簡介

Python 是很獨特的程式語言。身為一名軟體開發者，我已經開始愛上 Python 的本質與特色。能夠寫好 Python 程式會產生一種特別的藝術美感。我很喜歡為某些問題找出最具「Python 風格（Pythonic）」的解決方案，然後再回頭思考，嗯，這應該找不出其他答案了吧？

可惜的是，多年來我出於本能，希望透過我原本熟悉的其他程式語言的觀點來看待 Python。雖然我能閱讀和寫出 Python 程式，但我未能看到某種做法的明顯特質。這就像只依賴翻譯字典來說西班牙語一樣。我能寫 Python，但我無法真正以 Python 的方式來思考。這個語言的本質特點對我而言是模糊的。

當我開始真正理解 Python，真正以 Python 的方式來思考時，我對這個語言有了全新的喜愛。解決方案變得顯而易見，程式設計變得讓人愉悅，而不再是一個謎團。

當一位全新的開發者開始學習 Python 時，他們沒有太多先入為主的觀念。他們沒有「原生程式語言」來阻礙對 Python 的初次接觸。但對於一位已經熟悉其他程式語言的開發者來說，轉向 Python 的過程在某些方面會更加困難。他們不僅需要學習新的東西，而且在很多方面還需要「忘掉」一些舊有的知識。

本書會您在這趟學習旅程中很好的一本導引指南。

碁峰
www.gotop.com.tw

本書適用讀者

如果您已經熟悉其他程式語言，現在想要學 Python，而且不想浪費時間在初學者導向的課程，那麼這本書很適合您。我將專注於「Python 風格（Pythonic）」的寫作方式，只會提供對底層一般性程式設計概念最少且針對性的解釋。

如果您是一位中階程度的 Python 開發者，您也會發現這本書很有用。儘管我自己多年來一直在使用 Python，但直到最近對於某些主題才「恍然大悟」。這本書提供了我希望當初一開始學習時就能得到的解釋。

如果您還沒學過程式設計，別灰心。市面上有數百本優秀的書籍和資源可以作為第一門程式語言學習 Python 的材料。我特別推薦 Eric Matthes 的《Python 程式設計的樂趣｜範例實作與專題研究的 20 堂程式設計課，第三版》（碁峰資訊，2023 出版）或 Al Sweigart 的《Python 自動化的樂趣｜搞定重複瑣碎&單調無聊的工作，第二版》（碁峰資訊，2020 出版）。學習之後，您可以回到這本書來強化和擴充您所學到的知識。

「簡單」的真意

從表面上看，這本書所討論的主題看起來一點也不簡單。您可能會想知道這麼厚的一本書怎麼可能會是「簡單」的！當我為本書英文版取名時用了「簡單」的字樣，我是在描述對這些主題的回顧觀點，而不是前瞻觀點。我們應該認識到，任何值得學習的主題，在初次接觸時都會感到難以克服。同樣地，任何一個適合現有軟體開發者的解釋都應該具有足夠的深度，以使其完全脫離不符合前瞻的「簡單」標籤。

在這本書中，我希望把每個主題的解釋都十分清晰，以至於在每個章節結束時，讀者都會覺得這裡的概念很明顯清楚。不論這個主題一開始看起來多麼複雜，讀者最終應該會留下一個「真的很簡單」的印象，這時候讀者可以確信自己已經像一位原生的 Python 開發者這樣思考了。

為了達到這種程度的理解，通常我會從最低層、最明確的形式開始解釋這個主題。一旦確定了這個基礎，我就會逐層增加，最終達到最常用的隱含和慣用形

碁峯
www.gotop.com.tw

式。這樣的做法希望能幫助讀者，對於程式語言中每個特性的工作原理有一個扎實且舒適的理解。

本書內容

本書共分為五個部分（Part）。不同於許多針對初學者的課程，我會假設讀者希望盡快開始撰寫高品質的程式碼（而非不是太過簡化的教學範例）。我的方法需要讀者配合進行一些初步的工作，但這樣做會有回報，可以確保讀者能更輕鬆地將新知識應用於實際專案中。

Part I：「Python 環境」（第 1 章至第 4 章）會讓您扎實地踏進 Python 的世界：介紹它的理念、工具、基本語法和專案結構。這會為您撰寫真正的程式碼打下堅實的基礎。

Part II：「必學的基本結構」（第 5 章至第 8 章）探索了 Python 的基本結構元素（變數、函式、類別和例外處理），並教您如何充分運用和發揮其潛力。

Part III：「資料與流程」（第 9 章至第 12 章）涵蓋了控制執行流程和操作資料的多種獨特方式。這部分的內容討論了資料結構、迴圈、迭代、生成器、共常式、檔案和 2 進位資料等主題。

Part IV：「進階概念」（第 13 章至第 17 章）揭開了讓您的程式碼更強大的進階技巧，包括繼承、反射和並行處理。這裡的內容涉及大多數書籍和教學課程只會略過的「可怕」主題。

Part V：「程式碼之外的議題」（第 18 章至第 20 章）教您如何測試、除錯和部署真實的專案。

最後，第 21 章則綜述了讀者在 Python 開發的旅程中可以繼續探索的未來眾多方向。

碁峰
www.gotop.com.tw

第 6 章
函式與 lambda

函式（function）是程式設計中最基本的概念之一，但 Python 的函式蘊含了許多意想不到的多功能特性。您可能還記得在第 3 章中提到過，函式是第一級物件，所以它們和其他物件一樣同等對待。這是事實，再加上動態型別的強大功能，讓函式的運用有著無限的可能性。

Python 完全支援**函數式程式設計**（**functional programming**），這是一種獨特的範式，我們可以在網路上不斷聽到「lambda」或是匿名函式這樣的術語名詞。如果您熟悉像 Haskell 或 Scala 這樣的程式語言，本章的很多概念可能對您來說是相當熟悉的。然而，如果您更習慣物件導向程式設計，像是 Java 或 C++，那麼這可能是您第一次接觸到這些概念。

在學習 Python 時，早點深入理解函數式程式設計是有道理的。您完全可以寫出符合慣用 Python 風格寫法的程式碼，而不需要建立類別（參考第 7 章）。相較之下，函式和函數式程式設計的概念支撐了 Python 中許多最強大的功能。

碁峰
www.gotop.com.tw

理論回顧：函數式程式設計

在深入研究 Python 函式之前，您需要先了解函數式程式設計範式。

如果您是從像 Haskell、Scala、Clojure 或 Elm 這樣的純函數式程式語言轉來的，這裡的內容可能不會給您太多有用的新資訊。您可以直接跳到「Python 函式基礎知識」這一小節。

如果您不是從純函數式語言轉來的，就算您之前用過一些函數式程式設計的原則，我建議您還是跟著這裡的內容繼續學習。大多數開發者都不知道這種範式所涵蓋的內容有多廣。

「函數式」是什麼？

要了解函數式程式設計是什麼，就必須先理解它不是什麼。您很可能之前都是用**程序式程式設計**或**物件導向程式設計**。這兩種範式都是命令式的語法，您會透過特定、具體的步驟來描述如何達成目標。

程序式程式設計是圍繞著控制區塊來組織建構，並且非常著重在**控制流程**。物件導向程式設計則是圍繞著類別和物件來組織建構，並且著重在狀態，特別是這些物件的屬性（成員變數）。（詳見第 7 章）

函數式程式設計是以**函數**為中心的。這個範式被認為是**宣告式**的語法，也就是說，問題被分解成抽象的步驟。程式邏輯與數學上是一致的，而且在不同的程式語言之間基本上是沒有不同的。

在函數式程式設計中，您對每一個步驟都撰寫一個函數。每個函數接受一個輸入並產生一個輸出，這是自成一體的，只做一件事情，它不關心程式的其他部分。函數也沒有狀態，這表示它們在彼此呼叫之間不會儲存資訊。一旦函數結束，所有區域名稱都會結束作用範圍。每次在相同的輸入上呼叫函數，它都會產生相同的輸出結果。

最重要的是，函數不應該有副作用，意思是它們不應該改變任何東西。如果您將一個串列傳給一個純函數，這個函數不應該改變那個串列。相反地，它應該輸出一個全新的串列（或預期的其他值）。

函數式程式設計最大的優點在於可以改變任何一個函數的實作，而不影響其他事物。只要輸入和輸出是一致的，任務是怎麼完成都無所謂。這種程式碼

碁峰
www.gotop.com.tw

和更緊密耦合的程式碼相比，它更容易進行除錯和重構，在緊密耦合的程式碼中，每個函式都依賴於其他函式的實作。

純或不純？

很容易以為只要涉及到函式和匿名函式（lambda），就算是在「做函數式程式設計」，這裡再次強調，這個範式是圍繞**純函式**來組織建構的，這些函式沒有副作用或狀態，每個函式只執行單一任務。

Python 的函數式程式設計行為通常被認為是「不純」的，主要是因為存在可變的資料型別。若想要確保函式沒有副作用，需要額外的努力，正如您在第 5 章所學的內容。

在適當的情況下，您可以在 Python 中撰寫純粹的函數式程式碼。然而，大多數 Python 程式設計師選擇只借鑒函數式程式設計的特定觀念和概念，並將它們與其他範式結合在一起運用。

在實際應用中，**除非您有明確且合理的原因**，**不然遵循**函數式程式設計的規則通常會有最好的效果。這些規則，從嚴格到寬鬆，排列如下所示：

1. 每個函式應該只做一件特定的事情。
2. 函式的實作不應該影響程式的其他部分的行為。
3. 避免副作用！唯一的例外是當一個函式屬於一個物件時。在這種情況下，該函式只應該能夠修改該物件的成員（詳見第 7 章）。
4. 一般來說，函式不應該擁有（或受到）狀態的影響。提供相同的輸入應該總是產生相同的輸出。

第 4 條規則最有可能有例外，特別是牽涉到物件和類別的情況。

總而言之，您會發現把整個大型的 Python 專案寫成純函數式是有點不切實際的做法。所以，最好的方式是將這個範式的原則和概念融入到您的程式風格中。

函數式程式設計的迷思

對於函數式程式設計有一個常見的誤解是它避免使用迴圈。事實上，因為迭代在這個範式中是基本的功能（您馬上就會看到），所以迴圈是必需的。觀念是避免處理控制流程，所以遞迴（函數呼叫自己）通常比手動迴圈更受歡

碁峰 www.gotop.com.tw

迎。不過，您並不是都能避免使用迴圈。如果在程式中出現一些迴圈也不用擔心。您的主要重點應該是寫出純函數。

了解一點，函數式程式設計並不是萬能的解決方案，它有很多優勢，特別適用於某些情況，但也不是沒有缺點。有些重要的演算法和集合，例如並查集和雜湊表，在純函數式程式設計中可能無法有效地實作，甚至無法實作。在某些情況下，這種範式的效能比替代方案還差，而且記憶體使用量更高。在純函數式程式碼中實作並行處理是十分困難的。

這些議題很快就會變得相當具有技術性。對於大多數開發者來說，只要了解純函數式程式設計存在這些問題就足夠了。如果您發現自己需要更多的資訊，關於這些問題在網路上有許多白皮書和討論可供查閱。

函數式程式設計是您知識庫中很棒的一部分，但要準備好將它與其他程式設計範式和方法結合運用。在程式設計中並沒有萬靈丹的解決方案。

Python 函式基礎知識

在第 3 章我稍微提到了函式的觀念。在這個基礎上，本章內容將逐步建立一個更複雜的範例。

首先，我會建構一個函式，其功能是能夠擲指定面數的單一骰子：

🗁Listing 6-1: dice_roll.py:1a

```
import random


def roll_dice(sides):
    return random.randint(1, sides)
```

這裡定義了一個名為 roll_dice() 的函式，接受一個叫 sides 的參數。這個函式被視為純函式，因為它沒有副作用，接受一個值作為輸入，並返回一個新的值作為輸出。我使用 return 關鍵字從函式返回一個值。

random 模組有很多函式可以產生隨機值。這裡使用 random.randint() 函式在 Python 中生成一個假隨機數。它生成一個在 1 到 20 之間的隨機數（在這個例子中是 sides 的值），用的是 random.randint(1, 20)。

碁峰
www.gotop.com.tw

以下是我運用函式的方式：

🗁Listing 6-2: dice_roll.py:2a

```
print("Roll for initiative...")
player1 = ❶ roll_dice(20)
player2 = roll_dice(20)
if player1 >= player2:
    print(f"Player 1 goes first (rolled {player1}).")
else:
    print(f"Player 2 goes first (rolled {player2}).")
```

隨後我呼叫這個函式，並傳遞值 20 作為引數❶，所以這個函式呼叫實際上就像擲一個 20 面的骰子。第一次呼叫函式的返回值綁定到 player1，第二次呼叫的返回值綁定到 player2。

NOTE

「參數（parameter）」和「引數（argument）」這兩個詞常常會混淆。參數就是在函式定義中的「插槽（slot）」，可以接受一些資料，而引數則是在函式呼叫時傳遞給參數的資料。這兩個詞的定義不僅適用於 Python 程式設計，也適用於一般的電腦程式設計。

因為我把 roll_dice() 定義成一個函式，所以想要使用幾次都可以。如果我想改變它的行為，只需要修改這個定義函式的所在，然後所有使用這個函式的地方都會受到影響。

假設我想一次擲多個骰子，然後將結果以元組（tuple）的形式返回。我可以重新撰寫 roll_dice() 函式來達到這個目的：

🗁Listing 6-3: dice_roll.py:1b

```
import random

def roll_dice(sides, dice):
    return tuple(random.randint(1, sides) for _ in range(dice))
```

為了讓函式可以擲多個骰子，我們加入了第二個參數，叫做 dice，代表要擲的骰子數量。第一個參數 sides 仍然代表骰子的面數。

在函式內頂端看起來有點嚇人的程式碼，是一個**產生器運算式**（**generator expression**），我會在第 10 章中介紹。暫時不用太擔心，您可以先認定這行程式就是會為每個擲的骰子產生一個隨機數，然後將結果打包成一個元組。

碁峯 www.gotop.com.tw

因為現在的函式在呼叫時有第二個參數，所以我會傳入兩個引數：

🗀Listing 6-4: dice_roll.py:2b

```
print("Roll for initiative...")
player1, player2 = roll_dice(20, 2)
if player1 >= player2:
    print(f"Player 1 goes first (rolled {player1}).")
else:
    print(f"Player 2 goes first (rolled {player2}).")
```

回傳的元組可以被**拆解**（**unpacked**），意思是元組裡面的每個項目都會被綁定到一個名稱，我可以使用這些名稱來存取值。左邊列出的名稱數量（用逗號分隔）和元組裡面「值」的數量必須要相符，不然 Python 就會拋出錯誤。（詳見第 9 章有關拆解和元組的內容。）

遞迴

遞迴（**recursion**）就是一個函式呼叫了自己。在您需要重複整個函式的邏輯，但迴圈不適用或感覺太雜亂時很有幫助，如下所示的範例。

例如，回到之前的擲骰子的函式，我可以使用遞迴來達到完全相同的結果，而不是之前使用的那個生成器運算式（不過實務中，一般都認定生成器運算式在 Python 中更加符合慣例）。

🗀Listing 6-5: dice_roll_recursive.py:1a

```
import random


def roll_dice(sides, dice):
    if dice < 1:
        return ()
    roll = random.randint(1, sides)
    return (roll, ) + roll_dice(sides, dice-1)
```

我把這個函式呼叫得到的骰子點數存放在 roll 裡。接下來，在遞迴呼叫中，我保持 sides 參數不變，同時將要擲的骰子數減一，以便計算剛才擲出骰子的數量。最後，我把從遞迴函式呼叫返回的元組和這次函式呼叫所擲出的結果合併在一起，然後回傳這個更長的元組。

使用方式基本上與之前一樣：

碁峰 www.gotop.com.tw

🗁Listing 6-6: dice_roll_recursive.py:2a

```
dice_cup = roll_dice(6, 5)
print(dice_cup)
```

如果您將每個回傳的數值按照從最深一層的遞迴呼叫到最外一層的順序列印出來，就會看到以下的樣貌：

🗁Listing 6-7: 從 roll_dice(6, 5) 遞迴呼叫的返回處理

```
()
(2,)
(3, 2)
(6, 3, 2)
(4, 6, 3, 2)
(4, 4, 6, 3, 2)
```

當剩下的 dice 骰子數量為 0 或負數時，就會返回一個空的元組，而不是再次遞迴呼叫。如果不這樣做，遞迴就會無限執行。幸運的是，Python 在某個點上會中止遞迴，而不是讓它消耗掉電腦的所有記憶體空間（某些程式語言可能會這樣做）。遞迴深度是指尚未返回的遞迴函式呼叫的數量，Python 將其限制在約一千次。

NOTE

通常在 CPython 中，實際的最大遞迴深度是 997，即使根據原始程式碼應該是 1000，這真的有點奇怪。

如果遞迴深度超過了限制，整個程式就會停止並產生錯誤訊息：

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

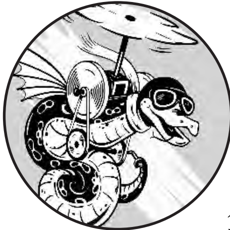
這就是為什麼在使用遞迴時，建立一些停止的機制是如此重要。在 roll_dice 函式中，這個停止的機制就在函式的最上面：

```
    if dice < 1:
        return ()
```

因為每次函式呼叫自己時 dice 都會遞減，早晚會變為 0。一旦變為 0，它就會回傳一個空的元組，而不會再產生更多的遞迴呼叫。隨後，剩下的遞迴呼叫就可以完成執行並返回。

碁峰 www.gotop.com.tw

第 16 章
非同步與並行



您應該很懂這些情況：您必須在老闆的要求下完成那份 TPS 報告、修復已經上線的錯誤、還得弄清楚是哪位同事借走了您的訂書機（又是 Jeff 對吧？），而且所有這些都必須在一天結束之前完成。您要如何搞定這一切呢？畢竟您無法複製自己，就算能複製，我想複印機的排隊隊伍已經排到公司門外了，所以您需要「**並行**（**concurrency**）」來應對這些任務。

在 Python 中也一樣。如果您的程式需要等待使用者輸入、在網路上發送資料、進行資料處理，同時還要更新使用者界面，那就需要「並行」處理這些任務，從而提升程式的反應速度。

在 Python 中，要做到並行處理有兩種選擇：一是使用執行緒（請參閱第 17 章），在這種情況下，作業系統負責多工處理（multitasking）；另一種則是使用非同步處理（asynchrony），由 Python 自己處理。本章內容將專注於後者。

碁峰
www.gotop.com.tw

理論回顧：並行

並行（Concurrency）就像是程式的多工（multitasking），快速地讓程式的注意力在多個任務之間切換。它不同於平行（parallelism），平行處理是指多個任務同時執行（請參閱第 17 章）。使用並行處理的程式受限於系統行程，在大多數 Python 的實作中，行程一次只能執行一件事情。

再回顧一下繁忙工作日的例子，您可以寫 TSP 報告、也可以找 Jeff 要回您的訂書針，但不能同時進行。就算您把 Jeff 叫到您的小辦公室，一邊填寫 TSP 報告一邊與他交談，您的注意力也是在這兩項任務之間分散切換，無論您的專注時間有多短。雖然從別人來看可能會覺得您是在同一時間做兩件事，但您實際上只是在不同的任務之間來回切換。

這有個重要的涵義：**並行處理實際上並不會加快執行時間**。總之，填寫 TSP 報告需要花費 10 分鐘，再花 5 分鐘詢問 Jeff。這兩個任務總共需要 15 分鐘，無論您是在與 Jeff 談話前完成報告還是分散注意力兩者兼顧。事實上，由於在不同任務之間切換需要額外的努力，所以使用並行處理可能會花更長時間，在程式設計中也是如此。這些任務在效能上受限於您大腦的能力，就像在電腦中的多工受限於到 CPU 速度。對於 CPU 效能受限的任務，並行處理並不會有什麼幫助。

在處理輸入 / 輸出效能受限的任務時，並行處理才真正能派上用場，例如在網路上接收檔案或等待使用者點按按鈕。舉例來說，請想像一下由於不知道管理層要多少份的原因，所以您必須隨時準備幾份會議議程的副本護貝裝訂好。每次護貝裝訂好一份議程大約需要幾分鐘的時間，而在這段時間內，您只需坐在那裡聽著護貝機的嗡嗡聲，這時您不需要注意和做些什麼。這不是個明智的時間運用方式，對吧？這是個輸入 / 輸出受限的任務，因為您的速度主要受到等待頁面護貝完成（輸出）的限制。現在假設您使用並行處理，將一頁紙放入護貝機，然後走開，到辦公室裡翻個底朝天尋找訂書針。每隔幾分鐘，您回去檢查一下護貝機，也許再送另一張紙進去，然後繼續尋找訂書針。當您在 Martha 的辦公桌抽屜找到訂書針時（不好意思，訂書針不是 Jeff 拿的！），會議議程的護貝也就完成了。

並行處理也對提升程式的反應速度有所幫助：即使在程式執行長時間或重要的任務，比如複雜的資料分析，它仍然可以對使用者輸入作出回應或更新進度條。實際上，沒有一個任務比以前更快，但程式不會卡住。

碁峰 www.gotop.com.tw

最後要說的是，並行處理對於定期執行的任務也很有用，例如不論程式的其餘部分在做什麼都每隔五分鐘儲存一次暫存檔案。

在 Python 中的非同步

如前所述，在 Python 中實作並行處理有兩種方式。**執行緒化**（**Threading**），也稱為**先佔式多工處理**（**pre-emptive multitasking**），牽涉到讓作業系統透過**執行緒**的單一執行串流中執行各個任務來管理多工處理。這多個執行緒仍然共用同一個系統**行程**（**process**，或譯**處理程序**、**進程**），這是執行中電腦程式的實例。如果您在電腦中開啟監控系統的工作管理員，就可以看到電腦上執行的執行緒清單，其中的任何一個行程都可以有多個執行緒。

傳統的執行緒有一些問題，這就是為什麼我會在第 17 章再回過頭來說明和討論。在 Python 中實作並行處理的另一種做法是**非同步**（**asynchrony**），也被稱為**協作多工**（**cooperative multitasking**）。這也是在 Python 中實作並行處理的最簡單方式，但這並不代表是輕鬆容易的！作業系統只是將您的程式視為在單一行程中執行，具有單一執行緒，實際上是 Python 本身來管理多工處理，再加上一些您的協助，這樣就能避開一些執行緒帶來的問題。然而，在 Python 中編寫出好的非同步程式碼還是需要一些事前思考和規劃。

請記住，非同步處理不等同於平行處理。在 Python 中，有一個叫做**全域直譯器鎖**（**Global Interpreter Lock**，**GIL**）的機制，不論系統有多少個核心可用，都會確保單個 Python 行程受限於單個 CPU 核心。因此，非同步處理和執行緒都無法實作平行處理。這聽起來可能像是設計缺陷，但試圖從 CPython 中消除 GIL 的努力已經證明比想像中更具技術挑戰性，而且到目前為止的結果是效能不佳。到本書截稿時，這些努力中最重要的因素之一，也就是 Larry Hastings 的 Gilectomy 進展幾乎已經停滯不前。這個 GIL 會讓 Python 執行更加順暢。

NOTE

有一些繞過 GIL 的方法，例如 Python 擴充模組，因為它們是用 C 語言編寫並以編譯的機器碼執行。每當您的邏輯走出 Python 直譯器，它也就超出了 GIL

碁峰 www.gotop.com.tw

的控制範圍，可以平行執行。我在這本書中不會深入探討這個議題，但在第 21 章會提到一些達到此目標的擴充模組。

早期 Python 實作非同步處理需要借助第三方程式庫，例如 Twisted。後來，Python 3.5 版加了原生實作非同步處理的語法和功能。但這些功能直到 Python 3.7 版才變得穩定，因此許多關於非同步處理的文章和網路上討論在目前來看是有點過時了。要取得最新資訊的最佳途徑始終是官方的說明文件。本書作者已盡力保持與 Python 3.10 版的最新資訊同步。

Python 借鑒了 C# 語言的兩個關鍵詞：async 和 await，以及一種特殊的協程，實作了非同步處理。（許多其他程式語言如 JavaScript、Dart 和 Scala 也實作了類似的語法）非同步執行由**事件迴圈**（**event loop**）管理和執行，事件迴圈負責多工處理。Python 為此提供了標準程式庫中的 asyncio 模組，我們會在本章的範例中使用它。

值得注意的是，在我撰稿時，除了基本用法之外，就算對一些 Python 專家來說，asyncio 還是顯得太過複雜。因此，我會專注於與非同步處理相關的基本概念，盡量避免不必要的詳細解釋或使用 asyncio。您看到的大部分內容都是純粹的非同步處理技巧，另外我也會特別提出例外情況。

當您準備深入研究非同步處理這個主題時，可以選擇使用 Trio 或 Curio 程式庫。這兩個程式庫很使用者友善，它們有很好的說明文件，是針對初學者撰寫的，並且會經常為 asyncio 的開發者提供設計指導。憑藉本章的知識，您應該能夠利用它們的說明文件學會相關的運用。

Curio 是由 David Beazley 開發的，他是 Python 和並行處理方面的專家，他的目標是讓 Python 中的非同步處理變得更容易理解。官方說明文件可以連到網站 https://curio.readthedocs.io/ 找到，官網主頁還包含一些關於 Python 非同步程式設計的優秀演講連結，也包括一些指導您如何編寫自己的非同步處理模組的演講（雖然您可能永遠不需要這麼做）。

Trio 是以 Curio 為基礎開發的，它更進一步強調了該程式庫的簡單性和易用性目標。在我撰寫本文時，Trio 還被視為實驗性質的程式庫，但它仍然足夠穩定，可以用於上線的環境。大多數 Python 開發者最常建議使用 Trio。您可以連到 https://trio.readthedocs.io/en/stable/ 找到官方說明文件。

碁峰 www.gotop.com.tw

在這一節的前面的內容中有提到 Twisted 程式庫，它在 20 年前就將非同步處理加到 Python 中，比非同步處理加到核心功能的時間還早。它使用了一些過時的模式，而不是現代的非同步處理工作流模型，但它仍然是一個活躍且具有多種用途的程式庫。許多熱門的程式庫在其內部還是使用它來進行相關處理。想要更多資訊，請參閱 https://twistedmatrix.com/。

您可以在 https://docs.python.org/3/library/asyncio.html 找到 asyncio 的官方說明文件。我建議只有在您已經熟悉透過 Trio 或 Curio 以及執行緒（第 17 章）類似概念的非同步程式設計後，再深入研究 asyncio。您理解了非同步處理和並行處理的概念與模式後會有助於您理解 asyncio 說明文件。

請記住，非同步處理在 Python 和整個電腦科學領域仍然相對年輕。非同步工作流模型大約在 2007 年首次出現在 F# 語言中，是以 Haskell 在 1999 年左右引入的概念以及 1990 年代初的一些論文為基礎。相比之下，執行緒的相關概念則可追溯到 20 世紀 60 年代末。許多非同步處理中的問題仍然沒有明確或者已有解決方案。誰知道呢？也許您會成為第一個解決其中某個問題的人！

範例場景：Collatz 遊戲，同步版本

為了正確示範這些概念是怎麼運作的，我會建立一支小程式，可以從並行處理中受益。由於所牽涉的問題比較複雜，本章和下一章的內容我會完全把焦點放在這個範例，這樣您就能熟悉其運作細節。

我會從一個**同步**（**synchronous**）處理的版本開始，這樣您會對我的舉例有清晰的概念。這個範例的複雜性將展示出並行處理中涉及的一些常見問題。對於這些概念的範例來說，簡單是效能的敵人。

在例子中，我會玩弄數學中被稱為 **Collatz 猜想**的奇怪現象，運作方式如下：

1. 從任意正整數 n 開始。
2. 如果 n 是偶數，序列中的下一個數應該是 n / 2。
3. 如果 n 是奇數，序列中的下一個數應該是 3 * n + 1。
4. 如果 n 是 1，停止。

碁峯 www.gotop.com.tw

第 18 章
套裝與發布

就算是寫出了世界上最好的程式碼，如果您沒有發布，也不會有多大的價值。一旦您的專案可以執行，就應該在繼續開發之前，先弄清楚如何將它套裝和發布。問題是，在 Python 中，套裝處理（packaging）有時候會讓人感到抓狂。

一般來說，問題不在於程式碼的套裝處理，這相對容易，而是在處理程式碼的相依性，尤其是非 Python 的相依性。發布（Distribution）可能會成為一個令人煩惱的問題，即使對經驗豐富的程式設計師來說也是如此，部分原因是因為 Python 是在各種不同的情境下被使用。然而，如果您了解事物的運作原理，就有堅實的基礎可以克服挫折，並成功把可執行的程式碼部署出去。

在這一章中，我會簡單解說如何套裝和發布 Python 專案，首先透過 Python Package Index，然後轉為可安裝的二進位檔案。為了實際示範，我會帶您瀏覽一個我自己寫的應用程式：Timecard。這個專案很適合當作範例，因為它既包含 Python 和系統的相依性，也包含一些非程式碼的資源，所有這些都需要以某種方式處理。這個程式專案的儲存庫可以在 GitHub 上找到：https://github.com/codemouse92/timecard/。我設定了 packaging_example 分支，只包含專案本身，不包含任何套裝檔案，您可以用它來練習。

碁峯
www.gotop.com.tw

本章主要是簡介套裝的過程，不論您打算使用哪種工具。然而，為了避免在解說眾多的套裝工具時讓大家迷失方向，我們會使用廣受歡迎的 setuptools 套件來套裝 Timecard 程式專案，因為它提供了現代套裝的大部分常用模式。

在這個過程中，我會提到許多其他常見的工具，包括一些受歡迎的第三方替代工具，但大部分不會深入介紹。如果您想更深入了解這些工具，書中有提供官方說明文件的連結供您參考。此外，如果您想深入了解套裝的一般知識，其中最好的資源是由社群維護的「Python Packaging User Guide」，您可以連到官網 https://packaging.python.org/ 找到這份使用手冊，其中包含許多更進階的主題，例如套裝 CPython 二進位擴充模組。

為了讓您對套裝的整個概念少一些不適，我也想提一下，Python 套裝的代表吉祥物是一隻快樂的紫色鴨嘴獸：是一種奇怪的小生物，看起來似乎由許多不同的部分組成，它很可愛、友善、還會下蛋。（最後一部分是個雙關語，在本章結束時您可能會理解。）如果您現在對套裝的概念和運用感到害怕，不妨去連到 https://monotreme.club/ 網站，感受一下 Python 套裝吉祥物的可愛和友善。他們還有貼紙。

規劃您的套裝處理

在您開始進行套裝處理之前，需要明確地知道您想達成什麼目標、為什麼要這樣做，以及如何進行。不幸的是，很少有開發者認識到這一必要性，大都是毫無計畫地著手編寫套裝腳本，缺乏真正的方向。這種臨時性的套裝方案可能會面臨脆弱性、不必要的複雜性、在不同系統間的可攜性不足，以及依賴關係的安裝出現問題或缺失等情況。

貨物崇拜程式設計的危險

為了鼓勵使用好的套裝工具和慣例，許多善意的人會提供 setup.py、setup.cfg 或其他在套裝中使用的檔案範本，並建議複製和修改這些範本來運用。這種做法被稱為**貨物崇拜程式設計**（**Cargo Cult Programming**），廣泛應用於 Python 的套裝處理中，這對專案和生態系統都有害。因為配置檔被盲目複製，錯誤、取巧和反模式就會像攜帶瘟疫的兔子一樣擴散。

碁峰
www.gotop.com.tw

在套裝處理中的錯誤不一定會導致安裝失敗或顯示有用的錯誤訊息。舉例來說，套裝某個程式庫（library）時的錯誤可能在使用該程式庫作為相依時才會表現出來。在這方面，程式的發布尤其讓人煩惱，因為許多相關的錯誤是針對特定平台的。有時您可以安裝套件，但程式會以出乎意料的方式失敗！問題追蹤程式中充滿了這類問題的記錄，其中很多記錄只是簡單地標記為「Cannot reproduce（無法重現）」，也許是因為這個錯誤只發生在某個特定版本的 Linux 和搭配特定版本的某個系統程式庫。

總之，別陷入貨物崇拜程式設計的誘惑！雖然從一個經過驗證的範本開始是合理的，但要設法了解其中每一行程式碼。仔細閱讀說明文件。確保您沒有遺漏範本可能忽略的必要參數，使用了一些已過時的選項，甚至調換了正確的程式碼行順序。（是的，最後一點確實是個重要問題。）

感恩 Python Packaging Authority（PyPA）工作小組已經做了很多工作，讓社群遠離上述這種情況。PyPA 是個半官方的團體，由希望改進 Python 套裝體驗的 Python 社群成員組成，任何維護專案的人都可以參加。他們詳細解釋了套裝範本每個部分的原由和目的，以及他們維護的 Python Packaging User Guide。

套裝的說明

正如您所發現的，有很多種方式可以對 Python 專案進行套裝和發布。我會把焦點放在 PyPA 建議的技術，但還有很多其他選擇。

無論最終您使用哪種套裝技術，它們必須能產生一個相當具有可攜性和穩定性「就能運作」的套件。您的終端使用者應該能夠在任何支援的系統上執行一組可預測的步驟，並成功執行您的程式碼。雖然安裝說明在不同平台之間可能有所不同，但您應該盡量減少終端使用者需要遵循的步驟數量。步驟越多，出錯的機會就越多！保持簡單，並盡量遵循每個平台的建議套裝和發布慣例。如果終端使用者在安裝您的專案時持續回報問題或混淆，那就要**修正套裝**了。

決定套裝的目標

最終，任何套裝工具的目標都是建立一個單一的**產物**，通常是一個檔案，可以安裝在終端使用者的環境上，不管是個人電腦、伺服器、虛擬機器，或其他硬

碁峰 www.gotop.com.tw