前言

和典型的婚紗不同，它很優雅 -- 在技術的用語中，「優雅」就像是一個只有短短幾行的演算法程式，卻能做出驚人的產出。

—Graeme Simsion，蘿西效應

歡迎你閱讀本書，書裡大部分的內容都是在講書名中的“SciPy”，所以就在這兒花一點時間談一下，有關書名中的 Elegant（優雅）部分（譯註：原文書名為 Elegant SciPy）。已經有很多手冊、導覽和文件網站是介紹 SciPy 函式庫的了，所以本書要更往前再進一步，不僅僅是教你如何寫能用的程式碼，而是要啟發你寫出超讚的程式。

在**蘿西效應**這本書中（詼諧愛情故事，如果有興趣，可以從**蘿西效應**的前傳**蘿西計畫**開始閱讀這個愛情故事（*https://en.wikipedia.org/wiki/The_Rosie_Project*）），Graeme Simsion 將 Elegant 的字義作了兩次不同的解讀，原來這個字的意思是視覺上很簡潔、有型和優美，就像第一代的 iPhone。而 Graeme Simsion 書中主角 Don Tillman，用電腦演算法來**定義** Elegant 這個字，也就是讀或寫出 Elegant 的程式碼時，沐浴在它散發的美麗優雅光芒下，感覺平靜。

一段好的程式碼，讓人感覺什麼都對了，當你看著該段程式碼時，會感覺到它的意圖明顯，通常很簡潔（但又不會過於簡潔讓人看不懂），執行時也很有**效率**。對於作者而言，看著一段 Elegant 的程式碼所帶來的快樂，是從程式碼背後的學問而來，這些學問可以在日後撰寫程式碰到問題時，帶來新的啟發。

諷刺的是，這些新啟發往往引誘我們賣弄小聰明，去寫出晦澀難懂的程式碼。PEP8（Python 設計文件）以及 PEP20（Python 精神守則）提醒著我們“讀一段程式碼的機會，比寫一段程式來得多”，所以“易讀才是重點”。

有好的抽像結構與明智使用的函式，才能造就一段簡潔優雅的程式碼，光是將一堆亂七八糟的函式集結起來是無法達成的。面對簡潔優雅的程式碼，一開始可能要花一兩分鐘想一下，最終想通時，你會有“阿 - 哈！就是這樣”的反應。加上清楚的變數與函式名稱，以及謹慎專業的寫下註解，可以輔助閱讀者理解程式碼，一旦理解了程式碼的各種構成後，應該可以感受到它的正確性不言而喻。

在紐約時報（*New York Times*），一位名為 J. Bradford Hipps 軟體工程師最近發表了一篇“to write better code,[one should] read Virginia Woolf”（譯註：想要寫好程式碼，就要讀維吉尼亞 . 吳爾芙）的文章（*http://nyti.ms/2sEOOwC*）：

> 實際上，軟體工程師的創造力比邏輯能力好。
>
> 面對程式編輯器的開發者，如同面對空白頁的作家一樣。[...] 他們同樣焦慮著，是要選擇“一直以來都是這麼做”，或是打破傳通的慾望。當程式碼模組完成或是文章完成後，衡量他們產出品質的標準也一樣：包括優雅、簡潔、一致性，甚至是不對稱的美麗。

本書的重點也在於此。

好了，現在關於 Elegant 的部分已經講完了，讓我們回到“SciPy”吧！

在不同的語境下，“SciPy”這個名詞可代表一個軟體函式庫、一個生態圈或是社群。讓 SciPy 這麼讚的主要原因，就是因為它有傑出的線上文件（*https://docs.scipy.org*）和教學（*http://www.scipy-lectures.org*），再出一本普通參考書已經沒有太大意義，所以本書的目標就是要用 SciPy 產生最好的程式碼。

本書的程式碼，都是從 NumPy、SciPy 或相關函式庫中，選取聰明又優雅程式碼。初學讀者可學到如何將這些函式庫應用在真實世界問題，我們也會用真實的科學數據來作為範例。

我們也希望這本書如同 SciPy 本身一樣，背後由社群驅動，所以我們從 Python 的生態圈裡取出具優雅原則程式碼，來當作我們的範例。

本書讀者

本書的目的是要讓你的 Python 能力晉級到全新境界，藉由範例中優良的程式碼來學習 Python。

我們設定讀者至少見過 Python，並知道變數、函式、迴圈或是用過一點 NumPy，甚至曾經用 Python 的進階教材，如 *Fluent Python*。如果以上描述與你不符，則在進入本書之前，請先閱讀一些初階的 Python 教學，例如 Software Carpentry（*http://software-carpentry.org*）。

也許你還不知道“SciPy stack”到底是一個函式庫或是 International House of Pancakes（譯註：IHOP，美國的連鎖餐聽）的一道點心，而且你也不知道最佳做法（best practice）是什麼。或是你是一位科研人員，已經讀過一些線上 Python 教學，也從其它的研究室或是前輩那取得了一些分析腳本，也試過亂搞一下這些腳本了。此時你可能在學習 Python 的過程之中感覺孤立無援，但事情並不是你想的那樣。

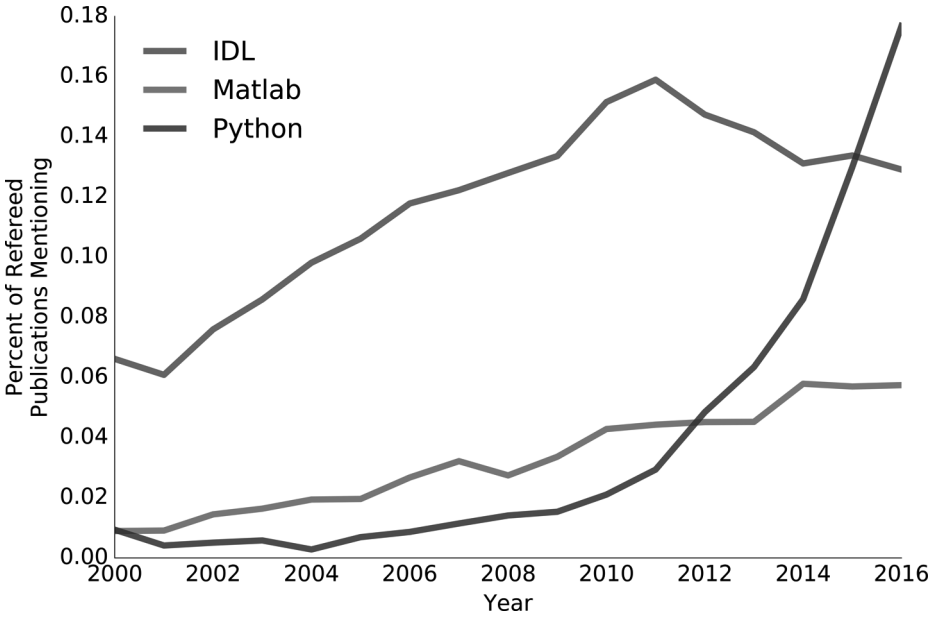
在我們的內容中，會教你如何在網路上查到你要的資料，也會告訴你郵件討論串、repository 及會遇見學者前輩的研討會在何處。

這是一本讓你讀一次，日後需要找靈感時會一直回來看的一本書（也許是回來找某段優雅的程式碼）。

為何選用 SciPy ？

在 Python 科研生態系中主流函式庫為 NumPy 和 SciPy，SciPy 這個軟體函式庫內含研究資料的函式，例如統計、訊號處理、影像處理及函數最佳化。SciPy 是架構在 NumPy 之上，NumPy 是 Python 的數值陣列計算函式庫。由於有了 NumPy 和 SciPy，整個應用和函式庫的生態系在過去幾年發展迅速，含括多種學科如：天文學、生物學、氣象學、氣候科學、材料學及其它。

這種增長沒有消退的跡象，在 2014 年 Thomas Robitaille 和 Chris Beaumont 說（*http://bit.ly/2sF5dRM*）Python 在天文學中使用率增加。而以下是我們更新（*http://bit.ly/2sF5i82*）他們的圖形至 2016 下半年：



顯然，SciPy 和相關的函式庫將會主導接下來幾年的科學研究資料分析。

Software Carpentry（*http://softwarecarpentry.org*）是另外一個例子，這個組織專門教科研要用的電腦技巧，大多數都使用 Python，目前需求仍然大過於供給。

什麼是 SciPy 生態圈？

> *SciPy*（唸作 *Sign Pie*）是一個以 *Python* 為基礎的開源軟體，應用於數學、科學和工程的生態圈。
>
> —*http://www.scipy.org*

SciPy 生態圈指的就是各種 Python 套件。在本書裡，我們會用到數個其中主要成員：

- **NumPy**（*http://www.numpy.org*）是 Python 中的科研基礎函式庫，它提供了有效率的陣列運算以及廣泛支援各種數字計算，包含線性代數、隨機數、傅利葉轉換等。NumPy 的殺手級應用是“N 維度陣列”或 ndarry，這些資料結構能有效率的儲存數值並定義任意維度的陣列（隨後會有更多介紹）。

- **SciPy**（*http://www.scipy.org/scipylib/index.html*）函式庫，是一群有效率的數值演算法的集合，應用在訊號處理、積分、最佳化和統計領域。SciPy 將這些演算法包裝成對使用者比較友善的介面。
- **Matplotlib**（*http://matplotlib.org*）是用來描繪 2D 圖（以及基礎的 3D 圖）的強效工具，它的名稱由來是因為其語法受 Matlab 啟發。。
- **IPython**（*https://ipython.org*）是一個 Python 的互動介面，可以讓你快速的測試資料。
- **Jupyter**（*http://jupyter.org*）是一個運作在你瀏覽器中的筆記本，讓你可以建立含有程式碼、文字和數學運算式，以及互動 widget[1] 的豐富文件格式文件。事實上，本書的內容文字就被轉換到 Jupyter 筆記本中，執行後產出本書（這樣一來我們才知道所有的範例程式能正確運作）。Jupyter 是由一個 IPython 擴展開始發展的，現在已經可以支援多種程式語言、包含 Cython、Julia、R、Octave、Bash、Pearl 和 Ruby。
- **pandas**（*http://pandas.pydata.org*）提供快速、欄為單位的資料結構在一個容易使用的套件中。它適合搭配標籤資料集合使用，例如：關連式資料庫的資料表、管理時間序列資料和滑動視窗。pandas 裡也有一些好用的資料分析工具，用來分析清理、群聚或描圖。
- **scikit-learn**（*http://scikit-learn.org*）為機器學習提供一個統一的介面。
- **scikit-image**（*http://scikit-image.org*）提供一個可整合其它的 SciPy 生態圈的影像分析工具。

SciPy 裡還有其它的 Python 套件，我們在書本也會看到一些。雖然目書重點放在 NumPy 和 SciPy 上，但許多關聯套件，是造就 Python 在科學研究界強大影響力的原因。

Python 2 到 Python3 的大變化

在你與 Python 相伴的旅程之中，你可能已經聽過一些人爭執著哪一版 Python 比較好。你可能會想不通，通常最新的版本不就是最好的嗎？（有雷劇透：最新的的確最好）

在 2008 年底，Python 核心開發者發布了 Python 3，主要更新是支援較好的 Unicode 處理、型態一致性和串流資料處理。但情況就像 Douglas Adams 嘲弄著宇宙的生成[2] “這件

1 Fernando Perez, “‘Literate computing’ and computational reproducibility: IPython in the age of data-driven journalism” (*http://bit.ly/2sFdfdl*) (blog post), April 19, 2013.

2 Douglas Adams, *The Hitchhiker’s Guide to the Galaxy* (London: Pan Books, 1979).

事已惹惱了一票人，而且被認為不是明智的行為。”，這樣的情況肇因於 Python 2.6 或 2.7 程式碼無法不加以修改直接轉換成 Python 3.0（雖然要改的地方也不是特別多）。

持續前進與向後相容之間，永遠都會有一點難以兩難。特別是底層的 C API 而言，Python 核心團隊覺得要有一些決斷才能擺脫過去的桎梏，著手將這個語言推向 21 世紀（Python 1.0 是 1994 年發表，已經超過 20 年了，在科技世界裡 20 年差不多是一輩子了）。

以下是 Python 3 進階版的其中一項變化：

```
print "Hello World!"  # Python 2 print statement
print("Hello World!") # Python 3 print function
```

只是加個括號幹嘛大驚小怪！呃，也是啦。不過萬你一想要把訊息改為輸出到其它的**串流輸出介面**呢？例如習慣上是將錯誤訊息指定送到**標準錯誤**（*standard error*）輸出。

```
print >>sys.stderr, "fatal error"  # Python 2
print("fatal error", file=sys.stderr)  # Python 3
```

改版以後看起來清楚多了，Python 2 版本的寫法令人困擾。

別外一個 Python 3 進階版變更是整數除法，改為人類習慣使用的除法形式。（註 >>> 表示我們是在 Python 的互動介面 shell 中下命令）

```
# Python 2
>>> 5 / 2
2
# Python 3
>>> 5 / 2
2.5
```

當 Python 3.5 在 2015 年發表了新的**矩陣乘法運算子**（*matrix multiplication*）@時，我們超興奮在第 5 章和第 6 章有這個運算子的使用範例！

Python 3 最大的變革應該就是支援 Unicode 吧！ Unicode 是一種編碼文字，讓使用者不限於只能使用英文字母，而是可以使用全世界的字母，在 Python 2 中使用 Unicode 需要另行定義，如：

```
beta = u"β"
```

但是在 Python 3，可以在任意地方使用 Unicode：

```
β = 0.5
print(2 * β)

1.0
```

Python 的核心團隊決定，世界所有語言的字母對 Python 而言都一樣尊貴。如果新的程式碼編寫者是來自非英語系國家時，這一點特別好用。不過考慮到共用性問題，我們基本上還是建議使用英文字母進行程式碼撰寫，但這個 Unicode 的支援，在 Jupyter 筆記本這種數學導向的工具裡還是特別好用。

在 IPython 終端或 Jupyter 筆記本中，若輸入一個 LaTeX 符號名稱，後面再接一個 TAB 鍵，它就會變成 Unicode，例如：`\beta<TAB>` 會變成 β。

Python 3 的更新使得部分既存 2.x 程式碼無法執行，有些變得執行速度比以前緩慢。若不考慮這些相容性問題的話，由於大多數的已知問題在 Python 3 已獲修正，所以我們建議所有使用者儘快昇級到 Python 3（Python 2.x 現在已進入維護階段，這個維護階段會到 2020 年為止），我們在本書裡也會使用到很多 Python 3 的新功能。

本書使用的版本是 **Python 3.6**。

關於更多昇級問題，可以閱讀 Ed Schofield 的 Python-Future 以及 Nick Coghlan 教學（*http://bit.ly/2sEZoUp*）。

SciPy 生態圈和社群

SciPy 是一個擁有許多功能的函式庫，和 NumPy 一起使用的話，它就變成 Python 的殺手級應用之一。它的功能已經是許多函式庫的基礎，你將會在本書內容中碰見那些函式庫。

上面提到的函式庫的作者與眾多使用者，在世界各地有很多活動和研討會，包括在美國 Austin 的年度 SciPy 研討會、EuroSciPy（歐洲）、SciPy India（印度）、PyData 以及其它。我們很建議你參加其中其中一個，見見那些在 Python 世界裡最好的科研軟體的作者們。如果你無到親臨現場，只是好奇想看看這些研討會是在做什麼，線上也有許多公開的演講連結（*https://www.youtube.com/user/EnthoughtMedia/playlists*）。

免費開源軟體（FOSS）

SciPy 社群支持開源軟體開發，幾乎所有 SciPy 函式庫的原始碼都可以讓任何人自由的讀取、編輯和使用。

如果你想要讓其它人使用你的程式碼，最好的方法就是讓它免費且公開，如果你使用了閉源程式碼軟體（相對於開源），但它的動作和你想要的有異，那就太不幸了。你只能寫郵件給開發者，並請他們加入你想要的新功能（最終可能也不會實現），或是自己重寫一套新的。如果程式碼是開源的，你就可以使用在本書學到的技巧輕易對它進行加入或修改功能。

而且，萬一你發現程式有問題，能夠直接取存原始碼進行修改，對開發者或使用者來說都輕鬆。即便你無法完全瞭解程式碼，也可以在問題分析上更深入一點，這能幫助開發者修正問題，對每個人來說都是學習的機會！

開放原始碼和開放科學

在科研界的程式寫作，以上所說的都很常見而且重要：科研用軟體通常都架構在前人的結果，或照自己的想法修改它。而且，由於科學研究持續推進發表，許多程式碼並沒有在發佈前經過詳盡的測試，多少隱含大小不一的問題。

另外一個讓程式碼開源的好理由是推廣研究的再利用，我們或許都讀過一些很酷的論文，下載程式碼並試著用它搭配我們自有的資料執行，不料發現我們的系統無法生成執行檔，或是無法執行，或發現有問題、功能不正常、結果不正確等情況。藉由將科研軟體開源，不僅是能提昇軟體的品質，也可以查看理論到底是如何實作的、作了哪些假設，或甚至就是內定值而已？開源能解決很多上述的問題，能讓其它科學家使用其它科學家的程式碼，促成新的合作機會或是加速科研的進度。

開放原始碼授權

如果你想讓別人使用你的程式碼，首先你**必須**對程式碼進行授權。如果你沒有做這個授權的動作，原則上它就不是開源的。即使你公開你的程式碼也一樣（例如：放在公眾可存取的 GitHub 上），如果沒有進行軟體的授權動作，別人就不能使用、編輯或是發布你的程式碼。

當在選擇授權種類時，首先你要決定想讓別人怎麼使用你的程式碼。你想要大家可以進行販售圖利？或是可銷售內含你程式碼的軟體？或是你想要限制只有免費散佈的軟體才能使用你的程式碼？

FOSS（Free and Open Source Software）授權可以大致分為兩種類：

- Premissive
- Copy-left

Premissive 授權，是指你讓任何人都可以用任何形式使用、編輯和發布你的程式碼，包括將你的程式碼放入商業軟體中。這個分類中，常見的有 MIT 和 BSD 兩種授權。SciPy 社群使用的是 New BSD License（又稱“Moditied BSD”或“3-Clause BSD”）。使用這授權表示會收到形形色色大眾對程式碼的修改，包括產業界公司和新創公司等。

Copy-left 授權，意指你讓任何人都可以用任何形式使用、編輯和發布你的程式碼，這種授權規定衍生的程式碼發布時，也要以 Copy-left 授權進行。在這個前提之下，Copy-left 授權產生使用者使用程式碼的限制。

最常見的 copy-left 授權是 GNU Public License，或稱 GPL。使用 Copy-left 授權最大的缺點是拒絕潛在使用者和營利事業，甚至是未來的你自己的使用！大幅度的降低會使用你程式碼的使用者總數，削減你軟體的成功。在科研界中可能會導致引用數變少。

如果在選擇授權種類上需要更多輔助資訊，你可以到 Choose a License 網站（*http://choosealicense.com*）。如果是在科研界，我推薦由 Washington 大學 Physical Sciences 的研究主任，也是全能的 SciPy 專家 Jake VanderPlas，他所發表的一篇部落文章“The Whys and Hows of Licensing Scientific Code”（*http://bit.ly/2sFj0HS*），我直接在下面引用 Jake 對於軟體授權切中要領的重點：

> …如果你只想從這篇文件中得到三個重點資訊，那就會是以下三點：
>
> 1. 永遠對你的程式碼進行授權動作，未授權的程式碼是閉鎖的程式碼，所以任何形式的開源授權都比閉鎖來得好（注意事項請接著看第 2 點）。
> 2. 永遠使用 GPL-compatible 的授權型態，GPL-compatible 授權讓你的程式碼有廣泛的相容性，型態包括 GPL、new BSD、MIT 及其它（注意事項請接著看第 3 點）。
> 3. 永遠使用 permissive 型態、BSD-style 授權。像 new BSD 或 MIT 這種 permissive 型態授權比 copyleft 型態如 GPL 或 LGPL 來得好。

本書中的所有程式碼都是 3-Clause BSD 授權，因為我們從別人取得的程式碼，大多都是 premissive 授權，或是其它類似授權（不一定是 BSD）。

至於你自己的程式碼，我們建議你遵守你的社群定義。在科研界的 Python 社群，通常是 3-Clause BSD，如果是 R 語言社群，就是 GPL 授權。

GitHub

關於發布程式碼時，應做開源授權這件事情已經說完了。希望可以有大量的人下載你的程式碼並使用它、修正錯誤，並加入新功能。但你要把程式碼放在哪，才能讓別人找到呢？問題修正以後及新增功能以後的程式碼要如何加回你的程式碼之中呢？你要怎麼持續追蹤這些問題和修正呢？你可以想像，如果沒有善加管理，事情很快就會失去控制了。

請到 GitHub。

GitHub（*https://github.com*）是一個托管、分享和開發程式碼的網站，它的基礎是 Git 版本控制軟體（*http://git-scm.com*）。對於學習如何使用 GitHub 有一些很好的資源，例如 Peter Bell 和 Brent Beer 的 *Introducing GitHub*。在 SciPy 生態圈裡大多數專案都托管在 GitHub 上，所以學習一下怎麼使用它是很值得的。

GitHub 對開源貢獻有巨大的影響，因為它允許使用者可以免費公開程式碼或是協同工作。不管是誰都可以建立複製一份程式碼（稱為 *fork*），並隨心所欲進行編輯。這些人最終把變更合併回原來程式碼的動作稱為 *pull request*。另外還有一些很好的功能像是問題管理和變更請求，或是管理誰能夠直接編輯你的程式碼。你甚至可以持續追蹤編緝、程式碼貢獻者，或其它有趣的東西。GitHub 的功能有一大堆，但我們在後面幾章會介紹其中一些，留下大部分讓你自己去探索。總歸來說，GitHub 已經在軟體開發界大眾化（圖 P-1），也降低了進入的門檻。

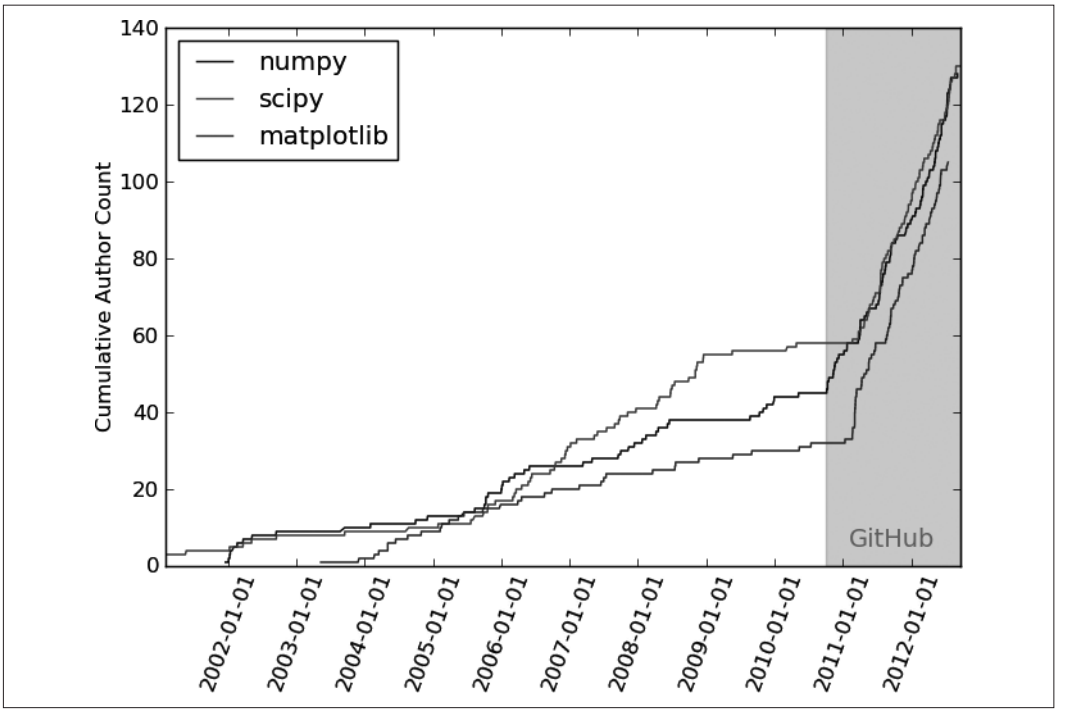


圖 P-1　GitHub 的影響（本圖取得作者 Jake VanderPlas 的同意後使用）

讓你自己在 SciPy 生態圈插旗

隨著你愈來愈熟悉 SciPy，並開始在你的研究中使用它，你可能會遇到某個套件缺乏你需要的功能，或你想要將某件事改良更有效率，或發現有問題。當你碰到這種情況時，就表示你該對 SciPy 生態圈作出貢獻的時候到了。

我們強烈的建議你試著做以下的動作，社群之所以存在，就是因為人們願意分享或改良既有程式碼，並且，如果每個人都貢獻一點點，集結起來力量就很大。但在這種大愛的貢獻之下，還是有一些實際的個人利益。藉由社群的力量，你可以變成更好的撰寫者，所有你所貢獻的程式碼，都會被別人檢查過，並給你一些回饋意見。而且，你會在過程中學到如何使用 Git 和 GitHub，它們是管理和分享你程式碼的有力工具。甚至可能在 SciPy 社群中取得廣大的科研人脈，或是令人驚喜的事業機會。

我想要你不只當自己只是 SciPy 使用者，請加入一個社群，並用你的力量促使科研界進步。

Py 的世界中充滿奇思妙想

萬一你擔心 SciPy 社群對菜鳥不友善，請記得社群是由一群像你的人或科學研究者所組成，通常具有很好的幽默感。

在 Python 的世界中，難免會看到一些類似蒙提·派森劇團（英語：Monty Python，也作 The Pythons，英國的一個超現實幽默表演團體）的幽默，像是 Airspeed Velocity（*http://spacetelescope.github.io/asv/using.html*）是用來測量你的軟體執行速度（之後會再講到），它的名稱是來自“*Monty Python and the Holy Grail*”裡的台詞“what is the airspeed velocity of an unladen swallow ？”。

另外一個有好玩名稱的套件叫“Sux”（譯註：是 sucks 的口語），功能是讓你在 Python 3 裡用 Python 2 的套件，名字是取“Six”的紐西蘭口音，而“Six”是讓你在 Python 2 裡用 Python 3 語法的套件。Sux 的語法可以舒緩你在 Python 3 時使用到只支援 Python 2 套件時的不爽心情：

```
import sux
p = sux.to_use('my_py2_package')（譯註：sux.to_use ==> sucks to use）
```

一般來說，Python 函式庫的命令都很歡樂，我期待你也可以享受於製造一些有趣的名稱！

協助

當我們 Google 想要做的事，或是查詢出現的錯誤訊息時，通常會看到 Stack Overflow（*http://stackoverflow.com/*）網站，它是一個優良的程式設計的問與答網站。如果你一下子找不到想找的東西，只要試著將你的搜尋關鍵字修改一下，看看是不是也有其它的人碰到類似的問題。

有時候，你可能真的是第一個碰到這個問題的人（特別可能在你使用一個全新的套件時發生），此時請不要放棄希望！之前我們說過，SciPy 社群是由一群散布網路各處友善的人士所組成，所以你的下一步是 Google“<library name> mailing list”，以找到郵件討論串來取得協助。函式庫作者和強者前輩會定期讀取這些郵件，而且也很歡迎新人加入討論。請注意，在張貼（post）訊息以前要先訂閱（*subscribe*）的規定，如果你沒有遵守，就表示會有某個人得手動檢查你的郵件是不是垃圾郵件，確認不是以後才能進行張貼，加入郵件討論串的步驟看起來有點煩人，不過我們還是高度推薦，因為這是學習的好地方！

第一章

優雅的 NumPy：科研界 Python 的基礎

[NumPy] 它無所不在，處處圍繞著我們，即使是現在，在這個房間裡。當你打開窗戶或是打開電視時都可以看到它的踪跡，上班 ... 去教堂 ... 繳稅時都可以感受到它。

—莫菲斯，駭客任務

本章會談到一些 SciPy 的統計及其它功能，著重在學習 NumPy 陣列，它是 Python 中用來作數值計算的資料結構。我們在接下來的內容中會看到 NumPy 的陣列是如何有效率地運算數值資料。

從本章開始到第二章，我們會用 The Cancer Genome Atlas（TCGA）專案的基因表現資料來預測皮膚癌病患的死亡率，並在過程中學習一些重要的 SciPy 觀念。在我們作死亡率預測之前，為了能在不同的個體和基因之間比較測量值，所以要先把基因表現（gene expression）資料做 RPKM 正規化（稍後會解釋什麼叫做基因表現）。

先從一小段程式範例開始介紹本章的主要概念。之後的每一章，我們都會以一段程式碼範例作為開頭，這些範例在 SciPy 生態圈的不同應用中，都堪稱優雅又強大的程式碼。在這一個範例中想強調的是 NumPy 的向量化和廣播規則，可用來有效率地操作資料陣列。

```
def rpkm(counts, lengths):
    """Calculate reads per kilobase transcript per million reads.

    RPKM = (10^9 * C) / (N * L)
```

```
    Where:
    C = Number of reads mapped to a gene
    N = Total mapped reads in the experiment
    L = Exon length in base pairs for a gene

    Parameters
    ----------
    counts: array, shape (N_genes, N_samples)
        RNAseq (or similar) count data where columns are individual samples
        and rows are genes.
    lengths: array, shape (N_genes,)
        Gene lengths in base pairs in the same order
        as the rows in counts.

    Returns
    -------
    normed : array, shape (N_genes, N_samples)
        The RPKM normalized counts matrix.
    """
    N = np.sum(counts, axis=0)  # sum each column to get total reads per sample
    L = lengths
    C = counts

    normed = 1e9 * C / (N[np.newaxis, :] * L[:, np.newaxis])

    return(normed)
```

這個範例展示了 NumPy 陣列的幾個用法，這些用法讓你的程式碼更簡潔：

- 陣列可以是一維，像串列一樣，但也可以是二維，像矩陣一樣，也可以是更高的維度。這個特性可以用來代表不同種類的數值資料，在我們的例子中所運算的是二維矩陣。
- 陣列可以和**座標軸**一起運作，在第一行，我們就藉指定 `axis=0` 來計算每個欄位的總合。
- 陣列容許將許多數值運算一起做。舉例來說，在函式結尾處，我們將基因計數的二維陣列（C）除以欄位加總的一維陣列（N），這就是廣播。待會就會有更多例子用來說明什麼是廣播。

在探索 NumPy 的強大功能之前，先看一下我們要使用的生物數據資料。

數據介紹：什麼是基因表現？

接下來要藉由分析基因表現來看 NumPy 和 SciPy 如何解決真實世界生物問題。我們將會用建構在 NumPy 之上的 pandas 函式庫讀取資料檔，然後將這些資料用 NumPy 陣列作計算。

分子生物學的中心法則（central dogma of molecular biology）（*https://en.wikipedia.org/wiki/Central_dogma_of_molecular_biology*）指出，一個細胞（或是一個有機體）運作的所需一切資訊，都被儲存於一個稱為**去氧核醣核酸**（*deoxyribonucleic acid*，DNA）的分子中。 這個分子有著重覆的骨幹，上面依次序排列**化學鹼基**（*base*）群（如圖 1-1）。鹼基群分為四種，分別為胞嘧啶（cytosine，C）、胸腺嘧啶（thymine，T）、腺嘌呤（adenine，A）以及鳥嘌呤（guanine，G）。

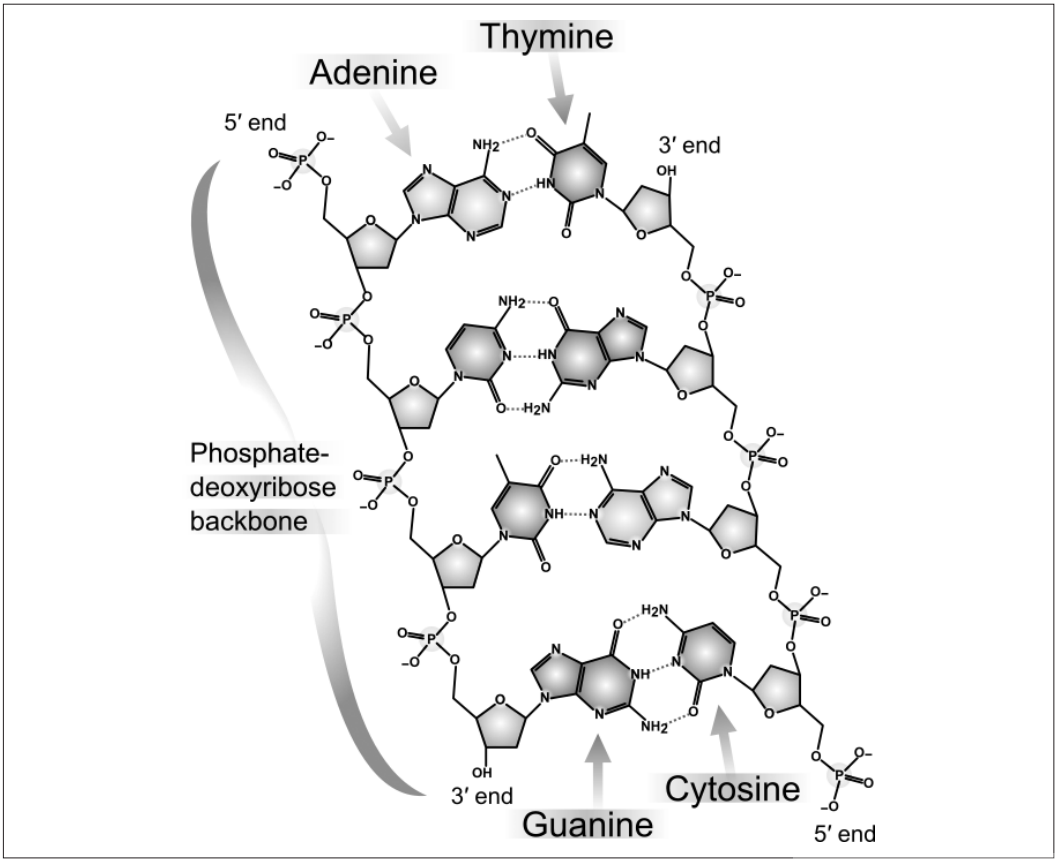


圖 1-1　DNA 的化學結構（此圖由 Madeleine Price Ball 提供，在 CC0 public domain license 授權下使用）

若想存取這些鹼基資訊，要先將 DNA **轉錄**為它的姐妹分子 mRNA（*messenger ribonucleic acid*）才行。mRNA 最終會被**轉譯**為蛋白質，蛋白質被細胞使用（如圖 1-2），DNA 中記錄如何生成蛋白質的區段（透過 mRNA）被稱為基因。

從一個特定基因可生成多少 mRNA，稱為此基因的表現。雖然在理想上我們應該測量的是生成的蛋白質量，但由於測量蛋白質的量比測量 mRNA 難上許多，還好基因生成 mRNA 的表現程度通常和它能生成多少蛋白質存在有相關關係[1]。所以，我們通常是測量生成 mRNA 的表現程度，並使用這個測量結果進行研究分析。接下來你會看到，我們不是用蛋白質，而是使用 mRNA 表現程度來預測生物差異。

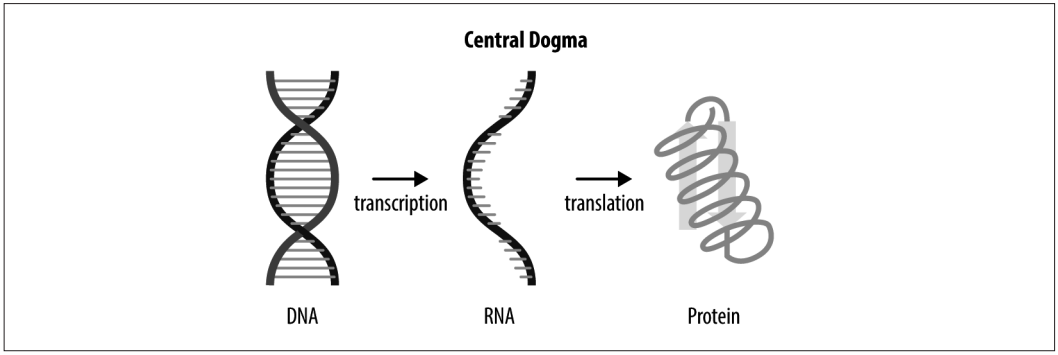


圖 1-2　分子生物學的中心法則

讓我們先瞭解一件事，就是你身體裡每個細胞裡的 DNA 都是一樣的，而不同種類細胞間的差異是由 DNA 轉錄為 RNA 時的**差異表現**（*differential expression*）造成：也就是說會有不同細胞的原因，是因為不同區段的 DNA 被處理，產生不同的分子（如圖 1-3）所造成。在本章和下一章裡會看到，藉由不同的基因表現就能區分出不同種類的癌症。

目前最先進的 mRNA 測量技術是 RNA 定序（RNA seq）。首先，要從生物組織樣本（例如病人的活體組織切片）取得 RNA，然後反轉錄回到 DNA（比較穩定），之後要用化學修改過的鹼基，而這些鹼基被併入 DNA 序列時會發亮，便可以被讀出。至今，高效的讀序機器也只能讀出簡短片段（通常大約是 100 個鹼基左右）。這樣的簡短片段被稱為“read”。我們測量數以百萬計的 read，用它們的鹼基排列順序來計算從每個基因來的 read 數量有多少（如圖 1-4），我們的分析就用這個計算數據結果來做。

1　Tobias Maier, Marc Güell, and Luis Serrano, “Correlation of mRNA and protein in complex biological samples” (*http://bit.ly/2sFtzLa*), FEBS Letters 583, no. 24 (2009).

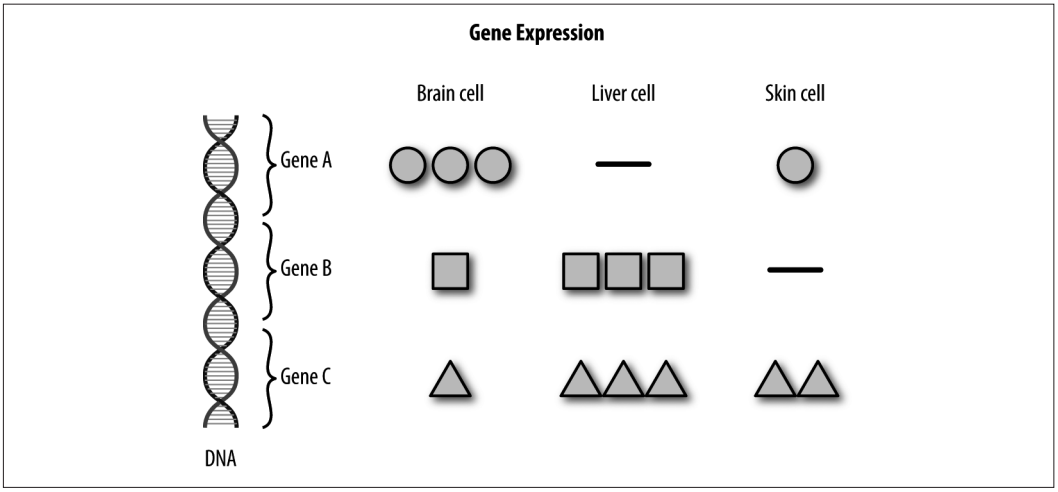


圖 1-3　基因表現

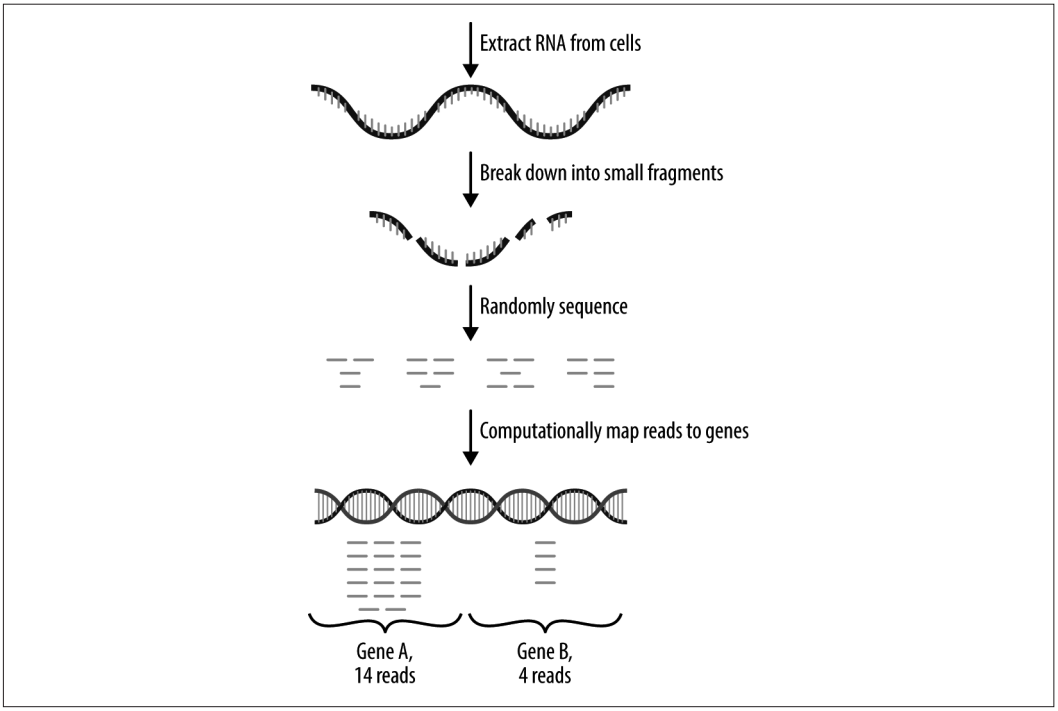


圖 1-4　RNA 定序（RNAseq）

表 1-1 是一個小型的基因表現計算數據範例

表 1-1　基因表現計算數據

	Cell type A	Cell type B
Gene 0	100	200
Gene 1	50	0
Gene 2	350	100

上面的計數表中，整數的意思是不同類細胞的不同基因共有多少 read。看到不同細胞對應的每種基因之數值差異了嗎？我們要用這些資訊來學習這兩種類細胞的差異。

用 Python 來表現這個數據的其中一個方法就是使用數個 list：

```
gene0 = [100, 200]
gene1 = [50, 0]
gene2 = [350, 100]
expression_data = [gene0, gene1, gene2]
```

上面不同細胞的基因表現被儲存在 Python 的整數 list 中，然後再把這些 list 整合成另一個 list（可以叫它 *meta-list*），可以用兩層的 list 索引來取出個別數據：

```
expression_data[2][0]
350
```

由於 Python 直譯器的關係，這些 list 數據指標的儲存很沒有效率。首先，由於 Python 的 list 儲存的單位是**物件**，所以上面的 gene2 並不是一個整數的 list，而是一個整數**指標**的 list（譯按：整數指標指向整數物件），這是沒有必要的浪費。另外，這也表示每個 list 以及其中每個整數，在你的電腦裡面是以隨機位置存放，現在的處理器在存取記憶體時，都是存取連續的區塊，所以分散在記憶體中很沒有效率。

這就是 NumPy 陣列要解決的問題。

NumPy N 維陣列

NumPy 的其中一種重要的資料型態就是 N 維陣列（稱 ndarray，或就叫它陣列）。N 維陣列是 SciPy 中許多厲害資料操作技巧的基礎，特別是接下來會談到的向量化和廣播，這些技巧讓我們可以寫出強大、簡潔的資料操作程式碼。

首先，讓我們瞭解一下 ndarray，這些陣列一定要是同質的：也就是所有在陣列裡的東西都要是一樣的型別，在我們的例子中，用的是整數。ndarray 被稱為 N 維陣列是因為它們可以有數個維度，一個維度在概念上意思就類似 Python 的 list。

```
import numpy as np

array1d = np.array([1, 2, 3, 4])
print(array1d)
print(type(array1d))

[1 2 3 4]
<class 'numpy.ndarray'>
```

陣列提供特定的屬性和方法，只要在陣列名稱後面接點就可以使用了。舉例來說，你可以取得陣列的 *shape*：

```
print(array1d.shape)

(4,)
```

結果如上，你可以看到輸出是一個數組架構，但裡面卻只有一個數字。你可能會好奇為什麼不能像 list 一樣用 len。其實是可以的，但如果遇到二維的情況 len 就不適用了。

以下才是我們用來表達表格 1-1 的方法：

```
array2d = np.array(expression_data)
print(array2d)
print(array2d.shape)
print(type(array2d))

[[100 200]
 [ 50   0]
 [350 100]]
(3, 2)
<class 'numpy.ndarray'>
```

現在你可以看到 shape 屬性彙整了資料陣列多個維度的 len。

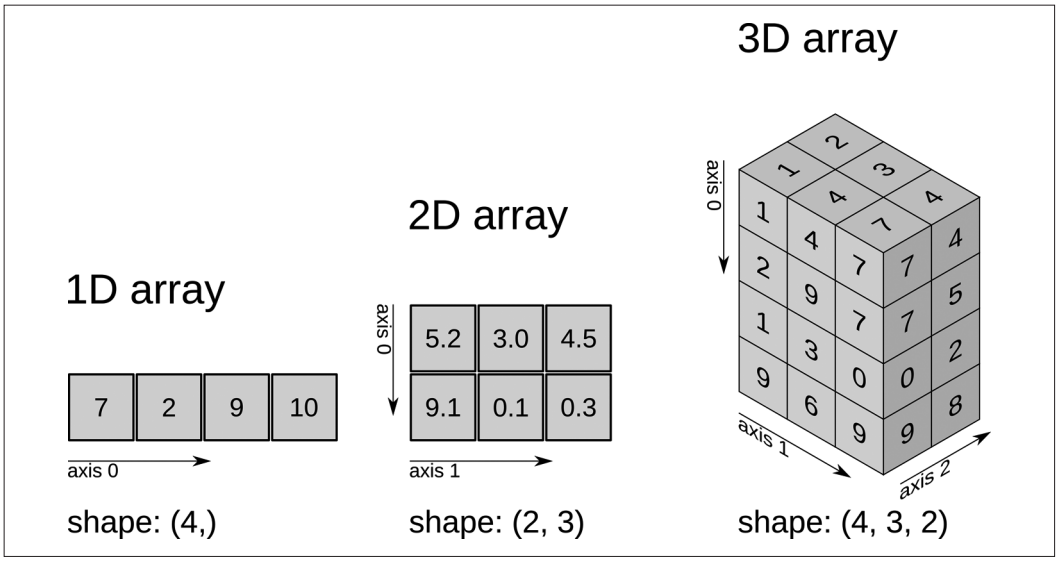


圖 1-5　以視覺化的方法呈現 NumPy 一維、二維和三維 ndarray

陣列還有另外一個叫 ndim 的屬性，意思是維度：

```
print(array2d.ndim)
```

```
2
```

當你開始用 NumPy 分析自己的數據後，就會更熟悉了。

NumPy 陣列還可以用在更高維度的資料，例如核磁共振造影（MRI）資料，它是三維空間的測量值，如果我們想儲存一段時間的 MRI 資料，就要用到四維陣列。

現在回到二維資料，後面的章節可能會用到更高維度的資料，並且會教你如何撰寫高維度資料的程式碼。

為何用 ndarray 不用 Python list ？

陣列運算比較快，因為它們支援低階語言 C 向量化運算。假設你有個 list，想把每個元素都乘上 5，一般 Python 的方法是寫一個迴圈，並遍歷 list 中所有元素，將每個元素個別乘 5。不過，如果你用陣列來做，元素乘以 5 的動作可以一次做完，由於在背後使用高度優化的 NumPy 函式庫，它迭代的速度非常地快。

```
import numpy as np

# Create an ndarray of integers in the range
# 0 up to (but not including) 1,000,000
array = np.arange(1e6)

# Convert it to a list
list_array = array.tolist()
```

讓我們實際用 IPython magic 中的 **timeit** 函式來比較一下，將每個元素乘 5 會花多少時間。首先是使用 list：

```
%timeit -n10 y = [val * 5 for val in list_array]

10 loops, average of 7: 102 ms +- 8.77 ms per loop (using standard deviation)
```

接著，使用 NumPy 內建的向量運算：

```
%timeit -n10 x = array * 5

10 loops, average of 7: 1.28 ms +- 206 µs per loop (using standard deviation)
```

結果快了超過 50 倍，而且程式碼更精簡！

使用陣列也省空間，在 Python 中每個在 list 中的元素都是物件，而且每個物件都還要給予合理的保留記憶體空間（或不合理？）。相反地，陣列中每個元素只用必要的空間。舉例來說，一個 64 位元整數型態的陣列，每個元素就是占 64 位元，加上額外一點點陣列運作用的空間，例如前面談到的 **shape** 屬性所占的空間，總合加起來還是比 Phython 的 list 所占用的空間小很多（如果你有興趣研究 Python 是如何作記憶體管理，可以參考 Jake VanderPlas 的部落格文章“Why Python Is Slow: Looking Under the Hood”（*http://bit.ly/2sFDbW8*））。

而且，在對陣列做計算時，你也可以直接**切出**需要的部分陣列值，而**不用做資料拷貝**的動作：

```
# Create an ndarray x
x = np.array([1, 2, 3], np.int32)
print(x)

[1 2 3]

# Create a "slice" of x
y = x[:2]
print(y)
```

```
[1 2]

# Set the first element of y to be 6
y[0] = 6
print(y)

[6 2]
```

請注意，雖然我們改變得是 y，但 x 也同時會改變。因為 y 是直接參照一樣的資料！

```
# Now the first element in x has changed to 6!
print(x)

[6 2 3]
```

所以使用陣列的參照時要特別小心，如果不想重疊到原始資料，則使用拷貝就可以了，要作拷貝也很簡單：

```
y = np.copy(x[:2])
```

向量化

稍早我們提到陣列處理的速度很快，NumPy 還有另外一個加速的利器就是**向量化**（*vectorization*）。向量化就是當你將陣列中每個元素都做同樣計算時，不需要使用 for 迴圈。這樣除了加速以外，另外一個好處就是讓程式碼更好讀。讓我們看一下範例：

```
x = np.array([1, 2, 3, 4])
print(x * 2)

[2 4 6 8]
```

上面建立了一個 x 陣列，包含 4 個值，我們把 x 中每個元素乘上單一數值 2。

```
y = np.array([0, 1, 2, 1])
print(x + y)

[1 3 5 5]
```

上面範例是把 x 的每個元素與 y 裡對應的元素相加，陣列 y 的 shape 與 x 相同。

以上範例既簡單又可以說明向量化，NumPy 讓以上範例執行速度快，遠超過用迴圈的方法（可用 IPython magic 的 `%timeit` 測量看看）。