Extracted from:

Learn to Program, Third Edition

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Updated for Ruby 3

Learn to Program

Third Edition

Chris Pine

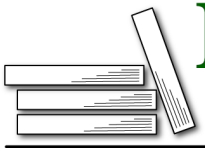
edited by Tammy Coron

The Facets of Ruby Series

Learn to Program, Third Edition

Chris Pine

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-817-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Introduction

I vividly remember writing my first program. (My memory is pretty horrible; I don't vividly remember many things, only things like waking up after oral surgery, watching the birth of my children, or writing my first program…you know, things like that.)

I suppose, looking back, that it was a fairly ambitious program for a newbie, maybe twenty or thirty lines of code. But I was a math major, after all, and we're 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 small. Eight grueling hours later, the program was finished. It worked, but I didn't much care at that point. It wasn't a triumphant moment for me.

It has been more than two decades, 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.

Why was it so hard? I mean, there I was, this reasonably bright guy with some fairly rigorous mathematical training—you'd think I'd 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 "didn't have what it took" or anything like that (which is kind of how I felt). 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, otherwise we'd never be able to get anything done. A favorite 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 made a sandwich before, you'd have to tell them how to do it:

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 look at that first step. To get out the two slices of bread, the computer would have to do the following:

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

But no, this isn't nearly good enough. For starters, how does it "locate bread"? You'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's *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. (Important!)
  - Close bread box.

- If bread is in cabinet:
  - Open cabinet door.
  - Pick up bread.
  - Remove hands from cabinet.
  - Close cabinet door.
- (And so on…)

And on and on it goes. What if no clean knife is available? What if no empty counter space is available? And you'd better hope—with everything you've got—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's why I had such a hard time writing that first program. Computers are way dumber than I was prepared for.

When you teach someone how to make a sandwich, your job is made much easier because they already know what a sandwich is. It's this common, informal understanding of "sandwichness" that allows them to fill in the gaps in your explanation. Step 3 tells you 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 simply know these things.

To make this clearer, I think it'll help to talk a bit about what programming is so that you have a sort of informal understanding of it.

What Is Programming?

Programming is telling your computer how to do something. Large tasks must be broken down into smaller tasks, which must be broken down into still smaller tasks, continuing down to the simplest 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 backwards. 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 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. But it doesn't work to start that way.)

That's why you're going to learn it differently. You'll first learn about some of the things your computer can do and then break some simple tasks down into steps your computer can handle. *Your* first program will be so easy, it won't even take you a minute.

One reason your first program will be so easy is that you'll be writing it in one of the more elegant programming languages, Ruby. It's a good choice for a first programming language.

Programming Languages

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 these elements to create meaning (sentences, paragraphs, and novels).

There are so many programming languages out there, each with their own strengths and weaknesses: Java, C++, Ruby, Lisp, Go, Erlang, and a few hundred more. As someone learning to program, how are you supposed to know which one to learn first? You let someone else pick for you. And in this case, Ruby is the language you'll be using.

Ruby is one of the easiest programming languages to learn. In my opinion, it's every bit as easy as some of the "beginner" languages out there. And since programming is hard enough as it is, let's make this as easy as we can.

Even though Ruby is as easy to learn as a language specifically geared toward beginners, it's a full, professional-strength programming language. By the end of this book, you won't need to go out and learn a "real" language. People have jobs writing Ruby code. Quite a few readers of this book's first and second editions have reached out to let me know that they're supporting their families writing Ruby code. Many websites are powered by Ruby. It's the real deal.

Ruby is also a fun programming language. In fact, the reason that Ruby's creator, Yukihiro Matsumoto, made Ruby in the first place was to create a programming language that would make programmers happy. This spirit of the joy of programming pervades the Ruby community. I know maybe a dozen programming languages at this point, and Ruby is still the first one I turn to.

But 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, writes `Hello world` to your screen. That's not nearly as obvious from looking at the Java version.

How about another comparison? Let's write a program to do *nothing*. Literally nothing at all. In Ruby, you don't need to *write* anything at all; a completely blank program is a valid Ruby program that does nothing. Which makes sense, right?

In Java, though, you need all of this:

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

You need all of that code simply to do nothing beyond saying, "Hey, I am a Java program, and I don't do anything."

So, these are the reasons why you'll use Ruby in this book. (My first program was *not* in Ruby, which is another reason why it was so painful.) Now let's get you all set up for writing Ruby programs.

Installation and Setup

You'll be using three main tools when you 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).

Although there's pretty much only 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 "silly mistakes" that beginning programmers make. Oh, all right, 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're working on), by helping you match up your parentheses, and most importantly 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 this book's examples.

With so many good editors (and so many bad ones), it can be hard to know which one to choose. I'll tell you which one I use in the setup appendix for your OS; that'll have to be good enough for now. ☺ But whatever you choose as your text editor, do *not* use something like Word, Pages, or Google Docs. Aside from being made for an entirely different purpose, they usually don't produce plain text (that is, text with no extra information about font, text size, or bold/italics/strikethrough), 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, and so on), I've placed the setup instructions in this book's appendices. Find the right one for your computer: Windows, macOS, or Linux.

Then come back here, because there's one last thing I want to talk to you about before we get started: the art of programming.

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 already know this. But focusing on correctness and bug-free programs misses a lot of what programming is all about. Programming isn't only 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 built in one go. They are talked about, sketched out, prototyped, played with, refactored, tuned, tested, tweaked, deleted, rewritten....

A program isn't built; it's grown.

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

Probably the first, most fundamental 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 from doing unnecessary 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 (as it's boring to do the same thing over and over), but it produces less buggy code, and it also produces code 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 up the introduction to this book. Looking at other technical books I own, they always seem to have a section here about "Who should read this book," "How to read this book," or something like that. Well, I think *you* should read this book, 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 stuff, so let's program!