The Pragmatic Programmers

Learn to Program

Chris Pine

The Facets of Ruby Series

Learn to Program

Chris Pine

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragmaticprogrammer.com

Printed in the United States of America.

ISBN 0-9766940-4-2

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

First printing, December 2005

Version: 2005-12-19

Contents

Introduction

I vividly remember writing my first program. (My memory is pretty horrible; I don't vividly remember many things, just things like waking up after oral surgery, or watching the birth of our children, or that time I was trying to flirt with this girl when she tells me that my zipper is down, or setting my shoes on fire in my middle-school P.E. class, or writing my first program…you know, things like that.)

I suppose, looking back, that it was a fairly ambitious program for a newbie (20 or 30 lines of code, I think). But I was a math major, after all, and we are supposed to be good at things like "logical thinking." So I went down to the Reed College computer lab, armed only with a book on programming and my ego, sat down at one of the Unix terminals there, and started programming. Well, maybe "started" isn't the right word. Or "programming." I mostly just sat there, feeling hopelessly stupid. Then ashamed. Then angry. Then just small. Eight grueling hours later, the program was finished. It worked, but I didn't much care at that point…it was not a triumphant moment.

It has been more than decade, but I can still feel the stress and humiliation in my stomach when I think about it.

Clearly, this was *not* the way to learn programming.

But why was it so hard? I mean, there I was, this reasonably bright guy with some fairly rigorous mathematical training—you'd think I would be able to get this! And I did go on to make a living programming, and even to write a book about it, so it's not like I just "didn't have what it took" or anything like that. No, in fact, I find programming to be pretty easy these days, for the most part.

So why was it so hard to tell a computer to do something only mildly complex? Well, it wasn't the "mildly complex" part that was giving me problems; it was the "tell a computer" part.

In any communication with humans, you can leave out all sorts of steps or concepts and let them fill in the gaps. In fact, you have to do this! We'd never be able to get anything done otherwise. The typical example is making a peanut butter and jelly sandwich. Normally, if you wanted someone to make you a peanut butter and jelly sandwich, you might simply say, "Hey, could you make me a peanut butter and jelly sandwich?" But if you were talking to someone who had never done it before, you would have to tell them how:

1. Get out two slices of bread (and put the rest back).
2. Get out the peanut butter, the jelly, and a butter knife.
3. Spread the peanut butter on one slice of bread and the jelly on the other one.
4. Put the peanut butter and jelly away, and take care of the knife.
5. Put the slices together, put the sandwich on a plate, and bring it to me. Thanks!

I imagine those would be sufficient instructions for a small child. Small children are needlessly, recklessly clever, though. What would you have to say to a computer? Well, let's just look at that first step:

1. a) Locate bread.
   b) Pick up bread.
   c) Move to empty counter.
   d) Set down bread on counter.
   e) Open bag of bread.

   ...

But no, this isn't nearly good enough. For starters, how does it "locate bread"? We'll have to set up some sort of database associating items with locations. The database will also need locations for peanut butter, jelly, knife, sink, plate, counter....

Oh, and what if the bread is in a bread box? You'll need to open it first. Or in a cabinet? Or in your fridge? Perhaps behind something else? Or what if it is *already on the counter??* You didn't think of that one, did you? So now we have this:

- Initialize item-to-location database.
- If bread is in bread box:
  - Open bread box.
  - Pick up bread.
  - Remove hands from bread box.
  - Close bread box.

- If bread is in cabinet:
  - Open cabinet door.
  - Pick up bread.
  - Remove hands from cabinet.
  - Close cabinet door.

  ...

And on and on it goes. What if no clean knife is available? What if there is no empty counter space at the moment? And you'd better pray to whatever forces you find comfort in that there's no twist-tie on that bread!

Even steps such as "open bread box" need to be explained...and this is why we don't have robots making sandwiches for us yet: it's not that we can't build the robots; it's that we can't program them to make sandwiches. It's because making sandwiches is *hard* to describe (but easy to do for smart creatures like us humans), and computers are good only for things that are (relatively) *easy* to describe (but hard to do for slow creatures like us humans).

And that is why I had such a hard time writing that first program: computers are way dumber than I was prepared for.

What Is Programming?

When you teach someone how to make a sandwich, your job is made much easier because they already know what a sandwich is. It is this common, informal understanding of "sandwichness" that allows them to fill in the gaps in your explanation. Step 3 says to spread the peanut butter on one slice of bread. It doesn't say to spread it on only one side of the bread or to use the knife to do the spreading (as opposed to, say, your forehead). You assume they just know these things.

Similarly, I think it will help to talk a bit about what programming is, in order to give you a sort of informal understanding of it.

Programming is telling your computer how to do something. Large tasks must be broken up into smaller tasks, which must be broken up into still smaller tasks, down until you get to the most basic tasks that you don't have to describe, the tasks your computer already knows how to do. (These are *really* basic things such as arithmetic or displaying some text on your screen.)

My biggest problem when I was learning to program was that I was trying to learn it backward. I knew what I wanted the computer to

do and tried working backward from that, breaking it down until I got to something the computer knew how to do. Bad idea. I didn't really know what the computer *could* do, so I didn't know what to break the problem down to. (Mind you, now that I do know, this is exactly how I program these days. But it just doesn't work to start out this way.)

That's why you're going to learn it differently. You'll learn first about those basic things your computer can do (a few of them), and then find some simple tasks that can be broken down into a few of these basic things. *Your* first program will be so easy, it won't even take you a minute.

Programming Languages

In order to tell your computer how to do something, you must use a programming language. A programming language is similar to a human language in that it's made up of basic elements (such as nouns and verbs) and ways to combine those elements to create meaning (sentences, paragraphs, and novels). There are many languages to choose from (C, Java, Ruby, Perl...), and some have a larger set of those basic elements than others. Ruby has a fine set and is one of the easiest to learn (as well as being elegant and forgiving and the name of my daughter, and so forth), so we'll use that one.

Perhaps the best reason for using Ruby is that Ruby programs tend to be short. For example, here's a small program in Java:

```
public class HelloWorld {
  public static void main(String []args) {
    System.out.println("Hello World");
  }
}
```

And here's the same program in Ruby:

```
puts 'Hello World'
```

This program, as you might guess from the Ruby version, just writes `Hello World` to your screen. It's not nearly as obvious from looking at the Java version.

How about this comparison: I'll write a program to do *nothing*! Nothing at all! In Ruby, you don't need to *write* anything at all; a completely blank program will work just fine.

In Java, though, you need all this:

```
public class DoNothing {
  public static void main(String[] args) {
  }
}
```

You need all that just to do nothing, just to say, "Hey, I am a Java program, and I don't do anything!" So that's why we'll use Ruby. (My first program was *not* in Ruby, which is another reason why it was so painful.)

The Art of Programming

An important part of programming is, of course, making a program that does what it's supposed to do. In other words, it should have no bugs. You know all this. However, focusing on correctness, on bug-free programs, misses a lot of what programming is all about. Programming is not just about the end product; it's about the process that gets you there. (Anyway, an ugly process will result in buggy code. This happens every time.)

Programs aren't just built in one go, like a bridge. They are talked about, sketched out, prototyped, played with, refactored, tuned, tested, tweaked, deleted, rewritten....

A program is not built; it is grown.

Because a program is always growing and always changing, it must be written with change in mind. I know it's not really clear yet what this means in practical terms, but I'll be bringing it up throughout the book.

Probably the first, most basic rule of good programming is to avoid duplication of code at all costs. This is sometimes called the *DRY rule*: Don't Repeat Yourself.

I usually think of it in another way: a good programmer cultivates the virtue of laziness. (But not just any laziness: you must be aggressively, proactively lazy!) Save yourself work whenever possible. If making a few

changes now means you'll be able to save yourself more work later, do it! Make your program a place where you can do the absolute minimum amount of work to get the job done. Not only is programming this way much more interesting (it's very boring to do the same thing over and over and over...), but it produces less buggy code, and it produces it faster. It's a win-win-win situation.

Either way you look at it (DRY or laziness), the idea is the same: make your programs flexible. When change comes (and it *always* does), you'll have a much easier time changing with it.

Well, that about wraps it up. Looking at other technical books I own, they always seem to have a section here about "Who should read this book" or "How to read this book" or something. Well...I think *you* should read it, and front-to-back always works for me. (I mean, I did put the chapters in this order for a reason, you know.) Anyway, I never read that crap, so let's program!

Chapter 1

Getting Started

We'll be using three main tools when we program: a text editor (to write your programs), the Ruby interpreter (to run your programs), and your command line (which is how you tell your computer which programs you want to run).

While there's pretty much just one Ruby interpreter and one command line, there are many text editors to choose from—and some are much better for programming than others. A good text editor can help catch many of those "stupid mistakes" that beginner programmers make...oh, alright, that *all* programmers make. It makes your code much easier for yourself and others to read in a number of ways: by helping with indentation and formatting, by letting you set markers in your code (so you can easily return to something you are working on), by helping you match up your parentheses, and most important by *syntax coloring* (coloring different parts of your code with different colors according to their meanings in the program). You'll see syntax coloring in the examples in this book.

With so many good editors (and so many bad ones), it can be hard to know which to choose. I'll tell you which ones I use, though; that will have to be good enough for now. :) But whatever you choose as your text editor, do *not* use a word processor! Aside from being made for an entirely different purpose, they usually don't produce plain text, and your code must be in plain text for your programs to run.

Since setting up your environment differs somewhat from platform to platform, (which text editors are available, how to install Ruby, how your command line works...), we'll look at setting up each platform covered in this book, one at a time.

1.1 Windows

First, let's install Ruby. Go get the One-Click Installer from the website http://rubyinstaller.rubyforge.org/wiki/wiki.pl by clicking [Download] and then clicking the highest-numbered version of Ruby you see there (version 1.8.2-15 as of this writing). When you run it, it will ask you where you want to install Ruby and which parts of it you want installed...just accept all the defaults.

Now let's make a folder on your desktop in which you'll keep all of your programs. Right-click your desktop, select `New`, and then select `Folder`. Name it something truly memorable, such as `programs`. Now double-click the folder to open it.

To make a blank Ruby program, right-click in the folder, select `New`, and then select `Ruby Program`. You can rename the file if you want, but make sure to keep the `.rb` file extension, since that's what tells your computer this is a Ruby program (and not an email or a picture of Mr. Bean or something).

Now, when you installed Ruby, you also installed a really nice text editor called SciTE (which is what I use when I'm on Windows or Linux). In order to use it to edit your new program, right-click your program, and select `Edit`. (When you get to the next chapter, you'll even write a program here, but for now let's just wait.)

To actually run your programs, you'll need to go to your command line. In your `Start` menu, select the `Accessories` folder, and then choose `Command Prompt`. You'll see something like this:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\chris>_
```

(That cursor at the end will probably be blinking; it's your computer's way of asking, "What would you like?")

So here we are, at the command line: your direct connection to the soul of your computer. You want to be somewhat careful way down here, since it's not *too* hard to do Bad Things (things such as erase everything on your computer). But if you don't try anything too wacky, you should be fine.

Now some of you overachievers may have noticed that you can run your programs straight from SciTE by pressing F5. However, this will not work for any but the simplest of programs. You will need to use the command line, so you might as well get used to it now.

Boy, when I was a kid, all we had was the command line! None of these fancy buttons or mice. We typed! Up hill! In the driving snow! And we liked it!

So here you are, basically just staring at your computer naked. It would only be polite to say "hello" at this point, so type *echo hello* on the command line, and press Enter. Your computer should reply with a friendly `hello` as well, making your screen look something like this:

```
C:\Documents and Settings\chris>echo hello
hello

C:\Documents and Settings\chris>_
```

And your cursor is blinking again in a "what's next?" sort of way. Now that you're acquainted, ask it to make sure Ruby is installed properly and to tell you the version number. We do this with *ruby -v*:

```
C:\Documents and Settings\chris>ruby -v
ruby 1.8.2 (2004-12-25) [i386-mswin32]

C:\Documents and Settings\chris>_
```

Great! All we have left now is to find your `programs` folder through your command line. It's on your desktop, so we need to go there first. We do this with *cd desktop*:

```
C:\Documents and Settings\chris>cd desktop

C:\Documents and Settings\chris\Desktop>_
```

So now we see what the `C:\Documents and Settings\chris` was all about: that's where we were on the command line. But now we're on the desktop (or `C:\Documents and Settings\chris\Desktop` according to the computer).

So why *cd*? Well, way back in the olden days, before CDs (when people were getting down to eight-tracks and phonographs and such) and when command lines roamed the earth in their terrible splendor, people didn't call them *folders* on your computer. After all, there were no pictures of folders (since this was before people had discovered crayons and Photoshop), so people didn't think of them as folders. They called them *directories*. So they didn't move from folder to folder; they changed directories. But if you actually try typing *change_directory desktop* all day long, you barely have time to get down to your funky eight-tracks. So it was shortened to just *cd*.

If you want to go back up a directory, you use *cd ..*:

```
C:\Documents and Settings\chris\Desktop> cd ..

C:\Documents and Settings\chris>_
```

And to see all the directories you can *cd* into from where you are, use *dir /ad*:

```
C:\Documents and Settings\chris> dir /ad
 Volume in drive C is System
 Volume Serial Number is 843D-8EDC

 Directory of C:\Documents and Settings\chris

07.10.2005  14:30    <DIR>          .
07.10.2005  14:30    <DIR>          ..
02.09.2005  10:45    <DIR>          Application Data
04.10.2005  16:19    <DIR>          Cookies
07.10.2005  14:24    <DIR>          Desktop
15.08.2005  13:17    <DIR>          Favorites
10.02.2005  02:50    <DIR>          Local Settings
05.09.2005  13:17    <DIR>          My Documents
15.08.2005  14:14    <DIR>          NetHood
10.02.2005  02:50    <DIR>          PrintHood
07.10.2005  15:23    <DIR>          Recent
10.02.2005  02:50    <DIR>          SendTo
10.02.2005  02:50    <DIR>          Start Menu
25.02.2005  14:57    <DIR>          Templates
25.02.2005  12:07    <DIR>          UserData
               0 File(s)              0 bytes
              15 Dir(s)   6~720~483~328 bytes free

C:\Documents and Settings\chris>_
```

And there you go!

1.2 Mac OS X

If you're using OS X, you're in luck! You can use the best (in my opinion) text editor, Ruby is already installed for you in OS X 10.2 (Jaguar) and

up, and you get to use a real command line (not that silly wanna-be command line we have to use on Windows)!

My absolute favorite editor is TextMate (http://macromates.com/). It's cute and sweet and has great Ruby support. The only drawback is that it's not free. But if you code as much as I do, it's worth the (fairly cheap) price. And if you're using a Mac, then I assume you are accustomed to getting the best...and paying for it! :) In any case, it has a fully functional free trial, so you can give it a try if you want. If you really need a free text editor, though, try TextWrangler (http://www.barebones.com/products/textwrangler/). It gets the job done.

If you decide to go with the built-in TextEdit editor (which I do not *advise), make sure you save your programs as plain text! (Select* `Make Plain Text` *from the* `Format` *menu.) Otherwise, your programs* will not work.

Next, you should make a folder on your desktop in which to keep your programs. Right-click (oops! "Ctrl-click") on your desktop, and select `New Folder`. You want to give it a name both descriptive and alluring, such as `programs`. Nice.

Now, let's get to know your computer a little better. The best way to really have a one-on-one with your computer is on the command line. You get there through the Terminal application (found in the Finder by navigating to `Applications/Utilities`). Open it, and you'll see something like this:

```
Last login: Sat Oct  8 12:05:33 on ttyp1
Welcome to Darwin!
mezzaluna:~ chris$ _
```

(That cursor at the end might be blinking, and it might be a vertical line instead of an underscore. Whatever it looks like, it's your computer's way of asking, "What would you like?")

So it's telling me when I last logged in (though if it's your first time, it might not say that), welcoming me to Darwin (the deep, dark internals of OS X), and giving me a *command prompt* and cursor. Prompts, like West-Coast hairdos, come in a variety of shapes, sizes, colors, and levels of expressivity. This isn't the prompt I normally use (nor is this the hairdo I normally use—I think this is the first time I've worn pigtails out of the house), but it's the default prompt. It's showing the name of this computer ("mezzaluna"), what two dots look like (":"), something else I'll tell you about in just a bit ("~"), who I am ("chris"), and then just a dollar sign ("$"). This is for good luck, I guess. Maybe it's trying to give my name a little bling bling. I don't know.

Anyway, here we are, at the command line: the heart and soul of your computer. You want to be somewhat careful what you do down here, since it's not *too* hard to do Bad Things here. (It's easier to delete everything on your computer than it is to get rid of that dollar sign, for example.) But if you don't try anything too rambunctious, you should be fine.

So here you are, basically just staring at your computer naked. It would only be polite to say "hello" at this point, so type *echo hello* on the command line, and press Return. Your computer should reply with a friendly `hello` as well, making your screen look something like this:

```
mezzaluna:~ chris$ echo hello
hello
mezzaluna:~ chris$ _
```

And your cursor is blinking again in a "what's next?" sort of way. Now that you're acquainted, ask your computer whether it has Ruby installed, and if so, which version. Do this with *ruby -v*:

```
mezzaluna:~ chris$ ruby -v
ruby 1.8.1 (2003-12-25) [powerpc-darwin]
```

So, that's good; I have Ruby 1.8.1 installed. At this very moment, 1.8.3 is the latest. But 1.8.*anything* is pretty good. If you have an earlier version, you can still use it, but a few examples in this book might not do exactly the same thing for you. (Almost everything should work, though.)

OK. Now that Ruby is ready to rumble, it's time to learn how to get around your computer from the command line and what that ~ in the prompt is all about.

The ~ is just a short way of saying "your home directory," which is just a geek way of saying "your default folder," which is still kind of geeky anyway. And I'm OK with that.

So that's where you are: your home directory. If you want to change to a different directory, you use *cd*. (No one wants to type *change-directory*, not even once. I mean, I had to just then, to make a point, but in general you really don't want to type it.)

```
mezzaluna:~ chris$ cd Desktop
mezzaluna:~/Desktop chris$ _
```

So my prompt changed, telling me that I'm now on my desktop, which is itself in my home directory. (Notice that *Desktop* was capitalized. If you don't capitalize it, your computer will get angry and begin to swear at you in computerese, with such insults as "No such" and "file" and the worst one of all: "bash.") You can go back up a directory with *cd ..*, which in this case would put you back in your home directory. And at any time, if you just type *cd* by itself, that takes you to your home directory, no matter where you are. This is just like the Return spell in Dragon Warrior (the original Dragon Warrior; I don't play any of these new-fangled "fun" versions...).

But we don't want either of those. We want to go to your `programs` folder (or directory, or whatever). Assuming you're still in your `Desktop` folder (if not, get there quick!), just do this:

```
mezzaluna:~/Desktop chris$ cd programs
mezzaluna:~/Desktop/programs chris$ _
```

But you probably could have guessed that.

As they say here in Norway: "Bra!" (See why I like it here? I'm not even allowed to tell you what they say for "five" and "six.") Now you're ready to program.

1.3 Linux

If you're using Linux, you probably already have a favorite text editor, you can figure out how to install Ruby from source, and you better already know where to find your command line. `:)`

If you don't have a text editor you're fond of, though, might I recommend SciTE? It's made specifically for programming, it plays well with Ruby, and it's free. You can download it from `http://www.scintilla.org/SciTE.html`. If you use another relatively popular editor (emacs, vim, etc.), you can probably find Ruby syntax highlighting rules and such for it.

Next, you'll want to see whether you have Ruby installed already. Type *which ruby* on your command line. If you see a scary-looking message which looks something like `/usr/bin/which: no ruby in (...)`,

then head over to http://ruby-lang.org, and download the latest stable source. Otherwise, see what version of Ruby you are running with *ruby -v*. If it is older than the latest stable version on the above download page, you might want to upgrade (because you are Linuxy and like having the latest, especially if you get to compile it yourself).

So that's what we'll do (because, hey, I'm Linuxy, too). After you have downloaded the tarball (`ruby-1.8.3.tar.gz` in my case, but maybe something newer for you) open it up with this command:

```
$ tar -xvzf ruby-1.8.3.tar.gz
```

Then you want to go into the directory you just created and configure the build:

```
$ cd ruby-1.8.3
$ ./configure
```

Once that's done, run *make* (and optionally *make test* to test it):

```
$ make
$ make test
```

Assuming that all went well, now you get to install it. You can be logged in as root to do this part, or you can run *sudo*:

```
$ sudo make install
```

Run one final *ruby -v*, just to make sure the gods are still smiling on you:

```
$ ruby -v
ruby 1.8.3 (2005-09-21) [powerpc-darwin8.2.0]
```

Perfect! (And, as you can see, the previous incantations work just fine on OS X as well as Linux.) Now all that's left is to create a directory somewhere to keep your programs in, *cd* into that directory, and you're all set!

Alright! Are you ready? Take a deep breath. Let's program!

Chapter 2

Numbers

Now that you've gotten everything ready, it's time to write your first program! Open your text editor, and type the following:

```
puts 1+2
```

Save your program (yep, that's a complete program!) as `calc.rb`. Now run your program by typing *ruby calc.rb* into your command line. It should put a `3` on your screen. See, programming isn't so hard, now is it?

2.1 Introduction to `puts`

So what's going on in that program? I'm sure you can guess what the `1+2` does; our program is basically the same as this:

```
puts 3
```

`puts` simply writes onto the screen whatever comes after it.

2.2 Integer and Float

In most programming languages (and Ruby is no exception) numbers without decimal points are called *integers*, and numbers with decimal points are usually called *floating-point numbers* or, more simply, *floats*.

Here are some integers:

```
5
-205
999999999999999999999999
0
```

And here are some floats:

```
54.321
0.001
-205.3884
0.0
```

In practice, most programs don't use floats; they use only integers. (After all, no one wants to look at 7.4 emails, browse 1.8 web pages, or listen to 5.24 of their favorite songs....) Floats are used more for academic purposes (physics experiments and such) and for audio and video (including 3D) programs. Even most money programs use integers; they just keep track of the number of pennies!

2.3 Simple Arithmetic

So far, we have all the makings of a simple calculator. (Calculators always use floats, so if you want your computer to act just like a calculator, you should also use floats.) You type numbers using the digit keys (either at the top of your keyboard or on the numeric keypad). For decimal points, you use the period (or full-stop, normally close to the M key on the bottom row, or over on the numeric keypad). Don't, however, type commas into your numbers. If you enter `1,000,000` you'll just confuse Ruby.

For addition and subtraction, we use + and -, as we saw. For multiplication, we use *, and for division we use /. Most keyboards have these keys in the numeric keypad on the far-right side, but you can also use Shift 8 and / (the same key as the ? key). Let's try to expand our `calc.rb` program a little. Try coding the program at the top of the next page.

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

This is what the program returns:

```
3.0
6.0
-3.0
4.5
```

(The spaces in the program are not important; they just make the code easier to read.) Well, that wasn't too surprising. Now let's try it with integers:

```
puts 1+2
puts 2*3
puts 5-8
puts 9/2
```

This is mostly the same, right?

```
3
6
-3
4
```

Uh...except for that last one! When you do arithmetic with integers, you'll get integer answers. When your computer can't get the "right" answer, it always rounds down. (Of course, `4` *is* the right answer in integer arithmetic for `9/2`. It just might not be the answer you were expecting.)

Perhaps you're wondering what integer division is good for. Well, let's say you're going to the movies but you only have $9. When I lived in Portland a few years back, you could see a movie at the Bagdad for two bucks. (It was cheaper for two people to go to the Bagdad and get a pitcher of beer, *good* beer, than to go see a movie at your typical

theater. And the seats all had tables in front of them! For your beer! It was heavenly!) Anyway, nostalgia aside, how many movies could you see at the Bagdad for nine bucks? `9/2...4` movies. You can see that 4.5 is definitely *not* the right answer in this case; they will not let you watch half of a movie or let half of you in to see a whole movie...some things just aren't divisible.

So now experiment with some programs of your own! If you want to write more complex expressions, you can use parentheses. For example:

```
puts 5 * (12-8) + -15
puts 98 + (59872 / (13*8)) * -51
```

```
5
-29227
```

2.4 A Few Things to Try

Write a program that tells you the following:

- How many hours are in a year?
- How many minutes are in a decade?
- How many seconds old are you? (I'm not going to check your answer, so be as accurate—or not—you want.)

Here's a tougher question:

- If I am 912 million seconds old (which I am, though I was in the 800 millions when I started this book), how old am I?