Industrial Standards

**Java Coding Standards**

**Oracle/Sun Java Code Conventions & Google Java Style Guide**

**Naming Conventions:**

* Classes: PascalCase (e.g., CustomerService)
* Methods and variables: camelCase (e.g., calculateTotalAmount)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_RETRY\_COUNT)
* Packages: lowercase with dots (e.g., com.company.module)

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 4 spaces (no tabs)
* Braces: Egyptian style (opening brace on same line)
* One class per file, file name matches class name

**Best Practices:**

* Use meaningful variable and method names
* Avoid deep nesting (maximum 3-4 levels)
* Implement proper exception handling with specific exception types
* Use StringBuilder for string concatenation in loops
* Follow SOLID principles and design patterns
* Maintain cyclomatic complexity below 10 per method

**Security Standards:**

* Input validation and sanitization
* Proper resource management (try-with-resources)
* Avoid hardcoded credentials
* Use parameterized queries for database operations

**Python Coding Standards**

[**Code Lay-out**](https://peps.python.org/pep-0008/#code-lay-out)

[Indentation](https://peps.python.org/pep-0008/#indentation)

Use 4 spaces per indentation level.

Continuation lines should align wrapped elements either vertically using Python’s implicit line joining inside parentheses, brackets and braces, or using a *hanging indent* [[1]](https://peps.python.org/pep-0008/#fn-hi). When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line:

*# Correct:*

*# Aligned with opening delimiter.*

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

*# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.*

**def** long\_function\_name(

var\_one, var\_two, var\_three,

var\_four):

print(var\_one)

*# Hanging indents should add a level.*

foo = long\_function\_name(

var\_one, var\_two,

var\_three, var\_four)

*# Wrong:*

*# Arguments on first line forbidden when not using vertical alignment.*

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

*# Further indentation required as indentation is not distinguishable.*

**def** long\_function\_name(

var\_one, var\_two, var\_three,

var\_four):

print(var\_one)

The 4-space rule is optional for continuation lines.

Optional:

*# Hanging indents \*may\* be indented to other than 4 spaces.*

foo = long\_function\_name(

var\_one, var\_two,

var\_three, var\_four)

When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it’s worth noting that the combination of a two character keyword (i.e. if), plus a single space, plus an opening parenthesis creates a natural 4-space indent for the subsequent lines of the multiline conditional. This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces. This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement. Acceptable options in this situation include, but are not limited to:

*# No extra indentation.*

**if** (this\_is\_one\_thing **and**

that\_is\_another\_thing):

do\_something()

*# Add a comment, which will provide some distinction in editors*

*# supporting syntax highlighting.*

**if** (this\_is\_one\_thing **and**

that\_is\_another\_thing):

*# Since both conditions are true, we can frobnicate.*

do\_something()

*# Add some extra indentation on the conditional continuation line.*

**if** (this\_is\_one\_thing

**and** that\_is\_another\_thing):

do\_something()

(Also see the discussion of whether to break before or after binary operators below.)

The closing brace/bracket/parenthesis on multiline constructs may either line up under the first non-whitespace character of the last line of list, as in:

my\_list = [

1, 2, 3,

4, 5, 6,

]

result = some\_function\_that\_takes\_arguments(

'a', 'b', 'c',

'd', 'e', 'f',

)

or it may be lined up under the first character of the line that starts the multiline construct, as in:

my\_list = [

1, 2, 3,

4, 5, 6,

]

result = some\_function\_that\_takes\_arguments(

'a', 'b', 'c',

'd', 'e', 'f',

)

[Tabs or Spaces?](https://peps.python.org/pep-0008/#tabs-or-spaces)

Spaces are the preferred indentation method.

Tabs should be used solely to remain consistent with code that is already indented with tabs.

Python disallows mixing tabs and spaces for indentation.

[Maximum Line Length](https://peps.python.org/pep-0008/#maximum-line-length)

Limit all lines to a maximum of 79 characters.

For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.

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 99 characters, provided that comments and docstrings are still wrapped at 72 characters.

The Python standard library is conservative and requires limiting lines to 79 characters (and docstrings/comments to 72).

The preferred way of wrapping long lines is by using Python’s implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Backslashes may still be appropriate at times. For example, long, multiple with-statements could not use implicit continuation before Python 3.10, so backslashes were acceptable for that case:

**with** open('/path/to/some/file/you/want/to/read') **as** file\_1, \

open('/path/to/some/file/being/written', 'w') **as** file\_2:

file\_2.write(file\_1.read())

(See the previous discussion on [multiline if-statements](https://peps.python.org/pep-0008/#multiline-if-statements) for further thoughts on the indentation of such multiline with-statements.)

Another such case is with assert statements.

Make sure to indent the continued line appropriately.

[Should a Line Break Before or After a Binary Operator?](https://peps.python.org/pep-0008/#should-a-line-break-before-or-after-a-binary-operator)

For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved away from its operand and onto the previous line. Here, the eye has to do extra work to tell which items are added and which are subtracted:

*# Wrong:*

*# operators sit far away from their operands*

income = (gross\_wages +

taxable\_interest +

(dividends - qualified\_dividends) -

ira\_deduction -

student\_loan\_interest)

To solve this readability problem, mathematicians and their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his *Computers and Typesetting* series: “Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations” [[3]](https://peps.python.org/pep-0008/#id7).

Following the tradition from mathematics usually results in more readable code:

*# Correct:*

*# easy to match operators with operands*

income = (gross\_wages

+ taxable\_interest

+ (dividends - qualified\_dividends)

- ira\_deduction

- student\_loan\_interest)

In Python code, it is permissible to break before or after a binary operator, as long as the convention is consistent locally. For new code Knuth’s style is suggested.

[Blank Lines](https://peps.python.org/pep-0008/#blank-lines)

Surround top-level function and class definitions with two blank lines.

Method definitions inside a class are surrounded by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. ^L) form feed character as whitespace; many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognize control-L as a form feed and will show another glyph in its place.

[Source File Encoding](https://peps.python.org/pep-0008/#source-file-encoding)

Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.

In the standard library, non-UTF-8 encodings should be used only for test purposes. Use non-ASCII characters sparingly, preferably only to denote places and human names. If using non-ASCII characters as data, avoid noisy Unicode characters like z̯̯͡a̧͎̺l̡͓̫g̹̲o̡̼̘ and byte order marks.

All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren’t English).

Open source projects with a global audience are encouraged to adopt a similar policy.

[Imports](https://peps.python.org/pep-0008/#imports)

* Imports should usually be on separate lines:
* *# Correct:*
* **import** os
* **import** sys
* *# Wrong:*
* **import** sys, os

It’s okay to say this though:

*# Correct:*

**from** subprocess **import** Popen, PIPE

* Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

* 1. Standard library imports.
  2. Related third party imports.
  3. Local application/library specific imports.

You should put a blank line between each group of imports.

* Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):
* **import** mypkg.sibling
* **from** mypkg **import** sibling
* **from** mypkg.sibling **import** example

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

**from** . **import** sibling

**from** .sibling **import** example

Standard library code should avoid complex package layouts and always use absolute imports.

* When importing a class from a class-containing module, it’s usually okay to spell this:
* **from** myclass **import** MyClass
* **from** foo.bar.yourclass **import** YourClass

If this spelling causes local name clashes, then spell them explicitly:

**import** myclass

**import** foo.bar.yourclass

and use myclass.MyClass and foo.bar.yourclass.YourClass.

* Wildcard imports (from <module> import \*) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools. There is one defensible use case for a wildcard import, which is to republish an internal interface as part of a public API (for example, overwriting a pure Python implementation of an interface with the definitions from an optional accelerator module and exactly which definitions will be overwritten isn’t known in advance).

When republishing names this way, the guidelines below regarding public and internal interfaces still apply.

[Module Level Dunder Names](https://peps.python.org/pep-0008/#module-level-dunder-names)

Module level “dunders” (i.e. names with two leading and two trailing underscores) such as \_\_all\_\_, \_\_author\_\_, \_\_version\_\_, etc. should be placed after the module docstring but before any import statements *except* from \_\_future\_\_ imports. Python mandates that future-imports must appear in the module before any other code except docstrings:

"""This is the example module.

This module does stuff.

"""

**from** \_\_future\_\_ **import** barry\_as\_FLUFL

\_\_all\_\_ = ['a', 'b', 'c']

\_\_version\_\_ = '0.1'

\_\_author\_\_ = 'Cardinal Biggles'

**import** os

**import** sys

[**String Quotes**](https://peps.python.org/pep-0008/#string-quotes)

In Python, single-quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.

For triple-quoted strings, always use double quote characters to be consistent with the docstring convention in [PEP 257](https://peps.python.org/pep-0257/).

[**Whitespace in Expressions and Statements**](https://peps.python.org/pep-0008/#whitespace-in-expressions-and-statements)

[Pet Peeves](https://peps.python.org/pep-0008/#pet-peeves)

Avoid extraneous whitespace in the following situations:

* Immediately inside parentheses, brackets or braces:
* *# Correct:*
* spam(ham[1], {eggs: 2})
* *# Wrong:*
* spam( ham[ 1 ], { eggs: 2 } )
* Between a trailing comma and a following close parenthesis:
* *# Correct:*
* foo = (0,)
* *# Wrong:*
* bar = (0, )
* Immediately before a comma, semicolon, or colon:
* *# Correct:*
* **if** x == 4: print(x, y); x, y = y, x
* *# Wrong:*
* **if** x == 4 : print(x , y) ; x , y = y , x
* However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted:
* *# Correct:*
* ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
* ham[lower:upper], ham[lower:upper:], ham[lower::step]
* ham[lower+offset : upper+offset]
* ham[: upper\_fn(x) : step\_fn(x)], ham[:: step\_fn(x)]
* ham[lower + offset : upper + offset]
* *# Wrong:*
* ham[lower + offset:upper + offset]
* ham[1: 9], ham[1 :9], ham[1:9 :3]
* ham[lower : : step]
* ham[ : upper]
* Immediately before the open parenthesis that starts the argument list of a function call:
* *# Correct:*
* spam(1)
* *# Wrong:*
* spam (1)
* Immediately before the open parenthesis that starts an indexing or slicing:
* *# Correct:*
* dct['key'] = lst[index]
* *# Wrong:*
* dct ['key'] = lst [index]
* More than one space around an assignment (or other) operator to align it with another:
* *# Correct:*
* x = 1
* y = 2
* long\_variable = 3
* *# Wrong:*
* x = 1
* y = 2
* long\_variable = 3

[Other Recommendations](https://peps.python.org/pep-0008/#other-recommendations)

* Avoid trailing whitespace anywhere. Because it’s usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker. Some editors don’t preserve it and many projects (like CPython itself) have pre-commit hooks that reject it.
* Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <=, >=, in, not in, is, is not), Booleans (and, or, not).
* If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator:
* *# Correct:*
* i = i + 1
* submitted += 1
* x = x\*2 - 1
* hypot2 = x\*x + y\*y
* c = (a+b) \* (a-b)
* *# Wrong:*
* i=i+1
* submitted +=1
* x = x \* 2 - 1
* hypot2 = x \* x + y \* y
* c = (a + b) \* (a - b)
* Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present. (See [Function Annotations](https://peps.python.org/pep-0008/#function-annotations) below for more about function annotations.):
* *# Correct:*
* **def** munge(input: AnyStr): ...
* **def** munge() -> PosInt: ...
* *# Wrong:*
* **def** munge(input:AnyStr): ...
* **def** munge()->PosInt: ...
* Don’t use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter:
* *# Correct:*
* **def** complex(real, imag=0.0):
* **return** magic(r=real, i=imag)
* *# Wrong:*
* **def** complex(real, imag = 0.0):
* **return** magic(r = real, i = imag)

When combining an argument annotation with a default value, however, do use spaces around the = sign:

*# Correct:*

**def** munge(sep: AnyStr = **None**): ...

**def** munge(input: AnyStr, sep: AnyStr = **None**, limit=1000): ...

*# Wrong:*

**def** munge(input: AnyStr=**None**): ...

**def** munge(input: AnyStr, limit = 1000): ...

* Compound statements (multiple statements on the same line) are generally discouraged:
* *# Correct:*
* **if** foo == 'blah':
* do\_blah\_thing()
* do\_one()
* do\_two()
* do\_three()

Rather not:

*# Wrong:*

**if** foo == 'blah': do\_blah\_thing()

do\_one(); do\_two(); do\_three()

* While sometimes it’s okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

*# Wrong:*

**if** foo == 'blah': do\_blah\_thing()

**for** x **in** lst: total += x

**while** t < 10: t = delay()

Definitely not:

*# Wrong:*

**if** foo == 'blah': do\_blah\_thing()

**else**: do\_non\_blah\_thing()

**try**: something()

**finally**: cleanup()

do\_one(); do\_two(); do\_three(long, argument,

list, like, this)

**if** foo == 'blah': one(); two(); three()

[**When to Use Trailing Commas**](https://peps.python.org/pep-0008/#when-to-use-trailing-commas)

Trailing commas are usually optional, except they are mandatory when making a tuple of one element. For clarity, it is recommended to surround the latter in (technically redundant) parentheses:

*# Correct:*

FILES = ('setup.cfg',)

*# Wrong:*

FILES = 'setup.cfg',

When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items is expected to be extended over time. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter (except in the above case of singleton tuples):

*# Correct:*

FILES = [

'setup.cfg',

'tox.ini',

]

initialize(FILES,

error=**True**,

)

*# Wrong:*

FILES = ['setup.cfg', 'tox.ini',]

initialize(FILES, error=**True**,)

[**Comments**](https://peps.python.org/pep-0008/#comments)

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.

You should use one or two spaces after a sentence-ending period in multi-sentence comments, except after the final sentence.

Ensure that your comments are clear and easily understandable to other speakers of the language you are writing in.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don’t speak your language.

[Block Comments](https://peps.python.org/pep-0008/#block-comments)

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

[Inline Comments](https://peps.python.org/pep-0008/#inline-comments)

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don’t do this:

x = x + 1 *# Increment x*

But sometimes, this is useful:

x = x + 1 *# Compensate for border*

[Documentation Strings](https://peps.python.org/pep-0008/#documentation-strings)

Conventions for writing good documentation strings (a.k.a. “docstrings”) are immortalized in [PEP 257](https://peps.python.org/pep-0257/).

* Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.
* [PEP 257](https://peps.python.org/pep-0257/) describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself:
* """Return a foobang
* Optional plotz says to frobnicate the bizbaz first.
* """
* For one liner docstrings, please keep the closing """ on the same line:
* """Return an ex-parrot."""

[**Naming Conventions**](https://peps.python.org/pep-0008/#naming-conventions)

The naming conventions of Python’s library are a bit of a mess, so we’ll never get this completely consistent – nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

[Overriding Principle](https://peps.python.org/pep-0008/#overriding-principle)

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

[Descriptive: Naming Styles](https://peps.python.org/pep-0008/#descriptive-naming-styles)

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

* b (single lowercase letter)
* B (single uppercase letter)
* lowercase
* lower\_case\_with\_underscores
* UPPERCASE
* UPPER\_CASE\_WITH\_UNDERSCORES
* CapitalizedWords (or CapWords, or CamelCase – so named because of the bumpy look of its letters [[4]](https://peps.python.org/pep-0008/#id8)). This is also sometimes known as StudlyCaps.

Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.

* mixedCase (differs from CapitalizedWords by initial lowercase character!)
* Capitalized\_Words\_With\_Underscores (ugly!)

There’s also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st\_mode, st\_size, st\_mtime and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

* \_single\_leading\_underscore: weak “internal use” indicator. E.g. from M import \* does not import objects whose names start with an underscore.
* single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g. :
* tkinter.Toplevel(master, class\_='ClassName')
* \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).
* \_\_double\_leading\_and\_trailing\_underscore\_\_: “magic” objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.

[Prescriptive: Naming Conventions](https://peps.python.org/pep-0008/#prescriptive-naming-conventions)

[Names to Avoid](https://peps.python.org/pep-0008/#names-to-avoid)

Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use ‘l’, use ‘L’ instead.

[ASCII Compatibility](https://peps.python.org/pep-0008/#ascii-compatibility)

Identifiers used in the standard library must be ASCII compatible as described in the [policy section](https://peps.python.org/pep-3131/#policy-specification) of [PEP 3131](https://peps.python.org/pep-3131/).

[Package and Module Names](https://peps.python.org/pep-0008/#package-and-module-names)

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. \_socket).

[Class Names](https://peps.python.org/pep-0008/#class-names)

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

[Type Variable Names](https://peps.python.org/pep-0008/#type-variable-names)

Names of type variables introduced in [PEP 484](https://peps.python.org/pep-0484/) should normally use CapWords preferring short names: T, AnyStr, Num. It is recommended to add suffixes \_co or \_contra to the variables used to declare covariant or contravariant behavior correspondingly:

**from** typing **import** TypeVar

VT\_co = TypeVar('VT\_co', covariant=**True**)

KT\_contra = TypeVar('KT\_contra', contravariant=**True**)

[Exception Names](https://peps.python.org/pep-0008/#exception-names)

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix “Error” on your exception names (if the exception actually is an error).

[Global Variable Names](https://peps.python.org/pep-0008/#global-variable-names)

(Let’s hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

Modules that are designed for use via from M import \* should use the \_\_all\_\_ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are “module non-public”).

[Function and Variable Names](https://peps.python.org/pep-0008/#function-and-variable-names)

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

mixedCase is allowed only in contexts where that’s already the prevailing style (e.g. threading.py), to retain backwards compatibility.

[Function and Method Arguments](https://peps.python.org/pep-0008/#function-and-method-arguments)

Always use self for the first argument to instance methods.

Always use cls for the first argument to class methods.

If a function argument’s name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class\_ is better than clss. (Perhaps better is to avoid such clashes by using a synonym.)

[Method Names and Instance Variables](https://peps.python.org/pep-0008/#method-names-and-instance-variables)

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python’s name mangling rules.

Python mangles these names with the class name: if class Foo has an attribute named \_\_a, it cannot be accessed by Foo.\_\_a. (An insistent user could still gain access by calling Foo.\_Foo\_\_a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of \_\_names (see below).

[Constants](https://peps.python.org/pep-0008/#constants)

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX\_OVERFLOW and TOTAL.

[Designing for Inheritance](https://peps.python.org/pep-0008/#designing-for-inheritance)

Always decide whether a class’s methods and instance variables (collectively: “attributes”) should be public or non-public. If in doubt, choose non-public; it’s easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won’t change or even be removed.

We don’t use the term “private” here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the “subclass API” (often called “protected” in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class’s behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

* Public attributes should have no leading underscores.
* If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, ‘cls’ is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

* For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 2: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

* If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python’s name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and \_\_getattr\_\_(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

[Public and Internal Interfaces](https://peps.python.org/pep-0008/#public-and-internal-interfaces)

Any backwards compatibility guarantees apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public and internal interfaces.

Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from the usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.

To better support introspection, modules should explicitly declare the names in their public API using the \_\_all\_\_ attribute. Setting \_\_all\_\_ to an empty list indicates that the module has no public API.

Even with \_\_all\_\_ set appropriately, internal interfaces (packages, modules, classes, functions, attributes or other names) should still be prefixed with a single leading underscore.

An interface is also considered internal if any containing namespace (package, module or class) is considered internal.

Imported names should always be considered an implementation detail. Other modules must not rely on indirect access to such imported names unless they are an explicitly documented part of the containing module’s API, such as os.path or a package’s \_\_init\_\_ module that exposes functionality from submodules.

[**Programming Recommendations**](https://peps.python.org/pep-0008/#programming-recommendations)

* Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).

For example, do not rely on CPython’s efficient implementation of in-place string concatenation for statements in the form a += b or a = a + b. This optimization is fragile even in CPython (it only works for some types) and isn’t present at all in implementations that don’t use refcounting. In performance sensitive parts of the library, the ''.join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementations.

* Comparisons to singletons like None should always be done with is or is not, never the equality operators.

Also, beware of writing if x when you really mean if x is not None – e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

* Use is not operator rather than not ... is. While both expressions are functionally identical, the former is more readable and preferred:
* *# Correct:*
* **if** foo **is** **not** **None**:
* *# Wrong:*
* **if** **not** foo **is** **None**:
* When implementing ordering operations with rich comparisons, it is best to implement all six operations (\_\_eq\_\_, \_\_ne\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_) rather than relying on other code to only exercise a particular comparison.

To minimize the effort involved, the functools.total\_ordering() decorator provides a tool to generate missing comparison methods.

[PEP 207](https://peps.python.org/pep-0207/) indicates that reflexivity rules *are* assumed by Python. Thus, the interpreter may swap y > x with x < y, y >= x with x <= y, and may swap the arguments of x == y and x != y. The sort() and min() operations are guaranteed to use the < operator and the max() function uses the > operator. However, it is best to implement all six operations so that confusion doesn’t arise in other contexts.

* Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier:
* *# Correct:*
* **def** f(x): **return** 2\*x
* *# Wrong:*
* f = **lambda** x: 2\*x

The first form means that the name of the resulting function object is specifically ‘f’ instead of the generic ‘<lambda>’. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

* Derive exceptions from Exception rather than BaseException. Direct inheritance from BaseException is reserved for exceptions where catching them is almost always the wrong thing to do.

Design exception hierarchies based on the distinctions that code *catching* the exceptions is likely to need, rather than the locations where the exceptions are raised. Aim to answer the question “What went wrong?” programmatically, rather than only stating that “A problem occurred” (see [PEP 3151](https://peps.python.org/pep-3151/) for an example of this lesson being learned for the builtin exception hierarchy)

Class naming conventions apply here, although you should add the suffix “Error” to your exception classes if the exception is an error. Non-error exceptions that are used for non-local flow control or other forms of signaling need no special suffix.

* Use exception chaining appropriately. raise X from Y should be used to indicate explicit replacement without losing the original traceback.

When deliberately replacing an inner exception (using raise X from None), ensure that relevant details are transferred to the new exception (such as preserving the attribute name when converting KeyError to AttributeError, or embedding the text of the original exception in the new exception message).

* When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause:
* **try**:
* **import** platform\_specific\_module
* **except** ImportError:
* platform\_specific\_module = **None**

A bare except: clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use except Exception: (bare except is equivalent to except BaseException:).

A good rule of thumb is to limit use of bare ‘except’ clauses to two cases:

* 1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.
  2. If the code needs to do some cleanup work, but then lets the exception propagate upwards with raise. try...finally can be a better way to handle this case.
* When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of errno values.
* Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs:
* *# Correct:*
* **try**:
* value = collection[key]
* **except** KeyError:
* **return** key\_not\_found(key)
* **else**:
* **return** handle\_value(value)
* *# Wrong:*
* **try**:
* *# Too broad!*
* **return** handle\_value(collection[key])
* **except** KeyError:
* *# Will also catch KeyError raised by handle\_value()*
* **return** key\_not\_found(key)
* When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.
* Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources:
* *# Correct:*
* **with** conn.begin\_transaction():
* do\_stuff\_in\_transaction(conn)
* *# Wrong:*
* **with** conn:
* do\_stuff\_in\_transaction(conn)

The latter example doesn’t provide any information to indicate that the \_\_enter\_\_ and \_\_exit\_\_ methods are doing something other than closing the connection after a transaction. Being explicit is important in this case.

* Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable):
* *# Correct:*
* **def** foo(x):
* **if** x >= 0:
* **return** math.sqrt(x)
* **else**:
* **return** **None**
* **def** bar(x):
* **if** x < 0:
* **return** **None**
* **return** math.sqrt(x)
* *# Wrong:*
* **def** foo(x):
* **if** x >= 0:
* **return** math.sqrt(x)
* **def** bar(x):
* **if** x < 0:
* **return**
* **return** math.sqrt(x)
* Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

startswith() and endswith() are cleaner and less error prone:

*# Correct:*

**if** foo.startswith('bar'):

*# Wrong:*

**if** foo[:3] == 'bar':

* Object type comparisons should always use isinstance() instead of comparing types directly:
* *# Correct:*
* **if** isinstance(obj, int):
* *# Wrong:*
* **if** type(obj) **is** type(1):
* For sequences, (strings, lists, tuples), use the fact that empty sequences are false:
* *# Correct:*
* **if** **not** seq:
* **if** seq:
* *# Wrong:*
* **if** len(seq):
* **if** **not** len(seq):
* Don’t write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them.
* Don’t compare boolean values to True or False using ==:
* *# Correct:*
* **if** greeting:
* *# Wrong:*
* **if** greeting == **True**:

Worse:

*# Wrong:*

**if** greeting **is** **True**:

* Use of the flow control statements return/break/continue within the finally suite of a try...finally, where the flow control statement would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite:
* *# Wrong:*
* **def** foo():
* **try**:
* 1 / 0
* **finally**:
* **return** 42

[Function Annotations](https://peps.python.org/pep-0008/#function-annotations)

With the acceptance of [PEP 484](https://peps.python.org/pep-0484/), the style rules for function annotations have changed.

* Function annotations should use [PEP 484](https://peps.python.org/pep-0484/) syntax (there are some formatting recommendations for annotations in the previous section).
* The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.
* However, outside the stdlib, experiments within the rules of [PEP 484](https://peps.python.org/pep-0484/) are now encouraged. For example, marking up a large third party library or application with [PEP 484](https://peps.python.org/pep-0484/) style type annotations, reviewing how easy it was to add those annotations, and observing whether their presence increases code understandability.
* The Python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refactorings.
* For code that wants to make a different use of function annotations it is recommended to put a comment of the form:
* *# type: ignore*

near the top of the file; this tells type checkers to ignore all annotations. (More fine-grained ways of disabling complaints from type checkers can be found in [PEP 484](https://peps.python.org/pep-0484/).)

* Like linters, type checkers are optional, separate tools. Python interpreters by default should not issue any messages due to type checking and should not alter their behavior based on annotations.
* Users who don’t want to use type checkers are free to ignore them. However, it is expected that users of third party library packages may want to run type checkers over those packages. For this purpose [PEP 484](https://peps.python.org/pep-0484/) recommends the use of stub files: .pyi files that are read by the type checker in preference of the corresponding .py files. Stub files can be distributed with a library, or separately (with the library author’s permission) through the typeshed repo [[5]](https://peps.python.org/pep-0008/#id9).

[Variable Annotations](https://peps.python.org/pep-0008/#variable-annotations)

[PEP 526](https://peps.python.org/pep-0526/) introduced variable annotations. The style recommendations for them are similar to those on function annotations described above:

* Annotations for module level variables, class and instance variables, and local variables should have a single space after the colon.
* There should be no space before the colon.
* If an assignment has a right hand side, then the equality sign should have exactly one space on both sides:
* *# Correct:*
* code: int
* **class** Point:
* coords: Tuple[int, int]
* label: str = '<unknown>'
* *# Wrong:*
* code:int *# No space after colon*
* code : int *# Space before colon*
* **class** Test:
* result: int=0 *# No spaces around equality sign*

Javascript Coding Standards

This is a summary of the [standard](https://github.com/standard/standard) JavaScript rules.

The best way to learn about standard is to just install it and give it a try on your code.

[**Rules**](https://standardjs.com/rules.html#rules)

* **Use 2 spaces** for indentation.

eslint: [indent](http://eslint.org/docs/rules/indent)

function hello (name) {

console.log('hi', name)

}

* **Use single quotes for strings** except to avoid escaping.

eslint: [quotes](http://eslint.org/docs/rules/quotes)

console.log('hello there') // ✓ ok

console.log("hello there") // ✗ avoid

console.log(`hello there`) // ✗ avoid

$("<div class='box'>") // ✓ ok

console.log(`hello ${name}`) // ✓ ok

* **No unused variables.**

eslint: [no-unused-vars](http://eslint.org/docs/rules/no-unused-vars)

function myFunction () {

var result = something() // ✗ avoid

}

* **Add a space after keywords.**

eslint: [keyword-spacing](http://eslint.org/docs/rules/keyword-spacing)

if (condition) { ... } // ✓ ok

if(condition) { ... } // ✗ avoid

* **Add a space before a function declaration's parentheses.**

eslint: [space-before-function-paren](http://eslint.org/docs/rules/space-before-function-paren)

function name (arg) { ... } // ✓ ok

function name(arg) { ... } // ✗ avoid

run(function () { ... }) // ✓ ok

run(function() { ... }) // ✗ avoid

* **Always use** === instead of ==.  
  Exception: obj == null is allowed to check for null || undefined.

eslint: [eqeqeq](http://eslint.org/docs/rules/eqeqeq)

if (name === 'John') // ✓ ok

if (name == 'John') // ✗ avoid

if (name !== 'John') // ✓ ok

if (name != 'John') // ✗ avoid

* **Infix operators** must be spaced.

eslint: [space-infix-ops](http://eslint.org/docs/rules/space-infix-ops)

// ✓ ok

var x = 2

var message = 'hello, ' + name + '!'

// ✗ avoid

var x=2

var message = 'hello, '+name+'!'

* **Commas should have a space** after them.

eslint: [comma-spacing](http://eslint.org/docs/rules/comma-spacing)

// ✓ ok

var list = [1, 2, 3, 4]

function greet (name, options) { ... }

// ✗ avoid

var list = [1,2,3,4]

function greet (name,options) { ... }

* **Keep else statements** on the same line as their curly braces.

eslint: [brace-style](http://eslint.org/docs/rules/brace-style)

// ✓ ok

if (condition) {

// ...

} else {

// ...

}

// ✗ avoid

if (condition) {

// ...

}

else {

// ...

}

* **For multi-line if statements,** use curly braces.

eslint: [curly](http://eslint.org/docs/rules/curly)

// ✓ ok

if (options.quiet !== true) console.log('done')

// ✓ ok

if (options.quiet !== true) {

console.log('done')

}

// ✗ avoid

if (options.quiet !== true)

console.log('done')

* **Always handle the** err function parameter.

eslint: [handle-callback-err](http://eslint.org/docs/rules/handle-callback-err)

// ✓ ok

run(function (err) {

if (err) throw err

window.alert('done')

})

// ✗ avoid

run(function (err) {

window.alert('done')

})

* **Declare browser globals** with a /\* global \*/ comment.  
  Exceptions are: window, document, and navigator.  
  Prevents accidental use of poorly-named browser globals like open, length, event, and name.
* /\* global alert, prompt \*/
* alert('hi')
* prompt('ok?')

Explicitly referencing the function or property on window is okay too, though such code will not run in a Worker which uses self instead of window.

eslint: [no-undef](http://eslint.org/docs/rules/no-undef)

window.alert('hi') // ✓ ok

* **Multiple blank lines not allowed.**

eslint: [no-multiple-empty-lines](http://eslint.org/docs/rules/no-multiple-empty-lines)

// ✓ ok

var value = 'hello world'

console.log(value)

// ✗ avoid

var value = 'hello world'

// blank line

// blank line

console.log(value)

* **For the ternary operator** in a multi-line setting, place ? and : on their own lines.

eslint: [operator-linebreak](http://eslint.org/docs/rules/operator-linebreak)

// ✓ ok

var location = env.development ? 'localhost' : 'www.api.com'

// ✓ ok

var location = env.development

? 'localhost'

: 'www.api.com'

// ✗ avoid

var location = env.development ?

'localhost' :

'www.api.com'

* **For var declarations,** write each declaration in its own statement.

eslint: [one-var](http://eslint.org/docs/rules/one-var)

// ✓ ok

var silent = true

var verbose = true

// ✗ avoid

var silent = true, verbose = true

// ✗ avoid

var silent = true,

verbose = true

* **Wrap conditional assignments** with additional parentheses. This makes it clear that the expression is intentionally an assignment (=) rather than a typo for equality (===).

eslint: [no-cond-assign](http://eslint.org/docs/rules/no-cond-assign)

// ✓ ok

while ((m = text.match(expr))) {

// ...

}

// ✗ avoid

while (m = text.match(expr)) {

// ...

}

* **Add spaces inside single line blocks.**

eslint: [block-spacing](http://eslint.org/docs/rules/block-spacing)

function foo () {return true} // ✗ avoid

function foo () { return true } // ✓ ok

* **Use camelcase when naming variables and functions.**

eslint: [camelcase](http://eslint.org/docs/rules/camelcase)

function my\_function () { } // ✗ avoid

function myFunction () { } // ✓ ok

var my\_var = 'hello' // ✗ avoid

var myVar = 'hello' // ✓ ok

* **Trailing commas not allowed.**

eslint: [comma-dangle](http://eslint.org/docs/rules/comma-dangle)

var obj = {

message: 'hello', // ✗ avoid

}

* **Commas must be placed at the end of the current line.**

eslint: [comma-style](http://eslint.org/docs/rules/comma-style)

var obj = {

foo: 'foo'

,bar: 'bar' // ✗ avoid

}

var obj = {

foo: 'foo',

bar: 'bar' // ✓ ok

}

* **Dot should be on the same line as property.**

eslint: [dot-location](http://eslint.org/docs/rules/dot-location)

console.

log('hello') // ✗ avoid

console

.log('hello') // ✓ ok

* **Files must end with a newline.**

eslint: [eol-last](http://eslint.org/docs/rules/eol-last)

* **No space between function identifiers and their invocations.**

eslint: [func-call-spacing](http://eslint.org/docs/rules/func-call-spacing)

console.log ('hello') // ✗ avoid

console.log('hello') // ✓ ok

* **Add space between colon and value in key value pairs.**

eslint: [key-spacing](http://eslint.org/docs/rules/key-spacing)

var obj = { 'key' : 'value' } // ✗ avoid

var obj = { 'key' :'value' } // ✗ avoid

var obj = { 'key':'value' } // ✗ avoid

var obj = { 'key': 'value' } // ✓ ok

* **Constructor names must begin with a capital letter.**

eslint: [new-cap](http://eslint.org/docs/rules/new-cap)

function animal () {}

var dog = new animal() // ✗ avoid

function Animal () {}

var dog = new Animal() // ✓ ok

* **Constructor with no arguments must be invoked with parentheses.**

eslint: [new-parens](http://eslint.org/docs/rules/new-parens)

function Animal () {}

var dog = new Animal // ✗ avoid

var dog = new Animal() // ✓ ok

* **Objects must contain a getter when a setter is defined.**

eslint: [accessor-pairs](http://eslint.org/docs/rules/accessor-pairs)

var person = {

set name (value) { // ✗ avoid

this.\_name = value

}

}

var person = {

set name (value) {

this.\_name = value

},

get name () { // ✓ ok

return this.\_name

}

}

* **Constructors of derived classes must call super.**

eslint: [constructor-super](http://eslint.org/docs/rules/constructor-super)

class Dog {

constructor () {

super() // ✗ avoid

this.legs = 4

}

}

class Dog extends Animal {

constructor () { // ✗ avoid

this.legs = 4

}

}

class Dog extends Animal {

constructor () {

super() // ✓ ok

this.legs = 4

}

}

* **Use array literals instead of array constructors.**

eslint: [no-array-constructor](http://eslint.org/docs/rules/no-array-constructor)

var nums = new Array(1, 2, 3) // ✗ avoid

var nums = [1, 2, 3] // ✓ ok

* **Avoid using arguments.callee and arguments.caller.**

eslint: [no-caller](http://eslint.org/docs/rules/no-caller)

function foo (n) {

if (n <= 0) return

arguments.callee(n - 1) // ✗ avoid

}

function foo (n) {

if (n <= 0) return

foo(n - 1) // ✓ ok

}

* **Avoid modifying variables of class declarations.**

eslint: [no-class-assign](http://eslint.org/docs/rules/no-class-assign)

class Dog {}

Dog = 'Fido' // ✗ avoid

* **Avoid modifying variables declared using const.**

eslint: [no-const-assign](http://eslint.org/docs/rules/no-const-assign)

const score = 100

score = 125 // ✗ avoid

* **Avoid using constant expressions in conditions (except loops).**

eslint: [no-constant-condition](http://eslint.org/docs/rules/no-constant-condition)

if (false) { // ✗ avoid

// ...

}

if (x === 0) { // ✓ ok

// ...

}

while (true) { // ✓ ok

// ...

}

* **No control characters in regular expressions.**

eslint: [no-control-regex](http://eslint.org/docs/rules/no-control-regex)

var pattern = /\x1f/ // ✗ avoid

var pattern = /\x20/ // ✓ ok

* **No debugger statements.**

eslint: [no-debugger](http://eslint.org/docs/rules/no-debugger)

function sum (a, b) {

debugger // ✗ avoid

return a + b

}

* **No delete operator on variables.**

eslint: [no-delete-var](http://eslint.org/docs/rules/no-delete-var)

var name

delete name // ✗ avoid

* **No duplicate arguments in function definitions.**

eslint: [no-dupe-args](http://eslint.org/docs/rules/no-dupe-args)

function sum (a, b, a) { // ✗ avoid

// ...

}

function sum (a, b, c) { // ✓ ok

// ...

}

* **No duplicate name in class members.**

eslint: [no-dupe-class-members](http://eslint.org/docs/rules/no-dupe-class-members)

class Dog {

bark () {}

bark () {} // ✗ avoid

}

* **No duplicate keys in object literals.**

eslint: [no-dupe-keys](http://eslint.org/docs/rules/no-dupe-keys)

var user = {

name: 'Jane Doe',

name: 'John Doe' // ✗ avoid

}

* **No duplicate case labels in switch statements.**

eslint: [no-duplicate-case](http://eslint.org/docs/rules/no-duplicate-case)

switch (id) {

case 1:

// ...

case 1: // ✗ avoid

}

* **Use a single import statement per module.**

eslint: [no-duplicate-imports](http://eslint.org/docs/rules/no-duplicate-imports)

import { myFunc1 } from 'module'

import { myFunc2 } from 'module' // ✗ avoid

import { myFunc1, myFunc2 } from 'module' // ✓ ok

* **No empty character classes in regular expressions.**

eslint: [no-empty-character-class](http://eslint.org/docs/rules/no-empty-character-class)

const myRegex = /^abc[]/ // ✗ avoid

const myRegex = /^abc[a-z]/ // ✓ ok

* **No empty destructuring patterns.**

eslint: [no-empty-pattern](http://eslint.org/docs/rules/no-empty-pattern)

const { a: {} } = foo // ✗ avoid

const { a: { b } } = foo // ✓ ok

* **No using eval().**

eslint: [no-eval](http://eslint.org/docs/rules/no-eval)

eval( "var result = user." + propName ) // ✗ avoid

var result = user[propName] // ✓ ok

* **No reassigning exceptions in catch clauses.**

eslint: [no-ex-assign](http://eslint.org/docs/rules/no-ex-assign)

try {

// ...

} catch (e) {

e = 'new value' // ✗ avoid

}

try {

// ...

} catch (e) {

const newVal = 'new value' // ✓ ok

}

* **No extending native objects.**

eslint: [no-extend-native](http://eslint.org/docs/rules/no-extend-native)

Object.prototype.age = 21 // ✗ avoid

* **Avoid unnecessary function binding.**

eslint: [no-extra-bind](http://eslint.org/docs/rules/no-extra-bind)

const name = function () {

getName()

}.bind(user) // ✗ avoid

const name = function () {

this.getName()

}.bind(user) // ✓ ok

* **Avoid unnecessary boolean casts.**

eslint: [no-extra-boolean-cast](http://eslint.org/docs/rules/no-extra-boolean-cast)

const result = true

if (!!result) { // ✗ avoid

// ...

}

const result = true

if (result) { // ✓ ok

// ...

}

* **No unnecessary parentheses around function expressions.**

eslint: [no-extra-parens](http://eslint.org/docs/rules/no-extra-parens)

const myFunc = (function () { }) // ✗ avoid

const myFunc = function () { } // ✓ ok

* **Use break to prevent fallthrough in switch cases.**

eslint: [no-fallthrough](http://eslint.org/docs/rules/no-fallthrough)

switch (filter) {

case 1:

doSomething() // ✗ avoid

case 2:

doSomethingElse()

}

switch (filter) {

case 1:

doSomething()

break // ✓ ok

case 2:

doSomethingElse()

}

switch (filter) {

case 1:

doSomething()

// fallthrough // ✓ ok

case 2:

doSomethingElse()

}

* **No floating decimals.**

eslint: [no-floating-decimal](http://eslint.org/docs/rules/no-floating-decimal)

const discount = .5 // ✗ avoid

const discount = 0.5 // ✓ ok

* **Avoid reassigning function declarations.**

eslint: [no-func-assign](http://eslint.org/docs/rules/no-func-assign)

function myFunc () { }

myFunc = myOtherFunc // ✗ avoid

* **No reassigning read-only global variables.**

eslint: [no-global-assign](http://eslint.org/docs/rules/no-global-assign)

window = {} // ✗ avoid

* **No implied eval().**

eslint: [no-implied-eval](http://eslint.org/docs/rules/no-implied-eval)

setTimeout("alert('Hello world')") // ✗ avoid

setTimeout(function () { alert('Hello world') }) // ✓ ok

* **No function declarations in nested blocks.**

eslint: [no-inner-declarations](http://eslint.org/docs/rules/no-inner-declarations)

if (authenticated) {

function setAuthUser () {} // ✗ avoid

}

* **No invalid regular expression strings in RegExp constructors.**

eslint: [no-invalid-regexp](http://eslint.org/docs/rules/no-invalid-regexp)

RegExp('[a-z') // ✗ avoid

RegExp('[a-z]') // ✓ ok

* **No irregular whitespace.**

eslint: [no-irregular-whitespace](http://eslint.org/docs/rules/no-irregular-whitespace)

function myFunc () /\*<NBSP>\*/{} // ✗ avoid

* **No using \_\_iterator\_\_.**

eslint: [no-iterator](http://eslint.org/docs/rules/no-iterator)

Foo.prototype.\_\_iterator\_\_ = function () {} // ✗ avoid

* **No labels that share a name with an in scope variable.**

eslint: [no-label-var](http://eslint.org/docs/rules/no-label-var)

var score = 100

function game () {

score: while (true) { // ✗ avoid

score -= 10

if (score > 0) continue score

break

}

}

* **No label statements.**

eslint: [no-labels](http://eslint.org/docs/rules/no-labels)

label:

while (true) {

break label // ✗ avoid

}

* **No unnecessary nested blocks.**

eslint: [no-lone-blocks](http://eslint.org/docs/rules/no-lone-blocks)

function myFunc () {

{ // ✗ avoid

myOtherFunc()

}

}

function myFunc () {

myOtherFunc() // ✓ ok

}

* **Avoid mixing spaces and tabs for indentation.**

eslint: [no-mixed-spaces-and-tabs](http://eslint.org/docs/rules/no-mixed-spaces-and-tabs)

* **Do not use multiple spaces except for indentation.**

eslint: [no-multi-spaces](http://eslint.org/docs/rules/no-multi-spaces)

const id = 1234 // ✗ avoid

const id = 1234 // ✓ ok

* **No multiline strings.**

eslint: [no-multi-str](http://eslint.org/docs/rules/no-multi-str)

const message = 'Hello \

world' // ✗ avoid

* **No new without assigning object to a variable.**

eslint: [no-new](http://eslint.org/docs/rules/no-new)

new Character() // ✗ avoid

const character = new Character() // ✓ ok

* **No using the Function constructor.**

eslint: [no-new-func](http://eslint.org/docs/rules/no-new-func)

var sum = new Function('a', 'b', 'return a + b') // ✗ avoid

* **No using the Object constructor.**

eslint: [no-new-object](http://eslint.org/docs/rules/no-new-object)

let config = new Object() // ✗ avoid

* **No using new require.**

eslint: [no-new-require](http://eslint.org/docs/rules/no-new-require)

const myModule = new require('my-module') // ✗ avoid

* **No using the Symbol constructor.**

eslint: [no-new-symbol](http://eslint.org/docs/rules/no-new-symbol)

const foo = new Symbol('foo') // ✗ avoid

* **No using primitive wrapper instances.**

eslint: [no-new-wrappers](http://eslint.org/docs/rules/no-new-wrappers)

const message = new String('hello') // ✗ avoid

* **No calling global object properties as functions.**

eslint: [no-obj-calls](http://eslint.org/docs/rules/no-obj-calls)

const math = Math() // ✗ avoid

* **No octal literals.**

eslint: [no-octal](http://eslint.org/docs/rules/no-octal)

const octal = 042 // ✗ avoid

const decimal = 34 // ✓ ok

const octalString = '042' // ✓ ok

* **No octal escape sequences in string literals.**

eslint: [no-octal-escape](http://eslint.org/docs/rules/no-octal-escape)

const copyright = 'Copyright \251' // ✗ avoid

* **Avoid string concatenation when using \_\_dirname and \_\_filename.**

eslint: [no-path-concat](http://eslint.org/docs/rules/no-path-concat)

const pathToFile = \_\_dirname + '/app.js' // ✗ avoid

const pathToFile = path.join(\_\_dirname, 'app.js') // ✓ ok

* **Avoid using \_\_proto\_\_.** Use getPrototypeOf instead.

eslint: [no-proto](http://eslint.org/docs/rules/no-proto)

const foo = obj.\_\_proto\_\_ // ✗ avoid

const foo = Object.getPrototypeOf(obj) // ✓ ok

* **No redeclaring variables.**

eslint: [no-redeclare](http://eslint.org/docs/rules/no-redeclare)

let name = 'John'

let name = 'Jane' // ✗ avoid

let name = 'John'

name = 'Jane' // ✓ ok

* **Avoid multiple spaces in regular expression literals.**

eslint: [no-regex-spaces](http://eslint.org/docs/rules/no-regex-spaces)

const regexp = /test value/ // ✗ avoid

const regexp = /test {3}value/ // ✓ ok

const regexp = /test value/ // ✓ ok

* **Assignments in return statements must be surrounded by parentheses.**

eslint: [no-return-assign](http://eslint.org/docs/rules/no-return-assign)

function sum (a, b) {

return result = a + b // ✗ avoid

}

function sum (a, b) {

return (result = a + b) // ✓ ok

}

* **Avoid assigning a variable to itself**

eslint: [no-self-assign](http://eslint.org/docs/rules/no-self-assign)

name = name // ✗ avoid

* **Avoid comparing a variable to itself.**

eslint: [no-self-compare](http://eslint.org/docs/rules/no-self-compare)

if (score === score) {} // ✗ avoid

* **Avoid using the comma operator.**

eslint: [no-sequences](http://eslint.org/docs/rules/no-sequences)

if (doSomething(), !!test) {} // ✗ avoid

* **Restricted names should not be shadowed.**

eslint: [no-shadow-restricted-names](http://eslint.org/docs/rules/no-shadow-restricted-names)

let undefined = 'value' // ✗ avoid

* **Sparse arrays are not allowed.**

eslint: [no-sparse-arrays](http://eslint.org/docs/rules/no-sparse-arrays)

let fruits = ['apple',, 'orange'] // ✗ avoid

* **Tabs should not be used**

eslint: [no-tabs](http://eslint.org/docs/rules/no-tabs)

* **Regular strings must not contain template literal placeholders.**

eslint: [no-template-curly-in-string](http://eslint.org/docs/rules/no-template-curly-in-string)

const message = 'Hello ${name}' // ✗ avoid

const message = `Hello ${name}` // ✓ ok

* **super() must be called before using this.**

eslint: [no-this-before-super](http://eslint.org/docs/rules/no-this-before-super)

class Dog extends Animal {

constructor () {

this.legs = 4 // ✗ avoid

super()

}

}

* **Only throw an Error object.**

eslint: [no-throw-literal](http://eslint.org/docs/rules/no-throw-literal)

throw 'error' // ✗ avoid

throw new Error('error') // ✓ ok

* **Whitespace not allowed at end of line.**

eslint: [no-trailing-spaces](http://eslint.org/docs/rules/no-trailing-spaces)

* **Initializing to undefined is not allowed.**

eslint: [no-undef-init](http://eslint.org/docs/rules/no-undef-init)

let name = undefined // ✗ avoid

let name

name = 'value' // ✓ ok

* **No unmodified conditions of loops.**

eslint: [no-unmodified-loop-condition](http://eslint.org/docs/rules/no-unmodified-loop-condition)

for (let i = 0; i < items.length; j++) {...} // ✗ avoid

for (let i = 0; i < items.length; i++) {...} // ✓ ok

* **No ternary operators when simpler alternatives exist.**

eslint: [no-unneeded-ternary](http://eslint.org/docs/rules/no-unneeded-ternary)

let score = val ? val : 0 // ✗ avoid

let score = val || 0 // ✓ ok

* **No unreachable code after return, throw, continue, and break statements.**

eslint: [no-unreachable](http://eslint.org/docs/rules/no-unreachable)

function doSomething () {

return true

console.log('never called') // ✗ avoid

}

* **No flow control statements in finally blocks.**

eslint: [no-unsafe-finally](http://eslint.org/docs/rules/no-unsafe-finally)

try {

// ...

} catch (e) {

// ...

} finally {

return 42 // ✗ avoid

}

* **The left operand of relational operators must not be negated.**

eslint: [no-unsafe-negation](http://eslint.org/docs/rules/no-unsafe-negation)

if (!key in obj) {} // ✗ avoid

if (!(key in obj)) {} // ✓ ok

* **Avoid unnecessary use of .call() and .apply().**

eslint: [no-useless-call](http://eslint.org/docs/rules/no-useless-call)

sum.call(null, 1, 2, 3) // ✗ avoid

* **Avoid using unnecessary computed property keys on objects.**

eslint: [no-useless-computed-key](http://eslint.org/docs/rules/no-useless-computed-key)

const user = { ['name']: 'John Doe' } // ✗ avoid

const user = { name: 'John Doe' } // ✓ ok

* **No unnecessary constructor.**

eslint: [no-useless-constructor](http://eslint.org/docs/rules/no-useless-constructor)

class Car {

constructor () { // ✗ avoid

}

}

* **No unnecessary use of escape.**

eslint: [no-useless-escape](http://eslint.org/docs/rules/no-useless-escape)

let message = 'Hell\o' // ✗ avoid

* **Renaming import, export, and destructured assignments to the same name is not allowed.**

eslint: [no-useless-rename](http://eslint.org/docs/rules/no-useless-rename)

import { config as config } from './config' // ✗ avoid

import { config } from './config' // ✓ ok

* **No whitespace before properties.**

eslint: [no-whitespace-before-property](http://eslint.org/docs/rules/no-whitespace-before-property)

user .name // ✗ avoid

user.name // ✓ ok

* **No using with statements.**

eslint: [no-with](http://eslint.org/docs/rules/no-with)

with (val) {...} // ✗ avoid

* **Maintain consistency of newlines between object properties.**

eslint: [object-property-newline](http://eslint.org/docs/rules/object-property-newline)

const user = {

name: 'Jane Doe', age: 30,

username: 'jdoe86' // ✗ avoid

}

const user = { name: 'Jane Doe', age: 30, username: 'jdoe86' } // ✓ ok

const user = {

name: 'Jane Doe',

age: 30,

username: 'jdoe86'

} // ✓ ok

* **No padding within blocks.**

eslint: [padded-blocks](http://eslint.org/docs/rules/padded-blocks)

if (user) {

// ✗ avoid

const name = getName()

}

if (user) {

const name = getName() // ✓ ok

}

* **No whitespace between spread operators and their expressions.**

eslint: [rest-spread-spacing](http://eslint.org/docs/rules/rest-spread-spacing)

fn(... args) // ✗ avoid

fn(...args) // ✓ ok

* **Semicolons must have a space after and no space before.**

eslint: [semi-spacing](http://eslint.org/docs/rules/semi-spacing)

for (let i = 0 ;i < items.length ;i++) {...} // ✗ avoid

for (let i = 0; i < items.length; i++) {...} // ✓ ok

* **Must have a space before blocks.**

eslint: [space-before-blocks](http://eslint.org/docs/rules/space-before-blocks)

if (admin){...} // ✗ avoid

if (admin) {...} // ✓ ok

* **No spaces inside parentheses.**

eslint: [space-in-parens](http://eslint.org/docs/rules/space-in-parens)

getName( name ) // ✗ avoid

getName(name) // ✓ ok

* **Unary operators must have a space after.**

eslint: [space-unary-ops](http://eslint.org/docs/rules/space-unary-ops)

typeof!admin // ✗ avoid

typeof !admin // ✓ ok

* **Use spaces inside comments.**

eslint: [spaced-comment](http://eslint.org/docs/rules/spaced-comment)

//comment // ✗ avoid

// comment // ✓ ok

/\*comment\*/ // ✗ avoid

/\* comment \*/ // ✓ ok

* **No spacing in template strings.**

eslint: [template-curly-spacing](http://eslint.org/docs/rules/template-curly-spacing)

const message = `Hello, ${ name }` // ✗ avoid

const message = `Hello, ${name}` // ✓ ok

* **Use isNaN() when checking for NaN.**

eslint: [use-isnan](http://eslint.org/docs/rules/use-isnan)

if (price === NaN) { } // ✗ avoid

if (isNaN(price)) { } // ✓ ok

* **typeof must be compared to a valid string.**

eslint: [valid-typeof](http://eslint.org/docs/rules/valid-typeof)

typeof name === 'undefimed' // ✗ avoid

typeof name === 'undefined' // ✓ ok

* **Immediately Invoked Function Expressions (IIFEs) must be wrapped.**

eslint: [wrap-iife](http://eslint.org/docs/rules/wrap-iife)

const getName = function () { }() // ✗ avoid

const getName = (function () { }()) // ✓ ok

const getName = (function () { })() // ✓ ok

* **The \* in yield\*expressions must have a space before and after.**

eslint: [yield-star-spacing](http://eslint.org/docs/rules/yield-star-spacing)

yield\* increment() // ✗ avoid

yield \* increment() // ✓ ok

* **Avoid Yoda conditions.**

eslint: [yoda](http://eslint.org/docs/rules/yoda)

if (42 === age) { } // ✗ avoid

if (age === 42) { } // ✓ ok

[**Semicolons**](https://standardjs.com/rules.html#semicolons)

* No semicolons. (see: [1](http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding), [2](https://web.archive.org/web/20201206065632/http:/inimino.org/~inimino/blog/javascript_semicolons), [3](https://www.youtube.com/watch?v=gsfbh17Ax9I))

eslint: [semi](http://eslint.org/docs/rules/semi)

window.alert('hi') // ✓ ok

window.alert('hi'); // ✗ avoid

* Never start a line with (, [, `, or a handful of other unlikely possibilities.

This is the only gotcha with omitting semicolons, and standard protects you from this potential issue.

(The full list is: [, (, `, +, \*, /, -, ,, ., but most of these will never appear at the start of a line in real code.)

eslint: [no-unexpected-multiline](http://eslint.org/docs/rules/no-unexpected-multiline)

// ✓ ok

;(function () {

window.alert('ok')

}())

// ✗ avoid

(function () {

window.alert('ok')

}())

// ✓ ok

;[1, 2, 3].forEach(bar)

// ✗ avoid

[1, 2, 3].forEach(bar)

// ✓ ok

;`hello`.indexOf('o')

// ✗ avoid

`hello`.indexOf('o')

Note: If you're often writing code like this, you may be trying to be too clever.

Clever short-hands are discouraged, in favor of clear and readable expressions, whenever possible.

Instead of this:

;[1, 2, 3].forEach(bar)

This is strongly preferred:

var nums = [1, 2, 3]

nums.forEach(bar)

[**Helpful reading**](https://standardjs.com/rules.html#helpful-reading)

* [An Open Letter to JavaScript Leaders Regarding Semicolons](http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding)
* [JavaScript Semicolon Insertion – Everything you need to know](https://web.archive.org/web/20201206065632/http:/inimino.org/~inimino/blog/javascript_semicolons)

[**And a helpful video:**](https://standardjs.com/rules.html#and-a-helpful-video)

* [Are Semicolons Necessary in JavaScript? - YouTube](https://www.youtube.com/watch?v=gsfbh17Ax9I)

All popular code minifiers in use today use AST-based minification, so they can handle semicolon-less JavaScript with no issues (since semicolons are not required in JavaScript).

[**Excerpt from**](https://standardjs.com/rules.html#excerpt-from-an-open-letter-to-javascript-leaders-regarding-semicolons)[***"An Open Letter to JavaScript Leaders Regarding Semicolons"***](http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding)**:**

[Relying on automatic semicolon insertion] is quite safe, and perfectly valid JS that every browser understands. Closure compiler, yuicompressor, packer, and jsmin all can properly minify it. There is no performance impact anywhere.

I am sorry that, instead of educating you, the leaders in this language community have given you lies and fear. That was shameful. I recommend learning how statements in JS are actually terminated (and in which cases they are not terminated), so that you can write code that you find beautiful.

In general, \n ends a statement unless: 1. The statement has an unclosed paren, array literal, or object literal or ends in some other way that is not a valid way to end a statement. (For instance, ending with . or ,.) 2. The line is -- or ++ (in which case it will decrement/increment the next token.) 3. It is a for(), while(), do, if(), or else, and there is no { 4. The next line starts with [, (, +, \*, /, -, ,, ., or some other binary operator that can only be found between two tokens in a single expression.

The first is pretty obvious. Even JSLint is ok with \n chars in JSON and parenthesized constructs, and with var statements that span multiple lines ending in ,.

The second is super weird. I’ve never seen a case (outside of these sorts of conversations) where you’d want to do write i\n++\nj, but, point of fact, that’s parsed as i; ++j, not i++; j.

The third is well understood, if generally despised. if (x)\ny() is equivalent to if (x) { y() }. The construct doesn’t end until it reaches either a block, or a statement.

; is a valid JavaScript statement, so if(x); is equivalent to if(x){} or, “If x, do nothing.” This is more commonly applied to loops where the loop check also is the update function. Unusual, but not unheard of.

The fourth is generally the fud-inducing “oh noes, you need semicolons!” case. But, as it turns out, it’s quite easy to *prefix* those lines with semicolons if you don’t mean them to be continuations of the previous line. For example, instead of this:

foo();

[1,2,3].forEach(bar);

you could do this:

foo()

;[1,2,3].forEach(bar)

The advantage is that the prefixes are easier to notice, once you are accustomed to never seeing lines starting with ( or [ without semis.

Ruby Coding Standards

[Source Encoding](https://rubystyle.guide/#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/#tabs-or-spaces)

Use only spaces for indentation. No hard tabs.

[Indentation](https://rubystyle.guide/#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/#max-line-length)

Limit lines to 80 characters.

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

Why Bother with 80 characters in a World of Modern Widescreen Displays?

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

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

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

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

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

[No Trailing Whitespace](https://rubystyle.guide/#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/#crlf)

Use Unix-style line endings.[[2](https://rubystyle.guide/" \l "_footnotedef_2" \o "View 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/#newline-eof)

End each file with a newline.

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

[Should I Terminate Expressions with ;?](https://rubystyle.guide/#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/#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*

[Operator Method Call](https://rubystyle.guide/#operator-method-call)

Avoid dot where not required for operator method calls.

*# bad*

num.**+** 42

*# good*

num **+** 42

[Spaces and Operators](https://rubystyle.guide/#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**

[Safe navigation](https://rubystyle.guide/#safe-navigation)

Avoid long chains of &.. The longer the chain is, the harder it becomes to track what on it could be returning a nil. Replace with . and an explicit check. E.g. if users are guaranteed to have an address and addresses are guaranteed to have a zip code:

*# bad*

user**&**.**address&**.**zip&**.**upcase**

*# good*

user **&&** user.**address**.**zip**.**upcase**

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

*# bad*

user **&&** user.**address** **&&** user.**address**.**zip** **&&** user.**address**.**zip**.**upcase**

*# good*

**class** **User**

**def** **zip**

address**&**.**zip**

**end**

**end**

user**&**.**zip&**.**upcase**

[Spaces and Braces](https://rubystyle.guide/#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/#no-space-bang)

No space after !.

*# bad*

**!** something

*# good*

**!**something

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

No space inside range literals.

*# bad*

1 **..** 3

'a' **...** 'z'

*# good*

1**..**3

'a'**...**'z'

[Indent when to case](https://rubystyle.guide/#indent-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/#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/#empty-lines-between-methods)

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

*# bad*

**def** **some\_method**

data **=** initialize(options)

data.**manipulate!**

data.**result**

**end**

**def** **some\_other\_method**

result

**end**

*# good*

**def** **some\_method**

data **=** initialize(options)

data.**manipulate!**

data.**result**

**end**

**def** **some\_other\_method**

result

**end**

[Two or More Empty Lines](https://rubystyle.guide/#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/#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/#empty-lines-around-access-modifier)

Use empty lines around access modifier.

*# bad*

**class** **Foo**

**def** **bar**; **end**

**private**

**def** **baz**; **end**

**end**

*# good*

**class** **Foo**

**def** **bar**; **end**

**private**

**def** **baz**; **end**

**end**

[Empty Lines around Bodies](https://rubystyle.guide/#empty-lines-around-bodies)

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

*# bad*

**class** **Foo**

**def** **foo**

**begin**

do\_something **do**

something

**end**

**rescue**

something

**end**

**true**

**end**

**end**

*# good*

**class** **Foo**

**def** **foo**

**begin**

do\_something **do**

something

**end**

**rescue**

something

**end**

**end**

**end**

[Trailing Comma in Method Arguments](https://rubystyle.guide/#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/#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/#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/#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/#leading-dot-in-multi-line-chains)

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

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

one.**two**.**three**.

**four**

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

one.**two**.**three**

.**four**

[Trailing .](https://rubystyle.guide/#trailing-dot-in-multi-line-chains)

When continuing a chained method call 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/ruby-style-guide/pull/176).

[Method Arguments Alignment](https://rubystyle.guide/#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/#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/#no-dsl-decorating)

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

**class** **Person** **<** ActiveRecord**::**Base

*# bad*

attr\_reader(:name, :age)

*# good*

attr\_reader :name, :age

*# bad*

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

*# good*

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

**end**

[Space in Method Calls](https://rubystyle.guide/#parens-no-spaces)

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

*# bad*

puts (x **+** y)

*# good*

puts(x **+** y)

[Space in Brackets Access](https://rubystyle.guide/#space-in-brackets-access)

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

*# bad*

collection [index\_or\_key]

*# good*

collection[index\_or\_key]

[Multi-line Arrays Alignment](https://rubystyle.guide/#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/#naming-conventions)

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

*— Phil Karlton*

[English for Identifiers](https://rubystyle.guide/#english-identifiers)

Name identifiers in English.

*# bad - identifier is a Bulgarian word, using non-ascii (Cyrillic) characters*

заплата **=** 1\_000

*# bad - identifier is a Bulgarian word, written with Latin letters (instead of Cyrillic)*

zaplata **=** 1\_000

*# good*

salary **=** 1\_000

[Snake Case for Symbols, Methods and Variables](https://rubystyle.guide/#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/#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**

[CapitalCase for Classes and Modules](https://rubystyle.guide/#camelcase-classes)

|  |  |
| --- | --- |
| **Note** | CapitalCase is also known as UpperCamelCase, CapitalWords and PascalCase. |

Use CapitalCase 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/#snake-case-files)

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

[Snake Case for Directories](https://rubystyle.guide/#snake-case-dirs)

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

[One Class per File](https://rubystyle.guide/#one-class-per-file)

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

[Screaming Snake Case for Constants](https://rubystyle.guide/#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/#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/#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/#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/#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/#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 will suppress their unused variable warnings.

*# bad*

result **=** hash.**map** { **|**k, v**|** v **+** 1 }

**def** **something**(x)

unused\_var, used\_var **=** something\_else(x)

*# some code*

**end**

*# good*

result **=** hash.**map** { **|**\_k, v**|** v **+** 1 }

**def** **something**(x)

\_unused\_var, used\_var **=** something\_else(x)

*# some code*

**end**

*# good*

result **=** hash.**map** { **|**\_, v**|** v **+** 1 }

**def** **something**(x)

\_, used\_var **=** something\_else(x)

*# some code*

**end**

[other Parameter](https://rubystyle.guide/#other-arg)

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

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

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

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

*# 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/#flow-of-control)

[for Loops](https://rubystyle.guide/#no-for-loops)

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

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 block*

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

[then in Multi-line Expression](https://rubystyle.guide/#no-then)

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

*# bad*

**if** some\_condition **then**

*# body omitted*

**end**

*# bad*

**case** foo

**when** bar **then**

*# body omitted*

**end**

*# bad*

**case** expression

**in** pattern **then**

*# body omitted*

**end**

*# good*

**if** some\_condition

*# body omitted*

**end**

*# good*

**case** foo

**when** bar

*# body omitted*

**end**

*# good*

**case** expression

**in** pattern

*# body omitted*

**end**

[Condition Placement](https://rubystyle.guide/#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 if](https://rubystyle.guide/#ternary-operator)

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/#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 if](https://rubystyle.guide/#no-semicolon-ifs)

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 vs if-else](https://rubystyle.guide/#case-vs-if-else)

Prefer case over if-elsif when compared value is the 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 if/case](https://rubystyle.guide/#use-if-case-returns)

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/#one-line-cases)

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

|  |  |
| --- | --- |
| **Note** | The alternative syntax when x: …​ has been removed as of Ruby 1.9. |

[Semicolon in when](https://rubystyle.guide/#no-when-semicolons)

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

[Semicolon in in](https://rubystyle.guide/#no-in-pattern-semicolons)

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

*# bad*

**case** expression

**in** pattern; do\_something

**end**

*# good*

**case** expression

**in** pattern **then** do\_something

**end**

[! vs not](https://rubystyle.guide/#bang-not-not)

Use ! instead of not.

*# bad - parentheses are required because of op precedence*

x **=** (not something)

*# good*

x **=** **!**something

[Double Negation](https://rubystyle.guide/#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/or](https://rubystyle.guide/#and-or-flow)

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

*# good: and/or for control flow*

x **=** extract\_arguments or **raise** ArgumentError, "Not enough arguments!"

user.**suspended?** and **return** :denied

*# bad*

*# and/or in conditions (their precedence is low, might produce unexpected result)*

**if** got\_needed\_arguments and arguments\_valid

*# ...body omitted*

**end**

*# in logical expression calculation*

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

*# good*

*# &&/|| in conditions*

**if** got\_needed\_arguments **&&** arguments\_valid

*# ...body omitted*

**end**

*# in logical expression calculation*

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

*# bad*

*# &&/|| for control flow (can lead to very surprising results)*

x **=** extract\_arguments **||** **raise**(ArgumentError, "Not enough arguments!")

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

*# bad*

*# Did author mean conditional return because `#log` could result in `nil`?*

*# ...or was it just to have a smart one-liner?*

x **=** extract\_arguments and log("extracted") and **return**

*# good*

*# If the intention was conditional return*

x **=** extract\_arguments

**if** x

**return** **if** log("extracted")

**end**

*# If the intention was just "log, then return"*

x **=** extract\_arguments

**if** x

log("extracted")

**return**

**end**

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

Why is using and and or as logical operators a bad idea?

Simply put - because they add some cognitive overhead, as they don’t behave like similarly named logical operators in other languages.

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

foo **=** **true** and **false** *# results in foo being equal to true. Equivalent to (foo = true) and false*

bar **=** **false** or **true** *# results in bar being equal to false. Equivalent to (bar = false) or true*

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

**true** or **true** and **false** *# => false (it's effectively (true or true) and false)*

**true** **||** **true** **&&** **false** *# => true (it's effectively true || (true && false)*

**false** or **true** and **false** *# => false (it's effectively (false or true) and false)*

**false** **||** **true** **&&** **false** *# => false (it's effectively false || (true && false))*

[Multi-line Ternary Operator](https://rubystyle.guide/#no-multiline-ternary)

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

[if as a Modifier](https://rubystyle.guide/#if-as-a-modifier)

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

*# bad*

**if** some\_condition

do\_something

**end**

*# good*

do\_something **if** some\_condition

*# another good option*

some\_condition and do\_something

[Multi-line if Modifiers](https://rubystyle.guide/#no-multiline-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/#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 vs unless](https://rubystyle.guide/#unless-for-negatives)

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 else with unless](https://rubystyle.guide/#no-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/#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/#safe-assignment-in-condition). |

[Multi-line while do](https://rubystyle.guide/#no-multiline-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 as a Modifier](https://rubystyle.guide/#while-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 vs until](https://rubystyle.guide/#until-for-negatives)

Prefer until over while for negative conditions.

*# bad*

do\_something **while** **!**some\_condition

*# good*

do\_something **until** some\_condition

[Infinite Loop](https://rubystyle.guide/#infinite-loop)

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

*# bad*

**while** **true**

do\_something

**end**

**until** **false**

do\_something

**end**

*# good*

**loop** **do**

do\_something

**end**

[loop with break](https://rubystyle.guide/#loop-with-break)

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

*# bad*

**begin**

puts val

val **+=** 1

**end** **while** val **<** 0

*# good*

**loop** **do**

puts val

val **+=** 1

**break** **unless** val **<** 0

**end**

[Explicit return](https://rubystyle.guide/#no-explicit-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 self](https://rubystyle.guide/#no-self-unless-required)

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/#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/#safe-assignment-in-condition)

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

*# bad (+ a warning)*

**if** v **=** array.**grep**(/foo/)

do\_something(v)

*# some code*

**end**

*# good (MRI would still complain, but RuboCop won't)*

**if** (v **=** array.**grep**(/foo/))

do\_something(v)

*# some code*

**end**

*# good*

v **=** array.**grep**(/foo/)

**if** v

do\_something(v)

*# some code*

**end**

[BEGIN Blocks](https://rubystyle.guide/#no-BEGIN-blocks)

Avoid the use of BEGIN blocks.

[END Blocks](https://rubystyle.guide/#no-END-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/#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/#exceptions)

[raise vs fail](https://rubystyle.guide/#prefer-raise-over-fail)

Prefer raise over fail for exceptions.

*# bad*

fail SomeException, 'message'

*# good*

**raise** SomeException, 'message'

[Raising Explicit RuntimeError](https://rubystyle.guide/#no-explicit-runtimeerror)

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

*# bad*

**raise** RuntimeError, 'message'

*# good - signals a RuntimeError by default*

**raise** 'message'

[Exception Class Messages](https://rubystyle.guide/#exception-class-messages)

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

*# bad*

**raise** SomeException.**new**('message')

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

*# good*

**raise** SomeException, 'message'

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

[return from ensure](https://rubystyle.guide/#no-return-ensure)

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

*# bad*

**def** **foo**

**raise**

**ensure**

**return** 'very bad idea'

**end**

[Implicit begin](https://rubystyle.guide/#begin-implicit)

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

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/#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/#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/#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**

[Standard Exceptions](https://rubystyle.guide/#standard-exceptions)

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

[Files](https://rubystyle.guide/#files)

[Reading from a file](https://rubystyle.guide/#file-read)

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

*## text mode*

*# bad (only when reading from beginning to end - modes: 'r', 'rt', 'r+', 'r+t')*

File.**open**(filename).**read**

File.**open**(filename, **&**:read)

File.**open**(filename) { **|**f**|** f.**read** }

File.**open**(filename) **do** **|**f**|**

f.**read**

**end**

File.**open**(filename, 'r').**read**

File.**open**(filename, 'r', **&**:read)

File.**open**(filename, 'r') { **|**f**|** f.**read** }

File.**open**(filename, 'r') **do** **|**f**|**

f.**read**

**end**

*# good*

File.**read**(filename)

*## binary mode*

*# bad (only when reading from beginning to end - modes: 'rb', 'r+b')*

File.**open**(filename, 'rb').**read**

File.**open**(filename, 'rb', **&**:read)

File.**open**(filename, 'rb') { **|**f**|** f.**read** }

File.**open**(filename, 'rb') **do** **|**f**|**

f.**read**

**end**

*# good*

File.**binread**(filename)

[Writing to a file](https://rubystyle.guide/#file-write)

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

*## text mode*

*# bad (only truncating modes: 'w', 'wt', 'w+', 'w+t')*

File.**open**(filename, 'w').**write**(content)

File.**open**(filename, 'w') { **|**f**|** f.**write**(content) }

File.**open**(filename, 'w') **do** **|**f**|**

f.**write**(content)

**end**

*# good*

File.**write**(filename, content)

*## binary mode*

*# bad (only truncating modes: 'wb', 'w+b')*

File.**open**(filename, 'wb').**write**(content)

File.**open**(filename, 'wb') { **|**f**|** f.**write**(content) }

File.**open**(filename, 'wb') **do** **|**f**|**

f.**write**(content)

**end**

*# good*

File.**binwrite**(filename, content)

[Release External Resources](https://rubystyle.guide/#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/#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**

[Atomic File Operations](https://rubystyle.guide/#atomic-file-operations)

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

*# bad - race condition with another process may result in an error in `mkdir`*

**unless** Dir.**exist?**(path)

FileUtils.**mkdir**(path)

**end**

*# good - atomic and idempotent creation*

FileUtils.**mkdir\_p**(path)

*# bad - race condition with another process may result in an error in `remove`*

**if** File.**exist?**(path)

FileUtils.**remove**(path)

**end**

*# good - atomic and idempotent removal*

FileUtils.**rm\_f**(path)

[Null Devices](https://rubystyle.guide/#null-devices)

Use the platform independent null device (File::NULL) rather than hardcoding a value (/dev/null on Unix-like OSes, NUL or NUL: on Windows).

*# bad - hardcoded devices are platform specific*

File.**open**("/dev/null", 'w') { **...** }

*# bad - unnecessary ternary can be replaced with `File::NULL`*

File.**open**(Gem.**win\_platform?** ? 'NUL' : '/dev/null', 'w') { **...** }

*# good - platform independent*

File.**open**(File**::**NULL, 'w') { **...** }

[Assignment & Comparison](https://rubystyle.guide/#assignment-comparison)

[Parallel Assignment](https://rubystyle.guide/#parallel-assignment)

Avoid the use of parallel assignment for defining variables. Parallel assignment is allowed when it is the return of a method call (e.g. Hash#values\_at), 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/#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/#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/#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/#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/#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/#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/#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? vs kind\_of?](https://rubystyle.guide/#is-a-vs-kind-of)

Prefer is\_a? over kind\_of?. The two methods are synonyms, but is\_a? is the more commonly used name in the wild.

*# bad*

something.**kind\_of?**(Array)

*# good*

something.**is\_a?**(Array)

[is\_a? vs instance\_of?](https://rubystyle.guide/#is-a-vs-instance-of)

Prefer is\_a? over instance\_of?.

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

*# bad*

something.**instance\_of?**(Array)

*# good*

something.**is\_a?**(Array)

[instance\_of? vs class comparison](https://rubystyle.guide/#instance-of-vs-class-comparison)

Use Object#instance\_of? instead of class comparison for equality.

*# bad*

var.**class** **==** Date

var.**class**.**equal?**(Date)

var.**class**.**eql?**(Date)

var.**class**.**name** **==** 'Date'

*# good*

var.**instance\_of?**(Date)

[== vs eql?](https://rubystyle.guide/#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/#blocks-procs-lambdas)

[Proc Application Shorthand](https://rubystyle.guide/#single-action-blocks)

Use the Proc call shorthand when the called 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/#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?

[Single-line do…​end block](https://rubystyle.guide/#single-line-do-end-block)

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

*# bad*

foo **do** **|**arg**|** bar(arg) **end**

*# good*

foo **do** **|**arg**|**

bar(arg)

**end**

*# bad*

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

*# good*

**->**(arg) { bar(arg) }

[Explicit Block Argument](https://rubystyle.guide/#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/#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/#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 called.

*# 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/#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/#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/#stabby-lambda-no-args)

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

*# bad*

l **=** **->**() { something }

*# good*

l **=** **->** { something }

[proc vs Proc.new](https://rubystyle.guide/#proc)

Prefer proc over Proc.new.

*# bad*

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

*# good*

p **=** proc { **|**n**|** puts n }

[Proc Call](https://rubystyle.guide/#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]

*# bad - most compact form, but might be confusing for newcomers to Ruby*

l **=** **->**(v) { puts v }

l**.**(1)

*# good - a bit verbose, but crystal clear*

l **=** **->**(v) { puts v }

l.**call**(1)

[Methods](https://rubystyle.guide/#methods)

[Short Methods](https://rubystyle.guide/#short-methods)

Avoid methods longer than 10 LOC (lines of code). Ideally, most methods will be shorter than 5 LOC. Empty lines do not contribute to the relevant LOC.

[Top-Level Methods](https://rubystyle.guide/#top-level-methods)

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

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

*# bad*

**def** **some\_method**; **end**

*# good*

**class** **SomeClass**

**def** **some\_method**; **end**

**end**

[No Single-line Methods](https://rubystyle.guide/#no-single-line-methods)

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

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

*# 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**

[Endless Methods](https://rubystyle.guide/#endless-methods)

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

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

*# bad*

**def** **fib**(x) **=** **if** x **<** 2

x

**else**

fib(x **-** 1) **+** fib(x **-** 2)

**end**

*# good*

**def** **the\_answer** **=** 42

**def** **get\_x** **=** @x

**def** **square**(x) **=** x **\*** x

*# Not (so) good: has side effect*

**def** **set\_x**(x) **=** (@x **=** x)

**def** **print\_foo** **=** puts("foo")

[Ambiguous Endless Method Definitions](https://rubystyle.guide/#ambiguous-endless-method-defintions)

Keywords with lower precedence than = can appear ambiguous when used after an endless method definition. This includes and, or, and the modifier forms of if, unless, while, and until. In these cases, the code may appear to include these keywords as part of the method body, but instead they actually modify the method definition itself.

In this cases, prefer using a normal method over an endless method.

*# bad*

**def** **foo** **=** **true** **if** bar

*# good - using a non-endless method is more explicit*

**def** **foo**

**true**

**end** **if** bar

*# ok - method body is explicit*

**def** **foo** **=** (**true** **if** bar)

*# ok - method definition is explicit*

(**def** **foo** **=** **true**) **if** bar

[Double Colons](https://rubystyle.guide/#double-colons)

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

*# bad*

SomeClass**::**some\_method

some\_object**::**some\_method

*# good*

SomeClass.**some\_method**

some\_object.**some\_method**

SomeModule**::**SomeClass**::**SOME\_CONST

SomeModule**::**SomeClass()

[Colon Method Definition](https://rubystyle.guide/#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/#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 Call Parentheses](https://rubystyle.guide/#method-call-parentheses)

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

*# bad*

x **=** Math.**sin** y

*# good*

x **=** Math.**sin**(y)

*# bad*

array.**delete** e

*# good*

array.**delete**(e)

*# bad*

temperance **=** Person.**new** 'Temperance', 30

*# good*

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

[Method Call with No Arguments](https://rubystyle.guide/#method-call-with-no-arguments)

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 Have "keyword" Status in Ruby](https://rubystyle.guide/#methods-that-have-keyword-status-in-ruby)

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

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

[Non-Declarative Methods That Have "keyword" Status in Ruby](https://rubystyle.guide/#non-declarative-methods-that-have-keyword-status-in-ruby)

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

*# good (most popular)*

puts temperance.**age**

system 'ls'

exit 1

*# also good (less popular)*

puts(temperance.**age**)

system('ls')

exit(1)

[Using super with Arguments](https://rubystyle.guide/#super-with-args)

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/#too-many-params)

Avoid parameter lists longer than three or four parameters.

[Optional Arguments](https://rubystyle.guide/#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'*

[Keyword Arguments Order](https://rubystyle.guide/#keyword-arguments-order)

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

*# bad*

**def** **some\_method**(foo: **false**, bar:, baz: 10)

*# body omitted*

**end**

*# good*

**def** **some\_method**(bar:, foo: **false**, baz: 10)

*# body omitted*

**end**

[Boolean Keyword Arguments](https://rubystyle.guide/#boolean-keyword-arguments)

Use keyword arguments when passing a 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/#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/#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**

[Merging Keyword Arguments](https://rubystyle.guide/#merging-keyword-arguments)

When passing an existing hash as keyword arguments, add additional arguments directly rather than using merge.

*# bad*

some\_method(**\*\***opts.**merge**(foo: **true**))

*# good*

some\_method(**\*\***opts, foo: **true**)

[Arguments Forwarding](https://rubystyle.guide/#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/rubocop/issues/7549*

**def** **some\_method**(**\***args)

other\_method(**\***args)

**end**

*# good*

**def** **some\_method**(**...**)

other\_method(**...**)

**end**

[Block Forwarding](https://rubystyle.guide/#block-forwarding)

Use Ruby 3.1’s anonymous block forwarding.

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

*# bad*

**def** **some\_method**(**&**block)

other\_method(**&**block)

**end**

*# good*

**def** **some\_method**(**&**)

other\_method(**&**)

**end**

[Private Global Methods](https://rubystyle.guide/#private-global-methods)

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

[Classes & Modules](https://rubystyle.guide/#classes-modules)

[Consistent Classes](https://rubystyle.guide/#consistent-classes)

Use a consistent structure in your class definitions.

**class** **Person**

*# extend/include/prepend go first*

**extend** SomeModule

**include** AnotherModule

prepend YetAnotherModule

*# inner classes*

**class** **CustomError** **<** StandardError

**end**

*# constants are next*

SOME\_CONSTANT **=** 20

*# afterwards we have attribute macros*

attr\_reader :name

*# followed by other macros (if any)*

validates :name

*# public class methods are next in line*

**def** **self.some\_method**

**end**

*# initialization goes between class methods and other instance methods*

**def** **initialize**

**end**

*# followed by other public instance methods*

**def** **some\_method**

**end**

*# protected and private methods are grouped near the end*

**protected**

**def** **some\_protected\_method**

**end**

**private**

**def** **some\_private\_method**

**end**

**end**

[Mixin Grouping](https://rubystyle.guide/#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/#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/#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/#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/#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/#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/#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/#solid-design)

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

[Define to\_s](https://rubystyle.guide/#define-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 Family](https://rubystyle.guide/#attr_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/#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/#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/#struct-new)

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

*# good*

**class** **Person**

attr\_accessor :first\_name, :last\_name

**def** **initialize**(first\_name, last\_name)

@first\_name **=** first\_name

@last\_name **=** last\_name

**end**

**end**

*# better*

Person **=** Struct.**new**(:first\_name, :last\_name) **do**

**end**

[Don’t Extend Struct.new](https://rubystyle.guide/#no-extend-struct-new)

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

*# bad*

**class** **Person** **<** Struct.**new**(:first\_name, :last\_name)

**end**

*# good*

Person **=** Struct.**new**(:first\_name, :last\_name)

[Don’t Extend Data.define](https://rubystyle.guide/#no-extend-data-define)

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

*# bad*

**class** **Person** **<** Data.**define**(:first\_name, :last\_name)

**end**

Person.**ancestors**

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

*# good*

Person **=** Data.**define**(:first\_name, :last\_name)

Person.**ancestors**

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

[Duck Typing](https://rubystyle.guide/#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/#no-class-vars)

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

**class** **Parent**

@@class\_var **=** 'parent'

**def** **self.print\_class\_var**

puts @@class\_var

**end**

**end**

**class** **Child** **<** Parent

@@class\_var **=** 'child'

**end**

Parent.**print\_class\_var** *# => will print 'child'*

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

[Leverage Access Modifiers (e.g. private and protected)](https://rubystyle.guide/#visibility)

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/#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/#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/#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/#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 self](https://rubystyle.guide/#class-and-self)

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

**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/#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/#classes-constructors)

[Factory Methods](https://rubystyle.guide/#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/#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/#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/#no-comments)

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

[Rationale Comments](https://rubystyle.guide/#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/#english-comments)

Write comments in English.

[Hash Space](https://rubystyle.guide/#hash-space)

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

[English Syntax](https://rubystyle.guide/#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/#no-superfluous-comments)

Avoid superfluous comments.

*# bad*

counter **+=** 1 *# Increments counter by one.*

[Comment Upkeep](https://rubystyle.guide/#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/#refactor-dont-comment)

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

*— old programmers maxim*[*through Russ Olsen*](https://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/#comment-annotations)

[Annotations Placement](https://rubystyle.guide/#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/#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/#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 purposes).

**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/#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/#todo)

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

[FIXME](https://rubystyle.guide/#fixme)

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

[OPTIMIZE](https://rubystyle.guide/#optimize)

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

[HACK](https://rubystyle.guide/#hack)

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

[REVIEW](https://rubystyle.guide/#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/#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/#magic-comments)

[Magic Comments First](https://rubystyle.guide/#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/#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/#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/#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/#collections)

[Literal Array and Hash](https://rubystyle.guide/#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/#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/#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/#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/#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 and last](https://rubystyle.guide/#first-and-last)

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

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

*# ok*

arr[0] *# => 1*

arr[**-**1] *# => 3*

*# (arguably) better*

arr.**first** *# => 1*

arr.**last** *# => 3*

*# good - assignments can only be done via []=*

arr[0] **=** 2

arr[**-**1] **=** 5

[Set vs Array](https://rubystyle.guide/#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/#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/#no-mutable-keys)

Avoid the use of mutable objects as hash keys.

[No Mutable Defaults](https://rubystyle.guide/#no-mutable-defaults)

Avoid the use of shared mutable objects as hash default values.

Creating a Hash in such a way will share the default value across all keys, causing unexpected behavior when modifying it.

For example, when the Hash was created with an Array as the argument, calling hash[:foo] << 'bar' will also change the value of all other keys that have not been explicitly assigned to.

*# bad*

Hash.**new**([])

Hash.**new**({})

Hash.**new**(Array.**new**)

Hash.**new**(Hash.**new**)

*# okay -- beware this will silently discard mutations and only remember assignments*

Hash.**new** { Array.**new** }

Hash.**new** { Hash.**new** }

Hash.**new** { {} }

Hash.**new** { [] }

*# good - frozen solution will raise an error when mutation is attempted*

Hash.**new**([].**freeze**)

Hash.**new**({}.**freeze**)

*# good - using a proc will create a new object for each key*

h **=** Hash.**new**

h.**default\_proc** **=** **->**(h, k) { [] }

h.**default\_proc** **=** **->**(h, k) { {} }

*# good - using a block will create a new object for each key*

Hash.**new** { **|**h, k**|** h[k] **=** [] }

Hash.**new** { **|**h, k**|** h[k] **=** {} }

[Hash Literals](https://rubystyle.guide/#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 Values](https://rubystyle.guide/#hash-literal-values)

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

*# bad*

hash **=** { one: one, two: two, three: three }

*# good*

hash **=** { one:, two:, three: }

[Hash Literal as Last Array Item](https://rubystyle.guide/#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/#no-mixed-hash-syntaxes)

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

*# bad*

{ a: 1, 'b' **=>** 2 }

*# good*

{ :a **=>** 1, 'b' **=>** 2 }

[Avoid Hash[] constructor](https://rubystyle.guide/#avoid-hash-constructor)

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

*# bad*

Hash[ary]

Hash[a, b, c, d]

*# good*

ary.**to\_h**

{a **=>** b, c **=>** d}

[Hash#key?](https://rubystyle.guide/#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/#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/#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 defaults](https://rubystyle.guide/#hash-fetch-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 falsey value we won't get the expected result*

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

*# good - fetch works correctly with falsey values*

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

[Use Hash Blocks](https://rubystyle.guide/#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 and Hash#fetch\_values](https://rubystyle.guide/#hash-values-at-and-hash-fetch-values)

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

*# bad*

email **=** data['email']

username **=** data['nickname']

*# bad*

keys **=** %w[email nickname].**freeze**

email, username **=** keys.**map** { **|**key**|** data[key] }

*# good*

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

*# also good*

email, username **=** data.**fetch\_values**('email', 'nickname')

[Hash#transform\_keys and Hash#transform\_values](https://rubystyle.guide/#hash-transform-methods)

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/#ordered-hashes)

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

[No Modifying Collections](https://rubystyle.guide/#no-modifying-collections)

Do not modify a collection while traversing it.

[Accessing Elements Directly](https://rubystyle.guide/#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/#provide-alternate-accessor-to-collections)

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

*# bad*

**def** **awesome\_things**

@awesome\_things

**end**

*# good*

**def** **awesome\_things**(index **=** **nil**)

**if** index **&&** @awesome\_things

@awesome\_things[index]

**else**

@awesome\_things

**end**

**end**

[map/find/select/reduce/include?/size](https://rubystyle.guide/#map-find-select-reduce-include-size)

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

[count vs size](https://rubystyle.guide/#count-vs-size)

Don’t use count as a substitute for size. For Enumerable objects other than Array it will iterate the entire collection in order to determine its size.

*# bad*

some\_hash.**count**

*# good*

some\_hash.**size**

[flat\_map](https://rubystyle.guide/#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/#reverse-each)

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

*# bad*

array.**reverse**.**each** { **...** }

*# good*

array.**reverse\_each** { **...** }

[Object#yield\_self vs Object#then](https://rubystyle.guide/#object-yield-self-vs-object-then)

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

*# bad*

obj.**yield\_self** { **|**x**|** x.**do\_something** }

*# good*

obj.**then** { **|**x**|** x.**do\_something** }

|  |  |
| --- | --- |
| **Note** | You can read more about the rationale behind this guideline [here](https://bugs.ruby-lang.org/issues/14594). |

[Slicing with Ranges](https://rubystyle.guide/#slicing-with-ranges)

Slicing arrays with ranges to extract some of their elements (e.g ary[2..5]) is a popular technique. Below you’ll find a few small considerations to keep in mind when using it.

* [0..-1] in ary[0..-1] is redundant and simply synonymous with ary.

*# bad - you're selecting all the elements of the array*

ary[0**..-**1]

ary[0**..nil**]

ary[0**...nil**]

*# good*

ary

* Ruby 2.6 introduced endless ranges, which provide an easier way to describe a slice going all the way to the end of an array.

*# bad - hard to process mentally*

ary[1**..-**1]

ary[1**..nil**]

*# good - easier to read and more concise*

ary[1**..**]

* Ruby 2.7 introduced beginless ranges, which are also handy in slicing. However, unlike the somewhat obscure -1 in ary[1..-1], the 0 in ary[0..42] is clear as a starting point. In fact, changing it to ary[..42] could potentially make it less readable. Therefore, using code like ary[0..42] is fine. On the other hand, ary[nil..42] should be replaced with ary[..42] or arr[0..42].

*# bad - hard to process mentally*

ary[**nil..**42]

*# good - easier to read*

ary[**..**42]

ary[0**..**42]

[Numbers](https://rubystyle.guide/#numbers)

[Underscores in Numerics](https://rubystyle.guide/#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/#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/#integer-type-checking)

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

timestamp **=** Time.**now**.**to\_i**

*# bad*

timestamp.**is\_a?**(Fixnum)

timestamp.**is\_a?**(Bignum)

*# good*

timestamp.**is\_a?**(Integer)

[Random Numbers](https://rubystyle.guide/#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/#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/#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/#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/#strings)

[String Interpolation](https://rubystyle.guide/#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/#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/#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/#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/#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/#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 to\_s](https://rubystyle.guide/#no-to-s)

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

*# bad*

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

*# good*

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

[String Concatenation](https://rubystyle.guide/#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 gsub](https://rubystyle.guide/#dont-abuse-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**('-', '\_')

[String#chars](https://rubystyle.guide/#string-chars)

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

|  |  |
| --- | --- |
| **Note** | These cases have the same behavior since Ruby 2.0. |

*# bad*

string.**split**(//)

string.**split**('')

*# good*

string.**chars**

[sprintf](https://rubystyle.guide/#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/#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/#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/#heredocs)

[Squiggly Heredocs](https://rubystyle.guide/#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/#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/#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/#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/#date-time)

[Time.now](https://rubystyle.guide/#time-now)

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

[No DateTime](https://rubystyle.guide/#no-datetime)

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

*# bad - uses DateTime for current time*

DateTime.**now**

*# good - uses Time for current time*

Time.**now**

*# bad - uses DateTime for modern date*

DateTime.**iso8601**('2016-06-29')

*# good - uses Date for modern date*

Date.**iso8601**('2016-06-29')

*# good - uses DateTime with start argument for historical date*

DateTime.**iso8601**('1751-04-23', Date**::**ENGLAND)

[Regular Expressions](https://rubystyle.guide/#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/#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/#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/#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/#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/#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/#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/#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/#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/#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/#multi-line-regexes)

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

|  |  |
| --- | --- |
| **Note** | That’s known as [free-spacing mode](https://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/#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 gsub with a Block or a Hash for Complex Replacements](https://rubystyle.guide/#gsub-blocks)

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/#percent-literals)

[%q shorthand](https://rubystyle.guide/#percent-q-shorthand)

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

*# bad (no interpolation needed)*

%(<div class="text">Some text</div>)

*# should be '<div class="text">Some text</div>'*

*# bad (no double-quotes)*

%(This is #{quality} style)

*# should be "This is #{quality} style"*

*# bad (multiple lines)*

%(<div>\n<span class="big">#{exclamation}</span>\n</div>)

*# should be a heredoc.*

*# good (requires interpolation, has quotes, single line)*

%(<tr><td class="name">#{name}</td>)

[%q](https://rubystyle.guide/#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/#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/#percent-x)

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

*# bad*

date **=** %x(date)

*# good*

date **=** `date`

echo **=** %x(echo `date`)

[%s](https://rubystyle.guide/#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/#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/#metaprogramming)

[No Needless Metaprogramming](https://rubystyle.guide/#no-needless-metaprogramming)

Avoid needless metaprogramming.

[No Monkey Patching](https://rubystyle.guide/#no-monkey-patching)

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

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

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

[Supply Location](https://rubystyle.guide/#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/#class-eval-define_method)

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

[eval Comment Docs](https://rubystyle.guide/#eval-comment-docs)

When using class\_eval (or other eval) with string interpolation, add a comment block showing its appearance if interpolated (a practice used in Rails code):

*# from activesupport/lib/active\_support/core\_ext/string/output\_safety.rb*

UNSAFE\_STRING\_METHODS.**each** **do** **|**unsafe\_method**|**

**if** 'String'.**respond\_to?**(unsafe\_method)

class\_eval **<<-**EOT, **\_\_FILE\_\_**, **\_\_LINE\_\_** **+** 1

def #{unsafe\_method}(\*params, &block) # def capitalize(\*params, &block)

to\_str.#{unsafe\_method}(\*params, &block) # to\_str.capitalize(\*params, &block)

end # end

def #{unsafe\_method}!(\*params) # def capitalize!(\*params)

@dirty = true # @dirty = true

super # super

end # end

EOT

**end**

**end**

[No method\_missing](https://rubystyle.guide/#no-method-missing)

Avoid using method\_missing for metaprogramming because backtraces become messy, the behavior is not listed in #methods, and misspelled method calls might silently work, e.g. nukes.luanch\_state = false. Consider using delegation, proxy, or define\_method instead. If you must use method\_missing:

* Be sure to [also define respond\_to\_missing?](https://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 public\_send](https://rubystyle.guide/#prefer-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 \_\_send\_\_](https://rubystyle.guide/#prefer-__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)

*# bad - Won't send a message to the receiver object. Instead it will send a message via UDP socket.*

u2.**send** :sleep, 0

*# good - Will actually send a message to the receiver object.*

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

[API Documentation](https://rubystyle.guide/#api-documentation)

[YARD](https://rubystyle.guide/#yard)

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

[RD (Block) Comments](https://rubystyle.guide/#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 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](https://rubystyle.guide/#_footnotedef_3)]

[Gemfile and Gemspec](https://rubystyle.guide/#gemfile-and-gemspec)

[No RUBY\_VERSION in the gemspec](https://rubystyle.guide/#no-ruby-version-in-the-gemspec)

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

*# bad*

Gem**::**Specification.**new** **do** **|**s**|**

**if** RUBY\_VERSION **>=** '2.5'

s.**add\_dependency** 'gem\_a'

**else**

s.**add\_dependency** 'gem\_b'

**end**

**end**

Fix by either:

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

[add\_dependency vs add\_runtime\_dependency](https://rubystyle.guide/#add_dependency_vs_add_runtime_dependency)

Prefer add\_dependency over add\_runtime\_dependency because add\_dependency is considered soft-deprecated and the Bundler team recommends add\_dependency.

*# bad*

Gem**::**Specification.**new** **do** **|**s**|**

s.**add\_runtime\_dependency** 'gem\_a'

**end**

*# good*

Gem**::**Specification.**new** **do** **|**s**|**

s.**add\_dependency** 'gem\_a'

**end**

See <https://github.com/rubygems/rubygems/issues/7799#issuecomment-2192720316> for details.

[Misc](https://rubystyle.guide/#misc)

[No Flip-flops](https://rubystyle.guide/#no-flip-flops)

Avoid the use of [flip-flop operators](https://en.wikipedia.org/wiki/Flip-flop_(programming)).

[No non-nil Checks](https://rubystyle.guide/#no-non-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/#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/#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/#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/#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 between](https://rubystyle.guide/#ranges-or-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/#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**

[Bitwise Predicate Methods](https://rubystyle.guide/#bitwise-predicate-methods)

Prefer bitwise predicate methods over direct comparison operations.

*# bad - checks any set bits*

(variable **&** flags).**positive?**

*# good*

variable.**anybits?**(flags)

*# bad - checks all set bits*

(variable **&** flags) **==** flags

*# good*

variable.**allbits?**(flags)

*# bad - checks no set bits*

(variable **&** flags).**zero?**

(variable **&** flags) **==** 0

*# good*

variable.**nobits?**(flags)

[No Cryptic Perlisms](https://rubystyle.guide/#no-cryptic-perlisms)

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

*# bad*

$:.**unshift** File.**dirname**(**\_\_FILE\_\_**)

*# good*

$LOAD\_PATH.**unshift** File.**dirname**(**\_\_FILE\_\_**)

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

*# bad*

print $', $$

*# good*

require 'English'

print $POSTMATCH, $PID

[Use require\_relative whenever possible](https://rubystyle.guide/#use-require_relative-whenever-possible)

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

*# bad*

require 'set'

require 'my\_gem/spec/helper'

require 'my\_gem/lib/something'

*# good*

require 'set'

require\_relative 'helper'

require\_relative '../lib/something'

This way is more expressive (making clear which dependency is internal or not) and more efficient (as require\_relative doesn’t have to try all of $LOAD\_PATH contrary to require).

[Always Warn](https://rubystyle.guide/#always-warn)

Write ruby -w safe code.

[No Optional Hash Params](https://rubystyle.guide/#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/#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/#optionparser)

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

[No Param Mutations](https://rubystyle.guide/#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/#three-is-the-number-thou-shalt-count)

Avoid more than three levels of block nesting.

[Functional Code](https://rubystyle.guide/#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 .rb to require](https://rubystyle.guide/#no-explicit-rb-to-require)

Omit the .rb extension for filename passed to require and require\_relative.

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

*# bad*

require 'foo.rb'

require\_relative '../foo.rb'

*# good*

require 'foo'

require 'foo.so'

require\_relative '../foo'

require\_relative '../foo.so'

[Avoid tap](https://rubystyle.guide/#avoid-tap)

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

*# bad*

Config.**new**(hash, path).**tap** **do** **|**config**|**

config.**check** **if** check

**end**

*# good*

config **=** Config.**new**(hash, path)

config.**check** **if** check

config

This is simpler and more efficient.

[Tools](https://rubystyle.guide/#tools)

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

[RuboCop](https://rubystyle.guide/#rubocop)

[RuboCop](https://github.com/rubocop/rubocop) is a Ruby static code analyzer and formatter, based on this style guide. RuboCop already covers a significant portion of the guide and has [plugins](https://docs.rubocop.org/rubocop/integration_with_other_tools.html) 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/#rubymine)

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

Kotlin Coding Standards

**Coding conventions﻿**

[Edit page](https://github.com/JetBrains/kotlin-web-site/edit/master/docs/topics/coding-conventions.md)Last modified: 24 April 2025

Commonly known and easy-to-follow coding conventions are vital for any programming language. Here we provide guidelines on the code style and code organization for projects that use Kotlin.

**Configure style in IDE﻿**

Two most popular IDEs for Kotlin - [IntelliJ IDEA](https://www.jetbrains.com/idea/) and [Android Studio](https://developer.android.com/studio/) provide powerful support for code styling. You can configure them to automatically format your code in consistence with the given code style.

**Apply the style guide﻿**

1. Go to Settings/Preferences | Editor | Code Style | Kotlin.
2. Click Set from....
3. Select Kotlin style guide.

**Verify that your code follows the style guide﻿**

1. Go to Settings/Preferences | Editor | Inspections | General.
2. Switch on Incorrect formatting inspection. Additional inspections that verify other issues described in the style guide (such as naming conventions) are enabled by default.

**Source code organization﻿**

**Directory structure﻿**

In pure Kotlin projects, the recommended directory structure follows the package structure with the common root package omitted. For example, if all the code in the project is in the org.example.kotlin package and its subpackages, files with the org.example.kotlin package should be placed directly under the source root, and files in org.example.kotlin.network.socket should be in the network/socket subdirectory of the source root.

On JVM: In projects where Kotlin is used together with Java, Kotlin source files should reside in the same source root as the Java source files, and follow the same directory structure: each file should be stored in the directory corresponding to each package statement.

**Source file names﻿**

If a Kotlin file contains a single class or interface (potentially with related top-level declarations), its name should be the same as the name of the class, with the .kt extension appended. It applies to all types of classes and interfaces. If a file contains multiple classes, or only top-level declarations, choose a name describing what the file contains, and name the file accordingly. Use [upper camel case](https://en.wikipedia.org/wiki/Camel_case), where the first letter of each word is capitalized. For example, ProcessDeclarations.kt.

The name of the file should describe what the code in the file does. Therefore, you should avoid using meaningless words such as Util in file names.

**Multiplatform projects﻿**

In multiplatform projects, files with top-level declarations in platform-specific source sets should have a suffix associated with the name of the source set. For example:

* jvmMain/kotlin/Platform. jvm .kt
* androidMain/kotlin/Platform. android .kt
* iosMain/kotlin/Platform. ios .kt

As for the common source set, files with top-level declarations should not have a suffix. For example, commonMain/kotlin/Platform.kt.

**Technical details﻿**

**Source file organization﻿**

Placing multiple declarations (classes, top-level functions or properties) in the same Kotlin source file is encouraged as long as these declarations are closely related to each other semantically, and the file size remains reasonable (not exceeding a few hundred lines).

In particular, when defining extension functions for a class which are relevant for all clients of this class, put them in the same file with the class itself. When defining extension functions that make sense only for a specific client, put them next to the code of that client. Avoid creating files just to hold all extensions of some class.

**Class layout﻿**

The contents of a class should go in the following order:

1. Property declarations and initializer blocks
2. Secondary constructors
3. Method declarations
4. Companion object

Do not sort the method declarations alphabetically or by visibility, and do not separate regular methods from extension methods. Instead, put related stuff together, so that someone reading the class from top to bottom can follow the logic of what's happening. Choose an order (either higher-level stuff first, or vice versa) and stick to it.

Put nested classes next to the code that uses those classes. If the classes are intended to be used externally and aren't referenced inside the class, put them in the end, after the companion object.

**Interface implementation layout﻿**

When implementing an interface, keep the implementing members in the same order as members of the interface (if necessary, interspersed with additional private methods used for the implementation).

**Overload layout﻿**

Always put overloads next to each other in a class.

**Naming rules﻿**

Package and class naming rules in Kotlin are quite simple:

* Names of packages are always lowercase and do not use underscores (org.example.project). Using multi-word names is generally discouraged, but if you do need to use multiple words, you can either just concatenate them together or use camel case (org.example.myProject).
* Names of classes and objects use upper camel case:

open class DeclarationProcessor { /\*...\*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /\*...\*/ }

**Function names﻿**

Names of functions, properties and local variables start with a lowercase letter and use camel case with no underscores:

fun processDeclarations() { /\*...\*/ }

var declarationCount = 1

Exception: factory functions used to create instances of classes can have the same name as the abstract return type:

interface Foo { /\*...\*/ }

class FooImpl : Foo { /\*...\*/ }

fun Foo(): Foo { return FooImpl() }

**Names for test methods﻿**

In tests (and only in tests), you can use method names with spaces enclosed in backticks. Note that such method names are only supported by Android runtime from API level 30. Underscores in method names are also allowed in test code.

class MyTestCase {

@Test fun `ensure everything works`() { /\*...\*/ }

@Test fun ensureEverythingWorks\_onAndroid() { /\*...\*/ }

}

**Property names﻿**

Names of constants (properties marked with const, or top-level or object val properties with no custom get function that hold deeply immutable data) should use all uppercase, underscore-separated names following the [screaming snake case](https://en.wikipedia.org/wiki/Snake_case) convention:

const val MAX\_COUNT = 8

val USER\_NAME\_FIELD = "UserName"

Names of top-level or object properties which hold objects with behavior or mutable data should use camel case names:

val mutableCollection: MutableSet<String> = HashSet()

Names of properties holding references to singleton objects can use the same naming style as object declarations:

val PersonComparator: Comparator<Person> = /\*...\*/

For enum constants, it's OK to use either all uppercase, underscore-separated ([screaming snake case](https://en.wikipedia.org/wiki/Snake_case)) names (enum class Color { RED, GREEN }) or upper camel case names, depending on the usage.

**Names for backing properties﻿**

If a class has two properties which are conceptually the same but one is part of a public API and another is an implementation detail, use an underscore as the prefix for the name of the private property:

class C {

private val \_elementList = mutableListOf<Element>()

val elementList: List<Element>

get() = \_elementList

}

**Choose good names﻿**

The name of a class is usually a noun or a noun phrase explaining what the class is: List, PersonReader.

The name of a method is usually a verb or a verb phrase saying what the method does: close, readPersons. The name should also suggest if the method is mutating the object or returning a new one. For instance sort is sorting a collection in place, while sorted is returning a sorted copy of the collection.

The names should make it clear what the purpose of the entity is, so it's best to avoid using meaningless words (Manager, Wrapper) in names.

When using an acronym as part of a declaration name, follow these rules:

* For two-letter acronyms, use uppercase for both letters. For example, IOStream.
* For acronyms longer than two letters, capitalize only the first letter. For example, XmlFormatter or HttpInputStream.

**Formatting﻿**

**Indentation﻿**

Use four spaces for indentation. Do not use tabs.

For curly braces, put the opening brace at the end of the line where the construct begins, and the closing brace on a separate line aligned horizontally with the opening construct.

if (elements != null) {

for (element in elements) {

// ...

}

}

In Kotlin, semicolons are optional, and therefore line breaks are significant. The language design assumes Java-style braces, and you may encounter surprising behavior if you try to use a different formatting style.

**Horizontal whitespace﻿**

* Put spaces around binary operators (a + b). Exception: don't put spaces around the "range to" operator (0..i).
* Do not put spaces around unary operators (a++).
* Put spaces between control flow keywords (if, when, for, and while) and the corresponding opening parenthesis.
* Do not put a space before an opening parenthesis in a primary constructor declaration, method declaration or method call.

class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {

foo(1)

}

* Never put a space after (, [, or before ], ).
* Never put a space around . or ?.: foo.bar().filter { it > 2 }.joinToString(), foo?.bar().
* Put a space after //: // This is a comment.
* Do not put spaces around angle brackets used to specify type parameters: class Map<K, V> { ... }.
* Do not put spaces around ::: Foo::class, String::length.
* Do not put a space before ? used to mark a nullable type: String?.

As a general rule, avoid horizontal alignment of any kind. Renaming an identifier to a name with a different length should not affect the formatting of either the declaration or any of the usages.

**Colon﻿**

Put a space before : in the following scenarios:

* When it's used to separate a type and a supertype.
* When delegating to a superclass constructor or a different constructor of the same class.
* After the object keyword.

Don't put a space before : when it separates a declaration and its type.

Always put a space after :.

abstract class Foo<out T : Any> : IFoo {

abstract fun foo(a: Int): T

}

class FooImpl : Foo() {

constructor(x: String) : this(x) { /\*...\*/ }

val x = object : IFoo { /\*...\*/ }

}

**Class headers﻿**

Classes with a few primary constructor parameters can be written in a single line:

class Person(id: Int, name: String)

Classes with longer headers should be formatted so that each primary constructor parameter is in a separate line with indentation. Also, the closing parenthesis should be on a new line. If you use inheritance, the superclass constructor call, or the list of implemented interfaces should be located on the same line as the parenthesis:

class Person(

id: Int,

name: String,

surname: String

) : Human(id, name) { /\*...\*/ }

For multiple interfaces, the superclass constructor call should be located first and then each interface should be located in a different line:

class Person(

id: Int,

name: String,

surname: String

) : Human(id, name),

KotlinMaker { /\*...\*/ }

For classes with a long supertype list, put a line break after the colon and align all supertype names horizontally:

class MyFavouriteVeryLongClassHolder :

MyLongHolder<MyFavouriteVeryLongClass>(),

SomeOtherInterface,

AndAnotherOne {

fun foo() { /\*...\*/ }

}

To clearly separate the class header and body when the class header is long, either put a blank line following the class header (as in the example above), or put the opening curly brace on a separate line:

class MyFavouriteVeryLongClassHolder :

MyLongHolder<MyFavouriteVeryLongClass>(),

SomeOtherInterface,

AndAnotherOne

{

fun foo() { /\*...\*/ }

}

Use regular indent (four spaces) for constructor parameters. This ensures that properties declared in the primary constructor have the same indentation as properties declared in the body of a class.

**Modifiers order﻿**

If a declaration has multiple modifiers, always put them in the following order:

public / protected / private / internal

expect / actual

final / open / abstract / sealed / const

external

override

lateinit

tailrec

vararg

suspend

inner

enum / annotation / fun // as a modifier in `fun interface`

companion

inline / value

infix

operator

data

Place all annotations before modifiers:

@Named("Foo")

private val foo: Foo

Unless you're working on a library, omit redundant modifiers (for example, public).

**Annotations﻿**

Place annotations on separate lines before the declaration to which they are attached, and with the same indentation:

@Target(AnnotationTarget.PROPERTY)

annotation class JsonExclude

Annotations without arguments may be placed on the same line:

@JsonExclude @JvmField

var x: String

A single annotation without arguments may be placed on the same line as the corresponding declaration:

@Test fun foo() { /\*...\*/ }

**File annotations﻿**

File annotations are placed after the file comment (if any), before the package statement, and are separated from package with a blank line (to emphasize the fact that they target the file and not the package).

/\*\* License, copyright and whatever \*/

@file:JvmName("FooBar")

package foo.bar

**Functions﻿**

If the function signature doesn't fit on a single line, use the following syntax:

fun longMethodName(

argument: ArgumentType = defaultValue,

argument2: AnotherArgumentType,

): ReturnType {

// body

}

Use regular indent (four spaces) for function parameters. It helps ensure consistency with constructor parameters.

Prefer using an expression body for functions with the body consisting of a single expression.

fun foo(): Int { // bad

return 1

}

fun foo() = 1 // good

**Expression bodies﻿**

If the function has an expression body whose first line doesn't fit on the same line as the declaration, put the = sign on the first line and indent the expression body by four spaces.

fun f(x: String, y: String, z: String) =

veryLongFunctionCallWithManyWords(andLongParametersToo(), x, y, z)

**Properties﻿**

For very simple read-only properties, consider one-line formatting:

val isEmpty: Boolean get() = size == 0

For more complex properties, always put get and set keywords on separate lines:

val foo: String

get() { /\*...\*/ }

For properties with an initializer, if the initializer is long, add a line break after the = sign and indent the initializer by four spaces:

private val defaultCharset: Charset? =

EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)

**Control flow statements﻿**

If the condition of an if or when statement is multiline, always use curly braces around the body of the statement. Indent each subsequent line of the condition by four spaces relative to the statement start. Put the closing parentheses of the condition together with the opening curly brace on a separate line:

if (!component.isSyncing &&

!hasAnyKotlinRuntimeInScope(module)

) {

return createKotlinNotConfiguredPanel(module)

}

This helps align the condition and statement bodies.

Put the else, catch, finally keywords, as well as the while keyword of a do-while loop, on the same line as the preceding curly brace:

if (condition) {

// body

} else {

// else part

}

try {

// body

} finally {

// cleanup

}

In a when statement, if a branch is more than a single line, consider separating it from adjacent case blocks with a blank line:

private fun parsePropertyValue(propName: String, token: Token) {

when (token) {

is Token.ValueToken ->

callback.visitValue(propName, token.value)

Token.LBRACE -> { // ...

}

}

}

Put short branches on the same line as the condition, without braces.

when (foo) {

true -> bar() // good

false -> { baz() } // bad

}

**Method calls﻿**

In long argument lists, put a line break after the opening parenthesis. Indent arguments by four spaces. Group multiple closely related arguments on the same line.

drawSquare(

x = 10, y = 10,

width = 100, height = 100,

fill = true

)

Put spaces around the = sign separating the argument name and value.

**Wrap chained calls﻿**

When wrapping chained calls, put the . character or the ?. operator on the next line, with a single indent:

val anchor = owner

?.firstChild!!

.siblings(forward = true)

.dropWhile { it is PsiComment || it is PsiWhiteSpace }

The first call in the chain should usually have a line break before it, but it's OK to omit it if the code makes more sense that way.

**Lambdas﻿**

In lambda expressions, spaces should be used around the curly braces, as well as around the arrow which separates the parameters from the body. If a call takes a single lambda, pass it outside parentheses whenever possible.

list.filter { it > 10 }

If assigning a label for a lambda, do not put a space between the label and the opening curly brace:

fun foo() {

ints.forEach lit@{

// ...

}

}

When declaring parameter names in a multiline lambda, put the names on the first line, followed by the arrow and the newline:

appendCommaSeparated(properties) { prop ->

val propertyValue = prop.get(obj) // ...

}

If the parameter list is too long to fit on a line, put the arrow on a separate line:

foo {

context: Context,

environment: Env

->

context.configureEnv(environment)

}

**Trailing commas﻿**

A trailing comma is a comma symbol after the last item in a series of elements:

class Person(

val firstName: String,

val lastName: String,

val age: Int, // trailing comma

)

Using trailing commas has several benefits:

* It makes version-control diffs cleaner – as all the focus is on the changed value.
* It makes it easy to add and reorder elements – there is no need to add or delete the comma if you manipulate elements.
* It simplifies code generation, for example, for object initializers. The last element can also have a comma.

Trailing commas are entirely optional – your code will still work without them. The Kotlin style guide encourages the use of trailing commas at the declaration site and leaves it at your discretion for the call site.

To enable trailing commas in the IntelliJ IDEA formatter, go to Settings/Preferences | Editor | Code Style | Kotlin, open the Other tab and select the Use trailing comma option.

**Enumerations﻿**

**Value arguments﻿**

**Class properties and parameters﻿**

**Function value parameters﻿**

**Parameters with optional type (including setters)﻿**

**Indexing suffix﻿**

**Parameters in lambdas﻿**

**when entry﻿**

**Collection literals (in annotations)﻿**

**Type arguments﻿**

**Type parameters﻿**

**Destructuring declarations﻿**

**Documentation comments﻿**

For longer documentation comments, place the opening /\*\* on a separate line and begin each subsequent line with an asterisk:

/\*\*

\* This is a documentation comment

\* on multiple lines.

\*/

Short comments can be placed on a single line:

/\*\* This is a short documentation comment. \*/

Generally, avoid using @param and @return tags. Instead, incorporate the description of parameters and return values directly into the documentation comment, and add links to parameters wherever they are mentioned. Use @param and @return only when a lengthy description is required which doesn't fit into the flow of the main text.

// Avoid doing this:

/\*\*

\* Returns the absolute value of the given number.

\* @param number The number to return the absolute value for.

\* @return The absolute value.

\*/

fun abs(number: Int): Int { /\*...\*/ }

// Do this instead:

/\*\*

\* Returns the absolute value of the given [number].

\*/

fun abs(number: Int): Int { /\*...\*/ }

**Avoid redundant constructs﻿**

In general, if a certain syntactic construction in Kotlin is optional and highlighted by the IDE as redundant, you should omit it in your code. Do not leave unnecessary syntactic elements in code just "for clarity".

**Unit return type﻿**

If a function returns Unit, the return type should be omitted:

fun foo() { // ": Unit" is omitted here

}

**Semicolons﻿**

Omit semicolons whenever possible.

**String templates﻿**

Don't use curly braces when inserting a simple variable into a string template. Use curly braces only for longer expressions.

println("$name has ${children.size} children")

**Idiomatic use of language features﻿**

**Immutability﻿**

Prefer using immutable data to mutable. Always declare local variables and properties as val rather than var if they are not modified after initialization.

Always use immutable collection interfaces (Collection, List, Set, Map) to declare collections which are not mutated. When using factory functions to create collection instances, always use functions that return immutable collection types when possible:

// Bad: use of a mutable collection type for value which will not be mutated

fun validateValue(actualValue: String, allowedValues: HashSet<String>) { ... }

// Good: immutable collection type used instead

fun validateValue(actualValue: String, allowedValues: Set<String>) { ... }

// Bad: arrayListOf() returns ArrayList<T>, which is a mutable collection type

val allowedValues = arrayListOf("a", "b", "c")

// Good: listOf() returns List<T>

val allowedValues = listOf("a", "b", "c")

**Default parameter values﻿**

Prefer declaring functions with default parameter values to declaring overloaded functions.

// Bad

fun foo() = foo("a")

fun foo(a: String) { /\*...\*/ }

// Good

fun foo(a: String = "a") { /\*...\*/ }

**Type aliases﻿**

If you have a functional type or a type with type parameters which is used multiple times in a codebase, prefer defining a type alias for it:

typealias MouseClickHandler = (Any, MouseEvent) -> Unit

typealias PersonIndex = Map<String, Person>

If you use a private or internal type alias for avoiding name collision, prefer the import ... as ... mentioned in [Packages and Imports](https://kotlinlang.org/docs/packages.html).

**Lambda parameters﻿**

In lambdas which are short and not nested, it's recommended to use the it convention instead of declaring the parameter explicitly. In nested lambdas with parameters, always declare parameters explicitly.

**Returns in a lambda﻿**

Avoid using multiple labeled returns in a lambda. Consider restructuring the lambda so that it will have a single exit point. If that's not possible or not clear enough, consider converting the lambda into an anonymous function.

Do not use a labeled return for the last statement in a lambda.

**Named arguments﻿**

Use the named argument syntax when a method takes multiple parameters of the same primitive type, or for parameters of Boolean type, unless the meaning of all parameters is absolutely clear from context.

drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)

**Conditional statements﻿**

Prefer using the expression form of try, if, and when.

return if (x) foo() else bar()

return when(x) {

0 -> "zero"

else -> "nonzero"

}

The above is preferable to:

if (x)

return foo()

else

return bar()

when(x) {

0 -> return "zero"

else -> return "nonzero"

}

**if versus when﻿**

Prefer using if for binary conditions instead of when. For example, use this syntax with if:

if (x == null) ... else ...

Instead of this one with when:

when (x) {

null -> // ...

else -> // ...

}

Prefer using when if there are three or more options.

**Guard conditions in when expression﻿**

Use parentheses when combining multiple boolean expressions in when expressions or statements with [guard conditions](https://kotlinlang.org/docs/control-flow.html#guard-conditions-in-when-expressions):

when (status) {

is Status.Ok if (status.info.isEmpty() || status.info.id == null) -> "no information"

}

Instead of:

when (status) {

is Status.Ok if status.info.isEmpty() || status.info.id == null -> "no information"

}

**Nullable Boolean values in conditions﻿**

If you need to use a nullable Boolean in a conditional statement, use if (value == true) or if (value == false) checks.

**Loops﻿**

Prefer using higher-order functions (filter, map etc.) to loops. Exception: forEach (prefer using a regular for loop instead, unless the receiver of forEach is nullable or forEach is used as part of a longer call chain).

When making a choice between a complex expression using multiple higher-order functions and a loop, understand the cost of the operations being performed in each case and keep performance considerations in mind.

**Loops on ranges﻿**

Use the ..< operator to loop over an open-ended range:

for (i in 0..n - 1) { /\*...\*/ } // bad

for (i in 0..<n) { /\*...\*/ } // good

**Strings﻿**

Prefer string templates to string concatenation.

Prefer multiline strings to embedding \n escape sequences into regular string literals.

To maintain indentation in multiline strings, use trimIndent when the resulting string does not require any internal indentation, or trimMargin when internal indentation is required:

println(**"""**

**Not**

**trimmed**

**text**

**"""**

)

​

println(**"""**

**Trimmed**

**text**

**"""**.trimIndent()

)

​

println()

​

**val** a = **"""Trimmed to margin text:**

**|if(a > 1) {**

**| return a**

**|}"""**.trimMargin()

​

println(a)

[Open in Playground →](https://play.kotlinlang.org/editor/v1/N4Igxg9gJgpiBcIBmBXAdgAgLYEMCWaAFAJQbAA6alGNADgE4EAuANkeSB15jRgHIQm1Xk0ZYsMKMJpMYADyE8aXTpyXFK0hszaEV3XhgAqYiVKUZZC6Rn0gAdKLxYAkmlhomJaRqpLtnrq%2B0gBuOCwYOBgAvLaqHCbOZpYQ2Dj0AOYElvJM8DaGNAA%2BeEiEUQB8GACMpBQWhRhFhvQwTCj0mDgFhUUAvnaOYgCy6VlEwf6MgUQ4k30gADQgTGNtAAosOExIEPRYCCAAVjhhS%2BAQWLR4LDD0AGp3AM54EGiHAEz21fYf1SB9IA%3D%3D)

Target: JVMRunning on v.2.1.21

Learn the difference between [Java and Kotlin multiline strings](https://kotlinlang.org/docs/java-to-kotlin-idioms-strings.html#use-multiline-strings).

**Functions vs properties﻿**

In some scenarios, functions with no arguments might be interchangeable with read-only properties. Although the semantics are similar, there are some stylistic conventions on when to prefer one to another.

Prefer a property over a function when the underlying algorithm:

* Does not throw.
* Is cheap to calculate (or cached on the first run).
* Returns the same result over invocations if the object state hasn't changed.

**Extension functions﻿**

Use extension functions liberally. Every time you have a function that works primarily on an object, consider making it an extension function accepting that object as a receiver. To minimize API pollution, restrict the visibility of extension functions as much as it makes sense. As necessary, use local extension functions, member extension functions, or top-level extension functions with private visibility.

**Infix functions﻿**

Declare a function as infix only when it works on two objects which play a similar role. Good examples: and, to, zip. Bad example: add.

Do not declare a method as infix if it mutates the receiver object.

**Factory functions﻿**

If you declare a factory function for a class, avoid giving it the same name as the class itself. Prefer using a distinct name, making it clear why the behavior of the factory function is special. Only if there is really no special semantics, you can use the same name as the class.

class Point(val x: Double, val y: Double) {

companion object {

fun fromPolar(angle: Double, radius: Double) = Point(...)

}

}

If you have an object with multiple overloaded constructors that don't call different superclass constructors and can't be reduced to a single constructor with default argument values, prefer to replace the overloaded constructors with factory functions.

**Platform types﻿**

A public function/method returning an expression of a platform type must declare its Kotlin type explicitly:

fun apiCall(): String = MyJavaApi.getProperty("name")

Any property (package-level or class-level) initialized with an expression of a platform type must declare its Kotlin type explicitly:

class Person {

val name: String = MyJavaApi.getProperty("name")

}

A local value initialized with an expression of a platform type may or may not have a type declaration:

fun main() {

val name = MyJavaApi.getProperty("name")

println(name)

}

**Scope functions apply/with/run/also/let﻿**

Kotlin provides a set of functions to execute a block of code in the context of a given object: let, run, with, apply, and also. For the guidance on choosing the right scope function for your case, refer to [Scope Functions](https://kotlinlang.org/docs/scope-functions.html).

Scala Coding Standards

[**Indentation**](https://docs.scala-lang.org/style/indentation.html)

Each level of indentation is 2 spaces. Tabs are not used. Thus, instead of indenting like this:

*// wrong!*

**class** **Foo** {

**def** **fourspaces** = {

**val** **x** = 4

..

}

}

You should indent like this:

*// right!*

**class** **Foo** {

**def** **twospaces** = {

**val** **x** = 2

..

}

}

The Scala language encourages a startling amount of nested scopes and logical blocks (function values and such). Do yourself a favor and don’t penalize yourself syntactically for opening up a new block. Coming from Java, this style does take a bit of getting used to, but it is well worth the effort.

Line Wrapping

There are times when a single expression reaches a length where it becomes unreadable to keep it confined to a single line (usually that length is anywhere above 80 characters). In such cases, the *preferred* approach is to simply split the expression up into multiple expressions by assigning intermediate results to values. However, this is not always a practical solution.

When it is absolutely necessary to wrap an expression across more than one line, each successive line should be indented two spaces from the *first*. Also remember that Scala requires each “wrap line” to either have an unclosed parenthetical or to end with an infix method in which the right parameter is not given:

**val** **result** = 1 + 2 + 3 + 4 + 5 + 6 +

7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 +

15 + 16 + 17 + 18 + 19 + 20

Without this trailing method, Scala will infer a semi-colon at the end of a line which was intended to wrap, throwing off the compilation sometimes without even so much as a warning.

Methods with Numerous Arguments

When calling a method which takes numerous arguments (in the range of five or more), it is often necessary to wrap the method invocation onto multiple lines. In such cases, put each argument on a line by itself, indented two spaces from the current indent level:

foo(

someVeryLongFieldName,

andAnotherVeryLongFieldName,

"this is a string",

3.1415)

This way, all parameters line up, but you don’t need to re-align them if you change the name of the method later on.

Great care should be taken to avoid these sorts of invocations well into the length of the line. More specifically, such an invocation should be avoided when each parameter would have to be indented more than 50 spaces to achieve alignment. In such cases, the invocation itself should be moved to the next line and indented two spaces:

*// right!*

**val** **myLongFieldNameWithNoRealPoint** =

foo(

someVeryLongFieldName,

andAnotherVeryLongFieldName,

"this is a string",

3.1415)

*// wrong!*

**val** **myLongFieldNameWithNoRealPoint** = foo(someVeryLongFieldName,

andAnotherVeryLongFieldName,

"this is a string",

3.1415)

Better yet, just try to avoid any method which takes more than two or three parameters!

[**Naming Conventions**](https://docs.scala-lang.org/style/naming-conventions.html)

Generally speaking, Scala uses “camel case” naming. That is, each word is capitalized, except possibly the first word:

UpperCamelCase

lowerCamelCase

Acronyms should be treated as normal words:

xHtml

maxId

instead of:

XHTML

maxID

Underscores in names (\_) are not actually forbidden by the compiler, but are strongly discouraged as they have special meaning within the Scala syntax. (But see below for exceptions.)

Classes/Traits

Classes should be named in upper camel case:

**class** **MyFairLady**

This mimics the Java naming convention for classes.

Sometimes traits and classes as well as their members are used to describe formats, documentation or protocols and generate/derive them. In these cases it is desirable to be close to a 1:1 relation to the output format and the naming conventions don’t apply. In this case, they should only be used for that specific purpose and not throughout the rest of the code.

Objects

Object names are like class names (upper camel case).

An exception is when mimicking a package or function. This isn’t common. Example:

**object** **ast** {

**sealed** **trait** **Expr**

**case** **class** **Plus**(e1: **Expr**, e2: **Expr**) **extends** **Expr**

...

}

**object** **inc** {

**def** **apply**(x: **Int**): **Int** = x + 1

}

Packages

Scala packages should follow the Java package naming conventions:

*// wrong!*

**package** **coolness**

*// right! puts only coolness.\_ in scope*

**package** com.novell.coolness

*// right! puts both novell.\_ and coolness.\_ in scope*

**package** com.novell

**package** **coolness**

*// right, for package object com.novell.coolness*

**package** com.novell

*/\*\**

*\* Provides classes related to coolness*

*\*/*

**package** **object** **coolness** {

}

*root*

It is occasionally necessary to fully-qualify imports using \_root\_. For example if another net is in scope, then to access net.liftweb we must write e.g.:

**import** \_root\_.net.liftweb.**\_**

Do not overuse \_root\_. In general, nested package resolves are a good thing and very helpful in reducing import clutter. Using \_root\_ not only negates their benefit, but also introduces extra clutter in and of itself.

Methods

Textual (alphabetic) names for methods should be in lower camel case:

**def** **myFairMethod** = ...

This section is not a comprehensive guide to idiomatic method naming in Scala. Further information may be found in the method invocation section.

Accessors/Mutators

Scala does *not* follow the Java convention of prepending set/get to mutator and accessor methods (respectively). Instead, the following conventions are used:

* **For accessors of properties, the name of the method should be the name of the property.**
* **In some instances, it is acceptable to prepend “`is`” on a boolean accessor (e.g. isEmpty). This should only be the case when no corresponding mutator is provided. Please note that the**[**Lift**](https://liftweb.net/)**convention of appending “\_?” to boolean accessors is non-standard and not used outside of the Lift framework.**
* **For mutators, the name of the method should be the name of the property with “\_=” appended. As long as a corresponding accessor with that particular property name is defined on the enclosing type, this convention will enable a call-site mutation syntax which mirrors assignment. Note that this is not just a convention but a requirement of the language.**
* **class Foo {**
* **def bar = ...**
* **def bar\_=(bar: Bar) {**
* **...**
* **}**
* **def isBaz = ...**
* **}**
* **val foo = new Foo**
* **foo.bar *// accessor***
* **foo.bar = bar2 *// mutator***
* **foo.isBaz *// boolean property***

Unfortunately, these conventions fall afoul of the Java convention to name the private fields encapsulated by accessors and mutators according to the property they represent. For example:

public **class** **Company** {

**private** **String** name;

public **String** getName() {

**return** name;

}

public void setName(**String** name) {

this.name = name;

}

}

In Scala, there is no distinction between fields and methods. In fact, fields are completely named and controlled by the compiler. If we wanted to adopt the Java convention of bean getters/setters in Scala, this is a rather simple encoding:

**class** **Company** {

**private** **var** **\_name**: **String** = **\_**

**def** **name** = \_name

**def** **name\_=**(name: **String**) {

\_name = name

}

}

While Hungarian notation is terribly ugly, it does have the advantage of disambiguating the \_name variable without cluttering the identifier. The underscore is in the prefix position rather than the suffix to avoid any danger of mistakenly typing name \_ instead of name\_. With heavy use of Scala’s type inference, such a mistake could potentially lead to a very confusing error.

Note that the Java getter/setter paradigm was often used to work around a lack of first class support for Properties and bindings. In Scala, there are libraries that support properties and bindings. The convention is to use an immutable reference to a property class that contains its own getter and setter. For example:

**class** **Company** {

**val** **string**: **Property**[**String**] = **Property**("Initial Value")

Parentheses

Scala allows a parameterless, zero-[arity](https://en.wikipedia.org/wiki/Arity) method to be declared with an empty parameter list:

**def** **foo1**() = ...

or with no parameter lists at all:

**def** **foo2** = ...

By convention, parentheses are used to indicate that a method has side effects, such as altering the receiver.

On the other hand, the absence of parentheses indicates that a method is like an accessor: it returns a value without altering the receiver, and on the same receiver in the same state, it always returns the same answer.

The callsite should follow the declaration; if declared with parentheses, call with parentheses.

These conventions are followed in the Scala standard library and you should follow them in your own code as well.

Additional notes:

* **Scala 3 errors if you leave out the parentheses at the call site. Scala 2 merely warns.**
* **Scala 3 and 2 both error if the call site has parentheses where the definition doesn’t.**
* **Java-defined methods are exempt from this distinction and may be called either way.**
* **If a method *does* take parameters, there isn’t any convention for indicating whether it also has side effects.**
* **Creating an object isn’t considered a side effect. So for example, Scala collections have an iterator method with no parens. Yes, you get a new iterator each time. And yes, iterators are mutable. But every fresh iterator is the same until it has been altered by calling a side-effecting method such as Iterator#next(), which *is* declared with parentheses. See this**[**2018 design discussion**](https://github.com/scala/collection-strawman/issues/520)**.**

Symbolic Method Names

Avoid! Despite the degree to which Scala facilitates this area of API design, the definition of methods with symbolic names should not be undertaken lightly, particularly when the symbols itself are non-standard (for example, >>#>>). As a general rule, symbolic method names have two valid use-cases:

* **Domain-specific languages (e.g. actor1 ! Msg)**
* **Logically mathematical operations (e.g. a + b or c :: d)**

In the former case, symbolic method names may be used with impunity so long as the syntax is actually beneficial. However, in the course of standard API design, symbolic method names should be strictly reserved for purely-functional operations. Thus, it is acceptable to define a >>= method for joining two monads, but it is not acceptable to define a << method for writing to an output stream. The former is mathematically well-defined and side-effect free, while the latter is neither of these.

As a general rule, symbolic method names should be well-understood and self documenting in nature. The rule of thumb is as follows: if you need to explain what the method does, then it should have a real, descriptive name rather than a symbols. There are some *very* rare cases where it is acceptable to invent new symbolic method names. Odds are, your API is not one of those cases!

The definition of methods with symbolic names should be considered an advanced feature in Scala, to be used only by those most well-versed in its pitfalls. Without care, excessive use of symbolic method names can easily transform even the simplest code into symbolic soup.

Constants, Values and Variables

Constant names should be in upper camel case. Similar to Java’s static final members, if the member is final, immutable and it belongs to a package object or an object, it may be considered a constant:

**object** **Container** {

**val** **MyConstant** = ...

}

The value: Pi in scala.math package is another example of such a constant.

Value and variable names should be in lower camel case:

**val** **myValue** = ...

**var** **myVariable**

Type Parameters (generics)

For simple type parameters, a single upper-case letter (from the English alphabet) should be used, starting with A (this is different than the Java convention of starting with T). For example:

**class** **List**[**A**] {

**def** **map**[**B**](f: **A** **=>** **B**): **List**[**B**] = ...

}

If the type parameter has a more specific meaning, a descriptive name should be used, following the class naming conventions (as opposed to an all-uppercase style):

*// Right*

**class** **Map**[**Key**, **Value**] {

**def** **get**(key: **Key**): **Value**

**def** **put**(key: **Key**, value: **Value**): **Unit**

}

*// Wrong; don't use all-caps*

**class** **Map**[**KEY**, **VALUE**] {

**def** **get**(key: **KEY**): **VALUE**

**def** **put**(key: **KEY**, value: **VALUE**): **Unit**

}

If the scope of the type parameter is small enough, a mnemonic can be used in place of a longer, descriptive name:

**class** **Map**[**K**, **V**] {

**def** **get**(key: **K**): **V**

**def** **put**(key: **K**, value: **V**): **Unit**

}

Higher-Kinds and Parameterized Type parameters

Higher-kinds are theoretically no different from regular type parameters (except that their [kind](https://en.wikipedia.org/wiki/Kind_(type_theory)) is at least \*=>\* rather than simply \*). The naming conventions are generally similar, however it is preferred to use a descriptive name rather than a single letter, for clarity:

**class** **HigherOrderMap**[**Key**[**\_**], **Value**[**\_**]] { ... }

The single letter form is (sometimes) acceptable for fundamental concepts used throughout a codebase, such as F[\_] for Functor and M[\_] for Monad.

In such cases, the fundamental concept should be something well known and understood to the team, or have tertiary evidence, such as the following:

**def** **doSomething**[**M**[**\_**]: **Monad**](**m**: **M**[**Int**]) = ...

Here, the type bound : Monad offers the necessary evidence to inform the reader that M[\_] is the type of the Monad.

Annotations

Annotations, such as @volatile should be in lower camel case:

**class** **cloneable** **extends** **StaticAnnotation**

This convention is used throughout the Scala library, even though it is not consistent with Java annotation naming.

Note: This convention applied even when using type aliases on annotations. For example, when using JDBC:

**type** **id** = javax.persistence.**Id** **@annotation.target.field**

**@id**

**var** **id**: **Int** = 0

Special Note on Brevity

Because of Scala’s roots in the functional languages, it is quite normal for local names to be very short:

**def** **add**(a: **Int**, b: **Int**) = a + b

This convention works because properly-written Scala methods are quite short, only spanning a single expression and rarely going beyond a few lines. Few local names are used (including parameters), and so there is no need to contrive long, descriptive names. This convention substantially improves the brevity of most Scala sources. This in turn improves readability, as most expressions fit in one line and the arguments to methods have descriptive type names.

This convention only applies to parameters of very simple methods (and local fields for very simply classes); everything in the public interface should be descriptive. Also note that the names of arguments are now part of the public API of a class, since users can use named parameters in method calls.

[**Types**](https://docs.scala-lang.org/style/types.html)

Inference

Use type inference where possible, but put clarity first, and favour explicitness in public APIs.

You should almost never annotate the type of a private field or a local variable, as their type will usually be immediately evident in their value:

**private** **val** **name** = "Daniel"

However, you may wish to still display the type where the assigned value has a complex or non-obvious form.

All public methods should have explicit type annotations. Type inference may break encapsulation in these cases, because it depends on internal method and class details. Without an explicit type, a change to the internals of a method or val could alter the public API of the class without warning, potentially breaking client code. Explicit type annotations can also help to improve compile times.

Function Values

Function values support a special case of type inference which is worth calling out on its own:

**val** **ls**: **List**[**String**] = ...

ls.map(str **=>** str.toInt)

In cases where Scala already knows the type of the function value we are declaring, there is no need to annotate the parameters (in this case, str). This is an intensely helpful inference and should be preferred whenever possible. Note that implicit conversions which operate on function values will nullify this inference, forcing the explicit annotation of parameter types.

Annotations

Type annotations should be patterned according to the following template:

value: Type

This is the style adopted by most of the Scala standard library and all of Martin Odersky’s examples. The space between value and type helps the eye in accurately parsing the syntax. The reason to place the colon at the end of the value rather than the beginning of the type is to avoid confusion in cases such as this one:

value :::

This is actually valid Scala, declaring a value to be of type ::. Obviously, the prefix-style annotation colon muddles things greatly.

Ascription

Type ascription is often confused with type annotation, as the syntax in Scala is identical. The following are examples of ascription:

* **Nil: List[String]**
* **Set(values: \_\*)**
* **"Daniel": AnyRef**

Ascription is basically just an up-cast performed at compile-time for the sake of the type checker. Its use is not common, but it does happen on occasion. The most often seen case of ascription is invoking a varargs method with a single Seq parameter. This is done by ascribing the \_\* type (as in the second example above).

Ascription follows the type annotation conventions; a space follows the colon.

Functions

Function types should be declared with a space between the parameter type, the arrow and the return type:

**def** **foo**(f: **Int** **=>** **String**) = ...

**def** **bar**(f: (**Boolean**, **Double**) **=>** **List**[**String**]) = ...

Parentheses should be omitted wherever possible (e.g. methods of arity-1, such as Int => String).

Arity-1

Scala has a special syntax for declaring types for functions of arity-1. For example:

**def** **map**[**B**](f: **A** **=>** **B**) = ...

Specifically, the parentheses may be omitted from the parameter type. Thus, we did *not* declare f to be of type (A) => B, as this would have been needlessly verbose. Consider the more extreme example:

*// wrong!*

**def** **foo**(f: (**Int**) **=>** (**String**) **=>** (**Boolean**) **=>** **Double**) = ...

*// right!*

**def** **foo**(f: **Int** **=>** **String** **=>** **Boolean** **=>** **Double**) = ...

By omitting the parentheses, we have saved six whole characters and dramatically improved the readability of the type expression.

Structural Types

Structural types should be declared on a single line if they are less than 50 characters in length. Otherwise, they should be split across multiple lines and (usually) assigned to their own type alias:

*// wrong!*

**def** **foo**(a: { **def** bar(a: **Int**, b: **Int**): **String**; **val** **baz**: **List**[**String** **=>** **String**] }) = ...

*// right!*

**private** **type** **FooParam** = {

**val** **baz**: **List**[**String** **=>** **String**]

**def** **bar**(a: **Int**, b: **Int**): **String**

}

**def** **foo**(a: **FooParam**) = ...

Simpler structural types (under 50 characters) may be declared and used inline:

**def** **foo**(a: { **val** bar: **String** }) = ...

When declaring structural types inline, each member should be separated by a semi-colon and a single space, the opening brace should be *followed* by a space while the closing brace should be *preceded* by a space (as demonstrated in both examples above).

Structural types are implemented with reflection at runtime, and are inherently less performant than nominal types. Developers should prefer the use of nominal types, unless structural types provide a clear benefit.

[**Nested Blocks**](https://docs.scala-lang.org/style/nested-blocks.html)

Curly Braces

Opening curly braces ({) must be on the same line as the declaration they represent:

**def** **foo** = {

...

}

Technically, Scala’s parser *does* support GNU-style notation with opening braces on the line following the declaration. However, the parser is not terribly predictable when dealing with this style due to the way in which semi-colon inference is implemented. Many headaches will be saved by simply following the curly brace convention demonstrated above.

Parentheses

In the rare cases when parenthetical blocks wrap across lines, the opening and closing parentheses should be unspaced and generally kept on the same lines as their content (Lisp-style):

(this + is a very ++ long \*

expression)

Parentheses also serve to disable semicolon inference, and so allow the developer to start lines with operators, which some prefer:

( someCondition

|| someOtherCondition

|| thirdCondition

)

A trailing parenthesis on the following line is acceptable in this case, for aesthetic reasons.

[**Declarations**](https://docs.scala-lang.org/style/declarations.html)

Classes

Class, object, and trait constructors should be declared all on one line, unless the line becomes “too long” (about 100 characters). In that case, put each constructor argument on its own line with [trailing commas](https://docs.scala-lang.org/sips/trailing-commas.html#motivation):

**class** **Person**(name: **String**, age: **Int**) {

…

}

**class** **Person**(

name: **String**,

age: **Int**,

birthdate: **Date**,

astrologicalSign: **String**,

shoeSize: **Int**,

favoriteColor: java.awt.**Color**,

) {

**def** **firstMethod**: **Foo** = …

}

If a class/object/trait extends anything, the same general rule applies, put it on one line unless it goes over about 100 characters, and then put each item on its own line with [trailing commas](https://docs.scala-lang.org/sips/trailing-commas.html#motivation); closing parenthesis provides visual separation between constructor arguments and extensions; empty line should be added to further separate extensions from class implementation:

**class** **Person**(

name: **String**,

age: **Int**,

birthdate: **Date**,

astrologicalSign: **String**,

shoeSize: **Int**,

favoriteColor: java.awt.**Color**,

) **extends** **Entity**

**with** **Logging**

**with** **Identifiable**

**with** **Serializable** {

**def** **firstMethod**: **Foo** = …

}

Ordering Of Class Elements

All class/object/trait members should be declared interleaved with newlines. The only exceptions to this rule are var and val. These may be declared without the intervening newline, but only if none of the fields have Scaladoc and if all of the fields have simple (max of 20-ish chars, one line) definitions:

**class** **Foo** {

**val** **bar** = 42

**val** **baz** = "Daniel"

**def** **doSomething**(): **Unit** = { ... }

**def** **add**(x: **Int**, y: **Int**): **Int** = x + y

}

Fields should *precede* methods in a scope. The only exception is if the val has a block definition (more than one expression) and performs operations which may be deemed “method-like” (e.g. computing the length of a List). In such cases, the non-trivial val may be declared at a later point in the file as logical member ordering would dictate. This rule *only* applies to val and lazy val! It becomes very difficult to track changing aliases if var declarations are strewn throughout class file.

Methods

Methods should be declared according to the following pattern:

**def** **foo**(bar: **Baz**): **Bin** = expr

Methods with default parameter values should be declared in an analogous fashion, with a space on either side of the equals sign:

**def** **foo**(x: **Int** = 6, y: **Int** = 7): **Int** = x + y

You should specify a return type for all public members. Consider it documentation checked by the compiler. It also helps in preserving binary compatibility in the face of changing type inference (changes to the method implementation may propagate to the return type if it is inferred).

Local methods or private methods may omit their return type:

**private** **def** **foo**(x: **Int** = 6, y: **Int** = 7) = x + y

**Procedure Syntax**

Avoid the (now deprecated) procedure syntax, as it tends to be confusing for very little gain in brevity.

*// don't do this*

**def** **printBar**(bar: **Baz**) {

println(bar)

}

*// write this instead*

**def** **printBar**(bar: **Bar**): **Unit** = {

println(bar)

}

**Modifiers**

Method modifiers should be given in the following order (when each is applicable):

1. Annotations, *each on their own line*
2. Override modifier (override)
3. Access modifier (protected, private)
4. Implicit modifier (implicit)
5. Final modifier (final)
6. def

**@Transaction**

**@throws(classOf[IOException])**

**override** **protected** **final** **def** **foo**(): **Unit** = {

...

}

**Body**

When a method body comprises a single expression which is less than 30 (or so) characters, it should be given on a single line with the method:

**def** **add**(a: **Int**, b: **Int**): **Int** = a + b

When the method body is a single expression *longer* than 30 (or so) characters but still shorter than 70 (or so) characters, it should be given on the following line, indented two spaces:

**def** **sum**(ls: **List**[**String**]): **Int** =

ls.map(**\_**.toInt).foldLeft(0)(**\_** + **\_**)

The distinction between these two cases is somewhat artificial. Generally speaking, you should choose whichever style is more readable on a case-by-case basis. For example, your method declaration may be very long, while the expression body may be quite short. In such a case, it may be more readable to put the expression on the next line rather than making the declaration line too long.

When the body of a method cannot be concisely expressed in a single line or is of a non-functional nature (some mutable state, local or otherwise), the body must be enclosed in braces:

**def** **sum**(ls: **List**[**String**]): **Int** = {

**val** **ints** = ls.map(**\_**.toInt)

ints.foldLeft(0)(**\_** + **\_**)

}

Methods which contain a single match expression should be declared in the following way:

*// right!*

**def** **sum**(ls: **List**[**Int**]): **Int** = ls **match** {

**case** hd :: tail **=>** hd + sum(tail)

**case** **Nil** **=>** 0

}

*Not* like this:

*// wrong!*

**def** **sum**(ls: **List**[**Int**]): **Int** = {

ls **match** {

**case** hd :: tail **=>** hd + sum(tail)

**case** **Nil** **=>** 0

}

}

**Multiple Parameter Lists**

In general, you should only use multiple parameter lists if there is a good reason to do so. These methods (or similarly declared functions) have a more verbose declaration and invocation syntax and are harder for less-experienced Scala developers to understand.

There are three main reasons you should do this:

1. For a fluent API

Multiple parameter lists allow you to create your own “control structures”:

**def** **unless**(exp: **Boolean**)(code: **=>** **Unit**): **Unit** =

**if** (!exp) code

unless(x < 5) {

println("x was not less than five")

}

1. Implicit Parameters

When using implicit parameters, and you use the implicit keyword, it applies to the entire parameter list. Thus, if you want only some parameters to be implicit, you must use multiple parameter lists.

1. For type inference

When invoking a method using only some of the parameter lists, the type inferencer can allow a simpler syntax when invoking the remaining parameter lists. Consider fold:

**def** **foldLeft**[**B**](z: **B**)(op: (**B**, **A**) **=>** **B**): **B**

**List**("").foldLeft(0)(**\_** + **\_**.length)

*// If, instead:*

**def** **foldLeft**[**B**](z: **B**, op: (**B**, **A**) **=>** **B**): **B**

*// above won't work, you must specify types*

**List**("").foldLeft(0, (b: **Int**, a: **String**) **=>** b + a.length)

**List**("").foldLeft[**Int**](0, **\_** + **\_**.length)

For complex DSLs, or with type names that are long, it can be difficult to fit the entire signature on one line. For those cases there are several different styles in use:

1. Split the parameter lists, one parameter per line with [trailing commas](https://docs.scala-lang.org/sips/trailing-commas.html#motivation) and parentheses being on separate lines adding to visual separation between the lists:
2. **protected** **def** **forResource**(
3. resourceInfo: **Any**,
4. )(
5. f: (**JsonNode**) **=>** **Any**,
6. )(**implicit**
7. urlCreator: **URLCreator**,
8. configurer: **OAuthConfiguration**,
9. ): **Any** = {
10. ...
11. }
12. Or align the open-paren of the parameter lists, one list per line:
13. **protected** **def** **forResource**(resourceInfo: **Any**)
14. (f: (**JsonNode**) **=>** **Any**)
15. (**implicit** urlCreator: **URLCreator**, configurer: **OAuthConfiguration**): **Any** = {
16. ...
17. }

**Higher-Order Functions**

It’s worth keeping in mind when declaring higher-order functions the fact that Scala allows a somewhat nicer syntax for such functions at call-site when the function parameter is curried as the last argument. For example, this is the foldl function in SML:

fun foldl (f: ('b \* 'a) -> 'b) (init: 'b) (ls: 'a list) = ...

In Scala, the preferred style is the exact inverse:

**def** **foldLeft**[**A**, **B**](ls: **List**[**A**])(init: **B**)(f: (**B**, **A**) **=>** **B**): **B** = ...

By placing the function parameter *last*, we have enabled invocation syntax like the following:

foldLeft(**List**(1, 2, 3, 4))(0)(**\_** + **\_**)

The function value in this invocation is not wrapped in parentheses; it is syntactically quite disconnected from the function itself (foldLeft). This style is preferred for its brevity and cleanliness.

Fields

Fields should follow the declaration rules for methods, taking special note of access modifier ordering and annotation conventions.

Lazy vals should use the lazy keyword directly before the val:

**private** **lazy** **val** **foo** = bar()

Function Values

Scala provides a number of different syntactic options for declaring function values. For example, the following declarations are exactly equivalent:

1. val f1 = ((a: Int, b: Int) => a + b)
2. val f2 = (a: Int, b: Int) => a + b
3. val f3 = (\_: Int) + (\_: Int)
4. val f4: (Int, Int) => Int = (\_ + \_)

Of these styles, (1) and (4) are to be preferred at all times. (2) appears shorter in this example, but whenever the function value spans multiple lines (as is normally the case), this syntax becomes extremely unwieldy. Similarly, (3) is concise, but obtuse. It is difficult for the untrained eye to decipher the fact that this is even producing a function value.

When styles (1) and (4) are used exclusively, it becomes very easy to distinguish places in the source code where function values are used. Both styles make use of parentheses, since they look clean on a single line.

Spacing

There should be no space between parentheses and the code they contain. Curly braces should be separated from the code within them by a one-space gap, to give the visually busy braces “breathing room”.

Multi-Expression Functions

Most function values are less trivial than the examples given above. Many contain more than one expression. In such cases, it is often more readable to split the function value across multiple lines. When this happens, only style (1) should be used, substituting braces for parentheses. Style (4) becomes extremely difficult to follow when enclosed in large amounts of code. The declaration itself should loosely follow the declaration style for methods, with the opening brace on the same line as the assignment or invocation, while the closing brace is on its own line immediately following the last line of the function. Parameters should be on the same line as the opening brace, as should the “arrow” (=>):

**val** **f1** = { (a: **Int**, b: **Int**) **=>**

**val** **sum** = a + b

sum

}

As noted earlier, function values should leverage type inference whenever possible.

[**Control Structures**](https://docs.scala-lang.org/style/control-structures.html)

All control structures should be written with a space following the defining keyword:

*// right!*

**if** (foo) bar **else** baz

**for** (i **<-** 0 to 10) { ... }

**while** (true) { println("Hello, World!") }

*// wrong!*

**if**(foo) bar **else** baz

**for**(i **<-** 0 to 10) { ... }

**while**(true) { println("Hello, World!") }

Curly-Braces

Curly-braces should be omitted in cases where the control structure represents a pure-functional operation and all branches of the control structure (relevant to if/else) are single-line expressions. Remember the following guidelines:

* **if - Omit braces if you have an else clause. Otherwise, surround the contents with curly braces even if the contents are only a single line.**
* **while - Never omit braces (while cannot be used in a pure-functional manner).**
* **for - Omit braces if you have a yield clause. Otherwise, surround the contents with curly-braces, even if the contents are only a single line.**
* **case - Always omit braces in case clauses.**

**val** **news** = **if** (foo)

goodNews()

**else**

badNews()

**if** (foo) {

println("foo was true")

}

news **match** {

**case** "good" **=>** println("Good news!")

**case** "bad" **=>** println("Bad news!")

}

Comprehensions

Scala has the ability to represent for-comprehensions with more than one generator (usually, more than one <- symbol). In such cases, there are two alternative syntaxes which may be used:

*// wrong!*

**for** (x **<-** board.rows; y **<-** board.files)

**yield** (x, y)

*// right!*

**for** {

x **<-** board.rows

y **<-** board.files

} **yield** (x, y)

While the latter style is more verbose, it is generally considered easier to read and more “scalable” (meaning that it does not become obfuscated as the complexity of the comprehension increases). You should prefer this form for all for-comprehensions of more than one generator. Comprehensions with only a single generator (e.g. for (i <- 0 to 10) yield i) should use the first form (parentheses rather than curly braces).

The exceptions to this rule are for-comprehensions which lack a yield clause. In such cases, the construct is actually a loop rather than a functional comprehension and it is usually more readable to string the generators together between parentheses rather than using the syntactically-confusing } { construct:

*// wrong!*

**for** {

x **<-** board.rows

y **<-** board.files

} {

printf("(%d, %d)", x, y)

}

*// right!*

**for** (x **<-** board.rows; y **<-** board.files) {

printf("(%d, %d)", x, y)

}

Finally, for comprehensions are preferred to chained calls to map, flatMap, and filter, as this can get difficult to read (this is one of the purposes of the enhanced for comprehension).

Trivial Conditionals

There are certain situations where it is useful to create a short if/else expression for nested use within a larger expression. In Java, this sort of case would traditionally be handled by the ternary operator (?/:), a syntactic device which Scala lacks. In these situations (and really any time you have a extremely brief if/else expression) it is permissible to place the “then” and “else” branches on the same line as the if and else keywords:

**val** **res** = **if** (foo) bar **else** baz

The key here is that readability is not hindered by moving both branches inline with the if/else. Note that this style should never be used with imperative if expressions nor should curly braces be employed.

[**Method Invocation**](https://docs.scala-lang.org/style/method-invocation.html)

Generally speaking, method invocation in Scala follows Java conventions. In other words, there should not be a space between the invocation target and the dot (.), nor a space between the dot and the method name, nor should there be any space between the method name and the argument-delimiters (parentheses). Each argument should be separated by a single space *following* the comma (,):

foo(42, bar)

target.foo(42, bar)

target.foo()

As of version 2.8, Scala now has support for named parameters. Named parameters in a method invocation should be treated as regular parameters (spaced accordingly following the comma) with a space on either side of the equals sign:

foo(x = 6, y = 7)

While this style does create visual ambiguity with named parameters and variable assignment, the alternative (no spacing around the equals sign) results in code which can be very difficult to read, particularly for non-trivial expressions for the actuals.

Arity-0

Scala allows the omission of parentheses on methods of arity-0 (no arguments):

reply()

*// is the same as*

reply

However, this syntax should *only* be used when the method in question has no side-effects (purely-functional). In other words, it would be acceptable to omit parentheses when calling queue.size, but not when calling println(). This convention mirrors the method declaration convention given above.

Observing this convention improves code readability and will make it much easier to understand at a glance the most basic operation of any given method. Resist the urge to omit parentheses simply to save two characters!

Arity-1 (Infix Notation)

Scala has a special punctuation-free syntax for invoking methods of arity-1 (one argument). This should generally be avoided, but with the following exceptions for operators and higher-order functions. In these cases it should only be used for purely-functional methods (methods with no side-effects).

*// recommended*

names.mkString(",")

*// also sometimes seen; controversial*

names mkString ","

*// wrong - has side-effects*

javaList add item

Symbolic Methods/Operators

Symbolic methods (operators) should always be invoked using infix notation with spaces separating the target, the operator, and the parameter:

*// right!*

"daniel" + " " + "spiewak"

a + b

*// wrong!*

"daniel"+" "+"spiewak"

a+b

a.+(b)

For the most part, this idiom follows Java and Haskell syntactic conventions. A gray area is short, operator-like methods like max, especially if commutative:

*// fairly common*

a max b

Symbolic methods which take more than one parameter are discouraged. When they exist, they may still be invoked using infix notation, delimited by spaces:

foo \*\* (bar, baz)

Such methods are fairly rare, however, and should normally be avoided during API design. For example, the use of the (now deprecated) /: and :\ methods should be avoided in preference to their better-known names, foldLeft and foldRight.

Higher-Order Functions

Invoking higher-order functions may use parens or braces, but in either case, use dot notation and omit any space after the method name:

names.map(**\_**.toUpperCase)

These are not recommended:

*// wrong! missing dot*

names map (**\_**.toUpperCase)

*// wrong! extra space*

names.map (**\_**.toUpperCase)

Experience has shown that these styles make code harder to read, especially when multiple such method calls are chained.

[**Files**](https://docs.scala-lang.org/style/files.html)

The unit of work for the compiler is a “compilation unit”, which is usually just an ordinary file. The Scala language places few restrictions on how code is organized across files. The definition of a class, or equivalently a trait or object, can’t be split over multiple files, so it must be contained within a single file. A class and its companion object must be defined together in the same file. A sealed class can be extended only in the same file, so all its subclasses must be defined there.

Similarly, there are no restrictions on the name of a source file or where it is located in the file system, although certain conventions are broadly honored in practice. Generally, the file is named after the class it contains, or if it has more than one class, the parent class.

For example, a file, Inbox.scala, is expected to contain Inbox and its companion:

**package** org.coolness

**class** **Inbox** { ... }

*// companion object*

**object** **Inbox** { ... }

The file may be located in a directory, org/coolness, following Java tooling conventions, but this is at the discretion of the developer and for their convenience.

It is natural to put the following Option ADT in a file, Option.scala:

**sealed** **trait** **Option**[**+A**]

**case** **class** **Some**[**A**](a: **A**) **extends** **Option**[**A**]

**case** **object** **None** **extends** **Option**[**Nothing**]

The related elements, Some and None, are easily discoverable, even in the absence of tooling.

When unrelated classes are grouped together, perhaps because they implement a feature or a model a domain, the source file receives a descriptive camelCase name. Some prefer this naming scheme for top-level terms. For example, object project would be found in project.scala. Similarly, a package object defined as package object model is located in package.scala in the model source directory.

Files created just for quick testing can have arbitrary names, such as demo-bug.scala.

[**Scaladoc**](https://docs.scala-lang.org/style/scaladoc.html)

It is important to provide documentation for all packages, classes, traits, methods, and other members. Scaladoc generally follows the conventions of Javadoc, but provides many additional features that simplify writing documentation for Scala code.

In general, you want to worry more about substance and writing style than about formatting. Scaladocs need to be useful to new users of the code as well as experienced users. Achieving this is very simple: increase the level of detail and explanation as you write, starting from a terse summary (useful for experienced users as reference), while providing deeper examples in the detailed sections (which can be ignored by experienced users, but can be invaluable for newcomers).

The Scaladoc tool does not mandate a documentation comment style.

The following examples demonstrate a single line summary followed by detailed documentation, in the three common styles of indentation.

Javadoc style:

*/\*\**

*\* Provides a service as described.*

*\**

*\* This is further documentation of what we're documenting.*

*\* Here are more details about how it works and what it does.*

*\*/*

**def** **member**: **Unit** = ()

Scaladoc style, with gutter asterisks aligned in column two:

*/\*\* Provides a service as described.*

*\**

*\* This is further documentation of what we're documenting.*

*\* Here are more details about how it works and what it does.*

*\*/*

**def** **member**: **Unit** = ()

Scaladoc style, with gutter asterisks aligned in column three:

*/\*\* Provides a service as described.*

*\**

*\* This is further documentation of what we're documenting.*

*\* Here are more details about how it works and what it does.*

*\*/*

**def** **member**: **Unit** = ()

Because the comment markup is sensitive to whitespace, the tool must be able to infer the left margin.

When only a simple, short description is needed, a one-line format can be used:

*/\*\* Does something very simple \*/*

**def** **simple**: **Unit** = ()

Note that, in contrast to the Javadoc convention, the text in the Scaladoc styles begins on the first line of the comment. This format saves vertical space in the source file.

In either Scaladoc style, all lines of text are aligned on column five. Since Scala source is usually indented by two spaces, the text aligns with source indentation in a way that is visually pleasing.

See [Scaladoc for Library Authors](https://docs.scala-lang.org/overviews/scaladoc/for-library-authors.html) for more technical info on formatting Scaladoc.

General Style

It is important to maintain a consistent style with Scaladoc. It is also important to target Scaladoc to both those unfamiliar with your code and experienced users who just need a quick reference. Here are some general guidelines:

* **Get to the point as quickly as possible. For example, say “returns true if some condition” instead of “if some condition return true”.**
* **Try to format the first sentence of a method as “Returns XXX”, as in “Returns the first element of the List”, as opposed to “this method returns” or “get the first” etc. Methods typically return things.**
* **This same goes for classes; omit “This class does XXX”; just say “Does XXX”**
* **Create links to referenced Scala Library classes using the square-bracket syntax, e.g. [[scala.Option]]**
* **Summarize a method’s return value in the @return annotation, leaving a longer description for the main Scaladoc.**
* **If the documentation of a method is a one line description of what that method returns, do not repeat it with an @return annotation.**
* **Document what the method *does do* not what the method *should do*. In other words, say “returns the result of applying f to x” rather than “return the result of applying f to x”. Subtle, but important.**
* **When referring to the instance of the class, use “this XXX”, or “this” and not “the XXX”. For objects, say “this object”.**
* **Make code examples consistent with this guide.**
* **Use the wiki-style syntax instead of HTML wherever possible.**
* **Examples should use either full code listings or the REPL, depending on what is needed (the simplest way to include REPL code is to develop the examples in the REPL and paste it into the Scaladoc).**
* **Make liberal use of @macro to refer to commonly-repeated values that require special formatting.**

Packages

Provide Scaladoc for each package. This goes in a file named package.scala in your package’s directory and looks like so (for the package parent.package.name.mypackage):

**package** parent.**package**.name

*/\*\* This is the Scaladoc for the package. \*/*

**package** **object** **mypackage** {

}

A package’s documentation should first document what sorts of classes are part of the package. Secondly, document the general sorts of things the package object itself provides.

While package documentation doesn’t need to be a full-blown tutorial on using the classes in the package, it should provide an overview of the major classes, with some basic examples of how to use the classes in that package. Be sure to reference classes using the square-bracket notation:

**package** my.**package**

*/\*\* Provides classes for dealing with complex numbers. Also provides*

*\* implicits for converting to and from `Int`.*

*\**

*\* ==Overview==*

*\* The main class to use is [[my.package.complex.Complex]], as so*

*\* {{{*

*\* scala> val complex = Complex(4,3)*

*\* complex: my.package.complex.Complex = 4 + 3i*

*\* }}}*

*\**

*\* If you include [[my.package.complex.ComplexConversions]], you can*

*\* convert numbers more directly*

*\* {{{*

*\* scala> import my.package.complex.ComplexConversions.\_*

*\* scala> val complex = 4 + 3.i*

*\* complex: my.package.complex.Complex = 4 + 3i*

*\* }}}*

*\*/*

*package complex {}*

Classes, Objects, and Traits

Document all classes, objects, and traits. The first sentence of the Scaladoc should provide a summary of what the class or trait does. Document all type parameters with @tparam.

**Classes**

If a class should be created using its companion object, indicate as such after the description of the class (though leave the details of construction to the companion object). Unfortunately, there is currently no way to create a link to the companion object inline, however the generated Scaladoc will create a link for you in the class documentation output.

If the class should be created using a constructor, document it using the @constructor syntax:

*/\*\* A person who uses our application.*

*\**

*\* @constructor create a new person with a name and age.*

*\* @param name the person's name*

*\* @param age the person's age in years*

*\*/*

**class** **Person**(name: **String**, age: **Int**) {

}

Depending on the complexity of your class, provide an example of common usage.

**Objects**

Since objects can be used for a variety of purposes, it is important to document *how* to use the object (e.g. as a factory, for implicit methods). If this object is a factory for other objects, indicate as such here, deferring the specifics to the Scaladoc for the apply method(s). If your object *doesn’t* use apply as a factory method, be sure to indicate the actual method names:

*/\*\* Factory for [[mypackage.Person]] instances. \*/*

**object** **Person** {

*/\*\* Creates a person with a given name and age.*

*\**

*\* @param name their name*

*\* @param age the age of the person to create*

*\*/*

**def** **apply**(name: **String**, age: **Int**) = {}

*/\*\* Creates a person with a given name and birthdate*

*\**

*\* @param name their name*

*\* @param birthDate the person's birthdate*

*\* @return a new Person instance with the age determined by the*

*\* birthdate and current date.*

*\*/*

**def** **apply**(name: **String**, birthDate: java.time.**LocalDate**) = {}

}

If your object holds implicit conversions, provide an example in the Scaladoc:

*/\*\* Implicit conversions and helpers for [[mypackage.Complex]] instances.*

*\**

*\* {{{*

*\* import ComplexImplicits.\_*

*\* val c: Complex = 4 + 3.i*

*\* }}}*

*\*/*

*object ComplexImplicits {}*

**Traits**

After the overview of what the trait does, provide an overview of the methods and types that must be specified in classes that mix in the trait. If there are known classes using the trait, reference them.

Methods and Other Members

Document all methods. As with other documentable entities, the first sentence should be a summary of what the method does. Subsequent sentences explain in further detail. Document each parameter as well as each type parameter (with @tparam). For curried functions, consider providing more detailed examples regarding the expected or idiomatic usage. For implicit parameters, take special care to explain where these parameters will come from and if the user needs to do any extra work to make sure the parameters will be available.

PHP Coding Standards

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](http://tools.ietf.org/html/rfc2119).

Overview

This specification extends, expands and replaces [PSR-2](https://www.php-fig.org/psr/psr-2/), the coding style guide and requires adherence to [PSR-1](https://www.php-fig.org/psr/psr-1/), the basic coding standard.

Like [PSR-2](https://www.php-fig.org/psr/psr-2/), the intent of this specification is to reduce cognitive friction when scanning code from different authors. It does so by enumerating a shared set of rules and expectations about how to format PHP code. This PSR seeks to provide a set way that coding style tools can implement, projects can declare adherence to and developers can easily relate to between different projects. When various authors collaborate across multiple projects, it helps to have one set of guidelines to be used among all those projects. Thus, the benefit of this guide is not in the rules themselves but the sharing of those rules.

[PSR-2](https://www.php-fig.org/psr/psr-2/) was accepted in 2012 and since then a number of changes have been made to PHP which has implications for coding style guidelines. Whilst [PSR-2](https://www.php-fig.org/psr/psr-2/) is very comprehensive of PHP functionality that existed at the time of writing, new functionality is very open to interpretation. This PSR, therefore, seeks to clarify the content of PSR-2 in a more modern context with new functionality available, and make the errata to PSR-2 binding.

**Previous language versions**

Throughout this document, any instructions MAY be ignored if they do not exist in versions of PHP supported by your project.

**Example**

This example encompasses some of the rules below as a quick overview:

<?php

**declare**(strict\_types=1);

**namespace** **Vendor**\**Package**;

**use** **Vendor**\**Package**\{**ClassA** **as** **A**, **ClassB**, **ClassC** **as** **C**};

**use** **Vendor**\**Package**\**SomeNamespace**\**ClassD** **as** **D**;

**use** **function** **Vendor**\**Package**\{**functionA**, **functionB**, **functionC**};

**use** **const** **Vendor**\**Package**\{**ConstantA**, **ConstantB**, **ConstantC**};

**class** **Foo** **extends** **Bar** **implements** **FooInterface**

{

**public** **function** **sampleFunction**(int $a, int $b = null): **array**

{

**if** ($a === $b) {

bar();

} **elseif** ($a > $b) {

$foo->bar($arg1);

} **else** {

BazClass::bar($arg2, $arg3);

}

}

**final** **public** **static** **function** **bar**()

{

// method body

}

}

2. General

**2.1 Basic Coding Standard**

Code MUST follow all rules outlined in [PSR-1](https://www.php-fig.org/psr/psr-1/).

The term 'StudlyCaps' in PSR-1 MUST be interpreted as PascalCase where the first letter of each word is capitalized including the very first letter.

**2.2 Files**

All PHP files MUST use the Unix LF (linefeed) line ending only.

All PHP files MUST end with a non-blank line, terminated with a single LF.

The closing ?> tag MUST be omitted from files containing only PHP.

**2.3 Lines**

There MUST NOT be a hard limit on line length.

The soft limit on line length MUST be 120 characters.

Lines SHOULD NOT be longer than 80 characters; lines longer than that SHOULD be split into multiple subsequent lines of no more than 80 characters each.

There MUST NOT be trailing whitespace at the end of lines.

Blank lines MAY be added to improve readability and to indicate related blocks of code except where explicitly forbidden.

There MUST NOT be more than one statement per line.

**2.4 Indenting**

Code MUST use an indent of 4 spaces for each indent level, and MUST NOT use tabs for indenting.

**2.5 Keywords and Types**

All PHP reserved keywords and types [[1]](http://php.net/manual/en/reserved.keywords.php)[[2]](http://php.net/manual/en/reserved.other-reserved-words.php) MUST be in lower case.

Any new types and keywords added to future PHP versions MUST be in lower case.

Short form of type keywords MUST be used i.e. bool instead of boolean, int instead of integer etc.

3. Declare Statements, Namespace, and Import Statements

The header of a PHP file may consist of a number of different blocks. If present, each of the blocks below MUST be separated by a single blank line, and MUST NOT contain a blank line. Each block MUST be in the order listed below, although blocks that are not relevant may be omitted.

* Opening <?php tag.
* File-level docblock.
* One or more declare statements.
* The namespace declaration of the file.
* One or more class-based use import statements.
* One or more function-based use import statements.
* One or more constant-based use import statements.
* The remainder of the code in the file.

When a file contains a mix of HTML and PHP, any of the above sections may still be used. If so, they MUST be present at the top of the file, even if the remainder of the code consists of a closing PHP tag and then a mixture of HTML and PHP.

When the opening <?php tag is on the first line of the file, it MUST be on its own line with no other statements unless it is a file containing markup outside of PHP opening and closing tags.

Import statements MUST never begin with a leading backslash as they must always be fully qualified.

The following example illustrates a complete list of all blocks:

<?php

/\*\*

\* This file contains an example of coding styles.

\*/

**declare**(strict\_types=1);

**namespace** **Vendor**\**Package**;

**use** **Vendor**\**Package**\{**ClassA** **as** **A**, **ClassB**, **ClassC** **as** **C**};

**use** **Vendor**\**Package**\**SomeNamespace**\**ClassD** **as** **D**;

**use** **Vendor**\**Package**\**AnotherNamespace**\**ClassE** **as** **E**;

**use** **function** **Vendor**\**Package**\{**functionA**, **functionB**, **functionC**};

**use** **function** **Another**\**Vendor**\**functionD**;

**use** **const** **Vendor**\**Package**\{**CONSTANT\_A**, **CONSTANT\_B**, **CONSTANT\_C**};

**use** **const** **Another**\**Vendor**\**CONSTANT\_D**;

/\*\*

\* FooBar is an example class.

\*/

**class** **FooBar**

{

// ... additional PHP code ...

}

Compound namespaces with a depth of more than two MUST NOT be used. Therefore the following is the maximum compounding depth allowed:

<?php

**use** **Vendor**\**Package**\**SomeNamespace**\{

**SubnamespaceOne**\**ClassA**,

**SubnamespaceOne**\**ClassB**,

**SubnamespaceTwo**\**ClassY**,

**ClassZ**,

};

And the following would not be allowed:

<?php

**use** **Vendor**\**Package**\**SomeNamespace**\{

**SubnamespaceOne**\**AnotherNamespace**\**ClassA**,

**SubnamespaceOne**\**ClassB**,

**ClassZ**,

};

When wishing to declare strict types in files containing markup outside PHP opening and closing tags, the declaration MUST be on the first line of the file and include an opening PHP tag, the strict types declaration and closing tag.

For example:

<?php **declare**(strict\_types=1) ?>

<html>

<body>

<?php

// ... additional PHP code ...

?>

</body>

</html>

Declare statements MUST contain no spaces and MUST be exactly declare(strict\_types=1) (with an optional semicolon terminator).

Block declare statements are allowed and MUST be formatted as below. Note position of braces and spacing:

**declare**(ticks=1) {

// some code

}

4. Classes, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

Any closing brace MUST NOT be followed by any comment or statement on the same line.

When instantiating a new class, parentheses MUST always be present even when there are no arguments passed to the constructor.

**new** Foo();

**4.1 Extends and Implements**

The extends and implements keywords MUST be declared on the same line as the class name.

The opening brace for the class MUST go on its own line; the closing brace for the class MUST go on the next line after the body.

Opening braces MUST be on their own line and MUST NOT be preceded or followed by a blank line.

Closing braces MUST be on their own line and MUST NOT be preceded by a blank line.

<?php

**namespace** **Vendor**\**Package**;

**use** **FooClass**;

**use** **BarClass** **as** **Bar**;

**use** **OtherVendor**\**OtherPackage**\**BazClass**;

**class** **ClassName** **extends** **ParentClass** **implements** \**ArrayAccess**, \**Countable**

{

// constants, properties, methods

}

Lists of implements and, in the case of interfaces, extends MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one interface per line.

<?php

**namespace** **Vendor**\**Package**;

**use** **FooClass**;

**use** **BarClass** **as** **Bar**;

**use** **OtherVendor**\**OtherPackage**\**BazClass**;

**class** **ClassName** **extends** **ParentClass** **implements**

\**ArrayAccess**,

\**Countable**,

\**Serializable**

{

// constants, properties, methods

}

**4.2 Using traits**

The use keyword used inside the classes to implement traits MUST be declared on the next line after the opening brace.

<?php

**namespace** **Vendor**\**Package**;

**use** **Vendor**\**Package**\**FirstTrait**;

**class** **ClassName**

{

**use** **FirstTrait**;

}

Each individual trait that is imported into a class MUST be included one-per-line and each inclusion MUST have its own use import statement.

<?php

**namespace** **Vendor**\**Package**;

**use** **Vendor**\**Package**\**FirstTrait**;

**use** **Vendor**\**Package**\**SecondTrait**;

**use** **Vendor**\**Package**\**ThirdTrait**;

**class** **ClassName**

{

**use** **FirstTrait**;

**use** **SecondTrait**;

**use** **ThirdTrait**;

}

When the class has nothing after the use import statement, the class closing brace MUST be on the next line after the use import statement.

<?php

**namespace** **Vendor**\**Package**;

**use** **Vendor**\**Package**\**FirstTrait**;

**class** **ClassName**

{

**use** **FirstTrait**;

}

Otherwise, it MUST have a blank line after the use import statement.

<?php

**namespace** **Vendor**\**Package**;

**use** **Vendor**\**Package**\**FirstTrait**;

**class** **ClassName**

{

**use** **FirstTrait**;

**private** $property;

}

When using the insteadof and as operators they must be used as follows taking note of indentation, spacing, and new lines.

<?php

**class** **Talker**

{

**use** **A**;

**use** **B** {

**A**::**smallTalk** **insteadof** **B**;

}

**use** **C** {

**B**::**bigTalk** **insteadof** **C**;

C::mediumTalk **as** FooBar;

}

}

**4.3 Properties and Constants**

Visibility MUST be declared on all properties.

Visibility MUST be declared on all constants if your project PHP minimum version supports constant visibilities (PHP 7.1 or later).

The var keyword MUST NOT be used to declare a property.

There MUST NOT be more than one property declared per statement.

Property names MUST NOT be prefixed with a single underscore to indicate protected or private visibility. That is, an underscore prefix explicitly has no meaning.

There MUST be a space between type declaration and property name.

A property declaration looks like the following:

<?php

**namespace** **Vendor**\**Package**;

**class** **ClassName**

{

**public** $foo = **null**;

**public** **static** int $bar = 0;

}

**4.4 Methods and Functions**

Visibility MUST be declared on all methods.

Method names MUST NOT be prefixed with a single underscore to indicate protected or private visibility. That is, an underscore prefix explicitly has no meaning.

Method and function names MUST NOT be declared with space after the method name. The opening brace MUST go on its own line, and the closing brace MUST go on the next line following the body. There MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis.

A method declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

<?php

**namespace** **Vendor**\**Package**;

**class** **ClassName**

{

**public** **function** **fooBarBaz**($arg1, &$arg2, $arg3 = [])

{

// method body

}

}

A function declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

<?php

**function** **fooBarBaz**($arg1, &$arg2, $arg3 = [])

{

// function body

}

**4.5 Method and Function Arguments**

In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Method and function arguments with default values MUST go at the end of the argument list.

<?php

**namespace** **Vendor**\**Package**;

**class** **ClassName**

{

**public** **function** **foo**(int $arg1, &$arg2, $arg3 = [])

{

// method body

}

}

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

When the argument list is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

<?php

**namespace** **Vendor**\**Package**;

**class** **ClassName**

{

**public** **function** **aVeryLongMethodName**(

ClassTypeHint $arg1,

&$arg2,

array $arg3 = []

) {

// method body

}

}

When you have a return type declaration present, there MUST be one space after the colon followed by the type declaration. The colon and declaration MUST be on the same line as the argument list closing parenthesis with no spaces between the two characters.

<?php

**declare**(strict\_types=1);

**namespace** **Vendor**\**Package**;

**class** **ReturnTypeVariations**

{

**public** **function** **functionName**(int $arg1, $arg2): **string**

{

**return** 'foo';

}

**public** **function** **anotherFunction**(

string $foo,

string $bar,

int $baz

): **string** {

**return** 'foo';

}

}

In nullable type declarations, there MUST NOT be a space between the question mark and the type.

<?php

**declare**(strict\_types=1);

**namespace** **Vendor**\**Package**;

**class** **ReturnTypeVariations**

{

**public** **function** **functionName**(?string $arg1, ?int &$arg2): ?**string**

{

**return** 'foo';

}

}

When using the reference operator & before an argument, there MUST NOT be a space after it, like in the previous example.

There MUST NOT be a space between the variadic three dot operator and the argument name:

**public** **function** **process**(string $algorithm, ...$parts)

{

// processing

}

When combining both the reference operator and the variadic three dot operator, there MUST NOT be any space between the two of them:

**public** **function** **process**(string $algorithm, &...$parts)

{

// processing

}

**4.6 abstract, final, and static**

When present, the abstract and final declarations MUST precede the visibility declaration.

When present, the static declaration MUST come after the visibility declaration.

<?php

**namespace** **Vendor**\**Package**;

**abstract** **class** **ClassName**

{

**protected** **static** $foo;

**abstract** **protected** **function** **zim**();

**final** **public** **static** **function** **bar**()

{

// method body

}

}

**4.7 Method and Function Calls**

When making a method or function call, there MUST NOT be a space between the method or function name and the opening parenthesis, there MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis. In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

<?php

bar();

$foo->bar($arg1);

Foo::bar($arg2, $arg3);

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line. A single argument being split across multiple lines (as might be the case with an anonymous function or array) does not constitute splitting the argument list itself.

<?php

$foo->bar(

$longArgument,

$longerArgument,

$muchLongerArgument

);

<?php

somefunction($foo, $bar, [

// ...

], $baz);

$app->get('/hello/{name}', **function** ($name) **use** ($app) {

**return** 'Hello ' . $app->escape($name);

});

5. Control Structures

The general style rules for control structures are as follows:

* There MUST be one space after the control structure keyword
* There MUST NOT be a space after the opening parenthesis
* There MUST NOT be a space before the closing parenthesis
* There MUST be one space between the closing parenthesis and the opening brace
* The structure body MUST be indented once
* The body MUST be on the next line after the opening brace
* The closing brace MUST be on the next line after the body

The body of each structure MUST be enclosed by braces. This standardizes how the structures look and reduces the likelihood of introducing errors as new lines get added to the body.

**5.1 if, elseif, else**

An if structure looks like the following. Note the placement of parentheses, spaces, and braces; and that else and elseif are on the same line as the closing brace from the earlier body.

<?php

**if** ($expr1) {

// if body

} **elseif** ($expr2) {

// elseif body

} **else** {

// else body;

}

The keyword elseif SHOULD be used instead of else if so that all control keywords look like single words.

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

<?php

**if** (

$expr1

&& $expr2

) {

// if body

} **elseif** (

$expr3

&& $expr4

) {

// elseif body

}

**5.2 switch, case**

A switch structure looks like the following. Note the placement of parentheses, spaces, and braces. The case statement MUST be indented once from switch, and the break keyword (or other terminating keywords) MUST be indented at the same level as the case body. There MUST be a comment such as // no break when fall-through is intentional in a non-empty case body.

<?php

**switch** ($expr) {

**case** 0:

**echo** 'First case, with a break';

**break**;

**case** 1:

**echo** 'Second case, which falls through';

// no break

**case** 2:

**case** 3:

**case** 4:

**echo** 'Third case, return instead of break';

**return**;

**default**:

**echo** 'Default case';

**break**;

}

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

<?php

**switch** (

$expr1

&& $expr2

) {

// structure body

}

**5.3 while, do while**

A while statement looks like the following. Note the placement of parentheses, spaces, and braces.

<?php

**while** ($expr) {

// structure body

}

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

<?php

**while** (

$expr1

&& $expr2

) {

// structure body

}

Similarly, a do while statement looks like the following. Note the placement of parentheses, spaces, and braces.

<?php

**do** {

// structure body;

} **while** ($expr);

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

<?php

**do** {

// structure body;

} **while** (

$expr1

&& $expr2

);

**5.4 for**

A for statement looks like the following. Note the placement of parentheses, spaces, and braces.

<?php

**for** ($i = 0; $i < 10; $i++) {

// for body

}

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first expression MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

<?php

**for** (

$i = 0;

$i < 10;

$i++

) {

// for body

}

**5.5 foreach**

A foreach statement looks like the following. Note the placement of parentheses, spaces, and braces.

<?php

**foreach** ($iterable **as** $key => $value) {

// foreach body

}

**5.6 try, catch, finally**

A try-catch-finally block looks like the following. Note the placement of parentheses, spaces, and braces.

<?php

**try** {

// try body

} **catch** (FirstThrowableType $e) {

// catch body

} **catch** (OtherThrowableType | AnotherThrowableType $e) {

// catch body

} **finally** {

// finally body

}

6. Operators

Style rules for operators are grouped by arity (the number of operands they take).

When space is permitted around an operator, multiple spaces MAY be used for readability purposes.

All operators not described here are left undefined.

**6.1. Unary operators**

The increment/decrement operators MUST NOT have any space between the operator and operand.

$i++;

++$j;

Type casting operators MUST NOT have any space within the parentheses:

$intValue = (int) $input;

**6.2. Binary operators**

All binary [arithmetic](http://php.net/manual/en/language.operators.arithmetic.php), [comparison](http://php.net/manual/en/language.operators.comparison.php), [assignment](http://php.net/manual/en/language.operators.assignment.php), [bitwise](http://php.net/manual/en/language.operators.bitwise.php), [logical](http://php.net/manual/en/language.operators.logical.php), [string](http://php.net/manual/en/language.operators.string.php), and [type](http://php.net/manual/en/language.operators.type.php) operators MUST be preceded and followed by at least one space:

**if** ($a === $b) {

$foo = $bar ?? $a ?? $b;

} **elseif** ($a > $b) {

$foo = $a + $b \* $c;

}

**6.3. Ternary operators**

The conditional operator, also known simply as the ternary operator, MUST be preceded and followed by at least one space around both the ? and : characters:

$variable = $foo ? 'foo' : 'bar';

When the middle operand of the conditional operator is omitted, the operator MUST follow the same style rules as other binary [comparison](http://php.net/manual/en/language.operators.comparison.php) operators:

$variable = $foo ?: 'bar';

7. Closures

Closures MUST be declared with a space after the function keyword, and a space before and after the use keyword.

The opening brace MUST go on the same line, and the closing brace MUST go on the next line following the body.

There MUST NOT be a space after the opening parenthesis of the argument list or variable list, and there MUST NOT be a space before the closing parenthesis of the argument list or variable list.

In the argument list and variable list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Closure arguments with default values MUST go at the end of the argument list.

If a return type is present, it MUST follow the same rules as with normal functions and methods; if the use keyword is present, the colon MUST follow the use list closing parentheses with no spaces between the two characters.

A closure declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

<?php

$closureWithArgs = **function** ($arg1, $arg2) {

// body

};

$closureWithArgsAndVars = **function** ($arg1, $arg2) **use** ($var1, $var2) {

// body

};

$closureWithArgsVarsAndReturn = **function** ($arg1, $arg2) **use** ($var1, $var2): **bool** {

// body

};

Argument lists and variable lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument or variable per line.

When the ending list (whether of arguments or variables) is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

The following are examples of closures with and without argument lists and variable lists split across multiple lines.

<?php

$longArgs\_noVars = **function** (

$longArgument,

$longerArgument,

$muchLongerArgument

) {

// body

};

$noArgs\_longVars = **function** () **use** (

$longVar1,

$longerVar2,

$muchLongerVar3

) {

// body

};

$longArgs\_longVars = **function** (

$longArgument,

$longerArgument,

$muchLongerArgument

) **use** (

$longVar1,

$longerVar2,

$muchLongerVar3

) {

// body

};

$longArgs\_shortVars = **function** (

$longArgument,

$longerArgument,

$muchLongerArgument

) **use** ($var1) {

// body

};

$shortArgs\_longVars = **function** ($arg) **use** (

$longVar1,

$longerVar2,

$muchLongerVar3

) {

// body

};

Note that the formatting rules also apply when the closure is used directly in a function or method call as an argument.

<?php

$foo->bar(

$arg1,

**function** ($arg2) **use** ($var1) {

// body

},

$arg3

);

8. Anonymous Classes

Anonymous Classes MUST follow the same guidelines and principles as closures in the above section.

<?php

$instance = **new** **class** {};

The opening brace MAY be on the same line as the class keyword so long as the list of implements interfaces does not wrap. If the list of interfaces wraps, the brace MUST be placed on the line immediately following the last interface.

<?php

// Brace on the same line

$instance = **new** **class** **extends** \**Foo** **implements** \**HandleableInterface** {

// Class content

};

// Brace on the next line

$instance = **new** **class** **extends** \**Foo** **implements**

\**ArrayAccess**,

\**Countable**,

\**Serializable**

{

// Class content

};

Go Coding Standards

**Formatting**[**¶**](https://go.dev/doc/effective_go#formatting)

Formatting issues are the most contentious but the least consequential. People can adapt to different formatting styles but it's better if they don't have to, and less time is devoted to the topic if everyone adheres to the same style. The problem is how to approach this Utopia without a long prescriptive style guide.

With Go we take an unusual approach and let the machine take care of most formatting issues. The gofmt program (also available as go fmt, which operates at the package level rather than source file level) reads a Go program and emits the source in a standard style of indentation and vertical alignment, retaining and if necessary reformatting comments. If you want to know how to handle some new layout situation, run gofmt; if the answer doesn't seem right, rearrange your program (or file a bug about gofmt), don't work around it.

As an example, there's no need to spend time lining up the comments on the fields of a structure. Gofmt will do that for you. Given the declaration

type T struct {

name string // name of the object

value int // its value

}

gofmt will line up the columns:

type T struct {

name string // name of the object

value int // its value

}

All Go code in the standard packages has been formatted with gofmt.

Some formatting details remain. Very briefly:

Indentation

We use tabs for indentation and gofmt emits them by default. Use spaces only if you must.

Line length

Go has no line length limit. Don't worry about overflowing a punched card. If a line feels too long, wrap it and indent with an extra tab.

Parentheses

Go needs fewer parentheses than C and Java: control structures (if, for, switch) do not have parentheses in their syntax. Also, the operator precedence hierarchy is shorter and clearer, so

x<<8 + y<<16

means what the spacing implies, unlike in the other languages.

**Commentary**[**¶**](https://go.dev/doc/effective_go#commentary)

Go provides C-style /\* \*/ block comments and C++-style // line comments. Line comments are the norm; block comments appear mostly as package comments, but are useful within an expression or to disable large swaths of code.

Comments that appear before top-level declarations, with no intervening newlines, are considered to document the declaration itself. These “doc comments” are the primary documentation for a given Go package or command. For more about doc comments, see “[Go Doc Comments](https://go.dev/doc/comment)”.

**Names**[**¶**](https://go.dev/doc/effective_go#names)

Names are as important in Go as in any other language. They even have semantic effect: the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

**Package names**[**¶**](https://go.dev/doc/effective_go#package-names)

When a package is imported, the package name becomes an accessor for the contents. After

import "bytes"

the importing package can talk about bytes.Buffer. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be typing that name. And don't worry about collisions *a priori*. The package name is only the default name for imports; it need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

Another convention is that the package name is the base name of its source directory; the package in src/encoding/base64 is imported as "encoding/base64" but has name base64, not encoding\_base64 and not encodingBase64.

The importer of a package will use the name to refer to its contents, so exported names in the package can use that fact to avoid repetition. (Don't use the import . notation, which can simplify tests that must run outside the package they are testing, but should otherwise be avoided.) For instance, the buffered reader type in the bufio package is called Reader, not BufReader, because users see it as bufio.Reader, which is a clear, concise name. Moreover, because imported entities are always addressed with their package name, bufio.Reader does not conflict with io.Reader. Similarly, the function to make new instances of ring.Ring—which is the definition of a *constructor* in Go—would normally be called NewRing, but since Ring is the only type exported by the package, and since the package is called ring, it's called just New, which clients of the package see as ring.New. Use the package structure to help you choose good names.

Another short example is once.Do; once.Do(setup) reads well and would not be improved by writing once.DoOrWaitUntilDone(setup). Long names don't automatically make things more readable. A helpful doc comment can often be more valuable than an extra long name.

**Getters**[**¶**](https://go.dev/doc/effective_go#Getters)

Go doesn't provide automatic support for getters and setters. There's nothing wrong with providing getters and setters yourself, and it's often appropriate to do so, but it's neither idiomatic nor necessary to put Get into the getter's name. If you have a field called owner (lower case, unexported), the getter method should be called Owner (upper case, exported), not GetOwner. The use of upper-case names for export provides the hook to discriminate the field from the method. A setter function, if needed, will likely be called SetOwner. Both names read well in practice:

owner := obj.Owner()

if owner != user {

obj.SetOwner(user)

}

**Interface names**[**¶**](https://go.dev/doc/effective_go#interface-names)

By convention, one-method interfaces are named by the method name plus an -er suffix or similar modification to construct an agent noun: Reader, Writer, Formatter, CloseNotifier etc.

There are a number of such names and it's productive to honor them and the function names they capture. Read, Write, Close, Flush, String and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method String not ToString.

**MixedCaps[¶](https://go.dev/doc/effective_go" \l "mixed-caps)**

Finally, the convention in Go is to use MixedCaps or mixedCaps rather than underscores to write multiword names.

**Semicolons**[**¶**](https://go.dev/doc/effective_go#semicolons)

Like C, Go's formal grammar uses semicolons to terminate statements, but unlike in C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them.

The rule is this. If the last token before a newline is an identifier (which includes words like int and float64), a basic literal such as a number or string constant, or one of the tokens

break continue fallthrough return ++ -- ) }

the lexer always inserts a semicolon after the token. This could be summarized as, “if the newline comes after a token that could end a statement, insert a semicolon”.

A semicolon can also be omitted immediately before a closing brace, so a statement such as

go func() { for { dst <- <-src } }()

needs no semicolons. Idiomatic Go programs have semicolons only in places such as for loop clauses, to separate the initializer, condition, and continuation elements. They are also necessary to separate multiple statements on a line, should you write code that way.

One consequence of the semicolon insertion rules is that you cannot put the opening brace of a control structure (if, for, switch, or select) on the next line. If you do, a semicolon will be inserted before the brace, which could cause unwanted effects. Write them like this

if i < f() {

g()

}

not like this

if i < f() // wrong!

{ // wrong!

g()

}

**Control structures**[**¶**](https://go.dev/doc/effective_go#control-structures)

The control structures of Go are related to those of C but differ in important ways. There is no do or while loop, only a slightly generalized for; switch is more flexible; if and switch accept an optional initialization statement like that of for; break and continue statements take an optional label to identify what to break or continue; and there are new control structures including a type switch and a multiway communications multiplexer, select. The syntax is also slightly different: there are no parentheses and the bodies must always be brace-delimited.

**If**[**¶**](https://go.dev/doc/effective_go#if)

In Go a simple if looks like this:

if x > 0 {

return y

}

Mandatory braces encourage writing simple if statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a return or break.

Since if and switch accept an initialization statement, it's common to see one used to set up a local variable.

if err := file.Chmod(0664); err != nil {

log.Print(err)

return err

}

In the Go libraries, you'll find that when an if statement doesn't flow into the next statement—that is, the body ends in break, continue, goto, or return—the unnecessary else is omitted.

f, err := os.Open(name)

if err != nil {

return err

}

codeUsing(f)

This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in return statements, the resulting code needs no else statements.

f, err := os.Open(name)

if err != nil {

return err

}

d, err := f.Stat()

if err != nil {

f.Close()

return err

}

codeUsing(f, d)

**Redeclaration and reassignment**[**¶**](https://go.dev/doc/effective_go#redeclaration)

An aside: The last example in the previous section demonstrates a detail of how the := short declaration form works. The declaration that calls os.Open reads,

f, err := os.Open(name)

This statement declares two variables, f and err. A few lines later, the call to f.Stat reads,

d, err := f.Stat()

which looks as if it declares d and err. Notice, though, that err appears in both statements. This duplication is legal: err is declared by the first statement, but only *re-assigned* in the second. This means that the call to f.Stat uses the existing err variable declared above, and just gives it a new value.

In a := declaration a variable v may appear even if it has already been declared, provided:

* this declaration is in the same scope as the existing declaration of v (if v is already declared in an outer scope, the declaration will create a new variable §),
* the corresponding value in the initialization is assignable to v, and
* there is at least one other variable that is created by the declaration.

This unusual property is pure pragmatism, making it easy to use a single err value, for example, in a long if-else chain. You'll see it used often.

§ It's worth noting here that in Go the scope of function parameters and return values is the same as the function body, even though they appear lexically outside the braces that enclose the body.

**For**[**¶**](https://go.dev/doc/effective_go#for)

The Go for loop is similar to—but not the same as—C's. It unifies for and while and there is no do-while. There are three forms, only one of which has semicolons.

// Like a C for

for init; condition; post { }

// Like a C while

for condition { }

// Like a C for(;;)

for { }

Short declarations make it easy to declare the index variable right in the loop.

sum := 0

for i := 0; i < 10; i++ {

sum += i

}

If you're looping over an array, slice, string, or map, or reading from a channel, a range clause can manage the loop.

for key, value := range oldMap {

newMap[key] = value

}

If you only need the first item in the range (the key or index), drop the second:

for key := range m {

if key.expired() {

delete(m, key)

}

}

If you only need the second item in the range (the value), use the *blank identifier*, an underscore, to discard the first:

sum := 0

for \_, value := range array {

sum += value

}

The blank identifier has many uses, as described in [a later section](https://go.dev/doc/effective_go#blank).

For strings, the range does more work for you, breaking out individual Unicode code points by parsing the UTF-8. Erroneous encodings consume one byte and produce the replacement rune U+FFFD. (The name (with associated builtin type) rune is Go terminology for a single Unicode code point. See [the language specification](https://go.dev/ref/spec#Rune_literals) for details.) The loop

for pos, char := range "日本\x80語" { // \x80 is an illegal UTF-8 encoding

fmt.Printf("character %#U starts at byte position %d\n", char, pos)

}

Prints

character U+65E5 '日' starts at byte position 0

character U+672C '本' starts at byte position 3

character U+FFFD '�' starts at byte position 6

character U+8A9E '語' starts at byte position 7

Finally, Go has no comma operator and ++ and -- are statements not expressions. Thus if you want to run multiple variables in a for you should use parallel assignment (although that precludes ++ and --).

// Reverse a

for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {

a[i], a[j] = a[j], a[i]

}

**Switch**[**¶**](https://go.dev/doc/effective_go#switch)

Go's switch is more general than C's. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the switch has no expression it switches on true. It's therefore possible—and idiomatic—to write an if-else-if-else chain as a switch.

func unhex(c byte) byte {

switch {

case '0' <= c && c <= '9':

return c - '0'

case 'a' <= c && c <= 'f':

return c - 'a' + 10

case 'A' <= c && c <= 'F':

return c - 'A' + 10

}

return 0

}

There is no automatic fall through, but cases can be presented in comma-separated lists.

func shouldEscape(c byte) bool {

switch c {

case ' ', '?', '&', '=', '#', '+', '%':

return true

}

return false

}

Although they are not nearly as common in Go as some other C-like languages, break statements can be used to terminate a switch early. Sometimes, though, it's necessary to break out of a surrounding loop, not the switch, and in Go that can be accomplished by putting a label on the loop and "breaking" to that label. This example shows both uses.

Loop:

for n := 0; n < len(src); n += size {

switch {

case src[n] < sizeOne:

if validateOnly {

break

}

size = 1

update(src[n])

case src[n] < sizeTwo:

if n+1 >= len(src) {

err = errShortInput

break Loop

}

if validateOnly {

break

}

size = 2

update(src[n] + src[n+1]<<shift)

}

}

Of course, the continue statement also accepts an optional label but it applies only to loops.

To close this section, here's a comparison routine for byte slices that uses two switch statements:

// Compare returns an integer comparing the two byte slices,

// lexicographically.

// The result will be 0 if a == b, -1 if a < b, and +1 if a > b

func Compare(a, b []byte) int {

for i := 0; i < len(a) && i < len(b); i++ {

switch {

case a[i] > b[i]:

return 1

case a[i] < b[i]:

return -1

}

}

switch {

case len(a) > len(b):

return 1

case len(a) < len(b):

return -1

}

return 0

}

**Type switch**[**¶**](https://go.dev/doc/effective_go#type_switch)

A switch can also be used to discover the dynamic type of an interface variable. Such a *type switch* uses the syntax of a type assertion with the keyword type inside the parentheses. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause. It's also idiomatic to reuse the name in such cases, in effect declaring a new variable with the same name but a different type in each case.

var t interface{}

t = functionOfSomeType()

switch t := t.(type) {

default:

fmt.Printf("unexpected type %T\n", t) // %T prints whatever type t has

case bool:

fmt.Printf("boolean %t\n", t) // t has type bool

case int:

fmt.Printf("integer %d\n", t) // t has type int

case \*bool:

fmt.Printf("pointer to boolean %t\n", \*t) // t has type \*bool

case \*int:

fmt.Printf("pointer to integer %d\n", \*t) // t has type \*int

}

**Functions**[**¶**](https://go.dev/doc/effective_go#functions)

**Multiple return values**[**¶**](https://go.dev/doc/effective_go#multiple-returns)

One of Go's unusual features is that functions and methods can return multiple values. This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns such as -1 for EOF and modifying an argument passed by address.

In C, a write error is signaled by a negative count with the error code secreted away in a volatile location. In Go, Write can return a count *and* an error: “Yes, you wrote some bytes but not all of them because you filled the device”. The signature of the Write method on files from package os is:

func (file \*File) Write(b []byte) (n int, err error)

and as the documentation says, it returns the number of bytes written and a non-nil error when n != len(b). This is a common style; see the section on error handling for more examples.

A similar approach obviates the need to pass a pointer to a return value to simulate a reference parameter. Here's a simple-minded function to grab a number from a position in a byte slice, returning the number and the next position.

func nextInt(b []byte, i int) (int, int) {

for ; i < len(b) && !isDigit(b[i]); i++ {

}

x := 0

for ; i < len(b) && isDigit(b[i]); i++ {

x = x\*10 + int(b[i]) - '0'

}

return x, i

}

You could use it to scan the numbers in an input slice b like this:

for i := 0; i < len(b); {

x, i = nextInt(b, i)

fmt.Println(x)

}

**Named result parameters**[**¶**](https://go.dev/doc/effective_go#named-results)

The return or result "parameters" of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values for their types when the function begins; if the function executes a return statement with no arguments, the current values of the result parameters are used as the returned values.

The names are not mandatory but they can make code shorter and clearer: they're documentation. If we name the results of nextInt it becomes obvious which returned int is which.

func nextInt(b []byte, pos int) (value, nextPos int) {

Because named results are initialized and tied to an unadorned return, they can simplify as well as clarify. Here's a version of io.ReadFull that uses them well:

func ReadFull(r Reader, buf []byte) (n int, err error) {

for len(buf) > 0 && err == nil {

var nr int

nr, err = r.Read(buf)

n += nr

buf = buf[nr:]

}

return

}

**Defer**[**¶**](https://go.dev/doc/effective_go#defer)

Go's defer statement schedules a function call (the *deferred* function) to be run immediately before the function executing the defer returns. It's an unusual but effective way to deal with situations such as resources that must be released regardless of which path a function takes to return. The canonical examples are unlocking a mutex or closing a file.

// Contents returns the file's contents as a string.

func Contents(filename string) (string, error) {

f, err := os.Open(filename)

if err != nil {

return "", err

}

defer f.Close() // f.Close will run when we're finished.

var result []byte

buf := make([]byte, 100)

for {

n, err := f.Read(buf[0:])

result = append(result, buf[0:n]...) // append is discussed later.

if err != nil {

if err == io.EOF {

break

}

return "", err // f will be closed if we return here.]

}

}

return string(result), nil // f will be closed if we return here.

}

Deferring a call to a function such as Close has two advantages. First, it guarantees that you will never forget to close the file, a mistake that's easy to make if you later edit the function to add a new return path. Second, it means that the close sits near the open, which is much clearer than placing it at the end of the function.

The arguments to the deferred function (which include the receiver if the function is a method) are evaluated when the *defer* executes, not when the *call* executes. Besides avoiding worries about variables changing values as the function executes, this means that a single deferred call site can defer multiple function executions. Here's a silly example.

for i := 0; i < 5; i++ {

defer fmt.Printf("%d ", i)

}

Deferred functions are executed in LIFO order, so this code will cause 4 3 2 1 0 to be printed when the function returns. A more plausible example is a simple way to trace function execution through the program. We could write a couple of simple tracing routines like this:

func trace(s string) { fmt.Println("entering:", s) }

func untrace(s string) { fmt.Println("leaving:", s) }

// Use them like this:

func a() {

trace("a")

defer untrace("a")

// do something....

}

We can do better by exploiting the fact that arguments to deferred functions are evaluated when the defer executes. The tracing routine can set up the argument to the untracing routine. This example:

func trace(s string) string {

fmt.Println("entering:", s)

return s

}

func un(s string) {

fmt.Println("leaving:", s)

}

func a() {

defer un(trace("a"))

fmt.Println("in a")

}

func b() {

defer un(trace("b"))

fmt.Println("in b")

a()

}

func main() {

b()

}

Prints

entering: b

in b

entering: a

in a

leaving: a

leaving: b

For programmers accustomed to block-level resource management from other languages, defer may seem peculiar, but its most interesting and powerful applications come precisely from the fact that it's not block-based but function-based. In the section on panic and recover we'll see another example of its possibilities.

**Data**[**¶**](https://go.dev/doc/effective_go#data)

**Allocation with new**[**¶**](https://go.dev/doc/effective_go#allocation_new)

Go has two allocation primitives, the built-in functions new and make. They do different things and apply to different types, which can be confusing, but the rules are simple. Let's talk about new first. It's a built-in function that allocates memory, but unlike its namesakes in some other languages it does not *initialize* the memory, it only *zeros* it. That is, new(T) allocates zeroed storage for a new item of type T and returns its address, a value of type \*T. In Go terminology, it returns a pointer to a newly allocated zero value of type T.

Since the memory returned by new is zeroed, it's helpful to arrange when designing your data structures that the zero value of each type can be used without further initialization. This means a user of the data structure can create one with new and get right to work. For example, the documentation for bytes.Buffer states that "the zero value for Buffer is an empty buffer ready to use." Similarly, sync.Mutex does not have an explicit constructor or Init method. Instead, the zero value for a sync.Mutex is defined to be an unlocked mutex.

The zero-value-is-useful property works transitively. Consider this type declaration.

type SyncedBuffer struct {

lock sync.Mutex

buffer bytes.Buffer

}

Values of type SyncedBuffer are also ready to use immediately upon allocation or just declaration. In the next snippet, both p and v will work correctly without further arrangement.

p := new(SyncedBuffer) // type \*SyncedBuffer

var v SyncedBuffer // type SyncedBuffer

**Constructors and composite literals**[**¶**](https://go.dev/doc/effective_go#composite_literals)

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example derived from package os.

func NewFile(fd int, name string) \*File {

if fd < 0 {

return nil

}

f := new(File)

f.fd = fd

f.name = name

f.dirinfo = nil

f.nepipe = 0

return f

}

There's a lot of boilerplate in there. We can simplify it using a *composite literal*, which is an expression that creates a new instance each time it is evaluated.

func NewFile(fd int, name string) \*File {

if fd < 0 {

return nil

}

f := File{fd, name, nil, 0}

return &f

}

Note that, unlike in C, it's perfectly OK to return the address of a local variable; the storage associated with the variable survives after the function returns. In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.

return &File{fd, name, nil, 0}

The fields of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as *field*:*value* pairs, the initializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

return &File{fd: fd, name: name}

As a limiting case, if a composite literal contains no fields at all, it creates a zero value for the type. The expressions new(File) and &File{} are equivalent.

Composite literals can also be created for arrays, slices, and maps, with the field labels being indices or map keys as appropriate. In these examples, the initializations work regardless of the values of Enone, Eio, and Einval, as long as they are distinct.

a := [...]string {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}

s := []string {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}

m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}

**Allocation with make**[**¶**](https://go.dev/doc/effective_go#allocation_make)

Back to allocation. The built-in function make(T, *args*) serves a purpose different from new(T). It creates slices, maps, and channels only, and it returns an *initialized* (not *zeroed*) value of type T (not \*T). The reason for the distinction is that these three types represent, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity, and until those items are initialized, the slice is nil. For slices, maps, and channels, make initializes the internal data structure and prepares the value for use. For instance,

make([]int, 10, 100)

allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. (When making a slice, the capacity can be omitted; see the section on slices for more information.) In contrast, new([]int) returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a nil slice value.

These examples illustrate the difference between new and make.

var p \*[]int = new([]int) // allocates slice structure; \*p == nil; rarely useful

var v []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:

var p \*[]int = new([]int)

\*p = make([]int, 100, 100)

// Idiomatic:

v := make([]int, 100)

Remember that make applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with new or take the address of a variable explicitly.

**Arrays**[**¶**](https://go.dev/doc/effective_go#arrays)

Arrays are useful when planning the detailed layout of memory and sometimes can help avoid allocation, but primarily they are a building block for slices, the subject of the next section. To lay the foundation for that topic, here are a few words about arrays.

There are major differences between the ways arrays work in Go and C. In Go,

* Arrays are values. Assigning one array to another copies all the elements.
* In particular, if you pass an array to a function, it will receive a *copy* of the array, not a pointer to it.
* The size of an array is part of its type. The types [10]int and [20]int are distinct.

The value property can be useful but also expensive; if you want C-like behavior and efficiency, you can pass a pointer to the array.

func Sum(a \*[3]float64) (sum float64) {

for \_, v := range \*a {

sum += v

}

return

}

array := [...]float64{7.0, 8.5, 9.1}

x := Sum(&array) // Note the explicit address-of operator

But even this style isn't idiomatic Go. Use slices instead.

**Slices**[**¶**](https://go.dev/doc/effective_go#slices)

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with explicit dimension such as transformation matrices, most array programming in Go is done with slices rather than simple arrays.

Slices hold references to an underlying array, and if you assign one slice to another, both refer to the same array. If a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. A Read function can therefore accept a slice argument rather than a pointer and a count; the length within the slice sets an upper limit of how much data to read. Here is the signature of the Read method of the File type in package os:

func (f \*File) Read(buf []byte) (n int, err error)

The method returns the number of bytes read and an error value, if any. To read into the first 32 bytes of a larger buffer buf, *slice* (here used as a verb) the buffer.

n, err := f.Read(buf[0:32])

Such slicing is common and efficient. In fact, leaving efficiency aside for the moment, the following snippet would also read the first 32 bytes of the buffer.

var n int

var err error

for i := 0; i < 32; i++ {

nbytes, e := f.Read(buf[i:i+1]) // Read one byte.

n += nbytes

if nbytes == 0 || e != nil {

err = e

break

}

}

The length of a slice may be changed as long as it still fits within the limits of the underlying array; just assign it to a slice of itself. The *capacity* of a slice, accessible by the built-in function cap, reports the maximum length the slice may assume. Here is a function to append data to a slice. If the data exceeds the capacity, the slice is reallocated. The resulting slice is returned. The function uses the fact that len and cap are legal when applied to the nil slice, and return 0.

func Append(slice, data []byte) []byte {

l := len(slice)

if l + len(data) > cap(slice) { // reallocate

// Allocate double what's needed, for future growth.

newSlice := make([]byte, (l+len(data))\*2)

// The copy function is predeclared and works for any slice type.

copy(newSlice, slice)

slice = newSlice

}

slice = slice[0:l+len(data)]

copy(slice[l:], data)

return slice

}

We must return the slice afterwards because, although Append can modify the elements of slice, the slice itself (the run-time data structure holding the pointer, length, and capacity) is passed by value.

The idea of appending to a slice is so useful it's captured by the append built-in function. To understand that function's design, though, we need a little more information, so we'll return to it later.

**Two-dimensional slices**[**¶**](https://go.dev/doc/effective_go#two_dimensional_slices)

Go's arrays and slices are one-dimensional. To create the equivalent of a 2D array or slice, it is necessary to define an array-of-arrays or slice-of-slices, like this:

type Transform [3][3]float64 // A 3x3 array, really an array of arrays.

type LinesOfText [][]byte // A slice of byte slices.

Because slices are variable-length, it is possible to have each inner slice be a different length. That can be a common situation, as in our LinesOfText example: each line has an independent length.

text := LinesOfText{

[]byte("Now is the time"),

[]byte("for all good gophers"),

[]byte("to bring some fun to the party."),

}

Sometimes it's necessary to allocate a 2D slice, a situation that can arise when processing scan lines of pixels, for instance. There are two ways to achieve this. One is to allocate each slice independently; the other is to allocate a single array and point the individual slices into it. Which to use depends on your application. If the slices might grow or shrink, they should be allocated independently to avoid overwriting the next line; if not, it can be more efficient to construct the object with a single allocation. For reference, here are sketches of the two methods. First, a line at a time:

// Allocate the top-level slice.

picture := make([][]uint8, YSize) // One row per unit of y.

// Loop over the rows, allocating the slice for each row.

for i := range picture {

picture[i] = make([]uint8, XSize)

}

And now as one allocation, sliced into lines:

// Allocate the top-level slice, the same as before.

picture := make([][]uint8, YSize) // One row per unit of y.

// Allocate one large slice to hold all the pixels.

pixels := make([]uint8, XSize\*YSize) // Has type []uint8 even though picture is [][]uint8.

// Loop over the rows, slicing each row from the front of the remaining pixels slice.

for i := range picture {

picture[i], pixels = pixels[:XSize], pixels[XSize:]

}

**Maps**[**¶**](https://go.dev/doc/effective_go#maps)

Maps are a convenient and powerful built-in data structure that associate values of one type (the *key*) with values of another type (the *element* or *value*). The key can be of any type for which the equality operator is defined, such as integers, floating point and complex numbers, strings, pointers, interfaces (as long as the dynamic type supports equality), structs and arrays. Slices cannot be used as map keys, because equality is not defined on them. Like slices, maps hold references to an underlying data structure. If you pass a map to a function that changes the contents of the map, the changes will be visible in the caller.

Maps can be constructed using the usual composite literal syntax with colon-separated key-value pairs, so it's easy to build them during initialization.

var timeZone = map[string]int{

"UTC": 0\*60\*60,

"EST": -5\*60\*60,

"CST": -6\*60\*60,

"MST": -7\*60\*60,

"PST": -8\*60\*60,

}

Assigning and fetching map values looks syntactically just like doing the same for arrays and slices except that the index doesn't need to be an integer.

offset := timeZone["EST"]

An attempt to fetch a map value with a key that is not present in the map will return the zero value for the type of the entries in the map. For instance, if the map contains integers, looking up a non-existent key will return 0. A set can be implemented as a map with value type bool. Set the map entry to true to put the value in the set, and then test it by simple indexing.

attended := map[string]bool{

"Ann": true,

"Joe": true,

...

}

if attended[person] { // will be false if person is not in the map

fmt.Println(person, "was at the meeting")

}

Sometimes you need to distinguish a missing entry from a zero value. Is there an entry for "UTC" or is that 0 because it's not in the map at all? You can discriminate with a form of multiple assignment.

var seconds int

var ok bool

seconds, ok = timeZone[tz]

For obvious reasons this is called the “comma ok” idiom. In this example, if tz is present, seconds will be set appropriately and ok will be true; if not, seconds will be set to zero and ok will be false. Here's a function that puts it together with a nice error report:

func offset(tz string) int {

if seconds, ok := timeZone[tz]; ok {

return seconds

}

log.Println("unknown time zone:", tz)

return 0

}

To test for presence in the map without worrying about the actual value, you can use the [blank identifier](https://go.dev/doc/effective_go#blank) (\_) in place of the usual variable for the value.

\_, present := timeZone[tz]

To delete a map entry, use the delete built-in function, whose arguments are the map and the key to be deleted. It's safe to do this even if the key is already absent from the map.

delete(timeZone, "PDT") // Now on Standard Time

**Printing**[**¶**](https://go.dev/doc/effective_go#printing)

Formatted printing in Go uses a style similar to C's printf family but is richer and more general. The functions live in the fmt package and have capitalized names: fmt.Printf, fmt.Fprintf, fmt.Sprintf and so on. The string functions (Sprintf etc.) return a string rather than filling in a provided buffer.

You don't need to provide a format string. For each of Printf, Fprintf and Sprintf there is another pair of functions, for instance Print and Println. These functions do not take a format string but instead generate a default format for each argument. The Println versions also insert a blank between arguments and append a newline to the output while the Print versions add blanks only if the operand on neither side is a string. In this example each line produces the same output.

fmt.Printf("Hello %d\n", 23)

fmt.Fprint(os.Stdout, "Hello ", 23, "\n")

fmt.Println("Hello", 23)

fmt.Println(fmt.Sprint("Hello ", 23))

The formatted print functions fmt.Fprint and friends take as a first argument any object that implements the io.Writer interface; the variables os.Stdout and os.Stderr are familiar instances.

Here things start to diverge from C. First, the numeric formats such as %d do not take flags for signedness or size; instead, the printing routines use the type of the argument to decide these properties.

var x uint64 = 1<<64 – 1

fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))

prints

18446744073709551615 ffffffffffffffff; -1 -1

If you just want the default conversion, such as decimal for integers, you can use the catchall format %v (for “value”); the result is exactly what Print and Println would produce. Moreover, that format can print *any* value, even arrays, slices, structs, and maps. Here is a print statement for the time zone map defined in the previous section.

fmt.Printf("%v\n", timeZone) // or just fmt.Println(timeZone)

which gives output:

map[CST:-21600 EST:-18000 MST:-25200 PST:-28800 UTC:0]

For maps, Printf and friends sort the output lexicographically by key.

When printing a struct, the modified format %+v annotates the fields of the structure with their names, and for any value the alternate format %#v prints the value in full Go syntax.

type T struct {

a int

b float64

c string

}

t := &T{ 7, -2.35, "abc\tdef" }

fmt.Printf("%v\n", t)

fmt.Printf("%+v\n", t)

fmt.Printf("%#v\n", t)

fmt.Printf("%#v\n", timeZone)

prints

&{7 -2.35 abc def}

&{a:7 b:-2.35 c:abc def}

&main.T{a:7, b:-2.35, c:"abc\tdef"}

map[string]int{"CST":-21600, "EST":-18000, "MST":-25200, "PST":-28800, "UTC":0}

(Note the ampersands.) That quoted string format is also available through %q when applied to a value of type string or []byte. The alternate format %#q will use backquotes instead if possible. (The %q format also applies to integers and runes, producing a single-quoted rune constant.) Also, %x works on strings, byte arrays and byte slices as well as on integers, generating a long hexadecimal string, and with a space in the format (% x) it puts spaces between the bytes.

Another handy format is %T, which prints the *type* of a value.

fmt.Printf("%T\n", timeZone)

prints

map[string]int

If you want to control the default format for a custom type, all that's required is to define a method with the signature String() string on the type. For our simple type T, that might look like this.

func (t \*T) String() string {

return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)

}

fmt.Printf("%v\n", t)

to print in the format

7/-2.35/"abc\tdef"

(If you need to print *values* of type T as well as pointers to T, the receiver for String must be of value type; this example used a pointer because that's more efficient and idiomatic for struct types. See the section below on [pointers vs. value receivers](https://go.dev/doc/effective_go#pointers_vs_values) for more information.)

Our String method is able to call Sprintf because the print routines are fully reentrant and can be wrapped this way. There is one important detail to understand about this approach, however: don't construct a String method by calling Sprintf in a way that will recur into your String method indefinitely. This can happen if the Sprintf call attempts to print the receiver directly as a string, which in turn will invoke the method again. It's a common and easy mistake to make, as this example shows.

type MyString string

func (m MyString) String() string {

return fmt.Sprintf("MyString=%s", m) // Error: will recur forever.

}

It's also easy to fix: convert the argument to the basic string type, which does not have the method.

type MyString string

func (m MyString) String() string {

return fmt.Sprintf("MyString=%s", string(m)) // OK: note conversion.

}

In the [initialization section](https://go.dev/doc/effective_go#initialization) we'll see another technique that avoids this recursion.

Another printing technique is to pass a print routine's arguments directly to another such routine. The signature of Printf uses the type ...interface{} for its final argument to specify that an arbitrary number of parameters (of arbitrary type) can appear after the format.

func Printf(format string, v ...interface{}) (n int, err error) {

Within the function Printf, v acts like a variable of type []interface{} but if it is passed to another variadic function, it acts like a regular list of arguments. Here is the implementation of the function log.Println we used above. It passes its arguments directly to fmt.Sprintln for the actual formatting.

// Println prints to the standard logger in the manner of fmt.Println.

func Println(v ...interface{}) {

std.Output(2, fmt.Sprintln(v...)) // Output takes parameters (int, string)

}

We write ... after v in the nested call to Sprintln to tell the compiler to treat v as a list of arguments; otherwise it would just pass v as a single slice argument.

There's even more to printing than we've covered here. See the godoc documentation for package fmt for the details.

By the way, a ... parameter can be of a specific type, for instance ...int for a min function that chooses the least of a list of integers:

func Min(a ...int) int {

min := int(^uint(0) >> 1) // largest int

for \_, i := range a {

if i < min {

min = i

}

}

return min

}

**Append**[**¶**](https://go.dev/doc/effective_go#append)

Now we have the missing piece we needed to explain the design of the append built-in function. The signature of append is different from our custom Append function above. Schematically, it's like this:

func append(slice []*T*, elements ...*T*) []*T*

where *T* is a placeholder for any given type. You can't actually write a function in Go where the type T is determined by the caller. That's why append is built in: it needs support from the compiler.

What append does is append the elements to the end of the slice and return the result. The result needs to be returned because, as with our hand-written Append, the underlying array may change. This simple example

x := []int{1,2,3}

x = append(x, 4, 5, 6)

fmt.Println(x)

prints [1 2 3 4 5 6]. So append works a little like Printf, collecting an arbitrary number of arguments.

But what if we wanted to do what our Append does and append a slice to a slice? Easy: use ... at the call site, just as we did in the call to Output above. This snippet produces identical output to the one above.

x := []int{1,2,3}

y := []int{4,5,6}

x = append(x, y...)

fmt.Println(x)

Without that ..., it wouldn't compile because the types would be wrong; y is not of type int.

**Initialization**[**¶**](https://go.dev/doc/effective_go#initialization)

Although it doesn't look superficially very different from initialization in C or C++, initialization in Go is more powerful. Complex structures can be built during initialization and the ordering issues among initialized objects, even among different packages, are handled correctly.

**Constants**[**¶**](https://go.dev/doc/effective_go#constants)

Constants in Go are just that—constant. They are created at compile time, even when defined as locals in functions, and can only be numbers, characters (runes), strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evaluatable by the compiler. For instance, 1<<3 is a constant expression, while math.Sin(math.Pi/4) is not because the function call to math.Sin needs to happen at run time.

In Go, enumerated constants are created using the iota enumerator. Since iota can be part of an expression and expressions can be implicitly repeated, it is easy to build intricate sets of values.

type ByteSize float64

const (

\_ = iota // ignore first value by assigning to blank identifier

KB ByteSize = 1 << (10 \* iota)

MB

GB

TB

PB

EB

ZB

YB

)

The ability to attach a method such as String to any user-defined type makes it possible for arbitrary values to format themselves automatically for printing. Although you'll see it most often applied to structs, this technique is also useful for scalar types such as floating-point types like ByteSize.

func (b ByteSize) String() string {

switch {

case b >= YB:

return fmt.Sprintf("%.2fYB", b/YB)

case b >= ZB:

return fmt.Sprintf("%.2fZB", b/ZB)

case b >= EB:

return fmt.Sprintf("%.2fEB", b/EB)

case b >= PB:

return fmt.Sprintf("%.2fPB", b/PB)

case b >= TB:

return fmt.Sprintf("%.2fTB", b/TB)

case b >= GB:

return fmt.Sprintf("%.2fGB", b/GB)

case b >= MB:

return fmt.Sprintf("%.2fMB", b/MB)

case b >= KB:

return fmt.Sprintf("%.2fKB", b/KB)

}

return fmt.Sprintf("%.2fB", b)

}

The expression YB prints as 1.00YB, while ByteSize(1e13) prints as 9.09TB.

The use here of Sprintf to implement ByteSize's String method is safe (avoids recurring indefinitely) not because of a conversion but because it calls Sprintf with %f, which is not a string format: Sprintf will only call the String method when it wants a string, and %f wants a floating-point value.

**Variables**[**¶**](https://go.dev/doc/effective_go#variables)

Variables can be initialized just like constants but the initializer can be a general expression computed at run time.

var (

home = os.Getenv("HOME")

user = os.Getenv("USER")

gopath = os.Getenv("GOPATH")

)

**The init function**[**¶**](https://go.dev/doc/effective_go#init)

Finally, each source file can define its own niladic init function to set up whatever state is required. (Actually each file can have multiple init functions.) And finally means finally: init is called after all the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been initialized.

Besides initializations that cannot be expressed as declarations, a common use of init functions is to verify or repair correctness of the program state before real execution begins.

func init() {

if user == "" {

log.Fatal("$USER not set")

}

if home == "" {

home = "/home/" + user

}

if gopath == "" {

gopath = home + "/go"

}

// gopath may be overridden by --gopath flag on command line.

flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")

}

**Methods**[**¶**](https://go.dev/doc/effective_go#methods)

**Pointers vs. Values**[**¶**](https://go.dev/doc/effective_go#pointers_vs_values)

As we saw with ByteSize, methods can be defined for any named type (except a pointer or an interface); the receiver does not have to be a struct.

In the discussion of slices above, we wrote an Append function. We can define it as a method on slices instead. To do this, we first declare a named type to which we can bind the method, and then make the receiver for the method a value of that type.

type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {

// Body exactly the same as the Append function defined above.

}

This still requires the method to return the updated slice. We can eliminate that clumsiness by redefining the method to take a *pointer* to a ByteSlice as its receiver, so the method can overwrite the caller's slice.

func (p \*ByteSlice) Append(data []byte) {

slice := \*p

// Body as above, without the return.

\*p = slice

}

In fact, we can do even better. If we modify our function so it looks like a standard Write method, like this,

func (p \*ByteSlice) Write(data []byte) (n int, err error) {

slice := \*p

// Again as above.

\*p = slice

return len(data), nil

}

then the type \*ByteSlice satisfies the standard interface io.Writer, which is handy. For instance, we can print into one.

var b ByteSlice

fmt.Fprintf(&b, "This hour has %d days\n", 7)

We pass the address of a ByteSlice because only \*ByteSlice satisfies io.Writer. The rule about pointers vs. values for receivers is that value methods can be invoked on pointers and values, but pointer methods can only be invoked on pointers.

This rule arises because pointer methods can modify the receiver; invoking them on a value would cause the method to receive a copy of the value, so any modifications would be discarded. The language therefore disallows this mistake. There is a handy exception, though. When the value is addressable, the language takes care of the common case of invoking a pointer method on a value by inserting the address operator automatically. In our example, the variable b is addressable, so we can call its Write method with just b.Write. The compiler will rewrite that to (&b).Write for us.

By the way, the idea of using Write on a slice of bytes is central to the implementation of bytes.Buffer.

**Interfaces and other types**[**¶**](https://go.dev/doc/effective_go#interfaces_and_types)

**Interfaces**[**¶**](https://go.dev/doc/effective_go#interfaces)

Interfaces in Go provide a way to specify the behavior of an object: if something can do *this*, then it can be used *here*. We've seen a couple of simple examples already; custom printers can be implemented by a String method while Fprintf can generate output to anything with a Write method. Interfaces with only one or two methods are common in Go code, and are usually given a name derived from the method, such as io.Writer for something that implements Write.

A type can implement multiple interfaces. For instance, a collection can be sorted by the routines in package sort if it implements sort.Interface, which contains Len(), Less(i, j int) bool, and Swap(i, j int), and it could also have a custom formatter. In this contrived example Sequence satisfies both.

type Sequence []int

// Methods required by sort.Interface.

func (s Sequence) Len() int {

return len(s)

}

func (s Sequence) Less(i, j int) bool {

return s[i] < s[j]

}

func (s Sequence) Swap(i, j int) {

s[i], s[j] = s[j], s[i]

}

// Copy returns a copy of the Sequence.

func (s Sequence) Copy() Sequence {

copy := make(Sequence, 0, len(s))

return append(copy, s...)

}

// Method for printing - sorts the elements before printing.

func (s Sequence) String() string {

s = s.Copy() // Make a copy; don't overwrite argument.

sort.Sort(s)

str := "["

for i, elem := range s { // Loop is O(N²); will fix that in next example.

if i > 0 {

str += " "

}

str += fmt.Sprint(elem)

}

return str + "]"

}

**Conversions**[**¶**](https://go.dev/doc/effective_go#conversions)

The String method of Sequence is recreating the work that Sprint already does for slices. (It also has complexity O(N²), which is poor.) We can share the effort (and also speed it up) if we convert the Sequence to a plain []int before calling Sprint.

func (s Sequence) String() string {

s = s.Copy()

sort.Sort(s)

return fmt.Sprint([]int(s))

}

This method is another example of the conversion technique for calling Sprintf safely from a String method. Because the two types (Sequence and []int) are the same if we ignore the type name, it's legal to convert between them. The conversion doesn't create a new value, it just temporarily acts as though the existing value has a new type. (There are other legal conversions, such as from integer to floating point, that do create a new value.)

It's an idiom in Go programs to convert the type of an expression to access a different set of methods. As an example, we could use the existing type sort.IntSlice to reduce the entire example to this:

type Sequence []int

// Method for printing - sorts the elements before printing

func (s Sequence) String() string {

s = s.Copy()

sort.IntSlice(s).Sort()

return fmt.Sprint([]int(s))

}

Now, instead of having Sequence implement multiple interfaces (sorting and printing), we're using the ability of a data item to be converted to multiple types (Sequence, sort.IntSlice and []int), each of which does some part of the job. That's more unusual in practice but can be effective.

**Interface conversions and type assertions**[**¶**](https://go.dev/doc/effective_go#interface_conversions)

[Type switches](https://go.dev/doc/effective_go#type_switch) are a form of conversion: they take an interface and, for each case in the switch, in a sense convert it to the type of that case. Here's a simplified version of how the code under fmt.Printf turns a value into a string using a type switch. If it's already a string, we want the actual string value held by the interface, while if it has a String method we want the result of calling the method.

type Stringer interface {

String() string

}

var value interface{} // Value provided by caller.

switch str := value.(type) {

case string:

return str

case Stringer:

return str.String()

}

The first case finds a concrete value; the second converts the interface into another interface. It's perfectly fine to mix types this way.

What if there's only one type we care about? If we know the value holds a string and we just want to extract it? A one-case type switch would do, but so would a *type assertion*. A type assertion takes an interface value and extracts from it a value of the specified explicit type. The syntax borrows from the clause opening a type switch, but with an explicit type rather than the type keyword:

value.(typeName)

and the result is a new value with the static type typeName. That type must either be the concrete type held by the interface, or a second interface type that the value can be converted to. To extract the string we know is in the value, we could write:

str := value.(string)

But if it turns out that the value does not contain a string, the program will crash with a run-time error. To guard against that, use the "comma, ok" idiom to test, safely, whether the value is a string:

str, ok := value.(string)

if ok {

fmt.Printf("string value is: %q\n", str)

} else {

fmt.Printf("value is not a string\n")

}

If the type assertion fails, str will still exist and be of type string, but it will have the zero value, an empty string.

As an illustration of the capability, here's an if-else statement that's equivalent to the type switch that opened this section.

if str, ok := value.(string); ok {

return str

} else if str, ok := value.(Stringer); ok {

return str.String()

}

**Generality**[**¶**](https://go.dev/doc/effective_go#generality)

If a type exists only to implement an interface and will never have exported methods beyond that interface, there is no need to export the type itself. Exporting just the interface makes it clear the value has no interesting behavior beyond what is described in the interface. It also avoids the need to repeat the documentation on every instance of a common method.

In such cases, the constructor should return an interface value rather than the implementing type. As an example, in the hash libraries both crc32.NewIEEE and adler32.New return the interface type hash.Hash32. Substituting the CRC-32 algorithm for Adler-32 in a Go program requires only changing the constructor call; the rest of the code is unaffected by the change of algorithm.

A similar approach allows the streaming cipher algorithms in the various crypto packages to be separated from the block ciphers they chain together. The Block interface in the crypto/cipher package specifies the behavior of a block cipher, which provides encryption of a single block of data. Then, by analogy with the bufio package, cipher packages that implement this interface can be used to construct streaming ciphers, represented by the Stream interface, without knowing the details of the block encryption.

The crypto/cipher interfaces look like this:

type Block interface {

BlockSize() int

Encrypt(dst, src []byte)

Decrypt(dst, src []byte)

}

type Stream interface {

XORKeyStream(dst, src []byte)

}

Here's the definition of the counter mode (CTR) stream, which turns a block cipher into a streaming cipher; notice that the block cipher's details are abstracted away:

// NewCTR returns a Stream that encrypts/decrypts using the given Block in

// counter mode. The length of iv must be the same as the Block's block size.

func NewCTR(block Block, iv []byte) Stream

NewCTR applies not just to one specific encryption algorithm and data source but to any implementation of the Block interface and any Stream. Because they return interface values, replacing CTR encryption with other encryption modes is a localized change. The constructor calls must be edited, but because the surrounding code must treat the result only as a Stream, it won't notice the difference.

**Interfaces and methods**[**¶**](https://go.dev/doc/effective_go#interface_methods)

Since almost anything can have methods attached, almost anything can satisfy an interface. One illustrative example is in the http package, which defines the Handler interface. Any object that implements Handler can serve HTTP requests.

type Handler interface {

ServeHTTP(ResponseWriter, \*Request)

}

ResponseWriter is itself an interface that provides access to the methods needed to return the response to the client. Those methods include the standard Write method, so an http.ResponseWriter can be used wherever an io.Writer can be used. Request is a struct containing a parsed representation of the request from the client.

For brevity, let's ignore POSTs and assume HTTP requests are always GETs; that simplification does not affect the way the handlers are set up. Here's a trivial implementation of a handler to count the number of times the page is visited.

// Simple counter server.

type Counter struct {

n int

}

func (ctr \*Counter) ServeHTTP(w http.ResponseWriter, req \*http.Request) {

ctr.n++

fmt.Fprintf(w, "counter = %d\n", ctr.n)

}

(Keeping with our theme, note how Fprintf can print to an http.ResponseWriter.) In a real server, access to ctr.n would need protection from concurrent access. See the sync and atomic packages for suggestions.

For reference, here's how to attach such a server to a node on the URL tree.

import "net/http"

...

ctr := new(Counter)

http.Handle("/counter", ctr)

But why make Counter a struct? An integer is all that's needed. (The receiver needs to be a pointer so the increment is visible to the caller.)

// Simpler counter server.

type Counter int

func (ctr \*Counter) ServeHTTP(w http.ResponseWriter, req \*http.Request) {

\*ctr++

fmt.Fprintf(w, "counter = %d\n", \*ctr)

}

What if your program has some internal state that needs to be notified that a page has been visited? Tie a channel to the web page.

// A channel that sends a notification on each visit.

// (Probably want the channel to be buffered.)

type Chan chan \*http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req \*http.Request) {

ch <- req

fmt.Fprint(w, "notification sent")

}

Finally, let's say we wanted to present on /args the arguments used when invoking the server binary. It's easy to write a function to print the arguments.

func ArgServer() {

fmt.Println(os.Args)

}

How do we turn that into an HTTP server? We could make ArgServer a method of some type whose value we ignore, but there's a cleaner way. Since we can define a method for any type except pointers and interfaces, we can write a method for a function. The http package contains this code:

// The HandlerFunc type is an adapter to allow the use of

// ordinary functions as HTTP handlers. If f is a function

// with the appropriate signature, HandlerFunc(f) is a

// Handler object that calls f.

type HandlerFunc func(ResponseWriter, \*Request)

// ServeHTTP calls f(w, req).

func (f HandlerFunc) ServeHTTP(w ResponseWriter, req \*Request) {

f(w, req)

}

HandlerFunc is a type with a method, ServeHTTP, so values of that type can serve HTTP requests. Look at the implementation of the method: the receiver is a function, f, and the method calls f. That may seem odd but it's not that different from, say, the receiver being a channel and the method sending on the channel.

To make ArgServer into an HTTP server, we first modify it to have the right signature.

// Argument server.

func ArgServer(w http.ResponseWriter, req \*http.Request) {

fmt.Fprintln(w, os.Args)

}

ArgServer now has the same signature as HandlerFunc, so it can be converted to that type to access its methods, just as we converted Sequence to IntSlice to access IntSlice.Sort. The code to set it up is concise:

http.Handle("/args", http.HandlerFunc(ArgServer))

When someone visits the page /args, the handler installed at that page has value ArgServer and type HandlerFunc. The HTTP server will invoke the method ServeHTTP of that type, with ArgServer as the receiver, which will in turn call ArgServer (via the invocation f(w, req) inside HandlerFunc.ServeHTTP). The arguments will then be displayed.

In this section we have made an HTTP server from a struct, an integer, a channel, and a function, all because interfaces are just sets of methods, which can be defined for (almost) any type.

**The blank identifier**[**¶**](https://go.dev/doc/effective_go#blank)

We've mentioned the blank identifier a couple of times now, in the context of [for range loops](https://go.dev/doc/effective_go#for) and [maps](https://go.dev/doc/effective_go#maps). The blank identifier can be assigned or declared with any value of any type, with the value discarded harmlessly. It's a bit like writing to the Unix /dev/null file: it represents a write-only value to be used as a place-holder where a variable is needed but the actual value is irrelevant. It has uses beyond those we've seen already.

**The blank identifier in multiple assignment**[**¶**](https://go.dev/doc/effective_go#blank_assign)

The use of a blank identifier in a for range loop is a special case of a general situation: multiple assignment.

If an assignment requires multiple values on the left side, but one of the values will not be used by the program, a blank identifier on the left-hand-side of the assignment avoids the need to create a dummy variable and makes it clear that the value is to be discarded. For instance, when calling a function that returns a value and an error, but only the error is important, use the blank identifier to discard the irrelevant value.

if \_, err := os.Stat(path); os.IsNotExist(err) {

fmt.Printf("%s does not exist\n", path)

}

Occasionally you'll see code that discards the error value in order to ignore the error; this is terrible practice. Always check error returns; they're provided for a reason.

// Bad! This code will crash if path does not exist.

fi, \_ := os.Stat(path)

if fi.IsDir() {

fmt.Printf("%s is a directory\n", path)

}

**Unused imports and variables**[**¶**](https://go.dev/doc/effective_go#blank_unused)

It is an error to import a package or to declare a variable without using it. Unused imports bloat the program and slow compilation, while a variable that is initialized but not used is at least a wasted computation and perhaps indicative of a larger bug. When a program is under active development, however, unused imports and variables often arise and it can be annoying to delete them just to have the compilation proceed, only to have them be needed again later. The blank identifier provides a workaround.

This half-written program has two unused imports (fmt and io) and an unused variable (fd), so it will not compile, but it would be nice to see if the code so far is correct.

package main

import (

"fmt"

"io"

"log"

"os"

)

func main() {

fd, err := os.Open("test.go")

if err != nil {

log.Fatal(err)

}

// TODO: use fd.

}

To silence complaints about the unused imports, use a blank identifier to refer to a symbol from the imported package. Similarly, assigning the unused variable fd to the blank identifier will silence the unused variable error. This version of the program does compile.

package main

import (

"fmt"

"io"

"log"

"os"

)

var \_ = fmt.Printf // For debugging; delete when done.

var \_ io.Reader // For debugging; delete when done.

func main() {

fd, err := os.Open("test.go")

if err != nil {

log.Fatal(err)

}

// TODO: use fd.

\_ = fd

}

By convention, the global declarations to silence import errors should come right after the imports and be commented, both to make them easy to find and as a reminder to clean things up later.

**Import for side effect**[**¶**](https://go.dev/doc/effective_go#blank_import)

An unused import like fmt or io in the previous example should eventually be used or removed: blank assignments identify code as a work in progress. But sometimes it is useful to import a package only for its side effects, without any explicit use. For example, during its init function, the [net/http/pprof](https://go.dev/pkg/net/http/pprof/) package registers HTTP handlers that provide debugging information. It has an exported API, but most clients need only the handler registration and access the data through a web page. To import the package only for its side effects, rename the package to the blank identifier:

import \_ "net/http/pprof"

This form of import makes clear that the package is being imported for its side effects, because there is no other possible use of the package: in this file, it doesn't have a name. (If it did, and we didn't use that name, the compiler would reject the program.)

**Interface checks**[**¶**](https://go.dev/doc/effective_go#blank_implements)

As we saw in the discussion of [interfaces](https://go.dev/doc/effective_go#interfaces_and_types) above, a type need not declare explicitly that it implements an interface. Instead, a type implements the interface just by implementing the interface's methods. In practice, most interface conversions are static and therefore checked at compile time. For example, passing an \*os.File to a function expecting an io.Reader will not compile unless \*os.File implements the io.Reader interface.

Some interface checks do happen at run-time, though. One instance is in the [encoding/json](https://go.dev/pkg/encoding/json/) package, which defines a [Marshaler](https://go.dev/pkg/encoding/json/" \l "Marshaler) interface. When the JSON encoder receives a value that implements that interface, the encoder invokes the value's marshaling method to convert it to JSON instead of doing the standard conversion. The encoder checks this property at run time with a [type assertion](https://go.dev/doc/effective_go#interface_conversions) like:

m, ok := val.(json.Marshaler)

If it's necessary only to ask whether a type implements an interface, without actually using the interface itself, perhaps as part of an error check, use the blank identifier to ignore the type-asserted value:

if \_, ok := val.(json.Marshaler); ok {

fmt.Printf("value %v of type %T implements json.Marshaler\n", val, val)

}

One place this situation arises is when it is necessary to guarantee within the package implementing the type that it actually satisfies the interface. If a type—for example, [json.RawMessage](https://go.dev/pkg/encoding/json/" \l "RawMessage)—needs a custom JSON representation, it should implement json.Marshaler, but there are no static conversions that would cause the compiler to verify this automatically. If the type inadvertently fails to satisfy the interface, the JSON encoder will still work, but will not use the custom implementation. To guarantee that the implementation is correct, a global declaration using the blank identifier can be used in the package:

var \_ json.Marshaler = (\*RawMessage)(nil)

In this declaration, the assignment involving a conversion of a \*RawMessage to a Marshaler requires that \*RawMessage implements Marshaler, and that property will be checked at compile time. Should the json.Marshaler interface change, this package will no longer compile and we will be on notice that it needs to be updated.

The appearance of the blank identifier in this construct indicates that the declaration exists only for the type checking, not to create a variable. Don't do this for every type that satisfies an interface, though. By convention, such declarations are only used when there are no static conversions already present in the code, which is a rare event.

**Embedding**[**¶**](https://go.dev/doc/effective_go#embedding)

Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to “borrow” pieces of an implementation by *embedding* types within a struct or interface.

Interface embedding is very simple. We've mentioned the io.Reader and io.Writer interfaces before; here are their definitions.

type Reader interface {

Read(p []byte) (n int, err error)

}

type Writer interface {

Write(p []byte) (n int, err error)

}

The io package also exports several other interfaces that specify objects that can implement several such methods. For instance, there is io.ReadWriter, an interface containing both Read and Write. We could specify io.ReadWriter by listing the two methods explicitly, but it's easier and more evocative to embed the two interfaces to form the new one, like this:

// ReadWriter is the interface that combines the Reader and Writer interfaces.

type ReadWriter interface {

Reader

Writer

}

This says just what it looks like: A ReadWriter can do what a Reader does *and* what a Writer does; it is a union of the embedded interfaces. Only interfaces can be embedded within interfaces.

The same basic idea applies to structs, but with more far-reaching implications. The bufio package has two struct types, bufio.Reader and bufio.Writer, each of which of course implements the analogous interfaces from package io. And bufio also implements a buffered reader/writer, which it does by combining a reader and a writer into one struct using embedding: it lists the types within the struct but does not give them field names.

// ReadWriter stores pointers to a Reader and a Writer.

// It implements io.ReadWriter.

type ReadWriter struct {

\*Reader // \*bufio.Reader

\*Writer // \*bufio.Writer

}

The embedded elements are pointers to structs and of course must be initialized to point to valid structs before they can be used. The ReadWriter struct could be written as

type ReadWriter struct {

reader \*Reader

writer \*Writer

}

but then to promote the methods of the fields and to satisfy the io interfaces, we would also need to provide forwarding methods, like this:

func (rw \*ReadWriter) Read(p []byte) (n int, err error) {

return rw.reader.Read(p)

}

By embedding the structs directly, we avoid this bookkeeping. The methods of embedded types come along for free, which means that bufio.ReadWriter not only has the methods of bufio.Reader and bufio.Writer, it also satisfies all three interfaces: io.Reader, io.Writer, and io.ReadWriter.

There's an important way in which embedding differs from subclassing. When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one. In our example, when the Read method of a bufio.ReadWriter is invoked, it has exactly the same effect as the forwarding method written out above; the receiver is the reader field of the ReadWriter, not the ReadWriter itself.

Embedding can also be a simple convenience. This example shows an embedded field alongside a regular, named field.

type Job struct {

Command string

\*log.Logger

}

The Job type now has the Print, Printf, Println and other methods of \*log.Logger. We could have given the Logger a field name, of course, but it's not necessary to do so. And now, once initialized, we can log to the Job:

job.Println("starting now...")

The Logger is a regular field of the Job struct, so we can initialize it in the usual way inside the constructor for Job, like this,

func NewJob(command string, logger \*log.Logger) \*Job {

return &Job{command, logger}

}

or with a composite literal,

job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}

If we need to refer to an embedded field directly, the type name of the field, ignoring the package qualifier, serves as a field name, as it did in the Read method of our ReadWriter struct. Here, if we needed to access the \*log.Logger of a Job variable job, we would write job.Logger, which would be useful if we wanted to refine the methods of Logger.

func (job \*Job) Printf(format string, args ...interface{}) {

job.Logger.Printf("%q: %s", job.Command, fmt.Sprintf(format, args...))

}

Embedding types introduces the problem of name conflicts but the rules to resolve them are simple. First, a field or method X hides any other item X in a more deeply nested part of the type. If log.Logger contained a field or method called Command, the Command field of Job would dominate it.

Second, if the same name appears at the same nesting level, it is usually an error; it would be erroneous to embed log.Logger if the Job struct contained another field or method called Logger. However, if the duplicate name is never mentioned in the program outside the type definition, it is OK. This qualification provides some protection against changes made to types embedded from outside; there is no problem if a field is added that conflicts with another field in another subtype if neither field is ever used.

**Concurrency**[**¶**](https://go.dev/doc/effective_go#concurrency)

**Share by communicating**[**¶**](https://go.dev/doc/effective_go#sharing)

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

**Goroutines**[**¶**](https://go.dev/doc/effective_go#goroutines)

They're called *goroutines* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

Prefix a function or method call with the go keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's & notation for running a command in the background.)

go list.Sort() // run list.Sort concurrently; don't wait for it.

A function literal can be handy in a goroutine invocation.

func Announce(message string, delay time.Duration) {

go func() {

time.Sleep(delay)

fmt.Println(message)

}() // Note the parentheses - must call the function.

}

In Go, function literals are closures: the implementation makes sure the variables referred to by the function survive as long as they are active.

These examples aren't too practical because the functions have no way of signaling completion. For that, we need channels.

**Channels**[**¶**](https://go.dev/doc/effective_go#channels)

Like maps, channels are allocated with make, and the resulting value acts as a reference to an underlying data structure. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

ci := make(chan int) // unbuffered channel of integers

cj := make(chan int, 0) // unbuffered channel of integers

cs := make(chan \*os.File, 100) // buffered channel of pointers to Files

Unbuffered channels combine communication—the exchange of a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

There are lots of nice idioms using channels. Here's one to get us started. In the previous section we launched a sort in the background. A channel can allow the launching goroutine to wait for the sort to complete.

c := make(chan int) // Allocate a channel.

// Start the sort in a goroutine; when it completes, signal on the channel.

go func() {

list.Sort()

c <- 1 // Send a signal; value does not matter.

}()

doSomethingForAWhile()

<-c // Wait for sort to finish; discard sent value.

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

A buffered channel can be used like a semaphore, for instance to limit throughput. In this example, incoming requests are passed to handle, which sends a value into the channel, processes the request, and then receives a value from the channel to ready the “semaphore” for the next consumer. The capacity of the channel buffer limits the number of simultaneous calls to process.

var sem = make(chan int, MaxOutstanding)

func handle(r \*Request) {

sem <- 1 // Wait for active queue to drain.

process(r) // May take a long time.

<-sem // Done; enable next request to run.

}

func Serve(queue chan \*Request) {

for {

req := <-queue

go handle(req) // Don't wait for handle to finish.

}

}

Once MaxOutstanding handlers are executing process, any more will block trying to send into the filled channel buffer, until one of the existing handlers finishes and receives from the buffer.

This design has a problem, though: Serve creates a new goroutine for every incoming request, even though only MaxOutstanding of them can run at any moment. As a result, the program can consume unlimited resources if the requests come in too fast. We can address that deficiency by changing Serve to gate the creation of the goroutines:

func Serve(queue chan \*Request) {

for req := range queue {

sem <- 1

go func() {

process(req)

<-sem

}()

}

}

(Note that in Go versions before 1.22 this code has a bug: the loop variable is shared across all goroutines. See the [Go wiki](https://go.dev/wiki/LoopvarExperiment) for details.)

Another approach that manages resources well is to start a fixed number of handle goroutines all reading from the request channel. The number of goroutines limits the number of simultaneous calls to process. This Serve function also accepts a channel on which it will be told to exit; after launching the goroutines it blocks receiving from that channel.

func handle(queue chan \*Request) {

for r := range queue {

process(r)

}

}

func Serve(clientRequests chan \*Request, quit chan bool) {

// Start handlers

for i := 0; i < MaxOutstanding; i++ {

go handle(clientRequests)

}

<-quit // Wait to be told to exit.

}

**Channels of channels**[**¶**](https://go.dev/doc/effective_go#chan_of_chan)

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. A common use of this property is to implement safe, parallel demultiplexing.

In the example in the previous section, handle was an idealized handler for a request but we didn't define the type it was handling. If that type includes a channel on which to reply, each client can provide its own path for the answer. Here's a schematic definition of type Request.

type Request struct {

args []int

f func([]int) int

resultChan chan int

}

The client provides a function and its arguments, as well as a channel inside the request object on which to receive the answer.

func sum(a []int) (s int) {

for \_, v := range a {

s += v

}

Return

}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}

// Send request

clientRequests <- request

// Wait for response.

fmt.Printf("answer: %d\n", <-request.resultChan)

On the server side, the handler function is the only thing that changes.

func handle(queue chan \*Request) {

for req := range queue {

req.resultChan <- req.f(req.args)

}

}

There's clearly a lot more to do to make it realistic, but this code is a framework for a rate-limited, parallel, non-blocking RPC system, and there's not a mutex in sight.

**Parallelization**[**¶**](https://go.dev/doc/effective_go#parallel)

Another application of these ideas is to parallelize a calculation across multiple CPU cores. If the calculation can be broken into separate pieces that can execute independently, it can be parallelized, with a channel to signal when each piece completes.

Let's say we have an expensive operation to perform on a vector of items, and that the value of the operation on each item is independent, as in this idealized example.

type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].

func (v Vector) DoSome(i, n int, u Vector, c chan int) {

for ; i < n; i++ {

v[i] += u.Op(v[i])

}

c <- 1 // signal that this piece is done

}

We launch the pieces independently in a loop, one per CPU. They can complete in any order but it doesn't matter; we just count the completion signals by draining the channel after launching all the goroutines.

const numCPU = 4 // number of CPU cores

func (v Vector) DoAll(u Vector) {

c := make(chan int, numCPU) // Buffering optional but sensible.

for i := 0; i < numCPU; i++ {

go v.DoSome(i\*len(v)/numCPU, (i+1)\*len(v)/numCPU, u, c)

}

// Drain the channel.

for i := 0; i < numCPU; i++ {

<-c // wait for one task to complete

}

// All done.

}

Rather than create a constant value for numCPU, we can ask the runtime what value is appropriate. The function [runtime.NumCPU](https://go.dev/pkg/runtime" \l "NumCPU) returns the number of hardware CPU cores in the machine, so we could write

var numCPU = runtime.NumCPU()

There is also a function [runtime.GOMAXPROCS](https://go.dev/pkg/runtime" \l "GOMAXPROCS), which reports (or sets) the user-specified number of cores that a Go program can have running simultaneously. It defaults to the value of runtime.NumCPU but can be overridden by setting the similarly named shell environment variable or by calling the function with a positive number. Calling it with zero just queries the value. Therefore if we want to honor the user's resource request, we should write

var numCPU = runtime.GOMAXPROCS(0)

Be sure not to confuse the ideas of concurrency—structuring a program as independently executing components—and parallelism—executing calculations in parallel for efficiency on multiple CPUs. Although the concurrency features of Go can make some problems easy to structure as parallel computations, Go is a concurrent language, not a parallel one, and not all parallelization problems fit Go's model. For a discussion of the distinction, see the talk cited in [this blog post](https://go.dev/blog/concurrency-is-not-parallelism).

**A leaky buffer**[**¶**](https://go.dev/doc/effective_go#leaky_buffer)

The tools of concurrent programming can even make non-concurrent ideas easier to express. Here's an example abstracted from an RPC package. The client goroutine loops receiving data from some source, perhaps a network. To avoid allocating and freeing buffers, it keeps a free list, and uses a buffered channel to represent it. If the channel is empty, a new buffer gets allocated. Once the message buffer is ready, it's sent to the server on serverChan.

var freeList = make(chan \*Buffer, 100)

var serverChan = make(chan \*Buffer)

func client() {

for {

var b \*Buffer

// Grab a buffer if available; allocate if not.

select {

case b = <-freeList:

// Got one; nothing more to do.

default:

// None free, so allocate a new one.

b = new(Buffer)]

}

load(b) // Read next message from the net.

serverChan <- b // Send to server.

}

}

The server loop receives each message from the client, processes it, and returns the buffer to the free list.

func server() {

for {

b := <-serverChan // Wait for work.

process(b)

// Reuse buffer if there's room.

select {

case freeList <- b:

// Buffer on free list; nothing more to do.

default:

// Free list full, just carry on.

}

}

}

The client attempts to retrieve a buffer from freeList; if none is available, it allocates a fresh one. The server's send to freeList puts b back on the free list unless the list is full, in which case the buffer is dropped on the floor to be reclaimed by the garbage collector. (The default clauses in the select statements execute when no other case is ready, meaning that the selects never block.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

**Errors**[**¶**](https://go.dev/doc/effective_go#errors)

Library routines must often return some sort of error indication to the caller. As mentioned earlier, Go's multivalue return makes it easy to return a detailed error description alongside the normal return value. It is good style to use this feature to provide detailed error information. For example, as we'll see, os.Open doesn't just return a nil pointer on failure, it also returns an error value that describes what went wrong.

By convention, errors have type error, a simple built-in interface.

type error interface {

Error() string

}

A library writer is free to implement this interface with a richer model under the covers, making it possible not only to see the error but also to provide some context. As mentioned, alongside the usual \*os.File return value, os.Open also returns an error value. If the file is opened successfully, the error will be nil, but when there is a problem, it will hold an os.PathError:

// PathError records an error and the operation and

// file path that caused it.

type PathError struct {

Op string // "open", "unlink", etc.

Path string // The associated file.

Err error // Returned by the system call.

}

func (e \*PathError) Error() string {

return e.Op + " " + e.Path + ": " + e.Err.Error()

}

PathError's Error generates a string like this:

open /etc/passwx: no such file or directory

Such an error, which includes the problematic file name, the operation, and the operating system error it triggered, is useful even if printed far from the call that caused it; it is much more informative than the plain "no such file or directory".

When feasible, error strings should identify their origin, such as by having a prefix naming the operation or package that generated the error. For example, in package image, the string representation for a decoding error due to an unknown format is "image: unknown format".

Callers that care about the precise error details can use a type switch or a type assertion to look for specific errors and extract details. For PathErrors this might include examining the internal Err field for recoverable failures.

for try := 0; try < 2; try++ {

file, err = os.Create(filename)

if err == nil {

return

}

if e, ok := err.(\*os.PathError); ok && e.Err == syscall.ENOSPC {

deleteTempFiles() // Recover some space.

continue

}

return

}

The second if statement here is another [type assertion](https://go.dev/doc/effective_go#interface_conversions). If it fails, ok will be false, and e will be nil. If it succeeds, ok will be true, which means the error was of type \*os.PathError, and then so is e, which we can examine for more information about the error.

**Panic**[**¶**](https://go.dev/doc/effective_go#panic)

The usual way to report an error to a caller is to return an error as an extra return value. The canonical Read method is a well-known instance; it returns a byte count and an error. But what if the error is unrecoverable? Sometimes the program simply cannot continue.

For this purpose, there is a built-in function panic that in effect creates a run-time error that will stop the program (but see the next section). The function takes a single argument of arbitrary type—often a string—to be printed as the program dies. It's also a way to indicate that something impossible has happened, such as exiting an infinite loop.

// A toy implementation of cube root using Newton's method.

func CubeRoot(x float64) float64 {

z := x/3 // Arbitrary initial value

for i := 0; i < 1e6; i++ {

prevz := z

z -= (z\*z\*z-x) / (3\*z\*z)

if veryClose(z, prevz) {

return z

}

}

// A million iterations has not converged; something is wrong.

panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))

}

This is only an example but real library functions should avoid panic. If the problem can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program. One possible counterexample is during initialization: if the library truly cannot set itself up, it might be reasonable to panic, so to speak.

var user = os.Getenv("USER")

func init() {

if user == "" {

panic("no value for $USER")

}

}

**Recover**[**¶**](https://go.dev/doc/effective_go#recover)

When panic is called, including implicitly for run-time errors such as indexing a slice out of bounds or failing a type assertion, it immediately stops execution of the current function and begins unwinding the stack of the goroutine, running any deferred functions along the way. If that unwinding reaches the top of the goroutine's stack, the program dies. However, it is possible to use the built-in function recover to regain control of the goroutine and resume normal execution.

A call to recover stops the unwinding and returns the argument passed to panic. Because the only code that runs while unwinding is inside deferred functions, recover is only useful inside deferred functions.

One application of recover is to shut down a failing goroutine inside a server without killing the other executing goroutines.

func server(workChan <-chan \*Work) {

for work := range workChan {

go safelyDo(work)

}

}

func safelyDo(work \*Work) {

defer func() {

if err := recover(); err != nil {

log.Println("work failed:", err)

}

}()

do(work)

}

In this example, if do(work) panics, the result will be logged and the goroutine will exit cleanly without disturbing the others. There's no need to do anything else in the deferred closure; calling recover handles the condition completely.

Because recover always returns nil unless called directly from a deferred function, deferred code can call library routines that themselves use panic and recover without failing. As an example, the deferred function in safelyDo might call a logging function before calling recover, and that logging code would run unaffected by the panicking state.

With our recovery pattern in place, the do function (and anything it calls) can get out of any bad situation cleanly by calling panic. We can use that idea to simplify error handling in complex software. Let's look at an idealized version of a regexp package, which reports parsing errors by calling panic with a local error type. Here's the definition of Error, an error method, and the Compile function.

// Error is the type of a parse error; it satisfies the error interface.

type Error string

func (e Error) Error() string {

return string(e)

}

// error is a method of \*Regexp that reports parsing errors by

// panicking with an Error.

func (regexp \*Regexp) error(err string) {

panic(Error(err))

}

// Compile returns a parsed representation of the regular expression.

func Compile(str string) (regexp \*Regexp, err error) {

regexp = new(Regexp)

// doParse will panic if there is a parse error.

defer func() {

if e := recover(); e != nil {

regexp = nil // Clear return value.

err = e.(Error) // Will re-panic if not a parse error.

}

}()

return regexp.doParse(str), nil

}

If doParse panics, the recovery block will set the return value to nil—deferred functions can modify named return values. It will then check, in the assignment to err, that the problem was a parse error by asserting that it has the local type Error. If it does not, the type assertion will fail, causing a run-time error that continues the stack unwinding as though nothing had interrupted it. This check means that if something unexpected happens, such as an index out of bounds, the code will fail even though we are using panic and recover to handle parse errors.

With error handling in place, the error method (because it's a method bound to a type, it's fine, even natural, for it to have the same name as the builtin error type) makes it easy to report parse errors without worrying about unwinding the parse stack by hand:

if pos == 0 {

re.error("'\*' illegal at start of expression")

}

Useful though this pattern is, it should be used only within a package. Parse turns its internal panic calls into error values; it does not expose panics to its client. That is a good rule to follow.

By the way, this re-panic idiom changes the panic value if an actual error occurs. However, both the original and new failures will be presented in the crash report, so the root cause of the problem will still be visible. Thus this simple re-panic approach is usually sufficient—it's a crash after all—but if you want to display only the original value, you can write a little more code to filter unexpected problems and re-panic with the original error. That's left as an exercise for the reader.

**A web server**[**¶**](https://go.dev/doc/effective_go#web_server)

Let's finish with a complete Go program, a web server. This one is actually a kind of web re-server. Google provides a service at chart.apis.google.com that does automatic formatting of data into charts and graphs. It's hard to use interactively, though, because you need to put the data into the URL as a query. The program here provides a nicer interface to one form of data: given a short piece of text, it calls on the chart server to produce a QR code, a matrix of boxes that encode the text. That image can be grabbed with your cell phone's camera and interpreted as, for instance, a URL, saving you typing the URL into the phone's tiny keyboard.

Here's the complete program. An explanation follows.

package main

import (

"flag"

"html/template"

"log"

"net/http"

)

var addr = flag.String("addr", ":1718", "http service address") // Q=17, R=18

var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {

flag.Parse()

http.Handle("/", http.HandlerFunc(QR))

err := http.ListenAndServe(\*addr, nil)

if err != nil {

log.Fatal("ListenAndServe:", err)

}

}

func QR(w http.ResponseWriter, req \*http.Request) {

templ.Execute(w, req.FormValue("s"))

}

const templateStr = `

<html>

<head>

<title>QR Link Generator</title>

</head>

<body>

{{if .}}

<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{.}}" />

<br>

{{.}}

<br>

<br>

{{end}}

<form action="/" name=f method="GET">

<input maxLength=1024 size=70 name=s value="" title="Text to QR Encode">

<input type=submit value="Show QR" name=qr>

</form>

</body>

</html>

`

The pieces up to main should be easy to follow. The one flag sets a default HTTP port for our server. The template variable templ is where the fun happens. It builds an HTML template that will be executed by the server to display the page; more about that in a moment.

The main function parses the flags and, using the mechanism we talked about above, binds the function QR to the root path for the server. Then http.ListenAndServe is called to start the server; it blocks while the server runs.

QR just receives the request, which contains form data, and executes the template on the data in the form value named s.

The template package html/template is powerful; this program just touches on its capabilities. In essence, it rewrites a piece of HTML text on the fly by substituting elements derived from data items passed to templ.Execute, in this case the form value. Within the template text (templateStr), double-brace-delimited pieces denote template actions. The piece from {{if .}} to {{end}} executes only if the value of the current data item, called . (dot), is non-empty. That is, when the string is empty, this piece of the template is suppressed.

The two snippets {{.}} say to show the data presented to the template—the query string—on the web page. The HTML template package automatically provides appropriate escaping so the text is safe to display.

The rest of the template string is just the HTML to show when the page loads. If this is too quick an explanation, see the [documentation](https://go.dev/pkg/html/template/) for the template package for a more thorough discussion.

And there you have it: a useful web server in a few lines of code plus some data-driven HTML text. Go is powerful enough to make a lot happen in a few lines.

I

Iaas Languages

**Terraform Standards (HCL)**

1. Project Structure

Organize code using:

modules/: Reusable components

environments/: Separate configs for dev, staging, prod

Common files:

main.tf, variables.tf, outputs.tf, versions.tf, backend.tf, providers.tf

Example :

terraform/

├── modules/

│ └── ec2/

│ ├── main.tf

│ ├── variables.tf

│ └── outputs.tf

├── environments/

│ └── dev/

│ ├── main.tf

│ ├── terraform.tfvars

│ └── backend.tf

├── provider.tf

└── versions.tf

2. File Naming Conventions

main.tf: Core resource definitions

variables.tf: All input variables with descriptions and types

outputs.tf: Export key resource values

backend.tf: Remote state config

versions.tf: Required provider and Terraform version

3. Variable Standards

variable "instance\_type" {

description = "EC2 instance type"

type = string

default = "t2.micro"

}

Always provide descriptions and types.

Avoid hardcoding values in main.tf.

4. Output Standards

output "instance\_ip" {

description = "Public IP of EC2"

value = aws\_instance.web.public\_ip

}

Use outputs.tf to expose critical data.

Use sensitive = true for secrets.

5. Use Modules

module "web" {

source = "./modules/ec2"

instance\_type = var.instance\_type

}

Promote reuse, readability, and testability.

Separate logic from environments.

6. Version Locking

terraform {

required\_version = ">= 1.3.0"

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 5.0"

}

}

}

Always pin provider and Terraform versions.

7. Remote State Management

terraform {

backend "s3" {

bucket = "my-terraform-state"

key = "dev/terraform.tfstate"

region = "us-east-1"

}

}

Use backends like S3, Azure Storage, or GCS.

Enable state locking where possible (e.g., DynamoDB for AWS).

8. Security Standards

Never hardcode secrets.

Use secret managers (AWS Secrets Manager, Vault).

Use sensitive = true in outputs.

9. Naming Conventions

Use lowercase and hyphens: web-server, db-instance

Resource names should reflect their function or environment

10. Validation & Linting

Format: terraform fmt -recursive

Validate: terraform validate

Lint: Use tools like tflint, checkov, terraform-compliance

**YAML standards**

**1. Template File Naming**

* Use .yaml format (preferred over .json for readability).
* Naming conventions:
  + networking.yaml
  + ec2-stack.yaml
  + vpc-module.yaml

**2. Template Structure (YAML)**

AWSTemplateFormatVersion: '2010-09-09'

Description: Provision an EC2 instance

Parameters:

InstanceType:

Type: String

Default: t2.micro

Description: EC2 instance type

Resources:

MyInstance:

Type: AWS::EC2::Instance

Properties:

ImageId: ami-0abcdef1234567890

InstanceType: !Ref InstanceType

Outputs:

InstanceId:

Description: EC2 Instance ID

Value: !Ref MyInstance

**3. Standards & Best Practices**

**Use YAML**

* Easier to read and comment.
* Less verbose than JSON.

**Follow This Logical Order:**

1. AWSTemplateFormatVersion
2. Description
3. Metadata (optional)
4. Parameters
5. Mappings
6. Conditions
7. Resources
8. Outputs

**4. Parameter Usage**

Parameters:

Env:

Type: String

AllowedValues: [dev, staging, prod]

Default: dev

Description: Deployment environment

* Always define defaults when possible.
* Use AllowedValues to restrict input.

**5. Outputs**

Outputs:

PublicIP:

Description: "Public IP of the instance"

Value: !GetAtt MyInstance.PublicIp

Export:

Name: !Sub "${AWS::StackName}-PublicIP"

* Use Outputs to pass values between stacks (cross-stack reference).
* Include Export names where needed.

**6. Modularization via Nested Stacks**

Use nested stacks to break down large templates.

yaml

CopyEdit

Resources:

NetworkStack:

Type: AWS::CloudFormation::Stack

Properties:

TemplateURL: https://s3.amazonaws.com/mybucket/networking.yaml

**7. Security Best Practices**

* Never hardcode secrets — use AWS::SecretsManager::Secret or SSM Parameter Store.
* Use NoEcho: true for sensitive parameters:

Parameters:

DBPassword:

Type: String

NoEcho: true

**8. Formatting & Linting**

Use tools like:

* [cfn-lint](https://github.com/aws-cloudformation/cfn-lint)
* prettier-plugin-cfn for formatting
* VS Code extension: **AWS CloudFormation Linter**

**9. Naming Conventions**

* Logical IDs: CamelCase (e.g., MySecurityGroup)
* Resource Names: kebab-case or snake\_case via Name tag (e.g., my-app-sg)

Tags:

- Key: Name

Value: !Sub "${Env}-app-server"

**10. Use Intrinsic Functions Properly**

Examples:

!Ref ResourceName

!GetAtt ResourceName.Attribute

!Sub "arn:aws:s3:::${BucketName}/\*"

!If, !Equals, !Join, !Select, !Split, !ImportValue

**Bash Scripting Standards**

**1. File Naming & Structure**

* Use .sh extension (e.g., deploy.sh)
* Shebang at top:

#!/bin/bash

* Group logic into **functions**.

**2. Script Formatting**

* Use **2 or 4 spaces** for indentation (no tabs).
* Keep lines under **80–100 characters**.
* Use lowercase for variables (my\_var), UPPER\_SNAKE\_CASE for constants.

**3. Safe Scripting Practices**

* Start scripts with:

set -euo pipefail

IFS=$'\n\t'

This ensures:

* + Exit on error (-e)
  + Treat unset vars as errors (-u)
  + Fail on pipeline errors (-o pipefail)

**4. Variable Declaration**

readonly SCRIPT\_VERSION="1.0.0"

name="example"

* Use readonly or declare -r for constants.

**5. Functions**

greet() {

echo "Hello, $1!"

}

* Use parentheses: function\_name() {}
* Prefer meaningful names: validate\_input, backup\_logs

**6. Input Validation**

if [[ $# -lt 1 ]]; then

echo "Usage: $0 <arg>" >&2

exit 1

fi

**7. Logging**

log\_info() {

echo "[INFO] $1"

}

log\_error() {

echo "[ERROR] $1" >&2

}

**8. Quoting & Globbing**

* Always quote variables: "$var"
* Avoid word splitting and globbing issues.

**9. Exit Codes**

* Use appropriate exit 0 (success) or exit 1 (error).
* Check return codes explicitly:

if ! cp file1 file2; then

echo "Copy failed" >&2

exit 1

fi

**10. Linting Tools**

* [shellcheck](https://www.shellcheck.net/) – Linter and static analyzer
* Format using shfmt

**Dockerfile Standards**

**Use Minimal, Specific Base Images**

**Standard**

Always use specific versions and minimal images (slim, alpine) to reduce size and attack surface.

**Example**

FROM python:3.11-slim

Avoid:

FROM python:latest

**Use Multi-Stage Builds**

**Standard**

Use multi-stage builds to separate build-time dependencies from runtime.

**Example**

FROM node:20 AS builder

WORKDIR /app

COPY package\*.json ./

RUN npm ci

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=builder /app/dist /usr/share/nginx/html

**Use .dockerignore to Avoid Copying Unnecessary Files**

**Standard**

Prevent sensitive files and unnecessary content from being sent to the Docker daemon.

**Example: .dockerignore**

.git

node\_modules

.env

Dockerfile\*

README.md

**Use COPY Instead of ADD**

**Standard**

Use COPY unless you specifically need ADD's capabilities (e.g., auto-extraction or remote URL download).

**Example**

COPY requirements.txt .

**Define Arguments (ARG) and Environment Variables (ENV) Properly**

**Standard**

Use ARG for build-time inputs and ENV for runtime environment configuration.

**Example**

ARG APP\_VERSION=1.0.0

ENV APP\_VERSION=$APP\_VERSION

From the command line:

docker build --build-arg APP\_VERSION=2.0.0 .

**Use Labels for Metadata**

**Standard**

Follow [OCI image labels](https://github.com/opencontainers/image-spec/blob/main/annotations.md).

**Example**

LABEL org.opencontainers.image.title="My App" \

org.opencontainers.image.version="1.0.0" \

org.opencontainers.image.authors="you@example.com"

**Reduce Image Layers**

**Standard**

Combine RUN instructions to minimize image layers and clean up temp files.

**Example**

RUN apt-get update && apt-get install -y curl git \

&& rm -rf /var/lib/apt/lists/\*

**Set a Working Directory**

**Standard**

Use WORKDIR instead of manually cd-ing.

**Example**

WORKDIR /app

COPY . .

**Specify ENTRYPOINT and CMD Correctly**

**Standard**

* ENTRYPOINT: fixed executable
* CMD: default arguments

**Example**

ENTRYPOINT ["python", "main.py"]

CMD ["--help"]

**Don't Run as Root**

**Standard**

Create and switch to a non-root user for security in production.

**Example**

RUN addgroup --system app && adduser --system --ingroup app appuser

USER appuser

**Add Health Checks**

**Standard**

Use HEALTHCHECK to monitor container status.

**Example**

HEALTHCHECK CMD curl --fail http://localhost:8080/health || exit 1

**Use BuildKit Variables for Advanced Builds**

**Standard**

Support BuildKit features using --build-arg and RUN --mount.

**Example: Using Build Arguments**

ARG TARGETPLATFORM

RUN echo "Building for $TARGETPLATFORM"

**Example: Using Build Secrets (BuildKit)**

# syntax=docker/dockerfile:1.4

RUN --mount=type=secret,id=mysecret cat /run/secrets/mysecret

Then build with:

DOCKER\_BUILDKIT=1 docker build --secret id=mysecret,src=mysecret.txt .