前言

這是一本討論 Python 軟體工程準則的書籍。

坊間有許多軟體工程書籍，也有許多資源提供 Python 相關資訊，但是你必須經過一番努力才能結合兩者，這也是本書試圖彌合的鴻溝。

用一本書來討論所有的軟體工程主題是不切實際的，因為這個領域太大了，而且有些書藉即使用盡篇幅也只不過討論了其中的某些主題而已。本書的重點是軟體工程的主要做法或準則，希望協助你編寫更容易維護的程式，同時告訴你如何利用 Python 的功能來編寫它們。

給你一個肺腑之言：軟體問題的解決方案不可能只有一個，通常你需要做出權衡取捨。每一種解決方案都有其優缺點，當你選擇它們時必須遵守一些準則，並坦然接受獲得好處應付出的代價。你通常沒有單一最佳解，但必須遵守一些規則，只要你遵守它們，就可以走在比較安全的道路上。這也是本書的宗旨：促使讀者遵循規則來做出最好的選擇，因為就算遇到困難，當你遵守良好的做法時，結果也會好很多。

講到良好的做法，雖然接下來的內容有些遵守既定且行之有效的準則，但有些比較武斷，不過這不代表你只能採取那種特定的做法。作者不認為自己是簡潔程式的權威，因為這種稱謂不切實際。我們鼓勵讀者做批判性的思考：選擇最適合專案的做法，並且自由地提出不同的意見。只要分歧的意見可以帶來啟發人心的辯論，我們就歡迎它。

我寫這本書的用意是分享 Python 帶來的樂趣，以及我從過往經驗中學到的典型表達風格，希望提升讀者在這種語言上的專業知識。

本書使用範例程式來說明主題。這些範例使用著作本書時的最新版 Python，即 Python 3.7，不過將來的版本應該也是相容的。書中的程式不需要綁定特定的平台，因此你只要使用 Python 解譯器就可以在任何作業系統上測試範例程式了。

多數範例的目標都是盡量維持程式的簡潔，所以我只用標準程式庫和一般的 Python 來編寫主程式與它的測試程式。有些章節需要使用額外的程式庫，我會在這些範例的 `requirements.txt` 檔案提供說明，以協助你執行它們。

本書將說明可讓程式更好、更容易閱讀且更容易維護的所有 Python 功能。我們不但會討論這種語言的功能，也會分析如何在 Python 中運用軟體工程技巧。讀者將會發現有些軟體工程技巧與 Python 的參考實作有所差異，有些準則或模式稍有不同，有些甚至根本沒辦法在 Python 中使用。瞭解這些情況意味著你已經更認識 Python 了。

本書對象

本書適合想要瞭解軟體設計或更深入瞭解 Python 的軟體工程從業者。我們假設讀者已經熟悉物件導向軟體設計的規則，而且已經寫過一些程式了。

就 Python 而言，本書適合各種等級的讀者。本書很適合用來學習 Python，因為它的複雜程度是循序漸進的。第一章會討論 Python 的基本知識，很適合用來學習這個語言主要的表達風格、函式，以及工具程式。本書不僅僅會用 Python 來解決問題，也會採取典型的寫法。

經驗豐富的程式員也能夠從本書獲益，因為有些章節會討論 Python 的進階主題，例如裝飾器（decorator）、描述器（descriptor），以及簡介非同步編程。本書可以協助讀者發現 Python 的更多層面，因為有些案例將從這種語言本身的內在開始分析。

我要特別強調本節第一句話的**從業者**這個字，這是一本務實的書籍，書中的範例僅限教學所需，但也企圖模擬實際軟體專案的情境。這不是一本學術著作，所以你應該仔細地思考我提出的定義、意見與建議，你應該以批判、務實的態度來看待這些建議，而不是將之視為教條。畢竟，實用性優於純粹性（practicality beats purit，出自 Zen of Python）。

本書內容

第一章，簡介、程式碼格式與工具介紹設定 Python 開發環境所需的主要工具。我會介紹 Python 開發者在開始使用這種語言時需要瞭解的基本知識，以及協助程式容易閱讀的準則，例如靜態分析、文件、型態檢查及程式碼格式化。

第二章，符合 *Python* 風格的程式討論幾種 Python 的典型表達風格，後續的章節會持續使用它。我們會探討 Python 的特定功能、它們的用法，並開始圍繞著“符合 Python 風格的程式通常是較優質的程式碼”這個概念來建構你的知識。

第三章，好程式的特徵將回顧“編寫可維護程式碼”的軟體工程一般原則。我們將探討這些概念，並藉由這種語言的工具來執行它們。

第四章，*SOLID* 原則討論物件導向軟體的設計原則。這個縮詞是一種軟體工程術語，你將會看到如何在 Python 中運用每一個字母代表的技術。基於 Python 語言的性質，並非所有技術都可以在 Python 中運用。

第五章，使用裝飾器來改善程式說明 Python 最棒的功能之一。在瞭解如何（為函式與類別）建立裝飾器之後，我們會實際使用它們來讓程式碼可供重複使用、劃分職責，與建立更細緻的函式。

第六章，藉由描述器來充分使用物件討論 Python 的描述器，它可將物件導向設計帶到新的層次。雖然描述器與框架和工具的關係較密切，但我們可以看到如何用它來改善程式碼的易讀性和重用（reuse，重複使用，爾後均譯為“重用”）性。

第七章，使用產生器告訴你產生器或許是 Python 最棒的功能。“迭代（iteration）是 Python 的核心元素”這件事讓大家認為 Python 成就了新的程式設計典範。一般來說，我們可以透過產生器與迭代器來思考程式的寫法。藉由學習產生器，你會進一步瞭解 Python 的協同程序（coroutine），以及非同步編程的基本知識。

第八章，單元測試與重構介紹單元測試對於聲稱具備可維護性的基礎程式的重要性。本章將回顧單元測試的重要性，並探討它的主要框架（`unittest` 與 `pytest`）。

第九章，常見的設計模式介紹如何實作 Python 最常見的設計模式，本章不從解決問題的觀點來說明，而是探討它們究竟如何利用更好且更容易維護的方案來處理問題。本章會提到 Python 有某些特質會將一些設計模式隱藏起來，也會採取務實的做法來實作其中的一些設計模式。

第十章，簡潔的結構主要討論一個概念：簡潔的程式是好結構的基礎。第一章談到的所有細節，以及在過程中重複討論的其他內容都會在部署系統時於整個設計中扮演重要的角色。

從本書獲得最大的利益

讀者必須熟悉 Python 的語法，並成功安裝 Python 解譯器，你可以從 `https://www.python.org/downloads/` 下載它。

建議你跟著操作本書的範例，並在電腦上測試程式。為此，強烈建議你用 Python 3.7 建立一個虛擬環境，並用這個解譯器執行程式。你可以到 `https://docs.python.org/3/tutorial/venv.html` 瞭解如何建立虛擬環境。

1

簡介、程式碼格式與工具

本章要討論簡潔程式的第一個概念，會從它是什麼，以及它有什麼意義開始說起。本章的重點是指出簡潔程式不僅僅是最好能夠擁有的好東西，或軟體專案的奢侈品，它其實是必要的。如果沒有高品質的程式碼，專案就會陷入持續累積技術債務進而失敗的風險。

順著這個方向更詳細地說，它是將程式碼格式化與文件化的概念。這聽起來或許也像是一個多餘的要求或工作，但同樣的，我們會說明它在保持基礎程式的可維護性和可用性時會發揮根本性的作用。

我們會分析在專案中採取好的程式編寫準則的重要性。明白“讓程式碼與參考文件保持一致”是件持續的任務之後，我們要說明如何借助自動化工具來減輕工作負擔。為此，我們會簡單地說明如何設置主要的工具，將它們融入組建程序，讓它們可在專案中自動運行。

閱讀本章後，你將瞭解簡潔程式碼的概念、為什麼將程式碼格式化與文件化是重要的任務，以及如何將這個程序自動化。藉此，你將建立“快速組織新專案結構以獲得良好程式品質”的思維模式。

閱讀本章後，你會學到：

- 簡潔程式的意義，它遠比在建構軟體的過程中排列程式的格式重要得多
- 就算如此，對軟體專案的可維護性而言，設計標準的格式仍然是個關鍵因素
- 如何使用 Python 的功能讓程式碼本身成為文件
- 如何設置工具來安排一致的程式碼佈局，讓團隊成員聚焦在問題的本質上

簡潔程式碼的意義

簡潔程式碼沒有唯一或嚴格的定義，我們應該也沒有正式的方法可以用來衡量程式簡潔與否，因此無法用某種工具來瞭解程式碼究竟夠不夠優秀、是不是不好，或是否容易維護。你當然可以執行檢查程式、linter、靜態分析程式等工具，這些工具有很大的幫助，它們是必要的，但還不夠。簡潔的程式碼不是機器或腳本可以判斷的東西（到目前為止），但它是身為專家的我們可以決定的東西。

因為我們幾十年來不斷使用“程式語言”這個字眼，讓我們認為它們是與機器溝通想法，以便執行程式的語言，但我們錯了。這不是全部的事實，只是事實的一部分。隱藏在程式語言底下的，其實就是和別的開發者溝通想法的語言。

這就是簡潔程式碼的本質所在，取決於別的工程師能否閱讀與維護你的程式碼。因此，身為專家的我們是唯一能夠判斷程式碼是否簡潔的人。回想一下，身為開發者，我們花在閱讀程式碼的時間遠比撰寫它還要長。每當我們想要修改或加入新功能時，都會先閱讀想要修改或擴展的程式附近的內容。語言（Python）是我們用來與彼此溝通的東西。

所以與其給你簡潔程式碼的定義（或我的定義），我想請你看完這本書，閱讀關於 Python 典型寫法的全部內容，瞭解一下良好與不良程式碼的差異，找出良好的程式與結構的特點，再自行定義。閱讀本書後，你就能夠自行判斷與分析程式，也會更透徹地瞭解簡潔程式碼。你會知道它是什麼，以及它的意義，無論你的定義是什麼。

簡潔程式碼的重要性

簡潔程式碼如此重要有許多原因，這些原因大都與易於維護、減少技術債務、有效地執行敏捷開發，以及管理成功的專案有關。

我想要討論的第一個概念與敏捷開發和持續交付有關。如果我們希望專案能夠以穩定、可預測的速度持續地提供功能，就必須擁有一個良好的、可維護的基礎程式。

想像一下，你在一條路上開著車往目的地前進。你要估計並告訴朋友到達的時間。如果車子是正常的，而且路況完美，我們找不到實際到達的時間比估計的時間慢很多的理由。如果路況很差，而且你必須走出車外搬走擋在路上的石頭，或閃開裂縫、每隔幾公里就要停車檢查引擎等等，你應該很難知道何時到達（或能否到達）。這個比喻應該很明顯，馬路就是程式碼。如果你想要以穩定、可預測的速度前進，程式碼就必須是易維護且易讀的。如果不是，每當產品經理想要加入新功能時，你就必須停下腳步，重構與修正技術債務。

在軟體中，技術債務是因為一時的妥協以及糟糕的決策產生的問題。在某種程度上，你可以從兩個角度看待技術債務。從現在到過去——如果現在面對的問題是以前寫的劣質程式造成的呢？從現在到未來——如果我們抄捷徑，而不是花時間寫正確的程式，我們會讓未來的自己面對什麼問題？

債務這個字眼取得很好，稱為債務是因為以後再修改程式比現在就修改還要困難，讓你必須付出債務滋生的利息。產生技術債務代表明日的程式碼會比今日更難以維護且昂貴（我們甚至可以衡量這一點），後天又會比明天更昂貴，以此類推。

每當團隊無法準時交出成果，必須停下腳步來修正並重構程式碼時，就是在付出技術債務的代價。

技術債務最糟糕的地方在於它代表了長期的潛在問題。它不會引起高度的關注，相反的，它是個沉默的隱患，分散在專案的各個部分，直到某日、某個特定時間突然甦醒，擾亂團隊的節奏。

格式化在簡潔程式碼之中的角色

簡潔程式碼與按照某種標準（例如 PEP-8，或是用專案準則來定義的自訂標準）來格式化和架構程式有關嗎？簡單的答案是：否。

簡潔的程式超越程式編寫準則、格式化、linting 工具與其他關於程式碼佈局的檢查。簡潔程式碼與實作高品質的軟體以及建立強健、易維護且避免技術債務的系統有關。100% 符合 PEP-8（或任何其他準則）的一段程式或整個軟體元件仍然可能不符合以上的需求。

但是，不注意程式碼的結構也有一些風險。因此，我們會先用比較差的程式結構來分析問題、討論如何處理它們，接著看看如何設置與使用 Python 專案工具來自動檢查與修正問題。

總之，我們可以說，簡潔的程式與 PEP-8 或編寫風格之類的東西沒有關係，它超越那些東西，代表對程式的可維護性與軟體品質而言更有意義的東西。但是我們將會看到，以正確的格式編寫程式對提升工作效率而言至關重要。

遵守專案的編寫風格指南

當你要用有品質的標準來開發專案時，至少應該要擁有一份程式編寫準則。本節將討論其背後的原因，接下來各節會開始介紹一些使用工具來自動執行這項工作的做法。

當我試著在程式佈局中尋找良好的特徵時，在腦海中第一個浮現的東西就是程式的一致性。我希望程式碼具備一致的結構，以方便閱讀與遵循。如果程式碼不正確或沒有一致的結構，團隊成員們各行其是，最終的程式將會讓你付出額外的精力與注意力才能正確地依循。這很容易出錯、具誤導性，且很容易就會出現 bug 或陷阱。

我們希望避免它，因此我們想要的恰恰與之相反——一眼就可以閱讀與理解的程式碼。

如果開發團隊的成員都認同程式的標準建構方式，就可以寫出相似許多的程式碼。因此，你可以快速地看出模式（很快就會討論這一點），認出模式後，就更容易瞭解事物與找出錯誤。例如在發生不對勁的情況時，你可以立刻想起曾經在模式中看過一些引起你的注意力的怪事。你會更仔細地檢查那個部分，大大提升找到錯誤的機會！

Code Complete 這部經典中談到，有一篇論文 *Perceptions in Chess*（1973）對此做了一個有趣的分析，它做一個實驗來確認不同的人如何瞭解或記憶不同的棋位。這項實驗的對象包括各種等級的棋手（新人、中級與西洋棋大師），並在棋盤擺了各種不同的棋位。研究者發現隨機擺放棋子時，新手的表現與大師一樣好，這只是一項記憶練習，每個人都可以用合理的表現完成。當棋子按照真實棋局的邏輯順序排列時（也就是具備一致性且符合模式），大師的表現比其他人好很多。

想像一下，如果在軟體中出現同樣的情況會如何。身為 Python 軟體工程師的我們就像是上述案例中的象棋大師。當程式碼被隨機構築、不遵守邏輯或任何標準時，我們發現錯誤的難度就跟新手沒什麼兩樣。另一方面，如果我們曾經看過結構化的程式，並且知道如何按照特定模式來快速理解程式，我們就有相當大的優勢。

具體來說，在 Python 中，你應該遵守的編寫風格就是 PEP-8。你可以根據當前專案的特殊情況來延伸它，或採取它的某些部分（例如每一行的長度、字串的注釋等等）。但是無論你使用一般的 PEP-8 或是擴展它，建議你持續使用它，而不是從頭開始擬出另一個不同的標準。

遵守這份文件的原因是它已經考慮許多 Python 語法的特殊性質了（通常不適用於其他語言），而且它是由實際開創 Python 語法的作者創造的。因此，應該很難有別的標準的準確性可以比得上 PEP-8 ，遑論改善。

尤其是在處理程式碼時，PEP-8 具備一些其他優良的特性，例如：

- **grep 的能力：**在程式碼中 grep 標記的能力，也就是在一些檔案裡面（以及這些檔案的一部分之中）搜尋特定字串的能力。這項標準有一個區分如何編寫“對變數賦值”與“傳給函式的關鍵字引數”的項目。

舉例來說，假如我們正在除錯，想要找出指派給參數 `location` 的值是在哪裡傳入的，我們可以執行下面的 `grep` 命令，它的結果將指出我們要找的檔案與行數：

```
$ grep -nr "location=" .
./core.py:13: location=current_location,
```

接下來我們想要知道這個變數在哪裡被賦值，下面的命令也可以指出我們要找的資訊：

```
$ grep -nr "location =" .
./core.py:10: current_location = get_location()
```

PEP-8 建立的標準是：如果你用關鍵字傳遞引數給函式，就不使用空格，但是如果對變數賦值就要使用空格。出於這個原因，我們可以調整搜尋的條件（第一次搜尋的 = 旁邊沒有空格，第二次使用一個空格）來提高搜尋的效率。這就是遵守規範的好處。

- **一致性：**如果程式碼看起來具備統一的格式，閱讀就會輕鬆很多。這在培訓新人時特別重要，如果你想要歡迎新的開發者加入專案，或是雇用新的程式員加入團隊（可能是比較缺乏經驗的），並且讓他們更熟悉程式（甚至可能包含幾個程式存放區），如果他們打開的檔案裡面的程式碼編排、文件、命名規範等等都是一致的，就會輕鬆許多。

- **程式品質：**藉著閱讀結構化的程式碼，你可以熟練地一眼就理解它（同樣類似 *Perceptions in Chess* 所說的），並且更容易找到 bug 與錯誤。檢查程式碼品質的工具也可以藉此提示潛在的 bug。程式碼靜態分析或許可協助減少每行程式的 bug 比率。

docstring 與註釋

本節介紹如何在 Python 程式碼內將程式碼文件化。良好的程式碼除了可說明自己的意圖之外，也是經過妥善文件化的。解釋它應該做什麼工作（而不是如何做）是很好的做法。

將程式碼文件化與在它裡面加上註解（comment）不一樣，這點非常重要。註解是不好的東西，應避免使用。文件化是“解釋資料的型態、提供它們的範例，或註解變數”之類的事情。

這件事與 Python 有密切關係，因為 Python 使用動態型態，讓我們很容易在函式與方法之間搞不清楚變數或物件的值。因此，說明這項資訊可方便程式碼的讀者。

此外還有一個專門與註釋（annotation）有關的理由。Mypy 之類的工具可以協助執行一些自動檢查，例如型態提示。你會發現加上註釋最終是可以獲得回報的。

docstring

基本上，docstring 是放在原始程式碼裡面的文件。它是放在程式碼某處的文字字串，目的是將那部分的邏輯文件化。

請留意**文件化**這個字眼。這個細節很重要，它代表解釋，不是辯解。docstring 不是註解，而是文件。

由於許多原因，在程式碼裡面加上註解是不好的做法。首先，使用註解代表你沒有用程式碼來表達想法，當你需要解釋為何或如何做某件事時，應該代表那段程式不夠好，讓初學者無法單純藉由觀看程式碼來瞭解它的意思。其次，它可能產生誤導。比起花時間閱讀複雜的段落，閱讀關於程式應該如何工作的註解之後才發現那段程式做的是不同的事情更糟糕。人們往往在修改程式之後忘了更改註解，造成被改過的程式旁邊的註解已經過時了，從而產生危險的誤導。

有時在罕見的情況下，我們不得不使用註解。或許是在第三方程式庫中有我們必須迴避的錯誤。在這種情況下，加入一小段描述性的註解可能是可接受的。

但是對 docstring 而言，情況完全不同。再說一次，它們不代表註解，而是屬於程式的特定元件（模組、類別、方法或函式）的文件。使用它們不但是可接受的，更是應該鼓勵的行為。盡可能地加入 docstring 是很好的習慣。

應該將它們放入程式的原因（甚至是必要的，取決於專案的標準）在於 Python 是動態型態語言，這意味著（舉例來說）函式的任何參數都可以接收任何值。Python 不會強制規定或檢查任何這類的東西。所以，如果你在程式裡面發現一個必須修改的函式，幸運的是，你也發現那個函式與參數都使用富描述性的名稱。但即使如此，你還是有可能不清楚應該傳給它什麼型態，就算知道了，它們又會被如何使用？

此時良好的 docstring 就派上用場了。將函式期望收到的輸入與產生的輸出文件化是很好的做法，可協助函式的讀者瞭解它應該如何工作。

看一下這個來自標準程式庫的好例子：

```
In [1]: dict.update??
Docstring:
D.update([E, ]**F) -> None.Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E:D[k] =
E[k]
If E is present and lacks a .keys() method, then does: for k, v in E:D[k]
= v
In either case, this is followed by: for k in F:D[k] = F[k]
Type: method_descriptor
```

這個關於 update 方法的字典的 docstring 提供了實用的資訊，指出它有兩種不同的用法：

1. 我們可以用 .keys() 來傳遞某個東西（例如另一個字典），它會使用以參數傳入的物件的鍵來更新原始字典：

```
>>> d = {}
>>> d.update({1: "one", 2: "two"})
>>> d
{1: 'one', 2: 'two'}
```

2. 我們可以傳遞鍵值組成的可迭代物，將它們拆開來 update（更新）：

```
>>> d.update([(3, "three"), (4, "four")])
>>> d
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

在任何情況下，字典都會用它收到的關鍵字引數來更新。

這項資訊對想要學習與瞭解新函式如何運作，以及如何使用它的人來說至關重要。

請注意，在第一個範例中，我們使用雙問號來取得函式的 docstring（dict.update??）。這是 IPython 互動式解譯器的功能。當你呼叫它時，它會印出你希望瞭解的物件的 docstring。既然我們可以這樣獲得標準程式庫函式的幫助，想像一下，當你在你寫的函式裡面加入 docstring，以便讓讀者（程式碼的使用者）用同樣的方式來瞭解做法時，會給他們帶來多大的方便。

docstring 不是與程式碼分開或獨立的東西。它是程式碼的一部分，而且你可以取得它。當你為物件定義 docstring 時，它就成為物件的一部分，你可以用物件的 __doc__ 屬性來取得它：

```
>>> def my_function():
... """Run some computation"""
... return None
...
>>> my_function.__doc__
'Run some computation'
```

這代表你也可以在執行階段讀取它，甚至可以用原始程式來生成或編譯文件。事實上，坊間也有做這些工作的工具。當你執行 Sphinx 時，它會幫你的專案建立基本的文件骨架。特別是當你加上 autodoc 副檔名（sphinx.ext.autodoc）時，這個工具會從程式碼取出 docstring 並將它們放在該函式的文件頁面內。

如果你有建立文件的工具，請公開它，讓它成為專案本身的一部分。當你參與開放原始碼專案時可以使用 read the docs，它會自動幫每一個版本分支產生文件（可設置的）。在公司的專案中，你可以使用同樣的工具或設置這些服務，但無論你做什麼決定，重點是準備好文件，讓團隊的所有成員都可以使用它。

不幸的是 docstring 有一個缺點——它與所有文件一樣，都需要你持續親自維護。當程式改變時，你就要更新它。另一個問題是為了讓 docstring 真正提供幫助，它們必須夠詳細，這需要使用好幾行文字。

維護正確的文件是軟體工程無法避免的挑戰。做這件事有它的道理，仔細想一下，文件需要人工編寫的原因是它的目的是要讓人閱讀的。如果它是自動產生的，可能就沒有太大的用途。為了讓文件產生價值，團隊的成員都必須同意它是需要人為干預的東西，因此需要付出心力。瞭解軟體的關鍵並非只有程式碼，它附帶的文件也是應交付的部分。因此，當函式被人修改時，更新那段程式對應的文件也一樣重要，無論那份文件是 wiki、使用者手冊、README 檔案，還是 docstring。

註釋

PEP-3107 加入了註釋的概念。它們的基本概念是提示程式的讀者“函式的引數值是什麼”。**提示**這個詞不是隨便使用的，註釋可以做型態提示，本章會在初步介紹註釋之後進一步說明。

註釋可用來指定已定義的變數應使用的型態。它不但與型態有關，也與“可以協助瞭解變數代表的意義”的任何詮釋資料有關。

請看以下的範例：

```
class Point:
    def __init__(self, lat, long):
        self.lat = lat
        self.long = long

def locate(latitude: float, longitude: float) -> Point:
    """用物體的座標在地圖中尋找它"""
```

我們在這裡使用 float 來指出 latitude 與 longitude 的期望型態。它們只是用來幫助函式的讀者瞭解函式希望收到的型態的資訊。Python 既不會檢查這些型態，也不會強制套用它們。

你也可以指定函式回傳值的預期型態。本例的 Point 是使用者定義的類別，它代表被回傳的東西是 Point 的實例。

但是可以拿來註釋的東西不是只有型態與內建物，基本上，在當前 Python 解譯器的範圍內有效的任何東西都可以放在那裡，例如用來解釋變數用途的字串、當成回呼（callback）的可呼叫物或驗證函式等等都可以。

隨著註釋的加入，Python 也加入新的特殊屬性，__annotations__。它可以用來讀取一個字典，字典裡面有我們定義的註釋名稱（鍵）及其對映的值。在本例中，字典的長相是：

```
>>> locate.__annotations__
 {'latitude': float, 'longitue': float, 'return': __main__.Point}
```

我們可以用它來產生文件、進行驗證，或是在必要時強制檢查程式碼。

談到用註釋來檢查程式碼，這正是 PEP-484 的用途。這個 PEP 指定了型態提示的基本概念，也就是透過註釋來檢查函式的型態。我引用 PEP-484 本身的話來再次聲明：

> “Python 以後都是動態型態語言，作者不希望強制使用型態提示，即使按照規範也是如此。”

型態提示的概念是用別的工具（非解譯器的）來檢查與評估型態在整個程式碼裡面的用法，並且在發現不相容時提示使用者。稍後的章節會說明做這些檢查的工具 Mypy，也會討論如何為專案設置工具及使用它們。你現在可以將它當成一種 linter，可在程式碼內檢查型態的語義（semantic）。這種工具有時可以在執行測試和檢查時，協助你提早發現 bug。因此，你最好為專案設置 Mypy，並且像其他的靜態分析工具一樣使用它。

但是型態提示的意義不是只有檢查程式碼的型態而已。Python 3.5 之後的版本加入新的型態模組，大大改善在 Python 程式裡面定義型態與註釋的方式。

它的基本理念是將語義延伸到更有意義的概念上，讓我們（人類）更方便瞭解程式碼的意思，或是某處所期望的事物。例如，當你有一個函式用一個參數來處理串列或 tuple 時，你可以將這兩種型態之一做成註釋，甚至用一個字串來解釋它。但是在使用這個模組時，你也可以告訴 Python 它需要一個可迭代物或是序列（sequence）。你甚至可以認出它的型態或值，例如，知道它接收了一個整數序列。

當我寫這本書時，Python 做了一項關於註釋的改善——從 Python 3.6 開始，我們可以直接註釋變數，而不是只能註釋函式的參數及其回傳型態。這是在 PEP-526 加入的功能，讓你不需要對已定義的變數賦值就可以宣告它的型態，例如：

```
class Point:
    lat: float
    long: float

>>> Point.__annotations__
{'lat': <class 'float'>, 'long': <class 'float'>}
```

註釋可取代 docstring 嗎？

這是一個好問題，因為在還沒有註解的舊版 Python 裡面，將函式參數型態或屬性型態文件化的做法是為它們加上 docstring。當時甚至有一些格式規範，規定如何用 docstring 加入函式的基本資訊，包括型態、各個參數與結果的意義，以及函式可能引發的異常狀況。

現在它們大部分都可以用註釋以更紮實的方式來處理了，所以可能有人會懷疑是否還需要使用 docstring。答案是肯定的，因為它們是相輔相成的。

沒錯，有些之前放在 docstring 裡面的資訊已經被移到註釋了，但是這只代表我們為 docstring 留下空間來建立更好的文件。特別是對動態與嵌套式資料型態而言，提供希望收到的資料的範例絕對是個好方法，因為這樣才能讓人更瞭解將要使用的東西。

請看以下的範例。假如有一個函式期望收到一個字典來驗證資料：

```
def data_from_response(response: dict) -> dict:
    if response["status"] != 200:
        raise ValueError
    return {"data": response["payload"]}
```

這個函式接收一個字典，並回傳另一個字典。如果 "status" 鍵的值不符合期望，它可能會發出例外，但是這段程式沒有提供進一步的資訊。例如，正確的 response 物件實例究竟長怎樣？ result 實例長怎樣？為了解決這兩個問題，有一種很好的做法是用文件來說明這個函式希望透過參數接收的資料，以及它回傳的資料。

我們來看一下能否藉助 docstring 來更詳細地說明：

```
def data_from_response(response: dict) -> dict:
    """如果回應是 OK，則回傳它的負載。

    - response:A dict like::

    {
        "status": 200, # <int>
        "timestamp": "....", # 當前日期時間的 ISO 格式字串
        "payload": { ... } # 含回傳資料的 dict
    }

    - Returns a dictionary like::

    {"data": { .. } }

    - Raises:
    - ValueError if the HTTP status is != 200
    """
    if response["status"] != 200:
        raise ValueError
    return {"data": response["payload"]}
```

這就更能夠讓人瞭解這個函式期望收到與回傳什麼東西了。文件是很好用的輸入機制，它不但可以幫助瞭解被四處傳遞的東西是什麼，當你做單元測試時，它也是很寶貴的來源。我們可以從這裡衍生資料當成測試的輸入來使用，並且能夠知道值是正確或錯誤的。事實上，測試程式也可以當成程式碼的可互動文件，不過這要留待稍後解釋。

我們現在可以知道鍵有哪些可能的值以及它們的型態，並且更具體地瞭解資料的外觀。如前所述，我們需要付出的代價是它占用好幾行的文字，而且需要詳細地說明才能發揮真正的效果。

設置強制執行基本品管的工具

本節將介紹如何設置基本工具來自動檢查程式碼，目的是利用部分的重複性驗證檢查。

之前提過，程式碼是讓我們人類瞭解的，所以只有我們可以判斷程式碼的好壞，這是重點所在。我們應該投資時間來審查程式碼、思考什麼是好程式，以及它是否容易閱讀和理解。當你閱讀同事編寫的程式時，要考慮下列問題：

- 同事容易理解和依循這些程式嗎？
- 它是否以問題領域為依據？
- 新同事能否理解它，並有效率地使用它？

之前談過，將程式碼格式化、使用一致的排版、做適當的縮排都是基礎程式必備的特徵，但這些還不夠，它們是我們這種具備高度品質意識的工程師認為理所當然的事情，所以我們所閱讀與撰寫的程式都不應只是符合這些基本的排版概念。因此，我們不想要把時間浪費在檢查這些項目上，而是要更有效率地查看程式的模式，以瞭解它真正的意義，並提供有價值的結果。

這些檢查都必須是自動化的。它們應該成為測試或檢查清單的一部分，而這些部分也應該成為持續整合版本的一部分。如果有未通過檢查的項目，我們就讓組建失敗。唯有如此，才能真正確保程式碼的結構永遠具備連續性。它也是一種可供團隊參考的客觀參數。與其讓團隊的一些工程師或領導者不斷在程式碼審查時做出關於 PEP-8 的評論，比較客觀的做法是直接讓組建自動失敗。

用 Mypy 做型態提示

Mypy（`http://mypy-lang.org/`）是執行 Python 靜態型態檢查的主力工具。它被安裝之後可以分析專案的所有檔案，檢查型態的使用是否有不一致的情況。它很方便的地方在於，在多數情況下，它可以在早期找到實際的 bug，但有時也會產生偽陽性（false positive）的結論。

你可以用 `pip` 安裝它，建議你將它加入設定檔，讓它成為專案的依賴項目：

```
$ pip install mypy
```

當你在虛擬環境安裝它並執行上面的命令之後，它就可以回報型態檢查的結果了。試著盡量採納它的報告，因為在多數情況下，它找到的東西都有助於避免悄悄溜進產品中的錯誤。但是這種工具不是完美的，如果你認為它的報告是偽陽性的，可以用這個標記來忽略那一行註解：

```
type_to_ignore = "something" # type: ignore
```

用 Pylint 來檢查程式

坊間有許多工具可檢查 Python 程式的結構（基本上就是遵守 PEP-8），例如 pycodestyle（之前稱為 PEP-8）、Flake8 等等。它們都是可設置的，只要執行它們提供的命令就可以使用它們。其中，我發現 Pylint 是最完整（且嚴謹）的一種。它也是可設置的。

同樣的，你只要用 `pip` 在虛擬環境安裝它就可以了：

```
$ pip install pylint
```

接著只要執行 `pylint` 命令就可以檢查程式了。

你也可以透過名為 `pylintrc` 的組態檔來配置 Pylint。

你可以在這個檔案裡面啟用或停用某些規則，或是將其他規則參數化（例如，改變欄位的最大長度）。

設定自動檢查

在 Unix 開發環境中，最常見的工作方式是使用 makefile。**makefile** 這種強大的工具可讓我們配置想要在專案中執行的命令，大部分都用來編譯、執行等等。此外，我們也可以在專案根目錄使用 makefile 來配置一些命令，來自動檢查程式碼的格式和規範。

你可以幫每一個測試指定目標，再一起執行測試，例如：

```
typehint:
mypy src/ tests/

test:
pytest tests/

lint:
pylint src/ tests/
```

```
checklist: lint typehint test

.PHONY: typehint test lint checklist
```

我們要執行的命令（在開發機器與持續整合環境版本中）是：

```
make checklist
```

它會用下面的步驟來執行每一項工作：

1. 檢查程式是否符合編寫準則（例如 PEP-8）
2. 接著在程式碼中檢查型態的使用
3. 最後執行測試

如果任何一個步驟失敗了，整個程序都會視為失敗。

除了在版本中配置這些自動化的檢查之外，讓團隊遵守某個規範，以及以自動化的方式來建構程式碼也是很好的做法。Black（`https://github.com/ambv/black`）之類的工具可以將程式碼自動格式化。坊間還有許多可以自動編輯程式碼的工具，但 Black 有趣的地方在於它是用獨特的形式來做這件事。它非常堅持己見且具備確定性（deterministic），因此程式碼最終必定有相同的排版方式。

例如，Black 的字串必定使用雙引號，參數的順序一定遵循相同的結構。這聽起來或許很沒彈性，但唯有這種做法可將程式碼的差異減至最低。如果程式碼始終遵循相同的結構，當你修改程式時，pull request 只會顯示實際的更改，不會有多餘的資訊。它的局限性比 PEP-8 強，但它也很方便，因為當你用工具直接將程式碼格式化時，就可以把焦點放在眼前的問題上，而不需要處理格式的問題。

在寫這本書時，它唯一可以設置的選項就是每一行的長度。其他的東西都會被專案的標準更正。

下面是符合 PEP-8，但不符合 `black` 規範的程式：

```
def my_function(name):
 """
 >>> my_function('black')
 'received Black'
 """
 return 'received {0}'.format(name.title())
```

接下來，我們可以執行下面的命令來將檔案格式化：

```
black -l 79 *.py
```

接著，我們可以看到這個工具寫了些什麼：

```
def my_function(name):
 """
 >>> my_function('black')
 'received Black'
 """
 return "received {0}".format(name.title())
```

較複雜的程式可能會被更改更多東西（結尾的逗號等等），但它的概念很易懂。再次強調，它很堅持己見，但使用工具來協助處理細節仍然是好方法。這也是 Golang 社群在很久以前就學會的事情，以致於有一種標準工具程式庫 `got fmt` 可根據語言的規範來將程式碼自動格式化。很棒的是，現在 Python 也有類似的工具了。

這些工具（Black、Pylint、Mypy 等等）可以和編輯器或 IDE 整合，讓你的工作更輕鬆。設置編輯器並且在你儲存檔案時（或藉由捷徑）做這類的修改是很棒的投資。

結論

我們已經初步瞭解簡潔程式碼的概念，並解釋如何實現它了，這些知識是本書接下來的內容的基石。

更重要的是，我們知道簡潔的程式比程式碼的結構與排版更重要。當我們檢查程式碼是否正確時，應該把焦點放在它能否呈現它的理念。簡潔的程式碼與易讀性、易維護性、盡量降低技術債務，以及“有效地溝通理念讓別人瞭解我們最初的意圖”息息相關。

但是我們也提到，出於許多原因，遵循程式碼編寫風格與準則也很重要。我們認為它們是必須但不充分的。由於它是每一個紮實的專案都應該遵守的基本要求，所以用工具來處理比較好。因此，將這些檢查自動化至關重要，所以我們必須知道如何設置 Mypy、Pylint 等工具。

下一章會更深入地討論 Python 專屬的程式碼，以及如何以符合風格的 Python 來表達想法。我們會討論讓程式碼更緊湊且有效的 Python 典型表達風格。透過分析你將會看到，一般而言，Python 完成工作的概念和手段與其他語言是不同的。