

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

Harris Coding Camp – Accelerated Track

Summer 2022

Who are we and why are we here?

# 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?

Policy jobs and the Harris curriculum rely on coding

- ▶ to quickly engage with policy data
- ▶ to conduct statistical analyses
- ▶ to make transparent, reproducible analyses

Examples

- ▶ Determine number of people eligible for debt relief
- ▶ Measure impact of debt forgiveness on future aid take-up

**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 I learn?

# This is just the beginning!

Camp covers:

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



# Learning philosophy

- ▶ You are learning a (new) language
- ▶ Coding can be frustrating
- ▶ Coding requires a different modality of thinking
- ▶ We learn by producing code and experimenting

**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!):
  - ▶ Learn coding by coding!
  - ▶ Work on problems with peers and have TA support
3. Additional help:
  - ▶ logistics: email Head TA
  - ▶ coding: Post questions to Canvas Discussion Board
4. Final project

## Final project (optional)

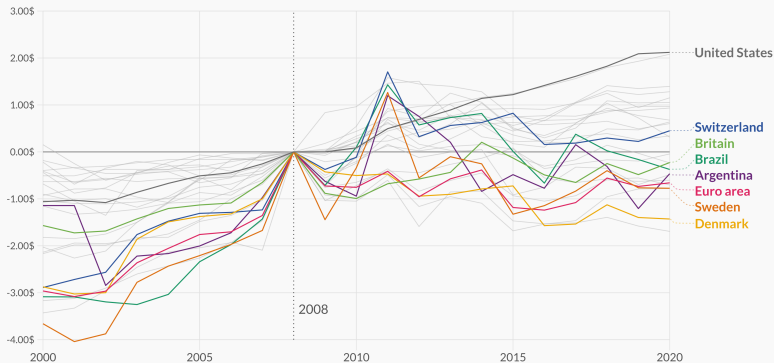
You'll know you're ready for policy school coding, if you can open a data set of interest to you and produce meaningful analysis. For the final project, you will:

- ▶ 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?

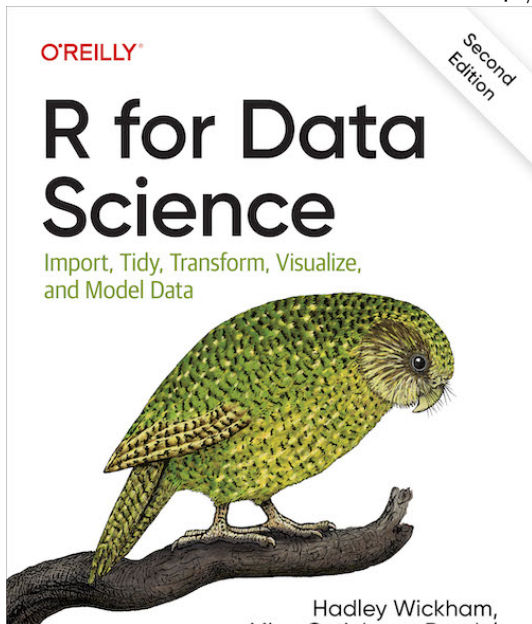
The index chart visualizes the price changes (in USD) of a Big Mac based on a 2008 as index year. The **Big Mac Index** is published by The Economist as an informal way to provide a test of the extent to which market exchange rates result in goods costing the same in different countries. It seeks to *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 <http://r4ds.hadley.nz/>



## Textbooks and Resources

- ▶ Run code in the console. Change code and re-run. Does it make sense?

*How to actually learn any new programming concept*




*Essential*

Changing Stuff and  
Seeing What Happens

# Learn to make it before your fake it.

*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

Team  
Q&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?

582

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 `<=` `<-`

share Improve this question

edited Mar 8 '18 at 9:30 asked Nov 15 '18 at 12:14

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

add a comment

7 Answers

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)
# 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)
# [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

- Google - Stack Overflow - ChatGPT

Provide credit where credit is due.

## A quick introduction to R and RStudio



# Using R and RStudio

We will

- ▶ Discuss what RStudio is
- ▶ Learn the basics of R
- ▶ Extend R with packages
- ▶ Bring data into R

# What's the difference between R and RStudio?

R is a programming language—all the nuts and bolts.

RStudio provides tools that make it easier to use R

- ▶ It's an “integrated development environment” (IDE) for R

Demo (see appendix for material)

## Try it yourself

1.  $30 + 6 \times 5^8 - \log(50) = ?$

2.  $-1^2 * (8 - \sqrt{16}) = ?$

3.  $568 \times \frac{135}{\log(1)} = ?$

4.  $\frac{15^0 - 1}{0} = ?$

Try it yourself: Did you get ..

```
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) # Inf is infinity (-Inf is negative)
```

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

```
(15^0 - 1)/0 # NaN means Not a Number
```

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

## Comparison and Logical Operators

## Compare 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
```

## Compare two objects and give back a *Boolean*

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

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

```
7 == 5 # "is equal to"
```

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

```
7 != 5 # "is not equal to"
```

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

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



# Logic operators

- ▶ & (and): Return TRUE if **both** terms are true
- ▶ | (or): Return TRUE if **either** terms are true
- ▶ ! (not): Return the opposite. (not true is false)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Try it yourself

Guess the output and then run the code 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 < 200) | (log(0.04978) + y/10 == 0)
```

## NA means Missing or Not Available

Common source of headaches

```
NA + 4
```

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

## NA requires it's own test

```
5 == NA
```

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

```
is.na(5)
```

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

```
5 != NA
```

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

```
!is.na(5)
```

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



## Reference slide

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		

## Variables and names

## 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
```

## Variable assignment

We can re-assign a variable as we wish.

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

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

```
my_number <- 2  
sqrt((12 * my_number) + 1)
```

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

## 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

```
# not so good  
s <- Z / secs
```

```
# better  
speed <- distance / time
```

```
# not so good  
x <- read_csv("fed_data.csv")
```

```
# better  
fed_data <- read_csv("fed_data.csv")
```

## Add comments using the # character

- ▶ Lets **future you** and teammates to follow what code is doing

```
# if you can do it with a name BETTER!, but  
my_number <- 4 # sometimes you need more info.
```

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

# Functions



## Using functions

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

```
sqrt(4)
```

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

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

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

## Capture outputs with names for later use

```
# mtcars is a built-in data set  
# so you can run this locally!  
lm(mpg ~ disp + cyl, data = mtcars)
```

```
##  
## Call:  
## lm(formula = mpg ~ disp + cyl, data = mtcars)  
##  
## Coefficients:  
## (Intercept)          disp           cyl  
##    34.66099      -0.02058      -1.58728
```

```
# notice no print out  
model_fit <- lm(mpg ~ disp + cyl, data = mtcars)
```

# Function arguments

Function inputs are called arguments.

Functions know what the argument refers to based on

- ▶ name
- ▶ position

## Arguments interpreted by *name*

I wrote a function `f` that expects `x` and `y` and returns  $2*x + y$

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

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

*Bad example* for demonstration purposes only

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

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

## Arguments interpreted by *position*

I wrote a function `f` that expects `x` and `y` and returns  $2*x + y$

```
# identical to f(x = 7, y = 0)  
f(7, 0)
```

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

# 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)
```

How to start working with data and packages

# 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")
```

## Q: What goes wrong here?

The package 'readr' provides a function to read .csv files called `read_csv()`.

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

A: 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:

```
# Run in console -- you only need to do this once!  
install.packages("tidyverse")
```

```
# Add to your script -- re-run everytime  
library(tidyverse)  
storms  
head(storms)
```

## See 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)      # base  
glimpse(wealth_data)   # tidyverse  
summary(wealth_data)   # base  
head(wealth_data)      # base
```

# Attributes of the Data

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

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

## Try it yourself: Dataset storms

- ▶ Look at the data in multiple ways. What do you see?
- ▶ What are the names of all the columns/variables in `storms`?
- ▶ How many rows/observations are included?
- ▶ How many columns/variables are included?

Code bank: `names()`, `nrow()`, `ncol()`, `View()`, `glimpse()` and `head()`



## Loading in data

## Loading data is as easy as ...

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

This requires that you consider:

- ▶ Format
- ▶ File location

# 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()

# File location

Where on your computer is the data stored?

- ▶ You'll need to understand your file system

## Detour: directory structure

Each file has a unique “address” or **file path**.

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

The files are stored in folders or directories, the “zip codes”.

- ▶ `~/Documents/coding_lab/`

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

## Detour: working directory

The 'working directory' is the folder R accesses by default

- ▶ `getwd()` shows your current working directory. Try it now!
- ▶ If you don't provide an explicit directory, R looks in current directory.

## Detour: working directory

Code works if `this_file.xlsx` is in the current working directory

```
library(readxl)
wealth_data <- read_xlsx("wealth_data.xlsx")
```

Otherwise you get an error like

Error: 'path' does not exist: 'wealth\_data.xlsx'

## Loading Data from original files

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

Set the directory as the **working directory**

```
# 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 also:

- ▶ 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

## Review: the basics

- ▶ How to navigate RStudio and run R code in the console and scripts
- ▶ How to use R operators for
  - ▶ math (+, ^)
  - ▶ comparisons (<=, !=)
  - ▶ logic (&, |)
- ▶ How to assign names
- ▶ How to use and learn about functions (Go ?!)

## Review: using packages and reading data

- ▶ 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

## Appendix: Demo

## RStudio basics (Demo)

- ▶ 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 (Demo)

**1. SOURCE**

Click "Run" to send your code to the console

This is where you write your code!

Your code will not be evaluated until you "Run" them to the console.

**2. CONSOLE**

This is where your code from the Source is evaluated by R.

You can also use the console to perform quick calculations that you don't need to save

**3. Environment / History**

Here you can see what objects are in your working space (Environment) or view your command history (History)

**4. Files / Plots / Packages / Help**

Here you can see file directories, view plots, see your packages, and access R Help

(This layout may be different than yours)

# RStudio Layout (Demo)

## Anatomy of RStudio

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

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

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 df <- as_tibble(df)
8
9 df <- df %>%
10   rename("Treatment" = "...") %>%
11   rename("Subject" = "...") %>%
12   rename("Task" = "...") %>%
13   rename("Time" = "...") %>%
14   rename("Choice" = "...") %>%
15   rename("Reward" = "...") %>%
16   rename("Trial" = "...") %>%
17   rename("Day" = "...")
18
19 df3 <- df %>%
20   filter(Choice != 0)
21 df3 <- df3 %>%
22   na.omit(Choice)
23
24 df <- df %>% #this block isn't working!
25   mutate(Choice = 1) %>%
26   group_by(Subject) %>%
27   summarise(Subtotal_Raw_Totals =
28     sum(Choice))
```



# Executing commands in R (Demo)

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

## Using R as a calculator (Demo)

$+$ ,  $-$ ,  $*$ , 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 (Demo)

- ▶ 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
```