讀我

這是一段學習體驗，而不是一本參考書，所有阻礙學習的東西，都會被刻意排除掉。第一次閱讀時，你必須從頭開始，因為本書對讀者的知識背景做了一些假設。

如果你有其他語言的程式撰寫經驗，將會有幫助。

大多數開發人員都是在學過其他程式語言才發現 Ruby。（他們通常是從其他語言來尋求庇護的。）我們會為有經驗的初學者提供足夠的基礎知識，但我們不會深入探討變數的相關細節或是 `if` 述句的運作原理。如果你之前能夠有一些這方面的經驗，本書閱讀起來將會比較輕鬆。

我們不會涵蓋每一個類別、程式庫和方法。

Ruby 隨附了大量內建的類別和方法。沒錯，它們都很有趣，但如果要全部涵蓋，本書的厚度將會變為兩倍。我們的重點會擺在初學者應該瞭解的核心類別和方法。確保你對它們能夠有深入的瞭解，並且確信你知道如何及何時使用它們。不管怎樣，一旦你看完《深入淺出 Ruby》，你將能夠拿起任何參考書籍，馬上查到我們遺漏的類別和方法。

不要略過任何活動。

習題與活動絕非附屬品，而是本書核心內容的一部分，有些可以幫助記憶，有些可以幫助理解，還有一些能夠幫助應用。所以，**請不要略過任何習題**。

重複是刻意且必要的。

我們冀望 Head First（深入淺出）系列能夠讓你**真正**學到東西，希望你讀完本書之後，能夠記住讀過的內容，然而，大部分參考用書並非以此為目標。本書的重點放在**學習**，所以，某些重要的內容會一再出現，企圖加深你的印象。

我們讓範例盡可能精簡。

讀者告訴我們，最讓他們懊惱的就是，在 200 列的例子中找尋他們需要瞭解的兩列程式碼。本書大部分的例子會儘可能呈現在最小的範圍內，好讓你想學習的部份既簡單又明瞭。不要假設所有的例子都是穩健或甚至是完整的—它們是專為**學習**而寫的，並不總是具有完整的功能。

本書的範例就擺在網站上，彈指之間便能夠取得。請參考本書網站：

http://headfirstruby.com/。

1 事半功倍

以自己想要的方式寫程式

你對 Ruby 這個瘋狂的語言感到疑惑，不知道它是否符合你的需要。讓我們來問你這幾個問題：**你想要有生產力嗎？**你認為像其他語言那樣使用額外的編譯器和程式庫以及類別檔和程式碼，就能夠讓你更快**完成產品**、**讓同事欽羨**以及**讓客戶滿意**？你喜歡能夠替你**處理細節**的語言嗎？如果你有時希望能夠停止維護煩人的樣板程式碼（boilerplate code）以便直接處理你的問題，那麼 Ruby 將會符合你的需要。Ruby 讓你得以**用較少的程式碼做更多的事情**。

Ruby 的設計哲學

回到 1990 年代的日本，一位名叫 Yukihiro Matsumoto（又名 "Matz"）的程式員夢想發展自己心目中的理想程式語言。他認為理想的程式語言應該：

- 好學易用
- 足以處理任何的程式設計工作
- 讓程式員專注在他們試圖解決的問題上
- 儘量少給程式員壓力
- 物件導向

環顧當時可供使用的程式語言，他認為沒有一個完全符合他的需要。於是他打算自己做一個。稱之為 Ruby。

對 Ruby 做了一段時間的修補後，Matz 於 1995 年向大眾釋出了 Ruby。從那時起，Ruby 社群做了一些令人驚訝的事情：

- 他們建造出了龐大的 Ruby 程式庫，可以協助你完成任何事，包括讀取 CSV（comma-separated value 的縮寫）檔案以及透過網路控制物件
- 他們撰寫了替代的解譯器，讓你的 Ruby 程式碼可以執行得快一點或是與其他語言整合
- 他們創建了 Ruby on Rails，一個廣受歡迎的 web 應用程式開發框架

Ruby 語言帶來了創造力與生產力的爆發。彈性與易用性是 Ruby 的基本原則；也就是說，你可以使用 Ruby 來完成任何的程式設計工作，所需要用到的程式碼比其他語言還少。

一旦你具備基礎知識後，你會同意：使用 Ruby 是一種樂趣！

彈性與易用性是 Ruby 的基本原則。

取得 Ruby

首先，你可以整天寫 Ruby 程式，但如果你不執行它，對你毫無用處。讓我們來確定一下電腦上是否已經安裝了可供使用的 Ruby **解譯器**（*interpreter*）。我們想要使用的是 2.0 或之後的版本。開啟一個終端機視窗〔也稱為**命令提示字元**（*command-line prompt*）〕並鍵入：

```
ruby -v
```

加上 -v 可讓 Ruby 顯示版本編號。

"ruby" 本身會啟動一個 Ruby 解譯器。

按下 Ctrl-C 會離開解譯器並返回你的作業系統的提示字元。

```
File Edit Window Help
$ ruby -v
ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-darwin11.4.2]
$ ruby
^Cruby: Interrupt

$
```

當你在提示字元之後鍵入 **ruby -v**，如果你看到這樣的回應，你就萬事俱備了：

```
ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-darwin11.4.2]
```

我們並不關心此輸出的其他部分，只要這個部分所指出的是 ruby 2.0 或之後的版本。

動手做！

如果你沒有 Ruby 2.0 或之後的版本可用，可至 **www.ruby-lang.org** 下載你的作業系統所使用的安裝程式。

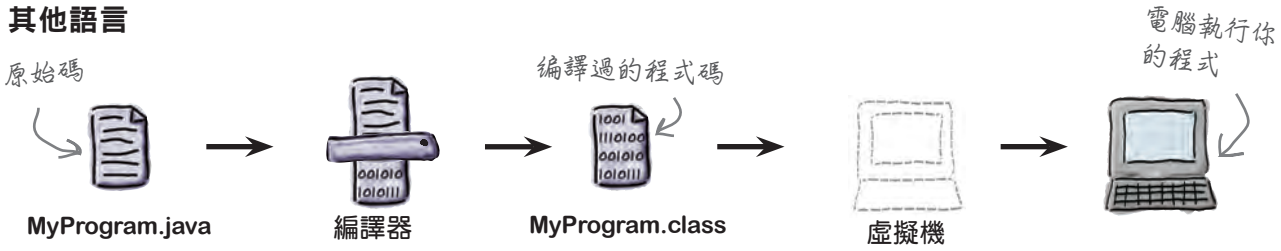
順便說一下，如果你不小心只鍵入了 `ruby`（漏掉了 `-v`），Ruby 將會等著你鍵入程式碼。要退出此模式，只要在壓下 Control 鍵的同時按下 C 鍵就行了。若你需要 Ruby 立即退出此模式，你可以隨時這麼做。

使用 Ruby

可供你執行的 Ruby 原始碼檔案，稱為**命令稿**（*script*），它們只是純文字檔案。欲執行 Ruby 命令稿，你只需要把你的 Ruby 程式碼存入檔案，並使用 Ruby 解譯器來執行該檔案。

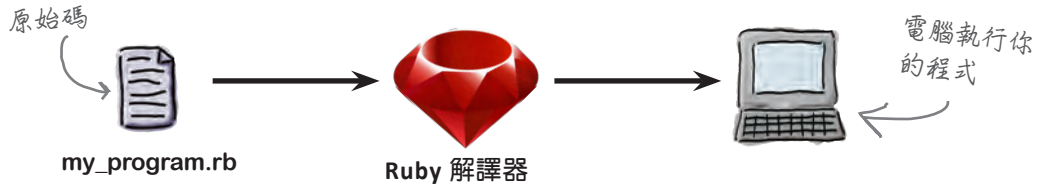
在 C++、C# 或 Java 等語言中，必須自己動手把程式碼編譯成 CPU 或虛擬機能夠瞭解的二進位格式。在這些語言中，程式碼在編譯之前是無法執行的。

其他語言



使用 Ruby，你可以跳過這一步。Ruby 會自動編譯命令稿中的原始碼。這意味著，程式碼撰寫好之後可以立即進行測試。

Ruby 的方式




```
puts "hello world"
```

鍵入你的原始碼。

儲存成：*hello.rb*

```
File Edit Window Help
$ ruby hello.rb
hello world
```

輸出結果在這裡！

使用 Ruby 解譯器執行你的原始碼。

以互動方式使用 Ruby

使用 Ruby 之類的語言還有一個很大的好處。你不僅不必在每次測試程式碼的時候執行編譯器，而且不必把它擺在命令稿的開頭。

Ruby 隨附了一個獨立的程式，稱為 **irb**（代表 **I**nteractive **Ruby**）。irb shell 允許你鍵入任何的 Ruby 運算式，而且會立即對它求值以及將結果顯示給你看。這是一個學習語言的好方式，因為你會立即得到反饋。但即使是 Ruby 專家也會使用 irb 來嘗試新的想法。

本書中，我們將會撰寫許多經由 Ruby 解譯器執行的命令稿。但每當你想要測試新的想法，啟動 irb 實驗一下，是一個好主意。

那麼，我們還等什麼？現在就讓我們進入 irb 來嘗試一些 Ruby 運算式。

使用 irb shell

開啟終端機視窗（terminal window）並鍵入 `irb`。這將會啟動互動式 Ruby 解譯器。（你將會知道它正在執行，因為提示字元〔prompt〕將會發生變化，不過你看到的結果可能跟此處不完全一樣。）

接著你就可以鍵入你想要執行的任何運算式，然後按下 Enter/Return 鍵。Ruby 將會立即對它求值並把結果顯示給你看。

irb 使用完畢，在提示字元（prompt）之後鍵入 `exit`，你就可以回到作業系統的提示字元。

你的第一個 Ruby 運算式

知道如何啟動 irb 之後，現在讓我們來嘗試一些運算式，看看會得到什麼結果！

在提示字元之後鍵入此運算式並按下 Return 鍵：

```
1 + 2
```

你將會看到此結果：

```
=> 3
```

數學運算和比較

Ruby 之數學運算符（math operator）的使用方式如同大多數其他的語言。+（加號）代表加法、-（減號）代表減法、*（乘號）代表乘法、/（除號）代表除法，而 ** 代表指數。

如果你鍵入：	Irb 會顯示：
`5.4 - 2.2`	`=> 3.2`
`3 * 4`	`=> 12`
`7 / 3.5`	`=> 2.0`
`2 ** 3`	`=> 8`

你可以使用 < 和 > 來比較兩個值，用於檢視哪個比較小、哪個比較大。你還可以使用 ==（兩個等號）來檢視兩個值是否相等。

```
4 < 6          => true
4 > 6          => false
2 + 2 == 5     => false
```

字串

字串（*string*）是一系列的文字符號。你可以使用字串來保存名稱、電子郵件地址、電話號碼以及一大堆其他東西。Ruby 的字串很特別，因為即使是非常大型的字串，處理起來也相當有效率（但是在許多其他的語言中並非如此）。

指定字串的最簡單方式為，使用一對雙引號（"）或單引號（'）來括住它。這兩種引號的運作方式有些不同；本章稍後將會加以討論。

```
"Hello"        => "Hello"
'world'        => "world"
```

變數

Ruby 允許我們建立**變數**（*variable*）—為數值取名字。

Ruby 中，變數的使用不必事先宣告；直接對它們賦值就可以建立它們。你可以使用 = 符號（一個等號）來對變數賦值。

如果你鍵入：	*Irb* 會顯示：
`small = 8`	`=> 8`
`medium = 12`	`=> 12`

變數名稱必須以一個小寫字母開頭，而且可以包含字母、數字和底線符號。

對變數賦值之後，你可以在有需要的時候取用變數裡的數值，儘管你可以直接使用原本的數值。

```
small + medium    => 20
```

Ruby 中，變數沒有資料型態的限制；它們可以保存你想要的任何值。你可以把字串賦值給一個變數，接著又把浮點數賦值給相同的變數，這是完全合法的。

```
pie = "Lemon"     => "Lemon"
pie = 3.14        => 3.14
```

+= 運算符讓你得以對變數中既有的值進行加法運算。

```
number = 3        => 3
number += 1       => 4
number            => 4

string = "ab"     => "ab"
string += "cd"    => "abcd"
string            => "abcd"
```

一般常識

以小寫字母來替變數命名。避免使用數字；一般人不會這麼做。使用底線符號來分隔名稱中的單字。

```
my_rank = 1
```

這種風格有時稱為 "snake case"（蛇形命名），因為底線符號讓名稱看起來好像在地上爬行。

一切皆為物件！

Ruby 是一個**物件導向**（*object-oriented*）語言。這意味著，你的資料隨附了有用的**方法**（你可以根據需要來執行的一段程式碼）。

在現代的語言中，把資料視為物件的情況相當常見，字串也不例外，因此字串當然具有可供呼叫的方法：

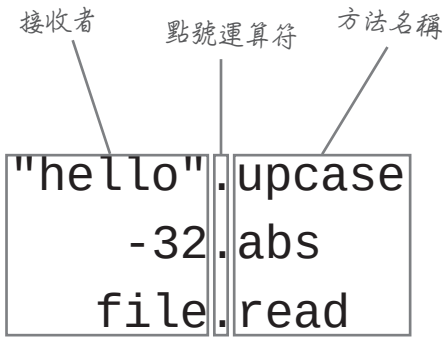
如果你鍵入：	irb 會顯示：
`"Hello".upcase`	`=> HELLO`
`"Hello".reverse`	`=> olleH`

然而，Ruby 最酷的是，一切皆為物件。連簡單如數字的資料也是物件。這意味著，數字也具備有用的方法。

`42.even?`	`=> true`
`-32.abs`	`=> 32`

呼叫一個物件上的方法

當你以這種方式進行呼叫，被你呼叫了方法的物件稱為**接收者**（*receiver* ）。它就位於點號運算符左側。你可以把呼叫物件的方法想成是在傳遞**訊息**（*message*）給它—就像在一張紙條上寫著：『嘿，你可以把你的大寫版本回傳給我嗎？』或是『你可以告訴我你的絕對值嗎？』



開啟一個新的終端機視窗，鍵入 **irb** 並按下 Enter/Return 鍵。接著為下面所列示的每個 Ruby 運算式，把你所猜測的求值結果寫在隨後的位置上。然後試著把每個運算式鍵入 irb 並按下 Enter 鍵。看看你的猜測是否與 irb 所傳回的結果相符！

```
42 / 6
```

.....

```
name = "Zaphod"
```

.....

```
name.upcase
```

.....

```
"Zaphod".upcase
```

.....

```
name.reverse
```

.....

```
name.upcase.reverse
```

.....

```
name.class
```

.....

```
name * 3
```

.....

```
5 > 4
```

.....

```
number = -32
```

.....

```
number.abs
```

.....

```
-32.abs
```

.....

```
number += 10
```

.....

```
rand(25)
```

.....

```
number.class
```

.....

開啟一個新的終端機視窗，鍵入 **irb** 並按下 Enter/Return 鍵。接著為下面所列示的每個 Ruby 運算式，把你所猜測的結果寫在隨後的位置上。然後試著把每個運算式鍵入 irb 並按下 Enter 鍵。看看你的猜測是否與 irb 所傳回的結果相符！

```
42 / 6
```

7

```
name = "Zaphod"
```

"Zaphod"

對一個變數賦值，會回傳你指定給它的值。

```
name.upcase
```

"ZAPHOD"

即使一個物件被存放在變數裡，你也可以呼叫它的方法。

```
"Zaphod".upcase
```

"ZAPHOD"

但是你甚至不必先把它存入變數！

```
name.reverse
```

"dohpaZ"

```
name.upcase.reverse
```

"DOHPAZ"

你可以對一個方法所回傳的值呼叫另一個方法。

```
name.class
```

String

一個物件的 class 方法可用於判斷它是屬於何種物件。

```
name * 3
```

"ZaphodZaphodZaphod"

你可以讓字串多次重複！

```
5 > 4
```

true

```
number = -32
```

-32

```
number.abs
```

32

```
-32.abs
```

32

```
number += 10
```

-22

這會把變數中的值加上 10，然後把結果賦值給變數。

```
rand(25)
```

一個隨機數字

每次的執行結果可能會有所不同（因為它是隨機的）

沒錯，這是一個方法呼叫，只是我們沒有指定接收者而已。稍後會做更進一步說明！

```
number.class
```

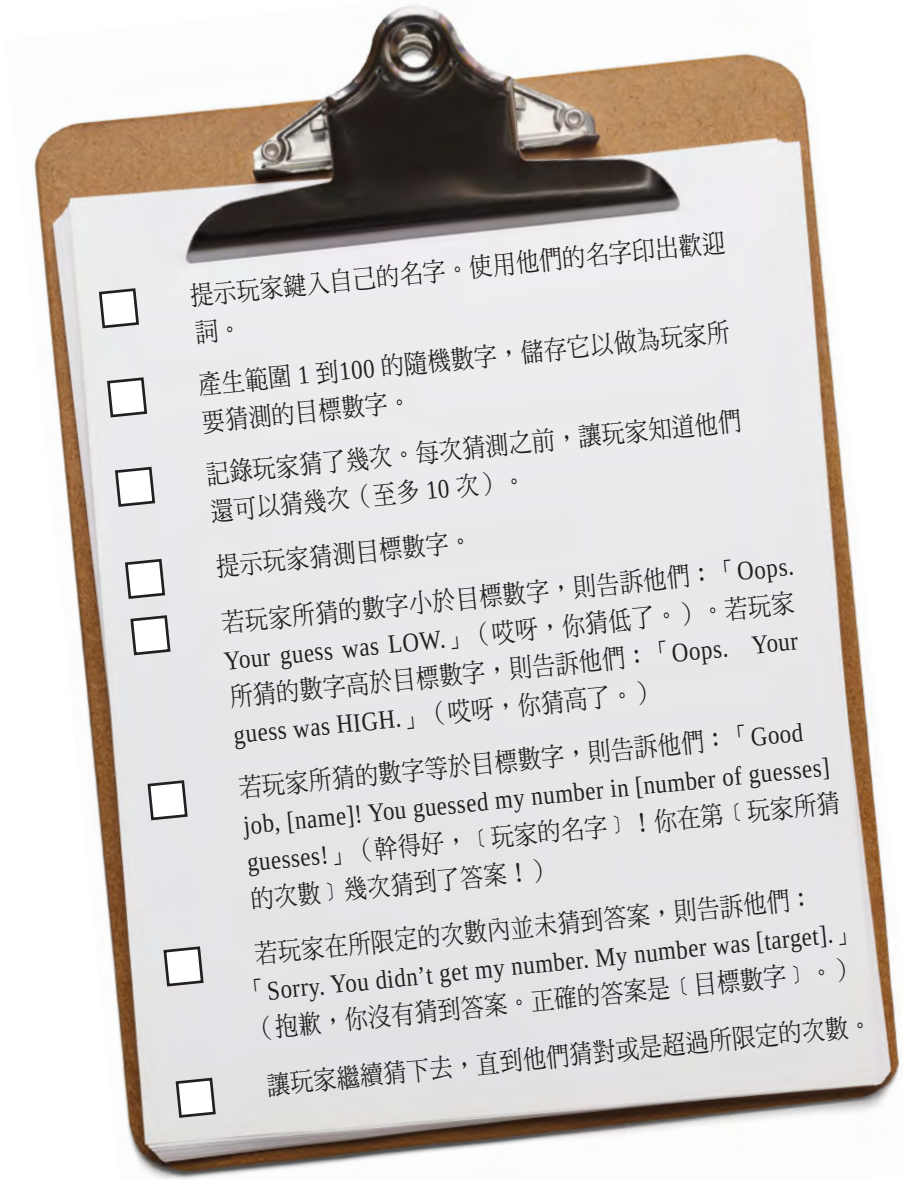
Fixnum

Fixnum 是一種整數型態。

讓我們來建構一個遊戲程式

第一章中，我們將會建構一個簡單的遊戲程式。如果這聽起來有點嚇人，不用擔心；使用 Ruby 來進行程式設計，這將會很容易！

讓我們來看一下，我們需要做哪些事情：



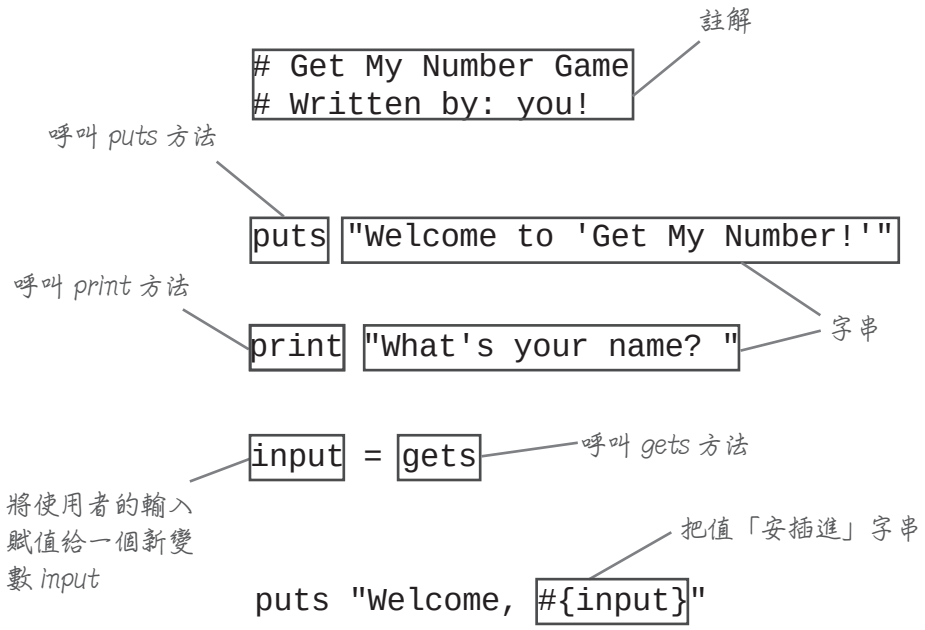
- ☐ 提示玩家鍵入自己的名字。使用他們的名字印出歡迎詞。
- ☐ 產生範圍 1 到100 的隨機數字，儲存它以做為玩家所要猜測的目標數字。
- ☐ 記錄玩家猜了幾次。每次猜測之前，讓玩家知道他們還可以猜幾次（至多 10 次）。
- ☐ 提示玩家猜測目標數字。
- ☐ 若玩家所猜的數字小於目標數字，則告訴他們：「Oops. Your guess was LOW.」（哎呀，你猜低了。）。若玩家所猜的數字高於目標數字，則告訴他們：「Oops. Your guess was HIGH.」（哎呀，你猜高了。）
- ☐ 若玩家所猜的數字等於目標數字，則告訴他們：「Good job, [name]! You guessed my number in [number of guesses] guesses!」（幹得好，〔玩家的名字〕！你在第〔玩家所猜的次數〕幾次猜到了答案！）
- ☐ 若玩家在所限定的次數內並未猜到答案，則告訴他們：「Sorry. You didn't get my number. My number was [target].」（抱歉，你沒有猜到答案。正確的答案是〔目標數字〕。）
- ☐ 讓玩家繼續猜下去，直到他們猜對或是超過所限定的次數。

遊戲設計師
Gary Richardott

輸入、儲存與輸出

我們的第一項要求是以使用者的名字來印出歡迎詞。為了實現這個目標，我們將需要撰寫一支命令稿，以便從使用者**取得輸入**、**儲存**該輸入，以及使用所儲存的值來**建立一些輸出**。

只要短短幾列 Ruby 程式碼，我們就可以做到這一切：



接下來我們將以幾頁的篇幅來說明此命令稿中每個組件的細節。但是在開始之前，讓我們先試著執行看看！

執行命令稿

我們已經撰寫了一支符合第一項要求的簡單命令稿：以玩家的名字印出歡迎詞。現在，你將會學到如何執行命令稿，使得你可以看到我們所建立的歡迎詞。

第一步

以你慣用的文字編輯器開啟一個新文件，鍵入如下的程式碼。

```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"
print "What's your name? "

input = gets

puts "Welcome, #{input}"
```

get_number.rb

第二步

儲存文件並以 *get_number.rb* 為檔名。

第三步

開啟終端機視窗，切換至你的程式所在之目錄。

第四步

鍵入 **ruby get _ number.rb** 以便執行程式。

第五步

你將會看到一段歡迎詞以及一段提示。鍵入你的名字並按下 Enter/Return 鍵。

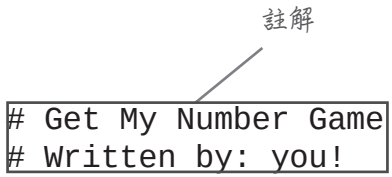
```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
```

讓我們以幾頁的篇幅進一步檢視此程式碼中每個部分。

註解

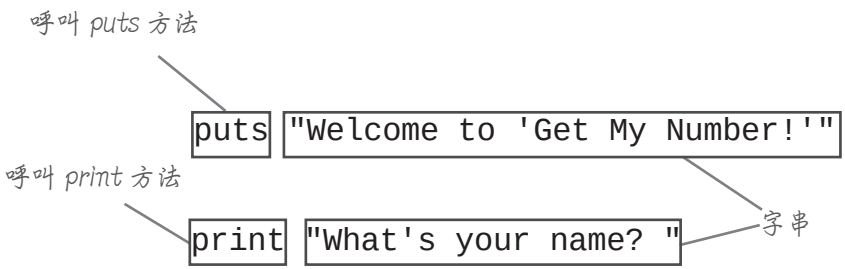
我們的原始碼一開始有兩列註解。Ruby 會忽略以井字號（#）開頭的所有內容，直到該列結束為止，這樣你就可以為你自己和你的同事留下說明或注意事項。

若你把一個 # 放入你的程式碼，那麼從該處開始直到該列結束，所有內容將會被視為註解。它的作用就好像 Java 或 JavaScript 中的雙斜線（//）。

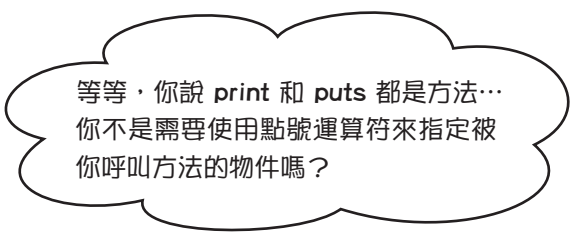


puts 與 print

程式碼中，我們首先會呼叫 puts 方法（puts 是 "put string"的簡寫）以便把文字顯示在標準輸出（通常是終端機）上。我們會把一個字串（內含所要顯示的文字）傳遞給 puts。



接著我們會把另一個字串傳遞給下一列的 print 方法，以便詢問使用者的名字。print 方法的行為就像 puts，但是 puts 會自動在字串結尾添加一個換列符號（如果字串結尾還沒有這個符號的話）以便跳到下一列，然而 print 並不會這麼做。為了美觀的理由，傳遞給 print 的字串中，我們使用了一個空格做為結尾，這樣我們所顯示的文字就不會與使用者所鍵入的名字黏在一起。



有時你不必為方法的呼叫指定接收者。

puts 和 print 這兩個方法很重要，而且經常被用到，因此 Ruby 的**頂層執行環境**（*top-level execution environment*）納入了這兩個方法以供呼叫。在你的 Ruby 程式碼中，你不必指定接收者就可以呼叫頂層環境中所定義的方法。第 2 章一開始將會介紹如何定義這樣的方法。

方法引數

puts 方法需要取得字串並把它印到標準輸出（你的終端機視窗）。

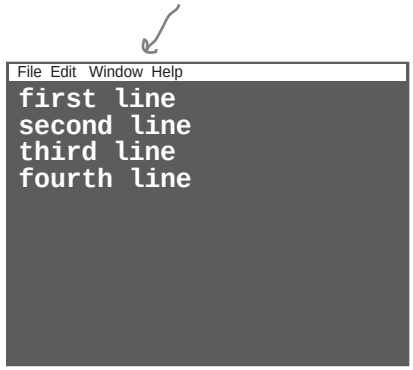
```
puts "first line"
```

被傳遞給 puts 方法的字串稱為方法**引數**（*argument*）

puts方法可以取得多個引數；只需要使用逗號隔開引數就行了。每個引數會自成一列地被印出。

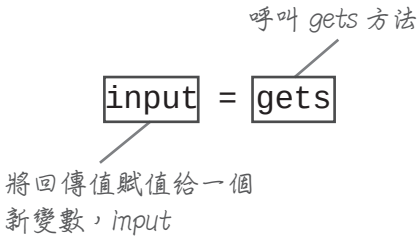
```
puts "second line", "third line", "fourth line"
```

你將會在你的終端機之上看到如下的結果。



gets

gets 方法（gets 是 "get string" 的簡寫）會從標準輸入讀取一列資料（你在終端機上所鍵入的字符）。當你呼叫 gets，會導致程式暫停直到使用者鍵入他們的名字以及按下 Enter 鍵，最後 gets 會回傳使用者所鍵入的文字。



如同 puts 和 print，你可以在程式碼中任何一處呼叫 gets 而不必指定接收者。

方法呼叫中的圓括號可以省略

Ruby 中，方法引數會被放在一對圓括號裡：

```
puts("one", "two")
```

但是你可以省略這一對圓括號，以 puts 呼叫為例，大多數的 Ruby 程式設計者傾向省略圓括號：

```
puts "one", "two"
```

如上所述，gets 方法會從標準輸入讀取一列資料。它（通常）不需要任何引數：

```
gets
```

一般 Ruby 程式設計者認為，若一個方法不需要引數，則可以省略圓括號。所以，即使這是個有效的程式碼，也要避免這麼做：

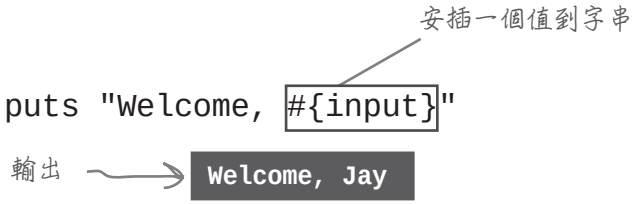
```
gets()
```
不要這麼做！

一般常識

如果沒有引數，方法呼叫可以省略圓括號。即使有引數，你也可以省略方法呼叫的圓括號，但這可能會讓程式碼變得較難閱讀。如果有疑義，請使用圓括號！

字串安插

最後，我們的命令稿會再次使用一個字串來呼叫 puts。這次比較特別，因為我們會把變數裡的值**安插**（*interpolate*）到字串中。每當被雙引號括住的字串包含 #{...} 標記時，Ruby 會使用大括號裡的值來填空（fill in the blank）。#{...} 標記可能會出現在字串中任何位置：開頭、結尾或是中間某處。



#{...} 標記之中不一定要使用變數—你可以使用任何的 Ruby 運算式。

```
puts "The answer is #{6 * 7}."
```

輸出 → `The answer is 42.`

注意，Ruby 只會在被**雙**引號括住的字串中進行安插操作。如果你在被**單**引號括住的字串中使用 #{...} 標記，它將會被按字面印出。

```
puts 'Welcome, #{input}'
```

輸出 → `Welcome, #{input}`

問：分號在哪裡？

答：Ruby 中，你可以使用分號來隔開指令述句（或簡稱述句），但是你一般不應該這麼做。（這會讓程式碼變得較難閱讀。）

```
puts "Hello";
puts "World";
```

不要這麼做！

Ruby 會把各自獨立的程式列視為不同的述句，因此不必使用分號。

```
puts "Hello"
puts "World"
```

問：其他的語言需要我把命令稿放入具有 main 方法的類別中。Ruby 不用嗎？

答：不用！這是 Ruby 令人驚豔的一個特點—簡單的程式不需要一堆儀式。只要寫幾個述句，就大功告成了！

Ruby 中，簡單的程式不需要一堆儀式。

字串裡藏了什麼？

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
```

嗯，這很容易。讓我們把一個驚嘆號添加到歡迎詞字串中所安插的值之後。

```
puts "Welcome to 'Get My Number!'"
print "What's your name? "

input = gets

puts "Welcome, #{input}!"
```

就是添加這個字符！

但如果我們再次執行程式，我們將會看到驚嘆號並未緊跟在使用者的名字之後，它跑到下一列去了！

呃，哦！為什麼跑到這裡來了？

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
!
```

為什麼會這樣？或許 `input` 變數中發生了什麼事…

但是，使用 `input` 方法，看不出有任何異樣。如果我們將下面這一列添加到該程式碼之前，將會看到這樣的輸出：

```
puts input
```

```
Jay
```

使用 inspect 和 p 方法來檢視物件

現在讓我們再試一次，這次使用的是 Ruby 程式中專門用於排除問題的方法，inspect。每個 Ruby 物件都會有 inspect 方法可供呼叫。它會把物件轉換成適合除錯的字串表達形式。也就是，它將揭示物件通常不會在程式輸出中呈現的面向。

下面是對我們的字串呼叫 inspect 的結果：

```
puts input.inspect
```

```
"Jay\n"
```

啊哈！

字串結尾的 \n 是什麼？我們將會在下一頁解開這個謎團…

印出 inspect 結果的需求是如此普遍，於是 Ruby 提供了另一個快捷的方式：p 方法。它的運作方式如同 puts，不過在印出結果之前，它會對每個引數呼叫 inspect。

下面的程式碼（呼叫 p）等效於前面的程式碼：

```
p input
```

```
"Jay\n"
```

記住 p 方法；之後的章節中，我們將會用它來協助 Ruby 程式碼的除錯！

inspect 方法將揭示物件通常不會在程式輸出中呈現的資訊。

字串中的規避序列

p方法的使用，揭示了使用者之輸入的結尾處出現了意想不到的資料：

```
p input                    "Jay\n"
```

結尾處的這兩個字符，倒斜線（\）與緊跟在後面的 n，實際上代表一個字符：一個換列符（newline character）。（之所以會這樣命名，是因為它會使得終端機的輸出換到下一列。）換列符之所以會出現在使用者輸入的結尾處，是因為當使用者完成輸入時，會按下 Return 鍵，這個額外的字符就這樣被記錄了下來。於是換列符就會出現在 gets方法的回傳值中。

倒斜線（\）與緊跟在後面的 n，稱為**規避序列**（*escape sequence*）—原始碼中，用於表示字串中不按正規方式呈現的字符。

最常被使用的規避序列是 \n（換列符，正如我們所看到的）以及 \t（跳格符，用於縮排）。

```
puts "First line\nSecond line\nThird line"
puts "\tIndented line"
```

```
First line
Second line
Third line
        Indented line
```

常被使用的規避序列

若你將此字符放入「被雙引號括住的」字串中…	…你會得到此字符…
\n	換列（newline）
\t	跳格（tab）
\"	雙引號（double-quote）
\'	單引號（single-quote）
\\	倒斜線（backslash）

通常，當你試圖把一個雙引號（"）放入被雙引號括住的字串中，它將會被視為字串的結尾符號，因而會導致錯誤[譯註]：

```
puts ""It's okay," he said."
```

錯誤 →

```
syntax error, unexpected
tCONSTANT
```

我們可以在雙引號之前放置一個倒斜線符號來規避雙引號，這樣即使在被雙引號括住的字串中放入雙引號也不會導致錯誤。

```
puts "\"It's okay,\" he said."        "It's okay," he said.
```

最後，因為 \ 是規避序列的起始標記，我們還需要一個方法來表示非規避序列起始標記的倒斜線符號。使用 \\ 我們就可以得到字面上的倒斜線。

```
puts "One backslash: \\"              One backslash: \
```

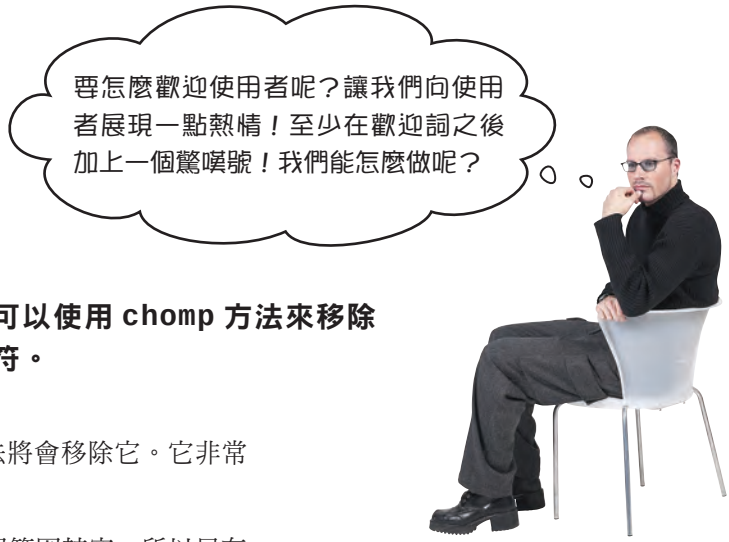
切記，這些規避序列多半僅適用於被**雙**引號括住的字串。在被**單**引號括住的字串中，大多數的規避序列只有字面上的作用。

```
puts '\n\t\"'                         \n\t\"
```

譯註：在 irb 中執行這列述句後，並須按下 CTRL D 以及 Return 鍵才會看到錯誤訊息。

呼叫字串物件上的 chomp 方法

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
!
```



我們可以使用 chomp 方法來移除換列符。

如果字串的結尾字符是一個換列符，chomp 方法將會移除它。它非常適合用於清理，例如，gets 所回傳的字符。

不同於 print、puts 和 gets，chomp 方法的應用範圍較窄，所以只有字串物件會提供此方法。這意味，我們需要把 input 變數中的字串指定為 chomp 方法的**接收者**（*receiver*）。我們需要對 input 使用點號運算符。

```
# Get My Number Game（猜數字）遊戲
# 作者：就是你！

puts "Welcome to 'Get My Number!'"
print "What's your name? "

input = gets

name = input.chomp

puts "Welcome, #{name}!"
```

我們將會把 chomp 的回傳值存入一個名為 name 的新變數。

呼叫 chomp 方法。

以 input 中的字串做為 chomp 方法的接收者。

點號運算符

歡迎詞中使用的是 name，而非 input。

chomp 會回傳相同的字串，但是結尾處的換列符已被移除。我們會將此字串存入名為 name 的變數，該變數是歡迎詞的一部分，它會連同歡迎詞一起被印出。

現在如果我們再次執行此程式，我們將會看到我們投入感情的新歡迎詞能夠正常呈現了！

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay!
```