

Ruby Safari



Ola Bini

computational metalinguist

ola.bini@gmail.com

<http://olabini.com/blog>

Sections

Regular expressions and strings

Procs and blocks

Expressions and operators

Collections

Loading

Regular expressions and strings

/foo(?#you can write here)bar/ \approx "foobar"

`/(\d{3})` # area code
`-?` # optional dash
`(\d{3})` # prefix
`-?` # optional dash
`(\d{4})` # line number
`/x`

Comments in regexps

The (?#comment) always allow comments in regular expressions

Using the /x flag allows the regexp to ignore whitespace

And Ruby comments count as whitespace

You have to use \s or backslash before whitespace to match it

Should you use it?

Yes

Sometimes

Martin's advice on dividing up a regexp with method names or local variables usually better

/x for whitespace formatting is a great aid for readability

```
m = /(?<area_code> \d{3})
```

```
-?
```

```
(?<prefix> \d{3})
```

```
-?
```

```
(?<line_number> \d{4})
```

```
/x.match("312-607-0542")
```

```
p m[:area_code], m[:prefix], m[:line_number]
```

Named capture groups

Can be referred to wherever you can refer to group indices

Only in 1.9

Or with Oniguruma

Should you use it?

Yes, always

Increases readability substantially

Makes comments less necessary

"Ola Bini" [^w+/] #=> "Ola"

"Ola Bini" [$\wedge w+ \backslash w+ /$] $\# \Rightarrow$ "Ola Bini"

"Ola Bini" [^w+\d+/] #=> nil

```
address = "Ola Bini <obini@gmail.com>"  
address[/<(.*?)>/, 1] #=> "obini@gmail.com"
```

"Ola Bini" [/<(.*?)>/, 1] #=> nil

String aref with regexps

String#[] takes a regular expression and an optional group

That group can be a symbol in 1.9

Will return the full match if no group is given

Will return the asked for group if one is given

Will return nil if the regexp doesn't match

Should you use it?

Yes

In fact, you should probably replace all your usages of `=~`, `$1` and friends with this

"foo bar"

‘foo #{no interpolation} bar’

`%{foo #{1+2} bar}`

%(foo (very flexible) bar)

`%[foo bar]`

%<foo bar>

`%?foo bar?`

%!foo bar!

%~foo bar~

`%q{foo #{1+2} {bar}}`

%Q[foo #{1+2} {bar}]

%s<a symbol>

`%x(ls -alF)`

%w[one two three]

%W[four $\{2+3\}$ six]

/foo bar/

`%r(foo bar)`

`%r[foo bar]`

`%r{foo bar}`

%r<foo bar>

%r!foo bar!

String delimiters

Strings can be delimited with “, ‘, and many versions with %

Any (almost) non-alphanumeric can be used before and after

But (), {}, [] and <> are treated specially

Several characters can be in-between for variations:

%q(foo) - non interpolated string

%Q(foo) - interpolated string (the default)

%s!foo! - creates a symbol

%w(foo bar) - Array of words

%W(foo bar) - Array of words with interpolation

%x[ls -aF] - Another version of backtick

Regex delimiters

%r and the same kind of delimiters as Strings allow regexp creation

Should you use it?

String delimiters - most of them, no!

Regex delimiters - most of them, no!

But when it makes things more readable, of course

What do you prefer?

```
"foo\"bar\"\\\"\\\""
```

or:

```
%(foo"bar""")
```

Or

```
/fox\/mouse\/horse\/\//
```

or:

```
%r[fox/mouse/horse///]
```

Procs and blocks

```
class Symbol
  def to_proc
    proc do |obj|
      obj.send self
    end
  end
end
```

```
[1,2,3].map(&:inspect)
```

The Symbol#to_proc trick

to_proc will be called when a & is used in a method call

if the object isn't already a Proc, of course

This will auto coerce a Symbol into a proc that will call the method represented by the symbol on the object sent as argument

The same approach can be used on other standard Ruby objects

What would a to_proc on a Regular Expression do?

Should you use it?

Yes

It increases readability and removes duplication:

```
foo.map {|x| x.foo}.filter {|y| y.focused?}.each{|z| z.print}
```

```
foo.map(&:foo).filter(&:focused?).each(&:print)
```

Beware the performance cost

```
class Proc
```

```
  def ===(other); !!self[other]  
  end; end
```

```
  def evenly_divisible_with(x)  
    proc { |n| n % x == 0 }  
  end
```

```
case 15
```

```
when evenly_divisible_with(2): 'Two'  
when evenly_divisible_with(3): 'Three'  
when evenly_divisible_with(5): 'Five'  
end
```

Proc triple equals

Case statements will call `===` on the left hand side

This allow you to encode some pretty strong things

But `===` is also already defined in many places

Regexp, Range, Class/Module, String, Symbol

Should you use it?

Mostly, encoding when statements with procs hide cost

In some domains it might be necessary

I have never encountered the need for it

But it's a good feature to know about

And the general `===` tricks are the more useful ones

```
def do_it(which)
  case which
  when :lambda: lambda { return 42 }
  when :proc: proc { return 43 }
  when :Proc: Proc.new { return 44 }
  end.call
  :real_exit
end
```

```
p do_it(:lambda) #=> :real_exit
p do_it(:proc) #=> :real_exit
p do_it(:Proc) #=> 44
```

Lambdas and Procs

Blocks reified with `proc` and `lambda` have `lambdaness`

This means they are more like internal methods

`Return` will return from the block

Blocks reified with `Proc.new` instead allows lexical return

From the method enclosing the place the block was created

This matches the behavior of `return` in unreified blocks

WHY???

`define_method`

In 1.9, the behavior of `proc` is now switched to `Proc.new`

Should you use it?

Well yes

But use it carefully

Use lambda where that is what you want

Since proc has changed meaning

Be mindful of the difference

Expressions and operators

```
ARGF.each_line do |line|
  if (/^#ifdef/ =~ line)..(/^#endif/ =~ line)
    puts line
  end
end
```

The flip flop operator

A range with boolean endpoints

Will keep the current state

And return true after the start point has been true once

And continue doing that until the end point returns true

Perl at its finest

Should you use it?

Almost never

But it works fine for some kinds of throwaway CLI scripts

```
class A
```

```
  def foo(x, y, z); p [x, y, z]; end  
end
```

```
class B < A
```

```
  def foo(*args)
```

```
    args[2] = 42
```

```
    super
```

```
  end
```

```
end
```

```
B.new.foo(1, 2, 3)
```

```
B.new.foo(1)
```

```
class C < A
  def foo(*args)
    args << 42
    super
  end
end

C.new.foo(1, 2, 3)
```

ZSuper and splatting

In some circumstances you can impact zsuper

It seems like a shared memory area bug

But it's used in the standard library...

Should you use it?

No, please don't!

```
def matcher(r)
  Class.new do
    define_method :"===" do |other|
      r === other
    end
  end
end
class Some < matcher /foo /
end
```

```
Some.new === "foo bar"
```

Any expression as super class

Ruby class definitions are flexible

So you can generate a class on the fly and inherit from

Most common in Struct idioms:

```
class Foo < Struct.new(:one, :two, :three)
```

Should you use it?

The Struct idiom is good and should be used

In most other places it's not obvious enough

Although it can be used to good effect in internal DSLs

```
def foo(flox = nil)
  puts "default argument" if flox.nil?
end
```

```
foo 42
```

```
foo #=> prints default argument
```

```
foo nil #=> prints default argument
```

```
def foo(flox = (flox_default=true;nil))  
  puts "default argument" if flox_default  
end
```

```
foo 42
```

```
foo #=> prints default argument
```

```
foo nil
```

Provided info for default arg

Ruby has no way of distinguishing giving arguments that are the same as the default argument

But default arguments are evaluated at the time of call

And any code can be execute here

You can even define new methods

Or classes

Or replace the whole currently running program

Or send all the missiles

Fun fact: CLOS has this built in. Only language I know that give provided information

Should you use it?

Almost always, never

But it can sometimes be a good way to create a nice API

Collections

```
["c", "y", "foo", "bar", "baxum"].
```

```
group_by(&:length) #=> {1 => ["c", "y"],  
# 3 => ["foo", "bar"],  
# 5 => ["baxum"]}
```

one = [1,2,3]

two = [9,8,7,6]

one.zip(two) $\#=>$ [[1,9],[2,8],[3,7]]

two.zip(one) $\#=>$ [[9,1],[8,2],[7,3],[6,nil]]

[3,4,5].all? { |n| n > 2 } #=> true
[3,4,5].any? { |n| n > 6 } #=> false
[3,4,5].none? { |n| n < 0 } #=> true
[3,4,5].one? { |n| n % 2 == 0 } #=> true

```
[3,4,5,6,7].each_cons(2) do |n|  
  p n  
end
```

```
# [3,4]  
# [4,5]  
# [5,6]  
# [6,7]
```

```
(3..10).each_slice(3) do |n|  
  p n  
end
```

```
# [3, 4, 5]  
# [6, 7, 8]  
# [9, 10]
```

```
["foo", "m", "aaaa"].  
max_by(&:length) #=> "aaaa"
```

Useful Enumerable methods

There are many useful Enumerable methods

If you don't know them all by heart you will duplicate their logic. Badly.

Remember that most of them take a block to customise behaviour

Should you use it?

Yes!

All the time

Every second

In fact, why aren't you using "partition" RIGHT NOW???

```
(50..60).reverse_each.  
  each_with_index.map(&:inspect)
```

Lazy enumerables

Most Enumerable methods that have to take a block will return an Enumerator if not given a block

An enumerator is a Enumerable

Which means you can combine Enumerable methods

Instead of building up each separate result in isolation

1.8.7 and 1.9

Should you use it?

Yes absolutely

No reason not to

Loading

```
puts DATA.read
```

```
__END__
```

```
Hello world
```

The data segment

All Ruby files can be ended with `__END__`

Everything after that is not evaluated when running the file

So anything can be stored there

You can access an IO object for reading it in the constant `DATA`

Common older pattern is to have tests for the file there

`eval(DATA.read) if $0 == __FILE__`

Should you use it?

Use with caution, but do use

Sometimes there is large amounts of data that a file need

But the feature is largely unknown, so not idiomatic anymore

```
require 'fileutils'
```

```
require 'stringio'
```

```
p $" #=> ["enumerator.so", "etc.bundle",  
#      "fileutils.rb", "stringio.bundle"]
```

```
$LOADED_FEATURES << "non_existing..."
```

```
require 'non_existing...' #=> false
```

Loaded features

You can find out all features that has been loaded

And you can modify the list

This list is used by require to remember

Should you use it?

It's a very useful tool for debugging

And occasionally to make changes in load patterns in 3rd-party libraries

Do use `$LOADED_FEATURES` and not `$"`

Things I haven't talked about

Most globals (\$whatever)

Taint and safe levels

Hook methods

Metaprogramming

Regex flags

Heredocs

Emacs style literals

defined?

bang bang

Questions?

OLA BINI

ThoughtWorks®

<http://olabini.com>
obini@thoughtworks.com

@olabini