EpiTeam R Style Guide

Epiconcept

2023

Contents

1 Introduction

The Epidemiology team aims to harmonise its R coding style across projects.

This document describes the style that we aim to apply. It is based on https://style.tidyverse.org/index.html, initially derived from Google's original R style guide https://google.github.io/styleguide/Rguide.html.

The most important thing about a style guide is that it provides consistency. Making code easier to read will also allow to focus on the content.

Some decisions described in this guide are driven by the wish to keep our R code as simple as possible. Non-R users should be able to read and understand our R code.

If you have any suggestion to this guide, please feel free to share them to the team at strap@epiconcept.fr

2 Syntax

2.1 Assignment

Proper assignment `<-` is preferred over the equal sign `=`. Never use the right assignment `->`:

```
# Good
results <- mean(iris$Sepal.Length)

# Bad
results = mean(iris$Sepal.Length)
mean(iris$Sepal.Length) -> results
```

Variable name and assignment should be on the same line:

```
# Good
iris_sepal <- iris %>%
  dplyr::select(Species, Sepal.Length, Sepal.Width) %>%
  dplyr::arrange(-Sepal.Length)

# Bad
iris_sepal <-
  iris %>%
  dplyr::select(Species, Sepal.Length, Sepal.Width) %>%
  dplyr::arrange(-Sepal.Length)
```

2.2 Naming

2.2.1 Variables

Use `CamelCase` for variable names (similarly to ECDC data warehouse format).

```
# Good
DayOne <- 1

# Bad
first_day_of_the_month <- 1
djm1 <- 1
```

2.2.2 Globals

Global variables and constants should be in `UPPERCASE` and `snake_case`:

```
# Good
COUNTRY_CODE <- c("CZ", "ES", "FR", "IT", "NO")
VACCINE_DELAY <- 14
```

2.2.3 Functions

Use a special `camelCase` for function names (first letter in lower case) and `lower case` for its arguments.

Because your function should reflect an action, start your function name with a verb (see also the fllowing section Function name).

```
# Good
doNothing <- function(x, y) {
  return()
}

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

2.2.4 Files

All file names should be in lower case: datafile, scripts, log files, image files, other output files. Some OS are not case sensitive and this could lead to unexpected issues.

The extension can mix case to respect R standards: `.R` extension for scripts, `.RData` extension for data. If needed `use_snake` case to separate words

```
# Good
do_nothing.R

# Bad
doNothing.R
```

2.3 Packages

Explicitly qualify package name (i.e., namespaces) when calling a function. Several R packages use sometimes the same function names and this may lead to conflicts between functions/packages and to unexpected error messages.

```
# Good
IrisSepal <- iris %>%
  dplyr::select(Species, Sepal.Length, Sepal.Width) %>%
  dplyr::arrange(-Sepal.Length)

# Bad
IrisSepal <- iris %>%
  select(Species, Sepal.Length, Sepal.Width) %>%
  arrange(-Sepal.Length)
```

2.4 Pipe %>%

Pipes can be used but with parsimony. Don't use them for short assignments (one line).

```
# Good
Species <- iris$Species

# Bad
Species <- iris %>%
  dplyr::pull(Species)
```

Try to avoid to use them for highly repeated assignments. Store intermediate results if this can help to check your code or if you can give meaningful names (if result is a step in your analyse).

Don't use the "in place pipe" `%<>%` as it requires another package and is not widely used.

```
# Good
iris <- iris %>%
  dplyr::select(Species, Sepal.Length, Sepal.Width) %>%
  dplyr::arrange(-Sepal.Length)

# Bad
iris %<>%
  dplyr::select(Species, Sepal.Length, Sepal.Width) %>%
  dplyr::arrange(-Sepal.Length)
```

3 Functions

3.1 Function name

A function must reflect an action. Therefore, strive to use verbs for function names. Ideally, try to be consistent and use `camelCase` for function name, and `lower case` for its arguments (see also the section above Functions).

3.2 Return

Use explicit returns. Do not rely on R's implicit return feature. It is better to be clear about your intent to `return()` an object.

```
# Good
add2Values <- function(x, y) {
  return(x + y)
}

# Bad
add2Values <- function(x, y) {
  x + y
}
```

3.3 Call

Please always specify the name of the arguments when calling a function.

```
library(tidyr)
who[1:6, 1:8]
```

```
## # A tibble: 6 x 8
##   country   iso2  iso3   year new_sp_m014 new_sp_m1524 new_sp_m2534 new_sp_m3544
##   <chr>     <chr> <chr> <int>       <int>        <int>        <int>        <int>
## 1 Afghanis~ AF    AFG    1980          NA           NA           NA           NA
## 2 Afghanis~ AF    AFG    1981          NA           NA           NA           NA
## 3 Afghanis~ AF    AFG    1982          NA           NA           NA           NA
## 4 Afghanis~ AF    AFG    1983          NA           NA           NA           NA
## 5 Afghanis~ AF    AFG    1984          NA           NA           NA           NA
## 6 Afghanis~ AF    AFG    1985          NA           NA           NA           NA
```

```
# Good
who %>% tidyr::pivot_longer(
  cols = new_sp_m014:newrel_f65,
  names_to = "variable",
  values_to = "value"
)
```

```
## # A tibble: 405,440 x 6
##    country     iso2  iso3   year variable     value
##    <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1980 new_sp_m014     NA
##  2 Afghanistan AF    AFG    1980 new_sp_m1524    NA
##  3 Afghanistan AF    AFG    1980 new_sp_m2534    NA
##  4 Afghanistan AF    AFG    1980 new_sp_m3544    NA
##  5 Afghanistan AF    AFG    1980 new_sp_m4554    NA
##  6 Afghanistan AF    AFG    1980 new_sp_m5564    NA
##  7 Afghanistan AF    AFG    1980 new_sp_m65      NA
##  8 Afghanistan AF    AFG    1980 new_sp_f014     NA
##  9 Afghanistan AF    AFG    1980 new_sp_f1524    NA
## 10 Afghanistan AF    AFG    1980 new_sp_f2534    NA
## # ... with 405,430 more rows
```

```
# Bad
who %>% tidyr::pivot_longer(
  new_sp_m014:newrel_f65,
  "variable",
  values_to = "value"
)
```

```
## # A tibble: 405,440 x 6
##    country     iso2  iso3   year variable     value
##    <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1980 new_sp_m014     NA
##  2 Afghanistan AF    AFG    1980 new_sp_m1524    NA
##  3 Afghanistan AF    AFG    1980 new_sp_m2534    NA
##  4 Afghanistan AF    AFG    1980 new_sp_m3544    NA
##  5 Afghanistan AF    AFG    1980 new_sp_m4554    NA
##  6 Afghanistan AF    AFG    1980 new_sp_m5564    NA
##  7 Afghanistan AF    AFG    1980 new_sp_m65      NA
##  8 Afghanistan AF    AFG    1980 new_sp_f014     NA
##  9 Afghanistan AF    AFG    1980 new_sp_f1524    NA
## 10 Afghanistan AF    AFG    1980 new_sp_f2534    NA
## # ... with 405,430 more rows
```

4 Code documentation

There are two types of comments. Those which help to structure the code (header, section break) and those which give information about your code.

4.1 Structure

Add sections to your script to structure your scripts and make it easier to navigate from chunk to chunk (shortcut: `Ctrl + Shift + r` for PC or `Cmd + Shift + R` for Mac). Use sub-headings as much as necessary.

This also allows the use of the RStudio tree view for navigating your code (see `Outline` button).

```
# 0 - Script information ------------------------------------------------

# 1 - R packages --------------------------------------------------------

# 2 - Import data -------------------------------------------------------

## 2.1 - Case-based -----------------------------------------------------

## 2.2 - Aggregated -----------------------------------------------------
```

Use a standard header when you create a R script (see the section R Studio snippets for epihead) .

It is useful to mark the end of a script because it helps to know which script has run properly.

```
# END of main.R ---------------------------------------------------------
```

4.2 Comments

Add comments to your code! Uncommented code will be very difficult to maintain. Insert simple comments (no section) before every chunk of code.

```
# Renaming variable
```

They should mainly explain **why** you are doing something. Comments are there to help people who don't know your project to understand your objectives. You don't need to explain the R syntax except for tricky constructions.

```
# Bad
# Renaming variable
```

```
# Better
# Renaming variable according to the data dictionary
# stored in 'whatever_data_disctionnary.csv'
```

Another example:

```
# Bad
# Checking data
```

```
# Better
# Checking for incorrect date format in vaccination dates
# to build list of inconsistencies (sent to country)
```

5 R Studio snippets

Snippets are shortcuts designed to insert code into your scripts.

Using snippets you will have harmonised pieces of code avoiding copy/paste from one project to another.

5.1 How to install them?

In RStudio, go to the tools menu, then "edit code snippets" options. It will open an editor containing a file with a list of all snippets already defined (probably the default from RStudio). Copy the code of the desired snippet such as epihead snippet, and paste it at the end of the snippets list.

5.2 How to add them in a script?

Write the name of the snippet you want to include in the script (ex: epihead). Press `tab` after. The code saved in the corresponding snippets will appear in your script.

5.3 Caution

The snippet keyword should be inserted at the beginning of the line, all the following snippet declaration should be indented with one space.

If some "epi" snippets previously inserted already exists remove them before pasting the new one.

If you get any red squares once you copy the next page's snippets, you can select the snippet code, then press `shift-tab` then `tab` to remove them.

5.4 List of snippets

5.4.1 epihead

Aim: To insert a standard header (You can put it at the top of each script)

Code to be inserted into snippets file:

```
  # Project Name :
  # Script Name  :
  # GitHub repo  :
  # Summary      :
  # Date created :
  # Author       :
  # Date reviewed:
  # Reviewed by  :

  # Description ---------------------------------------------------------------
  #


  # Changes Log ---------------------------------------------------------------
  #

  # START of SCRIPT  ----------------------------------------------------------


  # END of SCRIPT  ------------------------------------------------------------
```

5.4.2 todo

Aim: To insert a standardised todo marker.

Code to be inserted into snippets file:

```
  # ---------------------------------------------------------------------------
  # Todo : ${1:todo}
  # ---------------------------------------------------------------------------
```

5.5 Example of snippet use

```
# Project Name : I-MOVE-COVID-19
# Script Name  : restriction flowchart countries surveillance.do
```

```
# Summary      : Flowcharts that show the original data received, and the restrictions to the final dat
# Date created: August 2020
# Author       : Esther Kissling
# Date reviewed:
# Reviewed by  :

# ------------------------------------------------------------------------------------------------
# Description :
# This do-file displays the original data received by country and shows all the records dropped
# due to restrictions until the final dataset.
# While really this is an analysis do-file, it is used for data validation, so we keep it here
# (for the moment).
# The output automatically goes to the Excel spreadsheet "Restrictions flowchart surveillance.xlsx" -
#   in the folder Excel/data validation
#


# ------------------------------------------------------------------------------------------------
# Log version :
```

6 Tests: TO BE DEVELOPPED

7 NEWS: TO BE DEVELOPPED

8 GitHub: TO BE DEVELOPPED

9 Conclusion

But remember... this is just a guide, it is not there to cause us more problems than it solves ! Guide is there to help as much as possible, it's not an absolute ! If following those rules seems difficult, then we could re-evaluate them.

Have fun !