Code, Data, and Version Control: Best Practices for Economic Research

Brendan M. Price
Federal Reserve Board

April 2023

Email: brendan.m.price@frb.gov. Website: www.brendanmichaelprice.com. I am grateful to John Coglianese and Chris Nekarda for many helpful conversations. Any views or opinions expressed here are my own and do not necessarily represent the views or policies of the Board of Governors of the Federal Reserve System or its staff.

Roadmap

How should economists organize empirical research?

I. Workflow
II. Code
III. Data
IV. Version control

Ecumenical audience:

- RAs, PhD students, faculty
- Language-agnostic
- Related disciplines

Ask questions!

Prologue: Motivation

Brueghel the Elder, *Sloth*

Why bother?

Easy to neglect best practices

- Looming deadlines
- "I'll fix it later . . ."

Good workflows do entail upfront costs

But they're dwarfed by the benefits

- Better research process
- Better research product

Good workflow improves the research *process* . . .

It saves time

- Less debugging
- Less duplication of effort

It facilitates analysis

- Rapid prototyping
- Ease of exploration
- Reversible choices

It feels good

- Less frustration
- Less panic
- More pleasure in the craft

. . . and the research *product*

It promotes good science
- More discoveries
- Fewer mistakes

It complements presentation
- Better figures and tables
- Ease of answering questions

It has positive spillovers
- Shareable code
- Replication packages

All of which confers professional credibility

Virtuous cycles

Rich complementarities

- Better organization $\Longrightarrow$ simpler code
- Simpler code $\Longrightarrow$ easier to improve
- Faster code $\Longrightarrow$ cheaper analysis
- Cheaper analysis $\Longrightarrow$ deeper dives
- Version control $\Longrightarrow$ safer experimentation

So we should think holistically

Part I: Workflow

Brueghel the Elder, *The Harvesters*

Roadmap

I. **Workflow**
   - Reproducibility
   - Project organization
   - Project life cycle
   - Collaboration
   - Computing

II. Code

III. Data

IV. Version control

A working definition

Research is reproducible to the extent that

> *source code and data are sufficient for an outside researcher to replicate results exactly in a supported computing environment*

Intrinsic motivation:

- "Why am I getting different results?"

Extrinsic motivation:

- Increasingly policed
- Prominent retractions
- Journal requirements

The road to reproducible research

Main principles:

- Do everything in scripts
- Make projects self-contained
- Execute them in one click
- Test in a clean environment

Many pitfalls:

- Revisions to source data
- External dependencies
- Artifacts from previous runs
- Different computer setups
- Ambiguous instructions

The project directory

Give each project its own directory

- Maximally self-contained
- No gratuitous dependencies
- Install packages internally

Organize it coherently

- Separate files by function
- Use clear and concise names
- Exploit parallel structure

Keep it clean

- Avoid redundancy
- Minimize clutter

A flexible project template

Four (or more) subdirectories:

- `code`
- `data`
- `logs`
- `output`

We can add a few more:

- `libraries`
- `models`
- `paper`

Primary script: `main.sh (.do, .r, .py, ...)`

- Defines the order of execution
- Runs everything in sequence

A flexible project template (continued)

Subdivide `code` by function

- `code/build` (data preparation)
- `code/check` (data validation)
- `code/learn` (exploratory analysis)
- `code/share` (public-facing analysis)

Subdivide `data` by provenance

- `data/raw` (data as provided to you)
- `data/derived` (anything you created)

Use parallel structure, recycle names

- `code/harmonize.do` $\Longrightarrow$ `logs/harmonize.log`
- `code/learn/emp.do` $\Longrightarrow$ `output/learn/emp.pdf`

Other organizational principles

Use good nomenclature

- Intelligible
- Concise
- Memorable

Avoid redundancy

- Declare settings/paths/scalars in just one place
- Store repeated code in separate callable scripts

Ensure traceability

- Distinct input files $\implies$ distinct outputs
- Should be obvious what generated what

No circularity—directed acyclic graph

The project life cycle

Typical progression:

- Early: understand data, probe viability
- Later: fine-tune analysis, create nice figures

Project structure should evolve over time

- Reorganize directories
- Revisit nomenclature
- Tie up loose ends

Hierarchy should scale with complexity

- Simple project $\implies$ flatter directory structure
- Complex project $\implies$ subdirs, subsubdirs, ...
- Start simple, elaborate as needed

Leave the campsite better than you found it

Entropy is a fact of life

- Code decays
- Clutter piles up
- Bugs creep in

Streamline as you go

- Clarifies the code logic
- Reminds you what you did
- Controls bug population

Avoid clutter—then declutter

- “But what if I need it later?”
- Stay tuned for version control

Plan for a replication package

Common mistake: defer replication to the end

- Waste time revamping
- Scramble to meet deadlines
- Discover mistakes

Keep the end goal in mind

- Will the code meet journal standards?
- Will the code be runnable by a replicator?
- Will the code be useful to others?
- Does it look professional?

Stringent standard: *American Economic Review*

Getting along (is hard to do)

Collaborative work $\Longrightarrow$ added challenges

- Code works for me, crashes for you
- I edit a script, it messes with your analysis
- I can't read your code, you can't read mine

Three viable approaches:

- Designated coder: one person touches the code
- Partnership: multiple equal coders
- Surgical team: primary coder + others

Which approach?

Tailor to team and project

- Coding ability, attention to detail
- Comparative advantage, bandwidth
- Access to data, computing resources
- Project complexity, optimal language
- Interpersonal dynamics

Multiple coders $\Longrightarrow$ all the more vital to

- Use version control
- Adopt uniform conventions and style
- Be mindful of computing environments

Research computing

Invest in a good primary language

- Stata: great for economics—but costly, narrow
- R: increasingly popular—but many dialects
- Python: versatile, all-purpose—but newer to econ

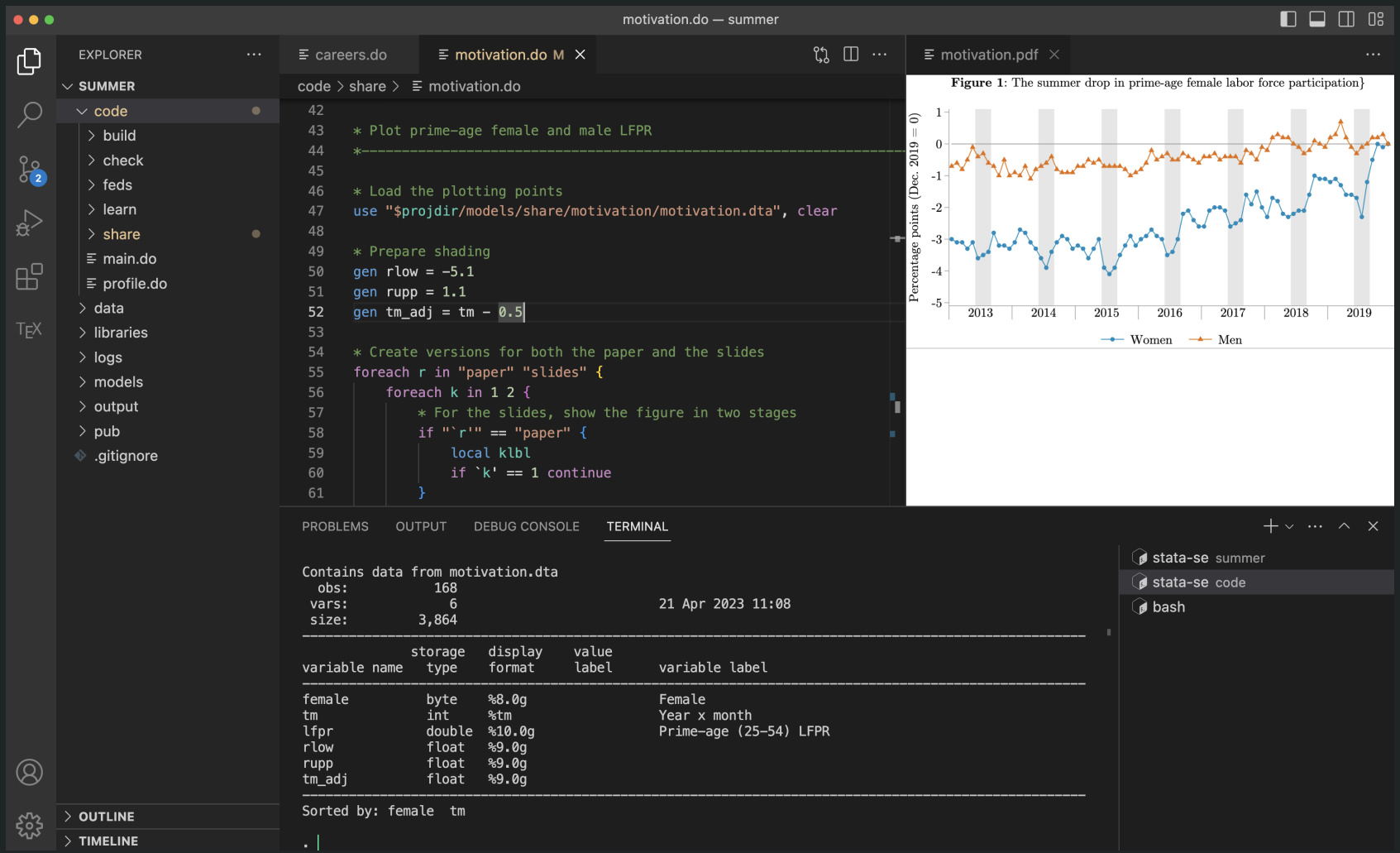
Supplement with the shell?

- Pro: useful lightweight utilities
  (preprocess text, scrape web, append pdfs)
- Con: reliance on OS impedes shareability

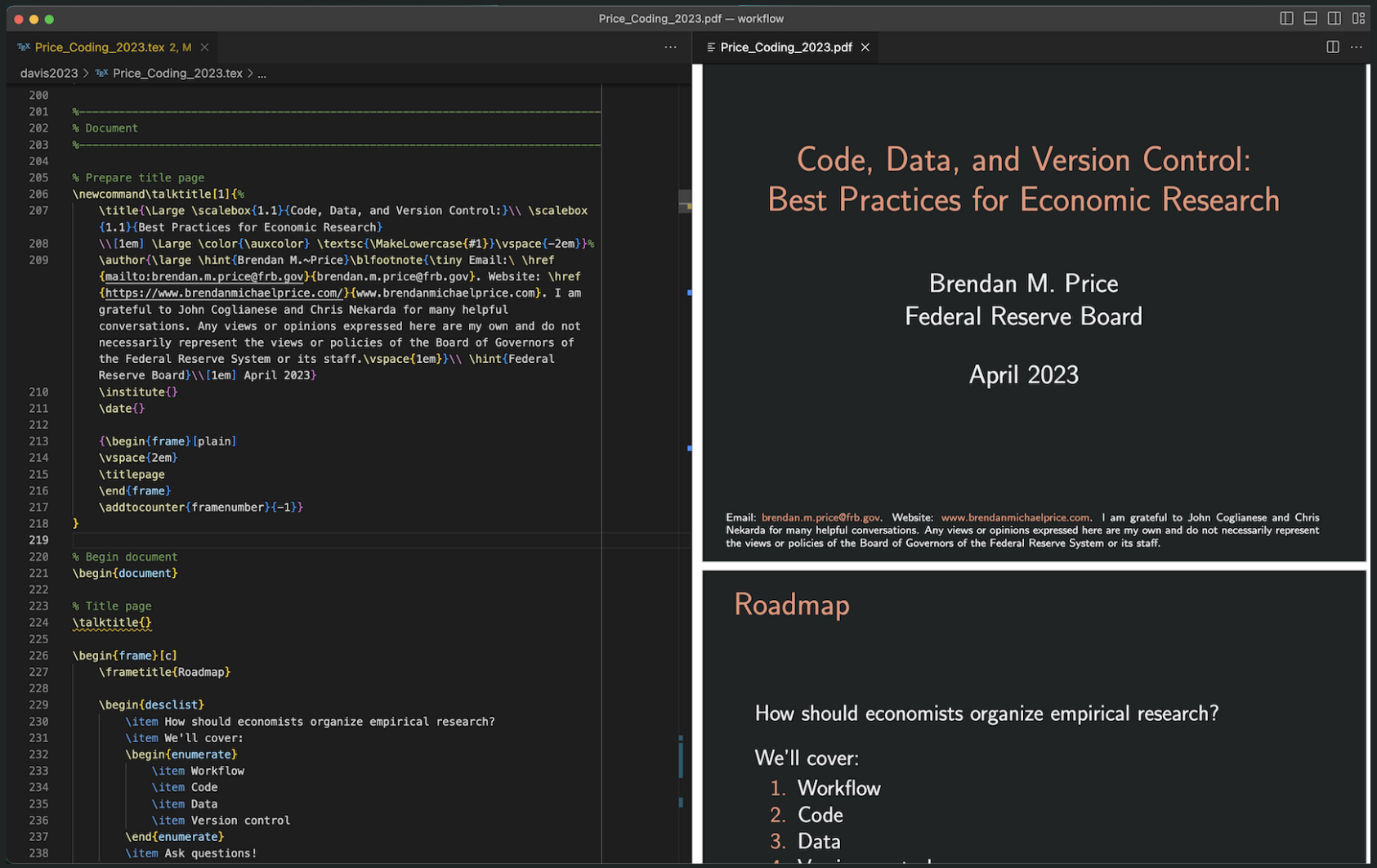
Highly recommended: Visual Studio Code

- Superb editing (syntax highlighting, multi-cursors)
- Integrated terminals, support for Git and LaTeX

VSCode: integrated terminal running Stata



VSCode: integrated LaTeX compilation



Go remote?

You may have access to a research server

Many pros . . .

- Faster processing
- Persistent sessions
- Shared access
- Regular backups

. . . but some cons

- Reliance on internet
- Server downtime
- Limited privileges
- May be harder to use GUIs

Learn a little Linux

Most servers run Linux*—so know the basics

- Navigation: `pwd`, `cd`, `ls`, `cp`, `mv`, `rm`, `mkdir`
- Permissions: `chmod`, `chown`, `chgrp`, `umask`
- Search and substitution: `find`, `grep`, `sed`
- Miscellaneous utilities: `cat`, `echo`, `head`, `less`
- Existential questions: `whoami`, `whatis man`

Powerful combos using pipes, I/O redirection, etc.

- Locate files containing a given string:

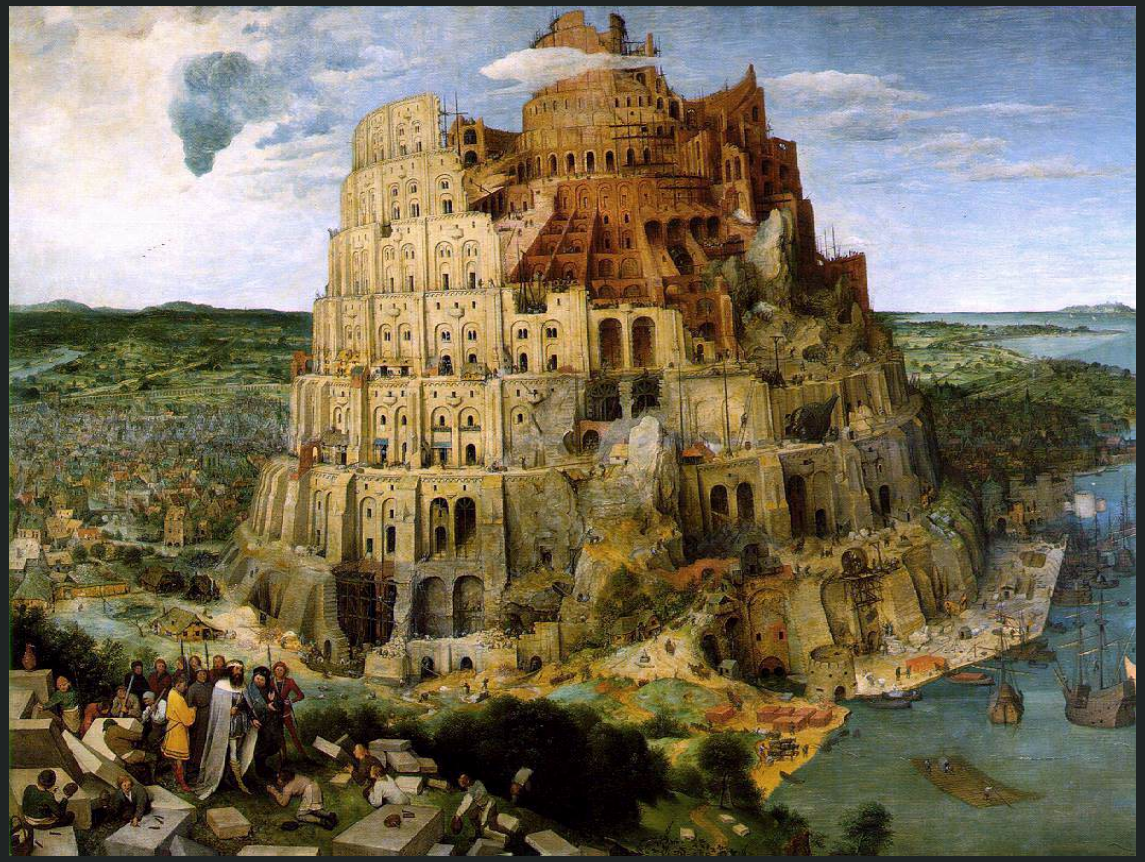
```
find haystack -name '*.py' -exec grep 'needle'
```

- Create a sorted list of lines containing a given string

```
cat counties.txt | grep 'Ohio' | sort > oh.txt
```

*MacOS is derived from Unix, so many Linux commands work here too.

Part II: Code



Brueghel the Elder, *The Tower of Babel*

Roadmap

I. Workflow

II. Code

- Desiderata
- Abstraction
- Readability
- Development
- Optimization

III. Data

IV. Version control

Desiderata

Good code is ...

- Concise
- Readable
- Robust
- Maintainable
- Extensible
- Efficient

These goals are often complementary

- Concise code runs faster
- Readable code is easier to maintain

Philosophy of continuous improvement

The root of all evil

The road to hell is paved with repetitive code

- Cut-and-paste within files
- Recycling across files

Curse of dimensionality

- Many groups (2 sexes $\times$ 3 age bins $\times$ ...)
- Many specs (raw, controls, FEs, more FEs ...)
- $\Longrightarrow$ Exponential growth

The horror, the horror

- Miserable to read, verify, debug
- Costly, error-prone to modify, extend

Automate repeated code

Abstract from repeated elements

- Encase repeated code blocks in loops
- Store repeated strings in macros
- Move repeated functionality into subroutines

Find the common threads

- Substantial overlap between `a`, `b`, `c`
- Loop over `a`, `b`, `c`
- Use conditionals to handle non-overlaps

Slimmer code, clearer logic

Abstraction in action

```
* Loop over sex
foreach f of numlist 0 1 {
    * Loop over outcome variables
    foreach yvar of varlist emp ump nlf lfe {
        * Specify controls and weights
        if "`yvar'" == "lfe" {
            local controls D.tmspline* D.weeks_elapsed
            local wtvar wtraked
        }
        else {
            local controls tmspline* weeks_elapsed
            local wtvar wtfinl
        }

        * Run the specification and save the estimates to disk
        quietly ivreg2 `yvar' ib5.month `controls' if female == `f' ///
            [aw = `wtvar'], bw($bandwidth) robust small
        process_estimates, path("share/overall") model("f`f'_`yvar'")
    }
}
```

Source: Price and Wasserman (2022).

Don't hardcode

Hardcoding = writing a literal instead of a variable

- User-specific filepaths
- Start/end of sample period
- Income thresholds

Main problem: multiple instances

- Annoying to change
- Hard to track down
- Hard to keep in sync

Obscures the logic, impedes readability

Instead: define variables in prominent places

- `main.do`, file header, `params.txt`

Write readable code

Readable code is easier to . . .

- Understand
- Debug
- Maintain
- Extend

The basic ingredients:

- Brevity balanced with clarity
- Good nomenclature
- Clear documentation
- Stylistic consistency

Be brief . . .

"Omit needless words."—Strunk and White

- Less to read, search, maintain, debug
- Better starting point for recycles

Tips for brevity:

- Automate extensively
- Look for one-liners
- Cut vestigial code

Fluency pays, as in this Stata example:

```
merge pid using cps.dta
assert _merge == 2 | _merge == 3
keep if _merge == 3
drop _merge
```

$\Longrightarrow$ `merge pid using cps.dta, assert(2 3) keep(3) nogen`

. . . but not overly brief

This is good:

```
gen sep_type = .
replace sep_type = 1 if sep == 1 & unemp == 1
replace sep_type = 2 if sep == 1 & nlf == 1
```

This is more compact, but harder to read:

```
gen sep_type = 1 if sep == 1 & unemp == 1
replace sep_type = 2 if sep == 1 & nlf == 1
```

And clever one-liners are not always best:

```
gen sep_type = unemp + 2 * nlf if sep == 1
```

Exploit visual parallels

Visually aligned code is easier to read:*

```
gen topcode = .
replace topcode =  999    if inrange(year, 1982, 1988)
replace topcode = 1923    if inrange(year, 1989, 1997)
replace topcode = 2884.61 if inrange(year, 1998, .)
```

Compare:

```
gen topcode = 999 if inrange(year, 1982, 1988)
replace topcode = 1923 if inrange(year, 1989, 1997)
replace topcode = 2884.61 if year >= 1998
```

Give related variables same prefixes, equal-length names

- Standardized prefixes $\Longrightarrow$ easy wildcard matching
- Standardized length $\Longrightarrow$ visual alignment

*Multiple spaces pair poorly with tab-indented code, since text editors expand tab indents to differing numbers of spaces.

Balance number vs. length of scripts

Each script should form a logical whole

- Unit of execution
- Locus of comprehension

More files or long files?

- Clutter vs. clarity
- Modular execution
- Ease of tracing version history

Rule of thumb: $>$500 lines feels long

A distribution of script lengths

Looked at a recent project . . .

```
find ./code -name '*.do' -exec wc -l {} | sort
```

Out of 39 code files:

- 12 files in `code/build`, 27 files in `code/share`
- 6 files under 100 lines, 2 files over 500 lines

My central tendency is 100–300 lines

Personal preference—but avoid bloat

Document for structure and clarity

Include a file header

- Purpose of the file
- Dependencies, caveats, known issues
- Version control handles author, date

Use section headers

- Breaks code into visually distinct units

Write informative comments

- Tautologous: `* Keep data from 1994 onward`
- Enlightening: `* Keep data post-CPS redesign`

Maintain comments as you would code

- Otherwise: code/comments contradict

Adopt a consistent style

Style encompasses:

- Naming conventions
- White space conventions
- Command abbreviations
- Preferred commands, approaches

No arguing about taste . . . but be consistent

- Easier to read own, others' code
- Easier to search for strings/patterns
- Easier to remember object names
- Easier to recycle across projects

Find a style guide for your language

Developing code

Sometimes makes sense to start rough . . .

- Ephemeral code
- Prototyping
- Time pressure
- Learning a new language

. . . but polish sooner rather than later

- Interactive $\rightarrow$ scripted
- Repetitive $\rightarrow$ automated
- Hardcoded $\rightarrow$ flexible
- Inefficient $\rightarrow$ optimized

Edit, edit, edit

Optimizing code

Two goals: efficient runtime, efficient storage

Theoretical trade-offs . . . but be on the Pareto frontier

- Space seldom binds
- But big files are slow and unwieldy

Locate and address bottlenecks

- Gentzkow and Shapiro: "profile slow code relentlessly"
- Develop a feel for fast and slow
- Use timers to record execution time

The need for speed

The sloth's excuse: "I'll only run it once"

Lots of reasons to rerun code:

- Ensuring it still works
- Incorporating new data
- Revisiting choices
- Purging artifacts
- Recycling across projects

Faster code $\implies$ quicker iteration

- Editing for readability
- Cleaning additional variables
- Modifying sample restrictions

Quicker code

Structure the data for speed

- Use minimal storage types
- Sort with an eye to future merges
- Operate on aggregates when you can

Leverage language

- Find faster commands, paradigms
- Farm out slow tasks to a faster language

Use intermediate files (judiciously)

- Store slow-to-build extracts
- Store regression estimates

Toggle slow code on/off (via flags, not comments)

Part III: Data

Brueghel the Elder, *Hunters in the Snow*

Roadmap

I. Workflow

II. Code

III. Data

- Provenance
- Data preparation
- Data validation
- Data analysis

IV. Version control

Provenance

Keep track of data provenance

- So you can re-download if needed
- So replicators can retrace your steps

Record where/how you got the source data

- Website linking to data extracts
- Exact URLs for direct downloads
- Instructions for navigating interface
- Accompanying documentation

Record *when* you got the data

- In case you later lose access
- In case data are revised over time

Protect the raw data

Keep the raw data pristine

- Store separately from derived files
- Never overwrite raw data(!)
- Limit and document file renames
  (but fine to store compressed)

Rare exception: prohibitively large files

- Retain subset of observations or variables
- Codify steps to downsize file
- Preprocess no more than necessary
- Save smaller file

Apart from that: clean only via code

Data preparation

Understand the data structure

- Unique identifiers
- Hierarchical relationships
- Completeness, missings
- Redundancies

Simplify and restructure

- Get rid of clutter
- Deal with inconsistencies
- Harmonize variables
- Improve nomenclature

Store cleaned extracts

Keep what you need

Raw data files contain lots of clutter

- Variables you'll never use
- Messy or gratuitous labels
- Early years with incomplete data

Keep what you need—and will bother to clean

- Economize on space (and speed)
- Retain what you can vouch for
- Can always reintroduce later

Eliminate redundancies

Raw data often contain redundant information

- Extra rows for state-wide totals
- Transformations of other variables

Verify redundancies, then eliminate them

- Make sure the data are internally consistent
- Drop anything you can recompute later

What if the data *aren't* internally consistent?

- Document any inconsistencies you find
- Deal with them on a case-by-case basis

Create clean extracts

Construct a minimal set of derived extracts

- Semi-cleaned extracts for data validation
- Fully cleaned extracts ready for estimation

Invest in high-quality extracts

- Unique identifiers
- Efficient data structures
- Clear, concise names and labels
- Sort by the unique IDs

Defer niche data processing further downstream

- Store `emp`—but compute `ln(emp)` on the fly

Data validation

It's (really) easy to make mistakes

- Raw data are wrong
- Raw data are right, but not what you think
- Typos, bad merges, floating-point issues ...

Solution: systematic data validation

- Reveals errors and false assumptions
- The sooner the better

Side benefits:

- Be better equipped to answer seminar questions
- Stumble on new ideas or identification strategies

Sweat the small stuff

Temptation: ignoring "small" bugs

- Issue only applies to a few observations
- Results make sense despite known bug

Most errors are canaries in coal mines

- Misunderstanding the data
- Misunderstanding the language

"Small" bugs often get amplified

Do not rest easy till you sort them out

Set tripwires

Data entail logical restrictions

- Additivities (population = sum across ages)
- Valid ranges ($0 \leq$ earnings $\leq$ income)
- Time invariance (birth state is fixed)

Set "tripwires": code crashes if an assertion fails

- Check your understanding of the data
- Document when identities do/don't hold
  (`assert popt == popm + popf if year >= 2003`)

Keep tripwires on—not commented out

- Old assumptions may cease to hold
- Add new data, redefine sample, update package ...

Know thy data

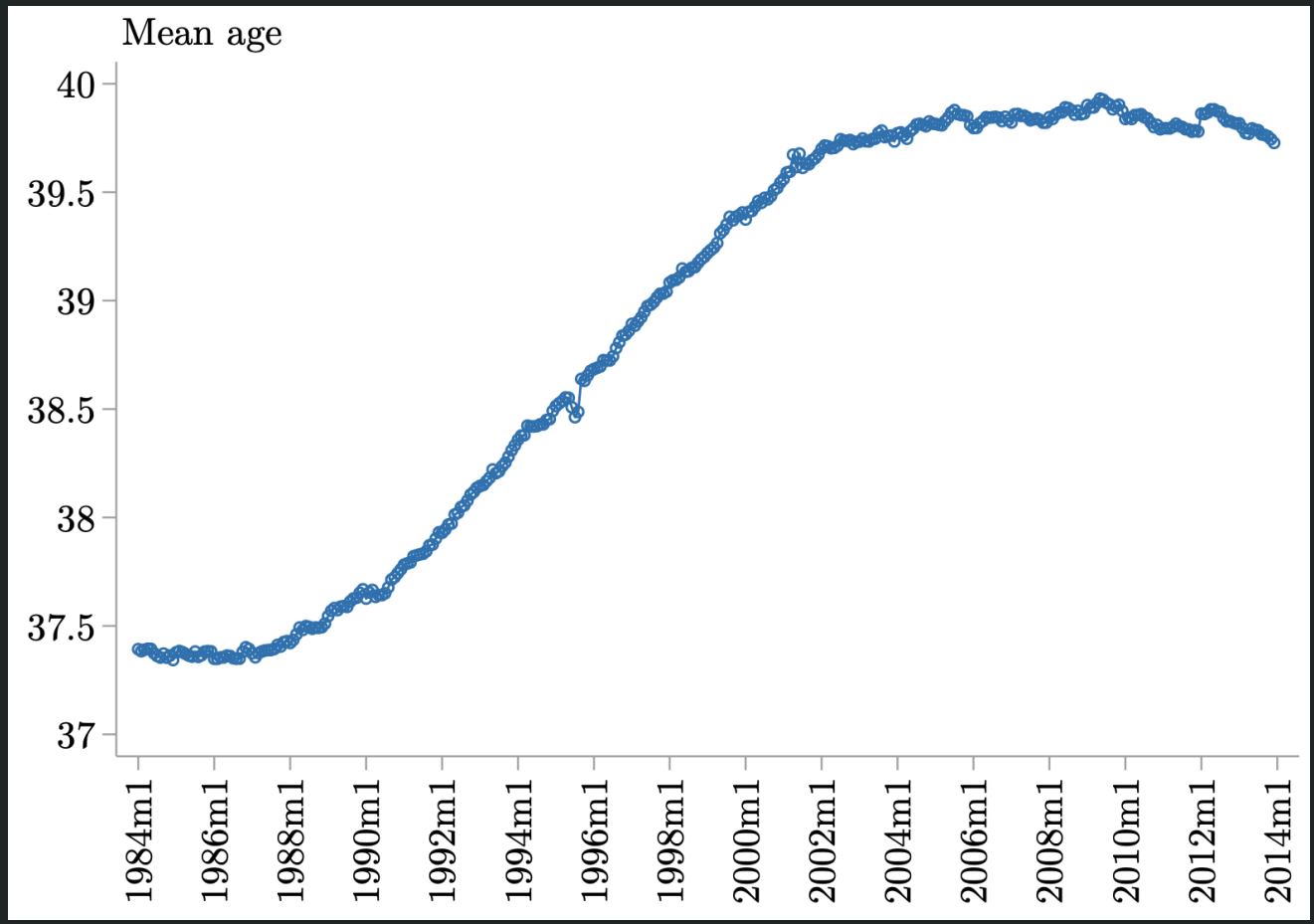
Look at the data!

- Browse individual observations
- Inspect means, SDs, correlations
- Plot the data over time

Ask yourself:

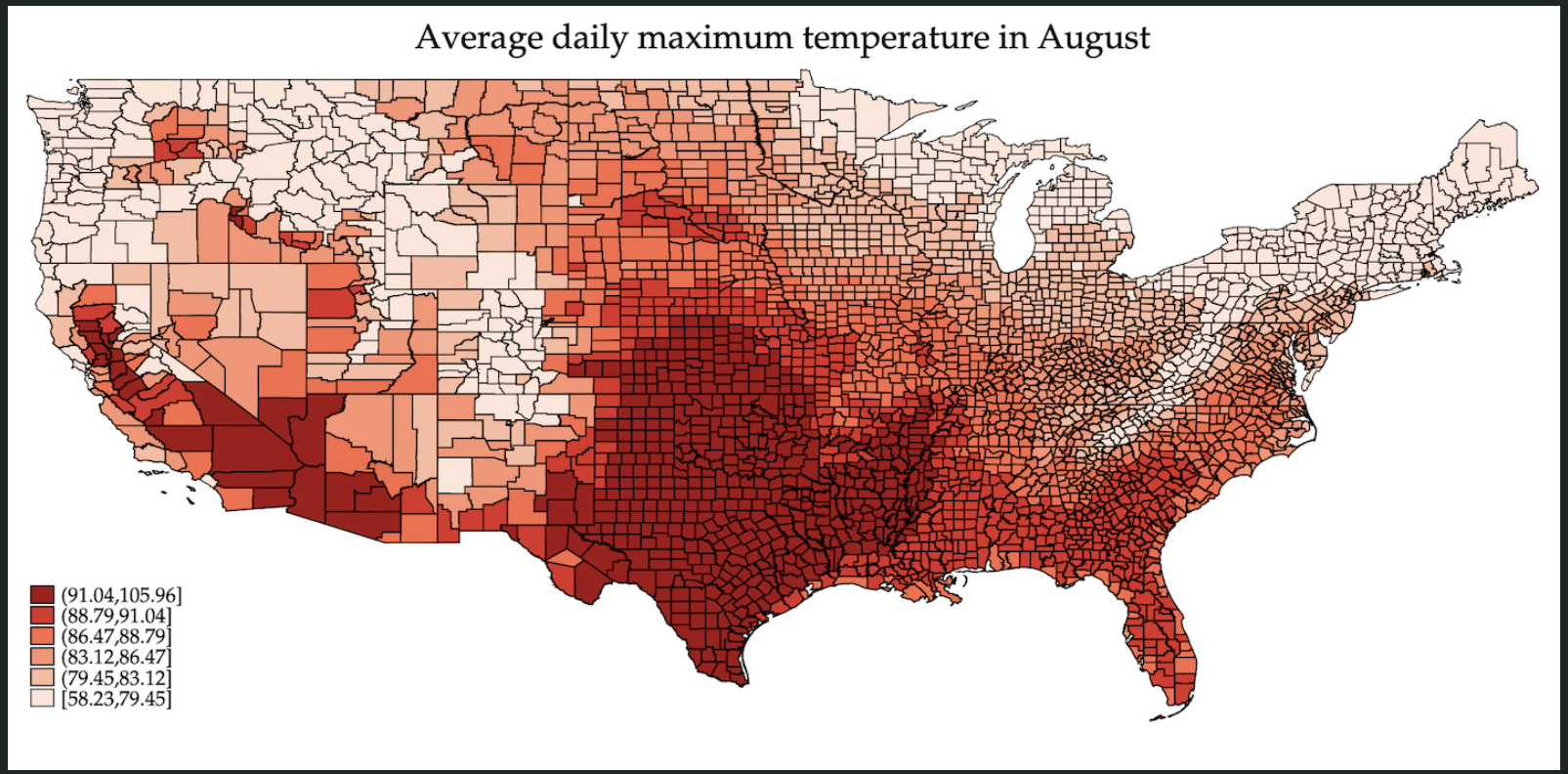
- Are the patterns plausible?
- Are there breaks in the data?
- Are there anomalies or outliers?
- Do expected features manifest?

Smooth series should evolve smoothly



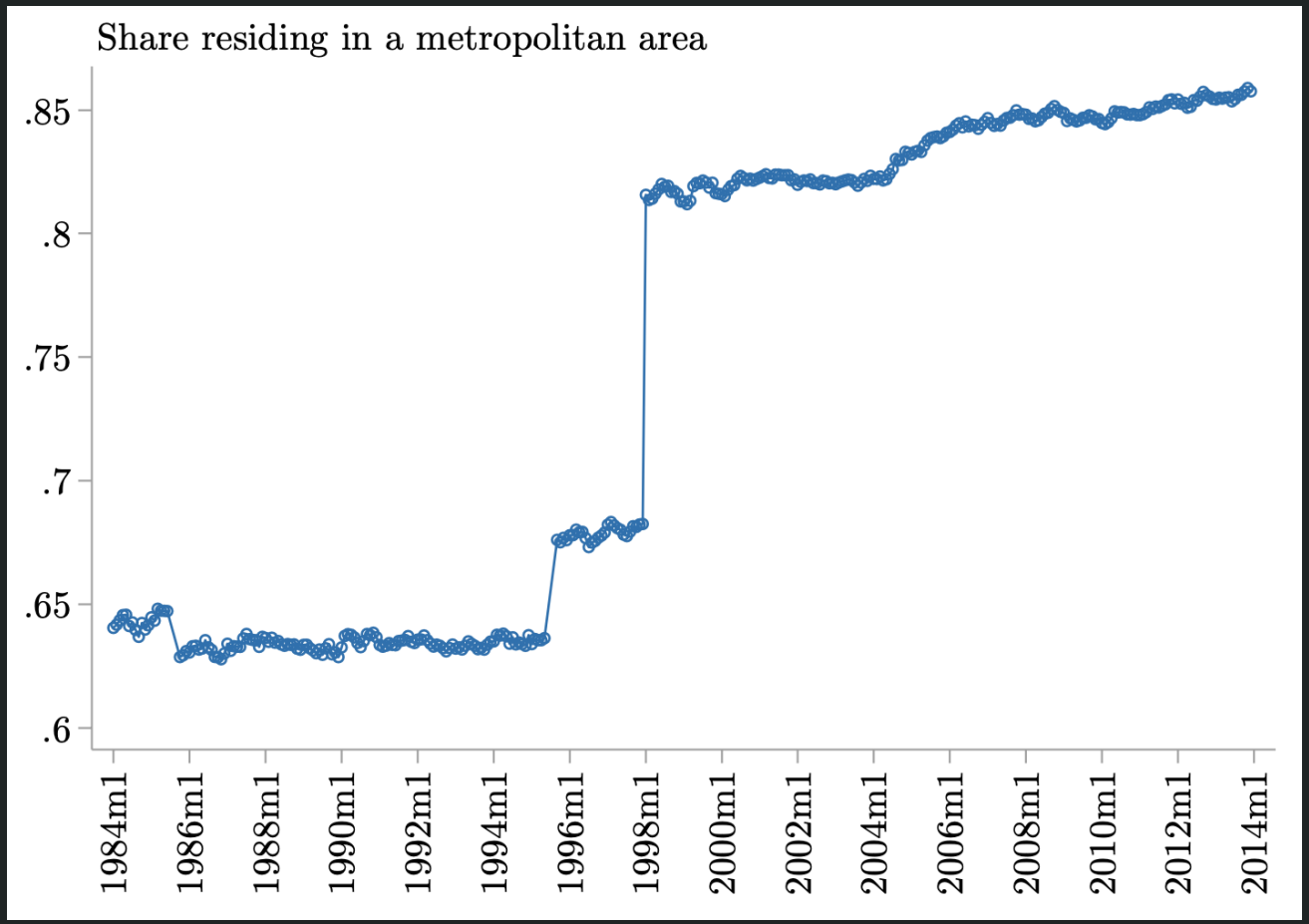
Source: IPUMS CPS data on prime-age US workers.

Hot places should be hot



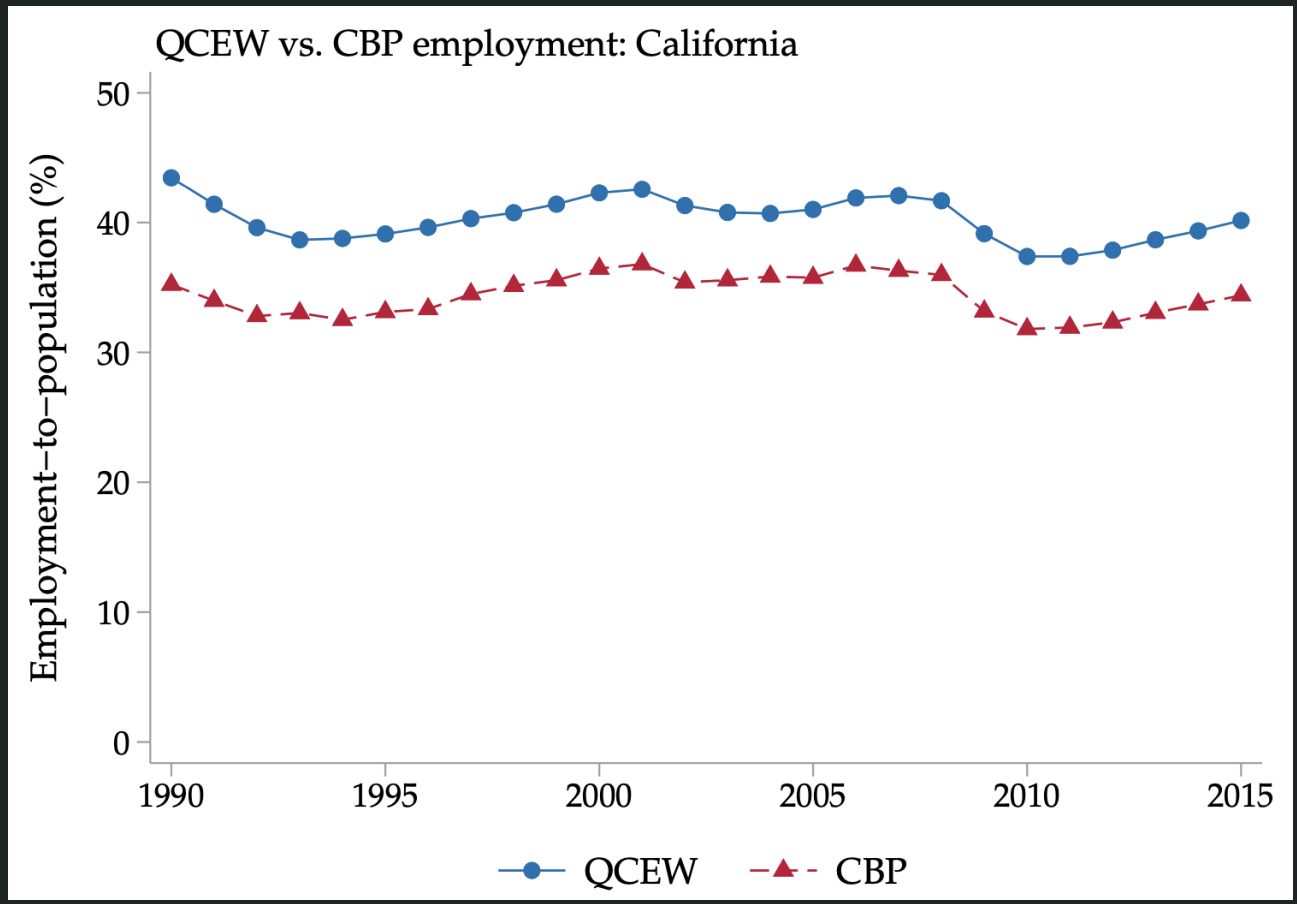
Source: North American Land Data Assimilation System, 1979–2011.

Plotting the data often reveals data seams



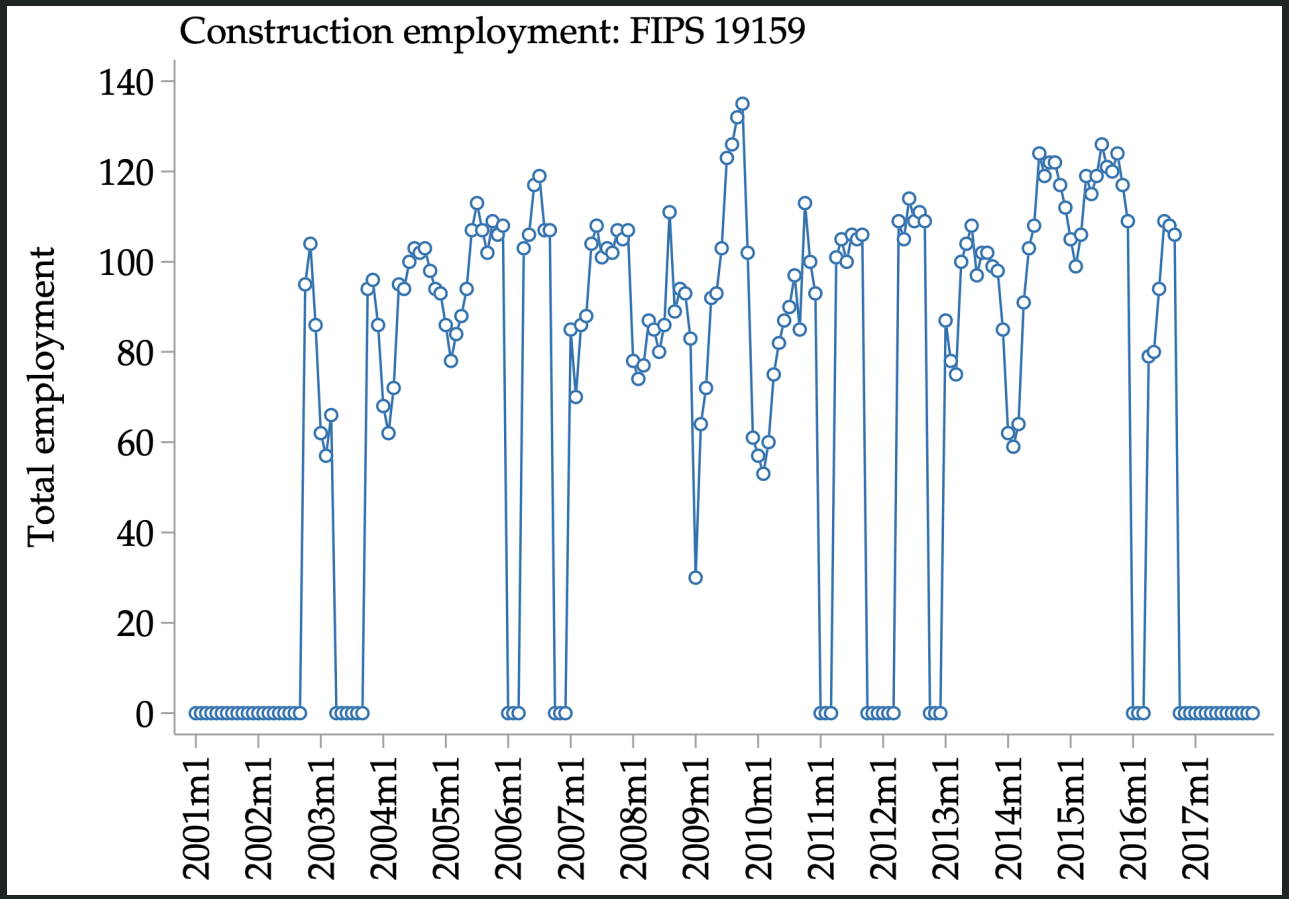
Source: IPUMS CPS data on prime-age US workers.

Validate one dataset against another



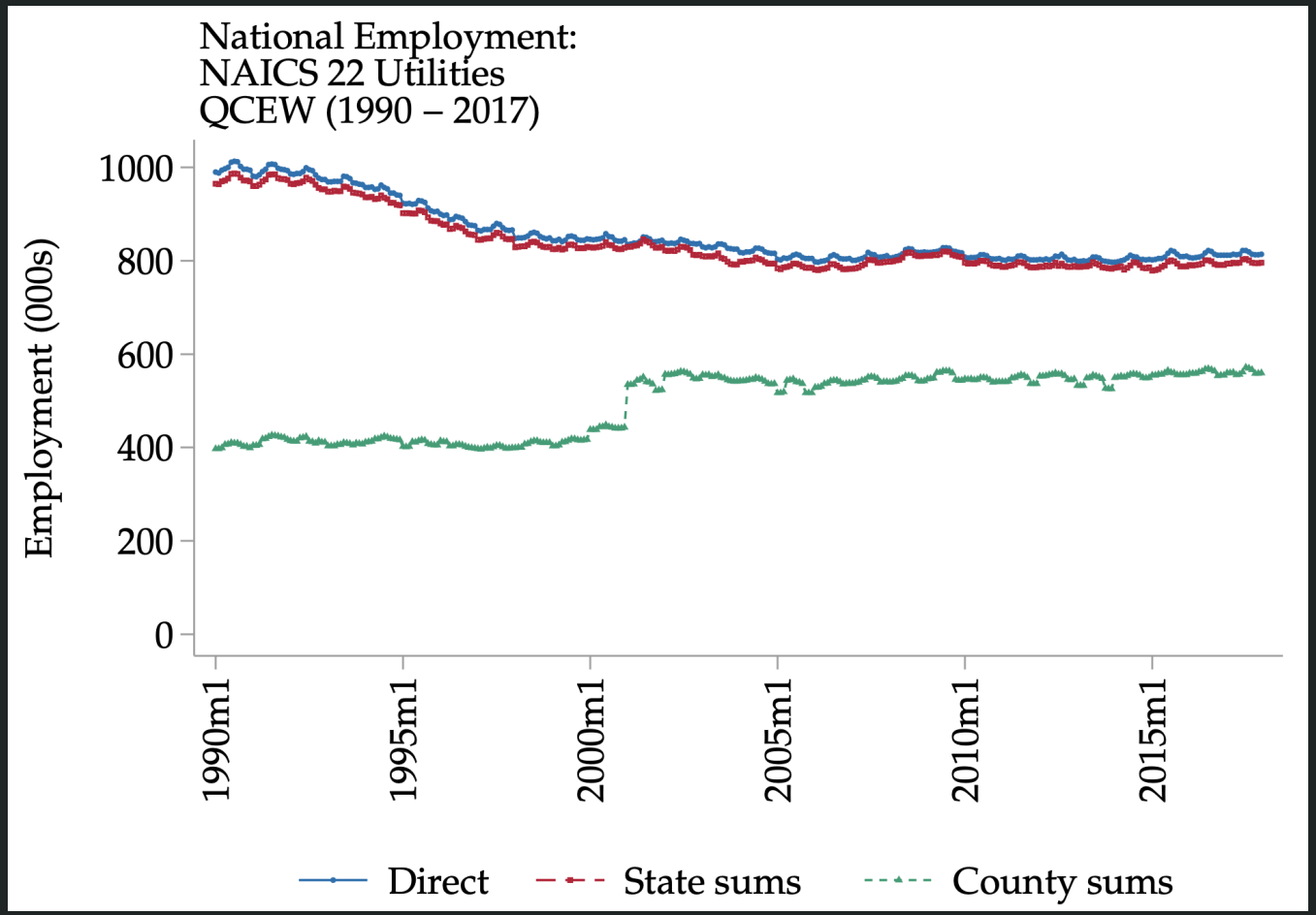
Source: Quarterly Census of Employment and Wages; County Business Patterns.

Visualize data suppressions



Source: Quarterly Census of Employment and Wages.

Examine data at different levels of aggregation



Source: Quarterly Census of Employment and Wages.

Lower the marginal cost of data analysis

Effective workflows yield . . .

- Quicker turnaround
- Fewer mistakes

Benefits scale with project complexity, duration

Quicker turnaround

Best practices save you time

- Automation $\Longrightarrow$ coding time
- Optimization $\Longrightarrow$ computational time
- Organization $\Longrightarrow$ search time

Faster turnaround means ...

- Cheaper data exploration
- Easier analytical modifications
- Less slog, more momentum
- Timelier feedback

Especially important with big, hard-to-access data

Fewer mistakes

Mistakes are costly if discovered . . .

- Backtracking, lost time
- Embarrassment, lost credibility
- Journal rejection, retraction

. . . but also if undiscovered

- Promising pilot flops at scale
- Measurement error attenuates result
- Erroneous science shapes the debate

Avoid unforced errors

Part IV: Version control

Brueghel the Elder, *Big Fish Eat Little Fish*

Roadmap

I. Workflow

II. Code

III. Data

IV. Version control

- Version chaos
- Version control in a nutshell
- Version control benefits
- Learning Git

Version chaos

We've all done it:

- `atus_v1.do`, `atus_v17.do`, `./v3/atus.do`
- `cps_bp.do`, `cps_FINAL_bp.do`, `cps_jul_bbag.do`

Confusion, error, terror, and strife

- Which version is authoritative?
- Who changed this? When?? WHY???
- How was the directory organized as of July?
- How does `v3` of one file relate to `v3` of another?
- When should I create a new version?
- Do I dare disturb the universe?

Even worse: multiple authors, machine migration

Poor man's version control

You might choose to go low-tech

- Git is unavailable
- Git is too hard(?!)
- Coauthors sharpening pitchforks

If you must: good organization helps

- Coherent, stable directory structure
- Readable, streamlined code

Archive the entire codebase at key junctures

- Rerun everything to ensure it's in sync
- Keep everything lightweight (`code`, `logs`, `output`)
- Apply a sortable date-stamp (`YYYYMMDD`)

Actual version control

Version control: a record of revisions to a set of files

- User saves snapshots of code and related files
- Easy to recover code from any given snapshot
- Easy to see how, when, by whom a file was edited
- Ideal for both solo and collaborative work

Everybody uses Git

- Versatile, reliable, fast, space-efficient

Usually paired with GitHub or GitLab

- Syncing across users, devices
- Discussion threads, project management

Some Git lingo

repository: a project containing version-controlled files

clone: a copy of the repo containing full project history

- Local clone on each coder's machine
- Remote clone on GitHub or GitLab

tracking: designating a (code) file for inclusion in the repo

commit: a snapshot of all tracked files at a particular time

pushing: uploading commits to GitHub/GitLab

pulling: downloading commits from GitHub/GitLab

The basic Git workflow

Initial configuration

- Set up a project directory as usual
- Initialize a Git repo (creates hidden `.git`)
- Link it to GitHub or GitLab

Commit and push an initial set of files

- Commit ID, author, timestamp, commit message

Further commits

- Create and edit files as usual
- Commit what/when you want to
- Tell Git what to ignore (derived files)

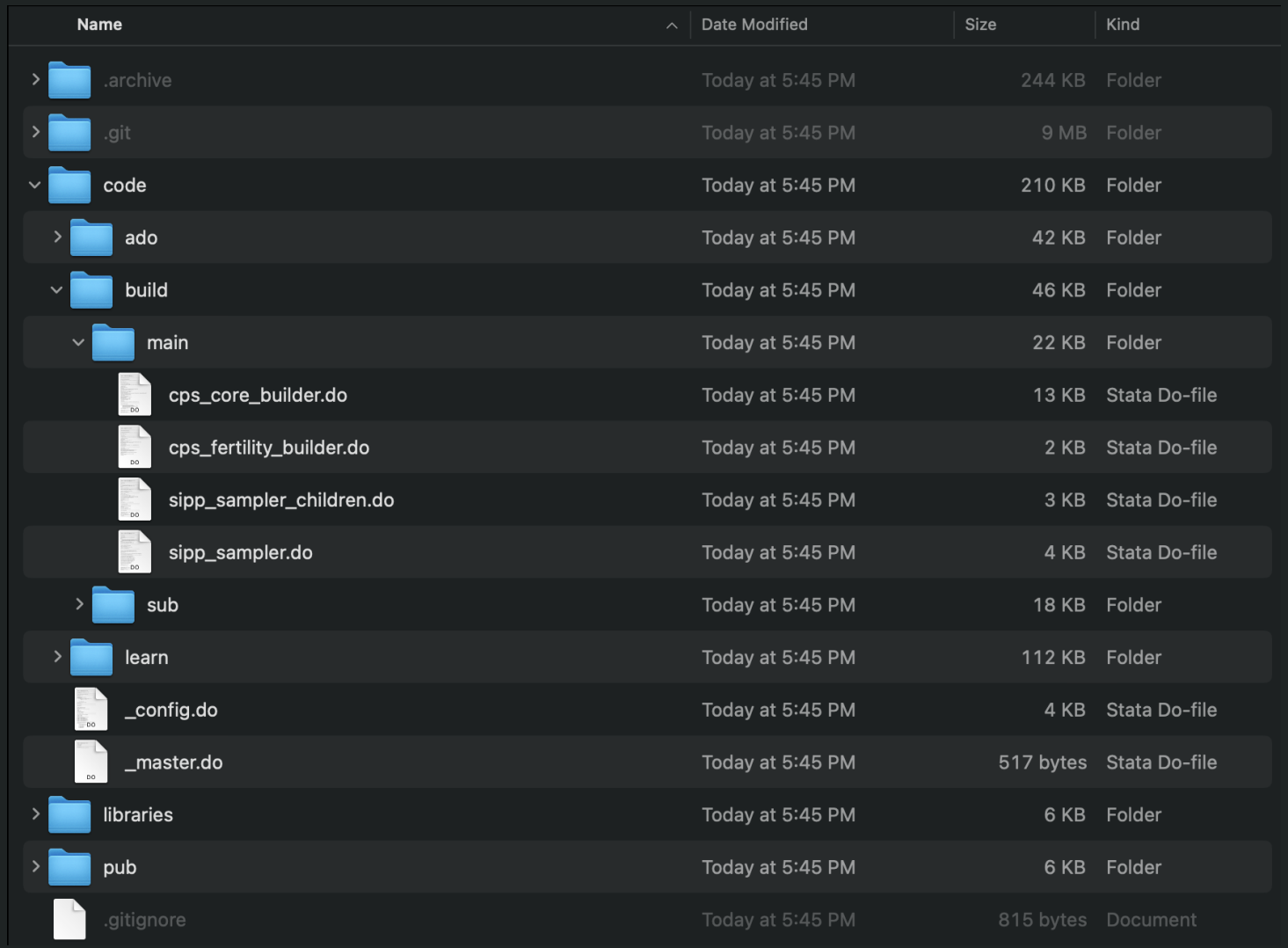
Inspect or revert to old versions as needed

A sample Git log

```
* | 6a4a6dc 2020-12-03 21:53:43 Merge branch 'develop'
|\|
| * 2199522 2020-12-03 21:41:20 Add analysis to guide our definition of mothers vs. non-mothers
| * 9bdcdba 2020-12-03 21:13:45 Update _main.do to execute additional .do files
| * 8897ac4 2020-12-03 21:10:06 Extend summer_drop.do sample period (1976-2019, instead of 1976-2017)
| * accbfe3 2020-12-03 20:32:52 Add code to construct our baseline, prime-age CPS estimation sample
* | 080cb98 2020-12-03 19:13:54 Merge branch 'develop' into 'master'
|\|
| * 082c567 2020-12-02 22:17:02 Assess detrending via restricted cubic splines
| * 0b22ab2 2020-12-02 18:29:32 Drop child observations with invalid personal identifiers
| * 8d80e66 2020-12-02 12:27:44 Fix bug in filepath for style files
* | d80bb5c 2020-12-02 16:44:34 Merge branch 'develop' into 'master'
|\|
| * b2d2382 2020-12-02 10:52:14 Add code to plot CPS variables over time
| * 549c090 2020-12-02 10:40:53 Add Stata style files
| * 9b62305 2020-12-01 22:31:34 Update cps_fertility_builder.do
| * 38f0e16 2020-12-01 22:16:19 Update _config.do to make code/ado/ssc/ if needed
| * 68f0379 2020-12-01 22:00:16 Update code to clean CPS basic monthly files
| * 19c6a23 2020-12-01 13:57:17 Incorporate updated CPS extracts
|/
* dd23431 2020-11-30 17:15:20 Change file nomenclature
* 44b9b49 2020-11-30 16:31:31 Add "replace" option to ssc install commands
* 81a4c0f 2020-09-25 20:55:45 (tag: v0.0) Incorporate Davis-era codebase
* 274d918 2020-06-14 22:20:21 Adapting old analyses to the new build
* ab72ae2 2020-06-14 20:37:45 Renaming folders, updating some files
* 071a6c1 2020-03-16 10:07:30 Updating CPS cleaning code (in progress)
* eded3e5 2020-03-07 22:59:38 Basic configuration
* 7d14d78 2020-03-07 21:40:22 Initial commit: utilities and file structure
```

Source: Price and Wasserman (2022).

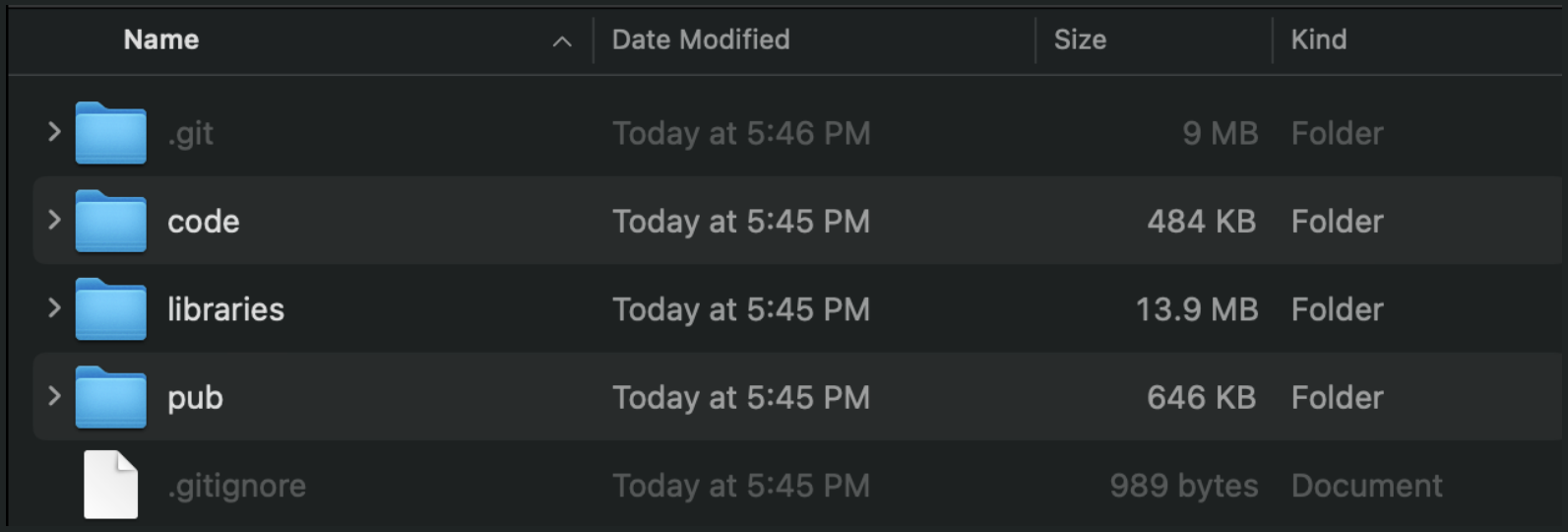
Meet Mr. 81a4c0f



Name	Date Modified	Size	Kind
.archive	Today at 5:45 PM	244 KB	Folder
.git	Today at 5:45 PM	9 MB	Folder
code	Today at 5:45 PM	210 KB	Folder
ado	Today at 5:45 PM	42 KB	Folder
build	Today at 5:45 PM	46 KB	Folder
main	Today at 5:45 PM	22 KB	Folder
cps_core_builder.do	Today at 5:45 PM	13 KB	Stata Do-file
cps_fertility_builder.do	Today at 5:45 PM	2 KB	Stata Do-file
sipp_sampler_children.do	Today at 5:45 PM	3 KB	Stata Do-file
sipp_sampler.do	Today at 5:45 PM	4 KB	Stata Do-file
sub	Today at 5:45 PM	18 KB	Folder
learn	Today at 5:45 PM	112 KB	Folder
_config.do	Today at 5:45 PM	4 KB	Stata Do-file
_master.do	Today at 5:45 PM	517 bytes	Stata Do-file
libraries	Today at 5:45 PM	6 KB	Folder
pub	Today at 5:45 PM	6 KB	Folder
.gitignore	Today at 5:45 PM	815 bytes	Document

Source: Price and Wasserman (2022).

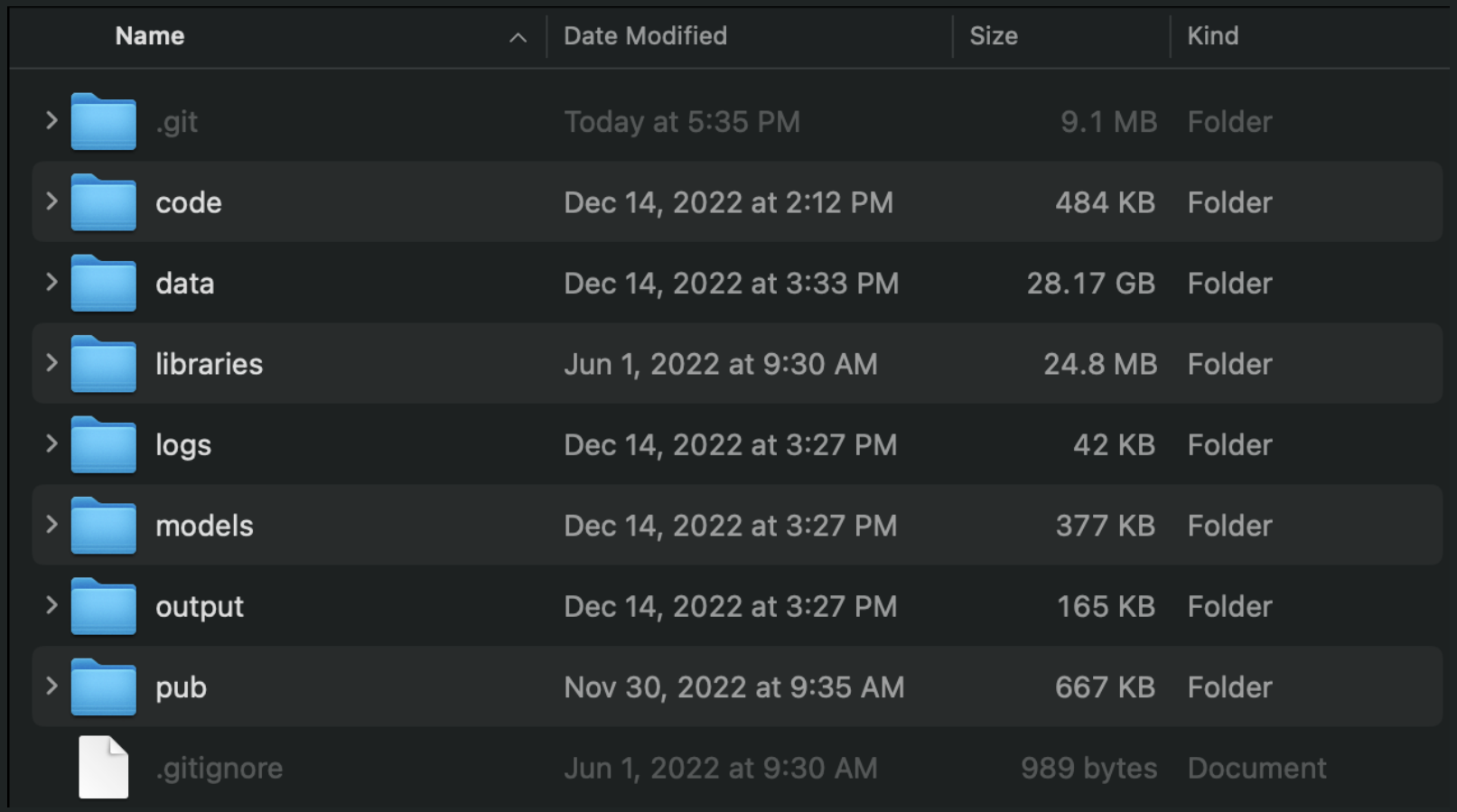
The latest version: tracked files only



Name	Date Modified	Size	Kind
.git	Today at 5:46 PM	9 MB	Folder
code	Today at 5:45 PM	484 KB	Folder
libraries	Today at 5:45 PM	13.9 MB	Folder
pub	Today at 5:45 PM	646 KB	Folder
.gitignore	Today at 5:45 PM	989 bytes	Document

Source: Price and Wasserman (2022).

The latest version: including derived files



Name	Date Modified	Size	Kind
.git	Today at 5:35 PM	9.1 MB	Folder
code	Dec 14, 2022 at 2:12 PM	484 KB	Folder
data	Dec 14, 2022 at 3:33 PM	28.17 GB	Folder
libraries	Jun 1, 2022 at 9:30 AM	24.8 MB	Folder
logs	Dec 14, 2022 at 3:27 PM	42 KB	Folder
models	Dec 14, 2022 at 3:27 PM	377 KB	Folder
output	Dec 14, 2022 at 3:27 PM	165 KB	Folder
pub	Nov 30, 2022 at 9:35 AM	667 KB	Folder
.gitignore	Jun 1, 2022 at 9:30 AM	989 bytes	Document

Source: Price and Wasserman (2022).

The perks of version control

Main benefit: code retrieval

- Organized archiving
- Convenient backups

Side benefit: lineage tracing

- Find all edits made to a given file
- Trace file, folder renames

Made for collaboration

- Work independently, merge together
- Link discussions to specific commits
- Share repos with the research community

Version control $\Longrightarrow$ better code

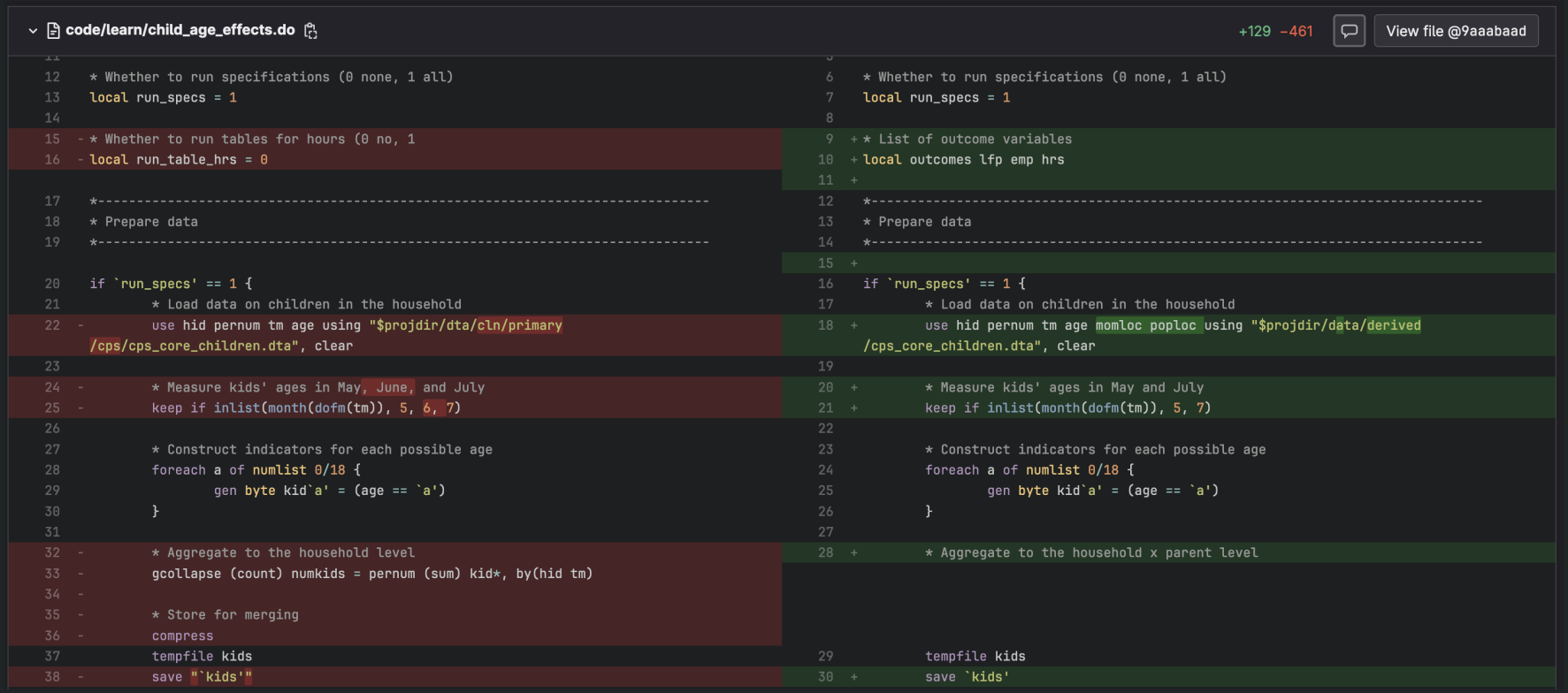
It promotes leanness

- Delete whatever you don't need now
- Easy to rerun code in a fresh clone

It disciplines your coding

- Workflow encourages modular thinking
- Easy to review edits before committing
- Easy to reverse mistakes

The power of the diff



code/learn/child_age_effects.do +129 −461 View file @9aaabaad

```
 * Whether to run specifications (0 none, 1 all)
local run_specs = 1

- * Whether to run tables for hours (0 no, 1
- local run_table_hrs = 0

*-------------------------------------------------------------------------------
* Prepare data
*-------------------------------------------------------------------------------

if `run_specs' == 1 {
        * Load data on children in the household
-       use hid pernum tm age using "$projdir/dta/cln/primary
/cps/cps_core_children.dta", clear

-       * Measure kids' ages in May, June, and July
-       keep if inlist(month(dofm(tm)), 5, 6, 7)

        * Construct indicators for each possible age
        foreach a of numlist 0/18 {
                gen byte kid`a' = (age == `a')
        }

-       * Aggregate to the household level
-       gcollapse (count) numkids = pernum (sum) kid*, by(hid tm)
-
-       * Store for merging
-       compress
        tempfile kids
-       save "`kids'"
```

```
 * Whether to run specifications (0 none, 1 all)
local run_specs = 1

+ * List of outcome variables
+ local outcomes lfp emp hrs
+
*-------------------------------------------------------------------------------
* Prepare data
*-------------------------------------------------------------------------------
+
if `run_specs' == 1 {
        * Load data on children in the household
+       use hid pernum tm age momloc poploc using "$projdir/data/derived
/cps_core_children.dta", clear

+       * Measure kids' ages in May and July
+       keep if inlist(month(dofm(tm)), 5, 7)

        * Construct indicators for each possible age
        foreach a of numlist 0/18 {
                gen byte kid`a' = (age == `a')
        }

+       * Aggregate to the household x parent level




        tempfile kids
+       save `kids'
```

Source: Price and Wasserman (2022).

How to learn Git

Git is well worth learning

- Hugely helpful for writing a dissertation
- Widely used in academia, policy, industry

Learning curve is a bit steep

- Start slow, stick with it
- Try it out in a solo project
- Over time: learn new tricks

Lots of good resources online

- But befriend an emergency contact

Postscript

Brueghel the Elder, *The Land of Cockaigne*

The big picture

Main message: think (hard) about workflow

Invest early in good habits

- Be organized
- Find ways to improve
- Figure out what works for you

Don't go it alone

- Talk to your classmates
- Read people's code
- Get the help you need

"Break any of these rules sooner than say anything outright barbarous."—George Orwell, *Politics and the English Language*

Further reading

Coding:

- Code and Data for the Social Sciences: A Practitioner's Guide, Matthew Gentzkow and Jesse Shapiro
- Coding for Economists, Ljubica Ristovska
- Coding Style Guide, Michael Stepner
- Best Practices for Computer Programming in Economics, Tal Gross

Version control:

- Git for Economists, Frank Pinter
- Pro Git, Scott Chacon and Ben Straub
- Version Control with Git, Jon Loeliger and Matthew McCullough

Plenty more at www.brendanmichaelprice.com