[RUBY COMMANDS](https://ruby-doc.org/core-3.1.0/Integer.html)

|  |  |
| --- | --- |
| Command Description RuboCop mode binding  M-x rubocop-check-project Runs RuboCop on the entire project C-c C-r p  M-x rubocop-check-directory Prompts from a directory on which to run RuboCop C-c C-r d  M-x rubocop-check-current-file Runs RuboCop on the currently visited file C-c C-r f  M-x rubocop-autocorrect-project Runs auto-correct on the entire project C-c C-r P  M-x rubocop-autocorrect-directory Prompts for a directory on which to run auto-correct C-c C-r D  M-x rubocop-autocorrect-current-file Runs auto-correct on the currently visited file. C-c C-r F  If you use them often you might want to enable rubocop-mode which will added some keybindings for them | Команда Описание Обвързване в режим RuboCop  M-x rubocop-check-project Стартира RuboCop за целия проект C-c C-r p  M-x rubocop-check-directory Подканва от директория, в която да стартирате RuboCop C-c C-r d  M-x rubocop-check-current-file Стартира RuboCop в текущо посещавания файл C-c C-r f  M-x rubocop-autocorrect-project Стартира автоматично коригиране на целия проект C-c C-r P  M-x rubocop-autocorrect-directory Подканва за директория, в която да се стартира автоматично коригиране C-c C-r D  M-x rubocop-autocorrect-current-file Стартира автоматично коригиране на текущо посетения файл. C-c C-r F  Ако ги използвате често, може да искате да активирате режим rubocop, който ще добави някои връзки с клавиши за тях |

**[Source Encoding](https://rubystyle.guide/" \l "utf-8)**

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?](https://rubystyle.guide/" \l "tabs-or-spaces)**

Use only spaces for indentation. No hard tabs.

**[Indentation](https://rubystyle.guide/" \l "spaces-indentation)**

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

# bad - four spaces

def some\_method

do\_something

end

# good

def some\_method

do\_something

end

**[Maximum Line Length](https://rubystyle.guide/" \l "max-line-length)**

Limit lines to 80 characters.

|  |  |
| --- | --- |
| Tip | Most editors and IDEs have configurations 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "crlf)**

Use Unix-style line endings.[[2](footnote.)]

|  |  |
| --- | --- |
| 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?](https://rubystyle.guide/" \l "newline-eof)**

End each file with a newline.

|  |  |
| --- | --- |
| Tip | This should be done by through editor configuration, not manually. |

**[Should I Terminate Expressions with](https://rubystyle.guide/" \l "no-semicolon) ;?**

Don’t use ; to terminate statements and expressions.

# bad

puts 'foobar'; # superfluous semicolon

# good

puts 'foobar'

**[One Expression Per Line](https://rubystyle.guide/" \l "one-expression-per-line)**

Use one expression per line.

# bad

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

# good

puts 'foo'

puts 'bar'

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

**[Spaces and Operators](https://rubystyle.guide/" \l "spaces-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.

# 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:

# bad

e = M \* c \*\* 2

# good

e = M \* c\*\*2

* Slash in rational literals:

# bad

o\_scale = 1 / 48r

# good

o\_scale = 1/48r

* Safe navigation operator:

# bad

foo &. bar

foo &.bar

foo&. bar

# good

foo&.bar

**[Spaces and Braces](https://rubystyle.guide/" \l "spaces-braces)**

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

# 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](https://rubystyle.guide/" \l "no-space-bang)**

No space after !.

# bad

! something

# good

!something

**[No Space inside Range Literals](https://rubystyle.guide/" \l "no-space-inside-range-literals)**

No space inside range literals.

# bad

1 .. 3

'a' ... 'z'

# good

1..3

'a'...'z'

**[Indent](https://rubystyle.guide/" \l "indent-when-to-case) when to case**

Indent when as deep as case.

# 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](https://rubystyle.guide/" \l "indent-conditional-assignment)**

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

# bad - pretty convoluted

kind = case year

when 1850..1889 then 'Blues'

when 1890..1909 then 'Ragtime'

when 1910..1929 then 'New Orleans Jazz'

when 1930..1939 then 'Swing'

when 1940..1950 then 'Bebop'

else 'Jazz'

end

result = if some\_cond

calc\_something

else

calc\_something\_else

end

# good - it's apparent what's going on

kind = case year

when 1850..1889 then 'Blues'

when 1890..1909 then 'Ragtime'

when 1910..1929 then 'New Orleans Jazz'

when 1930..1939 then 'Swing'

when 1940..1950 then 'Bebop'

else 'Jazz'

end

result = if some\_cond

calc\_something

else

calc\_something\_else

end

# good (and a bit more width efficient)

kind =

case year

when 1850..1889 then 'Blues'

when 1890..1909 then 'Ragtime'

when 1910..1929 then 'New Orleans Jazz'

when 1930..1939 then 'Swing'

when 1940..1950 then 'Bebop'

else 'Jazz'

end

result =

if some\_cond

calc\_something

else

calc\_something\_else

end

**[Empty Lines between Methods](https://rubystyle.guide/" \l "empty-lines-between-methods)**

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

def some\_method

data = initialize(options)

data.manipulate!

data.result

end

def some\_method

result

end

**[Two or More Empty Lines](https://rubystyle.guide/" \l "two-or-more-empty-lines)**

Don’t use several empty lines in a row.

# bad - It has two empty lines.

some\_method

some\_method

# good

some\_method

some\_method

**[Empty Lines around Attribute Accessor](https://rubystyle.guide/" \l "empty-lines-around-attribute-accessor)**

Use empty lines around attribute accessor.

# 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](https://rubystyle.guide/" \l "empty-lines-around-access-modifier)**

Use empty lines around attribute 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](https://rubystyle.guide/" \l "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

end

end

# good

class Foo

def foo

begin

do\_something do

something

end

rescue

something

end

end

end

**[Trailing Comma in Method Arguments](https://rubystyle.guide/" \l "no-trailing-params-comma)**

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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-trailing-backslash)**

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](https://rubystyle.guide/" \l "consistent-multi-line-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](https://rubystyle.guide/" \l "leading-dot-in-multi-line-chains) .**

When continuing a chained method invocation on another line, keep the . on the second line.

# bad - need to consult first line to understand second line

one.two.three.

four

# good - it's immediately clear what's going on the second line

one.two.three

.four

**[Trailing](https://rubystyle.guide/" \l "trailing-dot-in-multi-line-chains) .**

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

# bad - need to read ahead to the second line to know that the chain continues

one.two.three

.four

# good - it's immediately clear that the expression continues beyond the first line

one.two.three.

four

A discussion on the merits of both alternative styles can be found [here](https://github.com/rubocop-hq/ruby-style-guide/pull/176).

**[Method Arguments Alignment](https://rubystyle.guide/" \l "no-double-indent)**

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.

# starting point (line is too long)

def send\_mail(source)

Mailer.deliver(to: 'bob@example.com', from: 'us@example.com', subject: 'Important message', body: source.text)

end

# bad (double indent)

def send\_mail(source)

Mailer.deliver(

to: 'bob@example.com',

from: 'us@example.com',

subject: 'Important message',

body: source.text)

end

# 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](https://rubystyle.guide/" \l "no-braces-opts-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.

# bad

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

# good

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

**[DSL Method Calls](https://rubystyle.guide/" \l "no-dsl-decorating)**

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

class Person < ActiveRecord::Base

# bad

validates(:name, { presence: true, length: { within: 1..10 } })

# good

validates :name, presence: true, length: { within: 1..10 }

end

**[Space in Method Calls](https://rubystyle.guide/" \l "parens-no-spaces)**

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

# bad

puts (x + y)

# good

puts(x + y)

**[Multi-line Arrays Alignment](https://rubystyle.guide/" \l "align-multiline-arrays)**

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](https://rubystyle.guide/" \l "naming-conventions)**

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

— Phil Karlton

**[English for Identifiers](https://rubystyle.guide/" \l "english-identifiers)**

Name identifiers in English.

# bad - identifier using non-ascii 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](https://rubystyle.guide/" \l "snake-case-symbols-methods-vars)**

Use snake\_case for symbols, methods and variables.

# bad

:'some symbol'

:SomeSymbol

:someSymbol

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](https://rubystyle.guide/" \l "snake-case-symbols-methods-vars-with-numbers)**

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

# 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

**[CamelCase for Classes](https://rubystyle.guide/" \l "camelcase-classes)**

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

# 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](https://rubystyle.guide/" \l "snake-case-files)**

Use snake\_case for naming files, e.g. hello\_world.rb.

**[Snake Case for Directories](https://rubystyle.guide/" \l "snake-case-dirs)**

Use snake\_case for naming directories, e.g. lib/hello\_world/hello\_world.rb.

**[One Class per File](https://rubystyle.guide/" \l "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 CamelCase with snake\_case.

**[Screaming Snake Case for Constants](https://rubystyle.guide/" \l "screaming-snake-case)**

Use SCREAMING\_SNAKE\_CASE for other constants (those that don’t refer to classes and modules).

# bad

SomeConst = 5

# good

SOME\_CONST = 5

**[Predicate Methods Suffix](https://rubystyle.guide/" \l "bool-methods-qmark)**

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.

# bad

def even(value)

end

# good

def even?(value)

end

**[Predicate Methods Prefix](https://rubystyle.guide/" \l "bool-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?.

# 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](https://rubystyle.guide/" \l "dangerous-method-bang)**

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](https://rubystyle.guide/" \l "safe-because-unsafe)**

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](https://rubystyle.guide/" \l "underscore-unused-vars)**

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

# 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](https://rubystyle.guide/" \l "other-arg) 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 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).

# 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](https://rubystyle.guide/" \l "flow-of-control)**

**[for](https://rubystyle.guide/" \l "no-for-loops) 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.

arr = [1, 2, 3]

# bad

for elem in arr do

puts elem

end

# note that elem is accessible outside of the for loop

elem # => 3

# good

arr.each { |elem| puts elem }

# elem is not accessible outside each's block

elem # => NameError: undefined local variable or method `elem'

**[then](https://rubystyle.guide/" \l "no-then) in Multi-line Expression**

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

# bad

if some\_condition then

# body omitted

end

# bad

case foo

when bar then

# body omitted

end

# good

if some\_condition

# body omitted

end

# good

case foo

when bar

# body omitted

end

**[Condition Placement](https://rubystyle.guide/" \l "same-line-condition)**

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

# bad

if

some\_condition

do\_something

do\_something\_else

end

# good

if some\_condition

do\_something

do\_something\_else

end

**[Ternary Operator vs](https://rubystyle.guide/" \l "ternary-operator) if**

Prefer the ternary operator(?:) over if/then/else/end constructs. It’s more common and obviously more concise.

# bad

result = if some\_condition then something else something\_else end

# good

result = some\_condition ? something : something\_else

**[Nested Ternary Operators](https://rubystyle.guide/" \l "no-nested-ternary)**

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.

# bad

some\_condition ? (nested\_condition ? nested\_something : nested\_something\_else) : something\_else

# good

if some\_condition

nested\_condition ? nested\_something : nested\_something\_else

else

something\_else

end

**[Semicolon in](https://rubystyle.guide/" \l "no-semicolon-ifs) if**

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

# bad

result = if some\_condition; something else something\_else end

# good

result = some\_condition ? something : something\_else

**[case](https://rubystyle.guide/" \l "case-vs-if-else) vs if-else**

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

# 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](https://rubystyle.guide/" \l "use-if-case-returns) if/case**

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

# bad

if condition

result = x

else

result = y

end

# good

result =

if condition

x

else

y

end

**[One-line Cases](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-when-semicolons) when**

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

**[!](https://rubystyle.guide/" \l "bang-not-not) vs not**

Use ! instead of not.

# bad - parentheses are required because of op precedence

x = (not something)

# good

x = !something

**[Double Negation](https://rubystyle.guide/" \l "no-bang-bang)**

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?.

# 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](https://rubystyle.guide/" \l "no-and-or-or)/or**

The and and or keywords are banned. The minimal added readability is just not worth the high probability of introducing subtle bugs. For boolean expressions, always use && and || instead. For flow control, use if and unless; && and || are also acceptable but less clear.

# bad

# boolean expression

ok = got\_needed\_arguments and arguments\_are\_valid

# control flow

document.save or raise("Failed to save document!")

# good

# boolean expression

ok = got\_needed\_arguments && arguments\_are\_valid

# control flow

raise("Failed to save document!") unless document.save

# ok

# control flow

document.save || raise("Failed to save document!")

**[Multi-line Ternary Operator](https://rubystyle.guide/" \l "no-multiline-ternary)**

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

**[if](https://rubystyle.guide/" \l "if-as-a-modifier) as a Modifier**

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

# bad

if some\_condition

do\_something

end

# good

do\_something if some\_condition

# another good option

some\_condition && do\_something

**[Multi-line](https://rubystyle.guide/" \l "no-multiline-if-modifiers) 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](https://rubystyle.guide/" \l "no-nested-modifiers)**

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

# bad

do\_something if other\_condition if some\_condition

# good

do\_something if some\_condition && other\_condition

**[if](https://rubystyle.guide/" \l "unless-for-negatives) vs unless**

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

# 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](https://rubystyle.guide/" \l "no-else-with-unless) else with unless**

Do not use unless with else. Rewrite these with the positive case first.

# bad

unless success?

puts 'failure'

else

puts 'success'

end

# good

if success?

puts 'success'

else

puts 'failure'

end

**[Parentheses around Condition](https://rubystyle.guide/" \l "no-parens-around-condition)**

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

# 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](https://rubystyle.guide/" \l "safe-assignment-in-condition). |

**[Multi-line](https://rubystyle.guide/" \l "no-multiline-while-do) while do**

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

# 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](https://rubystyle.guide/" \l "while-as-a-modifier) as a Modifier**

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

# bad

while some\_condition

do\_something

end

# good

do\_something while some\_condition

**[while](https://rubystyle.guide/" \l "until-for-negatives) vs until**

Prefer until over while for negative conditions.

# bad

do\_something while !some\_condition

# good

do\_something until some\_condition

**[Infinite Loop](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "loop-with-break) 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](https://rubystyle.guide/" \l "no-explicit-return) return**

Avoid return where not required for flow of control.

# bad

def some\_method(some\_arr)

return some\_arr.size

end

# good

def some\_method(some\_arr)

some\_arr.size

end

**[Explicit](https://rubystyle.guide/" \l "no-self-unless-required) self**

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

# 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](https://rubystyle.guide/" \l "no-shadowing)**

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

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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-BEGIN-blocks) Blocks**

Avoid the use of BEGIN blocks.

**[END](https://rubystyle.guide/" \l "no-END-blocks) Blocks**

Do not use END blocks. Use Kernel#at\_exit instead.

# bad

END { puts 'Goodbye!' }

# good

at\_exit { puts 'Goodbye!' }

**[Nested Conditionals](https://rubystyle.guide/" \l "no-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.

# 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.

# 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](https://rubystyle.guide/" \l "exceptions)**

**[raise](https://rubystyle.guide/" \l "prefer-raise-over-fail) vs fail**

Prefer raise over fail for exceptions.

# bad

fail SomeException, 'message'

# good

raise SomeException, 'message'

**[Raising Explicit](https://rubystyle.guide/" \l "no-explicit-runtimeerror) 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-return-ensure) 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](https://rubystyle.guide/" \l "begin-implicit) begin**

Use *implicit begin blocks* where possible.

# 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](https://rubystyle.guide/" \l "contingency-methods)**

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

# 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](https://rubystyle.guide/" \l "dont-hide-exceptions)**

Don’t suppress exceptions.

# 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](https://rubystyle.guide/" \l "no-rescue-modifiers) rescue as a Modifier**

Avoid using rescue in its modifier form.

# 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](https://rubystyle.guide/" \l "no-exceptional-flows)**

Don’t use exceptions for flow of control.

# bad

begin

n / d

rescue ZeroDivisionError

puts 'Cannot divide by 0!'

end

# good

if d.zero?

puts 'Cannot divide by 0!'

else

n / d

end

**[Blind Rescues](https://rubystyle.guide/" \l "no-blind-rescues)**

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

# 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](https://rubystyle.guide/" \l "exception-ordering)**

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

# 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

**[Release External Resources](https://rubystyle.guide/" \l "release-resources)**

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

f = File.open('testfile')

begin

# .. process

rescue

# .. handle error

ensure

f.close if f

end

**[Auto-release External Resources](https://rubystyle.guide/" \l "auto-release-resources)**

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

# 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

File.open('testfile') do |f|

# some action on the file

end

**[Standard Exceptions](https://rubystyle.guide/" \l "standard-exceptions)**

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

**[Assignment & Comparison](https://rubystyle.guide/" \l "assignment-comparison)**

**[Parallel Assignment](https://rubystyle.guide/" \l "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.

# bad

a, b, c, d = 'foo', 'bar', 'baz', 'foobar'

# good

a = 'foo'

b = 'bar'

c = 'baz'

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](https://rubystyle.guide/" \l "values-swapping)**

Use parallel assignment when swapping 2 values.

# bad

tmp = x

x = y

y = tmp

# good

x, y = y, x

**[Dealing with Trailing Underscore Variables in Destructuring Assignment](https://rubystyle.guide/" \l "trailing-underscore-variables)**

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.

# bad

foo = 'one,two,three,four,five'

# 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](https://rubystyle.guide/" \l "self-assignment)**

Use shorthand self assignment operators whenever applicable.

# 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](https://rubystyle.guide/" \l "double-pipe-for-uninit)**

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

# bad

name = name ? name : 'Bozhidar'

# bad

name = 'Bozhidar' unless name

# 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.)  # bad - would set enabled to true even if it was false  enabled ||= true  # good  enabled = true if enabled.nil? |

**[Existence Check Shorthand](https://rubystyle.guide/" \l "double-amper-preprocess)**

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.

# 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](https://rubystyle.guide/" \l "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.

# 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](https://rubystyle.guide/" \l "no-case-equality)**

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?](https://rubystyle.guide/" \l "is-a-vs-kind-of) 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?](https://rubystyle.guide/" \l "is-a-vs-instance-of) 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?](https://rubystyle.guide/" \l "instance-of-vs-class-comparison) 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)

**[==](https://rubystyle.guide/" \l "eql) 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](https://rubystyle.guide/" \l "blocks-procs-lambdas)**

**[Proc Application Shorthand](https://rubystyle.guide/" \l "single-action-blocks)**

Use the Proc invocation shorthand when the invoked method is the only operation of a block.

# bad

names.map { |name| name.upcase }

# good

names.map(&:upcase)

**[Single-line Blocks Delimiters](https://rubystyle.guide/" \l "single-line-blocks)**

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.

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?

**[Explicit Block Argument](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-trailing-parameters-comma)**

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 }

# 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](https://rubystyle.guide/" \l "no-nested-methods)**

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 invoked.

# 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](https://rubystyle.guide/" \l "lambda-multi-line)**

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

# 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](https://rubystyle.guide/" \l "stabby-lambda-with-args)**

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](https://rubystyle.guide/" \l "stabby-lambda-no-args)**

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

# bad

l = ->() { something }

# good

l = -> { something }

**[proc](https://rubystyle.guide/" \l "proc) vs Proc.new**

Prefer proc over Proc.new.

# bad

p = Proc.new { |n| puts n }

# good

p = proc { |n| puts n }

**[Proc Invocation](https://rubystyle.guide/" \l "proc-call)**

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

# bad - looks similar to Enumeration access

l = ->(v) { puts v }

l[1]

# good - 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](https://rubystyle.guide/" \l "methods)**

**[Short Methods](https://rubystyle.guide/" \l "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.

**[No Single-line Methods](https://rubystyle.guide/" \l "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.

# bad

def too\_much; something; something\_else; end

# okish - notice that the first ; is required

def no\_braces\_method; body end

# okish - notice that the second ; is optional

def no\_braces\_method; body; end

# okish - valid syntax, but no ; makes it kind of hard to read

def some\_method() body end

# good

def some\_method

body

end

One exception to the rule are empty-body methods.

# good

def no\_op; end

**[Double Colons](https://rubystyle.guide/" \l "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 invocation.

# 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "method-parens)**

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

# bad

def some\_method\_with\_parameters param1, param2

# body omitted

end

# good

def some\_method\_with\_parameters(param1, param2)

# body omitted

end

**[Method Invocation Parentheses](https://rubystyle.guide/" \l "method-invocation-parens)**

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

# 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 Calls with No Arguments](https://rubystyle.guide/" \l "method-invocation-parens-no-args)**

Always omit parentheses for method calls with no arguments.

# bad

Kernel.exit!()

2.even?()

fork()

'test'.upcase()

# good

Kernel.exit!

2.even?

fork

'test'.upcase

**[Methods That are Part of an Internal DSL](https://rubystyle.guide/" \l "method-invocation-parens-internal-dsl)**

Always omit parentheses for methods that are part of an internal DSL (e.g., Rake, Rails, RSpec):

# bad

validates(:name, presence: true)

# good

validates :name, presence: true

**[Methods That Have "keyword" Status in Ruby](https://rubystyle.guide/" \l "method-invocation-parens-keyword)**

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. |

**[Declarative Methods That Have "keyword" Status in Ruby](https://rubystyle.guide/" \l "method-invocation-parens-declarative-keyword)**

Always omit parentheses for declarative methods (a.k.a. DSL methods or macro methods) that have "keyword" status in Ruby (e.g., various Module instance methods):

class Person

# bad

attr\_reader(:name, :age)

# good

attr\_reader :name, :age

# body omitted

end

**[Non-Declarative Methods That Have "keyword" Status in Ruby](https://rubystyle.guide/" \l "method-invocation-parens-non-declarative-keyword)**

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.

# good (most popular)

puts temperance.age

system 'ls'

exit 1

# also good (less popular)

puts(temperance.age)

system('ls')

exit(1)

**[Using](https://rubystyle.guide/" \l "super-with-args) super with Arguments**

Always use parentheses when calling super with arguments:

# 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](https://rubystyle.guide/" \l "too-many-params)**

Avoid parameter lists longer than three or four parameters.

**[Optional Arguments](https://rubystyle.guide/" \l "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.

# 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'

**[Boolean Keyword Arguments](https://rubystyle.guide/" \l "boolean-keyword-arguments)**

Use keyword arguments when passing boolean argument to a method.

# bad

def some\_method(bar = false)

puts bar

end

# bad - common hack before keyword args were introduced

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](https://rubystyle.guide/" \l "keyword-arguments-vs-optional-arguments)**

Prefer keyword arguments over optional arguments.

# 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](https://rubystyle.guide/" \l "keyword-arguments-vs-option-hashes)**

Use keyword arguments instead of option hashes.

# bad

def some\_method(options = {})

bar = options.fetch(:bar, false)

puts bar

end

# good

def some\_method(bar: false)

puts bar

end

**[Arguments Forwarding](https://rubystyle.guide/" \l "arguments-forwarding)**

Use Ruby 2.7’s arguments forwarding.

# 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-hq/rubocop/issues/7549

def some\_method(\*args)

other\_method(\*args)

end

# good

def some\_method(...)

other\_method(...)

end

**[Private Global Methods](https://rubystyle.guide/" \l "private-global-methods)**

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

**[Classes & Modules](https://rubystyle.guide/" \l "classes-modules)**

**[Consistent Classes](https://rubystyle.guide/" \l "consistent-classes)**

Use a consistent structure in your class definitions.

class Person

# extend and include go first

extend SomeModule

include AnotherModule

# inner classes

CustomError = Class.new(StandardError)

# constants are next

SOME\_CONSTANT = 20

# afterwards we have attribute macros

attr\_reader :name

# followed by other macros (if any)

validates :name

# public class methods are next in line

def self.some\_method

end

# 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](https://rubystyle.guide/" \l "mixin-grouping)**

Split multiple mixins into separate statements.

# bad

class Person

include Foo, Bar

end

# good

class Person

# multiple mixins go in separate statements

include Foo

include Bar

end

**[Single-line Classes](https://rubystyle.guide/" \l "single-line-classes)**

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

# 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](https://rubystyle.guide/" \l "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.

# 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](https://rubystyle.guide/" \l "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](https://cirw.in/blog/constant-lookup.html), 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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

# 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](https://rubystyle.guide/" \l "liskov)**

When designing class hierarchies make sure that they conform to the [Liskov Substitution Principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle).

**[SOLID design](https://rubystyle.guide/" \l "solid-design)**

Try to make your classes as [SOLID](https://en.wikipedia.org/wiki/SOLID) as possible.

**[Define](https://rubystyle.guide/" \l "define-to-s) to\_s**

Always supply a proper to\_s method for classes that represent domain objects.

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](https://rubystyle.guide/" \l "attr_family) Family**

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

# 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-extend-struct-new) 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)

**[Duck Typing](https://rubystyle.guide/" \l "duck-typing)**

Prefer [duck-typing](https://en.wikipedia.org/wiki/Duck_typing) over inheritance.

# 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](https://rubystyle.guide/" \l "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.](https://rubystyle.guide/" \l "visibility) 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](https://rubystyle.guide/" \l "indent-public-private-protected)**

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](https://rubystyle.guide/" \l "def-self-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](https://rubystyle.guide/" \l "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.

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.

class Fugitive < Westerner

def first\_name

'Nobody'

end

alias given\_name first\_name

end

**[alias\_method](https://rubystyle.guide/" \l "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.

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](https://rubystyle.guide/" \l "class-and-self) 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.

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](https://rubystyle.guide/" \l "no-constant-definition-in-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.

# 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](https://rubystyle.guide/" \l "classes-constructors)**

**[Factory Methods](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-comments)**

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

**[Rationale Comments](https://rubystyle.guide/" \l "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.

# 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](https://rubystyle.guide/" \l "english-comments)**

Write comments in English.

**[Hash Space](https://rubystyle.guide/" \l "hash-space)**

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

**[English Syntax](https://rubystyle.guide/" \l "english-syntax)**

Comments longer than a word are capitalized and use punctuation. Use [one space](https://en.wikipedia.org/wiki/Sentence_spacing) after periods.

**[No Superfluous Comments](https://rubystyle.guide/" \l "no-superfluous-comments)**

Avoid superfluous comments.

# bad

counter += 1 # Increments counter by one.

**[Comment Upkeep](https://rubystyle.guide/" \l "comment-upkeep)**

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

**[Refactor, Don’t Comment](https://rubystyle.guide/" \l "refactor-dont-comment)**

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

— old programmers maxim [*through Russ Olsen*](http://eloquentruby.com/blog/2011/03/07/good-code-and-good-jokes/)

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](https://rubystyle.guide/" \l "comment-annotations)**

**[Annotations Placement](https://rubystyle.guide/" \l "annotate-above)**

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

# 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](https://rubystyle.guide/" \l "annotate-keywords)**

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

# 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](https://rubystyle.guide/" \l "indent-annotations)**

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

def bar

# FIXME: This has crashed occasionally since v3.2.1. It may

# be related to the BarBazUtil upgrade.

baz(:quux)

end

**[Inline Annotations](https://rubystyle.guide/" \l "rare-eol-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.

def bar

sleep 100 # OPTIMIZE

end

**[TODO](https://rubystyle.guide/" \l "todo)**

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

**[FIXME](https://rubystyle.guide/" \l "fixme)**

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

**[OPTIMIZE](https://rubystyle.guide/" \l "optimize)**

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

**[HACK](https://rubystyle.guide/" \l "hack)**

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

**[REVIEW](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "magic-comments)**

**[Magic Comments First](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "separate-magic-comments-from-code)**

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

# 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](https://rubystyle.guide/" \l "collections)**

**[Literal Array and Hash](https://rubystyle.guide/" \l "literal-array-hash)**

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

# bad

arr = Array.new

hash = Hash.new

# good

arr = []

arr = Array.new(10)

hash = {}

hash = Hash.new(0)

**[%w](https://rubystyle.guide/" \l "percent-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.

# bad

STATES = ['draft', 'open', 'closed']

# good

STATES = %w[draft open closed]

**[%i](https://rubystyle.guide/" \l "percent-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.

# bad

STATES = [:draft, :open, :closed]

# good

STATES = %i[draft open closed]

**[No Trailing Array Commas](https://rubystyle.guide/" \l "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.

# 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](https://rubystyle.guide/" \l "no-gappy-arrays)**

Avoid the creation of huge gaps in arrays.

arr = []

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

**[first](https://rubystyle.guide/" \l "first-and-last) and last**

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

**[Set vs Array](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "symbols-as-keys)**

Prefer symbols instead of strings as hash keys.

# bad

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

# good

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

**[No Mutable Keys](https://rubystyle.guide/" \l "no-mutable-keys)**

Avoid the use of mutable objects as hash keys.

**[Hash Literals](https://rubystyle.guide/" \l "hash-literals)**

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

# bad

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

# good

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

**[Hash Literal as Last Array Item](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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 }

**[Hash#key?](https://rubystyle.guide/" \l "hash-key)**

Use Hash#key? instead of Hash#has\_key? and Hash#value? instead of Hash#has\_value?.

# bad

hash.has\_key?(:test)

hash.has\_value?(value)

# good

hash.key?(:test)

hash.value?(value)

**[Hash#each](https://rubystyle.guide/" \l "hash-each)**

Use Hash#each\_key instead of Hash#keys.each and Hash#each\_value instead of Hash#values.each.

# 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](https://rubystyle.guide/" \l "hash-fetch)**

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

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](https://rubystyle.guide/" \l "hash-fetch-defaults) defaults**

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

batman = { name: 'Bruce Wayne', is\_evil: false }

# bad - if we just use || operator with falsy value we won't get the expected result

batman[:is\_evil] || true # => true

# good - fetch works correctly with falsy values

batman.fetch(:is\_evil, true) # => false

**[Use Hash Blocks](https://rubystyle.guide/" \l "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.

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](https://rubystyle.guide/" \l "hash-values-at)**

Use Hash#values\_at when you need to retrieve several values consecutively from a hash.

# bad

email = data['email']

username = data['nickname']

# good

email, username = data.values\_at('email', 'nickname')

**[Hash#transform\_keys](https://rubystyle.guide/" \l "hash-transform-methods) 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.

# 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](https://rubystyle.guide/" \l "ordered-hashes)**

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

**[No Modifying Collections](https://rubystyle.guide/" \l "no-modifying-collections)**

Do not modify a collection while traversing it.

**[Accessing Elements Directly](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "map-find-select-reduce-include-size)/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](https://rubystyle.guide/" \l "count-vs-size) 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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 { ... }

**[Numbers](https://rubystyle.guide/" \l "numbers)**

**[Underscores in Numerics](https://rubystyle.guide/" \l "underscores-in-numerics)**

Add underscores to large numeric literals to improve their readability.

# bad - how many 0s are there?

num = 1000000

# good - much easier to parse for the human brain

num = 1\_000\_000

**[Numeric Literal Prefixes](https://rubystyle.guide/" \l "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.

# 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](https://rubystyle.guide/" \l "integer-type-checking)**

Use Integer to check 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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.

# 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "strings)**

**[String Interpolation](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "consistent-string-literals-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](https://rubystyle.guide/" \l "consistent-string-literals-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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "curlies-interpolate)**

Don’t leave out {} around instance and global variables being interpolated into a string.

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](https://rubystyle.guide/" \l "no-to-s) to\_s**

Don’t use Object#to\_s on interpolated objects. It’s invoked on them automatically.

# bad

message = "This is the #{result.to\_s}."

# good

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

**[String Concatenation](https://rubystyle.guide/" \l "concat-strings)**

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.

# bad

html = ''

html += '<h1>Page title</h1>'

paragraphs.each do |paragraph|

html += "<p>#{paragraph}</p>"

end

# good and also fast

html = ''

html << '<h1>Page title</h1>'

paragraphs.each do |paragraph|

html << "<p>#{paragraph}</p>"

end

**[Don’t Abuse](https://rubystyle.guide/" \l "dont-abuse-gsub) gsub**

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

url = 'http://example.com'

str = 'lisp-case-rules'

# bad

url.gsub('http://', 'https://')

str.gsub('-', '\_')

# good

url.sub('http://', 'https://')

str.tr('-', '\_')

**[sprintf](https://rubystyle.guide/" \l "sprintf)**

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

# bad

'%d %d' % [20, 10]

# => '20 10'

# 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](https://rubystyle.guide/" \l "named-format-tokens)**

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

# bad

format('Hello, %{name}', name: 'John')

# good

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

**[Long Strings](https://rubystyle.guide/" \l "heredoc-long-strings)**

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

# 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](https://rubystyle.guide/" \l "heredocs)**

**[Squiggly Heredocs](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "date-time)**

**[Time.now](https://rubystyle.guide/" \l "time-now)**

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

**[No](https://rubystyle.guide/" \l "no-datetime) 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-regexp-for-plaintext)**

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](https://rubystyle.guide/" \l "regexp-string-index)**

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](https://rubystyle.guide/" \l "non-capturing-regexp)**

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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "no-perl-regexp-last-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](https://rubystyle.guide/" \l "no-numbered-regexes)**

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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "multi-line-regexes)**

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

|  |  |
| --- | --- |
| Note | That’s known as [free-spacing mode](http://www.regular-expressions.info/freespacing.html). 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](https://rubystyle.guide/" \l "comment-regexes)**

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](https://rubystyle.guide/" \l "gsub-blocks) 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](https://rubystyle.guide/" \l "percent-literals)**

**[%q](https://rubystyle.guide/" \l "percent-q-shorthand) 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](https://rubystyle.guide/" \l "percent-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](https://rubystyle.guide/" \l "percent-r)**

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

# bad

%r{\s+}

# good

%r{^/(.\*)$}

%r{^/blog/2011/(.\*)$}

**[%x](https://rubystyle.guide/" \l "percent-x)**

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

# bad

date = %x(date)

# good

date = `date`

echo = %x(echo `date`)

**[%s](https://rubystyle.guide/" \l "percent-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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "metaprogramming)**

**[No Needless Metaprogramming](https://rubystyle.guide/" \l "no-needless-metaprogramming)**

Avoid needless metaprogramming.

**[No Monkey Patching](https://rubystyle.guide/" \l "no-monkey-patching)**

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

**[Block](https://rubystyle.guide/" \l "block-class-eval) class\_eval**

The block form of class\_eval is preferable to the string-interpolated form.

**[Supply Location](https://rubystyle.guide/" \l "class-eval-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](https://rubystyle.guide/" \l "class-eval-define_method)**

define\_method is preferable to class\_eval { def …​ }

**[eval](https://rubystyle.guide/" \l "eval-comment-docs) 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](https://rubystyle.guide/" \l "no-method-missing) 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?](http://blog.marc-andre.ca/2010/11/15/methodmissing-politely/)
* 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](https://rubystyle.guide/" \l "prefer-public-send) public\_send**

Prefer public\_send over send so as not to circumvent private/protected visibility.

# 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](https://rubystyle.guide/" \l "prefer-__send__) \_\_send\_\_**

Prefer \_\_send\_\_ over send, as send may overlap with existing methods.

require 'socket'

u1 = UDPSocket.new

u1.bind('127.0.0.1', 4913)

u2 = UDPSocket.new

u2.connect('127.0.0.1', 4913)

# Won't send a message to the receiver obj.

# Instead it will send a message via UDP socket.

u2.send :sleep, 0

# Will actually send a message to the receiver obj.

u2.\_\_send\_\_ ...

**[API Documentation](https://rubystyle.guide/" \l "api-documentation)**

**[YARD](https://rubystyle.guide/" \l "yard)**

Use [YARD](https://yardoc.org/) and its conventions for API documentation.

**[RD (Block) Comments](https://rubystyle.guide/" \l "no-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 a really a block comment syntax, but more of an attempt to emulate Perl’s [POD](https://perldoc.perl.org/perlpod.html) documentation system.

There’s an [rdtool](https://github.com/uwabami/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](https://github.com/uwabami/rdtool/blob/master/doc/rd-draft.rd). You can read more about it [here](https://ruby-doc.com/docs/ProgrammingRuby/html/rdtool.html).

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

**[Gemfile and Gemspec](https://rubystyle.guide/" \l "gemfile-and-gemspec)**

**[No](https://rubystyle.guide/" \l "no-ruby-version-in-the-gemspec) 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](https://rubystyle.guide/" \l "misc)**

**[No Flip-flops](https://rubystyle.guide/" \l "no-flip-flops)**

Avoid the use of flip-flops.

|  |  |
| --- | --- |
| Note | They are deprecated as of Ruby 2.6. |

**[No non-](https://rubystyle.guide/" \l "no-non-nil-checks)nil Checks**

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

# 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](https://rubystyle.guide/" \l "global-stdout)**

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.

# 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "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)

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](https://rubystyle.guide/" \l "ranges-or-between) between**

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

# 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](https://rubystyle.guide/" \l "predicate-methods)**

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

# 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](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "use-require_relative-whenever-possible) 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](https://rubystyle.guide/" \l "always-warn)**

Write ruby -w safe code.

**[No Optional Hash Params](https://rubystyle.guide/" \l "no-optional-hash-params)**

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

**[Instance Vars](https://rubystyle.guide/" \l "instance-vars)**

Use module instance variables instead of global variables.

# bad

$foo\_bar = 1

# good

module Foo

class << self

attr\_accessor :bar

end

end

Foo.bar = 1

**[OptionParser](https://rubystyle.guide/" \l "optionparser)**

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

**[No Param Mutations](https://rubystyle.guide/" \l "no-param-mutations)**

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

**[Three is the Number Thou Shalt Count](https://rubystyle.guide/" \l "three-is-the-number-thou-shalt-count)**

Avoid more than three levels of block nesting.

**[Functional Code](https://rubystyle.guide/" \l "functional-code)**

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

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](https://rubystyle.guide/" \l "no-explicit-rb-to-require) .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. |

# bad

require 'foo.rb'

require\_relative '../foo.rb'

# good

require 'foo'

require 'foo.so'

require\_relative '../foo'

require\_relative '../foo.so'

**[Tools](https://rubystyle.guide/" \l "tools)**

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

**[RuboCop](https://rubystyle.guide/" \l "rubocop)**

[RuboCop](https://github.com/rubocop-hq/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](https://docs.rubocop.org/en/stable/integration_with_other_tools/) 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](https://rubystyle.guide/" \l "rubymine)**

[RubyMine](http://www.jetbrains.com/ruby/)'s code inspections are [partially based](http://confluence.jetbrains.com/display/RUBYDEV/RubyMine+Inspections) on this guide.

**[History](https://rubystyle.guide/" \l "history)**

This guide started its life as an internal company Ruby coding guidelines (written by [Bozhidar Batsov](https://github.com/bbatsov)). Bozhidar had always been bothered as a Ruby developer about one thing - Python developers had a great programming style reference ([PEP-8](https://www.python.org/dev/peps/pep-0008/)) 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](https://rubystyle.guide/" \l "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](https://en.wikipedia.org/wiki/The_Elements_of_Style)
* [The Elements of Programming Style](https://en.wikipedia.org/wiki/The_Elements_of_Programming_Style)
* [PEP-8](https://www.python.org/dev/peps/pep-0008/)
* ["Programming Ruby"](https://pragprog.com/book/ruby4/programming-ruby-1-9-2-0)
* ["The Ruby Programming Language"](http://www.amazon.com/Ruby-Programming-Language-David-Flanagan/dp/0596516177)

**[Contributing](https://rubystyle.guide/" \l "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 [Patreon](https://www.patreon.com/bbatsov).

**[How to Contribute?](https://rubystyle.guide/" \l "how-to-contribute)**

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

* [Fork](https://help.github.com/articles/fork-a-repo) [rubocop-hq/ruby-style-guide](https://github.com/rubocop-hq/ruby-style-guide) on GitHub
* Make your feature addition or bug fix in a feature branch.
* Include a [good description](http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html) of your changes
* Push your feature branch to GitHub
* Send a [Pull Request](https://help.github.com/articles/using-pull-requests)

**[Colophon](https://rubystyle.guide/" \l "colophon)**

This guide is written in [AsciiDoc](http://asciidoc.org/) and is published as HTML using [AsciiDoctor](https://asciidoctor.org/). 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](https://rubystyle.guide/" \l "license)**

This work is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/deed.en_US)

**[Spread the Word](https://rubystyle.guide/" \l "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](https://rubystyle.guide/" \l "_footnoteref_1). Occasionally we might suggest to the reader to consider some alternatives, though.

[2](https://rubystyle.guide/" \l "_footnoteref_2). \*BSD/Solaris/Linux/macOS users are covered by default, Windows users have to be extra careful.

[3](https://rubystyle.guide/" \l "_footnoteref_3). According to this [Wikipedia article](https://en.wikipedia.org/wiki/Ruby_Document_format) the format used to be popular until the early 2000s when it was superseded by RDoc.

Last updated 2020-10-25 13:35:36 UTC