Reproducible Research

Mark Agerton

2024 Nov 19, ARE 202A

Intro

Overview

Today is about **tools** and **workflows**, not research process.

We want to

- To share some of the tools we use to make us more productive
- Help you figure out the right tool for the job
- Make it easier for you to use these tools

We are economists, not software engineers or computer scientists!

Course materials

Course materials are a public good.

Pull requests on Github are welcome and will enhance the public good!

https://github.com/magerton/research-computing-bootcamp/

Reproducability

3 kinds of reproducibility

1. **Methods:** *The ability to implement, as exactly as possible, the experimental and computational procedures, with the same data and tools, to obtain the same results.*
2. Results: *obtaining the same results from the conduct of an independent study*
3. Inferential: *do we interpret results from an independent or reanalysis study?*

(Goodman, Fanelli, and Ioannidis 2016)

Reproducible research

Make the *entire research process* transparent

Not just regression tables. It includes

- Every step of data-work (including downloading!)
- Your figures
- Sources for any fact you cite

Reproducible research is about **workflow** and **sharing**

Why bother with reproducible research?

1. We are scientists who (ostensibly) care about the truth, and need others to be able to verify it (altruistic).
2. We can build off one another's work (spillovers).
3. Journals care about it (*not* altruistic).
4. It'll make your life easier (in the long run) (also not altruistic).

New AER Reproducibility Guidelines (Methods)

- AER's Data and Code Availability Policy

  It is the policy of the American Economic Association to publish papers only if the data and code used in the analysis are clearly and precisely documented, and access to the data and code is clearly and precisely documented and is non-exclusive to the authors.

More on AER guidelines

- *For econometric, simulation, and experimental papers, the replication materials shall include*
  a. the data set(s),
  b. the programs used to create any final and analysis data sets from raw data,
  c. programs used to run the final models, and
  d. description sufficient to allow all programs to be run.
- Social Science Data Editors Verification guidance and Replication Template

Ten Simple Rules for Reproducible Computational Research

See Sandve et al. (2013)

1. For Every Result, Keep Track of How It Was Produced
   - Notion of "workflow"... from raw data to final table
2. Avoid Manual Data Manipulation Steps
   - Excel is evil
3. Archive the Exact Versions of All External Programs Used
   - Python `virtualenv`, R's ~~`packrat`~~ `renv`, Julia's `[Project/Manifest].toml` all help here
   - Also hardware
4. Version Control All Custom Scripts
   - Git!!!
5. Record All Intermediate Results, When Possible in Standardized Formats
   - Show my example from SLURM

Ten simple rules, continued

6. For Analyses That Include Randomness, Note Underlying Random Seeds
7. Always Store Raw Data behind Plots
   - Plots & tables should be generated from files
8. Generate Hierarchical Analysis Output, Allowing Layers of Increasing Detail to Be Inspected
9. Connect Textual Statements to Underlying Results
   - For each statement, where did it come from? Cite the source or note your data
10. Provide Public Access to Scripts, Runs, and Results
    - Github?
    - AER data site, Harvard Dataverse

How to do this?

- Gold Standard?
  - Create entire paper with one command. (Jay will talk about `Make`)
  - *Literate programming*: Rmarkdown, Jupyter notebooks, Stata's `dyndoc`
- Usually not practical: too many files!

Nuts and bolts of my projects

Mark's strategy

1. Document everything
2. Version control (almost) everything
3. Automate as much as practical

Benefits & costs

- On the plus side
  - Keeps me organized
  - Means others can understand what I'm doing
  - Lowers marginal cost of re-running analysis with new data, assumptions, etc
  - Reduces errors
- But
  - Higher fixed costs
  - Co-authors grumpy about new software

Starting a new project

- Everything lives in one folder
- Sublime Text and R projects
- Track everything with Git
- And keep only 1 version (no redundancy!)
- Everything is a text file (except data and .pdfs)
  - "Diff"-able (track changes)
  - Future-proofed
  - Searchable from command line, Sublime Text

Folder structure and filenames

- `README.md` in every folder explains what's there
- Save `data-raw`
  - Ideally download programatically code
  - `README.md` documents acquisition
  - Save md5 hash of data
  - Don't ever modify it!
- `data-intermediate`
  - Processed data
- `writeup`
  - `paper/` has .tex files
  - `yyyy-mm-presentation/`
  - `figures/`
  - `tables/`
  - various notes
- `code`
  - `master.do`, `run_all.sh` or `MAKE` script to run all analysis
  - Name files in order of analysis `00a - download prices.R`, `00b - download shapefiles.R`

What to do with parameters?

- Sometimes we re-use parameters across many files
- What if we change them? Or have to hard-code?
- Create `R/CONSTANTS.R` file (no redundancy!)

in `R/CONSTANTS.R`

```
update_constants <- function(grepfor, replacewith,
    file="R/CONSTANTS.R", update=TRUE) {
    orig_const <- readLines(file)
    orig_const[grep(grepfor, orig_const)] <- replacewith
    if (update) cat(orig_const, sep="\n", file="R/CONSTANTS.R")
    else return(orig_const)
}
```

in script where we compute beta that gets used elsewhere

```
tmptxt <- paste0("beta <- ", sprintf("%a", beta), "  #  beta = "
update_constants("^beta", tmptxt)
```

Can also create a `parameters.tex` file and use `SIunitx`

Write out parameters to `parameters.tex` as newcommand

- `\newcommand{\ChamberCount}{62}`
- In LaTeX, `\qty[round-mode=figures, round-precision=2]{\LeakContractActual}{\grams\per\second}` or `\ChamberCount{}`
- Slunitx can handle rounding

Staying organized with lengthy jobs on a cluster

- Jobs on cluster get a unique job ID. See example SLURM script.
- Keep a `README.md` log with *metadata* (data about data) listing job IDs
- In the log, print
  - unique "commit hash" to log that identifies current snapshot of the codebase
  - Lots of intermediate output
  - `sprintf` numbers in hexadecimal format so I can restart

```
commit_hash <- system('git -C . rev-parse HEAD', intern = TRUE)
out <- paste0("\nRepo commit ", commit_hash, "\n\n")
cat(out, file=STAT_PTH, append=T)
```

```
thet optimal (binary)  = [-0x1.9f0f1d52eece2p+1, ]
thet optimal (decimal) = [-3.2426 ]
```

Resources on project management / organization

- Software Carpentry Data Management lesson
- Other economists' workflows
  - Ryan Kellogg's Lab Wiki
  - Hunt Allcott's Lab Wiki
  - Gentzkow and Shapiro *Lab Wiki*
  - Gentzow and Shapiro *PDF Code and Data for the Social Sciences: A Practitioner's Guide*
  - Knittel and Metaxoglou (2016) *Working with Data: Two Empiricists' Experience*

AI

Generative AI in research

- I started using Claude.ai a lot
- Great when I have done original/creative thinking and need help with routine things: cutting/processing/coding
- Use a "project" so I can upload a bunch of documents
- Formula for a prompt
  - "Hi, I'm an economist doing XXXXXX."
  - "In project knowledge, I have uploaded XXXXXX"
  - "Please do XXXXXX and create an ***artifact*** with your results"
  - "***Before you start, do you have any questions?***"
- Use cases
  - Shorten abstract, ARE update, paper intro
  - Port a clustering algorithm to a new language & create unit tests based on peer reviewed paper + examples
  - Help me grind through tedious algebra for a model written in LaTeX
  - ReFormat LaTeX table
  - Summarize rules in the Federal Register
  - Make a `makefile` from a project repo
  - Write an abstract as a Shakespearan soliloquy

Example AI prompt

You are a quiz creator of highly diagnostic quizzes. You will make good, low-stakes tests and diagnostics. You will then ask me two questions: what, specifically, the quiz should test, and what audience the quiz is for. Once you have my answers you will construct several multiple-choice questions to quiz the audience on that topic. The questions should be highly relevant and go beyond just facts. Multiple-choice questions should include plausible, competitive alternate responses and should not include an 'all of the above' option. At the end of the quiz, you will provide an answer key and explain the right answer.

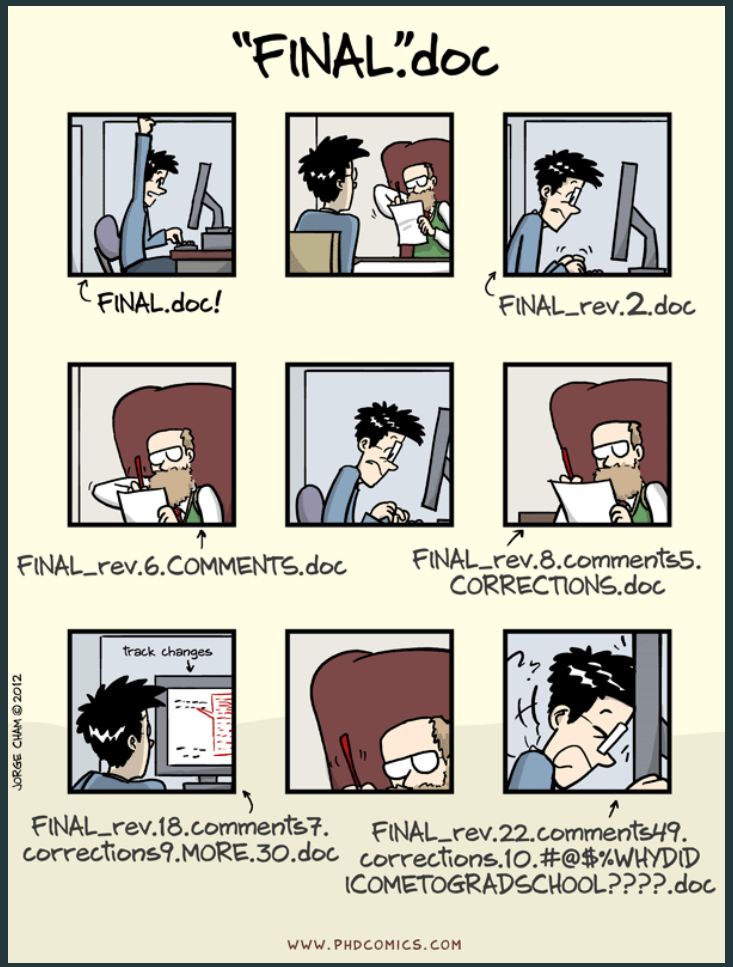
Source: HBS newsletter

Git

What is Git?

- A program that keeps track of file histories
- The biggest-baddest "Undo" button ever — roll back to any *commit*ted file
- Like Blockchain lite
  - Everyone gets the entire project history
  - Versions identified with cryptographic hash
- Searchable
- Can merge text file versions

#phdlife


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

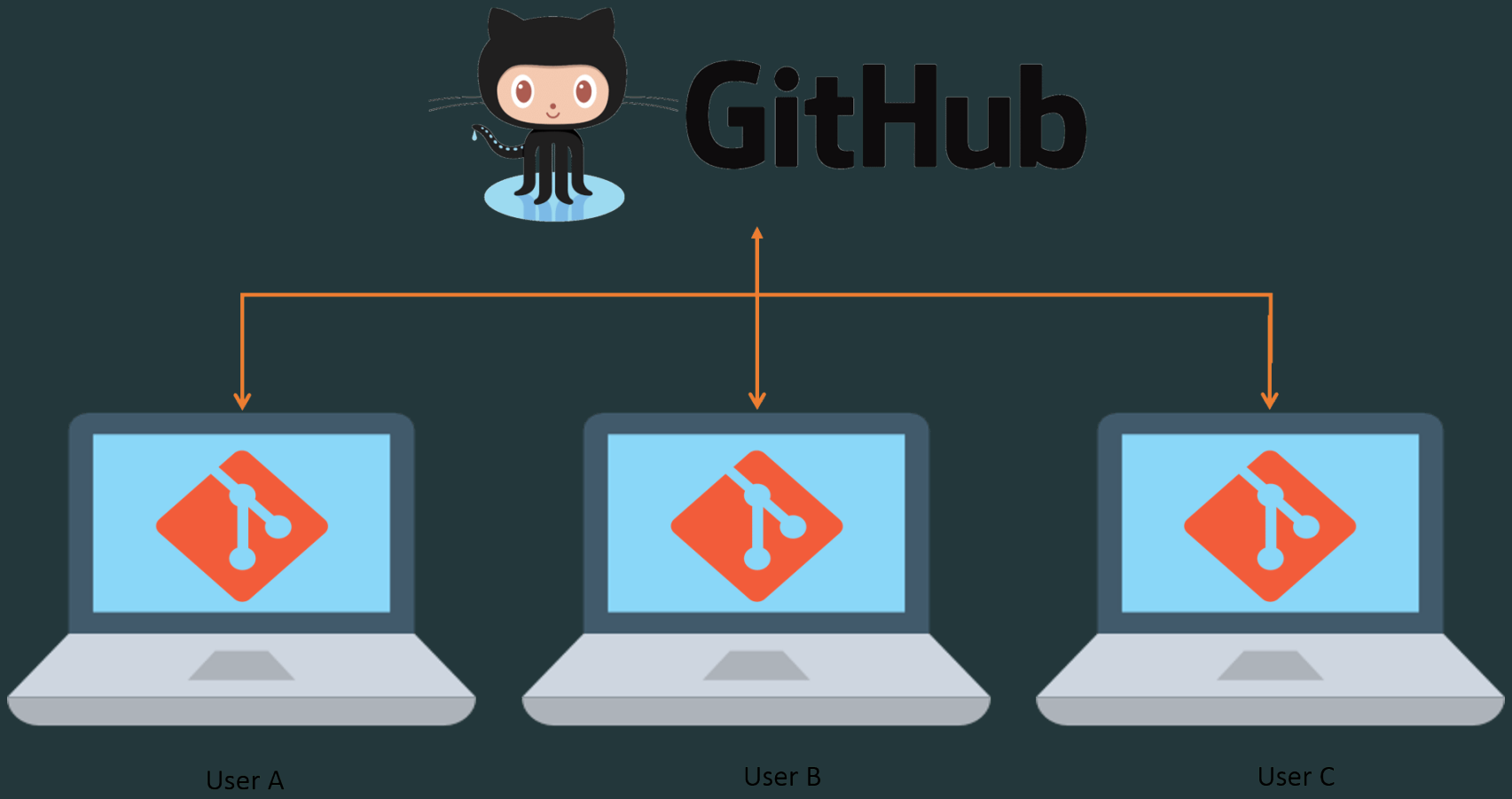
Git

Figure 1: *If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of 'It's really pretty simple, just think of branches as. . . ' and eventually you'll learn the commands that will fix everything.*

Git vs Dropbox

- Dropbox is better than nothing
  - Limits on how far back history saved
  - Can't choose what snapshots (*commits*) get saved
  - No metadata (*commit messages*) with saved versions
  - No tool to *merge* conflicting versions
  - Searching history is hard
  - Who changed what? (technical term: *blame*)
- But Git has higher fixed costs
- Git does NOT play well with Dropbox/Box/OneDrive/etc

Git vs Github



Git resources

- Ivan Rudik's lecture
- Grant McDermott's lecture
- QuantEcon lecture

Repositories vs commits

- **Repository**: folder where the entire history is tracked (*repo*)
  - Can be on your computer (*local*)
  - Or hosted online (*remote*)
  - Puts a folder `.git` in your direcotry
- **Commit**
  - "snapshot" of your repository at a point in time. 3 pieces of info
    1. A unique ID (*SHA-1 hash*)
    2. Content: pointer to *parent commit* + *diff* (changes to it)
    3. Metadata: *commit message*, author, timestamp
- **Github**: online service hosts repos
  - Can have git without Github
  - Public vs private
  - Adds extra functionality, project management
  - Great education discount

What I'll show you today

- Git GUIs
  - **Github Desktop** (what we do today)
  - Git Kraken
  - Sublime Merge
- Can also use command line, esp. for complicated situations

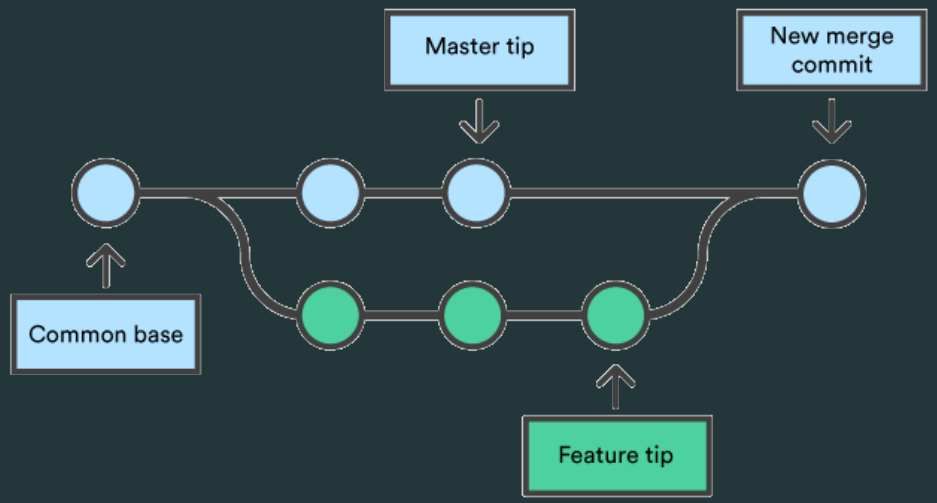
Workflow: Fork + Clone a repo

- Go to https://github.com/magerton/research-computing-bootcamp.git
- *Fork* the repo on the remote
  - Creates a *clone* of the repo in YOUR Github account
  - A *fork* keeps ability to sync original repo and yours
- *Clone* the repo from remote to local

```
# (makes new folder)
git clone https://github.com/[USERNAME]/research-computing-b
```

Branches

- Branches are sets of commits
- Keep `main` branch with functioning code
- Develop new features on *branches*
- When done, *merge* branches back to `main`



Workflow: Switching branches

```
## change working directory to new repo
cd my-first-test-repo

## add `-b` flag to create new branch
git checkout marks-test-branch
```

Workflow: staging changes + adding commit

```
## make a new file
echo "do Actually Relevant Econ" > note-to-self.md

## see changes
git status
git diff

## stage changes
git add *.md      # markdown files
git add -u        # tracked files
git add --all     # all files

git commit -m "Note to self" -m "About how econ is great"

## PUSH changes to remote
git push
```

Workflow: `checkout` branch and `merge` branches

```
## Switch back to master
git checkout main

## Merge in changes from `marks-test-branch` to current branch
git merge marks-test-branch
```

Workflow: Create a repo

- Make a new *private* repo on Github (remote): github.com/new
- Create a *local* repo and then push changes to remote

```
## new directory
mkdir my-first-test-repo

## cd into it
cd my-first-test-repo

## create readme
echo "# my-first-test-repo" >> README.md

## tells git to track new repo
git init

## STAGES the file
git add README.md
```

Authentication with Github Desktop & command line

- Github Desktop uses `https`-based authentication
- Command line works best with `ssh`
  - More from Arnon tomorrow
  - Github Authentication help
  - To switch from `https` to `ssh`
    - See Github help on changing a remote's URL

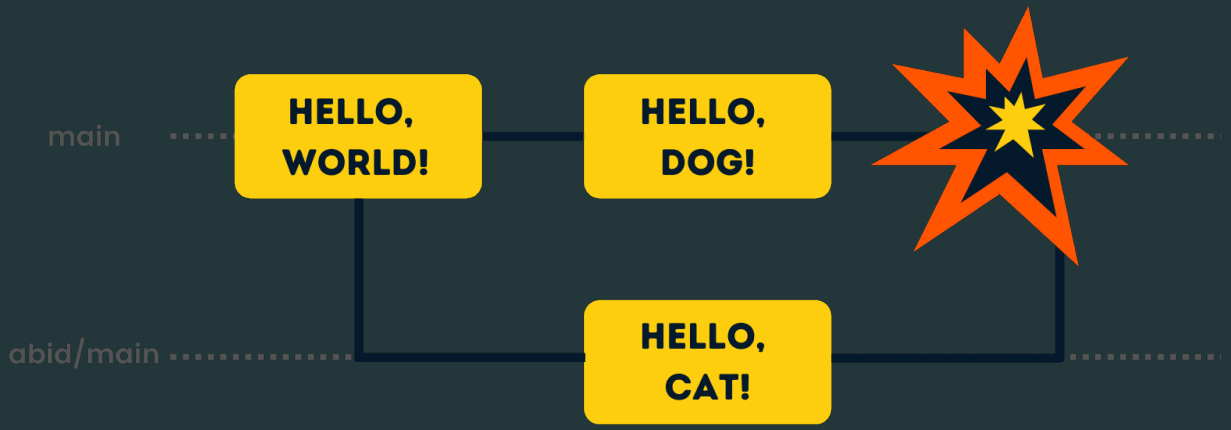
```
git remote -v
git remote set-url origin git@github.com:USERNAME/repository.git
```

Syncing between computers

- `push` changes to the `remote`
- `pull` changes from `remote` to `local`
  - `pull` is equivalent to

```
fetch
git merge FETCH_HEAD
```

`merge` conflicts



- Resolve in text editor (Microsoft VS Code is great for this!)

Important files

- `README.md` is for readers
  - Automagically displayed as HTML by Github
- `.gitignore` tells Git which files not to track
  - LaTeX temp files, big binary files, private keys
  - .gitignore has an example
- `LICENSE.md` tells how others can use, cite, modify, etc your work

What to do about binary files?

- Git is fantastic for tracking text files (.txt, .tex, .md, .R, .do, .py, .jl, .m)
- But not made for big binary files (.zip, .dta, .RData, .rda, .jld, .gz)
- Use `git-lfs` to handle binary files $\leq$ 2Gb
  - Store binary files on a separate server
  - Commits a text *pointer file* with unique *hash* and URL to repo
  - Way faster
  - 1 Gb for free, \$5/mo for 50 Gb

```
git lfs track my-big-file.zip  # single file
git lfs track "*.zip"          # all files ending in .zip. note
```

Project management on Github

- Permissions: who has the right to `push` to the repo?
  - repo > Settings > Collaborators
- Issue-tracking
  - Can reference issues in commit messages
  - Can tag other users

Organizing data

- What I've started doing
  - Save raw data into `./data-raw/`, with README.md on how to get it
  - Ingest data *programatically*
    - *Normalize* data as much as feasible. Redundancy is evil!
    - Save to `./data-intermediate/`
  - Further processing
- If data are huge, interesting guide on Software Carpentry

Data management

How I (try) to think about data

- Based on relational database theory
- Big idea is *normalization*
  - Redundancy is evil
  - Information should show up in one place only
  - Break out redundant parts into separate tables
- Why normalize?
  - Need less disk space
  - Lower chance of anomalies (misspellings, etc)
- Merge back together
  - Multiple tables painful in Stata
  - Easier with R (`dplyr` or my favorite, `data.table`)

Normalizing data: Suppose you get this Excel spreadsheet

LocID	Loc. nm.	PlantID	Plant nm.	Soil	Soil desc
11	Kirstenb. G.	431	Leaucadendron	A	Sandstone
		446	Protea	B	Sandstone/limestone
		482	Erica	C	Limestone
12	Karbon. M.	431	Leucadendron	A	Sandstone
		449	Restio	B	Sandstone/limestone

Example from MariaDB.com

Trying to create a table. . . .

LocID	Loc. nm.	PlantID	Plant nm.	Soil	Soil desc
11	Kirstenb. G.	431	Leaucadendron	A	Sandstone
NULL	NULL	446	Protea	B	Sandstone/limestone
NULL	NULL	482	Erica	C	Limestone
12	Karbon. M.	431	Leucadendron	A	Sandstone
NULL	NULL	449	Restio	B	Sandstone/limestone

- Rows 1–3 from one group
- Rows 4–5 from another
- How do you reference a row?

Each record needs to stand alone

LocID	Loc. nm.	PlantID	Plant nm.	Soil	Soil desc
11	Kirstenb. G.	431	Leaucadendron	A	Sandstone
11	Kirstenb. G.	446	Protea	B	Sandstone/limestone
11	Kirstenb.	482	Erica	C	Limestone
12	Karbon. M.	431	Leucadendron	A	Sandstone
12	Karbon. M.	449	Restio	B	Sandstone/limestone

- This is what our panels look like
- *Primary Key*
  - "specific choice of a minimal set of attributes (columns) that uniquely specify a tuple (row) in a relation (table)" Wikipedia
  - Here: `Loc.` AND `Plant` form a *composite* primary key
  - Could also be an ID number (row number?)

Why is this not great?

LocID	Loc. nm.	PlantID	Plant nm.	Soil	Soil desc
11	Kirstenb. G.	431	Leaucadendron	A	Sandstone
11	Kirstenb. G.	446	Protea	B	Sandst/limestone
11	Kirstenb.	482	Erica	C	Limestone
12	Karbon. M.	431	Leucadendron	A	Sandstone
12	Karb. M.	449	Restio	B	Sandstone/limestone

1. Table stores redundant information: *LocID* AND *Loc nm*
2. Misspellings!

Removing the fields not dependent on the entire key

LocID	PlantID
11	431
11	446
11	482
12	431
12	449

PlantID	Plant nm.	Soil
431	Leaucadendron	A
446	Protea	B
482	Erica	C
449	Restio	B

LocID	Loc. nm.
11	Kirstenb. G.
12	Karbon. M.

Soil	Soil desc
A	Sandstone
B	Sandstone/limestone
C	Limestone

- No redundancy. *Factor variables* in Stata, R or *Categorical Arrays* in Julia
- Join using *Foreign Key*: "a set of attributes that references a candidate key" Wikpedia

Dealing with data using R's `data.table`

I like R's `data.table` package if $<$ 2 billion rows

- Highly optimized
  - Much faster than `dplyr` if datasets are big (100k or more rows)
- In-place updating
  - `dplyr` and other packages copy dataframes if you modify them
  - Big deal if memory is an issue
- Parsimonious syntax
- Tight connection to SQL
- Joins
  - really fast
  - non-equi joins: `x.date <= y.date` or rolling joins (closest-to)

Software

What do I look for in software?

- Cross-platform
- Don't need the internet
- Everything is a text file $\implies$ need a good text editor
  - Syntax highlighting
  - Compile LaTeX & Markdown/Pandoc
  - Searchable with Regular Expressions
- Minimize redundancy $\implies$ one text format to rule them all!
  - Markdown/Pandoc
    - Compile to Word, LaTeX, PPT, Beamer, Reveal.js, HTML
    - Auto-formatted on Github
    - Lighter syntax than LaTeX
- Git, Github Desktop
- Zotero for bibliography (can share with people via Groups)
- Sublime Text 3 or VS Code for everything else

Example

```r
update_or_add_to_file <- function(grepfor, replacewith, file="pa
  # Read the file contents
  original_constants <- readLines(file, warn = FALSE)

  # Create the exact string to search for (without regex)
  search_string <- paste0("\\newcommand{\\", grepfor, "}")

  # Search for the exact line using fixed = TRUE
  linenos <- grep(search_string, original_constants, fixed = TRU

  # If the command is found, update the line, otherwise append
  if (length(linenos) == 0) {
    # Command doesn't exist, append the new line
    cat(replacewith, file = file, append = TRUE, sep = "\n")
  } else {
    # Command exists, replace it in the correct line
    original_constants[linenos] <- replacewith
```

References

References

Goodman, Steven N., Daniele Fanelli, and John P. A. Ioannidis. 2016. "What Does Research Reproducibility Mean?" *Science Translational Medicine* 8 (341): 341ps12–12. doi:10.1126/scitranslmed.aaf5027.

Sandve, Geir Kjetil, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. "Ten Simple Rules for Reproducible Computational Research." *PLOS Computational Biology* 9 (10): e1003285. doi:10.1371/journal.pcbi.1003285.