R Coding Style Guide

Anjali Silva, PhD

2022

Contents

Objective:

A brief introduction to good practices and coding style for maintainable R code.

Introduction:

Good coding practices are extremely important, but can be a volatile topic. This document is prepared for the purposes of R programming in Introduction to R course at the Data Sciences Institute, University of Toronto. This document adapts content from Tidyverse Style Guide (Wickham, H., 2021), Google's R Style Guide (Google, 2021) and R Coding Style Guide (Steipe, B., 2020). You may find some content covered in the above sources to slightly vary among each other. I recommend you to follow the rules outlined in this document. It is also important that you pick one style (e.g., when it comes to naming), and stick with it for the entire duration of the course. This document was prepared using R Markdown (Xie et al., 2018; Xie et al., 2020; Allaire et al., 2022).

Your goal should always be to code as clearly and explicitly as possible. One of the goals of setting a coding style is that it makes the code easy to read for people for whom R is not the first language, or even the language of choice. However, as Hadley Wickham puts it, "all style guides are fundamentally opinionated. Some decisions genuinely do make code easier to use, 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" (Wickham, H., 2021).

General

1. Comment your code.

Use comments to explain what you are doing when it is not immediately apparent and also to explain why you wrote it that way. It will help you when you look at your work months from now and can't remember anything. It will help you work with collaborators. It will also increase reproducibility if you release your code.

2. Use only <- for assignment, not =

```
# Good
totalValue <- x + y

priceTotal <- appPrice + priceIncrease

# Bad
totalValue = x + y

priceTotal = appPrice + priceIncrease
```

3. Use = when passing values into the arguments of functions.

```
# Good
averageValue <- mean(x = c(1:10),
                     trim = 0,
                     na.rm = TRUE)

# Bad
averageValue <- mean(x <- c(1:10),
                     trim <- 0,
                     na.rm <- TRUE)
```

4. Do not use right-hand assignment.

```
# Good
totalValue <- x + y

# Bad
x + y -> totalValue
```

5. Define global constants at the beginning of the code, use all caps names for such constants (e.g. MAXWIDTH).

6. Always use for (i in seq(along = x)) {...} rather than for (i in 1:length(x)) {...} because if x == NULL the loop is executed once, with an undefined variable.

```
# Good
xValue <- NULL
for (i in seq(along = xValue)) {
  cat("\n", xValue)
}
seq(along = xValue) # integer(0)

# Bad
for (i in 1:length(xValue)) {
  cat("\n", xValue)
}
1:length(xValue) # 1 0
```

7. Use the '< package >::< function >()' syntax to make it clear which function you mean. That's what package namespaces are for in the first place. More and more frequently, the functions in different packages have the same name.

```
install.packages("plyr")
library("plyr")
ls("package:plyr")

install.packages("Hmisc")
library("Hmisc")
ls("package:Hmisc")

# Both packages have a function called 'summarize'.
# To specify which package the function is coming from:
?plyr::summarize # for summarize function from plyr
?Hmisc::summarize # for summarize function from Hmisc
```

8. Don't use attach().

9. Don't use save() and load()

The built-in functions save() / load() saves one or more R objects and restores them to the same object name(s) they originally had. But there is no good way to know what that object name is. If you already have an object by that name in your global workspace, it will get overwritten.

The alternative is to use saveRDS() / readRDS(). saveRDS() serializes, compresses and saves a single R object, and readRDS() restores the object and returns it as a return value, thus it can be assigned:

```
# save object into a file named exampleFile.rds
saveRDS(object, file = "exampleFile.rds")

# save exampleFile.rds into an object called exampleObj
exampleObj <- readRDS("exampleFile.rds")
```

Layout

10. Limit yourself to 80 characters per line.

11. Don't write more than one expression on a line.

12. Don't repeat code. Use functions instead.

13. Whenever possible, make use of built-in functions. Do not write own functions when a built-in function is available for the same purpose.

```
sessionInfo()
# 7 base packages
# stats  graphics  grDevices  utils  datasets  methods  base

# To see built-in functions in R
ls("package:stats")
ls("package:graphics")
ls("package:grDevices")
ls("package:utils")
ls("package:datasets")
ls("package:methods")
ls("package:base")
```

14. Pay attention to collocation

One of the general principles of writing clear, maintainable code is collocation. This means that information items that can affect each other should be viewable on the same screen.

If the code for a function does not fit on approximately one printed page, you should probably break it up further.

If your loops or conditionals are nested more than three levels deep, you should rethink the logic.

Headers

15. Give your script files headers that state purpose, author, date and version information, and note bugs and issues.

```
# Purpose:
# Author:
# Date:
# Version:
# Bugs and Issues:
```

16. Give your functions headers that describe purpose, parameters (including required datatypes), and return values. Callers should be able to work with the function without having to read the code.

Sections

17. Use separators (# — SECTION ————————) to structure your code.

```
# --- SECTION -----------------)
```

18. In mathematical expressions, always use parentheses to define priority explicitly. Never rely on implicit operator priority.

```
(( 1 + 2 ) / 3 ) * 4
```

19. Do not use reserved words as object names.

20. The "if", "for", etc. are language keywords, not functions. Separate the following parenthesis from the keyword with a space.

```
# Good
if (xValue < 5) {
  ....
}
# Bad
if(xValue < 5) {
  ....
}
```

21. Never separate function names and their following parentheses with spaces.

```
# Good
print("Hello world")

# Bad
print ("Hello world")
```

22. Always use a space after a comma, and never before a comma. Except in subsetting expressions, where we don't want the comma to hide against the bracket.

```
# Good
yValue <- c(10, 15, 16)
print(1 / 3, digits = 10)
if (! id %in% IDs) { ...}
expressionProfiles[ , 1:3]


# Bad
yValue <- c(10,15,16)
print (1 / 3 ,digits=10)      # space before the comma
if (!id %in% IDs) { ...}      # "!" hides against the parenthesis
expressionProfiles[, 1:3]     # "," hides against the bracket
```

Names

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

```
# Good
L2ProblemSet.R # .R ensures this file is an R script
```

24. Use the concise camelCaseStyle for variable names. Don't use the confusing.dot.style or the rambling pothole_style (also called the snake case). However, if you are used to pothole_style and believes this style makes your code more legible, then you may use that style. Which ever style you pick, camelCaseStyle or pothole_style, stick to it.

25. Never reassign reserved words.

26. Don't use c as a variable name since c() is a function.

27. Don't call your data frames "df" since df() is a function.

Conditionals

28. 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
}
```

29. Make the FALSE behaviour explicit. Always use an else at the end of a conditional to define what your code does if the condition is not TRUE. Otherwise your reader will wonder whether your code covers all cases. What if your code should do nothing in the FALSE case? Make that explicit:

```
# Good
if (a > b) {
  tmp <- b
  b <- a
  a <- tmp
} else {
  ; # does nothing
}
```

Indent Style

30. Opening brace on the same line as the function or control declaration; (2) closing brace aligned with the declaration; (3) braces mandatory even if there is only one statement to execute.

```
if (length(x) > 1) {
  perm <- sample(x)
} else if (length(x) > 0) {
  perm <- x
} else  {
  perm <- NULL
}
```

31. Use spaces to align repeating parts of code in long function declarations, so errors become easier to spot.

Loops

32. Pre-allocate your result objects to have the correct size if at all possible. Growing objects dynamically with c(), cbind(), or rbind() is much, much slower.

Reminder: 5. Always use for (i in seq(along = x)) {...} rather than for (i in 1:length(x)) {...} because if x == NULL the loop is executed once, with an undefined variable.

```
# Good
xValue <- NULL
for (i in seq(along = xValue)) {
  cat("\n", xValue)
}
# Why?
seq(along = xValue) # integer(0)


# Bad
for (i in 1:length(xValue)) {
  cat("\n", xValue)
}
# Why?
1:length(xValue) # 1 0
```

Functions

33. Explicitly assign values to crucial function arguments, even if you think you know that that value is the language default.

```
# Ok
sort(x)


# Better
sort(x, decreasing = FALSE)
# This expression explicitly tells the reader what it is going to do
```

34. Always explicitly return values from functions, never rely on the implicit behaviour that returns the last expression.

```
# Good
AddValues <- function(x, y) {
  resultValue <- x + y + 1001L
  return(resultValue)
}


# Bad
AddValues <- function(x, y) {
  x + y + 1001L
}
```

35. If there is nothing to return because the function is invoked for its side effects of writing a file or plotting a graph, write it into your code that nothing will be returned. This prevents you from accidentally returning the result of last expression anyway (as the language does by default), or the reader might think you forgot something.

```
return(invisible(NULL))
```

36. In general, return only from the end of the function, not from multiple places.

```
testFunction <- function(argOne, argTwo) {
  ...
  ...
  ...

  return(resultValue)
}
```

Efficiency

37. If possible, do not grow data structures dynamically, but create the whole structure with "empty" values, then assign values to its elements. This is much faster.

```r
# This is really bad:
 system.time({
   N <- 100000
   v <- numeric()
   for (i in 1:N) {
       v <- c(v, sqrt(i))
   }
 })
# Duration will also vary depending on the machine used,
# R version, etc.
#    user  system elapsed
# 16.718  11.258  27.988

 # Even only writing directly to new elements is much, much better:
 system.time({
   N <- 100000
   v <- numeric()
   for (i in 1:N) {
       v[i] <- sqrt(i)
   }
 })
#   user  system elapsed
#  0.025   0.003   0.027

# That's about as fast as doing the same thing with a vapply()
# function. The fastest way is to pre-allocate memory, it actually comes
# close to the vectorized operation:
 system.time({
   N <- 100000
   v <- numeric(N)
   for (i in seq_along(v)) {
       v[i] <- sqrt(i)
   }
 })
#   user  system elapsed
#  0.008   0.000   0.007

# Using a vectorized operation is the fastest approach overall
# and the method of choice:
system.time({ v <- sqrt(1:100000) })
#   user  system elapsed
#  0.001   0.001   0.002
```

39. Add spaces and lines. The code will work without them, but it's hard to read.

```
# Bad
ads_data%>%gather(key="key",value="val")%>%mutate(is.missing=is.na(val)%>%
group_by(key,is.missing)%>%summarise(num.missing=n())

# Good
ads_data %>%
    gather(key = "key", value = "val") %>%
    mutate(is.missing = is.na(val)) %>%
    group_by(key, is.missing) %>%
    summarise(num.missing = n())
```

[END]

40. Always end the script with # [END] comment. This way you can be sure it was copied or saved completely and nothig has been omitted. This is important in teamwork. If even ONE team member does not adhere to this, it invalidates the efforts of EVERYONE.

```
...
# [END]
```

Further Readings:

Tidyverse Style Guide: https://style.tidyverse.org/index.html

Google's R Style Guide: https://style.tidyverse.org/index.html

ToCamelCaseorUnderscore: https://citeseerx.ist.psu.edu/viewdoc/summary;jsessionid=CCEC9E410DAC18876AE0225D4629C5C8?doi=10.1.1.158.9499

R coding style and organizing analytical projects: https://www.stat.ubc.ca/~jenny/STAT545A/block19_codeFormattingOrganization.html#r-coding-style-and-organizing-analytical-projects

References:

- Allaire, J. J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2022). rmarkdown: Dynamic Documents for R. R package version 2.14. URL https://rmarkdown.rstudio.com.
- Google. (2021). *styleguide*. Google's R Style Guide. Retrieved from https://google.github.io/styleguide/Rguide.html
- Steipe, B. (2020, September 18). *R coding style*. R coding style; software developmen.

Retrieved from http://steipe.biochemistry.utoronto.ca/abc/index.php/RPR-Coding_style

- Wickham, H. (2021). *The Tidyverse Style Guide*. Style Guide. Retrieved from https://style.tidyverse.org/index.html
- Xie, Y., Allaire, J. J., and Grolemund, G. (2018). R Markdown: The Definitive Guide. Chapman and Hall/CRC. ISBN 9781138359338. URL https://bookdown.org/yihui/rmarkdown.
- Xie, Y., Dervieux, C., and Riederer, E. (2020). R Markdown Cookbook. Chapman and Hall/CRC. ISBN 9780367563837. URL https://bookdown.org/yihui/rmarkdown-cookbook.