Styleguide

Das ist ein Auszug aus dem tidyverse-Styleguide. Der R-Styleguide von Google referenziert inzwischen explizit auf jenen von tidyverse, wobei Google einige eigene Modifikationen vornimmt: beispielsweise verwendet Google die **BigCamelCase**- Schreibweise anstelle der **unterscore**-Schreibweise, um Funktionsnamen von Variablennamen zu unterscheiden. Der nachfolgende Auszug enthält nur die Richtlinien für Dateinamen, Syntax und Funktionen. Weitere Richtlinien von tidyverse konzentrieren sich einerseits auf den Pipe-Operator und **ggplot2** und andererseits auf Richtlinien hinsichtlich dem Erstellen von R-Paketen.

Files

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.

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.

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`.

Syntax

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)
```

Spacing

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]
```

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`.

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

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

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

Embracing

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

```
# 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))
}
```

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
  ```

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.

Function calls

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.

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())
```

Control flow

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 }
```

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"
```

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:

```
# 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
}
```

Implicit type coercion

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

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

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)
```

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")
```

Semicolons

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

Assignment

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

```
# Good
x <- 5
# Bad
x = 5
```

Data

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'
```

Logical vectors

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

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.

Functions

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()
```

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.

`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)
}
```

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)
```