

# Lecture 1: An Introduction & Motivation for R Programming + Installing Packages and Reading Data

Harris Coding Camp – Accelerated Track

Summer 2022

# Welcome to Coding Camp!

- ▶ Why are we here?
- ▶ What are we going to do?
- ▶ A quick introduction to R and RStudio
- ▶ How to start working with data

# Teaching Members

- ▶ Instructors
  - ▶ Standard Track: Arthur Cheib, Sheng-Hao Lo
  - ▶ Accelerated Track: Ari Anisfeld
- ▶ Head TA: Rubina Hundal
  - ▶ All logistics issues
- ▶ You will also have several TAs who will be helping you along the way!
  - ▶ TA sessions
  - ▶ Canvas Discussion Board

# Why learn coding?

- ▶ Computation is an essential party of modern-day applied statistics and *quantitative* policy analysis
- ▶ Many public policy jobs and the Harris curriculum rely on programming
  - ▶ to quickly engage with policy data
  - ▶ to complete statistical analyses
- ▶ Examples
  - ▶ Determine how many people are eligible for debt forgiveness
  - ▶ Analyze changes in test scores among different groups due to online education
  - ▶ What policy problems do you want to tackle with data?

# Gauging your background

Most of you have some data experience. What are you bringing?

- ▶ Excel / Sheets
- ▶ Stata
- ▶ R
- ▶ Python
- ▶ C++? Julia? SPSS? SAS? Other software / languages?
- ▶ Excited for a challenge?

# Why R?

- ▶ R is a powerful programming language and statistical software environment
  - ▶ Great data manipulation and visualization suite
  - ▶ Strong statistical packages (e.g. program evaluation, machine learning)
- ▶ Open source and free
- ▶ Complete programming language with low barriers to entry
- ▶ We will use R for the entire Stats sequence in Fall and Winter

# What will we cover?

0. Motivation/Installation of R *[Today]*
  1. Installing Packages and Reading Data *[Today]*
  2. Basic Data Manipulation and Analysis *[3]*
  3. Data Visualization *[1]*
  4. More on Data Manipulation *[3]*
    - ▶ Grouped Analysis, Iteration, Functions
- ▶ In Stats 1/2 and other courses, you will build off of these lessons:
- ▶ extend your capabilities with the functions we teach you
  - ▶ introduce statistics functions
  - ▶ introduce new packages and tools based on needs

This is just the beginning of your programming journey!

# Learning philosophy

- ▶ We learn coding by experimenting with code
- ▶ Coding can be frustrating
- ▶ Coding requires a different modality of thinking
- ▶ We develop self-sufficiency by learning where to get help and how to ask for help
- ▶ Coding lab is for you



# How will we progress?

1. Live lectures:
  - ▶ Focus on main idea first
  - ▶ Try it yourself – you learn coding by coding!
2. Practice in TA sessions (Most important part!):
  - ▶ Again – you learn coding by coding!
  - ▶ Break up into small groups and work on problems with peer and TA support
3. Additional help:
  - ▶ Send emails to Head TA for logistics issues
  - ▶ Post questions to Canvas Discussion Board (Teaching team will monitor and reply)
4. Final project:
  - ▶ It's optional; more details on next slide

## Final project (optional)

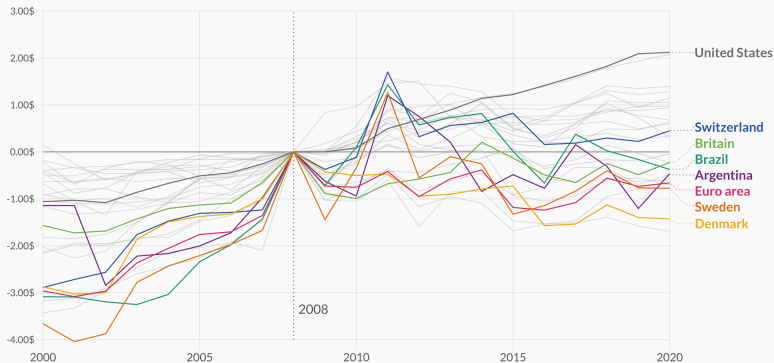
You'll know you're ready for policy school coding, if you can open a data set of interest to you and produce meaningful analysis. For the final project, you will:

- ▶ Pick a data set aligned with your policy interests (or not)
- ▶ Use programming skills to engage with data and make a data visualization showing something you learned from the data

# Final project goal

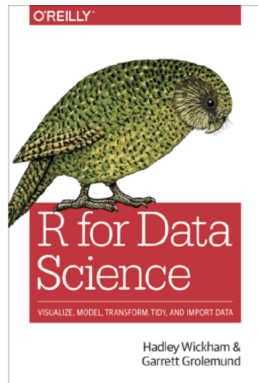
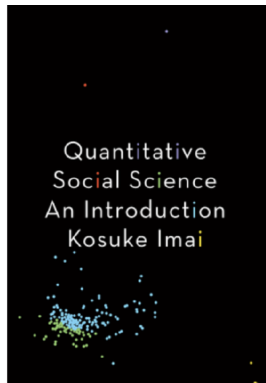
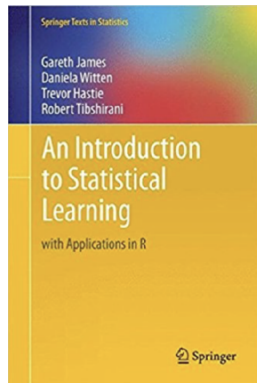
## Compared to the financial crisis in 2008, how much more or less do you have to pay for a Big Mac today?

The index chart visualizes the price changes (in USD) of a Big Mac based on a 2008 as index year. The **Big Mac Index** is published by The Economist as an informal way to provide a test of the extent to which market exchange rates result in goods costing the same in different countries. It seeks to *make exchange-rate theory a bit more digestible* and takes its name from the Big Mac, a hamburger sold at McDonald's restaurants.



Visualization by Cédric Scherer • Data by The Economist • The index chart shows the 27 countries that provide Big mac prices for all years from 2000 to 2020. In case a country was reported twice per year, the mean value was visualized.

# Textbooks and Resources



- ▶ Get situated with R for Data Science <https://r4ds.had.co.nz/>

# Textbooks and Resources

*How to actually learn any new programming concept*



*Essential*

Changing Stuff and  
Seeing What Happens

O RLY?

@ThePracticalDev

*The internet will make those bad words go away*



*Essential*

Googling the  
Error Message


O RLY?

*The Practical Developer  
@ThePracticalDev*

- ▶ Google is your friend for idiosyncratic problems

# Textbooks and Resources

*Cutting corners to meet arbitrary management deadlines*



*Essential*

## Copying and Pasting from Stack Overflow

O'REILLY®

*The Practical Developer*  
@ThePracticalDev

stackoverflow

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams

QA&A for work

Learn More

Search...

### What are the differences between "=" and "<=" in R?

What are the differences between the assignment operators `=` and `<=` in R?

I know that operators are slightly different, as this example shows

```
x <- y <- 5
x = y = 5
x <= y <- 5
x = y = 5
## Error in (x <- y) = 5 : could not find function "<=--"
```

But is this the only difference?

[assignment-operator](#) [faq](#)

share Improve this question

asked Mar 8 '18 at 9:30

rtore 5,500 ● 8 ● 48 ● 93

asked Nov 10 '18 at 12:14

cragflesce 42.4k ● 11 ● 107 ● 147

27 As noted [here](#) the origins of the `<=` symbol come from old APL keyboards that actually had a single `<=` key on them. — jordan Dec 12 '14 at 17:35

add a comment

7 Answers

active oldest votes

The difference in [assignment operators](#) is clearer when you use them to set an argument value in a function call. For example:

```
median(x = 1:10)
x
## Error: object "x" not found
```

✓ In this case, `x` is declared within the scope of the function, so it does not exist in the user workspace.

```
median(x <- 1:10)
x
## [1] 1 2 3 4 5 6 7 8 9 10
```

In this case, `x` is declared in the user workspace, so you can use it after the function call has been completed.

There is a general preference among the R community for using `<=` for assignment (other than in

- ▶ Stack Overflow is your another friend!
- ▶ ... In homework, give credit where it's due.

## Using R and RStudio

# Getting to business

We will

- ▶ Discuss what RStudio is
- ▶ Introduce minimal information to get started working with R
- ▶ Learn different types of operators
- ▶ Extending R with packages
- ▶ Bringing in data to R



## Why R *and* RStudio?

R is a language and environment for statistical computing and graphics.

RStudio is an “integrated development environment” for R.

In order to use RStudio, you must also have both RStudio and R installed on your computer (R is the “engine” for RStudio)

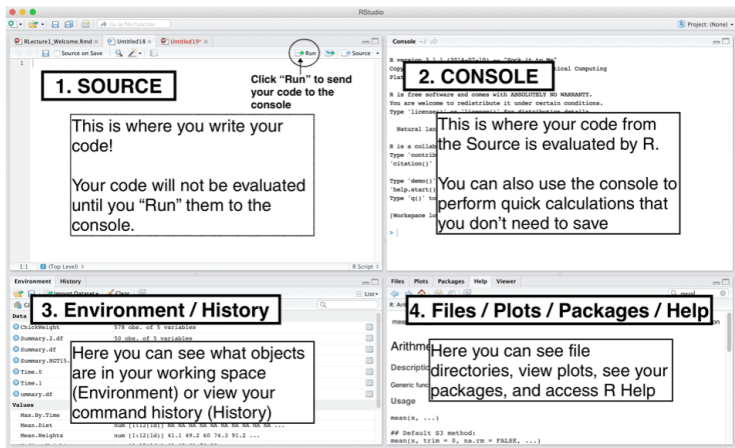
# Begin Live Demo

- ▶ Navigating R Studio
- ▶ Rmds and Scripts
- ▶ R as calculator

# RStudio basics

- ▶ It provides a console to access R directly
- ▶ A text editor to write R scripts and work with Rmds
- ▶ An environment and history tab that provide useful information about what objects you have in your R session
- ▶ A help / plots / files / packages etc. section

# RStudio Layout



(This layout may be different than yours)

## Anatomy of RStudio

The screenshot shows the RStudio interface with several panels and annotations:

- Source Editor (Top Left):** Contains R code for loading data and creating a data frame. A callout points to this area: "This window is a 'script'. Here's where you write and edit your programs."
- Environment/History (Top Right):** Shows the current environment with objects like 'data', 'df1', 'df2', and 'df3'. A callout points to this area: "Your data lives here."
- Files/Plots/Packages/Help/Viewer (Middle Right):** Shows the file explorer and other panels. A callout points to this area: "You'll find your files, plots, packages and other stuff over here."
- Console/Terminal (Bottom Left):** Shows the output of the R code. A callout points to this area: "Down here is the console. This is where your code runs and where R gets its work done."
- Documentation (Bottom Right):** Shows the help page for the 'ifelse' function. A callout points to this area: "You'll find your files, plots, packages and other stuff over here."

The R code in the Source Editor is as follows:

```
1 setwd("~/Downloads")
2 library(tidyverse)
3 library(data.table)
4 read_csv("data.csv")
5 df <- read_csv("data.csv", col_names = FALSE)
6 glimpse(df)
7
8 df <- df %>%
9   mutate("Treatment" = "...") %>%
10  mutate("Subject" = "...") %>%
11  mutate("Task" = "...") %>%
12  mutate("Time" = "...") %>%
13  mutate("Choice" = "...") %>%
14  mutate("Reward" = "...") %>%
15  mutate("Trial" = "...") %>%
16  mutate("Day" = "...") %>%
17
18 df3 <- df %>%
19   filter(Choice != 0)
20 df3 <- df3 %>%
21   group_by(Subject) %>%
22   summarise(Choice = sum(Choice))
23
24 df <- df %>% this block isn't working!
25
26 group_by(Subject) %>%
27   summarise(Choice = sum(Choice))
28
```

The console output shows the R version and the execution of the code:

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[workspace loaded from ~/Documents/Masters/PhD/SP_BBS/R/Data]
>
```

# Executing commands in R

Three ways to execute commands in R:

1. Type/copy commands directly into the console
2. R scripts (.R files)
  - ▶ This is just a text file full of R commands
  - ▶ Can execute one command at a time, several commands at a time, or the entire script
3. 'code chunks' in RMarkdown (.Rmd files)
  - ▶ Can execute one command at a time, one chunk at a time, or "knit" the entire document
  - ▶ More on this later

## Using R as a calculator

$+$ ,  $-$ ,  $*$ , and  $/$ . Also,  $^$  (Exponent).

```
7 + 5
```

```
## [1] 12
```

```
(4 + 6) * 3 - 2
```

```
## [1] 28
```

```
7 / 5
```

```
## [1] 1.4
```

```
2^4
```

```
## [1] 16
```

## Using R as a calculator

- ▶ R has many built-in mathematical functions
- ▶ To call a function, we type its *name*, followed by parentheses
- ▶ Anything we type inside the parentheses is called the function's *arguments*

```
sin(1)    # trigonometric functions
```

```
## [1] 0.841471
```

```
log(1)    # natural logarithm
```

```
## [1] 0
```

```
exp(0.5)  #  $e^{(1/2)}$ 
```

```
## [1] 1.648721
```

```
sqrt(4)   # square root of 4
```

```
## [1] 2
```



End Live Demo

## Try it yourself

1.  $30 + 6 \times 5^8 - \log(50) = ?$
2.  $-1^2 * (8 - \text{sqrt}(16)) = ?$
3.  $568 \times \frac{135}{\log(1)} = ?$
4.  $\frac{15^0 - 1}{0} = ?$

Try it yourself: Did you get ..

```
# answer
```

```
30 + 6*5^8 - log(50)
```

```
## [1] 2343776
```

```
(-1)^2 * (8 - sqrt(16)) # ambiguous (-1)^2 or -1^2
```

```
## [1] 4
```

```
568*135/log(1)
```

```
## [1] Inf
```

```
(15^{0} - 1)/0
```

```
## [1] NaN
```

## Operators can return special values

Inf is infinity. You can have either positive or negative infinity.

```
1 / 0
```

```
## [1] Inf
```

```
-5 / 0
```

```
## [1] -Inf
```

NaN means Not a Number. It's an undefined value.

```
0 / 0
```

```
## [1] NaN
```

NA means Missing or Not Available. (More later!)

```
NA + 4
```

```
## [1] NA
```

# Comparison Operators

These are also binary operators; they take two objects, and give back a *Boolean*

```
7 > 5 # greater than
```

```
## [1] TRUE
```

```
7 < 5 # less than
```

```
## [1] FALSE
```

```
7 >= 5 # greater than or equal to
```

```
## [1] TRUE
```

# Comparison Operators

```
7 <= 5 # less than or equal to
```

```
## [1] FALSE
```

```
7 == 5 # equality (two equals signs, read as "is equal to")
```

```
## [1] FALSE
```

```
7 != 5 # inequality (read as "is not equal to")
```

```
## [1] TRUE
```

Notice: == is a comparison operator, = is an assignment operator

& (and): Return TRUE if **all** terms are true.

- ▶ TRUE & TRUE -> TRUE
- ▶ TRUE & FALSE -> FALSE
- ▶ FALSE & TRUE -> FALSE
- ▶ FALSE & FALSE -> FALSE

```
(5 < 7) & (6 * 7 == 42)
```

```
## [1] TRUE
```

```
(5 < 7) & (6 * 7 < 42)
```

```
## [1] FALSE
```

```
(5 > 7) & (6 * 7 == 42)
```

```
## [1] FALSE
```

| (or): Return TRUE if **any** terms are true.

- ▶ TRUE | FALSE -> TRUE
- ▶ FALSE | TRUE -> TRUE
- ▶ TRUE | TRUE -> TRUE
- ▶ FALSE | FALSE -> FALSE

```
(5 < 7) | (6 * 7 < 42)
```

```
## [1] TRUE
```

```
(5 > 7) | (6 * 7 == 42)
```

```
## [1] TRUE
```

```
(5 > 7) | (6 * 7 != 42)
```

```
## [1] FALSE
```



# Logical Operations

operator	definition	operator	definition
<	less than	<code>x   y</code>	x OR y
<=	less than or equal to	<code>is.na(x)</code>	test if x is NA
>	greater than	<code>!is.na(x)</code>	test if x is not NA
>=	greater than or equal to	<code>x %in% y</code>	test if x is in y
==	exactly equal to	<code>!(x %in% y)</code>	test if x is not in y
!=	not equal to	<code>!x</code>	not x
<code>x &amp; y</code>	x AND y		

## Try it yourself

Guess the output of the following codes, and then run the codes to check your answer:

```
x <- 6  
(x < 9) & (x > 3)  
(x < 9) | (x > 7)  
(x > 8) | (x > 9)
```

```
x = 20  
y = 30  
(x == 20) & (y == 30)  
(x == 20) | (y == 50)  
(x + 5^8 - log(50) < 2000000) | (3*y - log(500) == 0)
```

## Basic syntax: Variable assignment

We can think of a variable as a *container* with a name, such as

- ▶ `x`
- ▶ `stats_score`
- ▶ `harris_gpa_average`

Each container can contain *one or more* values

## Basic syntax: Variable assignment

We use `<-` for assigning variables in R.

```
my_number <- 4  
my_number
```

```
## [1] 4
```

You can also use `=` for assigning variables

```
x = 5  
x
```

```
## [1] 5
```

## Basic syntax: Add comments using the # character

- ▶ Allow others (and future you) to have an easier time following what the code is doing

```
# comments help us remember ...  
my_number <- 4  
my_number - 10 # This should be equal to -6
```

```
## [1] -6
```

- ▶ Anything after # is ignored by R when executes code

## Variable assignment

We can re-assign a variable as we wish. This is useful if we want to try the same math with various different numbers.

```
my_number <- 2
```

```
sqrt((12 * my_number) + 1)
```

## Variable assignment: Use meaningful names

We assign all sorts of objects to names including data sets and statistical models so that we can refer to them later.

- ▶ **use names that are meaningful**

```
# mtcars is a built-in data set, so you can run this local  
model_fit <- lm(mpg ~ disp + cyl + hp, data = mtcars)  
summary(model_fit)
```

# Using functions

Functions are procedures that take an input and provide an output.

```
sqrt(4)
```

```
## [1] 2
```

```
median(c(3, 4, 5, 6, 7 ))
```

```
## [1] 5
```



## Function arguments

Function inputs are called arguments.

Functions know what the argument is supposed to do based on

- ▶ name
- ▶ position

e.g. `f` is a function that expects `x` and `y` and returns  $2*x + y$

```
# 2 * 7 + 0 = 14  
f(7, 0)
```

```
## [1] 14
```

```
# 2 * 0 + 7 = 7  
f(y = 7, x = 0)
```

```
## [1] 7
```

# Finding help with ?

```
?sum
```

## ► Description

`sum` returns the sum of all the values present in its arguments.

## ► Usage (API)

```
sum(..., na.rm = FALSE)
```

## ► Arguments

`...` numeric or complex or logical vectors.

## ► Examples (scroll down!)

```
sum(1, 2, 3, 4, 5)
```

# Review: intro to R and Rstudio

We learned

- ▶ How to navigate RStudio and run R code in the console and scripts
- ▶ How to use R operators for math and comparisons
- ▶ How to assign names to objects for future enjoyment
- ▶ How to use functions and find help with ?

## Extend R with Packages and Load in Data

# What are packages?

Packages are collections of *functions* and *data sets* developed by the community.

Benefits:

- ▶ Don't need to code everything from scratch (those are powerful tools!)
- ▶ Often functions are optimized using C or C++ code to speed up certain steps

# installing and loading packages

To use a package we need two steps:

- ▶ install/download once from the internet

```
install.packages("readxl")  # do this one time  
                             # directly in console
```

- ▶ load it *each time* we restart R

```
library(readxl) # add this to your script / Rmd  
               # every time you want to use  
read_xlsx("some_data.xls")
```

- ▶ `package::command()` lets you call a function without loading the library

```
readxl::read_xlsx("some_data.xls")
```

## common package error

The package 'readr' provides a function to read .csv files called `read_csv()`. What goes wrong here?

```
install.packages("haven")  
our_data <- read_csv("my_file.csv")  
  
Error in read_csv("my_file.csv") :  
  could not find function "read_csv"
```

## common package error

We need to load the package using `library()`!

```
library(readr)  
our_data <- read_csv("my_file.csv")
```

We can also use with one line of code:

```
our_data <- readr::read_csv("some_data.xls")
```



## tidyverse: set of useful packages

Think of the tidyverse packages providing a new dialect for R.

```
library(tidyverse)
## -- Attaching packages -----
## v ggplot2 3.3.0   v purrr   0.3.4
## v tibble  2.1.3   v dplyr   0.8.5
## v tidyr    1.0.2   v stringr 1.4.0
## v readr    1.3.1   v forcats 0.5.0
## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

# Try it yourself

Say we'd like to conduct data analysis using some powerful tools:

- ▶ Install the package `tidyverse`
- ▶ Then import it with the `library()` function
- ▶ Then run the command `storms` directly. What do you get?

## So you found some data

Say you find a spreadsheet on the internet and want to start exploring it with R.

Loading data is as easy as

```
library(readr)
housing_data <- read_csv("texas_housing_data.csv")
```

This requires that you consider:

- ▶ File type
- ▶ File location
- ▶ Funky formatting

# Loading data of various formats

Most common formats and their readers.

file type	package	function
.csv	readr (tidyverse)	read_csv()
.csv	utils (base R)	read.csv()
.dta (stata)	haven	read_dta()
.xlsx	readxl	read_xlsx()

## Detour: directory structure

Computer hard drives are organized using a file system. In this way, each file has a unique “address” or **file path**.

- ▶ `~/Documents/coding_lab/texas_housing_data.csv`

The files are stored in folders or directories which are analogous to “zip codes”.

- ▶ `~/Documents/coding_lab/`

In Windows, file paths usually start with `C://...`

## Detour: working directory

The 'working directory' is the folder in which you are currently working

- ▶ This is where R looks for files/data

```
# If the data is in the same folder as the current working directory  
fed_data <- read_xlsx("SCE-Public-LM-Quarterly-Microdata.xlsx")
```

getwd() shows your current working directory.

## Loading Data from original files

If the data were not in your current working directory, you could:

1. Locate the directory that your file is in
2. Set the directory as the **working directory**
3. Load the data into R using the correct function

```
# After setting the correct directory  
library(readr)  
setwd("/the/path/to/the/right/folder/")  
wealth_data <- read_csv("wealth_data.csv")
```

## Detour: Alternatives

If the data were not in your current working directory, you could:

- ▶ change the current working directory:  
`setwd("~/Documents/coding_lab")` and use `read_csv("file.csv")`
- ▶ give the absolute address:  
`read_csv("~/Documents/coding_lab/file.csv")`
- ▶ give a relative address: `read_csv("coding_lab/file.csv")`
  - ▶ this assumes “~/Documents” is the current working directory
- ▶ move the file to the current working directory



# Overview of the Data

- ▶ `View()`: look at the details of data (`view()` works if tidyverse loaded)
- ▶ `glimpse()`: structure of data frame – name, type and preview of data in each column
- ▶ `summary()`: displays min, 1st quartile, median, mean, 3rd quartile and max
- ▶ `head()`: shows first 6 rows

```
view(wealth_data)
glimpse(wealth_data)
summary(wealth_data)
head(wealth_data)
```

# Attributes of the Data

- ▶ `names()` or `colnames()`: both show the names of columns of a data frame
- ▶ `dim()`: returns the dimensions of data frame (i.e. number of rows and number of columns)
- ▶ `nrow()`: number of rows
- ▶ `ncol()`: number of columns

```
names(wealth_data)
dim(wealth_data)
nrow(wealth_data)
ncol(wealth_data)
```

# Try it yourself: Dataset storms

```
# Load dataset  
storms
```

- ▶ Try `view()`, `glimpse()` and `head()`. What do you see?
- ▶ What are the names of all the columns/variables included in `storms`?
- ▶ How many rows/observations are included?
- ▶ How many columns/variables are included?

# Review: using packages and reading data

We learned

- ▶ How to download packages from the internet with `install.packages()`
- ▶ How to load packages for use in R with `library()`
- ▶ How to distinguish between data formats (`csv`, `xlsx`, `dta`)
- ▶ How to navigate the file structure (`getwd()`, `setwd()`)
- ▶ How to programatically read data in to R
- ▶ How to get basic “see” data

## Next up

### Lab sessions:

- ▶ *today*: Review today's material
- ▶ *tomorrow*: Learn about Rmds. Load data (for your final project).
- ▶ Progress marker: I can load *relevant* policy data into R.

### Lecture:

- ▶ *Thursday*: Vectors + data types & Data in base R