The tidyverse style guide

Hadley Wickham

Contents

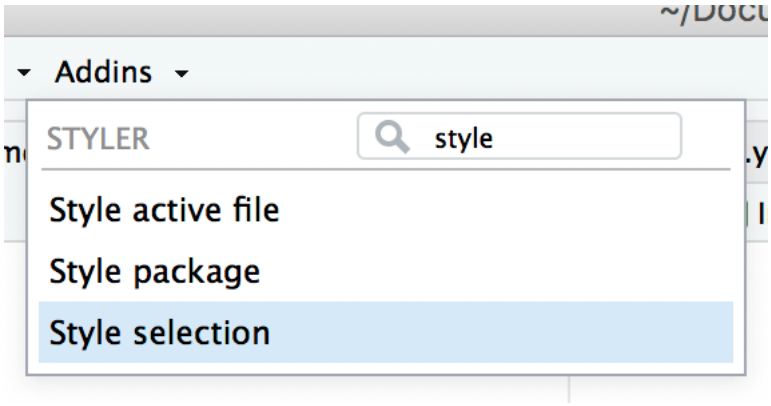
Welcome

Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. This site describes the style used throughout the tidyverse. It was originally derived from Google's R style guide, but has evolved and expanded considerably over the years.

All style guides are fundamentally opinionated. Some decisions genuinely do make code easier to use (especially matching indenting to programming structure), but many decisions are arbitrary. The most important thing about a style guide is that it provides consistency, making code easier to write because you need to make fewer decisions.

Two R packages support this style guide:

- styler allows you to interactively restyle selected text, files, or entire projects. It includes an RStudio add-in, the easiest way to re-style existing code.



- lintr performs automated checks to confirm that your conform to the style guide.

Part I

Analyses

Chapter 1

Files

1.1 Names

File names should be meaningful and end in `.R`. Avoid using special characters in file names - stick with numbers, letters, -, and _.

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

If files should be run in a particular order, prefix them with numbers. If it seems likely you'll have more than 10 files, left pad with zero:

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

If you later realise that you've missed some steps, it's tempting to use `02a`, `02b`, etc. However, I think it's generally better to bite the bullet and rename all files.

Pay attention to capitalization, since you, or some of your collaborators, might be using an operating system with a case-insensitive file system (e.g., Microsoft Windows or OS X) which can lead to problems with (case-sensitive) revision control systems. Prefer file names that are all lower case, and never have names that differ only in their capitalization.

1.2 Structure

Use commented lines of - and = to break up your file into easily readable chunks.

```
# Load data ---------------------------

# Plot data ---------------------------
```

If your script uses add-on packages, load them all at once at the very beginning of the file. This is more transparent than sprinkling `library()` calls throughout your code or having hidden dependencies that are loaded in a startup file, such as `.Rprofile`.

Chapter 2

Syntax

2.1 Object names

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> — Phil Karlton

Variable and function names should use only lowercase letters, numbers, and `_`.

Use underscores (`_`) to separate words within a name. Base R uses dots in function names (`contrib.url()`) and class names (`data.frame`), but it's better to reserve dots exclusively for the S3 object system. In S3, methods are given the name `function.class`; if you also use `.` in function and class names you end up with confusing methods like `as.data.frame.data.frame()`.

Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!).

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

Where possible, avoid re-using names of common functions and variables. This will cause confusion for the readers of your code.

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

2.2 Spacing

Put a space before and after = when naming arguments in function calls. Most infix operators (==, +, -, <-, etc.) are also surrounded by spaces, except those with relatively high precedence: ^, :, ::, and :::. Tildes

(~) in formulas with both LHS and RHS are surrounded by a space, if there is no LHS, put no space after ~.

Always put a space after a comma, never before, just like in regular English.
To highlight operator precedence, use parentheses rather than irregular spacing.

```
# Good
average <- mean((feet / 12) + inches, na.rm = TRUE)
sqrt(x^2 + y^2)
x <- 1:10
base::get
y ~ x
tribble(
  ~col1, ~col2,
  "a", "b"
)

# Bad
average<-mean(feet/12 + inches,na.rm=TRUE)
sqrt(x ^ 2 + y ^ 2)
x <- 1 : 10
base :: get
y~x
```

Place a space before (, except when it's part of a function call.

```
# Good
if (debug) show(x)
plot(x, y)

# Bad
if(debug)show(x)
plot (x, y)
```

Extra spacing (i.e., more than one space in a row) is ok if it improves alignment of equal signs or assignments (<-).

```
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

Do not place spaces around code in parentheses or square brackets (unless there's a comma, in which case see above).

```
# Good
if (debug) do(x)
diamonds[5, ]

# Bad
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

Do not put a space after the tidy evaluation bang-bang (!!) and bang-bang-bang (!!!) operators.

```
# Good
call(!!xyz)
```

```
# Bad
call(!! xyz)
call( !! xyz)
call(! !xyz)
```

2.3 Argument names

A function's arguments typically fall into two broad categories: one supplies the **data** to compute on; the other controls **details** of computation. When you call a function, you typically omit the names of data arguments, because they are used so commonly. If you override the default value of an argument, use the full name:

```
# Good
mean(1:10, na.rm = TRUE)

# Bad
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

Avoid partial matching.

2.4 Indenting

Curly braces, `{}`, define the most important hierarchy of R code. To make this hierarchy easy to see, always indent the code inside `{}` by two spaces.

A symmetrical arrangement helps finding related braces: The opening brace is the last, the closing brace is the first non-whitespace character in a line. Code that is related to a brace (e.g., an `if` clause, a function declaration, a trailing comma, ...) must be on the same line.

```
# Good
if (y < 0 && debug) {
  message("y is negative")
}

if (y == 0) {
  if (x > 0) {
    log(x)
  } else {
    message("x is negative or zero")
  }
} else {
  y ^ x
}

test_that("call1 returns an ordered factor", {
  expect_s3_class(call1(x, y), c("factor", "ordered"))
})

tryCatch(
  {
```

```
    x <- scan()
    cat("Total: ", sum(x), "\n", sep = "")
  },
  interrupt = function(e) {
    message("Aborted by user")
  }
)

# Bad
if (y < 0 && debug)
message("Y is negative")

if (y == 0)
{
    if (x > 0) {
      log(x)
    } else {
  message("x is negative or zero")
    }
} else { y ^ x }
```

It's ok to drop the curly braces if you have a very simple and short `if` statement that fits on one line. If you have any doubt, it's better to use the full form.

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

2.5 Long lines

Strive to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font. If you find yourself running out of room, this is a good indication that you should encapsulate some of the work in a separate function. If a function call is too long to fit on a single line, use one line each for the function name, each argument, and the closing `)`. This makes the code easier to read and to change later.

```
# Good
do_something_very_complicated(
  something = "that",
  requires = many,
  arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
                              )
```

As described under Argument names, you can omit the argument names for very common arguments (i.e. for arguments that are used in almost every invocation of the function). Short unnamed arguments can also go on the same line as the function name, even if the whole function call spans multiple lines.

```
map(x, f,
  extra_argument_a = 10,
```

```
  extra_argument_b = c(1, 43, 390, 210209)
)
```

You may also place several arguments on the same line if they are closely related to each other, e.g., strings in calls to `paste()` or `stop()`. When building strings, where possible match one line of code to one line of output.

```
# Good
paste0(
  "Requirement: ", requires, "\n",
  "Result: ", result, "\n"
)

# Bad
paste0(
  "Requirement: ", requires,
  "\n", "Result: ",
  result, "\n")
```

2.6 Assignment

Use <-, not =, for assignment.

```
# Good
x <- 5

# Bad
x = 5
```

2.7 Semicolons

Don't put ; at the end of a line, and don't use ; to put multiple commands on one line.

2.8 Quotes

Use ", not ', for quoting text. The only exception is when the text already contains double quotes and no single quotes.

```
# Good
"Text"
'Text with "quotes"'
'<a href="http://style.tidyverse.org">A link</a>'

# Bad
'Text'
'Text with "double" and \'single\' quotes'
```

Chapter 3

Functions

3.1 Naming

Use verbs for function names, where possible.

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()
```

3.2 Long lines

If a function definition runs over multiple lines, indent the second line to where the definition starts.

```
# Good
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
  # As usual code is indented by two spaces.
}

# Bad
long_function_name <- function(a = "a long argument",
  b = "another argument",
  c = "another long argument") {
  # Here it's hard to spot where the definition ends and the
  # code begins
}
```

3.3 `return()`

Only use `return()` for early returns. Otherwise rely on R to return the result of the last evaluated expression.

```
# Good
find_abs <- function(x, y) {
  if (x > 0) return(x)
  x * -1
}
add_two <- function(x, y) {
  x + y
}

# Bad
add_two <- function(x, y) {
  return(x + y)
}
```

If your function is called primarily for its side-effects (like printing, plotting, or saving to disk), it should return the first argument invisibly. This makes it possible to use the function as part of a pipe. `print` methods should usually do this, like this example from httr:

```
print.url <- function(x, ...) {
  cat("Url: ", build_url(x), "\n", sep = "")
  invisible(x)
}
```

3.4 Comments

In code, use comments to explain the "why" not the "what" or "how". Each line of a comment should begin with the comment symbol and a single space: #.

3.5 Design

There are two main design principles to bear in mind:

- A function should do one thing well.

  A function should be called either because it has side-effects or because it returns a value; not both. Strive to keep blocks within a function on one screen. 20-30 lines per function are common. For functions that are significantly longer, consider splitting it into smaller functions.

- A function should be easily understandable in isolation.

  Avoid global options. If your function has a transient side-effect (i.e. you need to create a temporary file or set an option), clean up after yourself with `on.exit()`.

Chapter 4

Pipes

4.1 Introduction

Use %>% when you find yourself composing three or more functions together into a nested call, or creating intermediate objects that you don't care about. Put each verb on its own line. This makes it simpler to rearrange them later, and makes it harder to overlook a step. It is ok to keep a one-step pipe in one line.

```
# Good
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)

foo_foo %>% fall("asleep")

# Bad
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)

iris %>% group_by(up) %>% summarize_all(mean) %>%
  ungroup %>% gather(field_mice, foo_foo, -on) %>%
  arrange(head)

bop(
  scoop(
    hop(foo_foo, through = forest),
    up = field_mice
  ),
  on = head
)
```

(If you're not familiar with Litte)

Avoid using the pipe when:

- You need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- There are meaningful intermediate objects that could be given informative names.

There's one exception to this rule: sometimes it's useful to include a short pipe as an argument to a function in a longer pipe. Carefully consider whether the code is more readable with a short inline pipe (which doesn't require a lookup elsewhere) or if it's better to move the code outside the pipe and give it an evocative name.

```
# Good
x %>%
  select(a, b, w) %>%
  left_join(y %>% select(a, b, v), by = c("a", "b"))

x_join <-
  x %>%
  select(a, b, w)
y_join <-
  y %>%
  filter(!u) %>%
  gather(a, v, -b) %>%
  select(a, b, v)
left_join(x_join, y_join, by = c("a", "b"))

# Bad
x %>%
  select(a, b, w) %>%
  left_join(y %>% filter(!u) %>% gather(a, v, -b) %>% select(a, b, v), by = c("a", "b"))
```

4.2 Spacing and indenting

%>% should always have a space before it and a new line after it. After the first step, each line should be indented by two spaces.

```
# Good
iris %>%
  group_by(Species) %>%
  summarize_if(is.numeric, mean) %>%
  ungroup() %>%
  gather(measure, value, -Species) %>%
  arrange(value)

# Bad
iris %>% group_by(Species) %>% summarize_all(mean) %>%
ungroup %>% gather(measure, value, -Species) %>%
arrange(value)
```

4.3 No arguments

magrittr allows you to omit `()` on functions that don't have arguments. Avoid this.

```
# Good
x %>%
  unique() %>%
  sort()
```

```
# Bad
x %>%
  unique %>%
  sort
```

4.4 Long lines

If the arguments to a function don't all fit on one line, put each argument on its own line and indent:

```
iris %>%
  group_by(Species) %>%
  summarise(
    Sepal.Length = mean(Sepal.Length),
    Sepal.Width = mean(Sepal.Width),
    Species = n_distinct(Species)
  )
```

4.5 Assignment

Use a separate line for the target of the assignment followed by <-.

Personally, I think you should avoid using -> to create an object at the end of the pipe. While starting with the assignment is a little more work when writing the code, it makes reading the code easier. This is because the name acts as a heading, which reminds you of the purpose of the pipe.

```
# Good
iris_long <-
  iris %>%
  gather(measure, value, -Species) %>%
  arrange(-value)

# Bad
iris_long <- iris %>%
  gather(measure, value, -Species) %>%
  arrange(-value)

iris %>%
  gather(measure, value, -Species) %>%
  arrange(-value) ->
  iris_long
```

4.6 Comments

In data analysis code, use comments to record important findings and analysis decisions. If you need comments to explain what your code is doing, consider rewriting your code to be clearer. If you discover that you have more comments than code, considering switching to RMarkdown.

Part II

Packages

Chapter 5

Code Documentation

5.1 Introduction

Documentation of code is essential, even if the only person using your code is future-you. Use roxygen2 with enabled markdown support to keep your documentation close to the code.

5.2 Title and description

For the title, describe concisely what the function does in the very first line of your function documentation. Titles should use sentence case but not end with a full stop.

```
#' Combine values into a vector or list
#'
#' This is a generic function which combines its arguments.
#'
```

There is no need to use the explicit `@title` or `@description` tags, except in the case of the description if it is multiple paragraphs or includes more complex formatting like a bulleted list.

```
#' Apply a function to each element of a vector
#'
#' @description
#' The map function transform the input, returning a vector the same length
#' as the input. `map()` returns a list or a data frame; `map_lgl()`,
#' `map_int()`, `map_dbl()` and `map_chr()` return vectors of the
#' corresponding type (or die trying); `map_dfr()` and `map_dfc()` return
#' data frames created by row-binding and column-binding respectively.
#' They require dplyr to be installed.
```

5.3 Indention

Always indent with one space after `#'`. If any description corresponding to a `roxygen` tag spans over multiple lines, add another two spaces of extra indention.

```
#' @param key The bare (unquoted) name of the column whose values will be used
#'   as column headings.
```

Alternatively, tags that span over multiple lines (like `@examples` and `@section`) can have the corresponding tag on its own line and then subsequent lines don't need extra indention.

```
#' @examples
#' 1 + 1
#' sin(pi)
```

The section 'Description' does not need extra indention either.

5.4 Documenting parameters

For most tags, like `@param`, `@seealso` and `@return`, the text should be a sentence, starting with a capital letter and ending with a full stop.

```
#' @param key The bare (unquoted) name of the column whose values will be used
#'   as column headings.
```

If some functions share parameters, you can use `@inheritParams` to avoid duplication of content in multiple places.

```
#' @inheritParams argument function_to_inherit_from
```

5.5 Capitalization and full stops

For all bullets, enumerations, argument descriptions and the like, use sentence case and put a period at the end of each text element, even if it is only a few words. However, avoid capitalization of function names or packages since R is case sensitive. Use a colon before enumerations or bulleted lists.

```
#' @details In the following, we present the bullets of the list:
#' * Four cats are few animals.
#' * forcats is a package.
```

5.6 Reference

Cross-referencing is encouraged, both within R's help file system as well as to external resources. Include parentheses after function names when referencing function calls.

```
#' @seealso [fct_lump()]
```

If you have a family of related functions, you can use the `@family` tag to automatically add appropriate lists and interlinks to the `@seealso` section. Family names are plural. In dplyr, the verbs `arrange`, `filter`, `mutate`, `slice`, `summarize` form the family of single table verbs.

```
#' @family single table verbs
```

5.7 Line break

Leave one line blank before / after each description.

```
#' @section Tidy data:
#' When applied to a data frame, row names are silently dropped. To preserve,
#' convert to an explicit variable with [tibble::rownames_to_column()].
#'
#' @section Scoped filtering:
#' The three [scoped] variants ([filter_all()], [filter_if()] and
#' [filter_at()]) make it easy to apply a filtering condition to a
#' selection of variables.
```

If you want to insert a line break within a section, also leave a line break in the roxygen comments.

```
#' @param ... Data frames to combine.
#'
#'   Each argument can either be a data frame, a list that could be a data
#'   frame, or a list of data frames.
#'
#'   When row-binding, columns are matched by name, and any missing
#'   columns with be filled with NA.
#'
#'   When column-binding, rows are matched by position, so all data
#'   frames must have the same number of rows. To match by value, not
#'   position, see [join()].
```

5.8 Code font

Text that contains valid R code should be marked as such.

- Function names, which should be followed by `()`, e.g. `tibble()`.
- Function arguments, e.g. `na.rm`.
- Values, e.g. `TRUE`, `FALSE`, `NA`, `NaN`, `...`, `NULL`
- Literal R code, e.g. `mean(x, na.rm = TRUE)`

Do not use code font for package names. If package name is ambiguous in the context, disambiguate with words, e.g. “the foo package”.

5.9 Internal functions

Internal functions should be documented with `#'` comments as per usual. Use the `@noRd` tag to prevent `.Rd` files from being generated.

```
#' Drop last
#'
#' Drops the last element from a vector.
#'
#' @param x A vector object to be trimmed.
#'
#' @noRd
```

Chapter 6

Error messages

An error message should start with a general statement of the problem then give a concise description of what went wrong. Consistent use of punctuation and formatting makes errors easier to parse.

(This guide is currently almost entirely aspirational; most of the bad examples come from existing tidyverse code.)

6.1 Problem statement

Every error message should start with a general statement of the problem. It should be concise, but informative. (This is hard!)

- If the cause of the problem is clear use "must":

```
dplyr::nth(1:10, "x")
#> Error: `n` must be a numeric vector, not a character vector

dplyr::nth(1:10, 1:2)
#> Error: `n` must have length 1, not length 2
```

  Clear cut causes typically involve incorrect types or lengths.

- If you can't state what was expected, use "can't":

```
mtcars %>% pull(b)
#> Error: Can't find column `b` in `.data`

as_vector(environment())
#> Error: Can't coerce `.x` to a vector

purrr::modify_depth(list(list(x = 1)), 3, ~ . + 1)
#> Error: Can't find specified `.depth` in `.x`
```

Use `stop(call. = FALSE)`, `rlang::abort()`, `Rf_errorcall(R_NilValue, ...)` to avoid cluttering the error message with the name of the function that generated it. That information is often not informative, and can easily be accessed via `traceback()` or IDE equivalent.

6.2 Error location

Do your best to reveal the location, name, and/or content of the troublesome component. The goal is to make it easy as possible for the user to find and fix the problem.

```
# GOOD
map_int(1:5, ~ "x")
#> Error: Each result must be a single integer:
#> * Result 1 is a character vector

# BAD
map_int(1:5, ~ "x")
#> Error: Each result must be a single integer
```

(It is often not easy to identify the exact problem; it may require passing around extra arguments so that error messages generated at a lower-level can know the original source. For frequently used functions, the effort is typically worth it.)

If the source of the error is unclear, avoid pointing the user in the wrong direction by giving an opinion about the source of the error:

```
# GOOD
pull(mtcars, b)
#> Error: Can't find column `b` in `.data`

tibble(x = 1:2, y = 1:3, z = 1)
#> Error: Columns must have consistent lengths:
#> * Column `x` has length 2
#> * Column `y` has length 3

# BAD: implies one argument at fault
pull(mtcars, b)
#> Error: Column `b` must exist in `.data`

pull(mtcars, b)
#> Error: `.data` must contain column `b`

tibble(x = 1:2, y = 1:3, z = 1)
#> Error: Column `x` must be length 1 or 3, not 2
```

If there are multiple issues, or an inconsistency revealed across several arguments or items, prefer a bulleted list:

```
# GOOD
purrr::reduce2(1:4, 1:2, `+`)
#> Error: `.x` and `.y` must have compatible lengths:
#> * `.x` has length 4
#> * `.y` has length 2

# BAD: harder to scan
purrr::reduce2(1:4, 1:2, `+`)
#> Error: `.x` and `.y` must have compatible lengths: `.x` has length 4 and
#> `.y` has length 2
```

6.3 Hints

If the source of the error is clear and common, you may want provide a hint as how to fix it:

```
dplyr::filter(iris, Species = "setosa")
#> Error: Filter specifications must be named
#> Did you mean `Species == "setosa"`?

ggplot2::ggplot(ggplot2::aes())
#> Error: Can't plot data with class "uneval".
#> Did you accidentally provide the results of aes() to the `data` argument?
```

Hints should always end in a question mark.

Hints are particularly important if the source of the error is far away from the root cause:

```
# BAD
mean[[1]]
#> Error in mean[[1]] : object of type 'closure' is not subsettable

# BETTER
mean[[1]]
#> Error: Can't subset a function.

# BEST
mean[[1]]
#> Error: Can't subset a function
#> Have you forgotten to define a variable named `mean`?
```

Good hints are difficult to write because, as above, you want to avoid steering users in the wrong direction. Generally, I avoid writing a hint unless the problem is common, and you can easily find a common pattern of incorrect usage (e.g. by searching StackOverflow).

6.4 Punctuation

- Errors should be written in sentence case, and should end in a full stop. Bullets should be formatted similarly; make sure to capitalise the first word (unless it's an argument or column name).
- Prefer the singular in problem statements:

  ```
  # GOOD
  map_int(1:2, ~ "a")
  #> Error: Each result must be coercible to a single integer:
  #> * Result 1 is a character vector

  # BAD
  map_int(1:2, ~ "a")
  #> Error: Results must be coercible to single integers:
  #> * Result 1 is a character vector
  ```

- If you can detect multiple problems, list up to five. This allows the user to fix multiple problems in a single pass without being overwhelmed by many errors that may have the same source.

  ```
  # BETTER
  map_int(1:10, ~ "a")
  #> Error: Each result must be coercible to a single integer:
  ```

```
#> * Result 1 is a character vector
#> * Result 2 is a character vector
#> * Result 3 is a character vector
#> * Result 4 is a character vector
#> * Result 5 is a character vector
#> * ... and 5 more problems
```

- Pick a natural connector between problem statement and error location: this may be ", not", ";", or ":" depending on the context.
- Surround the names of arguments in backticks, e.g. `` `x` ``. Use "column" to disambiguiate columns and arguments: `` Column `x` ``. Avoid "variable", because it is ambiguous.
- Ideally, each component of the error message should be less than 80 characters wide. Do not add manual line breaks to long error messages; they will not look correct if the console is narrower (or much wider) than expected. Instead, use bullets to break up the error into shorter logical components.

6.5 Before and after

More examples gathered from around the tidyverse.

```
dplyr::filter(mtcars, cyl)
#> BEFORE: Argument 2 filter condition does not evaluate to a logical vector
#> AFTER:  Each argument must be a logical vector:
#> * Argument 2 (`cyl`) is an integer vector

tibble::tribble("x", "y")
#> BEFORE: Expected at least one column name; e.g. `~name`
#> AFTER:  Must supply at least one column name, e.g. `~name`

ggplot2::ggplot(data = diamonds) + ggplot2::geom_line(ggplot2::aes(x = cut))
#> BEFORE: geom_line requires the following missing aesthetics: y
#> AFTER:  `geom_line()` must have the following aesthetics: `y`

dplyr::rename(mtcars, cyl = xxx)
#> BEFORE: `xxx` contains unknown variables
#> AFTER:  Can't find column `xxx` in `.data`

dplyr::arrange(mtcars, xxx)
#> BEFORE: Evaluation error: object 'xxx' not found.
#> AFTER:  Can't find column `xxx` in `.data`
```

Chapter 7

News

7.1 During development

Each user-facing change should be accompanied by a bullet in `NEWS.md`. The goal of the bullet is to:

- Briefly describe the change, starting with the name of the function. This can be similar to the commit message, but often the commit message will be developer facing, and the bullet will be user facing.
- Link to the related issue, if present.
- Credit the author, if the contribution was a PR.

```
* `drop_na()` no longer drops columns (@jennybryan, #245), and works with
  list-cols (#280). Equivalent of `NA` in a list column is any empty
  (length 0) data structure.
```

Each new bullet should be added to the top of the file (under the version heading).

It is not necessary to describe minor documentation changes.

7.2 Before release

Prior to release, the NEWS file needs to be thoroughly proofread and groomed. If there are many bullets, they should be grouped into related areas using level 2 headings (`##`). Typically, this will include a catch-all "Minor improvements and bug fixes".

If there are API breaking changes, these should appear at the top, including a description of the symptoms of the change, and what is needed to fix it.

```
## Breaking changes

* `separate()` now correctly uses -1 to refer to the far right position,
  instead of -2. If you depended on this behaviour, you'll need to condition
  on `packageVersion("tidyr") > "0.7.2"`
```

Within a section, bullets should be ordered alphabetically by the first function mentioned (which should be near the start of the sentence)