

Big Data and Economics

Lecture 2b: Clean Code

Kyle Coombs (adapted from Tyler Ransom + Scott Cunningham)

Bates College | [EC/DCS 368](#)

Table of contents

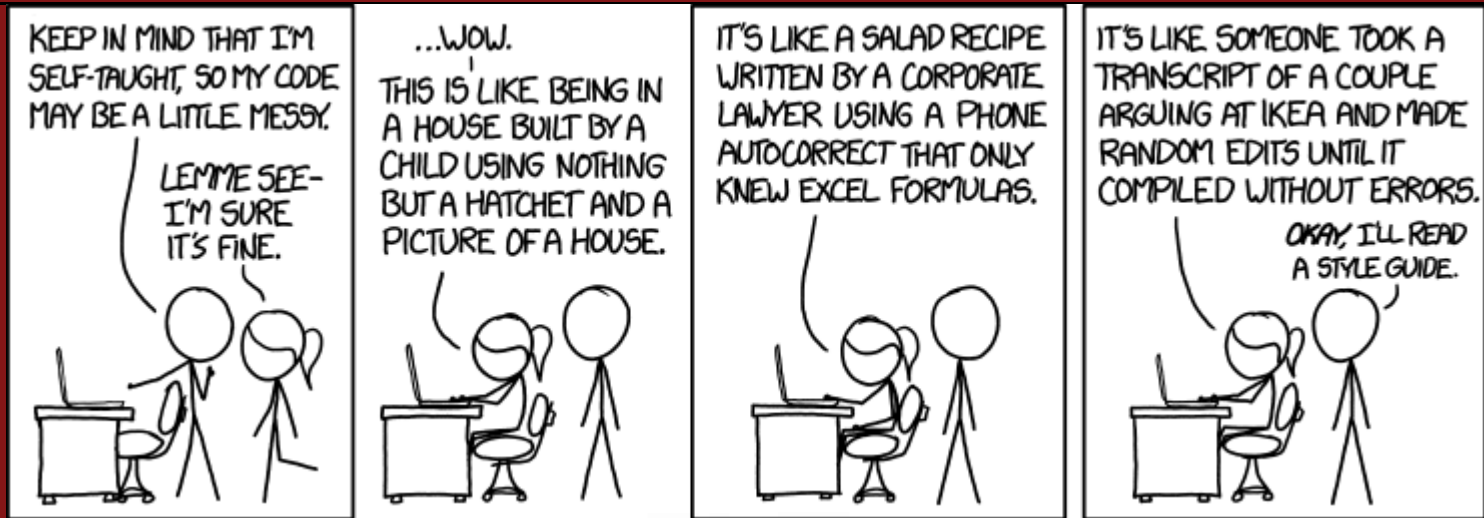
1. Prologue

2. 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

3. Appendix: FAQ

Prologue



Source: [xkcd](#)

Housekeeping

- **Presentations:** Sign-up in the [Presentations github repository](#)
- **Problem Set 1:** due on Sunday, January 29th at 11:59pm
- **Final Project Proposal:** due on Sunday, January 25th at 11:59pm
 - Create a fork of the Final Project repository and add me as a collaborator
 - List the names of you and your partner in the README.md file

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

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.
- Today and next class are about minimizing these struggles through Clean Code and a reproducible workflow

Student Presentation

Hidden Researcher Decisions

- Bottom line: empirical work is full of little arbitrary decisions
- These add up quickly
- It does not necessarily mean anything nefarious is going on or that the research is wrong
- But it underscores why we should be skeptical of empirical work
- And why we should work to clearly document any empirical work we do
- And why replication efforts by [University of Goettingen](#) or [datacolada](#) are so important
- Similarly why the American Economic Association's [Data and Code Availability Policy](#) matters so much

Clean Code

What is Clean Code?

Clean Code: Code that is easy to understand, easy to modify, and hence easy to debug

Clean code advances scientific progress

- Good science uses careful observations to iteratively test hypotheses/make 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
- Reduces "the influence of hidden researcher decisions" (Huntington-Klein et al. 2021)

Clean code increases personal/team sanity

- You will always make a mistake while coding -- what makes good programmers great is their ability to quickly identify and correct mistakes
- Clean code makes it easier to identify and correct mistakes
- Saves you stress in the long-run and makes 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?

1. Automation
2. Version Control¹
3. Organization of data and software files
4. Abstraction
5. Documentation
6. Time / task management
7. Test-driven development (unit testing, profiling, refactoring)
8. Pair programming

¹ Skipped today cause we covered it last class.

1. 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"

How to write scripts

Keep them modular

- Each script should do one thing and one thing only
- e.g. It takes an input in, it returns an output
 - Taking in a raw file and returning a cleaned version
 - Taking in two files and merging them
 - Taking in a cleaned file and returning a figure

Have a main script that runs all scripts in order

- This is the script that you run to reproduce your results
- You will rarely run it all at once, but it will be a nice way to organize your thoughts
- This is a further benefit of a well-organized directory -- you can easily see what scripts you need to run in what order
- Use `source('rscript.R')` to run an external script
- A main script could be a `.Rmd`, a `.R`, a `.sh`, a `.py`, a `.do` etc.

Main script

```
#File: main.Rmd or main.R
#By: Kyle Coombs
#What: Runs the project from start to finish in Python
#Date: 2023/09/12

#Install packages with housekeeping. Also put together paths.
source('housekeeping.R')
#User written functions can be sourced -- or you could write a package, your call
source(paste0(build, 'clean_functions.R'))
source(paste0(analysis, 'analysis_functions.R'))

#Import files
source(paste0(build, 'import_census.R'))
source(paste0(build, 'import_admin_data.R'))

#Clean files
source(paste0(build, 'clean_census.R'))
source(paste0(build, 'clean_admin_data.R'))

#Merge files 1 to 2
source(paste0(build, 'merge_census_admin.R'))

#Analysis
source('analysis/summary_stats.R')
source('analysis/basic_regression.R')

#Tables will likely be made with a host of R packages
source('analysis/make_sum_figures.R')
source('analysis/make_reg_figures.R')
source('analysis/make_sum_tables.R')
source('analysis/make_reg_tables.R')
```

Main script with functions

Main script as .Rmd

- In this class, your problem sets will be .Rmd files that you knit to PDF/HTML
- The .Rmd file will serve as your main script
- You can `source()` modular code files in code chunks
- PS1 will show you examples of doing this
- This guarantees your code runs from start to finish instead of only when you are working interactively

What's a housekeeping file?

A housekeeping file automates several tasks and goes at the start of every file in your project

```
# Housekeeping.R
# By: Your Name
# Date: YYYY-MM-DD
# What: This script loads the packages and data needed for the analysis.

## Package installation -- uncomment if running for the first time
#install.packages(c('here', 'tidyverse'))
library(here)
library(tidyverse)
library(haven)

## Directory creation

here::i_am('housekeeping.R')

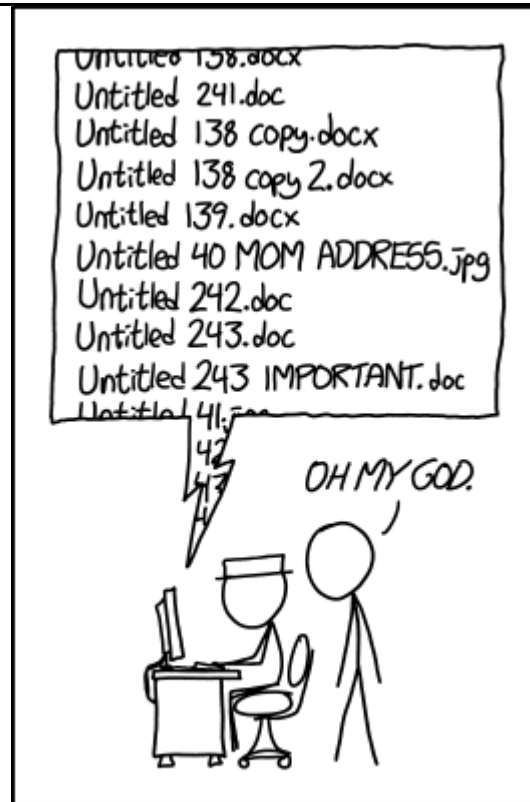
data_dir <- here::here('data')
raw_dir <- here::here(data_dir, 'raw')
clean_dir <- here::here(data_dir, 'clean')
output_dir <- here::here('output')
code_dir <- here::here('code')
processing_dir <- here::here(code_dir, 'processing')
analysis_dir <- here::here(code_dir, 'analysis')
documentation_dir <- here::here('documentation')

suppressWarnings({
  dir.create(data_dir)
  dir.create(raw_dir)
  dir.create(clean_dir)
  dir.create(documentation_dir)
  dir.create(code_dir)
  dir.create(processing_dir)
  dir.create(analysis_dir)
  dir.create(output_dir)
})
```

3a. 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

Don't be like this



PROTIP: NEVER LOOK IN SOMEONE
ELSE'S DOCUMENTS FOLDER.

Source: [xkcd](#)

What is a directory?

- All the files on your computer are organized in directories or folders
- When you are running a script, you are running it from a particular directory
 - This is *not necessarily* the directory where the script is located
 - It is the directory that your console is in
 - That means if you say `read.csv('my_data.csv')`, your computer looks for `my_data.csv` in that particular directory
 - If that file is not in that directory, you will get a `FileNotFound` error
 - In **R**, you can see what directory you are in using the `getwd()` function
 - It is also above the console in RStudio
 - You can change your working directory using the `setwd()` function

```
getwd()  
## [1] "C:/Users/kgcsp/OneDrive/Documents/Education/Big Data/big-data-class-materials/lectures/02-empirical-workflow"  
#setwd('lectures/02-empirical-workflow')
```

What is a directory path?

A path defines the location of a file or directory in a file system tree.

If I navigate to this file in my computer, the path is `C:\Users\kgcsp\OneDrive\Documents\Education\Big Data\big-data-class-materials\lectures\02-empirical-workflow\02-empirical-workflow.Rmd`

The name separates folders that chart the path from the **root** to the file

- **root**: the start of the file system tree (above that is `c:\`)
- Each folder along the tree is separated by a `\` or `/`

This is called an **absolute path**:

- It is long
- It is hard to remember
- It is not portable -- if I send this file to you, it won't work on your computer

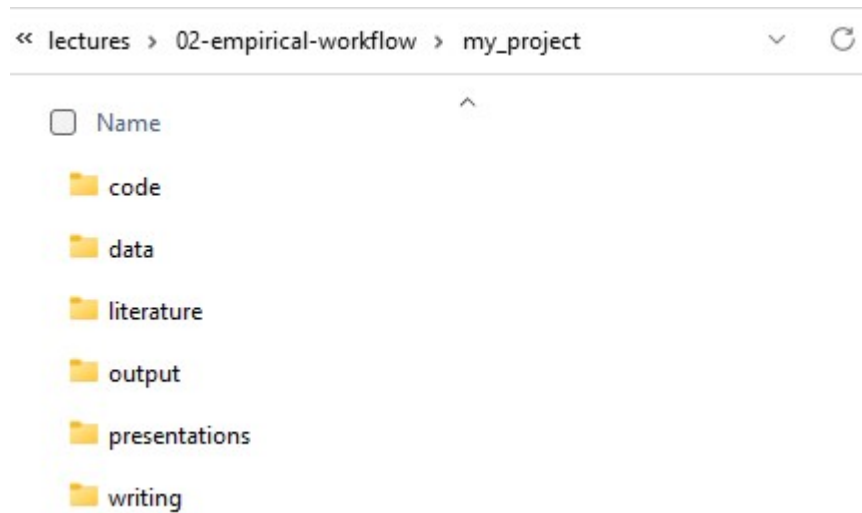
Relative paths solve a lot of this:

- The path to a file or directory starting from the current working directory
- If my current working directory is `/big-data-class-materials`, then I can use `lectures/02-empirical-workflow/02-empirical-workflow.Rmd`
- **This is portable** -- if I send this file to you and you have a copy of the `big-data-class-materials` repository on your computer, it will work on your computer

How I organize research projects

- Entire projects should *ideally* live within the same directory
- I have a folder called (`my_project`)
 - Within that folder I have subfolders:
 1. `data` for all data files a. `raw` for raw data files b. `clean` or `work` for cleaned data files c. `temp` for temporary data files
 2. `code` for all code files, and sometimes: a. `code/analysis` for code files that build/clean code a. `code/build` for code files that do analysis
 3. `output` for all output files a. `output/figures` for code files that make figures b. `output/tables` for code files that make tables
 4. `literature` or `articles` for all relevant literature
 5. `writing` for all writing files a. `writing/notes` for notes b. `writing/drafts` for drafts c. `writing/edits` for edits
 6. `presentations` for all presentations a. `presentations/slides` for slides b. `presentations/notes` for notes
- I'll further more or less as needed
- See GitHub folder for this lecture as an example
 - I also include a script `make_directory.sh` that automates this process

How I organize research projects



Source: My computer

What is the value of directories?

- All of the files in a directory are related to each other
- Can reference a file within the `data/raw` folder, from the `code/build` folder without writing out the full path `C:/Users/kylec/Documents/my_project/data/raw/my_data.csv`
- Can save objects of strings of path directories to use later using the `paste()` function

```
my_project <- 'my_project'
data <- paste(my_project, 'data', sep='/')
data_raw <- paste(data, 'raw', sep='/')
data_clean <- paste(data, 'clean', sep='/')
data_temp <- paste(data, 'temp', sep='/')
code <- paste(my_project, 'code', sep='/')
code_analysis <- paste(code, 'analysis', sep='/')
code_build <- paste(code, 'build', sep='/')

print(paste(data_raw, 'my_data.csv', sep='/'))
```

```
## [1] "my_project/data/raw/my_data.csv"
```

```
read.csv(paste(data_raw, 'my_data.csv', sep='/'))
```

```
##   this is my data
## 1    1  1  1  1
## 2    2  2  2  2
```

- This is a good way to make sure that your code is portable
- If you move your project to a different computer, you can just change the `my_project` variable and all the other paths will update automatically

Alternative to all the pastes is here()

- Better yet is the [here](#)
 - `here()` will find the root directory of your project and then you can navigate from there

```
#install.packages('here')  
library(here)
```

```
## here() starts at C:/Users/kgcsp/OneDrive/Documents/Education/Big Data/big-data-class-materials
```

```
here::i_am('my_project/code/build/.placeholder')
```

```
## here() starts at C:/Users/kgcsp/OneDrive/Documents/Education/Big Data/big-data-class-materials/lectures/02-empirical-workflow
```

```
here('data/raw', 'my_data.csv')
```

```
## [1] "C:/Users/kgcsp/OneDrive/Documents/Education/Big Data/big-data-class-materials/lectures/02-empirical-workflow/data/raw/my_data.csv"
```

- Can be less clunky than `paste()` and `sep="/"`
- Get lost in your directories? Use `here::here()` to identify your root directory
- Alternatively, double-click the `.Rproj` file to be redirected to the root directory of your project folder

Help! I am in code/, but I need

- You can use relative paths to navigate between directories
- `..` means "go up one directory"
 - `../data/raw` means "go up one directory, then down into `data/raw`"
- `.` means "stay in the current directory"
 - `./code/build` means "stay in the current directory, then down into `code/build`"
- `../..` means "go up two directories"
 - `../../data/raw` means "go up two directories, then down into `data/raw`"

Play around with them yourself!

3b. 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

What problems would this create?

county	state	cnty_pop	state_pop	region
36037	NY	3817735	43320903	1
36038	NY	422999	43320903	1
36039	NY	324920	.	1
36040	.	143432	43320903	1
.	NY	.	43320903	1
37001	VA	3228290	7173000	3
37002	VA	449499	7173000	3
37003	VA	383888	7173000	4
37004	VA	483829	7173000	3

Source: [Code and Data for the Social Sciences](#) (p. 19)

What's RDBM look like?

county	state	population			
36037	NY	3817735			
36038	NY	422999			
36039	NY	324920	state	population	region
36040	NY	143432	NY	43320903	1
37001	VA	3228290	VA	7173000	3
37002	VA	449499			
37003	VA	383888			
37004	VA	483829			

Source: [Code and Data for the Social Sciences](#) (p. 19)

4. 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
- In the context of R, abstraction means:
 - Write functions
 - Name your objects sensibly

Abstract to eliminate redundancy

- Sometimes you might find yourself repeating lines of code to accomplish a task

```
# Downloading a sequence of files from 2004 to 2020 gets tedious
download.file('https://data.nber.org/tax-stats/zipcode/2020/zipcode2020.zip',destfile=paste0(data_dir,'zipcode2020.zip',sep='/'))
download.file('https://data.nber.org/tax-stats/zipcode/2019/zipcode2019.zip',destfile=paste0(data_dir,'zipcode2019.zip',sep='/'))
download.file('https://data.nber.org/tax-stats/zipcode/2018/zipcode2018.zip',destfile=paste0(data_dir,'zipcode2018.zip',sep='/'))
# etc.
```

Notice any problems?

```
# Downloading a sequence of files from 2004 to 2020 with a loop
for (y in 2004:2020) {
  download.file(paste0('https://data.nber.org/tax-stats/zipcode/',y,'/zipcode',y,'.zip'),destfile=paste0(data_dir,'zipcode',y,'.zip'))
}
```

- We'll learn more about iteration later with the apply family in R
- There are many other forms of redundancy that can be eliminated with abstraction beyond iteration
- This is just the simplest to understand

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  
Bhat2 = (t(X)%*%X2)^(-1)%*%t(X2)%*%y
```

or

```
estimate_ols <- function(yvar, Xmat) {  
  Bhat = (t(Xmat)%*%Xmat)^(-1)%*%t(Xmat)%*%yvar  
  return(Bhat)  
}  
Bhat = estimate_ols(y,X)  
Bhat2 = estimate_ols(y,X2)
```

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 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
- 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)

5. 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

A README is documentation

- A README gives high-level information about the repository or data file:
 - This repository contains code that does X task
 - Simple use case: use this repository to replicate paper X in journal Y
- Onboarding instructions:
 - Add your name to this file in repository folder `the/folder/file.md`
 - Fork the repository and pull request changes
 - Configure your computer settings in this way to run this project
 - Guidelines/rules for contributing to the project
- Licensing information:
 - You can just take this code!
 - This is proprietary and we will sue you if you haven't paid us
- Dependencies:
 - To use this code or package or data, download packages `x`, `y`, `z`
- Changelog (short narrative commit history):
 - 9/23/2023 - KGC - added function `x` to do `y`

Documentation in R

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

Make your own documentation

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

```
library(roxygen2)
#' 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
}
```

- Use `roxygen::roxygenise()` to generate documentation for all functions in a file
- Read more [here](#)

6a. Time management

- Time management is key to writing clean code²
- 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

² Your professor needs this lecture too

6b. 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

7. Test-driven development

- 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
- Simplest test is to check if the code gives you the output you expected
- More complicated is to write a unit test
- **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 immediately
- 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 for basic unit tests
- They exist in every language
- In R it is called `stopifnot()`

```
x ← TRUE  
stopifnot(x)  
  
y ← FALSE  
stopifnot(y)
```

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

Troubleshooting tips

- Sometimes you've made several changes to your code and suddenly it stops running
 - Was it the new `if` statement?
 - That sick new vectorized function to replace the `for` loop?
 - A stray typo?
- How do you find the bug in hundreds of lines of code?
- Read your code to see if there is an obvious mistake
- **Binary search:** Comment¹ half your code, run the script, and see if the bug persists
 - If it does, the bug is in the other half
 - If it doesn't, the bug is in the commented half
 - Use `#` to comment out lines of code in R, or highlight and press `Ctrl+Shift+C`
- Repeat on each half until you narrow to set of lines
- If you can solve the bug from that line, great!
- If not, make a **Minimal reproducible example!**

¹ Comment in R with `#`. Comment in RMarkdown with `<!-- code -->`. Or highlight and press `Ctrl+Shift+C` in RStudio.

Minimal reproducible example (MRE)

- There's likely a ton of superfluous stuff in your code that is not relevant to the error
- **Minimal reproducible examples** (reprex) are a great way to isolate the error
 - **Minimal:** Use as little code as possible that still produces the same problem
 - **Complete:** Provide all parts needed to reproduce your problem in the question itself
 - **Reproducible:** Test the code you'll provide to make sure it reproduces the problem
- That means you should be able to copy and paste the code into R and run it yourself
 - Name all packages and data needed to reproduce error
 - Cut out irrelevant packages, steps, and data that are not relevant to the error
- Sometimes writing one will help you find the bug, sometimes it'll help a stranger find the bug in your code faster, and sometimes it'll identify a very real bug in the package itself
- MREs also help you **refactor** and **profile** your code

Min Reprex from RStudio community

- If someone does not have `hrbrthemes` installed, they will not be able to run your code.
 - You can remove this package from your code and still reproduce the error.

```
library(ggplot2) #For ggplot
library(datasets) #To load iris
library(hrbrthemes) #For the theme
data(iris)
df <- iris %>%
  mutate(Sepal.Length = Sepal.Length * 1000,
         Sepal.Width = Sepal.Width * 1000)

ggplot(data = df, x = Sepal.Length, y = Sepal.Width) +
  theme_modern_rc() +
  geom_point() +
  scale_x_log10() +
  labs(title = "Iris Sepal Width vs. Sepal Length",
       subtitle = "Log10 Scaled X Axis")
```

```
## Error in `geom_point()`:
## ! Problem while setting up geom.
## i Error occurred in the 1st layer.
## Caused by error in `compute_geom_1()`:
## ! `geom_point()` requires the following missing aesthetics: x and y
```

How to write MREs

- Cut out the unnecessary steps

```
library(ggplot2)

df <- data.frame(stringsAsFactors = FALSE,
                  Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5),
                  Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6)
)

ggplot(data = df, x = Sepal.Length, y = Sepal.Width) +
  geom_point()
```

```
## Error in `geom_point()`:
## ! Problem while setting up geom.
## i Error occurred in the 1st layer.
## Caused by error in `compute_geom_1()`:
## ! `geom_point()` requires the following missing aesthetics: x and y
```

```
#> Error: geom_point requires the following missing aesthetics: x, y
```

- You can use [reprex](#) to make sure that your code is reproducible by others.
- You can use [dput](#) to make sure that your data is reproducible by others.

Troubleshooting tips (cont.)

- Step back and ask if you're solving the right problem
 - e.g. I'm trying to make a plot, but I'm getting an error about a missing variable. Maybe I should check if I'm loading the right data
 - e.g. I have to create a long data set and I have annual files, but my code is merging instead of appending...
- Check for superfluous things you can remove
 - e.g. Wait, I don't need to include absolute file paths, I can use relative paths
 - Bonus: I'll make fewer typos!
- Try small fixes, then apply broadly
 - e.g. I think the problem is with how I wrote my file paths, let me try to get just one file path to work
- Change one thing at a time
 - e.g. The problem is either with my `paste0()` statement or the `ggsave` function, let me try to get the `paste0()` statement to work first

Troubleshooting tips (cont.)

- Embrace GitHub committing
 - When you have code that works, stage, commit and push it -- even if it is only a small piece of the puzzle
 - If it breaks, [revert](#)
 - This minimizes how much you need to re-do/keep track of
- Sometimes it is easier to change things on your side than it is to force a programming language to work a certain way
 - e.g. Rmarkdown doesn't like the character `#` in filepaths, but I can change the filepaths rather than trying to force Rmarkdown to accept it
- There's more than one way to skin a cat
 - e.g. If I can't get `read.csv()` to work, I'll try `read.table()`
 - e.g. This `googlesheets4` package doesn't seem to work -- what about `gsheet` or `googledrive`?
- With ChatGPT or Google, make very specific asks
 - e.g. "How do I get a file named `/my/path/name/my_file.pdf` into `other/folder/name/file.Rmd`?"

8. Pair programming - work with a buddy

- 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)

Next lecture: R basics, data wrangling,
tidyverse and data.table

Appendix

Main script with functions

name: main-with-functions

```
#File: main.Rmd or main.R
#By: Kyle Coombs
#What: Runs the project from start to finish in Python
#Date: 2023/09/12

#Install packages with housekeeping. Also put together paths.
source('housekeeping.R')
#User written functions can be sourced -- or you could write a package, your call
source(paste0(build,'clean_functions.R'))
source(paste0(analysis,'analysis_functions.R'))

#Import files
df1 <- read_csv(paste0(raw,'file1.csv'))
df2 <- read_parquet(paste0(raw,'file2.parquet'))
df3 <- read_dta(paste0(raw,'file3.dta'))

#Clean files
cleaned_df1 <- clean_df1(df1)
cleaned_df2 <- clean_df2(df2)
cleaned_df3 <- cf.clean_df3(df3)

#Merge files 1 to 2
merged_df1_df2 = merge(cleaned_df1, cleaned_df2, on=c('merge','vars'))

#Append file 1 to
append_df1_df2_df3 = rbind(merged_df1_df2, cleaned_df2)

#Analysis
sum_stats=summary_stats(append_df1_df2_df3,stats=c('mean','median','max'))
reg_results=basic_regression(append_df1_df2_df3)

#Tables will likely be made with a host of R packages
make_sum_figures(sum_stats)
make_figures(reg_results)
make_sum_tables(sum_stats)
make_tables(reg_results)
```

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](#)
- Wickham (2023) [Advanced R](#)
- Wickham and Grolemund (2023) [R for Data Science](#)
- Peng (2022) [R Programming for Data Science](#)

Non-textbook readings

- 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
- Ljubica "LJ" Ristovska's language agnostic guide to programming for economists
- Grant McDermott on Version Control using Github [Link](#)

Helpful for troubleshooting

- 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](#)

Learn by Immersion

- 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.
 - Help files are your grammar books.
 - ChatGPT is your phrasebook.
 - A great way to start coding is to see lots of other people's code and copy what you read.
- You must learn how to ask the “right” question:
 - Never: "Importing csv file into R not working."
 - Better: "read_csv R [specific error message]."
 - Better still: "read_csv tidyverse [specific error message]."

Abstract to eliminate redundancy (cont.)

What if you can't find an R function? Write your own!

```
set.seed(16)
prod1 = rnorm(1, 0, 1)*rnorm(1,4,6)
prod2 = rnorm(2, 0, 1)*rnorm(2,0,1)
prod3 = rnorm(3, 0, 1)*rnorm(3,15,78)
print(prod1)
## [1] 1.547257
print(prod2)
## [1] 1.2582691 0.6764943
print(prod3)
## [1] -60.06036 10.11156 24.32342
```

```
set.seed(16)
multiply_normals = function(count,mean1=0,sd1=1,mean2=0,sd2=1) {
  prod = rnorm(count,mean1,sd1)*rnorm(count,mean2,sd2)
  return(prod)
}
prod1=multiply_normals(1,mean2=4,sd2=6)
prod2=multiply_normals(2,mean2=0,sd2=1)
prod3=multiply_normals(3,mean2=15,sd2=78)

print(prod1)
## [1] 1.547257
print(prod2)
## [1] 1.2582691 0.6764943
print(prod3)
## [1] -60.06036 10.11156 24.32342
```

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

```
set.seed(0)
print(rnorm(1)+rnorm(1,5)+rnorm(1,10))
## [1] 17.26652
print(rnorm(1)+rnorm(1,5)+rnorm(1,10))
## [1] 15.14712
# New seed
set.seed(1)
print(rnorm(1)+rnorm(1,5)+rnorm(1,10))
## [1] 13.72156
print(rnorm(1)+rnorm(1,5)+rnorm(1,10))
## [1] 16.10432
# Reset seed
set.seed(0)
print(rnorm(1)+rnorm(1,5)+rnorm(1,10))
## [1] 17.26652
print(rnorm(1)+rnorm(1,5)+rnorm(1,10))
## [1] 15.14712
```

Refactoring

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

```
get_some_data ← function(config, outfile) {  
  if (config_ok(config)) {  
    if (can_write(outfile)) {  
      if (can_open_network_connection(config)) {  
        data ← parse_something_from_network()  
        if(makes_sense(data)) {  
          data ← beautify(data)  
          write_it(data, outfile)  
        }  
      }  
    }  
  }  
}
```

after refactoring becomes

```
get_some_data ← function(config, outfile) {  
  if (config_bad(config)) {  
    stop("Bad config")  
  }  
  
  if (!can_write(outfile)) {  
    stop("Can't write outfile")  
  }  
}
```

- 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 are two intro guides to profiling in R:
 - Using `system.time` and `Rprof` from R Programming for Data Science[<https://bookdown.org/rdpeng/rprogdatascience/profiling-r-code.html>]
 - Using `lineprof` from Advanced R[<http://adv-r.had.co.nz/Profiling.html>]

[Back to MREs](#)

Neat R functions to help reduce

```
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:

```
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
```