Bono Usu

A guide for good practice in R programming

Jon Calder & Lorenz Walthert

Contents

Preface

Bono usu is Latin for *good practice*. The aim of this book is to outline some good practice for R programming.

Chapter 1

Introduction

Welcome to *bono usu*, a book about good practice for R programming. This book tries to compile and further build upon resources on the topic of good practice in R programming in a somewhat broad sense. Mainly, Hadley Wickham's style section in Advanced R and Google's R Style Guide served as points of reference for this book. We acknowledge that there is no clear 'best practice' when it comes to many aspects of R code style, workflow etc, but there are certainly some principles which can direct one towards 'better practice'.

When collaborating with others on code projects it is hugely beneficial (if not essential) to agree on and abide by some style guidelines and workflow in order to facilitate the process of collaboration. The aim of this book is to bring together some popular opinion, convention, and/or suggested practice which has helped to inform us in facilitating our own consensus on 'good practice'.

The book covers the following topics:

- Code style guide: This chapter is about low level style use in R.
- Workflow & collaboration: In this chapter, we introduce some commonly used workflow tools, and explore how one can make the most of them, while taking a broader perspective on high-level practices in the organisation of projects.
- Package development: Finally, we address the topic of developing extensions for R.

The target audience of the book are R users that have reached an intermediate level. However, we try to include reference material to some fundamental concepts that one needs to be familiar with in order to follow the author's reasoning.

Chapter 2

Code style guide

This chapter deals mostly with low-level coding style. Every section starts with a code chunk to quickly highlight the principle of that section. Depending on how strongly we agree/disagree with a certain practice, we use `# good / bad` or `# advised / discouraged` to indicate the importance of adhering to a certain style.

After the summary, the pros and cons of certain approaches are discussed in more depth.

2.1 Naming

2.1.1 Naming files

```
# good
reduce_matrix.R
# bad
rM_1.r
```

At the point of creating files, it is important to choose a good name. The same applies for the directory where it is stored. Changing the name or location of a file later can cause a number of problems:

- outdated dependencies due to invalid paths.
- removed history in the version control system (e.g. not with `git mv`) if renaming is not done properly. [1]
- discontinuity in the history. It is hard to find renamed files.
- processes that take the file name as an input and return an output that is related to this name are problematic. For example, when creating an html file from an Rmd file via `knitr()`, the created object has the same name (but a different suffix) as the Rmd file. If you change the name, the old html file will still be there but a new html file with the new name will also be there. Most likely, you need to clean up manually.

Hence, you should avoid renaming files if possible. When naming files, the following considerations are worth taking into account:

- Sometimes, there are restrictions in place for file names (e.g. the package `testthat` used in package development requires that all files pertaining to testing start with "test" and live in a particular folder).

[1] See this website for a quick introduction to git, a popular version control system.

- Do use expressive but concise names. Describe the function of the file (`clean_temperature_data.R`), what it inputs or outputs (`daily_to_monthly_transformation.R`), how it relates to other files (e.g. `helper_sort.R` is a helper function for a sort routine) and so on.
- Do not use version numbers in the file name. Use a version control system like git instead.
- Use all lower case names. One motivation for this is that it can save keystrokes and improving typing efficiency. Possibly more important though is that when recalling functions (given that R is case-sensitive), either yourself or other R developers and users will have a harder time remembering what case you used for a function name. Instead of using camelCase, separate elements of the name with an underscore.
- It can be advantageous to use a combination of numbers and descriptive words as a file name. This allows you to a) select files efficiently with bash or even keyboard navigation in a user interface and b) can be informative if you want to maintain a certain order of the files (e.g. if scripts have to be run in a certain) order. However, note that if you use this system, you need to change the name of files if you want to insert an element at a position other than the end of the sequence.
- Since some systems are not case-sensitive, do not use file names that only differ in the case of the characters (e.g. `readme.md` and `README.md`).
- Choose unique file names, even across directories. This avoids confusion and allows you to find the file you are looking for very quickly with a search function or *go to* command such as the one in RStudio.
- Use only alpha-numeric characters where possible and avoid using special characters. While Linux only forbids the use of / in file names, Microsoft forbids <,>,:,“,/,„|,? and * in file names. So while it may be possible to use a range of special characters on some platforms, it is ill advised to do so and will likely cause compatibility issues and headaches for others. The only special characters you should really need to make use of are _ and -.
- Files with extremely long names might be very expressive, but that comes at a cost: they are not memorable, often truncated for displaying and hence not really helpful anymore.
- Short and concise names are preferable but do refrain from using abbreviations since that makes it hard for people to understand them if they lack prior or implicit knowledge about your code. If it is absolutely inevitable, provide a dictionary and a reference to it in the readme file.
- stick to a format if the content you are naming has one i.e. if your files refer to specific dates or times, always use the same representation, e.g YYMMDD. Other examples are names (firstname_lastname).
- Do not use synonyms, mix British or American English etc.
- Always use the same case for file extensions. Although `xzy.r` and `xyz.R` are both recognized as R files, we recommend the latter.

To sum it up, here are a couple of bad examples for the cases described above.

```
# version number in file name
cleaning_3.R
# special character
is_it_correct/.Rmd

# length
this_name_is_expressive_but_will_always_be_truncated_and_hence_useless.pdf
# (uncommon) abbreviations
vp8rm_fs.R
```

Do not use these pairs of examples simultaneously.

```
# avoid synonyms
quick_data_check.R
fast_data_check.R
# stick to one format
2016-02-15_tidy.RDS
15_Feb2016.csv
```

On the other hand, these file names meet the criteria outlined before.

```
# lower case names
showcase_outlier.Rmd
# underscore as a separator
combine_a_and_b.R
# files in order
01-get_data.R
02-clean_data.R
03-visualize_data.R
# uppercase extension for R
for_mint.R
```

2.1.2 Naming objects

```
# good
reduce_matrix # for a function
reduced_matrix # for a variable
# bad
convert.characters # resembles S3 method
```

A lot of rules that were described in the above section apply for naming objects (in the sense of bindings within namespaces) as well. In particular we would like to highlight the following:

- use verbs for functions and nouns for variables. This helps you distinguish objects into categories.
- do not use the dot within object names. It gives them the appearance of methods
- Avoid function names and variable names that already exist or have a particular meaning. For example, `data()` is a function in base R, so it might be easier to call your data `x`, `dta`, `raw` etc. Here are a couple of other examples of bad R object names.

```
# bad
F <- NULL
list <- "not an actual list"
mean <- "I mean he is not nice, he is quite the opposite"
sum <- function(...) summary(...)
```

- Think about how scoping rules might affect your code and name wisely.

2.2 Assignment

```
# good
a <- 2
# bad
a = 2
```

Although programmers coming from other languages may not immediately see the benefit of using the so called *assignment operator* instead of the equals sign, the main advantage is that that an *asignment context* can be distinguished with ease from other contexts, for example from a function call, from a list creation and from a comparison of two objects.

```
f(a = 1, b = TRUE)
list(a = 1:3,
     b = sample(letters[1:4]))
a == b
```

2.3 Quotes

```
# advised
"double quotes"
# discouraged
'single quotes'
```

In R, both double quotes and single quotes are available. The advantage of single quotes is that they result in a slightly cleaner visual representation of the code.

```
print('this is nice'); print("this is a bit less clean")
```

On the other hand, double quotes can be used to enclose single quotes, whereas the reverse is not possible. This can be useful to create a string like the following.

```
varname <- "index"
found <- sample(c("n't ", ""), size = 1)
print(paste("the variable '", varname, "' was", found, " found", sep = ""))
```

Note that in this example, single quotes behave just like any other character. Hence, is not necessary to *close* an open single quote. This allows us to also use them as contraction in `varname` without escape. Depending on whether or not compatibility with other programming languages is required, double quotes are probably also a safer option in certain cases.

For these reasons, we advise the use of double quotes, so all strings are wrapped in the same type of quotes.

2.4 Line indentation

Indenting code makes it more legible. For example, you can see where a conditional statement starts and where it ends.

```
# good
if (condition == TRUE){
  # quite a bit of code
  # ...
  if (another_condition == FALSE){
    # we are going even deeper...
  }
} else {
  # another block of code
}
```

Proper spacing is related to indentation and helps if a lot of consecutive lines are similar but not exactly the same. Below, the difference of the two lines can be spotted pretty much immediately.

```
# good
big_range   <- sample(1,   size = 1,   replace = FALSE)
small_range <- sample(100, size = 100, replace = TRUE)
```

That is because some extra space was added in certain places. Without this extra spaces, the code would look less compact.

```
# bad
big_range <- sample(1, size = 1, replace = FALSE)
small_range <- sample(100, size = 100, replace = TRUE)
```

Another place to make use of indentation is when vectors are created

```
# good
height <- list(Daniel = 1.98,
               Melody = 1.69,
               Peter  = 1.87)
```

There are two ways to indent: either using tabs or using spaces. We suggest two spaces for indentation, but in certain cases as the last example, it might be wise to use a more flexible indentation rule, i.e. to allow all elements of the list to appear exactly in line.
For RStudio users: By default, RStudio inserts spaces when you type tabs, so you can have the best of both worlds. If you prefer to press tab instead of spaces, just make sure that RStudio inserts the same amount of spaces as other people you collaborate with use. Otherwise, it will mess up the indentation. Regarding the more flexible indentation mentioned above, RStudio takes care of that too.

2.5 Spacing

```
#good
if (a == 3 | b > 4){
#...
max(c(1, 2, 3, NA), na.rm = TRUE)
a[[1]][3]

# bad
if (a==3|b>4){
# ...
max ( c(1,2,3,NA) ,na.rm=TRUE )
a[[1 ] ] [ 3]
```

- All binary operators such as +, /, & etc should be surrounded by a space.
- A comma is always followed by a space.
- A left parenthesis should be placed before each function call except `if`.

2.6 Line length

Lines should not exceed 80 characters. There are several reasons for this:[2]

- On a typical screen, the code typically gets displayed only in one tab or window, so space is limited. Since you will always have lines with very little code, it is a waste of screen space to have large differences in the line width.
- Line-by-line diffs in a version control system are more legible if lines are not too long.
- It is rare, but if code needs to be printed, endless lines are a nightmare.

2.7 Commenting code

Commenting code in a broad sense is structuring code. In a narrow sense, it is about explaining *why* something is done the was it is done, rarely the *how* also needs to be explained.
Comments always start with `#` in R. For three levels of granularity, we can use `###`, `##` and `#`.

[2]Some of the reasons outlined here are taken from this stackoverflow question.

2.7.1 Structuring code

To structure code, it can be grouped into segments, sections and subsections. We recommend starting with sections only, so as your code base grows, you can extend in both directions of granularity. However, since you don't want to take it too far with commenting, three levels will hardly be necessary in most cases. Also note that segments use roman numbers, sections arabic numbers and subsections letters. The idea is that segments are represented by the thickest *lines* (since they represent the top level) and subsections are represented by *loose lines*.

```
# good
### ============================================================================
### Segment I
### Description of what the first segment does
## ----------------------------------------------------------------------------
## Section 1
## Description of what the first section does
# ............................................................................
# Subsection a
# Description of what the first subsection does

# [your code goes here]

# ............................................................................
# Subsection b
# Description of what the second subsection does

# [your code goes here]


### ============================================================================
```

We discourage the use of plain `#` for a line of comment with the title of the section centered.

```
# bad
############################## A TITLE GOES HERE ##################################
```

The reasons are the following:

- `#` is rather dominant and has an irregular pattern, which is likely to cause distraction to the eye.
- Putting the title in the same line as the separator has two main disadvantages: For every separator, the width of the `#`s has to be adjusted plus it is tedious to get the title exactly centered. Also, no code is centered, so why should these markers be?
- Just plain `#` allows only one level of granularity.

2.7.2 Explaining code

```
# good
sum(x) # used to do xyz below
# bad
mean(x) # take the mean
```

If code is not extremely complex, the user wants to know *why* someone does something, not *how* (because he can read the code). Often, it makes sense to organize comments in blocks instead of commenting each and every line. If you are writing a function and its inputs and outputs need to be explained in more depth,

create a package for it and use the `roxygen2` package[3] to generate help files from a comment block at the top of your function. Comments should be put next to the function calls on the same line. If they do not fit there fully, move them to the line above the code

```
# good
a_function_you_want_to_call(x = rep(1:10, 3), y = rnorm(10)) # a quick comment

# longer comments that do not fit on the same line with the code are put above
a_function_you_want_to_call(x = rep(1:10, 3), y = rnorm(10))

# bad
a_function_you_want_to_call(x = rep(1:10, 3), y = rnorm(10)) # longer comments
# that do not fit on the same line with the code are not put like this
```

In line with the previous subsection, we recommend having a set of three levels of granularity available for explaining code. Start with `##` so you have an option for both lower level and higher level comments left.

2.8 Nested code

Line indentation is great, but if you hit the third level of nesting, you should maybe consider an approach that does not require so many sub-levels of code.

```
# bad
if (x == TRUE){
  for (i in 4:13){
    if (z[i] == TRUE){
      # ...
    } else if (t == TRUE){
# ....
```

In many cases, you can use functions to avoid nesting, in certain cases, the specific functions you want to use are functionals or the pipe. In more advanced settings, you might want to resort to *recursive functions*.[4]

2.8.1 Functionals

A great way to get rid of nested code related to for loops is to replace them with functionals. However, not all nested code can be replaced with functionals. We do not want to dive into that further here, since Hadley Wickham's book Advanced R covers the material quite nicely. However, let us just quickly discuss an example in the context of good style. The task is to return the data types of each column in the data frame `mtcars`.

```
# good
lapply(mtcars, class)
# bad
for (i in seq_len(ncol(mtcars))){
  print(class(mtcars[, i]))
}
```

Note that using a functional reduced the complexity of the code dramatically, which is certainly beneficial in terms of style. Next, let us consider a functional that uses an anonymous function.[5] One has to take a second glance to realize that the anonymous function is wrapped into another function. Also, note the expression `character(1)` that is supplied an argument of `vapply` and does not belong to the anonymous function.

[3] For more information on the package, check out the vignette.
[4] See an example of how to compute a fibbonacci series
[5] See an introduction to anonymous functions in Advanced R

```
x <- as.list(1:12)
vapply(x, function(r){
    if (r == 12){
      print("this is 12")
    } else {
      print("not 12")
    }
  }, character(1))
```

2.8.2 The pipe operator

On a related note, we should mention the pipe (`%>%`), a function that was introduced in the `magrittr` package. It allows us to chain multiple operations instead of nesting them, which significantly improves readibilty, See the following example provided in the package vignette. The following two code blocks evaluate to the same result.

```
# good
library(magrittr)
car_data <-
  mtcars %>%
  subset(hp > 100) %>%
  aggregate(. ~ cyl, data = ., FUN = . %>% mean %>% round(2)) %>%
  transform(kpl = mpg %>% multiply_by(0.4251)) %>%
  print

# bad
car_data <-
  transform(aggregate(. ~ cyl,
                      data = subset(mtcars, hp > 100),
                      FUN = function(x) round(mean(x, 2))),
            kpl = mpg*0.4251)
```

Formally, the pipe is defined as `x %>% f(y) = f(x, y)`, that is, the first argument of a function following the pipe is always the output of the statement before the pipe. Hence, the verbal interpretation is best expressed with the word *then*. In the above case: take mtcars, *then* subset it with hp $>$ 100, *then*, aggregate the data, *then* transform it, and *then* return it. Also, the pipe can help you to avoid creating a lot of throwaway variables on the fly that you don't need later on. The following two examples result in the same output.

```
scope <- 1:10
# good
max_of_min <-
  scope %>%
  min() %>%
  max(na.rm = T)

# bad
min <- min(scope)
max_of_min <- max(min, na.rm = TRUE)
```

In this example, we only have one argument for the function `min`, which is transfered with the pipe, so there is no need to supply any argument. Of course, this example is trivial but it generalizes to more complex situations.

2.9 Conditional statements

Much has been written about conditional statements. Since they are such an essential element of R programming and programming in general, it should be part of this book.

2.9.1 Booleans

Boolean values `TRUE` and `FALSE` can be abreviated with `T` and `F`, but it is bad practice since it is harder to spot them.

```
# good
if (cond == TRUE) xyz()
# bad
if (cond = F) xyz()
```

Boolean conditional statements can be abbreviated in the sense that they do not require a comparison counterpart. The following two *if statements* are identical, but the first is more explicit.

```
# adviced
if (cond == FALSE){
  # do xyz
}
# discouraged
if (!cond){
  # do xyz
}
```

The functions `any` and `all` are also helpful when it comes to conditional statements. Writing `all(boolean) == TRUE)` might worsen legibility, so we drop the comparison.

```
boolean <- sample(c(TRUE, FALSE), 10, replace= TRUE)
# good
if (all(boolean)){
  "all values are TRUE. That can't be true."
} else if (any(boolean)){
  "at least some values are TRUE. This is quite likely."
} else {
  "not a single TRUE value?"
}
```

In some situations, booleans allow you to do away completely with conditional statements. Whether tend to advocate for this since the amount of lines used with this approach is four times lower.

```
y = 0
k = 1
boolean = TRUE

# advised
x <- k * boolean * sin(y)
# discouraged
if(boolean == TRUE) {
  x <- k* sin(y)
} else {
  x <- 0
}
```

```
# advised
sin(k * boolean)
# discouraged
if(boolean == TRUE) {
  x <- sin(k)
} else {
  x <- sin(0)
}
```

2.9.2 Simplifying conditionals

In principal, R lets you write a whole *if-else-statement* in one line. That is five times less than using the verbose form. In the light of these relations, it might be ok to use the *one-liner* for very short statements. However, when the situation gets more complex, using a verbose form is adviced to maintain code which is well understandable.

```
# concise
if (fail == TRUE) message("fail") else message("no fail")
# verbose
if (fail == TRUE) {
  message("fail")
} else {
  message("no fail")
}
```

Especially for very short conditional statements, you might want to resort to the function `ifelse`.

```
ifelse(fail,
       "the test did fail",
       "the test did not fail")
```

`ifelse` can also take a vectorized input, which makes it even more useful to eliminate loops.

```
x <- -3:4
# good
sqrt(ifelse(x < 0, NA, x))

# bad
for (i in seq_along(x)){
  if (x[i] < 0){
    print(NA)
  } else {
    print(sqrt(x[i]))
  }
}
```

An alternative to `ifelse` is the function `switch()` if the condition is of class character, especially if there are many conditions to test. Note that switch expects a match and returns an invisible `NULL` if the condition does not match with any option given. Also, note that switch is not particularly flexible since it uses non-standard evaluation and cannot handle numeric input.[6]

```
outcome <- "excelled"
switch(outcome,
       fail  = "the test did fail",
```

[6] i.e. replacing *fail* below with the scalar 1 would raise an error

```
       just_passed = "lucky you",
       passed = "the test did not fail",
       excelled = "you must be Einstein")
```

If statements can also be used in other contexts:

```
# good
paste("the test did", if(fail == FALSE) "not", "fail")
```

A very verbose alternative would be:

```
# bad
if (fail == FALSE){
  "the test did not fail"
} else {
  "the test did fail"
}
```

If the duplication should be avoided, one needs to create a variable.

```
# bad
if (fail == FALSE){
  msg <- "not"
} else {
  msg <- NULL
}
paste("the test did", msg, "fail")
rm(msg)
```

We do not claim that the warpping the conditional statement into a function like `paste` in every siutuation, but in the above case, it seems to make the code more concise while not introducing confusion.

2.9.3 Formulation of conditions

The very same conditions can be formulated in different ways. There are some basic notions that are worth following. For example, note that double negation is not desirable in conditional statements since it is quite confusing in some cases, although it technically works fine. Also, it seems quite natural to put a *true* condition first, or the condition that is expected to apply mostly.

```
# good
if (fail == TRUE){
  message("the test failed")
}

# bad
if (fail != FALSE){
  message("the test failed")
}
```

Alternatively, you might want to define the variable `success = !fail` instead of `fail` in order to not get confused.

```
# good
if (success == TRUE){
  message("the test succeeded")
}
# bad
```

```
if(fail != FALSE){
  message("the test did not succeede")
}
```

2.9.4 Conditional assignment

When an assignment to a variable is conditional, the following is possible.

```
# good
x <-
  if (fail == TRUE) {
    rpois(10, lambda = 2)
  } else {
    rnorm(10)
  }
# bad
if (fail == TRUE) {
    x <- rpois(10, lambda = 2)
  } else {
    x<- rnorm(10)
  }
```

Obviously, this only works for simple cases where values are asigned to only one variable. It is advised in simple cases because it becomes evident very quickly that the whole point of this conditional statement is assigning a value to `x`.

2.9.5 Avoiding redundancy

Note that for conditional statements, the order matters if the conditions are not mutually exclusive.

```
boolean <- sample(c(TRUE, FALSE), 10, replace= TRUE)
# good
if (all(boolean)){
  "all values are TRUE. That can't be true."
} else if (any(boolean)){
  "at least some values are TRUE. This is quite likely."
} else {
  "not a single TRUE value?"
}
```

In other words, when all values in the above example are `TRUE`, then also some are. So exchanging the *if* and the *else if* statement will never print the message that all values are true, because the first condition already applies. To fix that you could be tempted to do the following:

```
if (any(boolean) & !all(boolean)){
  "at least some values are TRUE. This is quite likely."
} else if (all(boolean)){
  "all values are TRUE. That can't be true."
} else {
  "not a single TRUE value?"
}
```

Clearly changing the ordering of the conditional statements like this creates redundancy and should thus be avoided.

Also, lazy evaluation[7] can be useful in combination with conditional statements. For example, you might want to so something within a function based on an input value. However, you do not know whether the value was supplied in the first place. Here are two ways to deal with that situation.[8]

```
# good
if (!missing(x) && x > 10) {
  # your code
}
# bad
if (!missing(x)){
  if(x > 10 ){
   # your code
  }
}
```

2.10 Argument specification

```
# good
sample(100, size = 2, replace = FALSE)
# bad
sample(100, , FALSE)
```

Whether and how many arguments you want to name depends on the context. Naming each argument makes the code more accessible for inexperienced R users but makes it also less concise. Also, by definition, you do not need to specify an argument value if you want to adhere to the default. However, sometimes it can increase the legibility to do so, in particular when you want to highlight that you are using a function twice and specify the argument differently.

```
case_irrelevant <- grep("Look", "the value to look", ignore.case = TRUE)
case_relevant   <- grep("look", "the value to look", ignore.case = FALSE)
```

Regarding the order of the arguments, it is a good convention to stick to the default, even if you name the arguments. How long does it take you to spot the all differences in the following two commands?

```
# good
cut(3:19, breaks = 2, right = TRUE, include.lowest = FALSE)
# bad
cut(include.lowest = TRUE, breaks = 3:5, right = FALSE, x = 3:19)
```

[7] note that `&&` and `||` are the lazy equivalents of `&` and `|`, i.e. using `&` in the below example would raise an error because the second statement cannot be evaluated. With `&&` it does not get evaluated because the first statement returns `FALSE`

[8] If not used in the context of a function, you might use `exists("x")` instead of `missing(x)`

Chapter 3

Workflow & collaboration

Chapter 4

Package development