frontier-dev-section1 / **styleguide**

Branch: master ▾ **styleguide** / ruby.en.md Find file Copy path

kami Fix ruby example 60eac9a on 24 Aug 2015

0 contributors

659 lines (514 sloc) 19.4 KB Raw Blame History

Table of Contents

Ruby coding style

Introduction

This document defines the conventions for writing Ruby code at Cookpad Inc. These regulations are formed with respect to readability and maintainability, such that developers familiar with Ruby can easily read the code. To ensure readability and consistency within the code, the guide presents a number of rules to follow. Some of these rules must be followed, while others are simply a recommendation for the developer.

Ruby version

- **[SHOULD]** Projects using Ruby 2.0 should prefer Ruby 2.0 syntax where possible: e.g. keyword arguments, symbol array literal notation(`%i`).

Indentation

- **[MUST]** Use two spaces for 1-level of indent. Do not use the horizontal tab character.

- **[MUST]** When you want to pass a block in the last method call of a long method chain, you must extract the receiver of the last method call into a local variable. Afterwards, write the method call with block as a separate statement on the following line.

```
# good
posts = Post.joins(:user).
  merge(User.paid).
  where(created_at: target_date)
posts.each do |post|
  next if stuff_ids.include?(post.user_id)
  comment_count += post.comments.size
end

# bad
posts = Post.joins(:user).
  merge(User.paid).
  where(created_at: target_date).each do |post|
    next if stuff_ids.include?(post.user_id)
    comment_count += post.comments.size
  end
```

- **[SHOULD]** When breaking an expression over multiple lines, you must increase the indentation level.

```
# good
User.active.
  some_scope(foo).
  other_scope(bar)

# bad
User.active.
some_scope(foo).
other_scope(bar)
```

Whitespace

- **[MUST]** Do not put whitespace at the end of a line.

Empty lines

- **[MUST]** Do not put empty lines at the end of a file.

Character encoding and magic comments

- **[MUST]** Use UTF-8 as scripts' character encoding for no particular reason.
- **[SHOULD]** Do not use magic comments on Ruby 2.0+ and UTF-8 encoding because UTF-8 is the default encoding after Ruby 2.0.
- **[MUST]** Use the following style for magic comments when you need.

```
# coding: utf-8
```

Line columns

- **[SHOULD]** Keep lines shorter than 80 characters.
- **[MUST]** Keep lines shorter than 128 characters.

Numbers

- **[SHOULD]** Use underscores to separate every three-digits when writing long numbers.
  - eg: `1_000_000.001_023`
- **[SHOULD]** Use underscores to separate every four-digits when writing long binary and hexadecimal numbers.
  - eg: `0xABCD_1234`

- **[SHOULD]** Do not mix uppercase and lowercase letters in hexadecimal numbers.
- **[SHOULD]** Use `r` suffix to write fractional numbers, if you are using Ruby 2.1+
  - eg: `1/2r #=> (1/2)`
- **[SHOULD]** Use `Integer#quo` method for writing fractional numbers, if you are using Ruby 2.0+
  - eg: `1.quo(2) #=> (1/2)`
- **[SHOULD]** Use `i` or `ri` suffix to write complex numbers, if you are using Ruby 2.1+
  - eg: `1 + 2i #=> (1+2i)`

Strings

- **[SHOULD]** Use `''` to write empty strings.
- **[SHOULD]** Do not use `String.new` method with no arguments, unless there is a valid reason.
- **[MUST]** Use appropriate punctuation characters to minimize the number of escape sequences in string contents.
- **[SHOULD]** Use parentheses to write strings by `%` notation. You can use any kind of parentheses. In the following cases you can use non-parentheses characters for punctuations.

```
OPEN_PARENTHESES = %!({[!
```

- **[MUST]** Do not write only `Object#to_s` in string interpolation, such as `"#{obj.to_s}"` .
- **[SHOULD]** (Ruby 1.9+) Use `\u` notation to write Unicode characters rather than writing the bytes in `\x` , e.g. `"\u{3333}"` instead of `"\xE3\x8C\xB3"` . If you have to write a script for both Ruby 1.8 and 1.9, you may write the bytes in `\x` form.
- **[SHOULD]** Do not write string literals in loops (e.g. `while` , `until` , `for`).
- **[SHOULD]** Do not use `String#+` to concatenate any string objects to string literals. Use string interpolations.
- **[MUST]** Do not use `+=` to append string to string in a destructive manner. Use `String#<<` or `String#concat` .

Regular expressions

- **[SHOULD]** Do not make unnecessary backref groups. Use `(?: ... )` .
- **[SHOULD]** Use `x` option for writing complicated regular expressions. This option allows you to use line breaks, whitespace, and comments (`(?# ... )`) in the regular expressions to improve their readability.
  - You can find practical exampled in uri/common.rb.

Arrays

- **[MUST]** If you write an array in multiple lines, insert a white space between `[` and the first item, and align the head of each item.

```
# good
[ :foo,
  :bar,
  :baz
]

# bad
[:foo,
  :bar,
  :baz
]
```

- **[MUST]** If you write an array literal just after an assignment operator such as `=` , obey the following form denoted as "good".

```
# good
array = [
  :foo,
  :bar,
  :baz,
]

# bad
array = [ :foo,
          :bar,
          :baz, ]

# bad
array = [ :foo,
          :bar,
          :baz,
        ]

# bad
array = [
  :foo,
  :bar,
  :baz, ]
```

- **[SHOULD]** In the multi-line array literal, put `,` after the last item.
- **[SHOULD]** Use general delimited input (percent) syntax `%w(...)` or `%W(...)` for word arrays.

```
# good
words = %w(foo bar baz)

# bad
words = ['foo', 'bar', 'baz']
```

- **[MUST]** Use `[]` to write empty arrays.
- **[MUST]** Do not use `Array.new` without any arguments.
- **[SHOULD]** Use `Array.new(n, obj)` to generate an array with `n` same items. Do not use `[obj] * n` to reduce object allocation.
- **[SHOULD]** Use `[*range]` form instead of `Range#to_a` to convert range literals to arrays.

```
# good
[*1..10]  #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# bad
(1..10).to_a  #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Hashes

- **[MUST]** Put whitespaces between `{` and the first key, and between the last value and `}` when writing hash literals on a single line.

```
# good
{ hoge: 1, fuga: 2 }
# bad
{hoge: 1, fuga: 2}
```

- **[MUST]** Use new hash syntax in Ruby 1.9+ (`{ foo: 42 }`) if all keys can be written in that syntax:

```
# good
{ first: 42,
  second: 'foo',
}
# bad
{ :first => 42,
  :second => 'foo',
}
```

- **[SHOULD]** Use hash rocket (`{ :foo => 42 }`) for all keys if any of keys can't be written in new Hash syntax (e.g. `:'foo.bar'` , `:'foo-bar'` , `if` , `unless`)

```
# good
{ :cookpad => 42,
  :'cookpad.com' => 'foo',
}
# bad
{ cookpad: 42,
  :'cookpad.com' => 'foo',
}
```

- **[MUST]** Use `{}` to write empty hashes.
- **[MUST]** For multiline hashes, write the first item just after `{` , increment the level of indentation after the second items, put `}` on an independent line, and align the line of `}` to the head of the line of the first item.

```
# good
{ first: 42,
  second: 'foo',
}

# bad
{
  first: 42,
  second: 'foo',
}

# bad
{ first: 42,
second: 'foo',
}
```

- **[MUST]** If you write a hash literal just after an assignment operator, such as `=` , obey the following form denoted as "good".

```
# good
hash = {
  first: 42,
  second: 'foo',
}

# bad
hash = { first: 42,
         second: 'foo',
       }

# bad
hash = { first: 42,
         second: 'foo',
}

# bad
hash = { first: 42,
  second: 'foo',
}

# bad
hash = {
first: 42,
second: 'foo',
}
```

- **[SHOULD]** In the multi line hash literal, put `,` after the last item.
- **[SHOULD]** If Symbol literals can be used as keys of hashes, use it for faster item lookup.

```
# good
{ foo: 1, bar: 2 }
```

```
# bad
{ 'foo' => 1, 'bar' => 2 }
```

- **[SHOULD]** (Ruby 1.9+) If all the keys of hash literals are Symbol literals, use the form of `{ key: value }`. Put whitespace after `:`.

Operations

- **[SHOULD]** Put whitespace around operators, except for `**`.
- **[MUST]** Do not use `and`, `or`, and `not`.
    - But, you may use `or` for the special case: `expression or raise 'message'`.
- **[MUST]** Do not nest conditional operators.
- **[MUST]** Do not write conditional operators over multiple lines.

```
# good
fizzbuzz = if n % 3 == 0
    n % 5 == 0 ? 'fizzbuzz' : 'fizz'
  else
    n % 5 == 0 ? 'buzz' : "#{n}"
  end

# bad
fizzbuzz = n % 3 == 0 ? (n % 5 == 0 ? 'fizzbuzz' : 'fizz') : (n % 5 == 0 ? 'buzz' : "#{n}")

# bad
fizzbuzz = n % 3 == 0 ?
  (n % 5 == 0 ? 'fizzbuzz' : 'fizz') :
  (n % 5 == 0 ? 'buzz' : "#{n}")
```

Assignments

- **[MUST]** Parallel assignments can only be used for assigning literal values or results of methods without arguments, and for exchanging two variables or attributes.
- **[MUST]** Put whitespace around assignment operators.
- **[MUST]** Do not write assignment expressions in conditional clauses.

Control structures

- **[SHOULD]** Use `unless condition`, instead of `if !condition`.
- **[SHOULD]** Use `until condition`, instead of `while !condition`.
- **[SHOULD]** Do not use `else` for `unless`.
- **[MUST]** Put a line break just after the condition clause of `if`, `unless`, and `case`. Do not follow a body code.
- **[MUST]** Do not use `then` and `:` for the condition clause of `if`, `unless`, and `case`.
- **[MUST]** Put a line break just after the condition clause of `while` and `until`. Do not follow a body code.
- **[MUST]** Do not use `do` and `:` for the condition clause of `while` and `until`.
- **[SHOULD]** Do not write a logical expressions combined by `||` in the condition clause of `unless` and `until`.
- **[SHOULD]** Use modifier forms, if conditions and bodies are short.
- **[SHOULD]** Extract conditions as predicator methods with appropriate names if the conditions are long or multiple line.
- **[MUST]** If you write a control structure just after an assignment operator such as `=`, obey the following form denoted as "good".

```ruby
# good
result = if condition
    body_code
  end

# good
result =
  if condition
    body_code
  end

# bad
result = if condition
  body_code
end

# bad
result = if condition
           body_code
         end
```

- **[MUST]** Do not put any meaningless expressions after return, next, or break.
  - In the actions of controllers when writing method calls that declare continuation, such as redirect_to and render, you can put them after return or next.

Method calls

- **[MUST]** Do not omit parentheses of method calls except for the following permitted cases.
- **[MUST]** You must omit parentheses for method calls without any arguments.
- **[MUST]** For DSL-like methods (as in the examples below), you may omit parentheses when calling them. However, you should not omit parentheses for the case where the first argument is enclosed in parentheses, because this case generates syntax warning.
  - Global methods such as p, print, and puts.
  - Declarative methods used in definitions of classes and modules such as private and attr_reader, including them provided by frameworks like Rails.
  - Methods for declaring continuation for actions of controllers such as redirect_to and render provided by ActionController.
  - Other, DSL-like methods.
- You may omit the parentheses of the outermost method call when nesting method calls, provided that other guidelines are not violated
- **[MUST]** Do not put whitespace between a method name and parenthesis.

```ruby
# good
p(1 + 2)

# bad
p (1 + 2)
```

- **[SHOULD]** Omit `{ }` of a hash literal, when put on the end of an argument list.

```ruby
# good
foo(1, 2, foo: :bar, baz: 42)

# bad
foo(1, 2, { foo: :bar, baz: 42 })
```

- **[MUST]** Use `do` / `end` form for blocks of method calls where the return value of the block is unused. i.e. blocks executed for side-effects

- **[MUST]** Use `{ }` form (i.e. brace blocks) for blocks of method calls where the return value of the block used. e.g. as an argument of the other method call.

```
# good
puts [1, 2, 3].map {|i|
  i * i
}

# bad
puts [1, 2, 3].map do |i|
  i * i
end

# good
[1, 2, 3].map {|n|
  n * n
}.each {|n|
  puts Math.sqrt(n)
}

# bad
[1, 2, 3].map do |n|
  n * n
end.each do |n|
  puts Math.sqrt(n)
end
```

- **[MUST]** Use brace block for a method call written in one line.
- **[MUST]** Obey the following "good" form in `do` / `end` form of blocks.

```
# good
[1, 2, 3].each do |num|
  puts num
end

# bad
[1, 2, 3].each do |num|
    puts num
  end

# bad
[1, 2, 3].each do |num|
                  puts num
                end

# bad
[1, 2, 3].each do |num| puts num end
```

- **[MUST]** Put a whitespace before `{` of brace blocks.
- **[MUST]** For a brace block written in one line, put whitespace between `{` , `}` and the inner contents.

```
# good
[1, 2, 3].each {|num| puts num }
[1, 2, 3].each { |num| puts num }

# bad
[1, 2, 3].each {|num| puts num}

# bad
[1, 2, 3].each { |num| puts num}

# good
10.times { puts 'Hello world' }

# bad
10.times {puts 'Hello world' }

# bad
10.times {puts 'Hello world'}
```

```
# bad
10.times { puts 'Hello world'}
```

- **[SHOULD]** On a method call with long parameters, write these arguments over multiple lines according to the following regulations.
  - If the arguments are long, put a line break after an open parenthesis `(`, increment the level of indent from the next line and put each argument on a new line. Finally, put a closing parenthesis `)` on a new less indented line.

    ```
    Foo.new(
      arg,
      long_argument,
      key: value,
      long_key: long_value,
      pretty_so_much_very_long_key:
        pretty_so_much_very_tooooooooooooooooooooo_long_value,
    )
    ```

  - If the arguments are not long, continue the first argument after an open parenthesis `(`, increment the level of indentation from the next line and put each argument on a new line, and put a closing parenthesis `)` just after the last argument.

    ```
    Foo.new(arg,
            long_argument,
            key: value,
            long_key: long_value)
    ```

  - For writing a DSL-like method call in multiple lines, put the first argument just after the method name, increment the level of indentation from the next line, and put each argument on a new line.

    ```
    ActionMailer::Base.delivery_method :smtp,
      host: 'localhost',
      port: 25
    ```

BEGIN and END

- **[MUST]** Do not use `BEGIN` and `END` blocks.

Module and Class definitions

- **[MUST]** Use `alias_method` instead of `alias` to define aliases of methods.
- **[MUST]** use `attr_accessor`, `attr_reader`, and `attr_writer` to define accessors instead of `attr`.
- **[MUST]** In definitions of class methods, use `self.` prefix of method name to reduce the indentation level. However, it is fine to use `class << self` when you want to define both public and private class methods.

  ```
  class Foo
    # good
    def self.foo
    end

    # bad
    def Foo.foo
    end
  end
  ```

- **[MUST]** In definitions of private or protected class methods, define these methods and change their visibilities in `class << self` / `end`.

  ```
  class Foo
    # good
    class << self
      def foo
  ```

```
      end
      private :foo
    end

    # bad
    def self.foo
    end
    class <<self
      private :foo
    end
  end
```

- **[MUST]** If you use `private`, `protected`, and `public` with method name to change the visibilities of methods after definitions of the methods, do not put empty line between the definition of the method and these visibility-change methods.

```
class Foo
  # good
  def foo
  end
  private :foo

  # bad
  def foo
  end

  private :foo
end
```

- **[MUST]** If you use `private`, `protected`, and `public` without any arguments, align the lines of these method calls to their associated method definition. Put empty lines around the visibility-change methods.

```
# good
class Foo
  def foo
  end

  private

  def bar
  end
end

# bad
class Foo
  def foo
  end

private

  def bar
  end
end

# bad
class Foo
  def foo
  end

  private

    def bar
    end
end

# bad
class Foo
  def foo
  end

  private
  def bar
```

```
    end
  end
```

- **[SHOULD]** Use Markdown format to write documentation comments for public interfaces of modules and/or classes.
- **[SHOULD]** Follow YARD style to write documentation comments.
- **[MUST]** Do not put empty lines between documentation comments and the corresponding method definitions.
- **[MUST]** Do not give different responsibilities to a class.

Method definitions

- **[MUST]** On method definition, do not omit parentheses of parameter list, except for methods without parameters.
- **[MUST]** Do not put whitespace between method name and the parameter list.
- **[SHOULD]** Do not write codes that is not understandable without any comment.
  - Prefer to divide a method into smaller methods, which do not require comments to understand.
  - You may write comments for additional information, such as describing variables or citations for equations.
- **[MUST]** Do not give different responsibilities to a method.
- **[MUST]** Do not destructively modify arguments (i.e. cause side-effects).

```
# good
def your_method(str)
  new_str = str.sub('xxx', 'yyy')
end

# bad
def your_method(str)
  str.sub!('xxx', 'yyy')
end
```

Variables

- **[MUST]** Do not introduce new global variables (`$foo`) for any reason.
- **[MUST]** Do not use class variables (`@@foo`) for any reasons. Use `class_attribute` instead.
- **[MUST]** Name variables with valid English words without shortening. If a variable name gets too long, you can shorten the variable name by removing vowels, except for the first character. Alternatively, you choose well-known abbreviations. Additionally, you can use conventional variables such as `i` and `j` for loop counters.
- **[SHOULD]** Do not use a single variable for different purposes.
- **[SHOULD]** Minimize the scope of a local variable. Methods without any local variables are preferred.

Miscellaneous

- **[MUST]** Keep the side effects of destructive methods to a minimum.