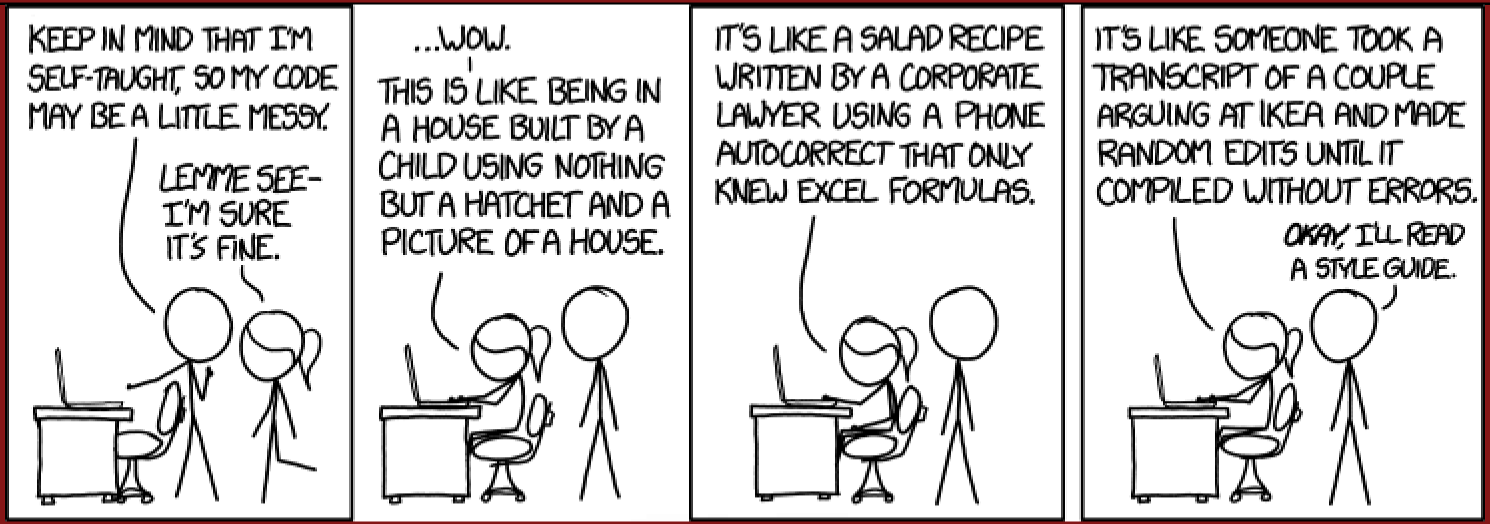
Big Data and Economics

The Empirical Workflow and Clean Code

Kyle Coombs (adapted from Tyler Ransom + Scott Cunningham)
Bates College | EC/DCS 368

Table of contents

Prologue



Source: xkcd

Forgot to mention

- **Office Hours:**
  - My office hours are 9am-10am on Tuesdays and 3pm-4pm on Wednesdays
  - My office is 276 Pettengill
  - I'm also available by appointment on Zoom
- **Problem Set 0:** due on Sunday, September 17th at 11:59pm

Attribution

- Today's material comes from these sources:

1. Clean Code by Tyler Ransom

2. *Code and Data for the Social Sciences: A Practitioner's Guide*, by Gentzkow and Shapiro

3. Causal Inference and Research Design by Scott Cunningham

4. Jenny Bryan's UseR 2018 keynote address

Also a small contribution from here and other sundry internet pages

Jargon

- There is a jargon in this class that won't make sense at first, I'll try to flag it as it comes
  - If I don't flag a term, look it up on ChatGPT
  - If it still doesn't make sense, ask me -- could be I'm using it idiosyncratically
- Here's a few terms:
  - **Local machine:** Your personal (or any) computer that isn't a server accessed via the internet
  - **Version Control:** Keep track of different iterations of a project/code
  - **Repository:** The location on GitHub of all project files and (commented) file revision history
  - **GUI:** A Graphical User Interface -- what you're used to pointing and clicking to navigate a computer and execute programs
  - **Command line:** Removes the "graphical" from GUI, instead you type all commands to navigate a computer and execute programs
    - R operates via the Command line, RStudio is a GUI
    - On Mac, this is called Terminal
    - Windows has Powershell, but it Powershell uses quite user-unfriendly commands
    - If you installed Git for Windows, you got *Git Bash*, which uses Bash (Linux) commands
    - You can also install Windows Subsystem for Linux to run Linux on a Windows machine

Clean Code

Reducing empirical chaos

Sad story

- Once upon a time there was a boy who was writing a job market paper on unemployment insurance during the pandemic
- This boy presented the findings a half dozen times, spoke to the media some, and generally thought he had cool results
- Several people suggested he look at a handful of other outcome series and try changing his analysis unit frequency from monthly to weekly
- He also knew that he needed to restrict his sample to reduce noise

The horror!

- But then after making these changes and re-running his code that took two days, his new sample dropped by 50 percent!
- He was, understandably, terrified.
- The young boy spent a week looking for the fix weeding through six different versions of the .do, .R, .dta, .csv, .sh, .py files with suffixes like *_v1* and *_test* and *_test2* and *_final_I_swear* and *_okay_i_lied*
- Finally he discovered the phrase:

```
df %>% filter(insample_new==0)
```

instead of

```
df %>% filter(insample_new==1)
```

- The boy was very frustrated and decided to work on these slides while re-running his code.

What is Clean Code?

- **Clean Code:** Code that is easy to understand, easy to modify, and hence easy to debug
- Clean code saves you and your collaborators time

Why clean code matters: Scientific

- Good science is based on careful observations
- Science progresses through iteratively testing hypotheses and making predictions
- Scientific progress is impeded if
  - mistaken previous results are erroneously given authority
  - previous hypothesis tests are not reproducible
  - previous methods and results are not transparent
- Thus, for science that involves computer code, clean code is a must

Why clean code matters: Personal and

- You will always make a mistake while coding
- What makes good programmers great is their ability to quickly identify and correct mistakes
- Developing a habit of clean coding from the outset of your career will help you more quickly identify and correct mistakes
- It will save you a lot of stress in the long-run
- It will make your collaborative relationships more pleasant

Why clean code is under-produced

- If clean code is so beneficial and important, why isn't there more of it?

1. **Competitive pressure** to produce research/products as quickly as possible
2. **End user** (journal editor, reviewer, reader, dean) **doesn't care what the code looks like**, just that the product works
3. In the moment, clean code **takes longer to produce** while seemingly conferring no benefit

How does one produce clean code?

Principles

How does one produce clean code?

- Automation
- Version control
- Organization of data and software files
- Abstraction
- Documentation
- Time / task management
- Test-driven development (unit testing, profiling, refactoring)
- Pair programming

Automation

- Gentzkow & Shapiro's two rules for automation:

1. Automate everything that can be automated

2. Write a single script that executes all code from beginning to end

- There are two reasons automation is so important

    - Reproducibility (helps with debugging and revisions)

    - Efficiency (having a code base saves you time in the future)

- A single script that shows the sequence of steps taken is the equivalent to "showing your work"

Version control

- We've discussed Git and GitHub in a previous slide deck
- Version control provides a principled way for you to easily undo changes, test out new specifications, and more

File organization

1. Separate directories by function

2. Separate files into inputs and outputs

3. Make directories portable

- To see how professionals do this, check out the source code for R's dplyr package

  - There are separate directories for source code (`/src`), documentation (`/man`), code tests (`/test`), data (`/data`), examples (`/vignettes`), and more

- When you use version control, it forces you to make directories portable (otherwise a collaborator will not be able to run your code)

  - use **relative** file paths, not absolute file paths

Data organization

- The key idea is to practice **relational data base management**
- A relational database consists of many smaller data sets
- Each data set is tabular and has a unique, non-missing key
- Data sets "relate" to each other based on these keys
- You can implement these practices in any modern statistical analysis software (R, Stata, SAS, Python, Julia, SQL, ...)
- Gentzkow & Shapiro recommend not merging data sets until as far into your code pipeline as possible

Abstraction

- What is abstraction? It means "reducing the complexity of something by hiding unnecessary details from the user"
- e.g. A dishwasher. All I need to know is how to put dirty dishes into the machine, and which button to press. I don't need to understand how the electrical wiring or plumbing work.
- In programming, abstraction is usually handled with functions
- Abstraction is usually a good thing
- But it can be taken to a harmful extreme: overly abstract code can be "impenetrable" which makes it difficult to modify or debug

Rules for Abstraction

- Gentzkow & Shapiro give three rules for abstraction:

1. Abstract to eliminate redundancy

2. Abstract to improve clarity

3. Otherwise, don't abstract

Abstract to eliminate redundancy

- Sometimes you might find yourself repeating lines of code with small modifications across the lines:

```
names <- c('one','two','three','four','five','one','two','three','four','five','one','two','three','four','five')

#Better
names_short <- c('one','two','three','four','five')
names_long <- c(names_short,names_short,names_short)

#Even better
name_repeater <- function(count,names_short=c('one','two','three','four','five')) {
    names_long <- rep(names_short, times = count)
    return(names_long)
}

print(names)
```

```
##  [1] "one"   "two"   "three" "four"  "five"  "one"   "two"   "three" "four"
## [10] "five"  "one"   "two"   "three" "four"  "five"
```

```
print(names_long)
```

```
##  [1] "one"   "two"   "three" "four"  "five"  "one"   "two"   "three" "four"
## [10] "five"  "one"   "two"   "three" "four"  "five"
```

```
print(name_repeater(3,names_short=names_short))
```

```
##  [1] "one"   "two"   "three" "four"  "five"  "one"   "two"   "three" "four"
## [10] "five"  "one"   "two"   "three" "four"  "five"
```

- Now if I need to make further changes to `name_repeater` I can do it once!

More complicated example

```
set.seed(16)
prod1  = rnorm(1, 0, 1)*rnorm(1,4,6)
prod2  = rnorm(2, 0, 1)*rnorm(2,4,6)
prod3  = rnorm(3, 0, 1)*rnorm(3,4,6)
print(prod1)
```

```
## [1] 1.547257
```

```
print(prod2)
```

```
## [1] 11.934479 -1.717951
```

```
print(prod3)
```

```
## [1] -7.4831177  0.9587218  4.7882622
```

```
set.seed(16)
multiply = function(count,mean1=0,sd1=1,mean2=4,sd2=6) {
    prod = rnorm(count,mean1,sd1)*rnorm(count,mean2,sd2)
    return(prod)
}
prod1=multiply(1)
prod2=multiply(2)
prod3=multiply(3)

print(prod1)
```

```
## [1] 1.547257
```

```
print(prod2)
```

```
## [1] 11.934479 -1.717951
```

Note on seeds

- When randomizing in any language, you aren't really randomizing
- You're producing pseudo-random numbers that return in a deterministic ordered list
- If you set the seed, you can reproduce the same "random" numbers
- This is useful for debugging and sharing code
- Use `set.seed` in R

Neat R functions to help reduce

```r
set.seed(16)
list1 = list() # Make an empty list to save output in
for (i in 1:3) { # Indicate number of iterations with "i"
    list1[[i]] = multiply(i) # Save output in list for each iteration
}
list1
```

```
## [[1]]
## [1] 1.547257
##
## [[2]]
## [1] 11.934479 -1.717951
##
## [[3]]
## [1] -7.4831177  0.9587218  4.7882622
```

A better way to eliminate this redundancy is to use the `map` function:

```r
set.seed(16)
map(1:3, multiply)
```

```
## [[1]]
## [1] 1.547257
##
## [[2]]
## [1] 11.934479 -1.717951
##
## [[3]]
## [1] -7.4831177  0.9587218  4.7882622
```

Abstract to improve clarity

- Consider the example of obtaining OLS estimates from a vector `y` and covariate matrix `X` that already exist on our workspace

- We could code this in two ways:

```
Bhat = (t(X)%*%X)^(-1)%*%t(X)%*%y
```

or

```
estimate_ols <- function(yvar, Xmat) {
    Bhat = (t(Xmat)%*%Xmat)^(-1)%*%t(Xmat)%*%yvar
    return(Bhat)
}
estimate_ols(y,X)
```

The second approach is easier to read and understand what the code is doing

Otherwise, don't abstract

- One could argue that the examples on the previous two slides are overly abstract
- OLS is a simple operation that only takes one line of code
- If we're only doing it once in our script, then it may not make sense to use the function version
- Similarly, it may not make sense to use the `name_repeater` function if I only need to use it to repeat five names three times
- This discussion points out that it can be difficult to know if one has reached the optimal level of abstraction
- As you're starting out programming, I would advise doing almost every inside of a function (i.e. err on the side of over-abstraction when starting out)

Documentation

1. Don't write documentation you will not maintain

2. Code should be self-documenting

- Generally speaking, commented code is helpful
- However, sometimes it can be harmful if, e.g. code comments contain dynamic information
- It may not be helpful to have to rewrite comments every time you change the code
- Code can be "self-documenting" by leveraging abstraction: function arguments make it easier to understand what is a variable and what is a constant

Documentation in R

- R has excellent built-in documentation called `Roxygen2`
- These make great documents above functions to increase readability
- Here's an example:

```
#' This is a sample function
#'
#' This function does something amazing.
#'
#' @param x A numeric input.
#' @return The result of the amazing operation.
#' @examples
#' amazing_function(5)
amazing_function <- function(x) {
  # function implementation
}
```

Other documentation in R

- **R Help System:** access using `?function_name`
- **Package vignettes:** access using `vignette("vignette_name")`
- **Cheatsheets:** access at Posit Cheatsheets

Time management

- Time management is key to writing clean code
- It is foolish to think that one can write clean code in a strained mental state
- Code written when you are groggy, overly anxious, or distracted will come back to bite you
- Schedule long blocks of time (1.5 hours - 3 hours) to work on coding where you eliminate distractions (email, social media, etc.)
- Stop coding when you feel that your focus or energy is dissipating

Task management

- When collaborating on code, it is essential to not use email or Slack threads to discuss coding tasks
- Rather, use a task management system that has dedicated messages for a particular point of discussion (bug in the code, feature to develop, etc.)
- I use GitHub issues for all of my coding projects
- For my personal task management, I use Trello to take all tasks out of my email inbox and put them in Trello's task management system
- GitHub and Trello also have Kanban-style boards where you can easily visually track progress on tasks

Workflow workflow workflow

The Cunningham Empirical Workflow Conjecture

- The cause of most of your errors is **not** due to insufficient knowledge of syntax in your chosen programming language
- The cause of most of your errors is due to a poorly designed **Empirical Workflow**

Empirical Workflow

- A workflow is a fixed set of routines you bind yourself to which when followed identifies the most common errors
  - Think of it as your morning routine: alarm goes off, go to wash up, make your coffee/tea, put pop tart in toaster, contemplate your existence in the universe until **ding**, eat pop tart repeat *ad infinitum*
- Finding the outlier errors is a different task; empirical workflows catch typical and common errors created by the modal data generating processes
- Empirical workflows follow a checklist

Why do we use checklists?

- I got engaged in July am planning a wedding in Princeton for next July
- I also moved to New England in August and am still unpacking
  - Extra weird I live part-time in MA with my fiance
- I am teaching two upper-level electives
- I am trying to submit several papers to conferences/journals this year
- Each of these tasks gets a checklist:
  - Wedding: ☐ Finalize tent configuration ☐ Pick wedding colors
  - Unpacking: ☐ Put books on shelves ☐ Buy dresser
  - Big Data: ☐ Prep GitHub demo ☐ Create presentations repo
  - Public Economics: ☐ Update solutions for PS1 ☐ Amend 2nd Welfare Theorem slides
  - etc.

To remember the obvious stuff you keep

- When I stop to think, I know I need to do everything on my checklists
- But then I forget when I move onto the next task
- Programming is the same, except you have an **empirical checklist**:
- The **empirical checklist**:
    - Covers the intermediate step between "getting the data" and "analyzing the data"
    - It largely focuses on ensuring data quality for the most common, easy to identify problems
    - It'll make you a better coauthor

Simple data checks

Look at the data

- Simple, yet non-negotiable, programming commands and exercises to check for data errors
- "Real eyes realize real lies" --Troy Ave vai some dude from my high school
- Let's look at a dataset with self-reported salaries (and other info) put together by Alison Green
- Note: This is an actively growing dataset -- the timestamps help show that!

```r
bp <- read.csv('data/messier_bp.csv')
bp
```

```
##    STOP.Blood.Pressure.Study            X2        X3         X4         X5
## 1                       <NA>          <NA>      <NA>       <NA>       <NA>
## 2                     pat_id Month of birth Day birth Year birth       Race
## 3                          1            11        30       1967      White
## 4                          2            12        12       1990  Caucasian
## 5                          3             5         4       1989      White
## 6                          3             3         8       1977      Other
## 7                          4             6        14       1963      Black
## 8                          5             4        19       1973      Black
## 9                          6             1        23       1986      White
## 10                         7            11         1       1956      Asian
## 11                         8             4        17       1961      Other
## 12                         8            12        24       1967      White
## 13                        11             9        27       1971      White
## 14                        12             5        16       1970  Caucasian
## 15                        13             7        18       1963      Black
## 16                        14             3         3       1989      Black
## 17                        15            12         8       1990      Asian
## 18                        16            11        13       1987      WHITE
## 19                        17             6        27       1980      WHITE
## 20                        19             4        11       1975      Black
## 21                        20             2         7       1967  Caucasian
```

Check factor variables

```
table(bp$race,bp$sex)
```

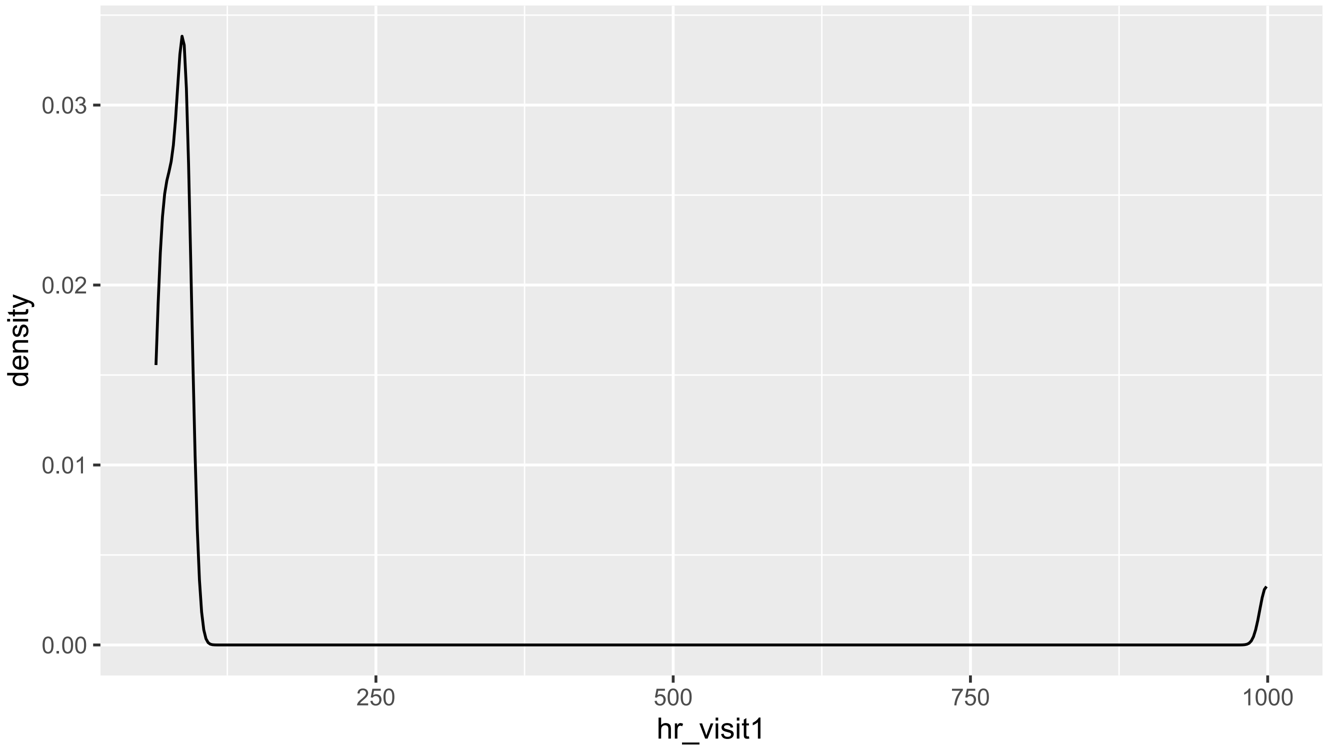
```
##
##              F  Female  M  Male
##   Asian      0       1  0     1
##   Black      1       2  1     3
##   Caucasian  0       3  0     1
##   Other      0       1  0     1
##   White      0       2  0     2
##   WHITE      0       1  0     1
```

Visualize the data

- Go beyond the eyeball and graph the data

```
# Get the first three rows of the data frame (or as many rows as needed)

#Make a density of the heart rate on visit 1:
ggplot(data=bp,aes(x=hr_visit1))+geom_density()
```



What might be going on here?

Other tricks:

- Check if the data are the right-size
- If you have a panel dataset is 50 states over 20 years, check if there are 1000 observations
- If not, find out why! Maybe there are 1020 because DC is (rightfully) included
- Search for outliers or oddities and work out possible explanations using:
    - Codebooks
    - Intuition
    - Emails to the source/creator of data

Test-driven development (unit testing,

- The only way to know that your code works is to test it!
- Test-driven development (TDD) consists of a suite of tools for writing code that can be automatically tested
- **unit testing** is nearly universally used in professional software development
- Unit testing is to software developers what washing hands is to surgeons

Unit testing

- Unit tests are scripts that check that a piece of code does everything it is supposed to do
- When professionals write code, they also write unit tests for that code at the same time
- If code doesn't pass tests, then bugs are caught on the front end
- Test coverage determines how much of the code base is tested. High coverage rates are a must for unit testing to be useful.
- R's dplyr package shows that all unit tests are passing and that tests cover 88% of the code base
- testthat is a nice step-by-step guide for doing this in R

Assertions

- Assert statements are extremely useful
- They exist in every langage
- In R it is called stopifnot()

```r
x <- TRUE
stopifnot(x)

y <- FALSE
stopifnot(y)
```

```
## Error: y is not TRUE
```

Refactoring

- Refactoring refers to the action of restructuring code without changing its external behavior or functionality. Think of it as "reorganizing"

- Example:

after refactoring becomes

- Nothing changed in the code except the number of characters in the function

- The new version may run faster, is more readable. The output is unchanged.

- Refactoring could also mean reducing the number of input arguments

- Jenny Bryan gave a great talk on refactoring

Profiling

- Profiling refers to checking the resource demands of your code
- How much processing time does your script take? How much memory?
- Clean code should be highly performant: it uses minimal computational resources
- Profiling and refactoring go hand in hand, along with unit testing, to ensure that code is maximally optimized
- Here is an intro guide to profiling in Julia using the `@time` macro

Pair programming

- An essential part of clean code is reviewing code
- An excellent way to review code is to do so at the time of writing
- **Pair programming** involves sitting two programmers at one computer
- One programmer does the writing while the other reviews
- This is a great way to spot silly typos and other issues that would extend development time
- It's also a great way to quickly refactor code at the start
- **I strongly encourage you to do pair programming on problem sets in this course!**
    - (Sometimes I will require it)

Appendix

Textbooks: Smarter people than me

- Cunningham (2021) Causal Inference: The Mixtape (Also, free version on his website)
- Huntington-Klein (2022) The Effect
- Angrist and Pischke (2009) Mostly Harmless Econometrics (MHE)
- Morgan and Winship (2014) Counterfactuals and Causal Inference (MW)
- Sweigart (2019) Automate The Boring Stuff With Python

- The help documentation associated with your language (no really)
- Jesse Shapiro's "How to Present an Applied Micro Paper"
- Gentzkow and Shapiro's coding practices manual
- Lubica "LJ" Ristovska's language agnostic guide to programming for economists
- Grant McDermott on Version Control using Github Link

- The help documentation associated with your language (no really)
- All languages: Stack Overflow, Stack Exchange
- Stata-specific (all hail Nick Cox): Statalist
- Cheatsheets: Stata, RStudio, Python
- Me: Sign up for office hours

- Just like learning a real language, no amount of talking today will teach you how to use any program.
  - You have to need to use it (immersion) to learn it.
  - Google is your dictionary.

Next lecture: Hidden Researcher Decisions