Day 4, Part 1: Managing Your Project

Brennan Terhune-Cotter and Matt Dye

https://github.com/brennangitsit/2023_IAM3_R

Agenda

1. Managing your project structure
2. Programming within a folder
3. Writing functions ¶

Project Management

Managing your project structure

- Having a clear and consistent project structure will make your life easier, especially as your projects grow.
- **Each study or project should have its own folder** with data, code, and output.
- Project structures vary depending on the project, but here are a few principles that define a general structure:
  - Raw data should be **separate** from processed data
  - Scripts and results should be **separate** so results aren't accidentally overwritten
  - You should have documentation within a clear and readable structure so that people (including you) can understand what's happening in your project folder ¶

My typical project folder

- **my_project**
  - **data**
    - raw_data
    - processed_data
  - **docs**
  - **scripts**
    - behavioral_analysis.R
    - recode_data.R
    - headliner_script.R
  - **results**
  - **outputs**
  - my_project.Rproj
  - README.md

(**folders** are bolded)

Data

- Data should always be kept in two separate folders:
  - the *raw data* that you never touch
  - the *processed data* that you can manipulate and is disposable (because you can always regenerate it from your scripts)
- If you work with big datafiles, often these will not be synced to the cloud or source control (Github - ask me about Git if you want)
- You can also split **processed data** into processed individual-level (first-level) data and group-level (second-level) data ¶

Scripts

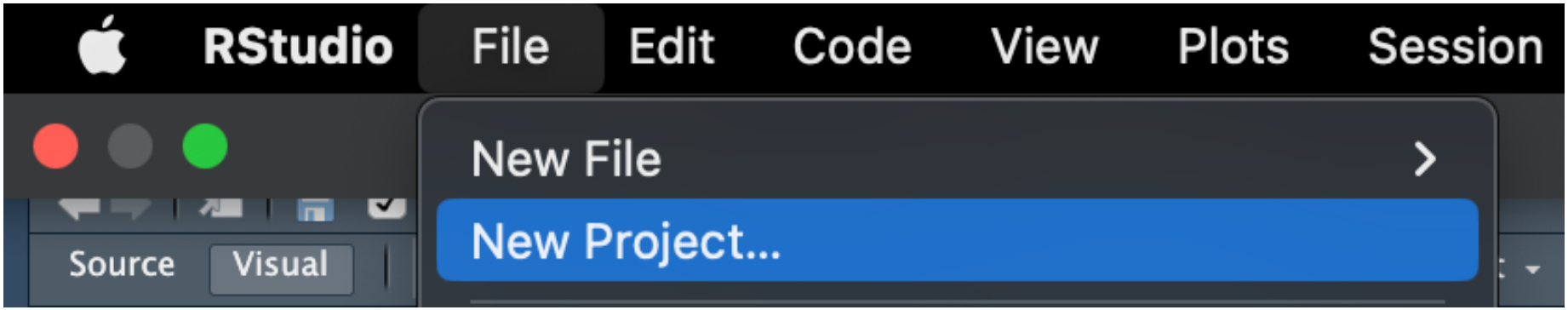
- A clean project structure might have:
  - scripts that *define* functions in the **scripts** folder
  - a headliner/master script in the main folder which *runs* those functions
    - (or multiple scripts for multiple pipelines) ¶

Results and/or Outputs

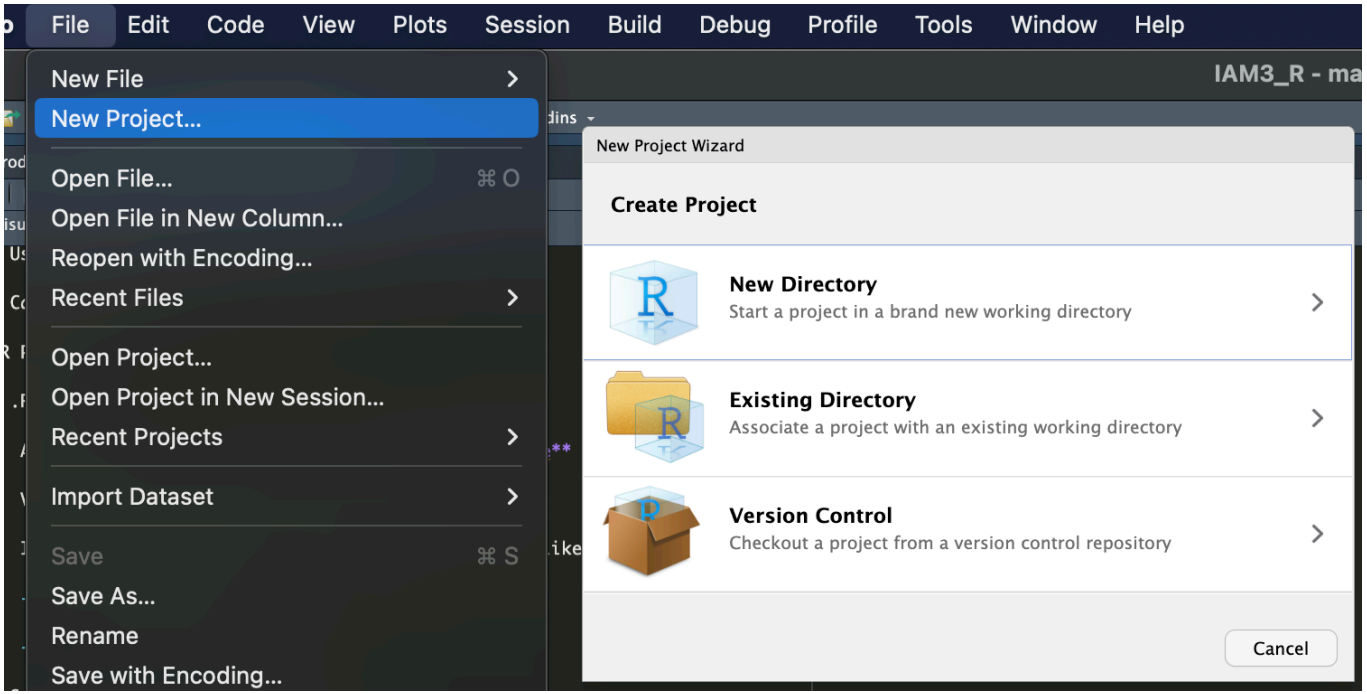
- The **results** folder can contain all your results (results, plots, write-ups) in however format you prefer.
- I like to have two folders:
  - **results** for statistical results, plots, things to show your lab, etc.
  - **outputs** for writing abstracts and posters, making reports, writing papers, etc. ¶

R Projects, Again

- An R project is a file which saves your **workspace**
- It is useful to have an .Rproj file for each project you have
- When you open the .Rproj, everything will look like the last time you saved it.
- Instead of opening RStudio or individual R files (like any other application):
  1. Open your .Rproj
  2. Open files from that .Rproj



Separate Studies, Separate R Projects



- .Rproj files are very useful when you have multiple projects!
- **Each study or project should have its own folder and R project**
- Have one .Rproj for each project you plan to use R for
  - You can put the .Rproj in the main folder or a subfolder (like `main/analysis`)

Coding Locally

Using local paths to stay within a folder

- All project-specific code should be written to work within the project's folder *no matter where the folder is*
- This means using local paths to load data and refer to other files within the folder
  - No slash in the beginning = local path (starts from working directory)
  - Slash in beginning = absolute path (starts from home directory)
  - `./` = current directory
  - `../` = go one directory "up"
- That project file (or analysis file that your scripts are in) will be your *working directory ¶*

```
localFilePath = "scripts/data_cleaning.R"
absoluteFilePath = "/Users/brennanwork/Library/CloudStorage/GoogleDrive-bterhunecotter@sdsu.edu/My
```

Working Directories

- Your *working directory* is where you run your scripts "from".
- You can temporarily change your working directory with `setwd()` ¶

```
1 getwd() # check your current working directory
```

```
[1] "/Users/brennanwork/Library/CloudStorage/GoogleDrive-bterhunecotter@sdsu.edu/My
Drive/IAM3_R/15th_Tidy_&_Manage_Data/15th_slides"
```

```
1 setwd("../") # change your working directory
2 getwd() # check it again
```

```
[1] "/Users/brennanwork/Library/CloudStorage/GoogleDrive-bterhunecotter@sdsu.edu/My
Drive/IAM3_R/15th_Tidy_&_Manage_Data"
```

Which folder is my working directory?

- The answer depends on how you run your code and how you opened the session:
  - If you run your code directly, and…
    - if you opened an .Rproj, then **it is that .Rproj's folder**
    - if you opened a new session of R or opened a .R file directly, then **it is your home folder** 😢
  - If you run the code in a script by using `source()`, then **it is that script's folder ¶**

> If you are just starting out with R, just save your .Rproj and all .R scripts to one folder. That folder will always be your working directory if you open R via your .Rproj :)
>
> Don't worry about using `source()` at this point!

Writing Local Paths

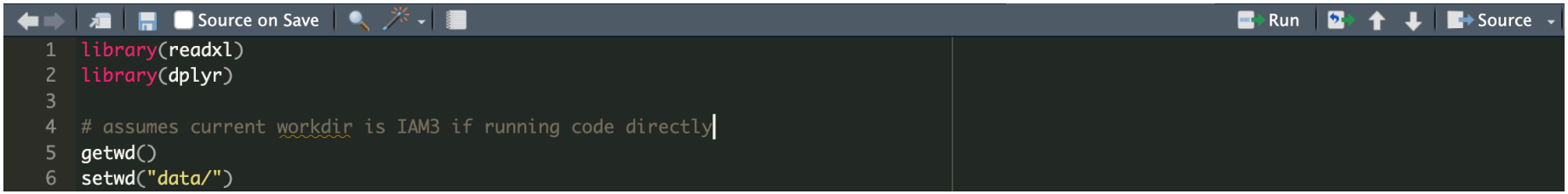
- project_folder
  - data
    - data.xlsx
  - analysis
    - project_analysis.R
    - project.Rproj

In the script `project_analysis.R`, opened in `project.Rproj`, how would you open `data.xlsx`?

```
1 open_xlsx("../data/data.xlsx")
```

The working directory would be `project_folder/analysis`, so you would go *up* one folder then back down into the data folder.

Source



- Sourcing code matters when you have multiple scripts that work with each other
- 99% of the time, I run my code directly, so the working directory is the .Rproj folder
- However, if I want to call code from another script, I *source* it and that code is run using the source folder as its working directory
- This matters if you want to ***define your own functions***! ¶

```
source("scripts/my_functions.R")
```

Defining and Running Functions

Functions

- *Functions* are the backbone of R. Everything that looks like `this_thing()` is a function.
- Most functions you use are already defined in packages, but you can write your own!
- Functions have:
  - one, multiple, or no arguments
  - **ONE** output object
    - either the last output in the function or whatever object is in `return()`
  - Global and local environments
    - Objects created within a function exist *only* within the function ¶

Writing Functions

Writing functions is relatively simple in concept, but can and will get complicated!

```
vee = 2
bai = 5
functionalFunction <- function(item1,item2) {
  item1+1
  item2+1
  #return(item1)
}

functionalFunction(vee, bai)
```

```
[1] 6
```

```
functionalFunction(bai, vee)
```

```
[1] 3
```

```
functionalFunction(item1, item2)
```

```
Error in eval(expr, envir, enclos): object 'item1' not found
```

Calling Functions

- Calling a function is simple, and you've been doing it this whole time:
- When you call a function, you name the actual objects you want to use. The **values** of these objects are passed to the function.

```
# Perform function and print output
name_of_function(argument1, argument2, argument3, ...)

# Assign function output to object
objectName <- name_of_function(argument1, argument2, argument3, ...)
```

Think about what it means that the **values** of the objects are passed to the function.

*Hint: sometimes you want to call the **names** of the objects, not their values!*

How to Write Functions

- Writing functions can be hard at first but will help you *organize your code* and avoid **spaghetti code**
- Writing functions makes your code more **modular** and **organized**
  - Each function has *one output*, meaning it has *one purpose*
  - Forces you to organize your code in a meaningful way (purpose by purpose)
  - Forces you to think about the main purpose of the code you are writing
- Each function can take a number of *arguments* which are environmental objects it will use and manipulate
  - When you write the function, you give the arguments any name you want
  - When you call the function, you write the actual names of the objects you want the function to use ¶

How to Write Functions

- Suppose you need to create identical plots for two datasets.
- You could write separate ggplots for each dataset and work on each of them manually
- OR you could write **one function** to create the ggplot, with any unique features entered as arguments!

subjid	test1_t1	test1_t2
iam3_001	19	20
iam3_002	18	18
iam3_003	19	17
iam3_004	17	22
iam3_005	19	21
iam3_006	17	18

How to Write Functions

These two chunks of code do the same thing:

```
library(ggplot2)
ggplot(data = mock_data,
       mapping = aes(x = test1_t1,
                     y = test1_t2)) +
  geom_point() +
  theme_classic() +
  labs(x = "Test Time 1",
       y = "Test Time 2",
       title = "Dataset 2")

ggplot(data = new_data,
       mapping = aes(x = test1_t1,
                     y = test1_t2)) +
  geom_point() +
  theme_classic() +
  labs(x = "Test Time 1",
       y = "Test Time 2",
       title = "Dataset 1")
```

```
my_plotting_function <-
  function(dataset, plot_title) {
  ggplot(data = dataset,
       mapping = aes(x = test1_t1,
                     y = test1_t2)) +
  geom_point() +
  theme_classic() +
  labs(x = "Test Time 1",
       y = "Test Time 2",
       title = plot_title)
}

library(ggplot2)
my_plotting_function(mock_data, "Dataset 1")
my_plotting_function(new_data, "Dataset 2")
```

Using ChatGPT to Write Functions

- If you know how to write functions, it's really easy to ask ChatGPT to write them for you.
- Functions have a limited set of inputs and a single output, and the function environment is closed, so whatever happens in a function is limited to that function.
  - Objects that are defined and used in a function won't affect the rest of your script
  - ChatGPT doesn't need to know what's in your global environment to write the function for you! ¶

Using ChatGPT to Write Functions

- Suppose I want to create a function where I can input `x_var` and `y_var` as arguments so I can create identical plots for different variables.
  - (this can be really helpful - think about plotting *accuracy* and *reaction time*)
- I just asked ChatGPT to write this function for me because I am a 🦥! ¶

Chatting with a LLM!

TE

Put this in an r function called my_function where age is x_var and score is y_var:

```
ggplot(data = mock_data,
    mapping = aes(x = age, y = score)) +
 geom_line()
```

Here is your request wrapped in an R function:

R Copy code

```r
my_function <- function(data, x_var, y_var) {
  # Load required library
  library(ggplot2)

  # Define aesthetic mapping using substitute to pass variable names correct
  aes_mapping <- aes_string(x = substitute(x_var), y = substitute(y_var))

  # Create and return the ggplot
  ggplot(data = data, mapping = aes_mapping) +
    geom_line()
}
```

Chatting with an LLM!

TE

For the above, why do you have to use substitute to pass variable names correctly?

When you pass variables as arguments to a function in R, the function doesn't receive the names of the variables you passed. Instead, it receives their values. This usually works fine, but it can cause problems when you're trying to use these variables in certain functions like `aes` in `ggplot2` that expect the names of variables, not their values.

The `aes_string` function in `ggplot2` allows you to pass variable names as strings, which can be useful in a function like this. However, if you were to pass `x_var` and `y_var` directly to `aes_string` without using `substitute`, it would treat "x_var" and "y_var" as the literal variable names it should look for in the data, not the values of these variables.

`substitute` is a base R function that allows you to capture the expression passed to the function, as it was typed, rather than its value. In this case, `substitute` is capturing the variable names passed to `my_function` as strings, which can then be correctly passed to `aes_string`.

Demanding more from an LLM!

TE

Write me a function that takes two dataframes as input, called lbgHeadliner and namesIDs. The dataframes each have the following variable names: llcn_id, name, name_first, and name_last. The function's ultimate goal is to use the information in namesIDs to give llcn_id to all the rows possible in lbgHeadliner. It should do the following, all within lbgHeadliner; namesIDs are used for reference only. These should be executed in order using if_else, so if a row matches a condition and is given an llcn_id, it is then left alone.

- Leave all llcn_ids that already exist alone.
- If an lbgHeadliner row has a name which matches an llcn_id in namesIDs, add the llcn_id in namesIDs to lbgHeadliner.
- If an lbgHeadliner row has name_first and name_last which both match a row in namesIDs, add that llcn_id to lbgHeadliner.
- If an lbgHeadliner row has name_last which matches a row AND the first four letters of name_first match, add that llcn_id to lbgHeadliner.
- Else, do nothing.

Here's a function which accomplishes the above requirements using a series of `` `if_else()` `` calls inside a `` `mutate()` ``:

Bullying an LLM!

TE

This code didn't work. It caused the following error: Error in `mutate()`:
! Problem while computing `result = map(.x = df$date_column, .f = ~standardize_dates(.x))`.
✖ `result` must be size 9 or 1, not 0.

I apologize for the confusion. The error message indicates that the **`map`** function doesn't get the expected output from **`standardize_dates`** function, possibly because the function is designed to operate on a vector of dates rather than individual dates.

TE

The above almost works, but now every value in each of the output columns is the entire vector.

I see. I apologize for misunderstanding your needs.

Sourcing Scripts

Separate Scripts

- If you have scripts to *define* functions, and your headliner/master script *runs* these functions, then your headliner script becomes really short and clear!
  - In your headliner, you *source* that file using `source()`, and the functions defined in that script appear in your environment 😎
- I'll show you something I wrote for a recent project that I'm proud of 😀 ¶

OYOLab

- Look at your project(s)' folder structure. Can you make them clearer or write better documentation?
- Try writing your own functions
- Or practice other concepts you've learned in this workshop