Ruby Style Guide

Introduction

> Role models are important.

– Officer Alex J. Murphy / RoboCop

Tip	You can find a beautiful version of this guide with much improved navigation at https://rubystyle.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, while 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 guidelines. We've tried to add the rationale behind the guidelines (if it's omitted we've assumed it's pretty obvious).

We didn't come up with all the guidelines out of nowhere - they are mostly based on the professional experience of the editors, feedback and suggestions from members of the Ruby community and various highly regarded Ruby programming resources, such as "Programming Ruby" and "The Ruby Programming Language".

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

You can generate a PDF copy of this guide using AsciiDoctor PDF, and an HTML copy with AsciiDoctor using the following commands:

```
# Generates README.pdf
asciidoctor-pdf -a allow-uri-read README.adoc

# Generates README.html
asciidoctor README.adoc
```

Tip	Install the `rouge` gem to get nice syntax highlighting in the generated document. `gem install rouge`

Tip	If you're into Rails or RSpec you might want to check out the complementary Ruby on Rails Style Guide and RSpec Style Guide.

Tip	RuboCop is a static code analyzer (linter) and formatter, based on this style guide.

Guiding Principles

> Programs must be written for people to read, and only incidentally for machines to execute.

– Harold Abelson
Structure and Interpretation of Computer Programs

It's common knowledge that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Ruby code. They are also meant to reflect real-world usage of Ruby instead of a random ideal. When we had to choose between a very established practice and a subjectively better alternative we've opted to recommend the established practice.[1]

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.

Ruby had existed for over 15 years by the time the guide was created, and the language's flexibility and lack of common standards have contributed to the creation of numerous styles for just about everything. Rallying people around the cause of community standards took a lot of time and energy, and we still have a lot of ground to cover.

Ruby is famously optimized for programmer happiness. We'd like to believe that this guide is going to help you optimize for maximum programmer happiness.

A Note about Consistency

> A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.

– Ralph Waldo Emerson

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one class or method is the most important.

However, know when to be inconsistent – sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

In particular: do not break backwards compatibility just to comply with this guide!

Some other good reasons to ignore a particular guideline:

- When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this style guide.
- To be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up someone else's mess (in true XP style).

- Because the code in question predates the introduction of the guideline and there is no other reason to be modifying that code.
- When the code needs to remain compatible with older versions of Ruby that don't support the feature recommended by the style guide.

Translations

Translations of the guide are available in the following languages:

- Chinese Simplified
- Chinese Traditional
- Egyptian Arabic
- French
- Japanese
- Korean
- Portuguese (pt-BR)
- Russian
- Spanish
- Vietnamese

Note	These translations are not maintained by our editor team, so their quality and level of completeness may vary. The translated versions of the guide often lag behind the upstream English version.

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)

Source Encoding

Use `UTF-8` as the source file encoding.

Tip	UTF-8 has been the default source file encoding since Ruby 2.0.

Tabs or Spaces?

Use only spaces for indentation. No hard tabs.

Indentation

Use two spaces per indentation level (aka soft tabs).

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

# good
def some_method
  do_something
end
```

Maximum Line Length

Limit lines to 80 characters.

Tip	Most editors and IDEs have configuration options to help you with that. They would typically highlight lines that exceed the length limit.

Why Bother with 80 characters in a World of Modern Widescreen Displays?
A lot of people these days feel that a maximum line length of 80 characters is just a remnant of the past and makes little sense today. After all - modern displays can easily fit 200+ characters on a single line. Still, there are some important benefits to be gained from sticking to shorter lines of code.

First, and foremost - numerous studies have shown that humans read much faster vertically and very long lines of text impede the reading process. As noted earlier, one of the guiding principles of this style guide is to optimize the code we write for human consumption.

Additionally, limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.

Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the line length limit up to 100 characters, or all the way up to 120 characters. Please, restrain the urge to go beyond 120 characters.

No Trailing Whitespace

Avoid trailing whitespace.

Tip	Most editors and IDEs have configuration options to visualize trailing whitespace and to remove it automatically on save.

Line Endings

Use Unix-style line endings.[2]

Tip	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`

Should I Terminate Files with a Newline?

End each file with a newline.

Tip	This should be done via editor configuration, not manually.

Should I Terminate Expressions with `;`?

Don't use `;` to terminate statements and expressions.

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

# good
puts 'foobar'
```

One Expression Per Line

Use one expression per line.

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

```ruby
# good
puts 'foo'
puts 'bar'

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

Operator Method Call

Avoid dot where not required for operator method calls.

```ruby
# bad
num.+ 42

# good
num + 42
```

Spaces and Operators

Use spaces around operators, after commas, colons and semicolons. Whitespace might be (mostly) irrelevant to the Ruby interpreter, but its proper use is the key to writing easily readable code.

```ruby
# bad
sum=1+2
a,b=1,2
class FooError<StandardError;end

# good
sum = 1 + 2
a, b = 1, 2
class FooError < StandardError; end
```

There are a few exceptions:

- Exponent operator:

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

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

- Slash in rational literals:

```ruby
# bad
o_scale = 1 / 48r
```

```ruby
# good
o_scale = 1/48r
```

- Safe navigation operator:

```ruby
# bad
foo &. bar
foo &.bar
foo&. bar

# good
foo&.bar
```

Safe navigation

Avoid chaining of `&.`. Replace with `.` and an explicit check. E.g. if users are guaranteed to have an address and addresses are guaranteed to have a zip code:

```ruby
# bad
user&.address&.zip

# good
user && user.address.zip
```

If such a change introduces excessive conditional logic, consider other approaches, such as delegation:

```ruby
# bad
user && user.address && user.address.zip

# good
class User
  def zip
    address&.zip
  end
end
user&.zip
```

Spaces and Braces

No spaces after `(`, `[` or before `]`, `)`. Use spaces around `{` and before `}`.

```ruby
# bad
some( arg ).other
[ 1, 2, 3 ].each{|e| puts e}

# good
```

```
some(arg).other
[1, 2, 3].each { |e| puts e }
```

`{` and `}` deserve a bit of clarification, since they are used for block and hash literals, as well as string interpolation.

For hash literals two styles are considered acceptable. 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.

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

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

With interpolated expressions, there should be no padded-spacing inside the braces.

```
# bad
"From: #{ user.first_name }, #{ user.last_name }"

# good
"From: #{user.first_name}, #{user.last_name}"
```

No Space after Bang

No space after `!`.

```
# bad
! something

# good
!something
```

No Space inside Range Literals

No space inside range literals.

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

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

Indent `when` to `case`

Indent `when` as deep as `case`.

```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
```

A Bit of History
This is the style established in both "The Ruby Programming Language" and "Programming Ruby". Historically it is derived from the fact that `case` and `switch` statements are not blocks, hence should not be indented, and the `when` and `else` keywords are labels (compiled in the C language, they are literally labels for `JMP` calls).

Indent Conditional Assignment

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

```ruby
# 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
```

```ruby
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
```

Empty Lines between Methods

Use empty lines between method definitions and also to break up methods into logical paragraphs internally.

```ruby
# bad
def some_method
  data = initialize(options)
  data.manipulate!
  data.result
end
```

```ruby
def some_other_method
  result
end

# good
def some_method
  data = initialize(options)

  data.manipulate!

  data.result
end

def some_other_method
  result
end
```

Two or More Empty Lines

Don't use several empty lines in a row.

```ruby
# bad - It has two empty lines.
some_method


some_method

# good
some_method

some_method
```

Empty Lines around Attribute Accessor

Use empty lines around attribute accessor.

```ruby
# bad
class Foo
  attr_reader :foo
  def foo
    # do something...
  end
end

# good
class Foo
  attr_reader :foo

  def foo
    # do something...
```

```
    end
  end
```

Empty Lines around Access Modifier

Use empty lines around access modifier.

```
# bad
class Foo
  def bar; end
  private
  def baz; end
end

# good
class Foo
  def bar; end

  private

  def baz; end
end
```

Empty Lines around Bodies

Don't use empty lines around method, class, module, block bodies.

```
# bad
class Foo

  def foo

    begin

      do_something do

        something

      end

    rescue

      something

    end

    true

  end
```

```
end

# good
class Foo
  def foo
    begin
      do_something do
        something
      end
    rescue
      something
    end
  end
end
```

Trailing Comma in Method Arguments

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

```
# 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)
```

Spaces around Equals

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

```
# 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).

Line Continuation in Expressions

Avoid line continuation with `\` where not required. In practice, avoid using line continuations for anything but string concatenation.

```
# bad (\ is not needed here)
result = 1 - \
          2

# bad (\ is required, but still ugly as hell)
result = 1 \
          - 2

# good
result = 1 -
          2

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

Multi-line Method Chains

Adopt a consistent multi-line method chaining style. There are two popular styles in the Ruby community, both of which are considered good - leading `.` and trailing `.`.

Leading `.`

When continuing a chained method call 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
```

Trailing `.`

When continuing a chained method call on another line, include the `.` on the first line to indicate that the expression continues.

```ruby
# 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.

Method Arguments Alignment

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

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

# 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
```

Implicit Options Hash

Important	As of Ruby 2.7 braces around an options hash are no longer optional.

Omit the outer braces around an implicit options hash.

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

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

DSL Method Calls

Omit both the outer braces and parentheses for methods that are part of an internal DSL (e.g., Rake, Rails, RSpec).

```ruby
class Person < ActiveRecord::Base
  # bad
  attr_reader(:name, :age)
  # good
  attr_reader :name, :age

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

Space in Method Calls

Do not put a space between a method name and the opening parenthesis.

```ruby
# bad
puts (x + y)

# good
puts(x + y)
```

Space in Brackets Access

Do not put a space between a receiver name and the opening brackets.

```ruby
# bad
collection [index_or_key]

# good
collection[index_or_key]
```

Multi-line Arrays Alignment

Align the elements of array literals spanning multiple lines.

```
# bad - single indent
menu_item = %w[Spam Spam Spam Spam Spam Spam Spam Spam
  Baked beans Spam Spam Spam Spam Spam]

# good
menu_item = %w[
  Spam Spam Spam Spam Spam Spam Spam Spam
  Baked beans Spam Spam Spam Spam Spam
]

# good
menu_item =
  %w[Spam Spam Spam Spam Spam Spam Spam Spam
     Baked beans Spam Spam Spam Spam Spam]
```

Naming Conventions

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

— Phil Karlton

English for Identifiers

Name identifiers in English.

```
# bad - identifier is a Bulgarian word, using non-ascii (Cyrillic) characters
заплата = 1_000

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

# good
salary = 1_000
```

Snake Case for Symbols, Methods and Variables

Use `snake_case` for symbols, methods and variables.

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

```ruby
someVar = 5

def someMethod
  # some code
end

def SomeMethod
  # some code
end

# good
:some_symbol

some_var = 5

def some_method
  # some code
end
```

Identifiers with a Numeric Suffix

Do not separate numbers from letters on symbols, methods and variables.

```ruby
# bad
:some_sym_1

some_var_1 = 1

var_10 = 10

def some_method_1
  # some code
end

# good
:some_sym1

some_var1 = 1

var10 = 10

def some_method1
  # some code
end
```

CapitalCase for Classes and Modules

Note	`CapitalCase` is also known as `UpperCamelCase`, `CapitalWords` and `PascalCase`.

Use `CapitalCase` for classes and modules. (Keep acronyms like HTTP, RFC, XML uppercase).

```ruby
# bad
class Someclass
  # some code
end

class Some_Class
  # some code
end

class SomeXml
  # some code
end

class XmlSomething
  # some code
end

# good
class SomeClass
  # some code
end

class SomeXML
  # some code
end

class XMLSomething
  # some code
end
```

Snake Case for Files

Use `snake_case` for naming files, e.g. `hello_world.rb`.

Snake Case for Directories

Use `snake_case` for naming directories, e.g. `lib/hello_world/hello_world.rb`.

One Class per File

Aim to have just a single class/module per source file. Name the file name as the class/module, but replacing `CapitalCase` with `snake_case`.

Screaming Snake Case for Constants

Use `SCREAMING_SNAKE_CASE` for other constants (those that don't refer to classes and modules).

```ruby
# bad
SomeConst = 5

# good
SOME_CONST = 5
```

Predicate Methods Suffix

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.

```ruby
# bad
def even(value)
end

# good
def even?(value)
end
```

Predicate Methods Prefix

Avoid prefixing predicate methods with the auxiliary verbs such as `is`, `does`, or `can`. These words are redundant and inconsistent with the style of boolean methods in the Ruby core library, such as `empty?` and `include?`.

```ruby
# bad
class Person
  def is_tall?
    true
  end

  def can_play_basketball?
    false
  end

  def does_like_candy?
    true
  end
end

# good
class Person
  def tall?
    true
  end

  def basketball_player?
```

```
    false
  end

  def likes_candy?
    true
  end
end
```

Dangerous Method Suffix

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.

```
# 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
```

Relationship between Safe and Dangerous Methods

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

```
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
```

Unused Variables Prefix

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 will suppress their unused variable warnings.

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

def something(x)
  unused_var, used_var = something_else(x)
  # some code
end

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

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

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

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

`other` Parameter

When defining binary operators and operator-alike methods, name the parameter `other` for operators with "symmetrical" semantics of operands. Symmetrical semantics means both sides of the operator are typically of the same or coercible types.

Operators and operator-alike methods with symmetrical semantics (the parameter should be named `other`): `+`, `-`, `*`, `/`, `%`, `**`, `==`, `>`, `<`, `|`, `&`, `^`, `eql?`, `equal?`.

Operators with non-symmetrical semantics (the parameter should not be named `other`): `<<`, `[]` (collection/item relations between operands), `===` (pattern/matchable relations).

Note that the rule should be followed only if both sides of the operator have the same semantics. Prominent exception in Ruby core is, for example, `Array#*(int)`.

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

# bad
def <<(other)
  @internal << other
end

# good
def <<(item)
  @internal << item
end

# bad
# Returns some string multiplied `other` times
def *(other)
  # body omitted
end

# good
# Returns some string multiplied `num` times
def *(num)
  # body omitted
end
```

Flow of Control

`for` Loops

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.

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

# bad
for elem in arr do
  puts elem
end
```

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

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

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

`then` in Multi-line Expression

Do not use `then` for multi-line `if` / `unless` / `when` / `in` .

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

# bad
case foo
when bar then
```

README.adoc

```ruby
# bad
case expression
in pattern then
  # body omitted
end

# good
if some_condition
  # body omitted
end

# good
case foo
when bar
  # body omitted
end

# good
case expression
in pattern
  # body omitted
end
```

Condition Placement

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

```ruby
# bad
if
  some_condition
  do_something
  do_something_else
end

# good
if some_condition
  do_something
  do_something_else
end
```

Ternary Operator vs `if`

Prefer 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
```

Nested Ternary Operators

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_e

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

Semicolon in `if`

Do not use `if x; …` . Use the ternary operator instead.

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

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

`case` vs `if-else`

Prefer `case` over `if-elsif` when compared value is the same in each clause.

```ruby
# bad
if status == :active
  perform_action
elsif status == :inactive || status == :hibernating
  check_timeout
else
  final_action
end

# good
case status
when :active
  perform_action
when :inactive, :hibernating
  check_timeout
else
  final_action
end
```

Returning Result from `if` / `case`

Leverage the fact that `if` and `case` are expressions which return a result.

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

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

One-line Cases

Use `when x then …` for one-line cases.

Note	The alternative syntax `when x: …` has been removed as of Ruby 1.9.

Semicolon in `when`

Do not use `when x; …`. See the previous rule.

Semicolon in `in`

Do not use `in pattern; …`. Use `in pattern then …` for one-line `in` pattern branches.

```ruby
# bad
case expression
in pattern; do_something
end

# good
case expression
in pattern then do_something
end
```

`!` vs `not`

Use `!` instead of `not`.

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

# good
x = !something
```

Double Negation

Avoid unnecessary uses of `!!`.

`!!` converts a value to boolean, but you don't need this explicit conversion in the condition of a control expression; using it only obscures your intention.

Consider using it only when there is a valid reason to restrict the result `true` or `false`. Examples include outputting to a particular format or API like JSON, or as the return value of a `predicate?` method. In these cases, also consider doing a nil check instead: `!something.nil?`.

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

# good
x = 'test'
if x
  # body omitted
end

# good
def named?
  !name.nil?
end

# good
def banned?
  !!banned_until&.future?
end
```

and / or

Do not use `and` and `or` in boolean context - `and` and `or` are control flow operators and should be used as such. They have very low precedence, and can be used as a short form of specifying flow sequences like "evaluate expression 1, and only if it is not successful (returned `nil`), evaluate expression 2". This is especially useful for raising errors or early return without breaking the reading flow.

```ruby
# good: and/or for control flow
x = extract_arguments or raise ArgumentError, "Not enough arguments!"
user.suspended? and return :denied

# bad
# and/or in conditions (their precedence is low, might produce unexpected result)
if got_needed_arguments and arguments_valid
  # ...body omitted
end
# in logical expression calculation
ok = got_needed_arguments and arguments_valid

# good
# &&/|| in conditions
if got_needed_arguments && arguments_valid
  # ...body omitted
end
# in logical expression calculation
```

```ruby
ok = got_needed_arguments && arguments_valid

# bad
# &&/|| for control flow (can lead to very surprising results)
x = extract_arguments || raise(ArgumentError, "Not enough arguments!")
```

Avoid several control flow operators in one expression, as that quickly becomes confusing:

```ruby
# bad
# Did author mean conditional return because `#log` could result in `nil`?
# ...or was it just to have a smart one-liner?
x = extract_arguments and log("extracted") and return

# good
# If the intention was conditional return
x = extract_arguments
if x
  return if log("extracted")
end
# If the intention was just "log, then return"
x = extract_arguments
if x
  log("extracted")
  return
end
```

Note	Whether organizing control flow with `and` and `or` is a good idea has been a controversial topic in the community for a long time. But if you do, prefer these operators over `&&` / `||`. As the different operators are meant to have different semantics that makes it easier to reason whether you're dealing with a logical expression (that will get reduced to a boolean value) or with flow of control.

Why is using `and` and `or` as logical operators a bad idea?
Simply put - because they add some cognitive overhead, as they don't behave like similarly named logical operators in other languages.

First of all, `and` and `or` operators have lower precedence than the `=` operator, whereas the `&&` and `||` operators have higher precedence than the `=` operator, based on order of operations.

```ruby
foo = true and false # results in foo being equal to true. Equivalent to (foo = true) a l
bar = false or true  # results in bar being equal to false. Equivalent to (bar = false) or tr
```

Also `&&` has higher precedence than `||`, where as `and` and `or` have the same one. Funny enough, even though `and` and `or` were inspired by Perl, they don't have different precedence in Perl.

```
true or true and false # => false (it's effectively (true or true) and false)
true || true && false # => true (it's effectively true || (true && false)
false or true and false # => false (it's effectively (false or true) and false)
false || true && false # => false (it's effectively false || (true && false))
```

Multi-line Ternary Operator

Avoid multi-line `?:` (the ternary operator); use `if`/`unless` instead.

`if` as a Modifier

Prefer modifier `if`/`unless` usage when you have a single-line body. Another good alternative is the usage of control flow `and`/`or`.

```
# bad
if some_condition
  do_something
end

# good
do_something if some_condition

# another good option
some_condition and do_something
```

Multi-line `if` Modifiers

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

```
# 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
```

Nested Modifiers

Avoid nested modifier `if` / `unless` / `while` / `until` usage. Prefer `&&` / `||` if appropriate.

```ruby
# bad
do_something if other_condition if some_condition

# good
do_something if some_condition && other_condition
```

`if` vs `unless`

Prefer `unless` over `if` for negative conditions (or control flow `||`).

```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
```

Using `else` with `unless`

Do not 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
```

Parentheses around Condition

Don't use parentheses around the condition of a control expression.

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

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

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

Multi-line `while do`

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

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

until x > 5 do
  # body omitted
end

# good
while x > 5
  # body omitted
end

until x > 5
  # body omitted
end
```

`while` as a Modifier

Prefer modifier `while/until` usage when you have a single-line body.

```ruby
# bad
while some_condition
  do_something
end

# good
do_something while some_condition
```

`while` vs `until`

Prefer `until` over `while` for negative conditions.

```
# bad
do_something while !some_condition

# good
do_something until some_condition
```

Infinite Loop

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

```
# bad
while true
  do_something
end

until false
  do_something
end

# good
loop do
  do_something
end
```

`loop` with `break`

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

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

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

Explicit `return`

Avoid `return` where not required for flow of control.

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

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

Explicit `self`

Avoid `self` where not required. (It is only required when calling a `self` write accessor, methods named after reserved words, or overloadable operators.)

```ruby
# 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
```

Shadowing Methods

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

```ruby
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
```

Safe Assignment in Condition

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*.

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

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

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

`BEGIN` Blocks

Avoid the use of `BEGIN` blocks.

`END` Blocks

Do not use `END` blocks. Use `Kernel#at_exit` instead.

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

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

Nested Conditionals

Avoid use of nested conditionals for flow of control.

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[:foo])
    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
end

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

Exceptions

`raise` vs `fail`

Prefer `raise` over `fail` for exceptions.

```
# bad
fail SomeException, 'message'

# good
raise SomeException, 'message'
```

Raising Explicit `RuntimeError`

Don't specify `RuntimeError` explicitly in the two argument version of `raise`.

```
# bad
raise RuntimeError, 'message'

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

Exception Class Messages

Prefer supplying an exception class and a message as two separate arguments to `raise`, instead of an exception instance.

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

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

`return` from `ensure`

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.

```
# bad
def foo
  raise
ensure
  return 'very bad idea'
end
```

Implicit `begin`

Use *implicit begin blocks* where possible.

```ruby
# 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
```

Contingency Methods

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

```ruby
# 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 }
```

Suppressing Exceptions

Don't suppress exceptions.

```ruby
# bad
begin
  do_something # an exception occurs here
rescue SomeError
end

# good
begin
  do_something # an exception occurs here
rescue SomeError
  handle_exception
end

# good
begin
  do_something # an exception occurs here
rescue SomeError
  # Notes on why exception handling is not performed
end

# good
do_something rescue nil
```

Using `rescue` as a Modifier

Avoid using `rescue` in its modifier form.

```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 => e
  handle_error(e)
end
```

Using Exceptions for Flow of Control

Don't use exceptions for flow of control.

```ruby
# bad
begin
  n / d
```

```ruby
rescue ZeroDivisionError
  puts 'Cannot divide by 0!'
end

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

Blind Rescues

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

```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
```

Exception Rescuing Ordering

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

```ruby
# bad
begin
  # some code
rescue StandardError => e
  # some handling
```

```
rescue IOError => e
  # some handling that will never be executed
end

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

Reading from a file

Use the convenience methods `File.read` or `File.binread` when only reading a file start to finish in a single operation.

```
## text mode
# bad (only when reading from beginning to end - modes: 'r', 'rt', 'r+', 'r+t')
File.open(filename).read
File.open(filename, &:read)
File.open(filename) { |f| f.read }
File.open(filename) do |f|
  f.read
end
File.open(filename, 'r').read
File.open(filename, 'r', &:read)
File.open(filename, 'r') { |f| f.read }
File.open(filename, 'r') do |f|
  f.read
end

# good
File.read(filename)

## binary mode
# bad (only when reading from beginning to end - modes: 'rb', 'r+b')
File.open(filename, 'rb').read
File.open(filename, 'rb', &:read)
File.open(filename, 'rb') { |f| f.read }
File.open(filename, 'rb') do |f|
  f.read
end

# good
File.binread(filename)
```

Writing to a file

Use the convenience methods `File.write` or `File.binwrite` when only opening a file to create / replace its content in a single operation.

```ruby
## text mode
# bad (only truncating modes: 'w', 'wt', 'w+', 'w+t')
File.open(filename, 'w').write(content)
File.open(filename, 'w') { |f| f.write(content) }
File.open(filename, 'w') do |f|
  f.write(content)
end

# good
File.write(filename, content)

## binary mode
# bad (only truncating modes: 'wb', 'w+b')
File.open(filename, 'wb').write(content)
File.open(filename, 'wb') { |f| f.write(content) }
File.open(filename, 'wb') do |f|
  f.write(content)
end

# good
File.binwrite(filename, content)
```

Release External Resources

Release external resources obtained by your program in an `ensure` block.

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

Auto-release External Resources

Use versions of resource obtaining methods that do automatic resource cleanup when possible.

```ruby
# bad - you need to close the file descriptor explicitly
f = File.open('testfile')
# some action on the file
f.close

# good - the file descriptor is closed automatically
```

```ruby
File.open('testfile') do |f|
  # some action on the file
end
```

Atomic File Operations

When doing file operations after confirming the existence check of a file, frequent parallel file operations may cause problems that are difficult to reproduce. Therefore, it is preferable to use atomic file operations.

```ruby
# bad - race condition with another process may result in an error in `mkdir`
unless Dir.exist?(path)
  FileUtils.mkdir(path)
end

# good - atomic and idempotent creation
FileUtils.mkdir_p(path)

# bad - race condition with another process may result in an error in `remove`
if File.exist?(path)
  FileUtils.remove(path)
end

# good - atomic and idempotent removal
FileUtils.rm_f(path)
```

Standard Exceptions

Prefer the use of exceptions from the standard library over introducing new exception classes.

Assignment & Comparison

Parallel Assignment

Avoid the use of parallel assignment for defining variables. Parallel assignment is allowed when it is the return of a method call, used with the splat operator, or when used to swap variable assignment. Parallel assignment is less readable than separate assignment.

```ruby
# bad
a, b, c, d = 'foo', 'bar', 'baz', 'foobar'

# good
a = 'foo'
b = 'bar'
c = 'baz'
```

```ruby
d = 'foobar'

# good - swapping variable assignment
# Swapping variable assignment is a special case because it will allow you to
# swap the values that are assigned to each variable.
a = 'foo'
b = 'bar'

a, b = b, a
puts a # => 'bar'
puts b # => 'foo'

# good - method return
def multi_return
  [1, 2]
end

first, second = multi_return

# good - use with splat
first, *list = [1, 2, 3, 4] # first => 1, list => [2, 3, 4]

hello_array = *'Hello' # => ["Hello"]

a = *(1..3) # => [1, 2, 3]
```

Values Swapping

Use parallel assignment when swapping 2 values.

```ruby
# bad
tmp = x
x = y
y = tmp

# good
x, y = y, x
```

Dealing with Trailing Underscore Variables in Destructuring Assignment

Avoid the use of unnecessary trailing underscore variables during parallel assignment. Named underscore variables are to be preferred over underscore variables because of the context that they provide. Trailing underscore variables are necessary when there is a splat variable defined on the left side of the assignment, and the splat variable is not an underscore.

```ruby
# bad
foo = 'one,two,three,four,five'
```

```ruby
# Unnecessary assignment that does not provide useful information
first, second, _ = foo.split(',')
first, _, _ = foo.split(',')
first, *_ = foo.split(',')

# good
foo = 'one,two,three,four,five'
# The underscores are needed to show that you want all elements
# except for the last number of underscore elements
*beginning, _ = foo.split(',')
*beginning, something, _ = foo.split(',')

a, = foo.split(',')
a, b, = foo.split(',')
# Unnecessary assignment to an unused variable, but the assignment
# provides us with useful information.
first, _second = foo.split(',')
first, _second, = foo.split(',')
first, *_ending = foo.split(',')
```

Self-assignment

Use shorthand self assignment operators whenever applicable.

```ruby
# 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
```

Conditional Variable Initialization Shorthand

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

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

# bad
name = 'Bozhidar' unless name
```

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

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

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

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

Existence Check Shorthand

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`.

```ruby
# 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
```

Identity Comparison

Prefer `equal?` over `==` when comparing `object_id`. `Object#equal?` is provided to compare objects for identity, and in contrast `Object#==` is provided for the purpose of doing value comparison.

```ruby
# bad
foo.object_id == bar.object_id

# good
foo.equal?(bar)
```

Similarly, prefer using `Hash#compare_by_identity` than using `object_id` for keys:

```
# bad
hash = {}
hash[foo.object_id] = :bar
if hash.key?(baz.object_id) # ...

# good
hash = {}.compare_by_identity
hash[foo] = :bar
if hash.key?(baz) # ...
```

Note that `Set` also has `Set#compare_by_identity` available.

Explicit Use of the Case Equality Operator

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.

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

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

Note	With direct subclasses of `BasicObject`, using `is_a?` is not an option since `BasicObject` doesn't provide that method (it's defined in `Object`). In those rare cases it's OK to use `===`.

`is_a?` vs `kind_of?`

Prefer `is_a?` over `kind_of?`. The two methods are synonyms, but `is_a?` is the more commonly used name in the wild.

```
# bad
something.kind_of?(Array)

# good
something.is_a?(Array)
```

`is_a?` vs `instance_of?`

Prefer `is_a?` over `instance_of?`.

While the two methods are similar, `is_a?` will consider the whole inheritance chain (superclasses and included modules), which is what you normally would want to do. `instance_of?`, on the other hand, only returns `true` if an object is an instance of that exact class you're checking for, not a subclass.

```
# bad
something.instance_of?(Array)

# good
something.is_a?(Array)
```

`instance_of?` vs class comparison

Use `Object#instance_of?` instead of class comparison for equality.

```
# bad
var.class == Date
var.class.equal?(Date)
var.class.eql?(Date)
var.class.name == 'Date'

# good
var.instance_of?(Date)
```

`==` vs `eql?`

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

```
# 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 Integer and Float 1
```

Blocks, Procs & Lambdas

Proc Application Shorthand

Use the Proc call shorthand when the called method is the only operation of a block.

```ruby
# bad
names.map { |name| name.upcase }

# good
names.map(&:upcase)
```

Single-line Blocks Delimiters

Prefer `{…}` over `do…end` for single-line blocks. Avoid using `{…}` for multi-line blocks (multi-line 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 = %w[Bozhidar Filipp 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(&:upcase)
```

Some will argue that multi-line 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?

Single-line `do … end` block

Use multi-line `do … end` block instead of single-line `do … end` block.

```ruby
# bad
foo do |arg| bar(arg) end

# good
foo do |arg|
  bar(arg)
end

# bad
```

```
->(arg) do bar(arg) end

# good
->(arg) { bar(arg) }
```

Explicit Block Argument

Consider using explicit block argument to avoid writing block literal that just passes its arguments to another block.

```
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
```

Trailing Comma in Block Parameters

Avoid comma after the last parameter in a block, except in cases where only a single argument is present and its removal would affect functionality (for instance, array destructuring).

```
# bad - easier to move/add/remove parameters, but still not preferred
[[1, 2, 3], [4, 5, 6]].each do |a, b, c,|
  a + b + c
end

# good
[[1, 2, 3], [4, 5, 6]].each do |a, b, c|
  a + b + c
end

# bad
[[1, 2, 3], [4, 5, 6]].each { |a, b, c,| a + b + c }
```

```ruby
# good
[[1, 2, 3], [4, 5, 6]].each { |a, b, c| a + b + c }

# good - this comma is meaningful for array destructuring
[[1, 2, 3], [4, 5, 6]].map { |a,| a }
```

Nested Method Definitions

Do not use nested method definitions, use lambda instead. Nested method definitions actually produce methods in the same scope (e.g. class) as the outer method. Furthermore, the "nested method" will be redefined every time the method containing its definition is called.

```ruby
# bad
def foo(x)
  def bar(y)
    # body omitted
  end

  bar(x)
end

# good - the same as the previous, but no bar redefinition on every foo call
def bar(y)
  # body omitted
end

def foo(x)
  bar(x)
end

# also good
def foo(x)
  bar = ->(y) { ... }
  bar.call(x)
end
```

Multi-line Lambda Definition

Use the new lambda literal syntax for single-line body blocks. Use the `lambda` method for multi-line blocks.

```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
```

Stabby Lambda Definition with Parameters

Don't omit the parameter parentheses when defining a stabby lambda with parameters.

```
# bad
l = ->x, y { something(x, y) }

# good
l = ->(x, y) { something(x, y) }
```

Stabby Lambda Definition without Parameters

Omit the parameter parentheses when defining a stabby lambda with no parameters.

```
# bad
l = ->() { something }

# good
l = -> { something }
```

`proc` vs `Proc.new`

Prefer `proc` over `Proc.new`.

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

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

Proc Call

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

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

# bad - most compact form, but might be confusing for newcomers to Ruby
l = ->(v) { puts v }
l.(1)

# good - a bit verbose, but crystal clear
l = ->(v) { puts v }
l.call(1)
```

Methods

Short Methods

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.

Top-Level Methods

Avoid top-level method definitions. Organize them in modules, classes or structs instead.

Note	It is fine to use top-level method definitions in scripts.

```ruby
# bad
def some_method; end

# good
class SomeClass
  def some_method; end
end
```

No Single-line Methods

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.

Note	Ruby 3 introduced an alternative syntax for single-line method definitions, that’s discussed in the next section of the guide.

```ruby
# 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.

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

Endless Methods

Only use Ruby 3.0's endless method definitions with a single line body. Ideally, such method definitions should be both simple (a single expression) and free of side effects.

Note	It's important to understand that this guideline doesn't contradict the previous one. We still caution against the use of single-line method definitions, but if such methods are to be used, prefer endless methods.

```ruby
# bad
def fib(x) = if x < 2
  x
else
  fib(x - 1) + fib(x - 2)
end

# good
def the_answer = 42
def get_x = @x
def square(x) = x * x

# Not (so) good: has side effect
def set_x(x) = (@x = x)
def print_foo = puts("foo")
```

Double Colons

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

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

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

Colon Method Definition

Do not use `::` to define class methods.

```
# bad
class Foo
  def self::some_method
  end
end

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

Method Definition Parentheses

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

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

# good
def some_method
  # body omitted
end
```

```ruby
# bad
def some_method_with_parameters param1, param2
  # body omitted
end

# good
def some_method_with_parameters(param1, param2)
  # body omitted
end
```

Method Call Parentheses

Use parentheses around the arguments of method calls, especially if the first argument begins with an open parenthesis `(`, as in `f((3 + 2) + 1)`.

```ruby
# bad
x = Math.sin y
# good
x = Math.sin(y)

# bad
array.delete e
# good
array.delete(e)

# bad
temperance = Person.new 'Temperance', 30
# good
temperance = Person.new('Temperance', 30)
```

Method Call with No Arguments

Always omit parentheses for method calls with no arguments.

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

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

Methods That Have "keyword" Status in Ruby

Always omit parentheses for methods that have "keyword" status in Ruby.

Note	Unfortunately, it's not exactly clear *which* methods have "keyword" status. There is agreement that declarative methods have "keyword" status. However, there's less agreement on which non-declarative methods, if any, have "keyword" status.

Non-Declarative Methods That Have "keyword" Status in Ruby

For non-declarative methods with "keyword" status (e.g., various `Kernel` instance methods), two styles are considered acceptable. By far the most popular style is to omit parentheses. Rationale: The code reads better, and method calls look more like keywords. A less-popular style, but still acceptable, is to include parentheses. Rationale: The methods have ordinary semantics, so why treat them differently, and it's easier to achieve a uniform style by not worrying about which methods have "keyword" status. Whichever one you pick, apply it consistently.

```ruby
# good (most popular)
puts temperance.age
system 'ls'
exit 1

# also good (less popular)
puts(temperance.age)
system('ls')
exit(1)
```

Using `super` with Arguments

Always use parentheses when calling `super` with arguments:

```ruby
# bad
super name, age

# good
super(name, age)
```

Important	When calling `super` without arguments, `super` and `super()` mean different things. Decide what is appropriate for your usage.

Too Many Params

Avoid parameter lists longer than three or four parameters.

Optional Arguments

Define optional arguments at the end of the list of arguments. Ruby has some unexpected results when calling methods that have optional arguments at the front of the list.

```ruby
# bad
def some_method(a = 1, b = 2, c, d)
  puts "#{a}, #{b}, #{c}, #{d}"
end

some_method('w', 'x') # => '1, 2, w, x'
some_method('w', 'x', 'y') # => 'w, 2, x, y'
some_method('w', 'x', 'y', 'z') # => 'w, x, y, z'

# good
def some_method(c, d, a = 1, b = 2)
  puts "#{a}, #{b}, #{c}, #{d}"
end

some_method('w', 'x') # => '1, 2, w, x'
some_method('w', 'x', 'y') # => 'y, 2, w, x'
some_method('w', 'x', 'y', 'z') # => 'y, z, w, x'
```

Keyword Arguments Order

Put required keyword arguments before optional keyword arguments. Otherwise, it's much harder to spot optional keyword arguments there, if they're hidden somewhere in the middle.

```ruby
# bad
def some_method(foo: false, bar:, baz: 10)
  # body omitted
end

# good
def some_method(bar:, foo: false, baz: 10)
  # body omitted
end
```

Boolean Keyword Arguments

Use keyword arguments when passing a boolean argument to a method.

```ruby
# bad
def some_method(bar = false)
  puts bar
end

# bad - common hack before keyword args were introduced
```

```ruby
def some_method(options = {})
  bar = options.fetch(:bar, false)
  puts bar
end

# good
def some_method(bar: false)
  puts bar
end

some_method              # => false
some_method(bar: true) # => true
```

Keyword Arguments vs Optional Arguments

Prefer keyword arguments over optional arguments.

```ruby
# bad
def some_method(a, b = 5, c = 1)
  # body omitted
end

# good
def some_method(a, b: 5, c: 1)
  # body omitted
end
```

Keyword Arguments vs Option Hashes

Use keyword arguments instead of option hashes.

```ruby
# bad
def some_method(options = {})
  bar = options.fetch(:bar, false)
  puts bar
end

# good
def some_method(bar: false)
  puts bar
end
```

Arguments Forwarding

Use Ruby 2.7's arguments forwarding.

```ruby
# bad
def some_method(*args, &block)
```

```
  other_method(*args, &block)
end

# bad
def some_method(*args, **kwargs, &block)
  other_method(*args, **kwargs, &block)
end

# bad
# Please note that it can cause unexpected incompatible behavior
# because `...` forwards block also.
# https://github.com/rubocop/rubocop/issues/7549
def some_method(*args)
  other_method(*args)
end

# good
def some_method(...)
  other_method(...)
end
```

Block Forwarding

Use Ruby 3.1's anonymous block forwarding.

In most cases, block argument is given name similar to `&block` or `&proc`. Their names have no information and `&` will be sufficient for syntactic meaning.

```
# bad
def some_method(&block)
  other_method(&block)
end

# good
def some_method(&)
  other_method(&)
end
```

Private Global Methods

If you really need "global" methods, add them to Kernel and make them private.

Classes & Modules

Consistent Classes

Use a consistent structure in your class definitions.

```ruby
class Person
  # extend/include/prepend go first
  extend SomeModule
  include AnotherModule
  prepend YetAnotherModule

  # inner classes
  class CustomError < StandardError
  end

  # 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

  # initialization goes between class methods and other instance methods
  def initialize
  end

  # followed by other 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
```

Mixin Grouping

Split multiple mixins into separate statements.

```ruby
# bad
class Person
  include Foo, Bar
end
```

```ruby
# good
class Person
  # multiple mixins go in separate statements
  include Foo
  include Bar
end
```

Single-line Classes

Prefer a two-line format for class definitions with no body. It is easiest to read, understand, and modify.

```ruby
# bad
FooError = Class.new(StandardError)

# okish
class FooError < StandardError; end

# ok
class FooError < StandardError
end
```

Note	Many editors/tools will fail to understand properly the usage of `Class.new`. Someone trying to locate the class definition might try a grep "class FooError". A final difference is that the name of your class is not available to the `inherited` callback of the base class with the `Class.new` form. In general it's better to stick to the basic two-line style.

File Classes

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.

```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
```

Namespace Definition

Define (and reopen) namespaced classes and modules using explicit nesting. Using the scope resolution operator can lead to surprising constant lookups due to Ruby's lexical scoping, which depends on the module nesting at the point of definition.

```
module Utilities
  class Queue
  end
end

# bad
class Utilities::Store
  Module.nesting # => [Utilities::Store]

  def initialize
    # Refers to the top level ::Queue class because Utilities isn't in the
    # current nesting chain.
    @queue = Queue.new
  end
end

# good
module Utilities
  class WaitingList
    Module.nesting # => [Utilities::WaitingList, Utilities]

    def initialize
      @queue = Queue.new # Refers to Utilities::Queue
```

```
    end
  end
end
```

Modules vs Classes

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

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

  def self.some_other_method
    # body omitted
  end
end

# good
module SomeModule
  module_function

  def some_method
    # body omitted
  end

  def some_other_method
    # body omitted
  end
end
```

module_function

Prefer the use of `module_function` over `extend self` when you want to turn a module's instance methods into class methods.

```
# 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
```

```ruby
# 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
```

Liskov

When designing class hierarchies make sure that they conform to the Liskov Substitution Principle.

SOLID design

Try to make your classes as SOLID as possible.

Define `to_s`

Always supply a proper `to_s` method for classes that represent domain objects.

```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
```

`attr` Family

Use the `attr` family of functions to define trivial accessors or mutators.

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

```
  end

  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
```

Accessor/Mutator Method Names

For accessors and mutators, avoid prefixing method names with `get_` and `set_`. It is a Ruby convention to use attribute names for accessors (readers) and `attr_name=` for mutators (writers).

```
# bad
class Person
  def get_name
    "#{@first_name} #{@last_name}"
  end

  def set_name(name)
    @first_name, @last_name = name.split(' ')
  end
end

# good
class Person
  def name
    "#{@first_name} #{@last_name}"
  end

  def name=(name)
    @first_name, @last_name = name.split(' ')
  end
end
```

attr

Avoid the use of `attr`. Use `attr_reader` and `attr_accessor` instead.

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

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

Struct.new

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

```
# 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 Struct.new

Don't extend an instance initialized by `Struct.new`. Extending it introduces a superfluous class level and may also introduce weird errors if the file is required multiple times.

```
# bad
class Person < Struct.new(:first_name, :last_name)
end

# good
Person = Struct.new(:first_name, :last_name)
```

Don't Extend Data.define

Don't extend an instance initialized by `Data.define`. Extending it introduces a superfluous class level.

```ruby
# bad
class Person < Data.define(:first_name, :last_name)
end

Person.ancestors
# => [Person, #<Class:0x0000000105abed88>, Data, Object, (...)]

# good
Person = Data.define(:first_name, :last_name)

Person.ancestors
# => [Person, Data, Object, (...)]
```

Duck Typing

Prefer [duck-typing](#) over inheritance.

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

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

No Class Vars

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

```
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.

Leverage Access Modifiers (e.g. **`private`** and **`protected`**)

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).

Access Modifiers Indentation

Indent the `public` , `protected` , and `private` methods as much as 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.

```
# good
class SomeClass
  def public_method
    # some code
  end

  private

  def private_method
    # some code
  end

  def another_private_method
```

```
    # some code
  end
end
```

Defining Class Methods

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

```
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 class methods.
  class << self
    def first_method
      # body omitted
    end

    def second_method_etc
      # body omitted
    end
  end
end
```

Alias Method Lexically

Prefer `alias` when aliasing methods in lexical class scope as the resolution of `self` in this context is also lexical, and it communicates clearly to the user that the indirection of your alias will not be altered at runtime or by any subclass unless made explicit.

```
class Westerner
  def first_name
    @names.first
  end

  alias given_name first_name
end
```

Since `alias`, like `def`, is a keyword, prefer bareword arguments over symbols or strings. In other words, do `alias foo bar`, not `alias :foo :bar`.

Also be aware of how Ruby handles aliases and inheritance: an alias references the method that was resolved at the time the alias was defined; it is not dispatched dynamically.

```ruby
class Fugitive < Westerner
  def first_name
    'Nobody'
  end
end
```

In this example, `Fugitive#given_name` would still call the original `Westerner#first_name` method, not `Fugitive#first_name`. To override the behavior of `Fugitive#given_name` as well, you'd have to redefine it in the derived class.

```ruby
class Fugitive < Westerner
  def first_name
    'Nobody'
  end

  alias given_name first_name
end
```

alias_method

Always use `alias_method` when aliasing methods of modules, classes, or singleton classes at runtime, as the lexical scope of `alias` leads to unpredictability in these cases.

```ruby
module Mononymous
  def self.included(other)
    other.class_eval { alias_method :full_name, :given_name }
  end
end

class Sting < Westerner
  include Mononymous
end
```

Class and self

When class (or module) methods call other such methods, omit the use of a leading `self` or own name followed by a `.` when calling other such methods. This is often seen in "service classes" or other similar concepts where a class is treated as though it were a function. This convention tends to reduce repetitive boilerplate in such classes.

```ruby
class TestClass
  # bad - more work when class renamed/method moved
  def self.call(param1, param2)
    TestClass.new(param1).call(param2)
  end

  # bad - more verbose than necessary
  def self.call(param1, param2)
    self.new(param1).call(param2)
  end

  # good
  def self.call(param1, param2)
    new(param1).call(param2)
  end

  # ...other methods...
end
```

Defining Constants within a Block

Do not define constants within a block, since the block's scope does not isolate or namespace the constant in any way.

Define the constant outside of the block instead, or use a variable or method if defining the constant in the outer scope would be problematic.

```ruby
# bad - FILES_TO_LINT is now defined globally
task :lint do
  FILES_TO_LINT = Dir['lib/*.rb']
  # ...
end

# good - files_to_lint is only defined inside the block
task :lint do
  files_to_lint = Dir['lib/*.rb']
  # ...
end
```

Classes: Constructors

Factory Methods

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

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

Disjunctive Assignment in Constructor

In constructors, avoid unnecessary disjunctive assignment (`||=`) of instance variables. Prefer plain assignment. In ruby, instance variables (beginning with an `@`) are nil until assigned a value, so in most cases the disjunction is unnecessary.

```
# bad
def initialize
  @x ||= 1
end

# good
def initialize
  @x = 1
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

No Comments

Write self-documenting code and ignore the rest of this section. Seriously!

Rationale Comments

If the *how* can be made self-documenting, but not the *why* (e.g. the code works around non-obvious library behavior, or implements an algorithm from an academic paper), add a comment explaining the rationale behind the code.

```ruby
# bad

x = BuggyClass.something.dup

def compute_dependency_graph
  ...30 lines of recursive graph merging...
end

# good

# BuggyClass returns an internal object, so we have to dup it to modify it.
x = BuggyClass.something.dup

# This is algorithm 6.4(a) from Worf & Yar's _Amazing Graph Algorithms_ (2243).
def compute_dependency_graph
  ...30 lines of recursive graph merging...
end
```

English Comments

Write comments in English.

Hash Space

Use one space between the leading `#` character of the comment and the text of the comment.

English Syntax

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

No Superfluous Comments

Avoid superfluous comments.

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

Comment Upkeep

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

Refactor, Don't Comment

> Good code is like a good joke: it needs no explanation.

– old programmers maxim
through 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)

Comment Annotations

Annotations Placement

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

```ruby
# bad
def bar
  baz(:quux) # FIXME: This has crashed occasionally since v3.2.1.
end

# good
def bar
  # FIXME: This has crashed occasionally since v3.2.1.
  baz(:quux)
end
```

Annotations Keyword Format

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

```ruby
# bad
def bar
  # FIXME This has crashed occasionally since v3.2.1.
  baz(:quux)
end

# good
def bar
  # FIXME: This has crashed occasionally since v3.2.1.
  baz(:quux)
end
```

Multi-line Annotations Indentation

If multiple lines are required to describe the problem, subsequent lines should be indented three spaces after the `#` (one general plus two for indentation purposes).

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

Inline Annotations

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
```

TODO

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

FIXME

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

OPTIMIZE

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

HACK

Use `HACK` to note code smells where questionable coding practices were used and should be refactored away.

REVIEW

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?`

Document Annotations

Use other custom annotation keywords if it feels appropriate, but be sure to document them in your project's `README` or similar.

Magic Comments

Magic Comments First

Place magic comments above all code and documentation in a file (except shebangs, which are discussed next).

```
# bad
# Some documentation about Person

# frozen_string_literal: true
class Person
end

# good
# frozen_string_literal: true

# Some documentation about Person
class Person
end
```

Below Shebang

Place magic comments below shebangs when they are present in a file.

```
# bad
# frozen_string_literal: true
#!/usr/bin/env ruby

App.parse(ARGV)

# good
#!/usr/bin/env ruby
# frozen_string_literal: true

App.parse(ARGV)
```

One Magic Comment per Line

Use one magic comment per line if you need multiple.

```
# bad
# -*- frozen_string_literal: true; encoding: ascii-8bit -*-
```

```
# good
# frozen_string_literal: true
# encoding: ascii-8bit
```

Separate Magic Comments from Code

Separate magic comments from code and documentation with a blank line.

```ruby
# bad
# frozen_string_literal: true
# Some documentation for Person
class Person
  # Some code
end

# good
# frozen_string_literal: true

# Some documentation for Person
class Person
  # Some code
end
```

Collections

Literal Array and Hash

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

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

# good
arr = []
arr = Array.new(10)
hash = {}
hash = Hash.new(0)
```

%w

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.

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

# good
STATES = %w[draft open closed]
```

%i

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.

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

# good
STATES = %i[draft open closed]
```

No Trailing Array Commas

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

```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]
```

No Gappy Arrays

Avoid the creation of huge gaps in arrays.

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

`first` and `last`

When accessing the first or last element from an array, prefer `first` or `last` over `[0]` or `[-1]`. `first` and `last` take less effort to understand, especially for a less experienced Ruby programmer or someone from a language with different indexing semantics.

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

# bad
arr[0]  # => 1
arr[-1] # => 3

# good
arr.first # => 1
arr.last  # => 3
```

Set vs Array

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.

Symbols as Keys

Prefer symbols instead of strings as hash keys.

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

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

No Mutable Keys

Avoid the use of mutable objects as hash keys.

Hash Literals

Use the Ruby 1.9 hash literal syntax when your hash keys are symbols.

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

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

Hash Literal Values

Use the Ruby 3.1 hash literal value syntax when your hash key and value are the same.

```
# bad
hash = { one: one, two: two, three: three }

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

Hash Literal as Last Array Item

Wrap hash literal in braces if it is a last array item.

```
# bad
[1, 2, one: 1, two: 2]

# good
[1, 2, { one: 1, two: 2 }]
```

No Mixed Hash Syntaxes

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.

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

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

Avoid Hash[] constructor

`Hash::[]` was a pre-Ruby 2.1 way of constructing hashes from arrays of key-value pairs, or from a flat list of keys and values. It has an obscure semantic and looks cryptic in code. Since Ruby 2.1, `Enumerable#to_h` can be used to construct a hash from a list of key-value pairs, and it should be preferred. Instead of `Hash[]` with a list of literal keys and values, just a hash literal should be preferred.

```
# bad
Hash[ary]
Hash[a, b, c, d]

# good
ary.to_h
{a => b, c => d}
```

Hash#key?

Use `Hash#key?` instead of `Hash#has_key?` and `Hash#value?` instead of `Hash#has_value?`.

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

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

Hash#each

Use `Hash#each_key` instead of `Hash#keys.each` and `Hash#each_value` instead of `Hash#values.each`.

```ruby
# bad
hash.keys.each { |k| p k }
hash.values.each { |v| p v }
hash.each { |k, _v| p k }
hash.each { |_k, v| p v }

# good
hash.each_key { |k| p k }
hash.each_value { |v| p v }
```

Hash#fetch

Use `Hash#fetch` when dealing with hash keys that should be present.

```ruby
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)
```

Hash#fetch defaults

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

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

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

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

Use Hash Blocks

Prefer the use of the block instead of the default value in `Hash#fetch` if the code that has to be evaluated may have side effects or be expensive.

```ruby
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, obtain_batman_powers) # obtain_batman_powers is an expensive call

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

Hash#values_at

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

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

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

Hash#transform_keys and Hash#transform_values

Prefer `transform_keys` or `transform_values` over `each_with_object` or `map` when transforming just the keys or just the values of a hash.

```ruby
# bad
{a: 1, b: 2}.each_with_object({}) { |(k, v), h| h[k] = v * v }
{a: 1, b: 2}.map { |k, v| [k.to_s, v] }.to_h

# good
{a: 1, b: 2}.transform_values { |v| v * v }
{a: 1, b: 2}.transform_keys { |k| k.to_s }
```

Ordered Hashes

Rely on the fact that as of Ruby 1.9 hashes are ordered.

No Modifying Collections

Do not modify a collection while traversing it.

Accessing Elements Directly

When accessing elements of a collection, avoid direct access via `[n]` by using an alternate form of the reader method if it is supplied. This guards you from calling `[]` on `nil`.

```
# bad
Regexp.last_match[1]

# good
Regexp.last_match(1)
```

Provide Alternate Accessor to Collections

When providing an accessor for a collection, provide an alternate form to save users from checking for `nil` before accessing an element in the collection.

```
# bad
def awesome_things
  @awesome_things
end

# good
def awesome_things(index = nil)
  if index && @awesome_things
    @awesome_things[index]
  else
    @awesome_things
  end
end
```

map / find / select / reduce / include? / size

Prefer `map` over `collect`, `find` over `detect`, `select` over `find_all`, `reduce` over `inject`, `include?` over `member?` 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.

`count` vs `size`

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.

```
# bad
some_hash.count

# good
some_hash.size
```

`flat_map`

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.

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

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

`reverse_each`

Prefer `reverse_each` to `reverse.each` because some classes that `include Enumerable` will provide an efficient implementation. Even in the worst case where a class does not provide a specialized implementation, the general implementation inherited from `Enumerable` will be at least as efficient as using `reverse.each`.

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

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

Object#yield_self vs Object#then

The method `Object#then` is preferred over `Object#yield_self`, since the name `then` states the intention, not the behavior. This makes the resulting code easier to read.

```ruby
# bad
obj.yield_self { |x| x.do_something }

# good
obj.then { |x| x.do_something }
```

Note	You can read more about the rationale behind this guideline here.

Numbers

Underscores in Numerics

Add underscores to large numeric literals to improve their readability.

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

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

Numeric Literal Prefixes

Prefer lowercase letters for numeric literal prefixes. `0o` for octal, `0x` for hexadecimal and `0b` for binary. Do not use `0d` prefix for decimal literals.

```ruby
# bad
num = 01234
num = 0O1234
num = 0X12AB
num = 0B10101
num = 0D1234
num = 0d1234

# good - easier to separate digits from the prefix
num = 0o1234
num = 0x12AB
num = 0b10101
num = 1234
```

Integer Type Checking

Use `Integer` to check the type of an integer number. Since `Fixnum` is platform-dependent, checking against it will return different results on 32-bit and 64-bit machines.

```
timestamp = Time.now.to_i

# bad
timestamp.is_a?(Fixnum)
timestamp.is_a?(Bignum)

# good
timestamp.is_a?(Integer)
```

Random Numbers

Prefer to use ranges when generating random numbers instead of integers with offsets, since it clearly states your intentions. Imagine simulating a roll of a dice:

```
# bad
rand(6) + 1

# good
rand(1..6)
```

Float Division

When performing float-division on two integers, either use `fdiv` or convert one-side integer to float.

```
# bad
a.to_f / b.to_f

# good
a.to_f / b
a / b.to_f
a.fdiv(b)
```

Float Comparison

Avoid (in)equality comparisons of floats as they are unreliable.

Floating point values are inherently inaccurate, and comparing them for exact equality is almost never the desired semantics. Comparison via the `==/!=` operators checks floating-point value representation to be exactly the same, which is very unlikely if you perform any arithmetic operations involving precision loss.

```ruby
# bad
x == 0.1
x != 0.1

# good - using BigDecimal
x.to_d == 0.1.to_d

# good - not an actual float comparison
x == Float::INFINITY

# good
(x - 0.1).abs < Float::EPSILON

# good
tolerance = 0.0001
(x - 0.1).abs < tolerance

# Or some other epsilon based type of comparison:
# https://www.embeddeduse.com/2019/08/26/qt-compare-two-floats/
```

Exponential Notation

When using exponential notation for numbers, prefer using the normalized scientific notation, which uses a mantissa between 1 (inclusive) and 10 (exclusive). Omit the exponent altogether if it is zero.

The goal is to avoid confusion between powers of ten and exponential notation, as one quickly reading `10e7` could think it's 10 to the power of 7 (one then 7 zeroes) when it's actually 10 to the power of 8 (one then 8 zeroes). If you want 10 to the power of 7, you should do `1e7`.

power notation	exponential notation	output
10 ** 7	1e7	10000000
10 ** 6	1e6	1000000
10 ** 7	10e6	10000000

One could favor the alternative engineering notation, in which the exponent must always be a multiple of 3 for easy conversion to the thousand / million / … system.

```
# bad
10e6
0.3e4
11.7e5
3.14e0

# good
1e7
3e3
1.17e6
3.14
```

Alternative : engineering notation:

```
# bad
3.2e7
0.1e5
12e4

# good
1e6
17e6
0.98e9
```

Strings

String Interpolation

Prefer string interpolation and string formatting to string concatenation:

```
# 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)
```

Consistent String Literals

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 and double quotes by default.

Note	The string literals in this guide are using single quotes by default.

Single Quote

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

```
# bad
name = "Bozhidar"

name = 'De\'Andre'

# good
name = 'Bozhidar'

name = "De'Andre"
```

Double Quote

Prefer double-quotes unless your string literal contains " or escape characters you want to suppress.

```
# bad
name = 'Bozhidar'

sarcasm = "I \"like\" it."

# good
name = "Bozhidar"

sarcasm = 'I "like" it.'
```

No Character Literals

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

```
# bad
char = ?c

# good
char = 'c'
```

Curlies Interpolate

Don't leave out `{}` around instance and global 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 - 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"

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

No `to_s`

Don't use `Object#to_s` on interpolated objects. It's called on them automatically.

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

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

String Concatenation

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.

```ruby
# bad
html = ''
html += '<h1>Page title</h1>'

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

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

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

Don't Abuse `gsub`

Don't use `String#gsub` in scenarios in which you can use a faster and more specialized alternative.

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

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

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

`String#chars`

Prefer the use of `String#chars` over `String#split` with empty string or regexp literal argument.

Note	These cases have the same behavior since Ruby 2.0.

```ruby
# bad
string.split(//)
string.split('')

# good
string.chars
```

`sprintf`

Prefer the use of `sprintf` and its alias `format` over the fairly cryptic `String#%` method.

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

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

# good
sprintf('%<first>d %<second>d', first: 20, second: 10)
# => '20 10'

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

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

Named Format Tokens

When using named format string tokens, favor `%<name>s` over `%{name}` because it encodes information about the type of the value.

```ruby
# bad
format('Hello, %{name}', name: 'John')

# good
format('Hello, %<name>s', name: 'John')
```

Long Strings

Break long strings into multiple lines but don't concatenate them with `+`. If you want to add newlines, use heredoc. Otherwise use `\`:

```ruby
# bad
"Lorem Ipsum is simply dummy text of the printing and typesetting industry. " +
"Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, " +
"when an unknown printer took a galley of type and scrambled it to make a type specimen book.'

# good
<<~LOREM
  Lorem Ipsum is simply dummy text of the printing and typesetting industry.
  Lorem Ipsum has been the industry's standard dummy text ever since the 1500s,
  when an unknown printer took a galley of type and scrambled it to make a type specimen book.
LOREM

# good
"Lorem Ipsum is simply dummy text of the printing and typesetting industry. "\
"Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, "\
"when an unknown printer took a galley of type and scrambled it to make a type specimen book.'
```

Heredocs

Squiggly Heredocs

Use Ruby 2.3’s squiggly heredocs for nicely indented multi-line strings.

```
# bad - using Powerpack String#strip_margin
code = <<-RUBY.strip_margin('|')
  |def test
  |  some_method
  |  other_method
  |end
RUBY

# also bad
code = <<-RUBY
def test
  some_method
  other_method
end
RUBY

# good
code = <<~RUBY
  def test
    some_method
    other_method
  end
RUBY
```

Heredoc Delimiters

Use descriptive delimiters for heredocs. Delimiters add valuable information about the heredoc content, and as an added bonus some editors can highlight code within heredocs if the correct delimiter is used.

```
# bad
code = <<~END
  def foo
    bar
  end
END

# good
code = <<~RUBY
  def foo
    bar
  end
```

```
RUBY

# good
code = <<~SUMMARY
  An imposing black structure provides a connection between the past and
  the future in this enigmatic adaptation of a short story by revered
  sci-fi author Arthur C. Clarke.
SUMMARY
```

Heredoc Method Calls

Place method calls with heredoc receivers on the first line of the heredoc definition. The bad form has significant potential for error if a new line is added or removed.

```
# bad
query = <<~SQL
  select foo from bar
SQL
.strip_indent

# good
query = <<~SQL.strip_indent
  select foo from bar
SQL
```

Heredoc Argument Closing Parentheses

Place the closing parenthesis for method calls with heredoc arguments on the first line of the heredoc definition. The bad form has potential for error if the new line before the closing parenthesis is removed.

```
# bad
foo(<<~SQL
  select foo from bar
SQL
)

# good
foo(<<~SQL)
  select foo from bar
SQL
```

Date & Time

Time.now

Prefer `Time.now` over `Time.new` when retrieving the current system time.

No `DateTime`

Don't use `DateTime` unless you need to account for historical calendar reform - and if you do, explicitly specify the `start` argument to clearly state your intentions.

```
# bad - uses DateTime for current time
DateTime.now

# good - uses Time for current time
Time.now

# bad - uses DateTime for modern date
DateTime.iso8601('2016-06-29')

# good - uses Date for modern date
Date.iso8601('2016-06-29')

# good - uses DateTime with start argument for historical date
DateTime.iso8601('1751-04-23', Date::ENGLAND)
```

Regular Expressions

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

– Jamie Zawinski

Plain Text Search

Don't use regular expressions if you just need plain text search in string.

```
foo = 'I am an example string'

# bad - using a regular expression is an overkill here
foo =~ /example/

# good
foo['example']
```

Using Regular Expressions as String Indexes

For simple constructions you can use regexp directly through string index.

```
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'
```

Prefer Non-capturing Groups

Use non-capturing groups when you don't use the captured result.

```
# bad
/(first|second)/

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

Do not mix named and numbered captures

Do not mix named captures and numbered captures in a Regexp literal. Because numbered capture is ignored if they're mixed.

```
# bad - There is no way to access `(BAR)` capturing.
m = /(?<foo>FOO)(BAR)/.match('FOOBAR')
p m[:foo] # => "FOO"
p m[1]    # => "FOO"
p m[2]    # => nil   - not "BAR"

# good - Both captures are accessible with names.
m = /(?<foo>FOO)(?<bar>BAR)/.match('FOOBAR')
p m[:foo] # => "FOO"
p m[:bar] # => "BAR"

# good - `(?:BAR)` is non-capturing grouping.
m = /(?<foo>FOO)(?:BAR)/.match('FOOBAR')
p m[:foo] # => "FOO"

# good - Both captures are accessible with numbers.
m = /(FOO)(BAR)/.match('FOOBAR')
p m[1] # => "FOO"
p m[2] # => "BAR"
```

Refer named regexp captures by name

Prefer using names to refer named regexp captures instead of numbers.

```
# bad
m = /(?<foo>FOO)(?<bar>BAR)/.match('FOOBAR')
p m[1] # => "FOO"
p m[2] # => "BAR"

# good
m = /(?<foo>FOO)(?<bar>BAR)/.match('FOOBAR')
```

```
p m[:foo] # => "FOO"
p m[:bar] # => "BAR"
```

Avoid Perl-style Last Regular Expression Group Matchers

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

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

# bad
process $1

# good
process Regexp.last_match(1)
```

Avoid Numbered Groups

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

```
# bad
/(regexp)/ =~ string
# some code
process Regexp.last_match(1)

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

Limit Escapes

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

Caret and Dollar Regexp

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/`).

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

Multi-line Regular Expressions

Use `x` (free-spacing) modifier for multi-line regexps.

Note	That's known as free-spacing mode. In this mode leading and trailing whitespace is ignored.

```
# bad
regex = /start\
\s\
(group)\
(?:alt1|alt2)\
end/

# good
regexp = /
  start
  \s
  (group)
  (?:alt1|alt2)
  end
/x
```

Comment Complex Regular Expressions

Use `x` modifier for complex regexps. This makes them more readable and you can add some useful comments.

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

Use `gsub` with a Block or a Hash for Complex Replacements

For complex replacements `sub` / `gsub` can be used with a block or a hash.

```
words = 'foo bar'
words.sub(/f/, 'f' => 'F') # => 'Foo bar'
words.gsub(/\w+/) { |word| word.capitalize } # => 'Foo Bar'
```

Percent Literals

%q shorthand

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.

```
# 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>)
```

%q

Avoid %() or the equivalent %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.

```
# 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?"'
quote = %q(<p class='quote'>"What did you say?"</p>)
```

%r

Use %r only for regular expressions matching *at least* one / character.

```
# bad
%r{\s+}
```

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

%x

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

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

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

%s

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.

Percent Literal Braces

Use the braces that are the most appropriate for the various kinds of percent literals.

- `()` for string literals (`%q`, `%Q`).
- `[]` for array literals (`%w`, `%i`, `%W`, `%I`) as it is aligned with the standard array literals.
- `{}` for regexp literals (`%r`) since parentheses often appear inside regular expressions. That's why a less common character with `{` is usually the best delimiter for `%r` literals.
- `()` for all other literals (e.g. `%s`, `%x`)

```
# bad
%q{"Test's king!", John said.}

# good
%q("Test's king!", John said.)

# bad
%w(one two three)
%i(one two three)

# good
```

```
%w[one two three]
%i[one two three]

# bad
%r((\w+)-(\d+))
%r{\w{1,2}\d{2,5}}

# good
%r{(\w+)-(\d+)}
%r|\w{1,2}\d{2,5}|
```

Metaprogramming

No Needless Metaprogramming

Avoid needless metaprogramming.

No Monkey Patching

Do not mess around in core classes when writing libraries (do not monkey-patch them).

Block `class_eval`

The block form of `class_eval` is preferable to the string-interpolated form.

Supply Location

When you use the string-interpolated form, always supply `__FILE__` and `__LINE__`, so that your backtraces make sense:

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

`define_method`

`define_method` is preferable to `class_eval { def … }`

`eval` Comment Docs

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):

```
# 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}(*params, &block)       # def capitalize(*params, &block)
```

```
        to_str.#{unsafe_method}(*params, &block)  #   to_str.capitalize(*params, &block)
      end                                         # end

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

No `method_missing`

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.luanch_state = false`. Consider using delegation, proxy, or `define_method` instead. If you must use `method_missing`:

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

# good
def method_missing(meth, *params, &block)
  if /^find_by_(?<prop>.*)/ =~ meth
    find_by(prop, *params, &block)
  else
    super
  end
end

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

Prefer `public_send`

Prefer `public_send` over `send` so as not to circumvent `private` / `protected` visibility.

```ruby
# We have an ActiveModel Organization that includes concern Activatable
module Activatable
  extend ActiveSupport::Concern

  included do
    before_create :create_token
  end

  private

  def reset_token
    # some code
  end

  def create_token
    # some code
  end

  def activate!
    # some code
  end
end

class Organization < ActiveRecord::Base
  include Activatable
end

linux_organization = Organization.find(...)

# bad - violates privacy
linux_organization.send(:reset_token)
# good - should throw an exception
linux_organization.public_send(:reset_token)
```

Prefer `__send__`

Prefer `__send__` over `send`, as `send` may overlap with existing methods.

```ruby
require 'socket'

u1 = UDPSocket.new
u1.bind('127.0.0.1', 4913)
u2 = UDPSocket.new
u2.connect('127.0.0.1', 4913)

# bad - Won't send a message to the receiver object. Instead it will send a message via UDP sc
u2.send :sleep, 0
```

```
# good - Will actually send a message to the receiver object.
u2.__send__ ...
```

API Documentation

YARD

Use YARD and its conventions for API documentation.

RD (Block) Comments

Don't use block comments. They cannot be preceded by whitespace and are not as easy to spot as regular comments.

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

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

From Perl's POD to RD
This is not really a block comment syntax, but more of an attempt to emulate Perl's POD documentation system.

There's an rdtool for Ruby that's pretty similar to POD. Basically `rdtool` scans a file for `=begin` and `=end` pairs, and extracts the text between them all. This text is assumed to be documentation in RD format. You can read more about it here.

RD predated the rise of RDoc and YARD and was effectively obsoleted by them.[3]

Gemfile and Gemspec

No **RUBY_VERSION** in the gemspec

The gemspec should not contain `RUBY_VERSION` as a condition to switch dependencies. `RUBY_VERSION` is determined by `rake release`, so users may end up with wrong dependency.

```
# bad
Gem::Specification.new do |s|
  if RUBY_VERSION >= '2.5'
```

```
    s.add_runtime_dependency 'gem_a'
  else
    s.add_runtime_dependency 'gem_b'
  end
end
```

Fix by either:

- Post-install messages.
- Add both gems as dependency (if permissible).
- If development dependencies, move to Gemfile.

Misc

No Flip-flops

Avoid the use of flip-flop operators.

No non-`nil` Checks

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

```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
```

Global Input/Output Streams

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.

```ruby
# bad
STDOUT.puts('hello')

hash = { out: STDOUT, key: value }

def m(out = STDOUT)
  out.puts('hello')
```

```
end

# good
$stdout.puts('hello')

hash = { out: $stdout, key: value }

def m(out = $stdout)
  out.puts('hello')
end
```

Note	The only valid use-case for the stream constants is obtaining references to the original streams (assuming you've redirected some of the global vars).

Warn

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`).

```
# bad
$stderr.puts 'This is a warning!'

# good
warn 'This is a warning!'
```

Array#join

Prefer the use of `Array#join` over the fairly cryptic `Array#*` with a string argument.

```
# bad
%w[one two three] * ', '
# => 'one, two, three'

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

Array Coercion

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

```
# bad
paths = [paths] unless paths.is_a?(Array)
```

```ruby
paths.each { |path| do_something(path) }

# bad (always creates a new Array instance)
[*paths].each { |path| do_something(path) }

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

Ranges or `between`

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

```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)
```

Predicate Methods

Prefer the use of predicate methods to explicit comparisons with `==`. Numeric comparisons are OK.

```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
```

No Cryptic Perlisms

Avoid using Perl-style special variables (like `$:`, `$;`, etc). They are quite cryptic and their use in anything but one-liner scripts is discouraged.

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

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

Use the human-friendly aliases provided by the `English` library if required.

```
# bad
print $', $$

# good
require 'English'
print $POSTMATCH, $PID
```

Use `require_relative` whenever possible

For all your internal dependencies, you should use `require_relative`. Use of `require` should be reserved for external dependencies

```
# bad
require 'set'
require 'my_gem/spec/helper'
require 'my_gem/lib/something'

# good
require 'set'
require_relative 'helper'
require_relative '../lib/something'
```

This way is more expressive (making clear which dependency is internal or not) and more efficient (as `require_relative` doesn't have to try all of `$LOAD_PATH` contrary to `require`).

Always Warn

Write `ruby -w` safe code.

No Optional Hash Params

Avoid hashes as optional parameters. Does the method do too much? (Object initializers are exceptions for this rule).

Instance Vars

Use module instance variables instead of global variables.

```ruby
# bad
$foo_bar = 1

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

Foo.bar = 1
```

OptionParser

Use `OptionParser` for parsing complex command line options and `ruby -s` for trivial command line options.

No Param Mutations

Do not mutate parameters unless that is the purpose of the method.

Three is the Number Thou Shalt Count

Avoid more than three levels of block nesting.

Functional Code

Code in a functional way, avoiding mutation when that makes sense.

```ruby
a = []; [1, 2, 3].each { |i| a << i * 2 }   # bad
a = [1, 2, 3].map { |i| i * 2 }             # good

a = {}; [1, 2, 3].each { |i| a[i] = i * 17 }                # bad
a = [1, 2, 3].reduce({}) { |h, i| h[i] = i * 17; h }        # good
a = [1, 2, 3].each_with_object({}) { |i, h| h[i] = i * 17 } # good
```

No explicit `.rb` to `require`

Omit the `.rb` extension for filename passed to `require` and `require_relative`.

Note	If the extension is omitted, Ruby tries adding '.rb', '.so', and so on to the name until found. If the file named cannot be found, a `LoadError` will be raised. There is an edge case where `foo.so` file is loaded instead of a `LoadError` if `foo.so` file exists when `require 'foo.rb'` will be changed to `require 'foo'`, but that seems harmless.

```ruby
# bad
require 'foo.rb'
require_relative '../foo.rb'

# good
require 'foo'
require 'foo.so'
require_relative '../foo'
require_relative '../foo.so'
```

Avoid `tap`

The method `tap` can be helpful for debugging purposes but should not be left in production code.

```ruby
# bad
Config.new(hash, path).tap do |config|
  config.check if check
end

# good
config = Config.new(hash, path)
config.check if check
config
```

This is simpler and more efficient.

Tools

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

RuboCop

RuboCop is a Ruby static code analyzer and formatter, based on this style guide. RuboCop already covers a significant portion of the guide and has plugins for most popular Ruby editors and IDEs.

Tip	RuboCop's cops (code checks) have links to the guidelines that they are based on, as part of their metadata.

RubyMine

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

History

This guide started its life in 2011 as an internal company Ruby coding guidelines (written by Bozhidar Batsov). Bozhidar had always been bothered as a Ruby developer about one thing - Python developers had a great programming style reference (PEP-8) and Rubyists never got an official guide, documenting Ruby coding style and best practices. Bozhidar firmly believed that style matters. He also believed that a great hacker community, such as Ruby has, should be quite capable of producing this coveted document. The rest is history…

At some point Bozhidar decided that the work he 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.

Bozhidar served as the guide's only editor for a few years, before a team of editors was formed once the project transitioned to RuboCop HQ.

Since the inception of the guide we'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.

Sources of Inspiration

Many people, books, presentations, articles and other style guides influenced the community Ruby style guide. Here are some of them:

- "The Elements of Style"
- "The Elements of Programming Style"
- The Python Style Guide (PEP-8)
- "Programming Ruby"
- "The Ruby Programming Language"

Contributing

The guide is still a work in progress - some guidelines are lacking examples, some guidelines don't have examples that illustrate them clearly enough. Improving such guidelines 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 our 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 one of the following platforms:

- GitHub Sponsors
- ko-fi
- Patreon
- PayPal

How to Contribute?

It's easy, just follow the contribution guidelines below:

- Fork rubocop/ruby-style-guide on GitHub
- Make your feature addition or bug fix in a feature branch.
- Include a good description of your changes
- Push your feature branch to GitHub
- Send a Pull Request

Colophon

This guide is written in AsciiDoc and is published as HTML using AsciiDoctor. The HTML version of the guide is hosted on GitHub Pages.

Originally the guide was written in Markdown, but was converted to AsciiDoc in 2019.

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?

1. Occasionally we might suggest to the reader to consider some alternatives, though.
2. *BSD/Solaris/Linux/macOS users are covered by default, Windows users have to be extra careful.
3. According to this Wikipedia article the format used to be popular until the early 2000s when it was superseded by RDoc.

Releases

No releases published

Sponsor this project

bbatsov Bozhidar Batsov

patreon.com/bbatsov

opencollective.com/rubocop

tidelift.com/funding/github/rubygems/rubocop

https://www.paypal.me/bbatsov

Learn more about GitHub Sponsors

Packages

No packages published

Contributors 236

+ 225 contributors

Deployments 195

github-pages 7 hours ago

+ 194 deployments