Extracted from:

Text Processing with Ruby

Extract Value from the Data That Surrounds You

This PDF file contains pages extracted from *Text Processing with Ruby*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Text Processing with Ruby

Extract Value from the Data That Surrounds You

Rob Miller

edited by Jacquelyn Carter

Text Processing with Ruby

Extract Value from the Data That Surrounds You

Rob Miller

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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 *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Cathleen Small; Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

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

Copyright © 2015 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-070-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 3

Shell One-Liners

We've looked at processing text in Ruby scripts, but there exists a stage of text processing in which writing full-blown scripts isn't the correct approach. It might be because the problem you're trying to solve is temporary, where you don't want the solution hanging around. It might be that the problem is particularly lightweight or simple, unworthy of being committed to a file. Or it might be that you're in the early stages of formulating a solution and are just trying to explore things for now.

In such cases, it would be advantageous to be able to process text from the command line, without having to go to the trouble of committing your thoughts to a file. This would allow you to quickly throw together text processing pipelines and scratch whatever particular itch that you have—either solving the problem directly or forming the foundation of a future, more solid solution.

Such processing pipelines will inevitably make use of standard Unix utilities, such as `cat`, `grep`, `cut`, and so on. In fact, those utilities might actually be sufficient—tasks like these are, after all, what they're designed for. But it's common to encounter problems that get just a little too complex for them, or that for some reason aren't well suited to the way they work. At times like these, it would nice if we could introduce Ruby into this workflow, allowing us to perform the more complex parts of the processing in a language that's familiar to us.

It turns out that Ruby comes with a whole host of features that make it a cinch to integrate it into such workflows. First, we need to discover how we can use it to execute code from the command line. Then we can explore different ways to process input within pipelines and some tricks for avoiding lengthy boilerplate—something that's very important when we're writing scripts as many times as we run them!

Arguments to the Ruby Interpreter

You probably learned on your first day of programming Ruby that you can invoke Ruby from the command line by passing it the filename of a script to run:

```
$ ruby foo.rb
```

This will execute the code found in foo.rb, but otherwise it won't do anything too special. If you've ever written Ruby on the command line, you'll definitely have started Ruby in this way.

What you might not know is that by passing options to the ruby command, you can alter the behavior of the interpreter. There are three key options that will make life much easier when writing one-liners in the shell. The first is essential, freeing you from having to store code in files; the second and third allow you to skip a lot of boilerplate code when working with input. Let's take a look at each them in turn.

Passing Code with the -e Switch

By default, the Ruby interpreter assumes that you'll pass it a file that contains code. This file can contain references to other files (require and load statements, for example), but Ruby expects us to pass it a single file in which execution will begin.

When it comes to using Ruby in the shell, this is hugely limiting. We don't want to have to store code in files; we want to be able to compose it on the command line as we go.

By using the -e flag when invoking Ruby, we can execute code that we pass in directly on the command line—removing the need to commit our script to a file on disk. (It might be helpful to remember -e as standing for *evaluate*, because Ruby is evaluating the code we pass contained within this option.) The universal "hello world" example, then, would be as follows:

```
$ ruby -e 'puts "Hello world"'
Hello world
```

Any code that we could write in a script file can be passed on the command line in this way. We could, though it wouldn't be much fun, define classes and methods, require libraries, and generally write a full-blown script, but in all likelihood we'll limit our code to relatively short snippets that just do a couple of things. Indeed, this desire to keep things short will lead to making choices that favor terseness over even readability, which isn't usually the choice we make when writing scripts.

This is the first step toward being able to use Ruby in an ad hoc pipeline: it frees us from having to write our scripts to the filesystem. The second step is to be able to read from input. After all, if we want our script to be able to behave as part of a pipeline, as we saw in the previous chapter, then it needs to be able to read from standard input.

The obvious solution might be to read from STDIN in the code that we pass in to Ruby, looping over it line by line as we did in the previous chapter:

```
$ printf "foo\nbar\n" | ruby -e 'STDIN.each { |line| puts line.upcase }'
FOO
BAR
```

But this is a bit clunky. Considering how often we'll want to process input line by line, it would be much nicer if we didn't have to write this tedious boilerplate every time. Luckily, we don't. Ruby offers a shortcut for just this use case.

Streaming Lines with the -n Switch

If we pass Ruby the -n switch as well as -e, Ruby will act as though the code we pass to it was wrapped in the following:

```
while gets
  # execute code passed in -e here
end
```

This means that the code we pass in the -e argument is executed once for each line in our input. The content of the line is stored in the $_ variable. This is one of Ruby's many global variables, sometimes referred to as *cryptic globals*, and it always points to the last line that was read by gets.

So instead of writing the clunky looping example that we saw earlier:

```
$ printf "foo\nbar\n" | ruby -e 'STDIN.each { |line| puts line.upcase }'
FOO
BAR
```

we can simply write:

```
$ printf "foo\nbar\n" | ruby -ne 'puts $_.upcase'
FOO
BAR
</code>
<p> There's more to <inlinecode>$_</inlinecode> than this, though.
  Ruby also defines some global methods that either act on
  <inlinecode>$_</inlinecode> or have it as a default argument.
  <ic>print</ic> is one of them: if you call it with no arguments,
  it will output the value of <inlinecode>$_</inlinecode>. So we
  can output the input that we receive with this short script:
</p>
[code language="session"]
$ printf "foo\nbar\n" | ruby -ne 'print'
foo
bar
```

This implicit behavior is particularly useful for filtering down the input to only those lines that match a certain condition—only those that start with f, for example:

```
$ printf "foo\nbar\n" | ruby -ne 'print if $_.start_with? "f"'
foo
```

This kind of conditional output can be made even more terse with another shortcut. As well as print, regular expressions also operate implicitly on $_. We'll be covering regular expressions in depth in Chapter 8, *Regular Expressions Basics*, on page ?, but if in the previous example we changed our start_with? call to use a regular expression instead, it would read:

```
$ printf "foo\nbar\n" | ruby -ne 'print if /^f/'
```

This one-liner is brief almost to the point of being magical; the subject of both the print statement and the if are both completely implicit. But one-liners like this are optimized more for typing speed than for clarity, and so tricks like this—which have a subtlety that might be frowned upon in more permanent scripts—are a boon.

There are also shortcut methods for manipulating input. If we invoke Ruby with either the -n or -p flag, Ruby creates two global methods for us: sub and gsub. These act just like their ordinary string counterparts, but they operate on $_ implicitly.

This means we can perform search and replace operations on our lines of input in a really simple way. For example, to replace all instances of COBOL with Ruby:

```
$ echo 'COBOL is the best!' | ruby -ne 'print gsub("COBOL", "Ruby")'
Ruby is the best!
```

We didn't need to call $_.gsub, as you might expect, since the gsub method operates on $_ automatically. This is a really handy shortcut.

Handy shortcuts are nice to have, but these techniques are fundamentally useful, too. This line-by-line looping that we achieve with -n enables many types of processing. We've seen *conditional output*, where we output only those lines that match certain criteria, and we've seen *filtering*, where we output a modified form of our input. Many scripts involve both of these steps, sometimes even multiple times.

In a full-blown script, we might not consider these to be distinct steps in a process, worthy of considering separately. We certainly wouldn't write multiple scripts, one to handle each one of them. But it can often help when writing a one-liner to frame things in this way. For example, let's take the following text file, which contains the names of students and their scores on a test:

```
Bob 40
Alice 98
Gillian 100
Fred 67
```

Let's imagine we want to output the name of any student who scored more than 50 on the test (sorry, Bob). There are two steps here: conditional output, to select only those students who scored more than 50; and filtering, to show only the name, rather than the name and score together. With a one-liner, it makes sense to treat those two things separately.

The first step is typically to output the contents of the file. This is an important psychological step, if nothing else, because it validates that the data is there and is in the format we're expecting:

```
$ cat scores.txt
Bob 40
Alice 98
Gillian 100
Fred 67
```

Next, we need to perform our filtering step. In this case, we need to take the second word of the line, convert it to a number, and check whether it's greater than 50:

```
$ cat scores.txt | \
  ruby -ne 'print if $_.split[1].to_i > 50'
Alice 98
Gillian 100
Fred 67
```

(The \ characters at the end of the line escape the line break, meaning that when you press Enter a newline is inserted rather than the shell executing the command you've typed. They're included here just to keep lines shorter on the page; although you can type them into your shell, in practice you wouldn't.)

Finally we perform our filter step, to output only the names:

```
$ cat scores.txt | \
  ruby -ne 'print if $_.split[1].to_i > 50' | \
  ruby -ne 'puts $_.split.first'
Alice
Gillian
Fred
```

Although we could potentially have performed these two steps together, breaking up the problem in this way has two important advantages. The first is performance: the two steps will run in parallel, so on greater input we might see a speedup. The second, though, is that we might discover that a particular step can be performed with an existing tool—in which case there's no need write any code of our own. That's the case here, in fact. We could have used `cut` for the second step:

```
$ cat scores.txt | \
  ruby -ne 'print if $_.split[1].to_i > 50' | \
  cut -d' ' -f1
Alice
Gillian
Fred
```

Here we tell `cut` that we want to delimit text by the space character, and to take the first field. If we keep our commands small and have them do just one job, then they're easier to compose together—either with more scripts of our own or with commands that already exist. That means solving problems more quickly and with less typing, which is always one of our goals when programming.

The `-n` switch is very useful, but Ruby has more one-liner gifts. There's another useful option that makes it even easier to write filter scripts.

Printing Lines with the -p Switch

We've seen that, when writing scripts to perform filter operations on input, we're very often taking a line of input, making a modification to it, and then outputting it. This pattern is common enough that Ruby offers us another switch to help: `-p`.

Used instead of `-n`, the `-p` switch acts similarly in that it loops over each of the lines in the input. However, it goes a bit further: after our code has finished, it always prints the value of `$_`. So, we can imagine it as:

```
while gets
  # execute code passed in -e here
  puts $_
```

```
end
```

Generally, it's useful when the code we pass using -e does something that modifies the current line, but where that output isn't conditional—in other words, where we're expecting to modify and output every line of the input.

So, returning to our find-and-replace example, we could use gsub! to modify the contents of $_:

```
$ echo 'COBOL is the best!' | ruby -pe '$_.gsub!("COBOL", "Ruby")'
Ruby is the best!
```

However, we can go shorter still. This operation is so common that Ruby provides a shortcut for it:

```
$ echo 'COBOL is the best!' | ruby -pe 'gsub("COBOL", "Ruby")'
Ruby is the best!
```

We know that the global gsub method operates on $_, but it also modifies it. In that sense, it operates like $_.gsub!. So after our call to gsub, $_ has been modified, and the implicit puts outputs our transformed text.