

Python 基礎

目次

- Module 1. Python技術及開發環境介紹 3
- Module 2. 資料型態與變數 33
- Module 3. 控制結構 if...else、for、while 49
- Module 4. 字串、串列、元組、字典、集合 57
- Module 5. 函式 75
- Module 6. 物件導向程式設計 81
- Module 7. 例外處理 99
- Module 8. 檔案處理 105
- Module 9. 模組 110
- Module 10. 裝飾器 118

Module 1.

Python技術及開發環境介紹

程式的執行方式

- **直譯式語言 (Interpreted Language)**

直譯式語言在執行程式碼時，**不會事先將整個程式碼編譯成機器語言**。而是透過**直譯器 (Interpreter)** 逐行讀取和執行。這意味著每次執行程式時，程式碼都會被重新翻譯一次。

- **編譯式語言 (Compiled Language)**

編譯式語言需要透過**編譯器 (Compiler)** 將程式碼**一次性翻譯成機器語言的可執行檔案**，再執行這個可執行檔。編譯過程發生在執行之前，程式碼編譯完成後，便不再需要原始碼或編譯器來執行程式。

認識 Python 應用

- Python 是一種跨平台 (Cross-platform)、開放原始碼 (Open Source)、擁有大量社群 (Communities)、物件導向 (Object-Oriented Language) 的程式語言，由 Guido van Rossum (Python 最初設計者兼程式架構師) 在 1989 年底發明。
- Python 的語法簡潔、明確，可以整合許多工具來開發各式各樣的功能與服務，例如網站建置、資料分析與視覺化、機器學習、深度學習、視窗表單設計、自動化、遊戲開發、資料庫操作等，都是 Python 的強項。
- Python 目前主流是 3.x 版本，過去的 2.x 版本已經不再被官方維護，強烈建議大家安裝 3.x 版來進行程式開發與學習。

建置Python開發環境

- Python 的環境建置來源，通常分為
 - 官方網站 (<https://www.python.org/>)
 - Anaconda (<https://www.anaconda.com/>)
 - Docker (https://hub.docker.com/_/python)



在不同時期，某些套件與Python版本存在相容性的問題

在安裝或使用某些特定套件時，務必要留意Python版本是否相容

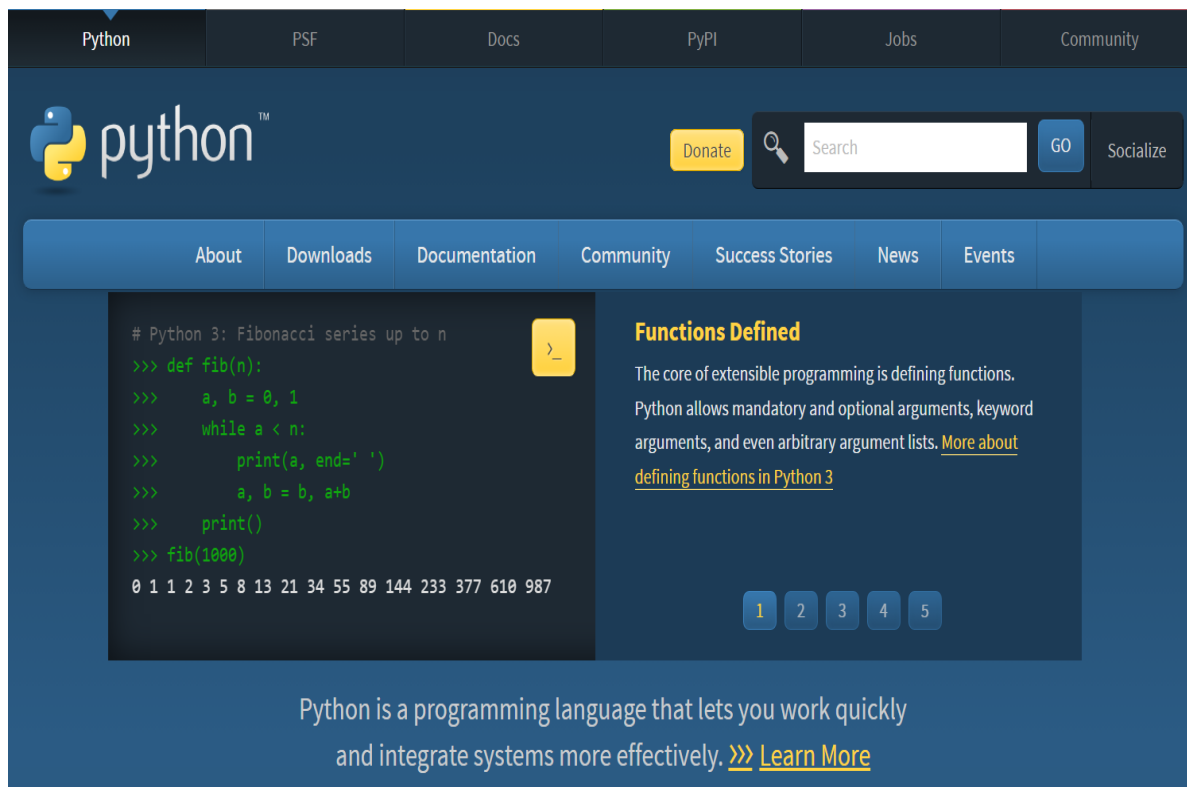
套件管理工具 pip

- pip 是 Python 的套件管理工具，可以用來安裝、更新和管理從 Python Package Index (PyPI) 以及其他第三方套件存儲庫中找到的 Python 套件。
- 安裝套件
 - `pip install <套件名稱>`
- 列出所有已安裝的套件
 - `pip list`
- 刪除套件
 - `pip uninstall <套件名稱>`

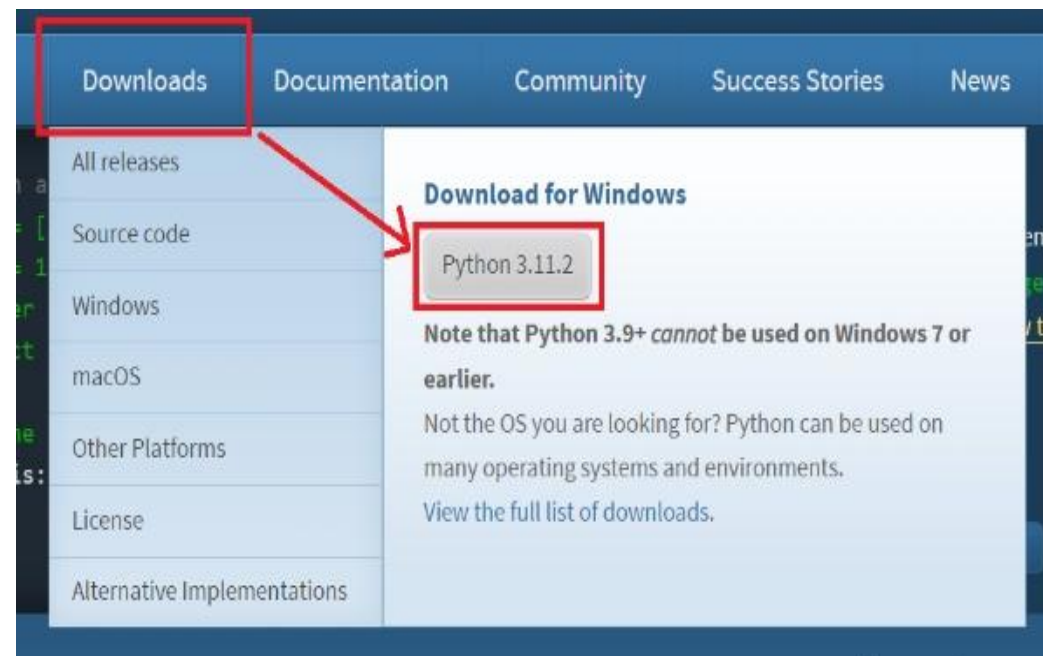
- Anaconda 也會有它的缺點，例如：
 - 下載安裝的檔案很大、時間很久，對進階的開發者而言，預先下載的工具不見得是開發者要的，換句話說，一開始會有很多短時間用不著，或是要學習到一個階段的時候，才有機會用到的工具。對於初學者來說，透過 Anaconda 建立第一個開發環境是很重要的，繁雜的流程，容易讓人望而卻步。
 - 安裝工具 conda & pip 容易讓人搞不清楚。conda 與 pip 都可以安裝開發環境當中的套件：
 - conda 主要是安裝開發的「環境」(Environment)，例如透過 conda 選定一個 python 版本，然後給予自訂名稱作為開發環境的識別，以後可以任意切換不同 python 版本的環境。
 - pip 純粹是套件安裝與管理工具，在不同的 conda 環境中，針對環境使用的 python 版本，來決定下載、安裝哪個 python 版本適用的套件。

安裝 Python 單一環境

- 官方網站 (<https://www.python.org/>)



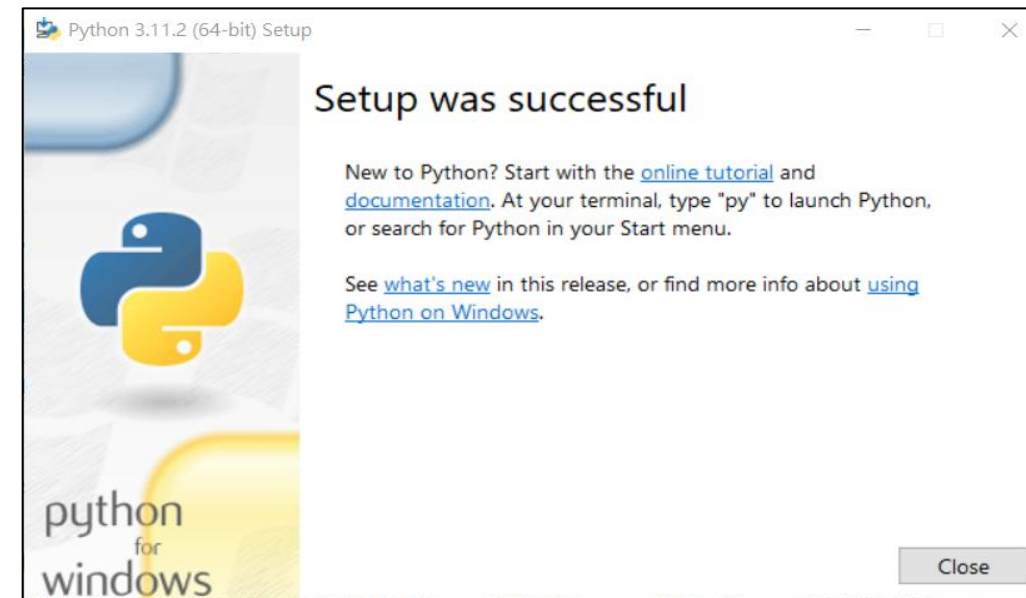
圖：python.org 首頁



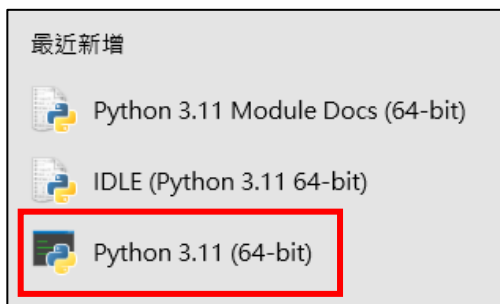
圖：游標移至 Download，
看到 Python3.xx 下載按鈕，
直接點它下載安裝程式



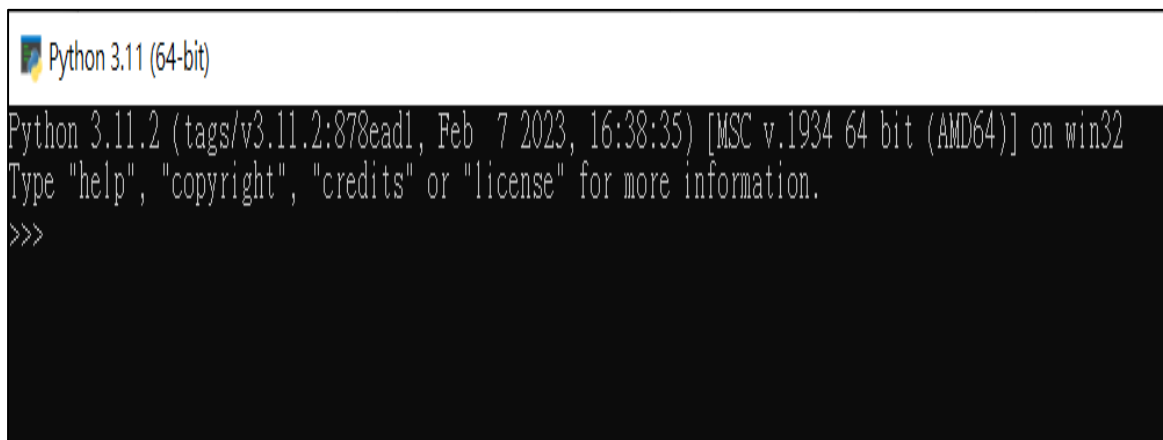
圖：打勾「Add python.exe to PATH」
按下上方的Install Now
(若有安全性提示，可以按下同意)



圖：安裝成功畫面



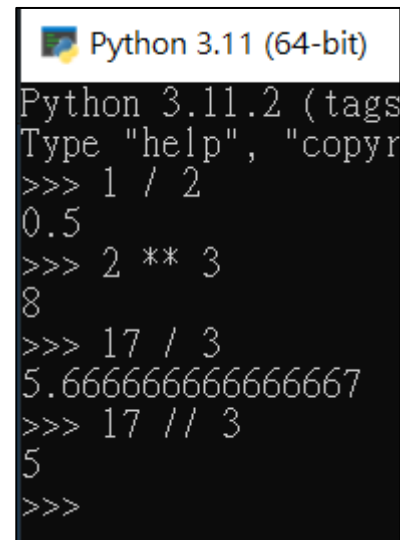
圖：到 Windows 選單，尋找最近新增，選擇「Python 3.xx (64-bit)」



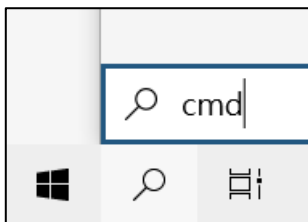
圖：Python 指令操作介面

範例

```
>>> 1 / 2 (輸入完，按下 Enter)
0.5
>>> 2 ** 3 (輸入完，按下 Enter)
8
>>> 17 / 3 (輸入完，按下 Enter)
5.666666666666667
>>> 17 // 3 (輸入完，按下 Enter)
5
```



圖：執行結果。
可輸入 `quit()` 或 `exit()`
再按 Enter 離開

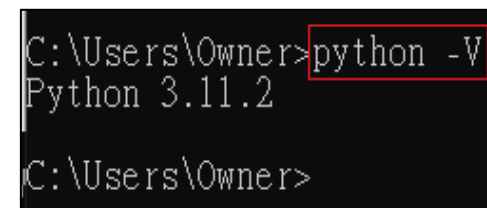


圖：搜尋 cmd

按下 Enter，進入命令提示字元

A screenshot of the Windows Command Prompt. The title bar reads '命令提示字元 - python'. The window content shows the following text: 'Microsoft Windows [版本 10.0.19045.2604] (c) Microsoft Corporation. 著作權所有，並保留一切權利。 C:\Users\Owner>python Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)] on win32 Type "help", "copyright", "credits" or "license" for more information. >>>'. The word 'python' in the command line is highlighted with a red box.

圖：在命令提示字元輸入「python」後，按下 Enter，可使用 Python

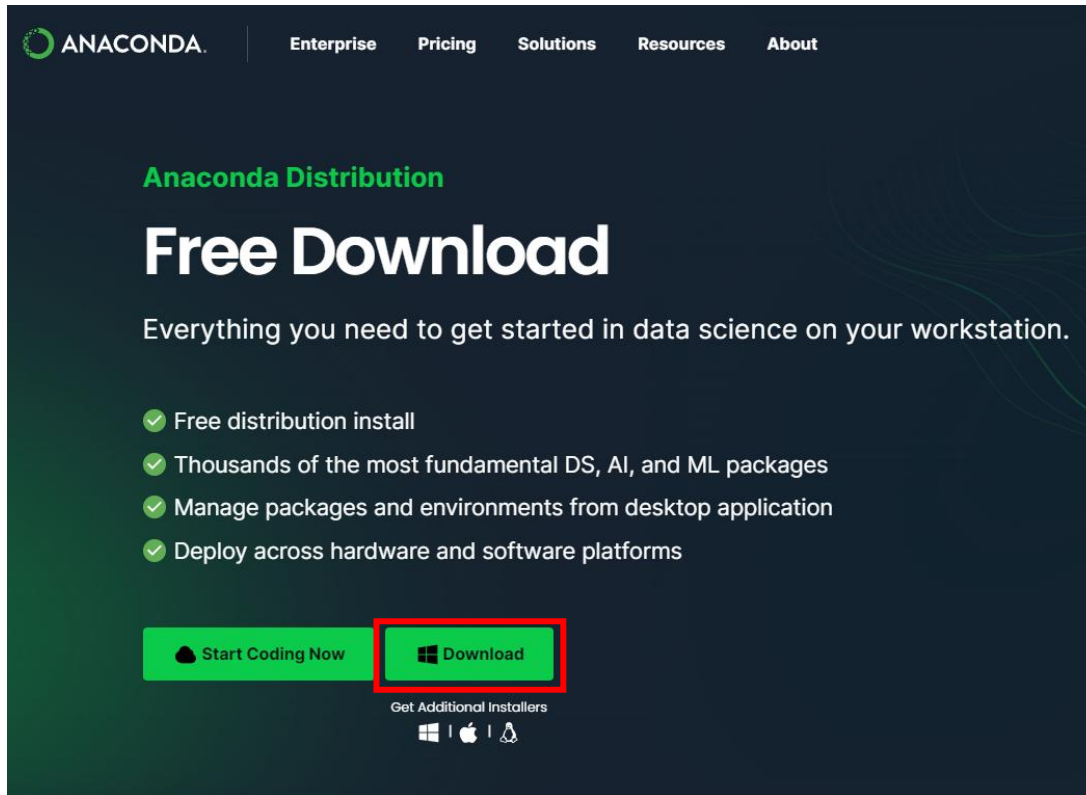


圖：可以在命令提示字元輸入「python -V」，按下 Enter，得知當前 Python 的版本

註：
若是安裝過程中，沒有打勾
「python.exe to PATH」，
無法在命令提示字元中使用
python 指令

安裝Anaconda開發軟體

- Anaconda - Individual Edition
<https://www.anaconda.com/download>

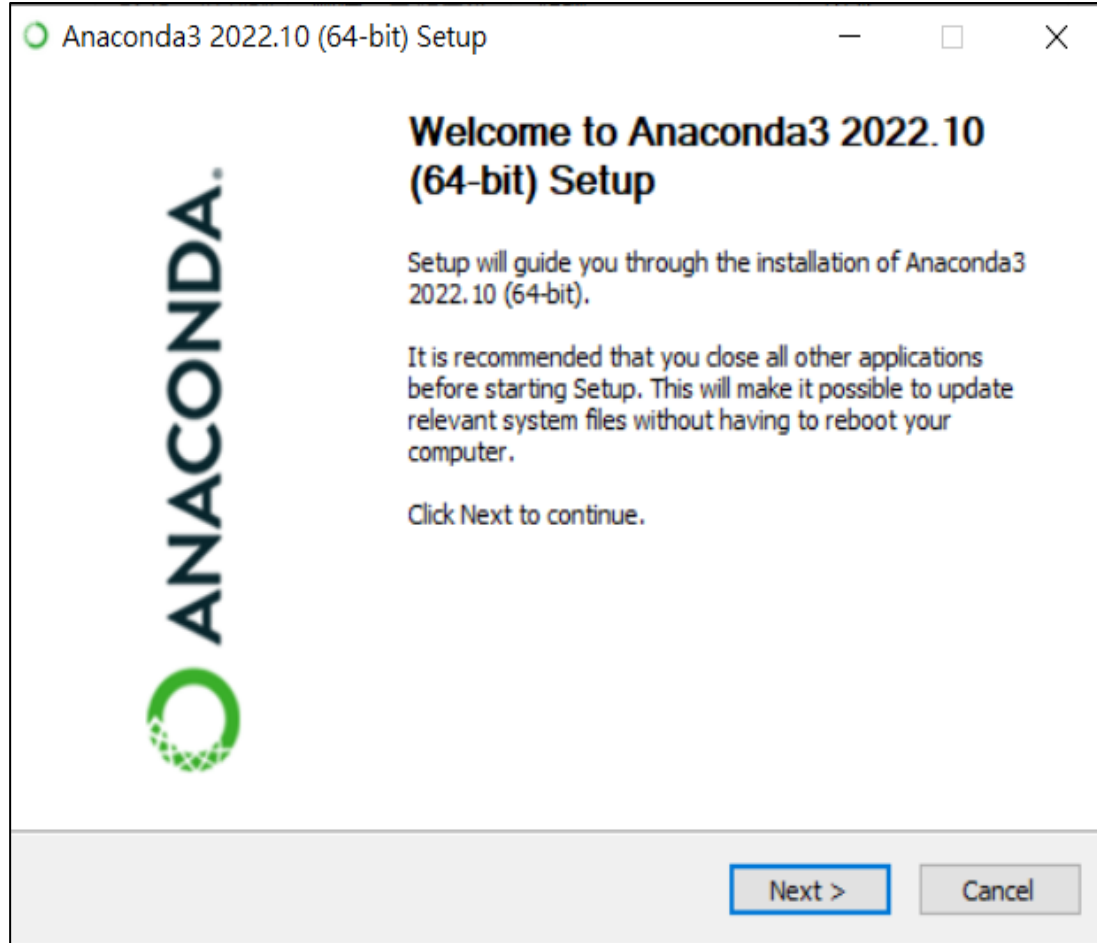


圖：至Anaconda下載頁面，選擇 Download

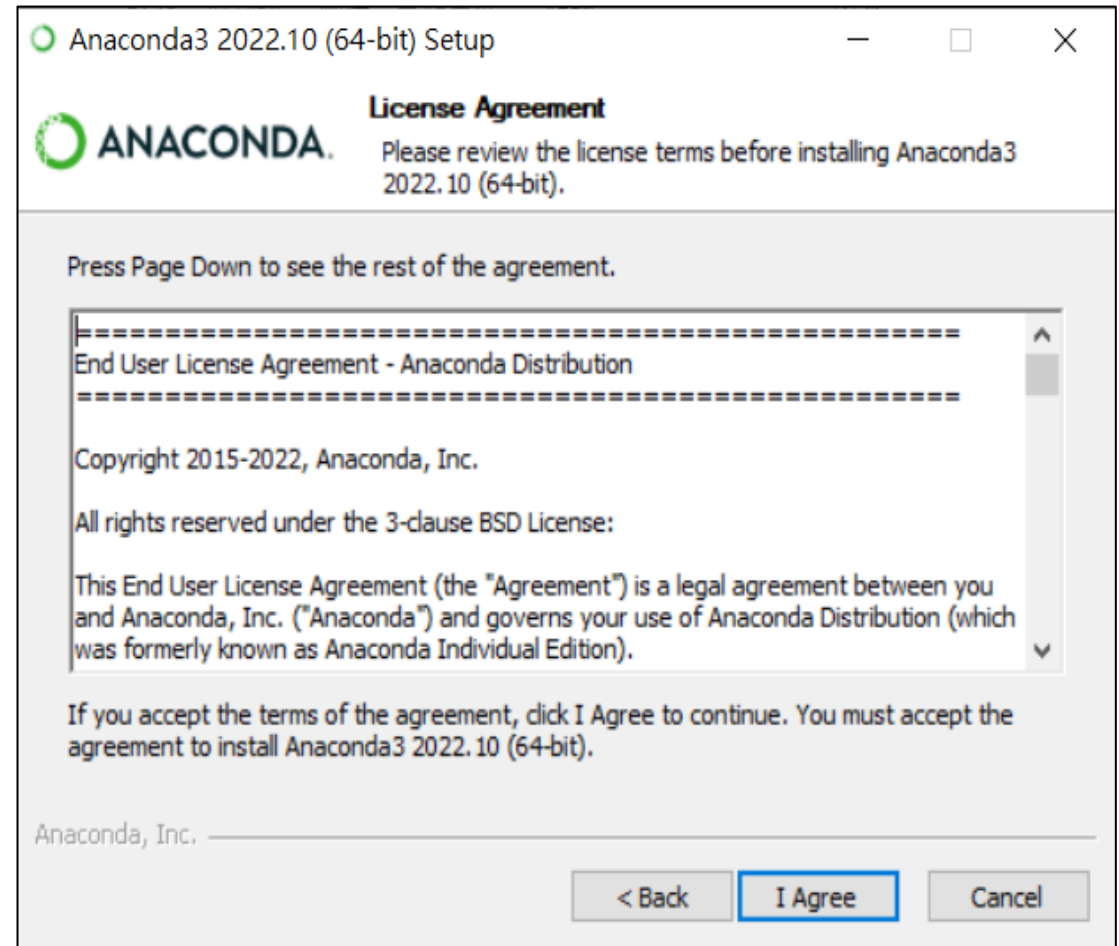


Anaconda3-202
2.10-Windows-x
86_64.exe

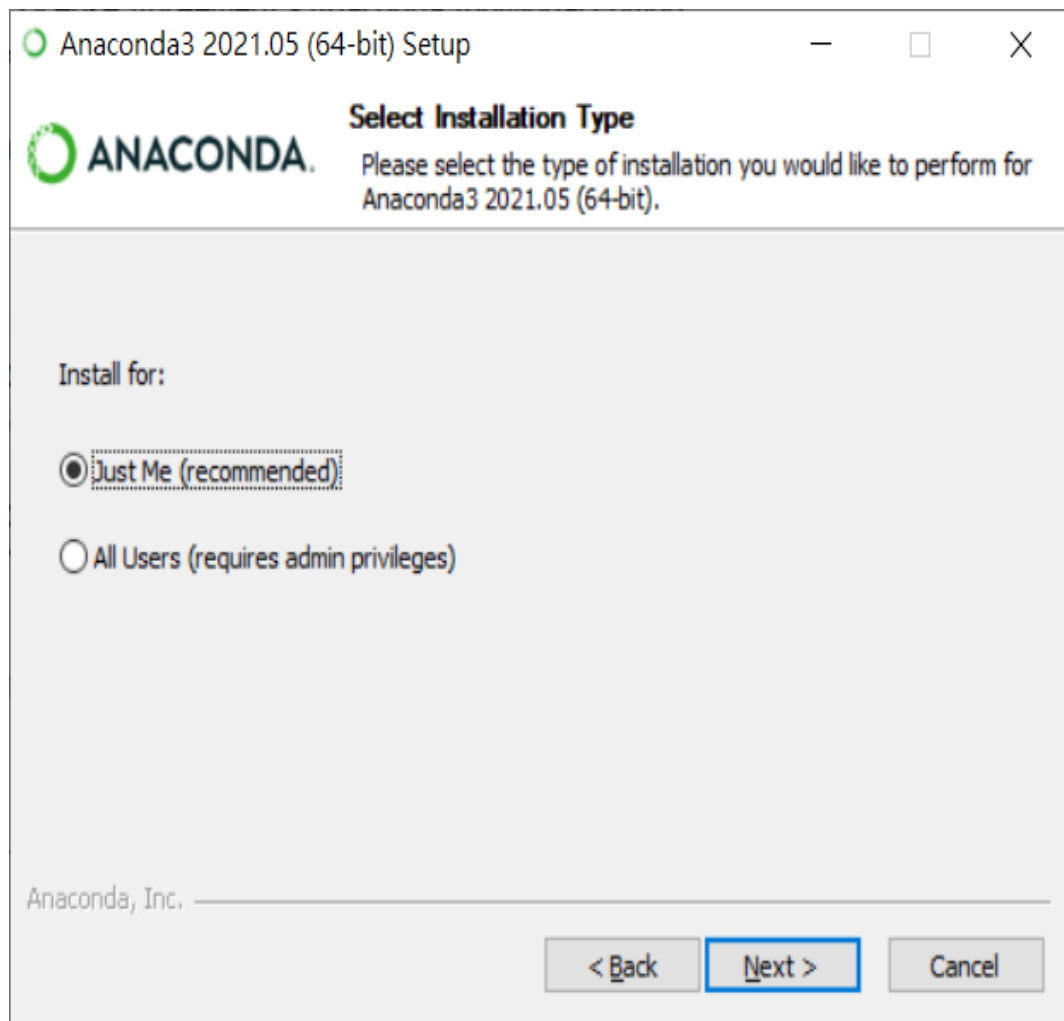
圖：下載後，快速點兩下進行安裝
有安全性警告，可按下執行或同意



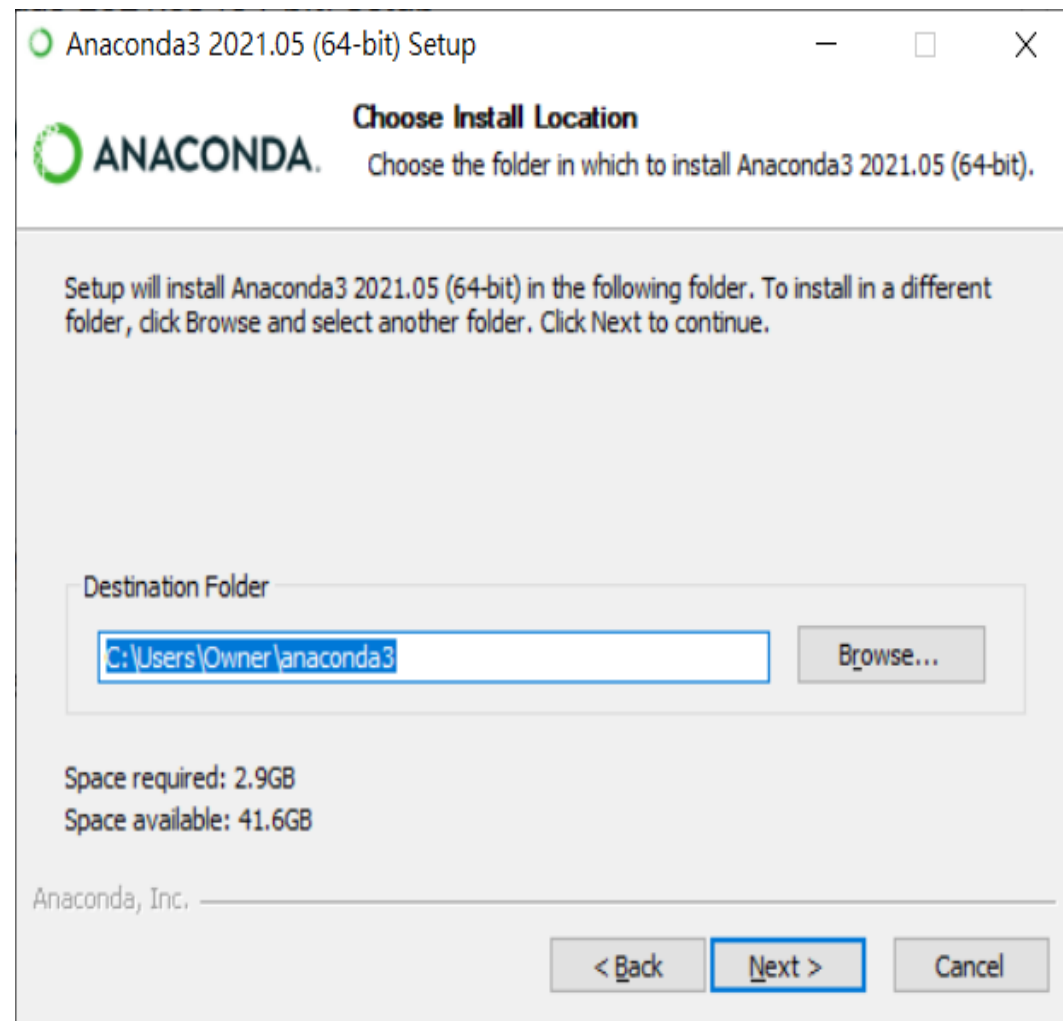
圖：按下「Next」



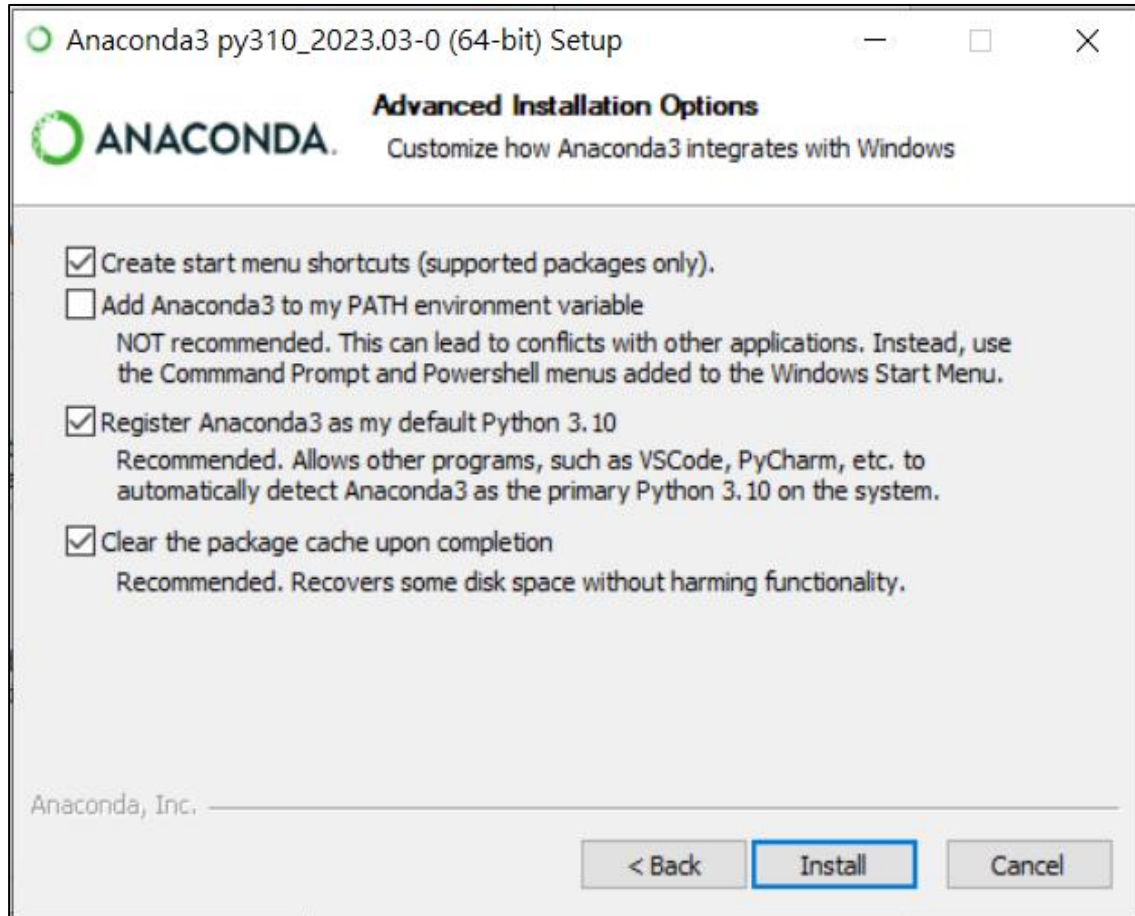
圖：按下「I Agree」



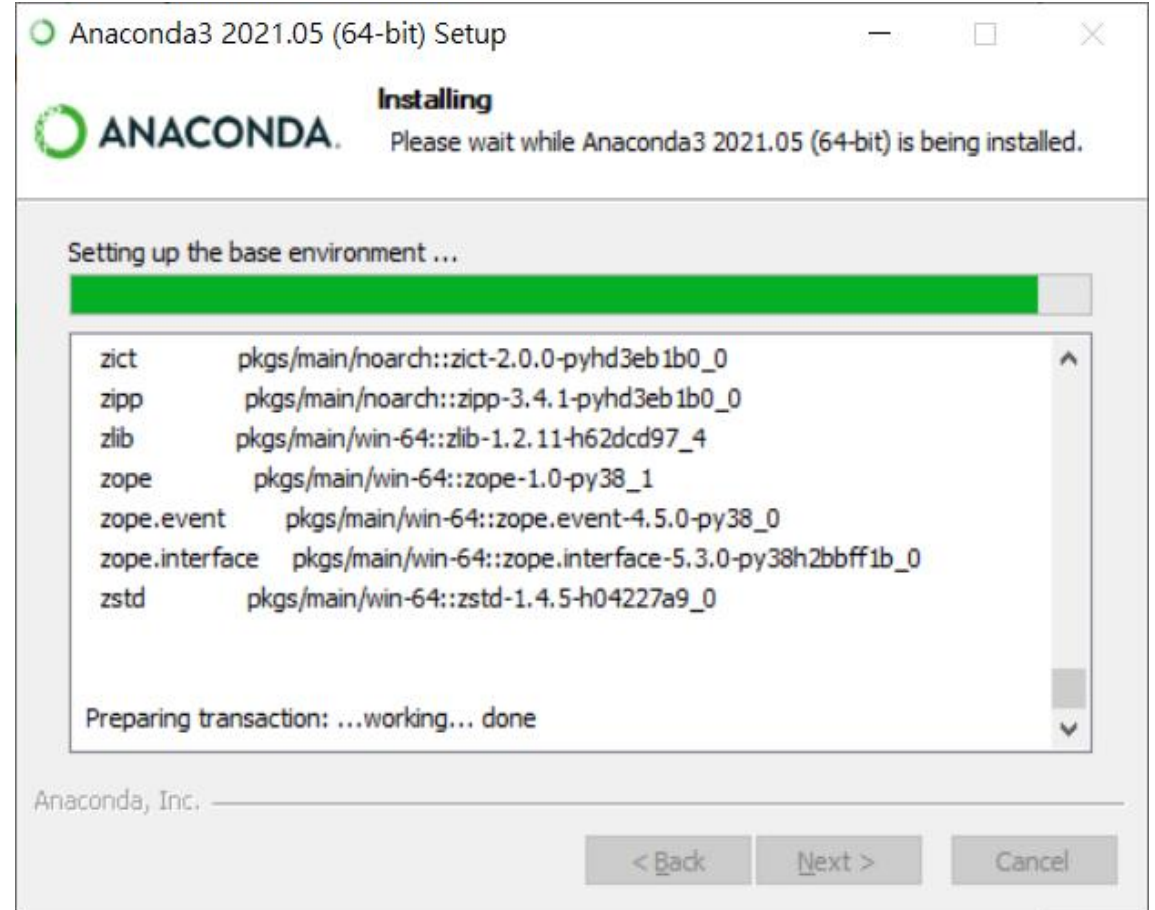
圖：依需求選擇後，按下「Next」



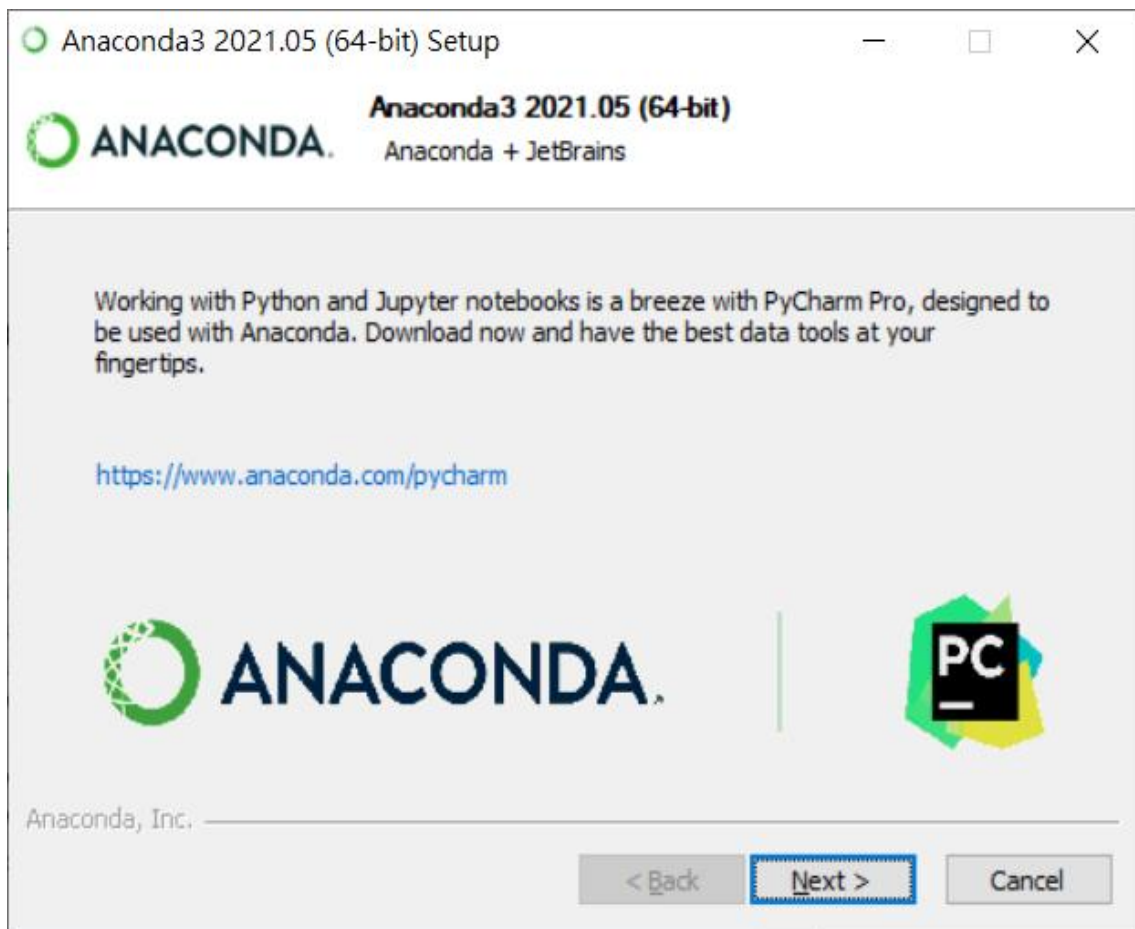
圖：安裝位置會在使用者資料夾中的 anaconda3，依需求設定，按下「Next」



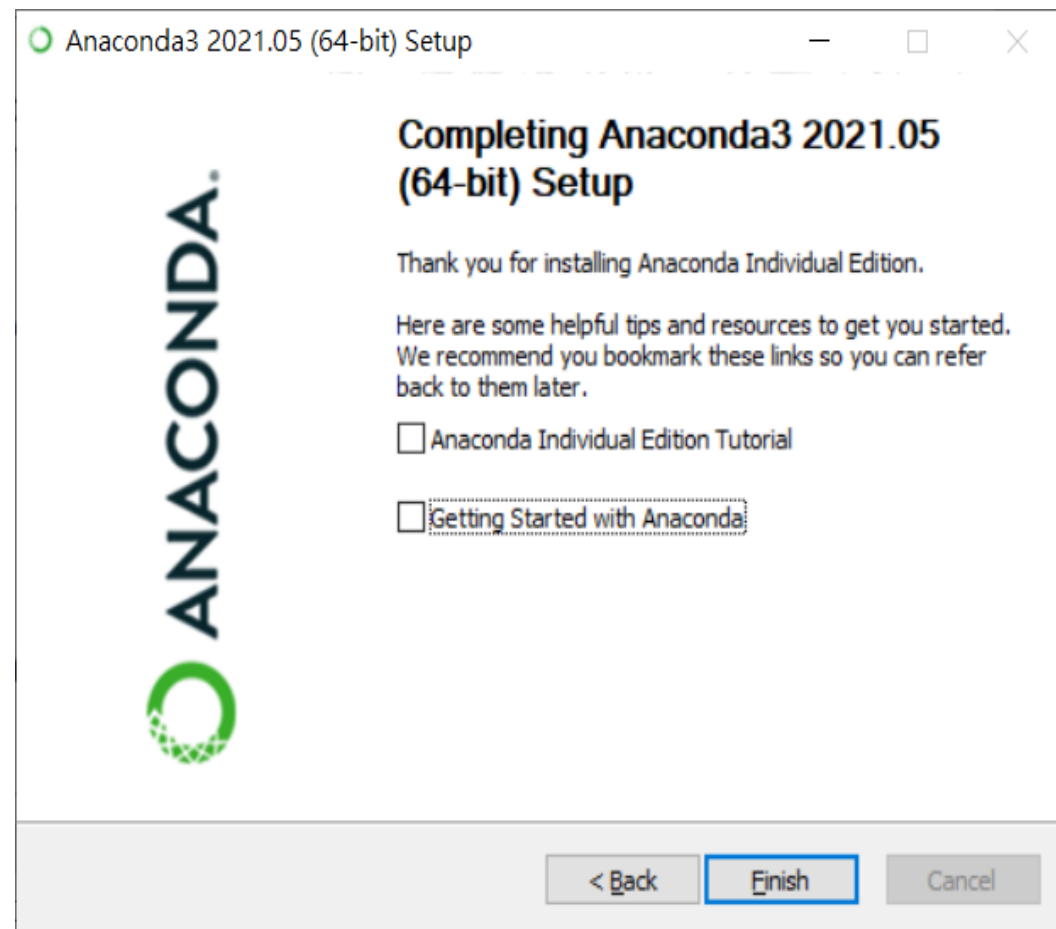
圖：依需求選擇（最後一個建議打勾，省空間），
按下「Install」，進行安裝



圖：按下「Show details」，
會看到安裝過程
完成後按下「Next」



圖：按下「Next」



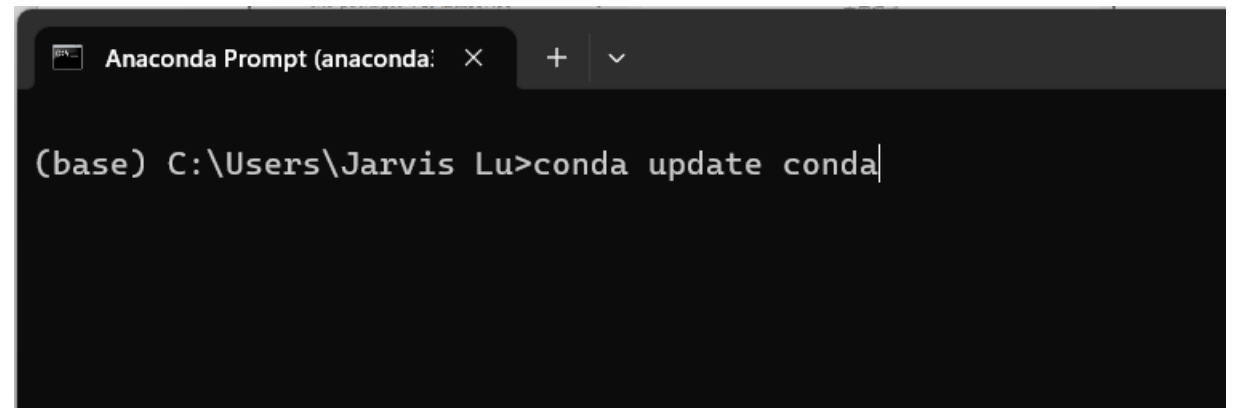
圖：取消勾選圖片中的兩個選項後，
按下「Finish」



圖：搜尋「anaconda prompt」，
按下「Anaconda Prompt (anaconda3)」



圖：出現 Anaconda Prompt，
類似 Windows 的命令提示字元



圖：輸入「conda update conda」，更新 conda 本身

```
Anaconda Prompt (anaconda3)

(base) C:\Users\Owner>conda list
# packages in environment at C:\Users\Owner\anaconda3:
#
# Name                    Version            Build    Channel
_ipyw_jlab_nb_ext_conf    0.1.0              py37_0
alabaster                  0.7.12             py37_0
anaconda                   2020.02            py37_0
anaconda-client            1.7.2              py37_0
anaconda-navigator         1.9.12             py37_0
anaconda-project           0.8.4              py_0
argh                        0.26.2             py37_0
asn1crypto                 1.3.0              py37_0
astroid                    2.3.3              py37_0
astropy                    4.0                py37he774522_0
atomicwrites               1.3.0              py37_1
attrs                      19.3.0             py_0
autopep8                   1.4.4              py_0
babel                      2.8.0              py_0
backcall                   0.1.0              py37_0
backports                  1.0                py_2
backports.functools_lru_cache 1.6.1              py_0
backports.shutil_get_terminal_size 1.0.0            py37_2
backports.tempfile         1.0                py_1
backports.weakref           1.0.post1          py_1
```

圖：輸入「conda list」，
會顯示目前預設安裝的套件
（在虛擬環境base下）

```
Anaconda Prompt (anaconda3)

werkzeug                    1.0.0              py_0
wheel                      0.34.2             py37_0
widgetsnbextension         3.5.1              py37_0
win_inet_pton              1.1.0              py37_0
win_unicode_console        0.5                py37_0
wincertstore               0.2                py37_0
winpty                     0.4.3              4
wrap                        1.11.2             py37he774522_0
xlrd                        1.2.0              py37_0
xlsxwriter                 1.2.7              py_0
xlwings                    0.17.1             py37_0
xlwt                       1.3.0              py37_0
xmldict                    0.12.0             py_0
xz                          5.2.4              h2fa13f4_4
yaml                       0.1.7              hc54c509_2
yapf                       0.28.0             py_0
zeromq                     4.3.1              h33f27b4_3
zict                       1.0.0              py_0
zipp                       2.2.0              py_0
zlib                       1.2.11             h62dc97_3
zstd                       1.3.7              h508b16e_0

(base) C:\Users\Owner>conda upgrade --all
```

圖：輸入「conda upgrade --all」，
更新當前所有套件

```
Anaconda Prompt (anaconda3) - conda upgrade --all

sphinxcontrib-qth~      1.0.2-py_0 --> 1.0.3-py_0
sphinxcontrib-ser~      1.1.3-py_0 --> 1.1.4-py_0
sphinxcontrib-web~      1.2.0-py_0 --> 1.2.1-py_0
spyder                   4.0.1-py37_0 --> 4.1.3-py37_0
spyder-kernels           1.8.1-py37_0 --> 1.9.1-py37_0
sqlalchemy               1.3.13-py37he774522_0 --> 1.3.16-py37he774522_0
sqlite                   3.31.1-he774522_0 --> 3.31.1-h2a8f88b_1
tornado                  6.0.3-py37he774522_3 --> 6.0.4-py37he774522_1
tqdm                     4.42.1-py_0 --> 4.46.0-py_0
wcwidth                  0.1.8-py_0 --> 0.1.9-py_0
werkzeug                 1.0.0-py_0 --> 1.0.1-py_0
xlswriter                1.2.7-py_0 --> 1.2.8-py_0
xlwings                  0.17.1-py37_0 --> 0.19.0-py37_0
xz                       5.2.4-h2fa13f4_4 --> 5.2.5-h62dcd97_0
zict                     1.0.0-py_0 --> 2.0.0-py_0
zipp                     2.2.0-py_0 --> 3.1.0-py_0
zlib                     1.2.11-h62dcd97_3 --> 1.2.11-h62dcd97_4

The following packages will be DOWNGRADED:

anaconda                 2020.02-py37_0 --> custom-py37_1
lzo                      2.10-h6df0209_2 --> 2.10-he774522_2

Proceed ([y]/n)?
```

圖：按下「y」後，再按鍵盤「Enter」

```
Anaconda Prompt (anaconda3)

- DEBUG menuinst_win32: __init__(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Owner\anaconda3'
, env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Owner\anaconda3\pythonw.exe, args are ['C:\Users\Owner\
anaconda3\cwp.py', 'C:\Users\Owner\anaconda3', 'C:\Users\Owner\anaconda3\pythonw.exe', 'C:\Users\Owner\ana
conda3\Scripts\spyder-script.py']
\ DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Owner\anaconda3\python.exe, args are ['C:\Users\Owner\
anaconda3\cwp.py', 'C:\Users\Owner\anaconda3', 'C:\Users\Owner\anaconda3\python.exe', 'C:\Users\Owner\ana
conda3\Scripts\spyder-script.py', '--reset']
done

(base) C:\Users\Owner>
```

圖：套件更新完成

```
Anaconda Prompt (anaconda3) - python
- DEBUG menuinst_win32:_init_(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Owner\anaconda3'
, env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Owner\anaconda3\pythonw.exe, args are ['C:\Users\Owner\
anaconda3\cwp.py', 'C:\Users\Owner\anaconda3', 'C:\Users\Owner\anaconda3\pythonw.exe', 'C:\Users\Owner\ana
conda3\Scripts\spyder-script.py']
\ DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Owner\anaconda3\python.exe, args are ['C:\Users\Owner\
anaconda3\cwp.py', 'C:\Users\Owner\anaconda3', 'C:\Users\Owner\anaconda3\python.exe', 'C:\Users\Owner\ana
conda3\Scripts\spyder-script.py', '--reset']
done

(base) C:\Users\Owner>python
Python 3.7.7 (default, May 6 2020, 11:45:54) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

圖：輸入「python」，進入 python 執行環境

```
Anaconda Prompt (anaconda3) - python
- DEBUG menuinst_win32:_init_(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Owner\anaconda3'
, env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Owner\anaconda3\pythonw.exe, args are ['C:\Users\Owner\
anaconda3\cwp.py', 'C:\Users\Owner\anaconda3', 'C:\Users\Owner\anaconda3\pythonw.exe', 'C:\Users\Owner\ana
conda3\Scripts\spyder-script.py']
\ DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Owner\anaconda3\python.exe, args are ['C:\Users\Owner\
anaconda3\cwp.py', 'C:\Users\Owner\anaconda3', 'C:\Users\Owner\anaconda3\python.exe', 'C:\Users\Owner\ana
conda3\Scripts\spyder-script.py', '--reset']
done

(base) C:\Users\Owner>python
Python 3.7.7 (default, May 6 2020, 11:45:54) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, world")
Hello, world
>>>
```

圖：輸入「print("Hello, world")」，
輸出「Hello, world」

安裝、切換與刪除 Conda 環境 (Env)

- 預設是 (base)，如果有切換環境的需求，例如手上處理著不同 Python 版本和其它相關套件的專案，需要不時切換版本來開發，此時可以建立一到多個 Conda 環境，需要的時候可以切換，不需要的時候可以刪除。
- 在執行以下的指令前，需要確認目前是否在 Anaconda Prompt 當中，或是支援 Conda 的 Terminal 環境。終端機顯示預設路徑時，最前面會有 (base)，代表目前正在預設的 Conda 環境當中。

Conda 指令

- 安裝 Conda 環境
 - **conda create --name <自訂環境名稱> [python=版本號]**
 - conda create --name python_basic python=3.10
- 列出所有 Conda 環境
 - conda env list
- 離開當前的 Conda 環境
 - conda deactivate
- 切換 / 啟動 Conda 環境
 - conda activate python_basic
- 刪除 Conda 環境
 - conda env remove -n python_basic

如何撰寫 Python 程式

- 使用 IDE

- 整合式開發環境 (Integrated Development Environment, IDE) 可以協助開發者方便使用類似記事本的程式碼輸入介面，側欄又有類似檔案總管的管理功能。常見的 Python IDE 有：

1. Visual Studio Code

2. Eclipse with PyDev

3. Sublime Text

4. Atom

5. Spyder

6. PyCharm

7. Vim

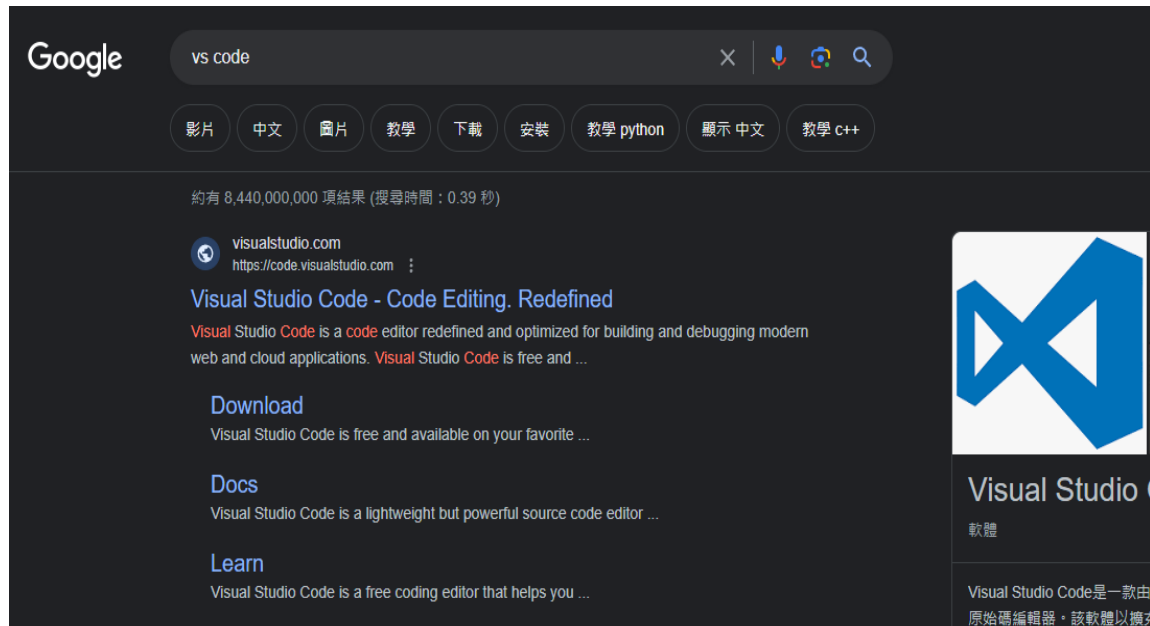
8. Emacs

9. Thonny

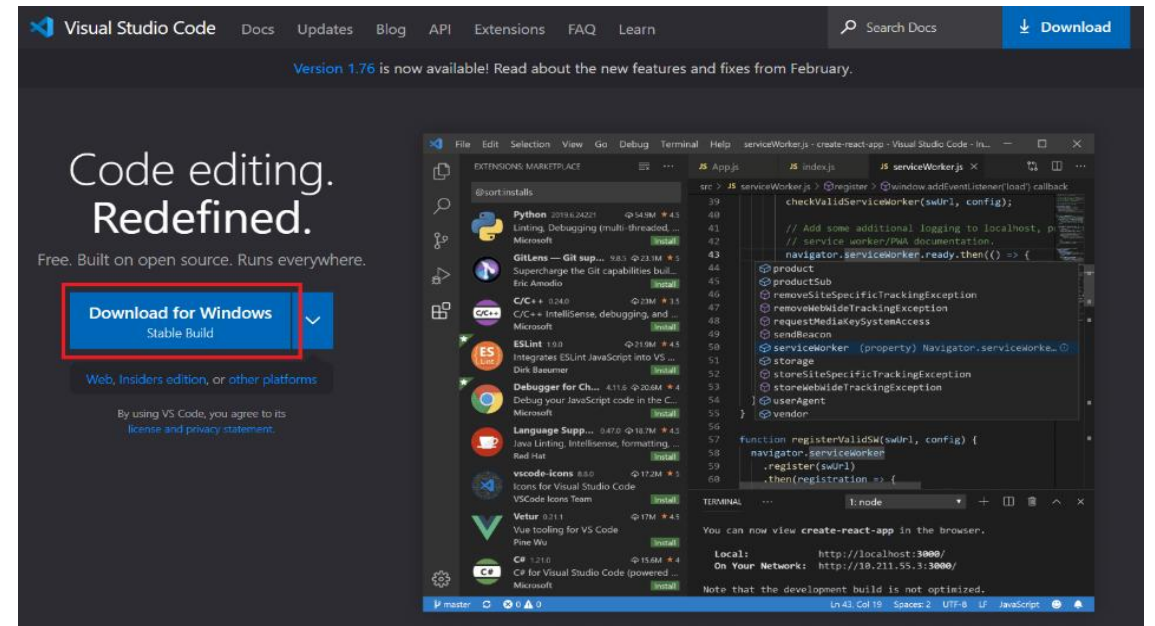
10. Wing

在課程中，我們選用「Visual Studio Code」（簡稱為 VS code）來作為課程使用的 IDE。VS code 是微軟開發的跨平台（Cross-platform）程式開發工具，除了開發程式之外，還可以使用豐富的擴充功能（Extensions）來提升開發的效率。

VS Code 安裝



圖：在 google 搜尋「vs code」，
進入 vs code 首頁



圖：網站會偵測使用者的作業系統，依需求選擇後，
按下 Download for XXX

Thanks for downloading VS Code for Windows!

Download not starting? Try this [direct download link](#).

Please take a few seconds and help us improve ... [click to take survey](#).

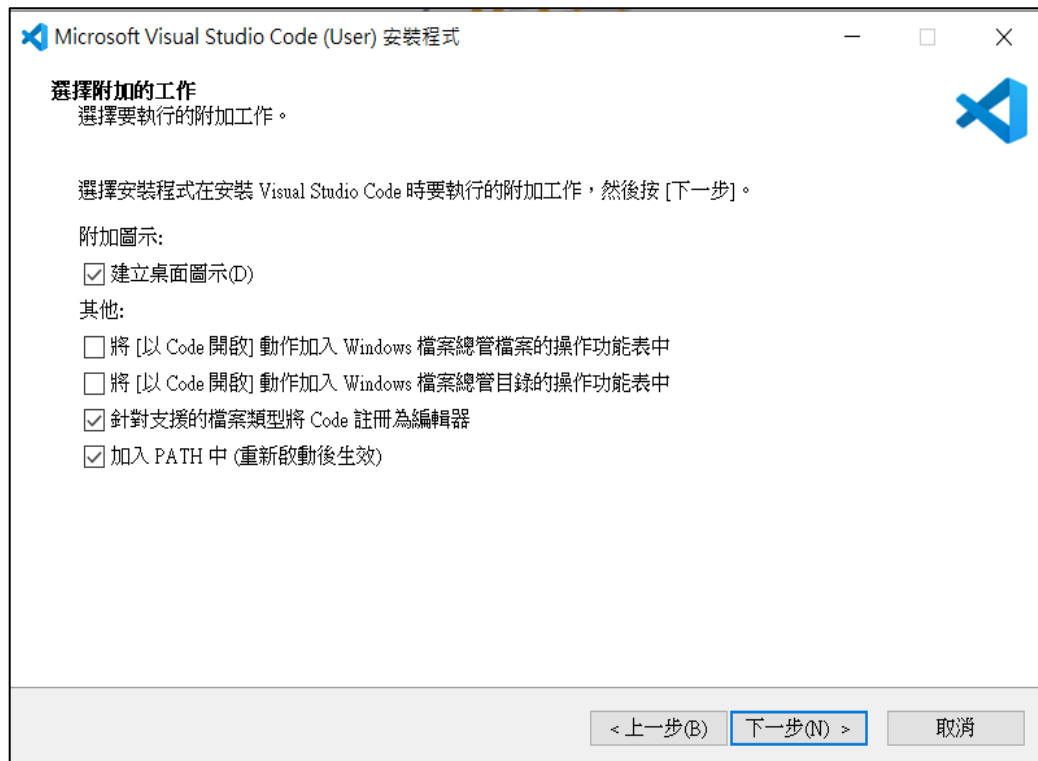
Getting Started

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages and runtimes (such as C++, C#, Java, Python, PHP, Go, .NET). Begin your journey with VS Code with these [introductory videos](#).

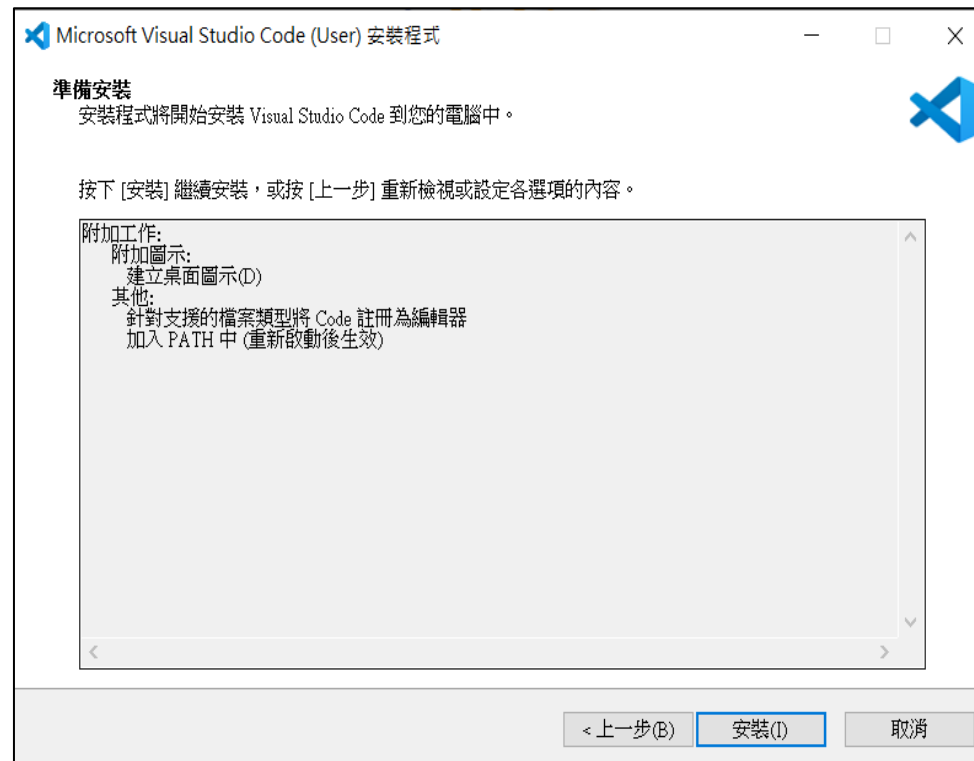
圖：看到這個畫面，安裝檔案會自動下載，
沒有下載的話，
可以按下「direct download link」



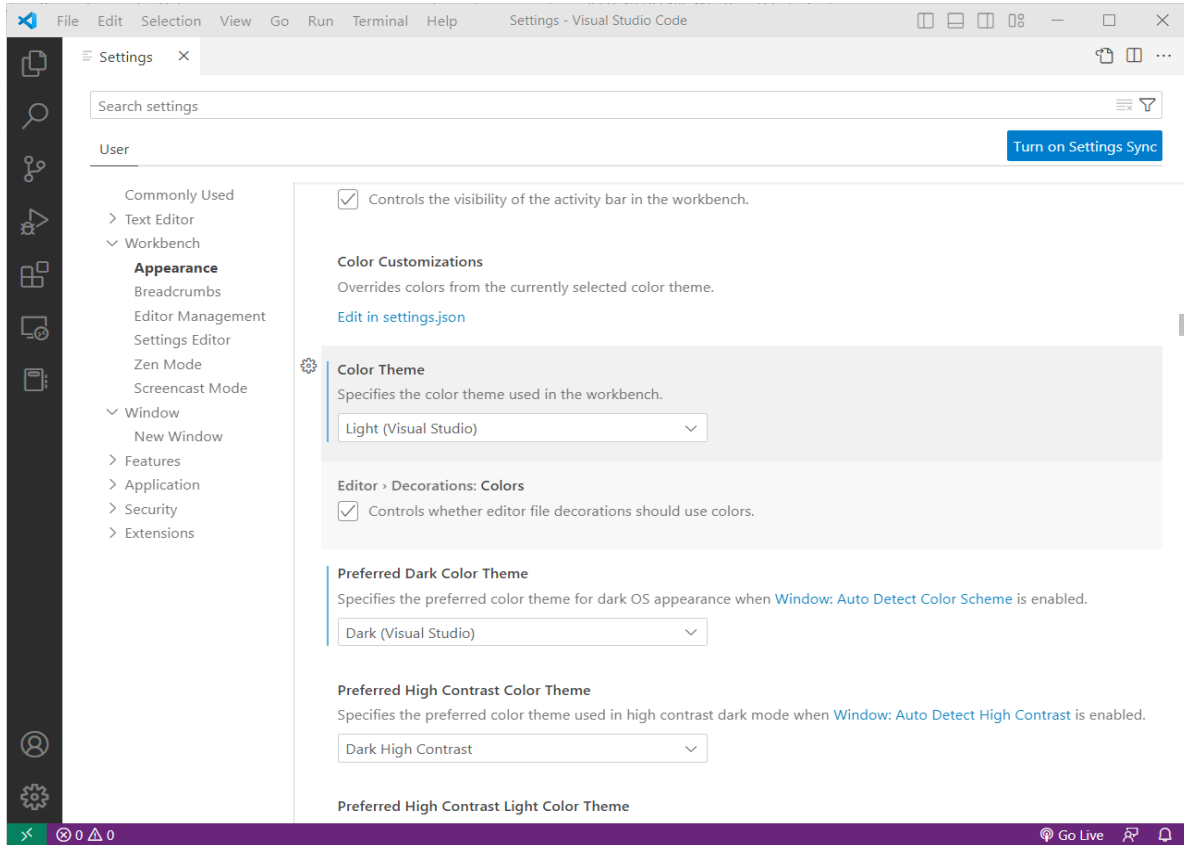
圖：勾選「我同意」，按「下一步」



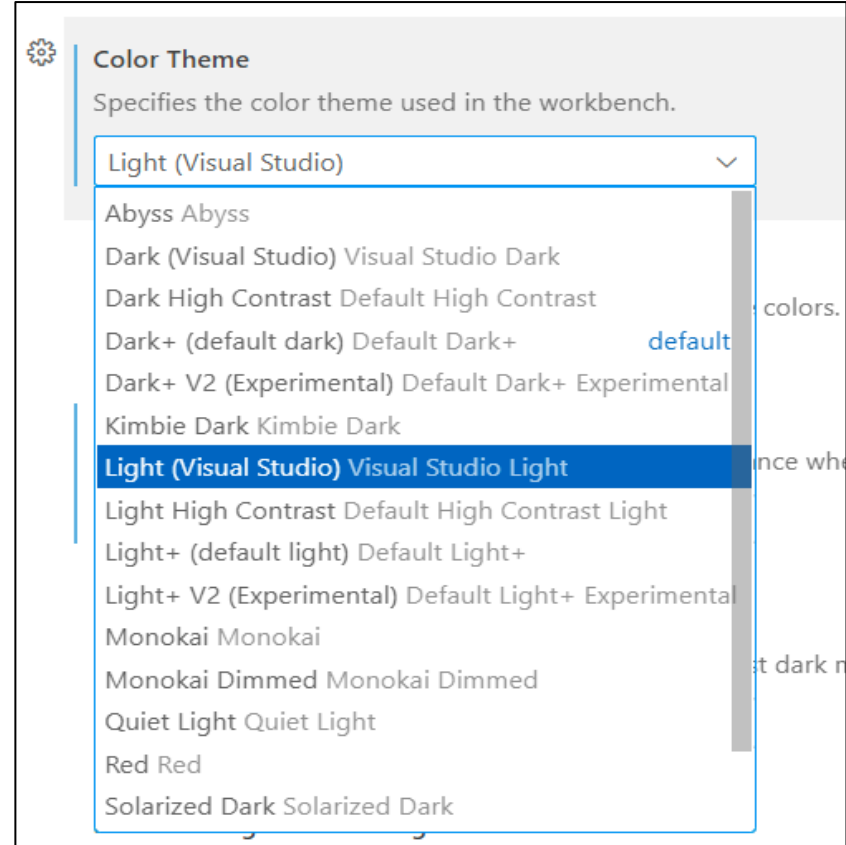
圖：勾選「建立桌面圖示」，按「下一步」



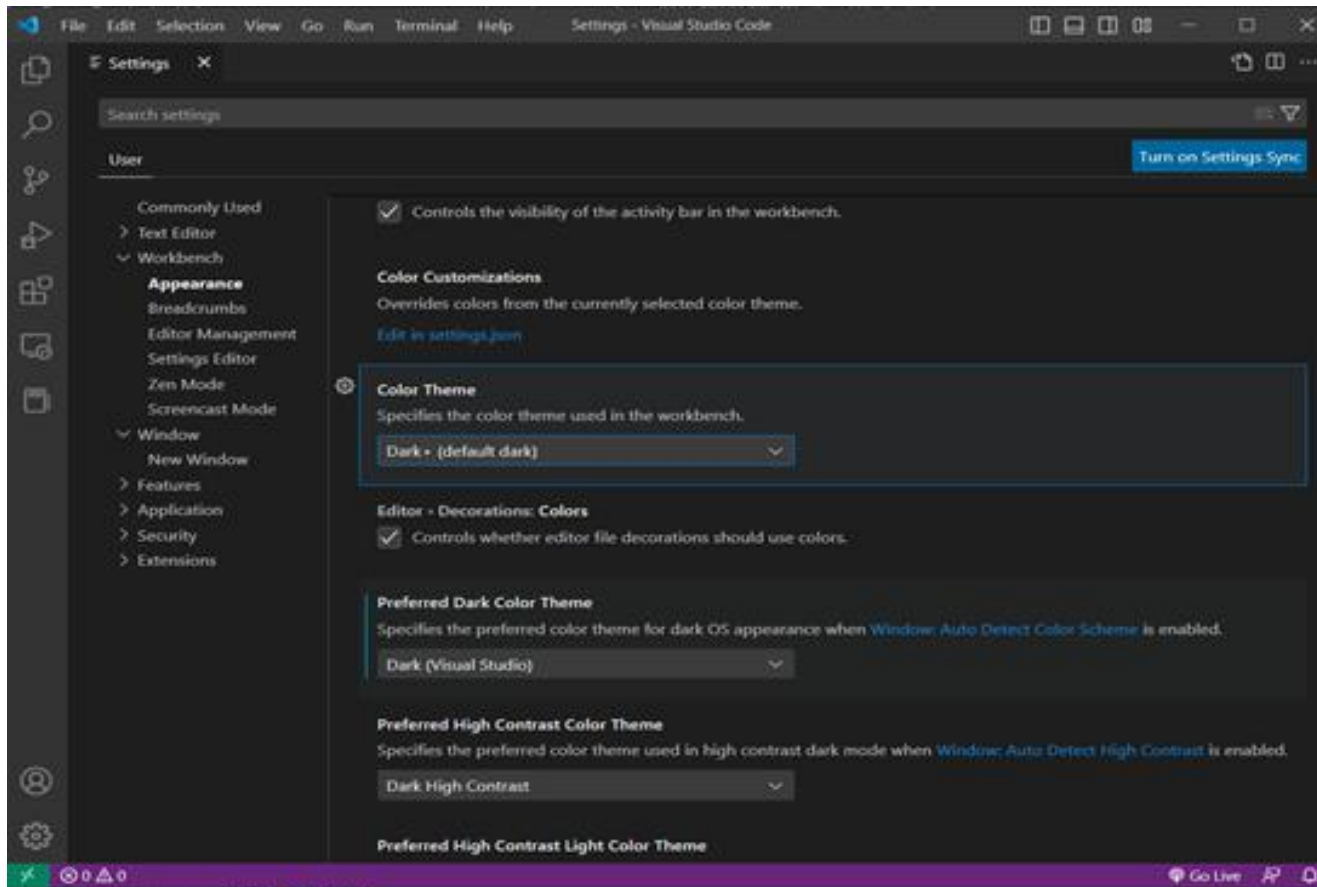
圖：按下「安裝」



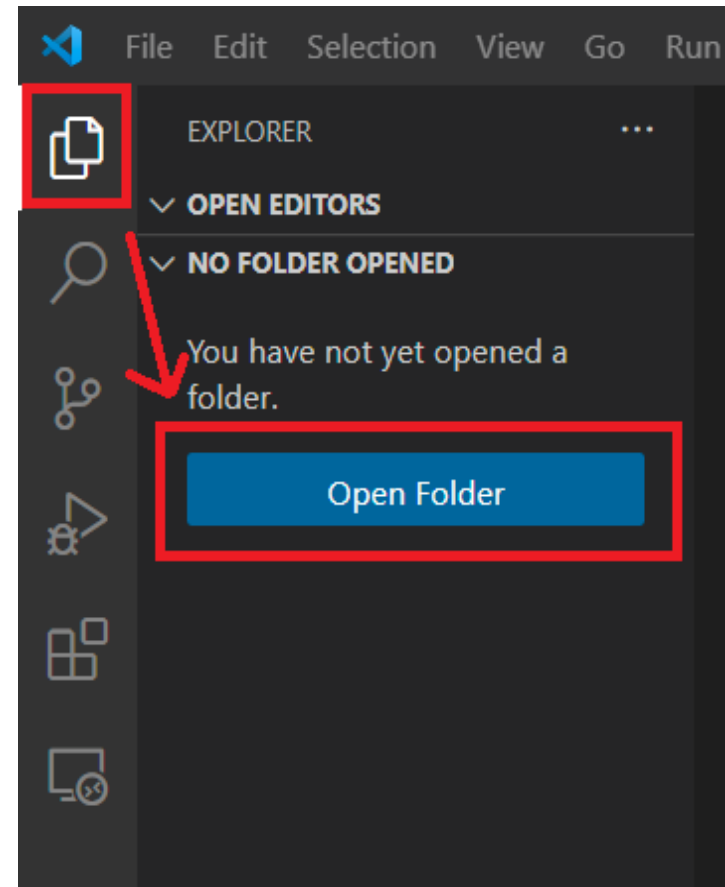
圖：如果想要換佈景顏色，
按下「Ctrl + ,」來開啟設定



圖：在 Color Theme 下方的選單中，
選擇偏好的佈景，例如選擇「Dark+」

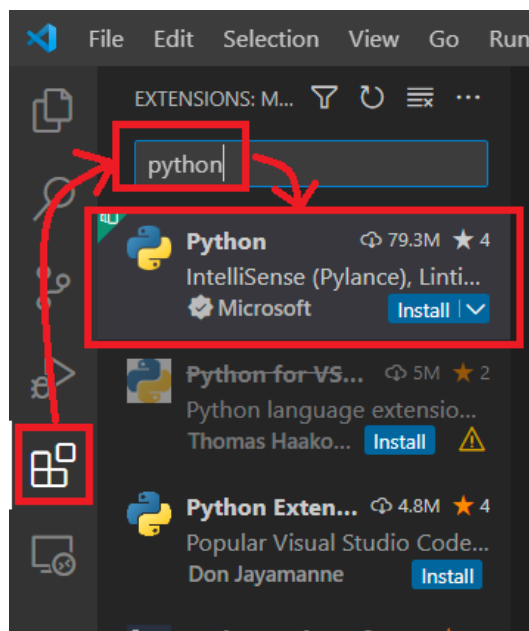


圖：選擇「Dark+」，可以立即套用選擇的佈景，
可以把左上方的頁籤 Settings 關掉（透過按下
Settings 右邊的 x 圖示）



圖：按下左上角檔案總管（Explorer），
再按下開啟資料夾（Open Folder），

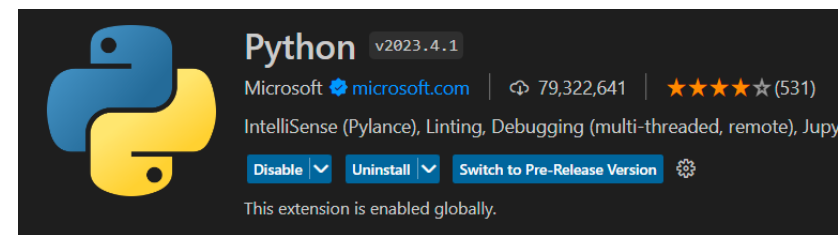
讓專案資料夾使用 Conda 環境



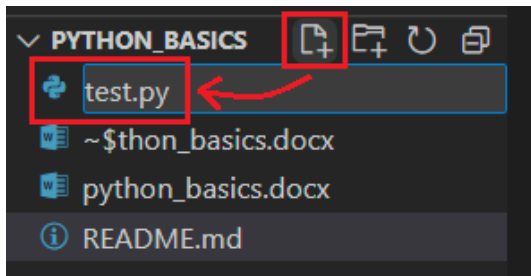
圖：按下擴充功能（Extensions），
在上方輸入「python」，
點選「Python」



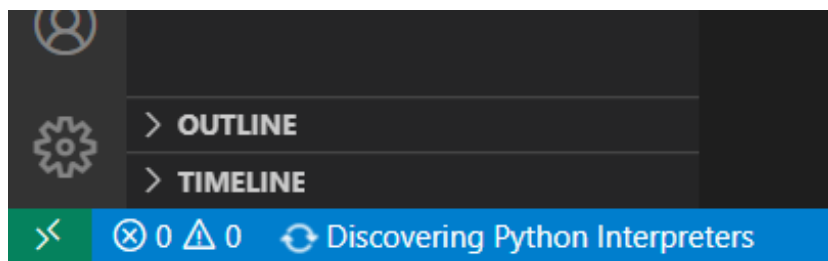
圖：按下 Install



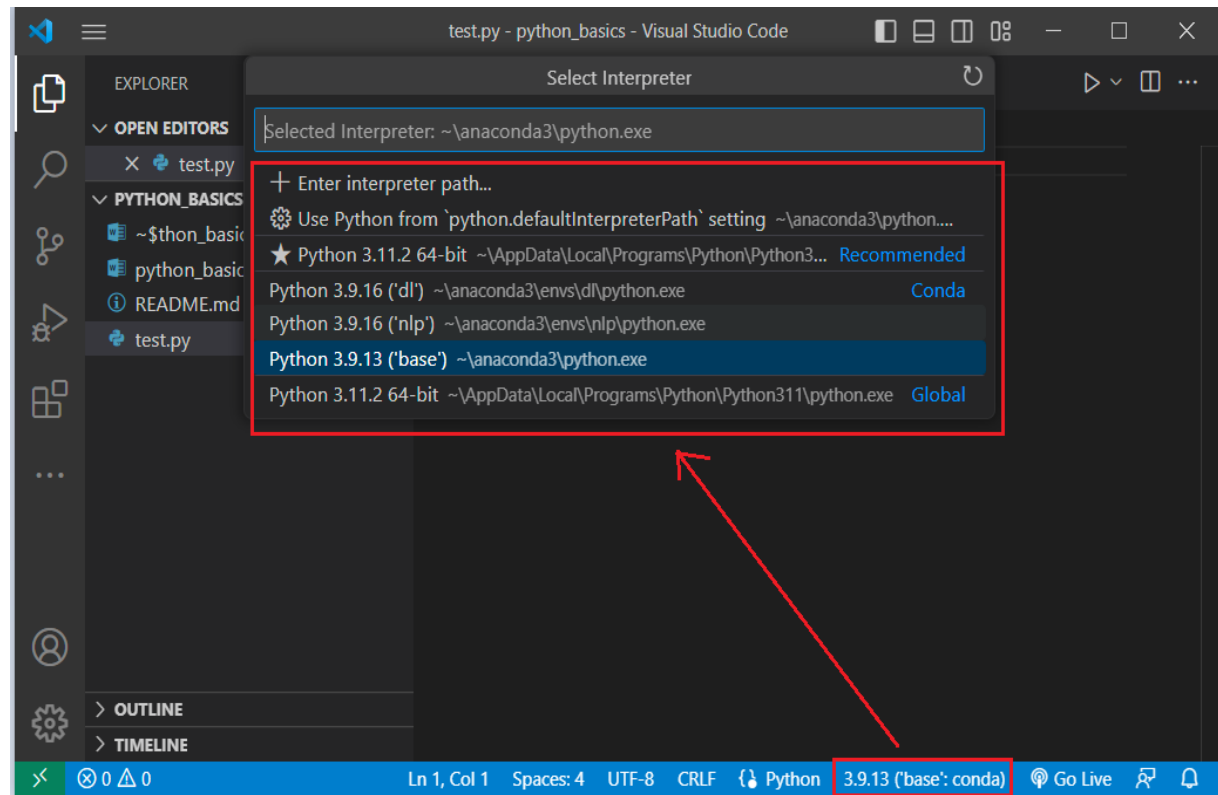
圖：出現 Uninstall 字眼，代表安裝完成，
關閉頁籤，回到專案資料夾



圖：新增檔案，輸入檔案為 test.py，
按下 Enter



圖：開啟 .py 的檔案後，
VS code 自動尋找直譯器



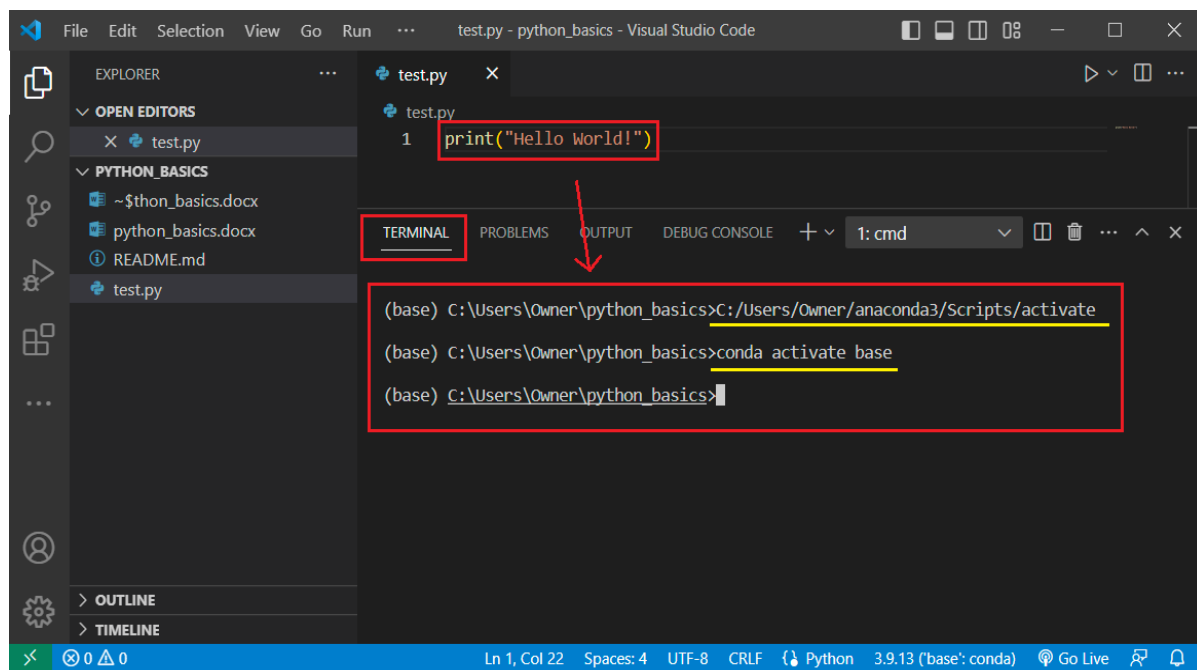
圖：按下右下角的 conda 版本圖示，
可以看到安裝過的環境列表，
可自由切換

建立與執行 .py 檔案

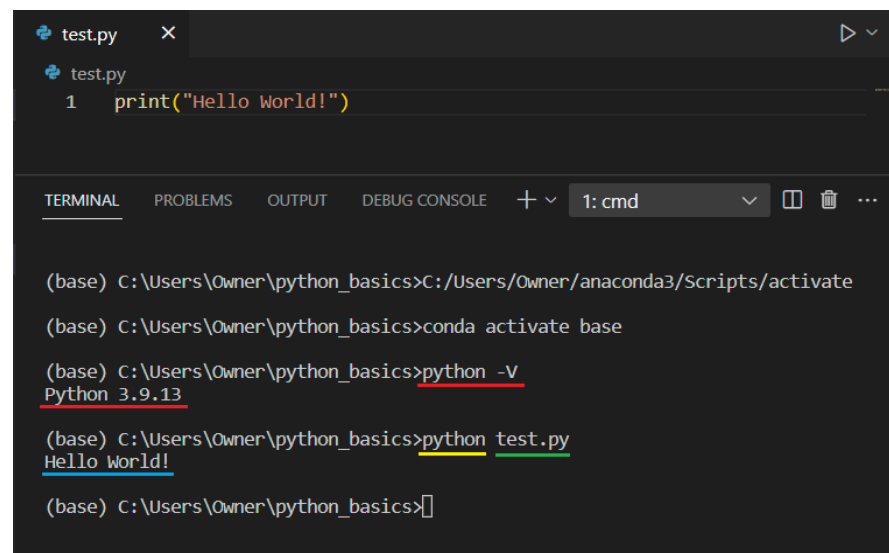
先在剛才的 test.py 檔案中，

輸入「`print("Hello World!")`」，

按下快速鍵「`Ctrl + s`」來儲存文字內容 (程式碼)。



圖：按下快速鍵「`Ctrl + ~`」，顯示終端機 (Terminal)，並啟動 Conda 環境



圖：先確認 base 環境的 python 版本，然後執行 test.py 程式，出現「Hello World!」

Module 2.

資料型態與變數

變數

- 變數 (variable) 就是一個暫存資料 (值) 的地方，會將資料儲存在電腦記憶體當中，原則上什麼值都給可以賦予、分配 (assign) 給它。使用明確、描述性的變數名可以使你的程式更易讀、更易理解。

2-1.py

```
x = 1  
name = "Alex"  
  
print(x)  
print(name)
```

變數命名規則、關鍵字

- 變數命名規則

- 由字母 (A-z)、數字 (0-9) 符號和底線 (_) 組成。
- 必須以字母 (A-z) 或底線 (_) 開頭，**不能以數字開頭**。
- 區分大小寫，name、Name、NAME 是不同的變數。

```
myvar = "John"  
my_var = "John"  
_my_var = "John"  
myVar = "John"  
MYVAR = "John"  
myvar2 = "John"
```

圖：正確的變數命名方式

```
2myvar = "John"  
my-var = "John"  
my var = "John"
```

圖：錯誤的變數命名方式

- **關鍵字**

- 所謂關鍵字 (keywords)，是用來保留給程式語言使用的內建語法或是變數名稱，不適合作為一般變數來使用，以下列出常見的 Python 關鍵字：

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

圖：參考官方文件 https://docs.python.org/3/reference/lexical_analysis.html#keywords

• 註解

- 註解 (comments) 在程式碼當中，通常是用於對撰寫的程式碼進行文字說明，或是希望其它行的程式碼暫時不要執行，先用單行註解或多行註解先將其關閉，待需要的時候才打開。

單行註解	多行註解	多行註解
# 單行註解	''' 多行註解 多行註解 多行註解 '''	'''' 這也是多行註解 這也是多行註解 這也是多行註解 ''''

註：

在 VS code 當中，可以反白幾行程式碼，之後再對它們按下 `Ctrl + /`，可以作到多個單行註解；如果要解除註解，對原先註解的程式碼進行反白，再按下 `Ctrl + /`。

資料型態

資料型態	說明	範例
整數 (Integer)	基本的數據類型，代表整數	<code>x = 10</code>
浮點數 (Float)	代表具有小數點的數字	<code>y = 2.5</code>
布林值 (Boolean)	用來表示真(True)或假(False) 在進行條件判斷或者邏輯運算時非常有用	<code>parse_ok = True</code>
字串 (String)	代表文字數據。你可以用單引號或雙引號來定義一個字串	<code>message = 'Hello'</code> <code>name = "John"</code>
列表 (List)	容納多個值，這些值可以是任意數據類型，並且可修改。 使用中括號「[]」將資料整理在一起。	<code>my_list = [1, 2, 3, 4]</code>
元組 (Tuple)	元組與列表類似，但是一旦創建就不能修改。 使用小括號「()」將資料整理在一起。	<code>my_tuple = (1, 2, 3)</code>
集合 (Set)	類似 list 和 tuple，但它儲存的資料「不能重複」。 使用大括號「{}」將資料整理在一起。	<code>my_set = {1, 2, 3}</code>
字典 (Dictionary)	是一種鍵值對 {key:value} 的數據結構。每個元素都有一個鍵和一個值，可以透過鍵(key)來查找對應的值(value)。	<code>my_dict = {'name': 'John', 'age': 5}</code>

運算子

- 算術運算子 (Arithmetic operators)
- 賦值運算子 (Assignment operators)
- 比較運算子 (Comparison operators)
- 邏輯運算子 (Logical operators)
- 成員運算子 (Membership operators)
- 身分運算子 (Identity operators)
- 位元運算子 (Bitwise Operators)

• 算術運算子

運算符號	說明	範例
+	加	9 + 4 回傳 13
-	減	9 - 4 回傳 5
*	乘	9 * 4 回傳 36
/	除	9 / 4 回傳 2.25
%	相除取得餘數	9 % 4 回傳 1
**	多少的幾次方	9 ** 4 回傳 6561
//	相除取得整數	9 // 4 回傳 2

- 賦值運算子 $a = 3, b = 4$

運算符號	說明	範例
=	單純賦值 (值由右往左給)	$c = a + b$ c的值為 7
+=	計算加法後賦值	$b += a$, 等同於 $b = b + a$ b的值為 7
-=	計算減法後賦值	$b -= a$, 等同於 $b = b - a$ b的值為 1
*=	計算乘法後賦值	$b *= a$, 等同於 $b = b * a$ b的值為 12
/=	計算除法後賦值	$b /= a$, 等同於 $b = b / a$ b的值為 1.3333333333333333
%=	計算取餘數後賦值	$b \% = a$, 等同於 $b = b \% a$ b的值為 1
**=	計算幾次方後賦值	$b ** = a$, 等同於 $b = b ** a$ b的值為 64
//=	計算取得整數後賦值	$b //= a$, 等同於 $b = b // a$ b的值為 1

• 比較運算子

運算符號	說明	範例
==	等於	5 == 5 回傳True
!=	不等於	5 != 6 回傳True
>	大於	5 > 6 回傳 False
<	小於	5 < 6 回傳 True
>=	大於等於	5 >= 6 回傳 False
<=	小於等於	5 <= 6 回傳 True

• 縮排

- Python 程式語言中，縮排不僅是為了使程式碼看起來更整齊、更易讀，更重要的是，它在 Python 中被用來表示程式結構
- 某一行以關鍵字 (if、for、try) 開始，以冒號「:」結尾，下一行就要縮排，來代表程式區塊的開始
- Python 建議使用 **4 個空格**來表達下一個區塊

<pre>score = 50 passing_score = 60</pre>	宣告 score 變數的值為 50 宣告 passing_score 變數的值為 60
<pre>if score < passing_score:</pre>	條件語句的開始，用「:」結尾
<pre> score = score + 10 print('加分')</pre>	程式碼前放了 4 個空格 () 縮排，代表 在同一個程式區塊內。
<pre> (下一行程式) (下一行程式)</pre>	如果下一行程式前面還有縮排， 代表都在同一個程式區塊範圍內。
<pre>print(score)</pre>	這是下一行程式，不受上方程式區塊的範圍影響

• 邏輯運算子

運算符號	說明	範例
and	若是兩邊都是 True，則回傳 True	<pre>name = 'Alex' age = 18 if name == 'Alex' and age == 18: print('正確')</pre>
or	若是兩邊其中一個是 True，則回傳 True	<pre>name = 'Alex' age = 18 if name == 'Alex' or age == 16: print('正確')</pre>
not	若變數的值是 True，則回傳 False；若變數的值是 False，則回傳 True	<pre>flag = False if not(flag): print('從 False 變成 True')</pre>

• 成員運算子

運算符號	說明	範例
in	檢查某個元素是否存在於某個容器中，在則回傳 True，反之則回傳 False。	<pre>myList = [9, 5, 2, 7] if 2 in myList: print('2 在 list 當中')</pre>
not in	檢查某個元素是否不存在於某個容器中，不在則回傳 True，在則回傳 False。	<pre>myList = [9, 5, 2, 7] if 3 not in myList: print('3 不在 list 當中')</pre>

• 身分運算子

運算符號	說明	範例
is	當兩個變數 <u>是</u> 參照同一個物件(同一個記憶體位置)，回傳 True。	<pre>x = 123 y = x # y 參照 x 的記憶體位置 if x is y: print('x 和 y 是同一物件')</pre>
is not	當兩個變數 <u>不是</u> 參照同一個物件(同一個記憶體位置)，回傳 True。	<pre>x = 123 y = 456 # y 隨機佔用記憶體位置 if x is not y: print('x 和 y 不是同一物件')</pre>

• 位元運算子

提示：位元運算會將運算元轉成二進制，再進行運算元之間的布林運算

運算符號	說明	範例
&	AND 運算 (相同為1，就是1)	<pre>x = 1 y = 3 print(x & y) # 輸出 1</pre>
	OR運算 (其中為1，就是1)	<pre>x = 1 y = 3 print(x y) # 輸出 3</pre>
^	XOR運算 (相同為0，相異為1)	<pre>x = 1 y = 3 print(x ^ y) # 輸出 2</pre>
~	補數運算 (有正負號，運算才有義意)	<pre>x = +3 print(~x)# 輸出 -4</pre>
<<	位元左移	<pre>x = 4 print(x << 1) # 輸出 8</pre>

• 運算子優先順序

運算子	說明
()	括號
**	冪、指數
+X -X ~X	一元運算子加號、一元運算子減號、位元運算的 not
* / // %	乘、除、除(只取商數)、相除取餘數
+ -	加、減
<< >>	位元左移、位元右移
&	位元運算 AND
^	位元運算 XOR
	位元運算 OR
== != > >= < <= is is not in not in	比較運算子 身分運算子 成員運算子
not	(邏輯運算)True變成False，或是 False 變成 True
and	(邏輯運算)且
or	(邏輯運算)或

2-3.py

```
# 算術運算子
print(9 + 4)
print(3 / 2)
print(2 ** 3)

# 賦值運算子
a, b = 9, 4
c, d = 4, 2
a += b # 等於 a = a + b
print(a)
d **= c # 等於 d = d ** c
print(d)

# 比較運算子 (用 if 判斷)
e = 4
if e == 4:
    print('e 等於 4')
if e > 2:
    print('e 大於 2')
if e != 5:
    print('e 不等於 5')
```

```
# 邏輯運算子
f = 8
name = 'Bill'
if f == 8 and name == 'Bill':
    print('ok')
if f == 6 or name == 'Bill':
    print('name 等於 Bill')

# 成員運算子
myList = [5, 5, 6, 6, 3, 3, 1, 2]
if 6 in myList:
    print('6 在 list 裡')
if 4 not in myList:
    print('4 不在 list 裡')
```

Module 3.

控制結構

if...else 、 for 、 while

控制結構

- Python的控制結構主要包括條件語句（**if-elif-else**）、迴圈（**for**、**while**）、以及迴圈控制語句（**break**、**continue**）、異常處理（**try-except**）

1. 條件語句（**if-elif-else**）：條件語句用來進行條件判斷，只有滿足某個條件，相對應的語句塊才會被執行。

- **if**

敘述包含一個結果為 **Boolean** 的運算，伴隨一個程式碼區塊

- **if...else**

如果 **if** 敘述的判斷式為真，則執行 **if** 後面的敘述，否則執行 **else** 後面的敘述

3-1.py

```
# if
num = 10

if num > 5:
    print("num 大於 5")

# if else
name = 'apple'
if name == 'apple':
    print('name is apple')
else:
    print("name is not apple")

# if elif (else)
name = 'Jarvis'
if name == "alex":
    print("name: alex")
elif name == "bill":
    print("name: bill")
```

2. 迴圈 (for 、 while) : 迴圈用來進行重複的操作。

- **for** 迴圈

通常用來遍歷一個序列 (如列表 List 或 字串 String)

- **while** 迴圈

在滿足某個條件的情況下不斷循環直到該條件變為 False 。

3. 迴圈控制語句 (break 、 continue) : 用來改變迴圈的執行流程。

- **break**

立即結束當前迴圈(不論剩幾次)

- **continue**

跳過當前迴圈的剩餘部分(continue 以下的程式碼)，直接進入下一輪迴圈。

- 使用 **for 迴圈** 遍歷可迭代的 (iterable) 資料結構

```
for char in 'contents':  
    print(char, end=", ")  
# 輸出 c, o, n, t, e, n, t, s
```

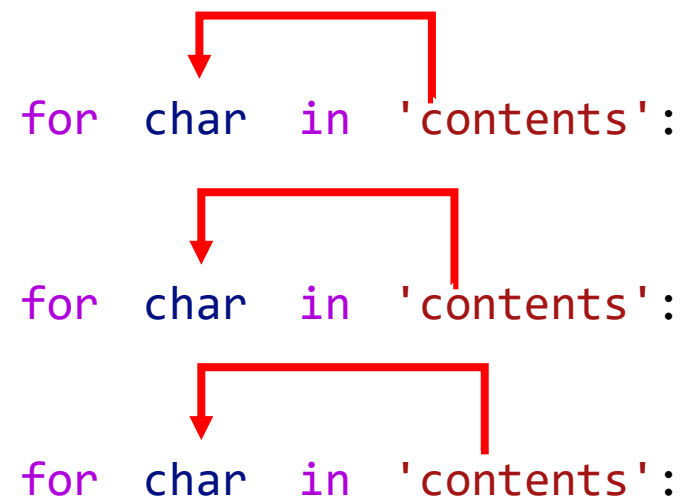
1. 迴圈第一次執行，將 **c** 賦值給 **char** 這個變數

2. 迴圈第二次執行，將 **o** 賦值給 **char** 這個變數

3. 迴圈第三次執行，將 **n** 賦值給 **char** 這個變數

⋮

一直執行到 **s** 結束



3-2.py

for 迴圈

'''

用法

range(n, m-1)

說明

會走訪 n 到 m-1 的數字

'''

```
for i in range(5, 8):  
    print(i, end = ",")
```

while 迴圈

count = 1

```
while count <= 5:  
    print(count, end="")  
    count = count + 1 # 或是寫成 count += 1
```

3-3.py

break

'''

當偵測到字母 t 時，就會強制結束迴圈

'''

```
for char in 'content':  
    if char == 't':  
        break  
    print(char, end="")
```

continue

'''

當偵測到字母 t 時，
會跳過本次迴圈剩下的程式碼 print(char)，
但不會結束迴圈，仍然會進入下一圈繼續執行

'''

```
for char in 'content':  
    if char == 't':  
        continue  
    print(char, end="")
```

補充：input()

- 若是希望能在程式執行期間，讓使用者可以自行輸入資料 (類似提示對話視窗 prompt)，之後整合到程式碼之後輸出結果，可以使用 input() 函式。

3-4.py

```
# 取得使用者輸入的資料
data = input('輸入文字後，按下 Enter: ')
print(data)

# 將使用者輸入強制轉型成 int
num = int(input('請輸入數字: '))
print(type(num))
print(num)
```

Module 4.

字串、串列、元組、 字典、集合

字串

- 「字串」(string) - 同時由兩個以上的字元 (character) 組成。
- 字串需要用「兩個單引號」或「兩個雙引號」包起來，
例如 '星期日' 或 "星期日"。
- 有索引 (index)，可通過索引來取得相對應的字元
- 字串是不可變的，無法更改已經創建字串中的單個字元。

4-1.py

字串變數初始化

```
string01 = "1,2,3,4"
```

```
print(string01)
```

替換字串

```
'''
```

```
string.replace(str1, str2)
```

將 string 中的 str1 替換成 str2

```
'''
```

```
string02 = "Alex"
```

```
string02 = string02.replace('ex', 'len')
```

```
print(string02)
```

去除兩側空格

```
'''
```

```
string.strip()
```

去除字串 string 左、右兩邊的空格

```
'''
```

```
string03 = "          ___ccc___          "
```

```
print(string03)
```

```
print(string03.strip())
```

字串變成小寫或大寫

```
'''
```

```
string.lower()
```

將字串 string 裡的字母全部改成小寫

```
'''
```

```
print("CAR".lower())
```

```
'''
```

```
string.upper()
```

將字串 string 裡的字母全部改成大寫

```
'''
```

```
print("good".upper())
```

補充：字串格式化

- 輸出結果的時候，若是需要嵌入必要的文字，建立字串樣版 (string template)，或是嘗試將文字進行排列與調整，可以將字串格式化。

- 使用 %

語法	說明
%s	插入 字串 的值到字串中
%f	插入 浮點數 的值到字串中
%d	插入 整數 的值到字串中
%%	輸出百分比 (類似跳脫字元)

- 使用 .format()
 - 使用 f-string (Python 版本 > 3.6)

• 使用 %

4-1-1.py

```
'''
格式化字串 - 使用 %
'''

# 多組文字
msg = '%s, %s!' % ('Hello', 'World')
print(msg)

# 整數
msg = 'I am %d years old.' % 5
print(msg)

# 文字與整數
msg = '%s is %d years old.' % ('Alex', 18)
print(msg)

# 指定寬度 (維持 10 個字元長度, 預設向右對齊)
msg = '%10s' % 'Hello'
print(msg)
```

```
# 靠左對齊 (維持 10 個字元長度, 向左對齊)
msg = '[-10s]' % 'Hello'
print(msg)

# 指定浮點數位數
msg = ' [%8.3f]' % 12.3456
print(msg)

# 指定文字長度上限 (只有文字, 才在格式化字串中加入「.」來限定字串長度)
msg = ' [%3s]' % 'Hello'
print(msg)

# 空白補 0
msg = ' [%06.2f]' % 3.1415926
print(msg)
```

• 使用 .format()

4-1-2.py

```
'''
格式化字串 - 使用 string.format()
'''

# 嵌入文字
msg = '{} {}'.format('Hello', 'World')
print(msg)

# 改變參數順序
msg = '{1} {0}'.format('Hello', 'World')
print(msg)

# 指定寬度
msg = '{:10}'.format('Hello')
print(msg)

# 靠右對齊
msg = '{:>10}'.format('Hello')
print(msg)
```

```
# 靠左對齊
msg = '{:<10}'.format('Hello')
print(msg)

# 置中對齊
msg = '{:^10}'.format('Hello')
print(msg)

# 指定浮點數位數
msg = '{:8.3f}'.format(12.3456)
print(msg)

# 指定文字長度上限（只有文字，才在格式化字串中
加入「.」來限定字串長度）
msg = '{:.3}'.format('Hello')
print(msg)

# 空白補 0
msg = '{:06.2f}'.format(3.1415926)
print(msg)
```

• 使用 f-string (Python 版本 > 3.6)

4-1-3.py

```
# f-string (又稱作 formatted string literals,  
version >= 3.6)  
name = "Jarvis"  
age = 18  
result = f"name: {name}, age: {age}"  
print(result)  
  
# 靠左對齊、靠右對齊  
result = f"name: [{name:<10}], age: [{age:>10}]"  
print(result)  
  
# 靠左對齊、靠右對齊，字數不足，補「␣」  
# (沒寫補什麼，預設空格)  
result = f"name: [{name:␣<10}], age: [{age:␣>10}]"  
print(result)
```

```
# 置中對齊  
result = f"name: [{name:^10}], age: [{age:^10}]"  
print(result)  
  
# 指定文字長度上限 (只有文字，才在格式化字串中加入  
# 「.」來限定字串長度)  
result = f"name: [{name:^10.2}]"  
print(result)
```

串列 List

- 或稱為「列表」，跟其它程式語言當中的「陣列」(array) 很類似。
- 使用「中括號」[] 將資料儲存起來。
- 可以想像成一個放置藥丸的盒子，從星期天開始。

[星期天, 星期一, 星期二, 星期三, 星期四, 星期五, 星期六]

0	1	2	3	4	5	6
---	---	---	---	---	---	---

- 這個存取的代號 0、1、2...等等，稱之為「索引」(index)。
- 可以儲存各種資料型態

4-2.py

```
# 初始化一個 list
ids = [1, 2, 3, 4, 5, 6]
print(ids)
```

```
# 新增元素在 list 尾端
ids.append(7)
print(ids)
```

```
# 修改索引位置的元素
ids[4] = 99
print(ids)
```

```
# 刪除索引位置的元素
ids.pop(4)
print(ids)
```

```
# 新增元素 9 在指定索引 1，其餘元素往後移
ids.insert(1, 9)
print(ids)
```

```
# 移除指定的值
ids.remove(9)
print(ids)
```

元組 tuple

- 有序號、索引概念。
- 允許重覆的值。
- 跟 list 很類似，只是**不可更改**元組的資料。
- 可以使用 for 迴圈讀出一筆筆的資料。

4-4.py

```
# Tuple 初始化: 第一種
myTuple01 = ("Big", "Data", "is", "Good")
print(myTuple01)

# Tuple 初始化: 第二種
myTuple02 = "AIOT", "is", "Good"
print(myTuple02)

# 透過指定索引輸出值
print(myTuple01[1])
print(myTuple02[1])

# 複數變數修改值
a = 10
b = 20
print("交換前: a = {}, b = {}".format(a, b))
a, b = b, a
print("交換後: a = {}, b = {}".format(a, b))
```

```
# tuple 不可以修改指定索引的值
myTuple = ("Big", "Data", "is", "Good")
myTuple[3] = "Best"
print(myTuple)

# 用 for 迴圈逐一輸出資料
for value in myTuple01:
    print(value)

# 用 len 計算 tuple 資料個數
print(len(myTuple02))
```

字典 dict

- 「字典」(dict)，跟其它程式語言的「物件」(Object) 很類似。
- 結構以**鍵值對** {key: value}，透過**大括號** {} 來儲存資料，
- key 必須是唯一的，value 可以是任何類型的數據
- 一個字典可以放置多組 {key: value}，每組之間用逗號分隔。
- 例如：{'name': 'Jarvis', 'age': 18, 'height': 172}
 'name' 就是字典其中的 key，
 'Jarvis' 就是此 key 對應的 value。

4-4.py

```
# 初始化 dict
dict01 = {"蘋果": 100, "橘子": 20, "番茄": 50}
print(dict01)

# 印出蘋果的價格
print(dict01["蘋果"])

# 修改橘子的價格
dict01["橘子"] = 30
print(dict01["橘子"])

# 刪除 番茄
del dict01["番茄"]
print(dict01)
```

```
dict02 = {
    'name': 'Jarvis',
    'age': 18,
    'info': {
        'nickname': 'boatman',
        'favorite_role': 'IronMan',
        'phone_number': [
            '0911111111',
            '0922222222',
            '0933333333'
        ]
    }
}

# 取得 nickname
print( dict02['info']['nickname'] )

# 取得所有 phone_number
for number in dict02['info']['phone_number']:
    print(number)
```

集合 set

- 無順序、索引概念
- **沒有重覆**(一樣的只會出現一次)
- 集合內的元素必須是不可變的（例如，數字、字符串和元組）
- 集合本身是可變的，可以新增或删除元素

4-5.py

```
# Set 初始化：第一種
mySet01 = {"Big", "Data", "is"}
print(mySet01)

# 一筆筆讀資料：for 迴圈
for data in mySet02:
    print(data)

# 總共有幾筆資料：len()
print(len(mySet02))

# 因為沒有序號、索引的概念，所以無法透過指定索引輸出
print(mySet01[0])

# 添加一筆資料：add()
mySet01.add("the")
mySet02.add("best")
```

此時新增重複的資料，不會增加

```
mySet01.add("Big")
print(mySet01)
```

添加多筆資料：update()

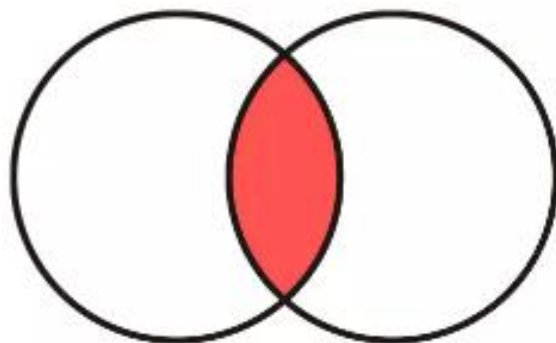
```
mySet01.update(["甲", "乙", "丙"])
mySet02.update(["子", "丑", "寅"])
print(mySet01)
print(mySet02)
```

刪除元素：discard() or remove()

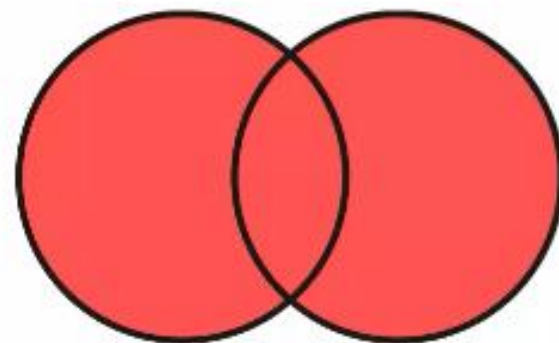
```
mySet01.discard("丙")
```

set 的四種運算型態

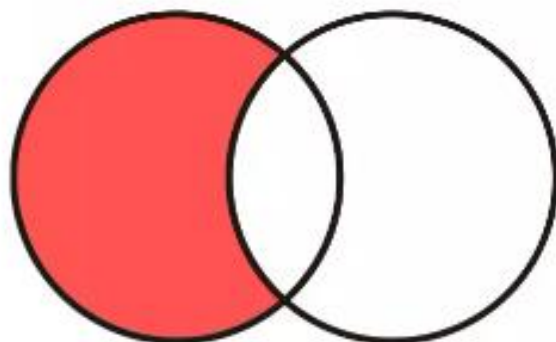
- 交集($A \& B$)
(Intersection)
- 聯集($A \mid B$)
(Union)
- 差集($A - B$)
(Difference)
- 對稱差集($A \wedge B$)
(Symmetric Difference)



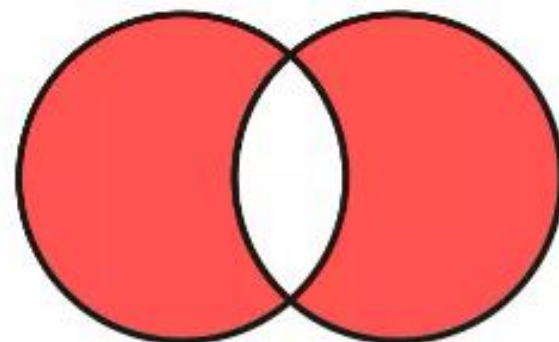
交集 intersection



聯集 union



差集 difference



對稱差集 symmetric_difference

- 序列物件分割 (Slicing)

- 序列物件包含 String、List 及 Tuple等，可透過其順序取得元素
- 中括號為常見物件的索引值取值 (Indexing) 或分割之語法
 - 索引值取值指的是取出一個值
 - 分割指的是取出一部分值

索引值：	0	1	2	3	4	5	6	7	8	9	10	11
	'H'	'e'	'l'	'l'	'o'	','	'W'	'o'	'r'	'l'	'd'	'!'
索引值：	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

圖：把 sequence 當成 list 來存取

4-6.py

```
# 初始化一個字串
myStr = 'Hello,World!'

'''
透過冒號 (:) 進行分割
'''

# 從頭取到 5 (不包含第 5 個元素，實際為 0 ~ 4)
print( myStr[0:5] ) # 索引從 0 開始，到 5 - 1 的索引位置

# 從頭取到 5 (不包含第 5 個元素，實際為 0 ~ 4)
print( myStr[:5] ) # 冒號前面留空，效果跟 [0:5] 一樣

# 從 7 取到尾
print( myStr[7:] ) # 冒號後面留空

# 從 7 取到 9 (不包含第 9 個元素，實際為 7 ~ 8)
print( myStr[7:9] )
```

```
'''
使用負號
'''

# 從倒數第 10 取到倒數第 7 (不包含倒數第 7 的元素)
print( myStr[-10:-7] )

# 從倒數第 5 取到尾
print( myStr[-5:] )

# 從頭取到倒數第 7 (不包含倒數第 7 的元素)
print( myStr[:-7] )

# 取得檔案全名
string06 = "/home/jarvis/ebook.txt"
list06 = string06.split("/")
print(list06[-1])
```

Module 5.

函式

定義函式

- 函式 (function) 將經常使用的程式碼打包起來，變成一種模組，每執行一次函式，相當於把模組的程式碼全部執行過，透過 `def` 關鍵字來定義。

基本用法

函式的定義方式

`def 函式名稱()`:

陳述句(statement, 代表程式要執行的動作)

執行函式，直接使用函式名稱，後面加上()

函式名稱()

• 變數傳遞

回傳值(結果)	傳遞變數
<pre># 傳遞在函式當中執行的結果 def 函式名稱(): return 您的執行結果 # 函式執行完畢後，會將結果回傳給變數 變數 = 函式名稱() # 定義函式 def get_name(): name = "Doraemon" return name # 執行函式 result = get_name() # 輸出回傳結果 print(result)</pre>	<pre># 傳遞在函式當中執行的結果 def 函式名稱(參數[, 參數2,...]): return 整合變數後的結果 # 函式執行完畢後，會將結果回傳給變數 變數 = 函式名稱(變數[, 變數2,...]) # 定義函式 def get_greeting(name): return "Hello, " + name # 執行函式 result = get_greeting("Jarvis") # 輸出回傳結果 print(result)</pre>

區域變數與全域變數

- 當 function 區塊 (block) 內外都有相同的變數名稱：
 - function 區塊內部的變數，叫作**區域變數 (local variable)**。
在函式內部會優先使用**區域變數**。
 - 如果變數在程式碼的最外層，沒有被任何 function 區塊 (block) 包起來，該變數叫作**全域變數 (global variable)**，整個程式都能存取。
- 想在函式內部使用、修改全域變數的值，必須在變數前加上 global 關鍵字。
- 不能在加上 global 的時候進行變數初始化。

- 使用 global 讓函式存取全域變數的值

區域變數無法修改全域變數	區域變數可以修改全域變數
<pre># 全域變數 name = 'Bill' # 定義函式 def 函式名稱(): # 區域變數 name = 'Doraemon' # 執行函式 函式名稱() # 輸出 name print(name) # 輸出 Bill</pre>	<pre># 全域變數 name = 'Bill' # 定義函式 def 函式名稱(): # 使用 global 關鍵字， # 讓函式可以取存全域變數的值 global name # 區域變數 name = 'Doraemon' # 執行函式 函式名稱() # 輸出 name print(name) # 輸出 Doraemon</pre>

匿名函式 Lambda

- 所謂匿名函式 (anonymous function)，許多程式語言都有支援，在程式語言 Python 當中叫作 Lambda，只需要撰寫一行的程式碼，就有一般函式的效果，相對其它函式定義方法來得簡潔。
- 將變數作為函式名稱 = lambda 參數1[,參數2,...]：執行回傳結果

一般函式	Lambda
<pre>def add(x, y): return x * y print(add(3, 5))</pre>	<pre>add = lambda x, y : x * y print(add(3, 5))</pre>
<pre>def abs(x): if x >= 0: return x else: return -x print(abs(5)) print(abs(-5))</pre>	<pre>abs = lambda x : x if x >= 0 else -x print(abs(5)) print(abs(-5))</pre>

Module 6.

物件導向程式設計

基本認識物件導向

- 物件導向程式設計 (Object-Oriented Programming) 是幾乎把所有在 Python 中的東西都看成物件 (object)，擁有它自己的屬性 (attribute) 和方法 (methods)。
- 例如車子有自己的烤漆顏色、4 門或 5 門、車身的寬高、型號等等，這些是屬性；車子可以按下喇叭、煞車、轉彎、加速等，都是它的方法。
- 任何東西都擁有自己的屬性和方法，將這樣的概念用在程式設計上，就叫作物件導向程式設計。

類別 (Class)

- 類別就像是一個樣式的名稱，裡面已經事先定義好基本的、通用的**屬性和方法**，例如人類通常會有兩隻手、兩條腿，同時也會爬行、走路、跑、跳等。

屬性

- 類別屬性 (class attribute) 是屬於類別本身的變數，而不是屬於類別的任何特定實例。這意味著所有該類別的實例都共享同一個類別屬性的值。當你更改類別屬性的值時，這個更改會影響到所有使用該屬性的實例。

方法 (Methods)

- 方法就是定義在 class 內的函式 (functions) ，都是執行一連串在程式區塊中定義好的程式碼 (code) ，在每一個類別 (class) 中，都會有一些 Python 內建且需要在一開始定義的函式。
- 在建立類別的時候，常常會看到 `__init__(self)` ，指的是建立物件 (或稱為實體) 的時候，預設第一時間會執行的方法，稱為建構子 (constructor) 或 建構函式。

- 在建構函式內設定參數，代表在建立該物件時一定要帶參數

```
class 類別名稱:
```

```
    def __init__(self, 參數1, 參數2):
```

```
        self.實例屬性1 = 參數1
```

```
        self.實例屬性2 = 參數2
```

```
        self.實例屬性3 = 自訂值
```

```
物件(實體) = 類別名稱('Hello', 'World')
```

```
print(物件.實例屬性1) # 輸出 Hello
```

```
print(物件.實例屬性2) # 輸出 World
```

```
print(物件.實例屬性3) # 輸出 在程式中的自訂值
```

```
class 類別名稱:  
    def __init__(self):  
        print('Python')  
    def 方法1(self):  
        return 'Hello World!'  
    def 方法2(self):  
        return 'Good job!'
```

```
物件(實體) = 類別名稱() # 建立物件(又稱實體化)時，直接輸出 Python  
print( 物件.方法1() ) # 輸出 Hello World!  
print( 物件.方法2() ) # 輸出 Good job!
```

```
class Car:    # 定義一個 類別 ( class ) , 叫做 Car

    wheels = 4    # 定義 Car 的 類別屬性 ( class Attribute ) , wheels

    def __init__(self, brand, model):

        self.brand = brand

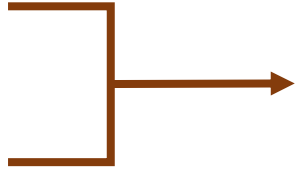
        self.model = model

        self.doors = 4

    def display_info(self):

        return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

```
class Car:
    wheels = 4
```



使用 Car 的 類別屬性
Car.wheels # 得到 4

```
def __init__(self, brand, model):
    self.brand = brand
    self.model = model
    self.doors = 4

def display_info(self):
    return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

```
class Car:
```

```
    wheels = 4
```

建構函式 類別的實例 建構函式的參數

```
def __init__(self, brand, model):
```

```
    self.brand = brand
```

```
    self.model = model
```

```
    self.doors = 4
```

```
def display_info(self):
```

```
    return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

```
class Car:
```

```
    wheels = 4
```

```
    def __init__(self, brand, model):
```

```
        self.brand = brand
```

```
        self.model = model
```

```
        self.doors = 4
```

```
    def display_info(self):
```

```
        return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

實例屬性



```
class Car:
```

```
    wheels = 4
```

```
    def __init__(self, brand, model):
```

```
        self.brand = brand
```

```
        self.model = model
```

```
        self.doors = 4
```

```
    def display_info(self):
```

```
        return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

實例屬性

用 參數 設定 實例屬性的值

```
class Car:
```

```
wheels = 4
```

創建 Car 類別的一個實例

car1 = Car("Toyota", "Camry")

使用 Car 的實例屬性

car1.brand # 得到 Toyota

```
def __init__(self, brand, model):
```

```
    self.brand = brand
```

```
    self.model = model
```

```
    self.doors = 4
```

```
def display_info(self):
```

```
    return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

```
class Car:
```

```
wheels = 4
```

創建 Car 類別的一個實例

```
car1 = Car("Toyota", "Camry")
```

```
def __init__(self, brand, model):
```

```
    self.brand = brand
```

```
    self.model = model
```

```
    self.doors = 4
```

```
def display_info(self): # 實例的方法(method)
```

```
    return f"The {self.brand} {self.model} has {Car.wheels} wheels."
```

使用實例的屬性

使用類別的屬性

使用 car1 物件的方法

car1.display_info()

繼承 (inheritance)

- 繼承 (Inheritance) 是建立一個新的類別 (class) 時，把前一個定義好的類別(父類別)內容拿過來使用，包括屬性和方法，同時也可以在新類別當中定義新的屬性和方法。
- 繼承的好處：
 - 代碼重用：可以使用父類的方法和屬性，無需重新編寫。
 - 擴展性：可以在不修改現有類別的情況下添加新功能。
 - 可維護性：修改父類的程式碼，所有子類都會獲得更新。

inheritance

```
class Vehicle:
    def info(self):
        return f"This is class : {self.__class__.__name__}"

print(Vehicle().info())

# class Car 繼承上面定義好的 class Vehicle
class Car(Vehicle):
    pass

# 可以使用 info() 這個 method
print(Car().info())
```

覆寫 (Override)

- 在 Python 中，覆寫父類別的方法非常直接。當子類別中有一個與父類別**同名的方法**時，允許子類別改變或擴展從父類別繼承的行為，這個方法就自動覆寫了父類別中的同名方法。



當我們在子類別 Override 某個方法，但又想保留父類別的原始方法時，可以使用 **super()** 來調用繼承的父類別方法

Override

父類別

```
class Vehicle:

    def __init__(self, brand, model):

        self.brand = brand

        self.model = model

    def display_info(self):

        return f"Brand: {self.brand}, Model: {self.model}"
```

Override

```
class Car(Vehicle):  
    def __init__(self, brand, model, car_type):  
        super().__init__(brand, model) # 在子類別中調用父類別的方法  
        self.car_type = car_type  
  
    def display_car_info(self):  
        return f"{self.display_info()}, Car Type: {self.car_type}"
```

Module 7.

例外處理

認識例外處理

- 例外 (Exception) 就是程式執行過程中，發生錯誤，影響程式的正常執行，所立刻執行的事件 (event)。
- 通常程式執行都會被中斷，偶爾也會顯示一些造成問題的原因；也可以自訂例外發生時的程式碼，不見得一定要使用預設的警示結果。

常見的事件類別

事件類別	說明
BaseException	基本例外事件類別
Exception	最常用的例外事件類別
IOError	輸入/輸出發生錯誤的事件類別
NameError	變數尚未宣告所發生的事件類別
IndexError	序列中沒有這個索引(index)的事件類別
RuntimeError	執行過程中發生錯誤的事件類別
SyntaxError	Python 語法錯誤的事件類別
KeyboardInterrupt	使用 Ctrl + C 取消程式執行的事件類別
IndentationError	縮排錯誤的事件類別
Warning	顯示警告訊息的事件類別 (通常不會中斷程式)
DeprecationWarning	功能、指令、語法等不再被支援的事件類別
RuntimeWarning	可能造成程式執行錯誤的事件類別
SyntaxWarning	程式碼語法可能有問題的事件類別

try-except 陳述

- 例外的處理，可以使用「try: ... except: ...」語法，try: 是預設的程式區塊，當 try: 程式區塊的程式碼在執行期間發生錯誤，會直接中斷，跳到 except: 程式區塊來繼續執行。

try:

預設正常執行程式的程式區塊

except 例外事件類別名稱1:

拋出例外時，符合例外事件類別名稱1，則在這個程式區塊繼續執行

except 例外事件類別名稱2:

拋出例外時，符合例外事件類別名稱2，則在這個程式區塊繼續執行

finally:

無論是否正常執行，最後都會執行這裡的程式區塊

7-1.py

```
# 可以連續定義數個例外處理機制，  
# 最符合的例外事件類別會先拋出  
try:  
    print(x)  
except NameError:  
    print("x 尚未宣告")  
except:  
    print("某個東西出錯了")  
  
# 多層例外處理：  
try:  
    print(x)  
    try:  
        print(y)  
    except NameError as err:  
        print("NameError:")  
        print(err)  
except Exception as err:  
    print("Exception:")  
    print(err)
```

例外處理 raise 陳述

- 如果有自訂例外的需求，可以使用 raise 語法來拋出例外訊息。

7-2.py

```
# 自訂例外處理
x = -1
if x < 0:
    raise Exception("數字小於 0")

# 自訂型別錯誤的例外處理
x = "hello"
if type(x) is not int:
    raise TypeError("只接受整數型別")
```

Module 8.

檔案處理

open() 函式

- 可以使用內建的 `open()` 函式來進行檔案處理，其中包含了 4 種檔案模式：
 - `"r"`: Read (讀取檔案)
 - `"a"`: Append (附加資料到檔案內容的最尾端或最後一行)
 - `"w"`: Write (寫入檔案)
 - `"x"`: Create (建立檔案)
 - `"t"`: Text (文字模式)
 - `"b"`: Binary (二進制模式，例如用於 `images` 的讀取)

讀取檔案

8-1.py

讀取檔案內容 - 現在常用的寫法（執行完畢會自動關閉檔案）

```
with open(file="./2-1.py", mode="r", encoding="utf-8") as f:  
    print( f.read() )
```

```
print("=" * 50) # 印出 50 個 等號
```

一行一行讀出來

```
with open(file="./2-1.py", mode="r", encoding="utf-8") as f:  
    for line in f:  
        print(line) # 每一行後面會用 \n 結尾，所以輸出會有多次斷行的效果
```

寫入檔案

8-2.py

```
# 文字內容
```

```
text = '''
```

```
傑出的人會努力規劃自己的人生，普通的人則是隨便命運怎麼安排  
一個是充實、有目的的過每一天，一個是隨興的過一天算一天
```

```
'''
```

```
# 寫入檔案
```

```
with open("./8-2.txt", "w", encoding="utf-8") as f:
```

```
    f.write(text + "\n")
```

```
# 寫入最後一行（附加文字在檔案內容最後一行）
```

```
with open("./8-2.txt", "a", encoding="utf-8") as f:
```

```
    f.write("People say that success just happens to you. It doesn't! " + "\n")
```

刪除檔案

8-2.py

```
# 匯入套件
import os

# 檔案路徑
file_path = './8-2.txt'

# 如果指定路徑的檔案存在，則進行刪除
if os.path.exists(file_path):
    os.remove(file_path)
else:
    print('Not found')
```

Module 9.

模組

定義模組

- 將常用的程式碼 (通常是函式) 整合在一個 .py 檔案中，變成一個函式庫 (library)，或是重複使用的工具，在其它程式需要用到的時候，可以不用重新撰寫，只要透過 import 語法將其引入、匯入，即可在當前的程式檔案裡使用。

myModule.py

計算幾次方

```
def pow(x, y):  
    return x**y
```

簡單斷句

```
def segment_sentence(text):  
    list_sentences = text.split(' ')  
    return list_sentences
```

使用模組-1

- 匯入模組的程式檔，通常是主程式；myModule.py 就是自訂模組，被主程式匯入。

9-1.py

```
import myModule

# 計算幾次方
num = myModule.pow(2, 3)
print(num)

# 簡單斷句
txt = "I will always love you"
list_result = myModule.segment_sentence(txt)
print(list_result)
```

使用模組-2

- 直接從模組匯入特定函式，這一種較為常用，明確的表示從模組引用了哪些函式。

9-2.py

```
# 直接從模組中匯入函式
from myModule import pow, segment_sentence

# 計算幾次方
num = pow(2, 3)
print(num)

# 簡單斷句
txt = "I will always love you"
list_result = segment_sentence(txt)
print(list_result)
```

__name__ 內建變數的用法

- __name__ 是內建變數，用來確認匯入模組的名稱 (例如 myModule，會自動去掉 .py)，如果執行的主程式檔案裡面有輸出 __name__，它的值就會是 __main__。

myModule.py	9-2.py
<pre># 計算幾次方 def pow(x, y): return x**y # 簡單斷句 def segment_sentence(text): list_sentences = text.split(' ') return list_sentences # 想先在模組裡測試，再給其它程式檔案匯入 print(pow(7, 2))</pre>	<pre># 直接從模組中匯入函式 from myModule import pow, segment_sentence # 計算幾次方 num = pow(2, 3) print(num) # 簡單斷句 txt = "I will always love you" list_result = segment_sentence(txt) print(list_result)</pre>

- 如果想要測試自訂模組的執行結果是否如預期，可以執行 myModule.py：

```
(base) C:\Users\Owner\python_basics>python myModule.py
49
(base) C:\Users\Owner\python_basics>
```

圖： myModule.py 執行結果

- 此時執行 9-2.py 的話，會發生什麼事？

```
(base) C:\Users\Owner\python_basics>python 9-2.py
49
8
['I', 'will', 'always', 'love', 'you']
(base) C:\Users\Owner\python_basics>
```

圖：突然輸出 49 這個模組內部測試結果

- 那是因為匯入模組的時候，Python 會將模組內部的程式碼 (myModule.py) 由上而下執行過一遍，連同模組的執行輸出也顯示在主要程式 (9-2.py) 裡。若是希望模組測試的結果，不要在主程式執行階段輸出，此時要將 myModule.py 改成：

myModule.py

```
'''  
將模組當作主程式來執行，  
此時 myModule.py 的  
__name__ 就會等於 __main__，  
__main__ 就是主程式的意思  
'''  
  
if __name__ == "__main__":  
    print( pow(7, 2) )
```

- 雖然主程式 (9-2.py) 執行時，模組的輸出不會顯示出來，實務上，通常都會在撰寫程式碼的檔案加上「`if __name__ == "__main__":`」，來確保每個程式進行模組化 (modularization) 時，都能避免掉不必要的麻煩。

9-2.py

```
# 直接從模組中匯入函式
from myModule import pow, segment_sentence

if __name__ == "__main__":
    # 計算幾次方
    num = pow(2, 3)
    print(num)

    # 簡單斷句
    txt = "I will always love you"
    list_result = segment_sentence(txt)
    print(list_result)
```

Module 10.

裝飾器

Decorator (裝飾器)

- 裝飾器是一種Python的高級功能，它允許你**修改或擴展函式**或方法的行為，而**無需直接修改它們的程式碼**。
- 裝飾器本身是一個函式，它接受一個函式作為參數並返回一個新的函式。通常，裝飾器的名稱以 **@** 符號開頭，並放在要裝飾的函式定義之前。

初建立 decorator

```
# 定義一個裝飾器函數

def my_decorator(func):

    def wrapper():

        print("在函式執行之前做一些事情")

        func() # 呼叫原始函式

        print("在函式執行之後做一些事情")

    return wrapper
```

```
# 使用裝飾器

@my_decorator

def say_hello():

    print("Hello decorator ! ")

# 調用裝飾後的函式

say_hello()
```

```
def my_decorator(func):  
  
    def wrapper():  
  
        print("在函式執行之前做一些事情")  
  
        func() # 執行 say_hello()  
  
        print("在函式執行之後做一些事情")  
  
    return wrapper
```

```
@my_decorator  
  
def say_hello():  
    print("Hello decorator !")  
  
say_hello()
```



當我們調用 `say_hello()` 時，它實際上執行的是 `wrapper` 函式，這允許我們在函式執行前後添加自己的邏輯。

*args 和 **kwargs

- ***args** 和 ****kwargs** 是Python中的特殊參數，它們允許函式接受不定數量的位置參數和關鍵字參數。
- ***args** 表示接受任意數量的位置參數，****kwargs** 表示接受任意數量的關鍵字參數。在這邊使用可以確保裝飾器函式應用於各種不同的函式，而不必關心它們的參數。