Best practices and open points

Stefan Linner

2022-09-03

Package Principles {MCS}

In developing the `tidyMC` package, we followed the principle of "fail early and often", i.e. if a function is misspecified, we want our function to fail as early as possible and also return readable and informative error messages. To achieve this goal, the first part of each function in our package consists of assertions to the input parameters, as these are specified by the user and are therefore error-prone. After making sure that the input parameters are correctly specified, the following core of the function should work without problems and can thus be executed. In our package, we ensure the class and general structure of the input parameters by using the `checkmate` package, as the `checkmate` commands can almost always be written in a readable line of code, produce readable error messages and are written in C, so we don't have to worry about execution time overhead.

Since one of our main focuses is to embed our functions in `tidyMC` into the `tidyverse` structure, we have used functions from packages of the `tidyverse` whenever possible. This is particularly evident in the following three cases: First, we use `stringr::str_c()` instead of `paste()` as this is generally faster and more stable. Second, we mostly work with tibbles instead of data.Frames, as they provide more consistent subsetting behaviour and we take advantage of tibbles' higher flexibility by storing lists in columns. Thirdly, for all repeated calculations that are normally handled by for loops, we implement their `purrr` counterparts. These are all implemented taking into account the type of the object being handled to reduce computation time.

To work collaboratively on our package code, we use the version control system `Git` and manage our Git project with `GitHub`. This ensures that our pieces of code are merged correctly and nothing gets lost along the way. It also allows us to track our development over time and easily revert to older versions of our package. We also relied heavily on Hadley Wickham's book, which is a great summary of best practices in package programming. As recommended by Wickham, when we wanted to use functions from another package, we added the appropriate package to the `NAMESPACE` file with `usethis::use_package()` and then used the form `package::function()` when calling a function from another package in our code. This way, all dependencies are clear to the reader. To reduce package dependencies outside the `tidyverse`, we tried to avoid using packages that are not part of the `tidyverse` whenever possible.

Our typical development workflow (inspired by Hadley Wickham) looks as follows:

1. we implement the desired feature in our code relying on `tidyverse` functions
2. we use `devtools::load_all()` to make our (updated) function available for experimentation
3. we run an example to check whether the functionality is implemented correctly
4. we try to take care about the function documentation using `roxgen2` right away, as at this point all requirements for the function and the functionality are best in our mind
5. we run `devtools::check()` and fix all occurring notes, warnings, and errors
6. a group member who was not involved in the coding runs several examples using the function to test its functionality

- Whenever a bug occurs, it is reported to the developer of the function, who takes care of fixing the bug
- the bug is converted into a failing test (using `testthat`) and the developer's task is to make this test pass. This test ensures that the same bug will not occur a second time.

Unresolved Issues

With the help of various online resources, in particular Stackoverflow, we were able to fix all errors, warnings and notes detected by `devtools::check()` in a (for us) satisfactory way except for one:

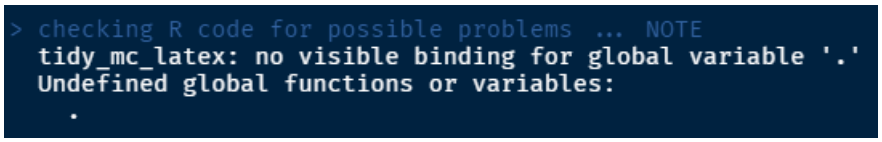

Figure 1: Unresolved Note.

Whenever this note occurs in combination with a `dplyr` function, this can be easily solved by using the `rlang::.data` pronoun after adding `rlang .data` to the `@importFrom` tag in your `roxygen2` header, as explained, e.g., in this post. However, if this note occurs not in combination with a `dplyr` function, but as in our case with `purrr::map()` there is (at least to our knowledge) no such nice solution, as also this stackoverflow post is not answered. Thus, we decided to use the second option presented in the (R-blogger post)[https://www.r-bloggers.com/2019/08/no-visible-binding-for-global-variable/] and set

```
. <- NULL
```

at the respective part in the code. This solution is often used when working with `data.table` package and deals with the note in the RCMD-check as the variable `.` then has a visible binding (to `NULL`). However, it feels a bit "hacky" to us and we would like to know if there is another more natural solution to that problem.