The tidyverse style guide

Hadley Wickham

Contents

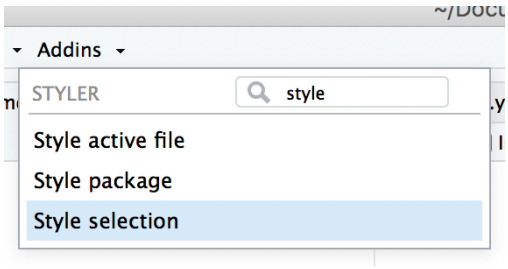
Welcome

Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. This site describes the style used throughout the tidyverse. It was derived from Google's original R Style Guide - but Google's current guide is derived from the tidyverse style guide.

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 you 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 Organisation

It's hard to describe exactly how you should organise your code across multiple files. I think the best rule of thumb is that if you can give a file a concise name that still evokes its contents, you've arrived at a good organisation. But getting to that point is hard.

1.3 Internal 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 (`_`) (so called snake case) to separate words within a name.

```
# Good
day_one
day_1

# Bad
DayOne
dayone
```

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()`.

If you find yourself attempting to cram data into variable names (e.g. `model_2018`, `model_2019`, `model_2020`), consider using a list or data frame instead.

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

# Bad
first_day_of_the_month
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

2.2.1 Commas

Always put a space after a comma, never before, just like in regular English.

```
# Good
x[, 1]

# Bad
x[,1]
x[ ,1]
x[ , 1]
```

2.2.2 Parentheses

Do not put spaces inside or outside parentheses for regular function calls.

```
# Good
mean(x, na.rm = TRUE)

# Bad
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
```

Place a space before and after `()` when used with `if`, `for`, or `while`.

```r
# Good
if (debug) {
  show(x)
}

# Bad
if(debug){
  show(x)
}
```

Place a space after `()` used for function arguments:

```r
# Good
function(x) {}

# Bad
function (x) {}
function(x){}
```

2.2.3 Embracing

The embracing operator, `{{ }}`, should always have inner spaces to help emphasise its special behaviour:

```r
# Good
max_by <- function(data, var, by) {
  data %>%
    group_by({{ by }}) %>%
    summarise(maximum = max({{ var }}, na.rm = TRUE))
}

# Bad
max_by <- function(data, var, by) {
  data %>%
    group_by({{by}}) %>%
    summarise(maximum = max({{var}}, na.rm = TRUE))
}
```

2.2.4 Infix operators

Most infix operators (`==`, `+`, `-`, `<-`, etc.) should always be surrounded by spaces:

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-feet*12+inches
mean(x, na.rm=TRUE)
```

There are a few exceptions, which should never be surrounded by spaces:

- The operators with high precedence: `::`, `:::`, `$`, `@`, `[`, `[[`, `^`, unary `-`, unary `+`, and `:`.

  ```
  # Good
  sqrt(x^2 + y^2)
  df$z
  x <- 1:10

  # Bad
  sqrt(x ^ 2 + y ^ 2)
  df $ z
  x <- 1 : 10
  ```

- Single-sided formulas when the right-hand side is a single identifier:

  ```
  # Good
  ~foo
  tribble(
    ~col1, ~col2,
    "a",   "b"
  )

  # Bad
  ~ foo
  tribble(
    ~ col1, ~ col2,
    "a", "b"
  )
  ```

  Note that single-sided formulas with a complex right-hand side do need a space:

  ```
  # Good
  ~ .x + .y
  ```

```
# Bad
~.x + .y
```

- When used in tidy evaluation `!!` (bang-bang) and `!!!` (bang-bang-bang) (because have precedence equivalent to unary `-`/`+`)

```
# Good
call(!!xyz)

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

- The help operator

```
# Good
package?stats
?mean

# Bad
package ? stats
? mean
```

2.2.5 Extra spaces

Adding extra spaces is ok if it improves alignment of `=` or `<-`.

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

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

Do not add extra spaces to places where space is not usually allowed.

2.3 Function calls

2.3.1 Named arguments

A function's arguments typically fall into two broad categories: one supplies the **data** to compute on; the other controls the **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.

2.3.2 Assignment

Avoid assignment in function calls:

```
# Good
x <- complicated_function()
if (nzchar(x) < 1) {
  # do something
}

# Bad
if (nzchar(x <- complicated_function()) < 1) {
  # do something
}
```

The only exception is in functions that capture side-effects:

```
output <- capture.output(x <- f())
```

2.4 Control flow

2.4.1 Code blocks

Curly braces, `{}`, define the most important hierarchy of R code. To make this hierarchy easy to see:

- `{` should be the last character on the line. Related code (e.g., an `if` clause, a function declaration, a trailing comma, …) must be on the same line as the opening brace.
- The contents should be indented by two spaces.
- `}` should be the first character on the 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 }
```

2.4.2 If statements

- If used, `else` should be on the same line as `}`.
- `&` and `|` should never be used inside of an `if` clause because they can return vectors. Always use `&&` and `||` instead.
- NB: `ifelse(x, a, b)` is not a drop-in replacement for `if (x) a else b`. `ifelse()` is vectorised (i.e. if `length(x) > 1`, then `a` and `b` will be recycled to match) and it is eager (i.e. both `a` and `b` will always be evaluated).

  If you want to rewrite a simple but lengthy `if` block:

  ```
  if (x > 10) {
    message <- "big"
  } else {
    message <- "small"
  }
  ```

  Just write it all on one line:

  ```
  message <- if (x > 10) "big" else "small"
  ```

2.4.3 Inline statements

It's ok to drop the curly braces for very simple statements that fit on one line, as long as they don't have side-effects.

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

Function calls that affect control flow (like `return()`, `stop()` or `continue`) should always go in their own `{}` block:

```r
# Good
if (y < 0) {
  stop("Y is negative")
}

find_abs <- function(x) {
  if (x > 0) {
    return(x)
  }
  x * -1
}

# Bad
if (y < 0) stop("Y is negative")

if (y < 0)
  stop("Y is negative")

find_abs <- function(x) {
  if (x > 0) return(x)
  x * -1
}
```

2.4.4 Implicit type coercion

Avoid implicit type coercion (e.g. from numeric to logical) in `if` statements:

```r
# Good
if (length(x) > 0) {
  # do something
}

# Bad
if (length(x)) {
  # do something
}
```

2.4.5 Switch statements

- Avoid position-based `switch()` statements (i.e. prefer names).
- Each element should go on its own line.
- Elements that fall through to the following element should have a space after `=`.

- Provide a fall-through error, unless you have previously validated the input.

```
# Good
switch(x,
  a = ,
  b = 1,
  c = 2,
  stop("Unknown `x`", call. = FALSE)
)

# Bad
switch(x, a = , b = 1, c = 2)
switch(x, a =, b = 1, c = 2)
switch(y, 1, 2, 3)
```

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 Named arguments, 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 Semicolons

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

2.7 Assignment

Use `<-`, not `=`, for assignment.

```
# Good
x <- 5

# Bad
x = 5
```

2.8 Data

2.8.1 Character vectors

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'
```

2.8.2 Logical vectors

Prefer `TRUE` and `FALSE` over `T` and `F`.

2.9 Comments

Each line of a comment should begin with the comment symbol and a single space: `#`

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, consider switching to R Markdown.

Chapter 3

Functions

3.1 Naming

As well as following the general advice for object names, strive to use verbs for function names:

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()
```

3.2 Long lines

There are two options if the function name and definition can't fit on a single line:

- Function-indent: place each argument on its own line, and indent to match the opening `(` of `function`:

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

- Double-indent: Place each argument of its own **double** indented line.

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

In both cases the trailing `)` and leading `{` should go on the same line as the last argument.

Prefer function-indent style to double-indent style when it fits.

These styles are designed to clearly separate the function definition from its body.

```
# 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, and to see all three function arguments
}
```

If a function argument can't fit on a single line, this is a sign you should rework the argument to keep it short and sweet.

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

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

Return statements should always be on their own line because they have important effects on the control flow. See also inline statements.

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

# Bad
find_abs <- function(x) {
  if (x > 0) return(x)
  x * -1
}
```

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: `#`.

```
# Good

# Objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)
```

```
# Bad

# Recurse only with bare lists
x <- map_if(x, is_bare_list, recurse)
```

Comments should be in sentence case, and only end with a full stop if they contain at least two sentences:

```
# Good

# Objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)

# Do not use `is.list()`. Objects like data frames must be treated
# as leaves.
x <- map_if(x, is_bare_list, recurse)


# Bad

# objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)

# Objects like data frames are treated as leaves.
x <- map_if(x, is_bare_list, recurse)
```

Chapter 4

Pipes

4.1 Introduction

Use `%>%` to emphasise a sequence of actions, rather than the object that the actions are being performed on.

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.

4.2 Whitespace

`%>%` should always have a space before it, and should usually be followed by a new line. After the first step, each line should be indented by two spaces. This structure makes it easier to add new steps (or rearrange existing steps) and harder to overlook a step.

```
# 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 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)
  )
```

4.4 Short pipes

A one-step pipe can stay on one line, but unless you plan to expand it later on, you should consider rewriting it to a regular function call.

```
# Good
iris %>% arrange(Species)

iris %>%
  arrange(Species)

arrange(iris, Species)
```

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"))
```

```
# Better
x_join <- x %>% select(a, b, w)
y_join <- y %>% select(a, b, v)
left_join(x_join, y_join, by = c("a", "b"))
```

4.5 No arguments

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

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

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

4.6 Assignment

There are three acceptable forms of assignment:

- Variable name and assignment on separate lines:

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

- Variable name and assignment on the same line:

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

- Assignment at the end of the pipe with `->`:

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

I think this is the most natural to write, but makes reading a little harder: when the name comes first, it can act as a heading to remind you of the purpose of the pipe.

The magrittr package provides the `%<>%` operator as a shortcut for modifying an object in place. Avoid this operator.

```
# Good
x <- x %>%
  abs() %>%
  sort()

# Bad
x %<>%
  abs() %>%
  sort()
```

Chapter 5

ggplot2

5.1 Introduction

Styling suggestions for + used to separate ggplot2 layers are very similar to those for %>% in pipelines.

5.2 Whitespace

+ should always have a space before it, and should be followed by a new line. This is true even if your plot has only two layers. After the first step, each line should be indented by two spaces.

If you are creating a ggplot off of a dplyr pipeline, there should only be one level of indentation.

```
# Good
iris %>%
  filter(Species == "setosa") %>%
  ggplot(aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point()

# Bad
iris %>%
  filter(Species == "setosa") %>%
  ggplot(aes(x = Sepal.Width, y = Sepal.Length)) +
    geom_point()

# Bad
```

```
iris %>%
  filter(Species == "setosa") %>%
  ggplot(aes(x = Sepal.Width, y = Sepal.Length)) + geom_point()
```

5.3 Long lines

If the arguments to a ggplot2 layer don't all fit on one line, put each argument on its own line and indent:

```
# Good
ggplot(aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point() +
  labs(
    x = "Sepal width, in cm",
    y = "Sepal length, in cm",
    title = "Sepal length vs. width of irises"
  )

# Bad
ggplot(aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point() +
  labs(x = "Sepal width, in cm", y = "Sepal length, in cm", title = "Sepal length vs. w
```

ggplot2 allows you to do data manipulation, such as filtering or slicing, within the `data` argument. Avoid this, and instead do the data manipulation in a pipeline before starting plotting.

```
# Good
iris %>%
  filter(Species == "setosa") %>%
  ggplot(aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point()

# Bad
ggplot(filter(iris, Species == "setosa"), aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point()
```

Part II

Packages

Chapter 6

Files

The majority of advice in Chapter 1 also applies to files in packages. Important differences are described below.

6.1 Names

- If a file contains a single function, give the file the same name as the function.
- If a file contains multiple related functions, give it a concise, but evocative name.
- Deprecated functions should live in a file with `deprec-` prefix.
- Compatibility functions should live in a file with `compat-` prefix.

6.2 Organisation

In a file that contains multiple functions, public functions and their documentation should appear first, with private functions appearing after all documented functions. If multiple public functions share the same documentation, they should all immediately follow the documentation block.

See 7 for more thorough guidance on documenting functions in packages.

```
# Bad
help_compute <- function() {
  # ... Lots of code ...
}
```

```
#' My public function
#'
#' This is where the documentation of my function begins.
#' ...
#' @export
do_something_cool <- function() {
  # ... even more code ...
  help_compute()
}
```

```
# Good
#' Lots of functions for doing something cool
#'
#' ... Complete documentation ...
#' @name something-cool
NULL
```

```
## NULL
```

```
#' @describeIn something-cool Get the mean
#' @export
get_cool_mean <- function(x) {
  # ...
}

#' @describeIn something-cool Get the sum
#' @export
get_cool_sum <- function(x) {
  # ...
}
```

Chapter 7

Documentation

7.1 Introduction

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

7.2 Title and description

Use the first line of your function documentation to provide a concise title that describes the function, dataset, or class. 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.
```

7.3 Indents and line breaks

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 `@description`, `@examples` and `@section`) can have the corresponding tag on its own line and then subsequent lines don't need to be indented.

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

Use line breaks before/after sections where needed:

```
#' @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.
```

7.4 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 function_to_inherit_from
```

7.5 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.
```

7.6 Cross-linking

Cross-referencing is encouraged, both within R's help file system as well as to external resources.

List closely related functions in `@seealso`. A single related function can be written as a sentence:

```
#' @seealso [fct_lump()] to automatically convert the rarest (or most common)
#'   levels to "other".
```

More recommendations should be organised in a bulleted list:

```
#' @seealso
#' * [tibble()] constructs from individual columns.
#' * [enframe()] converts a named vector into a two-column tibble (names and
#'   values).
#' * [name-repair] documents the details of name repair.
```

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
```

When linking to external resources either include the full url inline with `<>`, or the surrounding prose and link text should make it extremely clear where the hyperlink goes. Avoid text like "click here".

7.7 R code

Text that contains valid R code should be marked as such using backticks. This includes:

- 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)`
- Class names, e.g. "a tibble will have class `tbl_df` …"

Do not use code font for package names. If the package name is ambiguous in the context, disambiguate with words, e.g. "the foo package". Do not capitalize the function name if it occurs at the start of a sentence.

7.8 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 8

Tests

8.1 Organisation

The organisation of test files should match the organisation of `R/` files: if a function lives in `R/foofy.R`, then its tests should live in `tests/testthat/test-foofy.R`.

Use `usethis::use_test()` to automatically create a file with the correct name.

The file name will be displayed in output in order to get context.

Chapter 9

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.)

9.1 Problem statement

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

It is encouraged to be as informative as possible, but each sentence should be very simple to make localisation and translation possible. A Localization Horror Story: It Could Happen To You is a Good summary of the challenges of localising error messages. You might not support localised messages right now but you should make it as easy as possible to do it in the future.

Ideally each sentence should contain a single phrase, and should only mention one variable quantity. Because we avoid complex sentences, we prefer to lay out information in a bullet list. Start with a list of *contextual* information, and finish with a list of information about *faulty* user input. These lists should be prefixed with and respectively if UTF-8 is available (and in blue and red if colour is available), or the ASCII * character otherwise.

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

```
dplyr::nth(1:10, "x")
#> Error: `n` must be a numeric vector:
```

```
#>   You've supplied a character vector.

dplyr::nth(1:10, 1:2)
#> Error: `n` must have length 1
#>   You've supplied a vector of length 2.
```

Clear cut causes typically involve incorrect types or lengths.

- If you cannot 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`.
```

The problem statement should use sentence case and end with a full stop.

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 an IDE equivalent.

Use simple sentences layed out in a bullet list of and elements:

- The sentences should be short and bulleted:

```
# Good
vec_slice(letters, 100)
#> Must index an existing element:
#>   There are 26 elements.
#>   You've tried to subset element 100.

# Bad
vec_slice(letters, 100)
#> Must index an existing element.
#> There are 26 elements and you've tried to subset element 100.
```

- The contextual information should be first:

```
# Good
vec_slice(letters, 100)
#> Must index an existing element:
#>   There are 26 elements.
#>   You've tried to subset element 100.

# Bad
vec_slice(letters, 100)
#> Must index an existing element:
#>   You've tried to subset element 100.
#>   There are 26 elements.
```

9.2 Error location

Do your best to reveal the location, name, and/or content of the troublesome component. The goal is to make it as 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
```

If the list of issues might be long, make sure to truncate to only show the first few:

```
# Good
#> Error: NAs found at 1,000,000 locations: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

If you want to correctly pluralise the error message, consider using `ngettext()`. See the notes in `?ngettext()` for some challenges related to correct translation to other languages.

9.3 Hints

If the source of the error is clear and common, you may want to provide a hint as to how to fix it. If UTF-8 is available, prefix with (in blue if colour is available):

```
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).

9.4 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 disambiguate 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.

9.5 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 10

News

Each user-facing change to a package should be accompanied by a bullet in `NEWS.md`. Minor changes to documentation don't need to be documented, but it's worthwhile to draw attention to sweeping changes and to new vignettes.

10.1 Bullets

The goal of the bullet is to briefly describe the change so users of the packages can understand what's changed. This can be similar to the commit message, but written with a user (not developer) in mind.

New bullets should be added to the top of the file (under the first heading). Organisation of bullets will happen later, during the release process (Section 10.2.2).

10.1.1 General style

Strive to place the name of the function as close to the beginning of the bullet as possible. A consistent location makes the bullets easier to scan, and easier to organise prior to release.

```
# Good
* `ggsave()` now uses full argument names to avoid partial match warning (#2355).

# Bad
* Fixed partial argument matches in `ggsave()` (#2355).
```

Lines should be wrapped to 80 characters, and each bullet should end in a full stop.

Frame bullets positively (i.e. what now happens, not what used to happen), and use the present tense.

````
# Good
* `ggsave()` now uses full argument names to avoid partial match warnings (#2355).

# Bad
* `ggsave()` no longer partially matches argument names (#2355).
````

Many news bullets will be a single sentence. This is typically adequate when describing a bug fix or minor improvement, but you may need more detail when describing a new feature. For more complex features, include longer examples in fenced code blocks (```). These will be useful inspiration when you later write the blog post.

````
# Good
* In `stat_bin()`, `binwidth` now also takes functions.

# Better
* In `stat_bin()`, `binwidth` now also takes functions. The function is
  called with the scaled `x` values, and should return a single number.
  This makes it possible to use classical binwidth computations with ggplot2.

# Best
* In `stat_bin()`, `binwidth` now also takes functions. The function is
  called with the scaled `x` values, and should return a single number.
  With a little work, this makes it possible to use classical bin size
  computations with ggplot2.

  ```R
  sturges <- function(x) {
    rng <- range(x)
    bins <- nclass.Sturges(x)

    (rng[2] - rng[1]) / bins
  }
  ggplot(diamonds, aes(price)) +
    geom_histogram(binwidth = sturges) +
    facet_wrap(~cut)
  ```
````

10.1.2 Acknowledgement

If the bullet is related to an issue, include the issue number. If the contribution was a PR, and the author is not a package author, include their GitHub user

name. Both items should be wrapped in parentheses and will generally come before the final period.

```
# Good
* `ggsave()` now uses full argument names to avoid partial match warnings
  (@wch, #2355).

# Bad
* `ggsave()` now uses full argument names to avoid partial match warnings.

* `ggsave()` now uses full argument names to avoid partial match warnings.
  (@wch, #2355)
```

10.1.3 Code style

Functions, arguments, and file names should be wrapped in backticks. Function names should include parentheses; omit "the argument" or "the function"

```
# Good
* In `stat_bin()`, `binwidth` now also takes functions.

# Bad
* In the stat_bin function, "binwidth" now also takes functions.
```

10.1.4 Common patterns

The following excerpts from tidyverse news entries provide helpful templates to follow.

- New family of functions:

  ```
  * Support for ordered factors is improved. Ordered factors throw a warning
    when mapped to shape (unordered factors do not), and do not throw warnings
    when mapped to size or alpha (unordered factors do). Viridis is used as
    default colour and fill scale for ordered factors (@karawoo, #1526).

  * `possibly()`, `safely()` and friends no longer capture interrupts: this
    means that you can now terminate a mapper using one of these with
    Escape or Ctrl + C (#314).
  ```

- New function:

```
  * New `position_dodge2()` provides enhanced dogding for boxplots...

  * New `stat_qq_line()` makes it easy to add a simple line to a Q-Q plot.
    This line makes it easier to judge the fit of the theoretical distribution
    (@nicksolomon).
```

- New argument to existing function:

```
  * `geom_segment()` gains a `linejoin` parameter.
```

- Function argument changes behaviour:

```
  * In `separate()`, `col = -1` now refers to the far right position.
    Previously, and incorrectly, `col = -2` referred to the far-right
    position.
```

- Function changes behaviour:

```
  * `map()` and `modify()` now work with calls and pairlists (#412).

  * `flatten_dfr()` and `flatten_dfc()` now aborts with informative
     message if dplyr is not installed (#454).

  * `reduce()` now throws an error if `.x` is empty and `.init` is not
    supplied.
```

10.2 Organisation

10.2.1 Development

During development, new bullets should be added to the top of the file, immediately under a "development" heading:

```
# haven (development version)

* Second update.

* First update.
```

10.2.2 Release

Prior to release, the NEWS file needs to be thoroughly proofread and groomed.

Each release should have a level 1 heading (`#`) containing the package name and version number. For smaller packages or patch releases this amount of organisation may be sufficient. For example, here is the NEWS for modelr 0.1.2:

```
# modelr 0.1.2

* `data_grid()` no longer fails with modern tidyr (#58).

* New `mape()` and `rsae()` model quality statistics (@paulponcet, #33).

* `rsquare()` use more robust calculation 1 - SS_res / SS_tot rather
  than SS_reg / SS_tot (#37).

* `typical()` gains `ordered` and `integer` methods (@jrnold, #44),
  and `...` argument (@jrnold, #42).
```

If there are many bullets, the version heading should be followed by issues grouped into related areas with level 2 headings (`##`). Three commonly used sections are shown below:

```
# package 1.1.0

## Breaking changes

## New features

## Minor improvements and fixes
```

It is fine to deviate from these headings if another organisation makes sense. Indeed, larger packages will often require a finer break down. For example, ggplot2 2.3.0 included these headings:

```
# ggplot 2.3.0
## Breaking changes
## New features
### Tidy evaluation
### sf
### Layers: geoms, stats, and position adjustments
### Scales and guides
### Margins
```

```
## Extension points
## Minor bug fixes and improvements
### Facetting
### Scales
### Layers
### Coords
### Themes
### Guides
### Other
```

It is not worthwhile to organise bullets into headings during development, as it's not typically obvious what the groups will be in advance.

Within a section, bullets should be ordered alphabetically by the first function mentioned. If no function is mentioned, place the bullet at the top of the section.

10.2.3 Breaking changes

If there are API breaking changes (as discovered during revdepchecks) these should also appear in their own section at the top. Each bullet should include a description of the symptoms of the change, and what is needed to fix it. The bullet should also be repeated in the appropriate section.

```
## 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"`.
```

10.3 Blog post

For all major and minor releases, the latest news should be turned into a blog post. The blog post should highlight major user-facing changes, and point to the release notes for more details. Generally, you should focus on new features and major improvements, including examples showing the new features in action. You don't need to describe minor improvements and bug fixes, as the motivated reader can find these in the release notes.

Chapter 11

Git/GitHub

11.1 Commit messages

Follow standard git commit message advice. In brief:

- The first line is the subject, and should summarise the changes in the commit in under 50 characters. Use sentence case, but no period at the end.
- If additional details are required, add a blank line, and then provide explanation and context in paragraph format.
- If the commit fixes a GitHub issue include `Fixes #<issue-number>`. This will ensure that the issue is automatically closed when the commit is merged into master.

11.2 Pull requests

The title of a pull request should briefly describe the changes made. The title should be standalone and should not include the related issue number (i.e. don't write `Fixes #10`).

For very simple changes, you can leave the description blank as there's no need to describe what will be obvious from looking at the diff. For more complex changes, you should give an overview of the changes. If the PR fixes an issue, make sure to include `Fixes #<issue-number>` in the description.