

目录

笨方法更简单	5
读和写	5
注重细节	5
发现不同	6
不要复制贴上	6
对于坚持练习的一点提示	6
练习 0 准备工作	7
Mac OSX	8
OSX: 你应该看到的结果	9
Windows	10
Windows: 你应该看到的结果	10
Linux	12
Linux: 你应该看到的结果	13
给新手的告诫	14
练习 1 第一个程序	15
你应该看到的内容	16
加分练习	17
练习 2 注释和井号	18
练习 3 数字和数学表达式	19
你应该看到的结果	22
加分练习	23
练习 4: 变量和命名	24
你应该看到的结果	25
加分练习	26
更多的加分练习	26
练习 5: 更多的变量和打印	27
你应该看到的结果	28
加分练习	29
练习 6: 字符串和文字	29
加分练习	32
练习 8: 打印, 打印	35
加分练习	37
练习 9: 打印, 打印, 还是打印	37
加分练习	39
练习 10: 那是什么	39
加分练习	42
练习 11: 提问	42
加分练习	44
练习 12: 模块 (Module)	44
等一下! 功能 (Features) 还有另外一个名字	45
加分练习	45

练习 13: 参数、解包、参数.....	45
你应该看到的结果.....	47
加分练习.....	48
练习 14: 提示和传递.....	49
你应该看到的结果.....	51
加分练习.....	52
练习 15: 读取文件	52
你应该看到的结果.....	54
加分练习.....	55
练习 16: 读写文件	56
你应该看到的结果.....	58
加分练习.....	59
练习 17: 更多文件操作.....	60
你应该看到的结果.....	61
加分练习.....	63
练习 18: 命名、变量、程序代码、函式.....	63
你应该看到的结果.....	66
加分练习.....	66
练习 19: 函式和变量.....	67
你应该看到的结果.....	69
加分练习.....	71
练习 20: 函式和文件.....	72
你应该看到的结果.....	74
加分练习.....	75
练习 21: 函式的返回值.....	76
你应该看到的结果.....	78
加分练习.....	79
练习 22: 到现在你学到了哪些东西.....	80
你学到的东西.....	81
练习 23: 阅读一些程序代码.....	81
练习 24: 更多练习	83
你应该看到的结果.....	85
加分练习.....	87
练习 25: 更多更多的练习.....	87
你应该看到的结果.....	90
加分练习.....	93
练习 26: 恭喜你, 现在来考试了!	94
练习 27: 记住逻辑关系.....	95
逻辑术语.....	96
真值表.....	96
练习 28: 布尔 (Boolean) 表示式练习	98
你应该看到的结果.....	100
加分练习.....	101
练习 29: 如果 (if)	101

你应该看到的结果.....	104
加分练习.....	104
练习 30: Else 和 If.....	104
你应该看到的结果.....	107
加分练习.....	107
练习 31: 做出决定	107
你应该看到的结果.....	110
加分练习.....	114
练习 32: 循环和数组.....	114
你应该看到的结果.....	117
加分练习.....	119
练习 33: While 循环	119
你应该看到的结果.....	121
加分练习.....	123
练习 34: 存取数组里的元素.....	123
加分练习.....	125
练习 35: 分支 (Branches) 和函式 (Functions)	126
你应该看到的结果.....	131
加分练习.....	132
练习 36: 设计和测试.....	133
If 语句的规则	133
Rules For Loops	134
除错 (Debug) 的小技巧	134
家庭作业.....	134
练习 37: 复习各种符号.....	135
Keywords (关键字)	135
资料类型.....	138
字符串格式(String Formats).....	139
Operators	139
练习 38: 阅读代码	141
加分练习.....	142
练习 39: 数组的操作.....	143
你应该看到的结果.....	145
加分练习.....	146
练习 40: Hash, 可爱的 Hash.....	146
你应该看到的结果.....	152
加分练习.....	153
练习 41: 来自 Percal 25 号行星的哥顿人(Gothons).....	154
你应该看到的结果.....	167
加分练习.....	173
练习 43: 你来制作一个游戏.....	173
练习 44: 评估你的游戏.....	174
函数的风格.....	175
Classh (类) 的风格	175

代码风格.....	176
好的注释.....	177
评估你的游戏.....	177
练习 45: 对象, 类和从属关系.....	178
写完后的代码是什么样子.....	179
加分练习.....	185
练习 46: 一个项目骨架.....	186
骨架内容: Linux/OSX.....	186
安装 Gems	189
使用这个骨架.....	190
小测验.....	191
练习 47: 自动化测试.....	191
撰写 Test Case.....	192
测试指南.....	196
你应该看到的结果.....	197
加分练习.....	198
练习 49: 创造句子	208
匹配(Match) And 窥视(Peek)	210
句子的语法.....	212
关于异常(Exception)	218
你应该测试的东西.....	218
加分练习.....	218
练习 50: 你的第一个网站.....	219
安装 Sinatra	219
发生了什么事情?	223
修正错误.....	223
建立基本的模板.....	224
加分练习.....	227
练习 51: 从浏览器中取得输入.....	227
Web 运作原理	227
浏览器(browser).....	228
IP 位址 (Address)	228
连接(connection)	229
请求(request).....	229
服务器(server).....	230
响应(response)	230
表单(form)的运作原理	231
创建 HTML 表单	233
Creating A Layout Template	236
为表单撰写自动测试代码.....	239
加分练习.....	242

笨方法更简单

这本小书的目的是让你起步学习程序。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序语言。写练习题是每个程序员的必经之路：

1. 做每一道习题
2. 一字不差地写出每一个程序
3. 让程序运行起来 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序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「写程序」，但你会学到最基本的学习方法。

这本书的目的是教会你程序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，那你学习写程序也会成问题。尤其如果你连程序源代码中的那些奇怪符号都打不出来的话，就根本别提写程序了。没有这样基本技能的话，你将连最基本的代码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，你需要将程序代码写下来并且运行起来。这个过程也会让你对程序语言更加熟悉。

注重细节

区分好程序员和差程序员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。以写程序来讲，这样你得到的结果只能是毛病多多难以使用的代码。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

程序员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。有经验的程序员拿着两份仅有细微不同的程序，可以立即指出里边的不同点来。程序员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，即使有经验的程序员也会偶尔写错。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序里的错误和 **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，让你有能力读程序代码、写程序代码、观察程序代码。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序时，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序的过程中要记住一点，就是所谓的「万事开头难」，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“有天赋”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 多年经验的程序老手相比，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，却还是不知道写程序编程是怎么回事。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。

练习 0 准备工作

这道练习并没有程序代码。它的主要目的是让你在电脑上安装好 Ruby，你应该尽量照着指示操作。

这份教学已经预设你将使用 Ruby 1.9.2

你的系统里面可能已经装好了 Ruby。打开 `console` 并尝试运行：

```
$ ruby -v
```

ruby 1.9.2

如果你的系统内并没有 Ruby，不论你使用的是哪一套操作系统，我都强烈建议你使用 Ruby Version Manager (RVM) 安装 Ruby。

Mac OSX

你需要做下列任务来完成这个练习：

1. 用浏览器打开 <http://learnpythonthehardway.org/wiki/ExerciseZero> 下载并安装 gedit 文字编辑器。
2. 把 gedit 放到桌面或者快速启动列，这样以后你就可以方便使用它了。这两个选项在安装时可以看到。
 - a. 执行 **gedit**（也就是你的编辑器），我们要先改掉一些愚蠢的预设值。
 - b. 从 gedit menu 中打开 Preferences, 选择 Editor 页面。
 - c. 将 Tab width: 改为 2。
 - d. 选择(确认有勾选到该选项) Insert spaces instead of tabs。
 - e. 然后打开「Automatic indentation」选项。
 - f. 转到 View 页面，打开「Display line numbers」选项。
3. 找到「Terminal」程序。它的名字是 Command Promot，或者你可以直接执行 cmd。
4. 为它建立一个捷径，放到桌面或者是快速启动列中以方便使用。
5. 执行 Terminal，这个程序看上去不怎么地。
6. 在 Termnal 程序里执行 irb。在 Terminal 中执行程序的方式是输入程序的名称然后再敲一下 Return (Enter)。
 - a. 如果你执行 irb 但发现不存在(不认得 irb 这个指令)。请用 Ruby Version Manager (RVM) 安装 Ruby。
7. 敲击 CTRL-Z (Z) 退出 irb。
8. 这样你就应该能回到敲 irb 前的提示介面了。如果没有的话自己研究一下为什么。
9. 学着使用 Terminal 创建一个目录，你可以上网搜寻怎么做。
10. 学着使用 Terminal 进入一个目录，同样你可以上网搜寻。
11. 使用你的编辑器在你进入的目录下建立一个档案。你将建立一个档案。使用「Save」或者「Save As...」选项，然后选择这个目录。
12. 使用键盘切回到 Terminal 视图，如果不知道怎样使用键盘切换，你一样可以上网搜寻。

13. 回到 **Terminal**，看看你能不能使用命令列出你在目录里新建立的档案，在网路上搜寻怎么列出档案夹里的资料。 > **Note:** 如果你在使用 **gedit** 上有问题，很有可能这是 **non-English keyboards layout** 造成的，那么我会建议你改使用 <http://www.barebones.com/products/textwrangler/>。

OSX: 你应该看到的结果

以下是我在自己电脑的 **Terminal** 中练习上述练习时看到的内容。可能会跟你在自己电脑中看的到结果有些不同，所以看看你能不能搞清楚两者的差异。

```
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. 用浏览器打开 <http://learnpythonthehardway.org/wiki/ExerciseZero> 下载并安装 gedit 文字编辑器。
2. 把 gedit 放到桌面或者快速启动列，这样以后你就可以方便使用它了。这两个选项在安装时可以看到。**a.** 执行 **gedit**（也就是你的编辑器），我们要先改掉一些愚蠢的预设值。**b.** 从 gedit menu 中打开 Preferences，选择 Editor 页面。**c.** 将 Tab width: 改为 2。**d.** 选择(确认有勾选到该选项)Insert spaces instead of tabs。**e.** 然后打开「Automatic indentation」选项。**f.** 转到 View 页面，打开「Display line numbers」选项。
3. 找到「Terminal」程序。它的名字是 Command Prompt，或者你可以直接执行 **cmd**。
4. 为它建立一个捷径，放到桌面或者是快速启动列中以方便使用。
5. 执行 Terminal，这个程序看上去不怎么地。
6. 在 Terminal 程序里执行 **irb**。在 Terminal 中执行程序的方式是输入程序的名称然后再敲一下 Return (Enter)。**a.** 如果你执行 **irb** 但发现不存在(不认得 **irb** 这个指令)。请用 Ruby Version Manager (RVM) 安装 Ruby。
7. 敲击 CTRL-Z (Z) 退出 **irb**。
8. 这样你就应该能回到敲 **irb** 前的提示介面了。如果没有的话自己研究一下为什么。 ..
_Ruby Version Manager (RVM): <https://rvm.beginrescueend.com/>
9. 学着使用 Terminal 创建一个目录，你可以上网搜寻怎么做。
10. 学着使用 Terminal 进入一个目录，同样你可以上网搜寻。
11. 使用你的编辑器在你进入的目录下建立一个档案。你将建立一个档案。使用「Save」或者「Save As...」选项，然后选择这个目录。
12. 使用键盘切回到 Terminal 视图，如果不知道怎样使用键盘切换，你一样可以上网搜寻。
13. 回到 Terminal，看看你能不能使用命令列列出你在目录里新建立的档案，在网路上搜寻怎么列出档案夹里的资料。 > Warning: 对于 Ruby 来说 Windows 是个大问题。有时候你在一台电脑上装得好好的，但在另外一台电脑上却会漏掉一堆重要功能。如果遇到问题的话，你可以访问: <http://rubyinstaller.org/>。

Windows: 你应该看到的结果

```
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 installed 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>
```

你会看到的提示介面、Ruby 资讯，以及一些其他东西可能非常不一样，不过应该大致上不会差多少。如果你的系统差太多的话，反映给我们，我们会修正过来。

Linux

Linux 系统可谓五花八门，安装软体的方式也各有不同。我们假设作为 Linux 使用者的你应该知道如何安装软体了，以下是给你的操作指示：

1. 用浏览器打开 <http://learnpythonthehardway.org/wiki/ExerciseZero> 下载并安装 gedit 文字编辑器。
2. 把 gedit 放到 Window Manager 明显的位置，以方便之后使用。
 - a. 执行 gedit（也就是你的编辑器），我们要先改掉一些愚蠢的预设值。
 - b. 从 gedit menu 中打开 Preferences，选择 Editor 页面。
 - c. 将 Tab width: 改为 2。
 - d. 选择(确认有勾选到该选项) Insert spaces instead of tabs。
 - e. 然后打开「Automatic indentation」选项。
 - f. 转到 View 页面，打开「Display line numbers」选项。

3. 找到「Terminal」程序。它的名字可能是 GNOME Terminal\、`Konsole\、或者 xterm\。
4. 把 Terminal 也放到 Dock 上。
5. 执行 Terminal，这个程序看上去不怎么地。
6. 在 Terminal 程序里执行 irb。在 Terminal 中执行程序的方式是输入程序的名称然后再敲一下 Return (Enter)。a. 如果你执行 irb 但发现不存在(不认得 irb 这个指令)。请用 Ruby Version Manager (RVM) 安装 Ruby。
7. 敲击 CTRL-D (D) 退出 irb。
8. 这样你就应该能回到敲 irb 前的提示界面了。如果没有的话自己研究一下为什么。
9. 学着使用 Terminal 创建一个目录，你可以上网搜寻怎么做。
10. 学着使用 Terminal 进入一个目录，同样你可以上网搜寻。
11. 使用你的编辑器在你进入的目录下建立一个档案。你将建立一个档案。使用「Save」或者「Save As...」选项，然后选择这个目录。
12. 使用键盘切回到 Terminal 视图，如果不知道怎样使用键盘切换，你一样可以上网搜寻。
13. 回到 Terminal，看看你能不能使用命令列列出你在目录里新建立的档案，在网路上搜寻怎么列出档案夹里的资料。

Linux: 你应该看到的结果

```
$ 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
```

```
$
```

你会看到的提示介面、Ruby 资讯，以及一些其他东西可能非常不一样，不过应该大致上不会差多少。如果你的系统差太多的话，反映给我们，我们会修正过来。

给新手的告诫

你已经完成了这节练习，取决于你对电脑的熟悉程度，这个练习对你而言可能会有些难。如果你觉得有难度的话，你要自己克服困难，多花点时间学习一下。因为如果你不会这些基础操作的话，写程序对你来说将会是相当艰难的一件事。

如果有程序员叫你去使用 vim 或者 emacs，你应该拒绝他们。当你成为一个更好的程序员的时候，这些编辑器才会适合你使用。你现在需要的一个可以编辑文字的编辑器。我们使用 gedit 是因为它很简单，而且在不同的系统上面使用起来也是一样的。就连专业程序员也用 gedit，所以对于初学者而言它已经够用了。

总有一天你会听到有程序员建议你使用 Mac OSX 或者 Linux。如果他喜欢字体美观，他会叫你弄台 Mac OSX 电脑，如果他们喜欢操作控制而且留了一把大胡子，他会叫你安装 Linux。这里再度向你说明，只要是一台手上能用的电脑就够了。你需要的只有三样东西 gedit、一个 Terminal、还有 IRB。

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

最后要说的是这节练习的准备工作目的，也就是让你可以在以后的练习中顺利做到下面的这些事情：

1. 使用 gedit 编写程序代码。
2. 执行你写的练习答案。
3. 修改错误的练习答案。
4. 重复上述步骤。

练习 1 第一个程序

你应该在练习 0 中花了不少的时间，学会了如何安装文字编辑器、执行文字编辑器、以及如何运行 **Terminal**，如果你还没这么做的话，请别继续往前走，之后会有很多苦头的。请不要跳过前一个练习的内容继续前进，这也是本书唯一的一次在练习开头就做这样的警告。

```
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.'
```

将上面的内容写到一个档案中，取名为 `ex1.rb`。注意这样的命名方式，**Ruby** 文件最好以 `.rb` 结尾。

然后你需要在 **Terminal** 中输入以下内容来执行这段程序：`sh ruby ex1.rb`

如果你写对了的话，你应该看到和下面一样的内容。如果不一样，那就是你弄错了什么东西。不是电脑有问题，电脑没问题。

你应该看到的内容

```
$ ruby ex1.rb

Hello World!

Hello Again

I like typing this.

This is fun.

Yay! Printing.

''d much rather you 'not'."

I "said" do not touch this.

$
```


你也许会看到 `$` 前面会显示你所在的目录的名称，这倒是问题，但如果你的输出不一样的话，你需要找出为什么会不一样，然后把你的程序改对。如果你看到类似如下的错误信息：

```
ruby ex1.rb

ex1.rb:4: syntax error, unexpected tCONSTANT, expecting $end

puts "This is fun."

      ^
```

看懂这些内容对你来说是很重要的。因为你以后还会犯类似的错误。就是我现在也会犯这样的错误。让我们一行一行来看。

1. 首先我们在 **Terminal** 输入命令来执行 `ex1.rb` 脚本。
 2. **Ruby** 告诉我们 `ex1.rb` 档案的第 4 行有一个错误。
 3. 然后这一行的内容被打印了出来。
 4. 然后 **Ruby** 打印出了一个[^](插入符号，**caret**) 符号，用来指示错误的位置。
 5. 最后，它打印出了一行「语法错误(**SyntaxError**)」告诉你究竟是发生了怎么样的错误。
- 通常这些错误资讯都非常的难懂，不过你可以把错误资讯的内容复制到搜寻引擎里，然后你就能看到别人也遇到过这样的错误，而且你也许能搞清楚怎样解决这个问题。

加分练习

你还会有加分练习需要完成。加分练习里面的内容是供你尝试的。如果你觉得做不出来，你可以暂时跳过，过段时间再回来做。

在这个练习中，试试这些东西：

1. 让你的脚本再多打印一行。

2. 让你的脚本只打印其中的一行。
3. 在一行的开始位置放置一个 `#` (`octothorpe`) 符号。它的作用是什么？自己研究一下。
4. 从现在开始，除非特殊情况，我将不再解释每个练习的运作原理了。

Note: 井号有很多的英文代称，例如「`octothorpe` (八角帽)」、「`pound` (英镑符号)」、「`hash` (电话的 # 键)」、「`mesh` (网)」。

练习 2 注释和井号

程序里的注释是很重要的。它们可以用自然语言告诉你某段程序代码的功能是什么。在你想要临时移除一段程序代码时，你还可以用注释的方式将这段程序代码临时禁用。接下来的练习将让你学会注释：

```
# 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."
```

你应该看到的结果

```
$ ruby ex2.rb
```

```
I could have code like this.
```

```
This will run.
```

```
$
```

加分练习

1. 弄清楚#符号的作用。而且记住它的名称。（中文为井号，英文为 **octothorpe** 或者 **pound character**）。
2. 打开你的 `ex2.rb` 文件，从后往前逐行检查。从最后一行开始，倒着逐个单词单词检查回去。
3. 有发现更多错误嘛？有的话就改正过来。
4. 阅读你写的练习，把每个字符（**character**）都读出来。有没有发现更多的错误呢？有的话也一样改正过来。

练习 3 数字和数学表达式

每一种程序语言都包含处理数字和进行数学表达式的方法。不必担心，程序员经常撒谎说他们是多们厉害的数学天才，其实他们根本不是。如果他们真是数学天才，他们早就去从事数学相关的行业了，而不是写写广告程序和社交网路游戏，从人们身上偷赚点小钱而已。

这章练习里有很多的数学运算符号。我们来看一遍它们都叫什么。你要一边写一边念出它们的名称来。直到你念烦了为止。名称如下：

+ 加号

- 减号

/ 除号

* 乘号

% 百分比符号

< 小于符号

> 大于符号

<= 小于等于符号

>= 大于等于号

有没有注意到以上只是些符号，没有运算操作呢？写完下面的练习程序代码后，再回到上面的列表，写出每个符号的作用。例如+是用来做加法运算的。

```
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
```

你应该看到的结果

```
$ 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
```

```
$
```

加分练习

1. 使用#在程序代码每一行的前一行为自己写一个注解，说明一下这一行的作用。

2. 记得最开始时的 `IRB` 吧？再次打开 `IRB`，然后使用刚才学到的运算符号，把 Ruby 当做计算机玩玩。
3. 自己找个想要计算的东西，写一个 `.rb` 文件把它计算出来。
4. 有没有发现计算结果是「错」的呢？计算结果只有整数，没有小数部分。研究一下这是为什么，搜寻一下「浮点数(floating point number)」是什么东西。
5. 使用浮点数重写一遍 `ex3.rb`，让它的计算结果更准确(提示：`20.0` 是一个浮点数)。

练习 4：变量和命名

你已经学会了打印出东西和数学运算。下一步你要学的是「变量」。在开发程序中，变量只不过是用来代表某个东西的名称。开发人员通过使用变量名称可以让他们的程序读起来更像英语。而且因为开发人员的记性都不怎样，变量名称可以让他们更容易记住程序的内容。如果他们没有在写程序时使用好的变量名称，在下次读到原来写的程序码时对他们会大为头疼的。

如果你被这章练习难住了的话，记得我们之前教过的：找到不同点、注意细节：

1. 在每一行的上面写一行注释，给自己解释一下这一行的作用。
2. 倒着读你的 `.rb` 文档。
3. 朗读你的 `.rb` 文档，将每个字符也朗读出来。

```
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 `_` 是下划线 (underscore) 符号。你要自己学会怎样打出这个符号来。这个符号在变量里通常被用作假想的空隔，用来隔开单词。

你应该看到的结果

```
$ 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.
```



加分练习

当我刚开始写这个程序时我犯了个错误，Ruby 告诉我这样的错误资讯：

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

用你自己的话解释一下这个错误资讯，解释时记得使用行号，而且要说明原因。

更多的加分练习

1. 解释一下为什么程序里面用了 `4.0` 而不是 `4`。
2. 记住 `4.0` 是一个「浮点数」，自己研究一下这是什么意思。
3. 在每一个变量赋值的上一行加上一行注释。
4. 记住 `=` 的名称是等于 (`equal`)，它的作用是为东西取名。

5. 记住_是下划线(underscore)。
6. 将 IRB 作为计算机跑起来,就跟以前一样,不过这一次在计算过程中使用变量名称来做计算,常见的变量名称有 i、x、j 等等。

练习 5: 更多的变量和打印

我们现在要键入更多的变量并且将它们打印出来,这次我们将使用一个叫「格式化字符串(format string)」的东西,每一次你使用 " 将一些文字包起来,你就建立一个字符串。字符串是程序将信息展示给人的方式。你可以打印他们,可以将它们写入文档,还可以将它们发给网站服务器等等。字符串是很好用的东西,所以在这个练习中你将学会如何创造包含变量内容的字符串,使用专门的格式和语法将变量的内容放到字符串里,相当于来告诉 Ruby: “Hey 这是一个格式化字符串,把这些变量放到那几个位置上”

如常,即使你还不不懂这些内容,只要一字不差的键入就可以了。

```
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]
```

你应该看到的结果

```
$ 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.
```

```
$
```

加分练习

1. 修改所有的变量名称，把它们前面的 `my_` 去掉，确认将每一个地方的都改掉，不只是你使用 `=` 赋值过的地方。
2. 试着使用更多的格式化字符串。
3. 在网路上搜寻所有的 `Ruby` 格式化字符串。
4. 试着使用变量将英寸和磅转换成公分和公斤。不要直接键入答案，使用 `Ruby` 的数学计算来完成。

练习 6：字符串和文字

虽然你已经在程序中写过字符串了，你还没学过它们的用处。在这章练习中我们将使用复杂的字符串来建立一系列的变量，从中你将学到它们的用途。首先我们解释一下字符串是什么东西。

字符串通常是指你想要展示给别人的，或者是你想要从程序里「导出」的一小段字符。`Ruby` 可以通过文字里的双引号"或者是单引号'识别出字符串来。这在你

以前的 `puts` 练习中你已经见过很多次了。如果你把单引号或者双引号括起来的文字放到 `puts` 后面，他们就会被 **Ruby** 打印出来。

字符串可以包含你目前学过的格式化字符串。你只要将格式化的变量放到字符串中，跟着一个百分比符号 **% (percent)**，再紧跟着变量名称即可。唯一要注意的地方，是如果你想要在字符串中通过格式化字符串放入多个变量的结果，你需要将变量放到 `[]` 中括号 **(brackets)** 中，而且变量之间用 **, 逗号 (comma)** 隔开。就像你逛商店时说「我要买牛奶、面包、鸡蛋、汤」一样，只不过程序设计师说的是 `"[milk, eggs, bread, soup]"`。

另一种方式是使用字符串插值 **(string interpolation)** 这种技巧，将变量注入到你的字符串中。方法是使用 `#{} 井号和大括号 (pound and curly brace)`。与其使用这种格式化字符串

```
name1 = "Joe"

name2 = "Mary"

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

我们可以输入：

```
name1 = "Joe"

name2 = "Mary"

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

我们将输入大量的字符串、变量和格式化字符串，并且将它们打印出来。我们还将练习使用简写的变量名称。程序设计师喜欢使用恼人的隐晦简写来节省打字时间，所以我们现在就将提早学会这件事，这样你就能读懂并写出这些东西了。

```
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
```

你应该看到的结果

```
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.
```

加分练习

1. 遍历程序，在每一行的上面写一行注释，给自己解释这一行的作用。
2. 找到所有的「字符串包含字符串」的位置，总共有四个位置。
3. 你确定只有四个位置吗？你怎么知道的？说不定我在骗你呢。

4. 解释一下为什么 w 和 e 用+连起来就可以生成一个更长的字符串。

练习 7：更多印出

现在我们做一些练习，在练习过程中你需要输入一些代码，并让他运行起来，我不会做太多的解释，因为这节的内容都是以前熟悉的。这节练习的目的时巩固你之前学到的东西，我们几轮练习后再见。不要跳过这些练习，不要复制粘贴！

```
1 puts "Mary had a little lamb."
2 puts "Its fleece was white as %s." % 'snow'
3 puts "And everywhere that Mary went."
4 puts "." * 10 # what'd that do?
5
6 end1 = "C"
7 end2 = "h"
8 end3 = "e"
9 end4 = "e"
10 end5 = "s"
11 end6 = "e"
12 end7 = "B"
13 end8 = "u"
```

```
14 end9 = "r"
15 end10 = "g"
16 end11 = "e"
17 end12 = "r"
18
19 # notice how we are using print instead of puts here. change it
   to puts
20
   # and see what happens.
21
   print end1 + end2 + end3 + end4 + end5 + end6
22
   print end7 + end8 + end9 + end10 + end11 + end12
23
24
   # this just is polite use of the terminal, try removing it
25
   puts
```

你应该看到的结果

```
$ ruby ex7.rb
```

```
Mary had a little lamb.
```

```
Its fleece was white as snow.
```

```
And everywhere that Mary went.
```

```
.....
```

```
CheeseBurger
```

```
$
```

加分练习

接下来几节的加分练习时一样的。

1. 逆向阅读，在每一行的上面添加一行注释。
2. 到这阅读出来，找出自己的错误。
3. 从现在开始，把你的错误记录下来，写在一张纸上。
4. 在开始下一节练习时，阅读一遍你记录下来的错误。并尽量避免在下一个练习中犯同样的错误。
5. 记住，每个人都会犯错误。程序员与魔术师一样，他们希望大家任务他们从不犯错误，不过这只是表面的假象而已，他们无时无刻都在犯错。

练习 8：打印，打印

```
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."

]
```

你应该看到的结果

```
$ 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
```

```
I had this thing. That you could type up right. But it didn't  
sing. So I said goodnight.
```

```
$
```

加分练习

1. 自己检查结果，记录你犯过的错误，并且在下一个练习中尽量不犯相同的错误。

练习 9：打印，打印，还是打印

```
# 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
```

你应该看到的结果

```
$ 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.

加分练习

1. 自己检查结果，记录你犯过的错误，并且在下一个练习中尽量不犯相同的错误。

练习 10：那是什么

在练习 9 中我丢给你了一些新东西。我让你看到两种让字符串扩展到多行的方法。第一种方法是在月份中间用 `\n`(back-slash n) 隔开。这两个字符串的作用是在该位置上放入一个「新行(new line)」字符。

使用反斜线 `\`(back-slash) 可以将难打印出的字符放到字符串里。针对不同的符号有很多这样的所谓「跳脱序列(escape sequences)」，但有一个特殊的跳

脱序列，就是\双反斜线(**double back-slash**)。这两个字符组合会打印出一个反斜线来。接下来我们做几个练习，然后你就知道这些跳脱序列的意义了。

另外一种重要的跳脱序列是用来将单引号'和双引号"跳脱。想像你有一个用双引号括起来的字符串，你想要在字符串里的内容里再加入一组双引号进去，比如你想说" I "understand" joke."，**Ruby** 就认为"understand"前后的两个引号是字符串的边界，从而把字符串弄错。你需要一种方法告诉 **Ruby** 字符串里面的双引号不是真正的双引号。

要解决这个问题，你需要将双引号和单引号跳脱，让 **Ruby** 将引号也包含到字符串里面去。这里有一个例子：

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

第二种方法是使用文件语法(**document syntax**)，也就是<<NAME，你可以在键入 NAME 前放入\ 任意多行的文字。接下来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
```

你应该看到的结果

注意你打出来的 **tab** 字符，这节练习中的文字间隔符号对于答案的正确性是很重要的。

```
$ 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



加分练习

1. 上网搜寻一下还有哪些可用的跳脱字符。
2. 结合跳脱序列和格式化字符串，创造一种复杂的格式。

练习 11：提问

我已经出过很多有关于打印的练习，让你习惯写出简单的东西，但简单的东西都有点无聊，我们现在要做的的事是把数据(**data**)读到你的程序里面去。这对你可能会有点难度，你可能一下子搞不明白，不过相信我，无论如何先把练习做了再说。只要做几道练习你就明白了。

一般软体做的事情主要就是下面几件：

1. 接受人的输入。
2. 改变输入值。
3. 打印改变了的值。

到目前为止你只做了打印，但还不会接受或修改人的输入。你还不知道「输入(input)」是什么意思。闲话少说，我们还是开始做点练习看你能不能明白，下一道练习里面我们将会更有详细的解释。

```
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: 注意到我们是使用 `print` 而非 `puts` 吗？`print` 不会自动产生新行。这样你的答案就可以跟问题在同一行了。换句话说，`puts` 会自动产生新行。

你应该看到的结果

```
$ 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. $
```

加分练习

1. 上网搜寻一下 Ruby 的 `gets` 和 `chomp` 的功能是什么？
2. 你能找到 `gets.chomp` 别的用法吗？测试一下你上网找到的例子。
3. 用类似的格式再写一段，把问题改成你自己的问题。

练习 12: 模块 (Module)

看看这段 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`。这是一个 Ruby 中在你所写的脚本中加入其他来源（如：Ruby Gems 或者是你写的其他东西）的功能(**features**) 的方法。与其一次给你所有功能，Ruby 会问你你打算使用什么。这可使你的程序保持轻薄，又可当做之后其他程序设计师阅读你的程序时的参考。

等一下！功能 (Features) 还有另外一个名字

我在这里称呼他们为「功能(**features**)」。但实际上没人这样称呼。我这样做只是取了点巧，使你在学习时先不用理解「行话」。在继续进行之前你得先知道它们的真名 `modules`（模块）。

从现在开始我们将把这些我们 `require` 进来的功能称作 `modules`（模块）。

我会这样说：「你想要 `require open-uri module`。」也有人给它另外一个称呼：「函式库(**libraries**)」。但在这里我们还是先叫它们 `modules`（模块）吧。

加分练习

1. 上网搜寻 `require` 与 `include` 的差异点。它们有什么不同？
2. 你能 `require` 一段没有特别包含 `module` 的脚本吗？
3. 搞懂 Ruby 会去系统的哪里找你 `require` 的 `modules`。

练习 13：参数、解包、参数

在这节练习中，我们将涵盖另外一种将变数传递给脚本的方法（所谓脚本，就是你写的 .rb 文件）。你已经知道，如果要执行 ex13.rb，只要在命令列中执行 `ruby ex13.rb` 就可以了。这句命令中的 `ex13.rb` 部分就是所谓的「参数 (argument)」，我们现在要做的就是写一个可以接受参数的脚本。

将下面的程序写下来，后来我将详细解释

```
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「参数变数(argument variable)」，是一个非常标准的程序术语。在其他的程序语言你也可以看到它全大写的原因是因为它是一个「常数 (constant)」，意思是当它 被赋值之后你就不应该去改变它了。这个变数会接收当你运行 Ruby 脚本时所传入的参数。通过后面的练习你将对它有更多的了解。 你将对它有更多的了解。

第 1 行将 ARGV 「解包(unpack)」，与其将所有参数放到同一个变数下面，我们将每个参数赋予一个变数名称 `first`、`second` 以及 `third`。脚本本身的名称被存在一个特殊变数 `$0` 里，这是我们需要解包的部份。也许看来有些诡异，但「解包」可能是最好的描述方式了。它的涵义很简单：「将 ARGV 中的东西解包，然后将所有的参数依次赋予左边的变数名称」。

接下来就是正常的印出了。

你应该看到的结果

用下面的方法执行你的程序：

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

如果你每次使用不同的参数执行，你将看到下面的结果：

```
$ 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
```

你其实可以将「first」、「2nd」、「3rd」替换成任意三样东西。你可以将它们换成任意你想要的东西。

```
ruby ex13.rb stuff I like
```

```
ruby ex13.rb anything 6 7
```

加分练习

1. 传三个以下的参数给你的脚本。当有缺少参数时哪些数值会被使用到？
2. 再写两个脚本，其中一个接收更少的参数，另一个接收更多的参数。在参数解包时给它们取一些有意义的变数名称。
3. 结合 `gets.chomp` 和 `ARGV` 一起使用，让你的脚本从用户手上得到更多输入。

练习 14: 提示和传递

让我们使用 `ARGV` 和 `gets.chomp` 一起来向使用者提一些特别的问题。下一节练习你将会学习到如何读写文件，这节练习是下节的基础。在这道练习里我们将用一个简单的 `>` 作为提示符号。这和一些游戏中的方法类似，例如 **Zork** 或者 **Adventure** 这两款游戏。

```
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
```

注意到我们将用户提示符号设置为 `prompt`，这样我们就不用每次都要重打一遍了。如果你要将提示符号和修改成别的字串，你只要改一个地方就可以了。

非常顺手吧。

Important: 同时必须注意的是，我们也用了 `STDIN.gets` 取代了 `gets`。这是因为如果有东西在 `ARGV` 里，标准的 `gets` 会认为将第一个参数当成文件而尝试从里面读东西。在要从使用者的输入（如 `stdin`）读取资料的情况下我们必须明确地使用 `STDIN.gets`。

你应该看到的结果

当你执行这个脚本时，记住你需要把你的名字传给这个脚本，让 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.
```

加分练习

1. 查一下 **Zork** 和 **Adventure** 是两个怎样的游戏。看能不能抓到，然后玩玩看。
2. 将 `prompt` 变数改为完全不同的内容再执行一遍。
3. 给你的脚本再新增一个参数，让你的程式用到这个参数。
4. 确认你看懂了我如何结合 `<<SOMETHING` 形式的多行字串与 `# { }` 字串注入做的打印。

练习 15：读取文件

你已经学过了 `STDIN.gets` 和 `ARGV`，这些是你开始学习读取文件的必备基础。

你可能需要多多实验才能明白它的运作原理，所以你要细心练习，并且仔细检查结果。处理文件需要非常仔细，如果不仔细的话，你可能会把有用的文件弄坏或者清空。导致前功尽弃。

这节练习涉及到写两个文件。一个正常的 `ex15.rb` 文件，另外一个

`ex15_sample.txt`，第二个文件并不是脚本，而是供你的脚本读取的文字文件。

以下是后者的内容：

```
This is stuff I typed into a file.
```

```
It is really cool stuff.
```

```
Lots and lots of fun to have in here.
```

我们要做的是把该文件用我们的脚本「打开(`open`)」，然后打印出来。然而把文件名 `ex15_sample.txt`「写死(**Hard Coding**)」在程序代码不是一个好主意，这些数据应该是使用者输入的才对。如果我们碰到其他文件要处理，写死的文件

名就会给你带来麻烦了。解决方案是使用 ARGV 和 STDIN.gets 来从使用者端获取数据，从而知道哪些文件该被处理。

```
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()
```

这个脚本中有一些新奇的玩意，我们来快速地过一遍：

程序代码的 1-3 行使用 ARGV 来获取文件名，这个你已经熟悉了。接下来第 4 行我们使用一个新的命令 **File.open**。现在请在命令列执行 `ri File.open` 来读读它的说明。注意到这多像你的脚本，它接收一个参数，并且传回一个值，你可以将这个值赋予一个变数。这就是你打开文件的过程。

第 6 行我们打印出了一小行，但在第 7 行我们看到了新奇的东西。我们在 `txt` 上引入了一个函式。你从 `open` 获得的东西是一个 `file`（文件），文件本身也支援一些命令。它接受命令的方式是使用句点 **.** (**dot or period**)，紧跟着你的命令，然后参数。就像 `File.open` 做的事一样。差别是：当你说 `txt.read()` 时，你的意思其实是：「嘿 `txt`！执行你的 `read` 命令，无需任何参数！」

脚本剩下的部份基本差不多，不过我就把剩下的分析作为加分练习留给你自己了。

你应该看到的结果

我的脚本叫 “`ex15_sample.txt`”，以下是执行结果：

```
$ 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.
```

```
$
```

加分练习

这节的难度跨越有点大，所以你要尽量做好这节加分练习，然后再继续后面的章节。

1. 在每一行的上面用注释说明这一行的用途。
2. 如果你不确定答案，就问别人，或者是上网搜寻。大部分时候，只要搜寻「`ruby` 你要搜寻的东西」，就能得到你要的答案。比如搜寻一下「`ruby file.open`」。
3. 我使用了「命令」这个词，不过实际上他们的名字是「**函式(function)**」和「**方法(method)**」。上网搜寻一下这两者的意义和区别。看不懂也没关系，迷失在其他程式设计师的知识海洋里是很正常的一件事。
4. 删掉 9-15 行使用到 `STDIN.gets` 的部份，再执行一次脚本。
5. 只用 `STDIN.gets` 撰写这个脚本，想想哪种得到文件名的方法更好，以及为什么。
6. 执行 `ri File` 然后往下滚动直到看见 `read()` 命令（**函式/方法**）。看到很多其他的命令了吧。你可以玩其他试试。
7. 再次启动 `IRB`，然后在里面使用 `File.open` 打开一个文件，这种 `open` 和 `read` 的

方法也值得一学。 8. 让你的脚本针对 `txt` 和 `txt_again` 变数执行一下 `close()`，处理完文件后你需要将其关闭，这是很重要的一点。

练习 16: 读写文件

如果你做了上一个练习的加分练习，你应该已经了解了个种文件相关的命令（方法/函式）。你应该记住的命令如下：

- `close` - 关闭文件。跟你编辑器的文件->储存..是一样的意思。
- `read` - 读取文件内容。你可以把结果赋给一个变量。
- `readline` - 读取文件文字中的一行。
- `truncate` - 清空文件，请小心使用该命令。
- `write(stuff)` - 将 `stuff` 写入文件。

这是你现在应该知道的重要命令。有些命令需要接收参数，但这对我们并不重要。你只要记住 `write` 的用法就可以了。`write` 需要接收一个字符串作为参数，从而将该字符串写入文件。

让我们来使用这些命令做一个简单的文字编辑器吧：

```
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()
```

这是一个大文件，大概是你输入过的最大的文件。所以慢慢来，仔细检查，让它能够跑起来。有一个小技巧就是你可以让你的脚本一部分一部分地跑起来。先写 1-8 行，让它能跑起来，再多做 5 行，再接着几行，以此类推，直到整个脚本都可以跑起来为止。

你应该看到的结果

你将看到两样东西，一样是你新脚本的输出：

```
$ 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.

$
```

这是一个大文件，大概是你输入过的最大的文件。所以慢慢来，仔细检查，让它能够跑起来。有一个小技巧就是你可以让你的脚本一部分一部分地跑起来。先写 1-8 行，让它能跑起来，再多做 5 行，再接着几行，以此类推，直到整个脚本都可以跑起来为止。

加分练习

1. 如果你觉得自己没有弄懂的话，用我们的老方法，在每一行之前加上注释，为自己理清思路。就算不能理清思路，你也可以知道自己究竟具体哪里没弄清楚。
2. 写一个和上一个练习类似的脚本，使用 `read` 和 `ARGV` 读取你刚才新建立的文件。
3. 文件中重复的地方太多了。试着用一个 `target.write()` 将 `line1`, `line2`, `line3` 印出来，你可以使用字符串、格式化字符串以及跳脱字符串。
4. 找出为什么我们打开文件时要使用 `w` 模式，而你真的需要 `target.truncate()` 吗？去看 Ruby 的 `File.open` 函式找答案吧。

练习 17: 更多文件操作

现在让我们再学习几种文件操作。我们将编写一个 Ruby 脚本，将一个文件中的内容拷贝到另一个文件中。这个脚本很短，不过它会让你对于文件操作有更多的了解。

```
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()
```

你应该注意到了我们使用了一个很好用的函式 `File.exists?`。运作原理是将文件名字符串当做一个参数传入，如果文件存在的话，它会传回 `true`，如果不存在的话就传回 `false`。之后在这本书中，我们将会使用这个函式做更多的事情。

你应该看到的结果

如同你前面所写的脚本，运行该脚本需要两个参数，一个是待拷贝的文件位置，一个是要拷贝至的文件位置。如果我们使用以前的 `test.txt` 我们将看到如下的结果：

```
$ 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.
```

```
$
```

该命令对于任何文件应该都是有效的。试试操作一些别的文件看看结果。不过当心别把你的重要文件给弄坏了。

Warning: 你看到我用了 `cat` 这个指令了吧？它只能在 **Linux** 和 **OS** 下使用，**Windows** 用户可以使用 `type` 做到相同效果。

加分练习

1. 再多读读和 `require` 相关的资料，然后将 `IRB` 跑起来，试试 `require` 一些东西看能不能摸出门到。当然，即使搞不清楚也没关系。
2. 这个脚本实在有点烦人。没必要再拷贝之前都问一遍吧，没必要在屏幕上输出那么多东西。试着删掉脚本的一些功能，使它用起来更友善。
3. 看看你能把这个脚本改到多短，我可以把它写成一行。
4. 我使用了一个叫 `cat` 的东西，这个古老的命令用处是将两个文件「连接(`concatenate`)」在一起，不过实际上它最大的用处是印出打文件内容到屏幕是。你可以通过 `man cat` 命令了解到更多资讯。
5. 使用 `Windows` 的人，你们可以给自己找一个 `cat` 的替代品。关于 `man` 的东西就别想太多了。`Windows` 下没这个指令。
6. 找出为什么需要在程式码中写 `output.close()` 的原因。

练习 18: 命名、变量、程序代码、函式

好大的一个标题。接下来我要教你「函式 (`function`)」了！咚咚锵！说到函式，不一样的人会对它有不一样的理解和使用方法，不过我只会教你现在能用到的最简单的使用方式。

函式可以做三件事情：

1. 它们可以给程序代码片段取名，就跟「变量」给字符串和数字命名一样。
2. 它们可以接受参数，就跟你的脚本接受 `ARGV` 一样。
3. 通过使用 `#1` 和 `#2`，他们可以让你创造出「迷你脚本」或者「微命令」。

你可以在 `Ruby` 中使用 `def` 新建函式，我将让你创造四个不同的函式，它们运作起来和你的脚本一样。然后我会示范给你各个函式之间的关系。

```
# 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()
```

让我们把你一个函式 `puts_two` 分解一下, 这个函式和你写脚本的方式差不多, 因此看上去你应该会觉得比较眼熟:

1. 首先我们告诉 **Ruby** 创建一个函式, 使用 `def` 去「定义(define)」一个函式。
2. 紧跟着 `def` 的是函式的名称。本例中它的名称是「`puts_two`」, 但名字可以随便取, 就叫「`peanuts`」也没关系。但函式的名称最好能够表达出它的功能来。
3. 然后我们告诉函式我们需要 `args(asterisk args)`, 这和脚本的 `ARGV` 非常相似, 参数必须放在小括号 `()` 中才能正常运作。
4. 在定义(definition)之后, 后面的行都必须以 2 个空格缩排。其中缩排的第一行的作用是将参数解包, 就像我们在脚本中做的事一样。
5. 为了示范它的运作原理, 我们把解包后的每个参数都印出来。`puts_two` 的问题是它不是建立一个函式最简单的方法。在 **Ruby** 中我们可以直接跳过解包参数的过程直接使用 `()` 里面的名称作为变量名。就像 `puts_two_again` 实现的功能。

接下来的例子是 `print_one`, 它像你示范了函式如何接收单个参数。

最后一个例子是 `print_none`, 它向你示范了函式可以不接收任何参数。

Warning: 如果你不太能看懂上面的内容也别气馁。这是非常重要的。后面我们还有更多的练习向你示范如何创造和使用函式。现在你只要把函式理解成「迷你脚本」就可以了

你应该看到的结果

执行上面的脚本会看到如下结果：

```
$ ruby ex18.rb

arg1: 'Zed', arg2: 'Shaw'

arg1: 'Zed', arg2: 'Shaw'

arg1: 'First!'

I got nothin'.
```

你应该看出来函式是怎样运作的了。注意到函式的用法和你以前见过的 `File.exist?`、`File.open` 以及别的「命令」有点类似了吧？其实我只是为了让你容易礼节才叫他们「命令」。它们的本质其实就是函式。也就是说，你也可以在你自己的脚本中创造你自己的「命令」。

加分练习

为自己写一个函式检查清单以供后续参考。你可以写在一个索引卡片上随时阅读，直到你记住所有的要点为止。注意事项如下：

1. 函式定义是以 `def` 开始的吗？
2. 函式名称是以字符串和底线_组成的吗？
3. 函式名称是不是紧跟着括号 (？
4. 括号里是否包含参数？多个参数是否以逗号隔开？
5. 参数名称是否有重复？（不能使用重复的参数名）
6. 紧跟着参数的最后是否括号)？
7. 紧跟着函式定义的程序代码是否用了 2 个空格的缩排(indent)？
8. 函式结束的位置是不是「`end`」

当你执行（或者说「使用(`use`)」或者「调用(`call`)」一个函数时，记得检查下列几项事情：

1. 调用函式时是否使用了函式的名称？
2. 函式名称是否紧跟着 ()？（非必要，理想性的话应该要加）
3. 参数是否以逗号隔开？
4. 函式是否以) 结尾？

按照这两份检查清单里的内容检查你的练习，直到你不需要检查清单为止。

最后，将下面这句话阅读几遍：

「执行(`run`)函式」、「调用(`call`)函式」和「使用(`use`)函式」是同一个意思。

练习 19：函式和变量

函式这个概念也许承载了太多的数据量。不过别担心，只要坚持做这些练习，对照上个练习中的检查清单检查这次练习的关联，你最终会明白这些内容的。有一个你可能没有注意到的细节，我们现在强调一下，函式里面的变量和脚本里面的变量之间是没有连接的。下面的这个练习可以让你对这一点有更多的思考：

```
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)
```

通过这个练习，你看到我们给我们的函数 `cheese_and_crackers` 很多的参数，然后在函数里把他们打印出来。我们可以填数字、填变量进去函数，我们甚至可以将变量和数学运算结合在一起。

从一方面来说，函数的参数和我们生成变量时用的=赋值符号类似。事实上，如果一个物件你可以用=将其命名，你通常也可以将其作为参数传给一个函数。

你应该看到的结果

你应该研究一下脚本的输出，和你想像的结果对比一下看有什么不同。

```
$ 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.
```



加分练习

1. 倒着将脚本读完，在每一行上面添加一行注解，说明这行程式的作用。
2. 从最后一行开始，倒着阅读每一行，读出所有重要的符号来。
3. 自己边写出至少一个函式出来，然后用十种方法运行这个函式。

Published: October 15 2013

- category:
- ruby 68
- tags:
- ruby 67

- Tweet
-

blog comments powered by Disqus

- « 练习 18: 命名、变量、程序代码、函式
- |
- 练习 20: 函式和文件 »

Back to Top

about

- Eason Han - nbkhhc007@gmail.com
- github.com/easonhan007

- twitter.com/
- [Subscribe to RSS Feed](#)

Theme: the_minimum based on Jekyll-bootstrap.
Powered by Jekyll.

练习 20：函式和文件

回忆一下函式的要点，然后一边做这节练习，一边注意一下函式和文件是如何一起协作发挥作用的。

```
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)
```

特别注意一下，每次运行 `print_a_line` 时，我们是怎样传递当前的行号数据的。

你应该看到的结果

```
$ 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.
```

```
$
```

加分练习

1. 通读脚本，在每行之前加上注解，以理解脚本里发生的事情。
2. 每次 `print_a_line` 运行时，你都传递了一个叫 `current_line` 的变量。在每次调用函数时，打印出 `current_line` 的值，跟踪一下它在 `print_a_line` 中是怎样变成 `line_count` 的。
3. 找出脚本中每一个用到函式的地方。检查 `def` 一行，确认参数没有用错。
4. 上网研究一下 `file` 中的 `seek` 函数是做什么用的。试着运行 `ri file` 看看能不能从 `rdoc` 中学到更多。
5. 研究一下 `+=` 这个简写操作符号的作用，写一个脚本，把这个操作符号用在里边试一下。

练习 21：函式的返回值

你已经学过使用=给变量命名，以及将变量定义为某个数字换字符串。接下来我们将让你见证更多奇迹。我们要示范给你的是如何使用=来将变量设置为「一个函式的值」。有一件事你需要特别注意，但待会再说，先输入下面的脚本吧：

```
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?"
```

现在我们创造了我们自己的加减乘除数学函数：add、subtract、multiply 以及 divide。最重要的是函数的最后一行，例如 add 的最后一行是 `return a + b`，它实现的功能是这样的：

1. 我们调用函数时使用了两个参数：a 和 b。
2. 我们打印出这个函数的功能，这里就是计算加法（**ADDING**）。
3. 接下来我们告诉 **Ruby** 让他做某个回传的动作：我们将 `a+b` 的值返回（**return**）。或者你可以这么说：「我将 a 和 b 加起来，再把结果返回。」
4. **Ruby** 将两个数字相加，然后当函数结束时，它就可以将 `a + b` 的结果赋予给一个变量。

和本书里的很多其他东西一样，你要慢慢消化这些内容，一步一步执行下去，追踪一下究竟发生了什么。为了帮助你理解，本节的加分练习将让你解决一个谜题，并且让你学到点比较酷的东西。

你应该看到的结果

```
$ 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?

\$

加分练习

1. 如果你不是很确定 `return` 回来的值，试着自己写几个函数出来，让它们返回一些值。你可以将任何可以放在=右边的东西作为一个函数的返回值。

2. 这个脚本的结尾是一个谜题。我将一个函式的返回值当作了另外一个函式的参数。我将它们链接到了一起，紧跟写数学等式一样。这样可能有些难读，不过执行一下你就知道结果了。接下来，你需要试试看能不能用正常的方法实现和这个方程式一样的功能。
3. 一旦你解决了这个谜题，试着修改一下函式里的某些部分，然后看会有什么样的结果。你可以有目的地修改它，让它输出另外一个值。
4. 最后，倒过来做一次。写一个简单的等式，使用一样的函式来计算它。

这个练习可能会让你有些头大，不过还是慢慢来，把它当做一个游戏，解决这样的谜题正是写程式的乐趣之一。后面你还会看到类似的小谜题。

Published: October 15 2013

- category:
- ruby 68
- tags:
- ruby 67

- Tweet
-

blog comments powered by Disqus

- « 练习 20: 函式和文件
- |
- 练习 22: 到现在你学到了哪些东西 »

[Back to Top](#)

练习 22: 到现在你学到了哪些东西

这节以及下一节的练习中不会有任何代码，所以也不会有练习答案或者加分练习。其实这节练习可以说是一个巨型的加分练习。我将让你完成一个表格，让你回顾你到现在学到的所有东西。

首先，回到你的每一个练习的脚本里，把你碰到的每一个词和每一个符号（`symbol`, `character` 的别名）写下来。确保你的符号列表是完整的。下一步，在每一个关键词和符号后面写出它的名字，并且说明它的作用。如果你在书里找不到符号的名字，就上网找一下。如果你不知道某个关键字或者符号的作用，就回到用到该符号的章节通读一下，并且在脚本中测试一下这个符号的用处。

你也许会碰到一些横竖找不到答案的东西，只要把这些记在列表里，它可以提示你让你知道还有哪些东西不懂，等下次碰到的时候，你就不会轻易跳过了。等你记住了这份列表中的所有内容，就试着把这份列表默写一遍。如果发现自己漏掉或者无法从「记忆中」回想起某些内容，就回去再记一遍。

Warning: 做这节练习没有失败，只有尝试，请牢记这一点。

你学到的东西

这种记忆练习是枯燥无味的，所以知道它的意义很重要。它会让你明确目标，让你知道你所有努力的目的。

在这节练习中你学会的是各种符号的名称，这样读程序代码这件事对你来说会更加容易。这和学英语时记忆字母表和基本单词的意义是一样的，不同的是 `Ruby` 中会有一些你不熟悉的符号。

慢慢来，别让它成为你的负担。这些符号对你来说应该比较熟悉，所以记住它们应该不是很费力的事情。你可以一次花个 15 分钟，然后休息一下。适度让你的大脑休息一下可以让你学得更快，而且可以让你保持士气。

练习 23：阅读一些程序代码

经过上一周的练习，你应该已经牢记了你的符号列表。现在你需要再花一周的时间，应用这些知识，在网上阅读程序代码。这个任务初看会觉得很难。我将

直接把你丢到深水区呆几天，让你竭尽全力去读懂实实在在的专题里的程序代码。这节练习的目的不是 让你读懂，而是让你学会下面的技能：

1. 找到你需要的 **Ruby** 程序代码。
2. 通读程序代码，找到文件。
3. 尝试理解你找到的程序代码。
4. 以你现在的水准，你还不具备完全理解你找到的程序代码的能力，不过通过接触这些程序代码，你可以熟悉真正的程式专题会是什么样子。

当你做这节练习时，你可以把自己当成是一个人类学家来到了一片陌生的大陆，你只懂得一丁点本地语言，但你需要接触当地人并且生存下去。当然做练习不会碰到生存问题，这毕竟这不是荒野或者丛林。

你要做的事情如下：

1. 使用你的浏览器登录 github.com，搜寻「**ruby**」。
2. 随便找一个专题，然后点进去。
3. 点击 **Source** 标签，浏览目录和文件列表，直到你看到以 **.rb** 结尾的文件
4. 从头开始阅读你找到的程序代码。把它的功能用笔记记下来。
5. 如果你看到一些有趣的符号或者奇怪的字符串，你可以把它们记下来，日后再进行研究。

就是这样，你的任务是使用你目前学到的东西，看自己能不能读懂一些程序代码，看出它们的功能来。你可以先粗略地阅读，然后再细读。也许你还可以试试将难度比较大的部分一字不漏地朗读出来。

现在再试试其它的几个站：

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

在这些网站你可能还会看到以 **.c** 结尾的奇怪文件，不过你只需要看 **.rb** 结尾的文件就可以了。

最后一个有趣的事情是你可以在这四个网站搜索「**ruby**」以外的你感兴趣的话题，例如你可以搜索「**journalism**（新闻）」，「**cooking**（下厨）」，「**physics**

（物理）」，或者任何你感兴趣的话题。你也许会找到一些对你有用的，且可以直接拿来用的程序代码。

练习 24: 更多练习

你离这本书第一部分的结尾已经不远了，你应该已经具备了足够的 Ruby 基础知识，可以继续学习一些程序的原理了，但你应该做更多的练习。这个练习的内容比较长，它的目的是锻炼你的毅力，下一个练习也差不多是这样的，好好完成它们，做到完全正确，记得仔细检查。

```
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)
```

你应该看到的结果

```
$ 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.
```

```
$
```

加分练习

1. 记得仔细检查结果，从后往前倒着检查，把程序代码朗读出来，在不清楚的位置加上注释。
2. 故意将程序码改烂，执行并检查会发生什么样的错误，并且确认你有能力改正这些错误。

练习 25：更多更多的练习

我们将做一些关于函式和变量的练习，以确认你真正掌握了这些知识。这节练习对你来说可以说是一本道：写程序，逐行研究，弄懂它。不过这节练习还是有些不同，你不需要执行它，取而代之，你需要将它导入到 **Ruby** 通过自己执行函式的方式运行。

```
module Ex25

  def self.break_words(stuff)

    # This function will break up words for us.

    words = stuff.split(' ')

    words

  end

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` 运行，找出里面的错误，并把它们都改正过来。然后你需要接着下面的答案章节完成这节练习。

你应该看到的结果

这节练习我们将在你之前用来做算术的 **Ruby 编译器(IRB)**里，用交互的方式和你的 `.rb` 作交流。

这是我做出来的样子：

```
$ 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
```

```
$
```

我们来逐行分析一下每一步实现的是什么：

1. 在第 2 行你 `require` 了自己的 `./ex25.rb` Ruby 文件，和你做过的其他 `require` 一样 `$`。在 `require` 的时候你不需要加 `.rb` 后缀。这个过程里，你将这个文件当做了 `module` (模块) 来使用，你在这个模块里定义的函式也可以直接引用出来。
2. 第 4 行你创造了一个用来处理的 `sentence` (句子)。
3. 第 6 行你使用了 `Ex25` 模块引用了你的第一个函式 `Ex25.break_words`。其中的 `.` (dot, period) 符号可以告诉 Ruby: 「Hi, 我要执行 `Ex25` 里的那个叫 `break_word` 的函式!」
4. 第 8 行我们只是输入 `Ex25.sort_words` 来得到一个排序过的句子。
5. 10-15 行我们使用 `Ex25.print_first_word` 和 `Ex25.print_last_word` 将第一个和最后一个词打印出来。
6. 第 16 行比较有趣。我把 `words` 变量写错成了 `wrods`，所以 Ruby 在 17-18 行给了一个错误信息。
7. 第 19-20 行我们打印出了修改过后的词汇列表。第一个和最后一个词我们已经打印过了，所以在这里没有再打印出来。
8. 剩下的行你需要自己分析一下，就留作你的加分练习了。

加分练习

1. 研究答案中没有分析过的行，找出它们的来龙去脉。确认自己明白了自己使用的是模块 `Ex25` 中定义的函式。
2. 我们将我们的函式放在一个 `module` 里是因为他们拥有自己的 命名空间 (`namespace`)。这样如果有其他人写了一个函式也叫 `break_words`，这样就不会发生碰创。无论如何，输入 `Ex25.` 是一件很烦人的事。有一个比较方便的作法，你可以输入 `include Ex25`,

这相当于说：「我要将所有 `Ex25` 这个 `mudle` 里的所有东西 `include` 到我现在的 `module` 里。」

3. 试着在你正在使用 `IRB` 时，弄烂文件会发生什么事。你可能要执行 `CTRL-D` (`Windows` 下是 `CTRL-Z`) 才能把 `IRB` 关掉 `reload` 一次。

练习 26: 恭喜你，现在来考试了！

你已经差不多完成这本书的前半部分了，不过后半部分才是更有趣的。你将学到逻辑，并通过条件判断实现有用的功能。

在你继续学习之前，你有一道试题要做。这道试题很难，因为它需要你修正别人写的代码。当你成为程序员以后，你将需要经常面对别的程序员的代码，也许还有他们的傲慢态度，他们会经常说自己的代码是完美的。

这样的程序员是自以为是不在乎别人的蠢货。优秀的科学家会对他们自己的工作持怀疑态度，同样，优秀的程序员也会认为自己的代码总有出错的可能，他们会先假设是自己的代码有问题，然后用排除法清查所有可能是自己有问题的地方，最后才会得出「这是别人的错误」这样的结论。

在这节练习中，你将面对一个程度糟糕的程序员，并改好他的代码。我将练习 24 和 25 胡乱拷贝到了一个文件中，随机地删掉了一些字，然后添加了一些错误进去。大部分的错误是 `Ruby` 在执行时会告诉你的，还有一些算术错误是你要自己找出来的。再剩下下来的就是格式和拼写错误了

所有这些错误都是程序员很容易犯的，就算有经验的程序员也不例外。

你的任务是将此文件修改正确，用你所有的技能改进这个脚本。你可以先分析这个文件，或者你还可以把它像学期论文一样打印出来，修正里边的每一个缺陷，重复修正和运行的动作，直到这个脚本可以完美地运行起来。在整个过程中不要寻求帮助，如果你卡在某个地方无法进行下去，那就休息一会晚点再做。

就算你需要几天才能完成，也不要放弃，直到完全改对为止。

最后要说的是，这个练习的目的不是写程序，而是修正现有的程序，练习放在下面的网站：

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

从那里把代码复制过来，命名为 `ex26.rb`，这也是本书唯一一处允许你复制贴上的地方。

练习 27：记住逻辑关系

到此为止你已经学会了读写文件，命令行处理，以及很多 `Ruby` 数学运算功能。

今天，你将要开始学习逻辑了。你要学习的不是研究院里的高深逻辑理论，只是程序员每天都用到的让程序跑起来的基础逻辑知识。

学习逻辑之前你需要先记住一些东西。这个练习我要求你一个星期完成，不要擅自修改 `schedule`，就算你烦得不得了，也要坚持下去。这个练习会让你背下来一系列的逻辑表格，这会让你更容易地完成后面的练习。

需要事先警告你的是：这件事情一开始一点乐趣都没有，你会一开始就觉得它很无聊乏味，但它的目的是教你程序员必须的一个重要技能 — 一些重要的概念是必须记住的，一旦你明白了这些概念，你会获得相当的成就感，但是一开始你会觉得它们很难掌握，就跟和乌贼摔跤一样，而等到某一天，你会刷的一下豁然开朗。你会从这些基础的记忆学习中得到丰厚的回报。

这里告诉你一个记住某样东西，而不让自己抓狂的方法：在一整天里，每次记忆一小部分，把你最需要加强的部分标记起来。不要想着在两小时内连续不停地背诵，这不会有什么好的结果。不管你花多长时间，你的大脑也只会留住你在前 15 或者 30 分钟内看过的东西。

取而代之，你需要做的是创建一些索引卡片，卡片有两列内容，正面写下逻辑关系，反面写下答案。你需要做到的结果是：拿出一张卡片来，看到正面的表达式，

例如「**True or False**」，你可以立即说出背面的结果是「**True**」！坚持练习，直到你能做到这一点为止。

一旦你能做到这一点了，接下来你需要每天晚上自己在笔记本上写一份真值表出来。不要只是抄写它们，试着默写真值表，如果发现哪里没记住的话，就飞快地撇一眼这里的答案。这样将训练你的大脑让它记住整个真值表。

不要在这上面花超过一周的时间，因为你在后面的应用过程中还会继续学习它们。

逻辑术语

在 **Ruby** 中我们会用到下面的术语（符号或者词汇）来定义事物的真(**True**)或者假(**False**)。电脑的逻辑就是在程序的某个位置检查这些符号或者变数组合在一起表达的结果是真是假。

- **and** 和
- **or** 或
- **not** 非
- **!=** (**not equal**) 不等于
- **==** (**equal**) 等于
- **>=** (**greater-than-equal**) 大于等于
- **<=** (**less-than-equal**) 小于等于
- **true** 真
- **false** 假

其实你已经见过这些符号了，但这些词汇你可能还没见过。这些词汇(**and**, **or**, **not**)和你期望的效果其实是一样的，跟英语里的意思一模一样。

真值表

我们将使用这些符号来创建你需要记住的真值表。

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 (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

现在使用这些表格创建你自己的卡片，再花一个星期慢慢记住它们。记住一点，这本书不会要求你成功或者失败，只要每天尽力去学，在尽力的基础上多花一点功夫就可以了。

练习 28: 布尔（Boolean）表示式练习

上一节你学到的逻辑组合的正式名称是「布尔逻辑表示式(boolean logic expression)」。在程序中，布尔逻辑可以说是无处不在。它们是电脑运算的基础和重要组成部分，掌握它们就跟学音乐掌握音阶一样重要。

在这节练习中，你将在 **IRB** 里使用到上节学到的逻辑表示式。先为下面的每一个逻辑问题写出你认为的答案，每一题的答案要嘛为 **True** 要嘛为 **False**。写完以后，你需要将 **IRB** 运行起来，把这些逻辑语句输入进去，确认你写的答案是否正确。

```
true and true
```

```
false and true
```

```
1 == 1 and 2 == 1
```

```
"test" == "test"
```

```
1 == 1 or 2 != 1
```

```
true and 1 == 1
```

```
false and 0 != 0
```

```
true or 1 == 1
```

```
"test" == "testing"
```

```
1 != 0 and 2 == 1

"test" != "testing"

"test" == 1

not (true and false)

not (1 == 1 and 0 != 1)

not (10 == 1 or 1000 == 1000)

not (1 != 10 or 3 == 4)

not ("testing" == "testing" and "Zed" == "Cool Guy")

1 == 1 and not ("testing" == 1 or 1 == 0)

"chunky" == "bacon" and not (3 == 4 or 3 == 3)

3 == 3 and not ("testing" == "testing" or "Ruby" == "Fun")
```

在本节结尾的地方我会给你一个理清复杂逻辑的技巧。

所有的布尔逻辑式都可以用下面的简单流程得到结果：

1. 找到相等判断的部分 (`==` or `!=`)，将其改写为其最终值(`True` 或 `False`)。
2. 找到括号里的 `and/or`，先算出它们的值。
3. 找到每一个 `not`，算出他们反过来的值。
4. 找到剩下的 `and/or`，解出它们的值。
5. 等你都做完后，剩下的结果应该就是 `True` 或者 `False` 了。

下面我们以 #20 逻辑式示范一下：

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

接下来你将看到这个复杂表达式是如何逐级解析为一个单独结果的：

1. 出每一个等值判断：

- `3 != 4` 为 `True`: `true and not ("testing" != "test" or "Ruby" == "Ruby")`
- `"testing" != "test"` 为 `True`: `true and not (true or "Ruby" == "Ruby")`
- `"Ruby" == "Ruby"`: `true and not (true or true)`

1. 找到 `()` 中的每一个 `and/or` :

- `(true or true)` is `True`: `true and not (true)`

1. 找到每一个 `not` 并将其逆转：

- `not (true)` is `False`: `true and false`

1. 找到剩下的 `and/or`，解出它们的值：

- `true and false` is `False`

这样我们就解出了它最终的值为 `False` 。

布尔 (**Boolean**) 表示式练习 **Warning**: 杂的逻辑表达式一开始看上去可能会让你觉得很难。而且你也许已经碰壁过了，不过别灰心，这些「逻辑体操」式的训练只是让你逐渐习惯起来，这样后面你可以轻易应对程序里边更酷的一些东西。只要你坚持下去，不放过自己做错的地方就行了。如果你暂时不太能理解也没关系，弄懂的时候总会到来的。

你应该看到的结果

以下内容是在你自己猜测结果以后，通过和 **IRB** 对话得到的结果：

```
$ irb

ruby-1.9.2-p180 :001 > true and true

=> true

ruby-1.9.2-p180 :002 > 1 == 1 and 2 == 2

=> true
```

加分练习

1. Ruby 里还有很多和 `!=`、`==` 类似的操作符号。试着尽可能多的列出 Ruby 中的「等价运算符」。例如 `<` 或是 `<=`。
2. 写出每一个等价运算符的名称。例如 `!=` 叫「`not equal`（不等于）」。
3. 在 **IRB** 里测试新的布尔逻辑式。在敲 **Enter** 前你需要喊出它的结果。不要思考，凭自己的第一直觉就可以了。把表达式和结果用笔写下来再敲 **Enter**，最后看自己做对多少，做错多少。
4. 把练习 3 那张纸丢掉，以后你不再需要查询它了。

练习 29：如果（if）

这里是你接下要写的作业，这段介绍了 `if-statement`（**if** 语句）。把这段输入进去，让它能够正确执行。然后我们看看你是否有收获。

```
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
```

你应该看到的结果

```
$ 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.
```

```
$
```

加分练习

猜猜「`if` 语句」是什么，它有什么用处。在做下一道练习前，试着用自己的话回答下面的问题：

1. 你认为 `if` 对于它下一行的程序码做了什么？
2. 把练习 29 中的其它布尔表示式放到「`if` 语句」中会不会也可以运行呢？试一下。
3. 如果把变量 `people`、`cats` 和 `dogs` 的初始值改掉，会发生什么事情？

练习 30: Else 和 If

前一练习中你写了一些「**if** 语句 (**if-statements**)」，并且试图猜出它们是什么，以及实现的是什么功能。在你继续学习之前，我给你解释一下上一节的加分练习的答案。上一节的加分练习你做过了吧，有没有？

1. 你认为 **if** 对于它下一行的代码做了什么？**if** 语句为代码创建了一个所谓的「分支 (**branch**)」，就跟 RPG 游戏中的情节分支一样。**if** 语句告诉你的脚本：「如果这个布尔表示式为真，就执行接下来的代码，否则就跳过这一段。」
2. 把练习 29 中的其它布尔表示式放到 **if** 语句中会不会也可以执行呢？试一下。可以。而且不管多复杂都可以，虽然写复杂的东西通常是一种不好的写作风格。
3. 如果把变数 **people**、**cats** 和 **dogs** 的初始值改掉，会发生什么事情？因为你比较的对象是数字，如果你把这些数字改掉的话，某些位置的 **if** 语句会被演绎为 **True**，而它下面的代码块将被运行。你可以试着修改这些数字，然后在头脑里假想一下那一段代码会被运行。

把我的答案和你的答案比较一下，确认自己真正懂得代码「块(**block**)」的含义。这点对于你下一节的练习很重要，因为你将会写很多的 **if** 语句。

把这一段写下来，并让它运行起来：

```
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
```

你应该看到的结果

```
$ ruby ex30.rb
```

```
We should take the cars.
```

```
Maybe we could take the buses.
```

```
Alright, let's just take the buses.
```

```
$
```

加分练习

1. 猜想一下 `elsif` 和 `else` 的功能。
2. 将 `cars`、`people` 和 `buses` 的数量改掉，然后追溯每一个 `if` 语句。看看最后会打印出什么来。
3. 试着写一些复杂的布尔表示式，例如 `cars > people and buses < cars`。在每一行的上面写注解，说明这一行的功用。

练习 31：做出决定

这本书的上半部分，你打印出了一些东西，并且调用了函式，不过一切都是直线式进行的。你的脚本从最上面一行开始，一路运行到结束，但其中并没有决定程序流向的分支点。现在你已经学会了 `if`、`else` 和 `elsif`，你就可以开始建立包含条件判断的脚本了。上一个脚本中你写了一系列的简单提问测试。这节的脚本中，你将需要向使用者提问，依据使用者的答案来做出决定。把脚本写下来，多多捣鼓一阵子，看看他的运作原理是什么。

```
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
```

这里的重点是你可以在 `if` 语句中内部再放一个 `if` 语句。这是一个很强大的功能，可以用来建立「巢状(nested)」的决定 (decision)。

你需要理解 `if` 语句包含 `if` 语句的概念。做一下加分练习，这样你会确信自己真正理解了它们。

你应该看到的结果

我在玩一个小冒险游戏。我的水准不怎么样。

```
$ 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.
```

加分练习

为游戏添加新的部分，改变玩家做决定的位置。尽自己能力扩充这个游戏，不过别把游戏弄得太诡异了。

练习 32：循环和数组

现在你应该有能力写更有趣的程序出来了。如果你能够一直跟得上，你应该已意识到你能将之前学到的将 `if` 语句 和 「布尔表示式」这些东西结合起来，让程序做出一些聪明的事了。

然而，我们的城市还需要能很快地完成重复的事情。这节练习中我们将使用 `for-loop`(`for` 循环) 来建立和打印出各式的数组。在做练习的过程中，你会逐渐搞懂它们是怎么回事。现在我不会告诉你，你需要自己找到答案。

在你开始使用 `for` 循环之前，你需要在某个位置存放循环的结果。最后的方法是使用数组 `array`。一个数组，就是一个按照顺序存放东西的容器。数组并不复杂，你只是要学习一点新的语法。首先我们来看看如何建立一个数组：

```
hairs = ['brown', 'blond', 'red']
```

```
eyes = ['brown', 'blue', 'green']
```

```
weights = [1, 2, 3, 4]
```

你要做的是以[左中括号开头「打开」数组，然后写下你要放入数组的东西、用逗号, 隔开，就跟函式的参数一样，最后你需要用]右中括号结束数组的定义。

然后 Ruby 接收这个数组以及里面所有的内容，将其赋予给一个变数。

Warning: 对于不会写程序的人来说这是一个困难点。习惯性思维告诉你的大脑大地是平的。记得上一个练习中的巢状 `if` 语句吧，你可能觉得要理解它有些难度，因为生活中一般人不会去想这样的问题，但这样的问题在程序中几乎到处都是。你会看到一个函式调用用另外一个包含 `if` 语句的函式，其中又有巢状数组的数组。如果你看到这样的东西一时无法弄懂，就用纸笔记下来，手动分割下去，直到弄懂为止。

现在我们将使用循环建立一些数组，然后将它们打印出来：

```
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
```

你应该看到的结果

```
$ 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
```

```
$
```

加分练习

1. 注意一下 `range (0..5)`。查一下 `Range class` (类别) 并弄懂它。
2. 在第 24 行, 你可以直接将 `elements` 赋值为 `(0..5)`, 而不需使用 `for` 循环吗?
3. 在 `Ruby` 文件中可以找到关于数组的内容, 仔细阅读一下, 除了 `push` 以外, 数组还支持那些操作?

练习 33: While 循环

接下来是一个更在你意料之外的概念: `while-loop`(**while** 循环)。while 循环会一直执行它下面的代码块, 直到它对应的布尔表示式为 `false` 才会停下来。

等等，你还能跟的上这些术语吧？如果我们写了这样一个语句：`if items > 5` 或者是 `for fruit in fruits`，那就是在开始一个代码块 (**code block**)，新的代码块是需要被缩排的，最后再以 `end` 语句结尾。只有将代码用这样的方式格式化，**Ruby** 才能知道你的目的。如果你不太明白这一点，就回去看看「**if** 语句」、「**函式**」和「**for** 循环」章节，直到你明白为止。

接下来的练习将训练你的大脑去阅读这些结构化的与集，这和我们布尔表示式刻录到你的大脑中的过程有点类似。

回到 `while` 循环，它所作的和 `if` 语句类似，也是去检查一个布尔表示式的真假，不一样的是它下面的代码块不是只被执行一次，而是执行完后再次回到 `while` 所在的位置，如此重复进行，直到 `while` 表示式为 `false` 为止。

`while` 循环有一个问题：那就是有时它会永不结束。如果你的目的是循环到宇宙毁灭为止，那这样也挺好的，不过其他的情况下你的回总需要有一个结束点。

为了避免这样的问题，你需要遵循下面的规定：

1. 尽量少用 `while` 循环，大部分时候 `for` 循环是更好的选择。
2. 重复检查你的 `while` 语句，确定你测试的布尔表示式最终会变成 `false`。
3. 如果不确定，就在 `while` 循环的结尾印出你要测试的值。看看它的变化。

在这节练习中，你将通过上面的三样事情学会 `while` 循环：

```
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
```

你应该看到的结果

```
$ 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
```

加分练习

1. 将这个 `while` 循环改成一个函数，将测试条件 (`i < 6`) 中的 `6` 换成一个变量。
2. 使用这个函数重写你的脚本，并使用不同的数字进行测试。
3. 为函数添加另一个参数，这个参数用来定义第 8 行的 `+1`，这样你就可以让它任意加值了。
4. 再使用该函数重写一遍这个脚本。看看效果如何。
5. 接下来使用 `for` 循环和 `range` 把这个脚本再写一遍。你还需要中间的加值操作吗？如果你不去掉它，会有什么样的结果？

有可能你会碰到程序跑着停不下来了，这时你只要按着 `CTRL` 再敲 `c` (`CTRL-c`)，这样程式就会中断下来了。

练习 34：存取数组里的元素

数组非常有用，但只有你存取里面的内容时，它才能发挥出作用来。你已经学会了按顺序读出数组中的内容，但如果你要得到第 5 个元素该怎么办呢？你需要知道如何存取数组中的元素。存取第一个元素的方法是这样的：

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

bear = animals[0]
```

你定义一个 `animals` 的数组，然后你用 0 来存取第一个元素！？这是怎么回事啊？因为数学里面就是这样，所以 `Ruby` 的数组也是从 0 开始的。虽然看起来很奇怪，这样定义其实有它的好处。

最好的解释方式是将你平常使用数字的方式和开发人员使用数字的方式做比较。

假设你在观看上面数组中的四种动物：(['bear', 'tiger', 'penguin', 'zebra']) 的赛跑。而它们比赛的名次正好跟数组中的顺序一样。这是一场很刺激的比赛，因为这些动物没打算吃掉对方，而且比赛还真的举办起来了。结果你的朋友来晚了，他想知道谁赢了比赛，他会问你「嘿，谁是第 0 名？」吗？不会的，他会问「嘿，谁是第 1 名？」

这是因为动物的次序是很重要的。没有第一个就没有第二个，没有第二个话也不会有第三个。第零个是不存在的，因为零的意思是什么都没有。「什么都没有」怎么赢比赛嘛？完全不和逻辑。这样的数字我们称之为「序数(ordinal number)」

而开发人员不能用这种方式思考问题，因为他们可以从数组中的任何一个位置取出一个元素来。对开发人员来说，上述的数组更像是一叠卡片。如果他们想要 `tiger`，就抓它出来，如果想要 `zebra`，也一样抓出来。要随机地抓数组里的内容，数组的每一个元素都应该要有一个地址(address)，或者一个「索引(index)」，而最好的方式就是使用以 0 开头的索引。相信我说的这一点吧，这种方式获取元素会更容易。而这一类的数字被称为「基数(cardinal number)」，它意味着你可以任意抓取元素，所以我们需要一个 0 号元素。

那么，这些知识对你的数组操作有什么帮助呢？很简单，每次你对自己说：「我要第 3 只动物」时，你需要将「序数」转换成「基数」，只要将前者减 1 就可以了。第 3 只动物的索引是 2，也就是 `penguin`。由于你一辈子都在跟序数打交道，所以你需要这种方式来获得基数，只要减 1 就都搞定了。

记住：`ordinal ==` 有序，以 1 开始；`cardinal ==` 随机存取，以 0 开始。

让我们练习一下。定义一个动物列表，然后跟着做后面的练习，你需要写出所指位置的动物名称。如果我用的是「`first`」、「`second`」等说法。那说明我用的是叙述，所以你需要减去 1。如果我给你的是基数 (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.

对于上述某一条，以这样的格式写出一个完整的句子：「The 1st animal is at 0 and is a bear.」然后倒过来念「“The animal at 0 is the 1st animal and is a bear.”」

使用 `IRB` 去检查你的答按。

Hint: Ruby 还有一些便利的 `method` 是属于在数组中存取特定元素的用法。：

`animals.first` 和 `animals.last`。

加分练习

1. 上网搜索一下关于 序数 (ordinal number) 和基数 (cardinal number) 的知识并阅读一下。

2. 以你对于这些数字类型的了解，解释一下为什么今年是 2010 年。呢是：你不能随便挑选年份。
3. 再写一些数组，用一样的方式做出索引，确认自己可以在两种数字之间互相翻译。
4. 使用 IRB 检查自己的答按。

Warning: 会有开发人员告诉你，叫你去阅读一个叫「Dijkstra」的人写的关于数字的主题。我建议你还是不读为妙，除非你喜欢听一个在写程序这一行刚兴起时就停止了从事写程序工作的人对你大吼大叫。

练习 35: 分支 (Branches) 和函式 (Functions)

你已经学会了 `if` 语句、函式、还有数组。现在你要练习扭转一下思维了。把下面的代码写下来，看你是否能看懂它实现的是什么功能。

```
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

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

end

start()
```

你应该看到的结果

你可以结果：

```
$ 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!

\$

加分练习

1. 把这个游戏的地图画出来，把自己的路线也画出来。
2. 改正你所有的错误，包括拼写错误。
3. 为你不懂的函式写注解。记得 RDoc 中的注释吗？
4. 为游戏添加更多元素。通过怎样的方式可以简化并且扩充游戏的功能呢？

5. 这个 `gold_room` 游戏使用了奇怪的方式让你输入一个数字。这种方式会导致什么样的 `bug`? 你可以用比检查 `0`、`1` 更好的方式判断输入是否是数字吗? `to_i()` 这个函式可以给你一些头绪。

练习 36: 设计和测试

现在你已经学会了「`if` 语句」，我将给你一些使用 `for` 循环和 `while` 循环的规则，一面你日后碰到麻烦。我还会教你一些测试的小技巧，以便你能发现自己程式的问题。最后，你将需要设计一个和上节类似的小游戏，不过内容略有更改。

If 语句的规则

1. 每一个「`if` 语句」必伴随须一个 `else`。
2. 如果这个 `else` 因为没有意义，而永远都没被执行到，那你必须在 `else` 语句后面使用一个叫 `die` 的函式，让它印出错误并死给你看，这和上一节的练习类似，这样你可以找到很多的错误。
3. 千万不要使用超过两层的 `if` 语句，最好尽量保持只有 `1` 层。那你就需要把第二个 `if` 移到另一个函式里面。
4. 将 `if` 语句当做段落来对待，其中的每一个 `if`、`elsif`、`else` 组合就跟 一个段落的句子组合一样。在这种组合的最前面和最后面留一个空行以作区分。
5. 你的布尔测试应该很简单，如果它们很复杂的话，你需要将它们的运算式先放到一个变量里，并且为变量取一个好名字。

如果你遵循上面的规则，你就会写出比大部分开发人员都好的代码来。回到上一个练习中，看看我有没有遵循这些规则，如果没有的话，就将其改正过来。

Warning: 在日常写程式中不要成为这些规则的奴隶。在训练中，你需要通过这些规则的应用来巩固你学到的知识，而在实际写程式中这些规则有时其实很蠢。如果你觉得哪个规则很蠢，就别使用它。

Rules For Loops

1. 只有在循环循环永不停止时使用 `while` 循环，这意味着你可能永远都用不到。这条只有 `Ruby` 中成立，其他的语言另当别论。
2. 其他类型的循环都使用 `for` 循环，尤其是在循环的对象数量固定或者有限的情况下。

除错（Debug）的小技巧

1. 不要使用「`debugger`」。Debugger 所作的相当于对病人的全身扫描。你并不会得到某方面的有用信息，而且你会发现它输出的信息太多，而且大部分没有用，或者只会让你更困惑。
2. 最好的除错技巧是使用 `puts` 或 `p` 在各个你想要检查的关键环节将关键变量印出来，从而检查哪里是否有错。
3. 让程式一部分一部分地运行起来。不要等一个很长的脚本写完后才去运行它。写一点，运行一点，再修改一点。

家庭作业

写一个和上节练习类似的游戏。同类的任何题材的游戏都可以，花一个星期让它尽可能有趣一些。作为加分练习，你可以尽量多使用数组、函式、以及模块（记得练习 13 吗？），而且尽量多弄一些新的 `Ruby` 程式让你的游戏跑起来。

过有一点需要注意，你应该把游戏的设计先写出来。在你开始写代码之前，你应该设计出游戏的地图，创建出玩家会碰到的房间、怪物、以及陷阱等环节。

一旦搞定了地图，你就可以写写代码了。如果你发现地图有问题，就调整一下地图，让写代码和地图互相符合。

最后一个建议：每一个开发人员在开始一个新的大项目时，都会被非理性的恐惧影响到。为了避免这种恐惧，他们会拖延时间，到最后一事无成。我有时会这样，

每个人都会有这样的经历，避免这种情况的最好的方法是把自己要做的事情列出来，一次完成一样。

开始做吧。先做一个小一点的版本，扩充它让它变大，把自己要完成的事情一一列出来，然后逐个完成就可以了。

练习 37：复习各种符号

现在该复习你学过的符号和 **Ruby** 关键字了，而且你在本节还会学到一些新的东西。我在这里所作的是将所有的 **Ruby** 符号和关键字列出来，这些都是值得掌握的重点。

在这节课中，你需要复习每一个关键字，从记忆中想起它的作用并且写下来，接着上网搜索它真正的功能。有些内容可能是无法搜索的，所以这对你可能有些难度，不过你还是需要坚持尝试。

如果你发现记忆中的内容有误，就在索引卡片上写下正确的定义，试着将自己的记忆纠正过来。如果你就是不知道它的定义，就把它也直接写下来，以后再做研究。

最后，将每一种符号和关键字用在程序里，你可以用一个小程序来做，也可以尽量多写一些程序来巩固记忆。这里的关键点是明白各个符号的作用，确认自己没搞错，如果搞错了就纠正过来，然后将其用在程序里，并且通过这样的方式巩固自己的记忆。

Keywords（关键字）

`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
```

资料类型

针对每一种资料类型，都举出一些例子来，例如针对 **string**，你可以举出一些字。针对 **number**，你可以举出一些数字。

```
true
```

```
false
```

```
nil
```

```
constants
```

```
strings
```

```
numbers
```

```
ranges
```

```
arrays
```

```
hashes
```

字符串格式(String Formats)

一样的，在字符串中使用它们，确认它们的功能。

`\\`

`\'`

`\"`

`\a`

`\b`

`\f`

`\n`

`\r`

`\t`

`\v`

Operators

有些操作符号你可能还不熟悉，不过还是一一看过去，研究一下它们的功能，如果你研究不出来也没关系，记录下来日后解决。

::

[]

**

-(unary)

+(unary)

!

~

*

/

%

+

-

<<

>>

&

|

>

>=

<

<=

<=>

==

===

!=

=~

!~

&&

||

..

...

花一个星期学习这些东西，如果你能提前完成就更好了。我们的目的是复盖到所有的符号类型，确认你已经牢牢记住它们。另外很重要的一点是这样你可以找出自己还不知道哪些东西，为自己日后学习找到一些方向。

练习 38：阅读代码

现在去找一些 **Ruby** 代码阅读一下。你需要自己找代码，然后从中学习一些东西。你学到的东西已经足够让你看懂一些代码了，但你可能还无法理解这些代码的功能。这节课我要教给你的是：如何运用你学到的东西理解别人的代码。

首先把你想要理解的代码打印到纸上。没错，你需要打印出来，因为和屏幕输出相比，你的眼睛和大脑更习惯于接受纸质打印的内容。一次最多打印几页就可以了。

然后通读你列打印出来的代码并做好标记，标记的内容包括以下几个方面：

1. 函数以及函数的功能。
2. 每个变量的初始赋值。
3. 每个在程式的各个部分中多次出现的变量。它们以后可能会给你带来麻烦。
4. 任何不包含 **else** 的 **if** 语句。它们是正确的吗？
5. 任何可能没有结束点的 **while** 循环。
6. 最后一条，代码中任何你看不懂的部分都记下来。

接下来你需要通过注解的方式向自己解释代码的含义。解释各个函式的使用方法，各个变量的用途，以及任何其它方面的内容，只要能帮助你理解代码即可。

最后，在代码中比较难的各个部分，逐行或者逐个函式跟踪变量值。你可以再打印一份出来，在空白处写出你要「跟踪」的每个变量的值。

一旦你基本理解了代码的功能，回到电脑面前，在代码上重读一次，看看能不能找到新的问题点。然后继续找新的代码，用上述的方法去阅读理解，直到你不再需要纸质打印为止。

加分练习

1. 研究一下什么是「流程图(**flow chart**)」，并学着画一下。
2. 如果你在读代码的时候找出了错误，试着把它们改对，并把修改内容发给作者。
3. 不使用纸质打印时，你可以使用注解符号`#`在程序中加入笔记。有时这些笔记会对后来的读代码的人有很大的帮助。

练习 39：数组的操作

你已经学过了数组。在你学习“while 循环”的时候，你对数组进行过「pushed」动作，而且将数组的内容打印了出来。另外你应该还在加分练习里研究过 Ruby 文件，看了数组支持的其他操作。这已经是一段时间以前了，所以如果你不记得了的话，就回到本书的前面再复习一遍吧。

找到了吗？还记得吗？很好。那时候你对一个数组执行了 push 函式。不过，你也许还没有真正明白发生的事情，所以我们再来看看我们可以对数组进行什么样的操作。

当你看到像 `mystuff.append('hello')` 这样的程序时，你事实上已经在 Ruby 内部激发了一个连锁反应。以下是它的运作原理：

1. Ruby 看到你用到了 `mystuff`，于是就去找到这个变量。也许它需要倒着检查你有没有在哪里用=建立过这个变量，或者检查它是不是一个函式参数，或者看它是不是一个全局变量。不管哪种方式，它得先找到 `mystuff` 这个变量才行。
2. 一旦它找到了 `mystuff`，就轮到处理句点 `.`(period) 这个操作符号，而且开始查看 `mystuff` 内部的一些变量了。由于 `mystuff` 是一个数组，Ruby 知道 `mystuff` 支持一些函式。
3. 接下来轮到了处理 `push`。Ruby 会将「push」和 `mystuff` 支持的所有函式的名称一一对比，如果确实其中有一个叫 `push` 的函式，那么 Ruby 就会去使用这个函式。
4. 接下来 Ruby 看到了括号 `(parenthesis)` 并且意识到，「噢，原来这应该是一个函式」，到了这里，它就正常会呼叫这个函式了，不过这里的函式还要多一个参数才行。

一下子要消化这么多可能有点难度，不过我们将做几个练习，让你头脑中有一个深刻的打印象。下面的练习将字符串和列表混在一起，看看你能不能在里边找出点乐子来：

```
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!
```

你应该看到的结果

```
$ 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", "Corn"]
```

```
Let's do some things with stuff.
```

```
Oranges
```

```
Corn
```

```
Corn
```

```
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
```

```
Telephone#Sugar
```

```
$
```

加分练习

1. 上网阅读一些关于「物件导向程序(Object Oriented Programming)」的资料。晕了吧？嗯，我以前也是。别担心。你将从这本书学到足够用的关于物件导向程序的基础知识，而以后你还可以慢慢学到更多。
2. `something.methods` 和 `something` 的 `class` 有什么关系？
3. 如果你不知道我讲的是些什么东西，别担心。程序设计师为了显得自己聪明，于是就发明了 `Opject Oriented Programming`，简称为 `OOP`，然后他们就开始滥用这个东西了。如果你觉得这东西太难，你可以开始学一下「函式式程序(functional programming)」。

练习 40: Hash, 可爱的 Hash

接下来我要教你另外一种让你伤脑筋的容器型资料结构，因为一旦你学会这种资料结构，你将拥有超酷的能力。这是最有用的容器：**Hash**。

Ruby 将这种资料类型叫做「**Hash**」，有的语言里它的名称是「**dictionaries**」。这两种名字我都会用到，不过这并不重要，重要的是它们和数组的区别。你看，针对数组你可以做这样的事情：

```
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 >
```

你可以使用数字作为数组的「索引」，也就是你可以通过数字找到数组中的元素。而 **Hash** 所作的，是让你可以通过任何东西找到元素，不只是数字。是的，**Hash** 可以将一个物件和另外一个东西关联，不管它们的类型是什么，我们来看看：

```
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 >
```

你将看到除了通过数字以外，我们在 **Ruby** 还可以用字符串来从 **Hash** 中获取 **stuff**，我们还可以用字符串来在 **Hash** 中添加元素。当然它支持的不只有字符串，我们还可以做这样的事情：

```
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 >
```

在这里我使用了数字。其实我可以使用任何东西，不过这么说并不准确，不过你先这么理解就行了。

当然了，一个只能放东西进去的 **Hash** 是没啥意思的，所以我们还要有删除物件的方法，也就是使用 `delete` 这个关键字：

```
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 >
```

接下来我们要做一个练习，你必须「非常」仔细，我要求你将这个练习写下来，然后试着看懂它做了些什么。这个练习很有趣，做完以后你可能会有豁然开朗的感觉。

```
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
```

你应该看到的结果

```
$ 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) >
```

加分练习

1. 在 Ruby 文件中找到 Hash 相关的内容，学着对 Hash 做更多的操作。
2. 找出一些 Hash 无法做到的事情。例如比较重要的一个就是 Hash 的内容是无序的，你可以检查一下看看是否真是这样。
3. 试着把 for 循环执行到 Hash 上面，然后试着在 for 循环中使用 Hash 的 each 函式，看看会有什么样的结果。

练习 41: 来自 Percal 25 号行星的哥顿人 (Gothons)

你在上一节中发现 Hash 的秘密功能了吗？你可以解释给自己吗？让我来给你解释一下，顺便和你自己的理解对比看有什么不同。这里是我们要讨论的代码：

```
cities[:find] = method(:find_city)

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

你要记住一个函式也可以作为一个变量，为了要将一个代码区段储存在一个变量里，我们创造了一个东西叫「proc」，proc 是 procedure 缩写。在这段代码中，首先我们调用了 Ruby 内建的函式 method，它会回传一个 proc 版的 find_city 函式。然后我们将之除存在一个 Hash 里：key 是 :find，value 是 cities。。这和我们省和市关联起来的代码做的事情一样，只不过在这个情况里是个 proc。

好了，所以一旦我们知道 find_city 是在 Hash 中 :find 的位置，这就意味着我们可以去调用它。第二行代码可以分解成如下步骤：

1. Ruby 读到了 cities，然后知道了它是一个「Hash」。
2. 然后看到了[:find]，于是 Ruby 就从索引找到了 cities Hash 中对应的位置，并且获取了该位置的内容。
3. [:find] 这个位置的内容是我们的函式 find_city，所以 Ruby 就知道了这里表示一个函式，于是当它碰到.call 就开始了 proc 调用。
4. cities、state 这两个参数将被传递到函式 find_city 中，然后这个函式就被运行了。
5. find_city 接着从 cities 中寻找 states，并且回传它找到的内容，如果什么都没找到，就返回一个信息说它什么都没找到。
3. Ruby 接受 find_city 传回的资讯，最后将该资讯赋值给一开始的 city_found 这个变量。

我再教你一个小技巧。如果你倒着阅读的话，代码可能会变得更容易理解。让我们来试一下，一样是那行：

1. `state` 和 `city` 是...
2. 最为参数传递给...
3. 一个 `proc` 位于...
4. `:find` 然后寻找，目的地为...
5. `cities` 这个 `Hash`...
6. 最后印到屏幕上

还有一种方法读它，这回是「由里向外」。

1. 找到表示式的中心位置，此次为 `[:find]`。
2. 逆时针追溯，首先看到的是一个叫 `cities` 的 `Hash`，这样就知道了 `cities` 中的 `:find` 元素。
3. 上一步得到一个函式。继续逆时针寻找，看到的是参数。
4. 参数传递给函式后，函式会传回一个值。然后再逆时针寻找。
5. 最后，我们到了 `city_found=` 的赋值位置，并且得到了最终结果。

数十年的程序经验下来，我在读代码的过程中已经用不到上面的三种方法了。我只要瞄一眼就能知道它的意思。甚至给我一整页的代码，我也可以一眼瞄出里边的 `bug` 和错误。这样的技能是花了超乎常人的时间和精力才锻炼得来的。在磨练的过程中，我学会了下面三种读代码的方法：

1. 从前向后。
2. 从后向前。
3. 逆时针方向。

现在我们来写这次的练习，写完后再过一遍，这节练习其实挺有趣的。

代码不少，不过还是从头写完吧。确认它能运行，然后玩一下看看。

```
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 "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 Gothons' 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

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

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)

```

你应该看到的结果

```
$ 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.
```

加分练习

1. 解释一下返回至下一个房间的运作原理。
2. 建立更多的房间，让游戏规模变大。
3. 除了让每个函数打印出自己以外，试试学习一下「文件注解(doc comments)」。
4. 看看你能不能将房间描述写成文件注解，然后修改运行它的代码，让它把文档注解打印出来。
5. 一旦你用了文件注解作为房间描述，你还需要让这个函数打出用户提示吗？试着让运行函数的代码打出用户提示来，然后将用户输入传递到各个函数。你的函数应该只是一些 `if` 语句组合，将结果印出来，并且返回下一个房间。
5. 这其实是一个小版本的「有限状态机(finite state machine)」，找资料阅读了解一下，虽然你可能看不懂，但还是找来看看吧

练习 43：你来制作一个游戏

你要开始学会自食其力了。通过阅读这本书你应该已经学到了一点，那就是你需要的所有的信息网路上都有，你只要去搜索就能找到。唯一困扰你的就是如何使

用正确的词汇进行搜索。学到现在，你在挑选搜索关键字方面应该已经有些感觉了。现在已经是时候了，你需要尝试写一个大的专题，并让它运行起来。

以下是你的需求：

1. 制作一个截然不同的游戏。
2. 使用多个文件，并使用 `require` 调用这些文件。确认自己知道 `require` 的用法。
3. 对于每个房间使用一个 `class`，`class` 的命名要能体现出它的用处。例如 `GoldRoom`、`KoiPondRoom`。 [资讯](#)
4. 你的编辑器代码应该了解这些房间，所以创建一个 `class` 来调用并且记录这些房间。有很多种方法可以达到这个目的，不过你可以考虑让每个房间传回下一个房间，或者设置一个变数，让它指定下一个房间是什么。

其他的事情就全靠你了。花一个星期完成这件任务，做一个你能做出来的最好的游戏。使用你学过的任何东西（类，函数，`Hash`，数组.....）来改进你的程序。这节课的目的是教你如何构建 `class` 出来，而这些 `class` 又能调用到其它 `Ruby` 文件中的 `class`。

我不会详细地告诉你告诉你怎样做，你需要自己完成。试着下手吧，写程序就是解决问题的过程，这就意味着你要尝试各种可能性，进行实验，经历失败，然后丢掉你做出来的东西重头开始。当你被某个问题卡住的时候，你可以向别人寻求帮助，并把你的程序贴出来给他们看。如果有人刻薄你，别理他们，你只要集中精力在帮你的人身上就可以了。持续修改和清理你的代码，直到它完整可执行为止，然后再研究一下看它还能不能被改进。

祝你好运，下个星期你做出游戏后我们再见。

练习 44：评估你的游戏

这节练习的目的是检查评估你的游戏。也许你只完成了一半，卡在那里没有进行下去，也许你勉强做出来了。不管怎样，我们将串一下你应该弄懂的一些东西，并确认你的游戏里有使用到它们。我们将学习如何用正确的格式构建 `class`，使用 `class` 的一些通用习惯，另外还有很多的「书本知识」让你学习。

为什么我会让你先行尝试，然后才告诉你正确的做法呢？因为从现在开始你要学会「自给自足」，以前是我牵着你前行，以后就得靠你自己了。后面的练习我只会告诉你你的任务，你需要自己去完成，在你完成后我再告诉你如何可以改进你的作业。

一开始你会觉得很困难并且很不习惯，但只要坚持下去，你就会培养出自己解决问题的能力。你还会找出创新的方法解决问题，这比从课本中拷贝解决方案强多了。

函数的风格

以前我教过的怎样写好函数的方法一样是适用的，不过这里要添加几条：

由于各种各样的原因，程序员将 **class** (类)里边的函数称作 **method** (方法)。很大程度上这只是个市场策略（用来推销 **OOP**），不过如果你把它们称作「函数」的话，是会有罗嗦的人跳出来纠正你的。如果你觉得他们太烦了，你可以告诉他们从数学方面示范一下「函数」和「方法」究竟有什么不同，这样他们会很快闭嘴的。

在你使用 **class** 的过程中，很大一部分时间是告诉你的 **class** 如何「做事情」。给这些函数命名的时候，与其命名成一个名词，不如命名为一个动词，作为给 **class** 的一个命令。就和数组中的 **pop** 函数一样，它相当于说：「嘿，数组，把这东西给我 **pop** 出去。」它的名字不是 **removefromendoflist**，因为即使它的功能的确是这样，这一个字串也不是一个命令。

让你的函数保持简单小巧。由于某些原因，有些人开始学习 **class** 后就会忘了这一条。

Classh (类) 的风格

你的 `class` 应该使用「**camel case**（驼峰式大小写）」，例如你应该使用 `SuperGoldFactory` 而不是 `super_gold_factory`

你的 `initialize` 不应该做太多的事情，这会让 `class` 变得难以使用。

你的其它函数应该使用「**underscore format**（下划线隔词）」，所以你可以写 `my_awesome_hair`，而不是 `myawesomehair` 或者 `MyAwesomeHair`。

用一致的方式组织函数的参数。如果你的 `class` 需要处理 `users`、`dogs`、和 `cats`，就保持这个次序（特殊情况除外）。如果一个函数的参数是 `(dog, cat, user)`，另一个的是 `(user, cat, dog)`，这会让函数使用起来很困难。

不要对全局变量或者来自模组的变量进行重定义或者赋值，让这些东西自顾自就行了。

不要一根筋式地维持风格一致性，这是思维力底下的妖怪喽罗做的事情。一致性是好事情，不过愚蠢地跟着别人遵从一些白痴口号是错误的行为——这本身就是一种坏的风格。好好为自己着想吧。

代码风格

为了以方便他人阅读，在自己的代码之间留下一些空白。你将会看到一些很差的开发人员，他们写的代码还算通顺，但代码之间没有任何空间。这种风格在任何程序语言中都是坏习惯，人的眼睛和大脑会通过空白和垂直对齐的位置来扫描和区隔视觉元素，如果你的代码里没有任何空白，这相当于为你的程序码上了迷彩装。

如果一段代码你无法朗读出来，那么这段代码的可读性可能就有问题。如你找不到让某个东西易用的方法，试着也朗读出来。这样不仅会逼迫你慢速而且真正仔细阅读过去，还会帮你找到难读的段落，从而知道那些代码的易读性需要作出改进。

学着模仿别人的风格写程序，直到哪天你找到你自己的风格为止。

一旦你有了自己的风格，也别把它太当回事。开发人员工作的一部分就是和别人的代码打交道，有的人审美观就是很差。相信我，你的审美观某一方面一定也很差，只是你从未意识到而已。

如果你发现有人写的代码风格你很喜欢，那就模仿他们的风格。

好的注释

有程序员会告诉你，说你的代码需要有足够的可读性，这样你就无需写注释了。他们会以自己接近官腔的声音说「所以你永远都不应该写代码注释。」这些人要嘛是一些顾问型的人物，如果别人无法使用他们的代码，就会付更多钱给他们让他们解决问题。要嘛他们能力不足，从来没有跟别人合作过。别理会这些人，好好写你的注释。

写注释的时候，描述清楚为什么你要这样做。代码只会告诉你「这样实现」，而不会告诉你「为什么要这样实现」，而后者比前者更重要。

当你为函数写文件注释的时候，记得为别的代码使用者也写些东西。你不需要狂写一大堆，但一两句话谢谢这个函数的用法还是很有用的。

最后要说的是，虽然注释是好东西，太多的注释就不见得是了。而且注释也是需要维护的，你要尽量让注释短小精悍一语中的，如果你对代码做了更改，记得检查并更新相关的注释，确认它们还是正确的。

评估你的游戏

现在我要求你假装成是我，板起脸来，把你的代码打印出来，然后拿一支红笔，把代码中所有的错误都标出来。你要充分利用你在本章以及前面学到的知识。等你批改完了，我要求你把所有的错误改对。这个过程我需要你多重复几次，争

取找到更多的可以改进的地方。使用我前面教过的方法，把代码分解成最 细小的单元一一进行分析。

这节练习的目的是训练你对于细节的关注程度。等你检查完自己的代码，再找一段别人的代码用这种方法检查一遍。把代码打印出来，检查出所有代码和风格方面的错误，然后试着在不改坏别人代码的前提下把它们修改正确。

这周我要求你的事情就是批改和纠错，包含你自己的代码和别人的代码，再没有别的了。这节练习难度还是挺大，不过一旦你完成了任务，你学过的东西就会牢牢牢记在脑中。

练习 45：对象，类和从属关系

有一个重要的概念你需要弄明白，那就是 `Class`「类」和 `Object`「对象」的区别。问题在于，`class` 和 `object` 并没有真正的不同。它们其实是同样的东西，只是在不同的时间名字不同罢了。我用禅语来解释一下吧：

鱼 (Fish) 和 鲑鱼 (Salmon) 有什么区别？

这个问题有没有让你有点晕呢？说真的，坐下来想一分钟。我的意思是说，鱼和鲑鱼是不一样，不过它们其实也是一样的是不是？泥鳅是鱼的一种，所以说没什么不同，不过泥鳅又有些特别，它和别的种类的鱼的确不一样，比如鲑鱼和比目鱼就不一样。所以鲑鱼和鱼既相同又不同。怪了。

这个问题让人晕的原因是大部分人不会这样去思考问题，其实每个人都懂这一点，你无须去思考鱼和鲑鱼的区别，因为你知道它们之间的关系。你知道鲑鱼是鱼的一种，而且鱼还有别的种类，根本就没必要去思考这类问题。

让我们更进一步，假设你有一只水桶，里边有三条鲑鱼。假设你的好人卡多到没地方用，于是你给它们分别取名叫 **Frank**，**Joe**，**Mary**。现在想想这个问题：

Mary 和 鲑鱼 有什么区别？

这个问题一样的奇怪，但比起鱼和鲑鱼的问题来还好点。你知道 **Mary** 是一条鲑鱼，所以他也没什么不同，他只是鲑鱼的一个「实例(instance)」。**Joe** 和 **Frank** 一样也是鲑鱼的实例。我的意思是说，它们是由鲑鱼创建出来的，而且代表着和鲑鱼一样的属性。

所以我们的思维方式是（你可能会有点不习惯）：鱼是一个「类(class)」，鲑鱼是一个「类(class)」，而 **Mary** 是一个「对象(object)」。仔细想想，然后我再一点一点慢慢解释给你。

鱼是一个「类」，表示它不是一个真正的东西，而是一个用来描述具有同类属性的实例的概括性词汇。你有鳍？你有鳃？你住在水里？好吧那你就是一条鱼。

后来一个博士路过，看到你的水桶，于是告诉你：「小伙子，你这些鱼是鲑鱼。」专家一出，真相即现。并且专家还定义了一个新的叫做「鲑鱼」的「类」，而这个「类」又有它特定的属性。长鼻子？红肉？体型大？住在海里或是干净新鲜的水里？吃起来味道不错？那你就是一条鲑鱼。

最后一个厨师过来了，他跟博士说：「非也非也，你看到的是鲑鱼，我看到的是 **Mary**，而且我要把 **Mary** 淋上美味酱料做一道小菜。」于是你就有了一只叫做 **Mary** 的鲑鱼的「实例(instance)」（鲑鱼也是鱼的一个「实例」），并且你使用了它（把它塞到你的胃里了），这样它就是一「对象 (object)」。

这会你应该了解了：**Mary** 是鲑鱼的成员，而鲑鱼又是鱼的成员。这里的关系式：对象属于某个类，而某个类又属于另一个类。

写完后的代码是什么样子

这个概念有点诡异，不过实话说，你只要在建立和使用 **class** 的时候操心一下就可以了。我来给你两个区分 **Class** 和 **Object** 的小技巧。

首先针对类和对象，你需要学会两个说法，「**is-a**(是啥)」和「**has-a**(有啥)」。「是啥」要用在谈论「两者以类的关系互相关联」的时候，而「有啥」要用在「两者无共同点，仅是互为参照」的时候。

接下来，通读这段代码，将每一个注解为##??的位置标明他是「**is-a**」还是「**has-a**」的关系，并讲明白这个关系是什么。在代码的开始我还举了几个例子，所以你只要写剩下的就可以了。

记住，「是啥」指的是鱼和鲑鱼的关系，而「有啥」指的是鲑鱼和烤肉架的关系。

```
## 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
```

加分练习

1. 有没有办法把 Class 当作 Object 使用呢？
2. 在练习中为 `animals`、`fish`、还有 `people` 添加一些函数，让它们做一些事情。看看当函数在 `Animal` 这样的「基类(base class)」里和在 `Dog` 里有什么区别。
3. 找些别人的代码，理清里边的「是啥」和「有啥」的关系。
4. 使用 `Array` 和 `Hash` 建立一些新的一一对应的「has-many」的关系。
5. 你认为会有一种「has-many」的关系吗？阅读一下关于「多重继承(multiple inheritance)」的资料，然后尽量避免这种用法。

练习 46: 一个项目骨架

这里你将学会如何建立一个项目「骨架」目录。这个骨架目录具备让项目跑起来的所有基本内容。它里边会包含你的项目文件布局、自动化测试代码，模块，以及安装脚本。当你建立一个新项目的时候，只要把这个目录复制过去，改改目录的名字，再编辑里面的文件就行了。

骨架内容: Linux/OSX

首先使用下述命令创建你的骨架目录：

```
$ mkdir -p projects

$ cd projects/

$ mkdir skeleton

$ cd skeleton

$ mkdir bin lib lib/NAME test
```

我使用了一个叫 `projects` 的目录，用来存放我自己的各个项目。然后我在里边建立了一个叫做 `skeleton` 的文件夹，这就是我们新项目的基础目录。其中叫做 `NAME` 的文件夹是你的项目的主文件夹，你可以将它任意取名。

接下来我们要配置一些初始文件：

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

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

然后我们可以建立一个 `NAME.gemspec` 的文件在我们的项目的根目录，这个文件在安装项目的时候我们会用到它：

```
# -*- 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

```

编辑这个文件，把自己的联络方式写进去，然后放到那里就行了。

最后你需要一个简单的测试专用(我们会在下一节中提到 **Test**)的骨架文件叫 `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
```

安装 Gems

Gems 是 Ruby 的套件系统，所以你需要知道怎么安装它和使用它。不过问题就来了。我的本意是让这本书越清晰越干净越好，不过安装软件的方法是在是太多了，如果我要一步一步写下来，那 10 页都写不完，而且告诉你吧，我本来就是懒人。

所以我不会提供详细的安装步骤了，我只会告诉你需要安装哪些东西，然后让你自己搞定。这对你也有好处，因为你将打开一个全新的世界，里边充满了其他人发布的软件。

接下来你需要安装下面的软件套件：

- `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/>

不要只是手动下载并且安装这些软件套件，你应该看一下别人的建议，尤其看看针对你的操作系统别人是怎样建议你安装和使用的。同样的软件套件在不一样的操作系统上面的安装方式是不一样的，不一样版本的 **Linux** 和 **OSX** 会有不同，而 **Windows** 更是不同。

我要预先警告你，这个过程会是相当无趣。在业内我们将这种事情叫做「**yak shaving**(剃牦牛)」。它指的是在你做一件有意义的事情之前的一些准备工作，而这些准备工作又是及其无聊冗繁的。你要做一个很酷的 **Ruby** 项目，但是创建骨架目录需要你安装一些软件到件，而安装软件套件之前你还要安装 **package installer** (软件套件安装工具)，而要安装这个工具你还得先学会如何在你的操作系统下安装软件，真是烦不胜烦呀。

无论如何，还是克服困难吧。你就把它当做进入程式俱乐部的一个考验。每个开发人员都会经历这条道路，在每一段「酷」的背后总会有一段「烦」的。

使用这个骨架

剃牦牛的事情已经做的差不多了，以后每次你要新建一个项目时，只要做下面的事情就可以了：

1. 拷贝这份骨架目录，把名字改成你新项目的名字。

2. 再将 `NAME` 模块和 `NAME.rb` 更名为你需要的名字，它可以是你项目的名字，当然别的名字也行。
3. 编辑你的 `NAME.gemspec` 文件，让它包含你新项目的相关资讯。
4. 重命名 `test/test_NAME.rb`，让它的名字匹配到你模块的名字。
5. 开始写程式吧。

小测验

这节练习没有加分练习，不过需要你做一个小测验：

1. 找文件阅读，学会使用你前面安装了的软件套件。
2. 阅读关于 `NAME.gemspec` 的文件，看它里边可以做多少配置。
3. 建立一个项目，在 `NAME.rb` 里写一些代码。
4. 在 `bin` 目录下放一个可以运行的脚本，找材料学习一下怎样建立可以在系统下运行的 **Ruby** 脚本。
5. 确定你建立的 `bin` 脚本，有在 `NAME.gemspec` 中被参照到，这这样你安装时就可以连它安装进去。
6. 使用你的 `NAME.gemspec` 和 `gem build`、`gem install` 来安装你写的程式和确定它能用。然后使用 `gem uninstall` 去移除它。
7. 弄懂如何使用 **Bundler** 来自动建立一个骨架目录。

练习 47：自动化测试

为了确认游戏的功能是否正常，你需要一遍一遍地在你的游戏中输入命令。这个过程是很枯燥无味的。如果能写一小段代码用来测试你的代码岂不是更好？然后只要你对程序做了任何修改，或者添加了什么新东西，你只要「跑一下你的测试」，而这些测试能确认程序依然能正确运行。这些自动测试不会抓到所有的 **bug**，但可以让你无需重复输入命令运行你的代码，从而为你节约很多时间。

从这一章开始，以后的练习将不会有「你应该看到的结果」这一节，取而代之的是一个「你应该测试的东西」一节。从现在开始，你需要为自己写的所有代码写自动化测试，而这将让你成为一个更好的程序员。

我不会试图解释为什么你需要写自动化测试。我要告诉你的是，你想要成为一个开发人员，而程序的作用是让无聊冗繁的工作自动化，测试软件毫无疑问是无聊冗繁的，所以你还是写点代码让它为你测试的更好。

这应该是你需要的所有的解释了。因为你写单元测试的原因是让你的大脑更加强健。你读了这本书，写了很多代码让它们实现一些事情。现在你将更进一步，写出懂得你写的其他代码的代码。这个写代码测试你写的其他代码的过程将强迫你清楚的理解你之前写的代码。这会让你更清晰地了解你写的代码实现的功能及其原理，而且让你对细节的注意更上一个台阶。

撰写 Test Case

我们将拿一段非常简单的代码为例，写一个简单的测试，这个测试将建立在上节我们创建的项目骨架上面。

首先从你的项目骨架创建一个叫做 `ex47` 的项目。确认该改名称的地方都有改过，尤其是 `tests/ex47_tests.rb` 这处不要写错。

接下来建立一个简单的 `ex47/lib/game.rb` 文件，里边放一些用来被测试的代码。我们现在放一个傻乎乎的小 `class` 进去，用来作为我们的测试对象：

```
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
```

一旦你有了这个文件，修改你的 `unit test` 骨架变成这样：

```
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
```

这个文件 `require` 了你在 `lib/ex47.rb` 里建立的 `Room` 这个类，接下来我们要做的就是测试它。于是我们看到一系列的以 `test_` 开头的测试函数，它们就是所谓的「**Test Case**」，每一个 **Test Case** 里面都有一小段代码，它们会建立一个或者一些房间，然后去确认房间的功能和你期望的是否一样。它测试了基本的房间功能，然后测试了路径，最后测试了整个地图。

这里最重要的函数是 `assert_equal`，它保证了你设置的变量，以及你在 `Room` 里设置的路径和你的期望相符。如果你得到错误的结果的话，`Ruby` 的 `Test::Unit` 模块将会印出一个错误信息，这样你就可以找到出错的地方并且修正过来。

测试指南

在写测试代码时，你可以照着下面这些不是很严格的指南来做：

1. 测试脚本要放到 `tests/` 目录下，并且命名为 `test_NAME.rb`。这样做还有一个好处就是防止测试代码和别的代码互相混淆。
2. 为你的每一个模块写一个测试。
3. **Test Cast** 函数保持简短，但如果看上去不怎么整洁也没关系，**Test Cast** 一般都有点乱。

4. 就算 `Test Cast` 有些乱，也要试着让他们保持整洁，把里边重复的代码删掉。建立一些辅助函数来避免重复的代码。当你下次在改完代码需要改测试的时候，你会感谢我这一条建议的。重复的代码会让修改测试变得很难操作。
5. 最后一条是别太把测试当做一回事。有时候，更好的方法是把代码和测试全部删掉，然后重新设计代码。

你应该看到的结果

```
$ 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.
```

加分练习

1. 仔细阅读 `Test::Unit` 相关的文件，再去了解一下其他的替代方案。
2. 了解一下 `Rspec`，看看它是否可以干得更出色。
3. 改进你游戏里的 `Room`，然后用它重建你的游戏。这次重写，你需要一边写代码，一般把单元测试写出来。

练习 48：更进一步的使用者输入

你的游戏可能一路跑的很爽，不过你处理使用者输入的方式肯定让你不胜其烦。每一个房间都需要一套自己的语句，而且只有使用者输入正确后才能执行。你需要一个设备，他可以允许使用者以各种方式输入语句。例如下面的集中表达，应该都是被支持的才对：

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

也就是说，如果使用者的输入和重用英语很接近也应该是可以的，而你的游戏要识别出他们的意思。为了达到这个目的，我们将写一个模块专门做这件事。这个模块中会有若干个类，他们互相配合，解释使用这输入，并且将使用者输入转换成你的游戏能识别的命令。

英语的简单格式是这样的：

- 单词由空格隔开。
- 句子由单词组成。
- 语法控制句子的含义。

所以最好的开始方式是先搞定如何得到使用者输入的语义，并判断出他们是什么。

我们的游戏词汇

我在游戏里建立了下面这些词汇：

- 表示方向: north, south, east, west, down, up, left, right, back.
- 动词: go, stop, kill, eat.
- 修饰词: the, in, of, from, at, it
- 名词: door, bear, princess, cabinet.
- 数字词: 由 0-9 构成的数字。

说到名称，我们会碰到一个小问题，那就是不一样的房间用不一样的一组名词，不过让我们先挑一个小组出来写程序，以后在做改进。

如何断句

我们已经有了词汇表，为了缝隙句子的意思，接下来我们需要找到一个断句的方法。我们对于句子的定义「空格隔开的单词」，所以只要这样就可以：

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

目前做到这样就可以了，不过这招在相当一段时间内都不会有问题。

语句结构

一旦我们知道了如何将句子转换为词汇表，剩下的就是逐一坚持这些词汇，看他是什么类型。为了达到这个目的，我们将用到一个非常便利的 ruby 资料结构「struct」。「struct」其实就是一个可以把一串的 attributes 绑在一起的方式，使用 accessor 函数，但不需要写一个复杂的 class。它的建立方式就是这样：

```
1 Pair = Struct.new(:token, :word)  
2 first_word = Pair.new("direction", "north")  
3 second_word = Pair.new("verb", "go")
```



```
4 sentence = [first_word, second_word]
```

这建立了一对 (TOKEN, WORD) 可以让你看到 word 和在里面做的事。

这只是一个例子, 不过最后做出的样子也差不多。你接受使用者输入, 用 `split` 将其分隔成单词列表, 然后分析这些单词, 识别它们的类型, 最后重新组成一个句子。

扫描输入资料

现在你要写的词汇扫描器。这个扫描器会将使用者的输入字符串当成参数, 然后返回多个 (TOKEN, WORD) `struct` 组成的列表, 这个列表实现类似句子的功能。如果一个单词不在预定的词汇表内, 那它返回时 WORD 应该还在, 但 TOKEN 应该设置成一个专门的错误标记。这个错误标记将告诉使用者哪里出错了。

有趣的地方来了。我不会告诉你这些该怎么做, 但我会写一个「单元测试 (unit test)」, 而你要把你的扫描器写出来, 并且保证单元测试能够正常通过。

Exceptions And Numbers

有一件小事我会先帮帮你，那就是数字转换。为了做到这一点，我们会做一点弊，使用「异常(exception)」来做。「异常」指的是你在运行某个函数时得到的错误。你的函数在遇到错误时，就会「提出(raise)」一个「异常」，然后你就要去处理(handle)这个异常。加入你在 IRB 里写了这些东西：

```
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>'
```

这个 `ArgumentError` 就是 `Integer()` 函数抛出的一个异常。因为你给 `Integer()` 的参数不是一个数字。`Integer()` 函数其实也可以传回一个值来告诉你它遇到了一个错误，不过由于他只能返回整数值，所以很难做到这一点。它不能返回-1，因为这也是一个数字（负数）。`Integer()` 没有纠结在它「究竟应该返回什么」上面，而是提出了一个叫做 `TypeError` 的异常，然后你只要处理这个异常就可以了。

异常处理的方法是使用 `begin` 和 `rescue` 这两个关键字：

```
1 def convert_number(s)
2   begin
3     Integer(s)
4   rescue ArgumentError
5     nil
6   end
7 end
```

你要试着运行的程序代码放到「begin」的区域里，再将出错后要运行的代码当道「except」区域里。在这里，我们要试着调用 `Integer()` 去处理某个可能是数字的东西，如果中间出错，我们就「rescue」这个错误，然后返回「nil」。

在你写的扫描器里面，你应该使用这个函数来测试某个东西是不是数字。做完这个检查，你就可以声明这个单词时一个错误单词了。

What You Should Test

这里是你应该使用的测试档案 `test/test_lexicon.rb`:

```
1 require 'test/unit'
2 require_relative "../lib/lexicon"
3
4 class LexiconTests < Test::Unit::TestCase
5
6   Pair = Lexicon::Pair
7
8   @@lexicon = Lexicon.new()
9
10  def test_directions()
11    assert_equal([Pair.new(:direction, 'north')],
12  @@lexicon.scan("north"))
13
14    result = @@lexicon.scan("north south east")
15
16    assert_equal(result, [Pair.new(:direction, 'north'),
17      Pair.new(:direction, 'south'),
18      Pair.new(:direction, 'east')])
19  end
20
21  def test_verbs()
22    assert_equal(@@lexicon.scan("go"), [Pair.new(:verb,
23  'go')])
24  end
25 end
```

```
20     result = @@lexicon.scan("go kill eat")
21     assert_equal(result, [Pair.new(:verb, 'go'),
22                           Pair.new(:verb, 'kill'),
23                           Pair.new(:verb, 'eat')])
24 end
25
26 def test_stops()
27     assert_equal(@@lexicon.scan("the"), [Pair.new(:stop,
28 'the')])
29
30     result = @@lexicon.scan("the in of")
31
32     assert_equal(result, [Pair.new(:stop, 'the'),
33                           Pair.new(:stop, 'in'),
34                           Pair.new(:stop, 'of')])
35 end
36
37 def test_nouns()
38     assert_equal(@@lexicon.scan("bear"), [Pair.new(:noun,
39 'bear')])
40
41     result = @@lexicon.scan("bear princess")
42
43     assert_equal(result, [Pair.new(:noun, 'bear'),
```

```
39         Pair.new(:noun, 'princess']])
40     end
41
42     def test_numbers()
43         assert_equal(@@lexicon.scan("1234"), [Pair.new(:number,
44 1234)])
45
46         result = @@lexicon.scan("3 91234")
47
48         assert_equal(result, [Pair.new(:number, 3),
49
50             Pair.new(:number, 91234)])
51     end
52
53     def test_errors()
54         assert_equal(@@lexicon.scan("ASDFADFASDF"),
55 [Pair.new(:error, 'ASDFADFASDF')])
56
57         result = @@lexicon.scan("bear IAS princess")
58
59         assert_equal(result, [Pair.new(:noun, 'bear'),
60
61             Pair.new(:error, 'IAS'),
62
63             Pair.new(:noun, 'princess']])
64     end
65 end
```

```
end
```

记住你使用的专案框架来建立新专案项目，将这个 Test Case 写下来（不要复制粘贴！），然后编写你的扫描器，直至所有的测试都能通过。注意细节并确认结果一切工作良好。

设计提示

集中一次实现一个测试，尽量保持简单，只要把你的 `lexicon.rb` 词汇表中所有的单词放那里就可以了。不要修改输入的单词表，不过你需要建立自己的新列表，里边包含你的词汇元组。另外，记得使用 `include?` 函数来坚持这些词汇阵列，以确认某个单词时都存在你的词汇表中。

加分练习

1. 改进单元测试，让它覆盖到更多的词汇。
2. 向词汇列表中添加更多的词汇，并且更新单元测试程序代码。
3. 让你的扫描器能够识别任意大小写的单词。更新你的单元测试以及确定其功能。
4. 找到另外一种转换为数字的方法。
5. 我的解决方法用了 37 行代码，你的时更长还是更短呢！

练习 49：创造句子

从我们这个小游戏的词汇扫描器中，我们应该可以得到类似下面的列表（你的看起来可能格式会不太一样）：

```
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 >
```

现在让我们把它转化成游戏可以使用的东西，也就是一个 **Sentence** 类。

如果你还记得学校学过的东西的话，一个句子是由这样的结构组成的：

主语(Subject) + 谓语(动词 Verb) + 宾语(Object)

很显然实际的句子可能会比这复杂，而你可能已经在英语的语法课上面被折腾得够呛了。我们的目的，是将上面的 `struct` 列表转换为一个 `Sentence` 对象，而这个对象又包含主谓宾各个成员。

匹配(Match) And 窥视(Peek)

为了达到这个效果，你需要四样工具：

1. 一个循环存取 `struct` 列表的方法，这挺简单的。
 2. 「匹配」我们的主谓宾设置中不同种类 `struct` 的方法。
 3. 一个「窥视」潜在 `struct` 的方法，以便做决定时用到。
 4. 「跳过(skip)」我们不在乎的内容的方法，例如形容词、冠词等没有用处的词汇。
 5. 我们使用 `peek` 函数查看 `struct` 列表中的下一个成员，做匹配以后再对它做下一步动作。
- 让我们先看看这个 `peek` 函数：

```
def peek(word_list)

  begin

    word_list.first.token

  rescue

    nil

  end

end

end
```

很简单。再看看 `match` 函数：

```
def match(word_list, expecting)

  begin

    word = word_list.shift

    if word.token == expecting

      word

    else

      nil

    end

  rescue

    nil

  end

end
```

还是很简单，最后我们看看 `skip` 函数：

```
def skip(word_list, word_type)

  while peek(word_list) == word_type

    match(word_list, word_type)

  end

end
```

```
end
```

```
end
```

以你现在的水准，你应该可以看出它们的功能来。确认自己真的弄懂了它们。

句子的语法

有了工具，我们现在可以从 `struct` 列表来构建句子(`Sentence`)对象了。我们的处理流程如下：

1. 使用 `peek` 识别下一个单词。
2. 如果这个单词和我们的语法匹配，我们就调用一个函数来处理这部分语法。假设函数的名字叫 `parse_subject` 好了。
3. 如果语法不匹配，我们就 `raise` 一个错误，接下来你会学到这方面的内容。
4. 全部分析完以后，我们应该能得到一个 `Sentence` 对象，然后可以将其应用在我们的游戏中。

演示这个过程最简单的方法是把代码展示给你让你阅读，不过这节练习有个不一样的要求，前面是我给你测试代码，你照着写出代码来，而这次是我给你的程序，而你要为它写出测试代码来。

以下就是我写的用来解析简单句子的代码，它使用了 `ex48` 这个 `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
```

```
end

end

def match(word_list, expecting)

  begin

    word = word_list.shift

    if word.token == expecting

      word

    else

      nil

    end

  rescue

    nil

  end

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
```

end

关于异常(Exception)

你已经简单学过关于异常的一些东西，但还没学过怎样抛出(**raise**)它们。这节的代码示范了如何 **raise**。首先在最前面，你要定义好 `ParserException` 这个类，而它又是 `Exception` 的一种。另外要注意我们是怎样使用 `raise` 这个关键字来抛出异常的。

你的测试代码应该也要测试到这些异常，这个我也会示范给你如何实现。

你应该测试的东西

为《练习 49》写一个完整的测试方案，确认代码中所有的东西都能正常工作，其中异常的测试—输入一个错误的句子它会抛出一个异常来。

使用 `assert_raises` 这个函数来检查异常，在 `Test::Unit` 的文件里查看相关的内容，学着使用它写针对「执行失败」的测试，这也是测试很重要的一个方面。从文件中学会使用 `assert_raises`，以及一些别的函数。

写完测试以后，你应该就明白了这段代码的运作原理，而且也学会了如何为别人的代码写测试代码。相信我，这是一个非常有用的技能。

加分练习

1. 修改 `parse_method`，将它们放到一个类里边，而不仅仅是独立的方法函数。这两种设计你喜欢哪一种呢？
2. 提高 `parser` 对于错误输入的抵御能力，这样即使使用者输入了你预定义语汇之外的词语，你的代码也能正常运行下去。
3. 改进语法，让它可以处理更多的东西，例如数字。
4. 想想在游戏里你的 `Sentence` 类可以对使用者输入做哪些有趣的事情。

练习 50：你的第一个网站

这节以及后面的练习中，你的任务是把你前面建立的游戏做成网页版。这是本书的最后三个章节，这些内容对你来说难度会相当大，你要在上面花些时间才能做出来。在你开始这节练习以前，你必须已经成功地完成过了《练习 46》的内容，正确安装了 **RubyGems**，而且学会了如何安装软件包件以及如何建立项目骨架。如果你不记得这些内容，就回到《练习 46》重新复习一遍。

安装 Sinatra

在建立你的第一个网页应用程序之前，你需要安装一个「Web 框架」，它的名字叫 **Sinatra**。所谓的「框架」通常是指「让某件事情做起来更容易的软件包件」。在网页应用的世界里，人们建立了各种各样的「网页框架」，用来解决他们在建立网站时碰到的问题，然后把这些解决方案用软件包件的方式发布出来，这样你就可以利用它们引导建立你自己的项目了。

可选的框架类型有很多很多，不过在这里我们将使用 **Sinatra** 框架。你可以先学会它，等到差不多的时候再去接触其它的框架，不过 **Sinatra** 本身挺不错的，所以就算你一直使用也没关系。

使用 `gem` 安装 **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...
```

写一个简单的「Hello World」项目

现在你将做一个非常简单的「Hello World」项目出来，首先你要建立一个项目目录：

```
$ cd projects

$ bundle gem gothonweb
```

你最终的目的是把《练习 42》中的游戏做成一个 web 应用，所以你的项目名称叫做 `igothonweb`，不过在此之前，你需要建立一个最基本的 `Sinatra` 应用，将下面的代码放到 `lib/gothonweb.rb` 中：

```
require_relative "gothonweb/version"

require "sinatra"

module Gothonweb

  get '/' do

    greeting = "Hello, World!"

    return greeting

  end

end
```

然后使用下面的方法来运行这个 web 程序：

```
$ 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
```

最后，使用你的网页浏览器，打开 URL `http://localhost:4567/`，你应该看到两样东西，首先是浏览器里显示了 `Hello, world!`，然后是你的命令行终端显示了如下的输出：

```
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
```

这些是 **Sinatra** 打印出的 **log** 信息，从这些信息你可以看出服务器有在运行，而且能了解到程序在浏览器背后做了些什么事情。这些信息还有助于你发现程序

的问题。例如在最后一行它告诉你浏览器试图存取 `/favicon.ico`，但是这个文件并不存在，因此它返回的状态码是 `404 Not Found`。

到这里，我还没有讲到任何 **web** 相关的工作原理，因为首先你需要完成准备工作，以便后面的学习能顺利进行，接下来的两节练习中会有详细的解释。我会要求你用各种方法把你的 **Sinatra** 应用程序弄坏，然后再将其重新构建起来：这样做的目的是让你明白运行 **Sinatra** 程序需要准备好哪些东西。

发生了什么事情？

在浏览器访问到你的网页应用程序时，发生了下面一些事情：

1. 浏览器通过网路连接到你自己的电脑，它的名字叫做 `localhost`，这是一个标准称呼，表示的谁就是网路中你自己的这台电脑，不管它实际名字是什么，你都可以使用 `localhost` 来访问。它使用到 `port 4567`。
2. 连接成功以后，浏览器对 `lib/gothonweb.rb` 这个应用程序发出了 HTTP 请求 (`request`)，要求访问 `URL/`，这通常是一个网站的第一个 `URL`。
3. 在 `lib/gothonweb.rb` 里，我们有一个代码块，里面包含了 `URL` 的匹配关系。我们这里只定义了一组匹配，那就是 `/`。它的含义是：如果有人使用浏览器访问 `/` 这一级目录，**Sinatra** 将找到它，从而用它处理这个浏览器请求。
4. **Sinatra** 调用匹配到的代码块，这段代码只简单的回传了一个字符串回给浏览器。
5. 最后 **Sinatra** 完成了对于浏览器请求的处理将响应 (`response`) 回传给浏览器，于是你就看到了现在的页面。

确定你真的弄懂了这些，你需要画一个示意图，来理清信息是如何从浏览器传递到 **Sinatra**，再到 `/` 区段，再回到你的浏览器的。

修正错误

第一步，把第 6 行的 `greeting` 变量删掉，然后重新刷浏览器。你应该会看到一个错误画面，你可以通过这一页丰富的信息看出你的程序崩溃的原因。当然你已经知道出错的原因是 `greeting` 的赋值遗失了，不过 `Sinatra` 还是会给你一个挺好的错误页面，让你能找到出错的具体位置。试试在这个错误页面上做以下操作：

1. 看看 `sinatra.error` 变量。
2. 看看 `REQUEST_` 变量里的信息。里面哪些知识是你已经熟悉了的。这是浏览器发给你的 `gothonweb` 应用程序的信息。这些知识对于日常网页浏览没有什么用处，但现在你要学会这些东西，以便写出 `web` 应用程序来。

建立基本的模板

你已经试过用各种方法把这个 `Sinatra` 程序改错，不过你有没有注意到「Hello World」不是一个好 HTML 网页呢？这是一个 `web` 应用，所以需要一个好的 HTML 响应页面才对。为了达到这个目的，下一步你要做的是将「Hello World」以较大的绿色字体显示出来。

第一步是建立一个 `lib/views/index.erb` 文件，内容如下：

```
<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>
```

什么是一个 .erb 的文件？ERB 的全名是 Embedded Ruby。 .erb 文件其实是一个内嵌一点 Ruby 代码的 HTML。如果你学过 HTML 的话，这些内容你看上去应该很熟悉。如果你没学过 HTML，那你应该去研究一下，试着用 HTML 写几个网页，从而知道它的运作原理。既然这是一个 erb 模版，Sinatra 就会在模板中找到对应的位置，将参数的内容填充到模板中。例如每一个出现 `<%= greeting %>` 的位置，内容都会被替换成对应这个变量名的参数。

为了让你的 lib/gothonweb.rb 处理模板，你需要写一写代码，告诉 Sinatra 到哪里去找到模板进行加载，以及如何渲染(render)这个模板，按下面的方式修改你的文件：

```
require_relative "gothonweb/version"
```

```
require "sinatra"

require "erb"

module Gothonweb

  get '/' do

    greeting = "Hello, World!"

    erb :index, :locals => {:greeting => greeting}

  end

end
```

特别注意我改了/这个代码块最后一行的内容，这样它就可以调用 `erb` 然后把 `greeting` 变量传给它。

改好上面的程序后，刷新一下浏览器中的网页，你应该会看到一条和之前不同的绿色信息输出。你还可以在浏览器中通过「查看源代码(View Source)」看到模板被渲染成了标准有效的 HTML 源代码。

这么讲也许有些太快了，我来详细解释一下模板的运作原理吧：

1. 在 `lib/gothonweb.rb` 你添加了一个 `erb` 函数调用。
2. 这个 `erb` 函数知道怎么载入 `lib/views` 目录夹里的 `.erb` 的文件。它知道去抓哪些文件（在这个例子里是 `index.erb`）。因为你传了一个参数进去（`erb :index ...`）。
3. 现在，当浏览器读取/且 `lib/gothonweb.eb` 匹配然后执行 `get '/' do` 区段，它再也没有只是回传字符串 `greeting`，而是调用 `erb` 然后传入 `greeting` 作为一个变量。

4. 最后，你让 `lib/views/index.erb` 去检查 `greeting` 这个变量，如果这个变量存在的话，就打印出变量里的内容。如果不存在的话，就会打印出一个预设的内容。

要深入理解这个过程，你可以修改 `greeting` 变量以及 `HTML`，看看会有什么效果。然后也创作另外一个叫做 `lib/views/foo.erb` 的模板。然后把 `erb :index` 改成 `erb :foo`。从这个过程中你也可以看到，你传入给 `erb` 的第一个参数只要匹配到 `lib/views` 下的 `.erb` 文件名称，就可以被渲染出来了。

加分练习

1. 到 `Sinatra` 这个框架的官方网站去阅读更多文件。
2. 实验一下你在上述网站中看到的所有东西，包括他们的范例代码。
3. 阅读有关于 `HTML5` 和 `CSS3` 相关的东西，自己练习写几个 `.html` 和 `.css` 文件。
4. 如果你有一个懂 `Rails` 的朋友可以帮你的画，你可以自己试着使用 `Rails` 完成一下练习 50, 51, 52，看看结果会是什么样子。

练习 51: 从浏览器中取得输入

虽然能让浏览器显示「Hello World」是很有趣的一件事情，但是如果能让用户通过表单(`form`)向你的应用程序提交信息就更有意思了。这节练习中，我们将使用 `form` 改进你的 `web` 程序，并且搞懂如何为一个网站程序写自动化测试。

Web 运作原理

该学点无趣的东西了。在建立 `form` 前你需要先多学一点关于 `web` 的运作原理。这里讲的并不完整，但是相当准确，在你的程序出错时，它会帮你找到出错的原因。另外，如果你理解了 `form` 的应用，那么建立 `form` 对你来说就会更容易了。

我将以一个简单的图示讲起，它向你展示了 **web** 请求的各个不同的部分，以及信息传递的大致流程：

为了方便讲述 **HTTP** 请求(**request**) 的流程，我在每条线上面加了字母标签以作区别。

1. 你在浏览器中输入网址 `http://learnpythonthehardway.org/`，然后浏览器会通过你的电脑的网路设备发出 **request** (线路 A)。
2. 你的 **request** 被传送到网际网路 (线路 B)，然后再抵达远端服务器 (线路 C)，然后我的服务器将接受这个 **request**。
3. 我的服务器接受 **request** 后，我的 **web** 应用程序就去处理这个请求 (线路 D)，然后我的网页应用程序就会去运行 `/ (index)` 这个「处理程序(**handler**)」。
4. 在代码 `return` 的时候，我的服务器就会发出响应(**response**)，这个响应会再通过线路 D 传递到你的浏览器。
5. 这个网站所在的服务器将响应由线路 D 获取，然后通过线路 C 传至网际网路。
6. 响应通过网路网路由线路 B 传至你的电脑，电脑的网路卡再通过线路 A 将响应传给你的浏览器。
7. 最后，你的浏览器显示了这个响应的内容。

这段详解中用到了一些术语。你需要掌握这些术语，以便在谈论你的 **web** 应用时你能明白而且应用它们：

浏览器(browser)

这是你几乎每天都会用到的软件。大部分人不知道它真正的原理，他们只会把它叫作「网际网路」。它的作用其实是接收你输入到地址栏网址(例如 `http://learnpythonthehardway.org`)，然后使用该信息向该网址对应的服务器提出请求(**request**)。

IP 位址 (Address)

通常这是一个像 `http://learnpythonthehardway.org/` 一样的 URL (Uniform Resource Locator, 统一资源定位符), 它告诉浏览器该打开哪个网站。前面的 `http` 指出了你要使用的协议(protocol), 这里我们用的是「超文本传输协议(Hyper-Text Transport Protocol)」。你还可以试试 `ftp://ibiblio.org/`, 这是一个「FTP 文件传输协议(File Transport Protocol)」的例子。`learnpythonthehardway.org` 这部分是「主机名(hostname)」, 也就是一个便于人阅读和记忆的字串, 主机名会被匹配到一串叫作「IP 位址」的数字上面, 这个「IP 位址」就相当于网路中一台电脑的电话号码, 通过这个号码可以访问到这台电脑。最后, URL 中还可以尾随一个「路径」, 例如 `http://learnpythonthehardway.org/book/` 中的 `/book/`, 它对应的是服务器上的某个文件或者某些资源, 通过访问这样的网址, 你可以向服务器发出请求, 然后获得这些资源。网站地址还有很多别的组成部分, 不过这些是最主要的。

连接(connection)

一旦浏览器知道了协议(`http`)、服务器(`learnpythonthehardway.org`)、以及要获得的资源, 它就要去建立一个连接。这个过程中, 浏览器让操作系统 (Operating System, OS) 打开计算机的一个「端口(port)」(通常是 80 端口), 端口准备好以后, 操作系统会回传给你的程序一个类似文件的东西, 它做的事情就是通过网路传输 和接收资料, 让你的电脑和 `learnpythonthehardway.org` 这个网站所属的服务器之间实现资料交流。当你使用 `http://localhost:4567/` 访问你自己的站点时, 发生的事情其实是一样的, 只不过这次你告诉了浏览器要访问的是你自己的电脑(`localhost`), 要使用的端口不是默认的 80, 而是 4567。你还可以直接访问 `http://learnpythonthehardway.org:80/`, 这和不输入端口效果一样, 因为 HTTP 的默认端口本来就是 80。

请求(request)

你的浏览器通过你提供的地址建立了连接，现在它需要从远端服务器要到它（或你）想要的资源。如果你在 URL 的结尾加了 `/book/`，那你想要的就是 `/book/` 对应的文件或资源，大部分的服务器会直接为你调用 `/book/index.html` 这个文件，不过我们就假装不存在好了。浏览器为了获得服务器上的资源，它需要向服务器发送一个「请求」。这里我就不讲细节了，为了得到服务器上的内容，你必须先向服务器发送一个请求才行。有意思的是，「资源」不一定非要是文件。例如当浏览器向你的应用程序提出请求的时候，服务器返回的其实是你的代码生成的一些东西。

服务器(server)

服务器指的是浏览器另一端连接的电脑，它知道如何回应浏览器请求的文件和资源。大部分的 web 服务器只要发送文件就可以了，这也是服务器流量的主要部分。不过你学的是使用 Ruby 组建一个服务器，这个服务器知道如何接受请求，然后返回用 Ruby 处理过的字符串。当你使用这种处理方式时，你其实是假装把文件发给了浏览器，其实你用的都只是代码而已。就像你在《练习 50》中看到的，要构建一个「响应」其实也不需要多少代码。

响应(response)

这就是你的服务器回复给你的请求，传回至浏览器的 HTML，它里边可能有 css、javascript、或者图片等内容。以文件响应为例，服务器只要从磁盘读取文件，发送给浏览器就可以了，不过它还要将这些内容包在一个特别定义的「header」中，这样浏览器就会知道它获取的是什么类型的内容。以你的 web 应用程序为例，你发送的其实还是一样的东西，包括 header 也一样，只不过这些资料是你用 Ruby 代码即时生成的。

这个可以算是你能在网上找到的关于浏览器如何访问网站的最快的快速课程了。这节课应该可以帮你更容易地理解本节的练习，如果你还是不明白，就到处找资料多多了解这方面的信息，知道你明白为止。有一个很好的方法，就是你对照着上面的图示，将你在《练习 50》中创建的 web 程序中的内容分成几个部分，

让其中的各部分对应到上面的图示。如果你可以正确地将程序的各部分对应到这个图示，你就大致开始明白它的运作原理了。

表单(form)的运作原理

熟悉「表单」最好的方法就是写一个可以接收表单资料的程序出来，然后看你可以对它做些什么。先将你的 `lib/gothonsweb.rb` 修改成下面的样子：

```
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
```

重启你的 Sinatra（按 CTRL-C 后重新运行），确认它有运行起来，然后使用浏览器访问 <http://localhost:4567/hello>，这时浏览器应该会显示 “I just wanted to say Hello , Nobody.”，接下来，将浏览器的地址改成 <http://localhost:4567/hello?name=Frank>，然后你可以看到页面显示为 “Hello, Frank.”，最后将 `name=Frank` 修改为你自己的名字，你就可以看到它对你说 Hello 了。

让我们研究一下你的程序里做过的修改。

1. 我们没有直接为 `greeting` 赋值，而是使用了 `params Hash` 从浏览器获取数据。这 `Sinatra` 个函数会将一组在 `URL ?` 后面的部份的 `key / value` 组加进 `params Hash` 里。
2. 然后我从 `params[:name]` 中找到 `name` 的值，并为 `greeting` 赋值，这部份相信你已经很熟悉了。
3. 其他的内容和以前是一样的，我们就不再分析了。

URL 中该还可以包含多个参数。将本例的 URL 改成这样子：

<http://localhost:4567/hello?name=Frank&greet=Hola>。然后修改代码，让它去存取 `params[:name]` 和 `params[:greet]`，如下所示：

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


创建 HTML 表单

你可以通过 URL 参数实现表单提交，不过这样看上去有些丑陋，而且不方便一般人使用，你真正需要的是一个「POST 表单」，这是一种包含了<form>标签的特殊 HTML 文件。这种表单收集使用者输入并将其传递给你的 web 程序，这和你上面实现的目的基本是一样的。

让我们来快速建立一个，从中你可以看出它的运作原理。你需要创建一个新的 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>
```

然后将 `lib/gothonsweb.rb` 改成这样:

```
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
```

都写好以后，重启 **web** 程序，然后通过你的浏览器访问它。

这回你会看到一个表单，它要求你输入「一个问候语句(**A Greeting**)」和「你的名字(**Your Name**)」，等你输入完后点击「提交(**Submit**)」按钮，它就会输出一个正常的问候页面，不过这一次你的 **URL** 还是 **http://localhost:4567/hello**，并没有添加参数进去。

在 `hello_form.erb` 里面关键的一行是`<form action="/hello" method="POST">`，它告诉你的浏览器以下内容：

1. 从表单中的各个栏位收集使用者输入的资料。
2. 让浏览器使用一种 **POST** 类型的请求，将这些资料发送给服务器。这是另外一种浏览器请求，它会将表单栏位「隐藏」起来。
3. 将这个请求发送至 `/hello` URL，这是由 `action="/hello"` 告诉浏览器的。
4. 你可以看到两段 `<input>` 标签的名字属性 (`name`) 和代码中的变数是对应的，另外我们在 `class index` 中使用的不再只是 **GET** 方法，而是另一个 **POST** 方法。

这个新程序的运作原理如下：

1. 浏览器访问到 `web` 程序的 `/hello` 目录，它发送了一个 **GET** 请求，于是我们的 `get '/hello/'` 就运行了并传回了 `hello_form`。
2. 你填好了浏览器的表单，然后浏览器依照 `<form>` 中的要求，将资料通过 **POST** 请求的方式发给 `web` 程序。
3. `Web` 程序运行了 `post '/hello'` 而不是 `get '/hello/'` 来处理这个请求。
4. 这个 `post '/hello'` 完成了它正常的功能，将 `hello` 页面返回，这里并没有新的东西，只是一个新函数名称而已。

作为练习，在 `lib/views/index.erb` 中添加一个链接，让它指向 `/hello`，这样你可以反复填写并提交表单查看结果。确认你可以解释清楚这个链接的工作原理，以及它是如何让你实现在 `lib/views/index.erb` 和 `lib/views/hello_form.erb` 之间循环跳转的，还有就是要明白你新修改过的 `Ruby` 代码，你需要知道在什么情况下会运行到哪一部分代码。

Creating A Layout Template

在你下一节练习建立游戏的过程中，你需要建立很多的小 **HTML** 页面。如果你每次都写一个完整的网页，你会很快感觉到厌烦的。幸运的是你可以建立一个「外观 (`layout`)」模板，也就是一种提供了通用的 `headers` 和 `footers` 的外壳模板，你可以用它将你所有的其他网页包裹起来。好开发人员会尽可能减少重复动作，所以要做一个好开发人员，使用外观模板是很重要的。

将 `lib/views/index.erb` 修改成这样：

```
<% if greeting %>

  <p>I just wanted to say <em style="color: green; font-size:
2em;"><%= greeting %></em>.

<% else %>

  <em>Hello</em>, world!

<% end %>
```

然后把 `lib/views/hello_form.erb` 修改成这样：

```
<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>
```

面这些修改的目的，是将每一个页面顶部和底部的反复用到的「样板 (boilerplate)」代码剥掉。这些被剥掉的代码会被放到一个单独的 `lib/views/layout.erb` 文件中，从此以后，这些反复用到的代码就由 `lib/views/layout.erb` 来提供了。

上面的都改好以后，建立一个 `lib/views/layout.erb` 文件，内容如下：

```
<html>

  <head>

    <title>Gothons From Planet Percal #25</title>

  </head>

  <body>

    <%= yield %>

  </body>

</html>
```

Sinatra 预设会自动去找名字为 `layout` 的外观模板，并且使用它作为其他模板的「基础」模板。你也可以修改已经用作任何页面的基础模板的 `template`。重启你的程序观察一下，然后试着用各种方法修改你的 `layout` 模板，不要修改你别的模板，看看输出会有什么样的变化。

为表单撰写自动测试代码

使用浏览器测试 web 程序是很容易的，只要点刷新按钮就可以了。不过毕竟我们是开发人员嘛，如果我们可以写一些代码来测试我们的程序，为什么还要重复手动测试呢？接下来你要做的，就是为你的 web 程序写一个小测试。这会用到你在《练习 47》学过的一些东西，如果你不记得的话，可以回去复习一下。

我已经为此建立了一个简单的小函数，让你判断(assert) web 程序的响应，这个函数有一个很合适的名字，就叫 `assert_response`。创建一个 `tests/tools.rb` 文件，内容如下：

```
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
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
end
```

最后，执行 `test/test_gothonsweb.rb` 去测试你的程序：


```
$ 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` 函数库包含了一串很简单的 API 可以让你处理请求。他们是 `get`, `put`, `post`, `delete` 和 `head` 函数，模拟程序会遇到的各类类型请求。

所有假的 (mock) `request` 函数会有一样的参数模式：

```
get '/path', params={}, rack_env={}
```

- `/path` 是 `request` 路径，而且可以选择性的包含一个 `query string`。
- `params` 是一组 `query/post` 的 Hash 参数，一个 `request body` 字串，或者是 `nil`
- `rack_env` 是一个 Rack 环境值 Hash。这可以用来设置 `request` 的 `header` 和其他 `request` 相关的信息，例如 `session` 内的资料。

这样的运作方式就不用实际运作一个真的 **web** 服务器，如此一来你就可以使用自动化测试代码去测试，当然同时你也可以使用浏览器去测试一个执行中的服务器。

为了验证(validate) 函数的响应，你需要使用 `test/tools.rb` 中定义的 `assert_response` 函数，里面内容是：

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)
```

把你调用 `get` 或 `post` 得到的响应传递给这个函数，然后将你要检查的内容作为参数传递给这个函数。你可以使用 `contains` 参数来检查响应中是否包含指定的值，使用 `status` 参数可以检查指定的响应状态。这个小函数其实包含了很多的信息，所以你还是自己研究一下的比较好。

在 `test/test_gothonsweb.rb` 自动测试脚本中，我首先确认 `/foo` URL 传回了一个「404 Not Found」响应，因为这个 URL 其实是不存在的。然后我检查了 `/hello` 在 GET 和 POST 两种请求的情况下都能正常运作。就算你没有弄明白测试的原理，这些测试代码应该是很好读懂的。

花一些时间研究一下这个最新版的 **web** 程序，重点研究一下自动测试的运作原理。

加分练习

1. 阅读和 HTML 相关的更多资料，然后为你的表单设计一个更好的输出格式。你可以先在纸上设计出来，然后用 HTML 去实现它。

2. 这是一道难题，试着研究一下如何进行文件上传，通过网页上传一张图片，然后将其保存到磁盘中。
3. 更难的难题，找到 **HTTP RFC** 文件（讲述 **HTTP** 运作原理的技术文件），然后努力阅读一下。这是一篇很无趣的文件，不过偶尔你会用到里边的一些知识。
4. 又是一道难题，找人帮你设置一个 **web** 服务器，例如 **Apache**、**Nginx**、或者 **thttpd**。试着让服务器伺候一下你建立的 **.html** 和 **.css** 文件。如果失败了也没关系，**web** 服务器本来就都有点烂。
5. 完成上面的任务后休息一下，然后试着多建立一些 **web** 程序出来。你应该仔细阅读 **Sinatra** 中关于会话(**session**)的内容，这样你可以明白如何存留使用者的状态信息。