前言

> 計畫是這樣的：如果有人用了你看不懂的功能，那就直接斃了他，畢竟這樣做比學習新功能省事多了，而且不久後，倖存的程式設計師就只懂得編寫容易理解的、一小部分的 Python 0.9.6 了（眨眼）。[1]
>
> —— 傳奇的核心開發者、*Zen of Python* 一文的作者，*Tim Peters*

「Python 是易學、強大的程式語言。」這是 Python 3.10 官方教學的第一句話（*https://fpy.li/p-2*），這句話說得沒錯，但問題在於，這種語言是如此易學和易用，使得很多實際撰寫 Python 的程式設計師都只用了它強大功能的一部分而已。

具備一定經驗的程式設計師可以在幾小時內寫出實用的 Python 程式，但隨著極富生產力的前幾個小時慢慢延伸成好幾星期或好幾個月，很多開發者會開始使用別種語言的「腔調」來撰寫 Python 程式。即使 Python 是你學會的第一種語言，教導你的學術機構和入門書籍也會小心翼翼地避免介紹它的獨特功能。

在指導用過其他語言的學生使用 Python 時，身為教師的我發現另一項問題：我們只記得已知的事情，而這也是本書想要解決的問題。用過其他語言的人應該猜得到 Python 支援正規表達式與查詢文件。但是如果你沒看過 tuple unpacking（拆箱）或 descriptor（描述器），你應該不會去搜尋它們，最終不會去使用那些功能…因為它們是 Python 的獨門功能。

本書不是包羅萬象的 Python 參考書，本書的重點是 Python 獨特的功能，或是在其他熱門語言裡找不到的功能。本書也偏重核心語言和它的程式庫。我幾乎不談不屬於標準程

1 出自 comp.lang.python Usenet group，2002 年 12 月 23 日：“Acrimony in c.l.p”（*https://fpy.li/p-1*）。

式庫的程式包，即使到目前為止，Python 程式包清單已經有超過 60,000 個程式庫，而且其中好多程式庫都好用得不得了。

適合本書的人

本書是為了那些想要熟練地掌握 Python 3 的 Python 實踐型程式設計師而寫的。我在 Python 3.10 中測試了書中的範例，也在 Python 3.9 和 3.8 中測試了其中的絕大多數。如果範例需要使用 Python 3.10，我會清楚地標示出來。

如果你不確定你的 Python 知識是否足以閱讀本書，你可以先複習 Python 官方教學（*https://fpy.li/p-3*）的各種主題。除了一些新功能之外，本書不會解釋該教學介紹過的主題。

不適合本書的人

如果你正準備開始學習 Python，你可能會覺得這本書難以消化，不僅如此，如果你在學習 Python 的旅程中，太早閱讀這本書，你可能會誤以為所有的 Python 程式都必須使用特殊的方法與超編程（metaprogramming）技巧來編寫，太早學習抽象的概念不是好事，這和太早進行最佳化一樣。

這是一本五合一的書籍

我建議所有人都閱讀第 1 章，「Python 資料模型」。本書的核心讀者在看完第 1 章之後，應該可以輕鬆地跳到任何其他部分閱讀了，但每一個部分通常都假設你已經看了之前的章節。請將第 1 單元到第 5 單元視為本書裡的五本書。

這本書強調，你應該先使用現成的東西，再研究如何建構自己的東西。例如，在第 1 單元，第 2 章介紹了現成的 sequence 型態，包括一些比較冷門的型態，例如 `collections.deque`。我只會在第 3 單元介紹如何建構自訂的 sequence，該單元也會介紹如何利用 `collections.abc` 提供的抽象基礎類別（abstract base class，ABC）。如何建立自己的 ABC 在第 3 單元的更後面介紹，因為我認為在建立自己的 ABC 之前，你應該先習慣使用它。

這種做法有一些優點。第一，知道有哪些現成的工具可以避免你重新發明輪子。我們經常使用現成的集合類別，而不是自行編寫它們，所以我們要把更多注意力放在使用現成的工具上，而不是討論如何創造新的工具。我們也較常繼承既有的 ABC，而不是從新編寫 ABC。最後，我認為先瞭解抽象化的實際行為比較容易瞭解它們。

但這種做法也有缺點，那就是參考資料將會分散在各章裡。希望你可以體諒我的本意。

本書架構

本書的主題如下：

第 *1* 單元，資料結構

第 1 章介紹 Python Data Model（Python 資料模型），並解釋為何特殊方法（例如 `__repr__`）是讓所有型態的物件具備一致行為的關鍵。本書各處會更詳細地介紹各種特殊方法。此單元的其餘章節將介紹如何使用 collection 型態，包括 sequence、mapping、set 以及 str vs. bytes 分割 —— 這是讓大多數的 Python 3 使用者備感慶幸，也是讓 Python 2 使用者在遷移程式碼庫（codebase，以下簡稱「碼庫」）時極其痛苦的主因。本單元也會討論標準程式庫的高階類別建構器：具名 tuple 工廠（named tuple factory），以及 `@dataclass` decorator（修飾器）。我們將在第 2、3 與 5 章的各節討論 Python 3.10 新增的模式比對，這幾章的主題是 sequence 模式、mapping 模式，及類別模式。第 1 單元的最後一章討論物件的生命週期：參考（reference）、可變性（mutability），及記憶體回收（garbage collection）。

第 *2* 單元，函式即物件

本單元將討論「在這種語言裡，函式是一級物件」的概念，包括這句話是什麼意思、它如何影響一些流行的設計模式，以及如何利用 closure 來實作函式 decorator。這個部分也會討論 Python 的 callable 概念、函式屬性、自檢（introspection）、參數注解（parameter annotation），及 Python 3 新增的 `nonlocal` 宣告。第 8 章會介紹重要的新主題：函式簽章內的型態提示。

第 *3* 單元，類別與協定

這裡的焦點是「手工」製作類別，而不是使用第 5 章介紹的類別建構器。如同所有物件導向（OO）語言，Python 有它自己的功能，那些功能或許也可以在你我學過的類別型語言裡找到，或許找不到。本章解釋如何建構自己的 collection、抽象基礎

類別（ABC）、協定，以及如何處理多重繼承、如何實作運算子多載（在合理的時機）。第 15 章繼續探討型態提示。

第 4 單元，控制流程

傳統控制流程是以條件式、迴圈和子程序構成的，這個單元將討論超越傳統控制流程的語言結構和程式庫。我們從 generator（產生器）看起，然後討論 context manager（環境管理器）與 coroutine（協同程序），包括有難度但功能強大的 `yield from` 語法。第 18 章透過一個重要的範例來教你在一個簡單但功能豐富的語言直譯器裡使用模式比對。第 19 章「Python 的並行模型」是新的一章，介紹 Python 中並行與平行處理的替代方案、它們的限制，以及軟體架構如何讓 Python 在網路規模上運行。我重寫了關於非同步設計的一章，以強調語言的核心功能，例如 `await`、`async def`、`async for` 與 `async with`，並展示如何一起使用它們與 `asyncio` 和其他框架。

第 5 單元，超編程（Metaprogramming）

這個單元先回顧如何製作「可動態建立屬性的類別」，這種類別可以用來處理 JSON 資料組等半結構化資料。接下來討論熟悉的屬性機制，然後探討在 Python 的底層，物件屬性的存取如何透過 descriptor 來運作。本單元也會解釋函式、方法、descriptor 之間的關係。第 5 單元將一步一步地實作一個欄位驗證程式庫，來揭示一些微妙的問題，那些問題帶來最終章介紹的進階工具：類別 decorator 與 metaclass（元類別）。

實踐的方法

我們通常使用 Python 互動式主控台來探索這種語言及其程式庫。我認為應該強調這種學習工具的威力，尤其是對那些比較常使用靜態編譯型語言的讀者而言，因為那種語言通常不支援 read-eval-print（讀、執行、列印）迴路（REPL）。

`doctest`（*https://fpy.li/doctest*）是 Python 標準測試程式庫，它會模擬主控台執行階段，並驗證運算式的執行結果是否和所示的反應一致。我用 `doctest` 來檢查本書的大多數程式碼，包括主控台列表（listing）。你不需要使用 `doctest` 就可以跟著學習，甚至不需要知道它，`doctest` 的主要特點在於，它們看起來就像互動式 Python 主控台執行階段的紀錄，所以你可以輕鬆地自行嘗試範例。

為了解釋我們想要完成什麼事情，有時我會先展示 doctest，再展示可以讓它通過的程式碼。先確定我們要完成什麼工作，再思考怎麼完成它，可以幫助我們把注意力放在設計程式上。先寫測試程式是測試驅動開發法（TDD）的基礎，我發現它在教學時也很有用。如果你還不認識 `doctest`，你可以閱讀它的文件（*https://fpy.li/doctest*），以及探索本書的範例程式存放區（*https://fpy.li/code*）。

我也使用 `pytest` 來為一些比較大型的範例撰寫單元測試，我認為這種程式庫比標準程式庫的 `unittest` 模組更容易使用，功能也更強大。你可以在 OS 的命令 shell 裡輸入 `python3 -m doctest example_script.py` 或 `pytest` 來驗證本書大多數程式是否正確。位於範例程式存放區（*https://fpy.li/code*）的根目錄裡的 *pytest.ini* 組態檔確保 doctest 都是由 `pytest` 命令來收集並執行的。

肥皂箱：我的個人觀點

我從 1998 年以來，就一直在使用、教導及探討 Python，我很喜歡學習與比較各種程式語言、它們的設計，及它們背後的原理。在一些章節的結尾有「肥皂箱」專欄，用來表達我個人對 Python 與其他語言的觀點。如果你對這類討論沒有興趣，盡可跳過它們，它們絕對不是必要的內容。

本書網站：fluentpython.com

由於第二版介紹幾個新功能，包括型態提示、資料類別、模式比對，所以它比第一版多了將近 30% 的篇幅。為了讓這本書輕盈一些，我把一些內容移到 *fluentpython.com*。你會在某幾章的內容裡看到該網站文章的連結。這個網站也有一些配套章節。你可以從 O'Reilly Learning（*https://fpy.li/p-5*）訂閱服務取得完整的內容（*https://fpy.li/p-4*）。本書的範例程式存放區位於 GitHub（*https://fpy.li/code*）。

本書編排方式

以下是本書使用的字體規則：

斜體字（*Italic*）

代表新術語、URL、電子郵件地址、檔案名稱及副檔名。

第 1 章

Python 資料模型

> Guido 的語言設計美學很驚人。我認識的很多語言設計者都可以設計出理論上很優美的語言，但沒有人願意使用它們。然而，Guido 是少數能夠設計出理論上沒那麼優美的語言，卻可以讓人愉快地編寫程式的人。
>
> —— *Jython 建構者，AspectJ 共同創造者，.Net DLR 架構師，Jim Hugunin*[1]

Python 最棒的性質之一就是它的一致性。在使用 Python 一段時間之後，你就可以明智且正確地猜出新功能怎麼使用。

然而，如果你在學習 Python 之前學過其他的物件導向語言，你可能會覺得使用 `len(collection)` 而不是 `collection.len()` 很彆扭。這種明顯的彆扭只是冰山的一角，如果用正確的方式來理解，它是讓一切都 *Pythonic*（很 Python）的關鍵。這座冰山稱為 Python 資料模型，它是讓我們自己的物件能夠和最典型的語言功能良好互動的 API。

你可以將資料模型想成將 Python 描述成框架的東西，它將這個語言的元素的介面正式化，例如 sequence、函式、iterator、coroutine、類別、context manager…等等。

1 Samuele Pedroni 與 Noel Rappin 合著的 *Jython Essentials*（O'Reilly）的前言，「Story of Jython」（*https://fpy.li/1-1*）。

在使用框架時，我們會花很多時間來編寫被框架呼叫的方法。在利用 Python Data Model 來建構新類別時也一樣。Python 直譯器會呼叫特殊方法來執行基本的物件操作，這通常是以特殊語法來觸發的。特殊方法的名稱一定是以雙底線開頭，並以雙底線結尾。例如，`obj[key]` 語法是由 `__getitem__` 特殊方法支援的。為了計算 `my_collection[key]`，直譯器會呼叫 `my_collection.__getitem__(key)`。

要讓我們的物件支援以下的語言基本結構，並與之互動，我們就要編寫特殊方法：

- 集合（collection）
- 屬性存取
- 迭代（包括使用 `async for` 來進行非同步迭代）
- 運算子多載
- 函式與方法呼叫
- 字串表示與格式化
- 使用 `await` 來編寫非同步程式
- 建構和解構物件
- 使用 `with` 或 `async with` 陳述式來管理的環境

魔術與 dunder

魔術方法是特殊方法的俗稱，但如何唸出 `__getitem__` 這種特殊方法？我從作者和教師 Steve Holden 那裡知道，應稱之為 dunder-getitem。「dunder」是「double underscore before and after」的簡稱。這就是為什麼特殊方法也稱為 *dunder 方法*。*The Python Language Reference* 的「Lexical Analysis」（*https://fpy.li/1-3*）一章警告道：「在任何環境下以任何方式使用 `__*__` 這種名稱時，如果不遵守明確記載的用法，可能會在不提前警告的情況下發生故障。」

本章有哪些新內容

本章與第一版的差異不大，因為這是介紹 Python Data Model 的一章，而 Python Data Model 相當穩定。最主要的修改有：

- 在第 16 頁的「特殊方法概覽」的表格中，加入支援非同步程式設計與其他新功能的特殊方法。
- 在第 14 頁的「Collection API」中，我們在展示特殊方法用法的圖 1-2 裡加入 Python 3.6 新增的抽象基礎類別 `collections.abc.Collection`。

此外，在這裡與整本第二版裡，我採用 Python 3.6 新增的 f-string 語法，它比舊的字串格式化語法（使用 `str.format()` 方法與 `%` 運算子）更易讀，且通常更方便。

使用 `my_fmt.format()` 的理由之一在於，你必須在一個地方定義 `my_fmt`，但必須在另一個地方進行格式化操作。例如，當 `my_fmt` 有很多行，最好在常數裡定義時，或它一定來自組態檔或資料庫時。

很 Python 的撲克牌組

範例 1-1 很簡單，但它展示了僅實作 `__getitem__` 與 `__len__` 兩個特殊方法產生的強大效果。

範例 1-1　以撲克牌 sequence 來製作牌組

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

第一個要注意的是，我們使用 `collections.namedtuple` 來建構一個簡單的類別，以代表個別的撲克牌。我們使用 `namedtuple` 來建構物件的類別，它只有一堆屬性，沒有自訂的方

法，很像資料庫的紀錄。在這個範例裡，我們用它來表示牌組裡的每一張牌，例如這個主控台執行過程：

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

但是這個範例的重點是 FrenchDeck 類別。它很短，但發揮巨大的作用。首先，如同所有的標準 Python collection，牌組（deck）回傳它的撲克牌數量來回應 len() 函式：

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

讀取牌組的特定卡牌（例如第一張與最後一張）很簡單，因為 __getitem__ 方法提供這個功能：

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

需要另外寫一個方法來隨機抽牌嗎？不用。Python 已經有一個從 sequence 中隨機抽取項目的函式了：random.choice。我們用它來處理牌組實例：

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

我們看到，透過特殊方法來利用 Python Data Model 有兩項優點：

- 類別的使用者不需要記憶標準操作的方法名稱（如何獲得項目的數量？它是 .size() ？ .length() ？還是別的名稱？）。
- 你比較容易受惠於豐富的 Python 標準程式庫，並且避免重新發明輪胎，例如 random.choice 函式。

但它有更多好處。

因為我們的 `__getitem__` 將工作委託給 `self._cards` 的 `[]` 運算子，所以我們的牌組自動支援 slicing（切段）。下面是檢查全新牌組最上面三張牌的寫法，以及從索引值 12 開始選取 ACE，並且每次都跳過 13 張牌的寫法：

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

只要實作 `__getitem__` 特殊方法，牌組就是 iterable（可迭代的）：

```
>>> for card in deck:  # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

我們也可以反向迭代牌組：

```
>>> for card in reversed(deck):  # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```

在 doctest 裡的省略符號

可以的話，我會從 `doctest`（*https://fpy.li/doctest*）擷取 Python 主控台文字來確保準確性。當輸出太長時，我會用省略符號（...）來表示被省略的部分，例如上面程式的最後一行。在這種情況下，我會用 `# doctest: +ELLIPSIS` 指令來讓 doctest 可 pass。如果你在互動式主控台裡嘗試這些範例，可以完全省略 doctest 指令。

迭代通常是隱性的。如果 collection 沒有 `__contains__` 方法，`in` 運算子會做循序掃描。典型的例子：我們的 `FrenchDeck` 類別可使用 `in`，因為它是可迭代的。我們來試一下：

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

那排序呢？撲克牌的大小通常是按照數字大小（ace 最大）來排列，然後用花色來排列，依序是黑桃（最高）、紅心、方塊與梅花（最低）。下面是一個按照這個規則來定義撲克牌的點數大小的函式，對於梅花 2，它會回傳 0，對於黑桃 A 則回傳 51：

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

我們可以用 `spades_high` 以遞增順序列出牌組：

```
>>> for card in sorted(deck, key=spades_high):  # doctest: +ELLIPSIS
...      print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

雖然 `FrenchDeck` 隱性地繼承 `object`，但它大多數的功能都不是繼承來的，而是利用資料模型與組合（composition）。我們的 `FrenchDeck` 透過實作特殊方法 `__len__` 與 `__getitem__`，展現出 Python 標準 sequence 的行為，並獲得 Python 語言的核心功能（即迭代與 slicing）與標準程式庫帶來的好處，例如範例中使用的 `random.choice`、`reversed` 與 `sorted`。拜組合之賜，`__len__` 與 `__getitem__` 可以將所有工作委託給 `list` 物件，`self._cards`。

洗牌呢？

目前的 `FrenchDeck` 不能洗牌，因為它是不可變的（*immutable*）（譯注：在 Python 裡，immutable 是指無法被修改或變更的特性，其相反詞為 mutable，本書將其分別譯為「不可變的」與「可變的」。）：卡牌及其位置不能更改，除非你違反封裝原則，直接修改 `_cards` 屬性。在第 13 章，我們會加入 `oneline__setitem__` 來修改它。

特殊方法的用法

關於特殊方法，首先，要注意的是，它們是為了讓 Python 直譯器呼叫的，而不是讓你呼叫的。你不能寫成 `my_object.__len__()`，而是要寫成 `len(my_object)`，而且，如果 `my_object` 是使用者自訂類別實例，Python 會呼叫你寫的 `__len__` 方法。

但是直譯器在處理 `list`、`str`、`bytearray` 等內建型態或 NumPy 陣列等擴展型態時會抄捷徑。在 Python 中，以 C 寫成的變數大小的 collection 有一個稱為 `PyVarObject` 的 struct，[2] 它有一個保存 collection 裡的項目數量的 `ob_size` 欄位。所以，如果 `my_object` 是這種內建型態的實例，`len(my_object)` 會取出 `ob_size` 欄位的值，這比呼叫方法快多了。

特殊方法往往是私下呼叫的。例如，`for i in x:` 其實會讓 Python 呼叫 `iter(x)`，假設它是變數，接下來可能呼叫 `x.__iter__()`，或使用 `x.__getitem__()`，就像在 `FrenchDeck` 範例裡那樣。

一般情況下，你的程式不應該經常直接呼叫特殊方法。除非你要做大量的超編程，否則你應該較常編寫特殊方法，而不是直接呼叫它們。使用者的程式經常呼叫的特殊方法只有 `__init__`，呼叫它是為了在你自己寫的 `__init__` 內呼叫超類別的初始化程式（initializer）。

如果你需要呼叫特殊方法，比較好的做法通常是呼叫相關的內建函式（例如 `len`、`iter`、`str` …等）。這些內建函式會呼叫對應的特殊方法，但通常會提供其他的服務，而且對內建的型態而言，呼叫它們比呼叫方法更快。例子參見第 602 頁，第 17 章的「一起使用 iter 與 Callable」。

在下一節，我們要來看幾個最重要的特殊方法用法：

- 模擬數值型態
- 物件的字串表示法
- 物件的布林值
- 實作 collection

2　C 的 struct 是一種紀錄型態，它裡面有具名欄位。

模擬數值型態

有一些特殊方法可讓使用者的物件對 + 之類的運算子做出回應。我們會在第 16 章詳細討論這個主題，但是在此的目標是透過另一個簡單的例子來進一步說明特殊方法的用法。

我們要寫出一個類別來表示二維的向量，也就是在數學與物理學裡使用的 Euclidean 向量（參見圖 1-1）。

內建的 `complex` 型態可用來代表二維向量，但我們的類別經過擴展可以進一步表示 *n* 維向量。我們會在第 17 章做這件事。

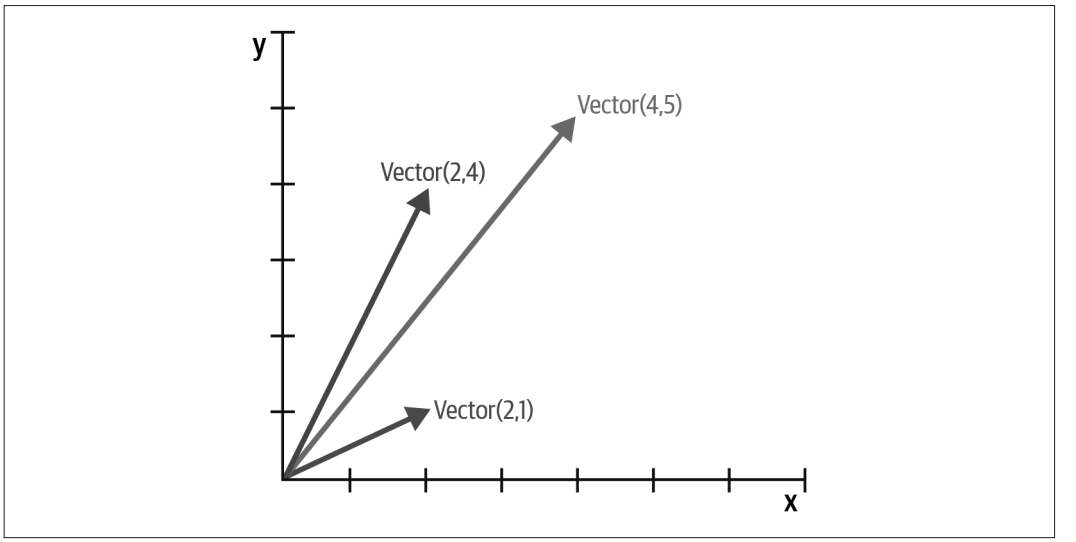

圖 1-1　二維向量加法，Vector(2, 4) + Vector(2, 1) 等於 Vector(4, 5)。

我們先藉著編寫一段主控台對話來設計這個類別的 API，之後可將這段對話當成 doctest 來使用。我們用下面的程式來檢查圖 1-1 的向量加法：

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

注意 + 運算子產生新的 `Vector`，並在主控台以貼心的方式顯示它。

內建的 abs 函式會回傳整數和浮點數的絕對值，以及複數的大小，所以為了保持一致，我們的 API 也使用 abs 來計算向量的大小：

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

我們也可以實作 * 來執行向量乘法（也就是將一個向量乘上一個數字，以產生同一個方向、大小加倍的新向量）：

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

範例 1-2 是實作上述操作的 Vector 類別，它使用特殊方法 __repr__、__abs__、__add__ 與 __mul__。

範例 *1-2*　簡單的二維向量類別

```
"""
vector2d.py: 展示一些特殊方法的極簡類別

我們為了教學而簡化它，所以沒有妥善的錯誤處理程式，
尤其是在 ``__add__`` 與 ``__mul__`` 方法裡。

稍後會大幅擴展這個範例。

Addition::

    >>> v1 = Vector(2, 4)
    >>> v2 = Vector(2, 1)
    >>> v1 + v2
    Vector(4, 5)

Absolute value::

    >>> v = Vector(3, 4)
    >>> abs(v)
    5.0

Scalar multiplication::

    >>> v * 3
    Vector(9, 12)
```

```
    >>> abs(v * 3)
    15.0

"""


import math

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

除了熟悉的 `__init__` 之外，我們寫了五個特殊方法。注意，它們在這個類別裡都沒有被直接呼叫，在 doctest 所敘述的類別用法裡也沒有直接呼叫它們。如前所述，只有 Python 直譯器會經常呼叫大多數的特殊方法。

範例 1-2 實作兩個運算子：`+` 與 `*`，以展示 `__add__` 與 `__mul__` 的基本用法。在這兩種用法中，方法會建立並回傳一個新的 `Vector` 實例，且不會修改它們的運算元 —— `self` 與 `other` 只被讀取。這是中綴（infix）運算子的預期行為，它的目的是建立新物件，不會碰觸它的運算元。我會在第 16 章進一步說明這件事。

如前所述，範例 1-2 允許將 `Vector` 乘上一個數字，但無法將一個數字乘上 `Vector`，這個行為違反純量乘法的交換律。我們會在第 16 章使用特殊方法 `__rmul__` 來修正這個問題。

在接下來的幾節裡，我們要討論 Vector 的其他特殊方法。

字串表示法

__repr__ 特殊方法是為了讓內建的 repr 呼叫的，目的是取得物件的字串表示法，以進行檢查。如果沒有自訂的 __repr__，Python 的主控台會顯示 Vector 實例 <Vector object at 0x10e100070>。

互動式主控台與偵錯器會針對運算式的執行結果呼叫 repr，正如使用 % 運算子的經典格式化語法中的 %r 占位符號，以及在 *f-string* str.format 方法中使用的新格式化字串語法（*https://fpy.li/1-4*）裡的 !r 轉換欄位所做的那樣。

注意，在 __repr__ 裡的 *f-string* 使用 !r 來取得想要顯示的屬性的標準表示法，這是很好的寫法，因為它會顯示 Vector(1, 2) 與 Vector('1', '2') 兩者之間最重要的差異 —— 後者無法在這個範例的背景下運作，因為建構式的引數必須是數字，而不是 str。

__repr__ 回傳的字串必須是明確的，而且應盡可能地符合原始碼，以便重新建立所表示的物件。這就是我們的 Vector 表示法看起來很像在呼叫類別的建構式（例如 Vector(3, 4)）的原因。

相較之下，__str__ 是讓 str() 內建方法呼叫的，而且 print 函式會私下使用它，所以應該回傳一個適合顯示給最終使用者看的字串。

讓 __repr__ 回傳同樣的字串有時很方便。你不需要編寫 __str__，因為繼承 object 類別的實作會將 __repr__ 當成回呼（callback）來呼叫。範例 5-2 是本書中具備自訂的 __str__ 的幾個例子之一。

用過具有 toString 方法的語言的程式設計師傾向實作 __str__ 而非 __repr__。如果你只想在 Python 中實作這些特殊方法之一，那就選擇 __repr__。

Python 鐵粉 Alex Martelli 與 Martijn Pieters 在 Stack Overflow 問題「What is the difference between __str__ and __repr__ in Python?（*https://fpy.li/1-5*」裡提供很棒的意見。

自訂型態的布林值

雖然 Python 有 `bool` 型態，但它接受布林背景下的任何物件，例如用來控制 `if` 或 `while` 陳述式的運算式，或 `and`、`or`、`not` 的運算元。Python 使用 `bool(x)` 來判斷 `x` 值可視為 *true* 還是 *false*，它只回傳 `True` 或 `False`。

在預設情況下，使用者自訂的類別的實例都被視為 true，除非它實作了 `__bool__` 或 `__len__`。基本上，`bool(x)` 會呼叫 `x.__bool__()`，並使用它的結果。如果你沒有實作 `__bool__`，Python 會試著呼叫 `x.__len__()`，如果它回傳零，`bool` 會回傳 `False`，否則 `bool` 會回傳 `True`。

我們的 `__bool__` 實作在概念上很簡單：如果向量的大小是零，那就回傳 `False`，否則回傳 `True`。因為 `__bool__` 應回傳布林值，我們使用 `bool(abs(self))` 來將大小轉換成布林值。在 `__bool__` 方法之外的地方幾乎都不需要明確地呼叫 `bool()`，因為任何物件都可以在布林背景之下使用。

注意，特殊方法 `__bool__` 使你的物件遵守 *The Python Standard Library* 文件中的「Built-in Types」一章（*https://fpy.li/1-6*）所定義的真值測試規則。

在實作 `Vector.__bool__` 時，較快的寫法是：

```
def __bool__(self):
    return bool(self.x or self.y)
```

這種寫法比較難理解，但可避免經歷 `abs`、`__abs__`、平方，與平方根。之所以需要明確地轉換成 `bool`，是因為 `__bool__` 必須回傳布林，而 `or` 會照原樣回傳其中一個運算元，例如 `x or y` 在 `x` 可視為 true 時，計算結果是 `x`，否則是 `y`，無論它是什麼。

Collection API

圖 1-2 是這個語言的基本 collection 型態的介面。圖中的類別都是 ABC，即**抽象基礎類別**。第 13 章會介紹 ABC 與 `collections.abc`。這一節的目的是綜述 Python 最重要的 collection 介面，介紹它們是怎麼用特殊方法來建構的。

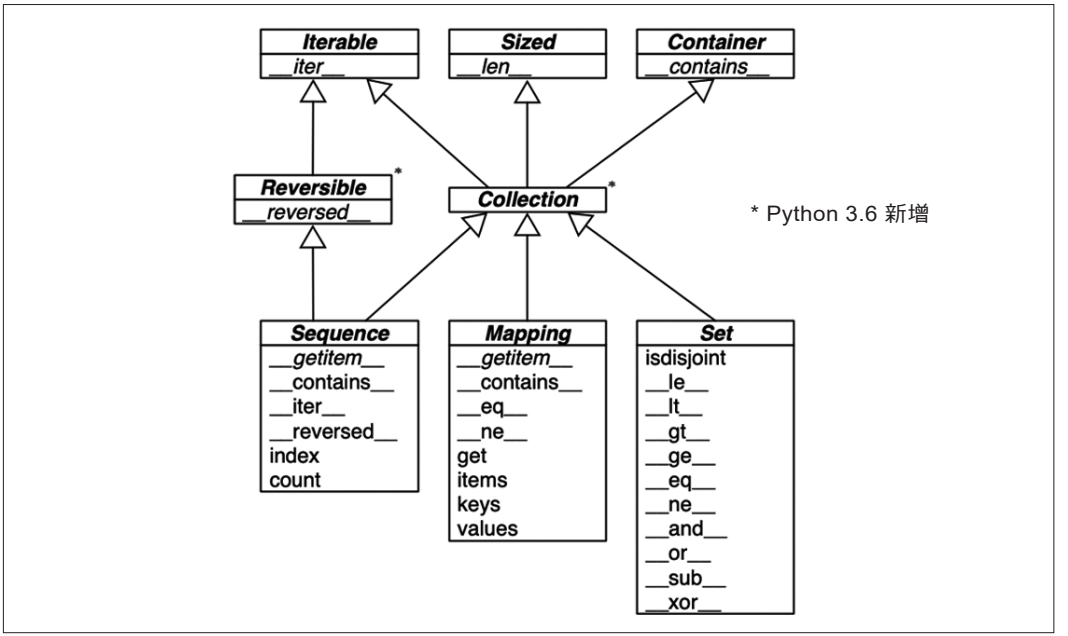

圖 1-2　基本 collection 型態的 UML 類別圖。名稱為斜體的類別是抽象的，所以它必須使用具體子類別來實作，例如 `list` 與 `dict`。其他的方法是具體實作，因此子類別可以繼承它們。

最上面的 ABC 都有一個特殊方法。`Collection` ABC（Python 3.6 新增）統合了每一個 collection 都應該實作的三個基本介面：

- 支援 `for`、unpacking（*https://fpy.li/1-7*）和其他迭代形式的 `Iterable`
- 支援 `len` 內建函式的 `Sized`
- 支援 `in` 運算子的 `Container`

Python 不要求具體類別實際繼承以上的任何一個 ABC。實作了 `__len__` 的任何類別都滿足 `Sized` 介面。

Collection 有三個非常重要的專門類別：

- `Sequence`，將 `list` 與 `str` 等內建型態的介面正式化
- `Mapping`，由 `dict`、`collections.defaultdict` …等實作
- `Set`，`set` 與 `frozenset` 內建型態的介面

只有 Sequence 是 Reversible（可相反的），因為 sequence 允許內容按任意順序，但 mapping 與 set 不允許。

Python 從 3.7 起正式定義 dict 型態是「有序的」，但這只意味著鍵的插入順序會被保留。你不能隨便重新排列 dict 的鍵的順序。

Set ABC 的特殊方法都實作了中綴運算子。例如，a & b 計算集合 a 與 b 的交集，它是在 __and__ 特殊方法裡實作的。

接下來的兩章將詳細介紹標準程式庫的 sequence、mapping 與 set。

接下來要討論 Python Data Model 定義的特殊方法的主要分類。

特殊方法概覽

The Python Language Reference 的「Data Model」一章（*https://fpy.li/dtmodel*）列出八十幾個特殊方法名稱。其中超過一半實作了算術、位元與比較運算子。我們在接下來的幾張表格列出可用的特殊方法。

表 1-1 是特殊方法名稱，不包括用來實作中綴運算子或核心數學函式（例如 abs）的那些。本書將介紹其中的大多數方法，包括最近加入的 __anext__ 等非同步特殊方法（於 Python 3.5 加入），以及類別自訂鉤點（hook），__init_subclass__（於 Python 3.6 加入）。

表 1-1　特殊方法名稱（不含運算子）

種類	方法名稱
字串 / bytes 表示法	__repr__ __str__ __format__ __bytes__ __fspath__
轉換成數字	__bool__ __complex__ __int__ __float__ __hash__ __index__
模擬 collection	__len__ __getitem__ __setitem__ __delitem__ __contains__
迭代	__iter__ __aiter__ __next__ __anext__ __reversed__
callable 或 coroutine 執行	__call__ __await__
環境管理	__enter__ __exit__ __aexit__ __aenter__
建立與銷毀實例	__new__ __init__ __del__
屬性管理	__getattr__ __getattribute__ __setattr__ __delattr__ __dir__
屬性 descriptor	__get__ __set__ __delete__ __set_name__

種類	方法名稱
抽象基礎類別	`__instancecheck__ __subclasscheck__`
類別超編程	`__prepare__ __init_subclass__ __class_getitem__ __mro_entries__`

表 1-2 是用特殊方法來支援的中綴與數值運算子。

其中，最近加入的名稱有 `__matmul__`、`__rmatmul__` 與 `__imatmul__`，它們是在 Python 3.5 加入的，以支援將 `@` 當成矩陣乘法的中綴運算子，第 16 章會詳細介紹。

表 1-2　運算子的特殊方法名稱與符號

運算子種類	符號	方法名稱
一元數值	`- + abs()`	`__neg__ __pos__ __abs__`
豐富比較	`< <= == != > >=`	`__lt__ __le__ __eq__ __ne__ __gt__ __ge__`
算術	`+ - * / // % @ divmod() round() ** pow()`	`__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__`
反向算術	（將運算元對調的算術運算子）	`__radd__ __rsub__ __rmul__ __rtruediv__ __rfloordiv__ __rmod__ __rmatmul__ __rdivmod__ __rpow__`
擴增賦值算術	`+= -= *= /= //= %= @= **=`	`__iadd__ __isub__ __imul__ __itruediv__ __ifloordiv__ __imod__ __imatmul__ __ipow__`
位元	`& | ^ << >> ~`	`__and__ __or__ __xor__ __lshift__ __rshift__ __invert__`
反向位元	（將運算元對調的位元運算子）	`__rand__ __ror__ __rxor__ __rlshift__ __rrshift__`
擴增賦值位元	`&= |= ^= <<= >>=`	`__iand__ __ior__ __ixor__ __ilshift__ __irshift__`

如果 Python 無法對著第一個運算元使用特殊方法，它會對著第二個運算元呼叫對應的反向特殊方法。擴增賦值是結合中綴運算子與變數賦值的一種捷徑，例如 `a += b`。

第 16 章會詳細解釋反向運算子與擴增賦值。

為什麼 len 不是一種方法？

我在 2013 年向核心開發者 Raymond Hettinger 問了這個問題，他的回答大致上引用了「The Zen of Python」（*https://fpy.li/1-8*）的一句話：「practicality beats purity.（實用勝於純粹）」。我在第 9 頁的「特殊方法的用法」中說過，x 是內建型態的實例時，len(x) 跑得非常快。Python 不會幫 CPython 的內建物件呼叫任何方法，它會直接從 C struct 的欄位讀取長度。取得集合內的項目數量是一種常見的操作，所以對於這種基本型態及其他多樣型態（如 str、list、memoryview …等）而言，這項操作必須高效運作。

換句話說，len 不被當成方法來呼叫，而是作為 Python Data Model 的一部分獲得特殊待遇，就像 abs 一樣。但透過特殊方法 __len__，你也可以讓 len 和你自訂的物件合作。在「高效的內建物件」和「語言的一致性」之間，取得合理的妥協。「The Zen of Python」也有這句話：「Special cases aren't special enough to break the rules.（即使是特例，也沒有特殊到可以破壞規則。）」

將 abs 與 len 想成一元運算子，而不是在物件導向語言裡常見的方法呼叫語法，也許會讓你比較容易體諒它們的函式外觀與感覺。事實上，ABC 語言（Python 的直系祖先，是 Python 的許多功能的源頭）有個相當於 len 的 # 運算子（寫法是 #s），將它當成中綴運算子來使用時的寫法是 x#s，此時會計算 x 在 s 裡面的數量，在 Python 中，這要用 s.count(x) 來取得，其中 s 是任意 sequence。

本章摘要

實作特殊方法可讓物件具備類似內建型態的的行為，可寫出富表現力的程式，進而讓社群認為你的程式「很 Python」。

Python 物件的基本要求是提供它自己的字串表示形式：一個用來進行除錯與記錄，另一個用來顯示給最終使用者看。這就是資料模型有特殊方法 __repr__ 與 __str__ 的原因。

模擬 sequence（像 FrenchDeck 範例中的那一個）是最常見的特殊方法用法之一。例如，資料庫程式庫通常將查詢結果包在類 sequence 的 collection 裡回傳。充分利用既有的 sequence 型態是第 2 章的主題。第 12 章會討論如何實作自己的 sequence，我們會在那裡建立 Vector 類別的多維延伸版本。

拜運算子多載之賜，Python 提供大量的數值型態，包括內建的型態，以及 `decimal.Decimal` 與 `fractions.Fraction`，全都支援中綴算術運算子。*NumPy* 資料科學程式庫支援矩陣與張量的中綴運算子。第 16 章會藉著改進 `Vector` 範例來展示如何實作運算子（包括反向運算子與擴增賦值）。

接下來將陸續討論大部分的 Python Data Model 特殊方法的用法與實作。

延伸讀物

The Python Language Reference 的「Data Model」一章（*https://fpy.li/dtmodel*）是本章與本書大多數內容的典範來源。

Alex Martelli、Anna Ravenscroft 與 Steve Holden 合著的 *Python in a Nutshell* 第 3 版（O'Reilly）對於資料模型做了精闢的介紹。他們對於屬性存取的介紹，是除了 CPython 的 C 原始碼之外，我所看過最權威的文獻。Martelli 也是 Stack Overflow 的多產貢獻者，他已經貼出 6,200 多個解答，他在 Stack Overflow 上的個人資訊位於 *https://fpy.li/1-9*。

David Beazley 有兩本書詳細討論在 Python 3 背景下的資料模型：*0Python Essential Reference* 第 4 版（Addison-Wesley），以及和 Brian K. Jones 合著的 *Python Cookbook* 第 3 版。

Gregor Kiczales、Jim desRivieres 與 Daniel G. Bobrow 合著的 *The Art of the Metaobject Protocol*（MIT Press）解釋了 metaobject 協定的概念，Python Data Model 是它的案例之一。

肥皂箱

資料模型還是物件模型？

大多數作者認為 Python 文件所說的「Python Data Model」是「Python object model（物件模型）」。Martelli、Ravenscroft 與 Holden 合著的 *Python in a Nutshell* 第三版，以及 David Beazley 的 *Python Essential Reference* 第四版是介紹 Python Data Model 的最佳著作，但他們將它稱為「object model」。在維基百科上，「object model」的第一個定義（*https://fpy.li/1-10*）是：「The properties of objects in general in a specific computer programming language.（特定計算機程式語言的一般物件屬性）」這也是 Python Data Model 的意義。我在本書裡使用「資料模型」，因為外界文件提到 Python 物件模型時喜歡使用這個詞，在 *The Python Language Reference* 裡，它也是與我們所討論的內容最相關的一章的標題（*https://fpy.li/dtmodel*）。

麻瓜方法

The Original Hacker's Dictionary（*https://fpy.li/1-11*）將 *magic* 定義成「尚未解釋的，或因為太複雜而無法解釋的」或「未被公開的功能，可做到本來不可能做到的事情。」

Ruby 社群將它們的特殊方法稱為魔術方法（*magic methods*）。Pythom 社群也有很多人使用這個名詞。我認為特殊方法並不是奇幻的魔術方法，Python 與 Ruby 都為它們的使用者提供了豐富的 metaobject 協定，那些協定都有完整的文件，讓你我這種麻瓜都能夠模仿語言直譯器的核心開發者可以使用的許多功能。

相較之下，以 Go 為例，這種語言的一些物件有一些魔術功能，但我們不能在自訂的型態裡模擬它們。例如，Go 的 array、string 與 map 可讓你用中括號來存取項目，例如 `a[i]`，但你無法讓你定義的新 collection 型態使用 `[]`。更慘的是，Go 沒有使用者等級的 iterable 介面或 iterable 物件的概念，所以它的 `for/range` 語法只支援五個「魔法」內建型態，包括 array、string 與 map。

或許 Go 的設計者將來會改進它的 metaobject 協定，但目前它的功能比 Python 或 Ruby 所提供還要有限得多。

metaobject

The Art of the Metaobject Protocol（AMOP）是我最喜歡的電腦書籍。我提到它是因為 *metaobject 協定*有助於思考 Python Data Model 及其他語言的類似功能。*metaobject* 是指該語言本身的基本物件。在這個背景下，*協定*是*介面*的同義詞。所以 *metaobject 協定*是物件模型的華麗同義詞，即語言核心構件的 API。

豐富的 metaobject 協定可讓我們擴展語言，以支援新的程式設計範式。*AMOP* 的第一作者 Gregor Kiczales 後來成為剖面導向程式設計（aspect-oriented programming）的先驅，以及 AspectJ（實作該範式的 Java 延伸版本）的初始作者。用 Python 這種動態語言來進行剖面導向程式設計容易得多，也有一些框架可做這種事。最重要的例子是 *zope.interface*（*https://fpy.li/1-12*），它是 Plone 內容管理系統（*https://fpy.li/1-13*）所使用的框架的一部分。