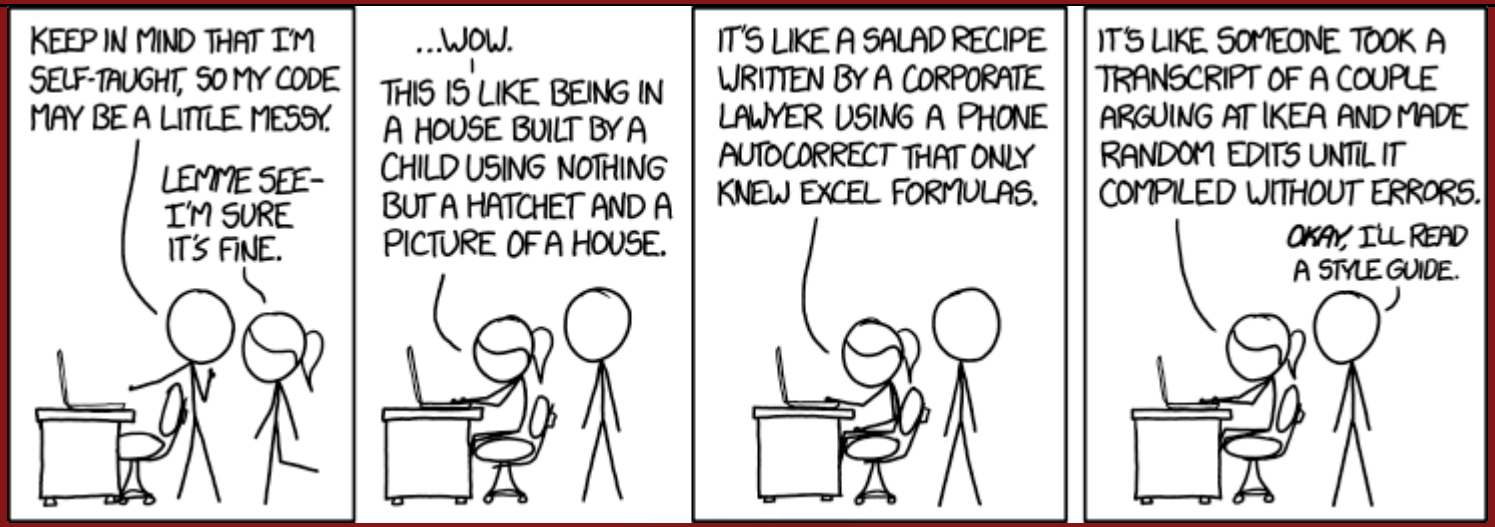
Data Science for Economists

Lecture 2b: Clean Code

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

Table of contents

Prologue



Source: xkcd

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

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

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

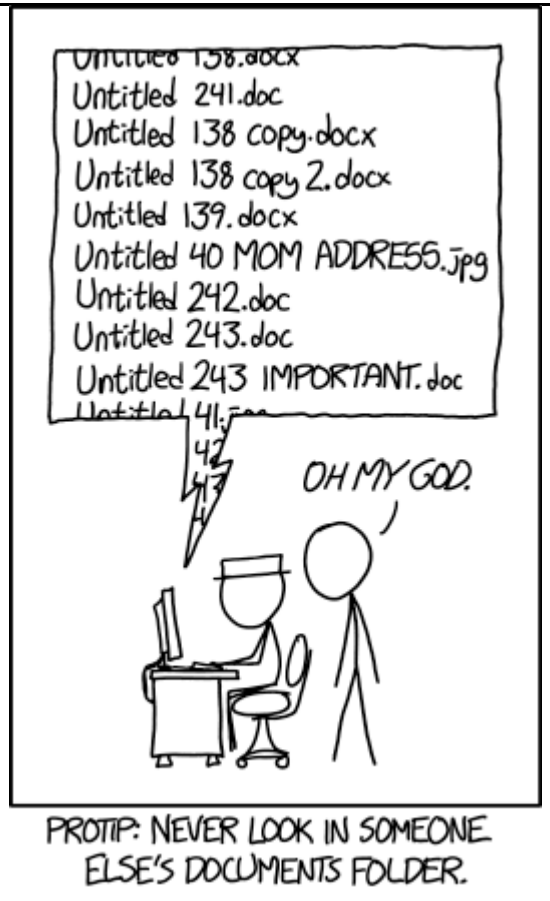
[1] Handled in Git lecture notes.

1a. 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



Source: xkcd

What is a working directory?

- All the files on your computer are organized in directories or folders
- When you are running a script, you are working from a particular directory
  - This is *not necessarily* the directory where the script is located
  - Your computer looks for `my_data.csv` in this directory when you execute `read.csv('my_data.csv')`
  - 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 double click the `.Rproj` file to set the working directory to the root of the project
  - You can also change your working directory using the `setwd()` function (avoid this within scripts)

```
getwd()
```

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

```
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, it is:

`C:\Users\kgcsp\OneDrive\Documents\Education\ECON368-DSE\big-data-class-materials\lectures\02-empirical-workflow\02-empirical-workflow.Rmd`[1]

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, hard to remember, and not portable across computers

Relative paths solve a lot of this:

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

[1] This is a Windows path, Mac and Linux paths use `/` instead of `\`. See appendix for slides on how to move between them using `..`

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 the `my_project` folder on GitHub (in the same folder as this lecture) as an example

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
- If you use `file.path()` or the **here** package, you can automate the slashes in your file paths
  - `file.path('data', 'raw', 'file.csv')` will work on Windows, Mac, and Linux
  - `here::here('data', 'raw', 'file.csv')` will do the same thing (see appendix for more on it)
- Then you do not need to worry about shifting around directories

1b. 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_state region_county
## 1   36037    NY  3817735  43320903            1             1
## 2   36038    NY   422999  43320903            1             1
## 3   36039    NY   324920        NA            1             1
## 4   36040  <NA>   143432  43320903            1             1
## 5      NA    NY       NA  43320903            1             1
## 6   37001    VA  3228290   7173000            3             3
## 7   37002    VA   449499   7173000            3             3
## 8   37003    VA   383888   7173000            3             4
## 9   37004    VA   483829   7173000            3             3
## 10     NA    VA       NA   7173000            3             3
```

Why is RDBM better?

```
##   county state cnty_pop
## 1  36037    NY  3817735
## 2  36038    NY   422999
## 3  36039    NY   324920
## 4  36040    NY   143432
## 5  37001    VA  3228290
## 6  37002    VA   449499
## 7  37003    VA   383888
## 8  37004    VA   483829
```

```
##   state state_pop region
## 1    NY  43320903      1
## 2    VA   7173000      3
```

Source: Example from Code and Data for the Social Sciences (p. 19)

3. 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)

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
- This is somewhat aligned with the structure of an essay
    - intro paragraph =/= body paragraph 1 =/= ... =/= conclusion
- Much like essays revisions, modular code makes it easier to debug and revise

Have a main script that runs all scripts in order

- A single script that shows the sequence of steps, i.e. "shows your work"
    - This script will run modular scripts in sequence to exactly reproduce your analysis
- You will rarely run it all at once, but it will be a nice way to organize your thoughts
- A benefit of a well-organized directory: easily see what scripts you need to run in what order

```r
#File: main.R.R
#By: Kyle Coombs
#What: Runs the project from start to finish in Python
#Date: 2024-09-0
# Instructions:
# Run this code from the root directory of your project

#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(paste0(analysis,'/summary_stats.R'))
source(paste0(analysis,'/basic_regression.R'))

#Tables will likely be made with a host of R packages
source(paste0(analysis,'/make_sum_figures.R'))
```

Main script with functions

Main script as .Rmd

- In this class, your problem sets will be `.Rmd` files that you knit
- The `.Rmd` file will serve as your main script
- You can `source()` modular code files in code chunks
- It improves chances your code runs from start to finish instead of only when working interactively
    - Means I can run (and grade) your code more easily!

What's a housekeeping file?

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

1. Load packages
2. Save strings of path directories to use later using the `file.path()` function to reference elsewhere[1]
   - If a folder name changes, you only need to change it in one place in your code
   - Use these strings to reference files in other scripts
     `read.csv(file.path(data_raw,'my_data.csv'))`
3. Create directories if they don't exist

[1] Alternatives to `file.path()` include `paste` and `here()`. Check appendix example for more information.

```r
# 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('tidyverse'))
library(tidyverse)
library(haven) # installed by tidyverse

## Directory objects
data_dir ← file.path('data')
raw_dir ← file.path(data_dir,'raw')
clean_dir ← file.path(data_dir,'clean')
output_dir ← file.path('output')
code_dir ← file.path('code')
processing_dir ← file.path(code_dir,'processing')
analysis_dir ← file.path(code_dir,'analysis')
documentation_dir ← file.path('documentation')

# Create directories
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)
})
```

4. Abstraction

- **Abstraction:** "reducing the complexity of something by hiding unnecessary details from the user"
- e.g. A dishwasher. I mainly need to know how to load it, put in soap, and press start. I don't need to understand the electrical wiring or plumbing.
- In programming, abstraction is usually handled with functions
- Abstraction is usually a good thing
- But it you can go too far: overly abstract code can be "impenetrable" and difficult to modify or debug

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,
download.file('https://data.nber.org/tax-stats/zipcode/2019/zipcode2019.zip',destfile=paste0(data_dir,
download.file('https://data.nber.org/tax-stats/zipcode/2019/zipcode2019.zip',destfile=paste0(data_dir,
# etc.
```

Notice any problems?

```
# Downloading a sequence of files from 20 wih a loop
lapply(2004:2020,function(y) {
    download.file(paste0('https://data.nber.org/tax-stats/zipcode/',y,'/zipcode',y,'.zip'),destfile=pas
})
```

- We'll learn more about iteration/for loops/appy statements later
- There are many forms of redundancy that can be eliminated with abstraction beyond iteration

Abstract to improve clarity

Say you want to round a number to the nearest of different integers:

1. Divide the number by there base integer
2. Round the result to the nearest whole number
3. Multiply by the base integer

I start coding and copy and paste the code for each integer:

```
rounded_157_nearest_5 ← round(157/5)*5
rounded_157_nearest_7 ← round(157/5)*7
```

Notice a problem?

Why not abstract with a function?

```
round_to_nearest ← function(x,base=5) {
    return(round(x/base)*base)
}

rounded_157_nearest_5 ← round_to_nearest(157,base=5)
rounded_157_nearest_7 ← round_to_nearest(157,base=7)
```

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

Otherwise, don't abstract

1. Write/use functions for tasks that are repeated
2. Write thoughtful variable names (e.g. `x100, x101` versus `household_income`, `household_size`)

- 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 everything inside of a function (i.e. err on the side of over-abstraction when starting out)
    - And look for opportunities to loop (or use apply functions)

5. Documentation

Documentation gives sufficient information to replicate work, but not so much that it is overwhelming[1]

Rules for 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 naming functions and variables thoughtfully

Documentation in R

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

[1] Anyone who has ever built IKEA furniture knows this all too well

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_dir download packages `X`, `Y`, `Z`
- Changelog (short narrative commit history):
  - 9/23/2023 - KGC - added function `X` to do `Y`

Documentation and problem sets

Documentation inevitably creates a host of issues on assignments.

It is challenging to give narrative technical instructions:

On a blank problem set:

- The reader (you) still needs to engage thoughtfully with the task
- The writer (me) needs to account for many misinterpretations!

On a completed problem set:

- The reader (me) is trying to guess what you were thinking
- The writer (you) may have made a mistake and not realized it

This challenge is a feature, not a bug.

My assignments are a learning experience of the robustness principle/Postel's law[1] (for people):

> "Be conservative in what you send, be liberal in what you accept." - Jon Postel

He was talking about internet protocols, but I see it as a general principle for communication:

- Conservative: Make instructions instructions as clear s
- Liberal: Give the benefit of the doubt and try to engage thoughtfully with documentation

[1] This quote was originally in reference to how to design programs that send and receive data.

6. Time and task management

Time management

- Time management is key to writing clean code[1]
- 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, avoid 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 and milestones for all of my coding projects including developing this class

[1] Your professor needs this lecture too

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
    - Whenever you make a change, check it against the output you expect
    - Ideally, check against a small example so it runs fast and is easy to confirm
- What if the code takes too long to check completely? Meet **unit tests**
- **Unit testing** is nearly universally used in professional software development

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 91% of the code base
- testthat is a nice step-by-step guide for doing this in R (I use it to autograde exercises)

Assertions

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

```
x ← TRUE
stopifnot(x)

y ← FALSE
stopifnot(y)
```

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

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)

Minimal reproducible example[1]

- Related to unit testing are minimal reproducible examples (aka MRE, reprex, minreps,...)
- The best way to isolate bugs is a minimal reproducible example
- If code throws an error, there's likely superfluous lines of code that are irrelevant to the error
    - The superfluous stuff makes it harder to read and replicate 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 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

[1] Postel's Law in action.

Min Reprex from RStudio community

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

```
library(ggplot2) #For ggplot
library(datasets) #To load irs
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)
dat <- iris[1:4,]
ggplot(data = dat, 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.
```

- You can use reprex to make sure that your code is reproducible by others and dput to make sure that your data is reproducible by others.

```
dput(iris[1:4,]) # copy/paste output of dput into your MRE
```

```
## structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6), Sepal.Width = c(3.5,
## 3, 3.2, 3.1), Petal.Length = c(1.4, 1.4, 1.3, 1.5), Petal.Width = c(0.2,
## 0.2, 0.2, 0.2), Species = structure(c(1L, 1L, 1L, 1L), levels = c("setosa",
## "versicolor", "virginica"), class = "factor")), row.names = c(NA,
## 4L), class = "data.frame")
```

A complete MRE[1]

Summary

When I try to make a scatterplot with ggplot, I am told that `geom_point()` requires missing aesthetics `x` and `y`. But I specified `x` and `y` in the `ggplot()` function. Can you help resolve?

Expected behavior

I expected the code to produce a scatterplot of `Sepal.Length` and `Sepal.Width` from the iris dataset.

Data

I used a subset of the iris dataset.

```
dat ← iris[1:4,]
```

Code and error message

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

```
## Error in `geom_point()`:
## ! Problem while setting up geom.
```

[1] Every forum has its own approach to MREs. Sometimes session info is not initially needed!

Try to write an MRE!

- Sync your fork of the exercise repository and open the folder for mre-exercise
- The file `mre-opatlas.Rmd` has a bug in it that has led to a host of problems when you look at the knit output, `mre-opatlas.md`
- Try to write an MRE
- I have already raised this as a poorly-written issue on GitHub. You can see the issue here
- Tips: https://aosmith16.github.io/spring-r-topics/slides/week09_reprex.html#1

Appendix

Shifting directories

Help! I need to run code from `code`, but need a file in `data/raw/file.csv`!

- 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!

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
```

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
```

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

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)) {
```

after refactoring becomes

```
get_some_data ← function(config, outfile) {
  if (config_bad(config)) {
    stop("Bad config")
  }

  if (!can_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 `Rprofs` 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
##
```

Alternative to file.path 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/0
```

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

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

- Can be less clunky than `paste()` and `sep="/"`
- Get lost in your directories? Use `file.path()` to identify your root directory