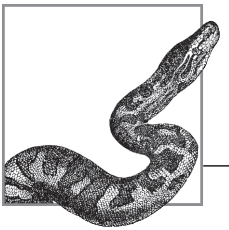


前言

Python 程式語言調和了表面上看似矛盾的許多特徵：優雅而務實、簡單卻強大，它非常高階（high-level），但在你需要擺弄位元和位元組時，並不會礙手礙腳，而且它既適合程式設計新手，對專家來說也很好。

本書的目標讀者是之前有過一些 Python 經驗的程式設計師，以及從其他語言過來，第一次接觸 Python 的程式設計老手。它提供了簡要的參考資訊，幫助讀者快速了解 Python 語言本身、其龐大標準程式庫中最常用的部分，以及一些最流行且實用的第三方模組和套件。Python 生態系統在豐富程度、涉獵範圍和複雜多樣性方面有了很大的進展，以致於百科全書式的單冊書籍不再是合理的希望。儘管如此，本書還是涵蓋了廣泛的應用領域，包括 Web 和網路程式設計、XML 處理、資料庫互動和高速數值計算。它還探討了 Python 的跨平台能力以及擴充 Python 或將其嵌入其他應用程式的基礎知識。

如何運用這本書

雖然你可以從頭開始一直線地閱讀本書，但我們希望它也能成為職業程式設計師的實用參考資料。你可以選擇使用索引來查詢感興趣的項目，或者閱讀特定的章節以了解其涵蓋的特有議題。無論你如何運用它，我們誠摯希望你喜歡閱讀這本書，它代表著我們團隊耗費一整年心血的最佳成果。

本書分為五個部分，包括以下內容。

第一部　開始使用 Python

第 *1* 章，「*Python* 簡介」

介紹 Python 語言的一般特點、它的實作（implementations）、從哪裡獲得幫助和資訊、如何參與 Python 社群，以及如何取得 Python 並在你的電腦上安裝，或是在瀏覽器中執行它。

第 *2* 章，「*Python* 直譯器」

介紹 Python 直譯器程式、它的命令列選項，以及如何用它來執行 Python 程式，或在互動式工作階段（interactive sessions）中使用它。這一章提到了用以編輯 Python 程式的文字編輯器和用來檢查 Python 原始碼的輔助程式，以及一些成熟的整合開發環境（integrated development environments），包括與標準 Python 一起免費提供的 IDLE。本章還包括從命令列執行 Python 程式。

第二部　核心 Python 語言和內建功能

第 *3* 章，「*Python* 語言」

涵蓋 Python 語法、內建資料型別、運算式、述句（statements）、流程控制，以及如何編寫和呼叫函式。

第 *4* 章，「物件導向的 *Python*」

涵蓋 Python 中的物件導向程式設計。

第 *5* 章，「型別注釋」

涵蓋如何將型別資訊添加到你的 Python 程式碼中，以便從現代程式碼編輯器獲得型別提示（type hinting）和自動完成（autocomplete）的幫助，並支援型別檢查器（type checkers）和 linter（程式碼品質檢測工具）的靜態型別檢查。

第 *6* 章，「例外」

涵蓋在錯誤和特殊情況下使用例外的方式、日誌記錄（logging），以及如何編寫程式碼在例外發生時自動進行清理工作。

第 *7* 章，「模組和套件」

涵蓋 Python 如何讓你將程式碼分類為模組和套件，如何定義和匯入模組，以及如何安裝第三方 Python 套件。本章還涵蓋使用虛擬環境（virtual environments）來隔離專案依存關係（project dependencies）的方式。

第 *8* 章，「核心內建功能和標準程式庫模組」

涵蓋內建的資料型別和函式，以及 Python 標準程式庫中一些最基本的模組（粗略地說，這個模組集合所提供的功能，在其他一些語言中是語言本身所內建的）。

第 *9* 章，「字串（*Strings*）與相關功能」

涵蓋 Python 的字串處理機能，包括 Unicode 字串、位元組字串（bytestrings）和字串字面值（string literals）。

第 *10* 章，「正規表達式」

涵蓋 Python 對正規表達式（regular expressions）的支援。

第三部　Python 程式庫和擴充模組（Extension Modules）

第 *11* 章，「檔案和文字運算」

涵蓋使用 Python 標準程式庫中的許多模組來處理檔案和文字，以及針對富文字（rich text）I/O 的特定平台擴充功能。本章還包括有關國際化（internationalization）和本地化（localization）的議題。

第 *12* 章，「續存和資料庫」

涵蓋 Python 的序列化（serialization）和續存（persistence）機制，以及它與 DBM 資料庫和關聯式（基於 SQL 的）資料庫的介面，特別是 Python 標準程式庫中便利的 SQLite。

第 *13* 章，「時間運算」

涵蓋 Python 中處理時間和日期的方式，使用標準程式庫和第三方擴充功能。

第 *14* 章，「自訂執行」

涵蓋在 Python 中實現進階執行控制（execution control）的方式，包括執行動態產生的程式碼和垃圾回收（garbage collection）的控制。本章還包括 Python 的一些內部型別，以及註冊「清理（cleanup）」函式以便在程式終止時執行的特殊議題。

第 *15* 章，「共時性：執行緒和行程」

涵蓋 Python 的共時執行（concurrent execution）功能，包括在一個行程中執行的多個執行緒（threads），以及在單一機器上執行的多個行程（processes）[1]。

第 *16* 章，「數值處理」

涵蓋 Python 在標準程式庫模組和第三方擴充套件中的數值計算功能；特別是如何使用十進位小數（decimal numbers）或分數（fractions）來代替預設的二進位浮點數（binary floating-point numbers）。本章還包括如何獲得並使用偽隨機（pseudorandom）數和真正的隨機數（random numbers），以及如何快速處理整個陣列（和矩陣）的數字。

第 *17* 章，「測試、除錯和最佳化」

本章所講述的工具和做法可以幫助你確保程式是正確的（也就是，它們做了應該做的事）、發現並修復程式中的錯誤，以及檢查並提高程式的效能。本章還包括警告（warnings）的概念和處理它們用的 Python 程式庫模組。

第四部　網路和 Web 程式設計

第 *18* 章，「網路基礎知識」

涵蓋使用 Python 進行網路通訊的基礎知識。

第 *19* 章，「客戶端網路協定模組」

涵蓋 Python 標準程式庫中用以編寫網路客戶端（network client）程式的模組，特別是用於處理客戶端的各種網路協定、傳送和接收電子郵件，以及 URL 處理。

1　第三版中關於非同步程式設計（asynchronous programming）的單獨章節在此版中被取消了，改在第 15 章的參考資訊中對這一日益增長的主題進行更徹底的講解。

第 *20* 章，「提供 *HTTP* 服務」

涵蓋如何在 Python 中為 Web 應用程式提供 HTTP 服務，透過流行的第三方輕量化 Python 框架，運用 Python 的 WSGI 標準介面來與 Web 伺服器互動。

第 *21* 章，「*Email*、*MIME* 和其他網路編碼」

涵蓋如何使用 Python 處理電子郵件訊息和其他網路結構化和編碼的文件（documents）。

第 *22* 章，「結構化文字：*HTML*」

涵蓋熱門的第三方 Python 擴充模組，用以處理、修改和產生 HTML 文件。

第 *23* 章，「結構化文字：*XML*」

涵蓋 Python 程式庫模組和流行的擴充功能，用以處理、修改和產生 XML 文件。

第五部　擴充、發佈（Distributing），以及版本升級和遷移（Migration）

第 24 章和第 25 章以摘要形式收錄在本書的印刷版本中，你可以在 *https://github.com/pynutshell/pynut4* 找到這些章節的全部內容。

第 *24* 章，「封裝程式和擴充功能」

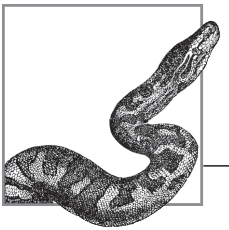
介紹封裝和分享 Python 模組與應用程式的工具和模組。

第 *25* 章，「擴充和內嵌標準型的 *Python*」

介紹如何使用 Python 的 C API、Cython 和其他工具來編寫 Python 擴充模組。

第 *26* 章，「*v3.7* 到 *v3.n* 的遷移」

涵蓋為 Python 使用者規劃及部署版本升級的相關主題和最佳實務做法，這些使用者包括個人、程式庫維護者、企業範圍的部署與支援人員。



5
型別注釋

用型別資訊來注釋（annotating）你的 Python 程式碼是一個選擇性的步驟，在開發和維護一個大型專案或一個程式庫的過程中可能會非常有幫助。靜態型別檢查器（static type checkers）和 lint 工具有助於識別和定位函式引數和回傳值中的資料型別不匹配。IDE 可以使用這些**型別注釋**（*type annotations*，也稱為**型別提示**，*type hints*）來改善自動完成，並提供彈出式說明文件。第三方套件和框架可以使用型別注釋來量身打造執行時期的行為，或者根據方法和變數的型別注釋來自動生成程式碼。

Python 中的型別注釋和檢查仍在持續發展中，並且觸及許多複雜的議題。本章涵蓋型別注釋的一些最常見的用例；你可以在本章結尾列出的資源中找到更全面的學習素材。

型別注釋支援因 *Python* 版本而異

Python 支援型別注釋的功能隨著版本的推進而不斷發展，其中有一些重要的添補和刪除。本章的其餘部分將描述 Python 最新版本（3.10 及之後的版本）中的型別注釋支援，並以註記的方式指出在其他版本中可能存在或不存在的功能。

歷史

Python 本質上是一種**動態定型**（*dynamically typed*）的語言。這讓你得以透過命名和使用變數來快速開發程式碼，而不需要宣告（declare）它

們。動態定型允許靈活的編程慣用語、泛用容器（generic containers）和多型（polymorphic）的資料處理，而不需要明確定義介面型別或類別階層架構。缺點是，在開發過程中，這種語言不會提供任何幫助來標示被傳入到函式或從函式回傳的型別不相容的變數。不像一些語言利用開發時的編譯步驟來檢測和回報資料型別問題，Python 仰賴開發人員維護全面的單元測試，特別是（雖然遠非唯一！[1]）透過在一系列測試案例中重現執行環境來發現資料型別錯誤。

型別注釋不強制施加

型別注釋在執行時期不會強制施加。Python 不會基於它們執行任何型別驗證或資料轉換；可執行的 Python 程式碼仍然要負責正確使用變數和函式引數。然而，型別注釋必須在語法上是正確的。一個後期匯入或動態匯入的模組若包含無效的型別注釋，就會在你執行的 Python 程式中提出一個 `SyntaxError` 例外，就像任何無效的 Python 述句一樣。

從歷史上看，沒有任何型別檢查常常被看作是 Python 的一個缺點，一些程式設計師把這當成選擇其他程式語言的理由。然而，社群希望 Python 保持其執行時期的型別自由，所以合乎邏輯的辦法是增加對靜態型別檢查的支援，這種檢查會在開發時期由類似 lint 的工具（會在下一節進一步描述）和 IDE 進行。有些人在基於剖析函式特徵式或說明文件字串的型別檢查方面進行了一些嘗試。Guido van Rossum 在 Python Developers 郵件列表（*https://oreil.ly/GFMBC*）中引用了幾個案例，表明型別注釋可能是有幫助的，例如在維護大型舊有源碼庫之時。透過注釋語法，開發工具可以進行靜態型別檢查，以強調與預期型別相衝突的變數和函式用法。

型別注釋的第一個正式版本使用特殊格式的註解（comments）來表示變數型別和回傳碼，如 PEP 484（*https://oreil.ly/61GSZ*）中所定義的，這是 Python 3.5 的一個臨時提案 PEP[2]。使用註解可以快速實作和試驗新的型別語法，而不需要修改 Python 編譯器本身[3]。第三方套件 `mypy`（*http://mypy-lang.org*）使用這些註解進行靜態型別檢查，獲得了廣泛

1 強大而廣泛的單元測試也將防止許多業務邏輯問題，那些問題是任何型別檢查都無法捕捉到的。因此，型別提示不是用來**取代**單元測試的，而是用來**補充**它們。

2 型別注釋的**語法**（*syntax*）在 Python 3.0 中就被引入，但後來才對其**語意**（*semantics*）做出規定。

3 這種做法也與當時仍在廣泛使用的 Python 2.7 程式碼相容。

的認可。隨著 Python 3.6 中 PEP 526（*https://oreil.ly/S8kI3*）被採納，型別注釋完全整合到 Python 語言本身，並在標準程式庫中加入了支援的 `typing` 模組。

型別檢查工具

隨著型別注釋成為 Python 既定的一個部分，型別檢查工具（type-checking utilities）和 IDE 外掛（plug-ins）也成為 Python 生態系統的一部分。

mypy

獨立的 `mypy`（*https://oreil.ly/6fMPM*）工具仍然是靜態型別檢查主要支柱，它總是與不斷發展的 Python 型別注釋形式保持同步（頂多差一個 Python 版本！）。`mypy` 也可以作為包括 Vim、Emacs 和 SublimeText 在內的編輯器以及 Atom、PyCharm 和 VS Code 等 IDE 的外掛使用（PyCharm、VS Code 和 Wing IDE 也在 `mypy` 之外加入了自己的型別檢查功能）。執行 `mypy` 最常用的命令單純是 **`mypy my_python_script.py`**。

你可以在 `mypy` 線上說明文件（*https://oreil.ly/rQPK0*）中找到更詳細的用法範例和命令列選項，以及一份速查表（cheat sheet，*https://oreil.ly/CT6FE*）作為方便的參考資訊。本節後面的程式碼範例將包括 `mypy` 錯誤訊息的例子，以說明使用型別檢查可以捕捉到的 Python 錯誤種類。

其他的型別檢查器

可以考慮使用的其他型別檢查器包括：

MonkeyType

> Instagram 的 MonkeyType（*https://oreil.ly/RHqNo*）使用 `sys.setprofile` 掛接器（hook）在執行時期動態偵測型別；就跟 `pytype`（見下文）一樣，它也可以生成一個 *.pyi*（殘根）檔案，代替或補充在 Python 程式碼檔案本身中插入的型別注釋。

`pydantic`

> `pydantic`（*https://oreil.ly/-zNQj*）也在執行時期工作，但是它不會產生殘根（stubs）或插入型別注釋；取而代之，它的主要目標是剖析輸入並確保 Python 程式碼收到乾淨的資料。正如線上說明文件（*https://oreil.ly/0Ucvm*）中所述，它還允許你為自己的環境擴充其驗證功能。請參閱第 696 頁的「FastAPI」，以了解一個簡單的例子。

Pylance

> Pylance（*https://oreil.ly/uB5XN*）是一個型別檢查模組，主要是為了將 Pyright（見下文）內嵌到 VS Code 中。

Pyre

> Facebook 的 Pyre（*https://oreil.ly/HJ-qQ*）也可以生成 *.pyi* 檔案。它目前不能在 Windows 上執行，除非你安裝了 Windows Subsystem for Linux（WSL，*https://oreil.ly/DwB82*）。

Pyright

> Pyright（*https://oreil.ly/wwuA8*）是 Microsoft 的靜態型別檢查工具，可作為一個命令列工具或 VS Code 的擴充功能。

`pytype`

> 來自 Google 的 `pytype`（*https://oreil.ly/QuhCB*）是一個靜態型別檢查器，除了型別注釋之外，它還專注於**型別推論**（*type inferencing*，甚至在沒有型別提示的情況下也能提供建議）。型別推論提供了強大的能力來檢測型別錯誤，即使在沒有注釋的程式碼中也是如此。`pytype` 也可以生成 *.pyi* 檔案，並把殘根檔（stub files）合併到 *.py* 原始碼中（最新版的 `mypy` 在這方面也跟進了）。目前，除非你先安裝 WSL（*https://oreil.ly/7G_j-*），否則 `pytype` 無法在 Windows 上執行。

來自多個主要軟體組織的型別檢查應用程式的湧現，證明了 Python 開發人員社群對使用型別注釋的廣泛興趣。

型別注釋語法

在 Python 中，一個**型別注釋**（*type annotation*）是用下列形式指定的：

```
identifier: type_specification
```

type_specification 可以是任何 Python 運算式，但通常涉及一或多個內建型別（舉例來說，僅僅只提到一個 Python 型別，就可以算是一個完全有效的運算式了）、或從 `typing` 模組匯入的屬性（會在下一節討論）。典型的形式為：

```
type_specifier[type_parameter, ...]
```

下面是用作變數型別注釋的型別運算式的一些例子：

```
import typing

# 一個 int
count: int

# 一個串列的 int，帶有一個預設值
counts: list[int] = []

# 一個有 str 鍵值的 dict，其值是包含 2 個 int 和一個 str 的元組
employee_data: dict[str, tuple[int, int, str]]

# 一個可呼叫物件，接受一個 str 或 byte 引數，並回傳一個 bool 值
str_predicate_function: typing.Callable[[str | bytes], bool]

# 一個帶有 str 鍵值的 dict，其值是接受和回傳
# 一個 int 的函式
str_function_map: dict[str, typing.Callable[[int], int]] = {
    'square': lambda x: x * x,
    'cube': lambda x: x * x * x,
}
```

請注意，**`lambda`** 並不接受型別注釋。

Python 3.9 和 3.10 中的型別語法變化

在本書所講述的 Python 版本中，型別注釋最重要的變化之一是在 Python 3.9 中新增了使用內建 Python 型別的支援，如這些例子中所示。

-3.9 在 Python 3.9 之前，這些注釋需要使用從 `typing` 模組匯入的型別名稱，如 `Dict`、`List`、`Tuple` 等。

3.10+ Python 3.10 新增了使用 | 表示替代型別（alternative types）的支援，以作為 Union[*atype*,*btype*, ...] 記號更可讀、更簡潔的替代方式。| 運算子也能用來以 *atype* | None 替換 Optional[*atype*]。

舉例來說，前面的 str_predicate_function 定義會接受以下形式之一，取決於你的 Python 版本：

```
# 在 3.10 之前，指定替代型別
# 需要使用 Union 型別
from typing import Callable, Union
str_predicate_function: Callable[Union[str, bytes], bool]

# 在 3.9 之前，諸如 list、tuple、dict、set 等
# 內建型別需要從 typing 模組
# 匯入型別
from typing import Dict, Tuple, Callable, Union
employee_data: Dict[str, Tuple[int, int, str]]
str_predicate_function: Callable[Union[str, bytes], bool]
```

要用回傳型別來注釋一個函式，請使用以下形式：

```
def identifier(argument, ...) -> type_specification :
```

其中每個 *argument* 都接受這種形式：

```
identifier[: type_specification[ = default_value]]
```

這裡有經過注釋的一個函式的例子：

```
def pad(a: list[str], min_len: int = 1, padstr: str = ' ') ->
list[str]:
    """ 給定一個字串串列和一個最小長度，
       回傳一個用「padding」字串延伸的串列複本，
       使其至少達到那個最小長度。
    """
    return a + ([padstr] * (min_len - len(a)))
```

注意，如果一個經過注釋的參數有一個預設值，PEP 8 建議在等號周圍使用空格。

尚未完全定義的前向參考型別（*Forward-Referencing Types*）

有時，一個函式或變數定義需要參考一個尚未定義的型別。這在類別方法或必須定義當前類別的引數或回傳值的方法中很常見。這些函式的特徵式是在編譯時期被剖析的，而那時型別還沒有被定義。舉例來說，這個 `classmethod` 會編譯失敗：

```
class A:
    @classmethod
    def factory_method(cls) -> A:
        # ... 方法主體在此 ...
```

由於 Python 在編譯 `factory_method` 時還沒有定義類別 `A`，所以程式碼會提出 `NameError`。

將回傳型別 `A` 用引號（quotes）圍起來可以解決這個問題：

```
class A:
    @classmethod
    def factory_method(cls) -> 'A':
        # ... 方法主體在此 ...
```

未來的 Python 版本可能會將型別注釋的估算推遲到執行時期，從而沒必要使用外圍的引號（Python 的 Steering Committee 正在評估各種可能性）。你可以使用 `from __future__ import annotations` 來預覽這種行為。

typing 模組

`typing` 模組支援型別提示。它包含在建立型別注釋時有用的定義，包括：

- 用於定義型別的類別和函式
- 用於修改型別運算式的類別和函式
- 抽象基礎類別（ABC）
- 協定（Protocols）
- 工具和裝飾器
- 用於定義自訂型別的類別

型別

typing 模組的最初實作包括與 Python 內建容器和其他型別、以及來自標準程式庫模組的型別相對應的型別定義。這些型別中有許多後來都被棄用了（見下文），但是有些仍然有用，因為它們並不直接對應於任何 Python 內建型別。表 5-1 列出在 Python 3.9 及之後版本中仍然有用的型別。

表 5-1　typing 模組中的實用定義

型別	說明
`Any`	匹配任何型別。
`AnyStr`	相當於 `str | bytes`。`AnyStr` 是用來注釋函式引數和回傳型別的，其中任一種字串型別都是可以接受的，但是在多個引數之間，或在引數和回傳型別之間都不應該混合使用。
`BinaryIO`	匹配具有二進位（`bytes`）內容的串流（streams），如那些從 `mode='b'` 的 `open`，或 `io.BytesIO` 回傳的串流。
`Callable`	`Callable[[`*`argument_type`*`, ...], `*`return_type`*`]` 為一個可呼叫物件定義型別特徵式。接受與可呼叫物件的引數相對應的型別構成的一個串列，以及函式回傳值的型別。如果可呼叫物件沒有引數，就用一個空串列 `[]` 表示。如果可呼叫物件沒有回傳值，則使用 **`None`** 作為 *`return_type`*。
`IO`	等同於 `BinaryIO | TextIO`。
`Literal` `[`*`expression`*`,...]`	**3.8+** 指定一個變數可以接受的有效值所成的一個串列。
`LiteralString`	**3.11+** 指定必須實作為字面引號值（literal quoted value）的一個 `str`。用於防止程式碼遭受注入（injection）攻擊。
`NoReturn`	用作「永遠執行」的函式之回傳型別，例如那些不會回傳的 `http.serve_forever` 或 `event_loop.run_forever` 呼叫。這不是為那些單純回傳而沒有明確值的函式所準備的；對於那些函式，請使用 `-> None`。關於回傳型別的更多討論，可以在第 226 頁的「為現有程式碼新增型別注釋（逐步定型）」中找到。
`Self`	**3.11+** 作為 **`return`** `self` 的實體函式之回傳型別（以及其他一些情況，如 PEP 673（*https://oreil.ly/NMMaw*）中的例子所示）。
`TextIO`	匹配有文字（`str`）內容的串列，比如那些從 `mode='t'` 的 `open`，或 `io.StringIO` 回傳的串流。

-3.9 在 3.9 之前，typing 模組中的定義被用來建立代表內建型別的型別，例如 List[int] 代表 int 的一個串列（list）。從 3.9 開始，那些名稱被棄用了，因為它們相應的內建或標準程式庫型別現在都支援 [] 語法了：一個 int 的串列現在可以單純使用 list[int] 來定型。表 5-2 列出在 Python 3.9 之前，對使用內建型別的型別注釋來說是必要的 typing 模組定義。

表 5-2　Python 內建型別和它們在 typing 模組中 3.9 之前的定義

內建型別	3.9 之前的 typing 模組等價物
dict	Dict
frozenset	FrozenSet
list	List
set	Set
str	Text
tuple	Tuple
type	Type
collections.ChainMap	ChainMap
collections.Counter	Counter
collections.defaultdict	DefaultDict
collections.deque	Deque
collections.OrderedDict	OrderedDict
re.Match	Match
re.Pattern	Pattern

型別運算式參數

在 typing 模組中定義的一些型別會修飾其他的型別運算式。表 5-3 中列出的型別會為 *type_expression* 中的經過修飾的型別提供額外的型別資訊或限制。

表 5-3　型別運算式參數

參數	用法和說明
Annotated	Annotated[*type_expression*, *expression*, ...] 3.9+ 用額外的詮釋資料（metadata）擴充了 *type_expression*。函式 *fn* 額外的詮釋資料值可以在執行時期使用 get_type_hints(*fn*, include_extras=**True**) 取得。
ClassVar	ClassVar[*type_expression*] 表示該變數是一個類別變數（class variable），不應該被指定為實體變數。
Final	Final[*type_expression*] 3.8+ 表示該變數不應該在子類別中被寫入或覆寫。
Optional	Optional[*type_expression*] 等同於 *type_expression* | **None**。通常用於預設值為 **None** 的具名引數（Optional 不會自動將 **None** 定義為預設值，所以你仍然必須在函式特徵式中用 =**None** 跟隨它）。3.10+ 隨著指定替代型別屬性的 | 運算子的出現，人們越來越傾向於選用 *type_expression* | **None**，而非使用 Optional[*type_expression*]。

抽象基礎類別

就像內建型別一樣，最初的 typing 模組實作包括與 collections.abc 模組中的抽象基礎類別（abstract base classes）相對應的型別定義。這些型別中有許多後來都被棄用了（見下文），但是有兩個定義仍然作為 ABC 的別名（aliases）被保留在 collections.abc 中（參閱表 5-4）。

表 5-4　抽象基礎類別的別名

型別	子類別必須實作的方法
Hashable	__hash__
Sized	__len__

-3.9 在 Python 3.9 之前，在 typing 模組中的下列定義代表了在 collections.abc 模組中定義的抽象基礎類別，例如 Sequence[int] 表示 int 的一個序列（sequence）。從 3.9 開始，typing 模組中的這些名稱被棄用了，因為它們在 collections.abc 中的相應型別現在支援 [] 語法了：

```
AbstractSet           Container        Mapping
AsyncContextManager   ContextManager   MappingView
AsyncGenerator        Coroutine        MutableMapping
AsyncIterable         Generator        MutableSequence
```

```
AsyncIterator       ItemsView       MutableSet
Awaitable           Iterable        Reversible
ByteString          Iterator        Sequence
Collection          KeysView        ValuesView
```

協定

typing 模組定義了幾個協定（*protocols*），這類似於其他一些語言所說的「介面（interfaces）」。協定是抽象基礎類別，主要用途是簡潔地表達對一個型別的限制，確保它包含特定的某些方法。目前在 typing 模組中定義的每個協定都與一個特殊方法有關，其名稱以 Supports 開頭，後面接著方法的名稱（然而，其他程式庫，如定義在 typeshed（*https://oreil.ly/adB9Z*）中的那些，不需要遵循同樣的約束）。協定可被用來當作判斷一個類別對於該協定能力支援情況的一個最簡抽象類別：類別想要遵從一個協定，需要做的就只是實作該協定的特殊方法。

表 5-5 列出 typing 模組中定義的協定。

表 5-5　typing 模組中的協定及其必要方法

協定	擁有方法
SupportsAbs	__abs__
SupportsBytes	__bytes__
SupportsComplex	__complex__
SupportsFloat	__float__
SupportsIndex 3.8+	__index__
SupportsInt	__int__
SupportsRound	__round__

一個類別若希望滿足 issubclass(*cls*, *protocol_type*)，或者讓它的實體滿足 isinstance(*obj*, *protocol_type*)，並不需要明確地繼承一個協定。該類別只需要實作協定中定義的方法就行了。舉例來說，想像一下，實作羅馬數字（Roman numerals）的一個類別：

```
class RomanNumeral:
    """ 代表羅馬數字和它們
       int 值的類別
    """
    int_values = {'I': 1, 'II': 2, 'III': 3, 'IV': 4, 'V': 5}
```

```
    def __init__(self, label: str):
        self.label = label

    def __int__(self) -> int:
        return RomanNumeral.int_values[self.label]
```

要建立這個類別的一個實體（例如為了表示一部電影的續集）並獲得其值，你可以使用下列程式碼：

```
>>> movie_sequel = RomanNumeral('II')
>>> print(int(movie_sequel))
2
```

RomanNumeral 滿足 issubclass 以及 isinstance 與 SupportsInt 的檢查，因為它實作了 __int__，儘管它沒有明確繼承自協定類別 SupportsInt[4]：

```
>>> issubclass(RomanNumeral, typing.SupportsInt)
True
>>> isinstance(movie_sequel, typing.SupportsInt)
True
```

工具與裝飾器

表 5-6 列出在 typing 模組中定義的常用函式和裝飾器；後面接著幾個例子。

表 5-6　在 typing 模組中定義的常用函式和裝飾器

函式 / 裝飾器	用法與說明
cast	cast(*type*, *var*) 向靜態型別檢查器發出訊號，指出 *var* 應該被視為型別 *type*。回傳 *var*；在執行時期，*var* 不會有任何變化、轉換或驗證。參閱表後的例子。
final	@final 3.8+ 用來在類別定義中裝飾一個方法，以在該方法於子類別中被覆寫時發出警告。也可以作為一個類別裝飾器，在類別本身被子類別化時發出警告。
get_args	get_args(*custom_type*) 回傳用於建構一個自訂型別（custom type）的引數。
get_origin	get_origin(*custom_type*) 3.8+ 回傳用於建構自訂型別的基礎型別（base type）。

4　而 SupportsInt 使用 runtime_checkable 裝飾器。

函式 / 裝飾器	用法與說明
`get_type_hints`	`get_type_hints(`*`obj`*`)` 回傳結果，就像存取 *`obj`*`.__annotations__` 一樣。能以選擇性的 `globalns` 和 `localns` 命名空間引數來解析以字串形式給定的前向型別參考（forward type references），或用選擇性的 Boolean `include_extras` 引數來包含任何使用 `Annotations` 新增的非定型注釋（nontyping annotations）。
`NewType`	`NewType(`*`type_name`*`, `*`type`*`)` 定義一個從 *`type`* 衍生出來的自訂型別。*`type_name`* 是一個字串，應該與 `NewType` 被指定的區域變數相匹配。對於區分常見型別的不同用途很有用，舉例來說，用於雇員姓名的 `str` 和用於部門名稱的 `str`。關於這個函式的更多資訊，請參閱第 223 頁的「NewType」。
`no_type_check`	`@no_type_check` 用來表示注釋不打算作為型別資訊使用。可應用於類別或函式。
`no_type_check_decorator`	`@no_type_check_decorator` 用來為另一個裝飾器新增 `no_type_check` 行為。
`overload`	`@overload` 用來允許定義出具有相同名稱、但在特徵式中具有不同型別的多個方法。參閱表後的例子。
`runtime_checkable`	`@runtime_checkable` **3.8+** 用來為自訂協定類別新增 `isinstance` 和 `issubclass` 支援。關於這個裝飾器的更多資訊，請參閱第 224 頁的「執行時期使用型別注釋」。
`TypeAlias`	*`name`*`: TypeAlias = `*`type_expression`* **3.10+** 用來區分型別別名（type alias）的定義和簡單的指定（simple assignment）。在 *`type_expression`* 是一個簡單的類別名稱，或指向尚未定義的類別的一個字串值的情況下最有用，那看起來可能像是一個指定。`TypeAlias` 只能在模組範疇（module scope）使用。一種常見的用法是讓我們更容易一致地重複使用一個冗長的型別運算式，例如：`Number: TypeAlias = int | float | Fraction`。關於這個注釋的更多資訊，請參閱第 222 頁的「TypeAlias」。
`type_check_only`	`@type_check_only` 用於表示該類別或函式僅在型別檢查時使用，在執行時期無法取用。
`TYPE_CHECKING`	一個特殊的常數，靜態型別檢查器估算為 **`True`**，但在執行時期被設定為 **`False`**。使用這個常數來跳過匯入緩慢、而且單純只用來支援型別檢查的大型模組之匯入（這樣在執行時期就不需要匯入）。

函式 / 裝飾器	用法與說明
TypeVar	TypeVar(*type_name*, **types*) 定義一個型別運算式元素，用於使用 Generic 的複雜泛用型別（generic types）。*type_name* 是一個字串，應該與 TypeVar 被指定的區域變數相匹配。如果沒有給出 *types*，那麼所關聯的 Generic 將接受任何型別。如果有給定 *types*，那麼 Generic 將只接受所提供的任何型別或其子類別的實體。也接受具名的 Boolean 引數 covariant 和 contravariant（兩者都預設為 False），以及引數 bound。這些在第 216 頁的「泛型和 TypeVar」以及 typing 模組的說明文件（*https://oreil.ly/069u4*）中會有更詳細的描述。

在型別檢查時期使用 overload 來標示必須以特定組合使用的具名引數。在這種情況下，必須用 str 鍵值和 int 值的一個對組來呼叫 fn，或者用單一的 bool 值：

```
@typing.overload
def fn(*, key: str, value: int):
    ...

@typing.overload
def fn(*, strict: bool):
    ...

def fn(**kwargs):
    # 實作放在這裡，包括不同具名引數
    # 的處理工作
    pass

# 有效呼叫
fn(key='abc', value=100)
fn(strict=True)

# 無效呼叫
fn(1)
fn('abc')
fn('abc', 100)
fn(key='abc')
fn(True)
fn(strict=True, value=100)
```

注意，overload 裝飾器純粹是用於靜態型別檢查。要想在執行時期根據參數型別實際分派（dispatch）到不同的方法，請使用 functools.singledispatch。

使用 cast 函式迫使型別檢查器在 cast 的範疇內將一個變數視為某個特定型別：

```
def func(x: list[int] | list[str]):
    try:
        return sum(x)
    except TypeError:
        x = cast(list[str], x)
        return ','.join(x)
```

謹慎使用 *cast*

cast（強制轉型）是覆寫任何推論或先前注釋的一種方式，這些推論或注釋可能存在於你程式碼中的特定位置。這可能會隱藏你程式碼中的實際型別錯誤，使型別檢查的過程不完整或不準確。前面例子中的 func 本身沒有引起任何 mypy 警告，但是如果被傳入混合了 int 和 str 的一個串列，則會在執行時期失敗。

定義自訂型別

正如 Python 的 **class** 語法允許創建新的執行時期型別和行為一樣，本節中討論的 typing 模組構造可以為高階型別檢查建立專門的型別運算式。

typing 模組包括三個類別，你的類別可以繼承這些類別，以獲得型別定義和其他預設功能，這些類別列於表 5-7。

表 5-7　用於定義自訂型別的基礎類別

Generic	Generic[*type_var*, ...] 定義出一個用於型別檢查的抽象基礎類別（type-checking abstract base class），適用於其方法會參考一或多個由 TypeVar 所定義的型別的那些類別。泛型在後面的章節中會有更詳細的描述。
NamedTuple	NamedTuple collections.namedtuple 的一個具型實作（typed implementation）。更多的細節和例子見第 218 頁的「NamedTuple」。
TypedDict	TypedDict 3.8+ 定義了一個型別檢查用的 dict，它有特定的鍵值和每個鍵值的值型別。細節請參閱第 219 頁的「TypedDict」。