Best Practices in Computing

Francisco Villamil

Applied Quantitative Methods II

MA in Social Sciences, Spring 2026

Today's goals

- Understand why computing practices matter for research

Today's goals

- Understand why computing practices matter for research
- Learn to organize a research project with a clear folder structure

Today's goals

- Understand why computing practices matter for research
- Learn to organize a research project with a clear folder structure
- Write better R code: functions, checks, and style

Today's goals

- Understand why computing practices matter for research
- Learn to organize a research project with a clear folder structure
- Write better R code: functions, checks, and style
- Understand the role of plain text and command line tools

Today's goals

- Understand why computing practices matter for research
- Learn to organize a research project with a clear folder structure
- Write better R code: functions, checks, and style
- Understand the role of plain text and command line tools
- Deepen your knowledge of version control with Git

Roadmap

Introduction

Project Organization

Writing Better Code

Plain Text and Tools

Version Control with Git

Wrap-up

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes
  - → Copy-pasting results into Word

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes
  - → Copy-pasting results into Word
  - → Renaming files by hand

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes
  - $\rightarrow$ Copy-pasting results into Word
  - $\rightarrow$ Renaming files by hand
  - $\rightarrow$ Running scripts in the wrong order

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes
  - → Copy-pasting results into Word
  - → Renaming files by hand
  - → Running scripts in the wrong order
- **Efficiency**: time invested in workflow saves time later

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes
  - → Copy-pasting results into Word
  - → Renaming files by hand
  - → Running scripts in the wrong order
- **Efficiency**: time invested in workflow saves time later
- On errors: Bisbee et al. (2022)

You come back to a project after 6 months.

You need to update one figure.

How long does it take you?

Roadmap

Introduction

Project Organization

Writing Better Code

Plain Text and Tools

Version Control with Git

Wrap-up

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  → Data collection / cleaning

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures
  - → Final document (paper, slides)

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures
  - → Final document (paper, slides)
- Each folder has:

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures
  - → Final document (paper, slides)
- Each folder has:
  - → A script (e.g., `analyze.R`)

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures
  - → Final document (paper, slides)
- Each folder has:
  - → A script (e.g., `analyze.R`)
  - → An `output/` subfolder for results

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures
  - → Final document (paper, slides)
- Each folder has:
  - → A script (e.g., `analyze.R`)
  - → An `output/` subfolder for results
- **Never** one giant script that does everything

Example: workflow_example project

workflow_example/

- Full example: github.com/franvillamil/workflow_example

File naming conventions

Bad	Good
`Final Data.csv`	`data_cleaned.csv`
`Datos educación.csv`	`datos_educacion.csv`
`analysis.R` (which one?)	`01_clean_data.R`
`figure1final2.pdf`	`fig_scatter_income.pdf`
`My Thesis Draft (3).docx`	`thesis_draft.tex`

- **No spaces** in file names (use underscores or hyphens)

File naming conventions

Bad	Good
`Final Data.csv`	`data_cleaned.csv`
`Datos educación.csv`	`datos_educacion.csv`
`analysis.R` (which one?)	`01_clean_data.R`
`figure1final2.pdf`	`fig_scatter_income.pdf`
`My Thesis Draft (3).docx`	`thesis_draft.tex`

- **No spaces** in file names (use underscores or hyphens)
- **No special characters** (accents, symbols)

File naming conventions

Bad	Good
`Final Data.csv`	`data_cleaned.csv`
`Datos educación.csv`	`datos_educacion.csv`
`analysis.R` (which one?)	`01_clean_data.R`
`figure1final2.pdf`	`fig_scatter_income.pdf`
`My Thesis Draft (3).docx`	`thesis_draft.tex`

- **No spaces** in file names (use underscores or hyphens)
- **No special characters** (accents, symbols)
- Use **numbered prefixes** for scripts that run in order

File naming conventions

Bad	Good
`Final Data.csv`	`data_cleaned.csv`
`Datos educación.csv`	`datos_educacion.csv`
`analysis.R` (which one?)	`01_clean_data.R`
`figure1final2.pdf`	`fig_scatter_income.pdf`
`My Thesis Draft (3).docx`	`thesis_draft.tex`

- **No spaces** in file names (use underscores or hyphens)
- **No special characters** (accents, symbols)
- Use **numbered prefixes** for scripts that run in order
- Use **descriptive names**: what is in the file, not when you made it

Integrating R and $\LaTeX$

- R produces output files (tables, figures)

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  → `analyze.R` → `analyses/output/table_models.tex`

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`
  - → `doc.tex`: `\input{../analyses/output/table_models}`

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`
  - → `doc.tex`: `\input{../analyses/output/table_models}`
  - → `doc.tex`: `\includegraphics{../plots/output/scatter}`

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`
  - → `doc.tex`: `\input{../analyses/output/table_models}`
  - → `doc.tex`: `\includegraphics{../plots/output/scatter}`
- **No copy-pasting**: update the analysis, recompile the document

Integrating R and LaTeX

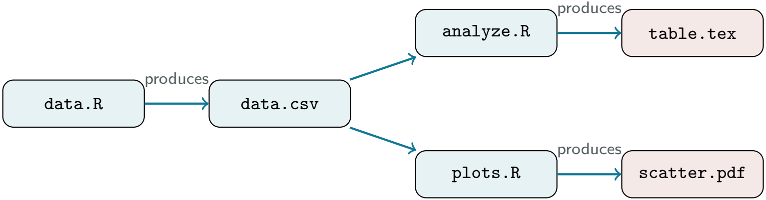
- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`
  - → `doc.tex`: `\input{../analyses/output/table_models}`
  - → `doc.tex`: `\includegraphics{../plots/output/scatter}`
- **No copy-pasting**: update the analysis, recompile the document
- The document always reflects the latest results

Automating the pipeline: Makefiles

```
all:  create_data/output/data.csv analyses/output/table_models.tex \
    plots/output/scatter.pdf
create_data/output/data.csv:  create_data/data.R
    Rscript --no-save create_data/data.R
analyses/output/table_models.tex:  analyses/analyze.R \
        create_data/output/data.csv
    Rscript --no-save analyses/analyze.R
plots/output/scatter.pdf:  plots/plots.R \
        create_data/output/data.csv
    Rscript --no-save plots/plots.R
```

Note: Makefiles require **tabs** (not spaces) for indentation

Makefiles: the logic



- `make` only re-runs what has **changed**
- The dependency graph ensures correct ordering

Roadmap

Introduction

Project Organization

Writing Better Code

Plain Text and Tools

Version Control with Git

Wrap-up

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug

DRY: Don’t Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug
  - → Shorter, cleaner scripts

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug
  - → Shorter, cleaner scripts
- Common candidates for functions:

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug
  - → Shorter, cleaner scripts
- Common candidates for functions:
  - → Cleaning steps applied to multiple datasets

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug
  - → Shorter, cleaner scripts
- Common candidates for functions:
  - → Cleaning steps applied to multiple datasets
  - → Running the same model with different variables

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug
  - → Shorter, cleaner scripts
- Common candidates for functions:
  - → Cleaning steps applied to multiple datasets
  - → Running the same model with different variables
  - → Producing plots with consistent formatting

Define constants at the top

Bad	**Good**
`df = df[df$year >= 2000, ]`	`start_year = 2000`
`...`	`...`
`df2 = df2[df2$year >= 2000, ]`	`df = df[df$year >= start_year, ]`
	`df2 = df2[df2$year >= start_year, ]`

- Put parameters and thresholds at the **top of the script**

Define constants at the top

Bad	Good
`df = df[df$year >= 2000, ]`	`start_year = 2000`
`...`	`...`
`df2 = df2[df2$year >= 2000, ]`	`df = df[df$year >= start_year, ]`
	`df2 = df2[df2$year >= start_year, ]`

- Put parameters and thresholds at the **top of the script**
- Change once, applies everywhere

Define constants at the top

Bad	Good
`df = df[df$year >= 2000, ]`	`start_year = 2000`
`...`	`...`
`df2 = df2[df2$year >= 2000, ]`	`df = df[df$year >= start_year, ]`
	`df2 = df2[df2$year >= start_year, ]`

- Put parameters and thresholds at the **top of the script**
- Change once, applies everywhere
- From the example project:

Define constants at the top

Bad	Good
`df = df[df$year >= 2000, ]`	`start_year = 2000`
`...`	`...`
`df2 = df2[df2$year >= 2000, ]`	`df = df[df$year >= start_year, ]`
	`df2 = df2[df2$year >= start_year, ]`

- Put parameters and thresholds at the **top of the script**
- Change once, applies everywhere
- From the example project:
  → `n_obs = 1000`

Define constants at the top

Bad	**Good**
`df = df[df$year >= 2000, ]`	`start_year = 2000`
`...`	`...`
`df2 = df2[df2$year >= 2000, ]`	`df = df[df$year >= start_year, ]`
	`df2 = df2[df2$year >= start_year, ]`

- Put parameters and thresholds at the **top of the script**
- Change once, applies everywhere

- From the example project:
  - → `n_obs = 1000`
  - → Used throughout — change it once to re-run with different sample size

Write checks and assertions

```
# After merging two datasets
merged = merge(df1, df2, by = "id")
if(nrow(merged) != nrow(df1)) {
  stop("Merge changed number of rows!")
}

# Check for duplicates
if(any(duplicated(df$id))) {
  stop("Duplicate IDs found!")
}

# Sanity check on values
if(any(df$age < 0 | df$age > 120, na.rm = TRUE)) {
  warning("Suspicious age values detected")
}
```

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`
  - → `print(table(df$treatment))`

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`
  - → `print(table(df$treatment))`
  - → `cat("Missing values:", sum(is.na(df$outcome)), "\n")`

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`
  - → `print(table(df$treatment))`
  - → `cat("Missing values:", sum(is.na(df$outcome)), "\n")`
- When the script runs, you get a **log** of what happened

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`
  - → `print(table(df$treatment))`
  - → `cat("Missing values:", sum(is.na(df$outcome)), "\n")`
- When the script runs, you get a **log** of what happened
- If something goes wrong later, the log tells you where

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`
  - → `print(table(df$treatment))`
  - → `cat("Missing values:", sum(is.na(df$outcome)), "\n")`
- When the script runs, you get a **log** of what happened
- If something goes wrong later, the log tells you where
- This is especially valuable when running scripts via `Rscript` or `make`

Code style matters

- Use **meaningful variable names**:

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`

Code style matters

- Use **meaningful variable names**:
  → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it
  - → Consistent indentation (2 spaces in R)

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it
  - → Consistent indentation (2 spaces in R)
- Write **comments** for non-obvious code

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it
  - → Consistent indentation (2 spaces in R)
- Write **comments** for non-obvious code
  - → Why, not what: `# Drop obs before 1990 (pre-reform)`

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it
  - → Consistent indentation (2 spaces in R)
- Write **comments** for non-obvious code
  - → Why, not what: `# Drop obs before 1990 (pre-reform)`
  - → Use section dividers: `# ============`

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it
  - → Consistent indentation (2 spaces in R)
- Write **comments** for non-obvious code
  - → Why, not what: `# Drop obs before 1990 (pre-reform)`
  - → Use section dividers: `# ============`
- Your code is read more often than it is written

Roadmap

Introduction

Project Organization

Writing Better Code

Plain Text and Tools

Version Control with Git

Wrap-up

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever
  - → **Version control**: Git tracks changes line by line

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever
  - → **Version control**: Git tracks changes line by line
  - → **Scriptable**: can be processed by other tools

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever
  - → **Version control**: Git tracks changes line by line
  - → **Scriptable**: can be processed by other tools
  - → **No vendor lock-in**: does not require specific software

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever
  - → **Version control**: Git tracks changes line by line
  - → **Scriptable**: can be processed by other tools
  - → **No vendor lock-in**: does not require specific software
- You are already using plain text: R scripts, LaTeX files

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever
  - → **Version control**: Git tracks changes line by line
  - → **Scriptable**: can be processed by other tools
  - → **No vendor lock-in**: does not require specific software
- You are already using plain text: R scripts, LaTeX files
- The goal: use it for **as much as possible**

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python
  - → **Sublime Text**: fast, lightweight, highly customizable

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python
  - → **Sublime Text**: fast, lightweight, highly customizable
- Why use a general editor?

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python
  - → **Sublime Text**: fast, lightweight, highly customizable
- Why use a general editor?
  - → Edit R, LaTeX, Python, Makefiles in the same tool

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python
  - → **Sublime Text**: fast, lightweight, highly customizable
- Why use a general editor?
  - → Edit R, LaTeX, Python, Makefiles in the same tool
  - → Better project navigation and search

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python
  - → **Sublime Text**: fast, lightweight, highly customizable
- Why use a general editor?
  - → Edit R, LaTeX, Python, Makefiles in the same tool
  - → Better project navigation and search
  - → Customizable snippets and shortcuts

The command line: basics

- The command line is how you talk directly to your computer

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory
  - → `mkdir` — create a directory

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory
  - → `mkdir` — create a directory
  - → `Rscript file.R` — run an R script

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory
  - → `mkdir` — create a directory
  - → `Rscript file.R` — run an R script
  - → `make` — run a Makefile

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory
  - → `mkdir` — create a directory
  - → `Rscript file.R` — run an R script
  - → `make` — run a Makefile
- You **already use it** for Git (`git add`, `git commit`, `git push`)

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory
  - → `mkdir` — create a directory
  - → `Rscript file.R` — run an R script
  - → `make` — run a Makefile
- You **already use it** for Git (`git add`, `git commit`, `git push`)
- Terminal on Mac/Linux, Git Bash or WSL on Windows

How do you currently organize your R projects?

One script or many? How do you keep track of output?

Roadmap

- Introduction
- Project Organization
- Writing Better Code
- Plain Text and Tools
- Version Control with Git
- Wrap-up

Git: recap from Session 1

- You have been using Git since the first session

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes
  - → `git commit -m "message"` — save a snapshot

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes
  - → `git commit -m "message"` — save a snapshot
  - → `git push` — upload to GitHub

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes
  - → `git commit -m "message"` — save a snapshot
  - → `git push` — upload to GitHub
  - → `git pull` — download from GitHub

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes
  - → `git commit -m "message"` — save a snapshot
  - → `git push` — upload to GitHub
  - → `git pull` — download from GitHub
- Today: going deeper into **good practices**

The `.gitignore` file

Track (source files):

- R scripts (`.R`)
- LaTeX source (`.tex`)
- Makefiles
- Documentation (`.md`)
- Small data files (`.csv`)

Do NOT track:

- Generated output (`.pdf`, `.png`)
- LaTeX auxiliary files (`.aux`, `.log`)
- Large data files
- System files (`.DS_Store`)
- Sensitive data

```
.gitignore:
*.pdf
*.aux
*.log
.DS_Store
output/
```

Good commit habits

- Commit **often**, in small logical units

Good commit habits

- Commit **often**, in small logical units

  $\rightarrow$ Not: one commit per day with everything

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:
  - → Bad: `"update"`, `"changes"`, `"asdf"`

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:
  - → Bad: `"update"`, `"changes"`, `"asdf"`
  - → Good: `"Add robust SE to main models"`

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:
  - → Bad: `"update"`, `"changes"`, `"asdf"`
  - → Good: `"Add robust SE to main models"`
  - → Good: `"Fix merge bug in cleaning script"`

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:
  - → Bad: `"update"`, `"changes"`, `"asdf"`
  - → Good: `"Add robust SE to main models"`
  - → Good: `"Fix merge bug in cleaning script"`
- Your commit history is a **lab notebook**

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:
  - → Bad: `"update"`, `"changes"`, `"asdf"`
  - → Good: `"Add robust SE to main models"`
  - → Good: `"Fix merge bug in cleaning script"`
- Your commit history is a **lab notebook**
  - → You should be able to reconstruct what you did and why

GitHub for replication materials

- Journals increasingly require **replication packages**

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results
  - → Data (if shareable) or instructions to obtain it

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results
  - → Data (if shareable) or instructions to obtain it
- Good examples in political science:

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results
  - → Data (if shareable) or instructions to obtain it
- Good examples in political science:
  - → Most published papers now have a GitHub repo

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results
  - → Data (if shareable) or instructions to obtain it
- Good examples in political science:
  - → Most published papers now have a GitHub repo
  - → Look at how others structure their replication packages

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results
  - → Data (if shareable) or instructions to obtain it
- Good examples in political science:
  - → Most published papers now have a GitHub repo
  - → Look at how others structure their replication packages
- Your `workflow_example`-style project is already a replication package

Collaboration with Git

- Git was designed for collaboration

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer
  - → `push` and `pull` to sync

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer
  - → `push` and `pull` to sync
- **Merge conflicts**: when two people edit the same line

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer
  - → `push` and `pull` to sync
- **Merge conflicts**: when two people edit the same line
  - → Git asks you to resolve manually

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer
  - → `push` and `pull` to sync
- **Merge conflicts**: when two people edit the same line
  - → Git asks you to resolve manually
  - → Rare if you communicate and divide tasks well

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer
  - → `push` and `pull` to sync
- **Merge conflicts**: when two people edit the same line
  - → Git asks you to resolve manually
  - → Rare if you communicate and divide tasks well
- Even for solo work: syncing between laptop and desktop

Roadmap

Introduction

Project Organization

Writing Better Code

Plain Text and Tools

Version Control with Git

Wrap-up

Key resources

- Kieran Healy, *The Plain Person's Guide to Plain Text Social Science*:

  → `plain-text.co`

- MIT, *The Missing Semester of Your CS Education*:

  → `missing.csail.mit.edu`

- Software Carpentry lessons (Unix Shell, Git):

  → `software-carpentry.org/lessons`

- Bruno Rodrigues, *Building Reproducible Analytical Pipelines with R*:

  → `raps-with-r.dev`

- Example project: `github.com/franvillamil/workflow_example`

Summary

- **Organize**: separate tasks into folders, each with script + output

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents
- **Code well**: DRY, constants at the top, checks and assertions

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents

- **Code well**: DRY, constants at the top, checks and assertions
- **Print diagnostics**: log what your scripts do

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents
- **Code well**: DRY, constants at the top, checks and assertions
- **Print diagnostics**: log what your scripts do
- **Use plain text**: portable, versionable, scriptable

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents

- **Code well**: DRY, constants at the top, checks and assertions
- **Print diagnostics**: log what your scripts do

- **Use plain text**: portable, versionable, scriptable
- **Use Git**: commit often, write good messages, share via GitHub

For next week

- Complete Assignment 5
- Next session: Panel Data
  - → Fixed effects and within-group variation
  - → The `fixest` package in R

Questions?