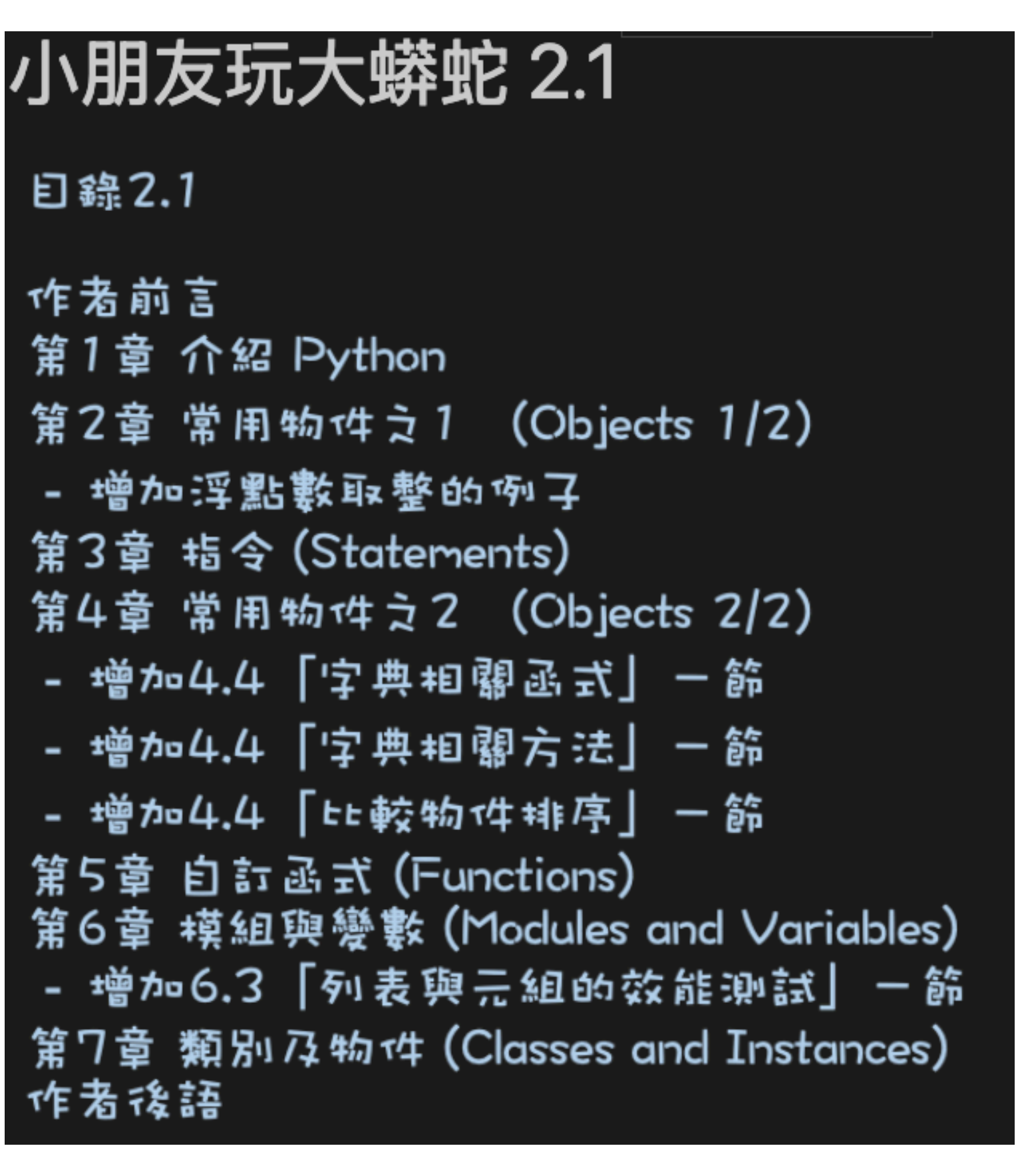
小朋友玩大蟒蛇 2.1

目錄2.1

作者前言
第1章 介紹 Python
第2章 常用物件之1 （Objects 1/2）
- 增加浮點數取整的例子
第3章 指令 (Statements)
第4章 常用物件之2 （Objects 2/2）
- 增加4.4「字典相關函式」一節
- 增加4.4「字典相關方法」一節
- 增加4.4「比較物件排序」一節
第5章 自訂函式 (Functions)
第6章 模組與變數 (Modules and Variables)
- 增加6.3「列表與元組的效能測試」一節
第7章 類別及物件 (Classes and Instances)
作者後語

第6章 模組與變數 (Modules and Variables)

6.1 介紹模組（Module）

為了可以呼叫別人寫好的程式，Python 設計模組的觀念，定義所有副檔名為 (.py) 的檔案都是可以被搜索的模組並可以被載入到我們的程式裡來呼叫。

Python 模組的架構讓人們可以方便疊床架屋，使得程式可以像蓋房子砌牆一樣從簡單的一磚一瓦一棟棟地建起一棟高樓，理論上可以完成一套複雜系統。

首先我們來學習如何載入模組。

介绍「進口」指令：import statement

語法

```
import <模組名稱>
from <模組名稱> import <變數名稱>
```

其中：

1. Python 的模組（Module）基本上就是一個檔案，副檔名是 ".py"。它可以是內建的、自訂的，或是他人寫好的
2. Python 系統啓動時，會從環境變數 PYTHONPATH 定義的目錄列載入成一個可搜索的目錄列表
3. 一旦進口指令（或稱引進指令）需要模組載入時，Python 會搜索檔案，名字叫 "<模組名稱>.py"
4. 若是在目錄列中，檔案找不到，Python 列印出錯誤信息並停止執行
5. 若 Python 搜索模組一旦成功，Python 會將該模組載入。我們後面的程式就可以呼叫模組裡面的所有的屬性和方法
6. 第二種進口指令是從進口模組當中載入，叫<變數名稱>的 一部份模組。它可以是屬性、方法或類別。類別下一章再詳細討論，下面先說明屬性和方法的呼叫語法。

呼叫語法

```
<模組名稱>.<模組內的屬性>
<模組名稱>.<模組內的方法>(參數列)
```

其中：

1. <模組內的屬性>就是進口模塊內定義和使用的變數
2. <模組內的方法>就是進口模塊內定義的函式

同學們很早就看到進口指令了，記得嗎？

- 在說明多行字串在使用手冊上 Python 利用 help() 發揮出來很大功用(提示：import greetings)

現在終於對進口指令能夠整合模塊的強大功能有了深刻印象了吧，看看一些例子如下：

例子1: sys 模組

sys 是 Python 的內建模組，提供與 Python 直譯器相關的變數和函數。以下是 sys 模組的主要功能：

1. 系統參數：

- sys.argv：獲取命令列參數
- sys.path：Python 模組的搜尋路徑
- sys.version：Python 直譯器的版本資訊

2. 標準輸入/輸出：

- sys.stdin：標準輸入
- sys.stdout：標準輸出
- sys.stderr：標準錯誤輸出

3. 程式控制：

- sys.exit([status])：退出程式
- sys.getdefaultencoding()：獲取系統預設字元編碼

4. 記憶體管理：

- sys.getsizeof(object)：獲取物件的記憶體大小
- sys.getrefcount(object)：獲取物件的參考計數

5. 執行環境：

- sys.platform：作業系統平台
- sys.modules：已載入的模組字典

使用範例：

```python
# 輸入系統模組 'sys'
import sys

# 系统模組中有變數名稱, version, 提供 Python 版本資訊
print(f'Python sys.version:\n{sys.version}\n')

# 系统模組中有變數名稱, executable, 提供 Python 執行程式所在的資料夾位置
print(f'Python sys.executable:\n{sys.executable}\n')

# 獲取命令列參數
print(f"程式名稱: {sys.argv[0]}\n")
print(f"參數: {sys.argv[1:]}\n")

# 檢查 Python 版本
if sys.version_info < (3, 6):
    print("需要 Python 3.6 或更新版本\n")
else:
    print(f"Python sys.version_info: {sys.version_info}\n")

# 修改預設編碼
print(f"預設編碼: {sys.getdefaultencoding()}\n")
```

```
Python sys.version:
3.13.2 | packaged by Anaconda, Inc. | (main, Feb  6 2025, 12:55:35) [Clang 14.0.6 ]

Python sys.executable:
/Users/michaelkao/miniconda3/envs/pex/bin/python

程式名稱: /Users/michaelkao/miniconda3/envs/pex/lib/python3.13/site-packages/ipykernel_launcher.py

參數: ['--f=/Users/michaelkao/Library/Jupyter/runtime/kernel-v3997fd3d5d68f99143d78c38bc85399a633be6f08.json']

Python sys.version_info: sys.version_info(major=3, minor=13, micro=2, releaselevel='final', serial=0)

預設編碼: utf-8
```

例子2: python-dotenv 模組

python-dotenv 是一個 Python 套件，主要用於管理應用程式的環境變數。以下是它的主要功能和用途：

1. 環境變數管理：

- 從 .env 檔案中載入環境變數 讓開發者可以將敏感資訊（如 API 金鑰、資料庫連線字串等）與程式碼分離

2. 開發便利性：

- 在開發環境中輕鬆管理不同的設定
- 無需在程式碼中硬編碼敏感資訊

3. 安全性：

- 防止敏感資訊被提交到版本控制系統
- 通常會將 .env 加入 .gitignore 中

4. 跨平台支援：

- 在不同作業系統上都能正常運作
- 自動處理不同系統的環境變數格式

5. 簡單易用：

- 只需幾行程式碼即可載入環境變數
- 支援多種檔案格式和編碼

使用範例：

```python
from dotenv import load_dotenv
import os

# 載入 .env 檔案
load_dotenv()

# 讀取環境變數
db_user = os.getenv('DB_USER')
print(f"DB_USER: {db_user}")
db_password = os.getenv('DB_PASSWORD')
print(f"DB_PASSWORD: {db_password}")
```

```
DB_USER: root
DB_PASSWORD: my-password
```

問：Python 搜索模組規則好像很複雜！可以檢視 Python 搜索的目錄列嗎？

答：可以。

Python 提供系統模组中，把搜索目錄列存到 sys.path 列表物件 (list) 裡。

檢視搜索目錄列，例子如下：

```python
import sys

print(f'sys.path 物件類型: {type(sys.path)}\n')

print(f'Python 搜索的目錄列: {sys.path}\n')

# 獲取命令列參數
print(f"程式名稱: {sys.argv[0]}\n")
print(f"參數: {sys.argv[1:]}\n")

# 檢查 Python 版本
if sys.version_info < (3, 12):
    print(f"需要 Python 3.12 或更新版本")
else:
    print(f"Python 版本: {sys.version_info}\n")

# 修改預設編碼
print(f"預設編碼: {sys.getdefaultencoding()}\n")
```

```
sys.path 物件類型: <class 'list'>

Python 搜索的目錄列: ['/Users/michaelkao/miniconda3/envs/pex/lib/python313.zip', '/Users/michaelkao/miniconda3/envs/pex/lib/python3.13', '/Users/michaelkao/miniconda
3/envs/pex/lib/python3.13/lib-dynload', '', '/Users/michaelkao/miniconda3/envs/pex/lib/python3.13/site-packages']

程式名稱: /Users/michaelkao/miniconda3/envs/pex/lib/python3.13/site-packages/ipykernel_launcher.py

參數: ['--f=/Users/michaelkao/Library/Jupyter/runtime/kernel-v3997fd3d5d68f99143d78c38bc85399a633be6f08.json']

Python 版本: sys.version_info(major=3, minor=13, micro=2, releaselevel='final', serial=0)

預設編碼: utf-8
```

6.2 變數的有效範圍

複習變數命名規則：

前面學習過的指派指令中，知道變數命名規則，現在複習一下：

1. 變數（variable）的名稱可以是任何大小寫的英文字母或下底線（_）或者數字的組成。
2. 變數名稱開頭不可以是數字。

除名稱外，變數在程式中也有範圍限制。

Python 程式中，當一個變數名稱第一次出現時，我們可以想像成 Python 必須有一張變數名稱表上登記該變數及其相關物件資料，包括記憶體儲存地址等。這一張變數名稱表 Python 叫它為 Variable Scope（變數範圍）。而且一個模組或函式都至少有一張變數範圍表管理我們的變數。

若在不同的模組或函式中有同名的變數被重覆定義時，Python 就得依底下四種變數範圍規則來認定變數的有效範圍。這四種範圍規則依序如下：

1. 局域變數 (Local variable) 範圍：局域變數只存在某一段程式區域中有效。例如上面學習過的定義函式指令中，參數列中的變數都是局域變數。
2. 外圍變數 (Enclosing variable) 範圍：外圍變數定義在外面縮格層的程式區塊中而內層程式區塊仍能有效訪問。內層變數必須用 'nonlocal' 來宣告。
3. 全域變數 (Global variable) 範圍：全域變數相對於上面的局域變數在整個程式中都有效。變數必須用 'global' 來宣告。
4. 內建變數 (Built-in variable) 範圍：Python 系統內建在 builtin 模組內的變數。

當一個變數須要被訪問時，Python 依上面 LEGB 規則順序搜索變數的定義。若還找不到變數的定義範圍的話，Python 列印出錯誤信息並停止執行。

問：可以檢查變數是屬那一種範圍嗎？

答：可以。

Python 提供兩個函式：locals () 和 globals ()

1. 'locals()'：返回一個局域字典供查詢變數是否在局域範圍
2. 'global()'：返回一個全域字典供查詢變數是否在全域範圍

局域和全域變數範圍的例子如下：

```python
# 局域和全域變數範圍的例子
def test():

    # 宣告全域變數 y
    print('\t 宣告全域變數 y')
    global y

    # 指派局域變數 x
    x = 'test()指派的局域變數 x'
    print(f'\t x = \'{x}\'')

    y = 'test()指派的全域變數 y'
    print(f'\t y = \'{y}\'')

    # 檢查變數有效範圍
    if 'x' in locals():
        print('\t test()中的 x 是局域變數')

    if 'y' in globals():
        print('\t test()中的 y 是全域變數\n')


print('主程式開始...')

# 指派全域變數 x
x = '主程式指派的全域變數 x'
print(f'x = \'{x}\'\n')

print(f'呼叫 test()中...')
test()

print('返回主程式...')
print(f'y = \'{y}\'')

if 'y' in globals():
    print(f'主程式中的 y 是全域變數\n')
```

```
主程式開始...
x = '主程式指派的全域變數 x'

呼叫 test()中...
         宣告全域變數 y
         x = 'test()指派的局域變數 x'
         y = 'test()指派的全域變數 y'
         test()中的 x 是局域變數
         test()中的 y 是全域變數

返回主程式...
y = 'test()指派的全域變數 y'
主程式中的 y 是全域變數
```

外圍變數範圍的例子如下：

```python
# 外圍變數範圍的例子

# 外層函式
def outer():
    print('\t 在 outer() 外圍範圍...')
    x = '指派外圍變數 x'
    y = '指派外圍變數 y'

    print(f'\t x = \'{x}\'')
    print(f'\t y = \'{y}\'\n')

    # 內層函式
    def inner():
        print('\t\t 在 inner() 局域範圍...')

        x = 'inner() 指派局域變數 x'
        print(f'\t\t x = \'{x}\'')

        # 宣告外圍變數 y
        print('\t\t 宣告外圍變數 y')
        nonlocal y
        y = 'inner() 指派外圍變數 y'
        print(f'\t\t y = \'{y}\'\n')

    print('\t 呼叫 inner()...')
    inner()
    print('\t 返回 outer() 外圍範圍...')
    print(f'\t x = \'{x}\'')
    print(f'\t y = \'{y}\'')

# 主程式
print('在主程式模組範圍...')
outer()
```

```
在主程式模組範圍...
         在 outer() 外圍範圍...
         x = '指派外圍變數 x'
         y = '指派外圍變數 y'

         呼叫 inner()...
                 在 inner() 局域範圍...
                 x = 'inner() 指派局域變數 x'
                 宣告外圍變數 y
                 y = 'inner() 指派外圍變數 y'

         返回 outer() 外圍範圍...
         x = '指派外圍變數 x'
         y = 'inner() 指派外圍變數 y'
```

內建變數範圍的例子如下：

```python
# 內建變數範圍的例子
import builtins

print(f'"builtins" 模組中內建變數 (屬性和方法)如下:')
print(f'{dir(builtins)}\n')

list_1 = [5, 1, 4, -8, 9]

num_min = min(list_1)
print(f'例子1: min({list_1}) = {num_min}')
print(f'例子2: abs({num_min}) = {abs(num_min)}')
```

```
"builtins" 模組中內建變數 (屬性和方法)如下:
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'E
xception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'Ind
entationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameE
rror', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessL
ookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'Syntax
Warning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeEr
ror', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc
__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callabl
e', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'execfile', 'fi
lter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'it
er', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'r
eversed', 'round', 'runfile', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

例子1: min([5, 1, 4, -8, 9]) = -8
例子2: abs(-8) = 8
```

6.3 比較列表與元組的效能測試

上面學習過的元組 (tuple) 與列表 (list) 兩者非常的類似，都是一個儲存資料的「容器」 物件，可以存入有順序的序列元素 (element)。複習一下！

元組和列表的差異：

1. 元組一旦建立，就不能修改內容，而列表可以修改。因為元組是不可變形 (Immutable Objects) ，列表卻是可變形的物件 (Mutable Object)。
2. 元組使用「小括號」，而列表使用「方括號」。
3. 如果元組裡只有一個元素，後方必須加上「逗號」(多個元素就不用)。

使用元組的好處：

1. 讀取速度比串列快。
2. 佔用的空間比較少。
3. 資料更安全 (因為無法修改)。

两者效能測試程式如下：

```python
"""Performance Testing between List and Tuple
列表與元組的效能測試
"""

import random
import glog as log
from glogTime import func_timer_decorator
from sys import getsizeof


@func_timer_decorator
def timeList(L):
    # 列表運算
    num = len(L)
    for x in L:
        i = random.randint(0, num - 1)
        j = random.randint(0, num - 1)
        _ = abs(L[i] - L[j])
        log.debug(f'x={x}, i={i}, j={j}')


@func_timer_decorator
def timeTuple(T):
    # 元組運算
    num = len(T)
    for x in T:
        i = random.randint(0, num - 1)
        j = random.randint(0, num - 1)
        _ = abs(T[i] - T[j])
        log.debug(f'x={x}, i={i}, j={j}')


from absl import app
from absl import flags

FLAGS = flags.FLAGS

flags.DEFINE_boolean('debug', False, 'Produces debugging output.')


def main(argv):
    if FLAGS.debug:
        log.setLevel("DEBUG")
    else:
        log.setLevel("INFO")

    # Number of times to process Lists vs Tuples
    maxNum = 250_000

    # 檢視佔用空間
    L = list(range(maxNum))
    T = tuple(range(maxNum))
    log.info(f'佔用空間  list[0-{maxNum-1}] = {getsizeof(L):,} bytes')
    log.info(f'佔用空間 tuple[0-{maxNum-1}] = {getsizeof(T):,} bytes\n')

    # 測試時間
    log.info(f'測試列表操作時間...')
    timeList(L)

    log.info(f'測試元組操作時間...')
    timeTuple(T)


if __name__ == '__main__':
    """
    在 command 視窗下執行程式並存檔如下：
    python <此檔案名稱> --debug 2> sample/perfListvsTuple.log

    在 JupyterNotebook 視窗下執行程式並存檔如下：
    !python <此檔案名稱> --debug 2> sample/perfListvsTuple.log
    """
    app.run(main)
```

注意：

上面程式 JupyterNotebook 執行上相容性可能有問題。改採用 Window 視窗下指令執行應該就可以。

```
!python perfListvsTuple.py --debug 2> sample/perfListvsTuple.log
```

--nodebug 模式

```
I1009 15:52:10.859112 4487075328 perfListvsTuple.py:53] 佔用空間
list[0-249999] = 2,000,056 bytes
I1009 15:52:10.859323 4487075328 perfListvsTuple.py:54] 佔用空間
tuple[0-249999] = 2,000,040 bytes

I1009 15:52:10.859386 4487075328 perfListvsTuple.py:57] 測試列表操作時間...
I1009 15:52:10.859450 4487075328 glogTime.py:21] 開始: timeList
I1009 15:52:12.273510 4487075328 glogTime.py:26] 結束: timeList 共花
0:00:01.413939 秒.

I1009 15:52:12.273718 4487075328 perfListvsTuple.py:60] 測試元組操作時間...
I1009 15:52:12.273791 4487075328 glogTime.py:21] 開始: timeTuple
I1009 15:52:13.528272 4487075328 glogTime.py:26] 結束: timeTuple 共花
0:00:01.254396 秒.
```

--debug 模式檢視檔案 'sample/perfListvsTuple.log' 如下：

```
I1009 15:53:49.059457 4534523392 glog.py:56] Log level set to DEBUG
I1009 15:53:49.087857 4534523392 perfListvsTuple.py:53] 佔用空間
list[0-249999] = 2,000,056 bytes
I1009 15:53:49.088048 4534523392 perfListvsTuple.py:54] 佔用空間
tuple[0-249999] = 2,000,040 bytes

I1009 15:53:49.088122 4534523392 perfListvsTuple.py:57] 測試列表操作時間...
I1009 15:53:49.088192 4534523392 glogTime.py:21] 開始: timeList
...
I1009 15:54:05.647750 4534523392 glogTime.py:26] 結束: timeList 共花
0:00:16.559461 秒.

I1009 15:54:05.647827 4534523392 perfListvsTuple.py:60] 測試元組操作時間...
I1009 15:54:05.647898 4534523392 glogTime.py:21] 開始: timeTuple
I1009 15:54:05.647974 4534523392 perfListvsTuple.py:30] x=0, i=205024,
...
I1009 15:54:21.435698 4534523392 glogTime.py:26] 結束: timeTuple 共花
0:00:15.787733 秒.
```

結論是：使用元組的確比列表讀取速度比串列快，而且佔用記憶體空間比較少。