前言

近來，Python 無疑是金融產業的主流策略技術平台之一。自從我在 2013 年寫了本書的第一版以來，我在許多談話與演說中，堅定地認為 Python 在金融領域的競爭優勢遠超過其他語言及平台。這個看法在 2018 年末已經成為無庸置疑的事實了，如今，世界各地的金融機構都直接充分利用 Python 程式，以及強大的資料分析、視覺化與機器學習程式包組成的 Python 生態系統。

除了金融領域之外，Python 通常也是程式語言入門課程（例如計算機科學程式）的首選，主要的原因除了 Python 的語法容易閱讀、可用許多範式（paradigm）來編寫之外，它也是人工智慧（AI）、機器學習（ML）和深度學習（DL）領域的一級公民。這些領域有許多流行的程式包和程式庫，它們要不是直接用 Python 編寫的（比如 ML 的 `scikit-learn`），就是提供 Python 包裝器（比如 DL 的 `TensorFlow`）。

金融界因為兩股力量的推動，正演變為新的時代。第一種，基本上，你已經可以用程式來取得所有金融資料了，一般來說，這是即時的，因此，它也是促成**數據驅動金融**（*data-driven finance*）的主因。在幾十年前，大多數的交易或投資決策都是由交易員及投資組合經理人透過報章雜誌或個人訪問所收集的資訊決定的，後來終端機出現了，可以透過電腦與電子通訊，將金融資料即時顯示在交易者和投資組合經理的面前。如今，個人（或團隊）甚至無法應付在一分鐘之內大量生成的金融資料，只有不斷提升處理速度與計算能力的電腦，才跟得上金融資料的數量與速度，換句話說，如今全世界絕大多數的股票交易量都是演算法和電腦執行的，不是人類交易員執行的。

第二股主要力量是在金融領域中日益重要的人工智慧。越來越多金融機構試著利用 ML 與 DL 演算法來改善業務，以及它們的交易和投資績效。在 2018 年初，第一本關於“金融機器學習”專書的出版，更是突顯了這一個趨勢，毫無疑問，未來還會有更多這類書籍的出版，這導致了所謂的 *AI 優先金融*（*AI-first finance*），這種做法使用靈活的、可參數化的 ML 與 DL 演算法來取代傳統的金融理論。傳統的理論或許很優雅，但是它們在數據驅動的 AI 優先金融時代只能發揮有限的作用。

Python 是應付這個金融時代挑戰的正確語言和生態系統。雖然本書也介紹基本的無監督和監督學習（以及深度神經網路）等 ML 演算法，但我們的重點仍然是 Python 的資料處理和分析能力。人工智慧在金融領域的重要性（包括現在與未來）必須用另一本書才能充分說明。然而，大多數的 AI、ML 與 DL 技術都需要大量的資料，因此，你絕對要先掌握數據驅動金融。

《*Python for Finance*》的第二版傾向升級，而非更新，例如，這一版加入完整的演算法交易（第 4 部分），這一個主題近來在金融界相當重要，也很受散戶的歡迎。這一版也加入比較入門的部分（第 2 部分），我先介紹基本的 Python 程式設計與資料分析主題，以便在本書稍後的部分加以應用。並且完全刪除第一版的一些章節，例如，我將關於 web 技術與程式包（例如 `Flask`）的部分移除了，因為坊間已經有許多專門探討這些主題的書籍可供參考了。

我想要在第二版探討更多關於金融的主題，將重點放在對金融數據科學、演算法交易和計算金融而言特別實用的 Python 技術上。如同第一版，這一版介紹的都是很實用的做法，我會先提供實作與說明，再介紹理論的細節。我通常會把焦點放在全局，而不是某個類別、方法或函式的神秘參數上。

介紹了第二版的基本做法之後，我想要強調的是，本書既不是 Python 語言的介紹書籍，也不是一般的金融理財書籍，你可以自己找到許多介紹這兩個領域的優秀資源。本書處於這兩個令人興奮的領域的交會點，我假設讀者有一些程式設計（不一定是 Python）與金融方面的背景知識，準備教這些讀者將 Python 及其生態系統應用在金融領域上。

你可以到 *http://py4fi.pqp.io* 免費註冊 Quant Platform，並從它那裡取得並執行本書的 Jupyter Notebooks 及程式碼。

我的公司（The Python Quants）與我本人都提供許多其他的資源，以協助你掌握金融資料科學、人工智慧、演算法交易及計算金融。你可以先瀏覽下列網站：

- 我們公司的網站（*http://tpq.io*）

第一章

為何在金融領域使用 Python

銀行其實是技術公司。

—Hugo Banziger

Python 程式語言

Python 是一種高階、多用途的程式語言，被用在技術領域以及其他廣泛的領域。你可以在 Python 的網站（*https://www.python.org/doc/essays/blurb*）看到下列的行動綱要：

> Python 是一種直譯式、物件導向、高階的程式語言，具備動態語意。由於 Python 有高階的內建資料結構，以及動態型態及動態綁定，它非常適合用來快速開發應用程式，也很適合當成膠水語言，將既有的元件連結起來。Python 的語法特別強調易讀性，既簡單且容易學習，因此可以降低程式的維護成本。Python 支援模組與程式包，鼓勵程式碼模組化與復用。Python 有原始碼與二進制形式的解譯器與廣泛的標準程式庫，可在所有平台上免費使用及發表。

這段話很好地指出 Python 為何成為現今主流的程式語言之一。如今，許多初學者與技術高超的專業開發者都在學校、大學、web 公司、大型企業、金融機構，以及任何一種科學領域廣泛地使用 Python。

Python 有以下幾項重要的特性：

開放原始碼

> Python 及多數的支援程式庫和工具都開放原始碼，它們的使用條款通常相當靈活且開放。

直譯的

CPython 這個參考作品也是這種語言的直譯器，可在執行期將 Python 程式碼轉換成可執行的位元組碼（byte code）。

多範式

Python 支援各種不同的編程及實作範式，例如物件導向與指令式、泛函或程序編程。

多用途

Python 可讓你快速、互動地開發程式碼，以及建構大型的應用程式；它可以用來進行低階的系統操作，以及高階的分析工作。

跨平台

Python 可在多數重要的作業系統中使用，例如 Windows、Linux、macOS 等。它可以用來建構桌上型及 web 應用程式，也可以在最大型的叢集和最強大的伺服器上使用，以及 Raspberry Pi 這種小型裝置（*http://www.raspberrypi.org*）。

動態型態

Python 的型態通常是在執行期推斷出來的，而非多數的編譯語言那樣靜態地宣告。

運用縮排

相較於多數其他程式語言，Python 使用縮排來標記程式段落，而不是使用小括號、中括號，或是分號。

資源回收

Python 使用自動化資源回收機制，所以程式員不需要管理記憶體。

關於 Python 語法以及 Python 究竟是怎麼一回事，Python Enhancement Proposal 20（也就是所謂的“Zen of Python”）提出主要的方針。你可以在每一個互動式 shell（殼層）使用 `import this` 來顯示它：

```
In [1]: import this
        The Zen of Python, by Tim Peters

        Beautiful is better than ugly.
        Explicit is better than implicit.
        Simple is better than complex.
        Complex is better than complicated.
        Flat is better than nested.
        Sparse is better than dense.
```

```
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python 簡史

雖然 Python 對一些人來說仍然是新鮮的事物，但它已經問世一段時間了。事實上，荷蘭人 Guido van Rossum 早在 1980 年代就開始開發它了，他目前還在積極地開發 Python，Python 社群授予他終身仁慈獨裁者（*Benevolent Dictator for Life*）頭銜。van Rossum 在擔任 Python 核心開發工作的積極推動者數十年後，於 2018 年 7 月退出這個職位。以下是 Python 的開發里程碑（*http://bit.ly/2DYWqCW*）：

- **Python 0.9.0** 在 1991 年發表（第一次發表）
- **Python 1.0** 在 1994 發表
- **Python 2.0** 在 2000 發表
- **Python 2.6** 在 2008 發表
- **Python 3.0** 在 2008 發表
- **Python 3.1** 在 2009 發表
- **Python 2.7** 在 2010 發表
- **Python 3.2** 在 2011 發表
- **Python 3.3** 在 2012 發表
- **Python 3.4** 在 2014 發表
- **Python 3.5** 在 2015 發表
- **Python 3.6** 在 2016 發表
- **Python 3.7** 在 2018 年 6 月發表

值得注意的是，自 2008 年以來，Python 有兩個主要的版本可供使用，它們依然被持續開發，更重要的是，我們可以平行使用它們，有時這會讓 Python 新手摸不著頭緒。在寫這本書時，我認為這種情況可能還會持續一段時間，因為有大量可供使用或位於生產環境中的程式碼仍然是 Python 2.6/2.7。本書的第一版使用 Python 2.7，第二版則完全使用 Python 3.7。

Python 生態系統

Python 生態系統並非只是個程式語言，它的主要特性在於，它有大量的程式包與工具可用。這些程式包與工具，要不是必須在你需要時**匯入**（例如畫圖程式庫），不然就是獨立的系統程序（例如 Python 的互動式開發環境）。匯入的意思，就是讓該程式包就緒，可供目前的名稱空間與 Python 直譯器程序使用。

Python 本身就有大量的程式包與模組，可在各方面增強基本直譯器的功能，它們統稱為 *Python 標準程式庫*（*Python Standard Library*）（*https://docs.python.org/3/library/index.html*）。例如，你不需要做任何匯入就可以執行基本的數學計算，但如果你需要更專門的數學函數，就必須透過 `math` 模組匯入它們：

```
In [2]: 100 * 2.5 + 50
Out[2]: 300.0

In [3]: log(1)  ❶

        ---------------------------------------------------------------
        NameError                         Traceback (most recent call last)
        <ipython-input-3-74f22a2fd43b> in <module>
        ----> 1 log(1)  ❶

        NameError: name 'log' is not defined

In [4]: import math  ❷

In [5]: math.log(1)  ❷
Out[5]: 0.0
```

❶ 如果沒有匯入程式包，就會出現錯誤。

❷ 匯入 `math` 模組之後，就可以執行計算了。

`math` 是每個 Python 版本都有的標準模組，你也可以單獨安裝許多其他程式包，並且像標準模組一樣使用它們。你可以從各種（web）資源取得這種程式包。但建議你使用

Python 程式包管理器，來確保所有程式庫彼此維持一致（第 2 章會進一步說明這個主題）。

截至目前為止的程式範例都是用互動式 Python 環境來展示的，它們是 IPython（*http://www.ipython.org*）與 Jupyter（*http://jupyter.org*）。在寫這本書時，它們是最流行的互動式 Python 環境。雖然 IPython 最初只是增強型互動式 Python shell，但它現在已經有許多一般只能在整合開發環境（IDE）看到的功能，例如剖析（profiling）與除錯。高階的文字 / 程式碼編譯器通常具備 IPython 缺乏的功能，例如 Vim（*http://vim.org*），它也可以和 IPython 整合。因此，我經常在開發 Python 的過程中，使用包含 IPython 與文字 / 程式碼編譯器的基本工具鏈。

IPython 在許多方面增強標準的互動式 shell，它提供改善的命令列歷史功能，也可以讓你輕鬆地檢視物件。例如，你只要在函式名稱的前面或後面加上一個 `?`，就可以印出輔助說明（`docstring`）（加上 `??` 可印出更多資訊）。

IPython 原本有兩個流行的版本：*shell* 版本，以及瀏覽器版本（*Notebook*）。由於 Notebook 版本如此實用且流行，它已經演變成獨立的、與語言無關的專案，目前稱為 Jupyter。在這個背景之下，Jupyter Notebook 繼承 IPython 大部分的優點就不足為奇了，它也提供了其他的好處，例如視覺化。

關於如何使用 IPython，請參考 VanderPlas（2016 年，第 1 章）。

Python 用戶頻譜

Python 不僅深受專業軟體開發者的喜愛，也獲得一般的開發者，以及領域專家及科學開發者的青睞。

專業軟體開發者發現，只要使用 Python 即可高效地建構大型的應用程式。它幾乎支援所有的程式設計範式、有許多強大的開發工具可用，而且理論上，任何工作都可以用 Python 來處理。這類用戶通常會建構自己的框架與類別，也會大量使用基本的 Python 及科學堆疊（scientific stack），並且充分活用生態系統。

科學開發者或**領域專家**通常大量使用某些程式包與框架，他們會建構自己的應用程式，並且隨著時間而不斷增強與優化它，也會根據自己的特殊需求來定制生態系統。這些用戶族群經常會執行較長的互動式對話（session）、快速地建立新程式的原型，以及探索他們的研究成果或領域資料集，或將它們視覺化。

一般的程式員喜歡用 Python 來處理他們認為很適合用 Python 處理的問題。例如，他們會造訪 matplotlib 展示網頁、複製並修改那裡的視覺化程式碼來滿足他們的特殊需求，這應該是對這個族群的成員而言，很有幫助的使用案例。

另一個重要的 Python 用戶族群是**程式初學者**，也就是剛開始學習編寫程式的人。現今，Python 已經成為大學、專科學校、甚至一般學校在教授編程時經常選擇的語言了[1]。主要的原因在於，它的基本語法非常容易學習與瞭解，即使對非開發者而言也是如此。此外，Python 支援幾乎所有編程風格，這一點也很有幫助[2]。

科學堆疊

有一組程式包被統稱為**科學堆疊**（*scientific stack*），其中有許多程式包，茲列出幾項如下：

NumPy（*http://www.numpy.org*）

NumPy 有一個多維陣列物件，可用來儲存同質或異質的資料。NumPy 也提供優化的函式 / 方法來處理這個陣列物件。

SciPy（*http://www.scipy.org*）

SciPy 包含一組子程式包及函式，它們實作了科學或金融界常用的標準功能，例如，三次樣條插值（cubic splines interpolation）或數值積分。

matplotlib（*http://www.matplotlib.org*）

這是最受歡迎的 Python 繪圖及視覺化程式包，提供 2D 與 3D 視覺化功能。

pandas（*http://pandas.pydata.org*）

pandas 以 NumPy 為基礎，提供更豐富的類別，可用來管理與分析時間序列及表格資料；pandas 也和 matplotlib 的繪圖功能，以及和 PyTables 的資料儲存及取回功能緊密整合。

---

1 例如，Python 是紐約城市大學巴魯克學院金融工程碩士課程（*http://mfe.baruch.cuny.edu*）所使用的主要語言。世界各地有許多大學都用本書的第一版來教授如何用 Python 進行財務分析，以及建構應用程式。

2 見 *http://wiki.python.org/moin/BeginnersGuide*，你可以在那裡找到許多協助剛開始學習 Python 的開發者及非開發者的寶貴資源。

`scikit-learn`（*http://scikit-learn.org*）

`scikit-learn` 是流行的機器學習（ML）程式包，為許多 ML 演算法（例如用來估計、分類或聚類的演算法）提供統一的應用程式設計介面（API）。

`PyTables`（*http://www.pytables.org*）

`PyTables` 是 `HDF5` 資料儲存包裝（*http://www.hdfgroup.org/HDF5/*）的包裝器（wrapper），這種程式包使用階層式資料庫 / 檔案格式來實作優化的磁碟 I/O 操作。

視特定的領域或問題而定，你可以在這個堆疊中加入其他的程式包，那些程式包往往是以上述的一或多個程式包為基礎的。但是，`NumPy ndarray` 類別（見第 4 章）與 `pandas DataFrame` 類別（見第 5 章）通常都是**最大公分母**或**最基本的元素**。

即使只將 Python 視為一種程式語言，世上也沒有太多語言可以在語法與優雅性等方面和它並駕齊驅。例如，Ruby 是一種經常被拿來與 Python 比較的流行語言，這項語言的網站（*http://www.ruby-lang.org*）這樣形容它：

> 它是一種開放原始碼的動態程式語言，把焦點放在簡單性與生產力。它具備優雅的語法，讀起來很自然，寫起來也很容易。

用過 Python 的人應該認同 Python 也可以用同一段文字來介紹自己。但是，對大部分的使用者來說，Python 與具備同樣特點的語言之間最大的差異在於 Python 提供的科學堆疊。所以，Python 不但是好用且優雅的語言，也可以取代專業領域的語言及工具，例如 Matlab 或 R。如果你是專業的 web 開發者或系統管理員，它也內建了許多你期望的東西。此外，Python 也擅長與領域專用語言（例如 R）互動，因此我們通常不需要決定**究竟要使用 *Python* 還是另一種語言**，只要決定讓哪一種語言成為主要語言即可。

金融界的技術

有了以上關於 Python 的“粗略概念”之後，我們要退一步，簡單地思考一下技術在金融扮演的角色，這可以讓我們更認識 Python 曾經扮演的角色，更重要的是，它在未來的金融產業中，可能扮演的角色。

在某種意義上，技術對於金融機構（例如，與生物技術公司相比）或財務部門（與物流等其他的企業職能相比）而言**沒有什麼特別的作用**。但是，近年來，在創新與監管（regulation）的刺激之下，銀行與其他金融機構（例如對沖基金）變得越來越像技術公司，而**不僅僅**是金融仲介機構。技術已經成為世界上幾乎所有金融機構的重要資產，可能帶來競爭優勢，也可能導致劣勢。我們可以用一些背景資訊來說明事態為何如此發展。

技術開銷

銀行和金融機構是每年對技術投注最多資源的行業。因此，下面這段話不僅說明技術對金融業的重要性，也指出金融業對技術行業的重要性：

> FRAMINGHAM, Mass., 2018 年 7 月 14 日—根據國際資料公司（IDC）最新公布的金融服務 IT 支出指南數據，全球金融服務公司於資訊技術（IT）方面的支出，在 2021 年，將從 2018 年的 4,400 億美元成長至接近 5,000 億美元。
>
> —IDC（*http://bit.ly/2RUAV8Y*）

講明白一點，銀行和其他金融機構正爭先恐後地將他們的業務與運維模式數位化：

> 據預測，在 2017 年，北美的銀行對新技術的投資將到達 199 億美元。
>
> 這些銀行致力開發既有的系統，以及新的技術解決方案，來提高它們在全球市場的競爭力，並吸引對新的線上和移動技術感興趣的客戶。對提供新創意和軟體解決方案給銀行的全球金融技術公司來說，這是個大好機會。
>
> —Statista（*http://bit.ly/2Q04KYr*）

如今，大型的跨國銀行通常會聘僱數千名開發人員來維護既有的系統和建構新系統。具備高度技術需求的大型投資銀行，往往每年編列數十億美元的技術預算。

用技術來推動

技術的發展也促進了金融部門的創新和效率的提高。通常，這個領域的專案都是在**數位化**的保護傘之下運行的。

> 過去幾年來，金融服務業在技術的主導之下，已經經歷了一場巨變。許多主管期望他們的 IT 部門提升效率，促進可以改變遊戲規則的創新，同時以某種方式降低成本，繼續支援舊有系統。與此同時，金融技術初創企業正蠶食著成熟的市場，它們從無到有，藉由不受舊有系統阻礙的解決方案來協助顧客，並取得領先地位。
>
> —普華永道 2016 年第 19 次全球 CEO 年度調查（*https://pwc.to/1OYTO2d*）

效率的提高，也迫使企業在越來越複雜的產品或交易中尋找競爭優勢。這反過來又實質增加風險，讓風險的管理以及監管變得越來越困難。在 2007 年和 2008 年發生的金融危機展示了這種發展帶來的風險。類似的情況，“演算法與電腦大暴走”也是金融市場可能出現的風險，2010 年 5 月發生的所謂“閃崩（*flash crash*）事件”戲劇性地體現這種情

況（*http://en.wikipedia.org/wiki/2010_Flash_Crash*），當時，許多股票與指數因為自動拋售機制而大幅下跌。第 4 部分將討論與金融商品演算法交易有關的主題。

技術與人才，是入場的門檻

一方面，在其他條件都不變的情況下，提升技術可隨著時間降低成本。另一方面，金融機構持續大量投資技術，除了意圖獲得市佔率之外，也為了捍衛目前的地位。現代企業若要積極參與某些金融領域，往往需要對技術與熟練的員工進行大規模投資。例如，考慮衍生商品分析領域：

> 採用內部策略進行 OTC（衍生商品）交易定價的公司，在整個軟體生命週期中，僅建立、維護和增強一個完整的衍生商品資料庫，就需要投資 2,500 萬至 3,600 萬美元。
>
> —Ding（2010）

建立完整的衍生商品分析資料庫不僅成本高、耗時長，還需要足夠的專家來做這件事，況且專家們還必須擁有合適的工具和技術來完成任務。隨著 Python 生態系統的發展，這種工作已經變得越來越有效率，而且這方面的預算與 10 年前相較之下已經精簡許多了。第 5 部分會介紹衍生商品分析，並且僅用 Python 和 Python 標準程式包來建構小型但強大且靈活的衍生商品定價程式庫。

長期資本管理公司（LTCM）在早期談到的一段話也進一步支持這種關於技術和人才的觀點。長期資本管理公司曾經是最受尊敬的定量對沖基金之一，但是在 1990 年代末破產了。

> Meriwether 花了 2,000 萬美元購置一套最先進的計算機系統，並聘請頂尖的金融工程師團隊來管理位於美國康乃狄克州格林威治的 LTCM。這是在產業層面進行風險管理。
>
> —Patterson（2010）

當初 Meriwether 需要耗資數千萬美元購買的計算能力，如今可能只要用數千美元就可以購得，甚至可以用靈活付款方案，向雲端提供商租用。第 2 章會展示如何在雲端設置基礎設施，藉以使用 Python 進行互動式金融分析、應用程式開發和部署。這種專業基礎設施的預算每個月只要幾美元起跳。另一方面，對大型金融機構來說，交易、定價和風險管理變得極其複雜，以致於他們必須部署具備數萬個計算核心的 IT 基礎設施。

不斷提升的速度、頻率與資料量

技術的進步對金融行業造成許多影響，其中最大的層面就是決定一項金融交易並執行它的**速度**和**頻率**。Lewis（2014）用生動的細節描述所謂的**閃電交易**（也就是以最快的速度進行交易）。

一方面，為了讓資料在更短暫的時間之內有效，我們必須即時做出反應；另一方面，交易速度與頻率的提升，讓資料量進一步地增加。這導致各種因素互相強化，將金融交易的平均時間尺度系統性地降低。這個趨勢在十年前就開始了：

> Renaissance 的 Medallion 基金在 2008 年利用閃電般快速的電腦，以及市場的極度波動，獲得驚人的 80% 收益。Jim Simons 是該年度全球對沖基金收入最高的人，淨賺 25 億美元。
>
> —Patterson（2010）

一檔股票在 30 年期間的逐日股價資料相當於 7,500 個收盤價，這種資料是當今大多數金融理論的基礎。例如，現代（或標準差）投資組合理論（MPT）、資本資產定價模型（CAPM）和風險值（VaR）都使用逐日股價資料。

相較之下，在一個典型的交易日中，蘋果公司（AAPL）的股價在一小時之內可能會報價 15,000 次左右，這大約是 30 年日收盤價的兩倍（見第 23 頁，“數據驅動與 AI 優先金融”之中的例子）。這種情況帶來一些挑戰：

資料處理

你再也不能只考慮與處理股票或其他金融商品的收盤價了，它們有“太多”資訊在一天之內出現，有些商品會每週 7 天，每天 24 小時出現龐大資訊。

分析速度

你往往必須在幾毫秒之內，甚至更快的時間內做出決策，這需要建構相應的分析功能，並即時分析大量的資料。

理論基礎

雖然傳統的金融理論與概念遠非完美無缺，但隨著時間的推移，它們也受到充分的測試（有時被很好的理由推翻）；就現今重要的毫秒及微秒時間尺度而言，仍然沒有在傳統概念上一致的金融概念與理論已被證實是穩健的。

這些挑戰通常只能透過現代技術來解決。另一件可能讓你驚訝的事實是，“缺乏一致的理論”這個問題往往是用技術方法來解決的，因為高速的演算法利用的是市場微觀結構元素（例如下單流（order flow）、買賣價差），而不是某種金融推理（reasoning）。

即時分析的興起

在金融產業中，有一門學科的重要性已經大幅提升：**金融和資料分析**。這個現象與業界對速度、頻率與資料量快速成長的認知密切相關。事實上，即時分析可視為業界對這一項趨勢做出的回應。

粗略地說，“金融和資料分析”是利用軟體和技術，結合（可能是先進的）演算法和其他方法來收集、處理和分析資料，藉以獲得見解、做出決策或滿足監管要求的學科。例如，評估銀行零售部門調整金融商品的定價結構，或是為投資銀行複雜的衍生商品投資組合大規模地隔夜計算信用評價調整（CVA）對銷售造成什麼影響。

在這個背景之下，金融機構面臨兩大挑戰：

大數據
: 早在“大數據”這個名詞出現之前，銀行與其他金融機構就必須處理大量的資料了；然而，單次分析任務需要處理的資料量已經隨著時間的過去有了巨幅成長，需要更高的計算能力，使用更大的記憶體和儲存容量。

即時
: 決策者以前可以依靠有組織的、定期的規劃，以及決策和（風險）管理程序，但是現今他們必須即時處理這些工作；以前必須在後台辦公室通宵達旦完成批次處理的工作，有一些都已經移至前端辦公室即時執行了。

我們同樣可以在此看到技術的進步與金融 / 商業實踐法之間的相互作用。一方面，我們運用現代技術，不斷改善分析方法的速度和能力；另一方面，技術的進步使得幾年前甚至幾個月前被認為不可能的（或是由於預算的限制而不可行）分析方法成為可行。

分析領域有一個主流趨勢是在中央處理器（CPU）端使用平行架構，並且在通用圖形處理單元（GPGPU）端使用大規模的平行架構。目前的 GPGPU 有成千上萬個計算核心，這讓我們必須徹底地反思“平行”對不同的演算法究竟有什麼意義。用戶往往需要學習新的編程範式與技術，才能充分運用這些硬體，這是技術方面的障礙之一。

在金融領域使用 Python

上一節提出技術在金融領域發揮作用的幾個層面：

- 金融產業的技術成本
- 技術是新業務與創新的促進因素
- 技術與人才是金融產業的門檻
- 提升速度、頻率與資料量
- 即時分析的興起

這一節要分析 Python 如何協助解決這些挑戰。但首先，在基本的層面上，我們要從語言與語法的角度來簡要地分析 Python 在金融領域的使用情況。

金融與 Python 語法

在金融背景之下第一次使用 Python 的人經常需要克服演算法問題，這一點和想要解出微分公式，計算積分，或將一些資料視覺化的科學家很像。一般來說，在這個階段，他們不太需要考慮諸如正式的開發程序、測試、製作文件，或部署之類的問題，但人們往往在這個階段愛上 Python。造成這種情況的主因，應該是 Python 的語法整體上和描述科學問題或金融演算法的數學語法非常接近。

我們可以用一個金融演算法來說明這一點：用蒙地卡羅模擬來評估歐式看漲選擇權的價格。我們使用 Black-Scholes-Merton（BSM）模型，這種模型的標的物風險因子遵循幾何布朗運動（Brownian motion）。

假設我們使用這些參數值來估價：

- 初始股票指數 $S_0 = 100$
- 歐式看漲選擇權的履約價 $K = 105$
- 選擇權有效期 $T = 1$ 年
- 固定無風險短期收益率 $r = 0.05$
- 固定波動率 $\sigma = 0 .2$

在 BSM 模型中，到期日價格是用公式 1-1 算出來的隨機變數，其中的 z 是標準常態分布的隨機變數。

公式 1-1 *Black-Scholes-Merton*（1973）到期日指數

$$S_T = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}z\right)$$

這是蒙地卡羅估價過程的演算法說明：

1. 從標準常態分布取 I 個偽亂數 $z(i), i \in \{1, 2, \ldots, I\}$。

2. 用 $z(i)$ 與公式 1-1 算出所有到期日指數 $S_T(i)$。

3. 計算選擇權在到期日的所有內在價值 $h_T(i) = \max\left(S_T(i) - K, 0\right)$。

4. 用公式 1-2 的蒙地卡羅公式估計選擇權現值。

公式 1-2 歐式選擇權蒙地卡羅估計式

$$C_0 \approx e^{-rT}\frac{1}{I}\sum_I h_T(i)$$

我們將這個問題與演算法轉換成 Python。下面的程式碼實作了所需的步驟：

```
In [6]: import math
        import numpy as np  ❶

In [7]: S0 = 100.  ❷
        K = 105.  ❷
        T = 1.0  ❷
        r = 0.05  ❷
        sigma = 0.2  ❷

In [8]: I = 100000  ❷

In [9]: np.random.seed(1000)  ❸

In [10]: z = np.random.standard_normal(I)  ❹

In [11]: ST = S0 * np.exp((r - sigma ** 2 / 2) * T + sigma * math.sqrt(T) * z)  ❺

In [12]: hT = np.maximum(ST - K, 0)  ❻

In [13]: C0 = math.exp(-r * T) * np.mean(hT)  ❼

In [14]: print('Value of the European call option: {:5.3f}.'.format(C0))  ❽
         Value of the European call option: 8.019.
```

❶ 本例使用 NumPy 程式包。

❷ 定義模型與模擬參數值。

❸ 使用固定的亂數產生器種子值。

❹ 取得標準常態分布的亂數。

❺ 模擬到期價。

❻ 算出選擇權到期日報酬。

❼ 計算蒙地卡羅估計式。

❽ 印出估計結果值。

這段程式有三個地方值得一提：

語法

Python 的語法非常接近數學語法，例如指派參數值的部分。

翻譯

每一段數學與（或）演算法通常都可以轉換成單行 Python 程式碼。

向量化

NumPy 的一大優點是它有紮實的、向量化的語法，例如可以用單行程式碼執行 100,000 次計算。

你可以在 IPython 或 Jupyter Notebook 等互動式環境使用這段程式。但是，準備供人重複使用的程式通常會用所謂的**模組**（或**腳本**）來整理，它們是一個使用 *.py* 副檔名的 Python 檔案（技術上是個文字檔）。你可以將這個例子寫成範例 1-1 這種模組，並且將它存成名為 *bsm_mcs_euro.py* 的檔案。

範例 1-1 歐式看漲選擇權蒙地卡羅估價

```
#
# 歐式看漲選擇權蒙地卡羅估價
# 使用 Black-Scholes-Merton 模型
# bsm_mcs_euro.py
#
# Python for Finance, 2nd ed.
# (c) Dr.Yves J. Hilpisch
```

```
#
import math
import numpy as np

# 參數值
S0 = 100. # 初始指數值
K = 105. # 履約價
T = 1.0 # 到期日
r = 0.05 # 無風險短期收益率
sigma = 0.2 # 波動率

I = 100000 # 模擬數

# 估價演算法
z = np.random.standard_normal(I) # 偽亂數
# 到期日價格
ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * math.sqrt(T) * z)
hT = np.maximum(ST - K, 0) # 到期日報酬
C0 = math.exp(-r * T) * np.mean(hT) # 蒙地卡羅估計式

# 結果輸出
print('Value of the European call option %5.3f.' % C0)
```

你可以從本節的演算法範例看到，Python 及其語法非常適合支援經典科學語言雙人組—英語及數學。在科學語言組合中加入 Python 可讓這個組合更全面：

- 用**英語**來撰寫與討論科學與金融等問題。
- 用**數學**來簡潔、準確地描述與建構抽象層面、演算法、複數等。
- 用 **Python** 來建立科學模型與實作抽象層面、演算法、複數等。

數學與 Python 語法

除了 Python 之外，其他程式語言的語法幾乎都無法如此接近數學語法。因此，你通常可以輕鬆地將數學翻譯成 Python 實作，寫出數值演算法，並且在金融領域中，高效地運用 Python 來建立原型、進行開發，與維護程式碼。

有些領域也經常使用**虛擬碼**，因此我們可以加入第 4 個語言家族成員。虛擬碼可以用比較技術化的方式來表示（舉例）金融演算法，它除了接近數學表示法之外，也很接近技術實作。虛擬碼除了考慮演算法本身之外，也考慮電腦的運作方式。

之所以採取這種做法，原因是使用大部分的程式語言（編譯式的）寫出來的實作與正式的數學表示法“相去甚遠”。大多數的程式語言都必須加入許多只在技術上需要的元素，因此，我們很難看出數學與程式碼指的是同一個東西。

如今，人們經常以**虛擬碼風格**來使用 Python，因為它的語法幾乎與數學相同，因此可將技術“開銷”維持在最低程度。之所以可以做到這一點，是因為這種語言內建一些高階概念，這些概念有其優勢，但往往也伴隨著風險與（或）其他成本。 但是，我們可以肯定地說，在必要時，你也可以在使用 Python 時，遵守在其他語言中，從一開始就嚴格要求的實作與編寫方式。在這個意義上，Python 可以同時提供最棒的**高階抽象**與**嚴謹實作**。

用 Python 來提升效率與生產力

從高層次來看，使用 Python 的好處可以用三個維度來衡量：

效率

Python 如何幫助你更快得到結果，節省成本與時間？

生產力

Python 如何用同樣的資源（人員、資產等）完成更多事情？

品質

Python 如何幫助人們完成其他技術無法完成的工作？

我們無法全面地討論這些層面，但是，它們可以指出一些可作為起點的論點。

用更短的時間得到結果

互動式資料分析是比較容易看出 Python 效率的領域之一。這個領域從 IPython、Jupyter Notebook 與 pandas 程式包等強大的工具獲益良多。

假設有一位正在撰寫碩士論文的財金系學生，他對標普 500 指數很有興趣。他想要分析過去幾年的歷史指數值，看看該指數的波動性是如何隨著時間變動，並且希望證明它的波動性與一些典型的模型假設相反，是隨著時間震盪，絕對不是固定的。他也想要將結果視覺化。這位學生主要工作是：

- 從網路取得指數值資料
- 計算對數報酬率年化滾動標準差（波動率）

- 畫出指數價格資料與波動率結果

這些工作非常複雜，甚至在不久前，一般認為只有專業金融分析師可以處理它們。現在就連財金系學生都可以輕鬆地處理這種問題。下面的程式展示它是如何運作的，此時你還不需要操心語法的細節（後續章節會詳細解釋每一個地方）：

```
In [16]: import numpy as np  ❶
         import pandas as pd  ❶
         from pylab import plt, mpl  ❷

In [17]: plt.style.use('seaborn')  ❷
         mpl.rcParams['font.family'] = 'serif'  ❷
         %matplotlib inline

In [18]: data = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                            index_col=0, parse_dates=True)  ❸
         data = pd.DataFrame(data['.SPX'])  ❹
         data.dropna(inplace=True)  ❹
         data.info()  ❺
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
         Data columns (total 1 columns):
         .SPX    2138 non-null float64
         dtypes: float64(1)
         memory usage:33.4 KB

In [19]: data['rets'] = np.log(data / data.shift(1))  ❻
         data['vola'] = data['rets'].rolling(252).std() * np.sqrt(252)  ❼

In [20]: data[['.SPX', 'vola']].plot(subplots=True, figsize=(10, 6));  ❽
```

❶ 匯入 NumPy 與 pandas。

❷ 匯入 matplotlib 並且為 Jupyter 設置繪圖風格與方法。

❸ pd.read_csv() 可以讀取遠端或本地的逗號分隔值（CSV）資料集。

❹ 讀取資料的子集合，並移除 NaN（“not a number”）值。

❺ 展示資料集的一些詮釋資訊（metainformation）。

❻ 以向量化的方式計算對數報酬率（在 Python 層面上“看不到迴圈”）。

❼ 計算滾動年化波動率。

❽ 最後，畫出兩個時間序列。

圖 1-1 是這個簡短的互動式對話畫出來的圖表。不可思議的是，我們只要用幾行程式，就可以完成金融分析領域常見的複雜工作：收集資料、執行複雜且重複的數學計算，以及將結果視覺化。這個例子說明使用 pandas 來處理整個時間序列幾乎與處理浮點數運算一樣簡單。

轉換成專業的金融背景，這個例子意味著，一旦金融分析師使用正確的 Python 工具，以及擁有高階抽象的程式包，他們就可以把注意力放在他們的專業領域上，而不是複雜的技術上。分析師也可以更快速地做出反應，近乎即時地提供有價值的見解，確保領先競爭對手一步。這個**提高效率**的例子也可以輕鬆地轉換成可測量的底線效應（measurable bottom-line effects）。

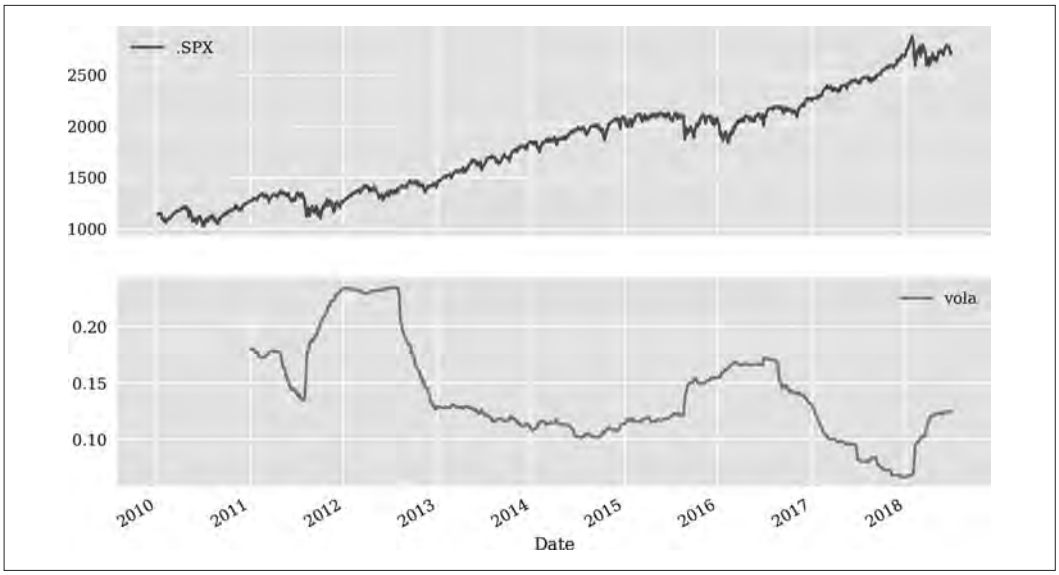

圖 1-1 標普 500 收盤價與年化波動率

確保高性能

一般認為 Python 的語法相當簡潔，用它來編寫程式的效率相對較高。但是，因為 Python 本身是一種直譯語言，人們經常認為 Python 處理計算密集的金融工作太慢了。事實上，當你使用某些實作方式時，Python 確實非常緩慢，但它**並非總是那麼慢**，它在幾乎任何一個應用領域都可以展現很高的性能。原則上，我們至少可以採取三種不同的策略來取得更好的性能：

採取慣用寫法與範式

一般來說，使用 Python 時，你可以用不同的做法來產生同樣的結果，但它們之間的性能特性有很大的不同，你“只要”選擇正確的方式（例如採取特定的實作法：正確地使用資料結構、避免迭代向量、或使用 pandas 之類的程式包），就可以明顯改善結果。

編譯

現在有許多性能程式包提供重要函式的已編譯版本，或靜態 / 動態地（在執行期或呼叫期）將 Python 程式碼編譯成機器碼。它們可讓這些函式的執行速度比單純使用 Python 程式碼快好幾個數量級；比較流行的程式包有 Cython 與 Numba。

平行化

許多計算工作，尤其是金融領域的，都可以透過平行執行來取得明顯的好處。平行執行對 Python 而言稀鬆平常，可以輕鬆地實作。

用 Python 來執行高性能計算

Python 本身不是一種高性能的計算技術。但是，Python 已經演變成一種理想的平台，可透過它來使用許多性能技術。也就是說，Python 已經變成一種使用高性能計算技術的膠水語言了。

本節舉一個簡單卻很實際的例子，這個例子涉及上述的三種策略（後續章節會詳細說明這些策略）。金融分析領域經常使用複雜的數學運算式來計算大型的數字陣列，Python 本身提供了這種工作所需的一切元素：

```
In [21]: import math
         loops = 2500000
         a = range(1, loops)
         def f(x):
             return 3 * math.log(x) + math.cos(x) ** 2
         %timeit r = [f(x) for x in a]
         1.59 s ± 41.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

在這個例子中，Python 直譯器大約用 1.6 秒來計算 f() 函式 2,500,000 次。我們也可以用 NumPy 完成同一項工作。NumPy 提供優化的（即預先編譯的）函式來處理這種陣列運算：

```
In [22]: import numpy as np
         a = np.arange(1, loops)
         %timeit r = 3 * np.log(a) + np.cos(a) ** 2
         87.9 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

使用 NumPy 可將執行時間顯著地減少至大約 88 毫秒。但是，還有一種程式包是專門為了處理這種工作而設計的，它稱為 numexpr，意思是 “numerical expressions”。它藉著編譯運算式來進一步改善 NumPy 功能的性能，例如，避免在記憶體內複製 ndarray 物件：

```
In [23]: import numexpr as ne
         ne.set_num_threads(1)
         f = '3 * log(a) + cos(a) ** 2'
         %timeit r = ne.evaluate(f)
         50.6 ms ± 4.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

這種更專用的方法可以進一步將執行時間減少至大約 50 毫秒。但是，numexpr 的內建功能也可以平行執行個別的運算。所以我們可以利用 CPU 的多執行緒：

```
In [24]: ne.set_num_threads(4)
         %timeit r = ne.evaluate(f)
         22.8 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

使用四個執行緒進一步將這個例子的執行時間減少至 23 毫秒以下，整體上，它將性能改善了 90 幾倍。特別要注意的是，進行這種改善時，你不需要修改基本的問題 / 演算法，也不需要知道任何關於編譯或平行化的細節，即使你不是專家，也可以在高層次上使用這些功能。只不過，在使用它們之前，你必須知道有哪些功能及選項可用。

這個例子展示 Python 提供許多選項，可讓你從既有的資源中得到更多好處，也就是**提升你的生產力**。當你執行平行化時，可以用同樣的時間完成循序法三倍的計算量—在這個例子中，你只要告訴 Python 使用多個 CPU 執行緒，而不是只有一個即可。

從原型製作到生產

從執行速度的角度來看，互動式分析的效率與性能絕對是 Python 值得考慮的兩大優點。然而，用 Python 來處理金融問題的另一個好處不容易在第一時間察覺，但經過仔細研究，你將發現它很有可能是金融機構的重大戰略要素。你可以端對端（**從原型製作階段到生產階段**）使用 Python。

現今全球的金融機構在進行金融開發時，通常採取分離式的雙步驟程序。一方面，他們讓**量化分析師**（“quants，寬客”）開發模型與製作技術原型，量化分析師喜歡使用 Matlab（*http://mathworks.com*）與 R（*https://www.r-project.org*）等工具與環境，來進

行快速、互動式應用程式開發。在這個開發階段，性能、穩定性、部署、訪問管理、版本控制等問題沒那麼重要，他們主要的目標是尋找概念的證明與（或）原型，來展示演算法或整體應用程式的主要期望功能。

完成原型之後，IT 部門及其*開發人員*接管工作，將既有的*原型程式*轉換成可靠的、易維護的、高性能的*生產程式*。這個階段通常會出現範式轉換（paradigm shift），因為他們通常使用 C++ 或 Java 等編譯語言來滿足部署與生產需求。他們通常也會在正式的開發過程中，使用專業工具、版本控制系統等等。

這種雙步驟的做法通常會產生一些意外的後果：

低效
: 原型碼無法重複使用、演算法必須實作兩次、浪費時間與資源的重複工作、轉換期間的風險

需要許多技能
: 不同的部門需要具備不同的技能，並且使用不同的語言來實作“同樣的東西”，他們不僅用不同的語言寫程式，也說著不同的語言

過時的程式碼
: 程式碼以不同的語言來提供與維護，通常使用不同的實作風格

但是使用 Python 可以將這個端對端流程*簡化*，包括最初的互動式原型製作，到寫出高度可靠、高效、易維護的生產程式。Python 可以簡化部門之間的溝通，也可以簡化員工培訓，因為公司只要使用一種主要語言，就可以涵蓋建構金融應用程式時的所有領域了。它也可以避免在不同的開發步驟中使用不同的技術造成的低效與麻煩。總之，Python 可以為金融分析、金融應用程式開發，以及演算法實作過程中的幾乎所有工作提供*一致的技術框架*。

數據驅動與 AI 優先金融

基本上，本書第一版在 2014 年所提出的關於技術與金融業的所有看法，在 2018 年 8 月修改本書的這一章時，看起來仍然非常實際且重要。不過，本節仍然要探討金融業的兩大趨勢，它們將會從根本上改造金融業，這兩大趨勢在過去幾年中已經成形了。