Chapter 18

Interactive Ruby Shell

Back on page 221 we introduced irb, a Ruby module that lets you enter Ruby programs interactively and see the results immediately. This chapter goes into more detail on using and customizing irb.

Command Line

irb is run from the command line:

```
irb [ irb-options ] [ ruby_script ] [ program arguments ]
```

The command-line options for irb are listed in Table 18.1 on the following page. Typically, you'll run irb with no options, but if you want to run a script and watch the blow-by-blow description as it runs, you can provide the name of the Ruby script and any options for that script.

Once started, irb displays a prompt and waits for input. In the examples that follow, we'll use irb's default prompt, which shows the current binding, the indent (nesting) level, and the line number.

At a prompt, you can type Ruby code. irb includes a Ruby parser, so it knows when statements are incomplete. When this happens, the prompt will end with an asterisk. You can leave irb by typing **exit** or **quit** or by entering an end-of-file character (unless IGNORE_EOF mode is set).

```
% irb
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0> 3 +
irb(main):003:0* 4
=> 7
irb(main):004:0> quit
%
```

Table 18.1. irb Command-Line Options

Option	Description
--back-trace-limit n	Displays backtrace information using the top n and last n entries. The default value is 16.
--context-mode *n*	See :CONTEXT_MODE on page 284.
-d	Sets $DEBUG to true (same as ruby -d).
-E *enc*	Same as Ruby's -E option.
-f	Suppresses reading ~/.irbrc.
-h, --help	Displays usage information.
-I *path*	Specifies the $LOAD_PATH directory.
--inf-ruby-mode	Sets up irb to run in inf-ruby-mode under Emacs. Same as --prompt inf-ruby --noreadline.
--inspect, --noinspect	Uses/doesn't use Object#inspect to format output (--inspect is the default, unless in math mode).
--irb_debug n	Sets internal debug level to n (only useful for irb development).
-m	Math mode (fraction and matrix support is available).
--noprompt	Does not display a prompt. Same as --prompt null
--prompt *prompt-mode*	Switches prompt. Predefined prompt modes are null, default, classic, simple, xmp, and inf-ruby.
--prompt-mode *prompt-mode*	Same as --prompt.
-r *module*	Requires *module*. Same as ruby -r.
--readline, --noreadline	Uses/doesn't use readline extension module.
--sample-book-mode	Same as --prompt simple.
--simple-prompt	Same as --prompt simple.
--single-irb	Nested irb sessions will all share the same context.
--tracer	Displays trace for execution of commands.
-U	Same as Ruby's -U option.
-v, --version	Prints the version of irb.

During an irb session, the work you do is accumulated in irb's workspace. Variables you set, methods you define, and classes you create are all remembered and may be used subsequently in that session.

```
irb(main):001:0> def fib_up_to(n)
irb(main):002:1>   f1, f2 = 1, 1
irb(main):003:1>   while f1 <= n
irb(main):004:2>     puts f1
irb(main):005:2>     f1, f2 = f2, f1+f2
irb(main):006:2>   end
irb(main):007:1> end
=> nil
irb(main):008:0> fib_up_to(4)
1
1
2
3
=> nil
```

Notice the nil return values. These are the results of defining the method and then running it—our method printed the Fibonacci numbers but then returned nil.

A great use of irb is experimenting with code you've already written. Perhaps you want to track down a bug, or maybe you just want to play. If you load your program into irb, you can then create instances of the classes it defines and invoke its methods. For example, the file code/fib_up_to.rb contains the following method definition:

Download **samples/irb_1.rb**

```
def fib_up_to(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
```

We can load this into irb and play with the method:

```
% irb
irb(main):001:0> load 'code/fib_up_to.rb'
=> true
irb(main):002:0> result = []
=> []
irb(main):003:0> fib_up_to(20) {|val| result << val}
=> nil
irb(main):004:0> result
=> [1, 1, 2, 3, 5, 8, 13]
```

In this example, we use load, rather than require, to include the file in our session. We do this as a matter of practice: load allows us to load the same file multiple times, so if we find a bug and edit the file, we could reload it into our irb session.

Tab Completion

If your Ruby installation has readline support, then you can use irb's completion facility. Once loaded (and we'll get to how to load it shortly), completion changes the meaning of the TAB key when typing expressions at the irb prompt. When you press TAB partway through a word, irb will look for possible completions that make sense at that point. If there is only one, irb will fill it in automatically. If there's more than one valid option, irb initially does nothing. However, if you hit TAB again, it will display the list of valid completions at that point.

For example, you may be in the middle of an irb session, having just assigned a string object to the variable a:

```
irb(main):002:0> a = "cat"
=> "cat"
```

You now want to try the method String#reverse on this object. You start by typing a.re and then hit TAB twice.

```
irb(main):003:0> a.re[TAB][TAB]
a.reject       a.replace      a.respond_to?  a.reverse      a.reverse!
```

irb lists all the methods supported by the object in a whose names start with *re*. We see the one we want, reverse, and enter the next character of its name, *v*, followed by the [TAB] key:

```
irb(main):003:0> a.rev[TAB]
irb(main):003:0> a.reverse
=> "tac"
irb(main):004:0>
```

irb responds to the [TAB] key by expanding the name as far as it can go, in this case completing the word reverse. If we keyed [TAB] twice at this point, it would show us the current options, reverse and reverse!. However, because reverse is the one we want, we instead hit [ENTER], and the line of code is executed.

Tab completion isn't limited to built-in names. If we define a class in irb, then tab completion works when we try to invoke one of its methods:

```
irb(main):004:0> class Test
irb(main):005:1>   def my_method
irb(main):006:2>   end
irb(main):007:1> end
=> nil
irb(main):008:0> t = Test.new
=> #<Test:0x35b724>
irb(main):009:0> t.my[TAB]
irb(main):009:0> t.my_method
```

Tab completion is implemented as an extension library. On some systems this is loaded by default. On others you'll need to load it when you invoke irb from the command line:

```
% irb -r irb/completion
```

You can also load the completion library when irb is running:

```
irb(main):001:0> require 'irb/completion'
=> true
```

If you use tab completion all the time, it's probably most convenient to put the require command into your .irbrc file:

```
require 'irb/completion'
```

Subsessions

irb supports multiple, concurrent sessions. One is always current; the others lie dormant until activated. Entering the command irb within irb creates a subsession, entering the jobs command lists all sessions, and entering fg activates a particular dormant session. This example also illustrates the -r command-line option, which loads in the given file before irb starts:

```
% irb -r code/fib_up_to.rb
irb(main):001:0> result = []
=> []
```

```
irb(main):002:0> fib_up_to(10) {|val| result << val }
=> nil
irb(main):003:0> result
=> [1, 1, 2, 3, 5, 8]
irb(main):004:0> #  Create a nested irb session
irb(main):005:0* irb
irb#1(main):001:0> result = %w{ cat dog horse }
=> ["cat", "dog", "horse"]
irb#1(main):002:0> result.map {|val| val.upcase }
=> ["CAT", "DOG", "HORSE"]
irb#1(main):003:0> jobs
=> #0->irb on main (#<Thread:0x331740>: stop)
#1->irb#1 on main (#<Thread:0x341694>: running)
irb#1(main):004:0> fg 0
irb(main):006:0> result
=> [1, 1, 2, 3, 5, 8]
irb(main):007:0> fg 1
irb#1(main):005:0> result
=> ["cat", "dog", "horse"]
```

Subsessions and Bindings

If you specify an object when you create a subsession, that object becomes the value of *self* in that binding. This is a convenient way to experiment with objects. In the following example, we create a subsession with the string “wombat” as the default object. Methods with no receiver will be executed by that object.

```
% irb
irb(main):001:0> self
=> main
irb(main):002:0> irb "wombat"
irb#1(wombat):001:0> self
=> "wombat"
irb#1(wombat):002:0> upcase
=> "WOMBAT"
irb#1(wombat):003:0> size
=> 6
irb#1(wombat):004:0> gsub(/[aeiou]/, '*')
=> "w*mb*t"
irb#1(wombat):005:0> irb_exit
irb(main):003:0> self
=> main
irb(main):004:0> upcase
NameError: undefined local variable or method `upcase' for main:Object
```

Configuration

irb is remarkably configurable. You can set configuration options with command-line options, from within an initialization file, and while you’re inside irb itself.

Initialization File

irb uses an initialization file in which you can set commonly used options or execute any required Ruby statements. When irb is run, it will try to load an initialization file from one of the following sources in order: ~/.irbrc, .irbrc, irb.rc, _irbrc, and $irbrc.

Within the initialization file, you may run any arbitrary Ruby code. You can also set configuration values. The list of configuration variables is given starting on the following page—the values that can be used in an initialization file are the symbols (starting with a colon). You use these symbols to set values into the IRB.conf hash. For example, to make SIMPLE the default prompt mode for all your irb sessions, you could have the following in your initialization file:

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
```

As an interesting twist on configuring irb, you can set IRB.conf[:IRB_RC] to a Proc object. This proc will be invoked whenever the irb context is changed and will receive the configuration for that context as a parameter. You can use this facility to change the configuration dynamically based on the context. For example, the following `.irbrc` file sets the prompt so that only the main prompt shows the irb level, but continuation prompts and the result still line up:

Download **samples/irb_5.rb**

```
IRB.conf[:IRB_RC] = lambda do |conf|
  leader = " " * conf.irb_name.length
  conf.prompt_i = "#{conf.irb_name} --> "
  conf.prompt_s = leader + ' \-" '
  conf.prompt_c = leader + ' \-+ '
  conf.return_format   = leader + " ==> %s\n\n"
  puts "Welcome!"
end
```

An irb session using this `.irbrc` file looks like the following:

```
% irb
Welcome!
irb --> 1 + 2
    ==> 3

irb --> 2 +
    \-+ 6
    ==> 8
```

Extending irb

Because the things you type into irb are interpreted as Ruby code, you can effectively extend irb by defining new top-level methods. For example, you may want to time how long certain things take. You can use the measure method in the Benchmark library to do this, but it's more convenient to wrap this in a helper method.

Add the following to your .irbrc file:

Download samples/irb_6.rb

```
def time(&block)
  require 'benchmark'
  result = nil
  timing = Benchmark.measure do
    result = block.()
  end
  puts "It took: #{timing}"
  result
end
```

The next time you start irb, you'll be able to use this method to get timings:

```
irb(main):001:0> time { 1000000.times { "cat".upcase }}
It took:   0.550000   0.000000   0.550000 (  0.545647)
=> 1000000
irb(main):002:0>
```

Interactive Configuration

Most configuration values are also available while you're running irb. The list starting on the current page shows these values as conf.*xxx*. For example, to change your prompt back to DEFAULT, you could use the following:

```
irb(main):001:0> 1 +
irb(main):002:0* 2
=> 3
irb(main):003:0> conf.prompt_mode = :SIMPLE
=> :SIMPLE
>> 1 +
?> 2
=> 3
```

irb Configuration Options

In the descriptions that follow, a label of the form :XXX signifies a key used in the IRB.conf hash in an initialization file, and conf.xxx signifies a value that can be set interactively. The value in square brackets at the end of the description is the option's default.

:AUTO_INDENT / conf.auto_indent_mode
: If true, irb will indent nested structures as you type them. [false]

:BACK_TRACE_LIMIT / conf.back_trace_limit
: Displays n initial and n final lines of backtrace. [16]

:CONTEXT_MODE
: What binding to use for new workspaces: 0→ proc at the top level, 1→ binding in a loaded, anonymous file, 2→ per thread binding in a loaded file, 3→ binding in a top-level function. [3]

:DEBUG_LEVEL / conf.debug_level
Sets the internal debug level to n. This is useful if you're debugging irb's lexer. [0]

:IGNORE_EOF / conf.ignore_eof
Specifies the behavior of an end of file received on input. If true, it will be ignored; otherwise, irb will quit. [false]

:IGNORE_SIGINT / conf.ignore_sigint
If false, ^C (Ctrl+c) will quit irb. If true, ^C during input will cancel input and return to the top level; during execution, ^C will abort the current operation. [true]

:INSPECT_MODE / conf.inspect_mode
Specifies how values will be displayed: true means use inspect, false uses to_s, and nil uses inspect in nonmath mode and to_s in math mode. [nil]

:IRB_RC
Can be set to a proc object that will be called when an irb session (or subsession) is started. [nil]

conf.last_value
The last value output by irb. [...]

:LOAD_MODULES / conf.load_modules
A list of modules loaded via the -r command-line option. [[]]

:MATH_MODE / conf.math_mode
If true, irb runs with the mathn library loaded (see page 767) and does not use inspect to display values. [false]

conf.prompt_c
The prompt for a continuing statement (for example, immediately after an if). [depends]

conf.prompt_i
The standard, top-level prompt. [depends]

:PROMPT_MODE / conf.prompt_mode
The style of prompt to display. [:DEFAULT]

conf.prompt_s
The prompt for a continuing string. [depends]

:PROMPT
See *Configuring the Prompt* on page 287. [{ ... }]

:RC / conf.rc
If false, do not load an initialization file. [true]

conf.return_format
The format used to display the results of expressions entered interactively. [depends]

:SAVE_HISTORY / conf.save_history
The number of commands to save between irb sessions. [nil]

:SINGLE_IRB
If true, nested irb sessions will all share the same binding; otherwise, a new binding will be created according to the value of :CONTEXT_MODE. [nil]

conf.thread
A read-only reference to the currently executing Thread object. [current thread]

:USE_LOADER / conf.use_loader
Specifies whether irb's own file reader method is used with load/require. [false]

:USE_READLINE / conf.use_readline
irb will use the readline library if available (see page 797) unless this option is set to false, in which case readline will never be used, or nil, in which case readline will not be used in inf-ruby-mode. [depends]

:USE_TRACER / conf.use_tracer
If true, traces the execution of statements. [false]

:VERBOSE / conf.verbose
In theory, switches on additional tracing when true; in practice, almost no extra tracing results. [true]

Commands

At the irb prompt, you can enter any valid Ruby expression and see the results. You can also use any of the following commands to control the irb session:[1]

help *ClassName*, *string*, **or** *symbol*
1.9
Displays the ri help for the given thing. To get the help for a method name, you'll probably want to pass a string, like this:

```
irb(main):001:0> help "String.encoding"
------------------------------------------------- String#encoding
 obj.encoding   => encoding
------------------------------------------------------------------
 Returns the Encoding object that represents the encoding of obj.
```

exit, quit, irb_exit, irb_quit
Quits this irb session or subsession. If you've used cb to change bindings (see below), exits from this binding mode.

conf, context, irb_context
Displays current configuration. Modifying the configuration is achieved by invoking methods of conf. The list starting on page 284 shows the available conf settings.

1. For some inexplicable reason, many of these commands have up to nine different aliases. We don't bother to show all of these.

For example, to set the default prompt to something subservient, you could use this:

```
irb(main):001:0> conf.prompt_i = "Yes, Master? "
=> "Yes, Master? "
Yes, Master? 1 + 2
```

cb, irb_change_binding ⟨ ***obj*** ⟩
Creates and enters a new binding (sometimes called a *workspace*) that has its own scope for local variables. If *obj* is given, it will be used as self in the new binding.

pushb *obj*, **popb**
Pushes and pops the current binding.

bindings
Lists the current bindings.

irb_cwws
Prints the object that's the binding of the current workspace.

irb ⟨ **obj** ⟩
Starts an irb subsession. If *obj* is given, it will be used as self.

jobs, irb_jobs
Lists irb subsessions.

fg n, **irb_fg** n
Switches into the specified irb subsession. n may be any of the following: an irb subsession number, a thread ID, an irb object, or the object that was the value of *self* when a subsession was launched.

kill n, **irb_kill** n
Kills an irb subsession. n may be any of the values as described for irb_fg.

source *filename*
Loads and executes the given file, displaying the source lines.

Configuring the Prompt

You have a lot of flexibility in configuring the prompts that irb uses. Sets of prompts are stored in the prompt hash, IRB.conf[:PROMPT].

For example, to establish a new prompt mode called MY_PROMPT, you could enter the following (either directly at an irb prompt or in the .irbrc file):

```
IRB.conf[:PROMPT][:MY_PROMPT] = { # name of prompt mode
  :PROMPT_I => '-->',             # normal prompt
  :PROMPT_S => '--"',             # prompt for continuing strings
  :PROMPT_C => '--+',             # prompt for continuing statement
  :RETURN => "    ==>%s\n"        # format to return value
}
```

Once you've defined a prompt, you have to tell irb to use it. From the command line, you can use the --prompt option. (Notice how the name of the prompt mode is automatically converted to uppercase, with hyphens changing to underscores.)

```
% irb --prompt my-prompt
```

If you want to use this prompt in all your future irb sessions, you can set it as a configuration value in your `.irbrc` file:

```
IRB.conf[:PROMPT_MODE] = :MY_PROMPT
```

The symbols PROMPT_I, PROMPT_S, and PROMPT_C specify the format for each of the prompt strings. In a format string, certain % sequences are expanded:

Flag	Description
%N	Current command.
%m	to_s of the main object (self).
%M	inspect of the main object (self).
%l	Delimiter type. In strings that are continued across a line break, %l will display the type of delimiter used to begin the string, so you'll know how to end it. The delimiter will be one of ", ', /,], or `.
%ni	Indent level. The optional number n is used as a width specification to printf, as printf("%nd").
%nn	Current line number (n used as with the indent level).
%%	A literal percent sign.

For instance, the default prompt mode is defined as follows:

```
IRB.conf[:PROMPT][:DEFAULT] = {
      :PROMPT_I => "%N(%m):%03n:%i> ",
      :PROMPT_S => "%N(%m):%03n:%i%l ",
      :PROMPT_C => "%N(%m):%03n:%i* ",
      :RETURN   => "=> %s\n"
}
```

Restrictions

Because of the way irb works, it is slightly incompatible with the standard Ruby interpreter. The problem lies in the determination of local variables.

Normally, Ruby looks for an assignment statement to determine whether something is a variable—if a name hasn't been assigned to, then Ruby assumes that name is a method call:

```
eval "var = 0"
var
```

produces:

```
prog.rb:2:in `<main>': undefined local variable or method `var' for
 main:Object (NameError)
```

In this case, the assignment is there, but it's within a string, so Ruby doesn't take it into account.

irb, on the other hand, executes statements as they are entered:

```
irb(main):001:0> eval "var = 0"
0
irb(main):002:0> var
0
```

In irb, the assignment was executed before the second line was encountered, so var is correctly identified as a local variable.

If you need to match the Ruby behavior more closely, you can place these statements within a begin/end pair:

```
irb(main):001:0> begin
irb(main):002:1*   eval "var = 0"
irb(main):003:1>   var
irb(main):004:1> end
NameError: undefined local variable or method `var'
(irb):3:in `irb_binding'
```

Saving Your Session History

If you have readline support in irb (that is, you can hit the up arrow key and irb recalls the previous command you entered), then you can also configure irb to remember the commands you enter between sessions. Simply add the following to your `.irbrc` file:

Download **samples/irb_14.rb**

```
IRB.conf[:SAVE_HISTORY] = 50     # save last 50 commands
```