The R Style Guide

Hadley Wickham and Garrett Grolemund

2017-03-07

Contents

Chapter 1

Style guide

Good coding style is like correct punctuation: you can manage without it, but it sure makes things easier to read. As with punctuation, there are many ways to style your code that you can choose from, but some ways are more reader-friendly than others. The following guide describes the style that I use. It is based on Google's R style guide, with a few tweaks. You don't have to use my style, but you really should use a consistent style.

Good style is important because while your code only has one author, it'll usually have multiple readers. This is especially true when you're writing code with others. In that case, it's a good idea to agree on a common style up-front. Since no style is strictly better than another, working with others may mean that you'll need to sacrifice some preferred aspects of your style.

The formatR package, by Yihui Xie, makes it easier to clean up poorly formatted code. It can't do everything, but it can quickly get your code from terrible to pretty good. Make sure to read the notes before using it.

Chapter 2

Notation and naming

2.1 File names

File names should be meaningful and end in `.R`.

```
# Good
fit_models.R
utility_functions.R

# Bad
foo.r
stuff.r
```

If files need to be run in sequence, prefix them with numbers:

```
0_download.R
1_parse.R
2_explore.R
```

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. Never use filenames that differ only in capitalization.

2.2 Object names

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

Variable and function names should be lowercase. Use an underscore (`_`) to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!).

Although standard R uses dots extensively in function names (`contrib.url()`), methods (`all.equal`), or class names (`data.frame`), it's better to use underscores. For example, the basic S3 scheme to define a method for a class, using a generic function, would be to concatenate them with a dot, like this `generic.class`. This can lead to confusing methods like `as.data.frame.data.frame()` whereas something like `print.my_class()` is unambiguous.

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

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

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

Chapter 3

Syntax

3.1 Spacing

Place spaces around all infix operators (`=`, `+`, `-`, `<-`, etc.). The same rule applies when using `=` in function calls. Always put a space after a comma, and never before (just like in regular English).

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

There's a small exception to this rule: `:`, `::` and `:::` don't need spaces around them.

```
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

Place a space before left parentheses, except in 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
```

3.2 Curly braces

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it's followed by `else` (or a closing parenthesis).

Always indent the code inside curly braces. When indenting your code, use two spaces. Never use tabs or mix tabs and spaces.

```
# 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
}

# 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 leave very short statements on the same line:

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

3.3 Line length

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.

3.4 Indentation

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") {
}
```

If a function call is too long, put the function name, each argument, and the closing parenthesis on a separate line. This makes the code easier to read and to change later. 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()`:

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

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

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

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

3.5 Assignment

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

```
# Good
x <- 5

# Bad
x = 5
```

3.6 Quotes

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

```
# Good
"Text"
```

```
## [1] "Text"
```

```
'Text with "quotes"'
```

```
## [1] "Text with \"quotes\""
```

```
# Bad
'Text'
```

```
## [1] "Text"
```

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

```
## [1] "Text with \"double\" and 'single' quotes"
```

Chapter 4

Functions

Use verbs for function names, where possible.

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()
```

Only use `return()` for early returns. Otherwise rely on R to return the result of the last row of a function.

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

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

Strive to keep blocks within a function on one screen, so around 20-30 lines maximum. Some even argue that if a *function* has 20 lines, it should be split into smaller functions.

Chapter 5

Pipes

Use the %>% operator from the tidyverse when you find yourself composing more than two functions together into a nested call, or when you find yourself creating unnecessaty interim objects to avoid nesting functions together.

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

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

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

When you use the %>% operator, 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
iris %>%
  group_by(Species) %>%
  summarize_all(mean) %>%
  ungroup %>%
  gather(measure, value, -Species) %>%
  arrange(value)

iris %>% arrange(Petal.Width)

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

```
  arrange(value)
```

5.1 When not to use the pipe

Do not use a pipe if:

- You need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- You have more than (say) ten steps in your pipe. In that case, divide your pipe into shorter pipes that create intermediate objects with meaningful names.

Chapter 6

Organisation

6.1 Commenting guidelines

Comment your code. Each line of a comment should begin with the comment symbol and a single space: `#`. Comments should explain the why, not the what of your code.

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

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

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