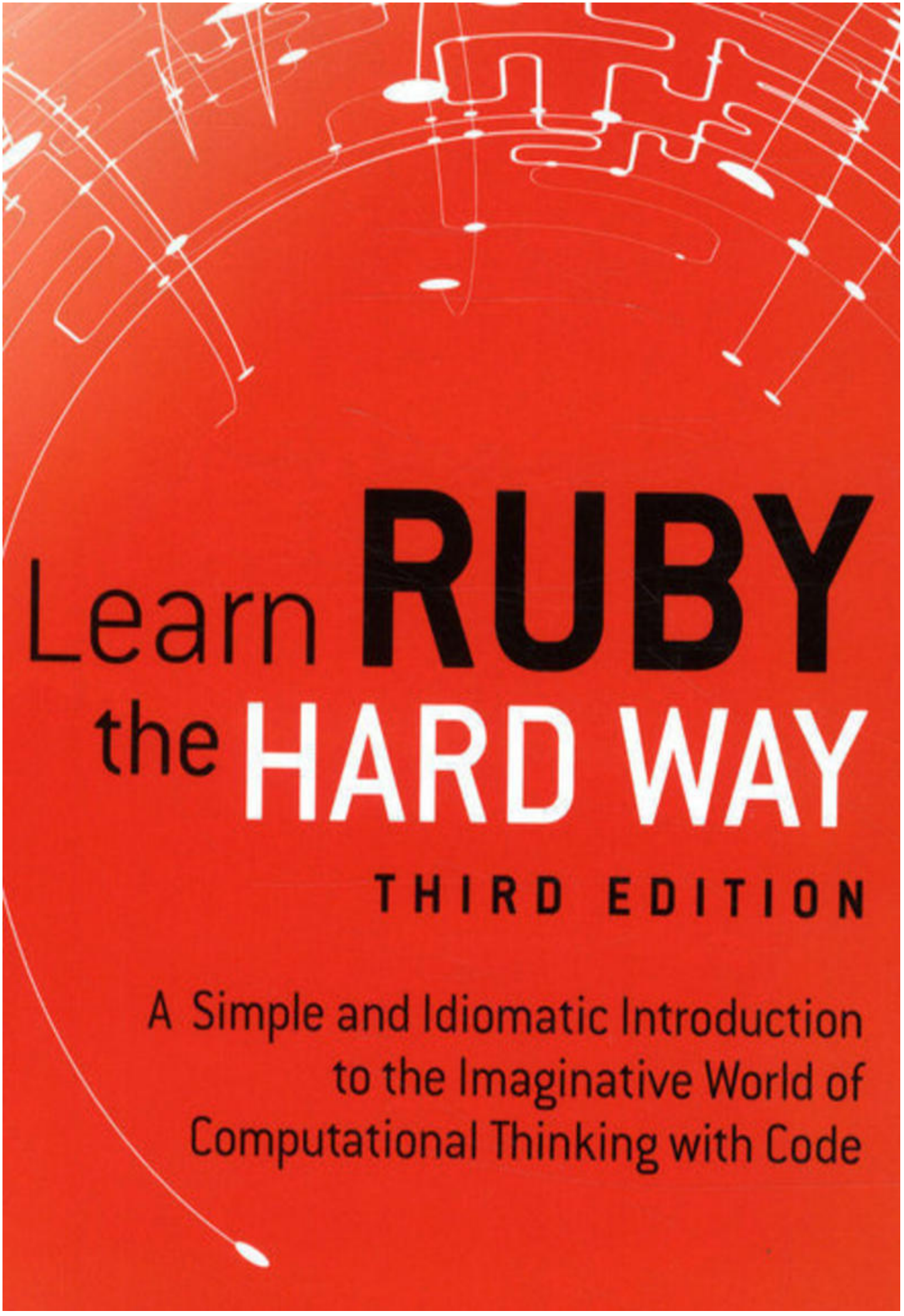
Learn RUBY
the HARD WAY
THIRD EDITION
A Simple and Idiomatic Introduction
to the Imaginative World of
Computational Thinking with Code

笨方法更简单

> 原文出处：http://lrthw.github.io/intro/

这本小书的目的是让你起步学习程式。虽然书名说是「笨办法」，但其实并非如此。所谓的「笨办法」是指本书教授的方式。在这本书的帮助下，你将通过非常简单的练习学会一门程式语言。写练习题是每个程序设计师的必经之路：

1. 做每一道习题
2. 一字不差地写出每一个程式
3. 让程式运行起来 就是这样了。刚开始这对你来说会非常难，但你需要坚持下去。如果你通读了这本书，每晚花个一两小时做做习题，你可以为自己读下一本程式书籍打下良好的基础。通过这本书你学到的可能不是真正的「写程式」，但你会学到最基本的学习方法。

这本书的目的是教会你程式新手所需的三种最重要的技能：「读和写」、「注重细节」、「发现不同」。

读和写

很显然，如果你连打字都成问题的话，那你学习写程式也会成问题。尤其如果你连程式原始码中的那些奇怪符号都打不出来的话，就根本别提写程式了。没有这样基本技能的话，你将连最基本的软体运作原理都难以学会。

为了让你记住各种符号的名字并对它们熟悉起来，你需要将程式码写下来并且运行起来。这个过程也会让你对程式语言更加熟悉。

注重细节

区分差程式设计师和差程式设计师的最重要的一个技能就是对于细节的注重程度。事实上这是任何行业区分好坏的标准。如果缺乏对于工作的每一个微小细节的注意，你的工作成果将缺乏重要的元素。以写程式来讲，这样你得到的结果只能是毛病多多难以使用的软体。

通过将本书里的每一个例子一字不差地打出来，你将通过实践训练自己，让自己集中精力到你作品的细节上面。

发现不同

程式设计师长年累月的工作会培养出一个重要技能，那就是对于不同点的区分能力。有经验的程式设计师拿着两份仅有细微不同的程式，可以立即指出里边的不同点来。程式设计师甚至造出工具来让这件事更加容易，不过我们不会用到这些工具。你要先用笨办法训练自己，等你具备一些相关能力的时候才可以使用这些工具。

在你做这些练习并且打字进去的时候，你一定会写错东西。这是不可避免的，即使有经验的程式设计师也会偶尔写错。你的任务是把自己写的东西和要求的正确答案对比，把所有的不同点都修正过来。这样的过程可以让你对于程式里的错误和 bug 更加敏感。

不要复制贴上

你必须手动将每个习题练习「打」出来。复制贴上会让这些练习变得毫无意义。这些习题的目的是训练你的双手和大脑思维，让你有能力读程式码、写程式码、观察程式码。如果你复制贴上的话，那你就是在欺骗自己，而且这些练习的效果也将大打折扣。

对于坚持练习的一点提示

在你通过这本书学习写程式时，我正在学习弹吉他。我每天至少训练2 小时，至少花一个小时练习音阶、和声、和琶音，剩下的时间用来学习音乐理论和歌曲演奏以及训练听力等。有时我一天会花8 个小时来练习，因为我觉得这是一件有趣的事情。对我来说，要学好一样东西，每天的练习是必不可少的。就算这天个人状态很差，或者说学习的课题实在太难，你也不必介意，只要坚持尝试，总有一天困难会变得容易，枯燥也会变得有趣了。

在你通过这本书学习写程式的过程中要记住一点，就是所谓的「万事起头难」，对于有价值的事情尤其如此。也许你是一个害怕失败的人，一碰到困难就想放弃。也许你是一个缺乏自律的人，一碰到「无聊」的事情就不想上手。也许因为有人夸你“有天分”而让你自视甚高，不愿意做这些看上去很笨拙的事情，怕有负你「神童」的称号。也许你太过激进，把自己跟有20多年经验的程式老手相比，让自己失去了信心。

不管是什么原因，你一定要坚持下去。如果你碰到做不出来的加分习题，或者碰到一节看不懂的习题，你可以暂时跳过去，过一阵子回来再看。只要坚持下去，你总会弄懂的。

一开始你可能什么都看不懂。这会让你感觉很不舒服，就像学习人类的自然语言一样。你会发现很难记住一些单词和特殊符号的用法，而且会经常感到很迷茫，直到有一天，忽然一下子你会觉得豁然开朗，以前不明白的东西忽然就明白了。如果你坚持练习下去，坚持去上下求索，你最终会学会这些东西的。也许你不会成为一个编程大师，但你至少会明白程序是怎么工作的。

如果你放弃的话，你会失去达到这个程度的机会。你会在第一次碰到不明白的东西时(几乎是所有的东西)放弃。如果你坚持尝试，坚持写习题，坚持尝试弄懂习题的话，你最终一定会明白里边的内容的。

如果你通读了这本书，却还是不知道写程式编程是怎么回事。那也没关系，至少你尝试过了。你可以说你已经尽过力但成效不佳，但至少你尝试过了。这也是一件值得你骄傲的事情。

License

unmodified.

习题 0: 准备工作

這道習題並沒有程式碼。它的主要目的是讓你在電腦上安裝好 Ruby，你應該儘量照著指示操作。

這份教學已經預設你將使用 Ruby 1.9.2

你的系統裡面可能已經裝好了 Ruby。打開 console 並嘗試運行:

```
$ ruby -v
ruby 1.9.2
```

如果你的系統內並沒有 Ruby，不論你使用的是哪一套作業系統，我都高度建議你使用 Ruby Version Manager (RVM) 安裝 Ruby。

Mac OSX

你需要做下列任務來完成這個習題：

1. 用瀏覽器打開 http://learnpythonthehardway.org/wiki/ExerciseZero 下載並安裝 `gedit` 文字編輯器。
2. 把 `gedit` 放到桌面或者快速啟動列，這樣以後你就可以方便使用它了。這兩個選項在安裝時可以看到。
    1. 執行 gedit（也就是你的編輯器），我們要先改掉一些愚蠢的預設值。
    2. 從 `gedit menu` 中打開 `Preferences` ，選擇 `Editor` 頁面。
    3. 將 `Tab width:` 改為 2。
    4. 選擇(確認有勾選到該選項) `Insert spaces instead of tabs` 。
    5. 然後打開 「Automatic indentation」 選項。
    6. 轉到 `View` 頁面，打開 「Display line numbers」 選項。

3. 找到 「Terminal」 程式。它的名字是 `Command Promot` ，或者你可以直接執行 `cmd` 。
4. 為它建立一個捷徑，放到桌面或者是快速啟動列中以方便使用。
5. 執行 Terminal，這個程式看上去不怎麼地。
6. 在 Termnal 程式裡執行 `irb` 。在 Terminal 中執行程式的方式是輸入程式的名稱然後再敲一下 Return (Enter)。
    1. 如果你執行 `irb` 但發現不存在(不認得 `irb` 這個指令)。請用 Ruby Version Manager (RVM) 安裝 Ruby。

7. 敲擊 CTRL-Z (Z) 退出 `irb` 。
8. 這樣你就應該能回到敲 `irb` 前的提示介面了。如果沒有的話自己研究一下為什麼。

9. 學著使用 Terminal 創造一個目錄，你可以上網搜尋怎麼做。
10. 學著使用 Terminal 進入一個目錄，同樣你可以上網搜尋。
11. 使用你的編輯器在你進入的目錄下建立一個檔案。你將建立一個檔案。使用 「Save」 或者 「Save As...」 選項，然後選擇這個目錄。
12. 使用鍵盤切回到 Terminal 視窗，如果不知道怎樣使用鍵盤切換，你一樣可以上網搜尋。
13. 回到 Terminal，看看你能不能使用命令列列出你在目錄裡新建立的檔案，在網路上搜尋怎麼列出檔案夾裡的資料。

> **Note:** 如果你在使用 gedit 上有問題，很有可能這是 non-English keyboards layout 造成的，那麼我會建議你改使用 http://www.barebones.com/products/textwrangler/。

OSX: 你應該看到的結果

以下是我在自己電腦的 Terminal 中練習上述習題時看到的內容。可能會跟你在自己電腦中看的到結果有些不同，所以看看你能不能搞清楚兩者的差異。

```
Last login: Sat Apr 24 00:56:54 on ttys001
~ $ irb
ruby-1.9.2-p180 :001 >
ruby-1.9.2-p180 :002 > ^D
~ $ mkdir mystuff
~ $ cd mystuff
mystuff $ ls
# ... Use Gedit here to edit test.txt....
mystuff $ ls
test.txt
mystuff $
```

Windows

> **Note:** Contributed by zhmark.

1. 用瀏覽器打開 http://learnpythonthehardway.org/wiki/ExerciseZero 下載並安裝 `gedit` 文字編輯器。
2. 把 `gedit` 放到桌面或者快速啟動列，這樣以後你就可以方便使用它了。這兩個選項在安裝時可以看到。 a. 執行 gedit（也就是你的編輯器），我們要先改掉一些愚蠢的預設值。 b. 從 `gedit menu` 中打開 `Preferences` ，選擇 `Editor` 頁面。 c. 將 `Tab width:` 改為 2。 d. 選擇(確認有勾選到該選項) `Insert spaces instead of tabs` 。 e. 然後打開 「Automatic indentation」 選項。 f. 轉到 `View` 頁面，打開 「Display line numbers」 選項。
3. 找到 「Terminal」 程式。它的名字是 `Command Promot` ，或者你可以直接執行 `cmd` 。
4. 為它建立一個捷徑，放到桌面或者是快速啟動列中以方便使用。
5. 執行 Terminal，這個程式看上去不怎麼地。

6. 在 Termnal 程式裡執行 `irb` 。在 Terminal 中執行程式的方式是輸入程式的名稱然後再敲一下 Return (Enter)。
    1. 如果你執行 `irb` 但發現不存在(不認得 `irb` 這個指令)。請用 Ruby Version Manager (RVM) 安裝 Ruby。
7. 敲擊 CTRL-Z (Z) 退出 `irb` 。
8. 這樣你就應該能回到敲 `irb` 前的提示介面了。如果沒有的話自己研究一下為什麼。 .. _Ruby Version Manager (RVM): https://rvm.beginrescueend.com/
9. 學著使用 Terminal 創造一個目錄，你可以上網搜尋怎麼做。
10. 學著使用 Terminal 進入一個目錄，同樣你可以上網搜尋。
11. 使用你的編輯器在你進入的目錄下建立一個檔案。你將建立一個檔案。使用 「Save」 或者 「Save As...」 選項，然後選擇這個目錄。
12. 使用鍵盤切回到 Terminal 視窗，如果不知道怎樣使用鍵盤切換，你一樣可以上網搜尋。
13. 回到 Terminal，看看你能不能使用命令列列出你在目錄裡新建立的檔案，在網路上搜尋怎麼列出檔案夾裡的資料。

> **Warning:** 對於 Ruby 來說 Windows 是個大問題。有時候你在一台電腦上裝得好好的，但在另外一台電腦上卻會漏掉一堆重要功能。如果遇到問題的話，你可以訪問： http://rubyinstaller.org/。

Windows: 你應該看到的結果

```
C:\Documents and Settings\you>irb
ruby-1.9.2-p180 :001 >
ruby-1.9.2-p180 :001 > ^Z

C:\Documents and Settings\you>mkdir mystuff

C:\Documents and Settings\you>cd mystuff

... Here you would use gedit to make test.txt in mystuff ...

C:\Documents and Settings\you\mystuff>
   <bunch of unimportant errors if you istalled it as non-admin - ignore them - hit Enter>
C:\Documents and Settings\you\mystuff>dir
 Volume in drive C is
 Volume Serial Number is 085C-7E02

 Directory of C:\Documents and Settings\you\mystuff

04.05.2010  23:32    <DIR>          .
04.05.2010  23:32    <DIR>          ..
04.05.2010  23:32                 6 test.txt
               1 File(s)              6 bytes
               2 Dir(s)  14 804 623 360 bytes free

C:\Documents and Settings\you\mystuff>
```

你會看到的提示介面、Ruby 資訊，以及一些其他東西可能非常不一樣，不過應該大致上不會差多少。如果你的系統差太多的話，反映給我們，我們會修正過來。

Linux

Linux 系統可謂五花八門，安裝軟體的方式也個有不同。我們假設作為 Linux 使用者的你應該知道如何安裝軟體了，以下是給你的操作指示：

1. 用瀏覽器打開 http://learnpythonthehardway.org/wiki/ExerciseZero 下載並安裝 `gedit` 文字編輯器。
2. 把 `gedit` 放到 Window Manager 明顯的位置，以方便之後使用。
    1. 執行 gedit（也就是你的編輯器），我們要先改掉一些愚蠢的預設值。
    2. 從 `gedit menu` 中打開 `Preferences` ，選擇 `Editor` 頁面。
    3. 將 `Tab width:` 改為 2。
    4. 選擇(確認有勾選到該選項) `Insert spaces instead of tabs` 。
    5. 然後打開 「Automatic indentation」 選項。
    6. 轉到 `View` 頁面，打開 「Display line numbers」 選項。

3. 找到 「Terminal」 程式。它的名字可能是 `GNOME Terminal` \、\ `Konsole` \、或者 `xterm` \。
4. 把 Terminal 也放到 Dock 上。

5. 執行 Terminal，這個程式看上去不怎麼地。
6. 在 Termnal 程式裡執行 `irb` 。在 Terminal 中執行程式的方式是輸入程式的名稱然後再敲一下 Return (Enter)。
    1. 如果你執行 `irb` 但發現不存在(不認得 `irb` 這個指令)。請用 Ruby Version Manager (RVM) 安裝 Ruby。

7. 敲擊 CTRL-D (D) 退出 `irb` 。
8. 這樣你就應該能回到敲 `irb` 前的提示介面了。如果沒有的話自己研究一下為什麼。
9. 學著使用 Terminal 創造一個目錄，你可以上網搜尋怎麼做。
10. 學著使用 Terminal 進入一個目錄，同樣你可以上網搜尋。
11. 使用你的編輯器在你進入的目錄下建立一個檔案。你將建立一個檔案。使用 「Save」 或者 「Save As...」 選項，然後選擇這個目錄。
12. 使用鍵盤切回到 Terminal 視窗，如果不知道怎樣使用鍵盤切換，你一樣可以上網搜尋。
13. 回到 Terminal，看看你能不能使用命令列列出你在目錄裡新建立的檔案，在網路上搜尋怎麼列出檔案夾裡的資料。

Linux: 你應該看到的結果

```
$ irb
ruby-1.9.2-p180 :001 >
ruby-1.9.2-p180 :002 > ^D
$ mkdir mystuff
$ cd mystuff
# ... Use gedit here to edit test.txt ...
$ ls
test.txt
$
```

你會看到的提示介面、Ruby 資訊，以及一些其他東西可能非常不一樣，不過應該大致上不會差多少。如果你的系統差太多的話，反映給我們，我們會修正過來。

給新手的告誡

你已經完成了這節習題，取決於你對電腦的熟悉程度，這個練習對你而言可能會有些難。如果你覺得有難度的話，你要自己克服困難，多花點時間學習一下。因為如果你不會這些基礎操作的話，寫程式對你來說將會是相當艱難的一件事。

如果有程式設計師叫你去使用 `vim` 或者 `emacs` ，你應該拒絕他們。當你成為一個更好的程式設計師的時候，這些編輯器才會適合你使用。你現在需要的一個可以編輯文字的編輯器。我們使用 `gedit` 是因為它很簡單，而且在不同的系統上面使用起來也是一樣的。就連專業程式設計師也用 `gedit` ，所以對於初學者而言它已經夠用了。

總有一天你會聽到有程式設計師建議你使用 Mac OSX 或者 Linux。如果他喜歡字體美觀，他會叫你弄台 Mac OSX 電腦，如果他們喜歡操作控制而且留了一把大鬍子，他會叫你安裝 Linux。這裡再度向你說明，只要是一台手上能用的電腦就夠了。你需要的只有三樣東西 `gedit` 、一個 Terminal、還有 `IRB` 。

Finally the purpose of this setup is so you can do three things very reliably while you work on the exercises:

最後要說的是這節習題的準備工作的目的，也就是讓你可以在以後的習題中順利做到下面的這些事情：

1. 使用 `gedit` 編寫程式碼。
2. *執行*你寫的習題答案。
3. *修改*錯誤的習題答案。
4. 重複上述步驟。

其他的事情只會讓你更困惑，所以還是堅持照著這個計畫進行吧。

习题 1: 第一个程式

你應該在習題 0 中花了不少的時間，學會了如何安裝文字編輯器、執行文字編輯器、以及如何運行 Terminal，如果你還沒這麼做的話，請別繼續往前走，之後會有很多苦頭的。請不要跳過前一個習題的內容繼續前進，這也是本書唯一的一次在習題開頭就做這樣的警告。

```
puts "Hello World!"
puts "Hello Again"
puts "I like typing this."
puts "This is fun."
puts 'Yay! Printing.'
puts "I'd much rather you 'not'."
puts 'I "said" do not touch this.'
```

將上面的內容寫到一個檔案中，取名為 `ex1.rb`。注意這樣的命名方式，Ruby 文件最好以 `.rb` 結尾。

然後你需要在 Terminal 中輸入以下內容來執行這段程式：

```
ruby ex1.rb
```

如果你寫對了的話，你應該看到和下面一樣的內容。如果不一樣，那就是你弄錯了什麼東西。不是電腦有問題，電腦沒問題。

你應該看到的內容

```
$ ruby ex1.rb
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I "said" do not touch this.
$
```

你也許會看到 `$` 前面會顯示你所在的目錄的名稱，這部是問題，但如果你的輸出不一樣的話，你需要找出為什麼會不一樣，然後把你的程式改對。

如果你看到類似如下的錯誤資訊:

```
ruby ex1.rb
ex1.rb:4: syntax error, unexpected tCONSTANT, expecting $end
puts "This is fun."
        ^
```

看懂這些內容對你來說是很重要的。因為你以後還會犯類似的錯誤。就是我現在也會犯這樣的錯誤。讓我們一行一行來看。

1. 首先我們在 Terminal 輸入命令來執行 `ex1.rb` 腳本。
2. Ruby 告訴我們 `ex1.rb` 檔案的第 4 行有一個錯誤。
3. 然後這一行的內容被印了出來。
4. 然後 Ruby 印出了一個 `^` (插入符號，caret) 符號，用來指示錯誤的位置。
5. 最後，它印出了一行「語法錯誤(SyntaxError)」告訴你究竟是發生了怎麼樣的錯誤。通常這些錯誤資訊都非常的難懂，不過你可以把錯誤資訊的內容複製到搜尋引擎裡，然後你就能看到別人也遇到過這樣的錯誤，而且你也許能搞清楚怎樣解決這個問題。

加分習題

你還會有加分習題需要完成。加分習題裡面的內容是供你嘗試的。如果你覺得做不出來，你可以暫時跳過， 過段時間再回來做。

在這個練習中，試試這些東西：

1. 讓你的腳本再多印一行。
2. 讓你的腳本只印其中的一行。
3. 在一行的開始位置放置一個 `#` (octothorpe) 符號。它的作用是什麼？自己研究一下。
4. 從現在開始，除非特別情況，我將不再解釋每個習題的運作原理了。

> **Note:** 井號有很多的英文代稱，例如「octothorpe (八角帽)」」、「pound(英镑符號)」、「hash(電話的 # 键)」、「mesh (網)」。

习题 2: 注释和井号

程式裡的註釋是很重要的。它們可以用自然語言告訴你某段程式碼的功能是什麼。在你想要臨時移除一段程式碼時，你還可以用註釋的方式將這段程式碼臨時禁用。接下來的練習將讓你學會註釋：

```
# A comment, this is so you can read your program later.
# Anything after the # is ignored by Ruby.

puts "I could have code like this." # and the comment after is ignored

# You can also use a comment to "disable" or comment out a piece of code:
# print "This won't run."

puts "This will run."
```

你應該看到的結果

```
$ ruby ex2.rb
I could have code like this.
This will run.
$
```

加分習題

1. 弄清楚 `#` 符號的作用。而且記住它的名稱。（中文為井號，英文為 octothorpe 或者 pound character)。
2. 打開你的 `ex2.rb` 檔案，從後往前逐行檢查。從最後一行開始，倒著逐個單詞單詞檢查回去。
3. 有發現更多錯誤嘛？有的話就改正過來。
4. 閱讀你寫的習題，把每個字符（charcter）都讀出來。有沒有發現更多的錯誤呢？有的話也一樣改正過來。

习题 3: 数字和数学计算

每一種程式語言都包含處理數字和進行數學計算的方法。不必擔心，程式設計師經常撒謊說他們是多們厲害的數學天才，其實他們根本不是。如果他們真是數學天才，他們早就去從事數學相關的行業了，而不是寫寫廣告程式和社交網路遊戲，從人們身上偷賺點小錢而已。

這章習題裡有很多的數學運算符號。我們來看一遍它們都叫什麼。你要一邊寫一邊念出它們的名稱來。直到你念煩了為止。名稱如下：

```
+ 加號
- 減號
/ 除號
* 乘號
% 百分比符號
< 小於符號
> 大於符號
<= 小於等於符號
>= 大於等於號
```

有沒有注意到以上只是些符號，沒有運算操作呢？寫完下面的練習程式碼後，再回到上面的列表，寫出每個符號的作用。例如 `+` 是用來做加法運算的。

```
puts "I will now count my chickens:"

puts "Hens", 25 + 30 / 6
puts "Roosters", 100 - 25 * 3 % 4

puts "Now I will count the eggs:"

puts 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6

puts "Is it true that 3 + 2 < 5 - 7?"

puts 3 + 2 < 5 - 7

puts "What is 3 + 2?", 3 + 2
puts "What is 5 - 7?", 5 - 7

puts "Oh, that's why it's false."

puts "How about some more."

puts "Is it greater?", 5 > -2
puts "Is it greater or equal?", 5 >= -2
puts "Is it less or equal?", 5 <= -2
```

你應該看到的結果

```
$ ruby ex3.rb
I will now count my chickens:
Hens
30
Roosters
97
Now I will count the eggs:
7
Is it true that 3 + 2 < 5 - 7?
false
What is 3 + 2?
5
What is 5 - 7?
-2
Oh, that's why it's false.
How about some more.
Is it greater?
true
Is it greater or equal?
true
Is it less or equal?
false
$
```

加分習題

1. 使用 `#` 在程式碼每一行的前一行為自己寫一個註解，說明一下這一行的作用。
2. 記得最開始時的 的 IRB 吧？再次打開 IRB，然後使用剛才學到的運算符號，把Ruby 當做計算機玩玩。
3. 自己找個想要計算的東西，寫一個 `.rb` 檔案把它計算出來。
4. 有沒有發現計算結果是「錯」的呢？計算結果只有整數，沒有小數部分。研究一下這是為什麼，搜尋一下「浮點數(floating point number)」是什麼東西。
5. 使用浮點數重寫一遍 `ex3.rb`，讓它的計算結果更準確(提示: 20.0 是一個浮點數)。

习题 4: 变数(variable)和命名

Exercise 4: Variables And Names

你已經學會了印出東西和數學運算。下一步你要學的是「變數」。在開發程式中，變數只不過是用來代表某個東西的名稱。程式設計師通過使用變數名稱可以讓他們的程式讀起來更像英語。而且因為程式設計師的記性都不怎樣，變數名稱可以讓他們更容易記住程式的內容。如果他們沒有在寫程式時使用好的變數名稱，在下一次讀到原來寫的程式碼時對他們會大為頭疼的。

如果你被這章習題難住了的話，記得我們之前教過的：找到不同點、注意細節:

1. 在每一行的上面寫一行註釋，給自己解釋一下這一行的作用。
2. 倒著讀你的 `.rb` 檔案。
3. 朗讀你的 `.rb` 檔案，將每個字符也朗讀出來。

```
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven

puts "There are #{cars} cars available."
puts "There are only #{drivers} drivers available."
puts "There will be #{cars_not_driven} empty cars today."
puts "We can transport #{carpool_capacity} people today."
puts "We have #{passengers} passengers to carpool today."
puts "We need to put about #{average_passengers_per_car} in each car."
```

> **Note:** `space_in_a_car` 中的 `_` 是 `底線(underscore)` 符號。你要自己學會怎樣打出這個符號來。這個符號在變數裡通常被用作假想的空隔，用來隔開單詞。

你應該看到的結果

```
$ ruby ex4.rb
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 passengers to carpool today.
We need to put about 3 in each car.
$
```

加分習題

當我剛開始寫這個程式時我犯了個錯誤，Ruby 告訴我這樣的錯誤資訊：

```
ex4.rb:8:in `<main>': undefined local variable or method `car_pool_capacity' for main:Object (NameError)
```

用你自己的話解釋一下這個錯誤資訊，解釋時記得使用行號，而且要說明原因。

更多的加分習題

1. 解釋一下為什麼程式裡面用了 4.0 而不是 4。
2. 記住 4.0 是一個「浮點數」，自己研究一下這是什麼意思。
3. 在每一個變數賦值的上一行加上一行註釋。
4. 記住 `=` 的名稱是等於(equal)，它的作用是為東西取名。
5. 記住 `_` 是底線(underscore)。
6. 將 IRB 作為計算機跑起來，就跟以前一樣，不過這一次在計算過程中使用變數名稱來做計算，常見的變數名稱有 `i` 、 `x` 、 `j` 等等。

习题 5: 更多的变数和印出

我們現在要鍵入更多的變數並且將它們印出來，這次我們將使用一個叫「格式化字串(format string)」的東西，每一次你使用 `"` 將一些文字包起來，你就建立一個字串。字串是程式將資訊展示給人的方式。你可以印出他們，可以將它們寫入檔案，還可以將它們發給網站伺服器等等。

字串是很好用的東西，所以在這個練習中你將學會如何創造包含變數內容的字串，使用專門的格式和語法將變數的內容放到字串裡，相當於來告訴 Ruby: “Hey 這是一個格式化字串，把這些變數放到那幾個位置上”

如常，即使你還不懂這些內容，只要一字不差的鍵入就可以了。

```
my_name = 'Zed A. Shaw'
my_age = 35 # not a lie
my_height = 74 # inches
my_weight = 180 # lbs
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'Brown'

puts "Let's talk about %s." % my_name
puts "He's %d inches tall." % my_height
puts "He's %d pounds heavy." % my_weight
puts "Actually that's not too heavy."
puts "He's got %s eyes and %s hair." % [my_eyes, my_hair]
puts "His teeth are usually %s depending on the coffee." % my_teeth

# this line is tricky, try to get it exactly right
puts "If I add %d, %d, and %d I get %d." % [
    my_age, my_height, my_weight, my_age + my_height + my_weight]
```

你應該看到的結果

```
$ ruby ex5.rb
Let's talk about Zed A. Shaw.
He's 74 inches tall.
He's 180 pounds heavy.
Actually that's not too heavy.
He's got Blue eyes and Brown hair.
His teeth are usually White depending on the coffee.
If I add 35, 74, and 180 I get 289.
$
```

加分習題

1. 修改所有的變數名稱，把它們前面的 `my_` 去掉，確認將每一個地方的都改掉，不只是你使用 `=` 賦值過的地方。
2. 試著使用更多的格式化字串。
3. 在網路上搜尋所有的 Ruby 格式化字串。
4. 試著使用變數將英吋和磅轉換成公分和公斤。不要直接鍵入答案，使用 Ruby 的數學計算來完成。

习题 6: 字串(string)和文字

雖然你已經在程式中寫過字串了，你還沒學過它們的用處。在這章習題中我們將使用複雜的字串來建立一系列的變數，從中你將學到它們的用途。首先我們解釋一下字串是什麼東西。

字串通常是指你想要展示給別人的，或者是你想要從程式裡「導出」的一小段字符。Ruby 可以通過文字裡的雙引號 `"` 或者是單引號 `'` 識別出字串來。這在你以前的 `puts` 練習中你已經見過很多次了。如果你把單引號或者雙引號括起來的文字放到 `puts` 後面，他們就會被 Ruby 印出來。

字串可以包含你目前學過的格式化字串。你只要將格式化的變數放到字串中，跟著一個百分比符號 `%` (percent)，再緊跟著變數名稱即可。唯一要注意的地方，是如果你想要在字串中通過格式化字串放入多個變數的結果，你需要將變數放到 `[]` 中括號(brackets) 中，而且變數之間用 `,` 逗號(comma)隔開。就像你逛商店時說「我要買牛奶、麵包、雞蛋、湯」一樣，只不過程式設計師說的是”[milk, eggs, bread, soup]”。

另一種方式是使用字串插值 (string interpolation) 這種技巧，將變數注入到你的字串中。方法是使用 `#{}` 井號和大括號(pound and curly brace)。與其使用這種格式化字串

```
name1 = "Joe"
name2 = "Mary"
puts "Hello %s, where is %s?" % [name1, name2]
```

我們可以鍵入：

```
name1 = "Joe"
name2 = "Mary"
puts "Hello #{name1}, where is #{name2}?"
```

我們將鍵入大量的字串、變數和格式化字串，並且將它們印出來。我們還將練習使用簡寫的變數名稱。程式設計師喜歡使用惱人的隱晦簡寫來節省打字時間，所以我們現在就將提早學會這件事，這樣你就能讀懂並寫出這些東西了。

```
x = "There are #{10} types of people."
binary = "binary"
do_not = "don't"
y = "Those who know #{binary} and those who #{do_not}."

puts x
puts y

puts "I said: #{x}."
puts "I also said: '#{y}'."

hilarious = false
joke_evaluation = "Isn't that joke so funny?! #{hilarious}"

puts joke_evaluation

w = "This is the left side of..."
e = "a string with a right side."

puts w + e
```

你應該看到的結果

```
There are 10 types of people.
Those who know binary and those who don't.
I said: There are 10 types of people..
I also said: 'Those who know binary and those who don't.'.
Isn't that joke so funny?! false
This is the left side of...a string with a right side.
```

加分習題

1. 遍歷程式，在每一行的上面寫一行註釋，給自己解釋這一行的作用。
2. 找到所有的「字串包含字串」的位置，總共有四個位置。
3. 你確定只有四個位置嗎？你怎麼知道的？說不定我在騙你呢。
4. 解釋一下為什麼 `w` 和 `e` 用 `+` 連起來就可以生成一個更長的字串。

习题 7: 更多印出

現在我們將做一批練習，在練習的過程中你需要鍵入程式碼，並且讓它們運行起來，我不會解釋太多，因為這節的內容都是以前熟悉的。這節練習的目的是鞏固你學到的東西，我們幾輪練習後再見。不要跳過這些習題，不要複製貼上！

```
puts "Mary had a little lamb."
puts "Its fleece was white as %s." % 'snow'
puts "And everywhere that Mary went."
puts "." * 10  # what'd that do?

end1 = "C"
end2 = "h"
end3 = "e"
end4 = "e"
end5 = "s"
end6 = "e"
end7 = "B"
end8 = "u"
end9 = "r"
end10 = "g"
end11 = "e"
end12 = "r"

# notice how we are using print instead of puts here. change it to puts
# and see what happens.
print end1 + end2 + end3 + end4 + end5 + end6
print end7 + end8 + end9 + end10 + end11 + end12

# this just is polite use of the terminal, try removing it
puts
```

你應該看到的結果

```
$ ruby ex7.rb
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.
..........
CheeseBurger
$
```

加分習題

接下來幾節的加分習題是一樣的。

1. 逆向閱讀，在每一行的上面加一行註釋。
2. 倒著閱讀出來，找出自己的錯誤。
3. 從現在開始，把你的錯誤記錄下來，寫在一張紙上。
4. 在開始下一節習題時，閱讀一遍你記錄下來的錯誤。並且儘量避免在下個練習中再犯相同的錯誤。
5. 記住，每個人都會犯錯。程式設計師和魔術師一樣，他們希望大家認為他們從不犯錯，不過這只是表象而已，他們每時每刻都在犯錯。

习题 8: 印出，印出

```
formatter = "%s %s %s %s"

puts formatter % [1, 2, 3, 4]
puts formatter % ["one", "two", "three", "four"]
puts formatter % [true, false, false, true]
puts formatter % [formatter, formatter, formatter, formatter]
puts formatter % [
    "I had this thing.",
    "That you could type up right.",
    "But it didn't sing.",
    "So I said goodnight."
]
```

你應該看到的結果

```
$ ruby ex8.rb
1 2 3 4
one two three four
true false false true
%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s
I had this thing. That you could type up right. But it didn't sing. So I said goodnight.
$
```

加分習題

1. 自己檢查結果，記錄你犯過的錯誤，並且在下個練習中儘量不犯相同的錯誤。

习题 9: 印出，印出，印出

```
# Here's some new strange stuff, remember type it exactly.

days = "Mon Tue Wed Thu Fri Sat Sun"
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"

puts "Here are the days: ", days
puts "Here are the months: ", months

puts <<PARAGRAPH
There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
PARAGRAPH
```

你應該看到的結果

```
$ ruby ex9.rb
Here are the days:
Mon Tue Wed Thu Fri Sat Sun
Here are the months:
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
```

加分習題

1. 自己檢查結果，記錄你犯過的錯誤，並且在下個練習中儘量不犯相同的錯誤。

习题 10: 那是什么？

在習題 9 中我丟給你了一些新東西。我讓你看到兩種讓字串擴展到多行的方法。第一種方法是在月份中間用 `\n` (back-slash n) 隔開。這兩個字串的作用是在該位置上放入一個「新行(new line)」字符。

使用反斜線 `\` (back-slash) 可以將難印出的字符放到字串裡。針對不同的符號有很多這樣的所謂「跳脫序列(escape sequences)」，但有一個特殊的跳脫序列，就是 `\` 雙反斜線(double back-slash)。這兩個字符組合會印出一個反斜線來。接下來我們做幾個練習，然後你就知道這些跳脫序列的意義了。

另外一種重要的跳脫序列是用來將單引號 `'` 和雙引號 `"` 跳脫。想像你有一個用雙引號括起來的字串，你想要在字串裡的內容裡再加入一組雙引號進去，比如你想說 `"I "understand" joke."` ，Ruby 就認為 `"understand"` 前後的兩個引號是字串的邊界，從而把字串弄錯。你需要一種方法告訴 Ruby 字串裡面的雙引號不是真正的雙引號。

要解決這個問題，你需要將雙引號和單引號跳脫，讓 Ruby 將引號也包含到字串裡面去。這裡有一個例子：

```
"I am 6'2\" tall."  # escape double-quote inside string
'I am 6\'2" tall.'  # escape single-quote inside string
```

第二種方法是使用文件語法(document syntax)，也就是 `<<NAME` ，你可以在鍵入 `NAME` 前放入\任意多行的文字。接下來你可以看到如何使用。

```
tabby_cat = "\tI'm tabbed in."
persian_cat = "I'm split\non a line."
backslash_cat = "I'm \\ a \\ cat."

fat_cat = <<MY_HEREDOC
I'll do a list:
\t* Cat food
\t* Fishies
\t* Catnip\n\t* Grass
MY_HEREDOC

puts tabby_cat
puts persian_cat
puts backslash_cat
puts fat_cat
```

你應該看到的結果

注意你打出來的 tab 字符，這節練習中的文字間隔符號對於答案的正確性是很重要的。

```
$ ruby ex10.rb
    I'm tabbed in.
I'm split
on a line.
I'm \ a \ cat.
I'll do a list:
    * Cat food
    * Fishies
    * Catnip
    * Grass

$
```

加分習題

1. 上網搜尋一下還有哪些可用的跳脫字符。
2. 結合跳脫序列和格式化字串，創造一種複雜的格式。

习题 11: 提问

我已經出過很多有關於印出的習題，讓你習慣寫出簡單的東西，但簡單的東西都有點無聊，我們現在要做的的事是把資料(data)讀到你的程式裡面去。這對你可能會有點難度，你可能一下子搞不明白，不過相信我，無論如何先把習題做了再說。只要做幾道練習你就明白了。

一般軟體做的事情主要就是下面幾件：

1. 接受人的輸入。
2. 改變輸入值。
3. 印出改變了的值。

到目前為止你只做了印出，但還不會接受或修改人的輸入。你還不知道「輸入(input)」是什麼意思。閒話少說，我們還是開始做點習題看你能不能明白，下一道習題裡面我們將會有更詳細的解釋。

```
print "How old are you? "
age = gets.chomp()
print "How tall are you? "
height = gets.chomp()
print "How much do you weigh? "
weight = gets.chomp()

puts "So, you're #{age} old, #{height} tall and #{weight} heavy."
```

> **Note:** 注意到我們是使用 `print` 而非 `puts` 嗎？ `print` 不會自動產生新行。這樣你的答案就可以跟問題在同一行了。換句話說， `puts` 會自動產生新行。

你應該看到的結果

$ ruby ex11.rb How old are you? 35 How tall are you? 6’ 2” How much do you weigh? 180lbs So, you’ re ‘35’ old, ‘6'2” ’ tall and ‘180lbs’ heavy. $

加分習題

1. 上網搜尋一下 Ruby 的 `gets` 和 `chomp` 的功能是什麼？
2. 你能找到 `gets.chomp` 別的用法嗎？測試一下你上網找到的例子。
3. 用類似的格式再寫一段，把問題改成你自己的問題。

习题 12: 模组 (Module)

看看這段 code

```
require 'open-uri'

open("http://www.ruby-lang.org/en") do |f|
  f.each_line {|line| p line}
  puts f.base_uri         # <URI::HTTP:0x40e6ef2 URL:http://www.ruby-lang.org/en/>
  puts f.content_type     # "text/html"
  puts f.charset          # "iso-8859-1"
  puts f.content_encoding # []
  puts f.last_modified    # Thu Dec 05 02:45:02 UTC 2002
end
```

在第一行是 require。這是一個 Ruby 中在你所寫的腳本中加入其他來源（如：Ruby Gems 或者是你寫的其他東西）的功能(features) 的方法。與其一次給你所有功能，Ruby 會問你你打算使用什麼。這可使你的程式保持輕薄，又可當做之後其他程式設計師閱讀你的程式時的參考。

等一下！功能 (Features) 還有另外一個名字

我在這裡稱呼他們為「功能(features)」。但實際上沒人這樣稱呼。我這樣做只是取了點巧，使你在學習時先不用理解「行話」。在繼續進行之前你得先知道它們的真名 `modules`（模組）。

從現在開始我們將把這些我們 require 進來的功能稱作 `modules`（模組）。我會這樣說：「你想要 require `open-uri` module。」也有人給它另外一個稱呼：「函式庫(libraries)」。但在這裡我們還是先叫它們 `modules`（模組）吧。

加分習題

1. 上網搜尋 `require` 與 `include` 的差異點。它們有什麼不同？
2. 你能 `require` 一段沒有特別包含 `module` 的腳本嗎？
3. 搞懂 Ruby 會去系統的哪裡找你 require 的 modules。

习题 13: 参数、解包、参数

在這節習題中，我們將涵蓋另外一種將變數傳遞給腳本的方法（所謂腳本，就是你寫的 .rb 檔案）。你已經知道，如果要執行 ex13.rb ，只要在命令列中執行 ruby ex13.rb 就可以了。這句命令中的 ex13.rb 部分就是所謂的「參數(argument)」，我們現在要做的就是寫一個可以接受參數的腳本。

將下面的程式寫下來，後來我將詳細解釋

```
first, second, third = ARGV

puts "The script is called: #{$0}"
puts "Your first variable is: #{first}"
puts "Your second variable is: #{second}"
puts "Your third variable is: #{third}"
```

ARGV 就是「參數變數(argument variable)」，是一個非常標準的程式術語。在其他的程式語言你也可以看到它全大寫的原因是因為它是一個「常數(constant)」，意思是當它被賦值之後你就不應該去改變它了。這個變數會接收當你運行 Ruby 腳本時所傳入的參數。通過後面的習題你將對它有更多的了解。 你将对它有更多的了解。

第 1 行將 ARGV 「解包(unpack)」，與其將所有參數放到同一個變數下面，我們將每個參數賦予一個變數名稱 first 、 second 以及 third 。腳本本身的名稱被存在一個特殊變數 $0 裡，這是我們不需要解包的部份。也許看來有些詭異，但「解包」可能是最好的描述方式了。它的涵義很簡單：「將 ARGV 中的東西解包，然後將所有的參數依次賦予左邊的變數名稱」。

接下來就是正常的印出了。

你應該看到的結果

用下面的方法執行你的程式：

```
ruby ex13.rb first 2nd 3rd
```

如果你每次使用不同的參數執行，你將看到下面的結果：

```
$ ruby ex13.rb first 2nd 3rd
The script is called: ex13.rb
Your first variable is: first
Your second variable is: 2nd
Your third variable is: 3rd

$ ruby ex13.rb cheese apples bread
The script is called: ex13.rb
Your first variable is: cheese
Your second variable is: apples
Your third variable is: bread

$ ruby ex13.rb Zed A. Shaw
The script is called: ex13.rb
Your first variable is: Zed
Your second variable is: A.
Your third variable is: Shaw
```

你其實可以將「first」、「2nd」、「3rd」替換成任意三樣東西。你可以將它們換成任意你想要的東西。

```
ruby ex13.rb stuff I like
ruby ex13.rb anything 6 7
```

加分習題

1. 傳三個以下的參數給你的腳本。當有缺少參數時哪些數值會被使用到？
2. 再寫兩個腳本，其中一個接收更少的參數，另一個接收更多的參數。在參數解包時給它們取一些有意義的變數名稱。
3. 結合 `gets.chomp` 和 `ARGV` 一起使用，讓你的腳本從用戶手上得到更多輸入。

习题 14: 提示和传递

讓我們使用 `ARGV` 和 `gets.chomp` 一起來向使用者提一些特別的問題。下一節習題你將會學習到如何讀寫檔案，這節習題是下節的基礎。在這道習題裡我們將用一個簡單的 `>` 作為提示符號。這和一些遊戲中的方法類似，例如 Zork 或者 Adventure 這兩款遊戲。

```
user = ARGV.first
prompt = '> '

puts "Hi #{user}, I'm the #{$0} script."
puts "I'd like to ask you a few questions."
puts "Do you like me #{user}?"
print prompt
likes = STDIN.gets.chomp()

puts "Where do you live #{user}?"
print prompt
lives = STDIN.gets.chomp()

puts "What kind of computer do you have?"
print prompt
computer = STDIN.gets.chomp()

puts <<MESSAGE
Alright, so you said #{likes} about liking me.
You live in #{lives}.  Not sure where that is.
And you have a #{computer} computer.  Nice.
MESSAGE
```

注意到我們將用戶提示符號設置為 `prompt`，這樣我們就不用每次都要重打一遍了。如果你要將提示符號和修改成別的字串，你只要改一個地方就可以了。

非常順手吧。

> **Important:** 同時必須注意的是，我們也用了 `STDIN.gets` 取代了 `gets`。這是因為如果有東西在 `ARGV` 裡，標準的 `gets` 會認為將第一個參數當成檔案而嘗試從裡面讀東西。在要從使用者的輸入（如 `stdin`）讀取資料的情況下我們必須明確地使用 `STDIN.gets`。

你應該看到的結果

當你執行這個腳本時，記住你需要把你的名字傳給這個腳本，讓 `ARGV` 可以接收到。

```
$ ruby ex14.rb Zed
Hi Zed, I'm the ex14.rb script.
I'd like to ask you a few questions.
Do you like me Zed?
> yes
Where do you live Zed?
> America
What kind of computer do you have?
> Tandy

Alright, so you said 'yes' about liking me.
You live in 'America'.  Not sure where that is.
And you have a 'Tandy' computer.  Nice.
```

加分習題

1. 查一下 Zork 和 Adventure 是兩個怎樣的遊戲。看能不能抓到，然後玩玩看。
2. 將 `prompt` 變數改為完全不同的內容再執行一遍。
3. 給你的腳本再新增一個參數，讓你的程式用到這個參數。
4. 確認你弄懂了我如何結合 `<<SOMETHING` 形式的多行字串與 `#{}` 字串注入做的印出。

习题 15: 读取档案

你已經學過了 `STDIN.gets` 和 `ARGV`，這些是你開始學習讀取檔案的必備基礎。你可能需要多多實驗才能明白它的運作原理，所以你要細心練習，並且仔細檢查結果。處理檔案需要非常仔細，如果不仔細的話，你可能會把有用的檔案弄壞或者清空。導致前功盡棄。

這節練習涉及到寫兩個檔案。一個正常的 `ex15.rb` 文件，另外一個是 `ex15_sample.txt`，第二個文件並不是腳本，而是供你的腳本讀取的文字檔案。以下是後者的內容：

```
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

我們要做的是把該檔案用我們的腳本「打開(open)」，然後印出來。然而把檔名 `ex15_sample.txt` 「寫死(Hard Coding」在程式碼不是一個好主意，這些資訊應該是使用者輸入的才對。如果我們碰到其他檔案要處理，寫死的檔名就會給你帶來麻煩了。解決方案是使用 `ARGV` 和 `STDIN.gets` 來從使用者端獲取資訊，從而知道哪些檔案該被處理。

```
filename = ARGV.first

prompt = "> "
txt = File.open(filename)

puts "Here's your file: #{filename}"
puts txt.read()

puts "Type the filename again:"
print prompt
file_again = STDIN.gets.chomp()

txt_again = File.open(file_again)

puts txt_again.read()
```

這個腳本中有一些新奇的玩意，我們來快速地過一遍：

程式碼的 1-3 行使用 `ARGV` 來獲取檔名，這個你已經熟悉了。接下來第 4 行我們使用一個新的命令 `File.open`。現在請在命令列執行 `ri File.open` 來讀讀它的說明。注意到這多像你的腳本，它接收一個參數，並且傳回一個值，你可以將這個值賦予一個變數。這就是你打開檔案的過程。

第 6 行我們印出了一小行，但在第 7 行我們看到了新奇的東西。我們在 `txt` 上呼叫了一個函式。你從 `open` 獲得的東西是一個 `file` （檔案），檔案本身也支援一些命令。它接受命令的方式是使用句點 `.` (dot or period)，緊跟著你的命令，然後參數。就像 `File.open` 做的事一樣。差別是：當你說 `txt.read()` 時，你的意思其實是：「嘿 txt！執行你的 read 命令，無需任何參數！」

腳本剩下的部份基本差不多，不過我就把剩下的分析作為加分習題留給你自己了。

你應該看到的結果

我的腳本叫 “ex15_sample.txt” ，以下是執行結果：

```
$ ruby ex15.rb ex15_sample.txt
Here's your file 'ex15_sample.txt':
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.

I'll also ask you to type it again:
> ex15_sample.txt
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.

$
```

加分習題

這節的難度跨越有點大，所以你要儘量做好這節加分習題，然後再繼續後面的章節。

1. 在每一行的上面用注釋說明這一行的用途。
2. 如果你不確定答案，就問別人，或者是上網搜尋。大部分時候，只要搜尋「ruby 你要搜尋的東西」，就能得到你要的答案。比如搜尋一下「ruby file.open」。
3. 我使用了「命令」這個詞，不過實際上他們的名字是「函式(function)」和「方法(method)」。上網搜尋一下這兩者的意義和區別。看不懂也沒關係，迷失在其他程式設計師的知識海洋裡是很正常的一件事。
4. 刪掉 9-15 行使用到 `STDIN.gets` 的部份，再執行一次腳本。
5. 只用 `STDIN.gets` 撰寫這個腳本，想想哪種得到檔名的方法更好，以及為什麼。
6. 執行 `ri File` 然後往下捲動直到看見 `read()` 命令（函式/方法）。看到很多其他的命令了吧。你可以玩其他試試。
7. 再次啟動 IRB，然後在裡面使用 `File.open` 打開一個文件，這種 open 和 read 的方法也值得一學。
8. 讓你的腳本針對 `txt` 和 `txt_again` 變數執行一下 `close()` ，處理完檔案後你需要將其關閉，這是很重要的一點。

习题 16: 读写档案

如果你做了上一個練習的加分習題，你應該已經了解了個個種文件相關的命令（方法/函式）。你應該記住的命令如下：

- close – 關閉檔案。跟你編輯器的 `文件->儲存..` 是一樣的意思。
- read – 讀取檔案內容。你可以把結果賦給一個變數。
- readline – 讀取檔案文字中的一行。
- truncate – 清空文件，請小心使用該命令。
- write(stuff) – 將 stuff 寫入檔案。

這是你現在應該知道的重要命令。有些命令需要接收參數，但這對我們並不重要。你只要記住 write 的用法就可以了。 `write` 需要接收一個字串作為參數，從而將該字串寫入檔案。

讓我們來使用這些命令做一個簡單的文字編輯器吧：

```
filename = ARGV.first
script = $0

puts "We're going to erase #{filename}."
puts "If you don't want that, hit CTRL-C (^C)."
puts "If you do want that, hit RETURN."

print "? "
STDIN.gets

puts "Opening the file..."
target = File.open(filename, 'w')

puts "Truncating the file.  Goodbye!"
target.truncate(target.size)

puts "Now I'm going to ask you for three lines."

print "line 1: "; line1 = STDIN.gets.chomp()
print "line 2: "; line2 = STDIN.gets.chomp()
print "line 3: "; line3 = STDIN.gets.chomp()

puts "I'm going to write these to the file."

target.write(line1)
target.write("\n")
target.write(line2)
target.write("\n")
target.write(line3)
target.write("\n")

puts "And finally, we close it."
target.close()
```

這是一個大檔案，大概是你鍵入過的最大的檔案。所以慢慢來，仔細檢查，讓它能夠跑起來。有一個小技巧就是你可以讓你的腳本一部分一部分地跑起來。先寫 1-8 行，讓它能跑起來，再多做 5 行，再接著幾行，以此類推，直到整個腳本都可以跑起來為止。

你應該看到的結果

你將看到兩樣東西，一樣是你新腳本的輸出：

```
$ ruby ex16.rb test.txt
We're going to erase 'test.txt'.
If you don't want that, hit CTRL-C (^C).
If you do want that, hit RETURN.
?
Opening the file...
Truncating the file.  Goodbye!
Now I'm going to ask you for three lines.
line 1: To all the people out there.
line 2: I say I don't like my hair.
line 3: I need to shave it off.
I'm going to write these to the file.
And finally, we close it.
$
```

這是一個大檔案，大概是你鍵入過的最大的檔案。所以慢慢來，仔細檢查，讓它能夠跑起來。有一個小技巧就是你可以讓你的腳本一部分一部分地跑起來。先寫 1-8 行，讓它能跑起來，再多做 5 行，再接著幾行，以此類推，直到整個腳本都可以跑起來為止。

加分習題

1. 如果你覺得自己沒有弄懂的話，用我們的老方法，在每一行之前加上註釋，為自己理清思路。就算不能理清思路，你也可以知道自己究竟具體哪裡沒弄清楚。
2. 寫一個和上一個習題類似的腳本，使用 `read` 和 `ARGV` 讀取你剛才新建立的文件。
3. 檔案中重複的地方太多了。試著用一個 `target.write()` 將 `line1` ， `line2` ， `line3` 印出來，你可以使用字串、格式化字串以及跳脫字串。
4. 找出為什麼我們打開檔案時要使用 `w` 模式，而你真的需要 `target.truncate()` 嗎？去看 Ruby 的 `File.open` 函式找答案吧。

习题 17: 更多的档案操作

現在讓我們再學習幾種檔案操作。我們將編寫一個 Ruby 腳本，將一個檔案中的內容拷貝到另一個檔案中。這個腳本很短，不過它會讓你對於檔案操作有更多的了解。

```
from_file, to_file = ARGV
script = $0

puts "Copying from #{from_file} to #{to_file}"

# we could do these two on one line too, how?
input = File.open(from_file)
indata = input.read()

puts "The input file is #{indata.length} bytes long"

puts "Does the output file exist? #{File.exists? to_file}"
puts "Ready, hit RETURN to continue, CTRL-C to abort."
STDIN.gets

output = File.open(to_file, 'w')
output.write(indata)

puts "Alright, all done."

output.close()
input.close()
```

你應該注意到了我們使用了一個很好用的函式 `File.exists?` 。運作原理是將檔名字串當做一個參數傳入，如果檔案存在的話，它會傳回 `true` ，如果不存在的話就傳回 `false` 。之後在這本書中，我們將會使用這個函式做更多的事情。

你應該看到的結果

如同你前面所寫的腳本，運行該腳本需要兩個參數，一個是待拷貝的檔案位置，一個是要拷貝至的檔案位置。如果我們使用以前的 `test.txt` 我們將看到如下的結果：

```
$ ruby ex17.rb test.txt copied.txt
Copying from test.txt to copied.txt
The input file is 81 bytes long
Does the output file exist? False
Ready, hit RETURN to continue, CTRL-C to abort.

Alright, all done.

$ cat copied.txt
To all the people out there.
I say I don't like my hair.
I need to shave it off.
$
```

該命令對於任何檔案應該都是有效的。試試操作一些別的檔案看看結果。不過當心別把你的重要檔案給弄壞了。

> **Warning:** 你看到我用 `cat` 這個指令了吧？它只能在 Linux 和 OX 下使用，Windows 用戶可以使用 `type` 做到相同效果。

加分習題

1. 再多讀讀和 `require` 相關的資料，然後將 IRB 跑起來，試試 require 一些東西看能不能摸出門到。當然，即使搞不清楚也沒關係。
2. 這個腳本實在有點煩人。沒必要再拷貝之前都問一遍吧，沒必要在螢幕上輸出那麼多東西。試著刪掉腳本的一些功能，使它用起來更友善。
3. 看看你能把這個腳本改到多短，我可以把它寫成一行。
4. 我使用了一個叫 `cat` 的東西，這個古老的命令用處是將兩個檔案「連接(con_cat_enate）」在一起，不過實際上它最大的用處是印出檔案內容到螢幕是。你可以通過 `man cat` 命令了解到更多資訊。
5. 使用 Windows 的人，你們可以給自己找一個 `cat` 的替代品。關於 `man` 的東西就別想太多了。Windows 下沒這個指令。
6. 找出為什麼需要在程式碼中寫 `output.close()` 的原因。

习题 18: 命名、变数、程式码、函式

好大的一個標題。接下來我要教你「函式 (function)」了！咚咚鏘！說到函式，不一樣的人會對它有不一樣的理解和使用方法，不過我只會教你現在能用到的最簡單的使用方式。

函式可以做三件事情：

1. 它們可以給程式碼片段取名，就跟「變數」給字串和數字命名一樣。
2. 它們可以接受參數，就跟你的腳本接受 `ARGV` 一樣。
3. 通過使用 #1 和 #2 ，他們可以讓你創造出「迷你腳本」或者「微命令」。

你可以在 Ruby 中使用 `def` 新建函式，我將讓你創造四個不同的函式，它們運作起來和你的腳本一樣。然後我會示範給你各個函式之間的關係。

```
# this one is like your scripts with argv
def puts_two(*args)
  arg1, arg2 = args
  puts "arg1: #{arg1}, arg2: #{arg2}"
end

# ok, that *args is actually pointless, we can just do this
def puts_two_again(arg1, arg2)
  puts "arg1: #{arg1}, arg2: #{arg2}"
end

# this just takes one argument
def puts_one(arg1)
  puts "arg1: #{arg1}"
end

# this one takes no arguments
def puts_none()
  puts "I got nothin'."
end

puts_two("Zed","Shaw")
puts_two_again("Zed","Shaw")
puts_one("First!")
puts_none()
```

讓我們把你一個函式 `puts_two` 肢解一下，這個函式和你寫腳本的方式差不多，因此看上去你應該會覺得比較眼熟：

1. 首先我們告訴 Ruby 創造一個函式，使用 `def` 去「定義(define)」一個函式。
2. 緊跟著 `def` 的是函式的名稱。本例中它的名稱是「puts_two」，但名字可以 隨便取，就叫「peanuts」也沒關係。但函式的名稱最好能夠表達出它的功能來。

3. 然後我們告訴函式我們需要 `args` (asterisk args)，這和腳本的 `ARGV` 非常相似，參數必須放在小括號 `()` 中才能正常運作。
4. 在定義(definition)之後，後面的行都必須以 2 個空格縮排。其中縮排的第一行的作用是將參數解包，就像我們在腳本中做的事一樣。
5. 為了示範它的運作原理，我們把解包後的每個參數都印出來。 `puts_two` 的問題是它不是建立一個函式最簡單的方法。在 Ruby 中我們可以直接跳過解包參數的過程直接使用 `()` 裡面的名稱作為變數名。就像 `puts_two_again` 實現的功能。

接下來的例子是 `print_one` ，它像你示範了函式如何接收單個參數。

最後一個例子是 `print_none` ，它向你示範了函式可以不接收任何參數。

> **Warning:** 如果你不太能看懂上面的內容也別氣餒。這是非常重要的。後面我們還有更多的習題向你示範如何創造和使用函式。現在你只要把函式理解成「迷你腳本」就可以了

你應該看到的結果

執行上面的腳本會看到如下結果：

```
$ ruby ex18.rb
arg1: 'Zed', arg2: 'Shaw'
arg1: 'Zed', arg2: 'Shaw'
arg1: 'First!'
I got nothin'.
$
```

你應該看出來函式是怎樣運作的了。注意到函式的用法和你以前見過的 `File.exist?` 、 `File.open` 以及別的「命令」有點類似了吧？其實我只是為了讓你容易禮節才叫他們「命令」。它們的本質其實就是函式。也就是說，你也可以在你自己的腳本中創造你自己的「命令」。

加分習題

為自己寫一個函式檢查清單以供後續參考。你可以寫在一個索引卡片上隨時閱讀，直到你記住所有的要點為止。注意事項如下：

1. 函式定義是以 `def` 開始的嗎？
2. 函式名稱是以字串和底線 `_` 組成的嗎？
3. 函式名稱是不是緊跟著括號 `(` ？
4. 括號裡是否包含參數？多個參數是否以逗號隔開？
5. 參數名稱是否有重複？（不能使用重複的參數名）
6. 緊跟著參數的最後是否括號 `)` ？
7. 緊跟著函式定義的程式碼是否用了 2 個空格的縮排（ `indent`)？
8. 函式結束的位置是不是「end」

當你執行（或者說「使用(use)」或者「呼叫(call)」一個函數時，記得檢查下列幾項事情：

1. 呼叫函式時是否使用了函式的名稱？
2. 函式名稱是否緊跟著 `()` ？（非必要，理想性的話應該要加）
3. 參數是否以逗號隔開？
4. 函式是否以 `)` 結尾？

按照這兩份檢查清單裡的內容檢查你的習題，直到你不需要檢查清單為止。

最後，將下面這句話閱讀幾遍：

> 「執行(run)函式」、「呼叫(call)函式」和「使用(use)函式」是同一個意思。

习题 19: 函式和变数

函式這個概念也許承載了太多的資訊量。不過別擔心，只要堅持做這些練習題，對照上個練習中的檢查清單檢查這次練習的關聯，你最終會明白這些內容的。

有一個你可能沒有注意到的細節，我們現在強調一下，函式裡面的變數和腳本裡面的變數之間是沒有連接的。下面的這個練習可以讓你對這一點有更多的思考：

```
def cheese_and_crackers(cheese_count, boxes_of_crackers)
  puts "You have #{cheese_count} cheeses!"
  puts "You have #{boxes_of_crackers} boxes of crackers!"
  puts "Man that's enough for a party!"
  puts "Get a blanket."
  puts # a blank line
end

puts "We can just give the function numbers directly:"
cheese_and_crackers(20, 30)

puts "OR, we can use variables from our script:"
amount_of_cheese = 10
amount_of_crackers = 50
cheese_and_crackers(amount_of_cheese, amount_of_crackers)

puts "We can even do math inside too:"
cheese_and_crackers(10 + 20, 5 + 6)

puts "And we can combine the two, variables and math:"
cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

通過這個練習，你看到我們給我們的函式 `cheese_and_crackers` 很多的參數，然後在函式裡把他們印出來。我們可以塞數字、塞變數進去函式，我們甚至可以將變數和數學運算結合在一起。

從一方面來說，函式的參數和我們生成變數時用的 `=` 賦值符號類似。事實上，如果一個物件你可以用 `=` 將其命名，你通常也可以將其作為參數傳給一個函式。

你應該看到的結果

你應該研究一下腳本的輸出，和你想像的結果對比一下看有什麼不同。

```
$ ruby ex19.rb
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.

OR, we can use variables from our script:
You have 10 cheeses!
You have 50 boxes of crackers!
Man that's enough for a party!
Get a blanket.

We can even do math inside too:
You have 30 cheeses!
You have 11 boxes of crackers!
Man that's enough for a party!
Get a blanket.

And we can combine the two, variables and math:
You have 110 cheeses!
You have 1050 boxes of crackers!
Man that's enough for a party!
Get a blanket.
$
```

加分習題

1. 倒著將腳本讀完，在每一行上面添加一行註解，說明這行程式的作用。
2. 從最後一行開始，倒著閱讀每一行，讀出所有重要的符號來。
3. 自己邊寫出至少一個函式出來，然後用十種方法運行這個函式。

习题 20: 函式和档案

回憶一下函式的要點，然後一邊做這節練習，一邊注意一下函式和檔案是如何一起協作發揮作用的。

```
input_file = ARGV[0]

def print_all(f)
  puts f.read()
end

def rewind(f)
  f.seek(0, IO::SEEK_SET)
end

def print_a_line(line_count, f)
  puts "#{line_count} #{f.readline()}"
end

current_file = File.open(input_file)

puts "First let's print the whole file:"
puts # a blank line

print_all(current_file)

puts "Now let's rewind, kind of like a tape."

rewind(current_file)

puts "Let's print three lines:"

current_line = 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)
```

特別注意一下，每次運行 `print_a_line` 時，我們是怎樣傳遞當前的行號資訊的。

你應該看到的結果

```
$ ruby ex20.rb test.txt
First let's print the whole file:

To all the people out there.
I say I don't like my hair.
I need to shave it off.

Now let's rewind, kind of like a tape.
Let's print three lines:
1 To all the people out there.
2 I say I don't like my hair.
3 I need to shave it off.

$
```

加分習題

1. 通讀腳本，在每行之前加上註解，以理解腳本裡發生的事情。
2. 每次 `print_a_line` 運行時，你都傳遞了一個叫 `current_line` 的變數。在每次呼叫函數時，印出 `current_line` 的值，跟踪一下它在 `print_a_line` 中是怎樣變成 `line_count` 的。
3. 找出腳本中每一個用到函式的地方。檢查 `def` 一行，確認參數沒有用錯。
4. 上網研究一下 file 中的 `seek` 函數是做什麼用的。試著運行 `ri file` 看看能不能從 `rdoc` 中學到更多。
5. 研究一下 `+=` 這個簡寫操作符號的作用，寫一個腳本，把這個操作符號用在裡邊試一下。

习题 21: 函式可以传回东西

你已經學過使用 `=` 給變數命名，以及將變數定義為某個數字換字串。接下來我們將讓你見證更多奇蹟。我們要示範給你的是如何使用 `=` 來將變數設置為「一個函式的值」。有一件事你需要特別注意，但待會再說，先輸入下面的腳本吧：

```
def add(a, b)
  puts "ADDING #{a} + #{b}"
  a + b
end

def subtract(a, b)
  puts "SUBTRACTING #{a} - #{b}"
  a - b
end

def multiply(a, b)
  puts "MULTIPLYING #{a} * #{b}"
  a * b
end

def divide(a, b)
  puts "DIVIDING #{a} / #{b}"
  a / b
end

puts "Let's do some math with just functions!"

age = add(30, 5)
height = subtract(78,4)
weight = multiply(90, 2)
iq = divide(100, 2)

puts "Age: #{age}, Height: #{height}, Weight: #{weight}, IQ: #{iq}"

# A puzzle for the extra credit, type it in anyway.
puts "Here is a puzzle."

what = add(age, subtract(height, multiply(weight, divide(iq, 2))))

puts "That becomes: #{what} Can you do it by hand?"
```

現在我們創造了我們自己的加減乘除數學函式：`add`、`subtract`、`multiply` 以及 `divide`。最重要的是函式的最後一行，例如 `add` 的最後一行是 `return a + b`，它實現的功能是這樣的：

1. 我們呼叫函式時使用了兩個參數：`a` 和 `b`。
2. 我們印出這個函式的功能，這裡就是計算加法（ADDING)。
3. 接下來我們告訴 Ruby 讓他做某個回傳的動作：我們將 `a+b` 的值傳回 (return)。或者你可以這麼

說：「我將 a 和 b 加起來，再把結果傳回。」

4. Ruby 將兩個數字相加，然後當函式結束時，它就可以將 a + b 的結果賦予給一個變數。

和本書裡的很多其他東西一樣，你要慢慢消化這些內容，一步一步執行下去，追蹤一下究竟發生了什麼。為了幫助你理解，本節的加分習題將讓你解決一個謎題，並且讓你學到點比較酷的東西。

你應該看到的結果

```
$ ruby ex21.rb
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50
Here is a puzzle.
DIVIDING 50 / 2
MULTIPLYING 180 * 25
SUBTRACTING 74 - 4500
ADDING 35 + -4426
That becomes:  -4391 Can you do it by hand?
$
```

加分習題

1. 如果你不是很確定 return 回來的值，試著自己寫幾個函式出來，讓它們傳回一些值。你可以將任何可以放在 `=` 右邊的東西作為一個函式的傳回值。
2. 這個腳本的結尾是一個謎題。我將一個函式的傳回值當作了另外一個函式的參數。我將它們鏈接到了一起，接跟寫數學等式一樣。這樣可能有些難讀，不過執行一下你就知道結果了。接下來，你需要試試看能不能用正常的方法實現和這個方程式一樣的功能。
3. 一旦你解決了這個謎題，試著修改一下函式里的某些部分，然後看會有什麼樣的結果。你可以有目的地修改它，讓它輸出另外一個值。
4. 最後，倒過來做一次。寫一個簡單的等式，使用一樣的函式來計算它。

這個習題可能會讓你有些頭大，不過還是慢慢來，把它當做一個遊戲，解決這樣的謎題正是寫程式的樂趣之一。後面你還會看到類似的小謎題。

习题 22: 到现在你学到了哪些东西？

這節以及下一節的習題中不會有任何代碼，所以也不會有習題答案或者加分習題。其實這節習題可以說是一個巨型的加分習題。我將讓你完成一個表格，讓你回顧你到現在學到的所有東西。

首先，回到你的每一個習題的腳本裡，把你碰到的每一個詞和每一個符號（symbol，character 的別名）寫下來。確保你的符號列表是完整的。

下一步，在每一個關鍵詞和符號後面寫出它的名字，並且說明它的作用。如果你在書裡找不到符號的名字，就上網找一下。如果你不知道某個關鍵字或者符號的作用，就回到用到該符號的章節通讀一下，並且在腳本中測試一下這個符號的用處。

你也許會碰到一些橫豎找不到答案的東西，只要把這些記在列表裡，它可以提示你讓你知道還有哪些東西不懂，等下次碰到的時候，你就不會輕易跳過了。

等你記住了這份列表中的所有內容，就試著把這份列表默寫一遍。如果發現自己漏掉或者無法從「記憶中」回想起某些內容，就回去再記一遍。

> **Warning:** 做這節習題沒有失敗，只有嘗試，請牢記這一點。

你學到的東西

這種記憶練習是枯燥無味的，所以知道它的意義很重要。它會讓你明確目標，讓你知道你所有努力的目的。

在這節練習中你學會的是各種符號的名稱，這樣讀程式碼這件事對你來說會更加容易。這和學英語時記憶字母表和基本單詞的意義是一樣的，不同的是 Ruby 中會有一些你不熟悉的符號。

慢慢來，別讓它成為你的負擔。這些符號對你來說應該比較熟悉，所以記住它們應該不是很費力的事情。你可以一次花個15 分鐘，然後休息一下。適度讓你的大腦休息一下可以讓你學得更快，而且可以讓你保持士氣。

习题 23: 阅读一些程式码

經過上一周的練習，你應該已經牢記了你的符號列表。現在你需要再花一周的時間，應用這些知識，在網上閱讀程式碼。這個任務初看會覺得很艱鉅。我將直接把你丟到深水區呆幾天，讓你竭盡全力去讀懂實實在在的專案裡的程式碼。這節練習的目的不是讓你讀懂，而是讓你學會下面的技能：

1. 找到你需要的 Ruby 程式碼。
2. 通讀程式碼，找到檔案。
3. 嘗試理解你找到的程式碼。
4. 以你現在的水準，你還不具備完全理解你找到的程式碼的能力，不過通過接觸這些程式碼，你可以熟悉真正的程式專案會是什麼樣子。

當你做這節習題時，你可以把自己當成是一個人類學家來到了一片陌生的大陸，你只懂得一丁點本地語言，但你需要接觸當地人並且生存下去。當然做習題不會碰到生存問題，這畢竟這不是荒野或者叢林。

你要做的事情如下：

1. 使用你的瀏覽器登錄 github.com，搜尋「ruby」。
2. 隨便找一個專案，然後點進去。
3. 點擊 Source 標籤，瀏覽目錄和檔案列表，直到你看到以 .rb 結尾的檔案
4. 從頭開始閱讀你找到的程式碼。把它的功能用筆記記下來。
5. 如果你看到一些有趣的符號或者奇怪的字串，你可以把它們記下來，日後再進行研究。

就是這樣，你的任務是使用你目前學到的東西，看自己能不能讀懂一些程式碼，看出它們的功能來。你可以先粗略地閱讀，然後再細讀。也許你還可以試試將難度比較大的部分一字不漏地朗讀出來。

現在再試試其它的幾個站：

- heroku.com
- rubygems.org
- bitbucket.org

在這些網站你可能還會看到以 .c 結尾的奇怪文件，不過你只需要看 .rb 結尾的檔案就可以了。

最後一個有趣的事情是你可以在這四個網站搜索 「ruby」以外的你感興趣的話題，例如你可以搜索「journalism（新聞）」，「cooking（下廚）」，「physics（物理）」，或者任何你感興趣的話題。你也許會找到一些對你有用的，且可以直接拿來用的程式碼。

习题 24: 更多练习

你離這本書第一部分的結尾已經不遠了，你應該已經具備了足夠的 Ruby 基礎知識，可以繼續學習一些程式的原理了，但你應該做更多的練習。這個練習的內容比較長，它的目的是鍛煉你的毅力，下一個習題也差不多是這樣的，好好完成它們，做到完全正確，記得仔細檢查。

```
puts "Let's practice everything."
puts "You\'d need to know \'bout escapes with \\ that do \n newlines and \t tabs."

poem = <<MULTI_LINE_STRING

\tThe lovely world
with logic so firmly planted
cannot discern \n the needs of love
nor comprehend passion from intuition
and requires an explanation
\n\t\twhere there is none.

MULTI_LINE_STRING

puts "--------------"
puts poem
puts "--------------"

five = 10 - 2 + 3 - 6
puts "This should be five: #{five}"

def secret_formula(started)
  jelly_beans = started * 500
  jars = jelly_beans / 1000
  crates = jars / 100
  return jelly_beans, jars, crates
end

start_point = 10000
beans, jars, crates = secret_formula(start_point)

puts "With a starting point of: #{start_point}"
puts "We'd have #{beans} beans, #{jars} jars, and #{crates} crates."

start_point = start_point / 10

puts "We can also do that this way:"
puts "We'd have %s beans, %s jars, and %s crates." % secret_formula(start_point)
```

你應該看到的結果

```
$ ruby ex24.rb
Let's practice everything.
You'd need to know 'bout escapes with \ that do
 newlines and    tabs.
--------------

    The lovely world
with logic so firmly planted
cannot discern
 the needs of love
nor comprehend passion from intuition
and requires an explanation

        where there is none.

--------------
This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000 jars, and 50 crates.
We can also do that this way:
We'd have 500000 beans, 500 jars, and 5 crates.
$
```

加分習題

1. 記得仔細檢查結果，從後往前倒著檢查，把程式碼朗讀出來，在不清楚的位置加上註釋。
2. 故意將程式碼改爛，執行並檢查會發生什麼樣的錯誤，並且確認你有能力改正這些錯誤。

习题 25: 更多更多的练习

我們將做一些關於函式和變數的練習，以確認你真正掌握了這些知識。這節習題對你來說可以說是一本道：寫程式，逐行研究，弄懂它。

不過這節習題還是有些不同，你不需要執行它，取而代之，你需要將它導入到 Ruby 通過自己執行函式的方式運行。

```
module Ex25
  def self.break_words(stuff)
    # This function will break up words for us.
    words = stuff.split(' ')
    words
  end

  def self.sort_words(words)
    # Sorts the words.
    words.sort()
  end

  def self.print_first_word(words)
    # Prints the first word and shifts the others down by one.
    word = words.shift()
    puts word
  end

  def self.print_last_word(words)
    # Prints the last word after popping it off the end.
    word = words.pop()
    puts word
  end

  def self.sort_sentence(sentence)
    # Takes in a full sentence and returns the sorted words.
    words = break_words(sentence)
    sort_words(words)
  end

  def self.print_first_and_last(sentence)
    # Prints the first and last words of the sentence.
    words = break_words(sentence)
    print_first_word(words)
    print_last_word(words)
  end

  def self.print_first_and_last_sorted(sentence)
    # Sorts the words then prints the first and last one.
    words = sort_sentence(sentence)
    print_first_word(words)
    print_last_word(words)
  end
end
```

首先以正常的方式 `ruby ex25.rb` 運行，找出裡面的錯誤，並把它們都改正過來。然後你需要接著下面的答案章節完成這節習題。

你應該看到的結果

這節練習我們將在你之前用來做算術的 Ruby 編譯器(IRB)裡，用交互的方式和你的 `.rb` 作交流。

這是我做出來的樣子：

```
$ irb
irb(main):001:0> require './ex25'
=> true
irb(main):002:0> sentence = "All good things come to those who wait."
=> "All good things come to those who wait."
irb(main):003:0> words = Ex25.break_words(sentence)
=> ["All", "good", "things", "come", "to", "those", "who", "wait."]
irb(main):004:0> sorted_words = Ex25.sort_words(words)
=> ["All", "come", "good", "things", "those", "to", "wait.", "who"]
irb(main):005:0> Ex25.print_first_word(words)
All
=> nil
irb(main):006:0> Ex25.print_last_word(words)
wait.
=> nil
irb(main):007:0> Ex25.wrods
NoMethodError: undefined method `wrods' for Ex25:Module
        from (irb):6
irb(main):008:0> words
=> ["good", "things", "come", "to", "those", "who"]
irb(main):009:0> Ex25.print_first_word(sorted_words)
All
=> nil
irb(main):010:0> Ex25.print_last_word(sorted_words)
who
=> nil
irb(main):011:0> sorted_words
=> ["come", "good", "things", "those", "to", "wait."]
irb(main):012:0> Ex25.sort_sentence(sentence)
=> ["All", "come", "good", "things", "those", "to", "wait.", "who"]
irb(main):013:0> Ex25.print_first_and_last(sentence)
All
wait.
=> nil
irb(main):014:0> Ex25.print_first_and_last_sorted(sentence)
All
who
=> nil
irb(main):015:0> ^D
$
```

我們來逐行分析一下每一步實現的是什麼：

1. 在第 2 行你 require 了自己的 `./ex25.rb` Ruby 檔案，和你做過的其他 require 一樣 $ 。在 require 的時候你不需要加 `.rb` 後綴。這個過程裡，你將這個檔案當做了一個 `module` (模組)來使用，你在這個模組裡定義的函式也可以直接呼叫出來。
2. 第 4 行你創造了一個用來處理的 `sentence` (句子)。
3. 第 6 行你使用了 `Ex25` 模組呼叫了你的第一個函式 `Ex25.break_words` 。其中的 `.` (dot, period)

符號可以告訴 Ruby：「Hi，我要執行 `Ex25` 裡的那個叫 `break_word` 的函式！」

4. 第 8 行我們只是輸入 `Ex25.sort_words` 來得到一個排序過的句子。
5. 10-15 行我們使用 `Ex25.print_first_word` 和 `Ex25.print_last_word` 將第一個和最後一個詞印出來。
6. 第 16 行比較有趣。我把 `words` 變數寫錯成了 `wrods`，所以 Ruby 在 17-18 行給了一個錯誤訊息。
7. 第 19-20 行我們印出了修改過後的詞彙列表。第一個和最後一個詞我們已經印過了，所以在這裡沒有再印出來。
8. 剩下的行你需要自己分析一下，就留作你的加分習題了。

加分習題

1. 研究答案中沒有分析過的行，找出它們的來龍去脈。確認自己明白了自己使用的是模組 Ex25 中定義的函式。
2. 我們將我們的函式放在一個 `module` 裡式因為他們擁有自己的 命名空間 (namespace)。這樣如果有其他人寫了一個函式也叫 `break_words`，這樣就不會發生碰創。無論如何，輸入 `Ex25.` 是一件很煩人的事。有一個比較方便的作法，你可以輸入 `include Ex25`，這相當於說：「我要將所有 Ex25 這個 mudle 裡的所有東西 include 到我現在的 module 裡。」
3. 試著在你正在使用 IRB 時，弄爛檔案會發生什麼事。你可能要執行 CTRL-D (Windows下是CTRL-Z) 才能把 IRB 關掉 reload 一次。

习题 26: 恭喜你，现在来考试了！

你已經差不多完成這本書的前半部分了，不過後半部分才是更有趣的。你將學到邏輯，並通過條件判斷實現有用的功能。

在你繼續學習之前，你有一道試題要做。這道試題很難，因為它需要你修正別人寫的程式碼。當你成為程式設計師以後，你將需要經常面對別的程式設計師的程式碼，也許還有他們的傲慢態度，他們會經常說自己的程式碼是完美的。

這樣的程式設計師是自以為是不在乎別人的蠢貨。優秀的科學家會對他們自己的工作持懷疑態度，同樣，優秀的程式設計師也會認為自己的程式碼總有出錯的可能，他們會先假設是自己的程式碼有問題，然後用排除法清查所有可能是自己有問題的地方，最後才會得出「這是別人的錯誤」這樣的結論。

在這節習題中，你將面對一個程度糟糕的程式設計師，並改好他的程式碼。我將習題 24 和25 胡亂拷貝到了一個檔案中，隨機地刪掉了一些字，然後添加了一些錯誤進去。大部分的錯誤是 Ruby 在執行時會告訴你的，還有一些算術錯誤是你要自己找出來的。再剩下來的就是格式和拼寫錯誤了

所有這些錯誤都是程式設計師很容易犯的，就算有經驗的程式設計師也不例外。

你的任務是將此檔案修改正確，用你所有的技能改進這個腳本。你可以先分析這個文件，或者你還可以把它像學期論文一樣印出來，修正裡邊的每一個缺陷，重複修正和運行的動作，直到這個腳本可以完美地運行起來。在整個過程中不要尋求幫助，如果你卡在某個地方無法進行下去，那就休息一會晚點再做。

就算你需要幾天才能完成，也不要放棄，直到完全改對為止。

最後要說的是，這個練習的目的不是寫程式，而是修正現有的程式，習題放在下面的網站：

http://ruby.learncodethehardway.org/book/exercise26.txt

從那裡把程式碼複製過來，命名為 `ex26.rb`，這也是本書唯一一處允許你複製貼上的地方。

习题 27: 记住逻辑关系

到此為止你已經學會了讀寫檔案，命令列處理，以及很多 Ruby 數學運算功能。

今天，你將要開始學習邏輯了。你要學習的不是研究院裡的高深邏輯理論，只是程式設計師每天都用到的讓程式跑起來的基礎邏輯知識。

學習邏輯之前你需要先記住一些東西。這個練習我要求你一個星期完成，不要擅自修改 schedule，就算你煩得不得了，也要堅持下去。這個練習會讓你背下來一系列的邏輯表格，這會讓你更容易地完成後面的習題。

需要事先警告你的是：這件事情一開始一點樂趣都沒有，你會一開始就覺得它很無聊乏味，但它的目的是教你程式設計師必須的一個重要技能 — 一些重要的概念是必須記住的，一旦你明白了這些概念，你會獲得相當的成就感，但是一開始你會覺得它們很難掌握，就跟和烏賊摔跤一樣，而等到某一天，你會刷的一下豁然開朗。你會從這些基礎的記憶學習中得到豐厚的回報。

這裡告訴你一個記住某樣東西，而不讓自己抓狂的方法：在一整天裡，每次記憶一小部分，把你最需要加強的部分標記起來。不要想著在兩小時內連續不停地背誦，這不會有什麼好的結果。不管你花多長時間，你的大腦也只會留住你在前15 或者30 分鐘內看過的東西。

取而代之，你需要做的是創建一些索引卡片，卡片有兩列內容，正面寫下邏輯關係，反面寫下答案。你需要做到的結果是：拿出一張卡片來，看到正面的表達式，例如「True or False」，你可以立即說出背面的結果是「True」！堅持練習，直到你能做到這一點為止。

一旦你能做到這一點了，接下來你需要每天晚上自己在筆記本上寫一份真值表出來。不要只是抄寫它們，試著默寫真值表，如果發現哪裡沒記住的話，就飛快地撇一眼這裡的答案。這樣將訓練你的大腦讓它記住整個真值表。

不要在這上面花超過一周的時間，因為你在後面的應用過程中還會繼續學習它們。

邏輯術語

在 Ruby 中我們會用到下面的術語（符號或者詞彙）來定義事物的真(True)或者假(False)。電腦的邏輯就是在程式的某個位置檢查這些符號或者變數組合在一起表達的結果是真是假。

- `and` 和
- `or` 或
- `not` 非
- `!=` (not equal) 不等於
- `==` (equal) 等於
- `>=` (greater-than-equal) 大於等於
- `<=` (less-than-equal) 小於等於

- true 真
- false 假

其實你已經見過這些符號了，但這些詞彙你可能還沒見過。這些詞彙(and, or, not)和你期望的效果其實是一樣的，跟英語裡的意思一模一樣。

真值表

我們將使用這些符號來創建你需要記住的真值表。

NOT	True?
not False	True
not True	False

OR	True?
True or False	True
True or True	True
False or True	True
False or False	False

AND	True?
True and False	False
True and True	True
False and True	False
False and False	False

NOT OR	True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	True?
not (True and False)	True
not (True and True)	False
not (False and True)	True

NOT AND	True?
not (False and False)	True

!=	True?
1 != 0	True
1 != 1	False
0 != 1	True
0 != 0	False

==	True?
1 == 0	False
1 == 1	True
0 == 1	False
0 == 0	True

現在使用這些表格創建你自己的卡片，再花一個星期慢慢記住它們。記住一點，這本書不會要求你成功或者失敗，只要每天盡力去學，在盡力的基礎上多花一點功夫就可以了。

习题 28: 布林（Boolean）表示式练习

上一節你學到的邏輯組合的正式名稱是「布林邏輯表示式(boolean logic expression)」。在程式中，布林邏輯可以說是無處不在。它們是電腦運算的基礎和重要組成部分，掌握它們就跟學音樂掌握音階一樣重要。

在這節練習中，你將在 IRB 裡使用到上節學到的邏輯表示式。先為下面的每一個邏輯問題寫出你認為的答案，每一題的答案要嘛為 True 要嘛為 False。寫完以後，你需要將 IRB 運行起來，把這些邏輯語句輸入進去，確認你寫的答案是否正確。

```
1\. true and true
2\. false and true
3\. 1 == 1 and 2 == 1
4\. "test" == "test"
5\. 1 == 1 or 2 != 1
6\. true and 1 == 1
7\. false and 0 != 0
8\. true or 1 == 1
9\. "test" == "testing"
10\. 1 != 0 and 2 == 1
11\. "test" != "testing"
12\. "test" == 1
13\. not (true and false)
14\. not (1 == 1 and 0 != 1)
15\. not (10 == 1 or 1000 == 1000)
16\. not (1 != 10 or 3 == 4)
17\. not ("testing" == "testing" and "Zed" == "Cool Guy")
18\. 1 == 1 and not ("testing" == 1 or 1 == 0)
19\. "chunky" == "bacon" and not (3 == 4 or 3 == 3)
20\. 3 == 3 and not ("testing" == "testing" or "Ruby" == "Fun")
```

在本節結尾的地方我會給你一個理清複雜邏輯的技巧。

所有的布林邏輯式都可以用下面的簡單流程得到結果：

1. 找到相等判斷的部分 (== or !=)，將其改寫為其最終值(True 或False)。
2. 找到括號裡的 and/or，先算出它們的值。
3. 找到每一個 not，算出他們反過來的值。
4. 找到剩下的 and/or，解出它們的值。
5. 等你都做完後，剩下的結果應該就是 True 或者 False 了。

下面我們以 #20 邏輯式示範一下：

```
3 != 4 and not ("testing" != "test" or "Ruby" == "Ruby")
```

接下來你將看到這個複雜表達式是如何逐級解析為一個單獨結果的：

1. 出每一個等值判斷:
   - `3 != 4` 為 **True**: `true and not ("testing" != "test" or "Ruby" == "Ruby")`
   - `"testing" != "test"` 為 **True**: `true and not (true or "Ruby" == "Ruby")`
   - `"Ruby" == "Ruby"` : `true and not (true or true)`

2. 找到 `()` 中的每一個 and/or :
   - `(true or true)` is **True**: `true and not (true)`

3. 找到每一個not 並將其逆轉:
   - `not (true)` is **False**: `true and false`

4. 找到剩下的and/or，解出它們的值:
   - `true and false` is **False**

這樣我們就解出了它最終的值為 False .

> **Warning:** 雜的邏輯表達式一開始看上去可能會讓你覺得很難。而且你也許已經碰壁過了，不過別灰心，這些「邏輯體操」式的訓練只是讓你逐漸習慣起來，這樣後面你可以輕易應對程式裡邊更酷的一些東西。只要你堅持下去，不放過自己做錯的地方就行了。如果你暫時不太能理解也沒關係，弄懂的時候總會到來的。

你應該看到的結果

以下內容是在你自己猜測結果以後，通過和 IRB 對話得到的結果：

```
$ irb
ruby-1.9.2-p180 :001 > true and true
 => true
ruby-1.9.2-p180 :002 > 1 == 1 and 2 == 2
 => true
```

加分習題

1. Ruby 裡還有很多和 `!=`、`==` 類似的操作符號。試著盡可能多的列出 Ruby 中的「等價運算符號」。例如 `<` 或是 `<=`。
2. 寫出每一個等價運算符號的名稱。例如 `!=` 叫「not equal（不等於」。
3. 在 IRB 裡測試新的布林邏輯式。在敲 Enter 前你需要喊出它的結果。不要思考，憑自己的第一直覺就可以了。把表達式和結果用筆寫下來再敲 Enter，最後看自己做對多少，做錯多少。

4. 把習題 3 那張紙丟掉，以後你不再需要查詢它了。

习题 29: 如果（if）

這裡是你接下要寫的作業，這段介紹了 `if-statement` （if 語句）。把這段輸入進去，讓它能夠正確執行。然後我們看看你是否有收穫。

```
people = 20
cats = 30
dogs = 15

if people < cats
  puts "Too many cats! The world is doomed!"
end

if people > cats
  puts "Not many cats! The world is saved!"
end

if people < dogs
  puts "The world is drooled on!"
end

if people > dogs
  puts "The world is dry!"
end

dogs += 5

if people >= dogs
  puts "People are greater than or equal to dogs."
end

if people <= dogs
  puts "People are less than or equal to dogs."
end

if people == dogs
  puts "People are dogs."
end
```

你應該看到的結果

```
$ ruby ex29.rb
Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs.
People are dogs.
$
```

加分習題

猜猜「if 語句」是什麼，它有什麼用處。在做下一道習題前，試著用自己的話回答下面的問題:

1. 你認為 if 對於它下一行的程式碼做了什麼？
2. 把習題 29 中的其它布林表示式放到「if 語句」中會不會也可以運行呢？試一下。
3. 如果把變數 people、cats 和 dogs 的初始值改掉，會發生什麼事情？

习题 30: Else 和 If

前一習題中你寫了一些「if 語句 (if-statements)」，並且試圖猜出它們是什麼，以及實現的是什麼功能。在你繼續學習之前，我給你解釋一下上一節的加分習題的答案。上一節的加分習題你做過了吧，有沒有？

1. 你認為 if 對於它下一行的代碼做了什麼？`if` 語句為程式碼創建了一個所謂的「分支(branch)」，就跟RPG遊戲中的情節分支一樣。if 語句告訴你的腳本：「如果這個布林表示式為真，就執行接下來的程式碼，否則就跳過這一段。」
2. 把習題29中的其它布林表示式放到 if 語句中會不會也可以執行呢？試一下。可以。而且不管多複雜都可以，雖然寫複雜的東西通常是一種不好的寫作風格。
3. 如果把變數 people、cats和 dogs 的初始值改掉，會發生什麼事情？因為你比較的對象是數字，如果你把這些數字改掉的話，某些位置的 if 語句會被演繹為 True，而它下面的程式區段將被運行。你可以試著修改這些數字，然後在頭腦裡假想一下那一段程式碼會被運行。

把我的答案和你的答案比較一下，確認自己真正懂得程式碼「區段(block)」的含義。這點對於你下一節的習題習很重要，因為你將會寫很多的if 語句。

把這一段寫下來，並讓它運行起來：

```
people = 30
cars = 40
buses = 15

if cars > people
  puts "We should take the cars."
elsif cars < people
  puts "We should not take the cars."
else
  puts "We can't decide."
end

if buses > cars
  puts "That's too many buses."
elsif buses < cars
  puts "Maybe we could take the buses."
else
  puts "We still can't decide."
end

if people > buses
  puts "Alright, let's just take the buses."
else
  puts "Fine, let's stay home then."
end
```

你應該看到的結果

```
$ ruby ex30.rb
We should take the cars.
Maybe we could take the buses.
Alright, let's just take the buses.
$
```

加分習題

1. 猜想一下 `elsif` 和 `else` 的功能。
2. 將 `cars`、 `people` 和 `buses` 的數量改掉，然後追溯每一個if語句。看看最後會印出什麼來。
3. 試著寫一些複雜的布林表示式，例如 `cars > people and buses < cars` 。 在每一行的上面寫註解，說明這一行的功用。

习题 31: 做出决定

這本書的上半部分，你印出了一些東西，並且呼叫了函式，不過一切都是直線式進行的。你的腳本從最上面一行開始，一路運行到結束，但其中並沒有決定程式流向的分支點。現在你已經學會了 `if`、`else` 和 `elsif`，你就可以開始建立包含條件判斷的腳本了。

上一個腳本中你寫了一系列的簡單提問測試。這節的腳本中，你將需要向使用者提問，依據使用者的答案來做出決定。把腳本寫下來，多多搗鼓一陣子，看看他的運作原理是什麼。

```
def prompt
  print "> "
end

puts "You enter a dark room with two doors.  Do you go through door #1 or door #2?"

prompt; door = gets.chomp

if door == "1"
  puts "There's a giant bear here eating a cheese cake.  What do you do?"
  puts "1\. Take the cake."
  puts "2\. Scream at the bear."

  prompt; bear = gets.chomp

  if bear == "1"
    puts "The bear eats your face off.  Good job!"
  elsif bear == "2"
    puts "The bear eats your legs off.  Good job!"
  else
    puts "Well, doing #{bear} is probably better.  Bear runs away."
  end

elsif door == "2"
  puts "You stare into the endless abyss at Cthuhlu's retina."
  puts "1\. Blueberries."
  puts "2\. Yellow jacket clothespins."
  puts "3\. Understanding revolvers yelling melodies."

  prompt; insanity = gets.chomp

  if insanity == "1" or insanity == "2"
    puts "Your body survives powered by a mind of jello.  Good job!"
  else
    puts "The insanity rots your eyes into a pool of muck.  Good job!"
  end

else
  puts "You stumble around and fall on a knife and die.  Good job!"
end
```

這裡的重點是你可以在 `if` 語句中內部再放一個 `if` 語句。這是一個很強大的功能，可以用來建立「巢狀(nested)」的決定 (decision)。

你需要理解 `if` 語句包含 `if` 語句的概念。做一下加分習題，這樣你會確信自己真正理解了它們。

你應該看到的結果

```
我在玩一個小冒險遊戲。我的水準不怎麼樣。
```

```
$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1\. Take the cake.
2\. Scream at the bear.
> 2
The bear eats your legs off.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1\. Take the cake.
2\. Scream at the bear.
> 1
The bear eats your face off.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 2
You stare into the endless abyss at Cthuhlu's retina.
1\. Blueberries.
2\. Yellow jacket clothespins.
3\. Understanding revolvers yelling melodies.
> 1
Your body survives powered by a mind of jello.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 2
You stare into the endless abyss at Cthuhlu's retina.
1\. Blueberries.
2\. Yellow jacket clothespins.
3\. Understanding revolvers yelling melodies.
> 3
The insanity rots your eyes into a pool of muck.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> stuff
You stumble around and fall on a knife and die.  Good job!

$ ruby ex31.rb
You enter a dark room with two doors.  Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.  What do you do?
1\. Take the cake.
2\. Scream at the bear.
> apples
Well, doing apples is probably better.  Bear runs away.
```

加分習題

為遊戲添加新的部分，改變玩家做決定的位置。盡自己能力擴充這個遊戲，不過別把遊戲弄得太詭異了。

习题 32: 回圈和阵列

現在你應該有能力寫更有趣的程式出來了。如果你能夠一直跟得上，你應該已意識到你能將之前學到的將 `if` 語句 和 「布林表示式」這些東西結合起來，讓程式做出一些聰明的事了。

然而，我們的城市還需要能很快地完成重複的事情。這節習題中我們將使用 `for-loop` (for 迴圈) 來建立和印出各式的陣列。在做習題的過程中，你將會逐漸搞懂它們是怎麼回事。現在我不會告訴你，你需要自己找到答案。

在你開始使用 for 迴圈之前，你需要在某個位置存放迴圈的結果。最後的方法是使用陣列 `array` 。一個陣列，就是一個按照順序存放東西的容器。陣列並不複雜，你只是要學習一點新的語法。首先我們來看看如何建立一個陣列：

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green']
weights = [1, 2, 3, 4]
```

你要做的是以 `[` 左中括號開頭「打開」陣列，然後寫下你要放入陣列的東西、用逗號 `,` 隔開，就跟函式的參數一樣，最後你需要用 `]` 右中括號結束陣列的定義。然後 Ruby 接收這個陣列以及裡面所有的內容，將其賦予給一個變數。

> **Warning:** 對於不會寫程式的人來說這是一個困難點。習慣性思維告訴你的大腦大地是平的。記得上一個練習中的巢狀 if 語句吧，你可能覺得要理解它有些難度，因為生活中一般人不會去想這樣的問題，但這樣的問題在程式中幾乎到處都是。你會看到一個函式呼叫用另外一個包含 if 語句的函式，其中又有巢狀陣列的陣列。如果你看到這樣的東西一時無法弄懂，就用紙筆記下來，手動分割下去，直到弄懂為止。

現在我們將使用迴圈建立一些陣列，然後將它們印出來：

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for-loop goes through an array
for number in the_count
  puts "This is count #{number}"
end

# same as above, but using a block instead
fruits.each do |fruit|
  puts "A fruit of type: #{fruit}"
end

# also we can go through mixed arrays too
for i in change
  puts "I got #{i}"
end

# we can also build arrays, first start with an empty one
elements = []

# then use a range object to do 0 to 5 counts
for i in (0..5)
  puts "Adding #{i} to the list."
  # push is a function that arrays understand
  elements.push(i)
end

# now we can puts them out too
for i in elements
  puts "Element was: #{i}"
end
```

你應該看到的結果

```
$ ruby ex32.rb
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got 'pennies'
I got 2
I got 'dimes'
I got 3
I got 'quarters'
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
Adding 4 to the list.
Adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
$
```

加分習題

1. 注意一下 range `(0..5)`。查一下 `Range` class (類別) 並弄懂它。
2. 在第 24 行，你可以直接將 `elements` 賦值為 `(0..5)`，而不需使用 for 迴圈嗎？
3. 在 Ruby 文件中可以找到關於陣列的內容，仔細閱讀一下，除了 `push` 以外，陣列還支援那些操作？

习题 33: While 回圈

接下來是一個更在你意料之外的概念：`while-loop` (while迴圈)。`while` 迴圈會一直執行它下面的程式碼區段，直到它對應的布林表示式為 `false` 才會停下來。

等等，你還能跟的上這些術語吧？如果我們寫了這樣一個語句：`if items > 5` 或者是 `for fruit in fruits`，那就是在開始一個程式碼區段 (code block)，新的程式碼區段是需要被縮排的，最後再以 `end` 語句結尾。只有將程式碼用這樣的方式格式化，Ruby 才能知道你的目的。如果你不太明白這一點，就回去看看「if 語句」、「函式」和「for 迴圈」章節，直到你明白為止。

接下來的店席將訓練你的大腦去閱讀這些結構化的與集，這和我們將布林表示式燒錄到你的大腦中的過程有點類似。

回到 `while` 迴圈，它所作的和 `if` 語句類似，也是去檢查一個布林表示式的真假，不一樣的是它下面的程式碼區段不是只被執行一次，而是執行完後再趟回到 `while` 所在的位置，如此重複進行，直到 while 表示式為 **false** 為止。

`while` 迴圈有一個問題：那就是有時它會永不結束。如果你的目的是循環到宇宙毀滅為止，那這樣也挺好的，不過其他的情況下你的迴總需要有一個結束點。

為了避免這樣的問題，你需要遵循下面的規定：

1. 盡量少用 `while` 迴圈，大部分時候 `for` 迴圈是更好的選擇。
2. 重複檢查你的 `while` 語句，確定你測試的布林表示式最終會變成 **false**。
3. 如果不確定，就在 `while` 迴圈的結尾印出你要測試的值。看看它的變化。

在這節練習中，你將通過上面的三樣事情學會 `while` 迴圈：

```
i = 0
numbers = []

while i < 6
  puts "At the top i is #{i}"
  numbers.push(i)

  i = i + 1
  puts "Numbers now: #{numbers}"
  puts "At the bottom i is #{i}"
end

puts "The numbers: "

for num in numbers
  puts num
end
```

你應該看到的結果

```
$ ruby ex33.rb
At the top i is 0
Numbers now:  [0]
At the bottom i is 1
At the top i is 1
Numbers now:  [0, 1]
At the bottom i is 2
At the top i is 2
Numbers now:  [0, 1, 2]
At the bottom i is 3
At the top i is 3
Numbers now:  [0, 1, 2, 3]
At the bottom i is 4
At the top i is 4
Numbers now:  [0, 1, 2, 3, 4]
At the bottom i is 5
At the top i is 5
Numbers now:  [0, 1, 2, 3, 4, 5]
At the bottom i is 6
The numbers:
0
1
2
3
4
5
```

加分習題

1. 將這個 `while` 迴圈改成一個函式，將測試條件 `(i < 6)` 中的 6 換成一個變數。
2. 使用這個函式重寫你的腳本，並使用不同的數字進行測試。
3. 為函式添加另一個參數，這個參數用來定義第 8 行的 `+1`，這樣你就可以讓它任意加值了。
4. 再使用該函式重寫一遍這個腳本。看看效果如何。
5. 接下來使用 `for` 迴圈和 range 把這個腳本再寫一遍。你還需要中間的加值操作嗎？如果你不去掉它，會有什麼樣的結果？

有可能你會碰到程序跑著停不下來了，這時你只要按著 CTRL 再敲 c (CTRL-c)，這樣程式就會中斷下來了。

习题 34: 存取阵列里的元素

陣列非常有用，但只有你存取裡面的內容時，它才能發揮出作用來。你已經學會了案順序讀出陣列中的內容，但如果你要得到第 5 個元素該怎麼辦呢？你需要知道如何存取陣列中的元素。存取第一個元素的方法是這樣的：

```
animals = ['bear', 'tiger', 'penguin', 'zebra']
bear = animals[0]
```

你定義一個 animals 的陣列，然後你用 `0` 來存取第一個元素！？這是怎麼回事啊？因為數學裡面就是這樣，所以 Ruby 的陣列也是從 0 開始的。雖然看起來很奇怪，這樣定義其實有它的好處。

最好的解釋方式勢將你平常使用數字的方式和程式設計師使用數字的方式做比較。

假設你在觀看上面陣列中的四種動物：(`['bear', 'tiger', 'penguin', 'zebra']`) 的賽跑。而它們比賽的名次正好跟陣列中的順樹一樣。這是一場很刺激的比賽，因為這些動物沒打算吃掉對方，而且比賽還真的舉辦起來了。結果你的朋友來晚了，他想知道誰贏了比賽，他會問你「嘿，誰是第 0 名？」嗎？不會的，他會問「嘿，誰是第 1 名？」

這是因為動物的次序是很重要的。沒有第一個就沒有第二個，沒有第二個話也不會有第三個。第零個是不存在的，因為零的意思是什麼都沒有。「什麼都沒有」怎麼贏比賽嘛？完全不和邏輯。這樣的數字我們稱之為「序數(ordinal number)」

而程式設計師不能用這種方式思考問題，因為他們可以從陣列中的任何一個位置取出一個元素來。對程式設計師來說，上述的陣列更像是一疊卡片。如果他們想要 tiger，就抓它出來，如果想要 zeebra，也一樣抓出來。要隨機地抓陣列裡的內容，陣列的每一個元素都應該要有一個地址(address)，或者一個「索引(index)」，而最好的方式就是使用以 0 開頭的索引。相信我說的這一點吧，這種方式獲取元素會更容易。而這一類的數字被稱為「基數(cardinal number)」，它意味著你可以任意抓取元素，所以我們需要一個 0 號元素。

那麼，這些知識對你的陣列操作有什麼幫助呢？很簡單，每次你對自己說：「我要第 3 隻動物」時，你需要將「序數」轉換成「基數」，只要將前者減 1 就可以了。第 3 隻動物的索引是 2，也就是 penguin。由於你一輩子都在跟序數打交道，所以你需要這種方式來獲得基數，只要減 1 就都搞定了。

記住：ordinal == 有序，以 1 開始；cardinal == 隨機存取，以 0 開始。

讓我們練習一下。定義一個動物列表，然後跟著做後面的習題，你需要寫出所指位置的動物名稱。如果我用的是「first」、「second」等說法。那說明我用的是敘述，所以你需要減去 1。如果我給你的是基數 (0, 1, 2)，你只要直接使用即可。

```
animals = ['bear', 'python', 'peacock', 'kangaroo', 'whale', 'platypus']
```

The animal at 1. The 3rd animal. The 1st animal. The animal at 3. The 5th animal. The animal at 2. The 6th animal. The animal at 4.

對於上述某一條，以這樣的格式寫出一個完整的句子：「The 1st animal is at 0 and is a bear.」然後倒過來念「 “The animal at 0 is the 1st animal and is a bear.」

使用 IRB 去檢查你的答案。

Hint: Ruby 還有一些便利的 method 是屬於在陣列中存取特定元素的用法。：`animals.first` 和 `animals.last`。

加分習題

1. 上網搜尋一下關於 序數 （ ordinal number) 和基數 (cardinal number) 的知識並閱讀一下。
2. 以你對於這些數字類型的了解，解釋一下為什麼今年是 2010 年。呢是：你不能隨便挑選年份。
3. 再寫一些陣列，用一樣的方式做出索引，確認自己可以在兩種數字之間互相翻譯。
4. 使用 IRB 檢查自己的答案。

> **Warning:** 會有程式設計師告訴你，叫你去閱讀一個叫「Dijkstra」的人寫的關於數字的主題。我建議你還是不讀為妙，除非你喜歡聽一個在寫程式這一行剛興起時就停止了從事寫程式工作的人對你大吼大叫。

习题 35: 分支 (Branches) 和函式 (Functions)

你已經學會了 `if` 語句、函式、還有陣列。現在你要練習扭轉一下思維了。把下面的代碼寫下來，看你是否能弄懂它實現的是什麼功能。

```
def prompt()
  print "> "
end

def gold_room()
  puts "This room is full of gold.  How much do you take?"

  prompt; next_move = gets.chomp
  if next_move.include? "0" or next_move.include? "1"
    how_much = next_move.to_i()
  else
    dead("Man, learn to type a number.")
  end

  if how_much < 50
    puts "Nice, you're not greedy, you win!"
    Process.exit(0)
  else
    dead("You greedy bastard!")
  end
end

def bear_room()
  puts "There is a bear here."
  puts "The bear has a bunch of honey."
  puts "The fat bear is in front of another door."
  puts "How are you going to move the bear?"
  bear_moved = false

  while true
    prompt; next_move = gets.chomp

    if next_move == "take honey"
      dead("The bear looks at you then slaps your face off.")
    elsif next_move == "taunt bear" and not bear_moved
      puts "The bear has moved from the door. You can go through it now."
      bear_moved = true
    elsif next_move == "taunt bear" and bear_moved
      dead("The bear gets pissed off and chews your leg off.")
    elsif next_move == "open door" and bear_moved
      gold_room()
    else
      puts "I got no idea what that means."
    end
  end
end
```

```
end

def cthulu_room()
  puts "Here you see the great evil Cthulu."
  puts "He, it, whatever stares at you and you go insane."
  puts "Do you flee for your life or eat your head?"

  prompt; next_move = gets.chomp

  if next_move.include? "flee"
    start()
  elsif next_move.include? "head"
    dead("Well that was tasty!")
  else
    cthulu_room()
  end
end

def dead(why)
  puts "#{why}  Good job!"
  Process.exit(0)
end

def start()
  puts "You are in a dark room."
  puts "There is a door to your right and left."
  puts "Which one do you take?"

  prompt; next_move = gets.chomp

  if next_move == "left"
    bear_room()
  elsif next_move == "right"
    cthulu_room()
  else
    dead("You stumble around the room until you starve.")
  end
end

start()
```

你應該看到的結果

你可以結果：

```
$ ruby ex35.rb
You are in a dark room.
There is a door to your right and left.
Which one do you take?
> left
There is a bear here.
The bear has a bunch of honey.
The fat bear is in front of another door.
How are you going to move the bear?
> taunt bear
The bear has moved from the door. You can go through it now.
> open door
This room is full of gold.  How much do you take?
> asf
Man, learn to type a number. Good job!
$
```

加分習題

1. 把這個遊戲的地圖畫出來，把自己的路線也畫出來。
2. 改正你所有的錯誤，包括拼寫錯誤。
3. 為你不懂的函式寫註解。記得 **RDoc** 中的註釋嗎？
4. 為遊戲添加更多元素。通過怎樣的方式可以簡化並且擴充遊戲的功能呢？
5. 這個 gold_room 遊戲使用了奇怪的方式讓你鍵入一個數字。這種方式會導致什麼樣的bug？你可以用比檢查 0、1更好的方式判斷輸入是否是數字嗎？ `to_i()` 這個函式可以給你一些頭緒。

习题 36: 设计和测试

現在你已經學會了「`if` 語句」，我將給你一些使用 `for` 迴圈和 `while` 迴圈的規則，一面你日後碰到麻煩。我還會教你一些測試的小技巧，以便你能發現自己程式的問題。最後，你將需要設計一個和上節類似的小遊戲，不過內容略有更改。

If 語句的規則

1.每一個「if語句」必伴隨須一個 `else`。 2. 如果這個 `else` 因為沒有意義，而永遠都沒被執行到，那你必須在 `else` 語句後面使用一個叫 `die` 的函式，讓它印出錯誤並死給你看，這和上一節的習題類似，這樣你可以找到很多的錯誤。 3. 千萬不要使用超過兩層的 `if` 語句，最好盡量保持只有 1 層。那你就需要把第二個 if 移到另一個函式裡面。 4. 將 if 語句當做段落來對待，其中的每一個 `if`、`elsif`、`else` 組合就跟一個段落的句子組合一樣。在這種組合的最前面和最後面留一個空行以作區分。 5. 你的布林測試應該很簡單，如果它們很複雜的話，你需要將它們的運算式先放到一個變數裡，並且為變數取一個好名字。

如果你遵循上面的規則，你就會寫出比大部分程式設計師都好的程式碼來。回到上一個練習中，看看我有沒有遵循這些規則，如果沒有的話，就將其改正過來。

> **Warning:** 在日常寫程式中不要成為這些規則的奴隸。在訓練中，你需要通過這些規則的應用來鞏固你學到的知識，而在實際寫程式中這些規則有時其實很蠢。如果你覺得哪個規則很蠢，就別使用它。

Rules For Loops

1. 只有在迴圈循環永不停止時使用 `while` 迴圈，這意味著你可能永遠都用不到。這條只有 Ruby 中成立，其他的語言另當別論。
2. 其他類型的迴圈都使用 `for` 迴圈，尤其是在迴圈的對象數量固定或者有限的情況下。

除錯（Debug) 的小技巧

1. 不要使用「debugger」。Debugger 所作的相當於對病人的全身掃描。你並不會得到某方面的有用資訊，而且你會發現它輸出的資訊太多，而且大部分沒有用，或者只會讓你更困惑。
2. 最好的除錯技巧是使用 `puts` 或 `p` 在各個你想要檢查的關鍵環節將關鍵變數印出來，從而檢查哪裡是否有錯。
3. 讓程式一部分一部分地運行起來。不要等一個很長的腳本寫完後才去運行它。寫一點，運行一點，再修改一點。

家庭作業

寫一個和上節練習類似的遊戲。同類的任何題材的遊戲都可以，花一個星期讓它盡可能有趣一些。作為加分習題，你可以盡量多使用陣列、函式、以及模組（記得習題 13 嗎？），而且盡量多弄一些新的 Ruby

程式讓你的遊戲跑起來。

過有一點需要注意，你應該把遊戲的設計先寫出來。在你開始寫程式碼之前，你應該設計出遊戲的地圖，創建出玩家會碰到的房間、怪物、以及陷阱等環節。

一旦搞定了地圖，你就可以寫寫程式碼了。如果你發現地圖有問題，就調整一下地圖，讓寫程式碼和地圖互相符合。

最後一個建議：每一個程式設計師在開始一個新的大項目時，都會被非理性的恐懼影響到。為了避免這種恐懼，他們會拖延時間，到最後一事無成。我有時會這樣，每個人都會有這樣的經歷，避免這種情況的最好的方法是把自己要做的事情列出來，一次完成一樣。

開始做吧。先做一個小一點的版本，擴充它讓它變大，把自己要完成的事情一一列出來，然後逐個完成就可以了。

习题 37: 复习各种符号

現在該複習你學過的符號和 Ruby 關鍵字了，而且你在本節還會學到一些新的東西。我在這裡所作的是將所有的 Ruby 符號和關鍵字列出來，這些都是值得掌握的重點。

在這節課中，你需要複習每一個關鍵字，從記憶中想起它的作用並且寫下來，接著上網搜索它真正的功能。有些內容可能是無法搜索的，所以這對你可能有些難度，不過你還是需要堅持嘗試。

如果你發現記憶中的內容有誤，就在索引卡片上寫下正確的定義，試著將自己的記憶糾正過來。如果你就是不知道它的定義，就把它也直接寫下來，以後再做研究。

最後，將每一種符號和關鍵字用在程式裡，你可以用一個小程式來做，也可以盡量多寫一些程式來鞏固記憶。這裡的關鍵點是明白各個符號的作用，確認自己沒搞錯，如果搞錯了就糾正過來，然後將其用在程序裡，並且通過這樣的方式鞏固自己的記憶。

Keywords（關鍵字）

- `alias`
- `and`
- `BEGIN`
- `begin`
- `break`
- `case`
- `class`
- `def`
- `defined?`
- `do`
- `else`
- `elsif`
- `END`
- `end`
- `ensure`
- `false`
- `for`
- `if`
- `in`
- `module`
- `next`
- `nil`

- not
- or
- redo
- rescue
- retry
- return
- self
- super
- then
- true
- undef
- unless
- until
- when
- while
- yield

資料類型

針對每一種資料類型，都舉出一些例子來，例如針對 string，你可以舉出一些字。針對number，你可以舉出一些數字。

- **true**
- **false**
- **nil**
- constants
- strings
- numbers
- ranges
- arrays
- hashes

字串格式(String Formats)

一樣的，在字符串中使用它們，確認它們的功能。

- \\
- \'
- \"
- \a
- \b

- `\f`
- `\n`
- `\r`
- `\t`
- `\v`

Operators

有些操作符號你可能還不熟悉，不過還是一一看過去，研究一下它們的功能，如果你研究不出來也沒關係，記錄下來日後解決。

- `::`
- `[]`
- `**`
- `-(unary)`
- `+(unary)`
- `!`
- `~`
- `*`
- `/`
- `%`
- `+`
- `-`
- `<<`
- `>>`
- `&`
- `|`
- `>`
- `>=`
- `<`
- `<=`
- `<=>`
- `==`
- `===`
- `!=`
- `=~`
- `!~`
- `&&`
- `||`
- `..`

- ...

花一個星期學習這些東西，如果你能提前完成就更好了。我們的目的是覆蓋到所有的符號類型，確認你已經牢牢記住它們。另外很重要的一點是這樣你可以找出自己還不知道哪些東西，為自己日後學習找到一些方向。

习题 38: 阅读程式码

現在去找一些 Ruby 程式碼閱讀一下。你需要自己找程式碼，然後從中學習一些東西。你學到的東西已經足夠讓你看懂一些程式碼了，但你可能還無法理解這些程式碼的功能。這節課我要教給你的是：如何運用你學到的東西理解別人的程式碼。

首先把你想要理解的程式碼印到紙上。沒錯，你需要印出來，因為和螢幕輸出相比，你的眼睛和大腦更習慣於接受紙質列印的內容。一次最多列印幾頁就可以了。

然後通讀你列印出來的代碼並做好標記，標記的內容包括以下幾個方面：

1. 函數以及函數的功能。
2. 每個變數的初始賦值。
3. 每個在程式的各個部分中多次出現的變數。它們以後可能會給你帶來麻煩。
4. 任何不包含else的 `if` 語句。它們是正確的嗎？
5. 任何可能沒有結束點的while循環。
6. 最後一條，代碼中任何你看不懂的部分都記下來。

接下來你需要通過註解的方式向自己解釋程式碼的含義。解釋各個函式的使用方法，各個變數的用途，以及任何其它方面的內容，只要能幫助你理解程式碼即可。

最後，在程式碼中比較難的各個部分，逐行或者逐個函式跟踪變數值。你可以再打印一份出來，在空白處寫出你要「追踪」的每個變數的值。

一旦你基本理解了程式碼的功能，回到電腦面前，在程式碼上重讀一次，看看能不能找到新的問題點。然後繼續找新的程式碼，用上述的方法去閱讀理解，直到你不再需要紙質列印為止。

加分習題

1. 研究一下什麼是「流程圖(flow chart)」，並學著畫一下。
2. 如果你在讀程式碼的時候找出了錯誤，試著把它們改對，並把修改內容發給作者。
3. 不使用紙質打印時，你可以使用註解符號#在程序中加入筆記。有時這些筆記會對後來的讀程式碼的人有很大的幫助。

习题 39: 阵列的操作

你已經學過了陣列。在你學習“`while` 迴圈的時候，你對陣列進行過「pushed」動作，而且將陣列的內容印了出來。另外你應該還在加分習題裡研究過 Ruby 文件，看了陣列支援的其他操作。這已經是一段時間以前了，所以如果你不記得了的話，就回到本書的前面再複習一遍吧。

找到了嗎？還記得嗎？很好。那時候你對一個陣列執行了 `push` 函式。不過，你也許還沒有真正明白發生的事情，所以我們再來看看我們可以對陣列進行什麼樣的操作。

當你看到像 `mystuff.append('hello')` 這樣的程式時，你事實上已經在 Ruby 內部激發了一個連鎖反應。以下是它的運作原理：

1. Ruby 看到你用到了 `mystuff`，於是就去找到這個變數。也許它需要倒著檢查看你有沒有在哪裡用 `=` 建立過這個變數，或者檢查它是不是一個函式參數，或者看它是不是一個全局變數。不管哪種方式，它得先找到 `mystuff` 這個變數才行。
2. 一旦它找到了 `mystuff`，就輪到處理句點 `.` (period)這個操作符號，而且開始查看 `mystuff` 內部的一些變數了。由於 `mystuff` 是一個陣列，Ruby 知道 `mystuff` 支援一些函式。
3. 接下來輪到了處理 `push`。Ruby會將 「push」和 `mystuff` 支援的所有函式的名稱一一對比，如果確實其中有一個叫 `push` 的函式，那麼Ruby就會去使用這個函式。
4. 接下來Ruby看到了括號(parenthesis)並且意識到, 「噢，原來這應該是一個函式」，到了這裡，它就正常會呼叫這個函式了，不過這裡的函式還要多一個參數才行。

一下子要消化這麼多可能有點難度，不過我們將做幾個練習，讓你頭腦中有一個深刻的印象。下面的練習將字符串和列表混在一起，看看你能不能在裡邊找出點樂子來：

```
ten_things = "Apples Oranges Crows Telephone Light Sugar"

puts "Wait there's not 10 things in that list, let's fix that."

stuff = ten_things.split(' ')
more_stuff = %w(Day Night Song Frisbee Corn Banana Girl Boy)

while stuff.length != 10
  next_one = more_stuff.pop()
  puts "Adding: #{next_one}"
  stuff.push(next_one)
  puts "There's #{stuff.length} items now."
end

puts "There we go: #{stuff}"

puts "Let's do some things with stuff."

puts stuff[1]
puts stuff[-1] # whoa! fancy
puts stuff.pop()
puts stuff.join(' ') # what? cool!
puts stuff.values_at(3,5).join('#') # super stellar!
```

你應該看到的結果

```
$ ruby ex39.rb
Wait there's not 10 things in that list, let's fix that.
Adding: Boy
There's 7 items now.
Adding: Girl
There's 8 items now.
Adding: Banana
There's 9 items now.
Adding: Corn
There's 10 items now.
There we go: ["Apples", "Oranges", "Crows", "Telephone", "Light", "Sugar", "Boy", "Girl", "Banana", "Co
rn"]
Let's do some things with stuff.
Oranges
Corn
Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
Telephone#Sugar
$
```

加分習題

1. 上網閱讀一些關於「物件導向程式(Object Oriented Programming)」的資料。暈了吧？嗯，我以前

也是。別擔心。你將從這本書學到足夠用的關於物件導向程式的基礎知識，而以後你還可以慢慢學到更多。

2. `something.methods` 和 something的 class 有什麼關係？
3. 如果你不知道我講的是些什麼東西，別擔心。程式設計師為了顯得自己聰明，於是就發明了Opject Oriented Programming，簡稱為OOP，然後他們就開始濫用這個東西了。如果你覺得這東西太難，你可以開始學一下「函式式程式(functional programming)」。

习题 40: Hash, 可爱的 Hash

接下來我要教你另外一種讓你傷腦筋的容器型資料結構，因為一旦你學會這種資料結構，你將擁有超酷的能力。這是最有用的容器：Hash。

Ruby 將這種資料類型叫做「Hash」，有的語言裡它的名稱是「dictionaries」。這兩種名字我都會用到，不過這並不重要，重要的是它們和陣列的區別。你看，針對陣列你可以做這樣的事情：

```
ruby-1.9.2-p180 :015 > things = ['a','b','c','d']
 => ["a", "b", "c", "d"]
ruby-1.9.2-p180 :016 > print things[1]
b => nil
ruby-1.9.2-p180 :017 > things[1] = 'z'
 => "z"
ruby-1.9.2-p180 :018 > print things[1]
z => nil
ruby-1.9.2-p180 :019 > print things
["a", "z", "c", "d"] => nil
ruby-1.9.2-p180 :020 >
```

你可以使用數字作為陣列的「索引」，也就是你可以通過數字找到陣列中的元素。而 Hash 所作的，是讓你可以通過任何東西找到元素，不只是數字。是的，Hash 可以將一個物件和另外一個東西關聯，不管它們的類型是什麼，我們來看看：

```
ruby-1.9.2-p180 :001 > stuff = {:name => "Rob", :age => 30, :height => 5*12+10}
 => {:name=>"Rob", :age=>30, :height=>70}
ruby-1.9.2-p180 :002 > puts stuff[:name]
Rob
 => nil
ruby-1.9.2-p180 :003 > puts stuff[:age]
30
 => nil
ruby-1.9.2-p180 :004 > puts stuff[:height]
70
 => nil
ruby-1.9.2-p180 :005 > stuff[:city] = "New York"
 => "New York"
ruby-1.9.2-p180 :006 > puts stuff[:city]
New York
 => nil
ruby-1.9.2-p180 :007 >
```

你將看到除了通過數字以外，我們在 Ruby 還可以用字串來從 Hash 中獲取 `stuff`，我們還可以用字串來往 Hash 中添加元素。當然它支持的不只有字串，我們還可以做這樣的事情：

```
ruby-1.9.2-p180 :004 > stuff[1] = "Wow"
 => "Wow"
ruby-1.9.2-p180 :005 > stuff[2] = "Neato"
 => "Neato"
ruby-1.9.2-p180 :006 > puts stuff[1]
Wow
 => nil
ruby-1.9.2-p180 :007 > puts stuff[2]
Neato
 => nil
ruby-1.9.2-p180 :008 > puts stuff
{:name=>"Rob", :age=>30, :height=>70, :city=>"New York", 1=>"Wow", 2=>"Neato"}
 => nil
ruby-1.9.2-p180 :009 >
```

在這裡我使用了數字。其實我可以使用任何東西，不過這麼說並不准確，不過你先這麼理解就行了。

當然了，一個只能放東西進去的 Hash是沒啥意思的，所以我們還要有刪除物件的方法，也就是使用 `delete` 這個關鍵字：

```
ruby-1.9.2-p180 :009 > stuff.delete(:city)
 => "New York"
ruby-1.9.2-p180 :010 > stuff.delete(1)
 => "Wow"
ruby-1.9.2-p180 :011 > stuff.delete(2)
 => "Neato"
ruby-1.9.2-p180 :012 > stuff
 => {:name=>"Rob", :age=>30, :height=>70}
ruby-1.9.2-p180 :013 >
```

接下來我們要做一個練習，你必須「非常」仔細，我要求你將這個練習寫下來，然後試著弄懂它做了些什麼。這個練習很有趣，做完以後你可能會有豁然開朗的感覺。

```
cities = {'CA' => 'San Francisco',
  'MI' => 'Detroit',
  'FL' => 'Jacksonville'}

cities['NY'] = 'New York'
cities['OR'] = 'Portland'

def find_city(map, state)
  if map.include? state
    return map[state]
  else
    return "Not found."
  end
end

# ok pay attention!
cities[:find] = method(:find_city)

while true
  print "State? (ENTER to quit) "
  state = gets.chomp

  break if state.empty?

  # this line is the most important ever! study!
  puts cities[:find].call(cities, state)
end
```

你應該看到的結果

```
$ ruby ex40.rb
State? (ENTER to quit) > CA
San Francisco
State? (ENTER to quit) > FL
Jacksonville
State? (ENTER to quit) > O
Not found.
State? (ENTER to quit) > OR
Portland
State? (ENTER to quit) > VT
Not found.
State? (ENTER to quit) >
```

加分習題

1. 在 Ruby 文件中找到 Hash 相關的內容，學著對 Hash 做更多的操作。
2. 找出一些 Hash 無法做到的事情。例如比較重要的一個就是 Hash 的內容是無序的，你可以檢查一下看看是否真是這樣。

3. 試著把 `for` 迴圈執行到 Hash 上面，然後試著在 `for` 迴圈中使用 Hash 的 each 函式，看看會有什麼樣的結果。

习题 41: 来自 Percal 25 号行星的哥顿人 (Gothons)

你在上一節中發現 Hash 的秘密功能了嗎？你可以解釋給自己嗎？讓我來給你解釋一下，順便和你自己的理解對比看有什麼不同。這裡是我們要討論的程式碼：

```
cities[:find] = method(:find_city)
puts cities[:find].call(cities, state)
```

你要記住一個函式也可以作為一個變數，為了要將一個程式碼區段儲存在一個變數裡，我們創造了一個東西叫「proc」，proc 是 procedure 縮寫。在這段程式碼中，首先我們呼叫了 Ruby 內建的函式 `method`，它會回傳一個 proc 版的 `find_city` 函式。然後我們將之除存在一個 Hash 裡：key 是 `:find`，value 是 `cities`。。這和我們將州和市關聯起來的程式碼做的事情一樣，只不過在這個情況裡是個 proc。

好了，所以一旦我們知道 `find_city` 是在Hash中 `:find` 的位置，這就意味著我們可以去呼叫它。第二行程式碼可以分解成如下步驟：

1. Ruby 讀到了 `cities`，然後知道了它是一個 「Hash」。
2. 然後看到了 `[:find]`，於是 Ruby 就從索引找到了 cities Hash 中對應的位置，並且獲取了該位置的內容。
3. `[:find]` 這個位置的內容是我們的函式 `find_city`，所以Ruby就知道了這裡表示一個函式，於是當它碰到 .call 就開始了 proc呼叫。
4. `cities`、`state` 這兩個參數將被傳遞到函式 `find_city` 中，然後這個函式就被運行了。
5. `find_city` 接著從 `cities` 中尋找 `states`，並且回傳它找到的內容，如果什麼都沒找到，就返回一個信息說它什麼都沒找到。
6. Ruby 接受 `find_city` 傳回的資訊，最後將該資訊賦值給一開始的 `city_found` 這個變數。

我再教你一個小技巧。如果你倒著閱讀的話，程式碼可能會變得更容易理解。讓我們來試一下，一樣是那行：

1. `state` 和 `city` 是...
2. 最為參數傳遞給...
3. 一個 proc 位於...
4. `:find` 然後尋找，目的地為...
5. `cities` 這個 Hash...
6. 最後印到螢幕上

還有一種方法讀它，這回是「由裡向外」。

1. 找到表示式的中心位置，此次為 `[:find]`。

2. 逆時針追溯，首先看到的是一個叫 `cities` 的 Hash，這樣就知道了 `cities` 中的 `:find` 元素。
3. 上一步得到一個函式。繼續逆時針尋找，看到的是參數。
4. 參數傳遞給函式後，函式會傳回一個值。然後再逆時針尋找。
5. 最後，我們到了 `city_found` =的賦值位置，並且得到了最終結果。

數十年的程式經驗下來，我在讀程式碼的過程中已經用不到上面的三種方法了。我只要瞄一眼就能知道它的意思。甚至給我一整頁的程式碼，我也可以一眼瞄出裡邊的 bug 和錯誤。這樣的技能是花了超乎常人的時間和精力才鍛煉得來的。在磨練的過程中，我學會了下面三種讀程式碼的方法：

1. 從前向後。
2. 從後向前。
3. 逆時針方向。

現在我們來寫這次的練習，寫完後再過一遍，這節習題其實挺有趣的。

程式碼不少，不過還是從頭寫完吧。確認它能運行，然後玩一下看看。

```
def prompt()
  print "> "
end

def death()
  quips = ["You died.  You kinda suck at this.",
    "Nice job, you died ...jackass.",
    "Such a luser.",
    "I have a small puppy that's better at this."]
  puts quips[rand(quips.length())]
  Process.exit(1)
end

def central_corridor()
  puts "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
  puts "your entire crew.  You are the last surviving member and your last"
  puts "mission is to get the neutron destruct bomb from the Weapons Armory,"
  puts "put it in the bridge, and blow the ship up after getting into an "
  puts "escape pod."
  puts "\n"
  puts "You're running down the central corridor to the Weapons Armory when"
  puts "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume"
  puts "flowing around his hate filled body.  He's blocking the door to the"
  puts "Armory and about to pull a weapon to blast you."

  prompt()
  action = gets.chomp()

  if action == "shoot!"
    puts "Quick on the draw you yank out your blaster and fire it at the Gothon."
    puts "His clown costume is flowing and moving around his body, which throws"
    puts "off your aim.  Your laser hits his costume but misses him entirely.  This"
    puts "completely ruins his brand new costume his mother bought him, which"
    puts "makes him fly into an insane rage and blast you repeatedly in the face until"
```

```
    puts "makes him fly into an insane rage and blast you repeatedly in the face until"
    puts "you are dead.  Then he eats you."
    return :death

  elsif action == "dodge!"
    puts "Like a world class boxer you dodge, weave, slip and slide right"
    puts "as the Gothon's blaster cranks a laser past your head."
    puts "In the middle of your artful dodge your foot slips and you"
    puts "bang your head on the metal wall and pass out."
    puts "You wake up shortly after only to die as the Gothon stomps on"
    puts "your head and eats you."
    return :death

  elsif action == "tell a joke"
    puts "Lucky for you they made you learn Gothon insults in the academy."
    puts "You tell the one Gothon joke you know:"
    puts "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
    puts "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
    puts "While he's laughing you run up and shoot him square in the head"
    puts "putting him down, then jump through the Weapon Armory door."
    return :laser_weapon_armory

  else
    puts "DOES NOT COMPUTE!"
    return :central_corridor
  end
end

def laser_weapon_armory()
  puts "You do a dive roll into the Weapon Armory, crouch and scan the room"
  puts "for more Gothons that might be hiding.  It's dead quiet, too quiet."
  puts "You stand up and run to the far side of the room and find the"
  puts "neutron bomb in its container.  There's a keypad lock on the box"
  puts "and you need the code to get the bomb out.  If you get the code"
  puts "wrong 10 times then the lock closes forever and you can't"
  puts "get the bomb.  The code is 3 digits."
  code = "%s%s%s" % [rand(9)+1, rand(9)+1, rand(9)+1]
  print "[keypad]> "
  guess = gets.chomp()
  guesses = 0

  while guess != code and guesses < 10
    puts "BZZZZEDDD!"
    guesses += 1
    print "[keypad]> "
    guess = gets.chomp()
  end

  if guess == code
    puts "The container clicks open and the seal breaks, letting gas out."
    puts "You grab the neutron bomb and run as fast as you can to the"
    puts "bridge where you must place it in the right spot."
    return :the_bridge
  else
    puts "The lock buzzes one last time and then you hear a sickening"
```

```
    puts "melting sound as the mechanism is fused together."
    puts "You decide to sit there, and finally the Gothons blow up the"
    puts "ship from their ship and you die."
    return :death
  end
end

def the_bridge()
  puts "You burst onto the Bridge with the netron destruct bomb"
  puts "under your arm and surprise 5 Gothons who are trying to"
  puts "take control of the ship.  Each of them has an even uglier"
  puts "clown costume than the last.  They haven't pulled their"
  puts "weapons out yet, as they see the active bomb under your"
  puts "arm and don't want to set it off."

  prompt()
  action = gets.chomp()

  if action == "throw the bomb"
    puts "In a panic you throw the bomb at the group of Gothons"
    puts "and make a leap for the door.  Right as you drop it a"
    puts "Gothon shoots you right in the back killing you."
    puts "As you die you see another Gothon frantically try to disarm"
    puts "the bomb. You die knowing they will probably blow up when"
    puts "it goes off."
    return :death

  elsif action == "slowly place the bomb"
    puts "You point your blaster at the bomb under your arm"
    puts "and the Gothons put their hands up and start to sweat."
    puts "You inch backward to the door, open it, and then carefully"
    puts "place the bomb on the floor, pointing your blaster at it."
    puts "You then jump back through the door, punch the close button"
    puts "and blast the lock so the Gothons can't get out."
    puts "Now that the bomb is placed you run to the escape pod to"
    puts "get off this tin can."
    return :escape_pod
  else
    puts "DOES NOT COMPUTE!"
    return :the_bridge
  end
end

def escape_pod()
  puts "You rush through the ship desperately trying to make it to"
  puts "the escape pod before the whole ship explodes.  It seems like"
  puts "hardly any Gothons are on the ship, so your run is clear of"
  puts "interference.  You get to the chamber with the escape pods, and"
  puts "now need to pick one to take.  Some of them could be damaged"
  puts "but you don't have time to look.  There's 5 pods, which one"
  puts "do you take?"

  good_pod = rand(5)+1
  print "[pod #]>"
  guess = gets.chomp()
```

```
  guess = gets.chomp()

  if guess.to_i != good_pod
    puts "You jump into pod %s and hit the eject button." % guess
    puts "The pod escapes out into the void of space, then"
    puts "implodes as the hull ruptures, crushing your body"
    puts "into jam jelly."
    return :death
  else
    puts "You jump into pod %s and hit the eject button." % guess
    puts "The pod easily slides out into space heading to"
    puts "the planet below.  As it flies to the planet, you look"
    puts "back and see your ship implode then explode like a"
    puts "bright star, taking out the Gothon ship at the same"
    puts "time.  You won!"
    Process.exit(0)
  end
end

ROOMS = {
  :death => method(:death),
  :central_corridor => method(:central_corridor),
  :laser_weapon_armory => method(:laser_weapon_armory),
  :the_bridge => method(:the_bridge),
  :escape_pod => method(:escape_pod)
}

def runner(map, start)
  next_one = start

  while true
    room = map[next_one]
    puts "\n--------"
    next_one = room.call()
  end
end

runner(ROOMS, :central_corridor)
```

你應該看到的結果

```
$ ruby ex41.rb

--------
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
```

```
Armory and about to pull a weapon to blast you.
> dodge!
Like a world class boxer you dodge, weave, slip and slide right
as the Gothon's blaster cranks a laser past your head.
In the middle of your artful dodge your foot slips and you
bang your head on the metal wall and pass out.
You wake up shortly after only to die as the Gothon stomps on
your head and eats you.

--------
Such a luser.

$ ruby ex41.rb

--------
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
> tell a joke
Lucky for you they made you learn Gothon insults in the academy.
You tell the one Gothon joke you know:
Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr.
The Gothon stops, tries not to laugh, then busts out laughing and can't move.
While he's laughing you run up and shoot him square in the head
putting him down, then jump through the Weapon Armory door.

--------
You do a dive roll into the Weapon Armory, crouch and scan the room
for more Gothons that might be hiding.  It's dead quiet, too quiet.
You stand up and run to the far side of the room and find the
neutron bomb in its container.  There's a keypad lock on the box
and you need the code to get the bomb out.  If you get the code
wrong 10 times then the lock closes forever and you can't
get the bomb.  The code is 3 digits.
[keypad]> 123
BZZZZEDDD!
[keypad]> 234
BZZZZEDDD!
[keypad]> 345
BZZZZEDDD!
[keypad]> 456
BZZZZEDDD!
[keypad]> 567
BZZZZEDDD!
[keypad]> 678
BZZZZEDDD!
[keypad]> 789
```

```
BZZZZEDDD!
[keypad]> 384
BZZZZEDDD!
[keypad]> 764
BZZZZEDDD!
[keypad]> 354
BZZZZEDDD!
[keypad]> 263
The lock buzzes one last time and then you hear a sickening
melting sound as the mechanism is fused together.
You decide to sit there, and finally the Gothons blow up the
ship from their ship and you die.

--------
You died.  You kinda suck at this.
```

加分習題

1. 解釋一下返回至下一個房間的運作原理。 2.建立更多的房間，讓遊戲規模變大。
2. 除了讓每個函式印出自己以外，試試學習一下「文件註解(doc comments)」。
3. 看看你能不能將房間描述寫成文件註解，然後修改運行它的程式碼，讓它把文檔註解打印出來。
4. 一旦你用了文件註解作為房間描述，你還需要讓這個函式打出用戶提示嗎？試著讓運行函數的代碼打出用戶提示來，然後將用戶輸入傳遞到各個函式。你的函式應該只是一些 `if` 語句組合，將結果印出來，並且返回下一個房間。
5. 這其實是一個小版本的「有限狀態機(finite state machine)」，找資料閱讀了解一下，雖然你可能看不懂，但還是找來看看吧

习题 42: 物以类聚

雖說將函式放到 Hash 裡是很有趣的一件事情，你應該也會想到「如果 Ruby 內建這件事情該多好」。事實上也的確有，那就是 `class` 這個關鍵字。你可以使用 `class` 創建更棒的 「函式 Hash」，比你在上節練習中做的強大多了。Class（類）有著各種各樣強大的功能和用法，但本書不會深入講這些內容，在這裡，你只要你學會把它們當作高級的「函式字典」使用就可以了。

用到「class」的程式語言被稱作「Object Oriented Programming（面向對象編程式語言」。這是一種傳統的寫程式的方式，你需要做出「東西」來，然後你「告訴」這些東西去完成它們的工作。類似的事情你其實已經做過不少了，只不過還沒有意識到而已。記得你做過的這個吧：

```
stuff = ['Test', 'This', 'Out']
puts stuff.join(' ')
```

其實你這裡已經使用了 `class`。 stuff 這個變數其實是一個 Array Class。而 `stuff.join()` 呼叫了 `Array` 函式中的 `join`，然後傳遞了字串 `' '`（就是一個空格），這也是一個 class —— 它是一個 `String` class (字符串類)。到處都是 class！

其實你這裡已經使用了 `class`。 stuff 這個變量其實是一個list `class`（列表類）。而’ ‘.join(stuff)裡調用函式join的字符串’ ‘（就是一個空格）也是一個 `class` ——它是一個string `class` (字符串類)。到處都是 `class` ！

還有一個概念是 object（物件），不過我們暫且不提。當你建立過幾個 `class` 後就會學到了。怎樣建立 `class` 呢？和你建立 `ROOMS` Hash 的方法差不多，但其實更簡單：

```
class TheThing
  attr_reader :number

  def initialize()
    @number = 0
  end

  def some_function()
    puts "I got called."
  end

  def add_me_up(more)
    @number += more
    return @number
  end
end

# two different things
a = TheThing.new
b = TheThing.new

a.some_function()
b.some_function()

puts a.add_me_up(20)
puts a.add_me_up(20)
puts b.add_me_up(30)
puts b.add_me_up(30)

puts a.number
puts b.number
```

看到了在 `@number` 前面的 `@` 吧？這是一個實例變數 (instance variable)。每個在 `TheThing` 中你建立的實例都會擁有 `@number` 中自己的值。我們不能透過直接打 `a.number` 直接拿到 number。除非我們特別使用 `attr_reader :number` ，宣告讓外界能存取資料。

若要讓 `@number` write-only，我們可以使用 `attr_writer :number` 。為了讓它可以既可讀又可寫，我們可以使用 `attr_accessor :number` 。Ruby 使用了這些優良的物件導向原則來封裝資料。

下來，看到 `initialize` 函式了嗎？這就是你為建立 `class` 設置內部變數的方式。你可以用以 `@` 符號開頭的方式去設定它們。另外看到我們使用了 `add_me_up()` 為你建立 `number` 加值。後面你可以看到我們怎樣可以使用這種方法為數字加值，然後印出來。

Class 是很強大的東西，你應該好好讀讀相關的東西。盡可能多找些東西讀並且多多實驗。你其實知道它們該怎麼用，只要試試就知道了。其實我馬上就要去練吉他了，所以我不會讓你寫練習了。你將使用 `class` 寫一個練習。

接下來我們將把習題 41 的內容重寫一遍，不過這回我們將使用 `class` ：

```
class Game

  def initialize(start)
    @quips = [
      "You died.  You kinda suck at this.",
      "Nice job, you died ...jackass.",
      "Such a luser.",
      "I have a small puppy that's better at this."
    ]
    @start = start
    puts "in init @start = " + @start.inspect
  end

  def prompt()
    print "> "
  end

  def play()
    puts "@start => " + @start.inspect
    next_room = @start

    while true
      puts "\n--------"
      room = method(next_room)
      next_room = room.call()
    end
  end

  def death()
    puts @quips[rand(@quips.length())]
    Process.exit(1)
  end

  def central_corridor()
    puts "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
    puts "your entire crew.  You are the last surviving member and your last"
    puts "mission is to get the neutron destruct bomb from the Weapons Armory,"
    puts "put it in the bridge, and blow the ship up after getting into an "
    puts "escape pod."
    puts "\n"
    puts "You're running down the central corridor to the Weapons Armory when"
    puts "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume"
    puts "flowing around his hate filled body.  He's blocking the door to the"
    puts "Armory and about to pull a weapon to blast you."

    prompt()
    action = gets.chomp()

    if action == "shoot!"
      puts "Quick on the draw you yank out your blaster and fire it at the Gothon."
      puts "His clown costume is flowing and moving around his body, which throws"
      puts "off your aim.  Your laser hits his costume but misses him entirely.  This"
      puts "completely ruins his brand new costume his mother bought him, which"
      puts "makes him fly into an insane rage and blast you repeatedly in the face until"
      puts "you are dead.  Then he eats you."
```

```
    return :death

  elsif action == "dodge!"
    puts "Like a world class boxer you dodge, weave, slip and slide right"
    puts "as the Gothon's blaster cranks a laser past your head."
    puts "In the middle of your artful dodge your foot slips and you"
    puts "bang your head on the metal wall and pass out."
    puts "You wake up shortly after only to die as the Gothon stomps on"
    puts "your head and eats you."
    return :death

  elsif action == "tell a joke"
    puts "Lucky for you they made you learn Gothon insults in the academy."
    puts "You tell the one Gothon joke you know:"
    puts "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
    puts "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
    puts "While he's laughing you run up and shoot him square in the head"
    puts "putting him down, then jump through the Weapon Armory door."
    return :laser_weapon_armory

  else
    puts "DOES NOT COMPUTE!"
    return :central_corridor
  end
end

def laser_weapon_armory()
  puts "You do a dive roll into the Weapon Armory, crouch and scan the room"
  puts "for more Gothons that might be hiding.  It's dead quiet, too quiet."
  puts "You stand up and run to the far side of the room and find the"
  puts "neutron bomb in its container.  There's a keypad lock on the box"
  puts "and you need the code to get the bomb out.  If you get the code"
  puts "wrong 10 times then the lock closes forever and you can't"
  puts "get the bomb.  The code is 3 digits."
  code = "%s%s%s" % [rand(9)+1, rand(9)+1, rand(9)+1]
  print "[keypad]> "
  guess = gets.chomp()
  guesses = 0

  while guess != code and guesses < 10
    puts "BZZZZEDDD!"
    guesses += 1
    print "[keypad]> "
    guess = gets.chomp()
  end

  if guess == code
    puts "The container clicks open and the seal breaks, letting gas out."
    puts "You grab the neutron bomb and run as fast as you can to the"
    puts "bridge where you must place it in the right spot."
    return :the_bridge
  else
    puts "The lock buzzes one last time and then you hear a sickening"
    puts "melting sound as the mechanism is fused together."
```

```
    puts "You decide to sit there, and finally the Gothons blow up the"
    puts "ship from their ship and you die."
    return :death
  end
end

def the_bridge()
  puts "You burst onto the Bridge with the netron destruct bomb"
  puts "under your arm and surprise 5 Gothons who are trying to"
  puts "take control of the ship.  Each of them has an even uglier"
  puts "clown costume than the last.  They haven't pulled their"
  puts "weapons out yet, as they see the active bomb under your"
  puts "arm and don't want to set it off."

  prompt()
  action = gets.chomp()

  if action == "throw the bomb"
    puts "In a panic you throw the bomb at the group of Gothons"
    puts "and make a leap for the door.  Right as you drop it a"
    puts "Gothon shoots you right in the back killing you."
    puts "As you die you see another Gothon frantically try to disarm"
    puts "the bomb. You die knowing they will probably blow up when"
    puts "it goes off."
    return :death

  elsif action == "slowly place the bomb"
    puts "You point your blaster at the bomb under your arm"
    puts "and the Gothons put their hands up and start to sweat."
    puts "You inch backward to the door, open it, and then carefully"
    puts "place the bomb on the floor, pointing your blaster at it."
    puts "You then jump back through the door, punch the close button"
    puts "and blast the lock so the Gothons can't get out."
    puts "Now that the bomb is placed you run to the escape pod to"
    puts "get off this tin can."
    return :escape_pod
  else
    puts "DOES NOT COMPUTE!"
    return :the_bridge
  end
end

def escape_pod()
  puts "You rush through the ship desperately trying to make it to"
  puts "the escape pod before the whole ship explodes.  It seems like"
  puts "hardly any Gothons are on the ship, so your run is clear of"
  puts "interference.  You get to the chamber with the escape pods, and"
  puts "now need to pick one to take.  Some of them could be damaged"
  puts "but you don't have time to look.  There's 5 pods, which one"
  puts "do you take?"

  good_pod = rand(5)+1
  print "[pod #]>"
  guess = gets.chomp()
```

```
    if guess.to_i != good_pod
      puts "You jump into pod %s and hit the eject button." % guess
      puts "The pod escapes out into the void of space, then"
      puts "implodes as the hull ruptures, crushing your body"
      puts "into jam jelly."
      return :death
    else
      puts "You jump into pod %s and hit the eject button." % guess
      puts "The pod easily slides out into space heading to"
      puts "the planet below.  As it flies to the planet, you look"
      puts "back and see your ship implode then explode like a"
      puts "bright star, taking out the Gothon ship at the same"
      puts "time.  You won!"
      Process.exit(0)
    end
  end
end

a_game = Game.new(:central_corridor)
a_game.play()
```

你應該看到的結果

這個版本的遊戲和你的上一版效果應該是一樣的，其實有些代碼都幾乎一樣。比較一下兩版程式碼，弄懂其中不同的地方，重點在需要理解這些東西：

1. 怎樣建立一個 `class` Game 並且放函式到裡面去。
2. `initialize` 是一個特殊的初始方法，怎樣預設重要的變數在裡面。
3. 你如何透過將在 `class` 下這個關鍵字再巢狀排列這些定義的方式為 `class` 添加函式。
4. 你如何透過在名稱底下加進巢狀內容來添加函式的。
5. `@` 的概念，還有它在 `initialize`、`play` 和 `death` 是怎樣被使用的。
6. 最後我們怎樣建立了一個 Game，然後透過 `play()` 讓所有的東西運行起來。

加分習題 研究一下**dict**是什麼東西，應該怎樣使用。 再為遊戲添加一些房間，確認自己已經學會使用 `class` 。 創建一個新版本，裡邊使用兩個 `class` ，其中一個是Map，另一個是Engine。提示:把play放到Engine裡面。

加分習題

1. 再為遊戲添加一些房間，確認自己已經學會使用 `class` 。
2. 建一個新版本，裡邊使用兩個 `class` ，其中一個是 `Map` ，另一個是 `Engine` 。提示:把 play 放到 `Engine` 裡面。

习题 43: 你来制作一个游戏

你要開始學會自食其力了。通過閱讀這本書你應該已經學到了一點，那就是你需要的所有的資訊網路上都有，你只要去搜尋就能找到。唯一困擾你的就是如何使用正確的詞彙進行搜尋。學到現在，你在挑選搜尋關鍵字方面應該已經有些感覺了。現在已經是時候了，你需要嘗試寫一個大的專案，並讓它運行起來。

以下是你的需求：

1. 製作一個截然不同的遊戲。
2. 使用多個檔案，並使用 `require` 呼叫這些檔案。確認自己知道 `require` 的用法。
3. 對於每個房間使用一個 `class` ， `class` 的命名要能體現出它的用處。例如 `GoldRoom` 、 `KoiPondRoom` 。
4. 你的執行器程式碼應該了解這些房間，所以創建一個 class 來呼叫並且記錄這些房間。有很多種方法可以達到這個目的，不過你可以考慮讓每個房間傳回下一個房間，或者設置一個變數，讓它指定下一個房間是什麼。

其他的事情就全靠你了。花一個星期完成這件任務，做一個你能做出來的最好的遊戲。使用你學過的任何東西（類，函數，Hash，陣列……）來改進你的程式。這節課的目的是教你如何構建 class 出來，而這些 class 又能調用到其它 Ruby 檔案中的 class。

我不會詳細地告訴你告訴你怎樣做，你需要自己完成。試著下手吧，寫程式就是解決問題的過程，這就意味著你要嘗試各種可能性，進行實驗，經歷失敗，然後丟掉你做出來的東西重頭開始。當你被某個問題卡住的時候，你可以向別人尋求幫助，並把你的程式貼出來給他們看。如果有人刻薄你，別理他們，你只要集中精力在幫你的人身上就可以了。持續修改和清理你的程式碼，直到它完整可執行為止，然後再研究一下看它還能不能被改進。

祝你好運，下個星期你做出遊戲後我們再見。

习题 44: 评估你的游戏

這節練習的目的是檢查評估你的遊戲。也許你只完成了一半，卡在那裡沒有進行下去，也許你勉強做出來了。不管怎樣，我們將串一下你應該弄懂的一些東西，並確認你的遊戲裡有使用到它們。我們將學習如何用正確的格式構建class，使用class 的一些通用習慣，另外還有很多的「書本知識」讓你學習。

為什麼我會讓你先行嘗試，然後才告訴你正確的做法呢？因為從現在開始你要學會「自給自足」，以前是我牽著你前行，以後就得靠你自己了。後面的習題我只會告訴你你的任務，你需要自己去完成，在你完成後我再告訴你如何可以改進你的作業。

一開始你會覺得很困難並且很不習慣，但只要堅持下去，你就會培養出自己解決問題的能力。你還會找出創新的方法解決問題，這比從課本中拷貝解決方案強多了。

函式的風格

以前我教過的怎樣寫好函式的方法一樣是適用的，不過這裡要添加幾條：

- 由於各種各樣的原因，程序員將 class (類)裡邊的函式稱作method（方法）。很大程度上這只是個市場策略（用來推銷OOP），不過如果你把它們稱作「函式」的話，是會有囉嗦的人跳出來糾正你的。如果你覺得他們太煩了，你可以告訴他們從數學方面示範一下「函式」和「方法」究竟有什麼不同，這樣他們會很快閉嘴的。
- 在你使用class的過程中，很大一部分時間是告訴你的 class如何「做事情」。給這些函式命名的時候，與其命名成一個名詞，不如命名為一個動詞，作為給class的一個命令。就和陣列中 的 `pop` 函式一樣，它相當於說：「嘿，陣列，把這東西給我 `pop` 出去。」它的名字不是 `remove_from_end_of_list`，因為即使它的功能的確是這樣，這一個字串也不是一個命令。
- 讓你的函式保持簡單小巧。由於某些原因，有些人開始學習 class 後就會忘了這一條。

Classh (類) 的風格

- 你的 class 應該使用「camel case（駝峰式大小寫）」，例如你應該使用 `SuperGoldFactory` 而不是 `super_gold_factory`
- 你的 `initialize` 不應該做太多的事情，這會讓 class 變得難以使用。
- 你的其它函式應該使用「underscore format（下劃線隔詞）」，所以你可以寫 `my_awesome_hair`，而不是 `myawesomehair` 或者 `MyAwesomeHair`。
- 用一致的方式組織函式的參數。如果你的 class 需要處理 users、dogs、和cats，就保持這個次序（特別情況除外）。如果一個函式的參數是(dog, cat, user)，另一個的是(user, cat, dog)，這會讓函式使用起來很困難。
- 不要對全局變數或者來自模組的變數進行重定義或者賦值，讓這些東西自顧自就行了。
- 不要一根筋式地維持風格一致性，這是思維力底下的妖怪嘍囉做的事情。一致性是好事情，不過愚蠢地跟著別人遵從一些白痴口號是錯誤的行為——這本身就是一種壞的風格。好好為自己著想吧。

程式碼風格

- 為了以方便他人閱讀，在自己的程式碼之間留下一些空白。你將會看到一些很差的程式設計師，他們寫的程式碼還算通順，但程式碼之間沒有任何空間。這種風格在任何程式語言中都是壞習慣，人的眼睛和大腦會通過空白和垂直對齊的位置來掃描和區隔視覺元素，如果你的程式碼裡沒有任何空白，這相當於為你的程式碼上了迷彩裝。
- 如果一段程式碼你無法朗讀出來，那麼這段程式碼的可讀性可能就有問題。如你找不到讓某個東西易用的方法，試著也朗讀出來。這樣不僅會逼迫你慢速而且真正仔細閱讀過去，還會幫你找到難讀的段落，從而知道那些程式碼的易讀性需要作出改進。
- 學著模仿別人的風格寫程式，直到哪天你找到你自己的風格為止。
- 一旦你有了自己的風格，也別把它太當回事。程式設計師工作的一部分就是和別人的程式碼打交道，有的人審美觀就是很差。相信我，你的審美觀某一方面一定也很差，只是你從未意識到而已。
- 如果你發現有人寫的程式碼風格你很喜歡，那就模仿他們的風格。

好的註釋

- 有程序員會告訴你，說你的程式碼需要有足夠的可讀性，這樣你就無需寫註釋了。他們會以自己接近官腔的聲音說「所以你永遠都不應該寫程式碼註釋。」這些人要嘛是一些顧問型的人物，如果別人無法使用他們的程式碼，就會付更多錢給他們讓他們解決問題。要嘛他們能力不足，從來沒有跟別人合作過。別理會這些人，好好寫你的註解。
- 寫註解的時候，描述清楚為什麼你要這樣做。程式碼只會告訴你「這樣實現」，而不會告訴你「為什麼要這樣實現」，而後者比前者更重要。
- 當你為函式寫文件註解的時候，記得為別的程式碼使用者也寫些東西。你不需要狂寫一大堆，但一兩句話謝謝這個函式的用法還是很有用的。
- 最後要說的是，雖然註解是好東西，太多的註解就不見得是了。而且註解也是需要維護的，你要盡量讓註解短小精悍一語中的，如果你對程式碼做了更改，記得檢查並更新相關的註解，確認它們還是正確的。

評估你的遊戲

現在我要求你假裝成是我，板起臉來，把你的程式碼打印出來，然後拿一支紅筆，把程式碼中所有的錯誤都標出來。你要充分利用你在本章以及前面學到的知識。等你批改完了，我要求你把所有的錯誤改對。這個過程我需要你多重複幾次，爭取找到更多的可以改進的地方。使用我前面教過的方法，把程式碼分解成最細小的單元一一進行分析。

這節練習的目的是訓練你對於細節的關注程度。等你檢查完自己的程式碼，再找一段別人的程式碼用這種方法檢查一遍。把程式碼打印出來，檢查出所有程式碼和風格方面的錯誤，然後試著在不改壞別人程式碼的前提下把它們修改正確。

這週我要求你的事情就是批改和糾錯，包含你自己的程式碼和別人的程式碼，再沒有別的了。這節習題難

度還是挺大，不過一旦你完成了任務，你學過的東西就會牢牢記在腦中。

习题 45: 物件、类和从属关系

有一個重要的概念你需要弄明白，那就是 Class「類」和 Object「物件」的區別。問題在於，class 和 object 並沒有真正的不同。它們其實是同樣的東西，只是在不同的時間名字不同罷了。我用禪語來解釋一下吧：

魚(Fish)和鮭魚(Salmon)有什麼區別？

這個問題有沒有讓你有點暈呢？說真的，坐下來想一分鐘。我的意思是說，魚和鮭魚是不一樣，不過它們其實也是一樣的是不是？泥鰍是魚的一種，所以說沒什麼不同，不過泥鰍又有些特別，它和別的種類的魚的確不一樣，比如鮭魚和比目魚就不一樣。所以鮭魚和魚既相同又不同。怪了。

這個問題讓人暈的原因是大部分人不會這樣去思考問題，其實每個人都懂這一點，你無須去思考魚和鮭魚的區別，因為你知道它們之間的關係。你知道鮭魚是魚的一種，而且魚還有別的種類，根本就沒必要去思考這類問題。

讓我們更進一步，假設你有一隻水桶，裡邊有三條鮭魚。假設你的好人卡多到沒地方用，於是你給它們分別取名叫Frank，Joe，Mary。現在想想這個問題：

Mary 和鮭魚有什麼區別？

這個問題一樣的奇怪，但比起魚和鮭魚的問題來還好點。你知道 Mary是一條鮭魚，所以他並沒什麼不同，他只是鮭魚的一個「實例(instance)」。Joe 和Frank 一樣也是鮭魚的實例。我的意思是說，它們是由鮭魚創建出來的，而且代表著和鮭魚一樣的屬性。

所以我們的思維方式是（你可能會有點不習慣）：魚是一個「類(class)」，鮭魚是一個「類(class)」，而 Mary 是一個「物件(object)」。仔細想想，然後我再一點一點慢慢解釋給你。

魚是一個「類」，表示它不是一個真正的東西，而是一個用來描述具有同類屬性的實例的概括性詞彙。你有鰭？你有鰾？你住在水裡？好吧那你就是一條魚。

後來一個博士路過，看到你的水桶，於是告訴你：「小伙子，你這些魚是鮭魚。」 專家一出，真相即現。並且專家還定義了一個新的叫做「鮭魚」的「類」，而這個「類」又有它特定的屬性。長鼻子？紅肉？體型大？住在海裡或是乾淨新鮮的水裡？吃起來味道不錯？那你就是一條鮭魚。

最後一個廚師過來了，他跟博士說：「非也非也，你看到的是鮭魚，我看到的是Mary，而且我要把 Mary 淋上美味醬料做一道小菜。」於是你就有了一隻叫做Mary 的鮭魚的「實例(instance)」（鮭魚也是魚的一個「實例」），並且你使用了它（把它塞到你的胃裡了），這樣它就是一個「物件(object)」。

這會你應該了解了：Mary 是鮭魚的成員，而鮭魚又是魚的成員。這裡的關係式：物件屬於某個類，而某個類又屬於另一個類。

寫成程式碼是什麼樣子

這個概念有點詭異，不過實話說，你只要在建立和使用class的時候操心一下就可以了。我來給你兩個區分 `Class` 和 `Object` 的小技巧。

首先針對類和物件，你需要學會兩個說法，「is-a(是啥)」和「has-a(有啥)」。「是啥」要用在談論「兩者以類的關係互相關聯」的時候，而「有啥」要用在「兩者無共同點，僅是互為參照」的時候。

接下來，通讀這段程式碼，將每一個註解為 `##??` 的位置標明他是「is-a」還是「has-a」的關係，並講明白這個關係是什麼。在程式碼的開始我還舉了幾個例子，所以你只要寫剩下的就可以了。

記住，「是啥」指的是魚和鮭魚的關係，而「有啥」指的是鮭魚和烤肉架的關係。

```
## Animal is-a object (yes, sort of confusing) look at the extra credit
class Animal

end

## ??
class Dog < Animal

  def initialize(name)
    ## ??
    @name = name
  end

end

## ??
class Cat < Animal

  def initialize(name)
    ## ??
    @name = name
  end

end

## ??
class Person

  attr_accessor :pet

  def initialize(name)
    ## ??
    @name = name

    ## Person has-a pet of some kind
    @pet = nil
  end

end
## ??
class Employee < Person
```

```
  def initialize(name, salary)
    ## ?? hmm what is this strange magic?
    super(name)
    ## ??
    @salary = salary
  end

end

## ??
class Fish

end

## ??
class Salmon < Fish

end

## ??
class Halibut < Fish

end

## rover is-a Dog
rover = Dog.new("Rover")

## ??
satan = Cat.new("Satan")

## ??
mary = Person.new("Mary")

## ??
mary.pet = satan

## ??
frank = Employee.new("Frank", 120000)

## ??
frank.pet = rover

## ??
flipper = Fish.new

## ??
crouse = Salmon.new

## ??
harry = Halibut.new
```

加分習題

1. 有沒有辦法把 `Class` 當作 `Object` 使用呢？
2. 在習題中為 animals、fish、還有people 添加一些函式，讓它們做一些事情。看看當函數在 Animal 這樣的「基類(base class)」裡和在 Dog 裡有什麼區別。
3. 找些別人的程式碼，理清裡邊的「是啥」和「有啥」的關係。
4. 使用 Array 和 Hash 建立一些新的一對應多的「has-many」的關係。
5. 你認為會有一種「has-many」的關係嗎？閱讀一下關於「多重繼承(multiple inheritance)」的資料，然後儘量避免這種用法。

习题 46: 一个专案骨架

這裡你將學會如何建立一個專案「骨架」目錄。這個骨架目錄具備讓專案跑起來的所有基本內容。它裡邊會包含你的專案檔案佈局、自動化測試程式碼，模組，以及安裝腳本。當你建立一個新專案的時候，只要把這個目錄複製過去，改改目錄的名字，再編輯裡面的檔案就行了。

骨架內容: Linux/OSX

首先使用下述命令創建你的骨架目錄：

```
$ mkdir -p projects
$ cd projects/
$ mkdir skeleton
$ cd skeleton
$ mkdir bin lib lib/NAME test
```

我使用了一個叫 projects 的目錄，用來存放我自己的各個專案。然後我在裡邊建立了一個叫做 skeleton 的檔案夾，這就是我們新專案的基礎目錄。其中叫做 NAME 的檔案夾是你的專案的主檔案夾，你可以將它任意取名。

接下來我們要配置一些初始檔案：

```
$ touch lib/NAME.rb
$ touch lib/NAME/version.rb
```

然後我們可以建立一個 `NAME.gemspec` 的檔案在我們的專案的根目錄，這個檔案在安裝專案的時候我們會用到它：

```
# -*- encoding: utf-8 -*-
$:.push File.expand_path("../lib", __FILE__)
require "NAME/version"

Gem::Specification.new do |s|
  s.name        = "NAME"
  s.version     = NAME::VERSION
  s.authors     = ["Rob Sobers"]
  s.email       = ["rsobers@gmail.com"]
  s.homepage    = ""
  s.summary     = %q{TODO: Write a gem summary}
  s.description = %q{TODO: Write a gem description}

  s.rubyforge_project = "NAME"

  s.files         = `git ls-files`.split("\n")
  s.test_files    = `git ls-files -- {test,spec,features}/*`.split("\n")
  s.executables   = `git ls-files -- bin/*`.split("\n").map{ |f| File.basename(f) }
  s.require_paths = ["lib"]
end
```

編輯這個檔案，把自己的聯絡方式寫進去，然後放到那裡就行了。

最後你需要一個簡單的測試專用(我們將會在下一節中提到 Test)的骨架檔案叫 `test/test_NAME.rb`：

```
require 'test/unit'

class MyUnitTests < Test::Unit::TestCase

  def setup
    puts "setup!"
  end

  def teardown
    puts "teardown!"
  end

  def test_basic
    puts "I RAN!"
  end

end
```

安裝 Gems

Gems 是 Ruby 的套件系統，所以你需要知道怎麼安裝它和使用它。不過問題就來了。我的本意是讓這本書越清晰越乾淨越好，不過安裝軟體的方法是在是太多了，如果我要一步一步寫下來，那10 頁都寫不完，而且告訴你吧，我本來就是個懶人。

所以我不會提供詳細的安裝步驟了，我只會告訴你需要安裝哪些東西，然後讓你自己搞定。這對你也有好處，因為你將打開一個全新的世界，裡邊充滿了其他人發佈的軟體。

接下來你需要安裝下面的軟體套件：

- git - http://git-scm.com/
- rake - http://rake.rubyforge.org/
- rvm - https://rvm.beginrescueend.com/
- rubygems - http://rubygems.org/pages/download
- bundler - http://gembundler.com/

不要只是手動下載並且安裝這些軟體套件，你應該看一下別人的建議，尤其看看針對你的操作系統別人是怎樣建議你安裝和使用的。同樣的軟體套件在不一樣的操作系統上面的安裝方式是不一樣的，不一樣版本的 Linux 和 OSX 會有不同，而 Windows 更是不同。

我要預先警告你，這個過程會是相當無趣。在業內我們將這種事情叫做「yak shaving(剃犛牛)」。它指的是在你做一件有意義的事情之前的一些準備工作，而這些準備工作又是及其無聊冗繁的。你要做一個很酷的 Ruby 專案，但是創建骨架目錄需要你安裝一些軟體到件，而安裝軟體套件之前你還要安裝package installer (軟件套件安裝工具)，而要安裝這個工具你還得先學會如何在你的操作系統下安裝軟體，真是煩不勝煩呀。

無論如何，還是克服困難吧。你就把它當做進入程式俱樂部的一個考驗。每個程式設計師都會經歷這條道路，在每一段「酷」的背後總會有一段「煩」的。

使用這個骨架

剃犛牛的事情已經做的差不多了，以後每次你要新建一個專案時，只要做下面的事情就可以了：

1. 拷貝這份骨架目錄，把名字改成你新專案的名字。
2. 再將 `NAME` 模組和 `NAME.rb` 更名為你需要的名字，它可以是你專案的名字，當然別的名字也行。
3. 編輯你的 `NAME.gemspec` 檔案，讓它包含你新專案的相關資訊。
4. 重命名 `test/test_NAME.rb` ，讓它的名字匹配到你模組的名字。
5. 開始寫程式吧。

小測驗

這節練習沒有加分習題，不過需要你做一個小測驗：

1. 找文件閱讀，學會使用你前面安裝了的軟體套件。
2. 閱讀關於 `NAME.gemspec` 的文件，看它裡邊可以做多少配置。
3. 建立一個專案，在 `NAME.rb` 裡寫一些程式碼。
4. 在 `bin` 目錄下放一個可以運行的腳本，找材料學習一下怎樣建立可以在系統下運行的 Ruby 腳本。
5. 確定你建立的 `bin` 教本，有在 `NAME.gemspec` 中被參照到，這這樣你安裝時就可以連它安裝進

去。

6. 使用你的 `NAME.gemspec` 和 `gem build`、`gem install` 來安裝你寫的程式和確定它能用。然後使用 `gem uninstall` 去移除它。
7. 弄懂如何使用 Bundler 來自動建立一個骨架目錄。

习题 47: 自动化测试

為了確認遊戲的功能是否正常，你需要一遍一遍地在你的遊戲中輸入命令。這個過程是很枯燥無味的。如果能寫一小段程式碼用來測試你的程式碼豈不是更好？然後只要你對程序做了任何修改，或者添加了什麼新東西，你只要「跑一下你的測試」，而這些測試能確認程序依然能正確運行。這些自動測試不會抓到所有的bug，但可以讓你無需重複輸入命令運行你的程式碼，從而為你節約很多時間。

從這一章開始，以後的練習將不會有「你應該看到的結果」這一節，取而代之的是一個「你應該測試的東西」一節。從現在開始，你需要為自己寫的所有程式碼寫自動化測試，而這將讓你成為一個更好的程序員。

我不會試圖解釋為什麼你需要寫自動化測試。我要告訴你的是，你想要成為一個程式設計師，而程序的作用是讓無聊冗繁的工作自動化，測試軟件毫無疑問是無聊冗繁的，所以你還是寫點程式碼讓它為你測試的更好。

這應該是你需要的所有的解釋了。因為你寫單元測試的原因是讓你的大腦更加強健。你讀了這本書，寫了很多程式碼讓它們實現一些事情。現在你將更進一步，寫出懂得你寫的其他程式碼的程式碼。這個寫程式碼測試你寫的其他程式碼的過程將強迫你清楚的理解你之前寫的程式碼。這會讓你更清晰地了解你寫的程式碼實現的功能及其原理，而且讓你對細節的注意更上一個台階。

撰寫 Test Case

我們將拿一段非常簡單的程式碼為例，寫一個簡單的測試，這個測試將建立在上節我們創建的項目骨架上面。

首先從你的專案骨架創建一個叫做 `ex47` 的專案。確認該改名稱的地方都有改過，尤其是 `tests/ex47_tests.rb` 這處不要寫錯。

接下來建立一個簡單的 `ex47/lib/game.rb` 檔案，裡邊放一些用來被測試的程式碼。我們現在放一個傻乎乎的小 class 進去，用來作為我們的測試對象：

```
class Room

  attr_accessor :name, :description, :paths

  def initialize(name, description)
    @name = name
    @description = description
    @paths = {}
  end

  def go(direction)
    @paths[direction]
  end

  def add_paths(paths)
    @paths.update(paths)
  end

end
```

一旦你有了這個檔案，修改你的 unit test 骨架變成這樣：

```
require 'test/unit'
require_relative '../lib/ex47'

class MyUnitTests < Test::Unit::TestCase

  def test_room()
    gold = Room.new("GoldRoom",
                    """This room has gold in it you can grab. There's a
                door to the north.""")
    assert_equal(gold.name, "GoldRoom")
    assert_equal(gold.paths, {})
  end

  def test_room_paths()
    center = Room.new("Center", "Test room in the center.")
    north = Room.new("North", "Test room in the north.")
    south = Room.new("South", "Test room in the south.")

    center.add_paths({:north => north, :south => south})
    assert_equal(center.go(:north), north)
    assert_equal(center.go(:south), south)
  end

  def test_map()
    start = Room.new("Start", "You can go west and down a hole.")
    west = Room.new("Trees", "There are trees here, you can go east.")
    down = Room.new("Dungeon", "It's dark down here, you can go up.")

    start.add_paths({:west => west, :down => down})
    west.add_paths({:east => start})
    down.add_paths({:up => start})

    assert_equal(start.go(:west), west)
    assert_equal(start.go(:west).go(:east), start)
    assert_equal(start.go(:down).go(:up), start)
  end

end
```

這個文件 require 了你在 `lib/ex47.rb` 裡建立的 `Room` 這個類，接下來我們要做的就是測試它。於是我們看到一系列的以 `test_` 開頭的測試函式，它們就是所謂的「Test Case」，每一個Test Case裡面都有一小段程式碼，它們會建立一個或者一些房間，然後去確認房間的功能和你期望的是否一樣。它測試了基本的房間功能，然後測試了路徑，最後測試了整個地圖。

這裡最重要的函數時 `assert_equal`，它保證了你設置的變數，以及你在 `Room` 裡設置的路徑和你的期望相符。如果你得到錯誤的結果的話，Ruby 的 `Test::Unit` 模組將會印出一個錯誤信息，這樣你就可以找到出錯的地方並且修正過來。

測試指南

在寫測試程式碼時，你可以照著下面這些不是很嚴格的指南來做：

1. 測試腳本要放到 `tests/` 目錄下，並且命名為 `test_NAME.rb`。這樣做還有一個好處就是防止測試程式碼和別的程式碼互相混掉。
2. 為你的每一個模組寫一個測試。
3. Test Cast 函式保持簡短，但如果看上去不怎麼整潔也沒關係，Test Cast一般都有點亂。
4. 就算Test Cast有些亂，也要試著讓他們保持整潔，把裡邊重複的程式碼刪掉。建立一些輔助函數來避免重複的程式碼。當你下次在改完程式碼需要改測試的時候，你會感謝我這一條建議的。重複的程式碼會讓修改測試變得很難操作。
5. 最後一條是別太把測試當做一回事。有時候，更好的方法是把程式碼和測試全部刪掉，然後重新設計程式碼。

你應該看到的結果

```
$ ruby test_ex47.rb
Loaded suite test_ex47
Started
...
Finished in 0.000353 seconds.

3 tests, 7 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 63537
```

That’s what you should see if everything is working right. Try causing an error to see what that looks like and then fix it.

加分習題

1. 仔細閱讀 `Test::Unit` 相關的文件，再去了解一下其他的替代方案。
2. 了解一下 `Rspec`，看看它是否可以幹得更出色。
3. 改進你遊戲裡的 Room，然後用它重建你的遊戲。這次重寫，你需要一邊寫程式碼，一般把單元測試寫出來。

习题 48: 更进阶的使用者输入

你的遊戲可能一路跑得很爽，不過你處理使用者輸入的方式肯定讓你不勝其煩了。每一個房間都需要一套自己的語句，而且只有使用者完全輸入正確後才能執行。你需要一個設備，它可以允許使用者以各種方式輸入語彙。例如下面的幾種表述都應該被支援才對：

- open door
- open the door
- go THROUGH the door
- punch bear
- Punch The Bear in the FACE

也就是說，如果使用者的輸入和常用英語很接近也應該是可以的，而你的遊戲要識別出它們的意思。為了達到這個目的，我們將寫一個模組專門做這件事情。這個模組裡邊會有若干個類，它們互相配合，接受使用者輸入，並且將使用者輸入轉換成你的遊戲可以識別的命令。

英語的簡單格式是這個樣子的：

- 單詞由空格隔開。
- 句子由單詞組成。
- 語法控制句子的含義。

所以最好的開始方式是先搞定如何得到使用者輸入的詞彙，並且判斷出它們是什麼。

我們的遊戲語彙

我在遊戲裡建立了下面這些語彙：

- 表示方向: north, south, east, west, down, up, left, right, back.
- 動詞: go, stop, kill, eat.
- 修飾詞: the, in, of, from, at, it
- 名詞: door, bear, princess, cabinet.
- 數字詞: 由 0-9 構成的數字。

說到名詞，我們會碰到一個小問題，那就是不一樣的房間會用到不一樣的一組名詞，不過讓我們先挑一小組出來寫程式，以後再做改進吧。

如何斷句

我們已經有了詞彙表，為了分析句子的意思，接下來我們需要找到一個斷句的方法。我們對於句子的定義是「空格隔開的單詞」，所以只要這樣就可以了：

```
stuff = gets.chomp()
words = stuff.split()
```

目前做到這樣就可以了，不過這招在相當一段時間內都不會有問題。

語彙結構

一旦我們知道瞭如何將句子轉化成詞彙列表，剩下的就是逐一檢查這些詞彙，看它們是什麼類型。為了達到這個目的，我們將用到一個非常便利的 Ruby 資料結構「struct」。「struct」其實就是一個可以把一串的 attrbutes 綁在一起的方式，使用 accessor 函式，但不需要寫一個複雜的 class。它的建立方式就像這樣：

```
Pair = Struct.new(:token, :word)
first_word = Pair.new("direction", "north")
second_word = Pair.new("verb", "go")
sentence = [first_word, second_word]
```

這建立了一對 (TOKEN, WORD) 可以讓你看到 word 和在裡面做事。

這只是一個例子，不過最後做出來的樣子也差不多。你接受使用者輸入，用split 將其分隔成單詞列表，然後分析這些單詞，識別它們的類型，最後重新組成一個句子。

掃描輸入資料

現在你要寫的是詞彙掃描器。這個掃描器會將使用者的輸入字符串當做參數，然後返回由多個(TOKEN, WORD) struct 組成的列表，這個列表實現類似句子的功能。如果一個單詞不在預定的詞彙表中，那它返回時 WORD 應該還在，但TOKEN 應該設置成一個專門的錯誤標記。這個錯誤標記將告訴使用者哪裡出錯了。

有趣的地方來了。我不會告訴你這些該怎樣做，但我會寫一個「單元測試(unit test)」，而你要把掃描器寫出來，並保證單元測試能夠正常通過。

Exceptions And Numbers

有一件小事情我會先幫幫你，那就是數字轉換。為了做到這一點，我們會作一點弊，使用「異常(exceptions)」來做。「異常」指的是你運行某個函數時得到的錯誤。你的函數在碰到錯誤時，就會「提出(raise)」一個「異常」，然後你就要去處理(handle)這個異常。假如你在 IRB 裡寫了這些東西：

```
ruby-1.9.2-p180 :001 > Integer("hell")
ArgumentError: invalid value for Integer(): "hell"
    from (irb):1:in `Integer'
    from (irb):1
    from /home/rob/.rvm/rubies/ruby-1.9.2-p180/bin/irb:16:in `<main>'
```

這個 ArgumentError 就是 Integer() 函式拋出的一個異常。因為你給 Integer() 的參數不是一個數字。 Integer() 函數其實也可以傳回一個值來告訴你它碰到了錯誤，不過由於它只能傳回整數值，所以很難做到這一點。它不能返回-1，因為這也是一個數字。 Integer() 沒有糾結在它「究竟應該返回什麼」上面，而是提出了一個叫做 TypeError 的異常，然後你只要處理這個異常就可以了。

處理異常的方法是使用 begin 和 rescue 這兩個關鍵字：

```
def convert_number(s)
  begin
    Integer(s)
  rescue ArgumentError
    nil
  end
end
```

你把要試著運行的程式碼放到「begin」的區段裡，再將出錯後要運行的程式碼放到「except」區段裡。在這裡，我們要試著呼叫 Integer() 去處理某個可能是數字的東西，如果中間出了錯，我們就「rescue」這個錯誤，然後返回 「nil」。

在你寫的掃描器裡面，你應該使用這個函數來測試某個東西是不是數字。做完這個檢查，你就可以聲明這個單詞是一個錯誤單詞了。

What You Should Test

這裡是你應該使用的測試檔案 test/test_lexicon.rb :

```
require 'test/unit'
require_relative "../lib/lexicon"

class LexiconTests < Test::Unit::TestCase

  Pair = Lexicon::Pair
  @@lexicon = Lexicon.new()

  def test_directions()
    assert_equal([Pair.new(:direction, 'north')], @@lexicon.scan("north"))
    result = @@lexicon.scan("north south east")
    assert_equal(result, [Pair.new(:direction, 'north'),
                Pair.new(:direction, 'south'),
                Pair.new(:direction, 'east')])
  end

  def test_verbs()
    assert_equal(@@lexicon.scan("go"), [Pair.new(:verb, 'go')])
    result = @@lexicon.scan("go kill eat")
    assert_equal(result, [Pair.new(:verb, 'go'),
                Pair.new(:verb, 'kill'),
                Pair.new(:verb, 'eat')])
  end
```

```
  def test_stops()
    assert_equal(@@lexicon.scan("the"), [Pair.new(:stop, 'the')])
    result = @@lexicon.scan("the in of")
    assert_equal(result, [Pair.new(:stop, 'the'),
                          Pair.new(:stop, 'in'),
                          Pair.new(:stop, 'of')])
  end

  def test_nouns()
    assert_equal(@@lexicon.scan("bear"), [Pair.new(:noun, 'bear')])
    result = @@lexicon.scan("bear princess")
    assert_equal(result, [Pair.new(:noun, 'bear'),
                          Pair.new(:noun, 'princess')])
  end

  def test_numbers()
    assert_equal(@@lexicon.scan("1234"), [Pair.new(:number, 1234)])
    result = @@lexicon.scan("3 91234")
    assert_equal(result, [Pair.new(:number, 3),
                          Pair.new(:number, 91234)])
  end

  def test_errors()
    assert_equal(@@lexicon.scan("ASDFADFASDF"), [Pair.new(:error, 'ASDFADFASDF')])
    result = @@lexicon.scan("bear IAS princess")
    assert_equal(result, [Pair.new(:noun, 'bear'),
                          Pair.new(:error, 'IAS'),
                          Pair.new(:noun, 'princess')])
  end

end
```

記住你要使用你的專案骨架來建立新專案項目，將這個 Test Case 寫下來（不許複製貼上！），然後編寫你的掃描器，直至所有的測試都能通過。注意細節並確認結果一切工作良好。

設計提示

集中一次實現一個測試，盡量保持簡單，只要把你的 `lexicon.rb` 詞彙表中所有的單詞放那裡就可以了。不要修改輸入的單詞表，不過你需要建立自己的新列表，裡邊包含你的語彙元組。另外，記得使用 `include?` 函式來檢查這些語彙陣列，以確認某個單詞是否在你的語彙表中。

加分習題

1. 改進單元測試，讓它覆蓋到更多的語彙。
2. 向語彙列表添加更多的語彙，並且更新單元測試程式碼。
3. 讓你的掃描器能夠識別任意大小寫的詞彙。更新你的單元測試以確認其功能。
4. 找出另外一種轉換為數字的方法。
5. 我的解決方案用了37 行程式碼，你的是更長還是更短呢？

习题 49: 创造句子

從我們這個小遊戲的詞彙掃描器中，我們應該可以得到類似下面的列表（你的看起來可能格式會不太一樣）：

```
ruby-1.9.2-p180 :003 > print Lexicon.scan("go north")
[#<struct Lexicon::Pair token=:verb, word="go">,
   #<struct Lexicon::Pair token=:direction, word="north">] => nil
ruby-1.9.2-p180 :004 > print Lexicon.scan("kill the princess")
[#<struct Lexicon::Pair token=:verb, word="kill">,
   #<struct Lexicon::Pair token=:stop, word="the">,
   #<struct Lexicon::Pair token=:noun, word="princess">] => nil
ruby-1.9.2-p180 :005 > print Lexicon.scan("eat the bear")
[#<struct Lexicon::Pair token=:verb, word="eat">,
   #<struct Lexicon::Pair token=:stop, word="the">,
   #<struct Lexicon::Pair token=:noun, word="bear">] => nil
ruby-1.9.2-p180 :006 > print Lexicon.scan("open the door and smack the bear in the nose")
[#<struct Lexicon::Pair token=:error, word="open">,
   #<struct Lexicon::Pair token=:stop, word="the">,
   #<struct Lexicon::Pair token=:noun, word="door">,
   #<struct Lexicon::Pair token=:error, word="and">,
   #<struct Lexicon::Pair token=:error, word="smack">,
   #<struct Lexicon::Pair token=:stop, word="the">,
   #<struct Lexicon::Pair token=:noun, word="bear">,
   #<struct Lexicon::Pair token=:stop, word="in">,
   #<struct Lexicon::Pair token=:stop, word="the">,
   #<struct Lexicon::Pair token=:error, word="nose">] => nil
ruby-1.9.2-p180 :007 >
```

現在讓我們把它轉化成遊戲可以使用的東西，也就是一個 Sentence 類。

如果你還記得學校學過的東西的話，一個句子是由這樣的結構組成的：

> 主語(Subject) + 謂語(動詞Verb) + 賓語(Object)

很顯然實際的句子可能會比這複雜，而你可能已經在英語的語法課上面被折騰得夠嗆了。我們的目的，是將上面的 struct 列表轉換為一個 Sentence 物件，而這個對象又包含主謂賓各個成員。

匹配(Match) And 窺視(Peek)

為了達到這個效果，你需要四樣工具：

1. 一個循環存取 struct 列表的方法，這挺簡單的。
2. 「匹配」我們的主謂賓設置中不同種類 struct 的方法。
3. 一個「窺視」潛在struct的方法，以便做決定時用到。
4. 「跳過(skip)」我們不在乎的內容的方法，例如形容詞、冠詞等沒有用處的詞彙。

5. 我們使用 peek 函式查看 struct 列表中的下一個成員，做匹配以後再對它做下一步動作。讓我們先看看這個 peek 函式：

```
def peek(word_list)
  begin
    word_list.first.token
  rescue
    nil
  end
end
```

很簡單。再看看 match 函式：

```
def match(word_list, expecting)
  begin
    word = word_list.shift
    if word.token == expecting
      word
    else
      nil
    end
  rescue
    nil
  end
end
```

還是很簡單，最後我們看看 skip 函式:

```
def skip(word_list, word_type)
  while peek(word_list) == word_type
    match(word_list, word_type)
  end
end
```

以你現在的水準，你應該可以看出它們的功能來。確認自己真的弄懂了它們。

句子的語法

有了工具，我們現在可以從 struct 列表來構建句子(Sentence)對象了。我們的處理流程如下：

1. 使用 `peek` 識別下一個單詞。
2. 如果這個單詞和我們的語法匹配，我們就調用一個函式來處理這部分語法。假設函式的名字叫 `parse_subject` 好了。
3. 如果語法不匹配，我們就 `raise` 一個錯誤，接下來你會學到這方面的內容。
4. 全部分析完以後，我們應該能得到一個 Sentence 物件，然後可以將其應用在我們的遊戲中。

演示這個過程最簡單的方法是把程式碼展示給你讓你閱讀，不過這節習題有個不一樣的要求，前面是我給

你測試程式碼，你照著寫出程式碼來，而這次是我給你的程序，而你要為它寫出測試程式碼來。

以下就是我寫的用來解析簡單句子的程式碼，它使用了 `ex48` 這個 Lexicon class。

```
class ParserError < Exception

end

class Sentence

  def initialize(subject, verb, object)
    # remember we take Pair.new(:noun, "princess") structs and convert them
    @subject = subject.word
    @verb = verb.word
    @object = object.word
  end

end

def peek(word_list)
  begin
    word_list.first.token
  rescue
    nil
  end
end

def match(word_list, expecting)
  begin
    word = word_list.shift
    if word.token == expecting
      word
    else
      nil
    end
  rescue
    nil
  end
end

def skip(word_list, token)
  while peek(word_list) == token
    match(word_list, token)
  end
end

def parse_verb(word_list)
  skip(word_list, :stop)

  if peek(word_list) == :verb
    return match(word_list, :verb)
  else
    raise ParserError.new("Expected a verb next.")
  end
```

```
  end

  def parse_object(word_list)
    skip(word_list, :stop)
    next_word = peek(word_list)

    if next_word == :noun
      return match(word_list, :noun)
    end
    if next_word == :direction
      return match(word_list, :direction)
    else
      raise ParserError.new("Expected a noun or direction next.")
    end
  end

  def parse_subject(word_list, subj)
    verb = parse_verb(word_list)
    obj = parse_object(word_list)

    return Sentence.new(subj, verb, obj)
  end

  def parse_sentence(word_list)
    skip(word_list, :stop)

    start = peek(word_list)

    if start == :noun
      subj = match(word_list, :noun)
      return parse_subject(word_list, subj)
    elsif start == :verb
      # assume the subject is the player then
      return parse_subject(word_list, Pair.new(:noun, "player"))
    else
      raise ParserError.new("Must start with subject, object, or verb not: #{start}")
    end
  end
```

關於異常(Exception)

你已經簡單學過關於異常的一些東西，但還沒學過怎樣拋出(raise)它們。這節的程式碼示範了如何 raise。首先在最前面，你要定義好 `ParserException` 這個類，而它又是 `Exception` 的一種。另外要注意我們是怎樣使用 `raise` 這個關鍵字來拋出異常的。

你的測試程式碼應該也要測試到這些異常，這個我也會示範給你如何實現。

你應該測試的東西

為《習題49》寫一個完整的測試方案，確認程式碼中所有的東西都能正常工作，其中異常的測試——輸入

一個錯誤的句子它會拋出一個異常來。

使用 `assert_raises` 這個函式來檢查異常，在 Test::Unit 的文件裡查看相關的內容，學著使用它寫針對「執行失敗」的測試，這也是測試很重要的一個方面。從文件中學會使用 `assert_raises`，以及一些別的函式。

寫完測試以後，你應該就明白了這段程式碼的運作原理，而且也學會了如何為別人的程式碼寫測試程式碼。相信我，這是一個非常有用的技能。

加分習題

1. 修改 `parse_` method，將它們放到一個類裡邊，而不僅僅是獨立的方法函式。這兩種設計你喜歡哪一種呢？
2. 提高parser 對於錯誤輸入的抵禦能力，這樣即使使用者輸入了你預定義語彙之外的詞語，你的程式碼也能正常運行下去。
3. 改進語法，讓它可以處理更多的東西，例如數字。
4. 想想在遊戲裡你的 Sentence 類可以對使用者輸入做哪些有趣的事情。

习题 50: 你的第一个网站

這節以及後面的習題中，你的任務是把前面建立的遊戲做成網頁版。這是本書的最後三個章節，這些內容對你來說難度會相當大，你要在上面花些時間才能做出來。在你開始這節練習以前，你必須已經成功地完成過了《習題46》的內容，正確安裝了 **RubyGems**，而且學會瞭如何安裝軟體套件以及如何建立專案骨架。如果你不記得這些內容，就回到《習題46》重新複習一遍。

安裝 Sinatra

在建立你的第一個網頁應用程式之前，你需要安裝一個「Web框架」，它的名字叫 **Sinatra**。所謂的「框架」通常是指「讓某件事情做起來更容易的軟體套件」。在網頁應用的世界裡，人們建立了各種各樣的「網頁框架」，用來解決他們在建立網站時碰到的問題，然後把這些解決方案用軟體套件的方式發佈出來，這樣你就可以利用它們引導建立你自己的專案了。

可選的框架類型有很多很多，不過在這裡我們將使用 Sinatra 框架。你可以先學會它，等到差不多的時候再去接觸其它的框架，不過 Sinatra 本身挺不錯的，所以就算你一直使用也沒關係。

使用 `gem` 安裝 Sinatra:

```
$ gem install sinatra
Fetching: tilt-1.3.2.gem (100%)
Fetching: sinatra-1.2.6.gem (100%)
Successfully installed tilt-1.3.2
Successfully installed sinatra-1.2.6
2 gems installed
Installing ri documentation for tilt-1.3.2...
Installing ri documentation for sinatra-1.2.6...
Installing RDoc documentation for tilt-1.3.2...
Installing RDoc documentation for sinatra-1.2.6...
```

寫一個簡單的「Hello World」專案

現在你將做一個非常簡單的「Hello World」專案出來，首先你要建立一個專案目錄：

```
$ cd projects
$ bundle gem gothonweb
```

你最終的目的是把《習題42》中的遊戲做成一個 web 應用，所以你的專案名稱叫做 `gothonweb`，不過在此之前，你需要建立一個最基本的 Sinatra應用，將下面的代碼放到 `lib/gothonweb.rb` 中：

```
require_relative "gothonweb/version"
require "sinatra"

module Gothonweb
  get '/' do
    greeting = "Hello, World!"
    return greeting
  end
end
```

然後使用下面的方法來運行這個 web 程式：

```
$ ruby lib/gothonweb.rb
== Sinatra/1.2.6 has taken the stage on 4567 for development with backup from WEBrick
[2011-07-18 11:27:07] INFO  WEBrick 1.3.1
[2011-07-18 11:27:07] INFO  ruby 1.9.2 (2011-02-18) [x86_64-linux]
[2011-07-18 11:27:07] INFO  WEBrick::HTTPServer#start: pid=6599 port=4567
```

最後，使用你的網頁瀏覽器，打開 URL `http://localhost:4567/`，你應該看到兩樣東西，首先是瀏覽器裡顯示了 `Hello, world!`，然後是你的命令行終端顯示了如下的輸出：

```
127.0.0.1 - - [18/Jul/2011 11:29:10] "GET / HTTP/1.1" 200 12 0.0015
localhost - - [18/Jul/2011:11:29:10 EDT] "GET / HTTP/1.1" 200 12
- -> /
127.0.0.1 - - [18/Jul/2011 11:29:10] "GET /favicon.ico HTTP/1.1" 404 447 0.0008
localhost - - [18/Jul/2011:11:29:10 EDT] "GET /favicon.ico HTTP/1.1" 404 447
- -> /favicon.ico
```

這些是 Sinatra 印出的 log 資訊，從這些資訊你可以看出服務器有在運行，而且能了解到程式在瀏覽器背後做了些什麼事情。這些資訊還有助於你發現程式的問題。例如在最後一行它告訴你瀏覽器試圖存取 `/favicon.ico`，但是這個文件並不存在，因此它返回的狀態碼是 `404 Not Found`。

到這裡，我還沒有講到任何 web 相關的工作原理，因為首先你需要完成準備工作，以便後面的學習能順利進行，接下來的兩節習題中會有詳細的解釋。我會要求你用各種方法把你的 Sinatra 應用程式弄壞，然後再將其重新構建起來：這樣做的目的是讓你明白運行 Sinatra 程式需要準備好哪些東西。

發生了什麼事情？

在瀏覽器訪問到你的網頁應用程式時，發生了下面一些事情：

1. 瀏覽器通過網路連接到你自己的電腦，它的名字叫做 `localhost`，這是一個標準稱謂，表示的誰就是網路中你自己的這台電腦，不管它實際名字是什麼，你都可以使用 `localhost` 來訪問。它使用到 `port 4567`。
2. 連接成功以後，瀏覽器對 lib/gothonweb.rb

這個應用程式發出了HTTP請求(request)，要求訪問URL /`，這通常是一個網站的第一個URL。

3. 在 lib/gothonweb.rb 裡，我們有一個程式碼區段，裡面包含了 URL 的匹配關係。我們這裡只定義了一組匹配，那就是「/」。它的含義是：如果有人使用瀏覽器訪問 / 這一級目錄，Sinatra 將找到它，從而用它處理這個瀏覽器請求。
4. Sinatra 呼叫匹配到的程式碼區段，這段程式碼只簡單的回傳了一個字串傳回給瀏覽器。
5. 最後 Sinatra 完成了對於瀏覽器請求的處理將響應(response)回傳給瀏覽器，於是你就看到了現在的頁面。

確定你真的弄懂了這些，你需要畫一個示意圖，來理清資訊是如何從瀏覽器傳遞到 Sinata，再到 / 區段，再回到你的瀏覽器的。

修正錯誤

第一步，把第 6 行的 greeting 變數刪掉，然後重新刷瀏覽器。你應該會看到一個錯誤畫面，你可以通過這一頁豐富的資訊看出你的程式崩潰的原因。當然你已經知道出錯的原因是 greeting 的賦值遺失了，不過 Sinatra還是會給你一個挺好的錯誤頁面，讓你能找到出錯的具體位置。試試在這個錯誤頁面上做以下操作：

1. 看看 sinatra.error 變數。
2. 看看 REQUEST_ 變數裡的資訊。裡面哪些知識是你已經熟悉了的。這是瀏覽器發給你的 gothonweb 應用程式的資訊。這些知識對於日常網頁瀏覽沒有什麼用處，但現在你要學會這些東西，以便寫出web應用程式來。

建立基本的模板

你已經試過用各種方法把這個Sinatra 程式改錯，不過你有沒有注意到「Hello World」不是一個好 HTML 網頁呢？這是一個 web 應用，所以需要一個合適的HTML 響應頁面才對。為了達到這個目的，下一步你要做的是將「Hello World」以較大的綠色字體顯示出來。

第一步是建立一個 lib/views/index.erb 檔案，內容如下：

```
<html>
  <head>
    <title>Gothons Of Planet Percal #25</title>
  </head>
<body>

 <% if greeting %>
  <p>I just wanted to say <em style="color: green; font-size: 2em;"><%= greeting %></em>.
 <% else %>
  <em>Hello</em>, world!
 <% end %>

</body>
</html>
```

什麼是一個 .erb 的檔案？ERB 的全名是 **E**mbedded **R**u**b**y。.erb 檔案其實是一個內嵌一點 Ruby 程式碼的 HTML。如果你學過HTML的話，這些內容你看上去應該很熟悉。如果你沒學過HTML，那你應該去研究一下，試著用HTML寫幾個網頁，從而知道它的運作原理。既然這是一個 erb 模版，Sinatra 就會在模板中找到對應的位置，將參數的內容填充到模板中。例如每一個出現 ` 的位置，內容都會被替換成對應這個變數名的參數。

為了讓你的 lib/gothonweb.rb 處理模板，你需要寫一寫程式碼，告訴Sinatra 到哪裡去找到模板進行加載，以及如何渲染(render)這個模板，按下面的方式修改你的檔案：

```
require_relative "gothonweb/version"
require "sinatra"
require "erb"

module Gothonweb
  get '/' do
    greeting = "Hello, World!"
    erb :index, :locals => {:greeting => greeting}
  end
end
```

特別注意我改了 / 這個程式碼區段最後一行的內容，這樣它就可以呼叫 erb 然後把 greeting 變數傳給它。

改好上面的程式後，刷新一下瀏覽器中的網頁，你應該會看到一條和之前不同的綠色資訊輸出。你還可以在瀏覽器中通過「查看原始碼(View Source)」看到模板被渲染成了標準有效的HTML 原始碼。

這麼講也許有些太快了，我來詳細解釋一下模板的運作原理吧：

1. 在 lib/gothonweb.rb 你添加了一個 erb 函式呼叫。
2. 這個 erb 函式知道怎麼載入 lib/views 目錄夾裡的 .erb 的檔案。它知道去抓哪些檔案（在這個例子裡是 index.erb）。因為你傳了一個參數進去（ erb :index ... ）。
3. 現在，當瀏覽器讀取 / 且 lib/gothonweb.eb 匹配然後執行 get '/' do 區段，它再也沒有只是回

傳字串 `greeting` ，而是呼叫 `erb` 然後傳入 `greeting` 作為一個變數。

4. 最後，你讓 `lib/views/index.erb` 去檢查 `greeting` 這個變數，如果這個變數存在的話，就印出變數裡的內容。如果不存在的話，就會印出一個預設的訊息。

要深入理解這個過程，你可以修改 `greeting` 變數以及 HTML ，看看會友什麼效果。然後也創作另外一個叫做 `lib/views/foo.erb` 的模板。然後把 `erb :index` 改成 `erb :foo` 。從這個過程中你也可以看到，你傳入給 `erb` 的第一個參數只要匹配到 `lib/views` 下的 `.erb`` 檔案名稱，就可以被渲染出來了。

加分習題

1. 到 Sinatra 這個框架的官方網站去閱讀更多文件。
2. 實驗一下你在上述網站中看到的所有東西，包括他們的範例程式碼。
3. 閱讀有關於 HTML5 和 CSS3 相關的東西，自己練習寫幾個 `.html` 和 `.css` 文件。
4. 如果你有一個懂 **Rails** 的朋友可以幫你的畫，你可以自己試著使用 Rails 完成一下習題 50,51,52，看看結果會是什麼樣子。

习题 51: 从浏览器中取得输入

雖然能讓瀏覽器顯示「Hello World」是很有趣的一件事情，但是如果能讓用戶通過表單(form)向你的應用程序提交資訊就更有趣了。這節習題中，我們將使用form 改進你的web 程式，並且搞懂如何為一個網站程式寫自動化測試。

Web 運作原理

該學點無趣的東西了。在建立 form 前你需要先多學一點關於 web的運作原理。這裡講的並不完整，但是相當準確，在你的程式出錯時，它會幫你找到出錯的原因。另外，如果你理解了form 的應用，那麼建立 form 對你來說就會更容易了。

我將以一個簡單的圖示講起，它向你展示了web 請求的各個不同的部分，以及資訊傳遞的大致流程：

為了方便講述HTTP 請求(request) 的流程，我在每條線上面加了字母標籤以作區別。

1. 你在瀏覽器中輸入網址http://learnpythonthehardway.org/，然後瀏覽器會通過你的電腦的網路設備發出request（ 線路A ）。
2. 你的request 被傳送到網際網路（ 線路B ），然後再抵達遠端服務器（ 線路C ），然後我的伺服器將接受這個request。
3. 我的伺服器接受 request 後，我的 web 應用程式就去處理這個請求（ 線路D ），然後我的軼頁應用程式就會去運行 / (index) 這個「處理程序(handler)」。
4. 在程式碼 return 的時候，我的伺服器就會發出響應(response)，這個響應會再通過 線路D 傳遞到你的瀏覽器。
5. 這個網站所在的伺服器將響應由 線路D 獲取，然後通過 線路C 傳至網際網路。
6. 響應通過網路網路由 線路B 傳至你的電腦，電腦的網路卡再通過 線路A 將響應傳給你的瀏覽器。
7. 最後，你的瀏覽器顯示了這個響應的內容。

這段詳解中用到了一些術語。你需要掌握這些術語，以便在談論你的 web 應用時你能明白而且應用它們：

瀏覽器(browser)

這是你幾乎每天都會用到的軟件。大部分人不知道它真正的原理，他們只會把它叫作「網際網路」。它的作用其實是接收你輸入到地址欄網址(例如http://learnpythonthehardway.org)，然後使用該資訊向該網址對應的伺服器提出請求(request)。

IP 位址（Address）

通常這是一個像 http://learnpythonthehardway.org/ 一樣的URL (Uniform Resource Locator，統一資源定位符），它告訴瀏覽器該打開哪個網站。前面的 http 指出了你要使用的協議(protocol)，這裡我們

用的是「超文本傳輸協議(Hyper-Text Transport Protocol)」。你還可以試試ftp://ibiblio.org/，這是一個「FTP文件傳輸協議(File Transport Protocol)‘的例子。 learnpythonthehardway.org 這部分是「主機名(hostname)」，也就是一個便於人閱讀和記憶的字串，主機名會被匹配到一串叫作「IP 位址」的數字上面，這個「IP 位址」就相當於網路中一台電腦的電話號碼，通過這個號碼可以訪問到這台電腦。最後，URL中還可以尾隨一個「路徑「，例如http://learnpythonthehardway.org/book/ 中的 /book/ ，它對應的是伺服器上的某個文件或者某些資源，通過訪問這樣的網址，你可以向伺服器發出請求，然後獲得這些資源。網站地址還有很多別的組成部分，不過這些是最主要的。

連接(connection)

一旦瀏覽器知道了協議(http)、伺服器(learnpythonthehardway.org)、以及要獲得的資源，它就要去建立一個連接。這個過程中，瀏覽器讓操作系統(Operating System, OS) 打開計算機的一個「埠號(port)」（通常是80埠號），埠號準備好以後，操作系統會回傳給你的程式一個類似檔案的東西，它所做的事情就是通過網路傳輸和接收資料，讓你的電腦和learnpythonthehardway.org這個網站所屬的伺服器之間實現資料交流。當你使用 http://localhost:4567/ 訪問你自己的站點時，發生的事情其實是一樣的，只不過這次你告訴了瀏覽器要訪問的是你自己的電腦(localhost)，要使用的端口不是默認的80，而是 4567 。你還可以直接訪問http://learnpythonthehardway.org:80/，這和不輸入埠號效果一樣，因為HTTP的默認埠號本來就是80。

請求(request)

你的瀏覽器通過你提供的地址建立了連接，現在它需要從遠端伺服器要到它（或你）想要的資源。如果你在URL的結尾加了 /book/ ，那你想要的就是 /book/ 對應的檔案或資源，大部分的伺服器會直接為你呼叫 /book/index.html 這個檔案，不過我們就假裝不存在好了。瀏覽器為了獲得伺服器上的資源，它需要向伺服器發送一個「請求」。這裡我就不講細節了，為了得到伺服器上的內容，你必須先向伺服器發送一個請求才行。有意思的是，「資源」不一定非要是檔案。例如當瀏覽器向你的應用程序提出請求的時候，伺服器返回的其實是你的程式碼生成的一些東西。

伺服器(server)

伺服器指的是瀏覽器另一端連接的電腦，它知道如何回應瀏覽器請求的檔案和資源。大部分的 web 伺服器只要發送檔案就可以了，這也是伺服器流量的主要部分。不過你學的是使用 Ruby 組建一個伺服器，這個伺服器知道如何接受請求，然後返回用 Ruby 處理過的字符串。當你使用這種處理方式時，你其實是假裝把檔案發給了瀏覽器，其實你用的都只是程式碼而已。就像你在《習題50》中看到的，要構建一個「響應」其實也不需要多少程式碼。

響應(response)

這就是你的伺服器回覆給你的請求，傳回至瀏覽器的HTML，它裡邊可能有css、javascript、或者圖像等內容。以檔案響應為例，伺服器只要從磁碟讀取檔案，發送給瀏覽器就可以了，不過它還要將這些內容包在一個特別定義的「header]」中，這樣瀏覽器就會知道它獲取的是什麼類型的內容。以你的web 應用程式為例，你發送的其實還是一樣的東西，包括 header 也一樣，只不過這些資料是你用 Ruby 程式碼即時

生成的。

這個可以算是你能在網上找到的關於瀏覽器如何訪問網站的最快的快速課程了。這節課程應該可以幫你更容易地理解本節的習題，如果你還是不明白，就到處找資料多多了解這方面的資訊，知道你明白為止。有一個很好的方法，就是你對照著上面的圖示，將你在《習題50》中創建的 web 程式中的內容分成幾個部分，讓其中的各部分對應到上面的圖示。如果你可以正確地將程式的各部分對應到這個圖示，你就大致開始明白它的運作原理了。

表單(form)的運作原理

熟悉「表單」最好的方法就是寫一個可以接收表單資料的程式出來，然後看你可以對它做些什麼。先將你的 lib/gothonsweb.rb 修改成下面的樣子：

```
require_relative "gothonweb/version"
require "sinatra"
require "erb"

module Gothonweb
  get '/' do
    greeting = "Hello, World!"
    erb :index, :locals => {:greeting => greeting}
  end

  get '/hello' do
    name = params[:name] || "Nobody"
    greeting = "Hello, #{name}"
    erb :index, :locals => {:greeting => greeting}
  end
end
```

重啟你的 Sinatra（按CTRL-C後重新運行），確認它有運行起來，然後使用瀏覽器訪問 http://localhost:4567/hello，這時瀏覽器應該會顯示 “I just wanted to say Hello , Nobody.” ，接下來，將瀏覽器的地址改成 http://localhost:4567/hello?name=Frank，然後你可以看到頁面顯示為 “Hello, Frank.” ，最後將 name=Frank 修改為你自己的名字，你就可以看到它對你說 Hello 了。

讓我們研究一下你的程式裡做過的修改。

1. 我們沒有直接為 greeting 賦值，而是使用了 params Hash 從瀏覽器獲取數據。這Sinatra 個函數會將一組在 URL ? 後面的部份的 key / value 組加進 prarms Hash 裡。
2. 然後我從 params[:name] 中找到 name 的值，並為 greeting 賦值，這部份相信你已經很熟悉了。
3. 其他的內容和以前是一樣的，我們就不再分析了。

URL中該還可以包含多個參數。將本例的URL改成這樣子：http://localhost:4567/hello?name=Frank&greet=Hola 。然後修改程式碼，讓它去存取 prarams[:name] 和 params[:greet] ，如下所示：

```
greeting = "#{greet}, #{name}"
```

創建HTML表單

你可以通過URL參數實現表單提交，不過這樣看上去有些醜陋，而且不方便一般人使用，你真正需要的是一個「POST表單」，這是一種包含了 `<form>` 標籤的特殊 HTML 檔案。這種表單收集使用者輸入並將其傳遞給你的web程式，這和你上面實現的目的基本是一樣的。

讓我們來快速建立一個，從中你可以看出它的運作原理。你需要創建一個新的HTML文件，叫做 `lib/views/hello_form.erb` ：

```
<html>
  <head>
    <title>Sample Web Form</title>
  </head>
<body>

<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
  A Greeting: <input type="text" name="greet">
  <br/>
  Your Name: <input type="text" name="name">
  <br/>
  <input type="submit">
</form>

</body>
</html>
```

然後將 `lib/gothonsweb.rb` 改成這樣：

```
require_relative "gothonweb/version"
require "sinatra"
require "erb"

module Gothonweb

  get '/' do
    greeting = "Hello, World!"
    erb :index, :locals => {:greeting => greeting}
  end

  get '/hello' do
    erb :hello_form
  end

  post '/hello' do
    greeting = "#{params[:greet] || "Hello"}, #{params[:name] || "Nobody"}"
    erb :index, :locals => {:greeting => greeting}
  end

end
```

都寫好以後，重啟 web 程式，然後通過你的瀏覽器訪問它。

這回你會看到一個表單，它要求你輸入「一個問候語句(A Greeting)」和「你的名字(Your Name)」，等你輸入完後點擊「提交(Submit)」按鈕，它就會輸出一個正常的問候頁面，不過這一次你的URL還是 http://localhost:4567/hello，並沒有添加參數進去。

在 `hello_form.erb` 裡面關鍵的一行是 `<form action="/hello" method="POST">`，它告訴你的瀏覽器以下內容：

1. 從表單中的各個欄位收集使用者輸入的資料。
2. 讓瀏覽器使用一種POST類型的請求，將這些資料發送給服務器。這是另外一種瀏覽器請求，它會將表單欄位「隱藏」起來。
3. 將這個請求發送至 `/hello` URL，這是由 `action="/hello"` 告訴瀏覽器的。
4. 你可以看到兩段 `<input>` 標籤的名字屬性(name)和程式碼中的變數是對應的，另外我們在 class index 中使用的不再只是 GET 方法，而是另一個 POST 方法。

這個新程式的運作原理如下：

1. 瀏覽器訪問到 web 程式的 `/hello` 目錄，它發送了一個 GET 請求，於是我們的 `get '/hello/` 就運行了並傳回了hello_form。
2. 你填好了瀏覽器的表單，然後瀏覽器依照 `<form>` 中的要求，將資料通過POST 請求的方式發給web 程式。
3. Web 程式運行了 `post '/hello'` 而不是不是 `get '/hello/` 來處理這個請求。
4. 這個 `post '/hello'` 完成了它正常的功能，將 `hello` 頁面返回，這裡並沒有新的東西，只是一個新函式名稱而已。

作為練習，在 lib/views/index.erb 中添加一個鏈接，讓它指向 /hello ，這樣你可以反覆填寫並提交表單查看結果。確認你可以解釋清楚這個鏈接的工作原理，以及它是如何讓你實現在 lib/views/index.erb 和 lib/views/hello_form.erb 之間循環跳轉的，還有就是要明白你新修改過的 Ruby 程式碼，你需要知道在什麼情況下會運行到哪一部分程式碼。

Creating A Layout Template

在你下一節練習建立遊戲的過程中，你需要建立很多的小 HTML 頁面。如果你每次都寫一個完整的網頁，你會很快感覺到厭煩的。幸運的是你可以建立一個「外觀 (layout」模板，也就是一種提供了通用的 headers 和 footers 的外殼模板，你可以用它將你所有的其他網頁包裹起來。好程式設計師會盡可能減少重複動作，所以要做一個好程式設計師，使用外觀模板是很重要的。

將 lib/views/index.erb 修改成這樣：

```
<% if greeting %>
  <p>I just wanted to say <em style="color: green; font-size: 2em;"><%= greeting %></em>.
<% else %>
  <em>Hello</em>, world!
<% end %>
```

然後把 lib/views/hello_form.erb 修改成這樣：

```
<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>
```

面這些修改的目的，是將每一個頁面頂部和底部的反覆用到的「樣板 (boilerplate)」程式碼剝掉。這些被剝掉的程式碼會被放到一個單獨的 lib/views/layout.erb 檔案中，從此以後，這些反覆用到的程式碼就由 lib/views/layout.erb 來提供了。

上面的都改好以後，建立一個 lib/views/layout.erb 檔案，內容如下：

```
<html>
  <head>
    <title>Gothons From Planet Percal #25</title>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Sinatra 預設會自動去找名字為 `layout` 的外觀模板，並且使用它作為其他模板的「基礎」模板。你也可以修改已經用作任何頁面的基礎模板的 template。重啟你的程式觀察一下，然後試著用各種方法修改你的layout模板，不要修改你別的模板，看看輸出會有什麼樣的變化。

為表單撰寫自動測試程式碼

使用瀏覽器測試 web 程式是很容易的，只要點刷新按鈕就可以了。不過畢竟我們是程式設計師嘛，如果我們可以寫一些程式碼來測試我們的程式，為什麼還要重複手動測試呢？接下來你要做的，就是為你的web程式寫一個小測試。這會用到你在《習題47》學過的一些東西，如果你不記得的話，可以回去複習一下。

我已經為此建立了一個簡單的小函式，讓你判斷(assert) web程序的響應，這個函數有一個很合適的名字，就叫 `assert_response`。創建一個 `tests/tools.rb` 檔案，內容如下：

```
require 'test/unit'

def assert_response(resp, contains=nil, matches=nil, headers=nil, status=200)

  assert_equal(resp.status, status, "Expected response #{status} not in #{resp}")

  if status == 200
    assert(resp.body, "Response data is empty.")
  end

  if contains
    assert((resp.body.include? contains), "Response does not contain #{contains}")
  end

  if matches
    reg = Regexp.new(matches)
    assert reg.match(contains), "Response does not match #{matches}"
  end

  if headers
    assert_equal(resp.headers, headers)
  end

end
```

最後，執行 `test/test_gothonsweb.rb` 去測試你的程式：

```
$ ruby test/test_gothonweb.rb
Loaded suite test/test_gothonweb
Started
.
Finished in 0.023839 seconds.

1 tests, 9 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 57414
```

`rack/test` 函式庫包含了一串很簡單的 API 可以讓你處理請求。他們是 `get`，`put`，`post`，`delete` 和 `head` 函式，模擬程式會遇到的各類類型請求。

所有假的 (mock) request 函式會有一樣的參數模式：

```
get '/path', params={}, rack_env={}
```

- `/path` 是 request 路徑，而且可以選擇性的包含一個 query string。
- `params` 是一組 query/post 的 Hash 參數，一個 request body 字串，或者是 nil
- `rack_env` 是一個 Rack 環境值 Hash。這可以用來設置 request 的 header 和其他 request 相關的資訊，例如 session 內的資料。

這樣的運作方式就不用實際運作一個真的 web 伺服器，如此一來你就可以使用自動化測試程式碼去測試，當然同時你也可以使用瀏覽器去測試一個執行中的伺服器。

為了驗證(validate) 函式的響應，你需要使用 `test/tools.rb` 中定義的 `assert_response` 函式，裡面內容是：

To validate responses from this function, use the `assert_response` function from `test/tools.rb` which has:

```
assert_response(resp, contains=nil, matches=nil, headers=nil, status=200)
```

把你呼叫 `get` 或 `post` 得到的響應傳遞給這個函數，然後將你要檢查的內容作為參數傳遞給這個函數。你可以使用 `contains` 參數來檢查響應中是否包含指定的值，使用 `status` 參數可以檢查指定的響應狀態。這個小函式其實包含了很多的資訊，所以你還是自己研究一下的比較好。

在 `test/test_gothonsweb.rb` 自動測試腳本中，我首先確認 `/foo` URL 傳回了一個「404 Not Found」響應，因為這個 URL其實是不存在的。然後我檢查了 `/hello` 在 GET 和 POST 兩種請求的情況下都能正常運作。就算你沒有弄明白測試的原理，這些測試程式碼應該是很好讀懂的。

花一些時間研究一下這個最新版的web程式，重點研究一下自動測試的運作原理。

加分習題

1. 閱讀和HTML 相關的更多資料，然後為你的表單設計一個更好的輸出格式。你可以先在紙上設計出來，然後用HTML 去實現它。
2. 這是一道難題，試著研究一下如何進行檔案上傳，通過網頁上傳一張圖像，然後將其保存到磁碟中。
3. 更難的難題，找到 HTTP RFC 文件（講述HTTP 運作原理的技術文件），然後努力閱讀一下。這是一篇很無趣的文件，不過偶爾你會用到裡邊的一些知識。
4. 又是一道難題，找人幫你設置一個 web 伺服器，例如Apache、Nginx、或者thttpd。試著讓伺服器伺服一下你建立的.html 和.css 文件。如果失敗了也沒關係，web 服務器本來就都有點爛。
5. 完成上面的任務後休息一下，然後試著多建立一些 web 程式出來。你應該仔細閱讀 Sinatra 中關於會話(session)的內容，這樣你可以明白如何存留使用者的狀態資訊。

习题 52: 创造你的网页游戏

這本書馬上就要結束了。本章的練習對你是一個真正的挑戰。當你完成以後，你就可以算是一個能力不錯的 Ruby 初學者了。為了進一步學習，你還需要多讀一些書，多寫一些程式，不過你已經具備進一步學習的技能了。接下來的學習就只是時間、動力、以及資源的問題了。

在本章習題中，我們不會去創建一個完整的遊戲，取而代之的是我們會為《習題42》中的遊戲建立一個“引擎(engine)”，讓這個遊戲能夠在瀏覽器中運行起來。這會涉及到將《習題42》中的遊戲「重構(refactor)」，將《習題47》中的架構混合進來，添加自動測試程式碼，最後建立一個可以運行遊戲的web引擎。

這是一節很「龐大」的習題。我預測你要花一周到一個月才能完成它。最好的方法是一點一點來，每晚上完成一點，在進行下一步之前確認上一步有正確完成。

重構《習題42》的遊戲

你已經在兩個練習中修改了 `gothonweb` 專案，這節習題中你會再修改一次。這種修改的技術叫做「重構(refactoring)」，或者用我喜歡的講法來說，叫「修修補補(fixing stuff)」。重構是一個程式術語，它指的是清理舊程式碼或者為舊程式碼添加新功能的過程。你其實已經做過這樣的事情了，只不過不知道這個術語而已。這是寫軟體過程的第二個自然屬性。

你在本節中要做的，是將《習題47》中的可以測試的房間地圖，以及《習題42》中的遊戲這兩樣東西歸併到一起，創建一個新的遊戲架構。遊戲的內容不會發生變化，只不過我們會通過“重構”讓它有一個更好的架構而已。

第一步是將 `ex47.rb` 的內容複製到 `gothonweb/lib/map.rb` 中，然後將 `ex47_tests.rb` 的內容複製到 `gothonweb/test/test_map.rb` 中，然後再次運行測試，確認他們還能正常運作。

> **Note:** 從現在開始我不會再向你展示運行測試的輸出了，我就假設你回去運行這些測試，而且知道怎樣的輸出是正確的。

將《習題47》的程式碼拷貝完畢後，你就該開始重構它，讓它包含《習題42》中的地圖。我一開始會把基本架構為你準備好，然後你需要去完成 `map.rb` 和 `map_tests.rb` 裡邊的內容。

首先要做的是使用 `Room` 類來構建基本的地圖架構：

```
class Room

  attr_accessor :name, :description, :paths

  def initialize(name, description)
    @name = name
    @description = description
    @paths = {}
```

```
    @paths = {}
  end

  def go(direction)
    @paths[direction]
  end

  def add_paths(paths)
    @paths.update(paths)
  end

end

central_corridor = Room.new("Central Corridor",
%q{
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
})

laser_weapon_armory = Room.new("Laser Weapon Armory",
%q{
Lucky for you they made you learn Gothon insults in the academy.
You tell the one Gothon joke you know:
Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr.
The Gothon stops, tries not to laugh, then busts out laughing and can't move.
While he's laughing you run up and shoot him square in the head
putting him down, then jump through the Weapon Armory door.

You do a dive roll into the Weapon Armory, crouch and scan the room
for more Gothons that might be hiding.  It's dead quiet, too quiet.
You stand up and run to the far side of the room and find the
neutron bomb in its container.  There's a keypad lock on the box
and you need the code to get the bomb out.  If you get the code
wrong 10 times then the lock closes forever and you can't
get the bomb.  The code is 3 digits.
})

the_bridge = Room.new("The Bridge",
%q{
The container clicks open and the seal breaks, letting gas out.
You grab the neutron bomb and run as fast as you can to the
bridge where you must place it in the right spot.

You burst onto the Bridge with the netron destruct bomb
under your arm and surprise 5 Gothons who are trying to
take control of the ship.  Each of them has an even uglier
clown costume than the last.  They haven't pulled their
```

```
weapons out yet, as they see the active bomb under your
arm and don't want to set it off.
})

escape_pod = Room.new("Escape Pod",
%q{
You point your blaster at the bomb under your arm
and the Gothons put their hands up and start to sweat.
You inch backward to the door, open it, and then carefully
place the bomb on the floor, pointing your blaster at it.
You then jump back through the door, punch the close button
and blast the lock so the Gothons can't get out.
Now that the bomb is placed you run to the escape pod to
get off this tin can.

You rush through the ship desperately trying to make it to
the escape pod before the whole ship explodes.  It seems like
hardly any Gothons are on the ship, so your run is clear of
interference.  You get to the chamber with the escape pods, and
now need to pick one to take.  Some of them could be damaged
but you don't have time to look.  There's 5 pods, which one
do you take?
})

the_end_winner = Room.new("The End",
%q{
You jump into pod 2 and hit the eject button.
The pod easily slides out into space heading to
the planet below.  As it flies to the planet, you look
back and see your ship implode then explode like a
bright star, taking out the Gothon ship at the same
time.  You won!
})

the_end_loser = Room.new("The End",
%q{
You jump into a random pod and hit the eject button.
The pod escapes out into the void of space, then
implodes as the hull ruptures, crushing your body
into jam jelly.
})

escape_pod.add_paths({
    '2' => the_end_winner,
    '*' => the_end_loser
})

generic_death = Room.new("death", "You died.")

the_bridge.add_paths({
    'throw the bomb' => generic_death,
    'slowly place the bomb' => escape_pod
})

laser_weapon_armory.add_paths({
```

```
    '0132' => the_bridge,
    '*' => generic_death
})

central_corridor.add_paths({
    'shoot!' => generic_death,
    'dodge!'=> generic_death,
    'tell a joke' => laser_weapon_armory
})

START = central_corridor
```

你會發現我們的 `Room` 類和地圖有一些問題：

1. 在進入一個房間以前會打出一段文字作為房間的描述，我們需要將這些描述和每個房間關聯起來，這樣房間的次序就不會被打亂了，這對我們的遊戲是一件好事。這些描述本來是在 `if-else` 結構中的，這是我們後面要修改的東西。
2. 原版遊戲中我們使用了專門的程式來生成一些內容，例如炸彈的激活鍵碼，艦艙的選擇等，這次我們做遊戲時就先使用預設值好了，不過後面的加分習題裡，我會要求你把這些功能再加到遊戲中。
3. 我為所有的遊戲中的失敗結尾寫了一個 `generic_death`，你需要去補全這個函式。你需要把原版遊戲中所有的失敗結尾都加進去，並確保程式碼能正確運行。
4. 我添加了一種新的轉換模式，以 `"*"` 為標記，用來在遊戲引擎中實現「catch-all」動作。

等你把上面的程式碼基本寫好以後，接下來就是引導你繼續寫下去的自動測試的內容 `test/test_map.rb` 了：

```
require 'test/unit'
require_relative '../lib/map'

class MapTests < Test::Unit::TestCase

  def test_room()
    gold = Room.new("GoldRoom",
              %q{This room has gold in it you can grab. There's a
              door to the north.})
    assert_equal(gold.name, "GoldRoom")
    assert_equal(gold.paths, {})
  end

  def test_room_paths()
    center = Room.new("Center", "Test room in the center.")
    north = Room.new("North", "Test room in the north.")
    south = Room.new("South", "Test room in the south.")

    center.add_paths({'north' => north, 'south' => south})
    assert_equal(center.go('north'), north)
    assert_equal(center.go('south'), south)
  end

  def test_map()
    start = Room.new("Start", "You can go west and down a hole.")
    west = Room.new("Trees", "There are trees here, you can go east.")
    down = Room.new("Dungeon", "It's dark down here, you can go up.")

    start.add_paths({'west' => west, 'down' => down})
    west.add_paths({'east' => start})
    down.add_paths({'up' => start})

    assert_equal(start.go('west'), west)
    assert_equal(start.go('west').go('east'), start)
    assert_equal(start.go('down').go('up'), start)
  end

  def test_gothon_game_map()
    assert_equal(START.go('shoot!'), generic_death)
    assert_equal(START.go('dodge!'), generic_death)

    room = START.go('tell a joke')
    assert_equal(room, laser_weapon_armory)
  end

end
```

你在這部分練習中的任務是完成地圖，並且讓自動測試可以完整地檢查過整個地圖。這包括將所有的 `generic_death` 物件修正為遊戲中實際的失敗結尾。讓你的程式碼成功運行起來，並讓你的測試越全面越好。後面我們會對地圖做一些修改，到時候這些測試將保證修改後的程式碼還可以正常工作。

會話(session)和用戶跟踪

在你的 web 程式運行的某個位置，你需要追踪一些信息，並將這些信息和用戶的瀏覽器關聯起來。在 HTTP 協議的框架中，web 環境是「無狀態(stateless)」的，這意味著你的每一次請求和你其它的請求都是相互獨立的。如果你請求了頁面A，輸入了一些資料，然後點了一個頁面B 的鏈接，那你在頁面A 輸入的數據就全部消失了。

解決這個問題的方法是為 web 程式建立一個很小的資料儲存功能，給每個瀏覽器賦予一個獨一無二的數字，用來跟踪瀏覽器所作的事情。這個功能通常適用資料庫或者是存儲在磁碟上的檔案來實現。在 Sinatra 這個框架中實現這樣的功能是很容易的，以下就是一個這樣的例子（使用 Rack middleware）：

```
require 'rubygems'
require 'sinatra'

use Rack::Session::Pool

get '/count' do
  session[:count] ||= 0
  session[:count] +=1
  "Count: #{session[:count]}"
end

get '/reset' do
  session.clear
  "Count reset to 0."
end
```

建立引擎

你應該已經寫好了遊戲地圖和它的單元測試程式碼碼。現在我要求你製作一個簡單的遊戲引擎，用來讓遊戲中的各個房間運轉起來，從玩家收集輸入，並且記住玩家到了那一幕。我們將用到你剛學過的會話來製作一個簡單的引擎，讓它可以：

1. 為新使用者啟動新的遊戲。
2. 將房間展示給使用者。
3. 接受使用者的輸入。
4. 在遊戲中處理使用者的輸入。
5. 顯示遊戲的結果，繼續遊戲的下一幕，知道玩家角色死亡為止。

為了建立這個引擎，你需要將我們久經考驗的 `lib/gothonsweb.rb` 搬過來，建立一個功能完備的、基於 session 的遊戲引擎。這裡的難點是我會先使用基本的 HTML 檔案創建一個非常簡單的版本，接下來將由你完成它，基本的引擎是這個樣子的：

```
require_relative "gothonweb/version"
require_relative "map"
require "sinatra"
require "erb"

module Gothonweb

  use Rack::Session::Pool

  get '/' do
    # this is used to "setup" the session with starting values
    p START
    session[:room] = START
    redirect("/game")
  end

  get '/game' do
    if session[:room]
      erb :show_room, :locals => {:room => session[:room]}
    else
      # why is there here? do you need it?
      erb :you_died
    end
  end

  post '/game' do
    action = "#{params[:action] || nil}"
    # there is a bug here, can you fix it?
    if session[:room]
      session[:room] = session[:room].go(params[:action])
    end
    redirect("/game")
  end

end
```

下一步，你應該刪除 `lib/views/hello_form.erb` 和 `lib/views/index.erb` 然後創作兩個在上述 code 提到的 template，這裡是一個非常簡單的 `lib/views/show_room.erb`：

```
<h1><%= room.name %></h1>

<pre>
<%= room.description %>
</pre>

<% if room.name == "death" %>
  <p>
  <a href="/">Play Again?</a>
  </p>
<% else %>
  <p>
  <form action="/game" method="POST">
    - <input type="text" name="action"> <input type="SUBMIT">
  </form>
  </p>
<% end %>
```

這就用來顯示遊戲中的房間的模板。接下來，你需要在使用者跑到地圖的邊界時，用一個模板告訴使用者他的角色的死亡信息，也就是 `lib/views/you_died.erb` 這個模板：

```
<h1>You Died!</h1>

<p>Looks like you bit the dust.</p>
<p><a href="/">Play Again</a></p>
```

準備好了這些文件，你現在可以做下面的事情了：

1. 讓測試代碼 `test/test_gothonsweb.rb` 再次運行起來，這樣你就可以去測試這個遊戲。由於 session 的存在，你可能頂多只能實現幾次點擊，不過你應該可以做出一些基本的測試來。
2. 執行 lib/gothonsweb.rb` 腳本，試玩一下你的遊戲。
3. 你需要和往常一樣刷新和修正你的遊戲，慢慢修改遊戲的HTML 檔案和引擎，直到你實現遊戲需要的所有功能為止。

你的期末考試

你有沒有覺著我一下子給了你超多的資訊呢？那就對了，我想要你在學習技能的同時可以有一些可以用來鼓搗的東西。為了完成這節習題，我將給你最後一套需要你自己完成的練習。你將注意到，到目前為止你寫的遊戲並不是很好，這只是你的第一版程式碼而已。你現在的任務是讓遊戲更加完善，實現下面的這些功能：

1. 修正程式碼中所有我提到和沒提到的bug，如果你發現了新的bug，你可以告訴我。
2. 改進所有的自動測試，讓你可以測試更多的內容，直到你可以不用瀏覽器就能測到所有的內容為止。
3. 讓HTML 頁面看上去更美觀一些。
4. 研究一下網頁登錄系統，為這個程式創建一個登錄界面，這樣人們就可以登錄這個遊戲，並且可以保

存遊戲高分。

5. 完成遊戲地圖，盡可能地把遊戲做大，功能做全。
6. 給用戶一個「幫助系統」，讓他們可以查詢每個房間裡可以執行哪些命令。
7. 為你的遊戲添加新功能，想到什麼功能就添加什麼功能。
8. 創建多個地圖，讓用戶可以選擇他們想要玩的一張來進行遊戲。你的 `lib/gothonsweb.rb` 應該可以運行提供給它的任意的地圖，這樣你的引擎就可以支持多個不同的遊戲。
9. 最後，使用你在習題 48 和49 中學到的東西來創建一個更好的輸入處理器。你手頭已經有了大部分必要的程式碼，你只需要改進語法，讓它和你的輸入表單以及遊戲引擎掛鉤即可。

祝你好運！

下一步

現在還不能說你是一個程式員。這本書的目的相當於給你一個「程式設計師棕帶」。你已經了解了足夠的寫程式基礎，並且有能力閱讀別的寫程式書籍了。讀完這本書，你應該已經掌握了一些學習的方法，並且具備了該有的學習態度，這樣你在閱讀其他 Ruby 書籍時也許會更順利，而且能學到更多東西。

> **Rob says:** 為了更有趣，我推薦你閱讀 Why’s (Poignant) Guide to Ruby: http://mislav.uniqpath.com/poignant-guide 大部份的程式內容現在都正在被 review。但是 Why 的心智無比聰明，而且他的書就像是一件藝術品。去看看它的一些 opensource 專案，你可以從他的程式碼裡學到許多東西。

或許，你現在已經可以開始鼓搗一些程式出來了。如果你手上有需要解決的問題，試著寫個程式解決一下。你一開始寫的東西可能很挫，不過這沒有關係。以我為例，我在學每一種語言的初期都是很挫的。沒有哪個初學者能寫出完美的代碼來，如果有人告訴你他有這本事，那他只是在厚著臉皮撒謊而已。

最後，記住學習寫程式是要投入時間的，你可能需要至少每天晚上練習幾個小時。順便告訴你，當你每晚學習 Ruby 的時候，我在努力學習彈吉他。我每天練習2 到4 小時，而且還在學習基本的音階。

每個人都是某一方面的菜鳥。

一个老程式设计师的建议

你已經完成了這本書而且打算繼續寫程式。也許這會成為你的一門職業，也許你只是作為業餘愛好玩玩。無論如何，你都需要一些建議以保證你在正確的道路上繼續前行，並且讓這項新的愛好為你帶來最大程度的享受。

我從事寫程式這行已經太長時間，長到對我來說寫程式已經是非常乏味的事情了。我寫這本書的時候，已經懂得大約20 種程式語言，而且可以在大約一天或者一個星期內學會一門程式語言(取決於這門語言有多古怪)。現在對我來說寫程式這件事情已經很無聊，已經談不上什麼興趣了。當然這不是說寫程式本身是一件無聊的事情，也不是說你以後也一定會這樣覺得，這只是我個人在當前的感覺而已。

在這麼久的旅程下來我的體會是：程式語言這東西並不重要，重要的是你用這些語言做的事情。事實上我一直知道這一點，不過以前我會周期性地被各種程式語言分神而忘記了這一點。現在我是永遠不會忘記這一點了，你也不應該忘記這一點。

你學到和用到的程式語言並不重要。不要被圍繞某一種語言的宗教把你扯進去，這只會讓你忘掉了語言的真正目的，也就是作為你的工具來實現有趣的事情。

寫程式作為一項智力活動，是唯一一種能讓你創建交互式藝術的藝術形式。你可以建立專案讓別人使用，而且你可以間接地和使用者溝通。沒有其他的藝術形式能做到如此程度的交互性。電影領著觀眾走向一個方向，繪畫是不會動的。而程式碼卻是雙向互動的。

寫程式作為一項職業只是一般般有趣而已。寫程式可能是一份好工作，但如果你想賺更多的錢而且過得更快樂，你其實開一間快餐分店就可以了。你最好的選擇是將你的程式技術作為你其他職業的秘密武器。

技術公司裡邊會寫程式的人多到一毛錢一打，根本得不到什麼尊敬。而在生物學、醫藥學、政府部門、社會學、物理學、數學等行業領域從事寫程式的人就能得到足夠的尊敬，而且你可以使用這項技能在這些領域做出令人驚異的成就。

當然，所有的這些建議都是沒啥意義的。如果你跟著這本書學習寫軟體而且覺得很喜歡這件事情的話，那你完全可以將其當作一門職業去追求。你應該繼續深入拓展這個近五十年來極少有人探索過的奇異而美妙的智力工作領域。若能從中得到樂趣當然就更好了。

最後我要說的是學習創造軟體的過程會改變你而讓你與眾不同。不是說更好或更壞，只是不同了。你也許會發現因為你會寫軟件而人們對你的態度有些怪異，也許會用「怪人「這樣的詞來形容你。也許你會發現因為你會戳穿他們的邏輯漏洞而他們開始討厭和你爭辯。甚至你可能會發現有人因為你懂得電腦怎麼運作而覺得你是個討厭的怪人。

對於這些我只有一個建議: 讓他們去死吧。這個世界需要更多的怪人，他們知道東西是怎麼工作的而且喜歡找到答案。當他們那樣對你時，只要記住這是你的旅程，不是他們的。「與眾不同」不是誰的錯，告訴你「與眾不同是一種錯」的人只是嫉妒你掌握了他們做夢都不能想到的技能而已。

你會寫程式。他們不會。這真他媽的酷。