前言

Python 程式語言擁有可能難以掌握的獨特優點與魅力。許多熟悉其他語言的程式設計師，會以一種狹隘的視野來接觸 Python，而非欣然接納其完整的能力。有些程式設計師則是往另一個方向走得太遠，過度使用可能在日後造成很大問題的 Python 功能。

本書深入介紹撰寫程式的 *Pythonic* 思維，即運用 Python 的最佳方式。我假設你已經對 Python 語言有了基本的了解，本書就是建立在這個基礎上。新手程式設計師將學習 Python 關鍵功能的最佳實務做法。有經驗的程式設計師將學會如何自信地擁抱新工具。

透過這本書，我希望能幫助你使用 Python 來實現你的目標，不論目標為何，或者至少幫助你在程式設計的旅途中獲得更多的樂趣。

本書涵蓋的內容

本書的每一章都包含一系列廣泛但相關的做法（items）。請隨意在各個做法之間跳轉，並按照你的興趣來進行。每個做法都包含簡明而具體的指引，解釋如何更有效地編寫 Python 程式。做法中包含該做什麼、要避免什麼、如何取得適當的平衡，以及為什麼這是最佳選擇的建議。做法之間會相互參考，讓你在閱讀時更容易拾遺補闕。

第三版涵蓋 Python 語言到版本 3.13 的內容（請參閱做法 1：「了解你使用的是哪個版本的 Python」）。與第二版相比，本書包含了 35 個全新的做法。第二版中的做法大多都經過修訂並納入其中，而許多做法都有了大幅度的

更新。對於某些做法，由於最佳實踐方式隨著 Python 在過去五年的成熟而演進，我的建議也完全改變了。

Python 採取「內建電池（batteries included）」的理念來設計其標準程式庫（standard library）。這些內建套件（built-in packages）有許多與 Python 慣用語是如此緊密地結合在一起，簡直就像是語言規格的一部分。完整的標準模組過於龐大，無法在這本書中全部涵蓋，但我已將我認為至關重要且需要知道和使用的模組納入其中。

Python 也有一個充滿活力的社群模組（community-built modules）生態系統，這些模組以很有價值的方式擴充了 Python 語言。儘管我在各個做法中提到了需要了解的重要套件，但這本書並不打算成為詳盡的參考手冊。同樣地，儘管 Python 的套件管理（package management）很重要，但我還是避免深入討論相關細節，因為它正在快速改變和演化。

第 1 章：Pythonic 思維

Python 社群已經開始使用 *Pythonic* 這個形容詞來描述遵循特定風格的程式碼。Python 的慣用語（idioms）是在使用 Python 語言和與其他程式設計師合作的過程中逐漸形成的。本章涵蓋在 Python 中做最常見事情的最佳方式。

第 2 章：字串和切片

Python 內建了處理字串（string）和序列（sequence）的語法、方法和模組。這些功能非常重要，幾乎在每個程式中都會看到它們。它們使 Python 成為一種優秀的語言，善於剖析（parsing）文字、檢查資料格式，以及與電腦使用的低階二進位表示法介接。

第 3 章：迴圈與迭代器

處理循序資料（sequential data）是程式的關鍵需求。Python 中的迴圈（loops）對於涉及內建資料型別、容器型別和使用者定義類別的常見任務來說，感覺自然又強大。Python 也支援迭代器（iterators），它們能以更函式化的方式來處理任意的資料串流，並帶來顯著的好處。

第 4 章：字典

Python 內建的字典（dictionary）型別是一種多用途的資料結構，可用於程式中的簿記工作。與簡單的串列（lists）相比，字典在新增和移除項目時提供更好的效能。Python 還擁有特殊的語法和相關的內建模組，可以強化字典，使其功能超越其他語言中雜湊表（hash tables）的預期表現。

第 5 章：函式

Python 中的函式有許多額外的功能，可以讓程式設計師的生活更輕鬆。有些類似於其他程式語言的功能，但許多是 Python 所獨有的。本章將介紹如何使用函式來闡明意圖、促進重複使用並減少臭蟲（bugs）。

第 6 章：概括式和產生器

Python 有特殊的語法來快速迭代串列、字典和集合，以產出衍生的資料結構。它也允許函式以漸進的方式回傳由可迭代的值所構成的一個串流（stream）。本章將介紹這些功能如何提供更好的效能、減少記憶體用量，並改善可讀性。

第 7 章：類別和介面

Python 是一種物件導向（object-oriented）的語言。要在 Python 中完成工作，通常需要編寫新的類別，並定義它們如何透過其介面（interfaces）和階層架構（hierarchies）來互動。本章將介紹如何使用類別來表達物件的預期行為。

第 8 章：元類別和屬性

元類別（metaclasses）和動態屬性（dynamic attributes）是 Python 強大的功能。然而，它們也能讓你實作出極度怪異和出乎意料的行為。本章涵蓋使用這些機制的常見慣用語，以確保你遵循**最不意外法則**（*rule of least surprise*）。

第 9 章：共時性和平行處理

藉由執行緒（threads）和非同步協程（asynchronous coroutines）等功能，Python 可以輕鬆寫出看似正在同時做許多不同事情的共時程式（concurrent

programs）。Python 也可以透過系統呼叫、子行程（subprocesses）和特殊模組來平行處理工作。本章將介紹如何在這些有細微差異的情況下，以最佳的方式運用 Python。

第 10 章：穩健性

使程式在遇到意外情況時能夠可靠運作，與製作具備正確功能的程式同樣重要。Python 有內建的功能和模組，可以幫助強化你的程式，讓它們在各種情況下都能保持穩健。

第 11 章：效能

Python 具備各種功能，可讓程式以相對較低的努力程度，達到令人驚豔的效能。透過這些功能，不僅能從主機系統榨出極致效能，還能保有 Python 高階本質所帶來的生產力優勢。

第 12 章：資料結構與演算法

Python 包含許多標準資料結構（data structures）和演算法（algorithms）經過最佳化的實作，可以幫助你以最少的努力達到高效能。這個語言還提供了通過實戰考驗的資料型別（data types）和輔助函式（helper functions），用於常見任務（例如，處理貨幣和時間），讓你可以專注於程式的核心需求。

第 13 章：測試與除錯

無論使用何種語言撰寫程式碼，你都應該測試程式碼，但對 Python 而言，測試尤其重要。Python 的動態特質會以獨特的方式增加執行時期出錯的風險。幸運的是，它們也使編寫測試和診斷程式故障變得更容易。本章將介紹 Python 用來測試和除錯的內建工具。

第 14 章：協作

協同開發 Python 程式需要你深思熟慮如何編寫程式碼。即使你是單獨工作，你也會想要了解如何使用他人所寫的模組。本章將介紹讓人們能夠在 Python 程式上共同作業的標準工具和最佳實務做法。

做法 18 使用 zip 平行處理迭代器

在 Python 中，你常常會遇到由相關物件組成的許多串列。串列概括式（list comprehensions）能讓你輕鬆地取得來源串列，並將運算式套用到每個項目，藉此產生另一個衍生串列（請參閱做法 40：「使用概括式取代 map 和 filter」）。舉例來說，這裡我拿一個名稱串列來建立一個對應的串列，其中包含每個名稱的字元數：

```
names = ["Cecilia", "Lise", "Marie"]
counts = [len(n) for n in names]
print(counts)

>>>
[7, 4, 5]
```

衍生串列（counts）中的項目，會透過它們在序列中的對應位置，與來源串列（names）中的項目產生關聯。為了在單一迴圈中存取這兩個串列的項目，我可以迭代來源串列（names）的長度，並使用 range 產生的偏移量來索引任一個串列。舉例來說，這裡我使用平行並進的索引動作（parallel indexing）來判斷哪個名稱最長：

```
longest_name = None
max_count = 0

for i in range(len(names)):
    count = counts[i]
    if count > max_count:
        longest_name = names[i]
        max_count = count

print(longest_name)

>>>
Cecilia
```

問題在於整個 for 述句在視覺上過於雜亂。索引運算 —— names[i] 和 counts[i] —— 使得程式碼難以閱讀。透過相同的迴圈索引 i 對兩個陣列進行索引操作似乎是多餘的。我可以使用 enumerate 內建函式（請參閱做法

```
>>>
Cecilia
Lise
Marie
```

新項目 "Rosalind" 並沒有出現在輸出中。為什麼？這正是 zip 的運作方式。它會持續產出元組，直到其中一個被包裹的迭代器耗盡為止。它的輸出長度只會跟最短的輸入一樣長。如果過早截斷對你的程式而言可能是種問題，你可以將 strict 關鍵字引數傳入 zip —— 這是自 Python 3.10 以來的新選項 —— 如果任何輸入在其他輸入之前耗盡，這會導致所回傳的產生器提出例外：

```
for name, count in zip(names, counts, strict=True):  # 改變了
    print(name)
```

```
>>>
Cecilia
Lise
Marie
Traceback ...
ValueError: zip() argument 2 is shorter than argument 1
```

又或者，你也可以使用 itertools 內建模組中的 zip_longest 函式來解決這種截斷問題，用預設值填補缺少的項目（請參閱做法 24：「考慮使用 itertools 來處理迭代器和產生器」）。

要記得的事

- ✦ 內建的 zip 函式可用來平行迭代多個迭代器。
- ✦ zip 會建立一個產出元組的惰性產生器（lazy generator），它可以用於無限長的輸入。
- ✦ 如果你提供給 zip 的迭代器長度不一，zip 會默默地將其輸出截斷為最短的迭代器。
- ✦ 如果你想要確保這種默默截斷的情形不會發生，而且有迭代器長度不符時應該導致執行時期錯誤，請將 strict 關鍵字引數傳遞給 zip 函式。

做法 55 優先選用公開屬性而非私有屬性

在 Python 中，類別的屬性（attributes）只有兩種可見性（visibility）：公開（*public*）和私有（*private*）：

```
class MyObject:
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

公開屬性可讓任何人在物件上使用點運算子（dot operator）以進行存取：

```
foo = MyObject()
assert foo.public_field == 5
```

私有欄位（private fields）是透過在屬性名稱前面加上雙底線（double underscore）來指定。它們可由包含它們的類別（containing class）之方法直接存取：

```
assert foo.get_private_field() == 10
```

然而，從類別外部直接存取私有欄位會引發例外：

```
foo.__private_field

>>>
Traceback ...
AttributeError: 'MyObject' object has no attribute '__private_field'
```

類別方法也能存取私有屬性，因為它們是在外圍的 `class` 區塊中宣告的：

```
class MyOtherObject:
    def __init__(self):
        self.__private_field = 71

    @classmethod
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field

bar = MyOtherObject()
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

做法 106 當精確度至關重要時，請使用 decimal

若要撰寫與數值資料互動的程式碼，Python 會是極佳的程式語言。Python 的整數（integer）型別可以表示任何實際大小的值。它的雙精度浮點（double-precision floating point）型別符合 IEEE 754 標準。此語言也提供一種用於虛數值（imaginary values）的標準複數（complex number）型別。然而，這些不足以應付每一種情況。

舉例來說，假設我想要計算透過可攜式衛星電話（portable satellite phone）撥打國際電話要向客戶收取的費用。我知道客戶通話的時間，以分鐘和秒計算（例如，3 分鐘 42 秒）。我也有從美國（United States）撥打電話到南極洲（Antarctica）的固定費率（例如，每分鐘 1.45 美元）。通話費用應該是多少？

使用浮點數運算，計算出的費用看起來很合理：

```
rate = 1.45
seconds = 3 * 60 + 42
cost = rate * seconds / 60
print(cost)

>>>
5.364999999999999
```

結果比正確數值（`5.365`）少了 `0.0001`，這是因為 IEEE 754 的浮點數表示法所致。我可能會想把這個數值無條件進位到 `5.37`，以適當地涵蓋客戶所產生的所有成本。然而，因為浮點數誤差，四捨五入到最接近的整數美分，實際上會減少最終費用（從 `5.364` 變成 `5.36`），而非增加（從 `5.365` 變成 `5.37`）：

```
print(round(cost, 2))

>>>
5.36
```

解決方案是使用 `decimal` 內建模組中的 `Decimal` 類別。`Decimal` 類別預設提供 28 個小數位數（decimal places）的定點（fixed point）數學運算 —— 而且如果需要，還能夠更高。這能避開 IEEE 754 浮點數中的精確度（precision）問題。這個類別也讓你對捨入（rounding）行為有更多的控制。