前言

什麼是資料科學？

這是一本關於使用 Python 來從事資料科學工作的書。首先要面對的問題是：「何謂『資料科學』（*Data Science*）？」這是一個很難明確定義的詞，尤其是在這個詞已經被濫用的情況下。有些人認為這個名詞是多餘的（畢竟，哪有不包含資料的科學呢）、或認為這是個可以為自己履歷加料的流行語，好吸引那些特別喜歡科技的招聘人員的目光。

在我心裡的想法是，這些批評忽略了一些重要的事情。儘管資料科學表面上充滿炒作，但在跨領域技能的許多應用領域中，它可能是我們最好的標籤。這個「**跨領域**」的部分是關鍵：在我的心目中，資料科學目前最好的定義是 Dew Conway 所畫的 Data Science Venn Diagram，這張圖於 2010 年 9 月首次出現在其部落格中（參閱圖 P-1）。

雖然這些圖中某些交集的標籤內容並沒有那麼正式，但這張圖抓住了一些我認為人們提到「資料科學」時的所指的本質：它根本上是一個跨學科的主題。資料科學由三個不同但相互重疊的領域所組成：**統計學家**的技能，瞭解如何對越來越大的資料集進行建模與整合；**電腦科學家**的技能：用於設計以及使用高效的演算法進行儲存、處理與視覺化這些資料；以及**領域專家**：那些我們認為在某些傳統項目中有著「良好的」訓練，可以提出適合的問題以及得到對的答案的人。

有鑑於此，我建議讀者不要將資料科學視為一個需要從頭學習的全新領域知識，而是將其視為是可以應用在你目前專業領域的一套新技能。無論你是要報導選舉結果、預測股票收益、最佳化線上廣告的點擊率、辨識在顯微照片下的微生物、在天文領域尋找新的

星體、或是在任何領域中用到資料，本書的目標，就是提供你在自身專業領域中，提出新問題並找到解答的能力。

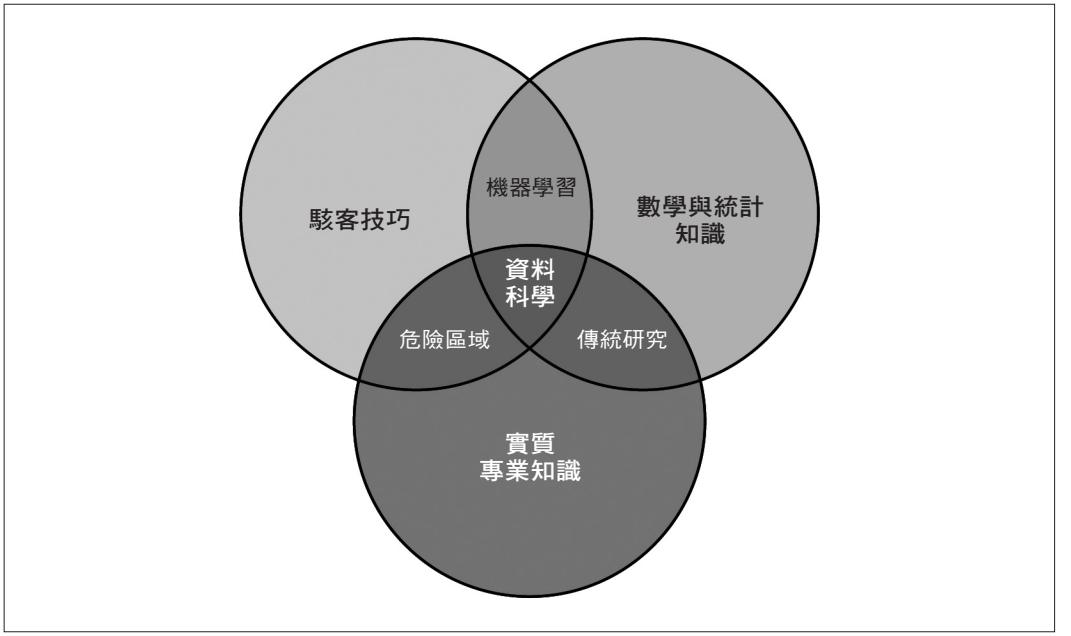

圖 P-1：由 Drew Conway 繪製的資料科學文氏圖。（資料來源：Drew Conway 授權，https://oreil.ly/PkOOw）

本書適用對象

我在華盛頓大學和許多技術研討會和見面會的教學場合中，最常被問到的問題是：「我該如何學習 Python ？」提問的人包括具有技術背景的學生、開發人員和研究人員，他們通常都已經具備編寫程式碼、使用計算及數值工具的經驗。其中大部分的人並不打算精通 Python，只是想要把它當成一個用來處理手邊大量資料並進行科學計算的工具。雖然網路上有大量的影音檔案、部落格文章及教學內容，但我對於這個問題長久以來都缺乏一個好的答案而感到沮喪，這給了我出版本書的動機。

本書並不是一本介紹 Python 或是一般程式設計的書籍。我假設本書的讀者已經熟悉了 Python 語言，知道如何定義函式、指定變數、呼叫物件方法、控制程式的流程以及利用 Python 執行一些基本任務。本書將能幫助 Python 的使用者瞭解如何運用 Python 資料科學堆疊（也就是我們將在接下來的章節介紹的程式庫以及相關的工具）來有效率地儲存、操作、以及進一步取得對於資料的洞見。

為何選用 Python ？

在過去的幾十年裡，Python 已經成為科學計算任務的一流工具，這些任務包括大規模數據集的分析與視覺化。這個發展也許會讓 Python 早期的支持者感到意外，因為 Python 在設計之初並沒有特別考慮到資料分析和科學計算。

Python 在資料科學領域特別有用的原因主要來自於這些大型具活躍的第三方套件生態系：*NumPy* 用來處理以同質性陣列為主的資料；*Pandas* 用來處理異質性和標籤類型的資料；*SciPy* 用來進行一般的科學計算工作；*Matplotlib* 用來處理具有發行等級品質的視覺化；*IPython* 用來進行互動式執行及程式碼共享；*Scikit-Learn* 可以進行機器學習；以及更多本書接下來要介紹的工具。

如果你正在尋找針對 Python 語言的指引，我建議你可以參考本書的姐妹專案《Python 旋風之旅》（*https://oreil.ly/jFtWj*）。這份簡短的報告介紹了 Python 的基礎與重要特色，適合已經熟悉一種以上程式語言的資料科學家閱讀。

本書大綱

本書每一篇都聚焦在一個特定的套件或工具，這些套件或工具都是 Python 資料科學中非常重要的組成。每一篇中再分割為較簡短且內容獨立的專章，讓每一章可以單獨地探討一個概念：

- 第一篇：〈Jupyter：更好用的 Python 開發環境〉介紹 IPython 與 Jupyter。這些套件提供的運算環境，是許多使用 Python 的資料科學家們執行作業的環境。
- 第二篇：〈NumPy 介紹〉聚焦在 NumPy 程式庫，這組程式庫提供 `ndarray`，它可以極高效率地在 Python 程式中儲存及操作密集資料陣列。
- 第三篇：〈使用 Pandas 操作資料〉介紹 Pandas 程式庫，它提供了 `DataFrame` 資料結構，可以高效地在 Python 程式中儲存和操作標籤及欄位式的資料。

- 第四篇：〈使用 Matplotlib 視覺化資料〉中則集中介紹 Matplotlib，這個程式庫提供了 Python 在資料視覺化上靈活的運用能力。
- 第五篇：「機器學習」聚焦在 Scikit-Learn 程式庫，這組程式庫對於一些重要的一些機器學習演算法，提供了高效且簡潔的 Python 實作。

PyData 的世界當然不會只有這 6 個套件，而且它每天仍在持續地成長。考慮到這一點，我也會在本書中介紹其他有趣的工作、專案、以及套件，它們將 Python 的可能性推到更遠。然而，這 6 個專案是目前在 Python 的資料科學領域中是最基本且有最多成果的，我認為就算即使這個生態系持續不斷地成長，仍然會圍繞著它們。

安裝注意事項

安裝 Python 及程式庫套組以啟用科學計算能力相當直覺。本節提供一些你在設定自己的電腦時需要留意的事項。

雖然有許多不同的方法可以安裝 Python，我最推薦的是在資料科學領域最受歡迎的 Anaconda 發佈版本，它在 Windows、Linux、或是 macOS 的作業方式都很類似。Anaconda 有兩個主要的發佈版本：

- Miniconda（*https://oreil.ly/dH7wJ*）提供了 Python 直譯器以及一個被稱為 *conda* 的命令列工具。conda 是一個 Python 套件的跨平台管理工具，就像是我們在 Linux 作業系統中使用的 apt 或是 yum 一樣。如果你是 Linux 使用者的話，一定可以很快上手。
- Anaconda（*https://oreil.ly/ndxjm*）包含 Python 以及 conda，還有許許多多資料科學使用的預裝套件。因為這些綑綁套件包的容量較大，全部安裝起來需要許多 GB 的硬碟空間。

包含在 Anaconda 中的套件均可以在 Miniconda 中手動安裝，基於這個理由，我推薦從 Miniconda 開始。

為了方便本書的閱讀，請下載並安裝 Miniconda 套件，並確認你選用的是 Python 3 版本，然後使用以下的指令安裝在本書中所有會使用到的套件：

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn jupyter
```

在本書中，我們也會使用到更多在 Python 資料科學生態系中的特定工具，安裝的方式通常也是跟上述的方式一樣簡單，只要輸入 **`conda install`** 以及**套件名稱**就可以了。如果

第 4 章

瞭解 Python 的資料型態

有效率的資料驅動科學與計算，需要瞭解資料如何儲存與操作。本章會瀏覽和比較資料陣列是如何在原生的 Python 中處理，以及 NumPy 如何改良它。瞭解這些基本的差異將會讓你在閱讀本書接下來的內容時，能有更深入的理解。

Python 的使用者經常是因為它的易用性而被吸引進來，其中一個部分就是動態型別。一些靜態型別的語言像是 C 或是 Java 需要對每一個變數做明確地宣告，而像是 Python 此類動態型別的程式語言則會跳過這個規格。例如，在 C 語言中，要指定一個特定的操作如下：

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

而在 Python 中，相同的操作如下所示：

```
# Python code
result = 0
for i in range(100):
    result += i
```

它們主要的差異為：在 C 語言中，每一個變數的資料型態需要明確地宣告，而 Python 的型態則是動態推導的。也就是說，可以指定任意型態的資料給任一變數：

```
# Python code
x = 4
```

```
x = "four"
```

在此例中，把 x 的內容從整數改為字串，同樣的事情在 C 語言中則會造成（視編譯器設定而定）編譯錯誤或是預期之外的結果：

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

此種彈性讓 Python 這一類的動態型別語言非常方便且易於使用。瞭解這樣的情況是如何運作的，對於學習使用 Python 有效率地分析資料非常有用。但是動態型別的特性也指向一個事實，就是 Python 的變數不會只是儲存值而已，它們還必須包含關於型別的額外資訊。以下的小節將探討這個問題。

Python 的整數不僅僅只是整數

標準的 Python 實作是以 C 語言寫成的。這表示每一個 Python 物件都是一個精巧設計的 C 語言結構，這個結構包含不只是值，還有其他的資訊。例如，當我們在 Python 中定義了一個 integer，像是 `x = 10000`，x 不僅僅只是一個「原始」的 integer，它實際上是一個指向複合式 C 語言結構的指標，在這個結構中包含了許多的值。檢視在 Python 3.10 的原始碼，可以發現長整數型態定義，實際上看起來是以下這個樣子（假設 C 語言的巨集是已經被展開之後的情況）：

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

在 Python 3.10 中，一個 integer 實際上包含了 4 個部分：

- ob_refcnt：參考的計數，用來協助 Python 處理記憶體的配置和解除。
- ob_type：設定變數的型態。
- ob_size：用來指定接下來的資料成員之記憶體大小。
- ob_digit：用來儲存打算在 Python 變數中表示的實際整數值。

由此可知，和其他像是 C 語言這種編譯式程式語言比起來，Python 相對來說在儲存整數時多了一些額外的負擔，如圖 4-1 所示：

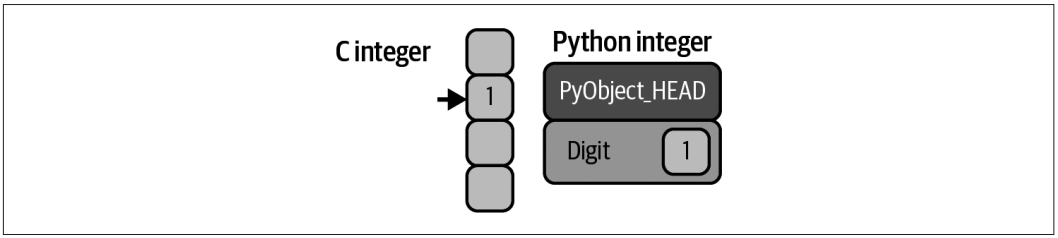

圖 4-1：C 語言和 Python 的整數差異

如圖所示，PyObject_HEAD 是結構的一部分，包含了參考計數器、型態編碼，以及其他在前面提到的部分。

要留意不同的部分是：一個 C 語言的整數是一個以位元組編碼的整數值在記憶體中位址的一個標籤，而 Python 整數則是一個指標，指向記憶體中一個包含所有 Python 物件資訊，其中含有放置整數值的那些位元組。在 Python 整數結構中，這些額外的資訊讓 Python 可以被自由及動態地編寫程式碼，然而，所有 Python 型態的額外資訊也是要付出成本，尤其是那些結合了許多物件的結構會特別明顯。

Python 的串列不僅僅只是串列

接下來讓我們來看看，當使用 Python 的資料結構去儲存許多的 Python 物件時會是什麼情形。在 Python 中，標準的可修改多元素容器是串列，建立一個整數的串列的方法如下：

```
In [1]: L = list(range(10))
        L
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: type(L[0])
Out[2]: int
```

建立一個字串型態的 list 也是類似的方法：

```
In [3]: L2 = [str(c) for c in L]
        L2
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
In [4]: type(L2[0])
Out[4]: str
```

因為 Python 是動態型別，因此我們可以建立異質的 list，如下所示：

```
In [5]: L3 = [True, "2", 3.0, 4]
        [type(item) for item in L3]
Out[5]: [bool, str, float, int]
```

但是這樣的彈性附帶了額外的成本：允許這些任意的型態，在串列中的每一個項目必須包含它自己的型態資訊、參考計數、以及其他的資訊，也就是說，每個項目都是一個完整的 Python 物件。當所有的變數都是相同型態的特殊情況下，大部分的資訊都是多餘的：也就是如果把它們儲存成固定型態的陣列會較有效率。動態型態的串列和固定型態（NumPy 類型）陣列的差異，請參考圖 4-2。

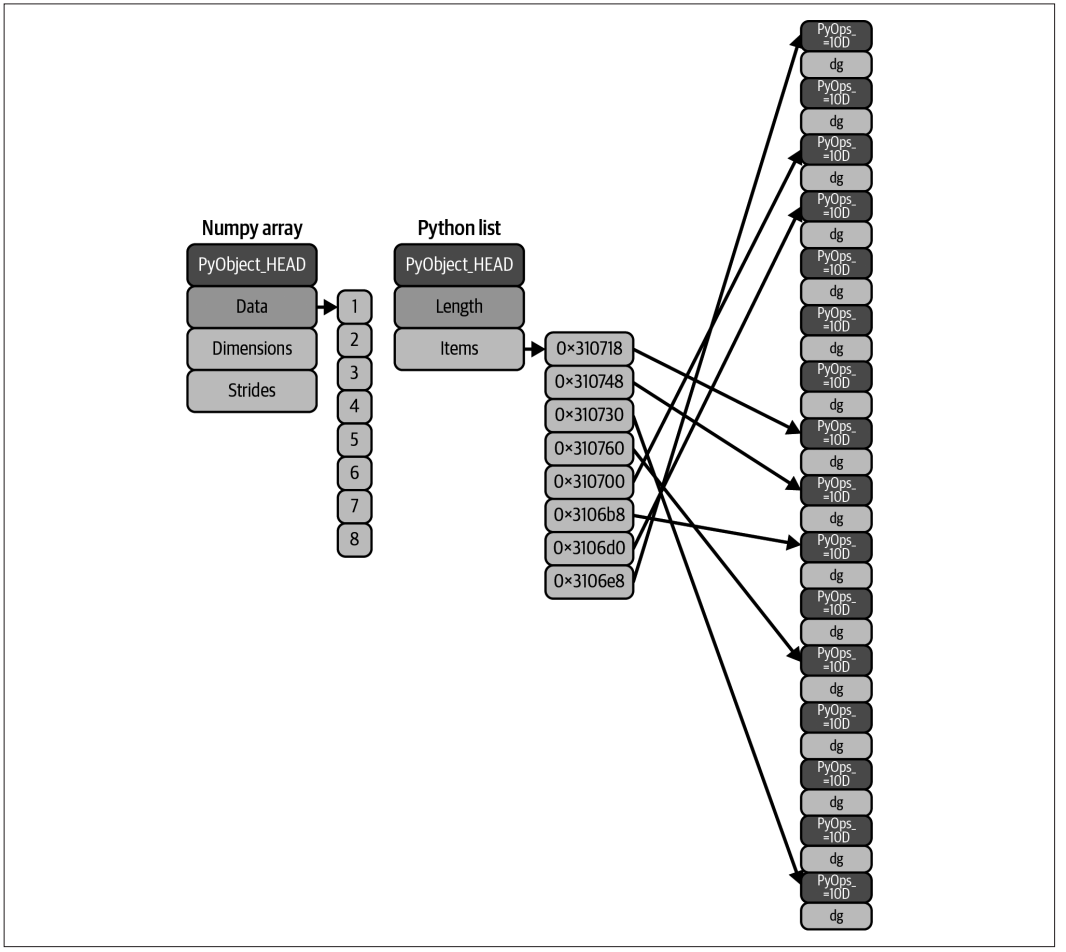

圖 4-2：C 語言和 Python 串列的差異

在實作層級中，陣列本質上包含了指向連結資料區塊的指標。在另一方面，Python 串列的指標則是指向一群指標，這些指標依序指向各自完整的 Python 物件，就像是之前看到的 Python 整數一樣。此種串列的優點就是彈性，它可以放入任何想要的型態，而像是 NumPy 類型的固定型態陣列就缺乏這樣的彈性，但是在儲存和操作資料上卻更有效率。

Python 的固定型態陣列

Python 提供幾種不同的方式可以有效率地儲存資料在固定型態、資料緩衝區中。內建的 `array` 模組（從 Python 3.3 開始）可以用來建立單一型態的稠密陣列：

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A
Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

在上述的程式碼中，「`i`」是用來設定接下來的內容之型態為整數（integer）的型態編碼。

然而更好用的是 NumPy 套件中的 `ndarray` 物件。雖然 Python 的 `array` 物件提供以陣列為基礎的資料是有效率的儲存方式，NumPy 增加了對於資料更有效率的操作。我們會在後續的章節中探討這些操作，接下來要說明幾種建立 NumPy 陣列的不同方法。

從 Python Lists 建立 NumPy 陣列

我們從匯入標準的 NumPy，並把它命令為 `np` 這個別名開始：

```
In [7]: import numpy as np
```

現在我們可以使用 `np.array`，從 Python 的串列來建立陣列：

```
In [8]: # Integer array
        np.array([1, 4, 2, 5, 3])
Out[8]: array([1, 4, 2, 5, 3])
```

不同於 Python 的串列，NumPy 限制所有在陣列中的內容需為同樣的型態。如果型態不符合，NumPy 將會試著自動轉換其型態，在這裡，整數會被轉換為浮點數：

```
In [9]: np.array([3.14, 4, 2, 3])
Out[9]: array([3.14, 4.  , 2.  , 3.  ])
```

如果你想要明確地設定陣列中的型態，可以使用 dtype 這個參數：

```
In [10]: np.array([1, 2, 3, 4], dtype=np.float32)
Out[10]: array([1., 2., 3., 4.], dtype=float32)
```

最後，不像是 Python 串列總是一維的順序，NumPy 陣列可以包含多個維度。在這裡展示如何利用串列來初始化一個多維度的陣列：

```
In [11]: # 使用巢狀式串列建立一個多維度的陣列
         np.array([range(i, i + 3) for i in [2, 4, 6]])
Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

如上所示，在內層的每一個串列，會被當作是產出結果的二維陣列每一列的內容。

從無到有建立陣列

特別是在大型的陣列，使用程序從無到有建立 NumPy 陣列會更有效率。底下是幾個例子：

```
In [12]: # 建立一個內容全為 0，長度為 10 的整數陣列
         np.zeros(10, dtype=int)
Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
In [13]: # 建立一個內容全為 1 的 3x5 浮點數陣列
         np.ones((3, 5), dtype=float)
Out[13]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])

In [14]: # 建立一個填滿 3.14 內容的 3x5 陣列
         np.full((3, 5), 3.14)
Out[14]: array([[3.14, 3.14, 3.14, 3.14, 3.14],
                [3.14, 3.14, 3.14, 3.14, 3.14],
                [3.14, 3.14, 3.14, 3.14, 3.14]])

In [15]: # 建立一個依序填滿的陣列
         # 從 0 開始，到 20 結束，每次以 2 為間隔
         #（這和內建的 range() 函式類似）
         np.arange(0, 20, 2)
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [16]: # 建立一個 5 個值的陣列，在 0 到 1 之間平均分佈
         np.linspace(0, 1, 5)
```

```
Out[16]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])

In [17]: # 建立一個均勻分佈的 3x3 陣列
         # 在 0 到 1 之間亂數值
         np.random.random((3, 3))
Out[17]: array([[0.09610171, 0.88193001, 0.70548015],
                [0.35885395, 0.91670468, 0.8721031 ],
                [0.73237865, 0.09708562, 0.52506779]])

In [18]: # 建立一個 3x3 的陣列，內容為常態分佈的亂數值
         # 平均是 0 而標準差為 1
         np.random.normal(0, 1, (3, 3))
Out[18]: array([[-0.46652655, -0.59158776, -1.05392451],
                [-1.72634268,  0.03194069, -0.51048869],
                [ 1.41240208,  1.77734462, -0.43820037]])

In [19]: # 建立一個 3x3 的陣列，內容為介於範圍是 [0, 10] 的整數亂數
         np.random.randint(0, 10, (3, 3))
Out[19]: array([[4, 3, 8],
                [6, 5, 0],
                [1, 1, 4]])

In [20]: # 建立一個 3x3 的單位矩陣（identity matrix）
           np.eye(3)
Out[20]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])

In [21]: # 建立一個 3 個整數的未初始化陣列
         # 這些值會是原本就存在那些記憶體中的值
         np.empty(3)
Out[21]: array([1., 1., 1.])
```

NumPy 的標準資料型態

NumPy 的陣列內含單一型態的值，因此充份瞭解這些型態的細節以及它們的限制是非常重要的。因為 NumPy 是使用 C 語言建立的，這些型態對於 C、Fortran 和其他類似程式語言的使用者來說應該會相當熟悉。

標準的 NumPy 資料型態列在表 4-1 中。在建構陣列時，可以透過字串來指定型態：

```
np.zeros(10, dtype='int16')
```

或使用相對應的 NumPy 物件：

```
np.zeros(10, dtype=np.int16)
```

除此之外還有一些更進階的型態規格可以設定，像是指定大端序（big-endian）或是小端序（little endian）數字等。更多的資訊，請參考 NumPy 的說明文件（*https:// numpy.org/*）。NumPy 也支援複合的資料型態，這部分將會在第 12 章加以討論。

表 4-1：標準的 NumPy 資料型態

資料型態	說明
bool_	布林值（True 或 False），以一個位元組來儲存
int_	預設的整數型態（和 C 的 long 一樣，一般不是 int64 就是 int32）
intc	和 C 語言的 int 一樣（一般是 int32 或 int64）
intp	用來做為索引的整數（和 C 的 ssize_t 一樣；一般不是 int32 就是 int64）
int8	位元組（-128 到 127）
int16	整數（-32768 到 32767）
int32	整數（-2147483648 到 2147483647）
int64	整數（-9223372036854775808 到 9223372036854775807）
uint8	無號整數（0 到 255）
uint16	無號整數（0 到 65535）
uint32	無號整數（0 到 4294967295）
uint64	無號整數（0 到 18446744073709551615）
float_	float64 的簡稱
float16	半精度浮點數：正負號位元，5 位元指數，10 位元尾數
float32	單精度浮點數：正負號位元，8 位元指數，23 位元尾數
float64	倍精度浮點數：正負號，11 位元指數，52 位元尾數
complex_	complex128 的簡稱
complex64	複數，使用 2 個 32 位元浮點數表示
complex128	複數，使用 2 個 64 位元浮點數表示

第 5 章

NumPy 陣列基礎

在 Python 中的資料操作幾乎就是在 NumPy 進行陣列操作的同義字：就連一些新式的工具像是 Pandas（在第三篇中會加以介紹）也是建立在 NumPy 陣列之上。這一章將展示使用 NumPy 陣列操作以存取資料和子陣列，以及 split（分割）、reshape（重塑）、以及 join（串接）陣列的例子。雖然在這裡展示的這些操作看起來有些不易閱讀，而且過於學術，但它們包含了貫穿本書其他範例的建構方塊，請你還是好好熟悉它們吧！

底下是將在這裡涵蓋的基本陣列操作之分類：

陣列的屬性

決定陣列的大小、形狀、記憶體使用、以及資料型態。

陣列的索引

取得以及設定陣列元素個別的值。

陣列的切片

在大陣列中取得和設定其中較小的子陣列。

陣列的重塑

改變陣列的形狀（shape）

陣列的組合和分割

把多個陣列組合成 1 個陣列，以及把 1 個陣列切割成許多個陣列。

NumPy 陣列屬性

先來討論一些有用的陣列屬性。我們一開始先定義 3 個隨機亂數陣列：一維陣列、二維陣列以及三維陣列，並使用 NumPy 的亂數產生器，在亂數產生前先設定一個 seed 以確保每一次程式碼執行時，都會產生相同的亂數陣列：

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)  # 指定一個種子值確保每次執行時
                                                # 均產生同樣的亂數內容

        x1 = rng.integers(10, size=6)  # 一維陣列
        x2 = rng.integers(10, size=(3, 4))  # 二維陣列
        x3 = rng.integers(10, size=(3, 4, 5))  # 三維陣列
```

每一個陣列分別有 `ndim`（維數）、`shape`（每一個維度的大小）、`size`（整個陣列的大小總和）、以及 `dtype`（每一個元素的資料型態）等屬性：

```
In [2]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)
        print("dtype:   ", x3.dtype)
Out[2]: x3 ndim:  3
        x3 shape: (3, 4, 5)
        x3 size:  60
        dtype:    int64
```

對於資料型態更多的討論請參考第 4 章。

陣列索引：存取單一個陣列元素

如果你習慣 Python 的標準串列索引方式，在 NumPy 中進行索引也會非常容易上手。

在一維陣列中，可以在中括號中指定想要的索引，以存取到第 *i* 個值（從 0 開始算），就像是在操作 Python 的串列一樣：

```
In [3]: x1
Out[3]: array([9, 4, 0, 3, 8, 6])

In [4]: x1[0]
Out[4]: 9

In [5]: x1[4]
Out[5]: 8
```

你也可以使用負數，從陣列的末端回推你想要的索引位置：

```
In [6]: x1[-1]
Out[6]: 6

In [7]: x1[-2]
Out[7]: 8
```

在多維陣列中，可以使用以半形逗號分隔的方式（列 , 行），存取陣列中的資料項：

```
In [8]: x2
Out[8]: array([[3, 1, 3, 7],
               [4, 0, 2, 3],
               [0, 0, 6, 9]])
In [9]: x2[0, 0]
Out[9]: 3
In [10]: x2[2, 0]
Out[10]: 0
In [11]: x2[2, -1]
Out[11]: 9
```

也可以透過前面所說明的索引方式修改指定位置的陣列值：

```
In [12]: x2[0, 0] = 12
         x2
Out[12]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])
```

要留意的是，NumPy 與 Python 的串列不同，它的陣列元素都必須是固定且單一的型態。這表示，你可能會在試著於整數陣列中插入一個浮點數值時，這個值將會被靜悄悄地截去小數部分。可千萬別不小心被這個自動轉換的行為害到。

```
In [13]: x1[0] = 3.14159  # 此數字的小數部分會被截掉！
         x1
Out[13]: array([3, 4, 0, 3, 8, 6])
```

陣列切片：存取子陣列

如同之前使用中括號存取個別的陣列元素一般，我們也可以再加上切片的符號（冒號「:」字元）改為存取其中的子陣列。NumPy 切片語法和在標準的 Python 串列中使用的語法一樣，存取陣列 x 的一個切片，可以使用以下的方式：

```
x[start:stop:step]
```

如果有任一個值沒有指定的話，則它們的預設值分別是 start=0、stop=<size of dimension> 和 step=1。接著分別來看看在一維陣列以及多維度陣列中操作子陣列的方法。

一維子陣列

以下是一些存取一維陣列子陣列的操作範例：

```
In [14]: x1
Out[14]: array([3, 4, 0, 3, 8, 6])

In [15]: x1[:3]  # 前面 3 個元素
Out[15]: array([3, 4, 0])

In [16]: x1[3:]  # 所有在索引 3 之後的元素
Out[16]: array([3, 8, 6])

In [17]: x1[1:4]  # 中間的子陣列
Out[17]: array([4, 0, 3])

In [18]: x1[::2]  # 間隔 2 的所有元素
Out[18]: array([3, 0, 8])

In [19]: x1[1::2]  # 從索引值 1 開始間隔 2 的所有元素
Out[19]: array([4, 3, 6])
```

當 step 是負值時，你可能會感到困惑。在這個情況下，start 和 stop 的預設值將會彼此互換。因此，這裡提供了一個反向取得陣列的簡便方式：

```
In [20]: x1[::-1]  # 反轉所有的元素
Out[20]: array([6, 8, 3, 0, 4, 3])

In [21]: x1[4::-2]  # 從索引值 4 開始，反向往前取得間隔 2 的所有元素
Out[21]: array([8, 0, 3])
```

多維子陣列

多個維度的切片也是使用同樣的方式，只要使用逗號來分隔多個切片值的指定內容就可以了。例如：

```
In [22]: x2
Out[22]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])
```

```
In [23]: x2[:2, :3]  # 前 2 列以及前 3 欄
Out[23]: array([[12,  1,  3],
                [ 4,  0,  2]])
In [24]: x2[:3, ::2]  # 前 3 列，間隔為 2 的欄
Out[24]: array([[12,  3],
                [ 4,  2],
                [ 0,  6]])

In [25]: x2[::-1, ::-1]  # 反向取出所有列和欄的值
Out[25]: array([[ 9,  6,  0,  0],
                [ 3,  2,  0,  4],
                [ 7,  3,  1, 12]])
```

存取陣列中的個別列或欄的所有值是一個常用的例行程序。此程序可以透過結合索引和切片的技巧來完成。以下使用單獨一個「:」符號就可以取出整欄的值：

```
In [26]: x2[:, 0]  # x2 陣列的第 1 欄
Out[26]: array([12,  4,  0])

In [27]: x2[0, :]  # x2 陣列的第 1 列
Out[27]: array([12,  1,  3,  7])
```

在上面這個讀取整列值的方法中，後面那個「:」符號可以省略，讓語法更加精簡：

```
In [28]: x2[0]  # 和 x2[0, :] 具有相同的效果
Out[28]: array([12,  1,  3,  7])
```

把子陣列視為未複製（No-Copy）的視圖

不同於 Python 串列的切片，NumPy 陣列的切片是以視圖的方式傳回而不是複製出陣列資料中的值。請參考我們之前使用的 2 維陣列：

```
In [29]: print(x2)
Out[29]: [[12  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

從該陣列中取出一個 2x2 的子的陣列：

```
In [30]: x2_sub = x2[:2, :2]
         print(x2_sub)
Out[30]: [[12  1]
          [ 4  0]]
```

現在我們修改這個子陣列的內容，你會看到原始的那個陣列值也被改變了：

```
In [31]: x2_sub[0, 0] = 99
         print(x2_sub)
Out[31]: [[99  1]
          [ 4  0]]

In [32]: print(x2)
Out[32]: [[99  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

有些使用者會對此種處理方式感到訝異，但此種方式最能表現出它的優點。例如：當我們在操作一個大型的資料集時，可以僅存取這個資料集的正在處理中的部分內容，而不需要在資料緩衝區中去複製那些還未處理的大量資料。

建立陣列的複本

雖然陣列視圖是一個不錯的特性，但有時候也是需要明確地從陣列或子陣列中複製出資料。只要使用 `copy()` 方法就可以簡單地做到：

```
In [33]: x2_sub_copy = x2[:2, :2].copy()
         print(x2_sub_copy)
Out[33]: [[99  1]
          [ 4  0]]
```

現在即使你修改子陣列，原來的那個陣列的內容也不會被更動了：

```
In [34]: x2_sub_copy[0, 0] = 42
         print(x2_sub_copy)
Out[34]: [[42  1]
          [ 4  0]]

In [35]: print(x2)
Out[35]: [[99  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

陣列重塑

另一個有用的操作是對陣列的重塑，可以利用 `reshape` 方法來達成。例如，如果你想要把 1 到 9 的數字放到一個 3x3 的陣列中，可以使用以下這樣的方式操作：

```
In [36]: grid = np.arange(1, 10).reshape(3, 3)
         print(grid)
Out[36]: [[1 2 3]
          [4 5 6]
          [7 8 9]]
```

不過要注意的是，透過此方式重塑陣列，原來的陣列和重塑之後的陣列尺寸要能夠符合。如果可能的話，reshape 方法會使用原有陣列中未複製的視圖，但這在不連續的記憶體緩衝中就不一定如此。

另外一個常用的重塑用法，是把一維陣列放進一個二維陣列中當作是它的其中一列或是一欄：

```
In [37]: x = np.array([1, 2, 3])
         x.reshape((1, 3))  # 透過 reshape 建立列向量
Out[37]: array([[1, 2, 3]])

In [38]: x.reshape((3, 1))  # 透過 reshape 建立欄向量
Out[38]: array([[1],
                [2],
                [3]])
```

有一種便捷的做法是在切片的語法中使用 np.newaxis：

```
In [39]: x[np.newaxis, :]  # 透過 newaxis 建立列向量
Out[39]: array([[1, 2, 3]])

In [40]: x[:, np.newaxis]  # 透過 newaxis 建立欄向量
Out[40]: array([[1],
                [2],
                [3]])
```

本書接下來的內容中將會經常看到此種類型的轉換。

陣列的串接和分割

所有前面執行的程序都是針對單一個陣列，當然也可以把多個陣列結合成一個，或是反過來把一個陣列分割成多個陣列。以下就來看看這些操作。

陣列的串接

在 NumPy 中串接兩個陣列，主要是以 np.concatenate、np.vstack、以及 np.hstack 這幾個程序來完成。np.concatenate 使用一個陣列的元組或是串列當作是第一個參數，如下所示：

```
In [41]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])
Out[41]: array([1, 2, 3, 3, 2, 1])
```

你也可以一次串接 2 個以上的陣列：

```
In [42]: z = np.array([99, 99, 99])
         print(np.concatenate([x, y, z]))
Out[42]: [ 1  2  3  3  2  1 99 99 99]
```

也可以使用在二維陣列上：

```
In [43]: grid = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [44]: # 沿著第一軸串接
         np.concatenate([grid, grid])
Out[44]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In [45]: # 沿著第二軸串接 (zero-indexed)
         np.concatenate([grid, grid], axis=1)
Out[45]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

如果要在不同維度陣列間進行串接操作，使用 np.vstack（垂直堆疊）以及 np.hstack（水平堆疊）函式會比較清楚：

```
In [46]: # 垂直地堆疊在陣列上
         np.vstack([x, grid])
Out[46]: array([[1, 2, 3],
                [1, 2, 3],
                [4, 5, 6]])

In [47]: # 水平地堆疊在陣列上
         y = np.array([[99],
                       [99]])
         np.hstack([grid, y])
```

```
Out[47]: array([[ 1,  2,  3, 99],
                [ 4,  5,  6, 99]])
```

同樣地，對於更高維度的陣列，`np.dstack` 則會沿著第三軸堆疊到陣列上。

分割陣列

和串接相反的操作是分割，其函式分別是 `np.split`、`np.hsplit`、以及 `np.vsplit`。這幾個函式可以透過一組陣列索引值的串列來指定要分割的點：

```
In [48]: x = [1, 2, 3, 99, 99, 3, 2, 1]
         x1, x2, x3 = np.split(x, [3, 5])
         print(x1, x2, x3)
Out[48]: [1 2 3] [99 99] [3 2 1]
```

請留意，N 個分割的點會產生 $N+1$ 個子陣列。`np.hsplit` 以及 `np.vsplit` 函式也是類似的行為：

```
In [49]: grid = np.arange(16).reshape((4, 4))
         grid
Out[49]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In [50]: upper, lower = np.vsplit(grid, [2])
         print(upper)
         print(lower)
Out[50]: [[0 1 2 3]
          [4 5 6 7]]
         [[ 8  9 10 11]
          [12 13 14 15]]

In [51]: left, right = np.hsplit(grid, [2])
         print(left)
         print(right)
Out[51]: [[ 0  1]
          [ 4  5]
          [ 8  9]
          [12 13]]
         [[ 2  3]
          [ 6  7]
          [10 11]
          [14 15]]
```

同樣地，`np.dsplit` 也是沿著第三軸進行分割用的函式。