Extracted from:

Learn to Program, Third Edition

The Pragmatic Bookshelf

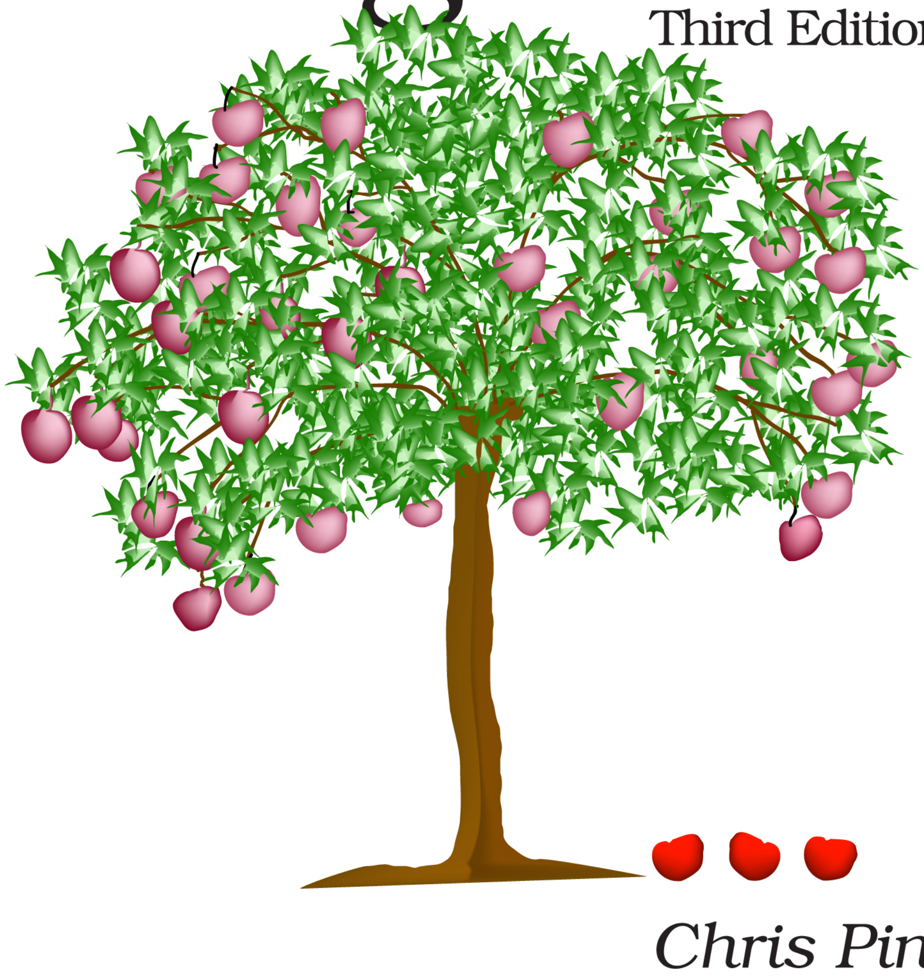
Raleigh, North Carolina

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

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

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

CHAPTER 4

Conversions and Input

So far, you've seen objects like integers, floats, and strings, and you've set up variables to point to these objects. Now it's time for you to get them all to play nicely together.

Let's suppose you have a program that you want to print 25. Using the following code won't work because you can't add numbers and strings together:

```
var1 =  2
var2 = "5"
puts var1 + var2
```

Part of the problem is that Ruby doesn't know if you were trying to print 7 (2 + 5) or 25 ("2" + "5"), so it's your job to specify which. But before you can do that, there's something else you need to do first.

To add the values together in some meaningful way, you first need to convert the objects to be the same type. In other words, you need to get the string version of the integer—in this case var1—or the integer version of the string—in this case var2.

Let's look at how you do that.

Numbers to Strings and Back Again

To get the string version of an object, you simply write .to_s after it like this:

```
var1 =  2
var2 = "5"
puts var1.to_s + var2
```

```
❮ 25
```

Similarly, appending .to_i gives the integer version of an object, and .to_f gives the float version. Let's look at what these methods do (and *don't* do) a little more closely:

```
var1 =  2
var2 = "5"
puts var1.to_s + var2
puts var1 + var2.to_i
```

```
25
7
```

Notice that, even after you got the string version of var1 by calling to_s, var1 was always pointing at 2 and never at "2". Unless you explicitly reassign var1 (which requires an = sign), it'll continue to point at 2 for the life of the program.

Let's try some more interesting (and a few just plain weird) conversions:

```
Line 1  puts "15".to_f
     2  puts "99.999".to_f
     3  puts "99.999".to_i
     4  puts ""
     5  puts "5 is my favorite number!".to_i
     6  puts "Who asked you about 5 or whatever?".to_i
     7  puts "Your mama did.".to_f
     8  puts ""
     9  puts "stringy".to_s
    10  puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

stringy
3
```

So, this program probably gave you some surprises. The conversion on line 1 is pretty standard, giving 15.0. After that, you converted the string "99.999" to a float and to an integer. The float did what was expected; the integer was, as always, rounded down.

Next, you had examples of some *unusual* strings being converted into numbers. On line 5, to_i ignores the first thing it doesn't understand (and the rest of the string from that point on). So the string on line 5 was converted to 5; but on lines 6 and 7, since those strings started with letters, they were ignored completely. When this happens, Ruby picks zero.

Finally, you saw that the last two conversions did nothing at all, as you'd expect.

Let Me Tell You a Secret

There's something strange about the puts method. Have a look at this:

```
puts  20
puts  20.to_s
puts "20"
```

```
20
20
20
```

Why do these three all print the same thing? Well, the last two should, since 20.to_s is "20". But what about the first example, the integer 20? For that matter, what does it even mean to write the *integer* 20? When you write a *2* and then a *0* on a piece of paper, you're writing a string, not an integer. The integer 20 is the number of fingers and toes I have; it isn't a *2* followed by a *0*.

Well, here's the big secret behind the puts method: before puts tries to write out an object, it uses to_s to get the string version of that object. In fact, the *s* in puts stands for *string*; puts actually means *put string*.

And here's a *bonus secret*: string interpolation does the same thing. Check it out:

```
puts "my favorite number really is #{2+3}"
```

```
my favorite number really is 5
```

This actually ends up being convenient, as it's one less conversion that you have to think about.

Now let's check out how you can get strings directly from the user, which will allow you to write all sorts of fun programs.

Getting Strings from the User

To get strings from the user, you use the gets method. If puts means *put string*, you can probably guess what gets means. And just as puts always spits out strings, gets retrieves only strings. So, where do these strings come from?

From you—well, from your keyboard, anyway! Of course, since your keyboard makes only strings, that works out beautifully. What actually happens is that gets simply sits there, reading what you type until you press Enter. Try it out:

```
puts gets
```

```
⇒ Is there an echo in here?
❮ Is there an echo in here?
```

As expected, whatever you type will get repeated back to you. Run it a few times and try typing different things.

Did It Work?

Maybe you didn't need any help installing Ruby, so you skipped Chapter 1. Hey, no problem.

Maybe you've done a little programming before, so you skipped Chapter 2. That's totally fine.

The only thing is that you missed some stuff there that you need to know now. If you haven't been running your programs from the command line, then you'll almost certainly have problems with gets, and you're going to be using it a lot from now on. So, if you saved your program as example.rb, you should run your program by typing ruby example.rb into your command line. If you're having trouble getting around on your command line, refer to the OS-specific appendix at the back of this book.

Now that you know how to use gets, you can make interactive programs. These will be *way more interesting* than the programs you've written so far. Still, one issue remains with gets that'll bite you, so let's look at what it is and how to avoid it.

Cleaning Up User Input

Let's make a program that greets you by name. Write the following program:

```
puts "Hello there, and what's your name?"
name = gets
puts "Your name is #{name}? What a lovely name!"
puts "Pleased to meet you, #{name}. :)"
```

I ran it, and this is what happened:

```
❮ Hello there, and what's your name?
⇒ Chris
❮ Your name is Chris
  ? What a lovely name!
  Pleased to meet you, Chris
  . :)
```

Yikes! This program first prompts for a name, and then it prints a message. But something's wrong. When I typed the letters C, h, r, i, and s, and then pressed Enter, the gets method got all of the letters in my name *and* the Enter. Well, that's not good. Fortunately, there's a method that deals with exactly this sort of thing: chomp. The chomp method takes off any newline characters hanging out at the end of your string. Let's try that program again, but this time using chomp to help tidy up the response:

```
puts "Hello there, and what's your name?"
name = gets.chomp
puts "Your name is #{name}? What a lovely name!"
puts "Pleased to meet you, #{name}. :)"
```

```
❮ Hello there, and what's your name?
⇒ Chris
❮ Your name is Chris? What a lovely name!
  Pleased to meet you, Chris. :)
```

Much better. Notice that since name is pointing to gets.chomp, you don't ever have to say name.chomp; name was already chomped. (Of course, if you did chomp it again, it wouldn't do anything; it has no more Enter characters to chomp off. You could chomp on that string all day, and it wouldn't change it. Like week-old bubble gum.)

Now you're ready to write your own truly interactive programs! Try it out with these exercises. (Or ignore these and try it out with your own ideas. That works, too.)

A Few Things to Try

- *Full name greeting.* Write a program that asks for a person's first name, then middle name, and then last name. Finally, have the program greet the person using their full name.

- *Bigger, better favorite number.* Write a program that asks for a person's favorite number. Have your program add 1 to the number, and then suggest the result as a bigger and better favorite number. (Please be tactful about it, though.)