WWZ Workshop
Clean Code in Context

Dr. Anthea Alberto (RISE)
antheajeanne.alberto@unibas.ch

4.3.2024
13:00-16:00

Agenda

13:00-13:15	Recap Crash Course
13:15-14:15	File Structure, Data Formats and More
14:15-14:30	Break
14:30-15:15	Group Work
15:15-16:00	Example and Q&A

"I like my code to be **elegant and efficient.** The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. **Clean code does one thing well.**"[1]

- *Bjarne Stroustrup, inventor of C++ (emphasis mine)*

[1] Cited in Martin, R. (2015): Clean Code. Upper Saddle River, NJ: Prentice Hall.

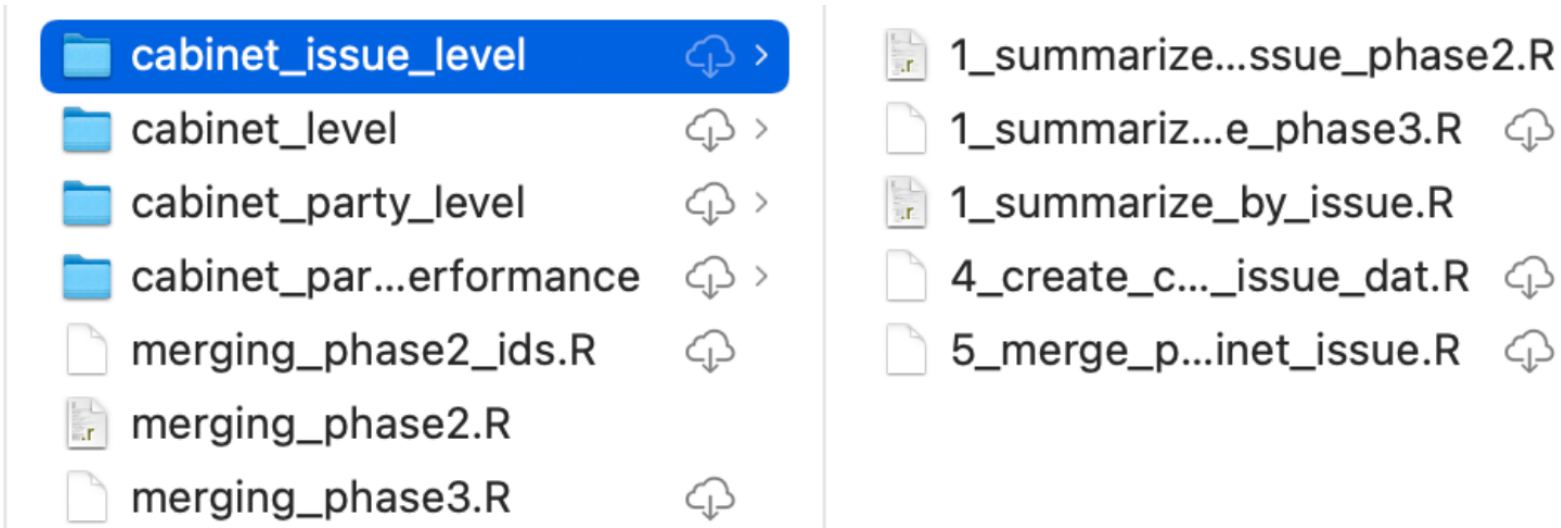
Why you should use clean code

- It saves time & increases efficiency
  - Clean code will reduce the effort of trying to understand a script when coming back to it after a while
- It makes collaboration easier
  - There's no need for project partners to explain their code when you can just run & understand it yourself
- Having well organized scripts will make it easier to go back for reference
  - You'll likely reuse your code a lot during your PhD (and later)
- It is important for **reproducibility**
  - Being mindful of your code while working on a project will save time later when putting together reproduction materials

Recap

- Code for people, not machines
- Use the right names
- Adhere to standards and be consistent
- Use comments & avoid unnecessary ones
- Use a lab journal to keep track of different parameters, models etc.
- **DRY**: Don't Repeat Yourself
- **YAGNI**: You Ain't Gonna Need It
- **KISS**: Keep It Simple, Stupid

File Structure



A good start, but not ideal

File Structure[2]

Why do it? The arguments for clean code apply here as well:

- It simplifies cooperation
  - Create a more streamlined analysis workflow
  - No time wasted when trying to understand project structure
  - Reproducibility
- Future you will be grateful
  - Copy-paste structure for future projects
  - Less time needed to refamiliarize yourself with an older project & its code

[2] Based on this explainer by MIT Communication Lab.

File Structure

General good practice:

- Have one main folder per project
- Subfolders: number and type depends on project (see here for examples)
- Separate raw data from edited data
- Be consistent with naming subfolders and files

Naming conventions

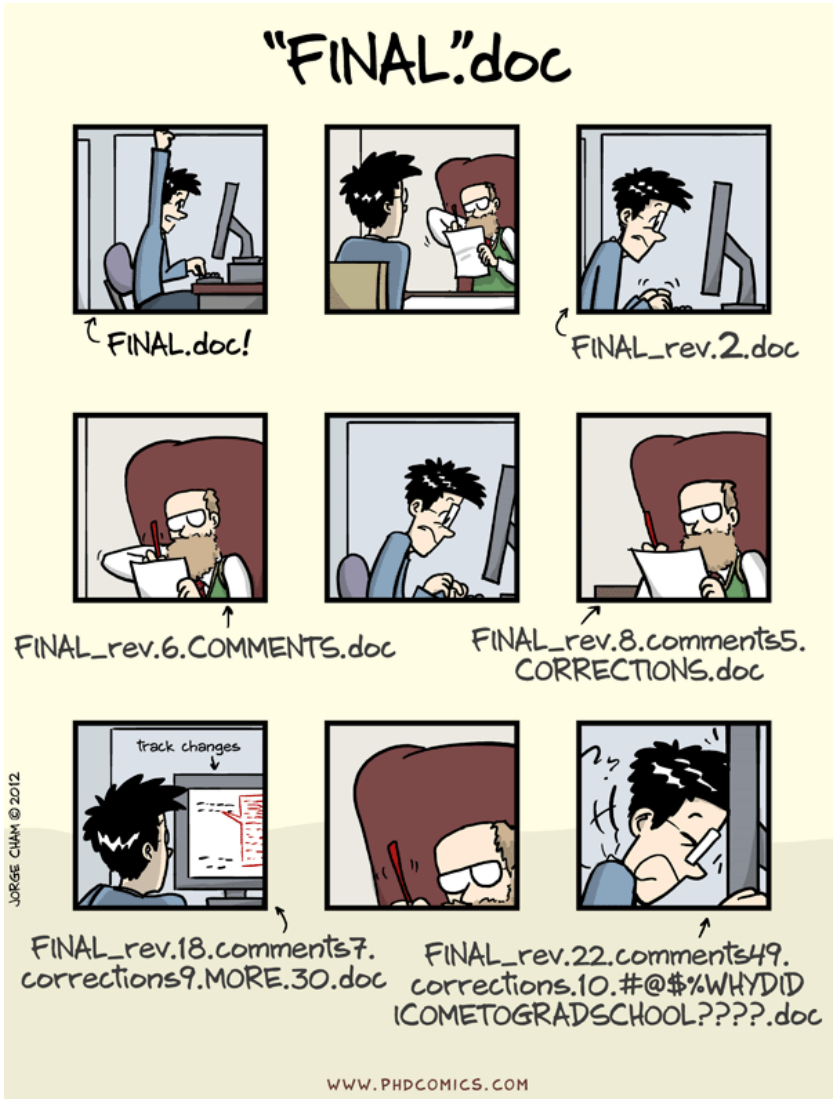
Use underscores, hyphens or periods as delimiters.

Never use spaces!

Another option is camelCase: e.g. `firstNameLastName`

Mixing delimiters can work well in some cases (e.g. camelCase and periods; `20240305.AntheaAlberto.v1.csv`), but I would avoid mixing hyphens, underscores and periods.

Naming conventions



Naming conventions

Instead of the confusing `_final_final_thisone` et al., consider using something like `_v1`, `_v2` etc.

If you use dates (which can make a lot of sense): ideally use YYYYMMDD format (or variations thereof, like YYYY-MM-DD).

Folder structure

The type of folder structure really depends on the *granularity* of your project.

```
├── Makefile
├── README.md          <- If you don't have a README
│                         did you even make a repo?
│
├── data               <- OTU tables (if small enough),
│                         QIIME 2 outputs, metadata excel
│                         files, trees, etc.
│
├── src                <- All code: scripts, notebooks, etc.
│
└── final              <- Final figures, supp files, tables.
```

Example folder structure (Source: Claire Duvallet)

Folder structure

Separate raw data from edited data.

I also recommend this for coding in general: rename a data frame when you edit (subset, aggregate...) it.

If you mess up or want to do something differently, you won't have to reload the data.

This sound negligible, but it can be a time saver if your data frame takes a while to load.

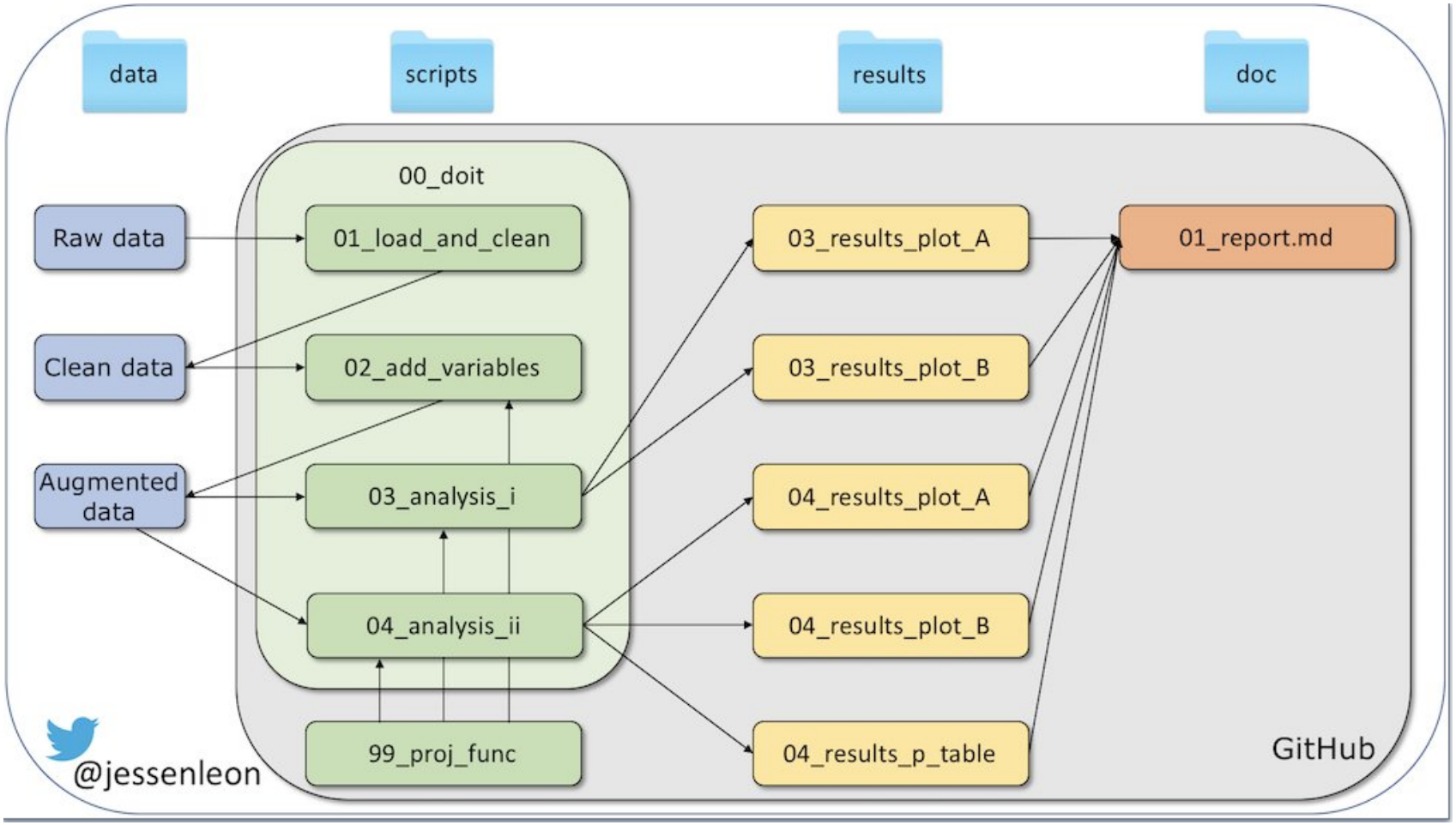
Folder structure

Case Study 2: A Simple Hierarchy

```
PROJECT/
├── bin/            <- compiled binaries.
├── data/
│   ├── raw/
│   └── clean/
│
├── figures/        <- figures used in place of a "results" folder.
├── scripts/
│   ├── process/    <- scripts to maniuplate data between raw, cleaned, final stages.
│   └── plot/         <- intermediate plotting.
│
├── src
│   ├── model1/     <- various experimental models.
│   ├── model2/
│   └── model3/
│
├── LICENSE
├── Makefile
└── readme.md
```

Source: Case studies from mitcommlab

Folder structure/scripts



Source: Leon Eyrich Jessen on Twitter

Folder structure/scripts

Some people prefer to have everything in one script, others like multiple scripts for different tasks.

I don't think there's a definite consensus, but you should definitely have different scripts for different analyses/chapters.

If you have an operation that takes a long time, it might be a good idea to save the results as an `.RData` file and make a new script were you simply load it.

You can find a discussion here.

Examples & Resources

Examples for different project types here.

Tips, tricks and philosophies on computational work by Claire Duvallet

File formats

There are a lot of them.

Generally, you should strive to use something that is usable across operating systems; something not Mac or Windows specific.

I'd recommend .rtf (text) and .csv (spreadsheets), and, more generally, LaTeX.

LaTeX

LaTeX is a typesetting software designed to work independently of OS.

The learning curve is quite steep at the beginning, but it's worth it.

I can recommend Overleaf, an online LaTeX editor.

I teach an intro course on LaTeX for GRACE Transferable Skills (also in HS24).

File encoding

If possible, use **UTF-8** encoding for all your documents.

UTF-8 solves most internationalization issues (meaning it can handle a variety of alphabets without returning gibberish).

Still, you should avoid using special characters (for example ä, ü, ö) in file and folder names etc, since this can lead to issues.

File encoding

In R and Python you can specifically choose UTF-8 as encoding when saving a file (useful when working with text data, currency signs...).

You can also change a document's encoding when loading it into your environment. This may require some trial and error, however.

R Markdown

R Markdown is a good tool for producing nice-looking documents (PDFs or html).

It allows you to write "code chunks", display their output and comment on it - ideal for home works, reports etc.

You can find an introduction plus tips & tricks here: R Markdown

Integrating GitHub with RStudio

Follow the instructions here if you want to integrate GitHub with RStudio.

After following the steps, you'll be able to make commits directly via RStudio, and not use the Desktop app as an in-between.

Integrating GitHub with RStudio

12.4 Make local changes, save, commit

From RStudio, modify the `README.md` file, e.g., by adding the line "This is a line from RStudio". Save your changes.

Commit these changes to your local repo. How?

From RStudio:

- Click the "Git" tab in upper right pane.
- Check "Staged" box for `README.md`.
- If you're not already in the Git pop-up, click "Commit".
- Type a message in "Commit message", such as "Commit from RStudio".
- Click "Commit".

Source: Happy Git with R Chapter 12

Group Work

Group Work

In groups of 3, make a list of things to think about when starting a new project. What needs to be clarified to make your life easier?

- Items on the list can refer to working alone or with others
- You can include your own experiences as well as things you've learned in the course
- The main aim is to have some sort of checklist you can refer to: it should be helpful to you & other students

- Showcase prepared for teaching purposes
- A visualization project supported by RISE (naming could use work, though!)