# Guidelines for data analysis

Cedric Arisdakessian

2022-07-28

This document provide some guidelines when analyzing data your own projects.

## Coding principles

### 1. Clarity

When analyzing data, many people dive right in to the code and cary on until it produces the outputs they are expecting. Once it works, they save the file, export the figures, share it and move on. Unless you are a very experienced coder and can write excellent code on the first try, this is usually not a good idea for several reasons:

- As your project advances, you will need to adjust some parts of your code, maybe add a few steps, change some parameters. If your code is not clear and badly structured, it will take you much more time and energy to do those changes.
- Other people might not be be able to help you: If the code is congested/complicated, it will take them much more concentration and motivation to understand what you're doing. Even if they do, they might just be burnt out by the time they actually get to the core of the issue.
- If you have to get back into it days or weeks after you wrote it, you might not be able to understand your own code.

You will find here a few guiding principles to keep your code clear along with a few examples to illustrate the main points.

#### Before you start

Before writing any code, you should ask yourself the following questions (when applicable)

- What is my workflow / the main steps of my analysis ?
- Is there packages/functions already doing what I want to do or do I need to code it from scratch? In general, unless you want to do something extremely specific, you should find a function for anything.
- Is there any parameters/options that I might be playing with and that are important for the workflow? If so, you might consider defining those variables at the beginning of your script to spot them and modify them easily.
- What kind of data structure should I use for my data? In most cases, it will be a dataframe. But some other times you might need a list or an object specific to your field. In some projects, using the right data structure can significantly simplify your code (e.g. phyloseq objects in 16S data analysis). This can also have a significant impact on the speed of your code (but this is a more advanced topic on data structures and complexity in computer science)

**As you are coding**

- Add **comments**, especially for parts of the code that took you a long time to write. It is also totally acceptable to add a URL to a stackoverflow issue that helped you solve a problem. You can add comments in R using the # symbol before the text.

```
table <- matrix(0, 3, 5) # make a matrix with 3 rows and 5 columns filled with zeros
```

- I like to use the named arguments for functions instead of their value directly. You can know the arguments' names in the function's documentation by using a question mark before the function (e.g. `?matrix`). In the previous example, it might be more readable to write this instead:

```
table <- matrix(data=0, nrow=3, ncol=5) # initialization
```

- Use **clear variable names**. If the variable names are clear enough, you might not even need comments.

```
# Without the comment, we don't really know what "a" is
a <- c("#000000", "#008941", "#FF90C9") # the hex codes for the colors I'm using
```

```
# Instead, with a clear variable name, we don't even need the comment:
color_palette_hex <- c("#000000", "#008941", "#FF90C9")
```

- Don't hesitate to **aerate your code**. It's okay to spread simple code across multiple lines. Break long lines of arguments if it's too long

*Example 1*

```
ints <- 0:10

# All on a single line, hard to read
for(i in ints) { if (i %% 2 == 0) { print("i is even") } else { print("i is odd")  } }
```

```
ints <- 0:10

# More readable, easier to spot missing parenthesis/brackets/curly braces
for(i in ints) {
  if (i %% 2 == 0) {
    print("i is even")
  } else {
    print("i is odd")
  }
}
```

*Example 2*

```
# The line wraps around, harder to read
function_with_long_arguments(very_long_argument_number_one="very_long_value", very_long_argument_number
```

```
# More lines but easier to parse
function_with_long_arguments(
  very_long_argument_number_one="very_long_value",
  very_long_argument_number_two="another_long_value"
)
```

- Make your code **modular** and **don't copy-paste blocks of code** to repeat a procedure, it is very bug-prone: if you were to change something in your procedure, you would need to fix every occurence of code you copy-pasted. What if your forget one? Instead, use for loops and functions.

```
# What if we want to change t-test for another type of test? Or if we want
# to also subset the data to include Oahu samples only?

# t-test for the well data
subset <- data[data$type == "well", ]
t.test(subset$x, subset$y)
# t-test for the pond data
subset <- data[data$type == "pond", ]
t.test(subset$x, subset$y)
```

```
# Better with a for loop. Only one spot in the code to modify
for (type in c("well", "pond")) {
  subset <- data[data$type == type, ]
  t.test(subset$x, subset$y)
}
```

```
# A function is a viable solution too:

t_test_on_subset <- function(data, value) {
  subset <- data[data$type == value, ]
  t.test(subset$x, subset$y)
}

# it is ok to just repeat the function call
t_test_on_subset(data, "well")
t_test_on_subset(data, "pond")

# or to loop
for (type in c("well", "pond")) {
  t_test_on_subset(data, type)
}
```

### After coding

Once it works, take some time to check your code, be critical of any part that looks complicated, congested or unclear. Is there any code duplication you can avoid? Any part you wrote in a rush and that could be rewritten more clearly?

Remove any code blocks you commented and that you will not be using anymore.

### 2. Reproducibility

When you are analyzing data, it is pretty common to encounter functions that incorporate randomness (e.g. random subsampling when normalizing 16S data). Each time you use these functions, you get a different result, which makes your results change each time. Although this randomness is desired to avoid any sort of biases in the function behaviour, it complicates your analyses for multiple reasons:

- First, it complicates bug fixing. Depending on the output of the random function, some errors might occur later in your code or not. If the conditions to produce this error are very rare, it makes it hard to identify what causes it and therefore to fix it.
- Second, it's less ideal when you share your code with the community, or if you want to publish it. If people cannot reproduce your results, they might not trust your work.

For these reasons, most random functions include a "seed" argument, which forces the behaviour or this function to be deterministic. A seed is a fixed integer, that you can set to any value, and everytime you use this function with the same seed, you will get the same results. Naturally, different seeds should also lead to different outcomes. It is also possible that your function does not provide a seed argument (it is the case for most basic random number generators in R base), in which case you should be able to set a seed for before using your command. See below for examples

```
# here we choose a seed of "123"
# Generates 3 random numbers between 0 and 1
set.seed(123)
runif(n=3, min=0, max=1)

# set seed "42" in 16S subsampling
phyloseq::rarefy_even_depth(mb, seed=42)
```

## File management

In general, you want to avoid mixing all of the documents for your project in the same folder next to each other. What I propose in this section does not necessarily fit all projects, but should work well in most cases. A good rule of thumb is that your foldera should not be too congested (e.g. less than 10-15 files), but shouldn't contain a single file either.

Keep all of your work in the same folder, named after your project name and make a different subfolder for each of those categories:

a. Project code (name suggestions: "src", "code", ...) with your .R or .Rmd files.
b. Project data (name suggestions: "data"): The data files generated in the context of your project. Don't hesitate to add subfolders if there are different types of data. For example, you could have raw sequencing reads in the "reads" folder, and then the reads processed by the pipeline in another one called "cmaiki-pipeline-outputs". If there are different types of metadata files (for example temporal/spatial), you can also consider making a folder for that.
c. Project outputs (name suggestions: "outputs", "results") for any file that your code produces. If you are generating a lot of data, you can divide this folder by output type (e.g. figures / stats / tables / ... ) or by type of analysis (ordination / permanova / deseq2 / ...) or a combination of both.
d. External files not specific to your project (name suggestions: "resources", "external", "db" or whichever name makes the most sense in your case). This can be various databases you have downloaded and use for your analysis, primer sequences, etc... If these files are shared across multiple projects, it can also be a good idea to instead have a specific location on your computer outside the project directory with all of these files. For more advanced command-line users, you can also consider using symbolic links of the external files required in your projects.

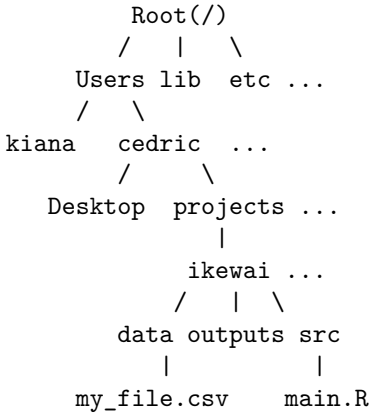
In general, it's also good to keep a text file (usually markdown formatted) to keep notes about your project. You can leave this file at the root of your project folder (for instance called "notes.md" or "README.md")

In summary, here's a visual example of what your project folder could look like:

```
- my_project
  README.md
  - external/[...]
  - src/
      main.Rmd
  - data/
      - pipeline-run/[...]
      metadata.csv
  - outputs/[...]
```

## Absolute vs Relative path

Whenever you want to load a data file into R (or any tool really), you need be able to specify its location (called its *path*) on your computer. Your computer filesystem is organized in a tree-like structure and starts at the root (called "/" in Linux and MacOS, and C:/ in Windows). A branching occurs whenever you have a folder/subfolder. See below an example of a macOS file structure:




Let's say we have a script in our src/ folder called "main.R" and we want to load "my_file.csv". We can refer to it in two ways:

- using the **absolute path**: this is the full path starting from the root to the file. In our example, the absolute path to `my_file.csv` is: `/Users/cedric/projects/ikewai/data/my_file.csv`
- using the **relative path**: this is the path relative to my working directory. If I set my working directory to be where where script is (i.e. `src`), the relative path to `my_file.csv` is `../data/my_file.csv`. Note that ".." means the parent directory of my current location. In RStudio, you can set your working directory using the menu: `Session > Set Working Directory > ...`