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. 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.

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 more in 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- outdated dependenicies due to invalid paths.
- removed history in the version control system if renaming is not done properly (e.g. not with `git mv`).
- 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 a html file of an Rmd file via `knittr()`, 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 (i.e. the package `testthat` used in package development requires all files with test to start with "test" and live in a particular folder).
- Do use expressive but concise names. Describe the function of the file (`clean-temperatur_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. It saves typing. Separate elements of the name with an underscore.
- It can be advantageous to use a combination of numers 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 filenames that only differ in the case of the characters (e.g. `readme.md` and `README.md`).
- Choose unique file names, even accross 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.
- Try to avoid using special characters (like ?, \ etc.) since they are not allowed, can cause compatibility issues, need to be escaped and so forth.
- 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 preferrable but do refrain from using abbreviations since that makes it hard for people to understand them if they lack prior or implict 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 respresentation, e.g YYMMDD. Other examples are names (firstname_lastname).
- Do not use synonoms, mix British or American English etc.
- Always use the same spelling of extensions. Although `xzy.r` and `xyz.R` are both recognized as R files, we recommend the latter.

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

```
# synonyms
quick_data_check.R
fast_data_check.R
# version number in file name
cleaning_3.R
# case-sensitivity
check_input.R
Check_Input.R
# suffix
convert_id.r
convert_id.R
# special character
is_it_correct?.Rmd
# format
2016-02-15_tidy.RDS
15_Feb2016.csv
# length
this_name_is_expressive_but_will_always_be_truncated_and_hence_useless.pdf
# (uncommon) abbreviations
vp8rm_fs.R
```

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 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.

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

- 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 = ""))
```

```
## [1] "the variable 'index' wasn't  found"
```

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.

Chapter 3

Workflow & collaboration

Chapter 4

Package development