

Debugging Ruby programs 101

How to use Ruby's debugger to fix problems in your code

Skill Level: Introductory

[Pat Eyer](mailto:pat.eyler@gmail.com) (pat.eyler@gmail.com)

Co-founder

Seattle Ruby Brigade

06 Sep 2005

This tutorial shows how to use the Ruby debugger to find and fix problems with your Ruby code. It also presents several other methods for finding and fixing problems in your code. After presenting an overview of debugging and a primer for the Ruby debugger, we move on to show the debugger in use.

Section 1. Before you start

Prerequisites

This tutorial teaches you how to use the debugger shipped with Ruby. It also presents several other methods for finding and fixing problems in your code. After presenting an overview of debugging and a primer for the Ruby debugger, this tutorial moves on to show the debugger in use.

This tutorial is written for the novice Ruby Programmer. To complete it successfully, you should have Ruby installed on your system, and you need some experience with the language.

To run the examples or sample code in this tutorial, you need a current version of Ruby (V1.8.2, as of this writing). Everything else you need is built right into that version.

Section 2. An introduction to debugging

First, some other options

The debugger is a powerful and important tool, but it shouldn't be the only one you use. With other tools and skills at your disposal, sometimes firing up the debugger to solve your problem makes as much sense as trying to pound in screws with a hammer.

I recommend four practices that will help you avoid spending time in the debugger:

- Unit testing
- Pair programming (or, to a lesser extent, good code reviews)
- Strategic use of the `print` or `puts` method
- Use of the `-c` and `-d` flags

I discuss each of these individually on the following pages. If you already have a handle on them (or you're convinced that all you need is a hammer), you can skip ahead.

Good reasons to have a debugger and to know how to use one do exist. Sometimes a bug is subtle enough that it's hard to catch without a debugger. Stepping through code to see what it's really doing is an excellent way to get a feel for a new body of code you've just taken over. Besides, do you really want to pound nails with a screwdriver?

Unit testing

Unit testing, particularly when you write your code test first, helps you to write code correctly as you go along. Because you test methods individually to ensure that they behave correctly even when they receive incorrect or marginal input, bugs show up quickly and are localized to your most recent change.

To see how unit testing works in practice, let's look at an example. In developing `r43`, I wrote an initial version that handled the keys only if they were strings. When one of my tests tried to use a numerical string, the test failed and the bug was immediately obvious, as was a deeper flaw in my design. Fortunately, both were easy to fix after the unit tests unearthed them.

Pair programming

Pair programming puts two sets of eyes on your code at all times. If you can't pair with another programmer, getting a detailed code review is a second choice. You don't get all the benefits, but it still helps. This practice helps reduce tactical and strategic bugs.

I think of bugs as falling into two large classes: tactical and strategic. *Tactical bugs* are the ones you think of most often: Your code behaves incorrectly -- miscalculating a result, for example. *Strategic bugs* occur when your code behaves correctly, but is at odds with the application's design -- such as calculating the correct result for the wrong process.

The following example describes a pair-programming session adding a new method to the `r43` library. Tom and Bill are adding the `get_person` method. Tom starts by writing a `test_get_person` method for the unit test, and Bill begins to write the code to pass the test. (Trading roles like this is called *ping-pong pairing* -- a silly-sounding name for a good idea.)

As Bill is writing code, Tom watches what he writes and asks about the pieces that don't make sense. Seeing that Bill is writing his own `http get` statement, Tom can pull things back by asking, "Don't we already have a `get_response` method taking care of that? If not, we might need to refactor a bit here before we start repeating ourselves." If Bill's mistake is tactical, rather than strategic, this kind of cooperation catches things just as quickly.

Using print and puts

Using the `print` or `puts` method might seem like an archaic alternative to the debugger, but they're a quick and proven method of watching a variable during program execution without having to step through that execution in a debugger. Listing 1 shows a simple example -- a script to find bash users by reading through `/etc/passwd`.

Listing 1. shell_grabber.rb

```
file = "/etc/passwd"
File.open(file).each do |line|
  re = %r{bash}
  if line =~ re then
    print line.split(':')[0], " uses bash\n"
  end
end
```

This script prints a user (`alanc`), who doesn't use `bash` at all. You can use the debugger to find this information, or you can modify your code to look like Listing 2, and run it like this: `$./shell_grabber.rb | grep alanc`. Running this code shows you what `alanc`'s `/etc/passwd` entry looks like and why it shows up in your `bash` users list. Because his name, Alan Cabash, is matched by the regex `/bash/`, you need to change your regex to something more specific, like `%r{/bin/bash}`.

Listing 2. Debugging with print

```
file = "/etc/passwd"
File.open(file).each do |line|
  re = %r{bash}
  if line =~ re then
    puts line
    print line.split(':')[0], " uses bash\n"
  end
end
```

Ruby command-line switches

Two switches can help you debug problems more effectively: `-c` and `-d`. Unlike many command-line switches, they don't interact well with each other. (For example, `-cd` behaves like `-c`.) Each offers a different kind of help, so I'll explain them independently.

The `-c` switch causes the Ruby interpreter to compile, but not execute, the file. Unless your file contains fatal syntax errors, you'll see the message `Syntax OK` upon completion. If your code has fatal errors, you'll get a warning message like `has_errors.rb:4: odd number list for Hash`.

To use the `-c` switch, run your code like this: `ruby -c has_errors.rb`.

The `-d` flag sets the `$DEBUG` variable (running your code in debug mode, but not in a debugger). This flag can be useful in conjunction with the debugging pattern of using `print` and `puts` methods.

To use the `-d` switch, run your code like this: `ruby -d debuggable.rb`. Listing 3 shows a script set up to use the `$DEBUG` flag, and its output with and without the `-d` switch.

Listing 3. Using the -d switch

```
$ cat debuggable.rb
#!/usr/local/bin/ruby
a = ARGV[0].to_i
```

```
a.upto(5) do |foo|
  if $DEBUG then
    puts foo
  end
  print "."
end
puts

$ ruby debuggable.rb
.....
$ ruby -d debuggable.rb
0
.1
.2
.3
.4
.5
.
$
```

Ruby also has a `-w` switch, but the error messages generated without it seem to be just as useful. For example, trying to access a variable before you've declared it typically generates the error `undefined local variable or method 'foo' for main:Object (NameError)`. (Note, Ruby isn't statically typed, so declaring a variable isn't as big a deal as it is in C, the Java™ programming language, or Perl.)

Section 3. A debugger primer

Getting to know the debugger

The Ruby debugger isn't a separate program. Rather, it's a library you can pull into any Ruby program to debug it. Running the debugger is as easy as `ruby -rdebug r43.rb` (assuming you want to debug the `r43` library). This drops you into the debugger and gives you an interactive prompt that looks something like Listing 4.

Listing 4. Entering the Ruby debugger

```
ruby -rdebug r43.rb
Debug.rb
Emacs support available.

r43.rb:5:require 'net/http'
(rdb:1)
```

You can also run the debugger against individual Ruby expressions, or one-liners, composed on the command line. In this case, you do something like `ruby -rdebug`

-e1. (Yes, the numeral 1 is a complete, if not very interesting, Ruby script.)

At this point, you can use the debugger to execute arbitrary bits of Ruby and watch what it does -- arguably, `irb` provides the same ability and is a lot easier to deal with.

After you enter the debugger, you can execute a variety of commands. I'll briefly cover each of these in a moment. They're broken into groups as follows:

- **Administrative commands**
 - `h[elp]`
 - `p`
 - `q[uit]`
- **Basic navigation**
 - `c[ontinue]`
 - `l[ist]`
 - `n[ext]`
 - `s[tep]`
- **Flow control commands**
 - `b[reak]`
 - `cat[ch]`
 - `del[ete]`
 - `wat[ch]`
- **Inspection commands**
 - `disp[lay]`
 - `m[ethod]`
 - `m[ethod] i[nstance]`
 - `trace`
 - `v[ariable] c[onstant]`
 - `v[ariable] g[lobal]`
 - `v[ariable] i[nstance]`
 - `v[ariable] l[ocal]`

- `undisp[lay]`
- **Frame-related commands**
 - `down`
 - `fin[ish]`
 - `f[rame]`
 - `up`
 - `w[here]`

The debugger has three more classes of commands, but instead of adding them to this list, I'll explain them here.

First are the thread-related commands. I haven't talked about Ruby's threading model in any of the tutorials so far, so I won't address these here.

The second class is Ruby commands executed in the debugger. The debugger parses and executes as normal Ruby code any command not shown in the preceding list.

The third class is the empty command -- when the debugger sees an empty command, it repeats the last issued command.

Jargon

Before I describe the debugging commands, a short course in Ruby and debugging jargon is in order. I'll present the most important terms here (take good notes; there will be a quiz later):

- **Breakpoint** -- A point at which the debugger should stop running code and return control to the command line.
- **Catchpoint** -- An exception the debugger should watch for. If it's caught, the debugger should return control to the command line.
- **Constant** -- An unchanging value. In Ruby, the name of a constant begins with a capital letter. For example, `Cat = "Anya"` creates a constant named `Cat` with a value of `"Anya"`.
- **Expression** -- An executable bit of Ruby code.
- **Frame** (also called a *stack frame*) -- The portion of memory allocated to Ruby that's in use by the current scope of the running program.
- **Global** -- A variable that can be accessed from any part of the running

program.

- **Instance** -- An individual case of an object or something related to it (like an instance method).
- **Line** -- Normally, a piece of code on one line. In Ruby, some "lines" span more than one actual line of code, and some actual lines of code have more than one "line," as far as the debugger is concerned.
- **Stepping** -- To move through code, normally (but not always) one line at a time.
- **Watchpoint** -- An expression the debugger should watch. If it evaluates to true, the debugger should return control to the command line.

Administrative commands

Entering the `h[elp]` command gives you a listing of the available commands with a brief summary of what each one does.

The `p` command allows you to print the value of an arbitrary expression. Because the return value of calling a variable (or an object, to be more precise) is the value of that variable (or object), you can use `p` to print the value of a variable. Listing 5 shows some examples of `p`.

Listing 5. Using the `p` command

```
(rdb:1) foo = "bar"
"bar"
(rdb:1) p foo
"bar"
(rdb:1) p 1+2
3
```

Use `q[uit]` to break out of a running debugging session. When you use this command, the debugger prompts you to be sure you really want to exit, as shown in Listing 6. (You can also quit by pressing **Ctrl-D** or by typing `exit`. These options don't prompt you, however; they exit immediately.)

Listing 6. Quitting the debugger

```
(rdb:1) q
Really quit? (y/n) y
$
```

Oddly enough, the Python interpreter takes a different approach to handling multiple exit commands. It recognizes both `exit` and `quit`, but not `q` by itself. Instead of

exiting the interpreter, Python recommends you try **Ctrl-D**.

Basic navigation

The `c[ontinue]` command executes the program until it ends or hits the next breakpoint (discussed in [Flow control commands](#)). If `c` doesn't hit any breakpoints and continues to the end of the program, the debugger exits, as shown in Listing 7.

Listing 7. The `c[ontinue]` command in the debugger

```
$ ruby -rdebug averager.rb
Debug.rb
Emacs support available.

averager.rb:3:class Array
(rdb:1) c
.....
$
```

Use the `l[ist]` command to list the program. By default, it lists the five lines on either side of your current position in the program (shown with a `=>` marker); succeeding calls show the 10 lines following the previous call.

The `l` command takes one optional argument: either a `-` or a number range like "5-20." The former lists 10 lines before your current position, and the latter shows the lines in the range listed. Listing 8 shows an example.

Listing 8. How to `l[ist]` a program in the debugger

```
$ ruby -rdebug scripts/averager.rb
Debug.rb
Emacs support available.

scripts/averager.rb:3:class Array
(rdb:1) l
[-2, 7] in scripts/averager.rb
1
2
=> 3 class Array
4
5 def average
6 result = 0
7 self.each { |x| result += x }
(rdb:1) l
[8, 17] in scripts/averager.rb
8 result / self.size.to_f
9 end
10
11 end
12
13 max_loop = (ARGV.shift || 5).to_i
14 max_size = (ARGV.shift || 100_000).to_i
15 a = (1..max_size).to_a
```

```

16
17 1.upto(max_loop) do
(rdb:1) 1 1-10
[1, 10] in scripts/averager.rb
1
2
=> 3 class Array
4
5 def average
6 result = 0
7 self.each { |x| result += x }
8 result / self.size.to_f
9 end
10
(rdb:1) 1 -
[-9, 0] in scripts/averager.rb
(rdb:1)

```

The `n[ext]` command goes over one line, passing over methods. You can give it an optional numeric argument, in which case it proceeds to that line (unless it hits a breakpoint first). It also passes over lines that close a code block (such as `end` statements or `}`).

In Listing 9, a breakpoint is set at line 19 to help show how this command works. Notice that although the debugger starts at line 3 of the program, the first `n` command passes right over the `class Array` definition to line 13.

The next command, `n 20`, stops at the breakpoint on line 19 instead of going all the way to line 20. A third command, `n` alone, falls through line 20 (the `end` of a code block) to line 18. A fourth `n` goes from line 18 to 19, as you might expect.

Listing 9. The `n[ext]` command in the debugger

```

(rdb:1) 1
[-2, 7] in averager.rb
1
2
=> 3 class Array
4
5 def average
6 result = 0
7 self.each { |x| result += x }
(rdb:1) n
averager.rb:13:max_loop = (ARGV.shift || 5).to_i
(rdb:1) n 20
Breakpoint 1, toplevel at averager.rb:19
averager.rb:19: $stderr.print "."
(rdb:1) 1
[14, 23] in averager.rb
14 max_size = (ARGV.shift || 100_000).to_i
15 a = (1..max_size).to_a
16
17 1.upto(max_loop) do
18 avg = a.average
=> 19 $stderr.print "."
20 end
21 $stderr.puts ""
22

```

```
(rdb:1) n
.averager.rb:18: avg = a.average
(rdb:1) n
averager.rb:19: $stderr.print "."
(rdb:1)
```

The `s[tep]` command is very much like the `n` command, except that it steps into the next line (including methods), instead of over it. You can also give it an optional numeric argument, indicating that it should step that many lines ahead -- subject to the same exceptions as for `n`.

Listing 10 shows how `s` works. Again, a breakpoint is set on line 19.

Listing 10. The `s[tep]` command in the debugger

```
(rdb:1) l
[-2, 7] in averager.rb
1
2
=> 3 class Array
4
5 def average
6 result = 0
7 self.each { |x| result += x }
(rdb:1) s 17
averager.rb:7:
(rdb:1) l
[2, 11] in averager.rb
2
3 class Array
4
5 def average
6 result = 0
=> 7 self.each { |x| result += x }
8 result / self.size.to_f
9 end
10
11 end
(rdb:1)
```

Hey, what just happened? We asked the debugger to step forward 17 lines, and it went only three. Well, the debugger hit line 18 (nine lines, according to the debugger) and went back into the `average` method defined earlier. At that point, it began stepping through the block defined and stopped eight executable lines into the work.

Flow control commands

Use `b[reak]` to set or list breakpoints for your debugging session. You can set breakpoints by position in a file or class, or by an expression. For example, if you want to set a breakpoint at line 19 in a program, you use `b 19`.

Although that's the simplest way to set a breakpoint, it's not the only way. You can set breakpoints by method name and specify the file or class that contains the line or method. You can also use `b` to list all the breakpoints in the current session.

Listing 11 shows several uses of the `b` command.

Listing 11. The `b[reak]` command in the debugger

```
(rdb:1) b 19
Set breakpoint 1 at averager.rb:19
(rdb:1) n 20
Breakpoint 1, toplevel at averager.rb:19
averager.rb:19:
(rdb:1) l
[14, 23] in averager.rb
14 max_size = (ARGV.shift || 10).to_i
15 a = (1..max_size).to_a
16
17 1.upto(max_loop) do
18 avg = a.average
=> 19 $stderr.print "."
20 end
21 $stderr.puts ""
22
(rdb:1) b average
Set breakpoint 2 at averager.rb:average
(rdb:1) c
.Breakpoint 2, average at averager.rb:average
averager.rb:5:
(rdb:1) l
[0, 9] in averager.rb
1
2
3 class Array
4
=> 5 def average
6 result = 0
7 self.each { |x| result += x }
8 result / self.size.to_f
9 end
(rdb:1) b
Breakpoints:
  1 averager.rb:19
  2 averager.rb:average
(rdb:1)
```

The `cat[ch]` points watch for an exception to be thrown, then catch it. They are defined like this: `cat <Exception>`. The command `cat` without arguments shows a listing of which catchpoints are currently set. Listing 12 shows an example.

Listing 12. Catchpoints in the debugger

```
(rdb:1) cat NotImplementedError
Set catchpoint NotImplementedError.
(rdb:1) c
.....
Loaded suite test_averager
```

```

Started
test_averager.rb:10: 'Need to write test_average' (NotImplementedError)
  from /usr/local/lib/ruby/1.8/test/unit/testsuite.rb:32:in 'run'
  from /usr/local/lib/ruby/1.8/test/unit/testsuite.rb:31:in 'each'
  from /usr/local/lib/ruby/1.8/test/unit/testsuite.rb:31:in 'run'
  from /usr/local/lib/ruby/1.8/test/unit/ui/testrunnermedi
ator.rb:44:in 'run_suite'
  from /usr/local/lib/ruby/1.8/test/unit/ui/console/testru
nner.rb:65:in 'start_mediator'
  from /usr/local/lib/ruby/1.8/test/unit/ui/console/testru
nner.rb:39:in 'start'
  from /usr/local/lib/ruby/1.8/test/unit/ui/testru
nnerutilities.rb:27:in 'run'
  from /usr/local/lib/ruby/1.8/test/unit/autorunner.rb:194:in 'run'
  from /usr/local/lib/ruby/1.8/test/unit/autorunner.rb:14:in 'run'
  from /usr/local/lib/ruby/1.8/test/unit.rb:285
  from /usr/local/lib/ruby/1.8/test/unit.rb:283
test_averager.rb:10:
(rdb:1) l
[5, 14] in test_averager.rb
5  require 'test/unit' unless defined? $ZENTEST and $ZENTEST
6  require 'averager'
7
8  class TestArray < Test::Unit::TestCase
9  def test_average
=> 10 raise NotImplementedError, 'Need to write test_average'
11 end
12
13 end
14
(rdb:1) cat
Catchpoint NotImplementedError.
(rdb:1)

```

Use the `del[ete]` command to delete one or all of the breakpoints currently set. Issued without arguments, it prompts for confirmation before deleting all the current breakpoints. Listing 13 shows some examples.

Listing 13. The `del[ete]` command in the debugger

```

(rdb:1) b
Breakpoints:
  1 averager.rb:6
  2 averager.rb:13
  3 averager.rb:average
  4 averager.rb:19

(rdb:1) del
Clear all breakpoints? (y/n) n
(rdb:1) del 1
(rdb:1) b
Breakpoints:
  2 averager.rb:13
  3 averager.rb:average
  4 averager.rb:19

(rdb:1) del
Clear all breakpoints? (y/n) y
(rdb:1) b
Breakpoints:

(rdb:1)

```

The `wat[ch]` command sets or displays watchpoints in the code under debugging. It takes a Ruby expression as an argument, and it returns when that argument is true. Listing 14 shows a simple example.

Listing 14. The `wat[ch]` command in the debugger

```
(rdb:1) l
[-1, 8] in debuggable.rb
1
2
3
=> 4 a = ARGV[0].to_i
5
6 a.upto(30) do |foo|
7 puts foo
8 end
(rdb:1) s
debuggable.rb:6:
(rdb:1) wat foo == 10
Set watchpoint 1
(rdb:1) c
1
2
3
4
5
6
7
8
9
Watchpoint 1, toplevel at debuggable.rb:7
debuggable.rb:7:
(rdb:1)
```

Inspection commands

The `disp[lay]` command allows you to set one or more expressions to display after each command. These expressions may be any valid Ruby code. Issued without arguments, `disp` prints a listing of currently displayed expressions. Listing 15 shows an example.

Listing 15. The `disp[lay]` command in the debugger

```
(rdb:1) disp max_loop
1: max_loop =
(rdb:1) n
averager.rb:13:
1: max_loop =
(rdb:1) n
averager.rb:13:
1: max_loop =
(rdb:1) l
[8, 17] in averager.rb
8 result / self.size.to_f
9 end
```

```

10
11 end
12
=> 13 max_loop = (ARGV.shift || 5).to_i
14 max_size = (ARGV.shift || 10).to_i
15 a = (1..max_size).to_a
16
17 1.upto(max_loop) do
(rdb:1) n
averager.rb:14:
1: max_loop = 5
(rdb:1) 1
[9, 18] in averager.rb
9 end
10
11 end
12
13 max_loop = (ARGV.shift || 5).to_i
=> 14 max_size = (ARGV.shift || 10).to_i
15 a = (1..max_size).to_a
16
17 1.upto(max_loop) do
18 avg = a.average
(rdb:1) disp 1+3
2: 1+3 = 4
(rdb:1) s
averager.rb:17:
1: max_loop = 5
2: 1+3 = 4
(rdb:1) disp
1: max_loop =
2: 1+3 = 4
(rdb:1)

```

The `m[method]` command takes the name of a class or module as an argument and returns a list of that class' or module's instance methods. Listing 16 shows an example.

Listing 16. The `m[method]` command in the debugger

```

(rdb:1) m Array
& * + - =<=< =<==> == [] []= assoc at clear collect collect! compact
compact! concat delete delete_at delete_if each each_index empty?
eql? fetch fill first flatten flatten! frozen? hash include? index
indexes indices insert inspect join last length map map! nitems pack
pop pretty_print pretty_print_cycle push rassoc reject reject!
replace reverse reverse! reverse_each rindex select shift size slice
slice! sort sort! to_a to_ary to_s transpose uniq uniq! unshift
values_at zip |

```

To see the methods available on a specific object, use the `m[method] i[instance]` command. It takes an object as its sole argument. Listing 17 shows an example.

Listing 17. The `m[method] i[instance]` command in the debugger

```

(rdb:1) m i a
& * + - =<=< =<==> == == ~ [] []= __id__ __send__ all? any? assoc at
average class clear clone collect collect! compact compact! concat
delete delete_at delete_if detect display dup each each_index

```

```

each_with_index empty? entries eql? equal? extend fetch fill find
find_all first flatten flatten! freeze frozen? grep hash id include?
index indexes indices inject insert inspect instance_eval
instance_of? instance_variable_get instance_variable_set
instance_variables is_a? join kind_of? last length map map! max
member? method methods min nil? nitems object_id pack partition pop
pretty_print pretty_print_cycle pretty_print_inspect
pretty_print_instance_variables private_methods protected_methods
public_methods push rassoc reject reject! replace respond_to? reverse
reverse! reverse_each rindex select send shift singleton_methods size
slice slice! sort sort! sort_by taint tainted? to_a to_ary to_s
transpose type uniq uniq! unshift untaint values_at zip |
(rdb:1)

```

The `trace` command turns tracing on or off based on the value of its single argument. The command `trace on` turns tracing on, and `trace off` turns it off. When you turn on tracing, the debugger prints every line of code it executes. Listing 18 shows a sample debugging session using tracing.

Listing 18. The trace command in the debugger

```

build_dns.rb:4:class Inventory
(rdb:1) b 66
Set breakpoint 1 at build_dns.rb:66
(rdb:1) trace on
Trace on.
(rdb:1) c
#0:build_dns.rb:4:Class:>: class Inventory
#0:build_dns.rb:4:Class:<: class Inventory
#0:build_dns.rb:4:C: class Inventory
#0:build_dns.rb:6::-:attr_reader :hosts
#0:build_dns.rb:6:Module:>:attr_reader :hosts
#0:build_dns.rb:6:Module:>:attr_reader :hosts
#0:build_dns.rb:6:Module:<:attr_reader :hosts
#0:build_dns.rb:6:Module:<:attr_reader :hosts
#0:build_dns.rb:8::-:def initialize(inv_csv)
#0:build_dns.rb:8:Module:>:def initialize(inv_csv)
#0:build_dns.rb:8:Module:<:def initialize(inv_csv)
#0:build_dns.rb:23::-:def build_forward
#0:build_dns.rb:23:Module:>:def build_forward
#0:build_dns.rb:23:Module:<:def build_forward
#0:build_dns.rb:34::-:def build_reverse
#0:build_dns.rb:34:Module:>:def build_reverse
#0:build_dns.rb:34:Module:<:def build_reverse
#0:build_dns.rb:44::-:def build_dhcp
#0:build_dns.rb:44:Module:>:def build_dhcp
#0:build_dns.rb:44:Module:<:def build_dhcp
#0:build_dns.rb:4:E: class Inventory
#0:build_dns.rb:65::-: if __FILE__ == $0
#0:build_dns.rb:65::-: if __FILE__ == $0
#0:build_dns.rb:65:String:>: if __FILE__ == $0
#0:build_dns.rb:65:String:<: if __FILE__ == $0
#0:build_dns.rb:66::-:inv = Inventory.new(File.op
en("e_equinix_inv_by_rack.csv").read)
Breakpoint 1, toplevel at build_dns.rb:66
build_dns.rb:66: inv = Inventory.new(File.op
en("e_equinix_inv_by_rack.csv").read)
(rdb:1)

```

The `v[ariable] c[onstant]` command displays all the constants currently set that are associated with a class or module. You give the class or module as an

argument to the command.

Listing 19. The v[ariable] c[onstant] command in the debugger

```
(rdb:1) b 20
Set breakpoint 1 at const_obj.rb:20
(rdb:1) c
Breakpoint 1, toplevel at const_obj.rb:20
const_obj.rb:20:
(rdb:1) v c Foo
  First => 1
  Second => 2
  Third => 3
(rdb:1)
```

Use the v[ariable] g[lobal] command to see the values of the global variables currently set in your program. Even an empty Ruby script has a number of globals, as you can see in Listing 20.

Listing 20. The v[ariable] g[lobal] command in the debugger

```
$ ruby -r debug -e '0'
Debug.rb
Emacs support available.

-e:1:
(rdb:1) v g
$! => nil
$" => ["debug.rb", "tracer.rb", "pp.rb",
"prettyprint.rb", "readline.so"]
$$ => 7885
$& => nil
$& => nil
$' => nil
$' => nil
$* => []
$+ => nil
$+ => nil
$, => nil
$-0 => "\n"
$-F => nil
$-I => ["/usr/local/lib/ruby/site_ruby/1.8",
"/usr/local/lib/ruby/site_ruby/1.8/i686-linux",
"/usr/local/lib/ruby/site_ruby",
"/usr/local/lib/ruby/1.8",
"/usr/local/lib/ruby/1.8/i686-linux", "."]
$-K => "NONE"
$a => false
$d => false
$i => nil
$l => false
$p => false
$v => false
$w => false
$. => 914
$/ => "\n"
$0 => "-e"
$1 => nil
$2 => nil
$3 => nil
```

```

$4 => nil
$5 => nil
$6 => nil
$7 => nil
$8 => nil
$9 => nil
$: => ["/usr/local/lib/ruby/site_ruby/1.8",
"/usr/local/lib/ruby/site_ruby/1.8/i686-linux",
"/usr/local/lib/ruby/site_ruby",
"/usr/local/lib/ruby/1.8",
"/usr/local/lib/ruby/1.8/i686-linux", "."]
$; => nil
$< => ARGF
$= => false
$> => #<IO:0x401d5078>
$? => nil
$@ => nil
$DEBUG => false
$FILENAME => "-"
$KCODE => "NONE"
$LOADED_FEATURES => ["debug.rb", "tracer.rb",
"pp.rb", "prettyprint.rb",
"readline.so"]
$LOAD_PATH => ["/usr/local/lib/ruby/site_ruby/1.8",
"/usr/local/lib/ruby/site_ruby/1.8/i686-linux",
"/usr/local/lib/ruby/site_ruby",
"/usr/local/lib/ruby/1.8",
"/usr/local/lib/ruby/1.8/i686-linux", "."]
$PROGRAM_NAME => "-e"
$SAFE => 0
$VERBOSE => false
$\ => nil
$_ => nil
$' => nil
$' => nil
$deferr => #<IO:0x401d5064>
$defout => #<IO:0x401d5078>
$stderr => #<IO:0x401d5064>
$stdin => #<IO:0x401d508c>
$stdout => #<IO:0x401d5078>
$~ => nil
(rdb:1)

```

The `v[ariable] i[nstance]` command takes an object as an argument and displays all the instance variables associated with it. Listing 21 shows an example.

Listing 21. The `v[ariable] i[nstance]` command in the debugger

```

const_obj.rb:1:
(rdb:1) l
[-4, 5] in const_obj.rb
=> 1 class Foo
2
3 First = 1
4 Second = 2
5 Third = 3
(rdb:1) l
[6, 15] in const_obj.rb
6
7 def initialize
8 @first = "first"
9 @second = "second"
10 @third = "third"
11 end

```

```

12
13
14 end
15
(rdb:1) l
[16, 25] in const_obj.rb
16
17 foo = Foo.new
18
19 bar = Foo.new
20
21 puts "done"
(rdb:1) b 19
Set breakpoint 1 at const_obj.rb:19
(rdb:1) c
Breakpoint 1, toplevel at const_obj.rb:19
const_obj.rb:19:
(rdb:1) v i foo
  @first => "first"
  @second => "second"
  @third => "third"
(rdb:1)

```

To see a listing of variables available in your current scope, use the `v[ariable]` `l[ocal]` command. Listing 22 shows an example.

Listing 22. The `v[ariable]` `l[ocal]` command in the debugger

```

(rdb:1) l
[1, 10] in debuggable.rb
1
2
3
=> 4 a = ARGV[0].to_i
5
6 a.upto(30) do |foo|
7 puts foo
8 end
debuggable.rb:4:
(rdb:1) v l
  a => nil
(rdb:1) s
debuggable.rb:6:
(rdb:1) v l
  a => 1
(rdb:1) s
debuggable.rb:7:
(rdb:1) v l
  a => 1
  foo => 1
(rdb:1)

```

Having set the debugger to print the current value of one or more Ruby expressions with the `disp` command, you may find that you need to turn off one or all of your displayed expressions. The `undisp[lay]` command is just the ticket. Listing 23 contains a few examples of the `undisp` command.

Listing 23. The `undisp[lay]` command in the debugger

```
(rdb:1) disp a
1: a = 1
(rdb:1) disp foo
2: foo = 1
(rdb:1) s
1
debuggable.rb:7:
1: a = 1
2: foo = 2
(rdb:1) s
2
debuggable.rb:7:
1: a = 1
2: foo = 3
(rdb:1) s
3
debuggable.rb:7:
1: a = 1
2: foo = 4
(rdb:1) undisp
Clear all expressions? (y/n) n
(rdb:1) undisp 2
(rdb:1) s
4
debuggable.rb:7:
1: a = 1
(rdb:1) undisp
Clear all expressions? (y/n) y
(rdb:1)
```

Frame-related commands

The `down` command allows you to climb back down the stack if you're not already at its bottom. You can pass an optional argument to `down` specifying how many levels you want to descend.

You can step up through the stack using the `fin[ish]` command. This command continues execution until it hits the end of the outer stack frame.

The `up` command climbs up the stack. You can pass an argument to `up` specifying how many frames up you want to go.

You can find out what the stack trace looks like and what your location in it is with the `w[here]` command.

Listing 24 presents examples of the various frame-related commands together.

Listing 24. Using the frame commands in the debugger

```
(rdb:1) w
--> #1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
#2 ./r43.rb:261:in '_get_response'
#3 ./r43.rb:282:in 'get_response'
#4 ./r43.rb:566:in 'initialize'
#5 ./r43.rb:312:in '_get_response'
```

```

#6 ./r43.rb:526:in 'get_city'
#7 43_city.rb:10
(rdb:1) up
#2 ./r43.rb:261:in '_get_response'
(rdb:1) v 1
url => "get_city?api_key=1234&id=77"
(rdb:1) down
#1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
(rdb:1) w
--> #1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
#2 ./r43.rb:261:in '_get_response'
#3 ./r43.rb:282:in 'get_response'
#4 ./r43.rb:566:in 'initialize'
#5 ./r43.rb:312:in '_get_response'
#6 ./r43.rb:526:in 'get_city'
#7 43_city.rb:10
(rdb:1) up
#2 ./r43.rb:261:in '_get_response'
(rdb:1) w
#1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
--> #2 ./r43.rb:261:in '_get_response'
#3 ./r43.rb:282:in 'get_response'
#4 ./r43.rb:566:in 'initialize'
#5 ./r43.rb:312:in '_get_response'
#6 ./r43.rb:526:in 'get_city'
#7 43_city.rb:10
(rdb:1) up
#3 ./r43.rb:282:in 'get_response'
(rdb:1) w
#1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
#2 ./r43.rb:261:in '_get_response'
--> #3 ./r43.rb:282:in 'get_response'
#4 ./r43.rb:566:in 'initialize'
#5 ./r43.rb:312:in '_get_response'
#6 ./r43.rb:526:in 'get_city'
#7 43_city.rb:10
(rdb:1) down
#2 ./r43.rb:261:in '_get_response'
(rdb:1) down
#1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
(rdb:1) down
At stack bottom
#1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'
(rdb:1) fin
./r43.rb:634:
(rdb:1) w
--> #1 ./r43.rb:634:in '_from_xml'
#2 ./r43.rb:566:in 'initialize'
#3 ./r43.rb:312:in '_get_response'
#4 ./r43.rb:526:in 'get_city'
#5 43_city.rb:10
(rdb:1) up 3
#4 ./r43.rb:526:in 'get_city'
(rdb:1) down
#3 ./r43.rb:312:in '_get_response'
(rdb:1) w
#1 ./r43.rb:634:in '_from_xml'
#2 ./r43.rb:566:in 'initialize'
--> #3 ./r43.rb:312:in '_get_response'
#4 ./r43.rb:526:in 'get_city'
#5 43_city.rb:10
(rdb:1) finish
43_city.rb:11:
(rdb:1) w
--> #1 43_city.rb:11
(rdb:1)

```

Section 4. Putting it all together

The debugging cycle

In a presentation to the Seattle Ruby Brigade, Doug Beaver talked about *the basic debugging cycle*. (See [Resources](#) for this presentation.) He proposed that debugging sessions ought to follow this pattern:

1. Spawn a debugger on the program
2. Skip past class definitions
3. List the code
4. Set breakpoints
5. Continue
6. Inspect variables at a breakpoint and step through the code
7. Continue when you're done

In the remainder of this tutorial, I'll start with this pattern and gradually add some embellishments. I'll use code from the r43 library for the debugging sessions from here on out. I won't list the entire library in this tutorial, though -- just a method or two as they're debugged.

To see the whole thing, download the library (see [Resources](#)).

Basic debugging

Let's begin with a program that uses the r43 library. Listing 25 shows `43_city.rb`, a script that grabs cities matching city IDs listed on the command line.

Listing 25. `43_city.rb`

```
#!/usr/local/bin/ruby
require 'r43'
connection = R43::Request.new(1234)
```

```

if ARGV[0] != nil then
  ARGV.each do |my_city|
    city = connection.get_city(my_city)
    puts "#{city.city_id} has #{city.num_people} signed up."
  end
else
  puts "Can't find cities without an id"
end

```

As written, this code fails (well, it runs, but it doesn't list the cities by name as you'd prefer). Let's follow Doug's pattern and see what you can find.

Start by spawning a debugger on the program: `ruby -rdebug 43_city.rb 77`. After the debugger is running, skip past any class definitions (in this case, there aren't any), list the code, and set a breakpoint at line 9. With a breakpoint set, you can continue to that point. Listing 26 shows these steps.

Listing 26. A basic debugging session, part 1

```

(rdb:1) l
[-2, 7] in 43_city.rb
1
2
=>> 3 require 'r43'
4
5 connection = R43::Request.new(1234)
6
7
(rdb:1) l
[8, 17] in 43_city.rb
8 if ARGV[0] != nil then
9 ARGV.each do |my_city|
10 city = connection.get_city(my_city)
11 puts "#{city.city_id} has #{city.num_people} signed up."
12 end
13 else
14 puts "Can't find cities without an id"
15 end
(rdb:1) b 9
Set breakpoint 1 at 43_city.rb:9
(rdb:1) c
Breakpoint 1, toplevel at 43_city.rb:9
43_city.rb:9: ARGV.each do |my_city|
(rdb:1)

```

At this point, you can begin to look around and see what's going on. Let's see what's in `ARGV` and `my_city`. Listing 27 shows the commands and the responses they generate.

Listing 27. A basic debugging session, part 2

```

(rdb:1) p my_city
43_city.rb:9:undefined local variable or method 'my_city' for main:Object
(rdb:1) p ARGV

```

```
["77"]
(rdb:1) s
43_city.rb:10: city = connection.get_city(my_city)
(rdb:1) p my_city
"77"
(rdb:1)
```

Everything looks good so far. On line 9, `my_city` hadn't been set yet, but when you step ahead to line 10, you see that it's been set correctly. On the next step forward (note that you should use `n` here and skip over the call into `r43`), you'll get a `city` object from the `get_city` method of an `R43::Request` object. Then you can start to look at the instance variables of the object and at its methods. Listing 28 shows the next steps.

Listing 28. A basic debugging session, part 3

```
(rdb:1) n
43_city.rb:11:
(rdb:1) p city.city_id
77
(rdb:1) m i city
== == ~ __id__ __send__ city_id city_id= class clone country
country= display dup eql? equal? extend freeze frozen? goals goals=
hash id inspect instance_eval instance_of? instance_variable_get
instance_variable_set instance_variables is_a? kind_of? link link=
method methods name name= nil? num_people num_people= object_id
pretty_print pretty_print_cycle pretty_print_inspect
pretty_print_instance_variables private_methods protected_methods
public_methods region region= respond_to? send singleton_methods
taint tainted? to_a to_s type untaint
(rdb:1) p city.name
"Saint John"
(rdb:1)
```

Aha! There's the problem: You're asking for `city.city_id`, which returns a numerical ID for the city. After using `m i city` to see what other methods are available, you can see that `city.name` is available. Trying that in the debugger gives the name of a city -- just what you're looking for. Now you can make the appropriate change in the code and lay this bug to rest.

A cleaner way

OK -- so you've used the debugger to walk through a simple problem. You can certainly clean up this process, though. For example, instead of printing variables manually with `p my_city`, you can use the `disp` command and let Ruby do it for you. Listing 29 provides an example.

Listing 29. Using the `disp` command

```
43_city.rb:3:
(rdb:1) b 8
Set breakpoint 1 at 43_city.rb:8
(rdb:1) disp my_city
1: my_city =
(rdb:1) c
Breakpoint 1, toplevel at 43_city.rb:8
43_city.rb:8:
1: my_city =
(rdb:1) s
43_city.rb:8:
1: my_city =
(rdb:1) s
43_city.rb:9:
1: my_city =
(rdb:1) s
43_city.rb:10:
1: my_city = 77
(rdb:1)
```

What else can you do to make this process easier? Well, you know that nothing's going to happen until you get a value into `my_city`, so set a watchpoint like this: `wat my_city != nil`. It will trigger as soon as `my_city` has a real value. Although this step doesn't provide a huge benefit in this situation, if you didn't know where the value was being set, it could save you a lot of stepping around.

You might also be interested in seeing what's happening behind that `c` command. Use the `trace` command to find out.

Watch out, though: Because the debugger is now a part of your program, you'll find that in addition to all the code you skip over, a variety of debugger-specific code is mixed in. When I used the `disp`, `trace on`, and `c` commands to follow the same path shown in Listing 26, I ended up with 4,827 lines of code displayed in my debugger session.

Digging deeper

Let's go back to the `43_city.rb` example. The last time you ran through it, you didn't look into the `r43.rb` code at all. What if a bug lurked there? Well, it's easy enough to step into it, so let's give it a try. Listing 30 shows the first couple of steps.

Listing 30. Deeper debugging

```
43_city.rb:3:
(rdb:1) b 5
Set breakpoint 1 at 43_city.rb:5
(rdb:1) c
Breakpoint 1, toplevel at 43_city.rb:5
43_city.rb:5:
(rdb:1) s
./r43.rb:301:
(rdb:1) w
```

```
--> #1 ./r43.rb:301:in 'initialize'
#2 43_city.rb:5
(rdb:1) s
./r43.rb:226:
(rdb:1) w
--> #1 ./r43.rb:226:in 'initialize'
#2 ./r43.rb:301:in 'initialize'
#3 43_city.rb:5
(rdb:1) v 1
key => 1234
proxy_addr => nil
proxy_pass => nil
proxy_port => nil
proxy_user => nil
(rdb:1)
```

At this point, you're in the third stack frame and have a small collection of local variables. After stepping forward through the initialization of both the `R43::Request` and the `R43::Service` objects and into the body of the code, you climbed back up into the top-level stack frame. Then, on line 10, you begin to dive back down, as shown in Listing 31.

Listing 31. Deeper debugging, continued

```
43_city.rb:10:
(rdb:1) s
r43.rb:526:
(rdb:1) s
r43.rb:312:
(rdb:1) w
--> #1 ./r43.rb:312:in '_get_response'
#2 ./r43.rb:526:in 'get_city'
#3 43_city.rb:10
(rdb:1) v 1
url => "get_city?id=77"
```

As you continue to step forward through the `43_city.rb` code, you can watch yourself move through the underlying code. At times, you'll find yourself in the Ruby library code, like this: `--> #1 /usr/local/lib/ruby/1.8/net/http.rb:324:in 'start'`. At any step of the way, you can peek at the local variables, method instances, or surrounding code. It's all there, available to help sort out problems.

Section 5. Summary

More than the debugger

This tutorial presented a collection of debugging tools and patterns other than the debugger. It then ran through the commands available in a debugging session. Finally, it presented several cases showing the debugger in action.

Resources

Learn

- You can get more information about the [Ruby programming language](#).
- Another great site is [Ruby Central](#), maintained by Dave Thomas and Andy Hunt (the Pragmatic Programmers).
- The best place to go for Ruby documentation is the [Ruby-doc.org](#).
- Local Ruby Brigades (like the [Seattle.rb](#)) are great places to learn more about Ruby. You can find more information at the [RubyGarden Wiki](#) on Ruby User Groups.
- Doug Beaver's [Ruby Debugging guide](#) is a great resource for debugging Ruby code.
- I use the [r43 library](#) as an example in this tutorial.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Pat Eyler

Pat Eyler is a co-founder of the [Seattle Ruby Brigade](#) and has been an active member of the Ruby community for nearly five years. He currently maintains the r43 library. Pat has written about free software, Linux, Ruby, and networking for several print- and Web-based publishers.