Learn Ruby The Hard Way

Release 2.0

Zed A. Shaw and Rob Sobers

January 21, 2012

CONTENTS

Welcome to Learn Ruby the hard way. This is a translation of "Learn Python The Hard Way" to teach total beginners Ruby. It's in the same style, and the content is nearly the same, but it will teach you Ruby. If you have problems email help@learncodethehardway.org.

The Hard Way Is Easier

This simple book is meant to get you started in programming. The title says it's the hard way to learn to write code; but it's actually not. It's only the "hard" way because it's the way people *used* to teach things. With the help of this book, you will do the incredibly simple things that all programmers need to do to learn a language:

1. Go through each exercise.
2. Type in each sample exactly.
3. Make it run.

That's it. This will be *very* difficult at first, but stick with it. If you go through this book, and do each exercise for one or two hours a night, you will have a good foundation for moving onto another book. You might not really learn "programming" from this book, but you will learn the foundation skills you need to start learning the language.

This book's job is to teach you the three most essential skills that a beginning programmer needs to know: Reading and Writing, Attention to Detail, Spotting Differences.

Reading and Writing

It seems stupidly obvious, but, if you have a problem typing, you will have a problem learning to code. Especially if you have a problem typing the fairly odd characters in source code. Without this simple skill you will be unable to learn even the most basic things about how software works.

Typing the code samples and getting them to run will help you learn the names of the symbols, get familiar with typing them, and get you reading the language.

Attention to Detail

The one skill that separates bad programmers from good programmers is attention to detail. In fact, it's what separates the good from the bad in any profession. Without paying attention to the tiniest details of your work, you will miss key elements of what you create. In programming, this is how you end up with bugs and difficult-to-use systems.

By going through this book, and copying each example *exactly*, you will be training your brain to focus on the details of what you are doing, as you are doing it.

Spotting Differences

A very important skill – that most programmers develop over time – is the ability to visually notice differences between things. An experienced programmer can take two pieces of code that are slightly different and immediately start

pointing out the differences. Programmers have invented tools to make this even easier, but we won't be using any of these. You first have to train your brain the hard way, then you can use the tools.

While you do these exercises, typing each one in, you will be making mistakes. It's inevitable; even seasoned programmers would make a few. Your job is to compare what you have written to what's required, and fix all the differences. By doing so, you will train yourself to notice mistakes, bugs, and other problems.

Do Not Copy-Paste

You must *type* each of these exercises in, manually. If you copy and paste, you might as well just not even do them. The point of these exercises is to train your hands, your brain, and your mind in how to read, write, and see code. If you copy-paste, you are cheating yourself out of the effectiveness of the lessons.

A Note On Practice And Persistence

While you are studying programming, I'm studying how to play guitar. I practice it every day for at least 2 hours a day. I play scales, chords, and arpeggios for an hour at least and then learn music theory, ear training, songs and anything else I can. Some days I study guitar and music for 8 hours because I feel like it and it's fun. To me repetitive practice is natural and just how to learn something. I know that to get good at anything you have to practice every day, even if I suck that day (which is often) or it's difficult. Keep trying and eventually it'll be easier and fun.

As you study this book, and continue with programming, remember that anything worth doing is difficult at first. Maybe you are the kind of person who is afraid of failure so you give up at the first sign of difficulty. Maybe you never learned self-discipline so you can't do anything that's "boring". Maybe you were told that you are "gifted" so you never attempt anything that might make you seem stupid or not a prodigy. Maybe you are competitive and unfairly compare yourself to someone like me who's been programming for 20+ years.

Whatever your reason for wanting to quit, keep at it. Force yourself. If you run into an Extra Credit you can't do, or a lesson you just do not understand, then skip it and come back to it later. Just keep going because with programming there's this very odd thing that happens.

At first, you will not understand anything. It'll be weird, just like with learning any human language. You will struggle with words, and not know what symbols are what, and it'll all be very confusing. Then one day *BANG* your brain will snap and you will suddenly "get it". If you keep doing the exercises and keep trying to understand them, you will get it. You might not be a master coder, but you will at least understand how programming works.

If you give up, you won't ever reach this point. You will hit the first confusing thing (which is everything at first) and then stop. If you keep trying, keep typing it in, trying to understand it and reading about it, you will eventually get it.

But, if you go through this whole book, and you still do not understand how to code, at least you gave it a shot. You can say you tried your best and a little more and it didn't work out, but at least you tried. You can be proud of that.

License

Special Thanks

I'd like to thank a few people who helped with this edition of the book. First is my editor at *Pretty Girl Editing Services* who helped me edit the book and is just lovely all by herself. Then there's *Greg Newman*, who did the cover jacket and artwork, plus reviewed copies of the book. His artwork made the book look like a real book, and didn't mind that I totally forgot to give him credit in the first edition. I'd also like to thank *Brian Shumate* for doing the website landing page and other site design help, which I need a lot of help on.

Finally, I'd like to thank the hundreds of thousands of people who read the first edition and especially the ones who submitted bug reports and comments to improve the book. It really made this edition solid and I couldn't have done it without all of you. Thank you.

Exercise 0: The Setup

This exercise has no code. It is simply the exercise you complete to get your computer setup to run Ruby. You should follow these instructions as exactly as possible.

This tutorial assumes that you are using Ruby version 1.9.2.

Your system might already have Ruby installed. Open up a console and try running:

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

If you don't already have Ruby installed on your system, I highly recommend using Ruby Version Manager (RVM) to install it, regardless of which OS you are running.

Mac OSX

To complete this exercise, complete the following tasks:

1. Go to http://learnpythonthehardway.org/exercise0.html with your browser, get the `gedit` text editor, and install it.
2. Put gedit (your editor) in your Dock so you can reach it easily.
   (a) Run gedit so we can fix some stupid defaults it has.
   (b) Open `Preferences` from the `gedit menu` and select the `Editor` tab.
   (c) Change `Tab width`: to 2.
   (d) Select (make sure a check mark is in) `Insert spaces instead of tabs`.
   (e) Turn on "Automatic indentation" as well.
   (f) Open the `View` tab and turn on "Display line numbers".
3. Find your "Terminal" program. Search for it. You will find it.
4. Put your Terminal in your Dock as well.
5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `irb` (Interactive Ruby). You run things in Terminal by just typing their name and hitting RETURN.
   (a) If you run `irb` and it's not there (`irb` is not recognized..). Install it using Ruby Version Manager (RVM).
7. Hit CTRL-D (`^D`) and get out of `irb`.
8. You should be back at a prompt similar to what you had before you typed `irb`. If not find out why.
9. Learn how to make a directory in the Terminal. Search online for help.

10. Learn how to change into a directory in the Terminal. Again search online.
11. Use your editor to create a file in this directory. You will make the file, "Save" or "Save As...", and pick this directory.
12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can't figure it out.
13. Back in Terminal, see if you can list the directory to see your newly created file. Search online for how to list a directory.

Note: If you have problems with gedit, which is possible with non-English keyboard layouts, then I suggest you try Textwrangler found at http://www.barebones.com/products/textwrangler/ instead.

OSX: What You Should See

Here's me doing the above on my computer in Terminal. Your computer would be different, so see if you can figure out all the differences between what I did and what you should do.

```
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. Go to http://learnpythonthehardway.org/exercise0.html with your browser, get the `gedit` text editor, and install it. You do not need to be administrator to do this.
2. Make sure you can get to gedit easily by putting it on your desktop and/or in Quick Launch. Both options are available during setup.
   (a) Run gedit so we can fix some stupid defaults it has.
   (b) Open `Edit->Preferences` select the `Editor` tab.
   (c) Change `Tab width`: to 2.
   (d) Select (make sure a check mark is in) `Insert spaces instead of tabs`.
   (e) Turn on "Automatic indentation" as well.
   (f) Open the View tab turn on "Display line numbers".
3. Find your "Terminal" program. It's called `Command Prompt`. Alternatively just run `cmd`.
4. Make a shortcut to it on your desktop and/or `Quick Launch` for your convenience.

5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `irb` (Interactive Ruby). You run things in Terminal by just typing their name and hitting RETURN.
   (a) If you run `irb` and it's not there (`irb` is not recognized..). Install it using Ruby Version Manager (RVM).
7. Hit CTRL-Z (^Z), Enter and get out of `irb`.
8. You should be back at a prompt similar to what you had before you typed `irb`. If not find out why.
9. Learn how to make a directory in the Terminal. Search online for help.
10. Learn how to change into a directory in the Terminal. Again search online.
11. Use your editor to create a file in this directory. Make the file, "Save" or "Save As...", and pick this directory.
12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can't figure it out.
13. Back in Terminal, see if you can list the directory to see your newly created file. Search online for how to list a directory.

Warning: Windows is a big problem for Ruby. Sometimes you install Ruby and one computer will have no problems, and another computer will be missing important features. If you have problems, please visit: http://rubyinstaller.org/

Windows: What You Should See

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

C:\Documents and Settings\you>mkdir mystuff

C:\Documents and Settings\you>cd mystuff

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

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

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

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

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

You will probably see a very different prompt, Ruby information, and other stuff but this is the general idea. If your system is different let us know and we'll fix it.

Linux

Linux is a varied operating system with a bunch of different ways to install software. I'm assuming if you are running Linux then you know how to install packages so here are your instructions:

1. Go to http://learnpythonthehardway.org/wiki/ExerciseZero with your browser, get the `gedit` text editor, and install it.
2. Make sure you can get to gedit easily by putting it in your window manager's menu.
   (a) Run gedit so we can fix some stupid defaults it has.
   (b) Open `Preferences` select the `Editor` tab.
   (c) Change `Tab width`: to 2.
   (d) Select (make sure a check mark is in) `Insert spaces instead of tabs`.
   (e) Turn on "Automatic indentation" as well.
   (f) Open the `View` tab turn on "Display line numbers".
3. Find your "Terminal" program. It could be called GNOME Terminal, Konsole, or xterm.
4. Put your Terminal in your Dock as well.
5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `irb` (Interactive Ruby). You run things in Terminal by just typing their name and hitting RETURN.
   (a) If you run `irb` and it's not there (`irb` is not recognized..). Install it using Ruby Version Manager (RVM).
7. Hit CTRL-D (`^D`) and get out of `irb`.
8. You should be back at a prompt similar to what you had before you typed `irb`. If not find out why.
9. Learn how to make a directory in the Terminal. Search online for help.
10. Learn how to change into a directory in the Terminal. Again search online.
11. Use your editor to create a file in this directory. Typically you will make the file, "Save" or "Save As..", and pick this directory.
12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can't figure it out.
13. Back in Terminal see if you can list the directory to see your newly created file. Search online for how to list a directory.

Linux: What You Should See

```
$ 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
$
```

You will probably see a very different prompt, Ruby information, and other stuff but this is the general idea.

Warnings For Beginners

You are done with this exercise. This exercise might be hard for you depending on your familiarity with your computer. If it is difficult, take the time to read and study and get through it, because until you can do these very basic things you will find it difficult to get much programming done.

If a programmer tells you to use `vim` or `emacs`, tell them, "No." These editors are for when you are a better programmer. All you need right now is an editor that lets you put text into a file. We will use `gedit` because it is simple and the same on all computers. Professional programmers use `gedit` so it's good enough for you starting out.

A programmer will eventually tell you to use Mac OSX or Linux. If the programmer likes fonts and typography, they'll tell you to get a Mac OSX computer. If they like control and have a huge beard, they'll tell you to install Linux. Again, use whatever computer you have right now that works. All you need is `gedit`, a Terminal, and Ruby.

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

1. *Write* exercises using `gedit`.
2. *Run* the exercises you wrote.
3. *Fix* them when they are broken.
4. Repeat.

Anything else will only confuse you, so stick to the plan.

Exercise 1: A Good First Program

Remember, you should have spent a good amount of time in Exercise 0 learning how to install a text editor, run the text editor, run the Terminal, and work with both of them. If you haven't done that then do not go on. You will not have a good time. This is the only time I'll start an exercise with a warning that you should not skip or get ahead of yourself.

```
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.'
```

Type the above into a single file named `ex1.rb`. This is important as Ruby works best with files ending in `.rb`.

Then in Terminal run the file by typing:

```
ruby ex1.rb
```

If you did it right then you should see the same output I have below. If not, you have done something wrong. No, the computer is not wrong.

What You Should See

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

You may see the name of your directory before the `$` which is fine, but if your output is not exactly the same, find out why and fix it.

If you have an error it will look like this:

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

It's important that you can read these since you will be making many of these mistakes. Even I make many of these mistakes. Let's look at this line-by-line.

1. Here we ran our command in the terminal to run the ex1.rb script.
2. Ruby then tells us that the file ex1.rb has an error on line 4.
3. It then prints this line for us.
4. Then it puts a ^ (caret) character to point at where the problem is.
5. Finally, it prints out a "syntax error" and tells us something about what might be the error. Usually these are very cryptic, but if you copy that text into a search engine, you will find someone else who's had that error and you can probably figure out how to fix it.

Extra Credit

You will also have Extra Credit. The Extra Credit contains things you should try to do. If you can't, skip it and come back later.

For this exercise, try these things:

1. Make your script print another line.
2. Make your script print only one of the lines.
3. Put a # (octothorpe) character at the beginning of a line. What did it do? Try to find out what this character does.
4. From now on, I won't explain how each exercise works unless an exercise is different.

Note: An 'octothorpe' is also called a 'pound', 'hash', 'mesh', or any number of names. Pick the one that makes you chill out.

Exercise 2: Comments And Pound Characters

Comments are very important in your programs. They are used to tell you what something does in English, and they also are used to disable parts of your program if you need to remove them temporarily. Here's how you use comments in Ruby:

```
# 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:
# puts "This won't run."

puts "This will run."
```

What You Should See

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

Extra Credit

1. Find out if you were right about what the # character does and make sure you know what it's called (octothorpe or pound character).
2. Take your `ex2.rb` file and review each line going backwards. Start at the last line, and check each word in reverse against what you should have typed.
3. Did you find more mistakes? Fix them.
4. Read what you typed above out loud, including saying each character by its name. Did you find more mistakes? Fix them.

Exercise 3: Numbers And Math

Every programming language has some kind of way of doing numbers and math. Do not worry, programmers lie frequently about being math geniuses when they really aren't. If they were math geniuses, they would be doing math, not writing ads and social network games to steal people's money.

This exercise has lots of math symbols. Let's name them right away so you know what they are called. As you type this one in, say the names. When saying them feels boring you can stop saying them. Here are the names:

```
+ plus
- minus
/ slash
* asterisk
% percent
< less-than
> greater-than
<= less-than-equal
>= greater-than-equal
```

Notice how the operations are missing? After you type in the code for this exercise, go back and figure out what each of these does and complete the table. For example, + does addition.

```
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
```

What You Should See

```
$ 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
$
```

Extra Credit

1. Above each line, use the # to write a comment to yourself explaining what the line does.
2. Remember in Exercise 0 when you started IRB? Start IRB this way again and using the above characters and what you know, use Ruby as a calculator.
3. Find something you need to calculate and write a new `.rb` file that does it.
4. Notice the math seems "wrong"? There are no fractions, only whole numbers. Find out why by researching what a "floating point" number is.
5. Rewrite `ex3.rb` to use floating point numbers so it's more accurate (hint: 20.0 is floating point).

Exercise 4: Variables And Names

Now you can print things with `puts` and you can do math. The next step is to learn about variables. In programming a variable is nothing more than a name for something so you can use the name rather than the something as you code. Programmers use these variable names to make their code read more like English, and because they have lousy memories. If they didn't use good names for things in their software, they'd get lost when they tried to read their code again.

If you get stuck with this exercise, remember the tricks you have been taught so far of finding differences and focusing on details:

1. Write a comment above each line explaining to yourself what it does in English.
2. Read your `.rb` file backwards.
3. Read your `.rb` file out loud saying even the characters.

```
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: The _ in `space_in_a_car` is called an underscore character. Find out how to type it if you do not already know. We use this character a lot to put an imaginary space between words in variable names.

What You Should See

```
$ 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.
$
```

Extra Credit

When I wrote this program the first time I had a mistake, and Ruby told me about it like this:

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

Explain this error in your own words. Make sure you use line numbers and explain why.

Here's more extra credit:

1. Explain why the 4.0 is used instead of just 4.
2. Remember that 4.0 is a "floating point" number. Find out what that means.
3. Write comments above each of the variable assignments.
4. Make sure you know what = is called (equals) and that it's making names for things.
5. Remember _ is an underscore character.
6. Try running IRB as a calculator like you did before and use variable names to do your calculations. Popular variable names are also `i`, `x`, and `j`.

Exercise 5: More Variables And Printing

Now we'll do even more typing of variables and printing them out. This time we'll use something called a "format string". Every time you put " (double-quotes) around a piece of text you have been making a string. A string is how you make something that your program might give to a human. You print them, save them to files, send them to web servers, all sorts of things.

Strings are really handy, so in this exercise you will learn how to make strings that have variables embedded in them. You embed variables inside a string by using specialized format sequences and then putting the variables at the end with a special syntax that tells Ruby, "Hey, this is a format string, put these variables in there."

As usual, just type this in even if you do not understand it and make it exactly the same.

```
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]
```

What You Should See

```
$ 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.
$
```

Extra Credit

1. Change all the variables so there isn't the `my_` in front. Make sure you change the name everywhere, not just where you used = to set them.
2. Try more format sequences.
3. Search online for all of the Ruby format sequences.
4. Try to write some variables that convert the inches and pounds to centimeters and kilos. Do not just type in the measurements. Work out the math in Ruby.

Exercise 6: Strings And Text

While you have already been writing strings, you still do not know what they do. In this exercise we create a bunch of variables with complex strings so you can see what they are for. First an explanation of strings.

A string is usually a bit of text you want to display to someone, or "export" out of the program you are writing. Ruby knows you want something to be a string when you put either `"` (double-quotes) or `'` (single-quotes) around the text. You saw this many times with your use of `puts` when you put the text you want to go to the string inside `"` or `'` after the `puts`. Then Ruby displays it.

Strings may contain the format characters you have discovered so far. You simply put the formatted variables in the string, and then a `%` (percent) character, followed by the variable. The only catch is that if you want multiple formats in your string to print multiple variables, you need to put them inside `[ ]` (brackets) separated by `,` (commas). It's as if you were telling me to buy you a list of items from the store and you said, "I want milk, eggs, bread, and soup." Only as a programmer we say, "[milk, eggs, bread, soup]".

Another way of injecting variables into your strings is to use something called "string interpolation", which uses the `#{ }` (pound and curly brace) characters. So, instead of using format strings:

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

We can type:

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

We will now type in a whole bunch of strings, variables, formats, and print them. You will also practice using short abbreviated variable names. Programmers love saving themselves time at your expense by using annoying cryptic variable names, so let's get you started being able to read and write them early on.

```
1  x = "There are #{10} types of people."
2  binary = "binary"
3  do_not = "don't"
4  y = "Those who know #{binary} and those who #{do_not}."
5
6  puts x
7  puts y
8
9  puts "I said: #{x}."
10 puts "I also said: '#{y}'."
11
12 hilarious = false
13 joke_evaluation = "Isn't that joke so funny?! #{hilarious}"
14
15 puts joke_evaluation
16
17 w = "This is the left side of..."
18 e = "a string with a right side."
19
20 puts w + e
```

What You Should See

```
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.
```

Extra Credit

1. Go through this program and write a comment above each line explaining it.
2. Find all the places where a string is put inside a string. There are four places.
3. Are you sure there's only four places? How do you know? Maybe I like lying.
4. Explain why adding the two strings `w` and `e` with + makes a longer string.

Exercise 7: More Printing

Now we are going to do a bunch of exercises where you just type code in and make it run. I won't be explaining much since it is just more of the same. The purpose is to build up your chops. See you in a few exercises, and do not skip! Do not paste!

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

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

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

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

What You Should See

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

Extra Credit

For these next few exercises, you will have the exact same extra credit.

1. Go back through and write a comment on what each line does.
2. Read each one backwards or out loud to find your errors.
3. From now on, when you make mistakes write down on a piece of paper what kind of mistake you made.
4. When you go to the next exercise, look at the last mistakes you made and try not to make them in this new one.
5. Remember that everyone makes mistakes. Programmers are like magicians who like everyone to think they are perfect and never wrong, but it's all an act. They make mistakes all the time.

Exercise 8: Printing, Printing

```
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."
]
```

What You Should See

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

Extra Credit

1. Do your checks of your work, write down your mistakes, try not to make them on the next exercise.

Exercise 9: Printing, Printing, Printing

```
# 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 PARAGRAPH thing
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
PARAGRAPH
```

What You Should See

```
$ 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 PARAGRAPH thing
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
```

Extra Credit

Do your checks of your work, write down your mistakes, try not to make them on the next exercise.

Exercise 10: What Was That?

In Exercise 9 I threw you some new stuff, just to keep you on your toes. I showed you two ways to make a string that goes across multiple lines. In the first way, I put the characters `\n` (back-slash n) between the names of the months. What these two characters do is put a new line character into the string at that point.

This use of the `\` (back-slash) character is a way we can put difficult-to-type characters into a string. There are plenty of these "escape sequences" available for different characters you might want to put in, but there's a special one, the double back-slash which is just two of them `\\`. These two characters will print just one back-slash. We'll try a few of these sequences so you can see what I mean.

Another important escape sequence is to escape a single-quote `'` or double-quote `"`. Imagine you have a string that uses double-quotes and you want to put a double-quote in for the output. If you do this `"I "understand" joe."` then Ruby will get confused since it will think the `"` around `"understand"` actually ends the string. You need a way to tell Ruby that the `"` inside the string isn't a real double-quote.

To solve this problem you escape double-quotes and single-quotes so Ruby knows to include in the string. Here's an example:

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

The second way is by using here document syntax, which uses `<<NAME` and works like a string, but you also can put as many lines of text you as want until you type `NAME` again. We'll also play with these.

```
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
```

What You Should See

Look for the tab characters that you made. In this exercise the spacing is important to get right.

```
$ 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

$
```

Extra Credit

1. Search online to see what other escape sequences are available.
2. Combine escape sequences and format strings to create a more complex format.

Exercise 11: Asking Questions

Now it is time to pick up the pace. I have got you doing a lot of printing so that you get used to typing simple things, but those simple things are fairly boring. What we want to do now is get data into your programs. This is a little tricky because you have learn to do two things that may not make sense right away, but trust me and do it anyway. It will make sense in a few exercises.

Most of what software does is the following:

1. Take some kind of input from a person.
2. Change it.
3. Print out something to show how it changed.

So far you have only been printing, but you haven't been able to get any input from a person, or change it. You may not even know what "input" means, so rather than talk about it, let's have you do some and see if you get it. Next exercise we'll do more to explain it.

```
1 print "How old are you? "
2 age = gets.chomp()
3 print "How tall are you? "
4 height = gets.chomp()
5 print "How much do you weigh? "
6 weight = gets.chomp()
7
8 puts "So, you're #{age} old, #{height} tall and #{weight} heavy."
```

Note: Notice that we are using `print` instead of `puts` to do the prompting. `print` doesn't add a new line automatically, so your answer can go on the same line as the question. `puts` on the other hand, adds a newline automatically.

What You Should See

```
$ 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.
$
```

Extra Credit

1. Go online and find out what Rubys `gets` and `chomp` methods do.
2. Can you find other ways to use `gets.chomp`? Try some of the samples you find.
3. Write another "form" like this to ask some other questions.

Exercise 12: Libraries

Take a look at this code:

```
1  require 'open-uri'
2
3  open("http://www.ruby-lang.org/en") do |f|
4    f.each_line {|line| p line}
5    puts f.base_uri         # <URI::HTTP:0x40e6ef2 URL:http://www.ruby-lang.org/en/>
6    puts f.content_type     # "text/html"
7    puts f.charset          # "iso-8859-1"
8    puts f.content_encoding # []
9    puts f.last_modified    # Thu Dec 05 02:45:02 UTC 2002
10 end
```

On line 1 we have what's called a "require". This is how you add features to your script from the Ruby feature set or other sources (e.g., Ruby Gems, stuff you wrote yourself). Rather than give you all the features at once, Ruby asks you to say what you plan to use. This keeps your programs small, but it also acts as documentation for other programmers who read your code later.

Hold Up! Features Have Another Name

I call them "features" here (these little things you require to make your Ruby program do more) but nobody else calls them features. I just used that name because I needed to trick you into learning what they are without jargon. Before you can continue, you need to learn their real name: `libraries`.

From now on we will be calling these "features" that we require `libraries`. I'll say things like, "You want to require the `open-uri` library." They are also called "modules" by other programmers, but let's just stick with `libraries`.

Extra Credit

1. Research the difference between `require` and `include`. How are they different?
2. Can you `require` a script that doesn't contain a `library` specifically?
3. Figure out which directories on your system Ruby will look in to find the libraries you require.

Exercise 13: Parameters, Unpacking, Variables

In this exercise we will cover one more input method you can use to pass variables to a script (script being another name for your `.rb` files). You know how you type `ruby ex13.rb` to run the `ex13.rb` file? Well the `ex13.rb` part of the command is called an "argument". What we'll do now is write a script that also accepts arguments.

Type this program and I'll explain it in detail:

```
1 first, second, third = ARGV
2
3 puts "The script is called: #{$0}"
4 puts "Your first variable is: #{first}"
5 puts "Your second variable is: #{second}"
6 puts "Your third variable is: #{third}"
```

The `ARGV` is the "argument variable", a very standard name in programming, that you will find used in many other languages. It's in all caps because it's a constant, meaning you shouldn't change the value once it's been assigned. This variable holds the arguments you pass to your Ruby script when you run it. In the exercises you will get to play with this more and see what happens.

Line 1 "unpacks" `ARGV` so that, rather than holding all the arguments, it gets assigned to three variables you can work with: `first`, `second`, and `third`. The name of the script itself is stored in a special variable `$0`, which we don't need to unpack. This may look strange, but "unpack" is probably the best word to describe what it does. It just says, "Take whatever is in `ARGV`, unpack it, and assign it to all of these variables on the left in order."

After that we just print them out like normal.

What You Should See

Run the program like this:

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

This is what you should see when you do a few different runs with different arguments:

```
$ 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
```

You can actually replace "first", "2nd", and "3rd" with any three things. You do not have to give these parameters either, you can give any 3 strings you want:

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

Extra Credit

1. Try giving fewer than three arguments to your script. What values are used for the missing arguments?
2. Write a script that has fewer arguments and one that has more. Make sure you give the unpacked variables good names.
3. Combine `gets.chomp()` with `ARGV` to make a script that gets more input from a user.

Exercise 14: Prompting And Passing

Let's do one exercise that uses `ARGV` and `gets.chomp()` together to ask the user something specific. You will need this for the next exercise where we learn to read and write files. In this exercise we'll print a simple > prompt. This is similar to a game like Zork or Adventure.

```
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
```

Important: Also notice that we're using `STDIN.gets` instead of plain 'ol `gets`. That is because if there is stuff in `ARGV`, the default `gets` method tries to treat the first one as a file and read from that. To read from the user's input (i.e., `stdin`) in such a situation, you have to use it `STDIN.gets` explicitly.

What You Should See

When you run this, remember that you have to give the script your name for the `ARGV` arguments.

```
$ ruby ex14.rb Zed
Hi Zed, I'm the ex/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.
```

Extra Credit

1. Find out what Zork and Adventure were. Try to find a copy and play it.
2. Change the `prompt` variable to something else entirely.
3. Add another argument and use it in your script.
4. Make sure you understand how I combined a `<<SOMETHING` style multi-line string with `#{ }` string interpolation as the last print.

Exercise 15: Reading Files

Everything you've learned about `STDIN.gets` and `ARGV` is so you can start reading files. You may have to play with this exercise the most to understand what's going on, so do it carefully and remember your checks. Working with files is an easy way to erase your work if you are not careful.

This exercise involves writing two files. One is your usual `ex15.rb` file that you will run, but the other is named `ex15_sample.txt`. This second file isn't a script but a plain text file we'll be reading in our script. Here are the contents of that file:

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

What we want to do is "open" that file in our script and print it out. However, we do not want to just "hard code" the name `ex15_sample.txt` into our script. "Hard coding" means putting some bit of information that should come from the user as a string right in our program. That's bad because we want it to load other files later. The solution is to use `ARGV` and `STDIN.gets` to ask the user what file they want instead of "hard coding" the file's name.

```
1  filename = ARGV.first
2
3  prompt = "> "
4  txt = File.open(filename)
5
6  puts "Here's your file: #{filename}"
7  puts txt.read()
8
9  puts "I'll also ask you to type it again:"
10 print prompt
11 file_again = STDIN.gets.chomp()
12
13 txt_again = File.open(file_again)
14
15 puts txt_again.read()
```

A few fancy things are going on in this file, so let's break it down real quick:

Line 1-3 should be a familiar use of `ARGV` to get a filename and setting up the prompt. Next we have line 4 where we use a new command `File.open`. Right now, run `ri File.open` from the command line and read the instructions. Notice how like your own scripts, it takes a parameter and returns a value you can set to your own variable. You just opened a file.

Line 6 we print a little line, but on line 7 we have something very new and exciting. We call a function on `txt`. What you got back from `open` is a `file`, and it's also got commands you can give it. You give a file a command by using the `.` (dot or period), the name of the command, and parameters. Just like with `File.open`. The difference is that when you say `txt.read()` you are saying, "Hey txt! Do your read command with no parameters!"

The remainder of the file is more of the same, but we'll leave the analysis to you in the extra credit.

What You Should See

I made a file called "ex15_sample.txt" and ran my script.

```
$ 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.

$
```

Extra Credit

This is a big jump so be sure you do this extra credit as best you can before moving on.

1. Above each line write out in English what that line does.
2. If you are not sure ask someone for help or search online. Many times searching for "ruby THING" will find answers for what that THING does in Ruby. Try searching for "ruby file.open".
3. I used the name "commands" here, but they are also called "functions" and "methods". Search around online to see what other people do to define these. Do not worry if they confuse you. It's normal for a programmer to confuse you with their vast extensive knowledge.
4. Get rid of the part from line 9-15 where you use `STDIN.gets` and try the script then.
5. Use only `STDIN.gets` and try the script that way. Think of why one way of getting the filename would be better than another.
6. Run `ri File` and scroll down until you see the `read()` command (method/function). See all the other ones you can use? Try some of the other commands.
7. Startup IRB again and use `File.open` from the prompt. Notice how you can open files and run read on them right there?
8. Have your script also do a `close()` on the `txt` and `txt_again` variables. It's important to close files when you are done with them.

Exercise 16: Reading And Writing Files

If you did the extra credit from the last exercise you should have seen all sorts of commands (methods/functions) you can give to files. Here's the list of commands I want you to remember:

- close – Closes the file. Like `File->Save..` in your editor.
- read – Reads the contents of the file, you can assign the result to a variable.
- readline – Reads just one line of a text file.
- truncate – Empties the file, watch out if you care about the file.
- write(stuff) – Writes stuff to the file.

For now these are the important commands you need to know. Some of them take parameters, but we do not really care about that. You only need to remember that write takes a parameter of a string you want to write to the file.

Let's use some of this to make a simple little text editor:

```
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")
```

```
31
32 puts "And finally, we close it."
33 target.close()
```

> **Warning:** If you get errors in this script it is probably because you are using Ruby 1.8 and the book assumes Ruby 1.9. To see what version you have type: `ruby -v` If you need to install a newer version then go back to the beginning of the book where Exercise 0 tells you how.

That's a large file, probably the largest you have typed in. So go slow, do your checks, and make it run. One trick is to get bits of it running at a time. Get lines 1-8 running, then 5 more, then a few more, etc., until it's all done and running.

What You Should See

There are actually two things you will see, first the output of your new script:

```
$ 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.
$
```

Now, open up the file you made (in my case test.txt) in your editor and check it out. Neat right?

Extra Credit

1. If you feel you do not understand this, go back through and use the comment trick to get it squared away in your mind. One simple English comment above each line will help you understand, or at least let you know what you need to research more.
2. Write a script similar to the last exercise that uses read and argv to read the file you just created.
3. There's too much repetition in this file. Use strings, formats, and escapes to print out line1, line2, and line3 with just one `target.write()` command instead of 6.
4. Find out why we had to pass a `'w'` as an extra parameter to open. Hint: open tries to be safe by making you explicitly say you want to write a file.
5. If you open the file with `'w'` mode, then do you really need the `target.truncate()`? Go read the docs for Ruby's `File.open` function and see if that's true.

Exercise 17: More Files

Now let's do a few more things with files. We're going to actually write a Ruby script to copy one file to another. It'll be very short but will give you some ideas about other things you can do with files.

```
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()
```

Here we used a new method called `File.exists?`. This returns **true** if a file exists, based on its name in a string as an argument. It returns **false** if not. We'll be using this function in the second half of this book to do lots of things.

What You Should See

Just like your other scripts, run this one with two arguments, the file to copy from and the file to copy it to. If we use your `test.txt` file from before we get this:

```
$ 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.
$
```

It should work with any file. Try a bunch more and see what happens. Just be careful you do not blast an important file.

> **Warning:** Did you see that trick I did with `cat`? It only works on Linux or OSX, on Windows use `type` to do the same thing.

Extra Credit

1. Go read up on Ruby's `require` statement, and start Ruby to try it out. Try importing some things and see if you can get it right. It's alright if you do not.
2. This script is really annoying. There's no need to ask you before doing the copy, and it prints too much out to the screen. Try to make it more friendly to use by removing features.
3. See how short you can make the script. I could make this 1 line long.
4. Notice at the end of the WYSS I used something called `cat`? It's an old command that "con*cat*enates" files together, but mostly it's just an easy way to print a file to the screen. Type `man cat` to read about it.
5. Windows people, find the alternative to `cat` that Linux/OSX people have. Do not worry about `man` since there is nothing like that.
6. Find out why you had to do `output.close()` in the code.

Exercise 18: Names, Variables, Code, Functions

Big title right? I am about to introduce you to the function! Dum dum dah! Every programmer will go on and on about functions and all the different ideas about how they work and what they do, but I will give you the simplest explanation you can use right now.

Functions do three things:

1. They name pieces of code the way variables name strings and numbers.
2. They take arguments the way your scripts take `ARGV`.
3. Using #1 and #2 they let you make your own "mini scripts" or "tiny commands".

You can create a function (also called "methods") by using the word `def` in Ruby. I'm going to have you make four different functions that work like your scripts, and then show you how each one is related.

```
# 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()
```

Let's break down the first function, `puts_two` which is the most similar to what you already know from making scripts:

1. First we tell Ruby we want to make a function using `def` for "define".

2. On the same line as `def` we then give the function a name, in this case we just called it "puts_two" but it could be "peanuts" too. It doesn't matter, except that your function should have a short name that says what it does.
3. Then we tell it we want `*args` (asterisk args) which is a lot like your `ARGV` parameter but for functions.
4. After the definition, all the lines that are indented 2 spaces will become attached to this name, `puts_two`. Our first indented line is one that unpacks the arguments the same as with your scripts.
5. To demonstrate how it works we print these arguments out, just like we would in a script. Now, the problem with `puts_two` is that it's not the easiest way to make a function. In Ruby we can skip the whole unpacking args and just use the names we want right inside `()`. That's what `print_two_again` does.

After that you have an example of how you make a function that takes one argument in `puts_one`.

Finally you have a function that has no arguments in `puts_none`.

> **Warning:** This is very important. Do not get discouraged right now if this doesn't quite make sense. We're going to do a few exercises linking functions to your scripts and show you how to make more. For now just keep thinking "mini script" when I say "function" and keep playing with them.

What You Should See

If you run the above script you should see:

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

Right away you can see how a function works. Notice that you used your functions the way you use things like `File.exists?`, `File.open`, and other "commands". In fact, I've been tricking you because in Ruby those "commands" are just functions. This means you can make your own commands and use them in your scripts too.

Extra Credit

Write out a function checklist for later exercises. Write these on an index card and keep it by you while you complete the rest of these exercises or until you feel you do not need it:

1. Did you start your function definition with `def`?
2. Does your function name have only characters and _ (underscore) characters?
3. Did you put an open parenthesis `(` right after the function name?
4. Did you put your arguments after the parenthesis `(` separated by commas?
5. Did you make each argument unique (meaning no duplicated names).
6. Did you put a close parenthesis `)` after the arguments?
7. Did you indent all lines of code you want in the function 2 spaces?
8. Did you close your function body by typing "end"?

And when you run (aka "use" or "call") a function, check these things:

1. Did you call/use/run this function by typing its name?
2. Did you put (character after the name to run it? (this isn't required, but is idiomatic)
3. Did you put the values you want into the parenthesis separated by commas?
4. Did you end the function call with a) character.

Use these two checklists on the remaining lessons until you do not need them anymore.

Finally, repeat this a few times:

> "To 'run', 'call', or 'use' a function all mean the same thing."

Exercise 19: Functions And Variables

Functions may have been a mind-blowing amount of information, but do not worry. Just keep doing these exercises and going through your checklist from the last exercise and you will eventually get it.

There is one tiny point though that you might not have realized which we'll reinforce right now: The variables in your function are not connected to the variables in your script. Here's an exercise to get you thinking about this:

```
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)
```

This shows all different ways we're able to give our function `cheese_and_crackers` the values it needs to print them. We can give it straight numbers. We can give it variables. We can give it math. We can even combine math and variables.

In a way, the arguments to a function are kind of like our = character when we make a variable. In fact, if you can use = to name something, you can usually pass it to a function as an argument.

What You Should See

You should study the output of this script and compare it with what you think you should get for each of the examples in the script.

```
$ 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.
$
```

Extra Credit

1. Go back through the script and type a comment above each line explaining in English what it does.
2. Start at the bottom and read each line backwards, saying all the important characters.
3. Write at least one more function of your own design, and run it 10 different ways.

Exercise 20: Functions And Files

Remember your checklist for functions, then do this exercise paying close attention to how functions and files can work together to make useful stuff.

```
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)
```

Pay close attention to how we pass in the current line number each time we run `print_a_line`.

What You Should See

```
$ 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.

$
```

Extra Credit

1. Go through and write English comments for each line to understand what's going on.
2. Each time `print_a_line` is run you are passing in a variable `current_line`. Write out what `current_line` is equal to on each function call, and trace how it becomes `line_count` in `print_a_line`.
3. Find each place a function is used, and go check its def to make sure that you are giving it the right arguments.
4. Research online what the `seek` function for file does. Look at the `rdoc` documentation using the `ri` command and see if you can figure it out from there.
5. Research the shorthand notation += and rewrite the script to use that.

Exercise 21: Functions Can Return Something

You have been using the = character to name variables and set them to numbers or strings. We're now going to blow your mind again by showing you how to use = to set variables to be a value from a function. There will be one thing to pay close attention to, but first type this in:

```
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?"
```

We are now doing our own math functions for `add`, `subtract`, `multiply`, and `divide`. The important thing to notice is the last line where we say `a + b` (in `add`). What this does is the following:

1. Our function is called with two arguments: `a` and `b`.
2. We print out what our function is doing, in this case "ADDING".
3. Then we tell Ruby to do something kind of backward: we return the addition of `a + b`. You might say this as, "I add a and b then return them." In Ruby, the last evaluated statement in a method is its return value. You can be more explicit if you want and type `return a + b`, but that is totally optional.
4. Ruby adds the two numbers. Then when the function ends any line that runs it will be able to assign this a + b result to a variable.

As with many other things in this book, you should take this real slow, break it down and try to trace what's going on. To help there's extra credit to get you to solve a puzzle and learn something cool.

What You Should See

```
$ 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?
$
```

Extra Credit

1. If you aren't really sure what return values are, try writing a few of your own functions and have them return some values. You can return anything that you can put to the right of an =.
2. At the end of the script is a puzzle. I'm taking the return value of one function, and using it as the argument of another function. I'm doing this in a chain so that I'm kind of creating a formula using the functions. It looks really weird, but if you run the script you can see the results. What you should do is try to figure out the normal formula that would recreate this same set of operations.
3. Once you have the formula worked out for the puzzle, get in there and see what happens when you modify the parts of the functions. Try to change it on purpose to make another value.
4. Finally, do the inverse. Write out a simple formula and use the functions in the same way to calculate it.

This exercise might really whack your brain out, but take it slow and easy and treat it like a little game. Figuring out puzzles like this is what makes programming fun, so I'll be giving you more little problems like this as we go.

Exercise 22: What Do You Know So Far?

There won't be any code in this exercise or the next one, so there's no WYSS or Extra Credit either. In fact, this exercise is like one giant Extra Credit. I'm going to have you do a form of review what you have learned so far.

First, go back through every exercise you have done so far and write down every word and symbol (another name for 'character') that you have used. Make sure your list of symbols is complete.

Next to each word or symbol, write its name and what it does. If you can't find a name for a symbol in this book, then look for it online. If you do not know what a word or symbol does, then go read about it again and try using it in some code.

You may run into a few things you just can't find out or know, so just keep those on the list and be ready to look them up when you find them.

Once you have your list, spend a few days rewriting the list and double checking that it's correct. This may get boring but push through and really nail it down.

Once you have memorized the list and what they do, then you should step it up by writing out tables of symbols, their names, and what they do from memory. When you hit some you can't recall *from memory*, go back and memorize them again.

> **Warning:** The most important thing when doing this exercise is: "There is no failure, only trying."

What You are Learning

It's important when you are doing a boring mindless memorization exercise like this to know why. It helps you focus on a goal and know the purpose of all your efforts.

In this exercise you are learning the names of symbols so that you can read source code more easily. It's similar to learning the alphabet and basic words of English, except that Ruby's alphabet has extra symbols you might not know.

Just take it slow and do not hurt your brain. Hopefully by now these symbols are natural for you so this isn't a big effort. It's best to take 15 minutes at a time with your list and then take a break. Giving your brain a rest will help you learn faster with less frustration.

Exercise 23: Read Some Code

You should have spent last week getting your list of symbols straight and locked in your mind. Now you get to apply this to another week reading code on the internet. This exercise will be daunting at first. I'm going to throw you in the deep end for a few days and have you just try your best to read and understand some source code from real projects. The goal isn't to get you to understand code, but to teach you the following three skills:

1. Finding Ruby source code for things you need.
2. Reading through the code and looking for files.
3. Trying to understand code you find.
4. At your level you really do not have the skills to evaluate the things you find, but you can benefit from getting exposure and seeing how things look.

When you do this exercise, think of yourself as an anthropologist, trucking through a new land with just barely enough of the local language to get around and survive. Except, of course, that you will actually get out alive because the internet isn't a jungle. Anyway.

Here's what you do:

1. Go to github.com with your favorite web browser and search for "ruby".
2. Pick a random project and click on it.
3. Click on the Source tab and browse through the list of files and directories until you find a .rb file.
4. Start at the top and read through it, taking notes on what you think it does.
5. If any symbols or strange words seem to interest you, write them down to research later.

That's it. Your job is to use what you know so far and see if you can read the code and get a grasp of what it does. Try skimming the code first, and then read it in detail. Maybe also try taking very difficult parts and reading each symbol you know outloud.

Now try several other sites:

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

On each of these sites you may find weird files ending in .c so stick to .rb files like the ones you have written in this book.

A final fun thing to do is use the above four sources of Ruby code and type in topics you are interested in instead of "ruby". Search for "journalism", "cooking", "physics", or anything you are curious about. Chances are there's some code out there you could use right away.

Exercise 24: More Practice

You are getting to the end of this section. You should have enough Ruby "under your fingers" to move onto learning about how programming really works, but you should do some more practice. This exercise is longer and all about building up stamina. The next exercise will be similar. Do them, get them exactly right, and do your checks.

```
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)
```

What You Should See

```
$ 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.
$
```

Extra Credit

1. Make sure to do your checks: read it backwards, read it out loud, put comments above confusing parts.
2. Break the file on purpose, then run it to see what kinds of errors you get. Make sure you can fix it.

Exercise 25: Even More Practice

We're going to do some more practice involving functions and variables to make sure you know them well. This exercise should be straight forward for you to type in, break down, and understand.

However, this exercise is a little different. You won't be running it. Instead *you* will import it into your Ruby interpreter and run the functions yourself.

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

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

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

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

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

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

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

```
41      print_first_word(words)
42      print_last_word(words)
43    end
44  end
```

First, run this like normal with `ruby ex25.rb` to find any errors you have made. Once you have found all of the errors you can and fixed them, you will then want to follow the WYSS section to complete the exercise.

What You Should See

In this exercise we're going to interact with your `.rb` file inside the Ruby interpreter (IRB) you used periodically to do calculations.

Here's what it looks like when I do it:

```
$ 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
$
```

Let's break this down line by line to make sure you know what's going on:

1. Line 2 you require your `./ex25.rb` Ruby file, just like other requires you have done. Notice you do not need to put the `.rb` at the end to require it. When you do this you make a `module` that has all your functions in it to use.
2. Line 4 you made a `sentence` to work with.
3. Line 6 you use the `Ex25` module and call your first function `Ex25.break_words`. The `.` (dot, period) symbol is how you tell Ruby, "Hey, inside `Ex25` there's a function called `break_words` and I want to run it."
4. Line 8 we do the same thing with `Ex25.sort_words` to get a sorted sentence.
5. Lines 10-15 we use `Ex25.print_first_word` and `Ex25.print_last_word` to get the first and last word printed out.
6. Line 16 is interesting. I made a mistake and typed the `words` variable as `wrods` so Ruby gave me an error on Lines 17-18.
7. Lines 19-20 is where we print the modified words list. Notice that since we printed the first and last one, those words are now missing.
8. The remaining lines are for you to figure out and analyze in the extra credit.

Extra Credit

1. Take the remaining lines of the WYSS output and figure out what they are doing. Make sure you understand how you are running your functions in the Ex25 module.
2. The reason we put our functions in a `module` is so they have their own namespace. If someone else writes a function called `break_words`, we won't collide. However, if typing `Ex25.` is annoying, you can type `include Ex25` which is like saying, "Include everything from the Ex25 module in my current module."
3. Try breaking your file and see what it looks like in Ruby when you use it. You will have to quit IRB with CTRL-D to be able to reload it.

Exercise 26: Congratulations, Take A Test!

You are almost done with the first half of the book. The second half is where things get interesting. You will learn logic and be able to do useful things like make decisions.

Before you continue, I have a quiz for you. This quiz will be very hard because it requires you to fix someone else's code. When you are a programmer you often have to deal with another programmer's code, and also with their arrogance. They will very frequently claim that their code is perfect.

These programmers are stupid people who care little for others. A good programmer assumes, like a good scientist, that there's always some probability their code is wrong. Good programmers start from the premise that their software is broken and then work to rule out all possible ways it could be wrong before finally admitting that maybe it really is the other guy's code.

In this exercise, you will practice dealing with a bad programmer by fixing a bad programmer's code. I have poorly copied exercises 24 and 25 into a file and removed random characters and added flaws. Most of the errors are things Ruby will tell you, while some of them are math errors you should find. Others are formatting errors or spelling mistakes in the strings.

All of these errors are very common mistakes all programmers make. Even experienced ones.

Your job in this exercise is to correct this file. Use all of your skills to make this file better. Analyze it first, maybe printing it out to edit it like you would a school term paper. Fix each flaw and keep running it and fixing it until the script runs perfectly. Try not to get help, and instead if you get stuck take a break and come back to it later.

Even if this takes days to do, bust through it and make it right.

Finally, the point of this exercise isn't to type it in, but to fix an existing file. To do that, you must go to:

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

Copy-paste the code into a file named `ex26.rb`. This is the only time you are allowed to copy-paste.

Exercise 27: Memorizing Logic

Today is the day you start learning about logic. Up to this point you have done everything you possibly can reading and writing files, to the terminal, and have learned quite a lot of the math capabilities of Ruby.

From now on, you will be learning logic. You won't learn complex theories that academics love to study, but just the simple basic logic that makes real programs work and that real programmers need every day.

Learning logic has to come after you do some memorization. I want you to do this exercise for an entire week. Do not falter. Even if you are bored out of your mind, keep doing it. This exercise has a set of logic tables you must memorize to make it easier for you to do the later exercises.

I'm warning you this won't be fun at first. It will be downright boring and tedious but this is to teach you a very important skill you will need as a programmer. You will need to be able to memorize important concepts as you go in your life. Most of these concepts will be exciting once you get them. You will struggle with them, like wrestling a squid, then one day snap you will understand it. All that work memorizing the basics pays off big later.

Here's a tip on how to memorize something without going insane: Do a tiny bit at a time throughout the day and mark down what you need to work on most. Do not try to sit down for two hours straight and memorize these tables. This won't work. Your brain will really only retain whatever you studied in the first 15 or 30 minutes anyway.

Instead, what you should do is create a bunch of index cards with each column on the left on one side (True or False) and the column on the right on the back. You should then pull them out, see the "True or False" and be able to immediately say "True!" Keep practicing until you can do this.

Once you can do that, start writing out your own truth tables each night into a notebook. Do not just copy them. Try to do them from memory, and when you get stuck glance quickly at the ones I have here to refresh your memory. Doing this will train your brain to remember the whole table.

Do not spend more than one week on this, because you will be applying it as you go.

The Truth Terms

In Ruby we have the following terms (characters and phrases) for determining if something is "true" or "false". Logic on a computer is all about seeing if some combination of these characters and some variables is True at that point in the program.

- `and`
- `or`
- `not`
- != (not equal)
- == (equal)
- >= (greater-than-equal)

- <= (less-than-equal)
- true
- false

You actually have run into these characters before, but maybe not the phrases. The phrases (and, or, not) actually work the way you expect them to, just like in English.

The Truth Tables

We will now use these characters to make the truth tables you need to memorize.

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

Now use these tables to write up your own cards and spend the week memorizing them. Remember though, there is no failing in this book, just trying as hard as you can each day, and then a little bit more.

Exercise 28: Boolean Practice

The logic combinations you learned from the last exercise are called "boolean" logic expressions. Boolean logic is used everywhere in programming. They are essential fundamental parts of computation and knowing them very well is akin to knowing your scales in music.

In this exercise you will be taking the logic exercises you memorized and start trying them out in IRB. Take each of these logic problems, and write out what you think the answer will be. In each case it will be either true or false. Once you have the answers written down, you will start IRB in your terminal and type them in to confirm your answers.

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

I will also give you a trick to help you figure out the more complicated ones toward the end.

Whenever you see these boolean logic statements, you can solve them easily by this simple process:

1. Find equality test (== or !=) and replace it with its truth.
2. Find each and/or inside a parenthesis and solve those first.
3. Find each not and invert it.
4. Find any remaining and/or and solve it.
5. When you are done you should have true or false.

I will demonstrate with a variation on #20:

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

Here's me going through each of the steps and showing you the translation until I've boiled it down to a single result:

1. Solve each equality test:

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

2. Find each and/or in parenthesis `()`:
   - `(true or true)` is **True**: `true and not (true)` 3 Find each not and invert it:
   - `not (true)` is **False**: `true and false`
3. Find any remaining and/or and solve them:
   - `true and false` is **False**

With that we're done and know the result is false.

Warning: The more complicated ones may seem very hard at first. You should be able to give a good first stab at solving them, but do not get discouraged. I'm just getting you primed for more of these "logic gymnastics" so that later cool stuff is much easier. Just stick with it, and keep track of what you get wrong, but do not worry that it's not getting in your head quite yet. It'll come.

What You Should See

After you have tried to guess at these, this is what your session with IRB might look like:

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

Extra Credit

1. There are a lot of operators in Ruby similar to `!=` and `==`. Try to find out as many "equality operators" as you can. They should be like: `<` or `<=`.
2. Write out the names of each of these equality operators. For example, I call `!=` "not equal".
3. Play with IRB by typing out new boolean operators, and before you hit enter try to shout out what it is. Do not think about it, just the first thing that comes to mind. Write it down then hit enter, and keep track of how many you get right and wrong. Throw away that piece of paper from #3 away so you do not accidentally try to use it later.

Exercise 29: What If

Here is the next script of Ruby you will enter, which introduces you to the `if-statement`. Type this in, make it run exactly right, and then we'll try see if your practice has paid off.

```
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
```

What You Should See

```
$ 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.
$
```

Extra Credit

In this extra credit, try to guess what you think the `if-statement` is and what it does. Try to answer these questions in your own words before moving onto the next exercise:

1. What do you think the `if` does to the code under it?
2. Can you put other boolean expressions from Ex. 27 in the `if-statement`? Try it.
3. What happens if you change the initial variables for people, cats, and dogs?

Exercise 30: Else And If

In the last exercise you worked out some `if-statements`, and then tried to guess what they are and how they work. Before you learn more I'll explain what everything is by answering the questions you had from extra credit. You did the extra credit right?

1. What do you think the if does to the code under it? An `if` statement creates what is called a "branch" in the code. It's kind of like those choose your own adventure books where you are asked to turn to one page if you make one choice, and another if you go a different direction. The `if-statement` tells your script, "If this boolean expression is True, then run the code under it, otherwise skip it."
2. Can you put other boolean expressions from Ex. 27 in the `if` statement? Try it. Yes you can, and they can be as complex as you like, although really complex things generally are bad style.
3. What happens if you change the initial variables for people, cats, and dogs? Because you are comparing numbers, if you change the numbers, different `if-statements` will evaluate to **True** and the blocks of code under them will run. Go back and put different numbers in and see if you can figure out in your head what blocks of code will run.

Compare my answers to your answers, and make sure you really understand the concept of a "block" of code. This is important for when you do the next exercise where you write all the parts of `if-statements` that you can use.

Type this one in and make it work too.

```
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
```

What You Should See

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

Extra Credit

1. Try to guess what `elsif` and `else` are doing.
2. Change the numbers of `cars`, `people`, and `buses` and then trace through each `if-statement` to see what will be printed.
3. Try some more complex boolean expressions like `cars > people and buses < cars`. Above each line write an English description of what the line does.

Exercise 31: Making Decisions

In the first half of this book you mostly just printed out things and called functions, but everything was basically in a straight line. Your scripts ran starting at the top, and went to the bottom where they ended. If you made a function you could run that function later, but it still didn't have the kind of branching you need to really make decisions. Now that you have `if`, `else`, and `elsif` you can start to make scripts that decide things.

In the last script you wrote out a simple set of tests asking some questions. In this script you will ask the user questions and make decisions based on their answers. Write this script, and then play with it quite a lot to figure it out.

```
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
```

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

A key point here is that you are now putting the `if-statements` *inside* `if-statements` as code that can run. This is very powerful and can be used to create "nested" decisions, where one branch leads to another and another.

Make sure you understand this concept of `if-statements` inside `if-statements`. In fact, do the extra credit to really nail it.

What You Should See

Here is me playing this little adventure game. I do not do so well.

```
$ 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.
```

Extra Credit

Make new parts of the game and change what decisions people can make. Expand the game out as much as you can before it gets ridiculous.

Exercise 32: Loops And Arrays

You should now be able to do some programs that are much more interesting. If you have been keeping up, you should realize that now you can combine all the other things you have learned with `if-statements` and boolean expressions to make your programs do smart things.

However, programs also need to do repetitive things very quickly. We are going to use a `for-loop` in this exercise to build and print various arrays. When you do the exercise, you will start to figure out what they are. I won't tell you right now. You have to figure it out.

Before you can use a `for-loop`, you need a way to store the results of loops somewhere. The best way to do this is with an `array`. An array is a container of things that are organized in order. It's not complicated; you just have to learn a new syntax. First, there's how you make an array:

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

What you do is start the array with the `[` (left-bracket) which "opens" the array. Then you put each item you want in the array separated by commas, just like when you did function arguments. Lastly you end the array with a `]` (right-bracket) to indicate that it's over. Ruby then takes this array and all its contents, and assigns them to the variable.

> **Warning:** This is where things get tricky for people who can't program. Your brain has been taught that the world is flat. Remember in the last exercise where you put `if-statements` inside `if-statements`? That probably made your brain hurt because most people do not ponder how to "nest" things inside things. In programming this is all over the place. You will find functions that call other functions that have `if-statements` that have arrays with arrays inside arrays. If you see a structure like this that you can't figure out, take out pencil and paper and break it down manually bit by bit until you understand it.

We now will build some arrays using some loops and print them out:

```
1  the_count = [1, 2, 3, 4, 5]
2  fruits = ['apples', 'oranges', 'pears', 'apricots']
3  change = [1, 'pennies', 2, 'dimes', 3, 'quarters']
4
5  # this first kind of for-loop goes through an array
6  for number in the_count
7    puts "This is count #{number}"
8  end
9
10 # same as above, but using a block instead
11 fruits.each do |fruit|
12   puts "A fruit of type: #{fruit}"
13 end
14
15 # also we can go through mixed arrays too
16 for i in change
```

```
17    puts "I got #{i}"
18  end
19
20  # we can also build arrays, first start with an empty one
21  elements = []
22
23  # then use a range object to do 0 to 5 counts
24  for i in (0..5)
25    puts "Adding #{i} to the list."
26    # push is a function that arrays understand
27    elements.push(i)
28  end
29
30  # now we can puts them out too
31  for i in elements
32    puts "Element was: #{i}"
33  end
```

What You Should See

```
$ 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
$
```

Extra Credit

1. Take a look at how you used the range `(0..5)`. Look up the `Range` class to understand it.

2. Could you have avoided that `for-loop` entirely on line 24 and just assigned `(0..5)` directly to `elements`?
3. Find the Ruby documentation on arrays and read about them. What other operations can you do to arrays besides `push`?

Exercise 33: While Loops

Now to totally blow your mind with a new loop, the `while-loop`. A `while-loop` will keep executing the code block under it as long as a boolean expression is **True**.

Wait, you have been keeping up with the terminology right? That if we write a statement such as `if items > 5` or `for fruit in fruits` we are starting a code block. Then we indent the lines that follow, which are said to be within the block, until we reach an `end` statement, which closes the block. This is all about structuring your programs so that Ruby knows what you mean. If you do not get that idea then go back and do some more work with `if-statements`, functions, and the `for-loop` until you get it.

Later on we'll have some exercises that will train your brain to read these structures, similar to how we burned boolean expressions into your brain.

Back to `while-loops`. What they do is simply do a test like an `if-statement`, but instead of running the code block once, they jump back to the "top" where the `while` is, and repeat. It keeps doing this until the expression is **False**.

Here's the problem with `while-loops`: sometimes they do not stop. This is great if your intention is to just keep looping until the end of the universe. Otherwise you almost always want your loops to end eventually.

To avoid these problems, there's some rules to follow:

1. Make sure that you use `while-loops` sparingly. Usually a `for-loop` is better.
2. Review your `while` statements and make sure that the thing you are testing will become **False** at some point.
3. When in doubt, print out your test variable at the top and bottom of the `while-loop` to see what it's doing.

In this exercise, you will learn the `while-loop` by doing the above three things:

```
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
```

What You Should See

```
$ 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
```

Extra Credit

1. Convert this `while` loop to a function that you can call, and replace 6 in the test `(i < 6)` with a variable.
2. Now use this function to rewrite the script to try different numbers.
3. Add another variable to the function arguments that you can pass in that lets you change the `+ 1` on line 8 so you can change how much it increments by.
4. Rewrite the script again to use this function to see what effect that has.
5. Now, write it to use `for-loops` and ranges instead. Do you need the incrementor in the middle anymore? What happens if you do not get rid of it?

If at any time that you are doing this it goes crazy (it probably will), just hold down CTRL and hit c (CTRL-c) and the program will abort.

Exercise 34: Accessing Elements Of Arrays

Arrays are pretty useful, but unless you can get at the things in them they aren't all that great. You can already go through the elements of a list in order, but what if you want say, the 5th element? You need to know how to access the elements of an array. Here's how you would access the first element of an array:

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

You take a list of animals, and then you get the first one using `0`?! How does that work? Because of the way math works, Ruby start its lists at 0 rather than 1. It seems weird, but there's many advantages to this, even though it is mostly arbitrary.

The best way to explain why is by showing you the difference between how you use numbers and how programmers use numbers.

Imagine you are watching the four animals in our array above (`['bear', 'tiger', 'penguin', 'zebra']`) run in a race. They win in the order we have them in this array. The race was really exciting because the animals didn't eat each other and somehow managed to run a race. Your friend, however, shows up late and wants to know who won. Does your friend say, "Hey, who came in zeroth?" No, he says, "Hey, who came in first?"

This is because the order of the animals is important. You can't have the second animal without the first animal, and can't have the third without the second. It's also impossible to have a "zeroth" animal since zero means nothing. How can you have a nothing win a race? It just doesn't make sense. We call these kinds of numbers "ordinal" numbers, because they indicate an ordering of things.

Programmers, however, can't think this way because they can pick any element out of a list at any point. To a programmer, the above list is more like a deck of cards. If they want the tiger, they grab it. If they want the zebra, they can take it too. This need to pull elements out of lists at random means that they need a way to indicate elements consistently by an address, or an "index", and the best way to do that is to start the indices at 0. Trust me on this, the math is way easier for these kinds of accesses. This kind of number is a "cardinal" number and means you can pick at random, so there needs to be a 0 element.

So, how does this help you work with arrays? Simple, every time you say to yourself, "I want the 3rd animal," you translate this "ordinal" number to a "cardinal" number by subtracting 1. The "3rd" animal is at index 2 and is the penguin. You have to do this because you have spent your whole life using ordinal numbers, and now you have to think in cardinal. Just subtract 1 and you will be good.

Remember: ordinal == ordered, 1st; cardinal == cards at random, 0.

Let's practice this. Take this list of animals, and follow the exercises where I tell you to write down what animal you get for that ordinal or cardinal number. Remember if I say "first", "second", etc. then I'm using ordinal, so subtract 1. If I give you cardinal (0, 1, 2) then use it directly.

```
 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.

For each of these, write out a full sentence of the form: "The 1st animal is at 0 and is a bear." Then say it backwards, "The animal at 0 is the 1st animal and is a bear."

Use your Ruby to check your answers.

Hint: Ruby has also a few convenience methods for accessing particular elements in an array: `animals.first` and `animals.last`

Extra Credit

1. Read about ordinal and cardinal numbers online.
2. With what you know of the difference between these types of numbers, can you explain why this really is 2011? (Hint, you can't pick years at random.)
3. Write some more arrays and work out similar indexes until you can translate them.
4. Use Ruby to check your answers to this as well.

Warning: Programmers will tell you to read this guy named "Dijkstra" on this subject. I recommend you avoid his writings on this unless you enjoy being yelled at by someone who stopped programming at the same time programming started.

Exercise 35: Branches and Functions

You have learned to do `if-statements`, functions, and arrays. Now it's time to bend your mind. Type this in, and see if you can figure out what it's doing.

```
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

start()
```

What You Should See

Here's me playing the game:

```
$ 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!
$
```

Extra Credit

1. Draw a map of the game and how you flow through it.
2. Fix all of your mistakes, including spelling mistakes.
3. Write comments for the functions you do not understand. Remember **RDoc** comments?
4. Add more to the game. What can you do to both simplify and expand it.
5. The `gold_room` has a weird way of getting you to type a number. What are all the bugs in this way of doing it? Can you make it better than just checking if "1" or "0" are in the number? Look at how `to_i()` works for clues.

Exercise 36: Designing and Debugging

Now that you know `if-statements`, I'm going to give you some rules for `for-loops` and `while-loops` that will keep you out of trouble. I'm also going to give you some tips on debugging so that you can figure out problems with your program. Finally, you are going to design a similar little game as in the last exercise but with a slight twist.

Rules For If-Statements

1. Every `if-statement` must have an `else`.
2. If this `else` should never be run because it doesn't make sense, then you must use a `die` function in the `else` that prints out an error message and dies, just like we did in the last exercise. This will find many errors.
3. Never nest `if-statements` more than 2 deep and always try to do them 1 deep. This means if you put an `if` in an `if` then you should be looking to move that second `if` into another function.
4. Treat `if-statements` like paragraphs, where each `if`,''elsif'',''else'' grouping is like a set of sentences. Put blank lines before and after.
5. Your boolean tests should be simple. If they are complex, move their calculations to variables earlier in your function and use a good name for the variable.

If you follow these simple rules, you will start writing better code than most programmers. Go back to the last exercise and see if I followed all of these rules. If not, fix it.

> **Warning:** Never be a slave to the rules in real life. For training purposes you need to follow these rules to make your mind strong, but in real life sometimes these rules are just stupid. If you think a rule is stupid, try not using it.

Rules For Loops

1. Use a `while-loop` only to loop forever, and that means probably never. This only applies to Ruby, other languages are different.
2. Use a `for-loop` for all other kinds of looping, especially if there is a fixed or limited number of things to loop over.

Tips For Debugging

1. Do not use a "debugger". A debugger is like doing a full-body scan on a sick person. You do not get any specific useful information, and you find a whole lot of information that doesn't help and is just confusing.

2. The best way to debug a program is to use `puts` or `p` to print out the values of variables at points in the program to see where they go wrong.
3. Make sure parts of your programs work as you work on them. Do not write massive files of code before you try to run them. Code a little, run a little, fix a little.

Homework

Now write a similar game to the one that I created in the last exercise. It can be any kind of game you want in the same flavor. Spend a week on it making it as interesting as possible. For extra credit, use arrays, functions, and modules (remember those from Ex. 13?) as much as possible, and find as many new pieces of Ruby as you can to make the game work.

There is one catch though, write up your idea for the game first. Before you start coding you must write up a map for your game. Create the rooms, monsters, and traps that the player must go through on paper before you code.

Once you have your map, try to code it up. If you find problems with the map then adjust it and make the code match.

One final word of advice: Every programmer becomes paralyzed by irrational fear starting a new large project. They then use procrastination to avoid confronting this fear and end up not getting their program working or even started. I do this. Everyone does this. The best way to avoid this is to make a list of things you should do, and then do them one at a time.

Just start doing it, do a small version, make it bigger, keep a list of things to do, and do them.

Exercise 37: Symbol Review

It's time to review the symbols and Ruby words you know, and to try to pick up a few more for the next few lessons. What I've done here is written out all the Ruby symbols and keywords that are important to know.

In this lesson take each keyword, and first try to write out what it does from memory. Next, search online for it and see what it really does. It may be hard because some of these are going to be impossible to search for, but keep trying.

If you get one of these wrong from memory, write up an index card with the correct definition and try to "correct" your memory. If you just didn't know about it, write it down, and save it for later.

Finally, use each of these in a small Ruby program, or as many as you can get done. The key here is to find out what the symbol does, make sure you got it right, correct it if you do not, then use it to lock it in.

Keywords

- `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`

Data Types

For data types, write out what makes up each one. For example, with strings write out how you create a string. For numbers write out a few numbers.

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

String Escapes Sequences For string escape sequences, use them in strings to make sure they do what you think they do.

- `\\`
- `\'`

- \"
- \a
- \b
- \f
- \n
- \r
- \t
- \v

Operators Some of these may be unfamiliar to you, but look them up anyway. Find out what they do, and if you still can't figure it out, save it for later.

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

- `&&`
- `||`
- `..`
- `...`

Spend about a week on this, but if you finish faster that's great. The point is to try to get coverage on all these symbols and make sure they are locked in your head. What's also important is to find out what you do not know so you can fix it later.

Exercise 38: Reading Code

Now go find some Ruby code to read. You should be reading any Ruby code you can and trying to steal ideas that you find. You actually should have enough knowledge to be able to read, but maybe not understand what the code does. What I'm going to teach you in this lesson is how to apply things you have learned to understand other people's code.

First, print out the code you want to understand. Yes, print it out, because your eyes and brain are more used to reading paper than computer screens. Make sure you only print a few pages at a time.

Second, go through your printout and take notes of the following:

1. Functions and what they do.
2. Where each variable is first given a value.
3. Any variables with the same names in different parts of the program. These may be trouble later.
4. Any `if-statements` without `else` clauses. Are they right?
5. Any `while-loops` that might not end.
6. Finally, any parts of code that you can't understand for whatever reason.

Third, once you have all of this marked up, try to explain it to yourself by writing comments as you go. Explain the functions, how they are used, what variables are involved, anything you can to figure this code out.

Lastly, on all of the difficult parts, trace the values of each variable line by line, function by function. In fact, do another printout and write in the margin the value of each variable that you need to "trace".

Once you have a good idea of what the code does, go back to the computer and read it again to see if you find new things. Keep finding more code and doing this until you do not need the printouts anymore.

Extra Credit

1. Find out what a "flow chart" is and write a few.
2. If you find errors in code you are reading, try to fix them and send the author your changes.
3. Another technique for when you are not using paper is to put # comments with your notes in the code. Sometimes, these could become the actual comments to help the next person.

Exercise 39: Doing Things To Arrays

You have learned about arrays. When you learned about `while-loops` you "pushed" numbers onto the end of an array and printed them out. There was also extra credit where you were supposed to find all the other things you can do to arrays in the Ruby documentation. That was a while back, so go find in the book where you did that and review if you do not know what I'm talking about.

Found it? Remember it? Good. When you did this you had a list, and you "called" the function `push` on it. However, you may not really understand what's going on so let's see what we can do to lists, and how doing things with "on" them works.

When you type Ruby code that reads `mystuff.push('hello')` you are actually setting off a chain of events inside Ruby to cause something to happen to the `mystuff` array. Here's how it works:

1. Ruby sees you mentioned `mystuff` and looks up that variable. It might have to look backwards to see if you created with =, look and see if it is a function argument, or maybe it's a global variable. Either way it has to find the `mystuff` array first.
2. Once it finds `mystuff` it then hits the `.` (period) operator and starts to look at variables that are a part of `mystuff`. Since `mystuff` is an array, it knows that `mystuff` has a bunch of functions.
3. It then hits `push` and compares the name "push" to all the ones that `mystuff` says it responds to. If `push` is in there (it is) then it grabs that to use.
4. Next Ruby sees the `(` (parenthesis) and realizes, "Oh hey, this should be a function." At this point it calls (aka runs, executes) the function.

That might be a lot to take in, but we're going to spend a few exercises getting this concept firm in your brain. To kick things off, here's an exercise that mixes strings and lists for all kinds of fun.

```
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]
```

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

What You Should See

```
$ 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", "
Let's do some things with stuff.
Oranges
Corn
Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
Telephone#Sugar
$
```

Extra Credit

1. Go read about "Object Oriented Programming" online. Confused? Yeah I was too. Do not worry. You will learn enough to be dangerous, and you can slowly learn more later. Read up on what a "class" is in Ruby. *Do not read about how other languages use the word "class"*. That will only mess you up.
2. What's the relationship between `something.methods` and the "class" of `something`?
3. If you do not have any idea what I'm talking about do not worry. Programmers like to feel smart so they invented Object Oriented Programming, named it OOP, and then used it way too much. If you think that's hard, you should try to use "functional programming".

Exercise 40: Dictionaries, Oh Lovely Dictionaries

Now I have to hurt you with another container you can use, because once you learn this container a massive world of ultra-cool will be yours. It is the most useful container ever: the hash.

Ruby calls them "hashes", other languages call them, "dictionaries". I tend to use both names, but it doesn't matter. What does matter is what they do when compared to arrays. You see, an array lets you do this:

```
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 >
```

You can use numbers to "index" into an array, meaning you can use numbers to find out what's in arrays. You should know this by now, but what a hash does is let you use *anything*, not just numbers. Yes, a hash associates one thing to another, no matter what it is. Take a look:

```
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 >
```

You will see that instead of just numbers we're using symbols, which are just lightweight strings (think name tags) in Ruby, to say what we want from the `stuff` hash. We can also put new things into the hash with symbols. It doesn't have to be symbols though, we can also do this:

```
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 >
```

In this one I just used numbers. I could use anything. Well almost but just pretend you can use anything for now.

Of course, a hash that you can only put things in is pretty stupid, so here's how you delete things, with the `delete` function:

```
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 >
```

We'll now do an exercise that you must study *very* carefully. I want you to type this exercise in and try to understand what's going on. It is a very interesting exercise that will hopefully make a big light turn on in your head very soon.

```
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
```

What You Should See

```
$ 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) >
```

Extra Credit

1. Go find the Ruby documentation for hashes and try to do even more things to them.
2. Find out what you can't do with hashes. A big one is that they do not have order, so try playing with that.
3. Try doing a `for-loop` over them, and then try the `each` method of iterating through a hash.

Exercise 41: Gothons From Planet Percal #25

Did you figure out the secret of the method in the hash from the last exercise? Can you explain it to yourself? I'll explain it and you can compare your explanation with mine. Here are the lines of code we are talking about:

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

Remember that code can be stored in variables too. In order to store a block of code in a variable, we create something called a `proc`, which is short for procedure. In this code, first we are calling Ruby's built-in method called `method`, which returns a proc version of the `find_city` method, which we then store in the hash `cities` with the key `:find`. This is the same as all the others where we set states to some cities, but in this case it's actually the proc.

Alright, so once we know that `find_city` is in the hash at `:find`, that means we can do work with it. The 2nd line of code (used later in the previous exercise) can be broken down like this:

1. Ruby reads `cities` and finds that variable, it's a `hash`.
2. Then there's `[:find]` which will index into the cities hash and pull out whatever is at `:find`.
3. What is at `[:find]` is our proc `find_city` so Ruby then knows it's got a proc, and when it hits `.call` it calls the proc code.
4. The parameters `cities`, `state` are passed to the proc `find_city`, and it runs because it's called.
5. `find_city` then tries to look up `states` inside `cities`, and returns what it finds or a message saying it didn't find anything.
6. Ruby takes what `find_city` returned, and prints it out using the `puts` method we've been using all along.

Here's a trick. Sometimes these things read better in English if you read the code backwards. This is how I would do it for that same line (remember backwards):

1. `state` and `city` are...
2. passed as parameters to...
3. a proc at...
4. `:find` inside...
5. the hash `cities`...
6. and finally printed on the screen

Here's another way to read it, this time "inside-out".

1. Find the center item of the expression, in this case `[:find]`.
2. Go counter-clock-wise and you have a hash `cities`, so this finds the element `:find` in `cities`.

3. That gives us a proc. Keep going counter-clock-wise and you get to the parameters.
4. The parameters are passed to the proc, and that returns a result. Go counter-clock-wise again.
5. Finally, we are at the `puts` statement, and we have our end result.

After decades of programming I don't even think about these three ways to read code. I just glance at it and know what it means. I can even glance at a whole screen of code, and all the bugs and errors jump out at me. That took an incredibly long time and quite a bit more study than is sane. To get that way, I learned these three ways of reading most any programming language:

1. Front to back.
2. Back to front.
3. Counter-clock-wise.

Try them out when you have a difficult statement to figure out.

Now type in your next exercise, then go over it. This one is gonna be fun.

```
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 Gothon's blaster cranks a laser past your head."
```

```
41      puts "In the middle of your artful dodge your foot slips and you"
42      puts "bang your head on the metal wall and pass out."
43      puts "You wake up shortly after only to die as the Gothon stomps on"
44      puts "your head and eats you."
45      return :death
46
47    elsif action == "tell a joke"
48      puts "Lucky for you they made you learn Gothon insults in the academy."
49      puts "You tell the one Gothon joke you know:"
50      puts "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
51      puts "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
52      puts "While he's laughing you run up and shoot him square in the head"
53      puts "putting him down, then jump through the Weapon Armory door."
54      return :laser_weapon_armory
55
56    else
57      puts "DOES NOT COMPUTE!"
58      return :central_corridor
59    end
60  end
61
62  def laser_weapon_armory()
63    puts "You do a dive roll into the Weapon Armory, crouch and scan the room"
64    puts "for more Gothons that might be hiding.  It's dead quiet, too quiet."
65    puts "You stand up and run to the far side of the room and find the"
66    puts "neutron bomb in its container.  There's a keypad lock on the box"
67    puts "and you need the code to get the bomb out.  If you get the code"
68    puts "wrong 10 times then the lock closes forever and you can't"
69    puts "get the bomb.  The code is 3 digits."
70    code = "%s%s%s" % [rand(9)+1, rand(9)+1, rand(9)+1]
71    print "[keypad]> "
72    guess = gets.chomp()
73    guesses = 0
74
75    while guess != code and guesses < 10
76      puts "BZZZZEDDD!"
77      guesses += 1
78      print "[keypad]> "
79      guess = gets.chomp()
80    end
81
82    if guess == code
83      puts "The container clicks open and the seal breaks, letting gas out."
84      puts "You grab the neutron bomb and run as fast as you can to the"
85      puts "bridge where you must place it in the right spot."
86      return :the_bridge
87    else
88      puts "The lock buzzes one last time and then you hear a sickening"
89      puts "melting sound as the mechanism is fused together."
90      puts "You decide to sit there, and finally the Gothons blow up the"
91      puts "ship from their ship and you die."
92      return :death
93    end
94  end
95
96  def the_bridge()
97    puts "You burst onto the Bridge with the netron destruct bomb"
98    puts "under your arm and surprise 5 Gothons who are trying to"
```

```
99    puts "take control of the ship.  Each of them has an even uglier"
100   puts "clown costume than the last.  They haven't pulled their"
101   puts "weapons out yet, as they see the active bomb under your"
102   puts "arm and don't want to set it off."
103
104   prompt()
105   action = gets.chomp()
106
107   if action == "throw the bomb"
108     puts "In a panic you throw the bomb at the group of Gothons"
109     puts "and make a leap for the door.  Right as you drop it a"
110     puts "Gothon shoots you right in the back killing you."
111     puts "As you die you see another Gothon frantically try to disarm"
112     puts "the bomb. You die knowing they will probably blow up when"
113     puts "it goes off."
114     return :death
115
116   elsif action == "slowly place the bomb"
117     puts "You point your blaster at the bomb under your arm"
118     puts "and the Gothons put their hands up and start to sweat."
119     puts "You inch backward to the door, open it, and then carefully"
120     puts "place the bomb on the floor, pointing your blaster at it."
121     puts "You then jump back through the door, punch the close button"
122     puts "and blast the lock so the Gothons can't get out."
123     puts "Now that the bomb is placed you run to the escape pod to"
124     puts "get off this tin can."
125     return :escape_pod
126   else
127     puts "DOES NOT COMPUTE!"
128     return :the_bridge
129   end
130 end
131
132 def escape_pod()
133   puts "You rush through the ship desperately trying to make it to"
134   puts "the escape pod before the whole ship explodes.  It seems like"
135   puts "hardly any Gothons are on the ship, so your run is clear of"
136   puts "interference.  You get to the chamber with the escape pods, and"
137   puts "now need to pick one to take.  Some of them could be damaged"
138   puts "but you don't have time to look.  There's 5 pods, which one"
139   puts "do you take?"
140
141   good_pod = rand(5)+1
142   print "[pod #]>"
143   guess = gets.chomp()
144
145   if guess.to_i != good_pod
146     puts "You jump into pod %s and hit the eject button." % guess
147     puts "The pod escapes out into the void of space, then"
148     puts "implodes as the hull ruptures, crushing your body"
149     puts "into jam jelly."
150     return :death
151   else
152     puts "You jump into pod %s and hit the eject button." % guess
153     puts "The pod easily slides out into space heading to"
154     puts "the planet below.  As it flies to the planet, you look"
155     puts "back and see your ship implode then explode like a"
156     puts "bright star, taking out the Gothon ship at the same"
```

```
157       puts "time.  You won!"
158       Process.exit(0)
159     end
160   end
161
162   ROOMS = {
163     :death => method(:death),
164     :central_corridor => method(:central_corridor),
165     :laser_weapon_armory => method(:laser_weapon_armory),
166     :the_bridge => method(:the_bridge),
167     :escape_pod => method(:escape_pod)
168   }
169
170   def runner(map, start)
171     next_one = start
172
173     while true
174       room = map[next_one]
175       puts "\n--------"
176       next_one = room.call()
177     end
178   end
179
180   runner(ROOMS, :central_corridor)
```

What You Should See

```
$ 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.
```

Extra Credit

1. Explain how returning the next room works.
2. Add cheat codes to the game so you can get past the more difficult rooms.
3. Instead of having each function print itself, learn about "here document" strings.
4. Write the room description as here document strings, and change the runner to use them.
5. Once you have here document strings as the room description, do you need to have the function prompt even? Have the runner prompt the user, and pass that in to each function. Your functions should just be `if-statements` printing the result and returning the next room.
6. This is actually a small version of something called a "finite state machine". Read about them. They might not make sense but try anyway.

Exercise 42: Gothons Are Getting Classy

While it's fun to put functions inside of hashes, you'd think there'd be something in Ruby that does this for you. There is: the `class` keyword. Using `class` is how you create an even more awesome "hash with functions" than the one you made in the last exercise. Classes have all sorts of powerful features and uses that I could never go into in this book. Instead, you'll just use them like they're fancy `hashes` with functions.

A programming language that uses classes is called "Object Oriented Programming". This is an old style of programming where you make "things" and you "tell" those things to do work. You've been doing a lot of this. A whole lot. You just didn't know it. Remember when you were doing this:

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

You were actually using classes. The variable stuff is actually an Array class. The `stuff.join(' ')` is calling the `join` function of the `Array` and passing `' '` (just an empty space), which is also a class, a `String` class. It's all classes!

Well, and objects, but let's just skip that word for now. You'll learn what those are after you make some classes. How do you make classes? Very similar to how you made the `ROOMS` hash, but easier:

```
class TheThing
  attr_reader :number

  def initialize()
    @number = 0
  end

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

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

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

a.some_function()
b.some_function()

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

```
puts b.add_me_up(30)

puts a.number
puts b.number
```

See the `@` symbol before the `@number` variable? That makes it an instance variable. Every instance of `TheThing` that you create will have its own value for `@number`. Instance variables are hidden away inside the object. We can't get at the name simply by typing `a.number` *unless* we explicitly make that data readable to the outside world.

By including the `attr_reader :number` line. To make `@number` write-only, we could do `attr_writer :number`. And to make it read/write we could do `attr_accessor :number`. Ruby uses the good object-oriented principle of encapsulating data.

Next, see the `initialize` method? That is how you set up a Ruby class with internal variables. You can set them with the `@` symbol just like I showed you here. See also how we then use this in `add_me_up()` later which lets you add to the `@number` you created. Later you can see how we use this to add to our number and print it.

Classes are very powerful, so you should read everything you can about them and play with them. You actually know how to use them, you just have to try it. In fact, I want to play some guitar right now so I'm not going to give you an exercise to type. You're going to write an exercise using classes.

Here's how we'd do exercise 41 using classes instead of the thing we created:

```
class Game

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

  def prompt()
    print "> "
  end

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

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

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

  def central_corridor()
    puts "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
    puts "your entire crew.  You are the last surviving member and your last"
```

```
37      puts "mission is to get the neutron destruct bomb from the Weapons Armory,"
38      puts "put it in the bridge, and blow the ship up after getting into an "
39      puts "escape pod."
40      puts "\n"
41      puts "You're running down the central corridor to the Weapons Armory when"
42      puts "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume"
43      puts "flowing around his hate filled body.  He's blocking the door to the"
44      puts "Armory and about to pull a weapon to blast you."
45
46      prompt()
47      action = gets.chomp()
48
49      if action == "shoot!"
50        puts "Quick on the draw you yank out your blaster and fire it at the Gothon."
51        puts "His clown costume is flowing and moving around his body, which throws"
52        puts "off your aim.  Your laser hits his costume but misses him entirely.  This"
53        puts "completely ruins his brand new costume his mother bought him, which"
54        puts "makes him fly into an insane rage and blast you repeatedly in the face until"
55        puts "you are dead.  Then he eats you."
56        return :death
57
58      elsif action == "dodge!"
59        puts "Like a world class boxer you dodge, weave, slip and slide right"
60        puts "as the Gothon's blaster cranks a laser past your head."
61        puts "In the middle of your artful dodge your foot slips and you"
62        puts "bang your head on the metal wall and pass out."
63        puts "You wake up shortly after only to die as the Gothon stomps on"
64        puts "your head and eats you."
65        return :death
66
67      elsif action == "tell a joke"
68        puts "Lucky for you they made you learn Gothon insults in the academy."
69        puts "You tell the one Gothon joke you know:"
70        puts "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
71        puts "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
72        puts "While he's laughing you run up and shoot him square in the head"
73        puts "putting him down, then jump through the Weapon Armory door."
74        return :laser_weapon_armory
75
76      else
77        puts "DOES NOT COMPUTE!"
78        return :central_corridor
79      end
80    end
81
82    def laser_weapon_armory()
83      puts "You do a dive roll into the Weapon Armory, crouch and scan the room"
84      puts "for more Gothons that might be hiding.  It's dead quiet, too quiet."
85      puts "You stand up and run to the far side of the room and find the"
86      puts "neutron bomb in its container.  There's a keypad lock on the box"
87      puts "and you need the code to get the bomb out.  If you get the code"
88      puts "wrong 10 times then the lock closes forever and you can't"
89      puts "get the bomb.  The code is 3 digits."
90      code = "%s%s%s" % [rand(9)+1, rand(9)+1, rand(9)+1]
91      print "[keypad]> "
92      guess = gets.chomp()
93      guesses = 0
94
```

```
95      while guess != code and guesses < 10
96        puts "BZZZZEDDD!"
97        guesses += 1
98        print "[keypad]> "
99        guess = gets.chomp()
100     end
101
102     if guess == code
103       puts "The container clicks open and the seal breaks, letting gas out."
104       puts "You grab the neutron bomb and run as fast as you can to the"
105       puts "bridge where you must place it in the right spot."
106       return :the_bridge
107     else
108       puts "The lock buzzes one last time and then you hear a sickening"
109       puts "melting sound as the mechanism is fused together."
110       puts "You decide to sit there, and finally the Gothons blow up the"
111       puts "ship from their ship and you die."
112       return :death
113     end
114   end
115
116   def the_bridge()
117     puts "You burst onto the Bridge with the netron destruct bomb"
118     puts "under your arm and surprise 5 Gothons who are trying to"
119     puts "take control of the ship.  Each of them has an even uglier"
120     puts "clown costume than the last.  They haven't pulled their"
121     puts "weapons out yet, as they see the active bomb under your"
122     puts "arm and don't want to set it off."
123
124     prompt()
125     action = gets.chomp()
126
127     if action == "throw the bomb"
128       puts "In a panic you throw the bomb at the group of Gothons"
129       puts "and make a leap for the door.  Right as you drop it a"
130       puts "Gothon shoots you right in the back killing you."
131       puts "As you die you see another Gothon frantically try to disarm"
132       puts "the bomb. You die knowing they will probably blow up when"
133       puts "it goes off."
134       return :death
135
136     elsif action == "slowly place the bomb"
137       puts "You point your blaster at the bomb under your arm"
138       puts "and the Gothons put their hands up and start to sweat."
139       puts "You inch backward to the door, open it, and then carefully"
140       puts "place the bomb on the floor, pointing your blaster at it."
141       puts "You then jump back through the door, punch the close button"
142       puts "and blast the lock so the Gothons can't get out."
143       puts "Now that the bomb is placed you run to the escape pod to"
144       puts "get off this tin can."
145       return :escape_pod
146     else
147       puts "DOES NOT COMPUTE!"
148       return :the_bridge
149     end
150   end
151
152   def escape_pod()
```

```
153       puts "You rush through the ship desperately trying to make it to"
154       puts "the escape pod before the whole ship explodes.  It seems like"
155       puts "hardly any Gothons are on the ship, so your run is clear of"
156       puts "interference.  You get to the chamber with the escape pods, and"
157       puts "now need to pick one to take.  Some of them could be damaged"
158       puts "but you don't have time to look.  There's 5 pods, which one"
159       puts "do you take?"
160
161       good_pod = rand(5)+1
162       print "[pod #]>"
163       guess = gets.chomp()
164
165       if guess.to_i != good_pod
166         puts "You jump into pod %s and hit the eject button." % guess
167         puts "The pod escapes out into the void of space, then"
168         puts "implodes as the hull ruptures, crushing your body"
169         puts "into jam jelly."
170         return :death
171       else
172         puts "You jump into pod %s and hit the eject button." % guess
173         puts "The pod easily slides out into space heading to"
174         puts "the planet below.  As it flies to the planet, you look"
175         puts "back and see your ship implode then explode like a"
176         puts "bright star, taking out the Gothon ship at the same"
177         puts "time.  You won!"
178         Process.exit(0)
179       end
180     end
181   end
182
183   a_game = Game.new(:central_corridor)
184   a_game.play()
```

What You Should See

The output from this version of the game should be exactly the same as the previous version. In fact you'll notice that some of the code is nearly the same. Compare this new version of the game with the last one so you understand the changes that were made. Key things to really get are:

1. How you made a `class Game` and put functions inside it.
2. How `initialize` is a special initialization method that sets up important variables.
3. How you added functions to the class by nesting their definitions under the `class` keyword.
4. How you nested the contents of the functions under their names.
5. The concept of `@` and how it's used in `initialize`, `play`, and `death`.
6. How a Game was created at the end and then told to `play()` and how that got everything started.

Extra Credit

1. Add some rooms to make sure you know how to work with a class.

2. Create a two-class version of this, where one is the `Map` and the other is the `Engine`. Hint: `play` goes in the `Engine`.

Exercise 43: You Make A Game

You need to start learning to feed yourself. Hopefully as you have worked through this book, you have learned that all the information you need is on the internet, you just have to go search for it. The only thing you have been missing are the right words and what to look for when you search. Now you should have a sense of it, so it's about time you struggled through a big project and tried to get it working.

Here are your requirements:

1. Make a different game from the one I made.
2. Use more than one file, and use `require` to use them. Make sure you know what that is.
3. Use one class per room and give the classes names that fit their purpose. Like `GoldRoom`, `KoiPondRoom`.
4. Your runner will need to know about these rooms, so make a class that runs them and knows about them. There's plenty of ways to do this, but consider having each room return what room is next or setting a variable of what room is next.

Other than that I leave it to you. Spend a whole week on this and make it the best game you can. Use classes, functions, dicts, lists anything you can to make it nice. The purpose of this lesson is to teach you how to structure classes that need other classes inside other files.

Remember, I'm not telling you exactly how to do this because you have to do this yourself. Go figure it out. Programming is problem solving, and that means trying things, experimenting, failing, scrapping your work, and trying again. When you get stuck, ask for help and show people your code. If they are mean to you, ignore them, focus on the people who are not mean and offer to help. Keep working it and cleaning it until it's good, then show it some more.

Good luck, and see you in a week with your game.

Exercise 44: Evaluating Your Game

In this exercise you will evaluate the game you just made. Maybe you got part-way through it and you got stuck. Maybe you got it working but just barely. Either way, we're going to go through a bunch of things you should know now and make sure you covered them in your game. We're going to study how to properly format a class, common conventions in using classes, and a lot of "textbook" knowledge.

Why would I have you try to do it yourself and then show you how to do it right? From now on in the book I'm going to try to make you self-sufficient. I've been holding your hand mostly this whole time, and I can't do that for much longer. I'm now instead going to give you things to do, have you do them on your own, and then give you ways to improve what you did.

You will struggle at first and probably be very frustrated but stick with it and eventually you will build a mind for solving problems. You will start to find creative solutions to problems rather than just copy solutions out of textbooks.

Function Style

All the other rules I've taught you about how to make a function nice apply here, but add these things:

- For various reasons, programmers call functions that are part of classes `methods`. It's mostly marketing but just be warned that every time you say "function" they'll annoyingly correct you and say "method". If they get too annoying, just ask them to demonstrate the mathematical basis that determines how a "method" is different from a "function" and they'll shut up.
- When you work with classes much of your time is spent talking about making the class "do things". Instead of naming your functions after what the function does, instead name it as if it's a command you are giving to the class. Same as `pop` is saying "Hey array, `pop` this off." It isn't called `remove_from_end_of_list` because even though that's what it does, that's not a command to an `array`.
- Keep your functions small and simple. For some reason when people start learning about classes they forget this.

Class Style

- Your class should use "proper case" like `SuperGoldFactory` rather than `super_gold_factory`.
- Try not to do too much in your `initialize` functions. It makes them harder to use.
- Your other functions should use "underscore format" so write `my_awesome_hair` and not `myawesomehair` or `MyAwesomeHair`.
- Be consistent in how you organize your function arguments. If your class has to deal with users, dogs, and cats, keep that order throughout unless it really doesn't make sense. If you have one function takes (`dog`, `cat`, `user`) and the other takes (`user`, `cat`, `dog`), it'll be hard to use.

- Try not to use variables that come from the module or globals. They should be fairly self-contained.
- A foolish consistency is the hobgoblin of little minds. Consistency is good, but foolishly following some idiotic mantra because everyone else does is bad style. Think for yourself.

Code Style

- Give your code vertical space so people can read it. You will find some very bad programmers who are able to write reasonable code, but who do not add any spaces. This is bad style in any language because the human eye and brain use space and vertical alignment to scan and separate visual elements. Not having space is the same as giving your code an awesome camouflage paint job.
- If you can't read it out loud, it's probably hard to read. If you are having a problem making something easy to use, try reading it out loud. Not only does this force you to slow down and really read it, but it also helps you find difficult passages and things to change for readability.
- Try to do what other people are doing in Ruby until you find your own style.
- Once you find your own style, do not be a jerk about it. Working with other people's code is part of being a programmer, and other people have really bad taste. Trust me, you will probably have really bad taste too and not even realize it.
- If you find someone who writes code in a style you like, try writing something that mimics their style.

Good Comments

- There are programmers who will tell you that your code should be readable enough that you do not need comments. They'll then tell you in their most official sounding voice that, "Ergo you should never write comments." Those programmers are either consultants who get paid more if other people can't use their code, or incompetents who tend to never work with other people. Ignore them and write comments.
- When you write comments, describe why you are doing what you are doing. The code already says how, but why you did things the way you did is more important.
- When you write here doc dcomments for your functions, make the comments documentation for someone who will have to use your code. You do not have to go crazy, but a nice little sentence about what someone does with that function helps a lot.
- Finally, while comments are good, too many are bad, and you have to maintain them. Keep your comments relatively short and to the point, and if you change a function, review the comment to make sure it's still correct.

Evaluate Your Game

I want you now to pretend you are me. Adopt a very stern look, print out your code, and take a red pen and mark every mistake you find. Anything from this exercise and from other things you have known. Once you are done marking your code up, I want you to fix everything you came up with. Then repeat this a couple of times, looking for anything that could be better. Use all the tricks I've given you to break your code down into the smallest tiniest little analysis you can.

The purpose of this exercise is to train your attention to detail on classes. Once you are done with this bit of code, find someone else's code and do the same thing. Go through a printed copy of some part of it and point out all the mistakes and style errors you find. Then fix it and see if your fixes can be done without breaking their program.

I want you to do nothing but evaluate and fix code for the week. Your own code and other people's. It'll be pretty hard work, but when you are done your brain will be wired tight like a boxer's hands.

Exercise 45: Is-A, Has-A, Objects, and Classes

An important concept that you have to understand is the difference between a `Class` and an `Object`. The problem is, there is no real "difference" between a class and an object. They are actually the same thing at different points in time. I will demonstrate by a Zen koan:

```
What is the difference between a Fish and a Salmon?
```

Did that question sort of confuse you? Really sit down and think about it for a minute. I mean, a Fish and a Salmon are different but, wait, they are the same thing right? A Salmon is a kind of Fish, so I mean it's not different. But at the same time, becase a Salmon is a particular type of Fish and so it's actually different from all other Fish. That's what makes it a Salmon and not a Halibut. So a Salmon and a Fish are the same but different. Weird.

This question is confusing because most people do not think about real things this way, but they intuitively understand them. You do not need to think about the difference between a Fish and a Salmon because you know how they are related. You know a Salmon is a kind of Fish and that there are other kinds of Fish without having to understand that.

Let's take it one step further, let's say you have a bucket full of 3 Salmon and because you are a nice person, you have decided to name them Frank, Joe, and Mary. Now, think about this question:

```
What is the difference between Mary and a Salmon?
```

Again this is a weird question, but it's a bit easier than the Fish vs. Salmon question. You know that Mary is a Salmon, and so she's not really different. She's just a specific "instance" of a Salmon. Joe and Frank are also instances of Salmon. But, what do I mean when I say instance? I mean they were created from some other Salmon and now represent a real thing that has Salmon-like attributes.

Now for the mind bending idea: Fish is a `Class`, and Salmon is a `Class`, and Mary is an `Object`. Think about that for a second. Alright let's break it down real slow and see if you get it.

A Fish is a `Class`, meaning it's not a real thing, but rather a word we attach to instances of things with similar attributes. Got fins? Got gills? Lives in water? Alright it's probably a Fish.

Someone with a Ph.D. then comes along and says, "No my young friend, this Fish is actually Salmo salar, affectionately known as a Salmon." This professor has just clarified the Fish further and made a new `Class` called "Salmon" that has more specific attributes. Longer nose, reddish flesh, big, lives in the ocean or fresh water, tasty? Ok, probably a Salmon.

Finally, a cook comes along and tells the Ph.D., "No, you see this Salmon right here, I'll call her Mary and I'm going to make a tasty fillet out of her with a nice sauce." Now you have this instance of a Salmon (which also is an instance of a Fish) named Mary turned into something real that is filling your belly. It has become an `Object`.

There you have it: Mary is a kind of Salmon that is a kind of Fish. `Object` is a `Class` is a `Class`.

How This Looks In Code This is a weird concept, but to be very honest you only have to worry about it when you make new classes, and when you use a class. I will show you two tricks to help you figure out whether something is a `Class` or `Object`.

First, you need to learn two catch phrases "is-a" and "has-a". You use the phrase is-a when you talk about objects and classes being related to each other by a class relationship. You use has-a when you talk about objects and classes that are related only because they reference each other.

Now, go through this piece of code and replace each `##??` comment with a replacement comment that says whether the next line represents an is-a or a has-a relationship, and what that relationship is. In the beginning of the code, I've laid out a few examples, so you just have to write the remaining ones.

Remember, `is-a` is the relationship between Fish and Salmon, while `has-a` is the relationship between Salmon and Gills.

```
## 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
```

```
49
50  end
51
52  ## ??
53  class Fish
54
55  end
56
57  ## ??
58  class Salmon < Fish
59
60  end
61
62  ## ??
63  class Halibut < Fish
64
65  end
66
67  ## rover is-a Dog
68  rover = Dog.new("Rover")
69
70  ## ??
71  satan = Cat.new("Satan")
72
73  ## ??
74  mary = Person.new("Mary")
75
76  ## ??
77  mary.pet = satan
78
79  ## ??
80  frank = Employee.new("Frank", 120000)
81
82  ## ??
83  frank.pet = rover
84
85  ## ??
86  flipper = Fish.new
87
88  ## ??
89  crouse = Salmon.new
90
91  ## ??
92  harry = Halibut.new
```

Extra Credit

1. Is it possible to use a `Class` like it's an `Object`?
2. Fill out the animals, fish, and people in this exercise with functions that make them do things. See what happens when functions are in a "base class" like Animal vs. in say Dog.
3. Find other people's code and work out all the `is-a` and `has-a` relationships.
4. Make some new relationships that are lists and dicts so you can also have "has-many" relationships.
5. Do you think there's a such thing as a "is-many" relationship? Read about "multiple inheritance", then avoid it

if you can.

Exercise 46: A Project Skeleton

This will be where you start learning how to setup a good project "skeleton" directory. This skeleton directory will have all the basics you need to get a new project up and running. It will have your project layout, automated tests, modules, and install scripts.

Skeleton Contents: Linux/OSX

First, create the structure of your skeleton directory with these commands:

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

I use a directory named projects to store all the various things I'm working on. Inside that directory I have my skeleton directory that I put the basis of my projects into. The directory NAME will be renamed to whatever you are calling your project's main module when you use the skeleton.

Next we need to setup some initial files:

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

Then we can create a `NAME.gemspec` file in our project's root directory which we can use to install our project later if we want:

```
# -*- 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) }
```

```
19    s.require_paths = ["lib"]
20  end
```

Edit this file so that it has your contact information and is ready to go for when you copy it.

Finally you will want a simple skeleton file for unit tests (which will talk about more in the next lesson) named `test/test_NAME.rb`:

```
1   require 'test/unit'
2
3   class MyUnitTests < Test::Unit::TestCase
4
5     def setup
6       puts "setup!"
7     end
8
9     def teardown
10      puts "teardown!"
11    end
12
13    def test_basic
14      puts "I RAN!"
15    end
16
17  end
```

Installing Gems

Gems are packages of Ruby code that help you get things done, so you will need to know how to install them and use them. Here's the problem though. You are at a point where it's difficult for me to help you do that and keep this book sane and clean. There are so many ways to install software on so many computers that I'd have to spend 10 pages walking you through every step, and let me tell you I am a lazy guy.

Rather than tell you how to do it exactly, I'm going to tell you what you should install, and then tell you to figure it out and get it working. This will be really good for you since it will open a whole world of software you can use that other people have released to the world.

Next, install the following software packages:

- 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/

Do not just download these packages and install them by hand. Instead see how other people recommend you install these packages and use them for your particular system. The process will be different for most versions of Linux, OSX, and definitely different for Windows.

I am warning you, this will be frustrating. In the business we call this "yak shaving". Yak shaving is any activity that is mind numblingly irritatingly boring and tedious that you have to do before you can do something else that's more fun. You want to create cool Ruby projects, but you can't do that until you setup a skeleton directory, but you can't setup a skeleton directory until you install some packages, but you can't install packages until you install package installers, and you can't install package installers until you figure out how your system installs software in general, and so on.

Struggle through this anyway. Consider it your trial-by-annoyance to get into the programmer club. Every programmer has to do these annoying tedious tasks before they can do something cool.

Using The Skeleton

You are now done with most of your yak shaving. Whenever you want to start a new project, just do this:

1. Make a copy of your skeleton directory. Name it after your new project.
2. Rename (move) the NAME directory and `NAME.rb` file to be the name of your project.
3. Edit your `NAME.gemspec` file to have all the information for your project.
4. Rename `test/test_NAME.rb` to also have your project name.
5. Start coding.

Required Quiz

This exercise doesn't have extra credit but a quiz you should complete:

1. Read about how to use all of the things you installed.
2. Read about the `NAME.gemspec` file and all it has to offer.
3. Make a project and start writing some code in the `NAME.rb` script.
4. Put a script in the `bin` directory that you can run. Read about how you can make a Ruby script that's runnable for your system.
5. Make sure the `bin` script you created is referenced in your `NAME.gemspec` so that it gets installed.
6. Use your `NAME.gemspec` and `gem build gem install` to install your own library and make sure it works, then use `gem uninstall` to uninstall it.
7. Figure out how you can use Bundler to generate a skeleton directory automatically.

Exercise 47: Automated Testing

Having to type commands into your game over and over to make sure it's working is annoying. Wouldn't it be better to write little pieces of code that test your code? Then when you make a change, or add a new thing to your program, you just "run your tests" and the tests make sure things are still working. These automated tests won't catch all your bugs, but they will cut down on the time you spend repeatedly typing and running your code.

Every exercise after this one will not have a `What You Should See section`, but instead it will have a `What You Should Test section`. You will be writing automated tests for all of your code starting now, and this will hopefully make you an even better programmer.

I won't try to explain why you should write automated tests. I will only say that, you are trying to be a programmer, and programmers automate boring and tedious tasks. Testing a piece of software is definitely boring and tedious, so you might as well write a little bit of code to do it for you.

That should be all the explanation you need because your reason for writing unit tests is to make your brain stronger. You have gone through this book writing code to do things. Now you are going to take the next leap and write code that knows about other code you have written. This process of writing a test that runs some code you have written forces you to understand clearly what you have just written. It solidifies in your brain exactly what it does and why it works and gives you a new level of attention to detail.

Writing A Test Case

We're going to take a very simple piece of code and write one simple test. We're going to base this little test on a new project from your project skeleton.

First, make a `ex47` project from your project skeleton. Make sure you do it right and rename the library and get that first `ex47/test/test_ex47.rb` test file going right.

Next, create a simple file `ex47/lib/ex47.rb` where you can put the code to test. This will be a very silly little class that we want to test with this code in it:

```
1  class Room
2
3    attr_accessor :name, :description, :paths
4
5    def initialize(name, description)
6      @name = name
7      @description = description
8      @paths = {}
9    end
10
11   def go(direction)
12     @paths[direction]
13   end
14
```

```
15    def add_paths(paths)
16      @paths.update(paths)
17    end
18
19  end
```

Once you have that file, change unit test skeleton to this:

```
1   require 'test/unit'
2   require_relative '../lib/ex47'
3
4   class MyUnitTests < Test::Unit::TestCase
5
6     def test_room()
7       gold = Room.new("GoldRoom",
8                       """This room has gold in it you can grab. There's a
9                  door to the north.""")
10      assert_equal(gold.name, "GoldRoom")
11      assert_equal(gold.paths, {})
12    end
13
14    def test_room_paths()
15      center = Room.new("Center", "Test room in the center.")
16      north = Room.new("North", "Test room in the north.")
17      south = Room.new("South", "Test room in the south.")
18
19      center.add_paths({:north => north, :south => south})
20      assert_equal(center.go(:north), north)
21      assert_equal(center.go(:south), south)
22    end
23
24    def test_map()
25      start = Room.new("Start", "You can go west and down a hole.")
26      west = Room.new("Trees", "There are trees here, you can go east.")
27      down = Room.new("Dungeon", "It's dark down here, you can go up.")
28
29      start.add_paths({:west => west, :down => down})
30      west.add_paths({:east => start})
31      down.add_paths({:up => start})
32
33      assert_equal(start.go(:west), west)
34      assert_equal(start.go(:west).go(:east), start)
35      assert_equal(start.go(:down).go(:up), start)
36    end
37
38  end
```

This file requires the `Room` class you made in the `lib/ex47.rb` file so that you can do tests on it. There are then a set of tests that are functions starting with `test_`. Inside each test case there's a bit of code that makes a `Room` or a set of `Rooms`, and then makes sure the rooms work the way you expect them to work. It tests out the basic room features, then the paths, then tries out a whole map.

The important functions here are `assert_equal` which makes sure that variables you have set or paths you have built in a `Room` are actually what you think they are. If you get the wrong result, then Ruby's `Test::Unit` module will print out an error message so you can go figure it out.

Testing Guidelines

Follow these general loose set of guidelines when making your tests:

1. Test files go in `test/` and are named `test_NAME.rb`. This keeps your tests from clashing with your other code.
2. Write one test file for each module or class you make.
3. Keep your test cases (functions) short, but do not worry if they are a bit messy. Test cases are usually kind of messy.
4. Even though test cases are messy, try to keep them clean and remove any repetitive code you can. Create helper functions that get rid of duplicate code. You will thank me later when you make a change and then have to change your tests. Duplicated code will make changing your tests more difficult.
5. Finally, do not get too attached to your tests. Sometimes, the best way to redesign something is to just delete it, the tests, and start over.

What You Should See

```
$ 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.

Extra Credit

1. Go read about `Test::Unit` more, and also read about alternatives.
2. Learn about `Rspec` and see if you like it better.
3. Make your Room more advanced, and then use it to rebuild your game yet again but this time, unit test as you go.

Exercise 48: Advanced User Input

Your game probably was coming along great, but I bet how you handled what the user typed was becoming tedious. Each room needed its own very exact set of phrases that only worked if your player typed them perfectly. What you'd rather have is a device that lets users type phrases in various ways. For example, we'd like to have all of these phrases work the same:

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

It should be alright for a user to write something a lot like English for your game, and have your game figure out what it means. To do this, we're going to write a library that does just that. This module will have a few classes that work together to handle use input and convert it into something your game can work with reliably.

In a simple version of English the following elements:

- Words separated by spaces.
- Sentences composed of the words.
- Grammar that structures the sentences into meaning.

That means the best place to start is figuring out how to get words from the user and what kinds of words those are.

Our Game Lexicon

In our game we have to create a Lexicon of words:

- Direction words: north, south, east, west, down, up, left, right, back.
- Verbs: go, stop, kill, eat.
- Stop words: the, in, of, from, at, it
- Nouns: door, bear, princess, cabinet.
- Numbers: any string of 0 through 9 characters.

When we get to nouns, we have a slight problem since each room could have a different set of Nouns, but let's just pick this small set to work with for now and improve it later.

Breaking Up A Sentence

Once we have our lexicon of words we need a way to break up sentences so that we can figure out what they are. In our case, we've defined a sentence as "words separated by spaces", so we really just need to do this:

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

That's really all we'll worry about for now, but this will work really well for quite a while.

Lexicon Structs

Once we know how to break up a sentence into words, we just have to go through the list of words and figure out what "type" they are. To do that we're going to use a handy little Ruby structure called a "struct". A struct is a convenient way to bundle a number of attributes together, using accessor methods, without having to write an explicit class. It's created like this:

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

This creates a pair of (TOKEN, WORD) that lets you look at the word and do things with it.

This is just an example, but that's basically the end result. You want to take input from the user, carve it into words with split, then analyze those words to identify their type, and finally make a sentence out of them.

Scanning Input

Now you are ready to write your scanner. This scanner will take a string of input from a user and return a sentence that's composed of a list of structs with the (TOKEN, WORD) pairings. If a word isn't part of the lexicon then it should still return the WORD, but set the TOKEN to an error token. These error tokens will tell the user they messed up.

Here's where it gets fun. I'm not going to tell you how to do this. Instead I'm going to write a unit test und you are going to write the scanner so that the unit test works.

Exceptions And Numbers

There is one tiny thing I will help you with first, and that's converting numbers. In order to do this though, we're going to cheat and use exceptions. An exception is an error that you get from some function you may have run. What happens is your function "raises" an exception when it encounters an error, then you have to handle that exception. For example, if you type this into IRB:

```
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>'
```

That `ArgumentError` is an exception that the `Integer()` function threw because what you handed `Integer()` is not a number. The `Integer()` function could have returned a value to tell you it had an error, but since it only returns numbers, it'd have a hard time doing that. It can't return -1 since that's a number. Instead of trying to figure out what to return when there's an error, the `Integer()` function raises the `TypeError` exception and you deal with it.

You deal with an exception by using the `begin` and `rescue` keywords:

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

You put the code you want to "begin" inside the `begin` block, and then you put the code to run for the error inside the `rescue`. In this case, we want to call `Integer()` on something that might be a number. If that has an error, then we "rescue" it and return `nil` instead.

In your scanner that you write, you should use this function to test if something is a number. You should also do it as the last thing you check for before declaring that word an error word.

What You Should Test

Here are the files `test/test_lexicon.rb` that you should use:

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

```
31    end
32
33    def test_nouns()
34      assert_equal(@@lexicon.scan("bear"), [Pair.new(:noun, 'bear')])
35      result = @@lexicon.scan("bear princess")
36      assert_equal(result, [Pair.new(:noun, 'bear'),
37                    Pair.new(:noun, 'princess')])
38    end
39
40    def test_numbers()
41      assert_equal(@@lexicon.scan("1234"), [Pair.new(:number, 1234)])
42      result = @@lexicon.scan("3 91234")
43      assert_equal(result, [Pair.new(:number, 3),
44                    Pair.new(:number, 91234)])
45    end
46
47    def test_errors()
48      assert_equal(@@lexicon.scan("ASDFADFASDF"), [Pair.new(:error, 'ASDFADFASDF')])
49      result = @@lexicon.scan("bear IAS princess")
50      assert_equal(result, [Pair.new(:noun, 'bear'),
51                    Pair.new(:error, 'IAS'),
52                    Pair.new(:noun, 'princess')])
53    end
54
55  end
```

Remember that you will want to make a new project with your skeleton, type in this test case (do not copy-paste!) and write your scanner so that the test runs. Focus on the details and make sure everything works right.

Design Hints

Focus on getting one test working at a time. Keep this simple and just put all the words in your lexicon in lists that are in your `lexicon.rb` file. Do not modify the input list of words, but instead make your own new list with your lexicon pairs in it. Also, use the `include?` method with these lexicon arrays to check if a word is in the lexicon.

Extra Credit

1. Improve the unit test to make sure you cover more of the lexicon.
2. Add to the lexicon and then update the unit test.
3. Make your scanner handles user input in any capitalization and case. Update the test to make sure this actually works.
4. Find another way to convert the number.
5. My solution was 37 lines long. Is yours longer? Shorter?

Exercise 49: Making Sentences

What we should be able to get from our little game lexicon scanner is a list that looks like this (yours will be formatted differently):

```
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 >
```

Now let us turn this into something the game can work with, which would be some kind of Sentence class.

If you remember grade school, a sentence can be a simple structure like:

    Subject Verb Object

Obviously it gets more complex than that, and you probably did many days of annoying sentence graphs for English class. What we want is to turn the above lists of structs into a nice Sentence object that has subject, verb, and object.

Match And Peek

To do this we need four tools:

1. A way to loop through the list of structs. That's easy.
2. A way to "match" different types of structs that we expect in our Subject Verb Object setup.
3. A way to "peek" at a potential struct so we can make some decisions.

4. A way to "skip" things we do not care about, like stop words.
5. We use the peek function to say look at the next element in our struct array, and then match to take one off and work with it. Let's take a look at a first peek function:

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

Very easy. Now for the match function:

```
def match(word_list, expecting)
  begin
    word = word_list.shift

    if word.token == expecting
      word
    else
      nil
    end
  rescue
     nil
  end
end
```

Again, very easy, and finally our skip function:

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

By now you should be able to figure out what these do. Make sure you understand them.

The Sentence Grammar

With our tools we can now begin to build Sentence objects from our array of structs. What we do is a process of:

1. Identify the next word with `peek`.
2. If that word fits in our grammar, we call a function to handle that part of the grammar, say `parse_subject`.
3. If it doesn't, we `raise` an error, which you will learn about in this lesson.
4. When we're all done, we should have a Sentence object to work with in our game.

The best way to demonstrate this is to give you the code to read, but here's where this exercise is different from the previous one: You will write the test for the parser code I give you. Rather than giving you the test so you can write the code, I will give you the code, and you have to write the test.

Here's the code that I wrote for parsing simple sentences using the `ex48` Lexicon class:

```
1 class ParserError < Exception
2
3 end
```

```
4
5  class Sentence
6
7    def initialize(subject, verb, object)
8      # remember we take Pair.new(:noun, "princess") structs and convert them
9      @subject = subject.word
10     @verb = verb.word
11     @object = object.word
12   end
13
14 end
15
16 def peek(word_list)
17   begin
18     word_list.first.token
19   rescue
20     nil
21   end
22 end
23
24 def match(word_list, expecting)
25   begin
26     word = word_list.shift
27     if word.token == expecting
28       word
29     else
30       nil
31     end
32   rescue
33     nil
34   end
35 end
36
37 def skip(word_list, token)
38   while peek(word_list) == token
39     match(word_list, token)
40   end
41 end
42
43 def parse_verb(word_list)
44   skip(word_list, :stop)
45
46   if peek(word_list) == :verb
47     return match(word_list, :verb)
48   else
49     raise ParserError.new("Expected a verb next.")
50   end
51 end
52
53 def parse_object(word_list)
54   skip(word_list, :stop)
55   next_word = peek(word_list)
56
57   if next_word == :noun
58     return match(word_list, :noun)
59   end
60   if next_word == :direction
61     return match(word_list, :direction)
```

```
62    else
63      raise ParserError.new("Expected a noun or direction next.")
64    end
65  end
66
67  def parse_subject(word_list, subj)
68    verb = parse_verb(word_list)
69    obj = parse_object(word_list)
70
71    return Sentence.new(subj, verb, obj)
72  end
73
74  def parse_sentence(word_list)
75    skip(word_list, :stop)
76
77    start = peek(word_list)
78
79    if start == :noun
80      subj = match(word_list, :noun)
81      return parse_subject(word_list, subj)
82    elsif start == :verb
83      # assume the subject is the player then
84      return parse_subject(word_list, Pair.new(:noun, "player"))
85    else
86      raise ParserError.new("Must start with subject, object, or verb not: #{start}")
87    end
88  end
```

A Word On Exceptions

You briefly learned about exceptions, but not how to raise them. This code demonstrates how to do that with the `ParserError` at the top. Notice that it uses classes to give it the type of `Exception`. Also notice the use of `raise` keyword to raise the exception.

In your tests, you will want to work with these exceptions, which I'll show you how to do.

What You Should Test

For Exercise 49 is write a complete test that confirms everything in this code is working. That includes making exceptions happen by giving it bad sentences.

Check for an exception by using the function `assert_raise` from the Test::Unit documentation. Learn how to use this so you can write a test that is expected to fail, which is very important in testing. Learn about this function (and others) by reading the Test::Unit documentation.

When you are done, you should know how this bit of code works, and how to write a test for other people's code even if they do not want you to. Trust me, it's a very handy skill to have.

Extra Credit

1. Change the `parse_` methods and try to put them into a class rather than be just methods. Which design do you like better?

2. Make the parser more error resistant so that you can avoid annoying your users if they type words your lexicon doesn't understand.
3. Improve the grammar by handling more things like numbers.
4. Think about how you might use this Sentence class in your game to do more fun things with a user's input.

Exercise 50: Your First Website

These final three exercises will be very hard and you should take your time with them. In this first one you'll build a simple web version of one of your games. Before you attempt this exercise you must have completed Exercise 46 successfully and have a working **RubyGems** installed such that you can install packages and know how to make a skeleton project directory. If you don't remember how to do this, go back to Exercise 46 and do it all over again.

Installing Sinatra

Before creating your first web application, you'll first need to install the "web framework" called **Sinatra**. The term "framework" generally means "some package that makes it easier for me to do something". In the world of web applications, people create "web frameworks" to compensate for the difficult problems they've encountered when making their own sites. They share these common solutions in the form of a package you can download to bootstrap your own projects.

In our case, we'll be using the Sinatra framework, but there are many, many, many others you can choose from. For now, learn Sinatra then branch out to another one when you're ready (or just keep using Sinatra since it's good enough).

Using `gem` install 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...
```

Make A Simple "Hello World" Project

Now you're going to make an initial very simple "Hello World" web application and project directory using Sinatra. First, make your project directory:

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

You'll be taking the game from Exercise 42 and making it into a web application, so that's why you're calling it `gothonweb`. Before you do that, we need to create the most basic Sinatra application possible. Put the following code into `lib/gothonweb.rb`:

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

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

Then run the application like this:

```
$ 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
```

Finally, use your web browser and go to the URL `http://localhost:4567/` and you should see two things. First, in your browser you'll see `Hello, World!`. Second, you'll see your terminal with new output like this:

```
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
```

Those are log messages that Sinatra prints out so you can see that the server is working, and what the browser is doing behind the scenes. The log messages help you debug and figure out when you have problems. For example, it's saying that your browser tried to get `/favicon.ico` but that file didn't exist so it returned `404 Not Found` status code.

I haven't explained the way any of this web stuff works yet, because I want to get you setup and ready to roll so that I can explain it better in the next two exercises. To accomplish this, I'll have you break your Sinatra application in various ways and then restructure it so that you know how it's setup.

What's Going On?

Here's what's happening when your browser hits your application:

1. Your browser makes a network connection to your own computer, which is called `localhost` and is a standard way of saying "whatever my own computer is called on the network". It also uses `port 4567`.
2. Once it connects, it makes an HTTP request to the `lib/gothonweb.rb` application and asks for the `/` URL, which is commonly the first URL on any website.
3. Inside `lib/gothonweb.rb` you've got blocks of code that map to URLs. The only one we have is the `'/'` mapping. This means that whenever someone goes to `/` with a browser, Sinatra will find the code block to handle the request.
4. Sinatra calls the matching block, which simply returns a string for what Sinatra should send to the browser.
5. Finally, Sinatra has handled the request and sends this response to the browser which is what you are seeing.

Make sure you really understand this. Draw up a diagram of how this information flows from your browser, to Sinatra, then to the `/` block and back to your browser.

Fixing Errors

First, delete line 6 where you assign the `greeting` variable, then hit refresh in your browser. You should see an error page now that gives you lots of information on how your application just exploded. You know that the variable `greeting` is now missing, but Sinatra gives you this nice error page to track down exactly where. Do each of the following with this page:

1. Look at the `sinatra.error` variable.
2. Look at the `REQUEST_` variables and see if they match anything you're already familiar with. This is information that your web browser is sending to your gothonweb application. You normally don't even know that it's sending this stuff, so now you get to see what it does.

Create Basic Templates

You can break your Sinatra application, but did you notice that "Hello World" isn't a very good HTML page? This is a web application, and as such it needs a proper HTML response. To do that you will create a simple template that says "Hello World" in a big green font.

The first step is to create a `lib/views/index.erb` file that looks like this:

```
<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>
```

What is a `.erb` file? ERB stands for Embedded Ruby. `.erb` files are HTML with bits of Ruby code embedded within. If you know what HTML is then this should look fairly familiar. If not, research HTML and try writing a few web pages by hand so you know how it works. Since this is an `erb` template, Sinatra will fill in "holes" in the text depending on variables you pass in to the template. Every place you see `<%= greeting %>` will be a variable you'll pass to the template that alters its contents.

To make your `lib/gothonweb.rb` script do this, you need to add some code to tell Sinatra where to load the template and to render it. Take that file and change it like this:

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

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

Pay close attention to how I changed the last line of the `/` block so it calls `erb` passing in your greeting variable.

Once you have that in place, reload the web page in your browser and you should see a different message in green. You should also be able to do a View Source on the page in your browser to see that it is valid HTML.

This may have flown by you very fast, so let me explain how a template works:

1. In your `lib/gothonweb.rb` you've added a new `erb` method call.
2. The `erb` method knows how to load `.erb` files out of the `lib/views/` directory. It knows which file to grab (`index.erb` in this case) because you pass it as a parameter (`erb :index ...`).
3. Now, when the browser hits `/` and `lib/gothonweb.rb` matches and executes the `get '/' do` block, instead of just returning the string `greeting`, it calls `erb` and pass `greeting` to it as a variable.
4. Finally, you have the HTML in `lib/views/index.erb` that contains a bit of Ruby code that tests the `greeting variable`, and if it's there, prints one message using the `greeting`, or a default message.

To get deeper into this, change the `greeting` variable and the HTML to see what effect it has. Also create another template named `lib/views/foo.lib` and render that using `erb :foo` instead of `erb :index` like before. This will show you how the first parameter you pass to `erb` is just matched to a `.erb` file in `lib/views/`.

Extra Credit

1. Read the documentation at http://www.sinatrarb.com.
2. Experiment with everything you can find there, including their example code.
3. Read about HTML5 and CSS3 and make some `.html` and `.css` files for practice.
4. If you have a friend who knows **Rails** and is willing to help you, then consider doing Ex 50, 51, and 52 in Rails instead to see what that's like.

Exercise 51: Getting Input From A Browser

While it's exciting to see the browser display "Hello World", it's even more exciting to let the user submit text to your application from a form. In this exercise we'll improve our starter web application using forms and figure out how to do automated testing for a web application.

How The Web Works

Time for some boring stuff. You need to understand a bit more about how the web works before you can make a form. This description isn't complete, but it's accurate and will help you figure out what might be going wrong with your application. Also, creating forms will be easier if you know what they do.

I'll start with a simple diagram that shows you the different parts of a web request and how the information flows:

Figure 1.1: http request diagram

I've labeled the lines with letters so I can walk you through a regular request process:

1. You type in the url http://learnpythonthehardway.org/ into your browser and it sends the request out on `line (A)` to your computer's network interface.
2. Your request goes out over the internet on `line (B)` and then to the remote computer on `line (C)` where my server accepts the request.
3. Once my computer accepts it, my web application gets it on `line (D)`, and my web application code runs the `/` (index) handler.
4. The response comes out of my web server when I return it, and goes back to your browser over `line (D)` again.
5. The server running this site takes the response off `line (D)` then sends it back over the internet on `line (C)`.
6. The response from the server then comes off the internet on `line (B)`, and your computer's network interface hands it to your browser on `line (A)`.
7. Finally, your browser then displays the response.

In this description there are a few terms you should know so that you have a common vocabulary to work with when talking about your web application:

Browser

The software that you're probably using every day. Most people don't know what it really does, they just call it "the internet". Its job is to take addresses (like http://learnpythonthehardway.org) you type into the URL bar, then use that

information to make requests to the server at that address.

Address

This is normally a URL (Uniform Resource Locator) like http://learnpythonthehardway.org/ and indicates where a browser should go. The first part `http` indicates the protocol you want to use, in this case "Hyper-Text Transport Protocol". You can also try ftp://ibiblio.org/ to see how "File Transport Protocol" works. The `learnpythonthehardway.org` part is the "hostname", or a human readable address you can remember and which maps to a number called an IP address, similar to a telephone number for a computer on the Internet. Finally, URLs can have a trailing `path` like the `/book/` part of http://learnpythonthehardway.org/book/ which indicates a file or some resource *on* the server to retrieve with a request. There are many other parts, but those are the main ones.

Connection

Once a browser knows what protocol you want to use (http), what server you want to talk to (learnpythonthehardway.org), and what resource on that server to get, it must make a connection. The browser simply asks your Operating System (OS) to open a "port" to the computer, usually port 80. When it works the OS hands back to your program something that works like a file, but is actually sending and receiving bytes over the network wires between your computer and the other computer at "learnpythonthehardway.org". This is also the same thing that happens with http://localhost:8080/ but in this case you're telling the browser to connect to your own computer (localhost) and use port 4567 rather than the default of 80. You could also do http://learnpythonthehardway.org:80/ and get the same result, except you're explicitly saying to use port 80 instead of letting it be that by default.

Request

Your browser is connected using the address you gave. Now it needs to ask for the resource it wants (or you want) on the remote server. If you gave `/book/` at the end of the URL, then you want the file (resource) at `/book/`, and most servers will use the real file `/book/index.html` but pretend it doesn't exist. What the browser does to get this resource is send a request to the server. I won't get into exactly how it does this, but just understand that it has to send something to query the server for the request. The interesting thing is that these "resources" don't have to be files. For instance, when the browser in your application asks for something, the server is returning something your code generated.

Server

The server is the computer at the end of a browser's connection that knows how to answer your browser's requests for files/resources. Most web servers just send files, and that's actually the majority of traffic. But you're actually building a server in Ruby that knows how to take requests for resources, and then return strings that you craft using Ruby. When you do this crafting, *you* are pretending to be a file to the browser, but really it's just code. As you can see from Ex. 50, it also doesn't take much code to create a response.

Response

This is the HTML (css, javascript, or images) your server wants to send back to the browser as the answer to the browser's request. In the case of files, it just reads them off the disk and sends them to the browser, but it wraps the contents of the disk in a special "header" so the browser knows what it's getting. In the case of your application, you're still sending the same thing, including the header, but you generate that data on the fly with your Ruby code.

That is the fastest crash course in how a web browser accesses information on servers on the internet. It should work well enough for you to understand this exercise, but if not, read about it as much as you can until you get it. A really

good way to do that is to take the diagram, and break different parts of the web application you did in Exercise 50. If you can break your web application in predictable ways using the diagram, you'll start to understand how it works.

How Forms Work

The best way to play with forms is to write some code that accepts form data, and then see what you can do. Take your `lib/gothonweb.rb` file and make it look like this:

```
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
```

Restart Sinatra (hit CTRL-C and then run it again) to make sure it loads again, then with your browser go to `http://localhost:4567/hello` which should display, "I just wanted to say Hello, Nobody." Next, change the URL in your browser to `http://localhost:4567/hello?name=Frank` and you'll see it say "Hello, Frank." Finally, change the `name=Frank` part to be your name. Now it's saying hello to you.

Let's break down the changes I made to your script.

1. Instead of just a string for greeting I'm now using the `params` hash to get data from the browser. Sinatra takes all of the key/value pairs after the `?` part of the URL and adds them to the `params` hash for you to work with.
2. I then construct the `greeting` from the `name` value we extracted via the `params[:name]` hash lookup, which should be very familiar to you by now.
3. Everything else about the file is the same as before.

You're also not restricted to just one parameter on the URL. Change this example to give two variables like this: `http://localhost:4567/hello?name=Frank&greet=Hola`. Then change the code to get `params[:name]` and `params[:greet]` like this:

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

Creating HTML Forms

Passing the parameters on the URL works, but it's kind of ugly and not easy to use for regular people. What you really want is a "POST form", which is a special HTML file that has a `<form>` tag in it. This form will collect information from the user, then send it to your web application just like you did above.

Let's make a quick one so you can see how it works. Here's the new HTML file you need to create, in `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>
```

You should then change `lib/gothonweb.rb` to look like this:

```
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
```

Once you've got those written up, simply restart the web application again and hit it with your browser like before.

This time you'll get a form asking you for "A Greeting" and "Your Name". When you hit the Submit button on the form, it will give you the same greeting you normally get, but this time look at the URL in your browser. See how it's `http://localhost:4567/hello` even though you sent in parameters.

The part of the `hello_form.erb` file that makes this work is the line with `<form action="/hello" method="POST">`. This tells your browser to:

1. Collect data from the user using the form fields inside the form.
2. Send them to the server using a POST type of request, which is just another browser request that "hides" the form fields.
3. Send that to the `/hello` URL (as shown in the `action="/hello"` part).
4. You can then see how the two `<input>` tags match the names of the variables in your new code. Also notice that instead of just a GET method inside class index, I have another method POST.

How this new application works is:

1. The browser first hits the web application at `/hello` but it sends a GET, so our `get '/hello/'` block runs and returns the `hello_form`.
2. You fill out the form in the browser, and the browser does what the `<form>` says and sends the data as a POST.
3. The web application then runs the `post '/hello'` block rather than the `get '/hello'` block to handle this request.
4. This `post '/hello'` block then does what it normally does to send back the `hello` page like before. There's really nothing new in here, it's just moved into a new block.

As an exercise, go into the `lib/views/index.erb` file and add a link back to just `/hello` so that you can keep filling out the form and seeing the results. Make sure you can explain how this link works and how it's letting you cycle between `lib/views/index.erb` and `lib/views/hello_form.erb` and what's being run inside this latest Ruby code.

Creating A Layout Template

When you work on your game in the next Exercise, you'll need to make a bunch of little HTML pages. Writing a full web page each time will quickly become tedious. Luckily you can create a "layout" template, or a kind of shell that will wrap all your other pages with common headers and footers. Good programmers try to reduce repetition, so layouts are essential for being a good programmer.

Change `lib/views/index.erb` to be like this:

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

Change `lib/views/hello_form.erb` to be like this:

```
<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>
```

All we're doing is stripping out the "boilerplate" at the top and the bottom which is always on every page. We'll put that back into a single `lib/views/layout.erb` file that handles it for us from now on.

Once you have those changes, create a `lib/views/layout.erb` file with this in it:

```
<html>
  <head>
    <title>Gothons From Planet Percal #25</title>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Sinatra automatically looks for a layout template called `layout` by default to use as the *base* template for all other templates. You can customize which template is used as the base for any given page, too. Restart your application and then try to change the layout in interesting ways, but without changing the other templates.

Writing Automated Tests For Forms

It's easy to test a web application with your browser by just hitting refresh, but come on, we're programmers here. Why do some repetitive task when we can write some code to test our application? What you're going to do next is write a little test for your web application form based on what you learned in Exercise 47. If you don't remember Exercise 47, read it again.

I've created a simple little function for that lets you assert things about your web application's response, aptly named `assert_response`. Create the file `test/test_gothonweb.rb` with these contents:

```
require 'test/unit'

def assert_response(resp, contains=nil, matches=nil, headers=nil, status=200)

  assert_equal(resp.status, status, "Expected response #{status} not in #{resp}")

  if status == 200
    assert(resp.body, "Response data is empty.")
  end

  if contains
    assert((resp.body.include? contains), "Response does not contain #{contains}")
  end

  if matches
    reg = Regexp.new(matches)
    assert reg.match(contains), "Response does not match #{matches}"
  end

  if headers
    assert_equal(resp.headers, headers)
  end

end
```

Finally, run `test/test_gothonweb.rb` to test your web application:

```
$ 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
```

What I'm doing here is I'm actually importing the whole application from the `lib/gothonweb.rb` library, then running it manually.

The `rack/test` library we have included has a very simple API for processing requests. Its `get`, `put`, `post`, `delete`, and `head` methods simulate the respective type of request on the application.

All mock request methods have the same argument signature:

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

- `/path` is the request path and may optionally include a query string.
- `params` is a Hash of query/post parameters, a String request body, or nil.
- `rack_env` is a Hash of Rack environment values. This can be used to set request headers and other request related information, such as session data.

This works without running an actual web server so you can do tests with automated tests and also use your browser to test a running server.

To validate responses from this function, use the `assert_response` function from `test/test_gothonweb.rb` which has:

```
assert_response(resp, contains=nil, matches=nil, headers=nil, status=200)
```

Pass in the response you get from calling `get` or `post` then add things you want checked. Use the `contains` parameter to make sure that the response contains certain values. Use the `status` parameter to check for certain responses. There's actually quite a lot of information in this little function so it would be good for you to study it.

In the `test/test_gothonweb.rb` automated test I'm first making sure the `/foo` URL returns a "404 Not Found" response, since it actually doesn't exist. Then I'm checking that `/hello` works with both a GET and POST form. Following the test should be fairly simple, even if you might not totally know what's going on.

Take some time studying this latest application, especially how the automated testing works.

Extra Credit

1. Read even more about HTML, and give the simple form a better layout. It helps to draw what you want to do on paper and *then* implement it with HTML.
2. This one is hard, but try to figure out how you'd do a file upload form so that you can upload an image and save it to the disk.
3. This is even more mind-numbing, but go find the HTTP RFC (which is the document that describes how HTTP works) and read as much of it as you can. It is really boring, but comes in handy once in a while.
4. This will also be really difficult, but see if you can find someone to help you setup a web server like Apache, Nginx, or thttpd. Try to serve a couple of your .html and .css files with it just to see if you can. Don't worry if you can't, web servers kind of suck.
5. Take a break after this and just try making as many different web applications as you can. You should definitely read about sessions in Sinatra so you can understand how to keep state for a user.

Exercise 52: The Start Of Your Web Game

We're coming to the end of the book, and in this exercise I'm going to really challenge you. When you're done, you'll be a reasonably competent Ruby beginner. You'll still need to go through a few more books and write a couple more projects, but you'll have the skills to complete them. The only thing in your way will be time, motivation, and resources.

In this exercise, we won't make a complete game, but instead we'll make an "engine" that can run the game from Exercise 42 in the browser. This will involve refactoring Exercise 42, mixing in the structure from Exercise 47, adding automated tests, and finally creating a web engine that can run the games.

This exercise will be *huge*, and I predict you could spend anywhere from a week to months on it before moving on. It's best to attack it in little chunks and do a bit a night, taking your time to make everything work before moving on.

Refactoring The Exercise 42 Game

You've been altering the `gothonweb` project for two exercises and you'll do it one more time in this exercise. The skill you're learning is called "refactoring", or as I like to call it, "fixing stuff". Refactoring is a term programmers use to describe the process of taking old code, and changing it to have new features or just to clean it up. You've been doing this without even knowing it, as it's second nature to building software.

What you'll do in this part is take the ideas from Exercise 47 of a testable "map" of Rooms, and the game from Exercise 42, and combine them together to create a new game structure. It will have the same content, just "refactored" to have a better structure.

First step is to grab the code from `ex47.rb` and copy it to `gothonweb/lib/map.rb` and copy `ex47_tests.rb` file to `gothonweb/test/test_map.rb` and run the test suite again to make sure it keeps working.

Note: From now on I won't show you the output of a test run, just assume that you should be doing it and it'll look like the above unless you have an error.

Once you have the code from Exercise 47 copied over, it's time to refactor it to have the Exercise 42 map in it. I'm going to start off by laying down the basic structure, and then you'll have an assignment to make the `map.rb` file and the `map_tests.rb` file complete.

First thing to do is lay out the basic structure of the map using the `Room` class as it is now:

```
1  class Room
2
3    attr_accessor :name, :description, :paths
4
5    def initialize(name, description)
6      @name = name
7      @description = description
8      @paths = {}
```

```
  end

  def go(direction)
    @paths[direction]
  end

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

end

central_corridor = Room.new("Central Corridor",
%q{
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

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


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

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


the_bridge = Room.new("The Bridge",
%q{
The container clicks open and the seal breaks, letting gas out.
You grab the neutron bomb and run as fast as you can to the
bridge where you must place it in the right spot.

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

```
67  })
68
69
70  escape_pod = Room.new("Escape Pod",
71  %q{
72  You point your blaster at the bomb under your arm
73  and the Gothons put their hands up and start to sweat.
74  You inch backward to the door, open it, and then carefully
75  place the bomb on the floor, pointing your blaster at it.
76  You then jump back through the door, punch the close button
77  and blast the lock so the Gothons can't get out.
78  Now that the bomb is placed you run to the escape pod to
79  get off this tin can.
80
81  You rush through the ship desperately trying to make it to
82  the escape pod before the whole ship explodes.  It seems like
83  hardly any Gothons are on the ship, so your run is clear of
84  interference.  You get to the chamber with the escape pods, and
85  now need to pick one to take.  Some of them could be damaged
86  but you don't have time to look.  There's 5 pods, which one
87  do you take?
88  })
89
90
91  the_end_winner = Room.new("The End",
92  %q{
93  You jump into pod 2 and hit the eject button.
94  The pod easily slides out into space heading to
95  the planet below.  As it flies to the planet, you look
96  back and see your ship implode then explode like a
97  bright star, taking out the Gothon ship at the same
98  time.  You won!
99  })
100
101
102 the_end_loser = Room.new("The End",
103 %q{
104 You jump into a random pod and hit the eject button.
105 The pod escapes out into the void of space, then
106 implodes as the hull ruptures, crushing your body
107 into jam jelly.
108 })
109
110 escape_pod.add_paths({
111     '2' => the_end_winner,
112     '*' => the_end_loser
113 })
114
115 generic_death = Room.new("death", "You died.")
116
117 the_bridge.add_paths({
118     'throw the bomb' => generic_death,
119     'slowly place the bomb' => escape_pod
120 })
121
122 laser_weapon_armory.add_paths({
123     '0132' => the_bridge,
124     '*' => generic_death
```

```
})

central_corridor.add_paths({
    'shoot!' => generic_death,
    'dodge!'=> generic_death,
    'tell a joke' => laser_weapon_armory
})

START = central_corridor
```

You'll notice that there are a couple of problems with our Room class and this map:

1. We have to put the text that was in the `if-else` clauses that got printed before entering a room as part of each room. This means you can't shuffle the map around which would be nice. You'll be fixing that up in this exercise.
2. There are parts in the original game where we ran code that determined things like the bomb's keypad code, or the right pod. In this game we just pick some defaults and go with it, but later you'll be given extra credit to make this work again.
3. I've just made a `generic_death` ending for all of the bad decisions, which you'll have to finish for me. You'll need to go back through and add in all the original endings and make sure they work.
4. I've got a new kind of transition labeled `"*"` that will be used for a "catch-all" action in the engine.

Once you've got that basically written out, here's the new automated test `test/test_map.rb` that you should have to get yourself started:

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

class MapTests < Test::Unit::TestCase

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

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

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

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

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

```

```
33      assert_equal(start.go('west'), west)
34      assert_equal(start.go('west').go('east'), start)
35      assert_equal(start.go('down').go('up'), start)
36    end
37
38    def test_gothon_game_map()
39      assert_equal(START.go('shoot!'), generic_death)
40      assert_equal(START.go('dodge!'), generic_death)
41
42      room = START.go('tell a joke')
43      assert_equal(room, laser_weapon_armory)
44    end
45
46  end
```

Your task in this part of the exercise is to complete the map, and make the automated test completely validate the whole map. This includes fixing all the `generic_death` objects to be real endings. Make sure this works really well and that your test is as complete as possible because we'll be changing this map later and you'll use the tests to make sure it keeps working.

Sessions And Tracking Users

At a certain point in your web application you'll need to keep track of some information and associate it with the user's browser. The web (because of HTTP) is what we like to call "stateless", which means each request you make is independent of any other requests being made. If you request page A, put in some data, and click a link to page B, all the data you sent to page A just disappears.

The solution to this is to create a little data store (usually in a database, on disk, or in cookies) that uses a number unique to each browser to keep track of what that browser was doing. In Sinatra it's fairly easy, and here's an example showing how it's done using Rack middleware:

```
require 'rubygems'
require 'sinatra'

use Rack::Session::Pool

get '/count' do
  session[:count] ||= 0
  session[:count] +=1
  "Count: #{session[:count]}"
end

get '/reset' do
  session.clear
  "Count reset to 0."
end
```

Creating An Engine

You should have your game map working and a good unit test for it. I now want to make a simple little game engine that will run the rooms, collect input from the player, and keep track of where a play is in the game. We'll be using the sessions you just learned to make a simple game engine that will:

1. Start a new game for new users.

2. Present the room to the user.
3. Take input from the user.
4. Run their input through the game.
5. Display the results and keep going until they die.

To do this, you're going to take the trusty `lib/gothonweb.rb` you've been hacking on and create a fully working, session based, game engine. The catch is I'm going to make a very simple one with basic HTML files, and it'll be up to you to complete it. Here's the base engine:

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

module Gothonweb

  use Rack::Session::Pool

  get '/' do
    # this is used to "setup" the session with starting values
    p START
    session[:room] = START
    redirect("/game")
  end

  get '/game' do
    if session[:room]
      erb :show_room, :locals => {:room => session[:room]}
    else
      # why is this here? do you need it?
      erb :you_died
    end
  end

  post '/game' do
    action = "#{params[:action] || nil}"
    # there is a bug here, can you fix it?
    if session[:room]
      session[:room] = session[:room].go(params[:action])
    end
    redirect("/game")
  end

end
```

You should next delete `lib/views/hello_form.erb` and `lib/views/index.erb` and create the two templates mentioned in the above code. Here's a very simple `lib/views/show_room.erb`:

```
<h1><%= room.name %></h1>

<pre>
<%= room.description %>
</pre>

<% if room.name == "death" %>
  <p>
  <a href="/">Play Again?</a>
```

```
  </p>
<% else %>
  <p>
  <form action="/game" method="POST">
    - <input type="text" name="action"> <input type="SUBMIT">
  </form>
  </p>
<% end %>
```

That is the template to show a room as you travel through the game. Next you need one to tell someone they died in the case that they got to the end of the map on accident, which is `lib/views/you_died.erb`:

```
<h1>You Died!</h1>

<p>Looks like you bit the dust.</p>
<p><a href="/">Play Again</a></p>
```

With those in place, you should now be able to do the following:

1. Get the test `test/test_gothonweb.rb` working again so that you are testing the game. You won't be able to do much more than a few clicks in the game because of sessions, but you should be able to do some basics.
2. Run the `lib/gothonweb.rb` script and test out the game.
3. You should be able to refresh and fix the game like normal, and work with the game HTML and engine until it does all the things you want it to do.

Your Final Exam

Do you feel like this was a huge amount of information thrown at you all at once? Good, I want you to have something to tinker with while you build your skills. To complete this exercise, I'm going to give you a final set of exercises for you to complete on your own. You'll notice that what you've written so far isn't very well built, it is just a first version of the code. Your task now is to make the game more complete by doing these things:

1. Fix all the bugs I mention in the code, and any that I didn't mention. If you find new bugs, let me know.
2. Improve all of the automated tests so that you test more of the application and get to a point where you use a test rather than your browser to check the application while you work.
3. Make the HTML look better.
4. Research logins and create a signup system for the application, so people can have logins and high scores.
5. Complete the game map, making it as large and feature complete as possible.
6. Give people a "help" system that lets them ask what they can do at each room in the game.
7. Add any other features you can think of to the game.
8. Create several "maps" and let people choose a game they want to run. Your `lib/gothonweb.rb` engine should be able to run any map of rooms you give it, so you can support multiple games.
9. Finally, use what you learned in Exercises 48 and 49 to create a better input processor. You have most of the code necessary, you just need to improve the grammar and hook it up to your input form and the GameEngine.

Good luck!

Next Steps

You're not a programmer quite yet. I like to think of this book as giving you your "programming brown belt". You know enough to start another book on programming and handle it just fine. This book should have given you the mental tools and attitude you need to go through most Ruby books and actually learn something. It might even make it easy.

> **Rob says:** For fun, I recommend you check out Why's (Poignant) Guide to Ruby: http://mislav.uniqpath.com/poignant-guide Most of the actual programming content will be review by now, but Why is a brilliant mind and his book is a work of art. Check out some of his open source projects, which are still floating around. You can learn a lot by reading his code.

You could probably start hacking away at some programs right now, and if you have that itch, go ahead. Just understand anything you write will probably suck. That's alright though, I suck at every programming language I first start using. Nobody writes pure perfect gold when they're a beginner, and anyone who tells you they did is a huge liar.

Finally, remember that this is something you have to do at least a couple hours a night for a while before you can get good. If it helps, while you're struggling to learn Ruby every night, I'm hard at work learning to play guitar. I work at it about 2 or 4 hours a day and still practice scales.

Everyone is a beginner at something.

Advice From An Old Programmer

You've finished this book and have decided to continue with programming. Maybe it will be a career for you, or maybe it will be a hobby. You'll need some advice to make sure you continue on the right path, and get the most enjoyment out of your newly chosen activity.

I've been programming for a very long time. So long that it's incredibly boring to me. At the time that I wrote this book, I knew about 20 programming languages and could learn new ones in about a day to a week depending on how weird they were. Eventually though this just became boring and couldn't hold my interest anymore. This doesn't mean I think programming *is* boring, or that *you* will think it's boring, only that I find it uninteresting at this point in my journey.

What I discovered after this journey of learning is that it's not the languages that matter but what you do with them. Actually, I always knew that, but I'd get distracted by the languages and forget it periodically. Now I never forget it, and neither should you.

Which programming language you learn and use doesn't matter. Do *not* get sucked into the religion surrounding programming languages as that will only blind you to their true purpose of being your tool for doing interesting things.

Programming as an intellectual activity is the *only* art form that allows you to create interactive art. You can create projects that other people can play with, and you can talk to them indirectly. No other art form is quite this interactive. Movies flow to the audience in one direction. Paintings do not move. Code goes both ways.

Programming as a profession is only moderately interesting. It can be a good job, but you could make about the same money and be happier running a fast food joint. You're much better off using code as your secret weapon in another profession.

People who can code in the world of technology companies are a dime a dozen and get no respect. People who can code in biology, medicine, government, sociology, physics, history, and mathematics are respected and can do amazing things to advance those disciplines.

Of course, all of this advice is pointless. If you liked learning to write software with this book, you should try to use it to improve your life any way you can. Go out and explore this weird wonderful new intellectual pursuit that barely anyone in the last 50 years has been able to explore. Might as well enjoy it while you can.

Finally, I'll say that learning to create software changes you and makes you different. Not better or worse, just different. You may find that people treat you harshly because you can create software, maybe using words like "nerd". Maybe you'll find that because you can dissect their logic that they hate arguing with you. You may even find that simply knowing how a computer works makes you annoying and weird to them.

To this I have just one piece of advice: they can go to hell. The world needs more weird people who know how things work and who love to figure it all out. When they treat you like this, just remember that this is *your* journey, not theirs. Being different is not a crime, and people who tell you it is are just jealous that you've picked up a skill they never in their wildest dreams could acquire.

You can code. They cannot. That is pretty damn cool.