Prelude

> Role models are important.
> -- Officer Alex J. Murphy / RoboCop

One thing has always bothered me as a Ruby developer - Python developers have a great programming style reference (PEP-8) and we never got an official guide, documenting Ruby coding style and best practices. And I do believe that style matters. I also believe that a great hacker community, such as Ruby has, should be quite capable of producing this coveted document.

This guide started its life as our internal company Ruby coding guidelines (written by yours truly). At some point I decided that the work I was doing might be interesting to members of the Ruby community in general and that the world had little need for another internal company guideline. But the world could certainly benefit from a community-driven and community-sanctioned set of practices, idioms and style prescriptions for Ruby programming.

Since the inception of the guide I've received a lot of feedback from members of the exceptional Ruby community around the world. Thanks for all the suggestions and the support! Together we can make a resource beneficial to each and every Ruby developer out there.

By the way, if you're into Rails you might want to check out the complementary Ruby on Rails Style Guide.

The Ruby Style Guide

This Ruby style guide recommends best practices so that real-world Ruby programmers can write code that can be maintained by other real-world Ruby programmers. A style guide that reflects real-world usage gets used, and a style guide that holds to an ideal that has been rejected by the people it is supposed to help risks not getting used at all – no matter how good it is.

The guide is separated into several sections of related rules. I've tried to add the rationale behind the rules (if it's omitted I've assumed it's pretty obvious).

I didn't come up with all the rules out of nowhere - they are mostly based on my extensive career as a professional software engineer, feedback and suggestions from members of the Ruby community and various highly regarded Ruby programming resources, such as "Programming Ruby 1.9" and "The Ruby Programming Language".

There are some areas in which there is no clear consensus in the Ruby community regarding a particular style (like string literal quoting, spacing inside hash literals, dot position in multi-line method chaining, etc.). In such scenarios all popular styles are acknowledged and it's up to you to pick one and apply it consistently.

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in Ruby itself.

Many projects have their own coding style guidelines (often derived from this guide). In the event of any conflicts, such project-specific guides take precedence for that project.

You can generate a PDF or an HTML copy of this guide using Transmuter.

RuboCop is a code analyzer, based on this style guide.

Translations of the guide are available in the following languages:

- Chinese Simplified
- Chinese Traditional
- French
- Japanese
- Korean
- Portuguese
- Russian
- Spanish
- Vietnamese

Table of Contents

Source Code Layout

> Nearly everybody is convinced that every style but their own is ugly and unreadable. Leave out the "but their own" and they're probably right...
> -- Jerry Coffin (on indentation)

- Use `UTF-8` as the source file encoding. [link]
- Use two **spaces** per indentation level (aka soft tabs). No hard tabs. [link]

```
# bad - four spaces
def some_method
    do_something
end

# good
def some_method
  do_something
end
```

- Use Unix-style line endings. (*BSD/Solaris/Linux/OS X users are covered by default, Windows users have to be extra careful.) [link]
  - If you're using Git you might want to add the following configuration setting to protect your project from Windows line endings creeping in:

```
$ git config --global core.autocrlf true
```

- Don't use `;` to separate statements and expressions. As a corollary - use one expression per line. [link]

```
# bad
puts 'foobar'; # superfluous semicolon

puts 'foo'; puts 'bar' # two expressions on the same line

# good
puts 'foobar'

puts 'foo'
puts 'bar'

puts 'foo', 'bar' # this applies to puts in particular
```

- Prefer a single-line format for class definitions with no body. [link]

```
# bad
class FooError < StandardError
end

# okish
class FooError < StandardError; end

# good
FooError = Class.new(StandardError)
```

- Avoid single-line methods. Although they are somewhat popular in the wild, there are a few peculiarities about their definition syntax that make their use undesirable. At any rate - there should be no more than one expression in a single-line method. [link]

```
# bad
def too_much; something; something_else; end

# okish - notice that the first ; is required
def no_braces_method; body end

# okish - notice that the second ; is optional
def no_braces_method; body; end

# okish - valid syntax, but no ; makes it kind of hard to read
def some_method() body end

# good
def some_method
  body
end
```

One exception to the rule are empty-body methods.

```
# good
def no_op; end
```

- Use spaces around operators, after commas, colons and semicolons, around `{` and

before `}`. Whitespace might be (mostly) irrelevant to the Ruby interpreter, but its proper use is the key to writing easily readable code. [link]

```ruby
sum = 1 + 2
a, b = 1, 2
[1, 2, 3].each { |e| puts e }
class FooError < StandardError; end
```

The only exception, regarding operators, is the exponent operator:

```ruby
# bad
e = M * c ** 2

# good
e = M * c**2
```

`{` and `}` deserve a bit of clarification, since they are used for block and hash literals, as well as embedded expressions in strings. For hash literals two styles are considered acceptable.

```ruby
# good - space after { and before }
{ one: 1, two: 2 }

# good - no space after { and before }
{one: 1, two: 2}
```

The first variant is slightly more readable (and arguably more popular in the Ruby community in general). The second variant has the advantage of adding visual difference between block and hash literals. Whichever one you pick - apply it consistently.

As far as embedded expressions go, there are also two acceptable options:

```ruby
# good - no spaces
"string#{expr}"

# ok - arguably more readable
"string#{ expr }"
```

The first style is extremely more popular and you're generally advised to stick with it. The second, on the other hand, is (arguably) a bit more readable. As with hashes - pick one style and apply it consistently.

- No spaces after `(`, `[` or before `]`, `)`. [link]

```ruby
some(arg).other
[1, 2, 3].size
```

- No space after `!`. [link]

```ruby
# bad
! something

# good
!something
```

- No space inside range literals. [link]

```ruby
# bad
1 .. 3
'a' ... 'z'

# good
1..3
'a'..'z'
```

- Indent `when` as deep as `case`. I know that many would disagree with this one, but it's the style established in both "The Ruby Programming Language" and "Programming Ruby". [link]

```ruby
# bad
case
  when song.name == 'Misty'
    puts 'Not again!'
  when song.duration > 120
    puts 'Too long!'
  when Time.now.hour > 21
    puts "It's too late"
  else
    song.play
end

# good
case
when song.name == 'Misty'
```

```
  puts 'Not again!'
when song.duration > 120
  puts 'Too long!'
when Time.now.hour > 21
  puts "It's too late"
else
  song.play
end
```

- When assigning the result of a conditional expression to a variable, preserve the usual alignment of its branches. [link]

```
# bad - pretty convoluted
kind = case year
when 1850..1889 then 'Blues'
when 1890..1909 then 'Ragtime'
when 1910..1929 then 'New Orleans Jazz'
when 1930..1939 then 'Swing'
when 1940..1950 then 'Bebop'
else 'Jazz'
end

result = if some_cond
  calc_something
else
  calc_something_else
end

# good - it's apparent what's going on
kind = case year
       when 1850..1889 then 'Blues'
       when 1890..1909 then 'Ragtime'
       when 1910..1929 then 'New Orleans Jazz'
       when 1930..1939 then 'Swing'
       when 1940..1950 then 'Bebop'
       else 'Jazz'
       end

result = if some_cond
           calc_something
         else
           calc_something_else
         end

# good (and a bit more width efficient)
kind =
  case year
  when 1850..1889 then 'Blues'
  when 1890..1909 then 'Ragtime'
  when 1910..1929 then 'New Orleans Jazz'
  when 1930..1939 then 'Swing'
  when 1940..1950 then 'Bebop'
  else 'Jazz'
  end

result =
  if some_cond
    calc_something
  else
    calc_something_else
  end
```

- Use empty lines between method definitions and also to break up a method into logical paragraphs internally. [link]

```
def some_method
  data = initialize(options)

  data.manipulate!

  data.result
end

def some_method
  result
end
```

- Avoid comma after the last parameter in a method call, especially when the parameters are not on separate lines. [link]

```
# bad - easier to move/add/remove parameters, but still not preferred
some_method(
            size,
            count,
            color,
```

```
            )

# bad
some_method(size, count, color, )

# good
some_method(size, count, color)
```

- Use spaces around the = operator when assigning default values to method parameters: [link]

```
# bad
def some_method(arg1=:default, arg2=nil, arg3=[])
  # do something...
end

# good
def some_method(arg1 = :default, arg2 = nil, arg3 = [])
  # do something...
end
```

While several Ruby books suggest the first style, the second is much more prominent in practice (and arguably a bit more readable).

- Avoid line continuation \ where not required. In practice, avoid using line continuations for anything but string concatenation. [link]

```
# bad
result = 1 - \
         2

# good (but still ugly as hell)
result = 1 \
         - 2

long_string = 'First part of the long string' \
              ' and second part of the long string'
```

- Adopt a consistent multi-line method chaining style. There are two popular styles in the Ruby community, both of which are considered good - leading . (Option A) and trailing . (Option B). [link]
    - **(Option A)** When continuing a chained method invocation on another line keep the . on the second line.

```
# bad - need to consult first line to understand second line
one.two.three.
  four

# good - it's immediately clear what's going on the second line
one.two.three
  .four
```

    - **(Option B)** When continuing a chained method invocation on another line, include the . on the first line to indicate that the expression continues.

```
# bad - need to read ahead to the second line to know that the chain continues
one.two.three
  .four

# good - it's immediately clear that the expression continues beyond the first line
one.two.three.
  four
```

A discussion on the merits of both alternative styles can be found here.

- Align the parameters of a method call if they span more than one line. When aligning parameters is not appropriate due to line-length constraints, single indent for the lines after the first is also acceptable. [link]

```
# starting point (line is too long)
def send_mail(source)
  Mailer.deliver(to: 'bob@example.com', from: 'us@example.com', subject: 'Important message', body: source.text)
end

# bad (double indent)
def send_mail(source)
  Mailer.deliver(
      to: 'bob@example.com',
      from: 'us@example.com',
      subject: 'Important message',
      body: source.text)
end
```

```ruby
# good
def send_mail(source)
  Mailer.deliver(to: 'bob@example.com',
                 from: 'us@example.com',
                 subject: 'Important message',
                 body: source.text)
end

# good (normal indent)
def send_mail(source)
  Mailer.deliver(
    to: 'bob@example.com',
    from: 'us@example.com',
    subject: 'Important message',
    body: source.text
  )
end
```

- Align the elements of array literals spanning multiple lines. [link]

```ruby
# bad - single indent
menu_item = ['Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam',
  'Baked beans', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam']

# good
menu_item = [
  'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam',
  'Baked beans', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam'
]

# good
menu_item =
  ['Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam',
   'Baked beans', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam']
```

- Add underscores to large numeric literals to improve their readability. [link]

```ruby
# bad - how many 0s are there?
num = 1000000

# good - much easier to parse for the human brain
num = 1_000_000
```

- Use RDoc and its conventions for API documentation. Don't put an empty line between the comment block and the `def`. [link]
- Limit lines to 80 characters. [link]
- Avoid trailing whitespace. [link]
- End each file with a newline. [link]
- Don't use block comments. They cannot be preceded by whitespace and are not as easy to spot as regular comments. [link]

```ruby
# bad
=begin
comment line
another comment line
=end

# good
# comment line
# another comment line
```

Syntax

- Use `::` only to reference constants(this includes classes and modules) and constructors (like `Array()` or `Nokogiri::HTML()`). Do not use `::` for regular method invocation. [link]

```ruby
# bad
SomeClass::some_method
some_object::some_method

# good
SomeClass.some_method
some_object.some_method
SomeModule::SomeClass::SOME_CONST
SomeModule::SomeClass()
```

- Use `def` with parentheses when there are arguments. Omit the parentheses when the method doesn't accept any arguments. [link]

```ruby
# bad
def some_method()
  # body omitted
end

# good
def some_method
  # body omitted
end

# bad
def some_method_with_arguments arg1, arg2
  # body omitted
end

# good
def some_method_with_arguments(arg1, arg2)
  # body omitted
end
```

- Do not use `for`, unless you know exactly why. Most of the time iterators should be used instead. `for` is implemented in terms of `each` (so you're adding a level of indirection), but with a twist - `for` doesn't introduce a new scope (unlike `each`) and variables defined in its block will be visible outside it. [link]

```ruby
arr = [1, 2, 3]

# bad
for elem in arr do
  puts elem
end

# note that elem is accessible outside of the for loop
elem # => 3

# good
arr.each { |elem| puts elem }

# elem is not accessible outside each's block
elem # => NameError: undefined local variable or method `elem'
```

- Do not use `then` for multi-line `if/unless`. [link]

```ruby
# bad
if some_condition then
  # body omitted
end

# good
if some_condition
  # body omitted
end
```

- Always put the condition on the same line as the `if`/`unless` in a multi-line conditional. [link]

```ruby
# bad
if
  some_condition
  do_something
  do_something_else
end

# good
if some_condition
  do_something
  do_something_else
end
```

- Favor the ternary operator(`?:`) over `if/then/else/end` constructs. It's more common and obviously more concise. [link]

```ruby
# bad
result = if some_condition then something else something_else end

# good
result = some_condition ? something : something_else
```

- Use one expression per branch in a ternary operator. This also means that ternary operators must not be nested. Prefer `if/else` constructs in these cases. [link]

```ruby
# bad
```

```
some_condition ? (nested_condition ? nested_something : nested_something_else) : something_else

# good
if some_condition
  nested_condition ? nested_something : nested_something_else
else
  something_else
end
```

- Do not use `if x; ...`. Use the ternary operator instead. [link]

```
# bad
result = if some_condition; something else something_else end

# good
result = some_condition ? something : something_else
```

- Leverage the fact that `if` and `case` are expressions which return a result. [link]

```
# bad
if condition
  result = x
else
  result = y
end

# good
result =
  if condition
    x
  else
    y
  end
```

- Use `when x then ...` for one-line cases. The alternative syntax `when x: ...` has been removed as of Ruby 1.9. [link]
- Do not use `when x; ...`. See the previous rule. [link]
- Use `!` instead of `not`. [link]

```
# bad - braces are required because of op precedence
x = (not something)

# good
x = !something
```

- Avoid the use of `!!`. [link]

```
# bad
x = 'test'
# obscure nil check
if !!x
  # body omitted
end

x = false
# double negation is useless on booleans
!!x # => false

# good
x = 'test'
unless x.nil?
  # body omitted
end
```

- The `and` and `or` keywords are banned. It's just not worth it. Always use `&&` and `||` instead. [link]

```
# bad
# boolean expression
if some_condition and some_other_condition
  do_something
end

# control flow
document.saved? or document.save!

# good
# boolean expression
if some_condition && some_other_condition
  do_something
end
```

```ruby
# control flow
document.saved? || document.save!
```

- Avoid multi-line `?:` (the ternary operator); use `if/unless` instead. [link]
- Favor modifier `if/unless` usage when you have a single-line body. Another good alternative is the usage of control flow `&&/||`. [link]

```ruby
# bad
if some_condition
  do_something
end

# good
do_something if some_condition

# another good option
some_condition && do_something
```

- Avoid modifier `if/unless` usage at the end of a non-trivial multi-line block. [link]

```ruby
# bad
10.times do
  # multi-line body omitted
end if some_condition

# good
if some_condition
  10.times do
    # multi-line body omitted
  end
end
```

- Favor `unless` over `if` for negative conditions (or control flow `||`). [link]

```ruby
# bad
do_something if !some_condition

# bad
do_something if not some_condition

# good
do_something unless some_condition

# another good option
some_condition || do_something
```

- Do not use `unless` with `else`. Rewrite these with the positive case first. [link]

```ruby
# bad
unless success?
  puts 'failure'
else
  puts 'success'
end

# good
if success?
  puts 'success'
else
  puts 'failure'
end
```

- Don't use parentheses around the condition of an `if/unless/while/until`. [link]

```ruby
# bad
if (x > 10)
  # body omitted
end

# good
if x > 10
  # body omitted
end
```

Note that there is an exception to this rule, namely safe assignment in condition.

- Do not use `while/until condition do` for multi-line `while/until`. [link]

```ruby
# bad
while x > 5 do
  # body omitted
```

```ruby
end

until x > 5 do
  # body omitted
end

# good
while x > 5
  # body omitted
end

until x > 5
  # body omitted
end
```

- Favor modifier `while/until` usage when you have a single-line body. [link]

```ruby
# bad
while some_condition
  do_something
end

# good
do_something while some_condition
```

- Favor `until` over `while` for negative conditions. [link]

```ruby
# bad
do_something while !some_condition

# good
do_something until some_condition
```

- Use `Kernel#loop` instead of `while/until` when you need an infinite loop. [link]

```ruby
# bad
while true
  do_something
end

until false
  do_something
end

# good
loop do
  do_something
end
```

- Use `Kernel#loop` with `break` rather than `begin/end/until` or `begin/end/while` for post-loop tests. [link]

```ruby
# bad
begin
  puts val
  val += 1
end while val < 0

# good
loop do
  puts val
  val += 1
  break unless val < 0
end
```

- Omit parentheses around parameters for methods that are part of an internal DSL (e.g. Rake, Rails, RSpec), methods that have "keyword" status in Ruby (e.g. `attr_reader`, `puts`) and attribute access methods. Use parentheses around the arguments of all other method invocations. [link]

```ruby
class Person
  attr_reader :name, :age

  # omitted
end

temperance = Person.new('Temperance', 30)
temperance.name

puts temperance.age

x = Math.sin(y)
array.delete(e)
```

```
bowling.score.should == 0
```

- Omit the outer braces around an implicit options hash. [link]

```
# bad
user.set({ name: 'John', age: 45, permissions: { read: true } })

# good
user.set(name: 'John', age: 45, permissions: { read: true })
```

- Omit both the outer braces and parentheses for methods that are part of an internal DSL. [link]

```
class Person < ActiveRecord::Base
  # bad
  validates(:name, { presence: true, length: { within: 1..10 } })

  # good
  validates :name, presence: true, length: { within: 1..10 }
end
```

- Omit parentheses for method calls with no arguments. [link]

```
# bad
Kernel.exit!()
2.even?()
fork()
'test'.upcase()

# good
Kernel.exit!
2.even?
fork
'test'.upcase
```

- Prefer `{...}` over `do...end` for single-line blocks. Avoid using `{...}` for multi-line blocks (multiline chaining is always ugly). Always use `do...end` for "control flow" and "method definitions" (e.g. in Rakefiles and certain DSLs). Avoid `do...end` when chaining. [link]

```
names = ['Bozhidar', 'Steve', 'Sarah']

# bad
names.each do |name|
  puts name
end

# good
names.each { |name| puts name }

# bad
names.select do |name|
  name.start_with?('S')
end.map { |name| name.upcase }

# good
names.select { |name| name.start_with?('S') }.map { |name| name.upcase }
```

Some will argue that multiline chaining would look OK with the use of {...}, but they should ask themselves - is this code really readable and can the blocks' contents be extracted into nifty methods?

- Consider using explicit block argument to avoid writing block literal that just passes its arguments to another block. Beware of the performance impact, though, as the block gets converted to a Proc. [link]

```
require 'tempfile'

# bad
def with_tmp_dir
  Dir.mktmpdir do |tmp_dir|
    Dir.chdir(tmp_dir) { |dir| yield dir }  # block just passes arguments
  end
end

# good
def with_tmp_dir(&block)
  Dir.mktmpdir do |tmp_dir|
    Dir.chdir(tmp_dir, &block)
  end
end

with_tmp_dir do |dir|
```

```
    puts "dir is accessible as a parameter and pwd is set: #{dir}"
  end
```

- Avoid `return` where not required for flow of control. [link]

```
  # bad
  def some_method(some_arr)
    return some_arr.size
  end

  # good
  def some_method(some_arr)
    some_arr.size
  end
```

- Avoid `self` where not required. (It is only required when calling a self write accessor.) [link]

```
  # bad
  def ready?
    if self.last_reviewed_at > self.last_updated_at
      self.worker.update(self.content, self.options)
      self.status = :in_progress
    end
    self.status == :verified
  end

  # good
  def ready?
    if last_reviewed_at > last_updated_at
      worker.update(content, options)
      self.status = :in_progress
    end
    status == :verified
  end
```

- As a corollary, avoid shadowing methods with local variables unless they are both equivalent. [link]

```
  class Foo
    attr_accessor :options

    # ok
    def initialize(options)
      self.options = options
      # both options and self.options are equivalent here
    end

    # bad
    def do_something(options = {})
      unless options[:when] == :later
        output(self.options[:message])
      end
    end

    # good
    def do_something(params = {})
      unless params[:when] == :later
        output(options[:message])
      end
    end
  end
```

- Don't use the return value of = (an assignment) in conditional expressions unless the assignment is wrapped in parentheses. This is a fairly popular idiom among Rubyists that's sometimes referred to as *safe assignment in condition*. [link]

```
  # bad (+ a warning)
  if v = array.grep(/foo/)
    do_something(v)
    ...
  end

  # good (MRI would still complain, but RuboCop won't)
  if (v = array.grep(/foo/))
    do_something(v)
    ...
  end

  # good
  v = array.grep(/foo/)
  if v
    do_something(v)
    ...
```

```
end
```

- Use shorthand self assignment operators whenever applicable. [link]

```
# bad
x = x + y
x = x * y
x = x**y
x = x / y
x = x || y
x = x && y

# good
x += y
x *= y
x **= y
x /= y
x ||= y
x &&= y
```

- Use `||=` to initialize variables only if they're not already initialized. [link]

```
# bad
name = name ? name : 'Bozhidar'

# bad
name = 'Bozhidar' unless name

# good - set name to Bozhidar, only if it's nil or false
name ||= 'Bozhidar'
```

- Don't use `||=` to initialize boolean variables. (Consider what would happen if the current value happened to be `false`.) [link]

```
# bad - would set enabled to true even if it was false
enabled ||= true

# good
enabled = true if enabled.nil?
```

- Use `&&=` to preprocess variables that may or may not exist. Using `&&=` will change the value only if it exists, removing the need to check its existence with `if`. [link]

```
# bad
if something
  something = something.downcase
end

# bad
something = something ? something.downcase : nil

# ok
something = something.downcase if something

# good
something = something && something.downcase

# better
something &&= something.downcase
```

- Avoid explicit use of the case equality operator `===`. As its name implies it is meant to be used implicitly by `case` expressions and outside of them it yields some pretty confusing code. [link]

```
# bad
Array === something
(1..100) === 7
/something/ === some_string

# good
something.is_a?(Array)
(1..100).include?(7)
some_string =~ /something/
```

- Do not use `eql?` when using `==` will do. The stricter comparison semantics provided by `eql?` are rarely needed in practice. [link]

```
# bad - eql? is the same as == for strings
"ruby".eql? some_str

# good
"ruby" == some_str
1.0.eql? x # eql? makes sense here if want to differentiate between Fixnum and Float 1
```

- Avoid using Perl-style special variables (like `$:`, `$;`, etc.). They are quite cryptic and their use in anything but one-liner scripts is discouraged. Use the human-friendly aliases provided by the `English` library. [link]

```ruby
# bad
$:.unshift File.dirname(__FILE__)

# good
require 'English'
$LOAD_PATH.unshift File.dirname(__FILE__)
```

- Do not put a space between a method name and the opening parenthesis. [link]

```ruby
# bad
f (3 + 2) + 1

# good
f(3 + 2) + 1
```

- If the first argument to a method begins with an open parenthesis, always use parentheses in the method invocation. For example, write `f((3 + 2) + 1)`. [link]
- Always run the Ruby interpreter with the `-w` option so it will warn you if you forget either of the rules above! [link]
- Use the new lambda literal syntax for single line body blocks. Use the `lambda` method for multi-line blocks. [link]

```ruby
# bad
l = lambda { |a, b| a + b }
l.call(1, 2)

# correct, but looks extremely awkward
l = ->(a, b) do
  tmp = a * 7
  tmp * b / 50
end

# good
l = ->(a, b) { a + b }
l.call(1, 2)

l = lambda do |a, b|
  tmp = a * 7
  tmp * b / 50
end
```

- Prefer `proc` over `Proc.new`. [link]

```ruby
# bad
p = Proc.new { |n| puts n }

# good
p = proc { |n| puts n }
```

- Prefer `proc.call()` over `proc[]` or `proc.()` for both lambdas and procs. [link]

```ruby
# bad - looks similar to Enumeration access
l = ->(v) { puts v }
l[1]

# also bad - uncommon syntax
l = ->(v) { puts v }
l.(1)

# good
l = ->(v) { puts v }
l.call(1)
```

- Prefix with _ unused block parameters and local variables. It's also acceptable to use just _ (although it's a bit less descriptive). This convention is recognized by the Ruby interpreter and tools like RuboCop and will suppress their unused variable warnings. [link]

```ruby
# bad
result = hash.map { |k, v| v + 1 }

def something(x)
  unused_var, used_var = something_else(x)
  # ...
end
```

```ruby
# good
result = hash.map { |_k, v| v + 1 }

def something(x)
  _unused_var, used_var = something_else(x)
  # ...
end

# good
result = hash.map { |_, v| v + 1 }

def something(x)
  _, used_var = something_else(x)
  # ...
end
```

- Use `$stdout/$stderr/$stdin` instead of `STDOUT/STDERR/STDIN`. `STDOUT/STDERR/STDIN` are constants, and while you can actually reassign (possibly to redirect some stream) constants in Ruby, you'll get an interpreter warning if you do so. [link]
- Use `warn` instead of `$stderr.puts`. Apart from being more concise and clear, `warn` allows you to suppress warnings if you need to (by setting the warn level to 0 via `-W0`). [link]
- Favor the use of `sprintf` and its alias `format` over the fairly cryptic `String#%` method. [link]

```ruby
# bad
'%d %d' % [20, 10]
# => '20 10'

# good
sprintf('%d %d', 20, 10)
# => '20 10'

# good
sprintf('%{first} %{second}', first: 20, second: 10)
# => '20 10'

format('%d %d', 20, 10)
# => '20 10'

# good
format('%{first} %{second}', first: 20, second: 10)
# => '20 10'
```

- Favor the use of `Array#join` over the fairly cryptic `Array#*` with [link] a string argument.

```ruby
# bad
%w(one two three) * ', '
# => 'one, two, three'

# good
%w(one two three).join(', ')
# => 'one, two, three'
```

- Use `[*var]` or `Array()` instead of explicit `Array` check, when dealing with a variable you want to treat as an Array, but you're not certain it's an array. [link]

```ruby
# bad
paths = [paths] unless paths.is_a? Array
paths.each { |path| do_something(path) }

# good
[*paths].each { |path| do_something(path) }

# good (and a bit more readable)
Array(paths).each { |path| do_something(path) }
```

- Use ranges or `Comparable#between?` instead of complex comparison logic when possible. [link]

```ruby
# bad
do_something if x >= 1000 && x <= 2000

# good
do_something if (1000..2000).include?(x)

# good
do_something if x.between?(1000, 2000)
```

- Favor the use of predicate methods to explicit comparisons with ==. Numeric comparisons are OK. [link]

```ruby
# bad
if x % 2 == 0
end

if x % 2 == 1
end

if x == nil
end

# good
if x.even?
end

if x.odd?
end

if x.nil?
end

if x.zero?
end

if x == 0
end
```

- Don't do explicit non-`nil` checks unless you're dealing with boolean values. [link]

```ruby
# bad
do_something if !something.nil?
do_something if something != nil

# good
do_something if something

# good - dealing with a boolean
def value_set?
  !@some_boolean.nil?
end
```

- Avoid the use of `BEGIN` blocks. [link]
- Do not use `END` blocks. Use `Kernel#at_exit` instead. [link]

```ruby
# bad
END { puts 'Goodbye!' }

# good
at_exit { puts 'Goodbye!' }
```

- Avoid the use of flip-flops. [link]
- Avoid use of nested conditionals for flow of control. [link]

Prefer a guard clause when you can assert invalid data. A guard clause is a conditional statement at the top of a function that bails out as soon as it can.

```ruby
# bad
def compute_thing(thing)
  if thing[:foo]
    update_with_bar(thing)
    if thing[:foo][:bar]
      partial_compute(thing)
    else
      re_compute(thing)
    end
  end
end

# good
def compute_thing(thing)
  return unless thing[:foo]
  update_with_bar(thing[:foo])
  return re_compute(thing) unless thing[:foo][:bar]
  partial_compute(thing)
end
```

Prefer `next` in loops instead of conditional blocks.

```ruby
# bad
[0, 1, 2, 3].each do |item|
  if item > 1
    puts item
  end
```

```ruby
end

# good
[0, 1, 2, 3].each do |item|
  next unless item > 1
  puts item
end
```

- Prefer `map` over `collect`, `find` over `detect`, `select` over `find_all`, `reduce` over `inject` and `size` over `length`. This is not a hard requirement; if the use of the alias enhances readability, it's ok to use it. The rhyming methods are inherited from Smalltalk and are not common in other programming languages. The reason the use of `select` is encouraged over `find_all` is that it goes together nicely with `reject` and its name is pretty self-explanatory. [link]

- Don't use `count` as a substitute for `size`. For `Enumerable` objects other than `Array` it will iterate the entire collection in order to determine its size. [link]

```ruby
# bad
some_hash.count

# good
some_hash.size
```

- Use `flat_map` instead of `map` + `flatten`. This does not apply for arrays with a depth greater than 2, i.e. if `users.first.songs == ['a', ['b','c']]`, then use `map + flatten` rather than `flat_map`. `flat_map` flattens the array by 1, whereas `flatten` flattens it all the way. [link]

```ruby
# bad
all_songs = users.map(&:songs).flatten.uniq

# good
all_songs = users.flat_map(&:songs).uniq
```

- Use `reverse_each` instead of `reverse.each`. `reverse_each` doesn't do a new array allocation and that's a good thing. [link]

```ruby
# bad
array.reverse.each { ... }

# good
array.reverse_each { ... }
```

Naming

> The only real difficulties in programming are cache invalidation and naming things.
> -- Phil Karlton

- Name identifiers in English. [link]

```ruby
# bad - identifier using non-ascii characters
Đ·Đ°Đ¿Đ»Đ°ÑĐ° = 1_000

# bad - identifier is a Bulgarian word, written with Latin letters (instead of Cyrillic)
zaplata = 1_000

# good
salary = 1_000
```

- Use `snake_case` for symbols, methods and variables. [link]

```ruby
# bad
:'some symbol'
:SomeSymbol
:someSymbol

someVar = 5

def someMethod
  ...
end

def SomeMethod
  ...
end

# good
:some_symbol

def some_method
  ...
```

```ruby
end
```

- Use `CamelCase` for classes and modules. (Keep acronyms like HTTP, RFC, XML uppercase.) [link]

```ruby
# bad
class Someclass
  ...
end

class Some_Class
  ...
end

class SomeXml
  ...
end

# good
class SomeClass
  ...
end

class SomeXML
  ...
end
```

- Use `snake_case` for naming files, e.g. `hello_world.rb`. [link]
- Use `snake_case` for naming directories, e.g. `lib/hello_world/hello_world.rb`. [link]
- Aim to have just a single class/module per source file. Name the file name as the class/module, but replacing CamelCase with snake_case. [link]
- Use `SCREAMING_SNAKE_CASE` for other constants. [link]

```ruby
# bad
SomeConst = 5

# good
SOME_CONST = 5
```

- The names of predicate methods (methods that return a boolean value) should end in a question mark. (i.e. `Array#empty?`). Methods that don't return a boolean, shouldn't end in a question mark. [link]
- The names of potentially *dangerous* methods (i.e. methods that modify `self` or the arguments, `exit!` (doesn't run the finalizers like `exit` does), etc.) should end with an exclamation mark if there exists a safe version of that *dangerous* method. [link]

```ruby
# bad - there is no matching 'safe' method
class Person
  def update!
  end
end

# good
class Person
  def update
  end
end

# good
class Person
  def update!
  end

  def update
  end
end
```

- Define the non-bang (safe) method in terms of the bang (dangerous) one if possible. [link]

```ruby
class Array
  def flatten_once!
    res = []

    each do |e|
      [*e].each { |f| res << f }
    end

    replace(res)
  end
```

```
  def flatten_once
    dup.flatten_once!
  end
end
```

- When using `reduce` with short blocks, name the arguments `|a, e|` (accumulator, element). [link]
- When defining binary operators, name the argument `other`(<< and `[]` are exceptions to the rule, since their semantics are different). [link]

```
def +(other)
  # body omitted
end
```

Comments

> Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.
> -- Steve McConnell

- Write self-documenting code and ignore the rest of this section. Seriously! [link]
- Write comments in English. [link]
- Use one space between the leading `#` character of the comment and the text of the comment. [link]
- Comments longer than a word are capitalized and use punctuation. Use one space after periods. [link]
- Avoid superfluous comments. [link]

```
# bad
counter += 1 # Increments counter by one.
```

- Keep existing comments up-to-date. An outdated comment is worse than no comment at all. [link]

> Good code is like a good joke - it needs no explanation.
> -- Russ Olsen

- Avoid writing comments to explain bad code. Refactor the code to make it self-explanatory. (Do or do not - there is no try. --Yoda) [link]

Comment Annotations

- Annotations should usually be written on the line immediately above the relevant code. [link]
- The annotation keyword is followed by a colon and a space, then a note describing the problem. [link]
- If multiple lines are required to describe the problem, subsequent lines should be indented two spaces after the `#`. [link]

```
def bar
  # FIXME: This has crashed occasionally since v3.2.1. It may
  #   be related to the BarBazUtil upgrade.
  baz(:quux)
end
```

- In cases where the problem is so obvious that any documentation would be redundant, annotations may be left at the end of the offending line with no note. This usage should be the exception and not the rule. [link]

```
def bar
  sleep 100 # OPTIMIZE
end
```

- Use `TODO` to note missing features or functionality that should be added at a later date. [link]
- Use `FIXME` to note broken code that needs to be fixed. [link]
- Use `OPTIMIZE` to note slow or inefficient code that may cause performance problems. [link]
- Use `HACK` to note code smells where questionable coding practices were used and should be refactored away. [link]

- Use `REVIEW` to note anything that should be looked at to confirm it is working as intended. For example: `REVIEW: Are we sure this is how the client does X currently?` [link]
- Use other custom annotation keywords if it feels appropriate, but be sure to document them in your project's `README` or similar. [link]

Classes & Modules

- Use a consistent structure in your class definitions. [link]

```ruby
class Person
  # extend and include go first
  extend SomeModule
  include AnotherModule

  # inner classes
  CustomErrorKlass = Class.new(StandardError)

  # constants are next
  SOME_CONSTANT = 20

  # afterwards we have attribute macros
  attr_reader :name

  # followed by other macros (if any)
  validates :name

  # public class methods are next in line
  def self.some_method
  end

  # followed by public instance methods
  def some_method
  end

  # protected and private methods are grouped near the end
  protected

  def some_protected_method
  end

  private

  def some_private_method
  end
end
```

- Don't nest multi line classes within classes. Try to have such nested classes each in their own file in a folder named like the containing class. [link]

```ruby
# bad

# foo.rb
class Foo
  class Bar
    # 30 methods inside
  end

  class Car
    # 20 methods inside
  end

  # 30 methods inside
end

# good

# foo.rb
class Foo
  # 30 methods inside
end

# foo/bar.rb
class Foo
  class Bar
    # 30 methods inside
  end
end

# foo/car.rb
class Foo
  class Car
    # 20 methods inside
  end
```

```
end
```

- Prefer modules to classes with only class methods. Classes should be used only when it makes sense to create instances out of them. [link]

```
# bad
class SomeClass
  def self.some_method
    # body omitted
  end

  def self.some_other_method
  end
end

# good
module SomeModule
  module_function

  def some_method
    # body omitted
  end

  def some_other_method
  end
end
```

- Favor the use of `module_function` over `extend self` when you want to turn a module's instance methods into class methods. [link]

```
# bad
module Utilities
  extend self

  def parse_something(string)
    # do stuff here
  end

  def other_utility_method(number, string)
    # do some more stuff
  end
end

# good
module Utilities
  module_function

  def parse_something(string)
    # do stuff here
  end

  def other_utility_method(number, string)
    # do some more stuff
  end
end
```

- When designing class hierarchies make sure that they conform to the Liskov Substitution Principle. [link]
- Try to make your classes as SOLID as possible. [link]
- Always supply a proper `to_s` method for classes that represent domain objects. [link]

```
class Person
  attr_reader :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end

  def to_s
    "#{@first_name} #{@last_name}"
  end
end
```

- Use the `attr` family of functions to define trivial accessors or mutators. [link]

```
# bad
class Person
  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
```

```ruby
  def first_name
    @first_name
  end

  def last_name
    @last_name
  end
end

# good
class Person
  attr_reader :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
end
```

- Avoid the use of `attr`. Use `attr_reader` and `attr_accessor` instead. [link]

```ruby
# bad - creates a single attribute accessor (deprecated in 1.9)
attr :something, true
attr :one, :two, :three # behaves as attr_reader

# good
attr_accessor :something
attr_reader :one, :two, :three
```

- Consider using `Struct.new`, which defines the trivial accessors, constructor and comparison operators for you. [link]

```ruby
# good
class Person
  attr_accessor :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
end

# better
Person = Struct.new(:first_name, :last_name) do
end
```

- Don't extend a `Struct.new` - it already is a new class. Extending it introduces a superfluous class level and may also introduce weird errors if the file is required multiple times. [link]

- Consider adding factory methods to provide additional sensible ways to create instances of a particular class. [link]

```ruby
class Person
  def self.create(options_hash)
    # body omitted
  end
end
```

- Prefer duck-typing over inheritance. [link]

```ruby
# bad
class Animal
  # abstract method
  def speak
  end
end

# extend superclass
class Duck < Animal
  def speak
    puts 'Quack! Quack'
  end
end

# extend superclass
class Dog < Animal
  def speak
    puts 'Bau! Bau!'
  end
end

# good
```

```ruby
class Duck
  def speak
    puts 'Quack! Quack'
  end
end

class Dog
  def speak
    puts 'Bau! Bau!'
  end
end
```

- Avoid the usage of class (`@@`) variables due to their "nasty" behavior in inheritance. [link]

```ruby
class Parent
  @@class_var = 'parent'

  def self.print_class_var
    puts @@class_var
  end
end

class Child < Parent
  @@class_var = 'child'
end

Parent.print_class_var # => will print "child"
```

As you can see all the classes in a class hierarchy actually share one class variable. Class instance variables should usually be preferred over class variables.

- Assign proper visibility levels to methods (`private`, `protected`) in accordance with their intended usage. Don't go off leaving everything `public` (which is the default). After all we're coding in *Ruby* now, not in *Python*. [link]

- Indent the `public`, `protected`, and `private` methods as much the method definitions they apply to. Leave one blank line above the visibility modifier and one blank line below in order to emphasize that it applies to all methods below it. [link]

```ruby
class SomeClass
  def public_method
    # ...
  end

  private

  def private_method
    # ...
  end

  def another_private_method
    # ...
  end
end
```

- Use `def self.method` to define singleton methods. This makes the code easier to refactor since the class name is not repeated. [link]

```ruby
class TestClass
  # bad
  def TestClass.some_method
    # body omitted
  end

  # good
  def self.some_other_method
    # body omitted
  end

  # Also possible and convenient when you
  # have to define many singleton methods.
  class << self
    def first_method
      # body omitted
    end

    def second_method_etc
      # body omitted
    end
  end
end
```

Exceptions

- Signal exceptions using the `fail` method. Use `raise` only when catching an exception and re-raising it (because here you're not failing, but explicitly and purposefully raising an exception). [link]

```
begin
  fail 'Oops'
rescue => error
  raise if error.message != 'Oops'
end
```

- Don't specify `RuntimeError` explicitly in the two argument version of `fail/raise`. [link]

```
# bad
fail RuntimeError, 'message'

# good - signals a RuntimeError by default
fail 'message'
```

- Prefer supplying an exception class and a message as two separate arguments to `fail/raise`, instead of an exception instance. [link]

```
# bad
fail SomeException.new('message')
# Note that there is no way to do `fail SomeException.new('message'), backtrace`.

# good
fail SomeException, 'message'
# Consistent with `fail SomeException, 'message', backtrace`.
```

- Do not return from an `ensure` block. If you explicitly return from a method inside an `ensure` block, the return will take precedence over any exception being raised, and the method will return as if no exception had been raised at all. In effect, the exception will be silently thrown away. [link]

```
def foo
  begin
    fail
  ensure
    return 'very bad idea'
  end
end
```

- Use *implicit begin blocks* where possible. [link]

```
# bad
def foo
  begin
    # main logic goes here
  rescue
    # failure handling goes here
  end
end

# good
def foo
  # main logic goes here
rescue
  # failure handling goes here
end
```

- Mitigate the proliferation of `begin` blocks by using *contingency methods* (a term coined by Avdi Grimm). [link]

```
# bad
begin
  something_that_might_fail
rescue IOError
  # handle IOError
end

begin
  something_else_that_might_fail
rescue IOError
  # handle IOError
end

# good
def with_io_error_handling
   yield
rescue IOError
  # handle IOError
end
```

```
with_io_error_handling { something_that_might_fail }

with_io_error_handling { something_else_that_might_fail }
```

- Don't suppress exceptions. [link]

```ruby
# bad
begin
  # an exception occurs here
rescue SomeError
  # the rescue clause does absolutely nothing
end

# bad
do_something rescue nil
```

- Avoid using `rescue` in its modifier form. [link]

```ruby
# bad - this catches exceptions of StandardError class and its descendant classes
read_file rescue handle_error($!)

# good - this catches only the exceptions of Errno::ENOENT class and its descendant classes
def foo
  read_file
rescue Errno::ENOENT => ex
  handle_error(ex)
end
```

- Don't use exceptions for flow of control. [link]

```ruby
# bad
begin
  n / d
rescue ZeroDivisionError
  puts 'Cannot divide by 0!'
end

# good
if d.zero?
  puts 'Cannot divide by 0!'
else
  n / d
end
```

- Avoid rescuing the `Exception` class. This will trap signals and calls to `exit`, requiring you to `kill -9` the process. [link]

```ruby
# bad
begin
  # calls to exit and kill signals will be caught (except kill -9)
  exit
rescue Exception
  puts "you didn't really want to exit, right?"
  # exception handling
end

# good
begin
  # a blind rescue rescues from StandardError, not Exception as many
  # programmers assume.
rescue => e
  # exception handling
end

# also good
begin
  # an exception occurs here

rescue StandardError => e
  # exception handling
end
```

- Put more specific exceptions higher up the rescue chain, otherwise they'll never be rescued from. [link]

```ruby
# bad
begin
  # some code
rescue Exception => e
  # some handling
rescue StandardError => e
  # some handling that will never be executed
end
```

```ruby
# good
begin
  # some code
rescue StandardError => e
  # some handling
rescue Exception => e
  # some handling
end
```

- Release external resources obtained by your program in an ensure block. [link]

```ruby
f = File.open('testfile')
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close if f
end
```

- Favor the use of exceptions for the standard library over introducing new exception classes. [link]

Collections

- Prefer literal array and hash creation notation (unless you need to pass parameters to their constructors, that is). [link]

```ruby
# bad
arr = Array.new
hash = Hash.new

# good
arr = []
hash = {}
```

- Prefer `%w` to the literal array syntax when you need an array of words (non-empty strings without spaces and special characters in them). Apply this rule only to arrays with two or more elements. [link]

```ruby
# bad
STATES = ['draft', 'open', 'closed']

# good
STATES = %w(draft open closed)
```

- Prefer `%i` to the literal array syntax when you need an array of symbols (and you don't need to maintain Ruby 1.9 compatibility). Apply this rule only to arrays with two or more elements. [link]

```ruby
# bad
STATES = [:draft, :open, :closed]

# good
STATES = %i(draft open closed)
```

- Avoid comma after the last item of an `Array` or `Hash` literal, especially when the items are not on separate lines. [link]

```ruby
# bad - easier to move/add/remove items, but still not preferred
VALUES = [
           1001,
           2020,
           3333,
         ]

# bad
VALUES = [1001, 2020, 3333, ]

# good
VALUES = [1001, 2020, 3333]
```

- Avoid the creation of huge gaps in arrays. [link]

```ruby
arr = []
arr[100] = 1 # now you have an array with lots of nils
```

- When accessing the first or last element from an array, prefer `first` or `last` over `[0]` or `[-1]`. [link]

- Use `Set` instead of `Array` when dealing with unique elements. `Set` implements a collection of unordered values with no duplicates. This is a hybrid of `Array`'s intuitive

inter-operation facilities and `Hash`'s fast lookup. [link]

- Prefer symbols instead of strings as hash keys. [link]

```
# bad
hash = { 'one' => 1, 'two' => 2, 'three' => 3 }

# good
hash = { one: 1, two: 2, three: 3 }
```

- Avoid the use of mutable objects as hash keys. [link]
- Use the Ruby 1.9 hash literal syntax when your hash keys are symbols. [link]

```
# bad
hash = { :one => 1, :two => 2, :three => 3 }

# good
hash = { one: 1, two: 2, three: 3 }
```

- Don't mix the Ruby 1.9 hash syntax with hash rockets in the same hash literal. When you've got keys that are not symbols stick to the hash rockets syntax. [link]

```
# bad
{ a: 1, 'b' => 2 }

# good
{ :a => 1, 'b' => 2 }
```

- Use `Hash#key?` instead of `Hash#has_key?` and `Hash#value?` instead of `Hash#has_value?`. As noted here by Matz, the longer forms are considered deprecated. [link]

```
# bad
hash.has_key?(:test)
hash.has_value?(value)

# good
hash.key?(:test)
hash.value?(value)
```

- Use `Hash#fetch` when dealing with hash keys that should be present. [link]

```
heroes = { batman: 'Bruce Wayne', superman: 'Clark Kent' }
# bad - if we make a mistake we might not spot it right away
heroes[:batman] # => "Bruce Wayne"
heroes[:supermann] # => nil

# good - fetch raises a KeyError making the problem obvious
heroes.fetch(:supermann)
```

- Introduce default values for hash keys via `Hash#fetch` as opposed to using custom logic. [link]

```
batman = { name: 'Bruce Wayne', is_evil: false }

# bad - if we just use || operator with falsy value we won't get the expected result
batman[:is_evil] || true # => true

# good - fetch work correctly with falsy values
batman.fetch(:is_evil, true) # => false
```

- Prefer the use of the block instead of the default value in `Hash#fetch`. [link]

```
batman = { name: 'Bruce Wayne' }

# bad - if we use the default value, we eager evaluate it
# so it can slow the program down if done multiple times
batman.fetch(:powers, get_batman_powers) # get_batman_powers is an expensive call

# good - blocks are lazy evaluated, so only triggered in case of KeyError exception
batman.fetch(:powers) { get_batman_powers }
```

- Use `Hash#values_at` when you need to retrieve several values consecutively from a hash. [link]

```
# bad
email = data['email']
nickname = data['nickname']

# good
email, username = data.values_at('email', 'nickname')
```

- Rely on the fact that as of Ruby 1.9 hashes are ordered. [link]
- Do not modify a collection while traversing it. [link]

Strings

- Prefer string interpolation and string formatting instead of string concatenation: [link]

```
# bad
email_with_name = user.name + ' <' + user.email + '>'

# good
email_with_name = "#{user.name} <#{user.email}>"

# good
email_with_name = format('%s <%s>', user.name, user.email)
```

- Consider padding string interpolation code with space. It more clearly sets the code apart from the string. [link]

```
"#{ user.last_name }, #{ user.first_name }"
```

- Adopt a consistent string literal quoting style. There are two popular styles in the Ruby community, both of which are considered good - single quotes by default (Option A) and double quotes by default (Option B). [link]
    - **(Option A)** Prefer single-quoted strings when you don't need string interpolation or special symbols such as `\t`, `\n`, `'`, etc.

    ```
    # bad
    name = "Bozhidar"

    # good
    name = 'Bozhidar'
    ```

    - **(Option B)** Prefer double-quotes unless your string literal contains `"` or escape characters you want to suppress.

    ```
    # bad
    name = 'Bozhidar'

    # good
    name = "Bozhidar"
    ```

The second style is arguably a bit more popular in the Ruby community. The string literals in this guide, however, are aligned with the first style.

- Don't use the character literal syntax `?x`. Since Ruby 1.9 it's basically redundant - `?x` would interpreted as `'x'` (a string with a single character in it). [link]

```
# bad
char = ?c

# good
char = 'c'
```

- Don't leave out `{}` around instance and global variables being interpolated into a string. [link]

```
class Person
  attr_reader :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end

  # bad - valid, but awkward
  def to_s
    "#@first_name #@last_name"
  end

  # good
  def to_s
    "#{@first_name} #{@last_name}"
  end
end

$global = 0
# bad
puts "$global = #$global"
```

```ruby
# good
puts "$global = #{$global}"
```

- Don't use `Object#to_s` on interpolated objects. It's invoked on them automatically. [link]

```ruby
# bad
message = "This is the #{result.to_s}."

# good
message = "This is the #{result}."
```

- Avoid using `String#+` when you need to construct large data chunks. Instead, use `String#<<`. Concatenation mutates the string instance in-place and is always faster than `String#+`, which creates a bunch of new string objects. [link]

```ruby
# good and also fast
html = ''
html << '<h1>Page title</h1>'

paragraphs.each do |paragraph|
  html << "<p>#{paragraph}</p>"
end
```

- Don't use `String#gsub` in scenarios in which you can use a faster more specialized alternative. [link]

```ruby
url = 'http://example.com'
str = 'lisp-case-rules'

# bad
url.gsub("http://", "https://")
str.gsub("-", "_")

# good
url.sub("http://", "https://")
str.tr("-", "_")
```

- When using heredocs for multi-line strings keep in mind the fact that they preserve leading whitespace. It's a good practice to employ some margin based on which to trim the excessive whitespace. [link]

```ruby
code = <<-END.gsub(/^\s+\|/, '')
  |def test
  |  some_method
  |  other_method
  |end
END
# => "def test\n  some_method\n  other_method\nend\n"
```

Regular Expressions

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
> -- Jamie Zawinski

- Don't use regular expressions if you just need plain text search in string: `string['text']` [link]

- For simple constructions you can use regexp directly through string index. [link]

```ruby
match = string[/regexp/]             # get content of matched regexp
first_group = string[/text(grp)/, 1] # get content of captured group
string[/text (grp)/, 1] = 'replace'  # string => 'text replace'
```

- Use non-capturing groups when you don't use captured result of parentheses. [link]

```ruby
/(first|second)/   # bad
/(?:first|second)/ # good
```

- Don't use the cryptic Perl-legacy variables denoting last regexp group matches (`$1`, `$2`, etc). Use `Regexp.last_match[n]` instead. [link]

```ruby
/(regexp)/ =~ string
...

# bad
process $1

# good
process Regexp.last_match[1]
```

- Avoid using numbered groups as it can be hard to track what they contain. Named groups can be used instead. [link]

```
# bad
/(regexp)/ =~ string
...
process Regexp.last_match[1]

# good
/(?<meaningful_var>regexp)/ =~ string
...
process meaningful_var
```

- Character classes have only a few special characters you should care about: `^`, `-`, `\`, `]`, so don't escape `.` or brackets in `[]`. [link]

- Be careful with `^` and `$` as they match start/end of line, not string endings. If you want to match the whole string use: `\A` and `\z` (not to be confused with `\Z` which is the equivalent of `/\n?\z/`). [link]

```
string = "some injection\nusername"
string[/^username$/]   # matches
string[/\Ausername\z/] # doesn't match
```

- Use `x` modifier for complex regexps. This makes them more readable and you can add some useful comments. Just be careful as spaces are ignored. [link]

```
regexp = /
  start         # some text
  \s            # white space char
  (group)       # first group
  (?:alt1|alt2) # some alternation
  end
/x
```

- For complex replacements `sub`/`gsub` can be used with block or hash. [link]

Percent Literals

- Use `%()`(it's a shorthand for `%Q`) for single-line strings which require both interpolation and embedded double-quotes. For multi-line strings, prefer heredocs. [link]

```
# bad (no interpolation needed)
%(<div class="text">Some text</div>)
# should be '<div class="text">Some text</div>'

# bad (no double-quotes)
%(This is #{quality} style)
# should be "This is #{quality} style"

# bad (multiple lines)
%(<div>\n<span class="big">#{exclamation}</span>\n</div>)
# should be a heredoc.

# good (requires interpolation, has quotes, single line)
%(<tr><td class="name">#{name}</td>)
```

- Avoid `%q` unless you have a string with both `'` and `"` in it. Regular string literals are more readable and should be preferred unless a lot of characters would have to be escaped in them. [link]

```
# bad
name = %q(Bruce Wayne)
time = %q(8 o'clock)
question = %q("What did you say?")

# good
name = 'Bruce Wayne'
time = "8 o'clock"
question = '"What did you say?"'
```

- Use `%r` only for regular expressions matching *more than* one '/' character. [link]

```
# bad
%r(\s+)

# still bad
%r(^/(.*)$)
# should be /^\/(.*)$/

# good
%r(^/blog/2011/(.*)$)
```

- Avoid the use of `%x` unless you're going to invoke a command with backquotes in it(which is rather unlikely). [link]

```
# bad
date = %x(date)

# good
date = `date`
echo = %x(echo `date`)
```

- Avoid the use of `%s`. It seems that the community has decided `:"some string"` is the preferred way to create a symbol with spaces in it. [link]
- Prefer `()` as delimiters for all `%` literals, except `%r`. Since parentheses often appear inside regular expressions in many scenarios a less common character like `{` might be a better choice for a delimiter, depending on the regexp's content. [link]

```
# bad
%w[one two three]
%q{"Test's king!", John said.}

# good
%w(one two three)
%q("Test's king!", John said.)
```

Metaprogramming

- Avoid needless metaprogramming. [link]
- Do not mess around in core classes when writing libraries. (Do not monkey-patch them.) [link]
- The block form of `class_eval` is preferable to the string-interpolated form. - when you use the string-interpolated form, always supply `__FILE__` and `__LINE__`, so that your backtraces make sense: [link]

```
class_eval 'def use_relative_model_naming?; true; end', __FILE__, __LINE__
```

- `define_method` is preferable to `class_eval{ def ... }`
    - When using `class_eval` (or other `eval`) with string interpolation, add a comment block showing its appearance if interpolated (a practice used in Rails code): [link]

```
# from activesupport/lib/active_support/core_ext/string/output_safety.rb
UNSAFE_STRING_METHODS.each do |unsafe_method|
  if 'String'.respond_to?(unsafe_method)
    class_eval <<-EOT, __FILE__, __LINE__ + 1
      def #{unsafe_method}(*args, &block)       # def capitalize(*args, &block)
        to_str.#{unsafe_method}(*args, &block)  #   to_str.capitalize(*args, &block)
      end                                       # end

      def #{unsafe_method}!(*args)              # def capitalize!(*args)
        @dirty = true                           #   @dirty = true
        super                                   #   super
      end                                       # end
    EOT
  end
end
```

- Avoid using `method_missing` for metaprogramming because backtraces become messy, the behavior is not listed in `#methods`, and misspelled method calls might silently work, e.g. `nukes.launch_state = false`. Consider using delegation, proxy, or `define_method` instead. If you must use `method_missing`: [link]
    - Be sure to also define `respond_to_missing?`
    - Only catch methods with a well-defined prefix, such as `find_by_*` -- make your code as assertive as possible.
    - Call `super` at the end of your statement
    - Delegate to assertive, non-magical methods:

```
# bad
def method_missing?(meth, *args, &block)
  if /^find_by_(?<prop>.*)/ =~ meth
    # ... lots of code to do a find_by
  else
    super
  end
end

# good
def method_missing?(meth, *args, &block)
  if /^find_by_(?<prop>.*)/ =~ meth
```

```
    find_by(prop, *args, &block)
  else
    super
  end
end

# best of all, though, would to define_method as each findable attribute is declared
```

- Prefer `public_send` over `send` so as not to circumvent `private`/`protected` visibility. [link]

Misc

- Write `ruby -w` safe code. [link]
- Avoid hashes as optional parameters. Does the method do too much? (Object initializers are exceptions for this rule). [link]
- Avoid methods longer than 10 LOC (lines of code). Ideally, most methods will be shorter than 5 LOC. Empty lines do not contribute to the relevant LOC. [link]
- Avoid parameter lists longer than three or four parameters. [link]
- If you really need "global" methods, add them to Kernel and make them private. [link]
- Use module instance variables instead of global variables. [link]

```
# bad
$foo_bar = 1

# good
module Foo
  class << self
    attr_accessor :bar
  end
end

Foo.bar = 1
```

- Avoid `alias` when `alias_method` will do. [link]
- Use `OptionParser` for parsing complex command line options and `ruby -s` for trivial command line options. [link]
- Prefer `Time.now` over `Time.new` when retrieving the current system time. [link]
- Code in a functional way, avoiding mutation when that makes sense. [link]
- Do not mutate arguments unless that is the purpose of the method. [link]
- Avoid more than three levels of block nesting. [link]
- Be consistent. In an ideal world, be consistent with these guidelines. [link]
- Use common sense. [link]

Tools

Here's some tools to help you automatically check Ruby code against this guide.

RuboCop

RuboCop is a Ruby code style checker based on this style guide. RuboCop already covers a significant portion of the Guide, supports both MRI 1.9 and MRI 2.0 and has good Emacs integration.

RubyMine

RubyMine's code inspections are partially based on this guide.

Contributing

The guide is still a work in progress - some rules are lacking examples, some rules don't have examples that illustrate them clearly enough. Improving such rules is a great (and simple way) to help the Ruby community!

In due time these issues will (hopefully) be addressed - just keep them in mind for now.

Nothing written in this guide is set in stone. It's my desire to work together with everyone interested in Ruby coding style, so that we could ultimately create a resource that will be beneficial to the entire Ruby community.

Feel free to open tickets or send pull requests with improvements. Thanks in advance for your

help!

You can also support the project (and RuboCop) with financial contributions via gittip.

How to Contribute?

It's easy, just follow the contribution guidelines.

License


This work is licensed under a Creative Commons Attribution 3.0 Unported License

Spread the Word

A community-driven style guide is of little use to a community that doesn't know about its existence. Tweet about the guide, share it with your friends and colleagues. Every comment, suggestion or opinion we get makes the guide just a little bit better. And we want to have the best possible guide, don't we?

Cheers,
Bozhidar