。

語法：

```
import os
```

先舉個例子：

我想把某個檔案寫在某個資料夾內，但這個資料夾要先存在，  
所以我讓 python 幫我確認是否存在這個資料夾，如果沒有就讓程式幫我建立。

```
1 import os
2
3
4 save_file_dir = "file_dir"
5
6 if save_file_dir not in os.listdir("."):
7     os.mkdir("./" + save_file_dir)
8
9 data = {"table": ["", "A01", "A02", "A03", "A04"]}
10 file_a = open(save_file_dir + "/file_name_JJJ.json", "w")
11 json.dump(data, file_a)
12 file_a.close()
```

Data (D:) > Python > workspace > file\_dir

名稱

file\_name\_JJJ.json

函數	說明
<code>os.getcwd()</code>	取得現在的目錄
<code>os.remove(路徑)</code>	刪除指定的檔案
<code>os.mkdir(路徑)</code>	建立指定的目錄
<code>os.rmdir(路徑)</code>	刪除指定的目錄
<code>os.rename(變更前的名稱, 變更後的名稱)</code>	變更檔案名稱
<code>os.listdir(路徑)</code>	取得指定路徑的檔案名稱列表
<code>os.access(路徑, 模式)</code>	查詢指定路徑的存取權限(模式)
<code>os.chmod(路徑, 模式)</code>	變更指定路徑的存取權限(模式)
<code>os.getenv(環境變數名稱)</code>	取得環境變數的值
<code>os.stat(路徑)</code>	取得檔案的資訊

~如果需要其他的 os 模組就直接去看文件~

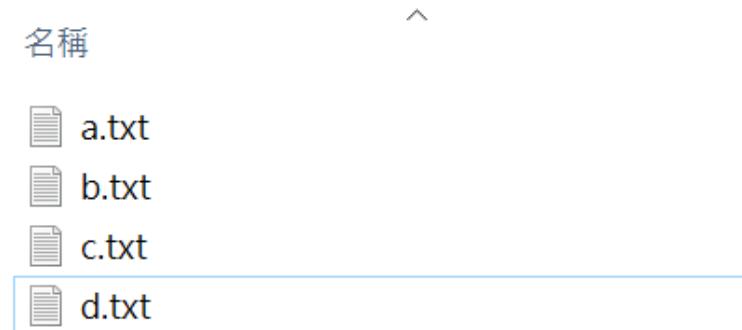
```
1 import os  
2  
3 print("取得現在目錄:", os.getcwd())
```

輸出結果

取得現在目錄: D:\Python\workspace

先準備一個目錄，  
裡面放幾個檔案

Data (D:) &gt; Python &gt; workspace &gt; os\_test



```
1 import os
2
3     path_t_1 = "D:\\Python\\workspace\\os_test"
4     print("1.\\t", os.listdir(path_t_1))
5
6     path_t_2 = "D:/Python/workspace/os_test"
7     print("2.\\t", os.listdir(path_t_2))
```

在 python 中絕對路徑的寫法有兩種：

1. 兩條反斜線(跳脫字元)
2. 一條斜線

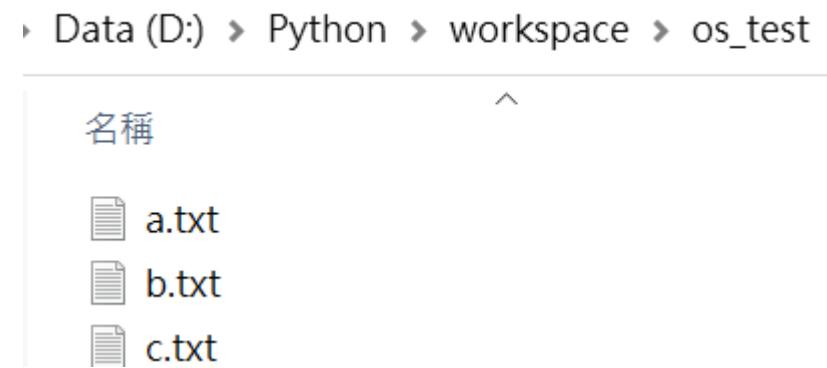
輸出結果

1. ['a.txt', 'b.txt', 'c.txt', 'd.txt']
2. ['a.txt', 'b.txt', 'c.txt', 'd.txt']

執行 remove(路徑) , 刪除指定檔案 :

```
1 import os  
2  
3 path_t_1 = "D:\Python\workspace\os_test"  
4 print("1. At", os.listdir(path_t_1))  
5  
6 os.remove(path_t_1 + "\d.txt")  
7 print("2. At", os.listdir(path_t_1))
```

輸出結果 1. ['a.txt', 'b.txt', 'c.txt', 'd.txt']  
2. ['a.txt', 'b.txt', 'c.txt']



執行 mkdir(路徑) , 建立指定資料夾 :

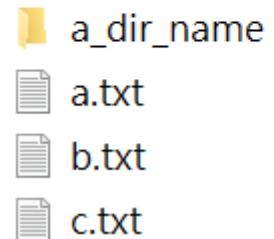
```
1 import os  
2  
3 path_t_1 = "D:\Python\workspace\os_test"  
4 print("1.\t", os.listdir(path_t_1))  
5  
6 os.mkdir(path_t_1 + "\a_dir_name")  
7 print("2.\t", os.listdir(path_t_1))
```

輸出結果

1. ['a.txt', 'b.txt', 'c.txt']
2. ['a.txt', 'a\_dir\_name', 'b.txt', 'c.txt']

Data (D:) > Python > workspace > os\_test

名稱



執行 rename(變更前名稱, 變更後名稱)，重新命名檔案或資料夾：

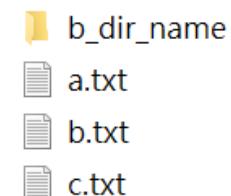
```
1 import os  
2  
3 path_t_1 = "D:\\Python\\workspace\\os_test"  
4 print("1.\\t", os.listdir(path_t_1))  
5  
6 os.rename(path_t_1 + "\\a_dir_name", path_t_1 + "\\b_dir_name")  
7 print("2.\\t", os.listdir(path_t_1))
```

輸出結果

1. ['a.txt', 'a\_dir\_name', 'b.txt', 'c.txt']
2. ['a.txt', 'b.txt', 'b\_dir\_name', 'c.txt']

Data (D:) > Python > workspace > os\_test

名稱



執行 rename(變更前名稱, 變更後名稱)，重新命名檔案或資料夾：

```
1 import os  
2  
3 path_t_1 = "D:\\Python\\workspace\\os_test"  
4 print("1.\\t", os.listdir(path_t_1))  
5  
6 os.rename(path_t_1 + "\\c.txt", path_t_1 + "\\e.txt")  
7 print("2.\\t", os.listdir(path_t_1))
```

輸出結果

1. ['a.txt', 'b.txt', 'b\_dir\_name', 'c.txt']
2. ['a.txt', 'b.txt', 'b\_dir\_name', 'e.txt']

Data (D:) > Python > workspace > os\_test

名稱

📁	b_dir_name
📄	a.txt
📄	b.txt
📄	e.txt

執行 `rmdir(路徑)`，刪除指定資料夾：

```
1 import os  
2  
3 path_t_1 = "D:\Python\workspace\os_test"  
4 print("1.\t", os.listdir(path_t_1))  
5  
6 os.rmdir(path_t_1 + "\b_dir_name")  
7 print("2.\t", os.listdir(path_t_1))
```

輸出結果

1. ['a.txt', 'b.txt', 'b\_dir\_name', 'e.txt']
2. ['a.txt', 'b.txt', 'e.txt']

Data (D:) > Python > workspace > os\_test

名稱

a.txt

b.txt

e.txt

`b_dir_name` 就刪除了。

執行 `access(路徑, 模式)`，查詢指定路徑的存取權限：

```
1 import os
2
3 path_t_1 = "D:\\Python\\workspace\\os_test"
4 print("1.\\t", os.access(path_t_1 + "\\a.txt", os.F_OK))
5 print("2.\\t", os.access(path_t_1 + "\\a.txt", os.W_OK))
6 print("3.\\t", os.access(path_t_1 + "\\a.txt", os.R_OK))
7 print("4.\\t", os.access(path_t_1 + "\\a.txt", os.X_OK))
8
9 print("5.\\t", os.access(path_t_1 + "\\d.txt", os.F_OK))
```

- 輸出結果
- 1. True
  - 2. True
  - 3. True
  - 4. True
  - 5. False

執行 chmod(路徑, 模式) , 可以變更指定路徑的存取權限：  
另外，可以變更的模式有非常非常多，所以難以列表，需要改特定模式還須查詢。

```
1 import os, stat  
2  
3 path_t_1 = "D:\Python\workspace\os_test"  
4 os.chmod(path_t_1 + "\a.txt", 0o700)  
5 print("1.\t", os.stat(path_t_1 + "\a.txt"))  
6 print("2.\t", os.access(path_t_1 + "\a.txt", os.W_OK))  
7 print("3.\t", os.access(path_t_1 + "\a.txt", os.R_OK))  
8 print("4.\t", os.access(path_t_1 + "\a.txt", os.X_OK))  
9 os.chmod(path_t_1 + "\a.txt", stat.S_IREAD)  
10 print("5.\t", os.stat(path_t_1 + "\a.txt"))  
11 print("6.\t", os.access(path_t_1 + "\a.txt", os.W_OK))  
12 print("7.\t", os.access(path_t_1 + "\a.txt", os.R_OK))  
13 print("8.\t", os.access(path_t_1 + "\a.txt", os.X_OK))
```

## 輸出結果

1. os.stat\_result(st\_mode=33206,
2. True
3. True
4. True
5. os.stat\_result(st\_mode=33060,
6. False
7. True
8. True

執行 `getenv(環境變數名稱)`，取得環境變數的值：

```
1 import os  
2  
3 key = "OS"  
4 value = os.getenv(key)  
5 print("1. At", value)  
6  
7 key = "Path"  
8 value = os.getenv(key)  
9 print("2. At", value)
```

輸出結果

1. Windows\_NT
2. D:\Python\workspace\venv\Scripts;C:\Program F

另外還有 os.path 模組，可以取得檔案或資料夾的相關資訊：

函數	說明
exists(路徑)	確認路徑是否存在
abspath(路徑)	取得絕對路徑
dirname(路徑)	取得目錄名稱
basename(路徑)	取得檔案名稱
isfile()	查詢是否為檔案
isdir()	查詢是否為目錄
getsize(路徑)	取得檔案大小
getatime(路徑)	取得最後存取時間(秒)
getmtime(路徑)	取得最後更新時間(秒)
getctime(路徑)	取得建立時間(秒)

```
1 import os.path  
2  
3 print("1.\t", os.listdir("./os_test"))  
4 print("2.\t", os.path.abspath("."))  
5 path_a = os.path.abspath(".")  
6 print("3.\t", os.path.dirname(path_a))
```

輸出結果

1. ['a.txt', 'b.txt', 'e.txt']
2. D:\Python\workspace
3. D:\Python

```
8     path_b = "D:\Python\workspace\os_test"  
9     print("4.\t", os.path.exists(path_b))  
10    print("5.\t", os.path.dirname(path_b))  
11    print("6.\t", os.path.basename(path_b))  
12    print("7.\t", os.path.isdir(path_b))  
13    print("8.\t", os.path.isfile(path_b))
```

輸出結果

- 4. True
- 5. D:\Python\workspace
- 6. os\_test
- 7. True
- 8. False

```
15 path_c = "D:\Python\workspace\os_test\a.txt"
16 print("9.", os.path.isdir(path_c))
17 print("10.", os.path.isfile(path_c))
18 print("11.", os.path.getsize(path_c))
19 print("12.", os.path.getatime(path_c))
20 print("13.", os.path.getmtime(path_c))
21 print("14.", os.path.getctime(path_c))
```

輸出結果

- 9. False
- 10. True
- 11. 0
- 12. 1635100483.3615556
- 13. 1635100483.3615556
- 14. 1635100483.3615556

12. 13. 14

顯示的秒是時間戳(timestamp)：  
以秒為單位的一個時間類型，  
起始時間為1970年1月1日0點0分0秒

# Module 19 :

## 日期與時間

19-1 處理日期與時間的資訊

19-2 指定日期時間的格式

19-3 時間 time

在程式中，同樣也包含數據分析時，我們最常會遇到資訊是時間。  
在 Python 程式中提供了 `datetime` 模組。

關於時間可以分成：

1. 建立與取得時間
2. 現在(當下)時間
3. 當下時間是在什麼時區
4. 由現在往後算的時間
5. 由現在往前算的時間
6. 時間戳
7. 取得過去時間後的處理

語法：  
`import datetime`

資料屬性/方法	說明
<code>datetime(年, 月, 日, 時, 分, 秒, 微秒, 時區)</code>	建立、取得時間日期(也可以指定年、月、日)
<code>datetime.now()</code>	取得現在日期時間
<code>datetime.today()</code>	取得現在日期
<code>datetime.fromtimestamp(時間戳)</code>	取得現在日期與時間戳
<code>datetime.strptime(日期時間字串, 格式)</code>	從指定格式的日期時間字串中取得實體物件
<code>日期時間.date()</code>	取得日期時間的 date
<code>日期時間.time()</code>	取得日期時間的 time
<code>日期時間.weekday()</code>	取得星期數
<code>日期時間.strftime(格式)</code>	取得指定格式的日期時間字串
<code>日期時間.year</code>	取得年
<code>日期時間.month</code>	取得月
<code>日期時間.day</code>	取得日
<code>日期時間.hour</code>	取得時
<code>日期時間.minute</code>	取得分
<code>日期時間.second</code>	取得秒
<code>日期時間.microsecond</code>	取得微秒
<code>日期時間.tzinfo</code>	取得時區
<code>timedelta(屬性 = 值)</code>	日期時間針對其屬性做減計算

```
1 import datetime as dt  
2  
3 print("1.\t", dt.datetime(2021, 10, 10, 12, 27, 30))  
4 print("2.\t", dt.datetime(2021, 10, 10))
```

輸出結果

1. 2021-10-10 12:27:30
2. 2021-10-10 00:00:00

現在本地時間：

```
1 import datetime as dt
2 date_time = dt.datetime.now()
3 print("1.\t現在日期與時間: ", date_time)
4 print("2.\t現在時間戳: ", date_time.timestamp())
5 print("3.\t現在日期: ", date_time.date())
6 print("4.\t現在星期數: ", date_time.weekday())
7 print("5.\t現在時間: ", date_time.time())
8 print("-" * 30)
9 print("6.\t年: ", date_time.year)
10 print("7.\t月: ", date_time.month)
11 print("8.\t日: ", date_time.day)
12 print("9.\t時: ", date_time.hour)
13 print("10.\t分: ", date_time.minute)
14 print("11.\t秒: ", date_time.second)
15 print("12.\t微秒: ", date_time.microsecond)
16 print("13.\t時區: ", date_time.tzinfo)
```

輸出結果

1. 現在日期與時間: 2021-10-25 17:32:15.735544
2. 現在時間戳: 1635154335.735544
3. 現在日期: 2021-10-25
4. 現在星期數: 0
5. 現在時間: 17:32:15.735544
- 
6. 年: 2021
7. 月: 10
8. 日: 25
9. 時: 17
10. 分: 32
11. 秒: 15
12. 微秒: 735544
13. 時區: None

```
17     print("-" * 30)
18     print("14.\t本地時間含時區: ", date_time.astimezone())
19     print("15.\t本地時間含時區: ", date_time.astimezone().tzinfo)
20     date_time_utc = dt.datetime.now(dt.timezone.utc)
21     print("16.\tUTC時間含時區: ", date_time_utc)
22     print("17.\tUTC時間含時區: ", date_time_utc.tzinfo)
```

輸出結果

14. 本地時間含時區: 2021-10-25 17:32:15.735544+08:00
15. 本地時間含時區: 台北標準時間
16. UTC時間含時區: 2021-10-25 09:32:15.735544+00:00
17. UTC時間含時區: UTC

時間的加減：

```
class datetime.timedelta  
(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)
```

```
1 import datetime as dt  
2  
3 print("1.\n", dt.timedelta(days=7))  
4 date_time = dt.datetime.now()  
5 print("2.\n", date_time)  
6 date_time_a = date_time + dt.timedelta(days=7)  
7 print("3.\n", date_time_a)  
8 date_time_b = date_time + dt.timedelta(days=30)  
9 print("4.\n", date_time_b)  
10 date_time_c = date_time + dt.timedelta(days=-6)  
11 print("5.\n", date_time_c)
```

輸出結果

1. 7 days, 0:00:00
2. 2021-10-25 17:49:06.752252
3. 2021-11-01 17:49:06.752252
4. 2021-11-24 17:49:06.752252
5. 2021-10-19 17:49:06.752252

```
1 import datetime as dt  
2  
3 date_time = dt.datetime.now()  
4 print("1.\t現在日期與時間: ", date_time)  
5  
6 date_now_stamp = date_time.timestamp()  
7 print("2.\t現在日期與時間轉成時間戳: ", date_now_stamp)  
8  
9 date_co = dt.datetime.fromtimestamp(date_now_stamp)  
10 print("3.\t時間戳轉成現在日期與時間: ", date_co)
```

輸出結果

1. 現在日期與時間: 2021-10-25 18:27:48.687116
2. 現在日期與時間轉成時間戳: 1635157668.687116
3. 時間戳轉成現在日期與時間: 2021-10-25 18:27:48.687116

上述的範例之中還有兩個方法還沒練習：

1. `datetime.strptime(日期時間字串, 格式)`
2. `日期時間.strftime(格式)`

而在練習前要先知道它的「格式」。

格式	說明	範圍/範例	格式	說明	範圍/範例
%c	時間日期		%M	分	00 ~ 59
%x	日期		%S	秒	00 ~ 59
%X	時間		%w	星期(數值)	0 (星期日) ~ 6 (星期六)
%Y	年(4位數)	2021	%A	星期(英文)	Sunday ~ Saturday
%y	年(2位數)	21	%a	星期(英文縮寫)	Sun ~ Sat
%B	月(英文)	January ~ December	%Z	時區	
%b	月(英文縮寫)	Jan ~ Dec	%p	AM/PM	
%m	月(含 0)	01 ~ 12	%j	年日數	001 ~ 366
%d	日(含 0)	01 ~ 31	%U	年週數(從星期一開始)	00 ~ 53
%H	時(24小時制)	00 ~ 23	%W	年週數(從星期日開始)	00 ~ 53
%I	時(12小時制)	01 ~ 12			

```
1 import datetime as dt
2 date_time = dt.datetime.now()
3 print("現在日期與時間: ", date_time)
4 print("1.\t", date_time.strftime("%c"))
5 print("2.\t", date_time.strftime("%x"))
6 print("3.\t", date_time.strftime("%X"))
7 print("4.\t", date_time.strftime("%Y"))
8 print("5.\t", date_time.strftime("%y"))
9 print("6.\t", date_time.strftime("%B"))
```

## 輸出結果

現在日期與時間: 2021-10-25 22:44:03.190384

1. Mon Oct 25 22:44:03 2021
2. 10/25/21
3. 22:44:03
4. 2021
5. 21
6. October

		輸出結果
10	print("7.\t", date_time.strftime("%b"))	
11	print("8.\t", date_time.strftime("%m"))	7. Oct
12	print("9.\t", date_time.strftime("%d"))	8. 10
13	print("10.\t", date_time.strftime("%H"))	9. 25
14	print("11.\t", date_time.strftime("%I"))	10. 22
15	print("12.\t", date_time.strftime("%M"))	11. 10
16	print("13.\t", date_time.strftime("%S"))	12. 44
		13. 03

		輸出結果
17	print("14.\t", date_time.strftime("%p"))	14. PM
18	print("15.\t", date_time.strftime("%w"))	15. 1
19	print("16.\t", date_time.strftime("%A"))	16. Monday
20	print("17.\t", date_time.strftime("%a"))	17. Mon
21	print("18.\t", date_time.strftime("%j"))	18. 298
22	print("19.\t", date_time.strftime("%U"))	19. 43
23	print("20.\t", date_time.strftime("%W"))	20. 43

```
25     date_time_tz = dt.datetime.now().astimezone()  
26     print("現在日期與時間含時區:", date_time_tz)  
27     print("21.\t", date_time_tz.strftime("%Z"))
```

### 輸出結果

現在日期與時間含時區: 2021-10-25 22:44:03.190384+08:00  
21. 台北標準時間

```
1 import datetime as dt
2 date_time = dt.datetime.now()
3 print("現在日期與時間: ", date_time)
4
5 a = dt.datetime.strftime(date_time, "%Y-%m-%d")
6 print("1.\t", a)
7 b = dt.datetime.strftime(date_time, "%y-%m-%d")
8 print("2.\t", b)
9 c = dt.datetime.strftime(date_time, "%H:%M:%S")
10 print("3.\t", c)
11 d = dt.datetime.strftime(date_time, "%Y-%m-%d %H:%M:%S")
12 print("4.\t", d)
```

## 輸出結果

- 現在日期與時間: 2021-10-25 22:51:28.187446
1. 2021-10-25
  2. 21-10-25
  3. 22:51:28
  4. 2021-10-25 22:51:28

剛剛學完 datetime 模組，接著學習 time 模組。

兩個模組間有許多函數是重複的，所以我們就跳過重複的地方，直接開門見山地說我們會拿 time 做那些事情。

語法：

import time

localtime() 的結果代表和 datetime 很類似但表示方法不同。

localtime() 內看得到的資料都可拿來用，取用方式如範例。

```
1 import time  
2 print(time.localtime())  
3  
4 print("1.\t", time.localtime().tm_year)  
5 print("2.\t", time.localtime().tm_hour)
```

輸出結果

```
time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1, tm_hour=22, tm_min=59, tm_sec=59, tm_wday=4, tm_yday=287, tm_isdst=0)
```

有些時候我們做爬網時會讓程式做 `time.sleep()`，讓他間隔幾秒再做動作。  
`time.sleep(5)` 表示等待時間 5 秒。

```
1 import time
2 time_a = time.time()
3 print("start time: ", time_a)
4 time.sleep(5)
5 time_b = time.time()
6 print("end time: ", time_b)
```

輸出結果

```
start time: 1635174380.7252877
end time: 1635174385.7345068
```

還記得之前學過 while 迴圈吧！還有同時介紹小撇部，在寫無窮迴圈時，一但執行下去就會跑得快，可以先放個 time.sleep() 進去，等到確認程式沒有問題再把time.sleep拿掉，或是在做秒數的調整。

```
1 import time
2
3     while True:
4         print("執行爬網相關程式", time.time())
5         time.sleep(5)
```

輸出結果

```
執行爬網相關程式 1635174670.0788016
執行爬網相關程式 1635174675.093915
執行爬網相關程式 1635174680.1083567
```

無窮會圈要小心，  
但有需求還是要用。

同樣目標的程式，時間花得越少越好，  
計算程式花多少時間或哪個 function 花多少時間，可不要左手拿碼錶右手按滑鼠。

```
1 import time
2
3 start_All = time.time()
4
5 start_a = time.time()
6 print("Do process A")
7 time.sleep(1.5)
8 end_a = time.time()
9 print("Process A track time:", end_a-start_a)
```

```
11 start_b = time.time()
12 print("Do process B")
13 time.sleep(2)
14 end_b = time.time()
15 print("Process B track time:", end_b-start_b)
16
17 end_All = time.time()
18 print("All process track time: ", end_All - start_All)
```

輸出結果

```
Do process A
Process A track time: 1.5000674724578857
Do process B
Process B track time: 2.000523328781128
All process track time: 3.5005908012390137
```

還記得之前學過如何用 python 操作 os 吧！，另外 strftime 也剛學過。  
如果我們每天都有資料要處理且依日期放置，  
那麼就可以幫你每天件新資料夾，當然新的檔案名稱也可以如法炮製。

```
1 import os
2 import time
3
4 date = time.strftime("%Y%m%d", time.localtime())
5 dir_name = "dir_AAA_" + date
6 path = "D:\\Python\\workspace\\"
7
8 if dir_name not in os.listdir(path):
9     os.mkdir(path + dir_name)
```

輸出結果



# Module 20 :

## 正規表示式

20-1 正規表示式

20-2 將字串作為模式進行檢索

20-3 搜尋字串開始與字串結束

20-4 表示任意一個字元

20-5 表示重複

20-6 最短匹配

20-7 群組化和選擇

20-8 若字串含有 meta 字元

20-9 代表字元類別

20-10 字元類別的簡易表示

20-11 對特別符號進行匹配

20-12 較複雜的正規表示式

20-13 其他正規表示式的方法

之前我們學過字串的操作，介紹了許多操作方法，接下來為了在操作字串有更大的彈性，可以更加仔細的做檢索或取代，所以我們要學習正規表示式。

使用正規表示式前要先匯入 re 模組。

語法：

```
import re
```

接著設計做檢索或取代的模式，接著做檢索或取代動作。

語法：

表示模式的變數 = re.compile(模式)

表示模式的變數.search(對象字串)

備註：

左邊的 serach 只是其中一種方法，還有其他方法之後會介紹，但在介紹前會先介紹模式的寫法。

模式會和對象字串做匹配，如果匹配成功則回傳值可以是實體物件，如果匹配不成功則會回傳 None。

```
1 import re
2
3 pattern_list = ["coffee", "tea"]
4 search_list = ["coffee", "milk"]
5
6 for pattern_value in pattern_list:
7     pattern_var = re.compile(pattern_value)
8     for search_value in search_list:
9         result = pattern_var.search(search_value)
10        print("模式:{0:^8}, 字串:{1:^8}".format(pattern_value, search_value), end="")
11        if result is not None:
12            print("匹配結果: ---- OK ----")
13        else:
14            print("匹配結果: ---- no ----")
15        print("-" * 30)
```

我們先實際操作簡單的練習：

## 輸出結果

模式: coffee , 字串: coffee 匹配結果: ----- OK -----

模式: coffee , 字串: milk 匹配結果: ----- no -----

-----

模式: tea , 字串: coffee 匹配結果: ----- no -----

模式: tea , 字串: milk 匹配結果: ----- no -----

如此一來就能拿匹配的字串做之後你想要的動作。

當然這者範例只是做簡單示範，  
pattern 還有更厲害的在後面。

Patten 可以用「meta 字元」協助我們更有彈性的設計。

meta 字元	說明
^	匹配字串開頭
\$	匹配字串結尾

```
1 import re
2
3 pattern_list = ["^ABC", "ABC$", "^ABC$"]
4 search_list = ["ABC", "ABCA", "CABC", "CABCA"]
5
6 for pattern_value in pattern_list:...
```

這裡的迴圈內容和上一範例一樣

## 20-3 搜尋字串開始與字串結束

輸出結果

模式: ^ABC , 字串: ABC 匹配結果: ----- OK -----

模式: ^ABC , 字串: ABCA 匹配結果: ----- OK -----

模式: ^ABC , 字串: CABC 匹配結果: ----- no -----

模式: ^ABC , 字串: CABCA 匹配結果: ----- no -----

-----  
模式: ABC\$ , 字串: ABC 匹配結果: ----- OK -----

模式: ABC\$ , 字串: ABCA 匹配結果: ----- no -----

模式: ABC\$ , 字串: CABC 匹配結果: ----- OK -----

模式: ABC\$ , 字串: CABCA 匹配結果: ----- no -----

-----  
模式: ^ABC\$ , 字串: ABC 匹配結果: ----- OK -----

模式: ^ABC\$ , 字串: ABCA 匹配結果: ----- no -----

模式: ^ABC\$ , 字串: CABC 匹配結果: ----- no -----

模式: ^ABC\$ , 字串: CABCA 匹配結果: ----- no -----

meta 字元	說明
.	匹配1個任意字元

```
1 import re
2
3 pattern_list = [".ABC", "..ABC", ".ABC."]
4 search_list = ["ABC", "ABCA", "CABC", "CABCA"]
5
6 for pattern_value in pattern_list:...
```

這裡的迴圈內容和上一範例一樣

## 20-4 表示任意一個字元

輸出結果

模式: .ABC , 字串: ABC 匹配結果: ----- no -----

模式: .ABC , 字串: ABCA 匹配結果: ----- no -----

模式: .ABC , 字串: CABC 匹配結果: ----- OK -----

模式: .ABC , 字串: CABCA 匹配結果: ----- OK -----

-----  
模式: ..ABC , 字串: ABC 匹配結果: ----- no -----

模式: ..ABC , 字串: ABCA 匹配結果: ----- no -----

模式: ..ABC , 字串: CABC 匹配結果: ----- no -----

模式: ..ABC , 字串: CABCA 匹配結果: ----- no -----

-----  
模式: .ABC. , 字串: ABC 匹配結果: ----- no -----

模式: .ABC. , 字串: ABCA 匹配結果: ----- no -----

模式: .ABC. , 字串: CABC 匹配結果: ----- no -----

模式: .ABC. , 字串: CABCA 匹配結果: ----- OK -----

meta 字元	說明
*	重覆0次以上
+	重覆1次以上
?	重覆0次或1次
{a}	重覆a次
{a,b}	重覆a~b次

注意：a, b 之間沒有空白

```
1 import re
2
3 pattern_list = ["A*", "A+", "A?"]
4 search_list = ["B", "A", "AA"]
5
6 for pattern_value in pattern_list:...
```

## 輸出結果

模式: A\*, 字串: B 匹配結果: ----- OK -----

模式: A\*, 字串: A 匹配結果: ----- OK -----

模式: A\*, 字串: AA 匹配結果: ----- OK -----

-----  
模式: A+, 字串: B 匹配結果: ----- no -----

模式: A+, 字串: A 匹配結果: ----- OK -----

模式: A+, 字串: AA 匹配結果: ----- OK -----

-----  
模式: A?, 字串: B 匹配結果: ----- OK -----

模式: A?, 字串: A 匹配結果: ----- OK -----

模式: A?, 字串: AA 匹配結果: ----- OK -----

注意：  
重複 0 次和重複 0 次以上的結果，  
因為 0 次，所以會匹配到。

## 20-5 表示重複

```
1 import re  
2  
3 pattern_list = ["A{2}", "A{2,}", "A{2,3}"]  
4 search_list = ["A", "AA", "AAA"]  
5  
6 for pattern_value in pattern_list:
```

## 輸出結果

模式: A{2}	, 字串:	A	匹配結果:	-----	no	-----
模式: A{2}	, 字串:	AA	匹配結果:	-----	OK	-----
模式: A{2}	, 字串:	AAA	匹配結果:	-----	OK	-----
<hr/>						
模式: A{2,}	, 字串:	A	匹配結果:	-----	no	-----
模式: A{2,}	, 字串:	AA	匹配結果:	-----	OK	-----
模式: A{2,}	, 字串:	AAA	匹配結果:	-----	OK	-----
<hr/>						
模式: A{2,3}	, 字串:	A	匹配結果:	-----	no	-----
模式: A{2,3}	, 字串:	AA	匹配結果:	-----	OK	-----
模式: A{2,3}	, 字串:	AAA	匹配結果:	-----	OK	-----

```
1 import re
2
3 pattern_list = ["A*?", "A+?"]
4 search_list = ["B", "AB", "ABC", "BBC", "CCA"]
5
6 for pattern_value in pattern_list:
7     pattern_var = re.compile(pattern_value)
8     for search_value in search_list:
9         result = pattern_var.search(search_value)
10        print("模式:{0:^8}, 字串:{1:^8}".format(pattern_value, search_value), end="")
11        if result is not None:
12            print("匹配結果: ---- OK ----", result)
13        else:
14            print("匹配結果: ---- no ----")
15        print("-" * 30)
```

meta 字元	說明
*?	重複 0 次以上之中最短部分
+?	重複 1 次以上之中最短部分

## 輸出結果

模式: A\*? , 字串: B 匹配結果: ----- OK ----- <re.Match object; span=(0, 0), match='>

模式: A\*? , 字串: AB 匹配結果: ----- OK ----- <re.Match object; span=(0, 0), match='>

模式: A\*? , 字串: ABC 匹配結果: ----- OK ----- <re.Match object; span=(0, 0), match='>

模式: A\*? , 字串: BBC 匹配結果: ----- OK ----- <re.Match object; span=(0, 0), match='>

模式: A\*? , 字串: CCA 匹配結果: ----- OK ----- <re.Match object; span=(0, 0), match='>

-----  
模式: A+? , 字串: B 匹配結果: ----- no -----

模式: A+? , 字串: AB 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='A'>

模式: A+? , 字串: ABC 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='A'>

模式: A+? , 字串: BBC 匹配結果: ----- no -----

模式: A+? , 字串: CCA 匹配結果: ----- OK ----- <re.Match object; span=(2, 3), match='A'>

最短匹配中 \*? 就算是重複 0 次都會匹配  
+? 匹配到的值是最短有匹配到的地方

meta 字元	說明
()	群組
	或著

```
1 import re
2
3 pattern_list = ["(AOA)", "AOAOAO"]
4 search_list = ["B", "AB", "AOA", "OAO", "QAOA"]
5
6 for pattern_value in pattern_list:...
```

## 輸出結果

模式: (AOA) , 字串: B 匹配結果: ----- no -----

模式: (AOA) , 字串: AB 匹配結果: ----- no -----

模式: (AOA) , 字串: AOA 匹配結果: ----- OK -----

模式: (AOA) , 字串: OAO 匹配結果: ----- no -----

模式: (AOA) , 字串: OAOA 匹配結果: ----- OK -----  
-----

模式: AOA|OAO , 字串: B 匹配結果: ----- no -----

模式: AOA|OAO , 字串: AB 匹配結果: ----- no -----

模式: AOA|OAO , 字串: AOA 匹配結果: ----- OK -----

模式: AOA|OAO , 字串: OAO 匹配結果: ----- OK -----

模式: AOA|OAO , 字串: OAOA 匹配結果: ----- OK -----

如果搜尋的字串中含有 meta 字元，那就在字元前面加入「\」。

```
1 import re
2
3 pattern_list = [r"(\.py)$", "(\.py)$", r"(\.py)$"]
4 search_list = ["b.py", "A.exe", ".py", "apy"]
5
6 for pattern_value in pattern_list:...
```

備註：

字串前加入，「r」是正規表示式的轉完原始字符。

## 20-8 若字串含有 meta 字元

輸出結果

模式:(\.\py)\\$ , 字串: b.py 匹配結果: ----- OK -----

模式:(\.\py)\\$ , 字串: A.exe 匹配結果: ----- no -----

模式:(\.\py)\\$ , 字串: .py 匹配結果: ----- OK -----

模式:(\.\py)\\$ , 字串: apy 匹配結果: ----- no -----

-----  
模式:(\.\py)\\$ , 字串: b.py 匹配結果: ----- OK -----

模式:(\.\py)\\$ , 字串: A.exe 匹配結果: ----- no -----

模式:(\.\py)\\$ , 字串: .py 匹配結果: ----- OK -----

模式:(\.\py)\\$ , 字串: apy 匹配結果: ----- no -----

-----  
模式: (\.\py)\\$ , 字串: b.py 匹配結果: ----- OK -----

模式: (\.\py)\\$ , 字串: A.exe 匹配結果: ----- no -----

模式: (\.\py)\\$ , 字串: .py 匹配結果: ----- OK -----

模式: (\.\py)\\$ , 字串: apy 匹配結果: ----- OK -----

## 20-9 代表字元類別

表示字元類別的方式是用「[]」中括號，括號內的字元是要匹配到的字元，  
如果不匹配到括號內的字元則加入「^」，以及加入「-」可以選擇一個區間。

模式	說明	符合的範例
[135]	1、3、5 中的任一字元	3
[0-9]	0~9 中的任一字元	6
[A-Z]	A-Z 中的任一字元	B
[A-Za-z]	A-Za-z 中的任一字元	h
[^012]	沒有 0、1、2 中的任一字元	4
[01][01]	00、01、10、11 中的任一組合	10
[A-Za-z][0-9]	1個英文後接1個數字	H2

輸出結果

```
1     import re
2
3     pattern_list = ["[012]", "[^012]"]
4     search_list = ["1", "12", "3"]
5
6     for pattern_value in pattern_list:...
```

模式: [012] , 字串: 1 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='1'>

模式: [012] , 字串: 12 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='1'>

模式: [012] , 字串: 3 匹配結果: ----- no -----

-----  
模式: [^012] , 字串: 1 匹配結果: ----- no -----

模式: [^012] , 字串: 12 匹配結果: ----- no -----

模式: [^012] , 字串: 3 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='3'>

```
1 import re  
2  
3 pattern_list = ["[abc]", "[^abc]"]  
4 search_list = ["a", "A", "a123"]  
5  
6 for pattern_value in pattern_list:
```

## 輸出結果

模式: [abc] , 字串: a 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='a'>  
模式: [abc] , 字串: A 匹配結果: ----- no -----  
模式: [abc] , 字串: a123 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='a'>  
-----  
模式: [^abc] , 字串: a 匹配結果: ----- no -----  
模式: [^abc] , 字串: A 匹配結果: ----- OK ----- <re.Match object; span=(0, 1), match='A'>  
模式: [^abc] , 字串: a123 匹配結果: ----- OK ----- <re.Match object; span=(1, 2), match='1'>

表記	說明
\s	空格
\S	非空格
\d	數字
\D	非數字
\w	數字或英文字
\W	非數字或英文字
\A	字串開頭
\Z	字串結尾

## 輸出結果

模式:[abc]\d , 字串: a

匹配結果: ----- no -----

模式:[abc]\d , 字串: A

匹配結果: ----- no -----

模式:[abc]\d , 字串: a123

匹配結果: ----- OK ----- &lt;re.Match object; span=(0, 2), match='a1'&gt;

-----

模式:[abc]\d{3}, 字串: a

匹配結果: ----- no -----

模式:[abc]\d{3}, 字串: A

匹配結果: ----- no -----

模式:[abc]\d{3}, 字串: a123

匹配結果: ----- OK ----- &lt;re.Match object; span=(0, 4), match='a123'&gt;

```
1 import re
2
3 pattern_list = [r"[abc]\d", r"[abc]\d{3}"]
4 search_list = ["a", "A", "a123"]
5
6 for pattern_value in pattern_list:...
```

輸出結果

```
1 import re
2
3 pattern_list = [r"^(0)[9]\d{8}", r"^(A-Z, a-z)[12]\d{8}"]
4 search_list = ["0912345678", "A123456789", "A323456789"]
5
6 for pattern_value in pattern_list:
```

模式:`^(0)[9]\d{8}`, 字串:0912345678匹配結果: ----- OK ----- <re.Match object; span=(0, 10), match='0912345678'>

模式:`^(0)[9]\d{8}`, 字串:A123456789匹配結果: ----- no -----

模式:`^(0)[9]\d{8}`, 字串:A323456789匹配結果: ----- no -----

-----  
模式:`^(A-Z, a-z)[12]\d{8}`, 字串:0912345678匹配結果: ----- no -----

模式:`^(A-Z, a-z)[12]\d{8}`, 字串:A123456789匹配結果: ----- OK ----- <re.Match object; span=(0, 10), match='A123456789'>

模式:`^(A-Z, a-z)[12]\d{8}`, 字串:A323456789匹配結果: ----- no -----

可以使用 group() 取得 match 到的資料

```
1 import re
2
3 pattern_list = [r"^[0][9]\d{8}", r"^[A-Z, a-z][12]\d{8}"]
4 search_list = ["0912345678", "A123456789", "A323456789"]
5
6 for pattern_value in pattern_list:
7     pattern_var = re.compile(pattern_value)
8     for search_value in search_list:
9         result = pattern_var.search(search_value)
10        print("模式:{0:^8}, 字串:{1:^8}".format(pattern_value, search_value), end="")
11        if result is not None:
12            print("匹配結果: ---- OK ----")
13            result_value = result.group()
14            print("取得匹配到的資料: ", result_value) ←只改這裡
15        else:
16            print("匹配結果: ---- no ----")
17        print("-" * 30)
```

輸出結果

模式: $^([0][9])\d{8}$ , 字串:0912345678匹配結果: ----- OK -----

取得匹配到的資料: 0912345678

模式: $^([0][9])\d{8}$ , 字串:A123456789匹配結果: ----- no -----

模式: $^([0][9])\d{8}$ , 字串:A323456789匹配結果: ----- no -----  
-----

模式: $^([A-Z, a-z][12])\d{8}$ , 字串:0912345678匹配結果: ----- no -----

模式: $^([A-Z, a-z][12])\d{8}$ , 字串:A123456789匹配結果: ----- OK -----

取得匹配到的資料: A123456789

模式: $^([A-Z, a-z][12])\d{8}$ , 字串:A323456789匹配結果: ----- no -----

搜尋匹配模式，取得要得部分。

模式	說明
str_a(?=str_b)	取 str_a，如果他後面有 str_b。
str_a(?!=str_b)	取 str_a，如果他後面沒有 str_b。
(?<=str_a)str_b	取 str_b，如果他前面有 str_a。
(?<!str_a)str_b	取 str_b，如果他前面沒有 str_a。

```
1 import re
2
3 pattern_list = [r"(news)\d{8}", r"(?<=news)\d{8}", r"(?<!news)\d{8}"]
4 search_list = ["abcdanews20211010abcd", "abcd20211010"]
5
6 for pattern_value in pattern_list:...
```

## 輸出結果

模式:(news)\d{8}，字串:abcdanews20211010abcd匹配結果: ----- OK -----

取得匹配到的資料: news20211010

模式:(news)\d{8}，字串:abcd20211010匹配結果: ----- no -----  
-----

模式:(?<news)\d{8}，字串:abcdanews20211010abcd匹配結果: ----- OK -----

取得匹配到的資料: 20211010

模式:(?<news)\d{8}，字串:abcd20211010匹配結果: ----- no -----  
-----

模式:(?<!news)\d{8}，字串:abcdanews20211010abcd匹配結果: ----- no -----

模式:(?<!news)\d{8}，字串:abcd20211010匹配結果: ----- OK -----

取得匹配到的資料: 20211010

```
1 import re
2
3 pattern_list = [r"\d{8}(?=news)", r"\d{8}(?!news)"]
4 search_list = ["abcda20211010newsabcd", "abcd20211010asd"]
5
6 for pattern_value in pattern_list:
```

輸出結果

模式:\d{8}(?=news)，字串:abcda20211010newsabcd匹配結果: ----- OK -----

取得匹配到的資料: 20211010

模式:\d{8}(?=news)，字串:abcd20211010asd匹配結果: ----- no -----

-----

模式:\d{8}(?!news)，字串:abcda20211010newsabcd匹配結果: ----- no -----

模式:\d{8}(?!news)，字串:abcd20211010asd匹配結果: ----- OK -----

取得匹配到的資料: 20211010

```
1 import re
2
3 pattern_list = ["[ ]", "@"]
4 search_list = ["_", "a_", "@"]
5
6 for pattern_value in pattern_list:
```

輸出結果

```
模式: [ ] , 字串: _ 匹配結果: ----- OK -----
模式: [ ] , 字串: a_ 匹配結果: ----- OK -----
模式: [ ] , 字串: @ 匹配結果: ----- no -----
-----
模式: @ , 字串: _ 匹配結果: ----- no -----
模式: @ , 字串: a_ 匹配結果: ----- no -----
模式: @ , 字串: @ 匹配結果: ----- OK -----
```

到目前為止，是否稍微了解正規表示式的語法以及使用的目的。  
比如爬網取得的資料有某種規律性，  
那麼除了字串的操作以外正規表示式提供彈性的方式，

另外，如果使用輸入某些資料，那麼程式可以在先檢查輸入的資料是否符合規範，  
例如身份證字號、帳號、信箱和電話等等。

前述的範例中有放一些身分證字號、電話的範例，  
而信箱是比較複雜，所以我們來談談正規表示式的語法用在信箱上。

```
1 import re
2
3 pattern_list = [r"^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)$"]
4 search_list = ["abcd1234@gmail.com", "abcd.1234@gmail.com",
5                 "abcd@1234@gmail.com", "ab=cd.1234@gmail.com",
6                 ".abcd1234@gmail.com", "abcd+1234@gmail.com"]
7 for pattern_value in pattern_list:...
```

## 輸出結果

模式:^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)\$, 字串:abcd1234@gmail.com 匹配結果: ----- OK -----  
模式:^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)\$, 字串:abcd.1234@gmail.com 匹配結果: ----- OK -----  
模式:^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)\$, 字串:abcd@1234@gmail.com 匹配結果: ----- no -----  
模式:^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)\$, 字串:ab=cd.1234@gmail.com 匹配結果: ----- no -----  
模式:^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)\$, 字串:.abcd1234@gmail.com 匹配結果: ----- no -----  
模式:^[^@=&=+\.\.][0-9A-Za-z\.][^@=\+]+@[a-z]+\.(com)\$, 字串:abcd+1234@gmail.com 匹配結果: ----- no -----

1. 開頭不可以是特殊符號(這個範例的特殊符號我沒全打上去)
2. 帳號內可以是數字、大寫英文、小寫英文、半形句號(.)，而顯示半形句號要用「\」。  
不可以是特殊符號，並且重複至少 1 個以上。
3. 接續第四個條件重複 1 次以上的匹配，一直到有 @ 的出現。
4. 目前想到的是以小寫：gmail、hotmail 都是英文小寫。
5. 結尾「\$」要是 .com 的群組「()」

```
pattern_list = [r"^[^@&=+\.\.](\d-zA-Za-zA-Z)[^@=+\.\.]+@[a-zA-Z]+\.(com)$"]
```

上一個範例對於信箱只須符合小寫就算是合格，  
但如果我要針對特別的信箱來源可以用比較嚴格的條件：

```
1 import re
2
3 net_list = ["gmail", "hotmail", "amail"]
4 symbol = "|"
5 net = symbol.join(net_list)
6
7 pattern_list = [r"^[^@&=+\.\.][0-9A-Za-z\.][^@=\+]+)@(" + net + ")+(.com)$"]
8 search_list = ["abcd1234@amail.com", "abcd.1234@hotmail.com", "abcd.1234@cmail.com"]
9 for pattern_value in pattern_list:...
```

## 輸出結果

模式:`^[^@&=+\.\.][0-9A-Za-z\.][^@=\+]+)@(.com)$`, 字串:`abcd1234@amail.com`匹配結果: ----- OK -----  
模式:`^[^@&=+\.\.][0-9A-Za-z\.][^@=\+]+)@(.com)$`, 字串:`abcd.1234@hotmail.com`匹配結果: ----- OK -----  
模式:`^[^@&=+\.\.][0-9A-Za-z\.][^@=\+]+)@(.com)$`, 字串:`abcd.1234@cmail.com`匹配結果: ----- no -----

之前的範例主要是為了介紹 meta 字元以及字元類別，所以只用 `search()` 當範例，不過正規化表示式還有其他的方法。

方法	說明
正規表示式. <code>search(搜尋對象的字串)</code>	在正規表示式中搜尋
正規表示式. <code>match(搜尋對象的字串)</code>	在正規表示式中搜尋(只有開始部分)
正規表示式. <code>.findall(搜尋對象的字串)</code>	在正規表示式中搜尋(匹配部分以串列方式回傳)
正規表示式. <code>sub(取代後的字串, 取代對象的字串, count = 0)</code>	取代正規表示式中匹配的部分
正規表示式. <code>split(分割對象字串, maxsplit=0)</code>	分割正規表示式中匹配的部分

使用 `sub()` 的範例如下：

當然如果要真正用在實際資料夾還要運用之前學過 python os 模組部分所學。  
`os.rename ...`

```
1 import re
2 pattern_str = r"(\.txt)"
3 dir_list = ["AAA.py", "BBB.exe", "CCC.txt", "DDD.txt"]
4 pattern_var = re.compile(pattern_str)
5 for dir_name in dir_list:
6     result = pattern_var.sub(".py", dir_name)
7     print("轉換前名稱{0:^11}, 轉換後名稱{1:^11}".format(dir_name, result))
```

輸出結果

轉換前名稱 AAA.py , 轉換後名稱 AAA.py  
轉換前名稱 BBB.exe , 轉換後名稱 BBB.exe  
轉換前名稱 CCC.txt , 轉換後名稱 CCC.py  
轉換前名稱 DDD.txt , 轉換後名稱 DDD.py

使用 `split()` 的範例如下：

```
1 import re
2
3 pattern_str = "-"
4 dir_list = ["0912-345-678", "0934-567-890"]
5 pattern_var = re.compile(pattern_str)
6 for dir_name in dir_list:
7     result = pattern_var.split(dir_name)
8     print(result)
```

輸出結果

```
['0912', '345', '678']
['0934', '567', '890']
```

```
1 import re
2
3 pattern_str = r"[-\*\+\ ]"
4 dir_list = ["886 0912-345-678", "0934*567*890"]
5 pattern_var = re.compile(pattern_str)
6 for dir_name in dir_list:
7     result = pattern_var.split(dir_name)
8     print(result)
```

輸出結果

```
['886', '0912', '345', '678']
['0934', '567', '890']
```

# Module 21： Multiprocessing 與 Multithreading

21-1 主程式的相關知識

21-2 PID 的相關知識

21-3 Multiprocessing

21-4 Multithreading

在學習 `multiprocess` 之前，我們必須先學習一個知識。  
許多其他的程式語言都規定要寫 `main`，才可以執行，  
如此一來在讀程式碼時就先找到 `main`，  
再從 `main` 開始讀取程式，比如 C 語言和 Java 就屬這類。

在 `python` 可以不用寫 `main` 便可執行，  
另外 `python` 常常會匯入其他的模組。  
而匯入其他的模組時。其他的模組內容也會執行。

## 21-1 主程式的相關知識

```
AAA.py
1 number = 50
2
3
4 def aaa():
5     print("HA! HA! HA!aaa")
6
7
8 aaa()

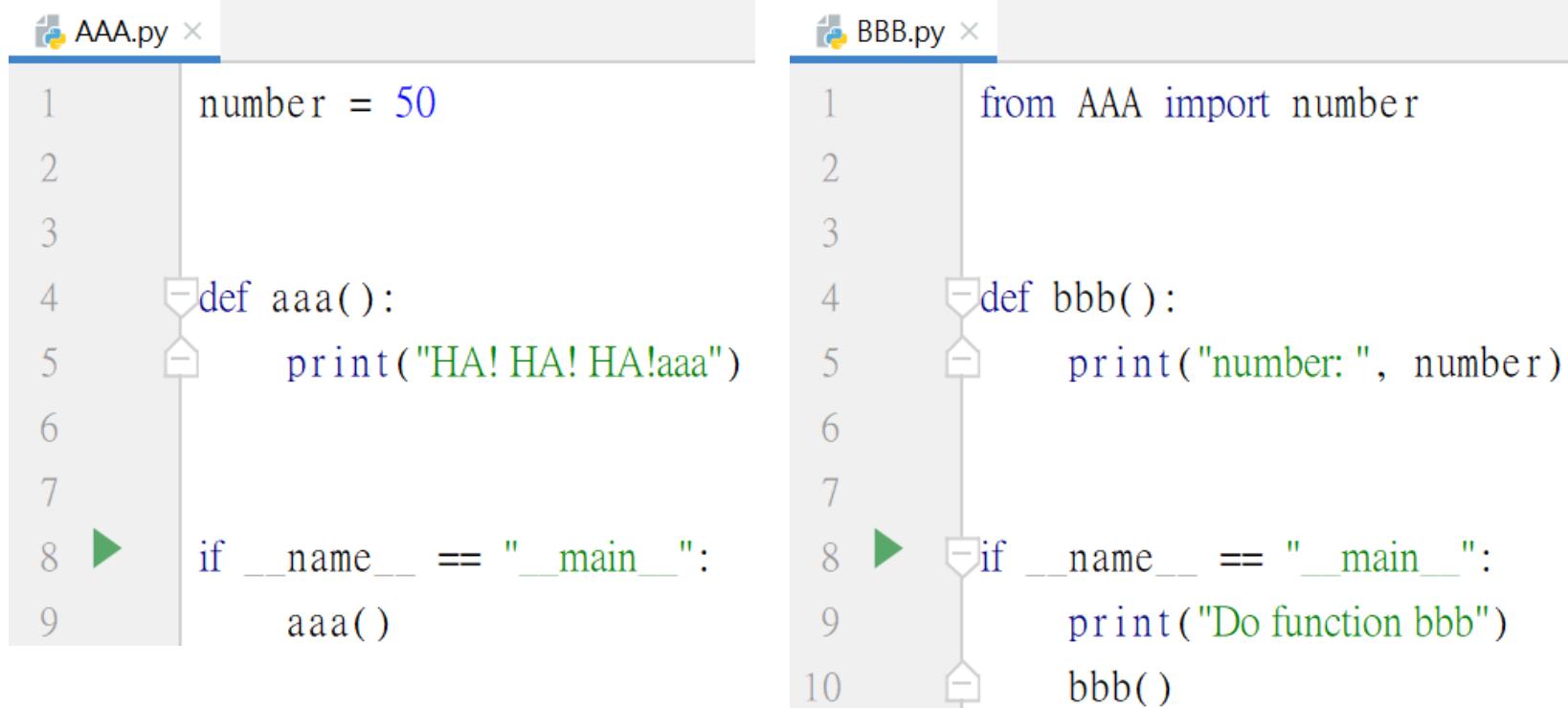
BBB.py
1 from AAA import number
2
3
4 def bbb():
5     print("number: ", number)
6
7
8     print("Do function bbb")
9     bbb()
```

執行 BBB.py 的結果如下，發現 aaa() 也會執行。

輸出結果  
HA! HA! HA!aaa  
Do function bbb  
number: 50

## 21-1 主程式的相關知識

如果我們讓程式加入 `if __name__ == '__main__':`



```
AAA.py
1     number = 50
2
3
4     def aaa():
5         print("HA! HA! HA!aaa")
6
7
8 ► if __name__ == "__main__":
9     aaa()

BBB.py
1     from AAA import number
2
3
4     def bbb():
5         print("number: ", number)
6
7
8 ► if __name__ == "__main__":
9     print("Do function bbb")
10    bbb()
```

執行 BBB.py 的結果如下，  
可以匯入 AAA.py ，  
但不會去執行 aaa 。

輸出結果

Do function bbb  
number: 50

原因是程式在我們當下執行的程式會把檔名自動帶為 main。

AAA.py

```
1 number = 50
2
3
4 def aaa():
5     print("HA! HA! HA!aaa")
6
7
8 # if __name__ == "__main__":
9 #     print("name: ", __name__)
10 aaa()
```

輸出結果 name: \_\_main\_\_  
HA! HA! HA!aaa

BBB.py

```
1 from AAA import number
2
3
4 def bbb():
5     print("number: ", number)
6
7
8 # if __name__ == "__main__":
9 #     print(__name__)
10 #     print("Do function bbb")
11 bbb()
```

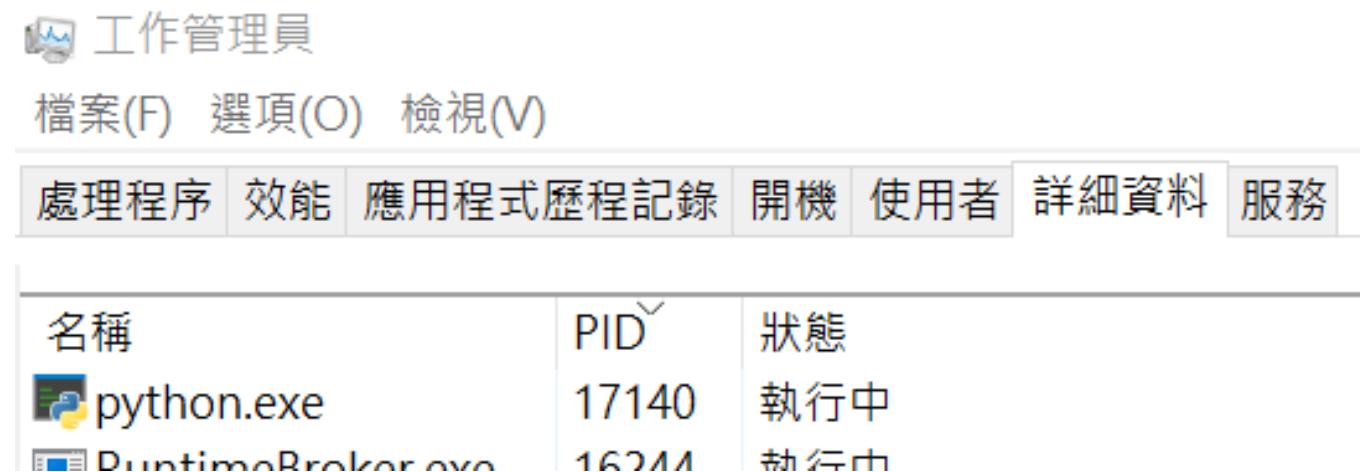
輸出結果  
name: AAA  
HA! HA! HA!aaa  
\_\_main\_\_  
Do function bbb  
number: 50

程式在執行會觸發載入記憶體中執行，藉由 PID來管理行程。  
換句話說，程式執行後在運算時期會產生 PID，  
PID 所表示的意思是 Process ID，也就是標示行程的一個數值，

```
1 import os
2
3
4
5 def get_drink():
6     print("PID:", os.getpid())
7     time.sleep(100)
8
9
10 get_drink()
```

輸出結果

PID: 17140



名稱	PID	狀態
python.exe	17140	執行中
PythonRuntimeBroker.exe	16214	執行中

Multiprocessing 和 Multithreading 很像，  
主要差別在於  
Multithreading 沒有 terminale，  
Multithreading 之間可以共享記憶體。

```
1 import os
2 import time
3
4
5 def get_drink(name, price):
6     print("PID:", os.getpid())
7     time.sleep(3)
8     print("產品名稱: {0}, 價格: {1}".format(name, price))
9
10
11 if __name__ == "__main__":
12     start_time = time.time()
13     data = {"coffee": 100, "tea": 90, "juice": 80}
14     for i in data:
15         get_drink(i, data[i])
16     end_time = time.time()
17     print("Total time: ", end_time - start_time)
```

首先我們先來寫個  
沒有 Multiprocessing 當成參考。

輸出結果 PID: 16532  
                產品名稱: coffee, 價格: 100  
                PID: 16532  
                產品名稱: tea, 價格: 90  
                PID: 16532  
                產品名稱: juice, 價格: 80  
                Total time: 9.04181981086731

我們發現程式執行的過程只在同一個 PID 運行。  
而且費時共需 9 秒。

有了以上的知識後我們來開始介紹 Multiprocessing，Multiprocessing 能夠為我們實現將工作分給不同的PID。

在使用 Multiprocessing 時，需要先匯入模組，

```
1 import multiprocessing as mp
2 import os
3 import time
4
5
6 def get_drink(name, price):
7
8
9
10
11
12 if __name__ == "__main__":
13     data = {"coffee": 100, "tea": 90, "juice": 80}
14     for i in data:
15         process_1 = mp.Process(target=get_drink, args=(i, data[i]))
16         process_1.start()
```

輸出結果

使用 start()  
開始執行 Multiprocessing 。

## 輸出結果

PID: 17272

PID: 14028

PID: 14432

產品名稱: juice, 價格: 80 產品名稱: coffee, 價格: 100

產品名稱: tea, 價格: 90

PID: 16408

PID: 16840

PID: 6776

產品名稱: juice, 價格: 80

產品名稱: coffee, 價格: 100

產品名稱: tea, 價格: 90

它會分給不同的 PID 進行運算，  
執行完後因不同資源分配，  
執行完的順序也是隨機。

計算花費多少時間：

```
1 import ...
4
5
6 def get_drink(name, price):...
10
11
12 ➤ if __name__ == "__main__":
13     start_time = time.time()
14     print("1.\tIn main PID:", os.getpid())
15     data = {"coffee": 100, "tea": 90, "juice": 80}
16     for i in data:
17         process_1 = mp.Process(target=get_drink, args=(i, data[i]))
18         process_1.start()
19     end_time = time.time()
20     print("2.\tIn main PID:", os.getpid())
21     print("total_time", end_time - start_time)
```

## 輸出結果

1. In main PID: 7148

2. In main PID: 7148

total\_time 0.015616655349731445

PID: 3352

PID: 2312

PID: 14260

產品名稱: coffee, 價格: 100 產品名稱: tea, 價格: 90 產品名稱: juice, 價格: 80

得到一個很奇怪的總費時間，為什麼!?

12 行程式以前和之前範例一樣。

```
12 ► if __name__ == "__main__":
13     start_time = time.time()
14     data = {"coffee": 100, "tea": 90, "juice": 80}
15     global process_1
16     for i in data:
17         process_1 = mp.Process(target=get_drink, args=(i, data[i]))
18         process_1.start()
19     else:
20         process_1.join()
21     end_time = time.time()
22     print("total_time", end_time - start_time)
```

使用 `.join()`，等待之前的工作都做完後，  
它會自動幫你解除，接著程式再往下執行。

```
輸出結果      PID: 16828
              PID: 11344
              PID: 7548
              產品名稱: coffee, 價格: 100
              產品名稱: tea, 價格: 90 產品名稱: juice, 價格: 80
              total_time 3.124929428100586
```

我們發現它的總費時是 3 秒，  
和之前的 9 秒相比快了需多。

可以設定一次傳幾個 PID 做運算，使用 map。

```
1  from multiprocessing import Pool  
2  
3  import os  
4  
5  
6  def get_calculate(i):  
7      print("PID: ", os.getpid())  
8      time.sleep(3)  
9      return i ** 3
```



含下頁

可以設定一次傳幾個 PID 做運算，使用 map 。

```
12 ► if __name__ == "__main__":
13     start_time = time.time()
14     data = [1, 2, 3, 4, 5, 6]
15     with Pool(processes=3) as process_1:
16         result = process_1.map(get_calculate, data)
17     print("result", result)
18     end_time = time.time()
19     print("total_time", end_time - start_time)
```

這裡設為 3 個，  
可自行調整。

```
輸出結果      PID: 8748
              PID: 6076
              PID: 14484
              PID: 6076
              PID: 14484
              PID: 8748
              result [1, 8, 27, 64, 125, 216]
              total_time 6.183637380599976
```

它會一次給三個 PID，執行完後接續，  
當然時間會增加。

```
1  from multiprocessing import Pool  
2  import os  
3  import time  
4  
5  
6  +def get_drink(name, price):...  
7  
8  
9  
10  
11  
12  ► if __name__ == "__main__":  
13      start_time = time.time()  
14      data = {"coffee": 100, "tea": 90, "juice": 80, "milk": 70, "latte": 60, "water": 50}  
15      with Pool(processes=3) as pool:  
16          for i in data:  
17              process_1 = pool.apply_async(get_drink, (i, data[i]))  
18              process_1.get()  
19      end_time = time.time()  
20      print("total_time", end_time - start_time)
```

可以設定一次傳幾個 PID 做運算，  
也可以用 `apply_async()`。

`.get()` 在 `apply_async()` 之後  
表示獲取執行結果。

傳完 3 個後接收再傳三個。

## 21-3 Multiprocessing

輸出結果

PID: 3744

PID: 236

PID: 12096

產品名稱: coffee, 價格: 100 產品名稱: tea, 價格: 90

產品名稱: juice, 價格: 80

PID: 12096

PID: 236

PID: 3744

產品名稱: latte, 價格: 60

產品名稱: milk, 價格: 70

產品名稱: water, 價格: 50

total\_time 6.151946067810059

它會一次給三個 PID，  
執行完後接續，  
時間會增加。

```
1 import multiprocessing as mp
2 import time
3
4
5 def get_count():
6     for i in range(100):
7         print(i)
8         time.sleep(1)
9
10
11 if __name__ == "__main__":
12     process_1 = mp.Process(target=get_count)
13     process_1.start()
14     time.sleep(8)
15     process_1.terminate()
```

Multiprocessing 可以用 terminate()  
來中止程序。

輸出結果	0
	1
	2
	3
	4
	5
	6
	7

Thread 和 Multiprocessing 很像，主要差別在於 Thread 沒有 terminate()，另外 thread 之間可以共享記憶體。

get\_num 中的  
PID 是察看自己的，  
而 global\_name 是看記憶體中的存放資料



```
1 import threading
2 import os
3 import time
4
5
6 def get_drink(name, price):
7     global global_name
8     global_name = name
9     print("PID: ", os.getpid())
10    time.sleep(3)
11    print("品名: {0}, 價格: {1}".format(name, price))
12
13
14 def get_num():
15     print("get_num PID: ", os.getpid(), end="\t")
16     print("In get num: ", global_name)
17     time.sleep(2)
```

```
20 ► if __name__ == "__main__":
21     start_time = time.time()
22     data = {"coffee": 100, "tea": 90, "juice": 80, "milk": 70, "water": 60}
23     global global_name
24     global process_1
25     for i in data:
26         process_1 = threading.Thread(target=get_drink, args=(i, data[i]))
27         process_2 = threading.Thread(target=get_num, )
28         process_1.start()
29         process_2.start()
30     else:
31         process_1.join()
32     end_time = time.time()
33     print("total_time", end_time - start_time)
```

Get\_num 中的 PID 是察看自己的，  
而 global\_name 是看記憶體中的存放資料

輸出結果

```
PID: 16496
get_num PID: 16496 In get num: coffee
PID: 16496
get_num PID: 16496 In get num: tea
PID: 16496
get_num PID: 16496 In get num: juice
PID: 16496
get_num PID: 16496 In get num: milk
PID: 16496
get_num PID: 16496 In get num: water
品名: milk, 價格: 70
品名: coffee, 價格: 100
品名: juice, 價格: 80
品名: tea, 價格: 90
品名: water, 價格: 60
total_time 3.0272276401519775
```

兩個 thread 都使用相同的 PID，  
另外 process\_2 可得到 process\_1 的資料

在時間上種共費時 3 秒

```
1 import multiprocessing as mp
2
3 import os
4
5 import time
6
7
8 def get_drink(name, price):
9     global num
10    num = name
11    print("PID: ", os.getpid())
12    time.sleep(5)
13    print("品名: {0}, 價格: {1}".format(name, price))
14
15
16 def get_num():
17     for i in range(3):
18         print("In get num: ", num)
19         time.sleep(2)
```

Multiprocessing 無法共享資料，  
PID 不相同。



Multiprocessing 無法共享資料，  
PID 不相同。

```
20  ► if __name__ == "__main__":
21      start_time = time.time()
22      data = {"coffee": 100, "tea": 90, "juice": 80, "milk": 70, "water": 60}
23      global process_1
24      for i in data:
25          process_1 = mp.Process(target=get_drink, args=(i, data[i]))
26          process_2 = mp.Process(target=get_num, )
27          process_1.start()
28          process_2.start()
29      else:
30          process_1.join()
31      end_time = time.time()
32      print("total_time", end_time - start_time)
```



含下頁

Multiprocessing 無法共享資料，  
PID 不相同。

```
File "D:\Python\workspace\Sample_22\Sample_22_2.py", line 17,  
    print("In get num: ", num)  
NameError: name 'num' is not defined
```

Thread 和 multiprocessing 都有 queue。queue 是能夠讓資料依序處理。

```
1 import threading
2 import queue
3 import os
4
5 def function_name():
6     while True:
7         print("PID: ", os.getpid())
8         value = queue_obj.get()
9         print("Do something {0}".format(value))
10        queue_obj.task_done()
```

1. 汇入 queue 套件

2. queue\_xxx.get() 会取得 queue\_xxx.put() 傳來的資料

3. task\_done() 会告訴列隊，該處理已完成。

```
14 queue_obj = queue.Queue()  
15 # turn-on the thread  
16 threading.Thread(target=function_name, daemon=True).start()  
17  
18 # send thirty task requests to the function_name  
19 data = ["coffee", "tea", "juice", "milk", "water"]  
20 for item in data:  
21     queue_obj.put(item)  
22  
23 # block until all tasks are done  
24 queue_obj.join()  
25 print("---- Program end ----")
```

4. 建立物件 queue\_obj。

5. 開啟對 function\_name 運作的 thread。

6. Daemon 的作用在於，當主程式結束時一並幫你把 thread 停止。

7. queue\_xxx.put() 和第 2. 點一並。

8. join()，等待之前的工作都做完後，它會自動幫你解除，接著程式再往下執行。

輸出結果

```
PID: 14268
Do something coffee
PID: 14268
Do something tea
PID: 14268
Do something juice
PID: 14268
Do something milk
PID: 14268
Do something water
PID: 14268
----- Program end -----
```

## 作業實作

1. 請用 for 迴圈撰寫一個程式，輸出結果如下：
2. 請用 while 迴圈撰寫一個程式，輸出結果如下：

Please enter a number: 5

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

輸出結果

提示:

```
print("{0:{1}}".format("*" * ( [REDACTED] ), "^\n" + str([REDACTED])))
```

程式有一筆資料如下：

```
1     performance = { "name_01": "A", "name_02": "B", "name_03": "C", "name_04": "C",
2                         "name_05": "B", "name_06": "A", "name_07": "B", "name_08": "C",
3                         "name_09": "C", "name_10": "A", "name_11": "A", "name_12": "B"}
```

請撰寫接下來的程式，分別將 A、B 和 C 歸類  
可得輸出結果如下：

```
class_A:  {'name_01': 'A', 'name_06': 'A', 'name_10': 'A', 'name_11': 'A'}
class_B:  {'name_02': 'B', 'name_05': 'B', 'name_07': 'B', 'name_12': 'B'}
class_C:  {'name_03': 'C', 'name_04': 'C', 'name_08': 'C', 'name_09': 'C'}
```

程式有一筆資料如下：

```
1     data = (31, 21, 54, 41, 62, 55, 18, 63)
2     print("原始資料:", data)
```

請撰寫接下來的程式，他會把偶數和奇數分別放到不同的串列裡，  
可得輸出結果如下：

```
原始資料: (31, 21, 54, 41, 62, 55, 18, 63)
odd_list: [31, 21, 41, 55, 63]
even_list: [54, 62, 18]
```

# Thank You!

- Any Questions?