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 principles for organizing a research project

Today's goals

- Understand why computing practices matter for research
- Learn principles for organizing a research project
- Integrate R output directly into LaTeX documents

Today's goals

- Understand why computing practices matter for research
- Learn principles for organizing a research project
- Integrate R output directly into LaTeX documents
- Write better R code: functions, checks, and style

Today's goals

- Understand why computing practices matter for research
- Learn principles for organizing a research project
- Integrate R output directly into LaTeX documents
- 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 principles for organizing a research project
- Integrate R output directly into LaTeX documents
- 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

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil
  - → Something to get through to reach results

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil
  - → Something to get through to reach results
- But code is part of your research

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil
  - → Something to get through to reach results
- But code is part of your research
  - → It will be **read** (by collaborators, reviewers, future you)

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil
  - → Something to get through to reach results
- But code is part of your research
  - → It will be **read** (by collaborators, reviewers, future you)
  - → It will **break** (after software updates, on another machine)

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil
  - → Something to get through to reach results
- But code is part of your research
  - → It will be **read** (by collaborators, reviewers, future you)
  - → It will **break** (after software updates, on another machine)
  - → It needs to be **maintained** (for revisions, new data)

Code is a scientific artifact

- Social scientists often treat computing as a necessary evil
  - → Something to get through to reach results
- But code is part of your research
  - → It will be **read** (by collaborators, reviewers, future you)
  - → It will **break** (after software updates, on another machine)
  - → It needs to be **maintained** (for revisions, new data)
- The standards of rigor you apply to your theory and research design should extend to your computing

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
  - → Copy-pasting results into Word
  - → Renaming files by hand
  - → 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

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

Three principles

- **Separation of concerns**

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**

Three principles

- **Separation of concens**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**
  - → Raw data → cleaned data → analysis → results → document

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**
  - → Raw data → cleaned data → analysis → results → document
  - → Each stage has clear inputs and outputs

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**
  - → Raw data → cleaned data → analysis → results → document
  - → Each stage has clear inputs and outputs
  - → No stage depends on a later stage

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**
  - → Raw data → cleaned data → analysis → results → document
  - → Each stage has clear inputs and outputs
  - → No stage depends on a later stage
- **Raw data is sacred**

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**
  - → Raw data → cleaned data → analysis → results → document
  - → Each stage has clear inputs and outputs
  - → No stage depends on a later stage
- **Raw data is sacred**
  - → Never overwrite it, never modify it by hand

Three principles

- **Separation of concerns**
  - → Code does analysis. Documents present it.
  - → Output files (tables, figures) are the interface between the two
- **The pipeline**
  - → Raw data → cleaned data → analysis → results → document
  - → Each stage has clear inputs and outputs
  - → No stage depends on a later stage
- **Raw data is sacred**
  - → Never overwrite it, never modify it by hand
  - → Code transforms raw data into everything else

Organizing a project: one example

```
PROJECT_FOLDER/
├ analyses/
│ ├ analyze.R
│ └ output/
│   └ table_models.tex
├ create_data/
│ ├ data.R
│ └ output/
│   └ data.csv
├ document/
│ ├ doc.tex
│ ├ doc.pdf
│ ├ doc.aux
│ └ ...
├ plots/
│ ├ output/
│ │ └ scatter.pdf
│ └ plots.R
├ Makefile
└ README.md
```

- Full example: github.com/franvillamil/workflow_example

Practical rules

- **Always use relative paths**

Practical rules

- **Always use relative paths**
  → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine

Practical rules

- **Always use relative paths**
  - → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  - → `read.csv("data/file.csv")` — works anywhere

Practical rules

- **Always use relative paths**
  - → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  - → `read.csv("data/file.csv")` — works anywhere
- **Never one giant script**

Practical rules

- **Always use relative paths**
  - → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  - → `read.csv("data/file.csv")` — works anywhere
- **Never one giant script**
  - → Break the project into discrete steps

Practical rules

- **Always use relative paths**
  - → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  - → `read.csv("data/file.csv")` — works anywhere
- **Never one giant script**
  - → Break the project into discrete steps
  - → Each script has clear inputs and outputs

Practical rules

- **Always use relative paths**
  → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  → `read.csv("data/file.csv")` — works anywhere
- **Never one giant script**
  → Break the project into discrete steps
  → Each script has clear inputs and outputs
- **Generated files are disposable**

Practical rules

- **Always use relative paths**
  - → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  - → `read.csv("data/file.csv")` — works anywhere
- **Never one giant script**
  - → Break the project into discrete steps
  - → Each script has clear inputs and outputs
- **Generated files are disposable**
  - → If a file can be reproduced by running a script, it is not a source file

Practical rules

- **Always use relative paths**
  - → `read.csv("/Users/fran/proj/data/file.csv")` — breaks on any other machine
  - → `read.csv("data/file.csv")` — works anywhere
- **Never one giant script**
  - → Break the project into discrete steps
  - → Each script has clear inputs and outputs
- **Generated files are disposable**
  - → If a file can be reproduced by running a script, it is not a source file
  - → Delete all output, re-run, and get the same results

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

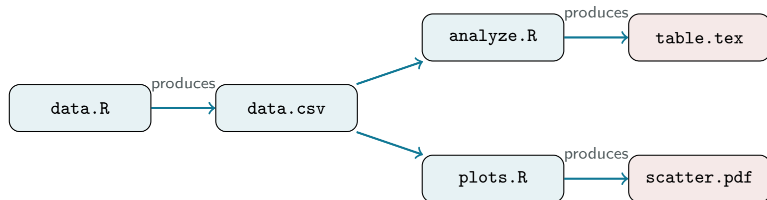
- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`
  - → `doc.tex`: `\input{../analyses/output/table_models}`
  - → `doc.tex`: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

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?
  - $\rightarrow$ 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`

Seeds and package versions

- Any code involving randomness must set a **seed**:

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script
  - → Same seed = same results every time

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script
  - → Same seed = same results every time
- Results can change across **package versions**

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script
  - → Same seed = same results every time
- Results can change across **package versions**
  - → Document which version of R and packages you used

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script
  - → Same seed = same results every time
- Results can change across **package versions**
  - → Document which version of R and packages you used
  - → Check your R session info with `sessionInfo()`

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script
  - → Same seed = same results every time
- Results can change across **package versions**
  - → Document which version of R and packages you used
  - → Check your R session info with `sessionInfo()`
  - → Options: `renv` (R) locks exact package versions

Seeds and package versions

- Any code involving randomness must set a **seed**:
  - → Simulation, bootstrapping, random sampling
  - → `set.seed(12345)` at the top of the script
  - → Same seed = same results every time
- Results can change across **package versions**
  - → Document which version of R and packages you used
  - → Check your R session info with `sessionInfo()`
  - → Options: `renv` (R) locks exact package versions
- Not paranoia — this has caused real problems in published work

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

What NOT to do

- Overwriting raw data with your code

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI
  → SPSS menus, Excel transformations, manual edits

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI
  - → SPSS menus, Excel transformations, manual edits
- "It works on my machine"

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI
  → SPSS menus, Excel transformations, manual edits
- "It works on my machine"
  → Almost always an absolute path problem

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI
  → SPSS menus, Excel transformations, manual edits
- "It works on my machine"
  → Almost always an absolute path problem
- "I fixed a few values in Excel"

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI
  → SPSS menus, Excel transformations, manual edits
- "It works on my machine"
  → Almost always an absolute path problem
- "I fixed a few values in Excel"
  → Undocumented, unreproducible, invisible to collaborators

What NOT to do

- Overwriting raw data with your code
- Results that require clicking through a GUI
  - → SPSS menus, Excel transformations, manual edits
- "It works on my machine"
  - → Almost always an absolute path problem
- "I fixed a few values in Excel"
  - → Undocumented, unreproducible, invisible to collaborators
- Scripts that must be run in a specific undocumented order

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

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

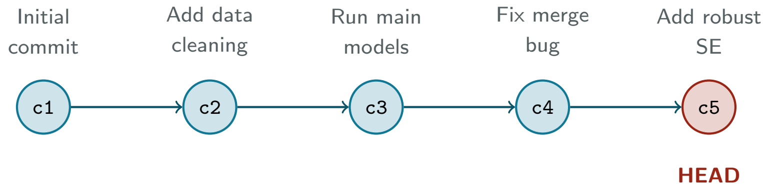
Git: recap from Session 1

- 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

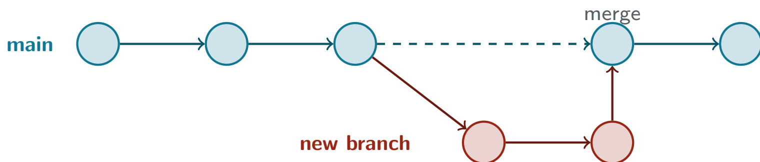
- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes
  - → `git commit -m "message"` — save a snapshot
  - → `git push` — upload to GitHub
  - → `git pull` — download from GitHub
- Today: going deeper into **good practices**

Git: commits as snapshots



- Each **commit** is a snapshot of your project at a point in time
- The **history** is a chain of commits, each with a message
- `HEAD` is a pointer to your current position

Branches: parallel lines of work



- A **branch** is an independent line of commits
- Useful for trying changes without breaking `main`
- **Merge** brings the work back together

The .gitignore file

Track (source files):

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

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

Do NOT track:

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

Good commit habits

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

Good commit habits

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

  → 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

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, ...)

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, . . .)
- From **day one** of a project, imagine you will submit one

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, . . .)
- From **day one** of a project, imagine you will submit one
  - → What would a reviewer need to go from raw data to published tables?

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, ...)
- From **day one** of a project, imagine you will submit one
  - → What would a reviewer need to go from raw data to published tables?
  - → Build that throughout — don't assemble it in a panic at the end

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, ...)
- From **day one** of a project, imagine you will submit one
  - → What would a reviewer need to go from raw data to published tables?
  - → Build that throughout — don't assemble it in a panic at the end
- A well-organized project is already a replication package:

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, ...)
- From **day one** of a project, imagine you will submit one
  - → What would a reviewer need to go from raw data to published tables?
  - → Build that throughout — don't assemble it in a panic at the end
- A well-organized project is already a replication package:
  - → Public repo with all scripts and a README

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, . . .)
- From **day one** of a project, imagine you will submit one
  → What would a reviewer need to go from raw data to published tables?
  → Build that throughout — don't assemble it in a panic at the end
- A well-organized project is already a replication package:
  → Public repo with all scripts and a README
  → Data (if shareable) or instructions to obtain it

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, . . .)
- From **day one** of a project, imagine you will submit one
  - → What would a reviewer need to go from raw data to published tables?
  - → Build that throughout — don't assemble it in a panic at the end
- A well-organized project is already a replication package:
  - → Public repo with all scripts and a README
  - → Data (if shareable) or instructions to obtain it
  - → Ideally: run one command and reproduce everything

The replication package as your north star

- Journals require **replication packages** (APSR, AJPS, JOP, ...)
- From **day one** of a project, imagine you will submit one
  - → What would a reviewer need to go from raw data to published tables?
  - → Build that throughout — don't assemble it in a panic at the end
- A well-organized project is already a replication package:
  - → Public repo with all scripts and a README
  - → Data (if shareable) or instructions to obtain it
  - → Ideally: run one command and reproduce everything
- Example: `github.com/franvillamil/vox_military`

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

Summary

- **Code is a scientific artifact**: treat it with rigor

Summary

- **Code is a scientific artifact**: treat it with rigor
- **Organize**: separation of concerns, clear pipeline, raw data is sacred

Summary

- **Code is a scientific artifact**: treat it with rigor
- **Organize**: separation of concerns, clear pipeline, raw data is sacred
- **Automate**: use Makefiles to run the pipeline

Summary

- **Code is a scientific artifact**: treat it with rigor

- **Organize**: separation of concerns, clear pipeline, raw data is sacred
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents

Summary

- **Code is a scientific artifact**: treat it with rigor
- **Organize**: separation of concerns, clear pipeline, raw data is sacred
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents
- **Code well**: DRY, constants, checks, seeds, readable style

Summary

- **Code is a scientific artifact**: treat it with rigor
- **Organize**: separation of concerns, clear pipeline, raw data is sacred
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents
- **Code well**: DRY, constants, checks, seeds, readable style
- **Use plain text**: portable, versionable, scriptable

Summary

- **Code is a scientific artifact**: treat it with rigor
- **Organize**: separation of concerns, clear pipeline, raw data is sacred
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents
- **Code well**: DRY, constants, checks, seeds, readable style
- **Use plain text**: portable, versionable, scriptable
- **Use Git**: commit often, write good messages, share via GitHub

For next week

- Complete Assignment 10 (computing practices)
- Next session: In-class exam + course review
  - → Exam covers sessions 1–9
  - → Second half: review and open questions

Questions?