序

提到高性能計算，你可能會想到模擬複雜氣象，或試著解讀遙遠恆星訊號的巨型電腦叢集，很多人認為只有需要建構特殊系統的人，才需要關心程式碼的性能，但是從你翻開這本書開始，你就朝著編寫高性能程式所需的理論和做法邁出一大步了。每一位程式員都可以了解如何建構高性能系統並得到好處。

有些 app 只能藉著編寫性能優化程式來提升性能，如果這是你的課題，你就來對地方了。但是除此之外還有更廣泛的 app 可以藉著使用高性能程式碼而獲益。

我們經常認為，使用新技術的能力可以推動創新，但我也喜歡可以將技術駕馭力提升好幾個數量級的能力。如果你可以讓處理某一件事情的時間成本或計算成本便宜十倍，你可以處理的 app 數量就會突然多得超乎想像。

我自己是在十幾年前工作時發現這一條原則的，當時我在一家社交媒體公司工作，我們必須分析好幾 TB 的資料，來了解用戶在社交媒體上究竟按下貓照片的次數比較多，還是按下狗照片的次數比較多。

當然，狗照片的次數比較多。貓只是品牌效果較好而已。

在當時，這樣子使用計算時間與基礎設施是非常輕率的舉動！由於我們有能力使用過往只能用於高價值 app 的技術（例如詐欺辨識）來處理普通的問題，所以我們開啟了全新可能性。我們可以利用從實驗得到的知識來建構一組進行搜尋與內容發現的新產品。

舉個你今天可能遇到的例子，假設有個機器學習系統可以從監視器影片中認出意外出現的動物或人。在使用高性能的系統時，你可以將那個模型放入相機本身來提高隱私性，即使模型在雲端上運行，你也可以使用少很多的計算資源與電力資源，從而為環境帶來好處，並降低你的營運成本，如此一來，你可以騰出資源來關注周遭的問題，做出更有價值的系統。

我們都想要做出高效、容易了解且高性能的系統。遺憾的是，我們往往只能從這三個要素中選出兩個（或一個）。本書是寫給想要滿足這三個要素的人看的。

本書與探討這個主題的其他書籍不同的地方有三個。首先，它是為我們這種程式員而寫的，它會告訴你做出某個選擇的所有背景。第二，Gorelick 與 Ozsvald 很好地組織和解釋支持該背景的理論。最後，在這本新版本中，你將了解實作這些方法時最實用的程式庫的特殊性質。

這是少數幾本可以改變你的編程思維的書籍之一，我已經把這本書送給許多人，讓他們有機會利用書中介紹的工具。書中的觀念將會讓你成為更棒的程式員，無論你使用哪一種語言或環境來工作。

好好享受這一場冒險！

—*Hilary Mason*，
Accel 常駐資料科學家

前言

Python 很容易學，你之所以看這本書，可能是因為雖然你的程式可以正確運行，但你想要讓它跑得更快，你想要讓程式更容易修改，並且能夠快速地反覆試驗新想法。在**容易開發**與**盡量按照希望的速度運行**之間的兩難很容易理解，也讓人很討厭，但這種情況是可以解決的。

有些人希望讓一系列的程序跑得更快；有些人在使用多核心架構、叢集或圖形處理單元時遇到麻煩；有些人需要可擴展的系統，能夠隨機應變，並且視資金狀況來處理工作量，且不失其可靠性；有些人則是發現他們的編程技術（通常是參考其他語言的）不像別人的案例那麼自然。

本書將探討以上所有主題，實際教你如何找出瓶頸，以及製作更快速且更容易擴展的解決方案。我們也會介紹一些過往的實戰經驗，幫助你避開別人承受過的打擊。

Python 很適合用來進行快速開發、生產部署，以及製作可縮放系統。它的生態系統有很多能為你擴展它的熱心人士，讓你可以把更多時間專注於更有挑戰性的任務上。

本書對象

我們使用 Python 很長一段時間了，所以知道為什麼有些東西跑得很慢，也曾經多次討論如何使用 Cython、numpy 與 PyPy 等技術來解決問題。或許你也用過其他語言，知道解決性能問題的手段不只一種。

雖然本書主要針對 CPU-bound（受 CPU 限制的）問題，但我們也關注資料傳輸與記憶體受限的解決方案。這些問題通常是科學家、工程師、定量分析師和學者面臨的問題。

我們也會介紹 web 開發者可能遇到的問題，包括移動資料，以及使用 PyPy 等即時（JIT）編譯器和非同步 I/O 來輕鬆提升性能。

具備 C（或 C++，也許 Java）的使用經驗是有幫助的，但它不是先決條件。Python 最常見的解譯器（CPython ——通常是當你在命令列輸入 python 時執行的標準解譯器）是用 C 寫成的，所以它的掛勾（hook）與程式庫都揭露了內部的 C 機制。我們介紹的許多技術都不預設你有任何 C 知識。

或許你對 CPU、記憶體架構、資料匯流排有初步的了解，但再次強調，這不是絕對必要的。

不適合本書的讀者

本書適合中高階 Python 程式員。雖然認真的 Python 程式員也可以跟著操作，但我們建議你具備堅實的 Python 基礎。

我們不會討論儲存系統優化。如果你遇到 SQL 或 NoSQL 瓶頸，這本書可能無法幫助你。

你將學到什麼

筆者一直以來都在業界和學術界處理大量的資料，所以多年來都有我想要更快得到答案！和「可擴展的架構」的需求。我們會盡量提供來之不易的經驗，避免你犯下我們犯過的錯誤。

在各章的開頭，我們會列出後續內容將要回答的問題（如果沒有，麻煩告訴我們，好在下次改版時修訂！）。

我們討論的主題有：

- 電腦機制的背景知識，讓你知道幕後發生的事情
- 串列與 tuple——在這些基本資料結構內有哪些語義和速度方面的微妙差異
- 字典與集合——在這些重要的資料結構內的記憶體配置策略和存取演算法
- 迭代器——如何將程式寫得更符合 Python 風格，以及使用迭代來開啟無限資料串流的大門
- 純 Python 法——如何有效地使用 Python 和它的模組

- 用 numpy 處理矩陣——如何大刀闊斧地使用貼心的 numpy 程式庫
- 編譯和即時計算——編譯成機器碼，以更快的速度來處理，用分析的結果來引導工作
- 並行——高效移動資料
- multiprocessing——使用內建的 multiprocessing 程式庫來進行平行計算、高效地共享 numpy 矩陣，以及程序間通訊（IPC）的成本與益處
- 叢集計算——轉換 multiprocessing 程式碼，讓它在本地或遠端叢集上的研究與生產系統中運行
- 使用更少 RAM——如何解決大型問題而不必購買大型的電腦
- 實戰經驗——從曾經遭受打擊的人吸取教訓，以避免它們

Python 3

Python 3 在 Python 2.7 經歷了 10 年的遷移過程並且被棄用之後，於 2020 年成為標準的 Python 版本。如果你還在使用 Python 2.7，這是不對的選擇——許多程式庫都不支援你的 Python 程式了，即使有支援，代價也會越來越高。請幫幫社群，遷移至 Python 3，並且在所有新專案中使用 Python 3。

本書使用 64-bit Python，雖然我們也支援 32-bit Python，但它在科學工作中罕見多了。我們希望所有程式庫都能正常工作，但數字精度可能會改變，因為它取決於計數時可用的 bit 數。64-bit 和 *nix 環境（通常是 Linux 或 Mac）在這個領域占主導地位。64-bit 可讓你處理更大量的 RAM。*nix 可讓你做出來的 app 能夠以易懂的方式和行為進行部署與設置。

如果你使用 Windows，你就要繫好安全帶了。雖然大部分的程式都可以正常運作，但有一些是 OS 專屬的，你必須自行研究 Windows 解決方案。Windows 用戶最大的困難就是安裝模組——在 Stack Overflow 等網站進行研究可以找到解決方案。如果你使用 Windows，或許在虛擬機器（例如 VirtualBox）中安裝 Linux 可以協助你更自由地進行試驗。

Windows 用戶絕對要考慮 Anaconda、Canopy、Python(x,y) 或 Sage 等包裝方案。這些版本也會讓 Linux 和 Mac 用戶輕鬆許多。

第一章

了解高性能 Python

看完這一章之後，你可以回答這些問題

- 電腦架構的元素有哪些？
- 常見的替代電腦架構有哪些？
- Python 如何將底層的電腦架構抽象化？
- 寫出高性能 Python 程式的障礙有哪些？
- 有哪些策略可以協助你成為高績效的程式員？

我們可以將電腦想成它可以移動一些資料位元，並且以特殊的方式轉換它們，以達到特定的結果。但是，這些動作有時間成本。因此，**高性能編程**可以視為可將這些操作最小化的工作，無論是藉著減少開銷（overhead）（即，寫出更高效的程式碼），或是藉著改變執行這些操作的方式，讓每一個操作都更有意義（即，找出更合適的演算法）。

我們把焦點放在降低程式碼的開銷，以便更深入地了解讓我們在裡面移動位元的實際硬體。因為 Python 已經很努力地將硬體的實際互動抽象化了，所以我們的做法乍看之下似乎是徒勞的，然而，藉著了解如何用最好的方法在實際的硬體中移動位元，以及 Python 的抽象如何移動位元，你可以寫出性能更高的 Python 程式。

基本電腦系統

構成電腦的底層元件可以簡單地分成三個基本部分：計算單元、記憶單元，及它們之間的連結。此外，這些單元都有不同的屬性（property），計算單元的屬性是它每秒可以做多少次計算，記憶單元的屬性是它可以保存多少資料，以及從它讀出和對它寫入的速度多快，最後，連結的屬性是將資料從一個地方搬到另一個地方有多快。

我們可以使用這些元件，以多個複雜程度來討論標準的工作站。例如，標準工作站可視為具備一個中央處理單元（CPU）作為計算單元，連接至兩個分開的記憶單元，隨機存取記憶體（RAM）與硬碟（它們分別有不同的功能與讀 / 寫速度），最後有一個匯流排，連接所有這些部分。但是，我們也可以更詳細地觀察，發現 CPU 本身也有多個記憶單元：L1、L2，有時甚至有 L3 與 L4 快取，雖然它們的容量很小，卻有極高的速度（從幾 KB 至數十 MB）。此外，新的電腦架構通常有新的配置（例如，Intel 的 SkyLake CPU 將前端匯流排換成 Intel Ultra Path Interconnect，並且重組許多連結）。最後，在工作站的這兩種概要結構中，我們都忽略了網路連結，它與許多其他計算與記憶單元相較之下是極緩慢的連結！

為了協助釐清這些複雜的問題，我們簡單介紹一下這些基本元素。

計算單元

電腦的**計算單元**是它的功能的核心——它可以將它收到的任何位元轉換成其他位元，或改變當前程序的狀態。CPU 是最常用的計算單元，然而，圖形處理單元（GPU）這種輔助計算單元也越來越流行。它們最初被用來提升電腦繪圖速度，但現在變得很適合用於數字應用，而且因其平行性質可同時進行許多計算，它也越來越實用。無論類型如何，計算單元都接收一系列的位元（例如，代表數字的位元），輸出另一組位元（例如，代表這些數字總和的位元）。除了對整數執行基本算術，以及對二進制數字執行實數和逐位元計算之外，有些計算單元也提供專門的操作，例如「fused multiply add」接收三個數字，`A`、`B` 與 `C`，然後回傳 `A * B + C`。

計算單元的主要屬性是可在一個週期之內執行的操作數，以及一秒內的週期數。第一個值是用指令平均週期數（IPC）來衡量的[1]，後者則是用它的時脈速度來衡量的。每當有新計算單元被做出來時，大家就會用這兩個指標來做比較。例如，Intel Core 系列有極高的 IPC，但時脈較低，Pentium 4 晶片則相反。另一方面，GPU 有極高的 IPC 與時脈，但它們有其他的問題，例如我們會在第 8 頁的「通訊層」討論的通訊緩慢。

此外，儘管增加時脈速度幾乎可以立即加快計算單元運行的所有程序的速度（因為它們每秒能夠執行更多計算），但具備較高的 IPC 時，你也可以藉著改變**向量化**（*vectorization*）程度來大幅影響計算能力。當 CPU 可以同時接收多個資料片段，並且能夠一次操作它們全部時，就會發生向量化。這種 CPU 指令稱為「單指令，多資料（single instruction, multiple data，SIMD）」。

總的來說，計算單元在過去十年的發展速度相當緩慢（見圖 1-1）。因為電晶體變小的速度被物理因素限制，時脈與 IPC 都停滯不前。因此，晶片製造商一直依靠其他方法來獲得更高的速度，包括同時多執行緒（一次可以執行多個執行緒）、更聰明的無序（out-of-order）執行，以及多核心架構。

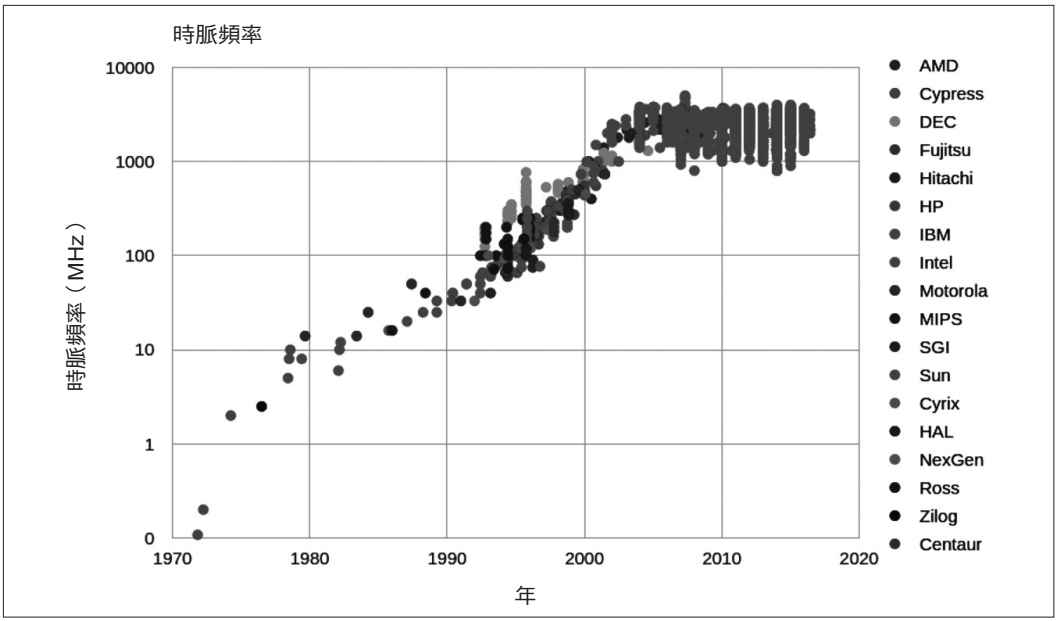

圖 1-1　CPU 時脈速度的歷史走勢（來自 CPU DB（*https://oreil.ly/JnJt2*））

1　請勿和縮詞相同的程序間通訊（interprocess communication）混為一談，第 9 章會討論這個主題。

超執行緒提供虛擬的第二 CPU 來承載作業系統（OS），聰明的硬體邏輯則試著在一顆 CPU 的執行單元內交錯執行兩個指令執行緒。成功的話，它可以讓一個執行緒提升 30% 之多。當這兩個執行緒處理的工作單位使用不同類型的執行單元時，這種做法通常有很好的效果，例如，一個執行緒執行浮點運算，另一個執行整數運算。

無序執行可讓編譯器發現有哪些線性的程式順序不需要依靠之前的工作結果，所以可讓工作以任何順序執行，或同時執行。只要有序的結果可以在正確的時間出現，程式就可以繼續正確執行，即使不同的工作沒有按照它們在程式中的順序執行。採取這種做法時，我們可以在其他指令被阻塞（例如等待記憶體存取）時執行某些指令，進而更充分地利用資源。

最後，對更高階的程式員來說，最重要的是流行的多核心架構。這種架構在同一個單元裡面放入多顆 CPU，可以提升總體性能，同時不妨礙各個單元的速度。這就是為何目前很難找到少於兩顆核心的電腦，當電腦有兩顆核心時，它就有兩個互相連接的實體計算單元。雖然這種設計可以提升每秒可以完成的操作總數，但它會讓你更難編寫程式！

在 CPU 中加入更多核心不一定可以提高程式的執行期速度。原因來自 *Amdahl* **定律**（*https://oreil.ly/GC2CK*）。簡單地說，Amdahl 定律是：如果需要在多顆核心內執行的程式有多個子程序必須在同一顆核心內處理，這就會阻礙使用多核心可獲得的最大速度。

打個比方，如果我們有一份市調想要讓 100 個人填寫，而那份調查需要用 1 分鐘完成，如果我們有一位負責調查的訪問者，我們就可以在 100 分鐘之內完成這項任務（即，這個人到受訪者 1 前面，問問題，等他回答，再到受訪者 2 前面）。這種讓一個人問問題並等待回答的方法很像循序程序。使用循序程序時，我們每次滿足一項操作，每一項操作都必須等待上一個完成。

但是，如果我們有兩位訪問者，我們就可以平行進行調查，只要用 50 分鐘就可以完成程序。之所以如此是因為每一個人在問問題的時候，不需要知道另一個人的任何事情。因此，這項工作可以輕鬆地拆開，訪問者之間沒有任何依賴性。

加入更多訪問者會提升更高速度，直到有 100 位訪問者為止。此時，這個程序需要 1 分鐘完成，會影響總時間的因素，只有受訪者回答問題的時間。加入更多訪問者不會讓速度進一步提升，因為額外的人員沒有工作可做，因為所有受訪者都已經被問問題了！此時，你只能藉著減少個別調查（這個問題的連續部分）的完成時間來降低總體市調時間。同樣地，在 CPU，我們可以加入更多能夠視需要執行不同計算區塊的核心，直到到達特定核心完成其工作所需的時間瓶頸為止。換句話說，任何平行計算的瓶頸一定是被分離的較小循序任務。

此外，在 Python 中使用多核心的主要障礙在於 Python 使用**全域解譯器鎖**（*global interpreter lock*，*GIL*）。GIL 可確保 Python 程序一次只執行一個指令，無論它目前使用多少核心。也就是說，即使 Python 程式一次可使用多個核心，無論何時也只有一個核心在運行 Python 指令。用上述的市調例子來說，這代表即使我們有 100 位訪問者，一次也只有一個人可以問一個問題並且聆聽回答。這根本消除擁有多位訪問者帶來的任何好處了！雖然這看起來確實是個障礙，尤其是因為目前的計算趨勢是使用多個計算單元而不是使用更快速的，但這個問題可以藉著使用這些工具來避免：`multiprocessing` 等標準程式庫工具（第 9 章）、`numpy` 或 `numexpr` 等技術（第 6 章）、Cython（第 7 章），或分散式計算模型（第 10 章）。

Python 3.2 也對 GIL 進行了重要的改寫（*https://oreil.ly/W2ikf*），讓系統靈活許多，減輕許多圍繞著單執行緒系統性能的煩惱。雖然 GIL 仍然會鎖定 Python，讓它一次只執行一個指令，但現在可以更好地在這些指令之間切換，讓工作開銷更低。

記憶單元

電腦的**記憶單元**是用來儲存位元的。這些位元可能代表程式中的變數，或代表圖像中的像素。因此，記憶單元的抽象也適用於主機板的暫存器以及你的 RAM 和硬碟。所有類型的記憶體單元之間最大的差別是它們讀 / 寫資料的速度。更複雜的是，讀 / 寫速度與資料被讀取的方式有很大的關係。

例如，大部分的記憶單元讀取一個大區塊資料的性能都遠比讀取許多小區塊更好（這稱為**連續讀取 vs. 隨機資料**）。如果你將這些記憶單元裡面的資料當成一本書的書頁，這意味著大部分的記憶單元在逐頁翻閱書本時的讀 / 寫速度，都比經常從某頁隨機翻到另一頁時更快。雖然所有記憶單元都有這種情況，但這種情況對各種記憶單元的影響程度有很大的不同。

除了讀 / 寫速度之外，記憶單元也有**延遲時間**（*latency*），也就是設備尋找想要使用的資料所花費的時間。對旋轉硬碟而言，這個延遲時間可能很長，因為磁碟需要物理性地加快轉速，並將讀取頭移到正確的位置。另一方面，對 RAM 而言，這個延遲時間可能很短，因為一切都是固定狀態的。以下簡要說明在標準工作站中常見的各種記憶單元，按讀 / 寫速度排序[2]：

旋轉硬碟

即使在電腦關機時，也可以進行持久保存的長期儲存設備。讀 / 寫速度通常很慢，因為它必須在物理上旋轉和移動磁碟。因為採取隨機存取模式，所以性能不彰，但有很大的容量（10 TB）。

固態硬碟

類似旋轉硬碟，有較快的讀 / 寫速度，但容量較小（1 TB）。

RAM

用來儲存應用程式碼與資料（例如任何變數）。具備快速讀 / 寫特性，在隨機存取模式下有很好的性能，但通常容量有限（64 GB）。

L1/L2 快取

極快的讀 / 寫速度。送往 CPU 的資料**必須**經過它們。容量非常小（幾 MB）。

圖 1-2 用目前的消費硬體的特性來展示這幾種記憶單元的差異。

2　這一節的速度來自 *https://oreil.ly/pToi7*。

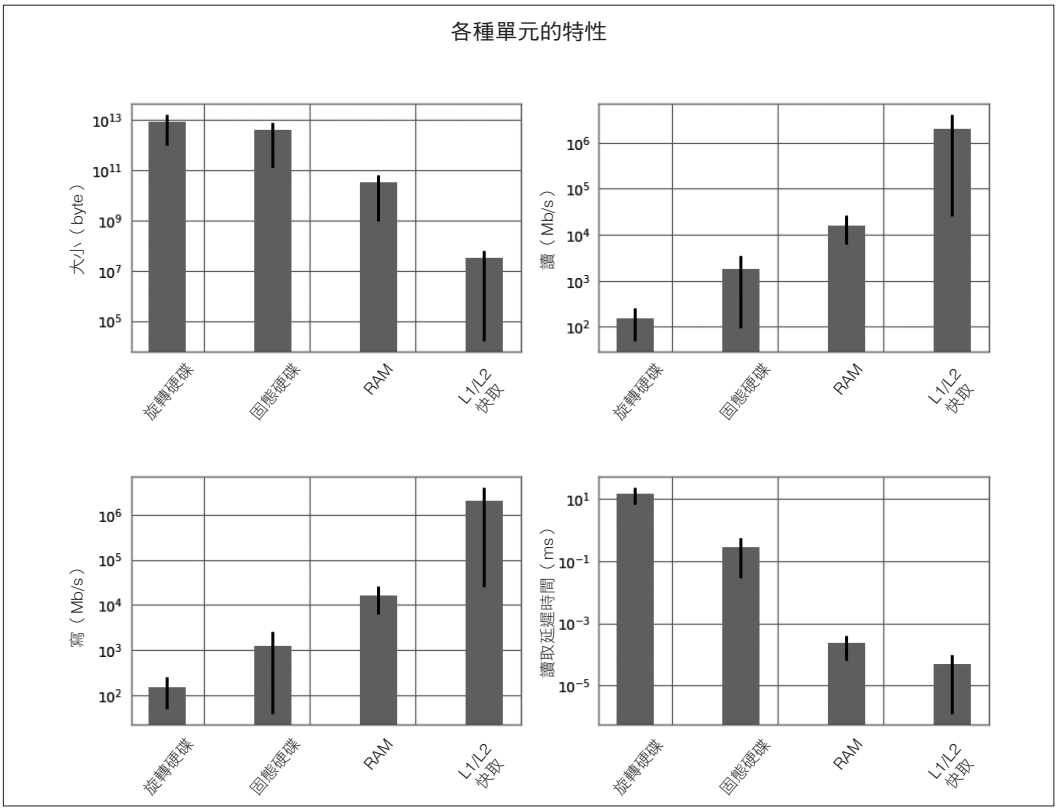

圖 1-2　不同記憶單元的特性值（2014 年 2 月的數據）

圖中有一個很明顯的趨勢是讀寫速度與容量成反比——當我們試著提高速度時，容量就會降低。因此，許多系統都採取記憶體分層法：最初，將完整的資料放在硬碟內，將它的一部分移至 RAM，再將小很多的子集合移到 L1/L2 快取。這種分層法可讓程式根據存取速度的需求在不同的地方保存記憶。當我們試著優化程式的記憶模式時，只是在優化要將哪些資料放在哪裡、如何布局（為了提升循序讀取的數量）、以及它在不同位置之間移動了多少次。此外，非同步 I/O 與搶占式快取之類的方法可以確保資料一定在需要的位置，而不會浪費計算時間——在執行其他計算時，這些程序大部分都可以獨立執行！

通訊層

最後，我們來看這些基本元素如何互相溝通。溝通模式有很多種，但它們都是所謂的**匯流排**（*bus*）的變體。

例如，**前端匯流排**（*frontside bus*）是 RAM 與 L1/L2 快取間的連結。它會將準備讓處理器轉換的資料移至預備處，準備進行計算，並且將完成的計算移出。此外還有其他的匯流排，例如外部匯流排，它是從硬體設備（例如硬碟與網路卡）到 CPU 與系統記憶體的主要路徑。外部匯流排通常比前端匯流排更慢。

事實上，L1/L2 快取的許多好處都要歸功於更快的匯流排。因為我們能夠在緩慢的匯流排（從 RAM 到快取）以大區塊的形式排列計算所需的資料，再以非常快的速度，以快取線（從快取到 CPU）提供它們，所以 CPU 無需等待太長的時間就可以進行更多的計算。

同樣地，使用 GPU 的許多缺點也來自連接它的匯流排：因為 GPU 通常是周邊設備，透過 PCI 匯流排來溝通，這種匯流排比前端匯流排慢非常多。因此，讓資料流入與流出 GPU 是相當費力的操作。異質計算（heterogeneous computing，也就是在前端匯流排接上 CPU 與 GPU 的計算模組）旨在降低資料傳輸成本，並且讓 GPU 計算成為比較可行的選項，即使在必須傳輸大量資料的情況下。

除了電腦內部的通訊區塊之外，網路也可以視為另一種通訊區塊。但是這種區塊比上述的區塊更柔韌（pliable）；網路設備可以接到記憶設備，例如網路附加儲存（network attached storage，NAS）設備或其他計算區塊，例如叢集內的計算節點。但是，網路通訊通常比上述的其他通訊類型慢非常多。前端匯流排每秒可以傳輸數十 gigabits，但網路只限於數十 megabits 的數量級。

顯然地，匯流排的主要屬性是它的速度：它可以在特定的時間內移動多少資料。這個屬性包含兩個數據：一次傳輸可以移動多少資料（匯流排寬度），以及匯流排每秒可以進行幾次傳輸（匯流排頻率）。注意，在一次傳輸之內移動的資料一定是循序的，它會將一塊資料讀出記憶體，並移到不同的位置。因此，匯流排的速度可以拆成這兩個數據，因為它們會分別影響計算的不同方面：因為寬度很大的匯流排可以在一次傳輸之內移動所有相關的資料，所以它對向量化的程式碼（或是需要循序讀取記憶體的任何程式碼）有益；另一方面，寬度很小，但是傳輸頻率很高的匯流排對必須從記憶體的隨機部分進行多次讀取的程式碼有利。有意思的是，電腦設計師可以透過主機板的物理布局改變這些屬性：當晶片被放在一起時，連接它們的物理接線比較短，所以可以加快傳輸速度。而且，接線的數量本身決定了匯流排的寬度（所以這個術語有實際的物理意義！）。

因為我們可以調整介面來為特定的應用提供正確的性能，所以介面有上百種類型就不足為奇了。圖 1-3 是常見的介面的位元率（bitrate）。注意，這與接線的延遲時間沒有任何關係，延遲時間代表回應資料請求所需的時間（不過延遲時間和電腦有很大的關係，它使用的介面有一些固有的基本限制）。

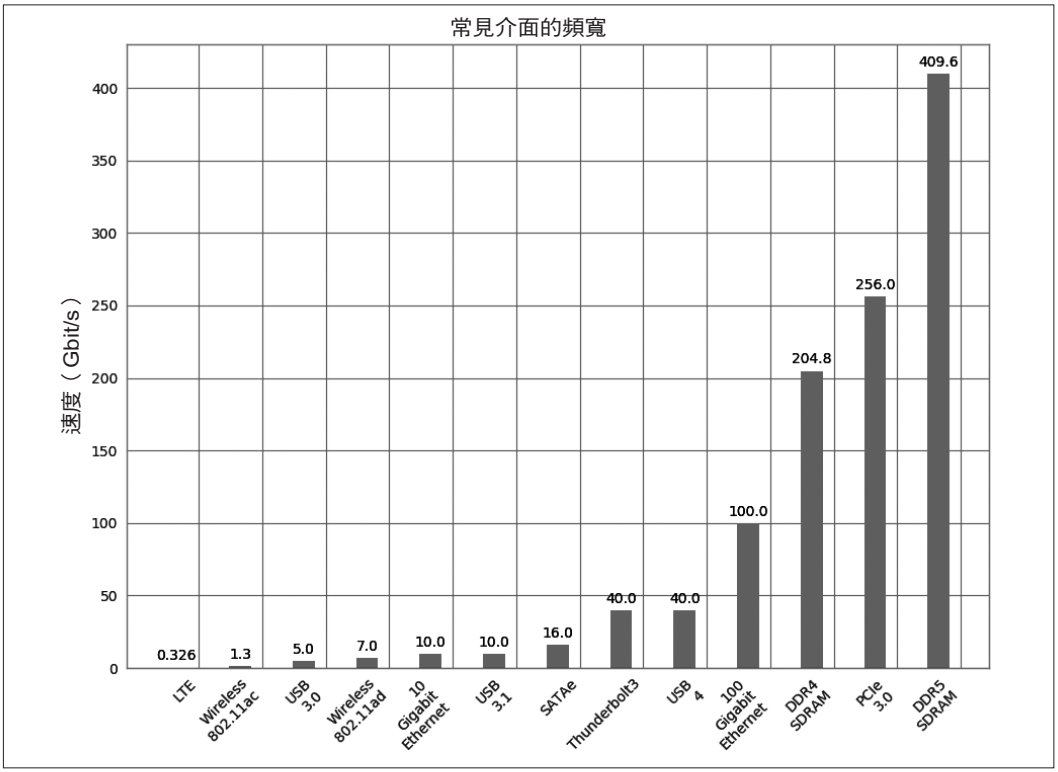

圖 1-3　各種常見介面的連接速度 [3]

將基本元素放在一起

了解電腦的基本元素不足以完全了解高性能編程的問題，這些元件的互動以及它們合作解決問題的方式也會帶來額外的複雜性。本節將探索一些玩具問題，說明理解的解決方案如何工作，以及 Python 如何處理它們。

3　此數據來自 *https://oreil.ly/7SC8d*。

警告：這一節可能會讓人氣餒，因為大部分的內容都暗指 Python 本質上無法處理性能問題。這不是事實，原因有二。第一，在所有這些「高性能計算元件」中，我們忽略了一個非常重要的因素：開發者，Python 的性能缺點可以用開發速度扳回。此外，本書將會介紹一些模組和原理，它們可以協助你相對輕鬆地緩解這裡提到的許多問題。結合這兩個層面，我們可以保持快速的 Python 開發思維，同時排除許多性能約束。

理想化的計算 vs. Python 虛擬機器

為了更加了解高性能編程的元素，我們來看一段簡單的程式，它的目的是檢查一個數字是不是質數：

```
import math

def check_prime(number):
    sqrt_number = math.sqrt(number)
    for i in range(2, int(sqrt_number) + 1):
        if (number / i).is_integer():
            return False
    return True

print(f"check_prime(10,000,000) = {check_prime(10_000_000)}")
# check_prime(10,000,000) = False
print(f"check_prime(10,000,019) = {check_prime(10_000_019)}")
# check_prime(10,000,019) = True
```

我們用抽象的計算模型來分析這段程式，再針對 Python 執行這段程式發生的事情進行比較。如同任何抽象，我們會忽略理想化的電腦與 Python 執行這段程式的方式之中的許多細微之處。但是，在解決問題之前，有一個很好的練習可以嘗試：想一下演算法的常見元素，以及讓計算元件一起找出解決方案的最佳方法可能是什麼。我們可以藉著了解這個理想的狀況，以及明白 Python 底層實際發生了什麼事，反覆地讓 Python 程式碼更接近最佳程式碼。

理想化計算

當程式開始時，我們在 RAM 儲存 `number` 的值。為了計算 `sqrt_number`，我們必須將 `number` 的值傳給 CPU。理想情況下，我們可以送一次值，它會被存放在 CPU 的 L1/L2 快取裡面，CPU 會進行計算，再將值傳回去給 RAM，將它儲存起來。這是理想化的情境，我們將「從 RAM 讀取 `number` 值」的次數降到最低，選擇從 L1/L2 快取讀取，因為這樣快很多。而且，藉著使用直接連接至 CPU 的 L1/L2 快取，我們也將透過前端匯流排傳遞的資料量最小化了。

將資料放在需要它的地方，並且盡量減少移動它的次數在進行優化時非常重要。「重資料（heavy data）」這個概念指的是移動資料所需的時間與勞力，它是應避免的東西。

至於程式中的迴圈，我們想要同時傳送 number 與一些 i 值給 CPU 來進行檢查，而不是一次傳送一個 i 值給 CPU。可以做到的原因是，CPU 不需要額外的時間成本就可以操作向量化，也就是說，它可以同時執行多個獨立的計算。所以我們想要將 number 傳至 CPU 快取，加上盡可能多的 i 值，只要快取可以保存即可。我們對著每一對 number/i 執行除法，檢查結果是不是整數，接下來，我們回傳一個訊號，指出是否有任何值是整數。如果有，函式結束。如果沒有，我們重複這項工作。如此一來，對於許多 i 值，我們只需要傳回一次結果，而不是透過緩慢的匯流排來傳遞每一個值。我們利用了 CPU 將計算向量化的功能，也就是在一次時脈週期之內，執行一個指令來處理多筆資料。

我用下面的程式來說明這種向量化的概念：

```
import math

def check_prime(number):
    sqrt_number = math.sqrt(number)
    numbers = range(2, int(sqrt_number)+1)
    for i in range(0, len(numbers), 5):
      # 下面這行不是有效的 Python 程式碼
        result = (number / numbers[i:(i + 5)]).is_integer()
        if any(result):
            return False
    return True
```

在此，我們設定了程序，讓除法與整數的檢查都是用整組的五個 i 值一次完成。如果我們正確地進行向量化，CPU 就可以用一個步驟執行這一行，而不是分別為每一個 i 進行計算。在理想情況下，any(result) 操作也會在 CPU 裡面發生，而不需要將結果傳回去 RAM。我們將在第 6 章更詳細地介紹向量化、它如何工作，以及它何時可以為你的程式帶來好處。

Python 的虛擬機器

Python 解譯器會努力試著將底層的計算元素抽象化。程式員不需要關心如何為陣列分配記憶體、如何安排記憶體，或是以什麼順序將它送給 CPU。這是 Python 的優點，因為它可讓你把注意力放在你想要實作的演算法上。但是，它也會帶來巨大的性能損失。

重點在於，我們要認識到，Python 的核心確實運行一組非常優化的指令。但是，訣竅是讓 Python 以正確的順序執行它們，以獲得更好的性能。例如，在接下來的例子中，我們很容易就可以看出來，search_fast 跑得比 search_slow 快，原因是它跳過未提前終止迴圈而導致的無謂計算，雖然這兩種做法的執行時間都是 O(n)。然而，當你處理衍生型態、特殊 Python 方法或第三方模組時，事情就更麻煩了。例如，你可以立即看出 search_unknown1 或是 search_unknown2 哪一個函式比較快嗎？

```
def search_fast(haystack, needle):
    for item in haystack:
        if item == needle:
            return True
    return False

def search_slow(haystack, needle):
    return_value = False
    for item in haystack:
        if item == needle:
            return_value = True
    return return_value

def search_unknown1(haystack, needle):
    return any((item == needle for item in haystack))

def search_unknown2(haystack, needle):
    return any([item == needle for item in haystack])
```

藉著分析來找出程式中緩慢的區域，並且尋找更高效的方式來進行同樣的計算，很像是在尋找這些無用的操作並移除它們；雖然最終結果是相同的，但計算次數與資料傳輸量將大幅減少。

這個抽象層會讓我們無法直接實現向量化。我們最初的質數程式會幫每一個 i 值執行一次迴圈迭代，而不是結合多次迭代。然而，從抽象向量化的例子中，我們可以看到它不是有效的 Python 程式，因為我們無法將一個浮點數除以一個串列。numpy 之類的外部程式庫可以藉著將數學運算向量化來協助處理這種情況。

此外，Python 的抽象機制，會讓我們難以藉著將接下來的計算所需的資料持續放在 L1/L2 快取裡面來進行優化。這有很多原因，首先，Python 的物件在記憶體裡面不是以最佳的方式安排的，原因是 Python 是一種資源回收（garbage-collected）語言，它會視需求自動配置與釋出記憶體。這些動作會造成記憶體碎片化，從而影響傳輸至 CPU 快取的動作。此外，我們根本無法直接在記憶體裡面改變資料結構的布局，這意味著在匯流排上面的單次傳輸可能不包含一次計算所需的所有資訊，即使匯流排寬度可以容納它們全部[4]。

第二，由於 Python 採用動態型態而且不會被編譯，導致更加根本性的問題。如同 C 程式員經年累月學來的教訓，編譯器通常比你聰明。當編譯器編譯靜態程式碼時，它可以用許多技巧來改變東西的布局方式，以及 CPU 執行某些指令的方式，從而優化它們。然而，Python 不會被編譯：更糟糕的是，它有動態型態，這意味著程式碼的功能可能會在執行期改變，所以很難用演算法來推斷任何優化機會。緩解這個問題的方法有很多種，最重要的是使用 Cython，它可讓 Python 程式碼被編譯，以及讓使用者建立「提示」，告訴編譯器程式碼的實際動態程度。

最後，如果你試著將這段程式碼平行化，上述的 GIL 可能會損害性能。舉例來說，假設我們修改程式碼，讓它使用多個 CPU 核心，因而各個核心都會取得從 2 到 `sqrtN` 的一段數字。每一個核心都可以為它的數字段落執行計算，當計算全部完成時，核心可以比較它們的計算結果。因為各個核心不知道是否找到解，所以我們無法提早終止迴圈，但我們可以降低各個核心執行的檢查次數（如果我們有 `M` 個核心，各個核心需要做 `sqrtN / M` 次檢查）。然而，因為 GIL，我們一次只能使用一個核心。這意味著我們運行的其實是與未平行化的版本一樣的程式碼，但我們無法提前終止了。要避免這個問題，我們可以將多執行緒換成多程序（使用 `multiprocessing` 模組），或使用 Cython 或跨語言函式（foreign function）。

那為什麼要使用 Python？

Python 具備高度的表達性，也容易學習，新程式員可以在很短的時間內完成很多工作。有許多 Python 程式庫包含以其他語言寫成的工具，來讓你可以輕鬆地呼叫其他系統；舉例來說，scikit-learn 機器學習系統包裝了 LIBLINEAR 與 LIBSVM（都是用 C 寫成的），而 `numpy` 程式庫包含 BLAS 與其他 C 和 Fortran 程式庫。因此，正確使用這些模組的 Python 程式可以跑得和 C 程式一樣快。

4　第 6 章會介紹如何取回這種控制權，並一路往下調整程式碼，直至記憶體使用模式。

Python 被稱為「內建電池（batteries included）」，因為它內建了許多重要的工具和穩定的程式庫，包括：

`unicode` 與 `bytes`

融入核心語言

`array`

高效地使用記憶體的基本型態陣列

`math`

基本數學運算，包含一些簡單的統計

`sqlite3`

以 SQL 檔案為主的流行引擎 SQLite3 的包裝

`collections`

各式各樣的物件，包括 deque、計數法與字典變體

`asyncio`

使用 async 與 await 語法來讓 I/O 綁定任務並行執行

在核心語言之外也有大量的程式庫，包括：

`numpy`

數值 Python 程式庫（用來處理任何矩陣相關工作的基本程式庫）

`scipy`

收集大量值得信賴的科學程式庫，它們通常包著德高望重的 C 與 Fortran 程式庫

`pandas`

用 `scipy` 與 `numpy` 來建構的資料分析程式庫，類似 R 的 data frames 或 Excel 試算表

scikit-learn

快速轉變為內定的機器學習程式庫，它是用 `scipy` 來建構的

`tornado`

可讓你輕鬆綁定來執行並行計算的程式庫

PyTorch 與 TensorFlow

Facebook 和 Google 提供的深度學習框架，強力支援 Python 與 GPU

`NLTK`、`SpaCy` 與 `Gensim`

深度支援 Python 的自然語言處理程式庫

資料庫綁定

可以和幾乎所有資料庫溝通，包括 Redis、MongoDB、HDF5 與 SQL

web 開發框架

可讓你高效建立網站的系統，例如 `aiohttp`、`django`、`pyramid`、`flask`、`tornado`

`OpenCV`

電腦視覺程式庫

API 程式庫

可讓你輕鬆訪問流行的 web API，例如 Google、Twitter 與 LinkedIn

你也可以使用各式各樣的託管環境與 shell 來滿足各種部署情境，包括：

- 標準版本，位於 *http://python.org*
- 用於簡單、輕量、可移植 Python 環境的 `pipenv`、`pyenv` 與 `virtualenv`。
- Docker，用來建立容易啟動與重現的開發或生產環境
- Anaconda 公司的 Anaconda，以科學為重點的環境
- Sage，一種類 Matlab 環境，包含整合開發環境（IDE）
- IPython，科學家和開發人員重度使用的互動式 Python shell
- Jupyter Notebook，使用瀏覽器的 IPython 擴展版本，被廣泛用來教學與展示

Python 有一項重要的優點是它可以讓你快速地建立原型。因為有各式各樣的支援程式庫可用，所以很容易測試想法是否可行，即使最初的實作可能極不穩定。

如果你想要讓算術程式跑得更快，你可以研究一下 `numpy`。如果你想要嘗試機器學習，可試試 scikit-learn。如果你想要清理和處理資料，`pandas` 是很好的選擇。

「雖然系統跑得更快了，但長遠來看，團隊的速度會不會反而變慢了？」一般來說，這是一個合理的問題。只要投入足夠的時間，我們一定可以從系統擠出更多性能，但是這樣可能會造成脆弱且倉促進行的優化，最終造成團隊的崩潰。

加入 Cython（見第 162 頁的「Cython」）可能會造成這種結果，Cython 是一種以編譯器為主的方法，可將 Python 程式碼註記為類 C 型態，以便使用 C 編譯器來編譯轉換後的程式碼。雖然它提升的速度令人印象深刻（通常可以用相對較少的勞力取得接近 C 的速度），但支援這段程式碼的代價也會增加。更明確地說，團隊可能很難支援這種新模組，因為團隊成員的編程能力必須達到一定的成熟度，才可以理解不使用可提升性能的 Python 虛擬機器時，會失去什麼和得到什麼。

如何寫出高性能的程式？

編寫高性能的程式只是讓成功的專案長期維持高性能的一個因素。比起提高解決方案的速度並且讓它更複雜，團隊的整體速度重要多了。團隊的整體速度有一些關鍵因素——好的結構、製作文件、除錯能力，以及共用的標準。

假設你有一個原型，你沒有對它進行徹底的測試，而且你的團隊也沒有檢查它。它看起來已經「夠好」了，並且被送至生產環境。因為你們從來沒有用結構化的方式來編寫它，所以它缺乏測試程式與文件。突然之間，有一段引發慣性（inertia-causing）的程式碼需要別人支援，而管理層通常無法量化團隊的成本。

由於這個解決方案難以維護，大家往往避之唯恐不及——它的架構沒有被重新整理過，也沒有可以協助團隊重構它的測試程式，沒有人喜歡碰到它，所以讓它維持運作的重責大任落在一個開發者身上。這可能會在壓力到來的時刻導致可怕的瓶頸，並且帶來重大的風險——如果那一位開發者離開專案怎麼辦？

這種開發形式通常會在管理層不了解難以維護的程式碼所導致的持續性慣性時出現。用長期的測試和文件來展示它可以協助團隊保持高效率，以及說服管理層分配時間來「清理」這段原型程式碼。

在研究環境中，經常有人用很爛的寫法建構許多 Jupyter Notebook，並且用它來試驗許多想法以及各種資料組。他們總是想要在之後的階段再「把它寫好」，但是那個之後的階段從不會發生。最終，雖然他們得到可運作的成果，卻缺少可以重現結果和測試它的基礎設施，且大家對成果缺乏信心。這種情況的風險因素同樣很高，而且大家不太信任結果。

有一種通用的方法有很好的效果：

讓它可以工作

先建立夠好的解決方案。先建構「拋棄式」的原型解決方案，用它設計更好的結構在第二版使用。先做一些規劃再開始寫程式絕對錯不了，否則，你會「花了整個下午寫程式，只為了節省一個小時的思考時間」。在某些領域，這種情況稱為「Measure twice, cut once」。

讓它正確

接下來，你要加入一些穩健的測試程式，並且為它們編寫文件以及明確的重現指令，讓其他的團隊成員可以駕馭它。

讓它更快

最後，你可以把注意力放在分析、編譯或平行化上面，並使用既有的測試套件來確認這個新的、更快速的解決方案依然可以按預期工作。

優秀的工作實踐法

我們有一些「必備」的東西，其中文件、優良的結構與測試程式都是關鍵要素。

專案等級的文件將會協助你維持一個簡潔的結構。它也可以在未來協助你和你的同事。如果你跳過這個部分，沒有人會感謝你（包括你自己）。在資料結構頂層將它寫成 *README* 檔是很好的起點，之後，你隨時可以視需求，將它擴展成 *docs* 資料夾。

請解釋專案的意圖、資料夾裡面有什麼、資料來自哪裡、哪些檔案至關重要，以及如何執行它們全部，包括如何執行測試。

Micha 也建議使用 Docker。頂層的 Dockerfile 可以讓將來的你知道，你需要從作業系統得到哪些程式庫來讓這個專案成功運行。它也可以排除在其他機器上執行這個程式，或將它部署至雲端環境的障礙。

加入 *tests/* 資料夾並加入一些單元測試。我們喜歡使用現代測試執行器 `pytest`，因為它的基礎是 Python 內建模組 `unittest`。你可以先寫幾個測試，再慢慢建構它們。然後使用 `coverage` 工具，它可以回報有幾行程式碼被測試程式覆蓋，有助於避免令人討厭的意外。

如果你繼承了舊的程式碼，而且它缺乏測試程式，預先加入一些測試有很高的價值。加入一些「整合測試」來檢查專案的整體流程，並確認執行某些輸入資料可取得特定的輸出結果，可以協助你在隨後進行修改時保持頭腦的清醒。

每當程式出現問題，就加入一個測試。被同一個問題困擾兩次是毫無價值的。

在每一個函式、類別與模組裡面用 docstring 來解釋程式碼一定可以幫助你。請說明函式可以**實現**什麼事情，可以的話，加入一個簡短的例子，來展示預期的輸出。如果你需要靈感，可參考 `numpy` 與 scikit-learn。

當你的程式碼變得太長（例如函式的長度超過螢幕畫面），請對它進行重構來讓它更短。較短的程式碼比較容易測試與支援。

當你在開發測試程式時，可以考慮接下來介紹的測試驅動開發法。當你確切地知道你需要開發什麼，並且手邊有可測試的例子時，這種方法將非常有效。

採取這種方法時，你會編寫測試、執行它們、看著它們失敗，再加入函式與必要的最簡邏輯來支持你寫好的測試。當測試都通過時，你就完成工作了。你將會發現，藉著事先釐清函式期望的輸入與輸出，函式的邏輯寫起來比較簡單。

當你無法事先定義測試時，有個問題會自然浮現：你真的了解函式需要做什麼事嗎？如果不了解，你可以用有效的方式正確編寫它嗎？如果你正處於創造過程，並且正在研究你還不了解的資料，這種方法就不太有效。

務必使用原始碼控制系統，當你不小心覆寫重要的程式時，你會慶幸自己決定使用它。養成經常 commit 的習慣（每天，甚至每 10 分鐘），並且每天將程式碼 push 至 repository。

堅持使用 `PEP8` 編碼標準。最好在 pre-commit 原始碼控制 hook 使用 `black`（武斷的程式碼格式化器），讓它幫你將程式碼改成標準格式。使用 `flake8` 來 lint 程式碼，以避免其他錯誤。

建立一個與作業系統隔開的環境可以讓你的工作更輕鬆。Ian 喜歡 Anaconda，而 Micha 喜歡 `pipenv` 和 Docker 的搭配。它們都是可行的解決方案，而且顯然比使用作業系統的全域 Python 環境更好！

切記，自動化是你的好幫手。手工的事情越少代表錯誤的機會越少。將組建系統自動化、使用自動化的測試套件執行器來進行持續整合，以及將部署系統自動化，可以將繁瑣且容易出錯的工作轉變成任何人都可以運行與支援的標準流程。

最後，切記，易讀性遠比賣弄小聰明重要。簡短卻複雜且難讀的程式對你和同事而言都難以維護，所以大家都很怕遇到這種程式碼。相反地，你應該編寫較長、較易讀的函式，為它編寫實用的文件來說明它會回傳什麼，並且使用測試程式來確認它確實如你預期地做事。

關於 Notebook 的優良實踐法

如果你正使用 Jupyter Notebook，它們很適合用來進行視覺化溝通，但它們會讓你變懶。如果你發現自己在 Notebook 中寫了很長的函式，請將它們做成 Python 模組，並加入測試程式。

考慮將 IPython 或 QTConsole 裡面的程式碼做成原型；將 Notebook 裡面的程式寫成函式，再將它們從 Notebook 提升為模組，輔以測試程式。最後，如果進行封裝和資料隱藏有幫助，將程式包在類別裡面。

盡量在整個 Notebook 裡面使用 `assert` 陳述式來確定函式的行為是否符合預期。除非你將函式重構為獨立的模組，否則在 Notebook 內測試程式並不容易，使用 `assert` 是進行驗證的簡便手段。在你將程式碼提取為模組，並且為它編寫單元測試之前，你都不應該完全相信它。

不要使用 `assert` 陳述式來檢查程式碼裡面的資料，雖然用這種方法來斷言某個條件是否符合很簡單，但這不是典型的 Python。為了讓別的開發人員更容易閱讀你的程式碼，你可以檢查你預期的資料狀態，當檢查失敗時，發出正確的例外。當函式遇到意外的值時，`ValueError` 是常見的例外。Bulwark 程式庫（*https://oreil.ly/c6QbY*）是一種專注於 Pandas 的測試框架，可檢查資料是否符合特定限制。

或許你也可以在 Notebook 的結尾加入一些健全性檢查——檢查邏輯是否正確，並且在產生預期的結果時，印出並展示它。當你在 6 個月之後回來閱讀這段程式時，你會感謝自己讓它如此易讀，而且從頭到尾都是正確的！

Notebook 不容易和原始碼控制系統共享程式碼。nbdime（*https://oreil.ly/PfR-H*）是一組持續成長的新工具，可讓你對你的 Notebook 進行差異比對（diff）。它十分方便，而且可協助你和同事合作。

找回工作的樂趣

生活有時很複雜。在筆者編寫此書第一版的五年中，我們和親友共同經歷了許多生活情境，包括抑鬱症、癌症、搬家、成功的業務退出、失敗，以及改變職業方向。這些外在事件難免影響任何人的工作與生活觀。

你一定要在新活動中繼續尋找樂趣。一旦你開始探索，就一定會遇到一些有趣的細節或需求。你可能會問「他們為什麼做出那個決定？」以及「如何採取不同的做法？」，突然之間，你就會開始自問如何改變或改善事物了。把值得慶祝的事情記錄下來。人們很容易忘記自己的成就，陷入日常的工作中。不停地追逐目標會讓人筋疲力盡，忘記自己獲得多少進步。

建議你列出值得慶祝的事情，並寫下你是怎麼慶祝它們的。Ian 有一份這種清單——每當他更新清單時，他就會驚喜地發現去年發生多少很酷的事情（而且如果沒有這份清單可能會忘記！）。在清單內，不要只寫下工作的里程碑，你可以加入興趣和運動，並慶祝你取得的成就。Micha 會優先考慮他的個人生活，花幾天的時間遠離電腦，去進行非技術性的專案。持續發展技能非常重要，但沒必要耗盡精力！

寫程式，尤其是專注於性能時，我們必須保持好奇心，以及鑽研技術細節的意願。遺憾的是，當你精疲力竭時，這種好奇心會率先消失，所以花點時間，確保你享受這趟旅程，並且保持快樂與好奇的心態。