styleGuide: codeInR

Contents

1. Style guide

1.1 Why a style guide

The importance of having a style guide orienting the development of RDBES R-code is to facilitate a common vocabulary and grammar in the code that makes editing and testing more collaborative and speeds up the process of package building.

There are many different styles and rules proposed for R-coding (see a sample in section ***References***). ICES itself provides a few general guidelines. The style proposed in this document aims to find a consensus between the main general global style rules proposed by a set of different sources [in brackets] and the personal opinions and preferences of some of those involved in RDBES development.

1.2 Main style advice [2,5,11]

"When coding, use common sense and most of all BE CONSISTENT".

This means a) write your code in consistent and predictable style; and b) if you are editing code made by others, adopt the local style, i.e., take a few minutes to look at the code around you and learn its style. Example: If others use spaces around their *if*-clauses, you should do that too; If their comments have "little boxes of stars around them", make your comments have "little boxes of stars around them" too.

1.3 Dependencies

Be careful when introducing libraries that are not already used in a project. Some libraries come with many additional dependencies, and changes to dependencies can challenge code-maintenance. If a problem can be efficiently solved using dependencies already introduced, consider doing so instead.

For instance, many people prefer coding in *tydiverse*, rather than in *base R* and we certainly don't want to miss out on their inputs. However, code in *tydiverse* will ultimately be more difficult to maintain as it requires multiple dependencies and that adds risk to the projects in terms of long-term maintenance.

So, if you know both *tidyverse* and *base R*, code your functions in *base R*. If you do not know or do not feel as efficient when using *base R*, then stick to *tidyverse* so you can keep contributing. Just be ready to accept your code is translated to *base R* at some stage of the packaging progress.

Also consider license constraints on dependencies. Some libraries come with licenses that restrict the kind of licenses the dependent code may be published under.

2. Naming

2.1 File names [1,2,3,10,11]

- End the file name with ".R" [caps!]
- Separate words in the file name with "_" not "." or " ".
- Be concise and meaningful in the naming
  - name the script of a function with the name of that function;
  - put one function per file
  - if the script prepares data, name it "01_Data_Preparation.R" not script.R.
- If your scripts are part of a larger set of steps pre-fix their filenames with numbers so they are easily sorted
  - e.g., "01_Preparation.R", "02_Model.R" not "Preparation.R" and "Model.R"

2.2 Variable names [1,2,3,4,9]

- Prefer nouns when naming your objects (e.g., results)
- Concise and meaningful
- Use camelCaps: initial lower case, then alternate case between words. [2,3,4, RDBES convention]
- Avoid using names of existing objects (RDBES tables, r-variables, r-functions, r-libraries) [1,9]

2.3 Function names [2,3,4,9]

- Concise and meaningful
- Prefer verbs when naming your functions (e.g., summarize)
- Avoid using names of existing objects (RDBES tables, r-variables, r-functions, r-libraries)
- Use camelCaps: initial lower case, then alternate case between words. [2,3,4, RDBES convention]
- Note: according to the above functions like read.csv today would likely be named readCSV

2.4 Package versions [11]

The tradition for R packages is to use version numbers that consist of three counters, for example 1.2-3. It's practical to have the three counters indicate the nature of changes between releases:

- The first counter (major) is incremented when existing user scripts will not give the same output as before. Breaking backward compatibility with a major release can be inconvenient for users, but is sometimes done to adopt an improved overall design.
- The second counter (minor) means new functions, new arguments, or the like. A minor release suggests that it's worthwhile for the user to read about the new functionality.
- The third counter (patch) is used for other improvements. A patch release may introduce bug fixes, improved documentation, etc.

3. General Syntax

3.1 Spaces [1,2,3,4]

- Use space after comma
- Use space around =, ==, +, -, /, *, <- [1,2]
- No space around :, ::, :::
- Use space before left parenthesis (e.g., in if or operations)
- No space before function calls [1,2,3]
- No space around "=" in function arguments [3,4]

3.2 Curly braces

- ident code inside the braces
- first curly brace on first line (not on its own line) [1,2,3]
- only short statements on the same line (and no curly braces)
- *else* statement surrounded by curly braces [2,3]

3.3 Semicolons [2]

- do not use them - separate instructions into different lines

3.4 Line length [1,2]

- Limit 80 characters so it is readable

3.5 Indentation [1,3,4,6,9]

- use spaces [3,4,9]
- indent the multiple instructions within functions or cycles [1,6]

3.6 Assignment

- use <- not =

3.7 Comments [1,2,3,5]

- use ####### to make sections more visible [1,5]
- add 1 space after #
- use capitals for aspects in development or that require special attention
- short comment above commands [2]
- inline comments separated by 2 spaces [3]

4. Functions

4.1 comment at the start of the function [2]

- If possible, use *roxygen2* to document your functions from the start
- Otherwise, functions should contain a comments section immediately below the function definition line. These comments should consist of a one-sentence description of the function; a list of the function's arguments, denoted by *Args:*, with a description of each (including the data type); and a description of the return value, denoted by *Returns:*. The comments should be descriptive enough that a caller can use the function without reading any of the function's code.

4.2 Order of arguments

- Data inputs first, control options after [2]
- To the extent possible, avoid hard-coding alterations of the input data inside the functions. If you do that, flag them with visible prints so assumptions can later be documented.

4.3 Dependencies [11]

- To the extent possible, avoid calling other packages within functions as these are sometimes not maintained. Ideally require only the core R packages, like base, graphics, and stats [11]
- If you have to call external packages, explicitly name them when you call the function (e.g., call "reshape2::melt" instead of "melt")

4.4 Testing

- Write a simple test for each function. Whenever a bug comes up, add a test for that bug before fixing it. This makes the code easier to maintain in the long run, and makes it much easier to reimplement code without reintroducing bugs. When programming packages there are good solutions for running automated tests ("testthat" with devtools, for instance)

4.5 Outputs/returns

- Ensure that the data types returned from functions are consistent. So that the functions can easily be included in programs, and not only in interactive-mode. E.g. avoid sometimes returning a matrix, and other times a vector depending on whether ncols or nrows are 1. Document clearly when non-expected data types might be returned (including NULL).

4.6 Global variables

- Variables defined outside the function: use them sparingly, and avoid writing functions that change them.

5. Plots

5.1 Order of arguments

- inputs at start, titles and labels next, general formatting last (cex, las, col, etc)

6. General layout of scripts

6.1 Opening [2,5,6]

- Copyright statement comment
- Author comments
- File description comment, including purpose of program, inputs, and outputs

6.2 Libraries and sources [2,5,6]

- *source()* and *library()* statements
- name all requirements and dependencies

6.3 Executed statements [2]

- Load data
- Transform data
- Outputs

7. Some additional Do's and Don'ts

7.1 Annotations during development

- Use Capitals so others see
    - Warnings: WARNING
    - Specific tunings: ATTENTION
    - Tips for improvements: WISHLIST (better at script start)

7.2 attach()

- avoid using it - risk of confounding variables [2]

7.3 errors

- Use *stop()* inside your functions to check assumptions on inputs (e.g., data type, dimensionality, presence of NAs)
- While developing a new function, signal with *stop()* options that you have not yet developed
- Use clear explicit messages in your stop arguments so others know why code broke or what needs to be developed [2]

7.4 setwd()

- limit its use - better to use a project directory named upfront [5] and take filenames etc. as function arguments. When *setwd()* is used to access resources like conversion-tables or GIS-files, consider packing as R-package, and put resource files in the package directory "inst" or "data" or in sysdata.rda. This avoids having to update *setwd()* when code is transferred to another computer.

7.5 Temporary object names

- v1, v2, v3 - vector
- t1, t2, t3 - table
- m1, m2, m3 - matrices
- df1, df2, df3 - data frames
- ls1, ls2, ls3 – lists
- Remove objects immediately when you don't need it further

7.6 Memory usage

- use *gc()* to clean memory fully - *rm()* cleans the object but not necessarily the memory associated to it

7.7 Loops

- use *apply*, *lapply* etc to avoid for loops [3]

7.8 Saving

- Don't use default workspace to save objects *.RData* [5,6]
- Save *sessionInfo()* so you can remember later what version of R and packages you used to run

that specific code

7.9 Passwords

- Do not store any passwords in the script (alternative, e.g., package keyring)

7.10 Other

- avoid mixing S3 and S4 [2]
- Review and test your code rigorously – once your code is ready, ensure that you test it rigorously on different input parameters. Ensure that the logic used in statements like *for*-loop, *if* statement, *if else* statement are correct. It is a nice idea to get your code reviewed by your colleague to ensure that the work is of high quality. [6]
- vectorize your code [5,6]. See example http://www.win-vector.com/blog/2019/01/what-does-it-mean-to-write-vectorized-code-in-r/

8. References

1

Hadley hickam

http://adv-r.had.co.nz/Style.html

2

google R style

https://google.github.io/styleguide/Rguide.xml

3

R style guide

http://jef.works/R-style-guide/

4

Bioconductor style guide

https://www.bioconductor.org/developers/how-to/coding-style/

5

Best Practices for Writing R

https://swcarpentry.github.io/r-novice-inflammation/06-best-practices-R/

6

R Best Practices: R you writing the R way!

https://www.quantinsti.com/blog/r-best-practices-r-you-writing-the-r-way/

7

John Myles White - Writing Better Statistical Programs in R

http://www.johnmyleswhite.com/notebook/2013/01/24/writing-better-statistical-programs-in-r/

8

CamelCase vs underscores: Revisited

https://whathecode.wordpress.com/2013/02/16/camelcase-vs-underscores-revisited/

CamelCase vs underscores: Scientific showdown

https://whathecode.wordpress.com/2011/02/10/camelcase-vs-underscores-scientific-showdown/

Consistent naming conventions in R

https://www.r-bloggers.com/consistent-naming-conventions-in-r/

9

R Style. An Rchaeological Commentary

https://cran.r-project.org/web/packages/rockchalk/vignettes/Rstyle.pdf

10

R style guide

https://csgillespie.wordpress.com/2010/11/23/r-style-guide/

11

R Package Development at ICES

https://github.com/ices-tools-prod/doc/blob/master/README.md