Project organization

Hitchhiker's Guide to Reproducible Research

Julia Wrobel and David Benkeser

Course Website

Basic principles

- Put everything in one version-controlled directory.
- Develop your own system.
- Be consistent, but look for ways to improve.
  - naming conventions, file structure
- Raw data are sacred. Keep them separate from everything else.
- Separate code and data.
- Use meaningful file names.
- Use YYYY-MM-DD date formatting.
- No absolute paths.

What to organize?

It is probably useful to have a system for organizing:

- data analysis projects;
- first-author papers;
- talks.

The systems should adhere to the same general principles, but different requirements may necessitate different structures.

Think about organization of a project from the outset!

Collaborative projects

Collaborative projects present a greater challenge.

- Not everyone is comfortable with LaTeX or git or ...

I don't have a great solution for this.

- Google drive/Word online helps to a certain extent, but you lose in other areas (reference management, math typesetting)
- Overleaf has gotten much better for LaTeX

Some advice:

- Address organization from the outset.
- Ideally, bring people on board to your (version controlled, reproducible) system.
- Keep open lines of communication (especially if using GitHub)

Example data analysis project (Julia)

```
YYYY_MM_PI_topic/
    data/
      data/raw_data.csv
      data/tidied_data.csv
    analysis/
      analysis/exploratory_data_analysis.Rmd
      analysis/modeling.Rmd
      analysis/manuscript_figures.Rmd
      analysis/report.Rmd
    source/
      source/clean_raw_data.R
      source/modeling_functions.R
    results/
    literature/
    README.md
```

Example data analysis project, cont'd (Julia)

I typically have other ancillary files in my root directory as well. These are files I don't (often) modify but are important for workflow or reproducibility:

```
YYYY_MM_PI_topic/
    YYYY_MM_PI_topic.Rproj
    .git
    .gitignore
```

Example data analysis project (David)

```
analysis/
    raw_data/
    data/
    R/
      R/00_clean_data.R
      R/01_fit_models.R
      R/02_make_figures.R
      R/03_summarize_results.R
      R/04_report.Rmd
    figs/
    sandbox/
      sandbox/exploratory.R
    ref_papers/
    Makefile
    README.md
    renv
```

Example paper organization

```
paper/
    analysis/
      analysis/README.md
      analysis/00_clean_data.R
      analysis/01_fit_models.R
      analysis/02_make_figures.R
      analysis/sandbox
    sim/
      sim/README.md
      sim/helper_functions.R
      sim/sim_script.R
      sim/run_sim_script.sh
    figs/
    notes/
    submitted/
    revision/
    final/
    README.md
```

Organizing data

Raw data are sacred... but may be a mess.

- You'll be surprised (and disheartened) by how many color-coded excel sheets you'll get in your life.

Tempting to edit raw data by hand. Don't!

- Everything scripted!

Use meta-data files to describe raw and cleaned data.

- structure as data (e.g., `.csv` so easy to read)

Organizing data

Hadley Wickham defined the notion of tidy data.

- Each variable forms a column.
- Each observation forms a row.
- Each observational unit forms a table.

ptid	day	age	drug	out
1	1	28	0	0
1	2	28	0	1
2	1	65	0	0
2	2	65	1	1
3	1	34	0	0
3	2	34	-	1

Exploring data

One of the first things we'll often do is open the data and start poking around.

- Could be informal, "getting to know you."
- Could be more formal, "see if anything looks interesting."

This is often done in an ad-hoc way:

- entering commands directly into R;
- making and saving plots "by hand";
- etc...

Slow down and document.

- Your future self will thank you!

Exploring data

Write out a set of comments describing what you are try to accomplish and fill in code from there.

- I do this for every coding project.
    - Data analysis, methods coding, package development

Leave a search-able comment tag by code to return to later

- I use e.g., `# TO DO: add math expression to labels; make colors prettier`.

Sets "the bones" of a formal analysis in place while allowing for some creative flow.

Exploring data

Other helpful ideas for formalizing exploratory data analysis:

- Informal `.Rmd` documents.
  - easy way to organize code/comments into readable format
- `.Rhistory` files
  - all the commands used in an R session
- `save` intermediate objects and workspaces
  - and document what they contain!
- `knitr::spin`
  - writing `.R` scripts with rendered-able comments

Automated project initiation using `projectr`

The `projectr` package sets my preferred directory structure for a new project.

- Borrowed *heavily* from `jeff-goldsmith/projectr`.

```
devtools::install_github("julia-wrobel/projectr")

projectr::proj_start(proj_dir = "~/projects/2024/202407_PERSON_PROJECT",
                     data_dir = "~/Data/202407_data")
```

proj_dir: where on your computer you want the project to live

- *~Documents/projects/2024/* is where I store new projects
- Whatever you decide, be consistent!

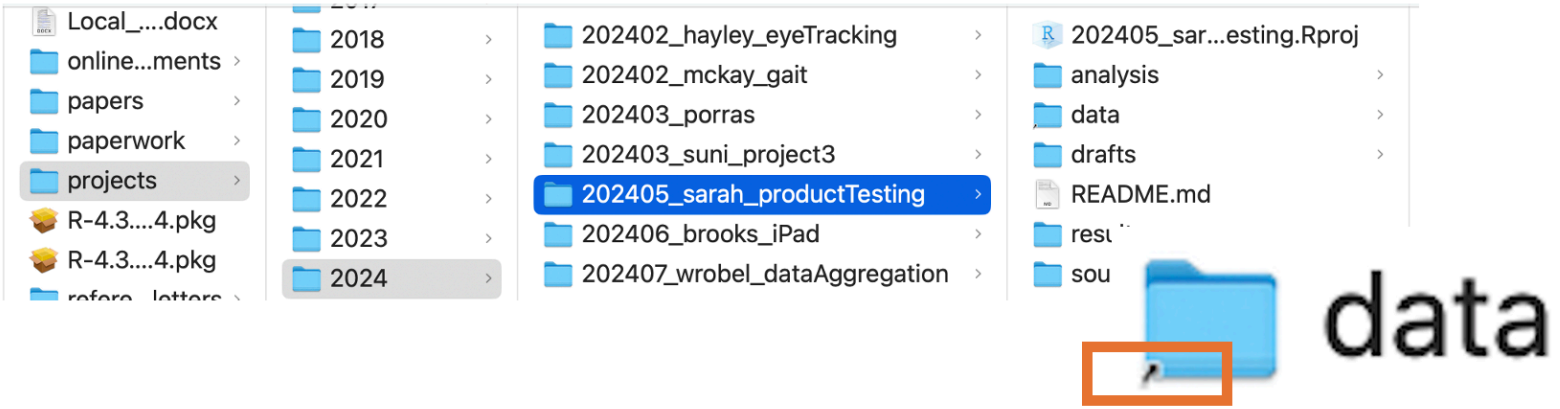
data_dir: where the data to live, if not within your project folder

- Sets up a symbolic link from the project directory to this folder

Symbolic links

There are great reasons NOT to put your raw data in the project folder

- iOS uploads many folders automatically to the cloud
- if you want to put the project on GitHub, you might want to exclude the data
- maybe data is stored on OneDrive/Box/AWS and you don't want to download a local copy



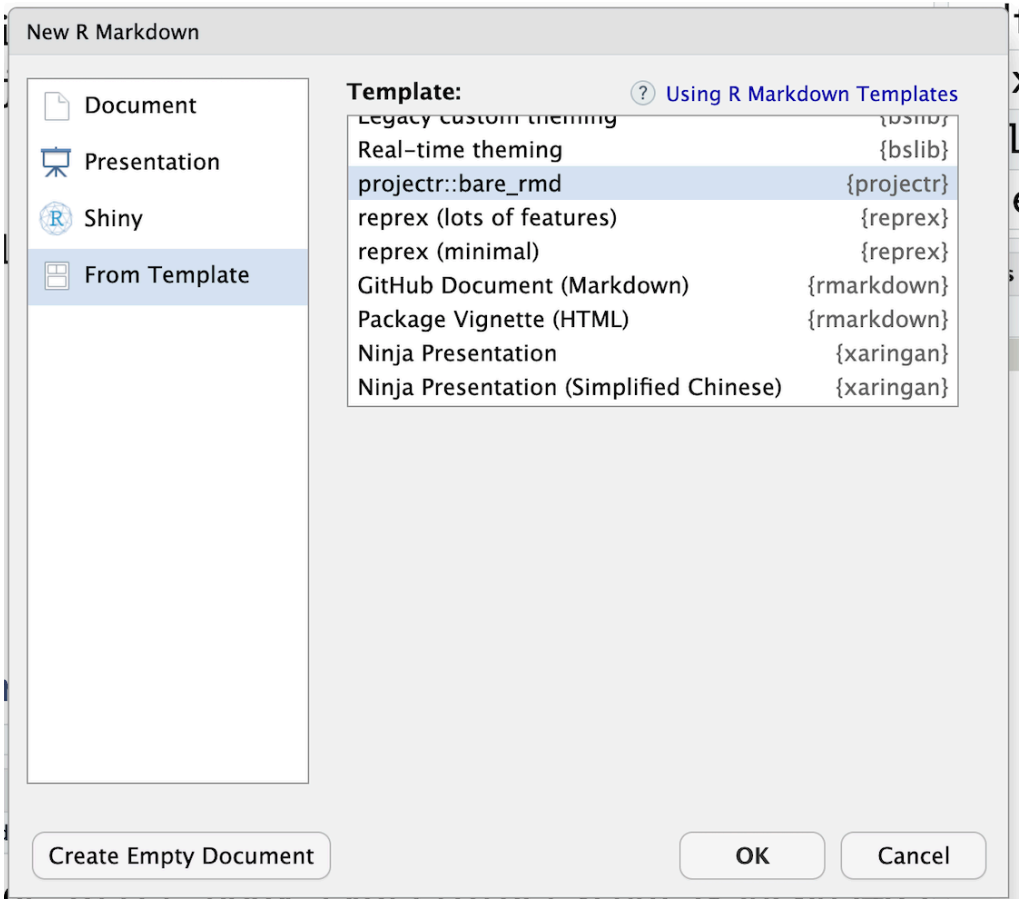
Setting up a symbolic link in the terminal

Pseudo code for setting up a symbolic link:

```
ln -s /path/to/target /path/to/symlink
```

Built-in `projectr` Rmd template

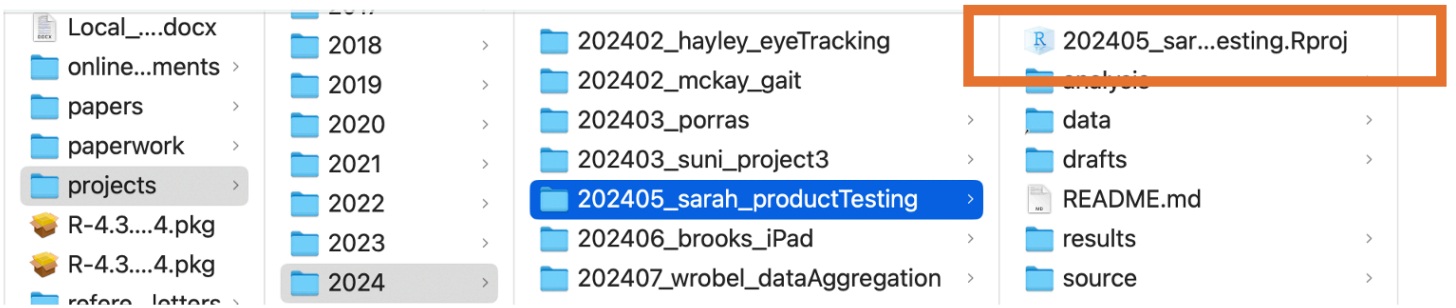
In Rstudio, click `File > New File > Rmarkdown`



.Rproj files

You may have noticed a file with the extension `.Rproj` in the productTesting folder

- These are called `R projects`
- `projectr::proj_start()` automatically sets up an `.Rproj`.



I'm going to try to convince you that these are the best.

Benefits of using R projects

Project organization:

- Relative file paths: ensures file paths are relative to the project directory, making scripts portable and easier to share.
- Separate workspaces: prevents conflicts between variables and packages across different projects.

Reproducibility

- Can hand off entire directory to someone else and have them rerun your analysis
- Works great with the `here` package

Double clicking the `202405_sarah_productTesting.Rproj` opens up an R Studio session and automatically sets your working directory to the `202405_sarah_productTesting` folder.

Example 1

We will walk through the following tasks together. We will be using this folder for the rest of the course so please set up your own folder as we go, and ask questions if you get lost!

1. Use `projectr::proj_start()` to initiate a new project called `20240722_sismid_repro`. Set up a directory in a separate location using the argument `data_dir`.
2. Download the data download script and save it in the `source` folder of your new project directory as `01_data_download.R`.
3. Make an Rmarkdown document that knits to html called `final_report.Rmd` and put it in the `analysis` folder of your project directory.

Example 1 cont'd

The `01_data_download.R` script should look like this:

```
library("RSocrata")
library(tidyverse)

# download longitudinal Covid WW concentration data from API
covid <- read.socrata(
  "https://data.cdc.gov/resource/g653-rqe2.json",
  app_token = Sys.getenv("TOKEN"),
  email     = Sys.getenv("EMAIL"),
  password  = Sys.getenv("PASSWORD")
) %>%
  mutate(date_downloaded = Sys.Date())

# download cross-sectional Covid WW concentration data from API, which w
counties <- read.socrata(
  "https://data.cdc.gov/resource/2ew6-ywp6.json",
  app_token = Sys.getenv("TOKEN"),
  email     = Sys.getenv("EMAIL"),
  password  = Sys.getenv("PASSWORD")
)
```

.Renviron files

In the previous slide you may have noticed code you may not be familiar with:

```
counties <- read.socrata(
  "https://data.cdc.gov/resource/2ew6-ywp6.json",
  app_token = Sys.getenv("TOKEN"),
  email     = Sys.getenv("EMAIL"),
  password  = Sys.getenv("PASSWORD")
)
```

.Renviron files set environment variables in R that you might not want to hard code into your scripts (e.g. API keys, passwords).

- Store in root directory of your project
- Syntax is `variable_name = "variable value"`, e.g. `TOKEN = "12345abc"`.
    - Access this variable using `Sys.getenv("TOKEN")`.

.gitignore files

A `.gitignore` file in Git is used to specify which files and directories should be ignored by Git when you make changes to a repository. This helps prevent unnecessary or sensitive files from being tracked and committed.

- `projectr::proj_start()` automatically creates a `.gitignore` file and puts it in your root directory
- ALWAYS put `.Renviron` in the `.gitignore` file if you want your API token and password to stay private!

David will go over `.gitignore` files in more detail later when he talks about **git** and **GitHub**.

Try it!

Let's walk through `Example 2`, which adds an .Renviron file.

The `here` package

No absolute paths.

- Absolute paths are the enemy of project reproducibility.

For `R` projects, the `here` package provides a simple way to use relative file paths.

- Read Jenny Bryan and James Hester's chapter on project-oriented work-flows.

The use of `here` is simple and best illustrated by example.

The `here` package

Consider this simple project structure.

```
my_project/
  my_project.Rproj
  data/
    my_data.csv
  output/
  R/
    R/my_analysis.R
  Rmd/
    Rmd/my_report.Rmd
```

Here, the folder `my_project` is the **root directory**.

- Where `.Rproj` lives
- All file paths should be relative to `my_project`!

The `here` package

Makes it easy to load data using a relative file path that works across different operating systems:

```
library(here)
# relative path using here()
here_path = here("data", "file_i_want.csv")
my_data = read.csv(here_path)
```

In contrast to:

```
# absolute path
ugly_path = "/Users/JWROB/projects/my_project/data/file_i_want.csv"
my_data = read.csv(ugly_path)
```

In contrast to:

```
# relative path using NOT using here()
relative_path = "./data/file_i_want.csv"
my_data = read.csv(relative_path)
```

The `here` package

`here` works, regardless of where the associated source file lives inside your project

- If you have an `.Rproj` file in your root directory of your project, here will set the location of the `.Rproj` to be the top-level directory
    - This is the behavior we want!
- These paths will "just work" during interactive development, without incessant fiddling with the working directory of your IDE's R process.
- I am oversimplifying the heuristics, feel free to read more.

The `here` package

What if I want to load data in a document that lives in a subfolder such as `my_project/analysis/code.Rmd`?

- Doesn't matter! You can use the same code within the `.Rmd` document to load the data

```
library(here)
path_to_data = here("data", "file_i_want.csv")
my_data = read.csv(path_to_data)
```

What if my data I want to access is nested in a subfolder of data, such as `my_project\data\raw_data\raw_file.csv`?

```
library(here)
path_to_data = here("data", "raw_data", "raw_file.csv")
my_data = read.csv(path_to_data)
```

The `here` package

If for some reason you don't use R projects (even though you should), you can still benefit from the `here` package.

Each `R` script or `Rmd` report, should contain a call to `here::i_am('path/to/this/file')` at the top.

- `path/to/this/file` should be replaced with the path relative to the project's **root directory**.
- `here::i_am` means use function `i_am` from `here` package.

For example, the file `R/my_analysis.R` might look like this.

```
# include at top of script
here::i_am('R/my_analysis.R')

# now add all your great R code...
```

Starting a new analysis

Once I've received data and decided to start an analysis, I'll typically follow these steps first:

1. Set up a new project directory using `projectr::proj_start()`
2. Open and do very basic exploration of the raw data
    - How many rows and columns do I have?
    - Is the data in the format I need for analysis?
3. Make sure I understand the columns in my data.
    - If a data dictionary doesn't exist, I create one
4. Make a data cleaning file that reads in the raw data and outputs a tidied dataset
    - Typically reduces data to only information necessary for the planned analysis

Analysis of Covid WW concentration data

We are interesting in analyzing wastewater concentration of SARS-CoV-2 over time at the county level. We will start with only counties in Georgia.

Longitudinal data contains concentrations over time, and cross-sectional data contains information about county each data collection site is located in.

- We will need to merge these two datasets
- We also want to subset to collection sites in Georgia only

Data analysis

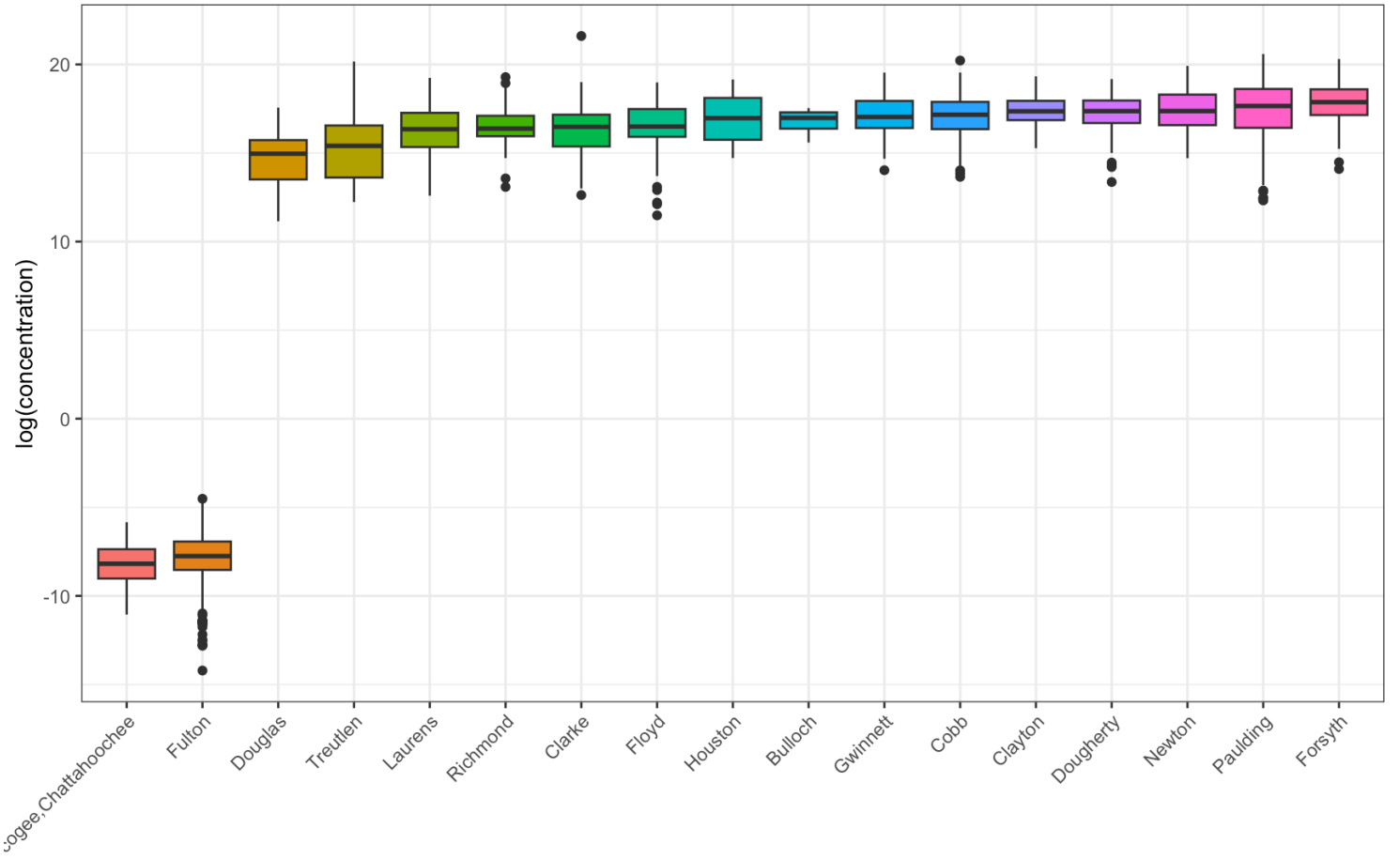
Using our cleaned data, we will calculate the median and interquartile range of the WW SARS-CoV-2 concentration by county.

```
library(tidyverse)

covid %>%
  group_by(county) %>%
  summarize(median = median(concentration),
            q25 = quantile(concentration, probs = .25),
            q75 = quantile(concentration, probs = .75),
            population_served = median(population_served)) %>%
  ungroup() %>%
  arrange(median)
```

Data visualization

Using our cleaned data, we will visualize the concentration by county and over time.



Example 3

We will walk through this together as well, using the `20240722_sismid_repro` project we set up already.

1. Using the `projectr` template, make an Rmarkdown document called `exploratory_analysis.Rmd` and put it in the `analysis` folder of your project directory. Load and explore the data. Take notes on what you learn. Add in brief descriptions of the key variables.
2. Download the data cleaning script and save it in the `source` folder of your new project directory as `02_data_cleaning.R`.
3. Download the data analysis script and save it in the `source` folder of your new project directory as `03_data_analysis.R`.
4. Download the data visualization script and save it in the `source` folder of your new project directory as `04_data_visualization.R`.
5. Open your `final_report.Rmd` document and source each of the scripts. Add comments to explain the document!

Pulling it all together

Knitting `final_report.Rmd` will ensure that if one step of the data analysis gets updated, it will be carried through the rest of the pipeline.

- Critical for reproducibility because a common error is to edit one piece of the code but not have changes follow through to the end of analysis

make: an alternative option, a command-line tool that automatically builds and compiles code by following instructions in a **Makefile**.

Parameterized reports

So far we have focused on analysis of counties in Georgia. What if we wanted to reproduce this analysis for any state in the US?

Parameterized reports in R markdown allow you to create a report template that can be reused across multiple similar scenarios. Examples include:

- Running a report that covers a specific time period
- Showing results for a specific geographic location

Declaring parameters

Parameters are specified using the `params` field within the YAML header of the R Markdown document. We can specify one or more parameters with each item on a new line:

```
---
title: My Document
output: html_document
params:
  year: 2024
  state: "ga"
  printcode: TRUE
---
```

It's worth noting that all standard R types that can be parsed by `yaml::yaml.load()` can be included as parameters, including `character`, `numeric`, `integer`, and `logical` types.

Using parameters

You can access the parameters within the knitting environment and the R console

- The values are contained within a list called `params`:
  - `params$year`
  - `params$state`

Parameters can also be used to control the behavior of `knitr`:

````
---
params:
  printcode: false  # or set it to true
---

```{r, setup, include=FALSE}
# set this option in the first code chunk in the document
knitr::opts_chunk$set(echo = params$printcode)
```
````

Knitting with parameters

There are a few ways in which a parameterized report can be knitted:

1. Using the `knit` button in R Studio. The default values listed in the YAML will be used.
2. `rmarkdown::render()` with the `params` argument. Allows you to override the default values listed in the YAML.

```
rmarkdown::render("MyDocument.Rmd", params = list(
  year = 2022,
  state = "nj",
  printcode = FALSE,
))
```

You don't have to explicitly state all parameters in the `params` argument. Any parameters not specified will default to the values in the YAML header.

Rendering parameterized reports

You can even integrate these into a function that can be used to create an output file with a different filename for each combination of parameters!

```
render_report = function(state, year) {
  rmarkdown::render(
    "MyDocument.Rmd", params = list(
      region = region,
      year = year
    ),
    output_file = paste0("Report-", region, "-", year, ".html")
  )
}
```

Try it: parameterized reports

In groups, try Exercises 1 and 2.