前言

結緣 Python

我初次接觸 Python 是在 2008 年末。那時接近大學畢業，我憑著在學校裡學到的微末 Java 知識四處求職。我從大學所在的城市南昌出發去了北京，借住在一位朋友的租屋處，他當時在巨鯨音樂網上班，用的主要程式設計語言正是 Python。

得知我正在尋找一份 Java 相關的工作，那位朋友跟我說：「寫 Java 程式有什麼用啊？ Python 比 Java 好玩多了，而且功能更強大，連 Google 都在用！」

在他的熱情「傳道」下，我對 Python 語言產生了好奇心，於是找了一份當時最流行的開源教程 *Dive into Python*，開始學起 Python。

老實說，之前在學校用 Java 和 C 語言的程式設計時，我很少體會到寫程式的快樂，也從未期待過自己將來要以寫程式為生。但神奇的是，在學了一些 Python 的基礎知識，並用它寫了幾個小工具後，我突然意識到原來自己很喜歡寫程式，並開始期待找到一份以 Python 為主要程式設計語言的開發工作 —— 也許這就是我和 Python 之間的緣分吧！

幸運的是，在當時的 CPyUG（中國 Python 社群）郵件群組裡，正好有一家南昌的公司在招募全職 Python 工程師。看到這個消息後，我立刻做出了決定：結束短暫的「北漂」生活，回到學校準備該職位的面試。後來，我成功通過了面試，最終在那家公司謀得了一份 Python 開發的實習工作，並從此開啟了後來十餘年的 Python 程式設計生涯。

為什麼寫這本書

回顧自己的工作經歷，我從中發現一件有意思的事：程式設計作為一項技能，或者說一門手藝，為新手帶來的「蜜月期」非常短暫。

一開始，我們對一門程式設計語言只是略懂皮毛，只要能用它實現想要的功能，就會非常開心。

如果再學會語言的一些高級用法，例如 Python 裡的裝飾器，把它應用在了專案程式裡，我們便整天笑得合不攏嘴。

但歡樂的時光總是特別短暫，一些類似的遭遇似乎總會不可避免地降臨到每個人頭上。

在接手了幾個被眾人稱為「坑」的老專案，或是親手寫了一些無人敢接手的程式後；在整日忙著修 bug，每寫一個新功能就引入三個新 bug 後……夜深人靜之時，坐在電腦前埋頭苦幹的我們總有那麼一些瞬間會突然意識到：程式設計最初帶給我們的快樂已悄然遠去，寫程式這件事現在變得有些痛苦。更誇張的是，一想到專案裡的爛程式，每天起床後最想做的一件事就是辭職。

造成上面這種困境的原因是多方面的，而其中最主要、最容易被我們直觀感受到的問題就是：爛的程式碼實在是太多了。後來，在親身經歷了許多個令人不悅的專案之後，我才慢慢看清楚：即便是兩個人實現同一個功能，最終結果看上去也一模一樣，但程式品質卻可能有著雲泥之別。

好的程式碼就像好文章，語言精練、層次分明，讓人讀了還想讀；而爛的程式碼則像糊成一團的義大利麵條，處處充斥著相似的邏輯，模組間的關係錯綜複雜，多看一眼都令人覺得眼睛會受傷。

在知道了「程式碼也分好壞」以後，我開始整天琢磨怎麼才能把程式碼寫得更好。我前前後後讀過一些書 —— 《程式大全》《重構》《設計模式》《程式整潔之道》 —— 毫無疑問，它們都是領域內首屈一指的經典好書，我從中學到了許多知識，至今受益匪淺。

這些領域內的經典圖書雖好，卻有個問題：它們大多是針對 Java 這類靜態型別語言所寫的，而 Python 這門動態型別的指令碼語言又和 Java 不太一樣。這些書裡的許多理念和例子，如果直接套用在 Python 裡，結果不盡如人意。

於是，話又說回來，要寫出好的 Python 程式碼，究竟得掌握哪些知識呢？在我看來，問題的答案可分為兩大部分。

- 第一部分：語言無關的基本知識，例如變數的命名原則、寫註解時的注意事項、寫條件分支語句的技巧，等等。這部分知識放之四海而皆準，可以運用在各種程式設計語言上，不僅在 Python 上。

- 第二部分：與 Python 語言高度相關的知識，例如自訂容器型態來改善程式碼、在正確的時機回傳例外狀況、活用生成器改善迴圈、用裝飾器設計純正的 API，等等。

當然，上面這種回答顯然過於簡陋，省略了太多細節。

為了更好地回答「如何寫出好的 Python 程式」這個問題，從 2016 年開始，我用業餘時間寫作了一系列相關的技術文章，命名為「Python 工匠」—— 正是這十幾篇文章構成了本書的骨架。此外，本書注重故事、注重案例的寫作風格也與「Python 工匠」系列一脈相承。

如果你也像我一樣，曾被爛程式碼所困，終日尋求寫好 Python 程式的方法，那麼我鄭重地將本書推薦給你。這是我多年的經驗彙集，相信會給你一些啟發。

目標讀者

本書適合以下類型的人閱讀：

- 以 Python 為主要開發語言的工程師
- 工作中需要寫一些 Python 程式的工程師
- 有其他語言程式設計經驗、想學習如何寫出優質 Python 程式的工程師
- 任何愛好程式設計、喜歡 Python 語言的讀者

本書內容以進階知識為主。書裡雖有少量基礎知識講解，但並不全面，描述得也並不詳盡。正因如此，如果你從未有過任何程式設計經驗，我並不建議你透過本書來入門 Python。

在 Python 入門學習方面，我推薦由人民郵電出版社圖靈公司出版的《Python 程式設計：從入門到實踐》。當你對 Python 有了一些瞭解、打好基礎後，再回過頭來閱讀本書，相信彼時你可以獲得更好的閱讀體驗。

程式運作環境

如果沒有特殊說明，書中所有的 Python 程式都是採用 Python3.8 版本編寫的。

結構與特色

❖ 五大部分

全書共計 13 章，按內容特色可歸納為五大部分。

第一部分　變數與基礎型別　由第 1 章、第 2 章和第 3 章組成。在學習一門程式語言的過程中，「如何操作變數」和「如何使用基礎型態」是兩個非常重要的知識點。透過學習這部分內容，你會知道如何善用變數來改善程式碼品質，掌握數值、字串及內建容器型態的使用技巧，避開常見盲點。

第二部分　語法結構　由第 4 章、第 5 章和第 6 章組成。條件分支、例外處理和迴圈語句是 Python 最常見的三種語法結構。它們雖然基礎，但很容易被誤用，從而變成爛程式的幫兇。本部分內容會帶你深入這三種語法結構，教你掌握如何用它們簡潔而精準地表達邏輯，寫出高品質的程式碼。

第三部分　函式與裝飾器　由第 7 章和第 8 章組成。函式是 Python 語言最重要的組成元素之一。正是因為有了函式，我們才獲得了有效率使用程式的能力。而裝飾器則可簡單視為基於函式的一種特殊物件 —— 它始於函式，但又不止於函式。這兩章介紹了許多與函式和裝飾器有關的「精華」，掌握它們，可以讓你在寫程式時事半功倍。

第四部分　物件導向程式設計　由第 9 章、第 10 章和第 11 章組成。眾所皆知，Python 是一門物件導向程式設計語言，因此「物件導向技術」自然是 Python 學習路上的重中之重。第 9 章圍繞 Python 語言的物件導向基礎概念和高級技巧展開。第 10 章和第 11 章則是為大家量身定制的物件導向設計進階知識。

第五部分　總結與延伸　由第 12 章和第 13 章組成。這部分內容可以看作是全書內容的總結和延伸。第 12 章匯總本書出現過的所有與「Python 物件模型」相關的知識點，並闡述它們與編寫優雅程式碼之間的重要關係。而最後的第 13 章則是一些與大型專案開發相關的經驗之談。

❖ 三大區塊

除了第 11 章和第 13 章等少數幾個純案例章節以外，其他章節都包含**基礎知識**、**案例故事**、**程式設計建議**三個常駐區塊。

其中，**基礎知識**部分涵蓋和該章主題有關的基礎知識內容。舉例來說，在第 6 章的**基礎知識**部分，你會學習有關迭代器與可迭代型態的基礎知識。不過，需

要提醒各位的是，本書中的基礎知識講解並不追求全面，僅包含筆者基於個人經驗挑選並認為比較關鍵的知識點。

如果說本書的基礎知識部分與其他同類書的內容大同小異，那麼**案例故事**與**程式設計建議**則是將本書與其他 Python 程式設計類圖書區分開來的關鍵。

在每一個**案例故事**部分，你會讀到一個或多個與該章主題相關的故事。例如，第 1 章講述了一位 Python 工程師去某公司參加面試的故事，讀完它，你會體會「變數與註解」究竟是如何影響了故事主人公的面試結果，最終深刻理解兩者是如何塑造我們對程式的第一印象的。

程式設計建議部分主要包含一些與該章節主題相關的建議。例如在第 4 章中，我一共介紹了 7 條與條件分支有關的建議。雖然內容包羅萬象，但書中的所有程式設計建議都是圍繞「如何寫好程式」這件事展開的。例如，我會建議你儘量刪除分支裡重複的程式碼、避開 or 運算子的陷阱，等等。

除了第 10 章與第 11 章同屬一個主題，有先後順序以外，本書的每一章都是獨立的。你可以隨意挑選自己感興趣的章節開始閱讀。

❖ 13 章內容

以下為各章內容要點。

第 1 章　變數與註解　要寫出一份品質良好的程式，運用好變數與註解不是加分項，而是必選項。在本書的開篇章，你將學習包括動態拆解等的一些 Python 變數的常見用法，瞭解編寫程式碼註解的幾項基本原則。而本章的案例故事「奇怪的冒泡排序演算法」，是全書趣味性最強的幾個故事之一，請一定不要錯過。

第 2 章　數值與字串　本章內容圍繞 Python 中最基礎的兩個資料型態展開。在基礎知識部分，我們會學習一些與數值和字串有關的基本操作。在案例故事部分，你會見到一個與程式可讀性有關的案例。在程式設計建議模組，你會學到一些與 Python 位元組相關的語言底層知識。

第 3 章　容器型態　由於 Python 語言的容器型態豐富，因此本章是全書篇幅最長的章節之一。在基礎知識部分，除了介紹每種容器的基本操作，我還會講解包括可變性、可雜湊性、深層複製與淺層複製在內的 Python 語言裡的許多重要概念。在案例故事部分，你會讀到一個與自訂容器型態相關的重構案例。

第 4 章　條件分支流程控制　條件分支是個讓人又愛又恨的東西：少了它，許多邏輯根本無法表達；而一旦被濫用，程式碼就會變得不堪入目。透過本章，你會學到在 Python 中編寫條件分支語句的一些常用技巧。在案例故事部分，我會說明有些條件分支語句其實沒必要存在，借助一些工具，我們甚至能讓它們完全消失。

第 5 章　例外與錯誤處理　例外就像數值和字串一樣，是組成 Python 語言的重要物件之一。在本章中，你需要先徹底搞清楚為什麼要在 Python 程式中多使用例外。之後，你會邂逅兩個與例外有關的案例故事，其中一個是我的親身工作經歷。

第 6 章　迴圈與可迭代物件　迴圈也許是所有程式設計語言裡最為重要的控制結構。要寫好 Python 裡的迴圈，不光要掌握迴圈語法本身，還得對迴圈的最佳拍檔 —— 可迭代物件了然於胸。在本章的基礎知識部分，我會詳細介紹可迭代物件的相關知識。

第 7 章　函式　Python 中的函式與其他程式設計語言裡的函式很相似，但又有著些許獨特之處。在本章中，你會學習與函式有關的一些常見技巧，例如：為何不應該用可變型態作為參數預設值、何時該用 None 作為回傳值，等等。案例故事會展示有趣的程式設計挑戰題，透過故事主人公的解題經歷，你會掌握為函式增加狀態的三種方式。在程式設計建議部分，你會讀到一份腳本案例程式碼，它完整詮釋了抽象級別對於函式的重要性。

第 8 章　裝飾器　裝飾器是一個獨特的 Python 語言特性。利用裝飾器，你可以實現許多既優雅又實用的工具。本章的基礎知識部分非常詳細，教你掌握如何建立幾種常用的裝飾器，例如用類別實現的裝飾器、使用可選參數的裝飾器等。在程式設計建議部分，我會展示裝飾器的一些常見使用場景，分析裝飾器的獨特性所在。相信學完本章內容之後，你一定可以變身為裝飾器方面的高手。

第 9 章　物件導向程式設計　Python 是一門物件導向程式設計語言，因此，好的 Python 程式碼離不開設計優良的類別和物件。在這一章中，你會讀到一些與 Python 類別有關的常用知識，例如什麼是類別方法、什麼是靜態方法，以及何時該使用它們等。此外，在本章的基礎知識部分，你還會詳細瞭解鴨子型別的由來，以及抽象類別如何影響了 Python 的型態系統。本章的案例故事是一個與類別繼承有關的長篇故事。它會告訴你為什麼繼承是一把雙刃劍，以及如何才能避開由繼承帶來的問題。

第 10 章和第 11 章　物件導向設計原則　要寫出好的物件導向程式，經典的 SOLID 設計原則是我們學習路上的必經之地。在這兩章裡，我會透過一個大的程式設計實戰專案詮釋 SOLID 原則的含義。透過學習這部分內容，你會掌握如何將 SOLID 原則運用到 Python 程式中。

第 12 章　資料模型與描述符　資料模型是最重要的 Python 進階知識，最重要之一。恰當地運用資料模型是寫出優值 Python 程式的關鍵所在。本章一開始會簡單回顧書中出現過的所有資料模型知識。在基礎知識部分，我會對運算子重新做一些簡單介紹。在案例故事部分，你會讀到一個與資料模型和集合型態有關的有趣故事。

第 13 章　開發大型專案　如何開發好一個大型專案，是個非常龐大的話題。在本章中，我精選了一些與之相關的重要主題，例如，在大型專案中使用哪些工具，能讓專案夥伴間的合作事半功倍，提升每個人的開發效率。在此之後，我會介紹兩個常用的 Python 單元測試工具。本章最後介紹了為大型專案編寫單元測試的 5 條建議。希望這些內容對你有所啟發。

圖示說明

本書使用以下圖示說明。

CHAPTER

1

變數與註解

程式設計是一個透過程式來表達思想的過程。聽上去蠻神秘，但我們早就做過類似的事情 —— 當年在小學課堂上寫出第一篇 500 字的作文，同樣也是在表達思想，只是二者方式不同，作文用的是詞語和句子，而程式設計用的是程式。

但程式設計與作文之間也有相通之處，程式中裡也有許多「詞語」和「句子」。大部分的變數名是詞語，而大多數註解本身就是句子。當我們看到一段程式時，最先注意到的，不是程式有幾層迴圈，用了什麼模式，而是**變數與註解**，因為它們是程式裡最接近自然語言的東西，最容易被大腦消化、理解。

正因如此，如果作者在編寫變數和註解時含糊不清、語焉不詳，其他人將很難搞清楚程式的真實意圖。就拿下面這行程式碼來說：

```
#  去掉 s 兩邊的空格，再處理
value = process(s.strip())
```

你能告訴我這段程式碼在做什麼嗎？當我看到它時，是這麼想的：

- 在 s 上呼叫 strip()，所以 s 可能是一個字串？不過為什麼要去掉兩邊的空格呢？
- process(...)，顧名思義，「處理」了一下 s，但具體是什麼處理呢？
- 處理結果賦值給了 value，value 代表「值」，但「值」又是什麼？
- 開頭的註解就更別提了，它說的就是程式碼本身，對理解程式沒有絲毫的幫助。

最後的結論是：「將一個可能是字串的東西兩端的空格去掉，然後處理一下，最後賦值給某個不明物體。」我只能理解到這種程度了。

但同樣是這段程式，如果我稍微調整一下變數的名字，加上一點點註解，就會變得截然不同：

```
#  使用者輸入可能會有空格，使用 strip() 去掉空格
username = extract_username(input_string.strip())
```

新程式碼讀上去是什麼感覺？是不是程式意圖變得容易理解多了？這就是變數與註解的魔力。

從電腦的角度來看，**變數（variable）**是用來從記憶體找到某個東西的標記。它叫「小明」「小華」還是「張三」「李四」，都無所謂。註解同樣如此，電腦一點都不關心你的註解寫得是否通順，用詞是否準確，因為它在執行程式時會忽略所有的註解。

但正是這些對電腦來說無關痛癢的東西，直接決定了人們對程式的「第一印象」。好的變數和註解並非為電腦而寫，而是為每個閱讀程式的人而寫（當然也包括你自己）。變數與註解是作者表達思想的基礎，是讀者理解程式的第一道門，它們對程式品質的貢獻毋庸置疑。

本章將對 Python 裡的變數和註解做簡單介紹，我會分享一些常用的變數命名原則，介紹編寫程式碼註解的幾種方式。在程式設計建議部分，我會列舉一些與變數和註解有關的好習慣。

我們從變數和註解開始，學習如何寫出給人留下美好「第一印象」的好程式吧！

1.1 基礎知識

本節將介紹一些與變數和註解相關的基礎知識。

1.1.1 變數常見用法

在 Python 裡，宣告一個變數特別簡單：

```
>>> author = 'piglei'
>>> print('Hello, {}!'.format(author))
Hello, piglei!
```

因為 Python 是一門動態型別的語言，所以我們不需先宣告變數型態，直接對變數賦值即可。

你也可以在一行語句裡同時操作多個變數，例如交換兩個變數所指向的值：

```
>>> author, reader = 'piglei', 'raymond'
>>> author, reader = reader, author ❶
>>> author
'raymond'
```

❶ 交換兩個變數

1. 變數開箱

變數開箱（unpacking）是 Python 裡的一種特殊賦值操作，允許我們把一個可迭代物件（例如串列）的所有成員，一次性賦值給多個變數：

```
>>> usernames = ['piglei', 'raymond']
#  注意：左側變數的個數必須和待展開的串列長度相等，否則會出現錯誤
>>> author, reader = usernames
>>> author
'piglei'
```

如果在設定陳述式左側添加小括弧（...），甚至可一次展開多層巢狀資料：

```
>>> attrs = [1, ['piglei', 100]]
>>> user_id, (username, score) = attrs
>>> user_id
1
>>> username
'piglei'
```

除了上面的普通開箱外，Python 還支援更靈活的動態開箱語法。只要用星號運算式（*variables）作為變數名，它便會貪婪[1]地捕捉多個值物件，並將捕捉到的內容作為串列賦值給 variables。

例如，下面 data 串列裡的資料就分為三段：頭為使用者，尾為分數，中間的都是水果名稱。透過把 *fruits 設定為中間的開箱變數，我們就能一次性開箱所有變數 —— fruits 會捕捉 data 去頭去尾後的所有成員：

```
>>> data = ['piglei', 'apple', 'orange', 'banana', 100]
>>> username, *fruits, score = data
>>> username
'piglei'
```

1 「貪婪」一詞在電腦領域具有特殊含義。比方說，某個行為要捕捉一批物件，它既可以選擇捕捉 1 個，也可以選擇捕捉 10 個，兩種做法都合法，但它總是選擇結果更多的那種：捕捉 10 個，這種行為就稱得上是「貪婪」。

```
>>> fruits
['apple', 'orange', 'banana']
>>> score
100
```

和常規的切片設定陳述式比起來，動態開箱語法要直觀許多：

```
# 1. 動態開箱
>>> username, *fruits, score = data
# 2. 切片賦值
>>> username, fruits, score = data[0], data[1:-1], data[-1]
# 兩種變數賦值方式完全相同
```

上面的變數開箱操作也可以在任何迴圈語句裡使用：

```
>>> for username, score in [('piglei', 100), ('raymond', 60)]:
... print(username)
...
piglei
raymond
```

2. 單底線變數名 _

在常用的諸多變數名中，單底線 _ 是比較特殊的一個。它常作為一個無意義的預留位置出現在賦值的語句中。_ 這個名字本身沒什麼特別之處，這算是大家約定俗成的一種用法。

舉個例子，如果想在開箱賦值時忽略某些變數，就可以使用 _ 作為變數名：

```
#  忽略展開時的第二個變數
>>> author, _ = usernames

#  忽略第一個和最後一個變數之間的所有變數
>>> username, *_, score = data
```

而在 Python 互動式命令列（直接執行 python 命令進入的互動環境）裡，_ 變數還有一層特殊含義 —— 默認儲存我們輸入的上個運算式的回傳值：

```
>>> 'foo'.upper()
'FOO'
>>> print(_) ❶
FOO
```

❶ 此時的 _ 變數儲存著上一個 .upper() 運算式的結果

1.1.2 為變數宣告型態

前面說過，Python 是動態型別語言，使用變數時不需要做任何型態宣告。在我看來，這是 Python 相比其他語言的一個重要優勢：它減少了我們的心智負擔，讓寫程式變得更容易。尤其對於許多程式新手來說，「不用宣告型態」無疑會讓學 Python 這件事變得簡單很多。

但任何事物都有其兩面性。動態型別所帶來的缺點是程式的可讀性會因此大打折扣。

試著讀讀下面這段程式：

```
def remove_invalid(items):
    """ 刪除 items 裡面無效的元素 """
    ... ...
```

你能告訴我，函式接收的 items 參數是什麼型態嗎？是一個裝滿數字的串列，還是一個裝滿字串的集合？只看上面這些程式碼，我們根本無從得知。

為了解決動態型別帶來的可讀性問題，最常見的辦法就是在說明字串（docstring）裡做文章。我們可以把每個函式參數的型態與說明全都寫在說明字串裡。

下面是增加了 Python 官方推薦的 Sphinx 格式文件後的結果：

```
def remove_invalid(items):
    """ 刪除 items 裡面無效的元素

    :param items: 待刪除物件
    :type items: 包含整數的串列，[int, ...]
    """
```

在上面的說明字串裡，我用 :type items: 註明 items 是個整數型態串列。任何人只要讀到這份文件，馬上就能知道參數型態，不用再猜來猜去。

當然，標註型態的辦法肯定不止上面這一種。在 Python 3.5 版本[2]以後，你可以用型態註解功能來直接註明變數型態。相比編寫 Sphinx 格式文件，我其實更推薦使用型態註解，因為它是 Python 的內建功能，而且正在變得越來越流行。

2 具體來說，針對變數的型態註解語法是在 Python 3.6 版本引入的，而 3.5 版本只支援註解函式參數。

要使用型態註解，只需在變數後添加型態，並用冒號隔開即可，例如 `func(value: str)` 表示函式的 `value` 參數為字串型態。

下面是為 `remove_invalid()` 函式添加型態註解後的樣子：

```
from typing import List

def remove_invalid(items: List[int]): ❶
    """ 刪除 items 裡面無效的元素 """
    ... ...
```

❶ `List` 表示參數為串列型態，`[int]` 表示裡面的成員是整數型態

「型態註解」只是一種有關型態的註解，不提供任何驗證功能。要驗證型態正確性，需要使用其他靜態型態檢查工具（如 mypy 等）。

平心而論，不管是編寫 Sphinx 格式文件，還是添加型態註解，都會增加編寫程式的工作量。同樣一段程式，標註變數型態比不標註一定要花費更多時間。

但從我的經驗來看，這些額外的時間投入，會帶來非常豐厚的回報：

- 程式碼更易讀，讀程式碼時可以直接看到變數型態。
- 大部分的現代化 IDE[3] 會讀取型態註解資訊，提供更智慧的輸入提示。
- 型態註解配合 mypy 等靜態型態檢查工具，能提升程式碼正確性（13.1.5 節）。

因此，我強烈建議在**多人參與的中大型 Python 專案**裡，至少使用一種型態註解方案 —— Sphinx 格式文件或官方型態註解都行。能直接看到變數型態的程式，總是會讓人更安心。

在 10.1.1 節中，你會看到更詳細的「型態註解」功能說明，以及更多使用了型態註解的程式碼。

3 IDE 是 integrated development environment（整合式開發環境）的縮寫，在滿足程式編輯的基本需求外，IDE 通常還集成了許多方便開發者的功能。常見的 Python IDE 有 PyCharm、VS Code 等。

1.1.3 變數命名原則

如果要從變數著手來破壞程式品質，辦法多到數也數不清，例如宣告了變數但是不用，或者宣告 100 個全域變數，等等。但如果要在這些辦法中選出破壞力最強的那個，非「幫變數胡亂命名」莫屬。

下面這段程式就是一個充斥著壞名字的「集大成」者。試著讀讀看有什麼感受：

```
data1 = process(data)
if data1 > data2:
    data2 = process_new(data1)
    data3 = data2
return process_v2(data3)
```

是不是想破頭都看不懂它要做什麼？壞名字對程式品質的破壞力可見一斑。

那麼問題來了，既然大家都知道上面這樣的程式碼不好，為何在程式世界裡，每天都有類似的程式碼被寫出來呢？我猜這是因為幫變數取個好名字真的很難。在電腦科學領域，有一句廣為流傳的「格言」：

> 電腦科學領域只有兩件難事：快取失敗和命名。
>
> —— *Phil Karlton*

這句話裡雖然一半嚴肅一半玩笑，但「命名」有時真的難到讓人抓狂。我常常呆坐在螢幕前，抓耳撓腮好幾分鐘，就是沒辦法為變數想出一個合適的名字。

要為變數取個好名字，主要靠的是經驗，有時還需加上一些靈感，但更重要的是遵守一些基本原則。下面就是我總結的幾條變數命名的基本原則。

1. 遵循 PEP 8 原則

為變數命名主要有兩種流派：一是透過大小寫界定單詞的駝峰命名派 CamelCase，二是透過底線連接的蛇形命名派 snake_case。這兩種流派沒有明顯的優劣之分，而是與個人喜好有關。

為了讓不同開發者寫出的程式風格儘量保持統一，Python 制定了官方的程式風格指南：PEP 8。這份風格指南裡有許多詳細的風格建議，例如應該用 4 個空格縮排，每行不超過 79 個字元，等等。其中當然也包含變數的命名規範：

- 對於普通變數，使用蛇形命名法，例如 max_value。
- 對於常數，採用全大寫字母，使用底線連接，例如 MAX_VALUE。

- 如果變數標記為「僅內部使用」，為其增加底線首碼，例如 `_local_var`。
- 當名字與 Python 關鍵字衝突時，在變數末尾追加底線，例如 `class_`。

除變數名以外，PEP 8 中還有許多其他命名規範，例如類別名稱應該使用駝峰風格（`FooClass`）、函式應該使用蛇形風格（`bar_function`），等等。為變數命名的第一條原則，就是一定要在格式上遵循以上規範。

PEP 8 是 Python 程式風格的事實標準。「程式碼符合 PEP 8 規範」應該作為對 Python 工程師的基本要求之一。如果一份程式碼的風格與 PEP 8 大相徑庭，就基本不必繼續討論它優雅與否了。

2. 描述性要足夠

寫作過程中的一項重要工作，就是為句子斟酌恰當的詞語。不同詞語的描述性強弱不同，例如「冬天的梅花」就比「花」的描述性更強。而變數名和普通詞語一樣，同樣有描述性強弱之分，如果程式碼大量使用描述性弱的變數名，讀者就很難理解程式碼的含義。

本章開頭的那兩段程式碼可以很好地解釋這個問題：

```
#  描述性弱的名字：看不懂在做什麼
value = process(s.strip())

#  描述性強的名字：嘗試從使用者輸入裡解析出一個使用者名名稱
username = extract_username(input_string.strip())
```

所以，在可接受的長度範圍內，變數名所指向的內容描述得越精確越好。表 1-1 是一些具體的例子。

表 1-1　描述性弱和描述性強的變數名範例

描述性弱的名字	描述性強的名字	說明
`data`	`file_chunks`	`data` 泛指所有的「資料」，但如果資料是來自檔案的小區塊，我們可以直接叫它 `file_chunks`
`temp`	`pending_id`	`temp` 泛指所有「臨時」的東西，但其實它存放的是一個等待處理的資料 ID，因此直接叫它 `pending_id` 更好

描述性弱的名字	描述性強的名字	說明
result(s)	active_member(s)	result(s) 經常用來表示函式執行的「結果」，但如果結果就是指「活躍會員」，那還是直接叫它 active_member(s) 吧

看到表 1-1 中的舉例，你可能會想：「也就是說左邊的名字都不好，永遠別用它們？」

當然不是這樣。判斷一個名字是否合適，一定要結合它所在的場景，脫離場景談名字是片面的，是沒有意義的。因此，在「說明」這一欄中，我們強調了這個判斷所適用的場景。

而在其他一些場景下，這裡「描述性弱」的名字也可能是好名字，例如把一個數學公式的計算結果叫作 value，就非常恰當。

3. 要儘量簡短

剛剛說到，變數名的描述性要儘量足夠，但描述性越強，通常名字也就越長（不信再看看表 1-1，第二欄的名字就比第一欄長）。如果不加思考地實作「描述性原則」，那你的程式碼裡可能會充斥著 how_many_points_needed_for_user_level3 這種名字，簡直像條蛇一樣長：

```
def upgrade_to_level3(user):
    """  如果積分滿足要求，將使用者升級到等級 3"""
    how_many_points_needed_for_user_level3 = get_level_points(3)
    if user.points >= how_many_points_needed_for_user_level3:
        upgrade(user)
    else:
        raise Error('積分不夠，必須要 {} 分 '.format(how_many_points_needed_for_
        user_level3))
```

如果一個特別長的名字重複出現，讀者不會認為它足夠精確，反而會覺得囉唆難讀。既然如此，怎麼才能在保證描述性的前提下，讓名字儘量簡短易讀呢？

我認為之中訣竅在於：為變數命名要結合程式情境和上下文。例如在上面的程式碼裡，upgrade_to_level3(user) 函式已經透過自己的名稱、文件表明了其目的，那在函式內部，我們完全可以把 how_many_points_needed_for_user_level3 直接刪減成 level3_points。

即使沒用特別長的名字，相信讀程式的人也肯定能明白，這裡的 level3_points 指的就是「升到等級 3 所需要的積分」，而不是其他含義。

到底多長的名字算是太長呢？我的經驗是儘量不要超過 4 個單詞。

4. 要匹配型態

雖然變數無須宣告型態，但為了提升可讀性，我們可以用型態註解語法為其加上型態。不過現實很殘酷，到目前為止，大部分 Python 專案沒有型態註解[4]，因此當你看到一個變數時，除了透過上下文猜測，沒法輕易知道它是什麼型態。

但是，對於變數名和型態的關係，通常會有一些「直覺上」的約定。如果在命名時遵守這些約定，就可以建立變數名和型態間的匹配關係，讓程式碼更容易理解。

❖ 匹配布林數值型態的變數名

布林值（bool）是一種很簡單的型態，它只有兩個可能的值：「是」（True）或「不是」（False）。因此，為布林值變數命名有一個原則：一定要讓讀到變數的人覺得它只有「肯定」和「否定」兩種可能。舉例來說，is、has 這些非黑即白的詞就很適合用來修飾這類名字。

表 1-2 中提供了一些更詳細的例子。

表 1-2　布林值變數名舉例

變數名	含義	說明
is_superuser	是否是超級使用者	是 / 不是
has_errors	有沒有錯誤	有 / 沒有
allow_empty	是否允許空值	允許 / 不允許

4　相比之下，型態註解在開源領域的接受度更高一些，許多流行的 Python 開源專案（例如 Web 開發框架 Flask 和 Tornado 等），很早就為程式加上了型態註解。

❖ 匹配 int/float 型態的變數名

當人們看到和數字有關的名字時，自然就會認定它們是 int 或 float 型態。這些名字可簡單分為以下幾種常見類型：

- 釋義為數字的所有單詞，例如 port（埠號）、age（年齡）、radius（半徑）等。
- 使用以 _id 結尾的單詞，例如 user_id、host_id。
- 使用以 length/count 開頭或者結尾的單詞，例如 length_of_username、max_length、users_count。

> ⚠ 最好別拿一個名詞的複數形式來當作 int 型態的變數名，例如 apples、trips 等，因為這類名字容易與那些裝著 Apple 和 Trip 的普通容器物件（List[Apple]、List[Trip]）混淆，建議用 number_of_apples 或 trips_count 這類複合詞來當作 int 型態的名字。

❖ 匹配其他型別的變數名

至於剩下的字串（str）、串列（list）、字典（dict）等其他數值型別，我們很難歸納出一個「由名字猜測型態」的統一公式。拿 headers 這個名字來說，它既可能是一個裝滿標頭資訊的串列（List[Header]），也可能是一個包含標頭資訊的字典（Dict[str, Header]）。

對於這些數值型別，強烈建議使用我們在 1.1.2 節中提到的方案，在程式中明確標註它們的型態詳情。

5. 超短命名

在眾多變數名裡，有一類非常特別，那就是只有一兩個字母的短名字。這些短名字一般可分為兩類，一類是那些大家約定俗成的短名字，例如：

- 陣列索引三劍客 i、j、k
- 某個整數 n
- 某個字串 s
- 某個例外 e
- 檔案物件 fp

我並不反對使用這類短名字，我自己也經常用，因為它們寫起來的確很方便。但如果條件允許，建議儘量用更精確的名字替代。例如，在表示使用者輸入的字串時，用 input_str 替代 s 會更明確一些。

另一類短名字，則是對一些其他常用名的縮寫。例如，在使用 Django 框架做國際化內容翻譯時，常常會用到 gettext 方法。為了方便，我們常把 gettext 縮寫成 _：

```
from django.utils.translation import gettext as _

print(_('待翻譯文字'))
```

如果你的程式中有一些長名字反覆出現，可以效仿上面的方式，為它們設定一些短名字作為別名。這樣可以讓程式碼變得更緊湊、更易讀。但同一個專案內的超短縮寫不宜太多，否則會適得其反。

其他技巧

除了上面這些規則外，下面再分享幾個為變數命名的小技巧：

- 在同一段程式內，不要出現多個相似的變數名，例如同時使用 users、users1、users3 這種序列；
- 可以嘗試換詞來簡化複合變數名，例如用 is_special 來代替 is_not_normal；
- 如果你苦思冥想都想不出一個合適的名字，請打開 GitHub[5]，到其他人的開源專案裡找找靈感吧！

1.1.4 註解基礎知識

註解（comment）是程式非常重要的組成部分。通常來說，註解泛指那些不影響程式實際行為的文字，它們主要起額外說明作用。

Python 裡的註解主要分為兩種，一種是最常見的程式內註解，透過在行首輸入 # 號來表示：

5 世界上規模最大的開源專案原始碼託管網站。

```
# 使用者輸入可能會有空格，使用 strip 去掉空格
username = extract_username(input_string.strip())
```

當註解包含多行內容時，同樣使用 # 號：

```
# 呼叫 strip() 去掉空格的好處：
# 1. 資料庫儲存時佔用空間更小
# 2. 不必因為使用者多打了一個空格而要求使用者重新輸入
username = extract_username(input_string.strip())
```

除使用 # 的註解外，另一種註解則是我們前面看到過的說明字串（docstring），這些文件也稱**介面註解**（interface comment）。

```
class Person:
    """ 人

   :param name: 姓名
   :param age: 年齡
   :param favorite_color: 最喜歡的顏色
   """

   def __init__(self, name, age, favorite_color):
      self.name = name
      self.age = age
      self.favorite_color = favorite_color
```

介面註解有好幾種流行的風格，例如 Sphinx 文件風格、Google 風格等，其中 Sphinx 文件風格目前應用得最為廣泛。上面的 `Person` 類別的介面註解就屬於 Sphinx 文件風格。

雖然註解一般不影響程式的執行結果，卻會極大地影響程式碼的可讀性。在編寫註解時，程式新手們常常會犯同型態的錯誤，以下是我整理的最常見的 3 種。

1. 用註解遮罩程式碼

有時候，人們會把註解當作臨時遮罩程式碼的工具。當某些程式碼暫時不需要執行時，就把它們都註解了，未來需要時再取消註解。

```
# 原始碼裡有大段大段暫時不需要執行的程式碼
# trip = get_trip(request)
# trip.refresh()
# ... ...
```

其實根本沒必要這麼做。這些被臨時註解掉的大段內容，對於閱讀程式碼的人來說是一種干擾，沒有任何意義。對於不再需要的程式碼，我們應該直接把它們刪掉，而不是註解掉。如果未來有人真的需要用到這些舊程式碼，他直接去 Git 倉庫歷史裡就能找到，畢竟版本控制系統就是專門做這個的。

2. 用註解複述程式碼

在編寫註解時，新手常犯的另一類錯誤是用註解複述程式碼。就像這樣：

```
# 呼叫 strip() 去掉空格
input_string = input_string.strip()
```

上面程式碼裡的註解完全是冗餘的，因為讀者從程式碼本身就能讀到註解裡的資訊。好的註解應該像下面這樣：

```
# 如果直接把帶空格的輸入傳遞到後端處理，可能會造成後端服務崩潰
# 因此使用 strip() 去掉首尾空格
input_string = input_string.strip()
```

註解作為程式之外的說明性文字，應該儘量提供那些讀者無法從程式碼裡讀出來的資訊。描述程式**為什麼**要這麼做，而不是簡單複述程式碼本身。

除了描述「為什麼」的解釋性註解外，還有一種註解也很常見：**指引性註解**。這種註解並不直接複述程式，而是簡明扼要地概括程式碼功能，起到「程式碼導讀」的作用。

例如，以下程式碼裡的註解就屬於指引性註解：

```
# 初始化存取服務的 client 物件
token = token_service.get_token()
service_client = ServiceClient(token=token)
service_client.ready()

# 呼叫服務取得資料，然後進行過濾
data = service_client.fetch_full_data()
for item in data:
    if item.value > SOME_VALUE:
        ...
```

指引性註解並不提供程式碼裡讀不到的東西 —— 如果沒有註解，耐心讀完所有程式碼，你也能知道程式做了什麼事。指引性註解的主要作用是降低程式碼的認知成本，讓我們能更容易理解程式碼的意圖。