前言

2018 年 1 月時，我準備教授一門針對沒有程式設計經驗的學生的程式設計課程。當時我想使用 Julia，卻發現坊間沒有以 Julia 作為學習第一種程式語言的教科書。雖然確實有一些教學課程解釋了 Julia 的基本概念，但它們都沒有針對如何以程式設計師的角度來進行思考。

我讀過 Allen B. Downey 的著作《*Think Python*》，書中包括了學好程式設計的所有關鍵要素。然而此書是以 Python 為本的。我的講義最初的草稿是參考許多著作的融合版本，但是我發現課上得愈久，內容和《*Think Python*》的章節竟然愈來愈相似。不久，我便有了將此書改寫成 Julia 版本的念頭。

本書所有素材都是以 Jupyter 筆記本的方式放在 Github 上。當我在 Julia Discourse 網站上公布我的課程進度後，引起極為熱烈的反應。一本以 Julia 作為學習程式設計的第一種語言的書顯然彌補了 Julia 世界遺失的一環。我詢問 Allen 是否可以將《*Think Python*》正式改版為 Julia 時，他立即回覆：“動手吧！”。他請我聯繫歐萊禮的編輯，一年後我終於完成了此書。

不過本書命運多舛。2018 年 8 月 Julia 1.0 版發行。和其他 Julia 的程式設計師一樣，我必須要更新我的程式碼。本書的所有範例在轉換為與歐萊禮相容的 AsciiDoc 檔案時，都已經重新進行測試。工具鏈（toolchain）和範例程式碼都必須與 Julia 1.0 版相符。還好 8 月不用上課…

我希望您閱讀此書時是一種享受，並期待您能學會如何像程式設計師一樣的寫程式及進行思考。

— *Ben Lauwens*

為什麼要使用 Julia ？

Julia 在 2012 年由 Alan Edelman、Stefan Karpinski、Jeff Bezanson 和 Viral Shah 等人所推出。它是一種免費的開放原始碼（open source）程式語言。

程式語言的選擇是主觀的。對我而言，下列 Julia 的特性特別重要：

- Julia 是為了高效能運算所開發的。
- Julia 使用多重分派，讓設計師可以選擇適用於應用程式的樣式（pattern）。
- Julia 為動態型別語言，便於交談式的使用。
- Julia 的進階語法很容易學習。
- Julia 為可選型別程式語言，它的（使用者定義之）資料型別可以讓程式碼變得簡潔而且不容易出錯。
- Julia 擁有大量的標準程式庫與無數的第三方套件可使用。

Julia 是一種獨一無二的語言，因為它解決了所謂的“雙語言問題”。您不再需要另一種語言來解決高效能運算問題。這不代表一切都是自動自發的。程式設計師必須負責克服造成瓶頸的程式碼，不過在 Julia 中這是可以自動完成的。

本書適合誰閱讀？

本書適合所有想要學寫程式的人。無須任何正規的先備知識。

本書漸進式的介紹新的概念，在後面章節會介紹更進階的主題。

《*Think Julia*》可用在高中或大學階段一學期的課程。

第一章

程式之道

本書的目標在於教導您像電腦科學家一樣思考。這種思考方式同時運用了來自數學、工程、和自然科學的一些最佳特性。電腦科學家和數學家一樣使用正規語言來表達概念（特別是計算）。他們像工程師一樣設計物品、將元件組合成系統並在多種可能性中評估得失。他們也像科學家一樣觀察一個複雜系統的行為後形成假說、並進行預測。

電腦科學家最重要的技能是**問題解決**（*problem solving*）。問題解決代表以系統化的方式闡述問題、以創意來思考解決方案、以及能清楚並且正確的表達解決方案。結果常常是讓學習程式的過程成為練習解題技巧的絕佳機會。這也是為何本章的標題是“程式之道”的原因。

在某個層面上，您會學到如何設計程式，而這技能原本就十分有用。在另一層面上，您會將程式設計當作是一種完成某件事情的方法。當我們繼續學習時，那件事情就會逐漸浮現。

何謂程式？

程式（*program*）就是說明如何執行運算之一連串指令（instruction）。這個運算可能與數學有關，例如解方程組或找出多項式的根。不過也可能是符號運算，例如搜尋並替換文件中的文字。又或是與圖形相關的，例如處理影像或播放影片。

不同的程式語言在細節上差異頗大，但所有語言仍然具有一些基本的指令類別：

輸入（*Input*）

從鍵盤、檔案、網路、或其他裝置取得資料。

輸出（*Output*）

在螢幕上顯示資料、將資料儲存於檔案、透過網路傳輸資料等。

數學運算（*Math*）

執行加、減法等基本數學運算。

條件化執行（*Conditional execution*）

檢查特定條件是否成立並執行適當之程式碼。

重複（*Repetition*）

重複的執行某動作，其用法通常具有不同的變化。

信不信由您，以上的類別就大概涵蓋全部的指令了。所有您曾經用過的程式，不論多麼複雜，都是由上面這些指令寫成的。所以您可以將程式設計想像成將一件複雜的工作拆解成一些小工作，再將這些小工作繼續拆解成更小的工作，直到它簡單到可以用上述的基本指令完成為止。

執行 Julia

開始使用 Julia 的挑戰之一為在您的電腦安裝它和其他相關軟體。如果您熟悉所使用的作業系統，尤其是命令行介面（command-line interface），那麼安裝 Julia 應該不困難。但對初學者而言，同時學習系統管理和程式設計可能會十分痛苦。

為了避免這個問題，我推薦您在瀏覽器上執行 Julia。當您比較熟悉 Julia 後，我會再針對 Julia 的安裝提出一些建議。

您可以在瀏覽器上使用 JuliaBox（*https://www.juliabox.com*）網站直接執行 Julia。並不需要安裝它－只要在瀏覽器上進行登入就可以開始運算（參見附錄 B）。

Julia *REPL*（Read–Eval–Print Loop）為一讀取並執行 Julia 程式碼的程式。您可以在 JuliaBox 中開啟一終端機並於命令行鍵入 **`Julia`**。開始執行後，您會看到下面畫面：

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.0 (2019-01-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

前面的幾行包含了 REPL 的相關資訊，您看到的可能會和這裏的不太一樣。不過您應該確認一下版本號碼至少要是 `1.0.0`。

最後一行是**命令提示符號**（*prompt*），告訴您 REPL 已經準備好讓您輸入指令了。如果您鍵入一行程式碼並按下 Enter 鍵，REPL 會顯示下列結果：

```
julia> 1 + 1
2
```

本書的程式碼片段皆可以逐字的複製與貼上，包括 `julia>` 提示符號以及任何輸出結果。

您現在已經準備好可以開始了。從現在起，我會假設您已經瞭解如何啟動 Julia REPL 和執行程式碼。

第一個程式

傳統上學習一個新的程式語言時所寫的第一個程式被稱為“Hello, World!”，因為它會在螢幕上顯示“Hello, World!”這些文字。在 Julia 中它長得像這樣：

```
julia> println("Hello, World!")
Hello, World!
```

這是**列印敘述**（*print statement*）的範例，雖然它並不會真的列印什麼東西到印表機，而是顯示在螢幕上。

程式中的雙引號標示出要顯示的文字的起點和終點，它們不會出現在輸出結果中。

小括號指出 `println` 是一個函數。我們在第 3 章中會再談到函數。

算術運算子

在“Hello, World!”程式後，下一步是算術運算。Julia 提供了**運算子**（*operator*）來表達像加法和乘法這類運算的符號。

運算子 `+`、`-`、和 `*` 會執行加法、減法、和乘法運算，如下面的範例所示：

```
julia> 40 + 2
42
julia> 43 - 1
42
julia> 6 * 7
42
```

運算子 `/` 則執行除法：

```
julia> 84 / 2
42.0
```

您可能會奇怪為何結果不是 `42` 而是 `42.0`？我會在下一節中解釋。

最後，運算子 `^` 會執行指數運算（exponentiation）；也就是將一數值開幾次方：

```
julia> 6^2 + 6
42
```

值與型別

值（*value*）是程式處理的基本事物之一，字母或數字就是例子。我們目前已經看過的值包括 `2`、`42.0`、和 `"Hello, World!"`。

這些值屬於不同的**型別**（*type*）：`2` 是**整數**（*integer*），`42.0` 是**浮點數**（*floating-point number*），而 `"Hello, World!"` 是**字串**（*string*），會這麼稱呼它是因為它是由字母所串起來構成的。

如果您不確定一個值的型別，REPL 可以告訴您：

```
julia> typeof(2)
Int64
julia> typeof(42.0)
Float64
julia> typeof("Hello, World!")
String
```

整數屬於型別 Int64，字串屬於 String，浮點數則屬於 Float64。

那麼像 "2" 和 "42.0" 這樣的數值又是如何呢？它們看來像數字，但又像字串一般被雙引號包圍。其實它們是字串：

```
julia> typeof("2")
String
julia> typeof("42.0")
String
```

當輸入一個大的整數時，您可能會習慣使用逗號來分隔數字群組，就像 1,000,000 這樣。在 Julia 中這不是合法的**整數**表示法，不過它仍然是合法的：

```
julia> 1,000,000
(1, 0, 0)
```

那完全不是我們所期待的結果！ Julia 將 1,000,000 看做是一連串由逗號分隔的整數。我們稍後會再學習到這類的序列。

您倒是可以用 1_000_000 來得到相似的效果。

正規與自然語言

自然語言（*natural language*）就是人類所說的語言，例如英文、西班牙文、和法文。它們並不是由人類所設計出來的（雖然人們試圖為它們加上某種秩序），而是自然演化而成的。

正規語言（*formal language*）則是人類為了特定應用所設計的語言。例如，數學的符號系統（notation）就是一種特別適合用以表達數字和符號間關係的正規語言。化學家使用正規語言來表達分子的化學結構。更重要的是，程式語言使用正規語言來表達運算。

正規語言會使用嚴格的**語法**（*syntax*）規則來管控敘述的結構。例如在數學上 3 + 3 = 6 是正確的語法，但 3 + = 3$6 不是。在化學中，$H_2O$ 為語法正確的化學式，但 $_2Zz$ 不是。

語法規則使用兩種調味料來組成敘述：**符記**（*token*）和**結構**（*structure*）。符記為語言的基本元素，例如文字、數字、和化學元素等。3 + = 3$6 的問題之一是 $ 不是數學的合法符記（根據我的認知）。同樣的，$_2Zz$ 也不合法，因為沒有元素的符號是 Zz。

語法規則的第二種類型涉及組合符記的方式。等式 3 + = 3 不合法，因為即使 + 和 = 是合法的符記，它們也不能接在一起出現。同樣在化學式中，下標總是出現在元素名稱後面，而不是前面。

This is @ well-structured Engli$h sentence with invalid t*kens in it.（這段文字結構正確但包含不合法的符記）。This sentence all valid tokens has, but invalid structure with.（這段文字使用合法的符記，但結構有問題）。

當您閱讀英文中的一句話或正規語言中的一個敘述時，您必須搞懂它的結構（雖然在自然語言中這通常是下意識完成的）。這個過程稱為**剖析**（*parsing*）。

雖然正規語言和自然語言具有很多共通的特性－符記、結構、和語法－它們還是有一些不同的：

歧義（*ambiguity*）

自然語言充滿著歧義，人們會利用前後文和其他資訊來處理歧義。正規語言的設計會盡量或完全避免歧義發生，亦即任何敘述都只有一種意涵，不論其前後文為何。

贅語（*redundancy*）

為了避免歧義與減少誤解，自然語言包含了不少贅語，使得文句過於冗長。正規語言較不會有贅語且較精簡。

生硬性（*literalness*）

自然語言充滿了俗語和隱喻。若我說“The penny dropped”（本句代表恍然大悟的意思），實際上大概不會真的有硬幣掉下來。正規語言則是望文生義。

由於我們生來便說著自然語言，有時要適應正規語言會有點困難。自然語言和正規語言間的差異就像詩詞和散文的差異一樣：

詩詞（*poetry*）

文字不只用於表達意義，也用於吟頌上。整體而言詩詞創造了效果或情感反應。贅語不但常見且被蓄意的使用。

散文（*prose*）

文字的字面意義更為重要，結構也對它的意義貢獻良多。散文比詩詞容易分析，不過仍然包含贅語。

程式（*program*）

電腦程式的意涵不包含任何歧義而且文即其義，並可以藉由符記與結構分析完全的瞭解其他的意義。

正規語言比自然語言更難懂，所以需要花更多時間來理解。此外，結構也很重要，所以由上到下、由左至右的讀法並不一定比較好。您要做的是在腦中剖析程式，找出其符記並解譯其結構。最後要注意的是細節很重要。拼字或標點符號的小錯在自然語言不會造成大礙，但在正規語言中會造成災難。

除錯

程式設計師會犯錯。由於某種離奇的原因，程式中的錯誤被稱為**臭蟲**（*bug*）。找出它們的過程則稱為**除錯**（*debugging*）。

程式設計，尤其是除錯，有時會引起強烈情緒反應。如果您正為一頑強的臭蟲而掙扎，你可能會感覺憤怒、沮喪或丟臉。

有證據證明人會像對待其他人一樣對待電腦。當它們表現好時，我們會認定它們是伙伴。當它們頑固難處時，我們會像對付頑固難處的人一樣對待它們。[1]

事先準備好面對這些反應可能有助於處理它們。其中一個作法是把電腦想像為一位具有某種長處（例如速度和精準度）以及弱點（例如缺乏同理心和無法看見大局）的員工。

您的工作是當一個好的主管：找出運用長處和降低弱點危害的方法。找出使用情緒來克服問題的方法，而不要讓情緒反應干涉了工作效率。

學習除錯可能會令人沮喪，但那是可以用在程式設計外的有用技能。在每一章的最後都有一節來說明我對除錯的建議。我希望對您有幫助！

1 Reeves, Byron, and Clifford Ivar Nass. 1996. "The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places." Chicago, IL: Center for the Study of Language and Information; New York: Cambridge University Press.

詞彙表

問題解決（*problem solving*）

列出問題、找出解答、並呈現解答的過程。

程式（*program*）

明確說明如何進行計算的一系列指令。

REPL

一個重複讀取輸入後執行、並輸出結果的程式。

提示符號（*prompt*）

REPL 所顯示的一組字元，用以表示它已準備好讀取使用者之輸入。

print 敘述（*print statement*）

一個讓 Julia REPL 在螢幕上顯示值的指令。

運算子（*operator*）

一個用以表達像加法、乘法、或字串連接等簡單計算的符號。

值（*value*）

程式所處理資料的基本單位，如數字或字串。

型別（*type*）

值的類別。目前我們已看過的型別包括整數（`Int64`）、浮點數（`Float64`）、和字串（`String`）。

整數（*integer*）

一種用來表達沒有小數的數字的型別。

浮點數（*floating-point*）

一種用來表達有小數的數字的型別。

字串（*string*）

一種用來表達一連串字元的型別。

自然語言（*natural language*）

任何一種由人類所使用且為自然演化而成的語言。

正規語言（*formal language*）

任何一種由人類為了特定目的（例如表達數學概念或電腦程式）而設計的語言。所有的程式語言都是正規語言。

語法（*syntax*）

用以管理程式結構的規則。

符記（*token*）

程式之語法結構之基本元素，可類比為自然語言之文字。

結構（*structure*）

符記組合的方式。

剖析（*parse*）

檢驗程式並分析其語法結構。

臭蟲（*bug*）

程式中的錯誤。

除錯（*debugging*）

找出並修正臭蟲的過程。

習題

在電腦前閱讀本書是一個好主意，因為您可以立即試作範例。

習題 1-1

當您試驗新的程式語言特點時，應該試著犯錯。例如，在“Hello, World!”程式中，試著看看如果您少打了一個雙引號會如何？如果兩個都沒有打呢？如果拼錯 `println` 呢？

這類的實驗能幫您記得您所讀的；它也能在設計程式時幫到您，因為您會知道錯誤訊息的含意。最好在現在故意犯錯，而不要等到稍晚不小心犯錯。

1. 在 print 敘述中，如果少了一個或二個括號會如何？
2. 若您要列印一字串，少了一個或二個雙引號會如何？
3. 您可以使用減號來表達負數，如 `-2`。那如果您在數字前放上加號呢？ `2++2` 會怎樣？
4. 在數學中，前置零是沒問題的，例如 `02`。您可以試試在 Julia 中會發生什麼事？
5. 如果您在兩個數值間沒有加入運算子會怎樣呢？

習題 1-2

啟動 Julia REPL 並用它來作為計算器。

1. 42 分 42 秒等於幾秒？
2. 10 公里（kilometer）等於幾英哩（mile）？注意一英哩等於 1.61 公里。
3. 如果您在 10 公里賽跑中跑出 37 分 48 秒的成績，您的平均配速（每英哩所花的時間）是多少？您的平均速度又是每小時多少英哩呢？

第二章

變數、運算式與敘述

程式語言最有力的特性之一是處理**變數**（*variable*）的能力。一個變數就是用來參照至某個值的名稱。

指定敘述

指定敘述（*assignment statement*）會建立一個新的變數並給它一個值：

```
julia> message = "And now for something completely different"
"And now for something completely different"
julia> n = 17
17
julia> π_val = 3.141592653589793
3.141592653589793
```

這個範例用了三次指定。第一次將一字串指定給名為 `message` 的變數。第二次將整數 `17` 指定給 `n`。第三次則將圓周率 π 的（近似）值指定給 `π_val`（**`\pi TAB`**）。

在紙上表達變數常用的方式是用一個箭頭從變數指向它的值。這種圖形稱為**狀態圖**（*state diagram*），因為它顯示了變數目前的狀態（可以把它看作是變數的心智狀態）。圖 2-1 顯示了上述範例的結果。

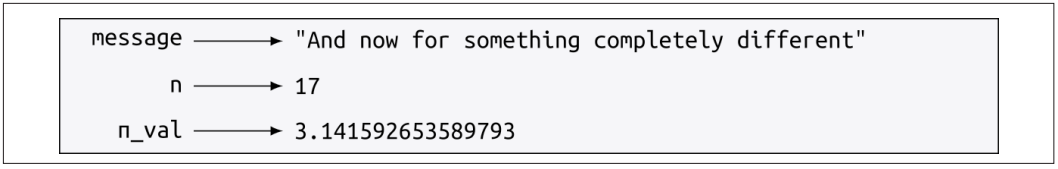

圖 2-1　狀態圖

變數名稱

程式設計師一般會使用有意義的名稱來稱呼變數－它們可用來說明變數的用途。

變數名稱的長度不限。它們可以包含幾乎所有的萬國碼（Unicode）字元（參見第 93 頁的“字元”小節），不過不能以數字開頭。雖然使用大寫字母是合法的，但是傳統上變數名稱都只用小寫字母。

萬國碼字元可用類似 LaTeX 中的縮寫方式，以定位字元完成（tab completion）方式在 Julia REPL 中輸入。

底線字元 _ 可出現在變數名稱內。它通常用於包含多個文字的名稱內，例如 `your_name` 或 `airspeed_of_unladen_swallow`。

若您給變數一個不合法的名稱，會產生語法錯誤（syntax error）：

```
julia> 76trombones = "big parade"
ERROR: syntax: "76" is not a valid function argument name
julia> more@ = 1000000
ERROR: syntax: extra token "@" after end of expression
julia> struct = "Advanced Theoretical Zymurgy"
ERROR: syntax: unexpected "="
```

`76trombones` 並不合法，因為它的開頭是一個數字。`more@` 也不合法，因為它包含一個不合法字元 `@`。不過 `struct` 是有什麼問題呢？

原來 `struct` 是 Julia 的一個**關鍵字**（*keyword*）。REPL 使用關鍵字來辨識程式的結構，因此它們不能被用於變數名稱。

Julia 的關鍵字如下：

```
abstract type       baremodule    begin       break       catch
const               continue      do          else        elseif
end                 export        finally     for         false
function            global        if          import      in
let                 local         macro       module      mutable struct
primitive type      quote         return      true        try
using               struct        where       while
```

您不須硬背這些字。在大多數的開發環境中，關鍵字會以不同顏色來顯示。一旦您想用它們來作為變數名稱時，您會知道它們是關鍵字的。

運算式與敘述

運算式（*expression*）是值、變數、和運算子的組合。值本身就是一個運算式，變數也是，所以下列所有的運算式都是合法的：

```
julia> 42
42
julia> n
17
julia> n + 25
42
```

當您在提示符號後輸入一個運算式時，REPL 會對它進行**賦值**（*evaluate*），也就是找出這個運算式的值。在這個範例中，n 的值是 17 而且 n + 25 的值為 42。

敘述（*statement*）是程式碼的單位，它會產生一些效果，例如建立變數或顯示一個值：

```
julia> n = 17
17
julia> println(n)
17
```

這裏的第一行為將一個值賦予 n 的指定敘述。第二行為列印敘述，用以顯示 n 的數值。

當您鍵入敘述時，REPL 會**執行**（*execute*）它，也就是它會做任何敘述所說的事。

腳本模式

直至目前為止我們都以**交談模式**（*interactive mode*）來執行 Julia，也就是您直接和 REPL 互動。交談模式在學習開始時很有用，不過隨著您寫的程式碼愈來愈長，它會變得笨手笨腳的。

另一種方式是將程式碼儲存至稱之為**腳本**（*script*）的檔案並以**腳本模式**（*script mode*）來執行 Julia。習慣上 Julia 的腳本檔名是以 *.jl* 結尾。

如果您知道如何在電腦上建立與執行腳本，那您已經準備好了。否則我會再一次推薦使用 JuliaBox。開啟一個文字檔、撰寫腳本、並將其存為一個延伸檔名為 *.jl* 的檔案。這個腳本可以在終端機中使用 **Julia 腳本名稱 .jl** 命令來執行。

由於 Julia 提供了兩種模式，您可以在進行腳本寫作前先以交談模式進行程式碼片段的測試。不過交談模式和腳本模式仍存在著令人混淆的差異。

舉例來說，如果您使用 Julia 作為計算器，您可能會鍵入：

```
julia> miles = 26.2
26.2
julia> miles * 1.61
42.182
```

第一行指定一個值給 `miles` 並顯示這個值。第二行為運算式，因此 REPL 對它進行賦值並顯示結果。結果顯示馬拉松賽跑的距離大約為 42 公里。

不過如果您將同樣的程式碼鍵入腳本並執行它，您不會得到任何的輸出。在腳本模式中運算式本身不會產生任何看得到的效果。Julia 是會對運算式進行賦值，但它不會顯示那數值，除非您告訴它要這麼做：

```
miles = 26.2
println(miles * 1.61)
```

這種行為一開始會造成混淆。

腳本通常包含一系列的敘述。如果有超過一個以上的敘述，結果會依敘述執行順序出現。

例如下列腳本：

```
println(1)
x = 2
println(x)
```

會輸出：

```
1
2
```

指定敘述不會產生輸出。

習題 2-1

為檢驗您所學，在 Julia REPL 鍵入下列敘述並觀察它們的結果：

```
5
x = 5
x + 1
```

現在將同樣的敘述放入腳本並執行它。結果為何？將每一敘述轉換為列印敘述再執行一次看看。

運算子優先順序

當運算式包含超過一個以上的運算子時，對其賦值的順序依**運算子優先順序**（*operator precedence*）而定。對於數學運算子而言，Julia 跟隨數學的傳統。縮寫 *PEMDAS* 讓您可以方便的記住這些規則：

- 括號（*P*arentheses）具有最高的優先順序，可被用來強制運算式依您所想要的順序進行賦值。由於括號內的運算式會優先運算，`2*(3-1)` 會是 `4`，且 `(1+1)^(5-2)` 會是 `8`。您也可以用括號來讓運算式較為易讀，如 `(minute * 100) / 60`，即使那不會對結果產生什麼變化。
- 指數（*E*xponentiation）具有次高優先順序，所以 `1+2^3` 會是 `9` 而不是 `27`，而 `2*3^2` 會是 `18` 而不是 `36`。
- 乘除（*M*ultiplication 和 *D*ivision）的優先順序高於加減（*A*ddition 和 *S*ubtraction）。所以 `2*3-1` 是 `5` 而不是 `4`，而 `6+4/2` 是 `8` 而不是 `5`。
- 具有同樣優先順序的運算子會由左至右進行賦值（除了指數）。所以運算式 `degrees / 2 * π` 會先進行除法後其結果再乘上 π。如果是要除以 2π 您可以使用括號，或寫成 `degrees / 2 / π` 或 `degrees / 2π`。

我不會刻意的去記運算子的優先順序。如果我一時無法判定其優先順序，我會使用括號來讓它變明確。

字串運算

一般而言，您不能在字串上執行數學運算，即使字串看來像是數字。所以下列的運算是不合法的：

```
"2" - "1"    "eggs" / "easy"    "third" + "a charm"
```

不過有兩個例外，`*` 和 `^`。

* 運算子執行**字串連接**（*string concatenation*），也就是將兩字串頭尾相連。例如：

```
julia> first_str = "throat"
"throat"
julia> second_str = "warbler"
"warbler"
julia> first_str * second_str
"throatwarbler"
```

^ 運算子也可以運作在字串上，用以表達重複。例如 "Spam"^3 的結果是 "SpamSpamSpam"。如果其中一個值是字串，另一個必須是整數。

* 和 ^ 可類比成乘法和指數運算。就像 4^3 等於 4*4*4 一樣，我們也期待 "Spam"^3 會和 "Spam"*"Spam"*"Spam" 相同，也真的如此。

註解

當程式愈來愈大愈來愈複雜時，它們也會愈來愈難讀。正規語言有時是很難讀的，我們常常無法只看一段程式碼就知道它在做什麼，或者為何那麼做。

基於這個原因，在程式中加上以自然語言來解釋程式作為的註記會是一個好主意。這些註記被稱為**註解**（*comment*），它們以 # 符號開頭：

```
# 計算花費的時數百分比
percentage = (minute * 100) / 60
```

在此案例中，註解自成一行。您也可以在一行的尾端加進註解：

```
percentage = (minute * 100) / 60   # 時數百分比
```

任何在 # 之後一直到這一行結尾之間的字元都會被忽略－它們不會對程式的執行產生影響。

註解最有用的地方是用在解釋程式碼中的不明顯特點。我們可以合理假設讀者知道程式碼在**做什麼**，不過更有用的是解釋**為何這樣做**。

以下這個註解和程式碼意義重複所以沒什麼用：

```
v = 5   # 將 5 指定給 v
```

以下這個註解則包含了從程式碼看不出來的有用資訊：

```
v = 5   # 速度，單位為公尺 / 秒
```

好的變數名稱可以降低使用註解的需求，不過太長的名稱會使得複雜的運算式變得難讀，所以要進行取捨。

除錯

程式會發生三種錯誤：語法錯誤、執行錯誤、以及語意錯誤。知道如何分辨它們可以讓我們更快找出它們：

語法錯誤（*syntax error*）

“語法”是指程式的結構以及這些結構之規則。例如括號一定要成對出現，所以 `(1 + 2)` 是合法的，但 `8)` 就是語法錯誤。

若您的程式中發生語法錯誤，Julia 會顯示錯誤訊息並結束程式的執行，故您無法執行該程式。在您程式設計生涯的前幾週，您可能會花許多時間在找出語法錯誤。當您經驗累積後，您犯錯次數會變少且會更快的找到它們。

執行錯誤（*runtime error*）

第二種錯誤叫做執行錯誤，會這麼稱呼是因為它們只在程式開始執行後才會出現。這類錯誤也被稱為**例外**（*exception*），因為它們指出某種（不好的）例外狀況已經發生。

執行錯誤在本書前幾章的簡單程式中極少出現，所以可能要再一陣子您才會碰到一個。

語意錯誤（*semantic error*）

第三種錯誤為“語意”錯誤，代表和意涵相關。若程式出現語意錯誤，它還是會執行且不會產生任何錯誤訊息，不過執行結果卻是不正確的。它其實是在做其他的事，那些您告訴它要做的事。

找出語意錯誤可能十分困難，因為那需要藉由程式的輸出進行反推並試圖找出它在做什麼。