DSCI351-351m-451: Class 10b Clean R Code

Profs: R. H. French, L. S. Bruckman, P. Leu, K. Davis, S. Cirlos

TAs: W. Oltjen, K. Hernandez, M. Li, M. Li, D. Colvin

03 November, 2022

Contents

10.2.3.1 Clean R Code Is Critical

- Over many years of experience delivering successful projects,
  - There is one common element across all these projects

A clean, readable, and concise codebase

- is the key to effective collaboration
- and provides the highest quality value to the client.

10.2.3.1.1 Code Review

- Code review is a crucial part
  - of maintaining a high-quality code process.
  - It is also a great way to
    - share best practices
    - and distribute knowledge among team members.
  - Code review as a must for every project.
    - Lets review best practices
    - recommended for all data science teams.

Having a well-established code review process does not change the fact

- that the data scientist is responsible for
  - writing good, clean code!
- Pointing out all of the code's basic mistakes

    - is painful, time-consuming,
- And distracts reviewers from going deep
    - into code logic
    - or improving the code's effectiveness.

Poorly written code can also harm team morale

- code reviewers are frustrated
    - while code creators might feel offended by a huge number of comments.

That is why before sending the code to review,

- developers need to make sure that the code is as "clean" as possible.

Also, note that there is not always a code reviewer that can come to the rescue.

- Sometimes you are on your own in a project.
- Even though you think the code is ok for you now,
    - consider rereading it in a few months
    - you want it to be clear to avoid wasting your own time later on.

Lets summarize

- the most common mistakes to avoid
- and outline best practices to follow
    - in programming in general.
- Follow these tips to speed up the code review iteration process
    - and be a better data scientist

10.2.3.1.2 Avoid Comments with Comments

- Adding comments to the code is a crucial developer skill.
    - However, a more critical and harder to master skill
        - is knowing when not to add comments.
    - Writing good comments is more of an art than a science.
    - It requires a lot of experience,
        - and you can write entire book chapters about it
        - (e.g., Clean Code: A Handbook of Agile Software Craftsmanship.

There are few simple rules that you should follow,

- to avoid comments about your comments:

The comments should add external knowledge to the reader:

- if they're explaining what is happening in the code itself,
    - it is a red flag that the code is not clean
    - and needs to be refactored.

10.2.3.1.3 Code Refactoring

- What is **code refactoring**
    - code refactoring is the process of restructuring existing computer code
        - changing the factoring
        - without changing its external behavior.
    - Refactoring is intended to improve
        - the design,
        - structure,
        - and/or implementation of the software
        - (its non-functional attributes),

    - while preserving its functionality.

Potential advantages of refactoring may include

- improved code readability
    - and reduced complexity;
- these can improve the source code's
    - maintainability
- and create a
    - simpler,
    - cleaner,
    - or more expressive internal architecture
    - or object model to improve extensibility.
- Another potential goal for refactoring is improved performance;
    - software engineers face an ongoing challenge
    - to write programs that perform faster
    - or use less memory.

10.2.3.1.4 Comments

- So in your code comments, if some hack was used,

    - then comments might be used to explain what is going on.
    - Comment required business logic
        * or exceptions added on purpose.
    - Try to think of what can be surprising to the future reader
        * and preempt their confusion.
    - Write only crucial comments!
        * Your comments should not be a dictionary of easily searchable information.

In general, comments are distracting

- and do not explain logic as well as the code does.

For example, I recently saw a comment like this in the code:

- `trimws(.) # this function trims leading/trailing white spaces`
    - This comment is is redundant.
- If the reader does not know what function trimws is doing,
    - it can be easily checked.
- A more robust comment here can be helpful,
    - e.g.: `trimws(.) # TODO(Marcin Dubel): Trimming white spaces is crucial here due to database entries inconsistency; data needs to be cleaned.`

10.2.3.1.5 Use `roxygen2` inline documentation

- When writing functions in R, I recommend using {`roxygen2`} comments

    - even if you are not writing a package.

```
library(roxygen2)
?roxygen2
```

`roxygne2` is a package used for building R packages

- Generate your
    - Rd documentation,
    - 'NAMESPACE' file,
    - and collation field
- using specially formatted comments.

- Writing documentation in-line with code
  - makes it easier to keep your documentation up-to-date
  - as your requirements change.
- 'Roxygen2' is inspired by the 'Doxygen' system for C++.
- Python3 has `[sphinx](https://en.wikipedia.org/wiki/Sphinx_(documentation_generator))`

`roxygen2` is an excellent tool for organizing the knowledge about

- the function goal,
  - parameters,
  - and output.

10.2.3.1.6 More on commenting

- Only write comments (as well as all parts of code) in English.

  - Making it understandable to all readers
    - might save you encoding issues that can appear
    - if you use special characters from your native language.

In case some code needs to be refactored/modified in the future,

- mark it with the # TODO comment.

Also, add some information

- to identify you as the author of this comment
  - (to contact in case details are needed)
- and a brief explanation of
  - why the following code is marked as TODO
  - and not modified right away.

Never leave commented-out code un-commented!

- It is ok to keep some parts for the future
  - or turn them off for a while,
  - but always mark the reason for this action.

Remember that the comments will stay in the code.

- If there is something that you would like to tell your reviewer,
  - but only once,
- add a comment to the Pull (Merge) Request
  - and not to the code itself.

Example: I recently saw removing part of the code with a comment like:

- "Removed as the logic changed."
- Ok, good to know,
  - but later that comment in the code looks odd and is redundant,
  - as the reader no longer sees the removed code.

10.2.3.1.7 Strings

- A common problem related to texts

  - is the readability of string concatenations.
  - What one encounters a lot
    - is an overuse of the paste function.
  - Don't get me wrong;
    - it is a great function when your string is simple,
    - e.g. `paste("My name is", my_name),`

    - but for more complicated forms, it is hard to read:

```
paste("My name is", my_name, "and I live in", my_city, "developing in",
      language, "for over", years_of_coding)
```

A better solution is to use

- `sprintf` functions
- or `glue`, e.g.

```
glue("My name is {my_name} and I live in {my_city} developing in {language}
     for over {years_of_coding}")
```

Isn't it clearer

- without all those commas
- and quotation marks?

When dealing with many code blocks,

- it would be great to extract them to separate locations,
    - e.g., to a .yml file.
- It makes both code and text blocks
    - easier to read and maintain.

The last tip related to texts:

- one of the debugging techniques,
    - often used in Shiny applications,
    - is adding `print()` statements.
- Double-check whether the prints are not left in the code
    - this can be quite embarrassing during code review!

10.2.3.2 Loops

- Loops are
    - one of the programming building blocks
    - and are a very powerful tool.

Nevertheless, they can be computationally heavy

- and thus need to be used carefully.

The rule of thumb that you should follow is:

- always double-check if looping is a good option.

It is hardly a case that

- you need to loop over rows in data.frame:
    - there should be a {`dplyr`} function
    - to deal with the problem more efficiently.

Another common source of issues is

- looping over elements
    - using the length of the object,
    - e.g. `for(i in 1:length(x)) ....` But what if the length of x is zero!
    - Yes, the loop will go another way
    - for iterator values 1, 0.
- That is probably not your plan.
    - Using `seq_along` or `seq_len` functions
    - are much safer.

Also, remember about the `apply` family of functions for looping.

- They are great
  - (not to mention {the `purrr` package} solutions)!
- Note that using `sapply`
  - might be commented by the reviewer as not stable
  - because this function chooses the type of the output itself!
- So sometimes it will be
  - a list,
  - sometimes a vector.
- Using vapply is safer,
  - as the programmer defines the expected output class.

10.2.3.3 Code Sharing

- Even if you are working alone,
  - you probably would like your program
    - to run correctly on other machines.
  - And how crucial it is
    - when you are sharing the code with the team!
  - To achieve this,
    - **never use absolute paths in your code**,
    - e.g. `/home/marcin/my_files/old_projects/september/project_name/file.txt`.
  - It won't be accessible for others.
    - Note that any violation of folder structure will crash the code.

As you should already **have an Rproject for all coding work**,

- you need to use paths related to the particular Rproject
  - in this case; it will be `./file.txt`.
- What is more, **one would suggest keeping all the paths**
  - as variables in a single place
- so that renaming a file requires one change in code,
  - not, e.g., twenty in six different files.

Sometimes your software needs to use some credentials or tokens,

- e.g., to a database or private repositories.
  - or an external API, like Google Maps
- You should never commit such secrets to the git repository!
  - Even if the entries are the same among the team.
- Usually, the good practice is to keep such values
  - in .Renviron file as environmental variables
  - that are loaded on start
  - and the file itself is ignored in the repo.
  - You can read more about .Renviron here.
- Or use the `keyring` package
  - It stores tokens or credentials
  - And exists for both R and Python3

10.2.3.3.1 Good Programming Practices

- Finally, let's focus on how you can improve your code.
  - First of all,
    - your code should be easily understandable and clean
  - even if you are working alone,

        - when you come back to code after a while,
        - it will make your life easier!

Use specific variable names,

- even if they seem to be lengthy
- the rule of thumb is that you should be able to guess
    - what is inside just by reading the name,
    - so `table_cases_per_country` is ok,
    - but `tbl1` is not.
- Avoid abbreviations.
    - Lengthy is preferable to vague.
- Keep consistent style for object names
    - (like camelCase or snake_case)
    - as agreed among your team members.

Do NOT abbreviate logical values

- such as `T` for `TRUE`
    - and `F` for `FALSE`
- the code will work,
    - but `T` and `F` are regular objects
    - that can be overwritten
- while `TRUE` and `FALSE` are special values
    - as defined in R.

Do not compare logical values using equations,

- like `if(my_logical == TRUE)`.
- If you can compare to `TRUE`,
    - it means your value is already logical,
    - so `if(my_logical)` is enough!
- If you want to double-check
    - that the value is `TRUE` indeed
    - (and not, e.g., NA),
    - you can use the isTRUE() function.

Make sure that your logic statements are correct.

- Check if you understand the difference in R
    - between single and double logical operators!

Good spacing is crucial for readability.

- Make sure that the rules are the same
    - and agreed upon in the team.
- It will make it easier to follow each other's code.
- The simplest solution is to stand on the shoulders of giants
    - and follow the tidyverse style guide.
    - Its the same as the Google R style guide.

However, checking the style in every line

- during the review is quite inefficient,
- so make sure to introduce `lintr` and `styler`
    - in your development workflow
- Our use the code diagnostics in Rstudio
- This can be lifesaving!

Recently we found an error in some legacy code

- that would have been automatically recognized by `lintr`:

```
sum_of_values <- first_element
  + second_element
```

This does not return the sum of the elements

- as the author was expecting.

Speaking of variable names

- this is known to be one of the hardest things in programming.
- Thus avoid it when it is unnecessary.

Note that R functions return, by default,

- the last created element,
- so you can easily replace that:

```
sum_elements <- function(first, second) {
  my_redundant_variable_name <- sum(first, second)
  return(my_redundant_variable_name)
}
```

With something shorter

- (and simpler,
  - you don't need to think about names):

```
sum_elements <- function(first, second) {
  sum(first, second)
}
```

On the other hand, please DO use additional variables

- anytime you repeat some function call or calculation!
- It will make it computationally more effective
  - and easier to be modified in the future.

Remember to keep your code DRY

- don't repeat yourself.
- If you copy-paste some code,
- think twice whether it
  - shouldn't be saved to a variable,
  - done in a loop,
  - or moved to a function.

10.2.3.3.2 Conclusion And there you have it

- five strategies to write clean R code
  - and leave your code reviewer commentless.
- These five alone will ensure you're writing great-quality code
  - that is easy to understand,
  - even years down the road.

10.2.3.4 Links

- Marcel Dubel, Clean Code
- Clean Code: A Handbook of Agile Software Craftsmanship