前言

本書是為誰而寫的？

如果你想學習程式設計，你就來對地方了。Python 是最適合初學者的程式語言之一，同時也是最搶手的技能之一。

你來得正是時候，因為現在學習程式設計可能比以往任何時候都容易。有了像 ChatGPT 這樣的虛擬助理（virtual assistants），你就不需要獨自學習了。在這本書中，我會推薦你運用這些工具來加速學習的方法。

這本書主要是為從未寫過程式的人和有過一些其他程式語言經驗的人所寫的。如果你對 Python 有相當的經驗，你可能會覺得前幾章進展太緩慢。

學寫程式的挑戰之一是你必須學習兩種語言：一種是程式語言本身；另一種是我們用來談論程式的詞彙。若只學習程式語言，當你需要解讀錯誤訊息、閱讀說明文件、與他人對話或使用虛擬助理時，你很可能會遇到問題。如果你已經寫過一些程式，但尚未學習這第二種語言，我希望你會發現本書對你有所幫助。

本書的目標

撰寫這本書時，我試著小心處理詞彙。每個術語第一次出現時，我都會給出定義。每章結尾都有詞彙表（glossary），回顧介紹過的術語。

我也盡量保持簡潔。閱讀本書所花的心力越少，能用於程式設計的就越多。

但是光靠看書是無法學會程式設計的，你必須多加練習。為此，本書每章結尾都有習題（exercises），讓你可以演練所學內容。

如果你仔細閱讀並持續練習，你一定會有所進步。但我現在要警告你，學習程式設計並不容易，即使是經驗豐富的程式設計師，也可能會遭遇挫折。在學習過程中，我會提出一些策略來幫助你編寫正確的程式並修正錯誤的程式。

導覽本書

本書的每一章都建立在前一章的基礎之上，因此你應該循序閱讀，並在繼續邁進之前花時間做習題。

前六章介紹算術（arithmetic）、條件式（conditionals）和迴圈（loops）等基本元素。這些章節還會介紹程式設計中最重要的概念——函式（functions），以及運用函式的強大方法——遞迴（recursion）。

第 7 章和第 8 章介紹字串（strings）——它們可以表示字母、字詞和句子——以及用來處理它們的演算法。

第 9 章到第 12 章介紹 Python 的核心資料結構——串列（lists）、字典（dictionaries）和元組（tuples），它們是編寫高效程式的強大工具。第 12 章介紹分析文字和隨機生成新文字的演算法。這類演算法是大型語言模型（large language models，LLM）的核心，因此本章將讓你大致了解 ChatGPT 等工具是如何運作的。

第 13 章是關於在長期儲存區——檔案和資料庫——中存放資料的方法。作為練習，你可以編寫程式來搜尋檔案系統，並找出重複的檔案。

第 14 章到第 17 章介紹物件導向程式設計（object-oriented programming，OOP），這是組織程式和所用資料的一種方法。許多 Python 程式庫都以物件導向風格寫成，所以這些章節將幫助你了解它們的設計——並定義你自己的物件。

這本書的目標並非涵蓋整個 Python 語言。取而代之，我把焦點放在能以最少的概念提供最大能力的語言子集。然而，Python 還有很多其他功能，你可以用它們有效率地解決一些常見的問題。第 18 章就介紹那些功能。

最後，第 19 章是我對繼續你程式設計之旅的臨別感言與建議。

第一章

程式設計作為一種思維方式

本書的第一個目標是教你如何使用 Python 撰寫程式。但學習程式設計意味著學習一種新的思考方式，因此本書的第二個目標是幫助你像電腦科學家（computer scientist）那樣思考。這種思考方式結合了數學、工程和自然科學的一些最佳特點。就跟數學家一樣，電腦科學家使用形式語言（formal languages）來表示想法（ideas），特別是計算（computations）。如同工程師，他們會設計東西，將元件組裝成系統，並評估各種替代選擇之間的利弊取捨。就像科學家，他們觀察複雜系統的行為，形成假說，並檢定預測。

我們將從最基本的程式設計元素開始，逐步深入。在本章中，我們將看到 Python 如何表示數字（numbers）、字母（letters）和字詞（words）。你將學習如何進行算術運算。

你也會開始學習程式設計的詞彙，包括運算子（operator）、運算式（expression）、值（value）和型別（type）等術語。這些詞彙非常重要，你需要透過它們來理解本書的其他內容、與其他程式設計師溝通，以及使用和理解虛擬助理。

算術運算子

算術運算子（arithmetic operator）是表示算術計算的符號（symbol）。舉例來說，加號 + 會執行加法：

```
30 + 12
```

```
42
```

減號 - 是執行減法的運算子：

```
43 - 1
```

```
42
```

星號 * 執行乘法：

```
6 * 7
```

```
42
```

而正斜線 / 則執行除法：

```
84 / 2
```

```
42.0
```

請注意，除法的結果是 `42.0`，而不是 `42`。這是因為在 Python 中有兩種型別的數字：

- **整數（integers）**，代表整數（whole numbers）值，以及
- **浮點數（floating-point numbers）**，表示帶有小數點（decimal point）的數字。

如果將兩個整數相加、相減或相乘，結果都會是整數。但若是兩個整數相除，結果會是浮點數。Python 提供另一個運算子 **//** 來執行**整數除法（integer division）**。整數除法的結果永遠是一個整數：

```
84 // 2
```

```
42
```

整數除法也稱為「地板除法（floor division）」，因為它總是向下捨入（朝向「地板」）：

```
85 // 2
```

```
42
```

```
int('126') / 3
```

```
42.0
```

如果你的字串包含數字和小數點，你可以使用 `float` 將其轉換為浮點數：

```
float('12.6')
```

```
12.6
```

撰寫一個大型整數時，你可能會想在數字組之間使用逗號（commas），例如 `1,000,000`。在 Python 中這是一種合法的運算式，但結果並不是一個整數：

```
1,000,000
```

```
(1, 0, 0)
```

Python 會將 `1,000,000` 解讀為以逗號分隔的整數序列。我們稍後會學習更多關於這種序列的知識。

你可以使用底線（underscores）來使大數字更容易閱讀：

```
1_000_000
```

```
1000000
```

形式語言和自然語言

自然語言（Natural languages）是人們使用的語言，例如英語（English）、西班牙語（Spanish）和法語（French）。它們不是由人類設計出來的，而是自然演化出來的。

形式語言（Formal languages）是人們為了特定的應用而設計的語言。舉例來說，數學家使用的記號（notation）就是一種形式語言，特別善於表示數字和符號之間的關係。同樣地，程式語言（programming languages）也是為了表達計算而設計的形式語言。

雖然形式語言和自然語言有一些共同的特點，但也有重要的差異：

歧義性（*Ambiguity*）

自然語言充滿歧義，人們利用情境的線索（contextual clues）和其他資訊來加以應付。形式語言被設計成幾乎或完全沒有歧義，也就是說任何程式都剛好只有一種意思，不論情境為何。

冗餘性（*Redundancy*）

為了彌補歧義並減少誤解，自然語言會有所贅餘。因此，自然語言通常比較冗長囉嗦。形式語言的冗餘較少，也較為簡潔。

字面性（*Literalness*）

自然語言充滿了慣用語（idiom）和隱喻（metaphor）。形式語言所代表的就是它們字面上的意思，不多也不少。

因為我們都是說自然語言長大的，所以有時候很難適應形式語言。形式語言的密度比自然語言高，所以閱讀所需的時間也比較長。此外，結構也很重要，所以從上至下，從左至右的閱讀方式不一定是最好的。最後，細節也很重要。拼寫和標點符號上的小錯誤，在自然語言中可以忽略不計，但在形式語言中卻會造成很大的差異。

除錯

程式設計師會犯錯。基於離奇的原因，程式錯誤被稱為 **bug（臭蟲）**，而追蹤錯誤的過程則稱為 **debugging（除錯）**。

程式設計，尤其是除錯，有時候會帶出強烈的情緒。如果你正糾結於一個難以解決的臭蟲，你可能會感到憤怒、傷心或羞愧。

為這些反應做好準備可能有助於處理它們。有種做法是將電腦視為具有某些優點（例如速度和精確度）和特定弱點（例如缺乏同理心和無法掌握大局）的員工。

你的任務則是成為一名優秀的經理人：想辦法運用優點並緩和缺點。並想辦法利用你的情緒來處理問題，而不要讓你的反應干擾你有效工作的能力。

學習除錯可能會令人沮喪，但這是一種寶貴的技能，對程式設計以外的許多活動都很有用。在每一章的最後，都有像這樣的小節，列出我對於除錯的建議。希望它們能有所幫助！

第八章

字串和正規表達式

字串（strings）與整數、浮點數和 Boolean 值不同。字串是一種**序列（sequence）**，這表示它包含以特定順序排列的多個值。在本章中，我們將了解如何存取構成字串的那些值，並使用處理字串的函式。

我們還會使用正規表達式（regular expressions），它是在字串中尋找模式（patterns）並進行搜尋（search）並取代（replace）等運算的強大工具。

作為練習，你將有機會將這些工具應用於一種名為 Wordle 的文字遊戲。

字串是一種序列

字串是一種字元序列（sequence of characters）。**字元（character）**可以是字母（幾乎包含所有全音素文字中的字母）、數字、標點符號或空白。

你可以使用方括號運算子（bracket operator，或稱「中括號運算子」）從字串中選出一個字元。這個範例述句會從 `fruit` 中選擇字元編號 `1`，並將其指定給 `letter`：

```
fruit = 'banana'
letter = fruit[1]
```

方括號中的運算式是**索引（index）**，之所以稱為 index，是因為它 *indicate*（指出）要選擇序列中的哪個字元。但結果可能與你預期的不同：

結果是 TypeError。在錯誤訊息中，object 是字串，而 item 是我們嘗試指定的字元。就目前而言，可以把**物件（object）**和值（value）當作同一回事，但我們之後會再修正這個定義。

造成這個錯誤的原因在於，字串是**不可變的（immutable）**，這表示你無法變更現有的字串。你所能做的頂多就是創建作為原始字串變體的一個新字串：

```
new_greeting = 'J' + greeting[1:]
new_greeting
```

```
'Jello, world!'
```

本範例將一個新的首字母串接至 greeting 的一個切片上。這對原始字串沒有影響：

```
greeting
```

```
'Hello, world!'
```

字串比較

關係運算子（relational operators）也適用於字串。若要檢視兩個字串是否相等，我們可以使用 == 運算子：

```
word = 'banana'

if word == 'banana':
    print('All right, banana.')
```

```
All right, banana.
```

其他關係運算子對於將字詞按照字母順序（alphabetical order）排列非常有用：

```
def compare_word(word):
    if word < 'banana':
        print(word, 'comes before banana.')
    elif word > 'banana':
        print(word, 'comes after banana.')
    else:
        print('All right, banana.')
```

```
compare_word('apple')
```

```
apple comes before banana.
```

Python 處理大寫（uppercase）和小寫（lowercase）字母的方式與人們處理的方式不同。所有的大寫字母都在所有的小寫字母之前，因此：

```
compare_word('Pineapple')
```

```
Pineapple comes before banana.
```

要解決這個問題，我們可以在進行比較之前，先將字串轉換為某種標準格式，例如全小寫。如果你要防禦拿著鳳梨（pineapple）當武器的人，請切記這一點。

字串方法

字串提供執行各種實用運算的方法。方法（method）類似於函式——它接受引數並回傳一個值——但語法不同。舉例來說，`upper` 方法接受一個字串，並回傳字母全大寫的一個新字串。

這裡不使用函式語法 `upper(word)`，而是使用方法語法 `word.upper()`：

```
word = 'banana'
new_word = word.upper()
new_word
```

```
'BANANA'
```

點號運算子的這種用法指明了方法的名稱 `upper`，以及要套用方法的字串之名稱 `word`。空括弧表示此方法不需要引數。

方法呼叫（method call）被稱為**調用（invocation）**；在本例中，我們會說我們在 `word` 上調用 `upper`。

第十二章

文字的分析與生成

到目前為止，我們已經涵蓋了 Python 的核心資料結構——串列、字典和元組——以及一些用到它們的演算法。在本章中，我們將使用它們來探索文字分析（text analysis）和 Markov 生成（generation）：

- 文字分析是描述文件中字詞之間統計關係的一種方式，像是一個字詞後面跟著另一個字詞的機率。
- Markov 生成是用與原始文字相似的字詞（words）和片語（phrases）生成新文字的一種方式。

這些演算法與大型語言模型（large language model，LLM）的某些部分類似，而大型語言模型是聊天機器人（chatbot）的關鍵元件。

我們會先計算每個字詞在一本書中出現的次數。接著我們會檢視成對的字詞（pairs of words），並製作一個串列，列出每個字詞後面可能出現的字詞。我們將製作一個簡單版的 Markov 產生器（generator），作為練習，你將有機會製作一個更一般化的版本。

不重複的字詞

作為文字分析的第一步，讓我們來讀取一本書——Robert Louis Stevenson 所著的《*The Strange Case of Dr. Jekyll and Mr. Hyde*》——並計數不重複的字詞（unique words）。本章的 notebook 中有下載這本書的指示：

```
filename = 'dr_jekyll.txt'
```

標點符號

要識別文字中的字詞，我們需要處理兩個問題：

- 當一行中出現連接號（dash）時，我們應該用空格（space）來取代它，這樣當我們使用 `split` 時，那些字詞就會被分開。
- 拆分字詞之後，我們可以使用 `strip` 剝除標點符號。

為了處理第一個問題，我們可以使用下列函式，它接受一個字串，將連接號換成空格，分割字串，並傳回結果串列：

```
def split_line(line):
    return line.replace('—', ' ').split()
```

請注意，`split_line` 只會取代連接號（dashes），不會取代連字號（hyphens）。

這裡有一個例子：

```
split_line('coolness—frightened')
```

```
['coolness', 'frightened']
```

現在，為了移除每個字詞開頭和結尾的標點符號，我們可以使用 `strip`，但我們需要一個串列，包含被視為標點符號的字元。

Python 字串中的字元採用 Unicode，這是一種國際標準，可用來表示幾乎所有全音素文字（alphabet）中的字母、數字、符號、標點符號等。`unicodedata` 模組提供一個 `category` 函式，我們可以用它來分辨哪些字元是標點符號。給定一個字母，它會回傳一個字串，其中包含該字母所屬類別的資訊：

```
import unicodedata

unicodedata.category('A')
```

```
'Lu'
```

`'A'` 的分類字串（category string）是 `'Lu'`——`'L'` 表示它是字母（letter），而 `'u'` 表示它是大寫（uppercase）。

由於 strip 是從開頭和結尾移除字元，因此含有連字號的字詞（hyphenated words）不會受到影響：

```
clean_word('pocket-handkerchief')
```

```
'pocket-handkerchief'
```

現在這裡是一個使用 split_line 和 clean_word 來辨識書中不重複字詞的迴圈：

```
unique_words2 = {}
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        unique_words2[word] = 1

len(unique_words2)
```

```
4005
```

根據這個更嚴格的字詞定義，大約會有四千個不重複的獨特字詞。而且我們可以確認，最長的那些字詞已經被清理過了：

```
sorted(unique_words2, key=len)[-5:]
```

```
['circumscription',
 'unimpressionable',
 'fellow-creatures',
 'chocolate-coloured',
 'pocket-handkerchief']
```

現在讓我們看看每個字詞的使用次數。

字詞頻率

以下迴圈會計算每個獨特字詞的頻率（frequency）：

```
word_counter = {}
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        if word not in word_counter:
            word_counter[word] = 1
        else:
            word_counter[word] += 1
```

我們第一次看到一個字詞時，會將其頻率初始化為 1。若之後再次看到相同的字詞，我們會遞增它的頻率。

若要檢視哪些字詞出現得最頻繁，我們可以使用 `items` 從 `word_counter` 取得鍵值與值對組（key-value pairs），然後根據對組的第二個元素（也就是頻率）排序。首先，我們定義一個函式來選取第二個元素：

```
def second_element(t):
    return t[1]
```

現在我們可以使用有兩個關鍵字引數（keyword arguments）的 `sorted`：

`key=second_element`

項目會根據字詞的頻率排序。

`reverse=True`

項目會以相反順序排序，最常出現的字詞會先出現。

```
items = sorted(word_counter.items(), key=second_element, reverse=True)
```

以下是最常出現的五個字詞：

```
for word, freq in items[:5]:
    print(freq, word, sep='\t')
```

```
1614    the
972     and
941     of
640     to
640     i
```

在下一節中，我們會將這個迴圈封裝在一個函式中。我們會用它來展示一項新功能——選擇性參數（optional parameters）。

選擇性參數

我們用過會接受選擇性參數的內建函式。例如，`round` 接受一個叫作 `ndigits` 的選擇性參數，表示要保留多少位小數：

```
round(3.141592653589793, ndigits=3)
```

```
3.142
```

但不只是內建函式，我們也可以寫出帶有選擇性參數的函式。舉例來說，以下函式接受兩個參數 `word_counter` 和 `num`：

```
def print_most_common(word_counter, num=5):
    items = sorted(word_counter.items(), key=second_element, reverse=True)

    for word, freq in items[:num]:
        print(freq, word, sep='\t')
```

第二個參數看起來像一個指定（assignment）述句，但它不是——它是一個選擇性參數。

若使用一個引數（argument）來呼叫這個函式，`num` 會得到**預設值（default value）**，也就是 `5`：

```
print_most_common(word_counter)
```

```
1614    the
972     and
941     of
640     to
640     i
```

生成文字

我們可以使用上一節的結果來產生新的文字，其中連續字詞之間的關係與原始文字相同。運作方式如下：

- 從文字中出現的任何字詞開始，我們查找其可能的後繼字詞，然後隨機選擇一個。
- 然後，使用選定的字詞，我們查找其可能的後繼字詞，並隨機選取一個。

我們可以重複這個過程，生成任意多的字詞。舉例來說，讓我們從 `'although'` 這個字開始。這裡是可能跟在它後面的字詞：

```
word = 'although'
successors = successor_map[word]
successors
```

```
['i', 'a', 'it', 'the', 'we', 'they', 'i']
```

我們可以使用 `choice` 以相等的機率從串列中選取：

```
word = random.choice(successors)
word
```

```
'i'
```

如果同一個字詞在串列中出現超過一次，它被選中的可能性就較大。

重複這些步驟，我們可以使用下面的迴圈來產生更長的序列：

```
for i in range(10):
    successors = successor_map[word]
    word = random.choice(successors)
    print(word, end=' ')
```

```
continue to hesitate and swallowed the smile withered from that
```

結果聽起來更像一個真正的句子，但仍沒有太大的意義。

我們可以在 `successor_map` 中使用一個以上的字詞作為鍵值，效果會更好。舉例來說，我們可以製作一個字典，從每個 bigram 或 trigram 映射到後繼字詞的串列。作為練習，你將有機會實作這種分析，看看結果會是怎樣。