Abstract

> Style is what separates the good from the great.
> -- Bozhidar Batsov

One thing has always bothered me as Ruby developer - Python devs 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.

This document was originally created when I, as the Technical Lead of the company which I work for, was asked by our CTO to create some internal documents describing good style and best practices for Ruby programming. I started off by building upon this existing style guide, since I concurred with many of the points in it. 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 of 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.

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 that is 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".

The guide is still a work in progress - some rules are lacking examples, some rules don't have examples that illustrate them clearly enough. In due time these issues will be addressed - just keep them in mind for now.

You can generate a PDF or an HTML copy of this guide using *rake*

To generate an HTML version

```
$ rake generate:html
```

You should have a README.html file generated

To generate an PDF version

```
$ rake generate:pdf
```

You should have a README.pdf file generated

To use these tasks you must have installed pygments and wkhtmltopdf

pygments can be installed using Python's easy_install command

```
sudo easy_install pygments
```

wkhtmltopdf can be installed in one of two methods

1. Install by hand (recommended):

   https://github.com/jdpace/PDFKit/wiki/Installing-WKHTMLTOPDF

2. Try using the wkhtmltopdf-binary gem (mac + linux i386)

   ```
   gem install wkhtmltopdf-binary
   ```

Formatting

> 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.
- Use two-space indent, no tabs. Tabs are represented by a different number of spaces on various operating systems (and their presentation can be manually configured as well) which usually results in code that looks different than intended in some (many) people's editors.
- Use Unix-style line endings. (Linux/OSX users are covered by default, Windows users have to be extra careful.)
  - 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
    ```

- 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.

  ```
  sum = 1 + 2
  a, b = 1, 2
  1 > 2 ? true : false; puts 'Hi'
  [1, 2, 3].each { |e| puts e }
  ```

  The only exception is when using the exponent operator:

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

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

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

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

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

```ruby
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

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
```

- Use an empty line before the return value of a method (unless it only has one line), and an empty line between `def`s.

```ruby
def some_method
  do_something
  do_something_else

  result
end

def some_method
  result
end
```

- Use RDoc and its conventions for API documentation. Don't put an empty line between the comment block and the `def`.
- Use empty lines to break up a method into logical paragraphs.
- Keep lines fewer than 80 characters.
  - Emacs users might want to put this in their config (e.g. `~/.emacs.d/init.el`):
  - Vim users might want to put this in their config (e.g. `~/.vimrc`):

```
" VIM 7.3+ has support for highlighting a specified column.
if exists('+colorcolumn')
    set colorcolumn=80
else
    " Emulate
    au BufWinEnter * let w:m2=matchadd('ErrorMsg', '\%80v.\+', -1)
endif
```

  - Textmate
- Avoid trailing whitespace.

- Emacs users might want to put this in their config (ideally combine this with the previous example):
- Vim users might want to put this in their `~/.vimrc`:

```
autocmd BufWritePre * :%s/\s\+$//e
```

Or if you don't want vim to touch possibly vital space based files, use:

```
set listchars+=trail:â||
```

Feel free to use some other character if you don't like the suggested one.

- Textmate users might want to take a look at the Uber Glory bundle.

Syntax

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

```
def some_method
  # body omitted
end

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

- Never use `for`, unless you know exactly why. Most of the time iterators should be used instead.

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

# bad
for elem in arr do
  puts elem
end

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

- Never use `then` for multi-line `if/unless`.

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

# good
if some_condition
  # body omitted
end
```

- Favor the ternary operator over if/then/else/end constructs. It's more common and obviously more concise.

```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.

```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
```

- Never use `if x: ...` - it is removed in Ruby 1.9. Use the ternary operator instead.

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

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

- Never use `if x; ...`. Use the ternary operator instead.
- Use `when x then ...` for one-line cases. The alternative syntax `when x: ...` is removed in Ruby 1.9.
- Never use `when x; ...`. See the previous rule.
- Use `&&/||` for boolean expressions, `and/or` for control flow. (Rule of thumb: If you have to use outer parentheses, you are using the wrong operators.)

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

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

- Avoid multi-line `?:` (the ternary operator), use `if/unless` instead.
- Favor modifier `if/unless` usage when you have a single-line body. Another good alternative is the usage of control flow `and/or`.

```ruby
# bad
if some_condition
  do_something
end

# good
do_something if some_condition
```

```ruby
# another good option
some_condition and do_something
```

- Favor `unless` over `if` for negative conditions (or control flow `or`).

```ruby
# bad
do_something if !some_condition

# good
do_something unless some_condition

# another good option
some_condition or do_something
```

- Never use `unless` with `else`. Rewrite these with the positive case first.

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

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

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

```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)
```

- 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.

```ruby
names = ["Bozhidar", "Steve", "Sarah"]

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

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

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

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

Some will argue that multiline chaining would look OK with the use of {...}, but they should ask themselves - it this code really readable and can't the blocks contents be extracted into nifty methods.

- Avoid `return` where not required.

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

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

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

```ruby
# 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 at all.

```ruby
# bad
result = 1 - \
         2

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

- Using the return value of = (an assignment) is ok.

```ruby
if v = array.grep(/foo/) ...
```

- Use `||=` freely.

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

- Avoid using Perl-style special variables (like $0-9, $`, etc.). They are quite cryptic and their use in anything but one-liner scripts is discouraged.
- Never put a space between a method name and the opening parenthesis.

```
# 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)`.
- Always run the Ruby interpreter with the `-w` option so it will warn you if you forget either of the rules above!

Naming

- Use `snake_case` for methods and variables.
- Use `CamelCase` for classes and modules. (Keep acronyms like HTTP, RFC, XML uppercase.)
- Use `SCREAMING_SNAKE_CASE` for other constants.
- The names of predicate methods (methods that return a boolean value) should end in a question mark. (i.e. `Array#empty?`).
- The names of potentially "dangerous" methods (i.e. methods that modify `self` or the arguments, `exit!`, etc.) should end with an exclamation mark.
- The length of an identifier determines its scope. Use one-letter variables for short block/method parameters, according to this scheme:

```
a,b,c: any object
d: directory names
e: elements of an Enumerable
ex: rescued exceptions
f: files and file names
i,j: indexes
k: the key part of a hash entry
m: methods
o: any object
r: return values of short methods
s: strings
v: any value
v: the value part of a hash entry
x,y,z: numbers
```

And in general, the first letter of the class name if all objects are of that type.

- When using `inject` with short blocks, name the arguments `|a, e|` (accumulator, element).
- When defining binary operators, name the argument `other`.

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

- Prefer `map` over *collect*, `find` over *detect*, `select` over *find_all*, `size` over *length*. This is not a hard requirement; if the use of the alias enhances readability, it's ok to use it.

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!
- Comments longer than a word are capitalized and use punctuation. Use [one space] after periods.

- Avoid superfluous comments.

```ruby
# bad
counter += 1 # increments counter by one
```

- Keep existing comments up-to-date. No comment is better than an outdated comment.

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

Annotations

- Annotations should usually be written on the line immediately above the relevant code.

- The annotation keyword is followed by a colon and a space, then a note describing the problem.

- If multiple lines are required to describe the problem, subsequent lines should be indented two spaces after the `#`.

```ruby
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.

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

- Use `TODO` to note missing features or functionality that should be added at a later date.

- Use `FIXME` to note broken code that needs to be fixed.

- Use `OPTIMIZE` to note slow or inefficient code that may cause performance problems.

- Use `HACK` to note code smells where questionable coding practices were used and should be refactored away.
- 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?`
- Use other custom annotation keywords if it feels appropriate, but be sure to document them in your project's `README` or similar.

Classes

- Always supply a proper `to_s` method.

```ruby
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.
- Consider adding factory methods to provide additional sensible ways to create instances of a particular class.
- Prefer duck-typing over inheritance.
- Avoid the usage of class (@@) variables due to their "nasty" behavior in inheritance.
- 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*.
- Indent the `public`, `protected`, and `private` methods as much the method definitions they apply to. Leave one blank line above them.

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

  private
  def private_method
    # ...
  end
end
```

- Use `def self.method` to define singleton methods. This makes the methods more resistant to refactoring changes.

```ruby
class TestClass
  # bad
```

```ruby
  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

- Don't suppress exceptions.
- Don't use exceptions for flow of control.
- Avoid rescuing the `Exception` class.

Strings

- Prefer string interpolation instead of string concatenation:

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

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

- Prefer single-quoted strings when you don't need string interpolation or special symbols such as `\t`, `\n`, `'`, etc.

```ruby
# bad
name = "Bozhidar"

# good
name = 'Bozhidar'
```

- Don't use `{}` around instance variables being interpolated into a string.

```ruby
class Person
  attr_reader :first_name, :last_name

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

  # bad
```

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

  # good
  def to_s
    "#@first_name #@last_name"
  end
end
```

- 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.

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

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

Percent Literals

- Use `%w` freely.

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

- Use `%()` for single-line strings which require both interpolation and embedded double-quotes. For multi-line strings, prefer heredocs.

```
# 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>)
```

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

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

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

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

- Avoid `%q`, `%Q`, `%x`, `%s`, and `%W`.
- Prefer `()` as delimiters for all `%` literals.

Misc

- Write `ruby -w` safe code.
- Avoid hashes as optional parameters. Does the method do too much?
- 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.
- Avoid parameter lists longer than three or four parameters.
- If you really have to, add "global" methods to Kernel and make them private.
- Use class instance variables instead of global variables.

```
#bad
$foo_bar = 1

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

Foo.bar = 1
```

- Avoid `alias` when `alias_method` will do.
- Use `OptionParser` for parsing complex command line options and `ruby -s` for trivial command line options.
- Write for Ruby 1.9. Don't use legacy Ruby 1.8 constructs.
    - Use the new JavaScript literal hash syntax.
    - Use the new lambda syntax.
    - Methods like `inject` now accept method names as arguments.

```
[1, 2, 3].inject(:+)
```

- Avoid needless metaprogramming.

Design

- Code in a functional way, avoiding mutation when that makes sense.
- Do not mutate arguments unless that is the purpose of the method.
- Do not mess around in core classes when writing libraries. (Do not monkey patch them.)
- Do not program defensively.
- Keep the code simple and subjective. Each method should have a single, well-defined responsibility.
- Avoid more than three levels of block nesting.
- Don't overdesign. Overly complex solutions tend to be brittle and hard to maintain.
- Don't underdesign. A solution to a problem should be as simple as possible, but no simpler than that. Poor initial design can lead to a lot of problems in the future.
- Be consistent. In an ideal world, be consistent with these guidelines.
- Use common sense.

Contributing

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!

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?