POST BY PETER COOPER ON MAY 26TH, 2008

# 21 Ruby Tricks You Should Be Using In Your Own Code

Permanent Link | Click here to add on del.icio.us | 77 Comments

I get to see a lot of Ruby code while writing for Ruby Inside. Most is very good, but sometimes we forget some of Ruby's shortcuts and tricks and instead reinvent the wheel. In this post I present 21 different Ruby tricks, from those that most experienced developers already use every day to those that are more obscure. Before writing this piece, for example, I had no idea about trick number 2! Whatever your level, a refresh may help you the next time you encounter similar scenarios.

***Note to beginners:*** *If you are still learning Ruby, check out my Beginning Ruby book (18 5-star reviews so far). It's only $26.39 on Amazon in print or Kindle format or available from the publisher in PDF format.*

### 1 - Extract regular expression matches quickly

A typical way to extract data from text using a regular expression is to use the *match* method. There is a shortcut, however, that can take the pain out of the process:

```
email = "Fred Bloggs <fred@bloggs.com>"
email.match(/<(.*?)>/)[1]           # => "fred@bloggs.com"
email[/<(.*?)>/, 1]                 # => "fred@bloggs.com"
email.match(/(x)/)[1]               # => NoMethodError [:(]
email[/(x)/, 1]                     # => nil
email[/([bcd]).*?([fgh])/, 2]       # => "g"
```

### 2 - Shortcut for Array#join

It's reasonably common knowledge that *Array#**, when supplied with a number, multiplies the size of the array by duplicate elements, but it's a lot lesser known that when supplied with a string *Array#** instead does a *join*!

```
%w{this is a test} * ", "               # => "this, is, a, test"
h = { :name => "Fred", :age => 77 }
h.map { |i| i * "=" } * "\n"            # => "age=77\nname=Fred"
```

### 3 - Format decimal amounts quickly

Formatting floating point numbers into a form used for prices can be done with *sprintf* or, alternatively, with a formatting interpolation:

```
money = 9.5
"%.2f" % money                      # => "9.50"
```

### 4 - Surround text quickly

The formatting interpolation technique from #3 comes out again, this time to insert a string inside another:

```
"[%s]" % "same old drag"            # => "[same old drag]"
```

You can use an array of elements to substitute in too:

```
x = %w{p hello p}
"<%s>%s</%s>" % x                   # => "<p>hello</p>"
```

### 5 - Delete trees of files

Don't resort to using the shell. Ruby comes with a handy file utilities library that can do your bidding:

```
require "fileutils"
FileUtils.rm_r "somedir"
```

Be careful how you use this one!

### 6 - Exploding enumerables

*can be used to "explode" enumerables (arrays and hashes). We'll let the examples do the talking:

```
a = %w{a b}
b = %w{c d}
[a + b]                         # => [["a", "b", "c", "d"]]
[*a + b]                        # => ["a", "b", "c", "d"]
```

```
a = { :name => "Fred", :age => 93 }
[a]                             # => [{:name => "Fred", :age =>93}]
[*a]                            # => [[:name, "Fred"], [:age, 93]]
```

```
a = %w{a b c d e f g h}
b = [0, 5, 6]
a.values_at(*b).inspect          # => ["a", "f", "g"]
```

### 7 - Cut down on local variable definitions

Instead of defining a local variable with some initial content (often just an empty hash or array), you can instead define it "on the go" so you can perform operations on it at the same time:

```
(z ||= []) << "test"
```

### 8 - Using non-strings or symbols as hash keys

It's very rare you see anyone use non-strings or symbols as hash keys. It's totally possible though, and sometimes handy (and, no, this isn't necessarily a great example!):

```
does = is = { true => "Yes", false => "No" }
does[10 == 50]                   # => "No"
is[10 > 5]                       # => "Yes"
```

### 9 - Use 'and' and 'or' to group operations for single liners

This is a trick that more confident Ruby developers use to tighten up their code and remove short multi-line if and unless statements:

```
queue = []
%w{hello x world}.each do |word|
  queue << word and puts "Added to queue" unless word.length <  2
end
puts queue.inspect

# Output:
#   Added to queue
#   Added to queue
#   ["hello", "world"]
```

### 10 - Do something only if the code is being implicitly run, not *required*

This is a very common pattern amongst experienced Ruby developers. If you're writing a Ruby script that could be used either as a library OR directly from the command line, you can use this trick to determine whether you're running the script directly or not:

```
if __FILE__ == $0
  # Do something.. run tests, call a method, etc. We're direct.
end
```

### 11 - Quick mass assignments

Mass assignment is something most Ruby developers learn early on, but it's amazing how little it's used relative to its terseness:

```
a, b, c, d = 1, 2, 3, 4
```

It can come in particularly useful for slurping method arguments that have been bundled into an array with *:

```ruby
def my_method(*args)
  a, b, c, d = args
end
```

If you want to get really smart (although this is more 'clever' than truly wise):

```ruby
def initialize(args)
  args.keys.each { |name| instance_variable_set "@" + name.to_s, args[name] }
end
```

### 12 - Use ranges instead of complex comparisons for numbers

No more *if x > 1000 && x < 2000* nonsense. Instead:

```ruby
year = 1972
puts  case year
        when 1970..1979: "Seventies"
        when 1980..1989: "Eighties"
        when 1990..1999: "Nineties"
      end
```

### 13 - Use enumerations to cut down repetitive code

```ruby
%w{rubygems daemons eventmachine}.each { |x| require x }
```

### 14 - The Ternary Operator

Another trick that's usually learned early on by Ruby developers, but again something I see quite rarely in less experienced developers' code. The ternary operator is not a fix-all, but it can sometimes make things tighter.

```ruby
puts x == 10 ? "x is ten" : "x is not ten"

# Or.. an assignment based on the results of a ternary operation:
LOG.sev_threshold = ENVIRONMENT == :development ? Logger::DEBUG : Logger::INFO
```

### 15 - Nested Ternary Operators

It may be asking for trouble but ternary operators can be nested within each other (after all, they only return objects, like everything else):

```ruby
qty = 1
qty == 0 ? "none" : qty == 1 ? "one" : "many"
# Just to illustrate, in case of confusion:
(qty == 0 ? "none" : (qty == 1 ? "one" : "many"))
```

### 16 - Fight redundancy with Ruby's logical provisions

I commonly see methods using this sort of pattern:

```ruby
def is_odd(x)
  # Wayyyy too long..
  if x % 2 == 0
    return false
  else
    return true
  end
end
```

Perhaps we can use a ternary operator to improve things?

```ruby
def is_odd(x)
  # Don't EVER put false and true in a ternary operator!!
  x % 2 == 0 ? false : true
end
```

It's shorter, and I see that pattern a **lot** but really you should go one step further and just rely on the *true / false* responses Ruby's comparison operators **already** give!

```ruby
def is_odd(x)
  # Use the logical results provided to you by Ruby already..
  x % 2 != 0
end
```

### 17 - See the whole of an exception's backtrace

```ruby
def do_division_by_zero; 5 / 0; end
begin
  do_division_by_zero
rescue => exception
  puts exception.backtrace
end
```

### 18 - Allow both single items AND arrays to be enumerated against

```ruby
# [*items] converts a single object into an array with that single object
# of converts an array back into, well, an array again
[*items].each do |item|
  # ...
end
```

### 19 - Rescue blocks don't need to be tied to a 'begin'

```ruby
def x
  begin
    # ...
  rescue
    # ...
  end
end
```

```ruby
def x
  # ...
rescue
  # ...
end
```

### 20 - Block comments

I tend to see this in more 'old-school' Ruby code. It's surprisingly under-used though, but looks a lot better than a giant row of pound signs in many cases:

```ruby
puts "x"
=begin
  this is a block comment
  You can put anything you like here!

  puts "y"
=end
puts "z"
```

### 21 - Rescue to the rescue

You can use *rescue* in its single line form to return a value when other things on the line go awry:

```ruby
h = { :age => 10 }
h[:name].downcase                       # ERROR
h[:name].downcase rescue "No name"        # => "No name"
```

If you want to post your own list of Ruby tricks to your blog, send trackback here or leave a comment, and I'll link to all of them in a future post. Alternatively, feel free to post your own Ruby tricks as comments here, or critique or improve on those above.

*As an aside, Ruby Inside is exactly two years old today. Thanks for your support! Intriguingly, the first post was **another Ruby trick** that I forgot to include above, so check that out too.*

Tweet This

This entry was posted on Monday, May 26th, 2008 at 7:11 pm and is filed under Compilation Posts, Cool, Ruby Tricks. Responses are currently closed, but you can trackback