The Pragmatic Programmers

Programming Ruby 3.3

The Pragmatic Programmers' Guide

Noel Rappin
with Dave Thomas
Edited by Katharine Dvorak

The Facets of Ruby Series

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.prag-prog.com.

If you're using Ruby as a scripting language, you'll be starting it from the command line. In this chapter, we'll look at how to use Ruby as a command-line tool and how to interact with your operating system environment. The two most common ways for a Ruby program to kick off from the command line are with the Ruby interpreter itself and with Rake, a utility that makes it easy to define a series of interrelated tasks. You also might want to create your own command-line programs, and Ruby can help with that as well.

Please note that some of the details of this chapter only apply to Unix-based systems like Linux, MacOS, and WSL.

Calling the Ruby Command

The most direct way to start the Ruby interpreter and run a Ruby program is by calling the `ruby` command from the command line. Regardless of the system in which Ruby is deployed, you have to start the Ruby interpreter somehow, and doing so with the `ruby` command gives us the opportunity to pass in command-line arguments both to Ruby itself and to the script being run.

A Ruby command-line call consists of three parts, none of which are required: options for the Ruby interpreter itself, the name of a program to run, and arguments for that program.

ruby ‹*options*› ‹–› ‹*programfile*› ‹*arguments*›*

You only need the double-dash if you're separating options to Ruby itself from options being passed to the program being run.

The simplest Ruby command that you are likely to use is `ruby` followed by a filename:

```
$ ruby my_code.rb
```

This command will cause the Ruby interpreter to load the `my_code.rb` file, parse it, and then execute it.

If the file has a syntax error, Ruby will attempt to locate the error and suggest where the problem is.

Here's an example:

rubyworld/syntax_error.rb

```
class HasAnError
  def this_method_ends
    p "it sure does"
  end

  def this_doesnt_end
    return "a thing"

  def this_one_is_also_right
    p "fine"
  end
end
```

```
$ ruby code/rubyworld/syntax_error.rb
code/rubyworld/syntax_error.rb: --> code/rubyworld/syntax_error.rb

Unmatched keyword, missing `end' ?

   1  class HasAnError
```

```
>  6    def this_doesnt_end
>  9    def this_one_is_also_right
> 11    end
  12  end
code/rubyworld/syntax_error.rb:12: syntax errors found (SyntaxError)
  10 |     p "fine"
  11 |   end
> 12 | end
     |    ^ expected an `end` to close the `class` statement
     |    ^ unexpected end-of-input, assuming it is closing
     |    ^ the parent top level context
```

Ruby notices the error—a missing end—and goes to some lengths attempts to find the location of the item missing the end and present the context to you. In this case, it gets it right.

Any options after the command ruby are sent to the Ruby interpreter. The Ruby interpreter options end with the first word on the command line that doesn't start with a hyphen or with the special flag -- (two hyphens).

There are ways to invoke the Ruby interpreter without passing it a filename. One way is to use the -e command-line option, which executes one line of script.

This lets us use Ruby as a powerful command-line calculator. Here's a one-liner that returns the first five square numbers:

```
ruby -e "p (1..5).map { _1 ** 2 }"
[1, 4, 9, 16, 25]
```

When you do this, remember that the command you run needs to be a string and that you have to print it, or you won't see the result.

You can also pipe a file into the command using Unix standard input and then access that file using Kernel#gets:

```
ruby -e 'puts "line: #{gets}"' < testfile
```

That works swimmingly, but it only processes the first line of the file. However, we can use Ruby's while expression clause to loop over the file in a single line:

```
$ ruby -e 'puts "line: #{$_}" while gets' < testfile
line: This is line one
line: This is line two
line: This is line three
line: And so on...
```

In this snippet, we're not only taking advantage of the while clause, we're also using the Ruby global $_, which contains the most recent text read in by a gets call. So, the while gets reads the line and puts it in $_ and the body of the statement prints out the line.

Still, that while at the end seems kind of awkward for something you might do often. If only there were some kind of shortcut:

```
$ ruby -ne 'puts "line: #{$_}"' < testfile
line: This is line one
line: This is line two
line: This is line three
line: And so on...
```

The -n command-line option wraps whatever else is sent to the Ruby interpreter in a while gets; <INPUT>; end loop. This is frequently a single line passed in using -e, but it doesn't have to be. You could have a script file that processes a single line of input and use -n to apply that script to an entire input.

Now, looking at it, that puts statement seems like boilerplate, and it turns out there's a shortcut for that as well…sort of:

```
ruby -pe '"line: #{$_}"' < testfile
This is line one
This is line two
This is line three
And so on...
```

The -p option behaves like n but also prints the line as it is being input, not the line that we're processing, which is sometimes helpful.

There's one more twist to the looping input, which is -a for auto-split mode. With -a set, the incoming gets line is automatically split using String#split, and the result goes into the global $F. The default delimiter is space, but you can set a delimiter with the command-line option -F, as in -F"\n". So this code uses -a to split the line from the input:

```
$ ruby -nae 'puts "line: #{$F}"' < testfile
line: ["This", "is", "line", "one"]
line: ["This", "is", "line", "two"]
line: ["This", "is", "line", "three"]
line: ["And", "so", "on..."]
```

And this code uses -F to also set a custom delimiter:

```
$ ruby -F"i" -nae 'puts "line: #{$F}"' < testfile
line: ["Th", "s ", "s l", "ne one\n"]
line: ["Th", "s ", "s l", "ne two\n"]
line: ["Th", "s ", "s l", "ne three\n"]
line: ["And so on...\n"]
```

And just to clear one thing up—the options can be stacked if they don't have any arguments, so -nae is identical to -n -a -e.

If no filename is present on the command line or if the filename is a single hyphen, Ruby reads the program source from standard input.

Arguments for the program itself follow the program name:

```
$ ruby -w - "Hello World"
```

In this snippet, -w will enable warnings, and then Ruby will read a program from standard input, and pass that program the string "Hello World" as an argument. We'll talk in a moment about how to deal with incoming command-line arguments.

Ruby Command-Line Options

Following is a complete list of Ruby's command-line options roughly organized by functionality.

Options That Determine What Ruby Runs

-0[octal]

The 0 flag (the digit zero) specifies the record separator character (\0, if no digit follows). -00 indicates paragraph mode: records are separated by two successive default record separator characters. 0777 reads the entire file at once (because it's an illegal character). Sets $/.

-a

Autosplit mode when used with -n or -p; equivalent to executing $F = $_.split at the top of each loop iteration.

-c

Checks syntax only; does not execute the program.

--copyright

Prints the copyright notice and exits.

-e 'command'

Executes *command* as one line of Ruby source. Several -es are allowed, and the commands are treated as multiple lines in the same program. If *programfile* is omitted when -e is present, execution stops after the -e commands have been run. Programs that run using -e can use ranges and regular expressions in conditions—ranges of integers compare against the current input line number, and regular expressions match against $_.

-F pattern

Specifies the input field separator ($;) used as the default for split (affects the -a option).

-h, --help

Displays a help screen. The -h option prints a short version, The --help prints a the complete version.

-l

Enables automatic line-ending processing; sets $\ to the value of $/ and chops every input line automatically.

-n

Assumes a while gets; ...; end loop around your program. For example, a simple grep command could be implemented as follows:

```
$ ruby -n -e "print if /wombat/"  *.txt
```

-p

Places your program code within the loop while gets; ...; print; end.

```
$ ruby -p -e "$_.downcase!" *.txt
```

--version

Displays the Ruby version number and exits.

Options That Change the Way the Interpreter Works

--backtrace-limit=num

Sets a limit on the number of lines of backtrace that are sent to standard error when the program sends a backtrace (when the program terminates unexpectedly, for example). The default value is -1, meaning unlimited backtrace.

-C directory

Changes working directory to directory before executing.

-d, --debug

Sets $DEBUG and $VERBOSE to true. This can be used by your programs to enable additional tracing.

--disable={FEATURE}

Disables one of the features described in Features That Can Be Enabled or Disabled, on page 8.

-Eex[:in], --encoding=ex[:in], external-encoding=encoding, internal-encoding=encoding

Specifies the default character encoding for data read from and written to the outside world. This can be used to set both the external encoding (the encoding to be assumed for file contents) and optionally the default internal encoding (the file contents are transcoded to this when read and transcoded from this when written). The format of the single *encoding* parameter is -E external, -E external:internal, or -E :internal.

--enable={FEATURE}

Enables one of the features described in Features That Can Be Enabled or Disabled, on page 8.

-i [extension]

Edits ARGV files in place. For each file named in ARGV, anything you write to standard output will be saved back as the contents of that file. A backup copy of the file will be made if the *extension* is supplied, as in the following code sample:

```
$ ruby -pi.bak -e "gsub(/Perl/, 'Ruby')" *.txt
```

--jit, --yjit

Enables one of the two just-in-time compilers available in Ruby. The Rust-based compiler can be enabled with --jit or --yjit. The JIT compilers are designed to improve program performance in long-running Ruby applications. Both compilers have several of their own command-line options.

-I directories

Specifies directories to be prepended to $LOAD_PATH ($:). Multiple -I options may be present. Multiple directories may appear following each -I, separated by a colon on Unix-like systems and by a semicolon on DOS/Windows systems.

--parser=PARSER

Specifies which parser to use, the default is Prism, the new parser, The older parser can be accessed with parse.y.

-r library

Requires the named library or gem before executing.

-S

Looks for the program file using the RUBYPATH or PATH environment variable.

-s

Any command-line switches found after the program filename, but before any filename arguments or before a --, are removed from ARGV and set to a global variable named for the switch. In the following example, the effect of this would be to set the variable $opt to "electric":

```
$ ruby -s prog -opt=electric ./mydata
```

-v, --verbose

Sets $VERBOSE to true, which enables verbose mode. Also prints the version number. In verbose mode, compilation warnings are printed. If no program filename appears on the command line, Ruby exits.

-w

Enables warning mode, which is like verbose mode, except it reads the program from standard input if no program files are present on the command line. We recommend running your Ruby programs with -w.

-W level|:category

Sets the level of warnings issued. With a *level* of 2 (or with no level specified), which is the equivalent to -w, additional warnings are given. If *level* is 1, it runs at the standard (default) warning level. With -W0, absolutely no warnings are given (including those issued using Object.warn). If a category is specified, then warnings are emitted related to that category. The available categories are: deprecated which warns when a deprecated feature is used, experimental which warns when an experimental feature is used, performance which warns for some specific performance issues, and strict_unused_block which warns if a block is passed to a method that does not use it.

-x [directory]

Strips off text before #!ruby line and changes working directory to *directory* if given.

-y --yydebug

Causes the parser log to be printed.

Other Options

--crash-report=TEMPLATE

Sets a template for a crash report file, if Ruby crashes with this option the template is used to generate a report file. The environment variable RUBY_CRASH_REPORT specifies the location.

--dump option...

Tells Ruby to dump various items of internal state. options... is a comma or space separated list containing one or more of insns, insns_without_opt, parsetree, parsetree_with_comment, and yydebug. This is intended for Ruby core developers.

Features That Can Be Enabled or Disabled

All of these features can be explicitly enabled or disabled from the command line, using an option such as --enable=gems or --disable=did_you_mean.

did_you_mean

When enabled, a NameError will also show the results of a search of the receiving object for similarly named messages that might have been the intended message. Helpful when you can't remember the name of the message you want. Enabled by default.

error_highlight

When enabled, error messages will have arrows highlighting the exact part of the line where the error was triggered. Useful in tracking down errors in a long line of code that chains multiple method calls. Enabled by default.

frozen-string-literal

When enabled, acts as if the magic comment # frozen_string_literal: true is placed at the front of all Ruby files. This comment causes all string literals to be implicitly frozen without freeze being called on them. Disabled by default.

gems

Stops Ruby from automatically loading RubyGems from require. Enabled by default.

rubyopt

Prevents Ruby from examining the RUBYOPT environment variable. You should probably disable this in an environment you want to secure. Enabled by default.

syntax_suggest

Enables the syntax_suggest tool which provides better handling of syntax errors when code is loaded.

yjit

Enables the yjit compiler. Disabled by default.