Extracted from:

Learn to Program

The Pragmatic Programmers

Learn to Program

Chris Pine

The Facets of Ruby Series

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.

Fourth printing, January 2009

Version: 2009-3-13

Chapter 11

Reading and Writing, Saving and Loading, Yin and...Something Else

Now if you were sitting here next to me, you'd probably ask why I didn't put on any clothes before you came over. (Well, I didn't know you were coming; you weren't here like ten seconds ago.) But if you had given more warning and I was wearing clothes, you'd be more likely to say something like, "Chris" (and it's a good thing you started with my name, because as anyone near and dear to me will tell you, it's best to make sure you have my attention *before* striking up a conversation, lest I interrupt you 45 seconds later with, "You aren't talking to me, are you? Uh...eel paste in your chair...*what?*"). So "Chris," you might say, "I still can't write a program that really *does* anything."

And I'd say, "Yep."

11.1 Doing Something

So far, after your program is done running, there's really no evidence that it ever ran (aside from your memory of it). Nothing on your computer has been changed at all. The least I can do is show how to save the output of your program. For example, let's say you wanted to save the output of your nifty new "99 Bottles of Beer" program. All you have to do is add a little bit onto the command line when you run it:

ruby 99bottles.rb > lyrics.txt

That's not even really programming; that's just a command-line trick. And it's not a terribly exciting one, since you can't really use it for a program with any kind of interactivity, or to save to more than one file, or to save at a time other than the end of the program.... But, hey—it's something. What's happening is that all of the program's output (from all of the `puts`es) is being grabbed and funneled to the named file instead of being printed on your screen.

11.2 The Thing about Computers...

Before we get to real saving and loading, you and I need to talk about something. Something important. It's about computers. The thing about computers—desktops, laptops, cell phones, you name it—is that, well, they suck. This is not by nature, mind you—it isn't intrinsic—and I yearn for the day when they won't suck. But for the moment, by (poor) design, they do. This is most powerfully seen when your computer loses a bunch of your information.

A few years back I was working on a project (thankfully I was the only one on the project at the time). To make a long story short, I dropped the database. All of the information in the database: gone. The very structure of the database (which was itself days and days of work for me): gone. It was all just gone. It felt like Scotty had beamed up my stomach but forgot the rest of me. I walked around for several hours, just feeling sick, not punching things that would certainly have injured my hand, muttering, "I can't believe I dropped the database...." It was horrible. You know how I did it? It was a mouse-click about 15 pixels too high, followed by a totally reflexive (at that point) hitting of the OK button on the confirmation pop-up. And it was all gone.

User error, you say? Yeah, I suppose it was. I certainly blamed myself. But at some point you have to ask yourself why is it so fast and so easy to screw things up so catastrophically. At some point you have to start blaming the computer.

Anyway, I'm telling you this because now that your programs can actually do something, it means they can do something *bad*. Now you have to be careful. Make backups. Make them on different computers if you can. Look into source code management systems. (I use Mercurial, and I love it.) I tend to favor things like Gmail, where I can use it from any machine (just in case my main computer melts...it happened once, with

actual melting) and where it's someone else's job to make sure I don't lose anything important.

From this point on, just be careful, OK?

11.3 Saving and Loading for Grown-ups

Now that you're good and afraid, let's get to it. A file is basically just a sequence of bytes. A string is also, ultimately, just a sequence of bytes. This makes saving strings to files pretty easy, at least conceptually. (And Ruby makes it pretty easy in practice.)

So here's a quick example where we save a simple string to a file and then read it back out again. (I'll just show you the program first, and then I'll talk some more about it.)

```
#  The filename doesn't have to end
#  with ".txt", but since it is valid
#  text, why not?
filename    = 'ListerQuote.txt'
test_string = 'I promise that I swear absolutely that ' +
              'I will never mention gazpacho soup again.'

#  The 'w' here is for write-access to the file,
#  since we are trying to write to it.
File.open filename, 'w' do |f|
  f.write test_string
end

read_string = File.read filename

puts(read_string == test_string)
```

```
true
```

`File.open` is how you open a file, of course. It creates the file object, calls it `f` (because that's what we said to call it), runs all the code until it gets to the **end**, and then closes the file. When you open a file, you always have to close it again. In most programming languages you have to remember to do this, but Ruby takes care of it for you at the **end**.

Reading files is even easier than writing them; with `File.read` Ruby takes care of everything behind the scenes. (I'm not sure why they made writing more complicated, but we'll fix that in just a bit.)

Well, that's all well and good if all you want to save and load are single strings. But what if you wanted to save an array of strings? Or an array of integers and floats? And what about all of the other classes of objects that we don't even cover until the next chapter? *What about the bunnies??*

Alright, one thing at a time. Now we can definitely save any kind of object, just as long as we have some well-defined way of converting from a general object to a string, and back again. So maybe an array would be represented as text separated by commas. But what if you wanted to save a string with commas? Well, maybe you could escape the commas somehow....

Figuring this all out would take us a ridiculous amount of time. I mean, it's pretty cool that you can do it at all, but you didn't pay good money for "pretty cool." No sir, this is a De-Luxe-Supremium book you have here. And for that, my friend, we need some serious saving. We need some full-frontal loading. Yes, when you're looking for De-Luxe-Supremium, you want YAML.

11.4 YAML

What is YAML? To know what YAML is, you have to see it for yourself. Or you could see a movie about it. Or I could just tell you.

The thing with geeky acronyms is that they are often recursive (which, hey, that's pretty cute, you've got to admit) and rarely informative (as likely to tell you what they aren't as what they are). Take LINUX, for example: Linux Is Not Uncle Xenophobe. There you go. Anyway, I think YAML stands for Yaml Ate My Landlord.

YAML is a format for representing objects as strings. You can use other formats, but YAML is nice because it's human-readable (and human-editable) as well as computer-readable. My wife actually writes YAML all the time, right there in her text editor. Then another program reads it in later. Pretty cool.

YAML is not actually part of the Ruby core (it is its own thing, and many other languages can use YAML), but it is part of the standard distribution. What does that mean? Well, when you install Ruby, you install YAML, too. But if you want to actually use YAML, you'll need to import it into your program. This is really easy, though, with the `require` method:

```
require 'yaml'  #  Told you it was easy.

test_array = ['Give Quiche A Chance',
              'Mutants Out!',
              'Chameleonic Life-Forms, No Thanks']

#  Here's half of the magic:
test_string = test_array.to_yaml
#  You see?  Kind of like "to_s", and it is in fact a string,
#  but it's a YAML description of "test_array".

filename = 'RimmerTShirts.txt'

File.open filename, 'w' do |f|
  f.write test_string
end

read_string = File.read filename

#  And the other half of the magic:
read_array = YAML::load read_string

puts(read_string == test_string)
puts(read_array  == test_array )
```

```
true
true
```

Simple. Just two extra lines of code (well, three if you count the `require` line at the top). So, I'm sure the question burning in all of our hearts is, "What does the YAML string look like?!" Run it yourself, and you'll see this in `RimmerTShirts.txt`:

```
---
- Give Quiche A Chance
- Mutants Out!
- "Chameleonic Life-Forms, No Thanks"
```

Pretty clear. I'm not sure why that last line is in double quotes—perhaps because of the dash? I wouldn't be surprised if leaving out the quotes worked, too; YAML is very forgiving...I think it added the quotes only for our benefit. (A quick check says yes, indeed, we can leave those quotes out.)

Wait a second, wait a second...I said that we were saving and loading only one string. But that file has four lines in it. What, one may be well-justified in asking, gives?

Well, it is one string. It's a four-line string. How does a string get to have four lines? It has three *newline characters*. You can add newline characters to strings by just hitting Enter in your code and continuing your string on the next line, though that doesn't play well with proper indentation and ends up looking ugly. There are somewhat less-ugly ways of dealing with it, but they require a different way of defining strings. When we want to make a string, we enclose some text in single quotes and *viola!* A string. But there are other ways—like, five other ways—to define strings. I don't even know what they all are. To learn them all, there are great references out there (I'll point them out in Chapter 15, *Beyond This Fine Book*, on page 151), but in the meantime, let's just learn one more that is commonly used....

11.5 Diversion: Double-Quoted Strings

So far, we've only used single-quoted strings. They are the easiest to use, in the same sense that a shovel is easier to use than a backhoe: when the job gets big enough, it stops being easier.

Consider multi-line strings:

```
buffy_quote_1 = '\'Kiss rocks\'?
Why would anyone want to kiss...
Oh, wait. I get it.'

buffy_quote_2 = "'Kiss rocks'?\n" +
                "Why would anyone want to kiss...\n" +
                "Oh, wait. I get it."

puts buffy_quote_1
puts
puts(buffy_quote_1 == buffy_quote_2)
```