Getting Started

POLS 7012: Introduction to Political Methodology

Joseph T. Ornstein (Joe)

University of Georgia

August 26, 2020

What Are We Doing Here?

What Are We Doing Here?

Modern political science is full of data analysis and mathematical thinking. You'll want to speak the language.

By the end of this course, you will be able to...

- "Wrangle" and summarize data
- Construct elegant and informative data visualizations
- Distinguish between patterns in data and random noise (**probability** and **inference**)
- Build models and conveniently represent data using **matrix algebra**
- Optimize objective functions using **calculus**

So...this is a weird semester...

The Plan:

- Face-to-face courses until Thanksgiving
- Simulcast over Zoom
- Slides and lecture recordings posted to the course website

Assignments:

- 1 Problem Set each week
- 1 Midterm + 1 Final Exam

Other Logistics:

- No required textbook, but lots of suggestions in the syllabus
- Weekly virtual office hours over Zoom

Introductions

Today's Objectives

By the end of this week, you will be able to:

- Write basic R scripts for loading and exploring data
- Navigate RStudio
- Explain several foundational mathematical/statistical concepts, including:
    - Logarithms
    - Functions
    - Vectors
    - Matrices
    - Mean & Median
    - Variance & Standard Deviation

R

What is R?

R is a programming language specifically designed for statistical computing.

It is widely used in academia and the data science community. By some measures it is the 8th most popular programming language.

Why use R?

Major Alternatives: Microsoft Excel, Stata, SPSS, SAS, Python

Short Answer:

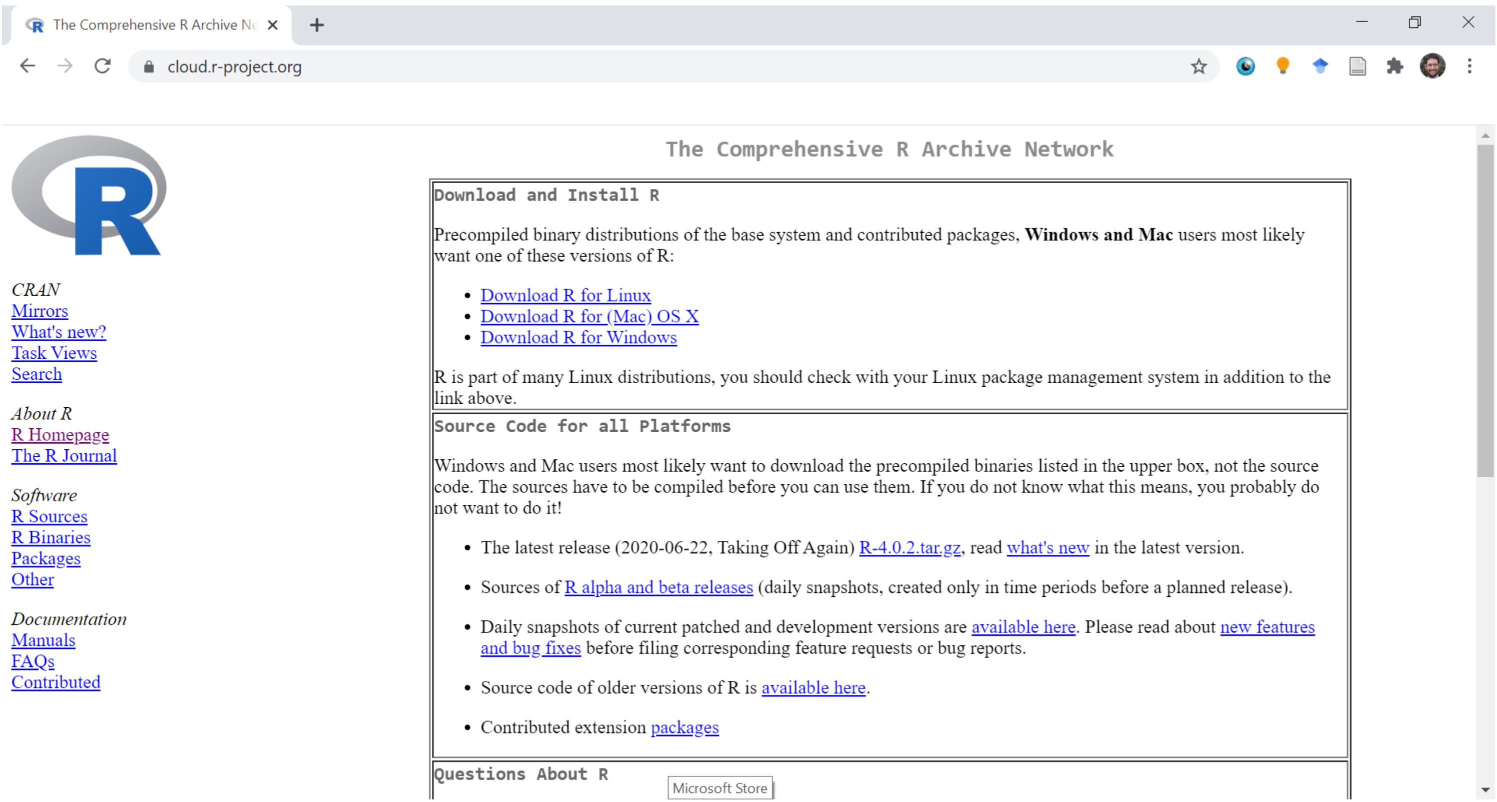
- I use R and I'm comfortable teaching it.

Long Answer:

- It's free
- It's open-source
- It's reproducible
- It has a huge user community building useful packages (add-ons)
- In academic social science, R is now fairly predominant (particularly among younger cohorts)
- It has a shorter learning curve than Python

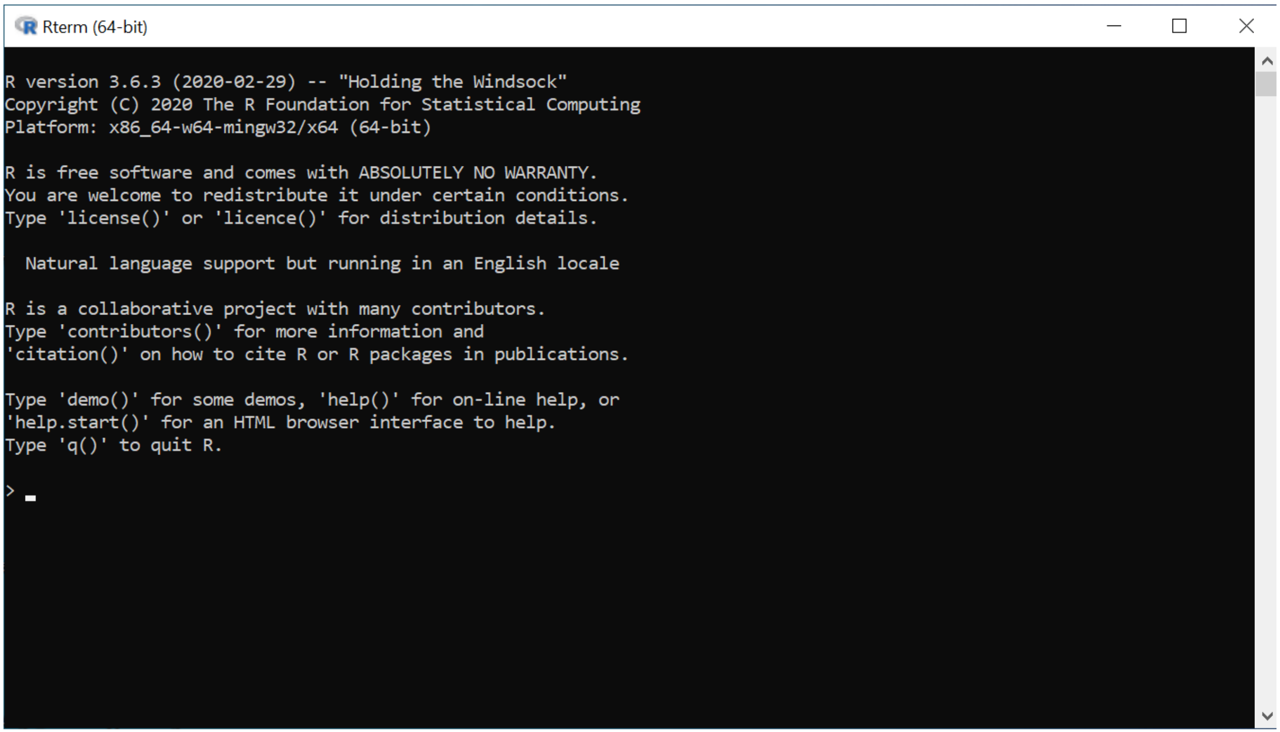
Download R

Here (click the picture):



Download R

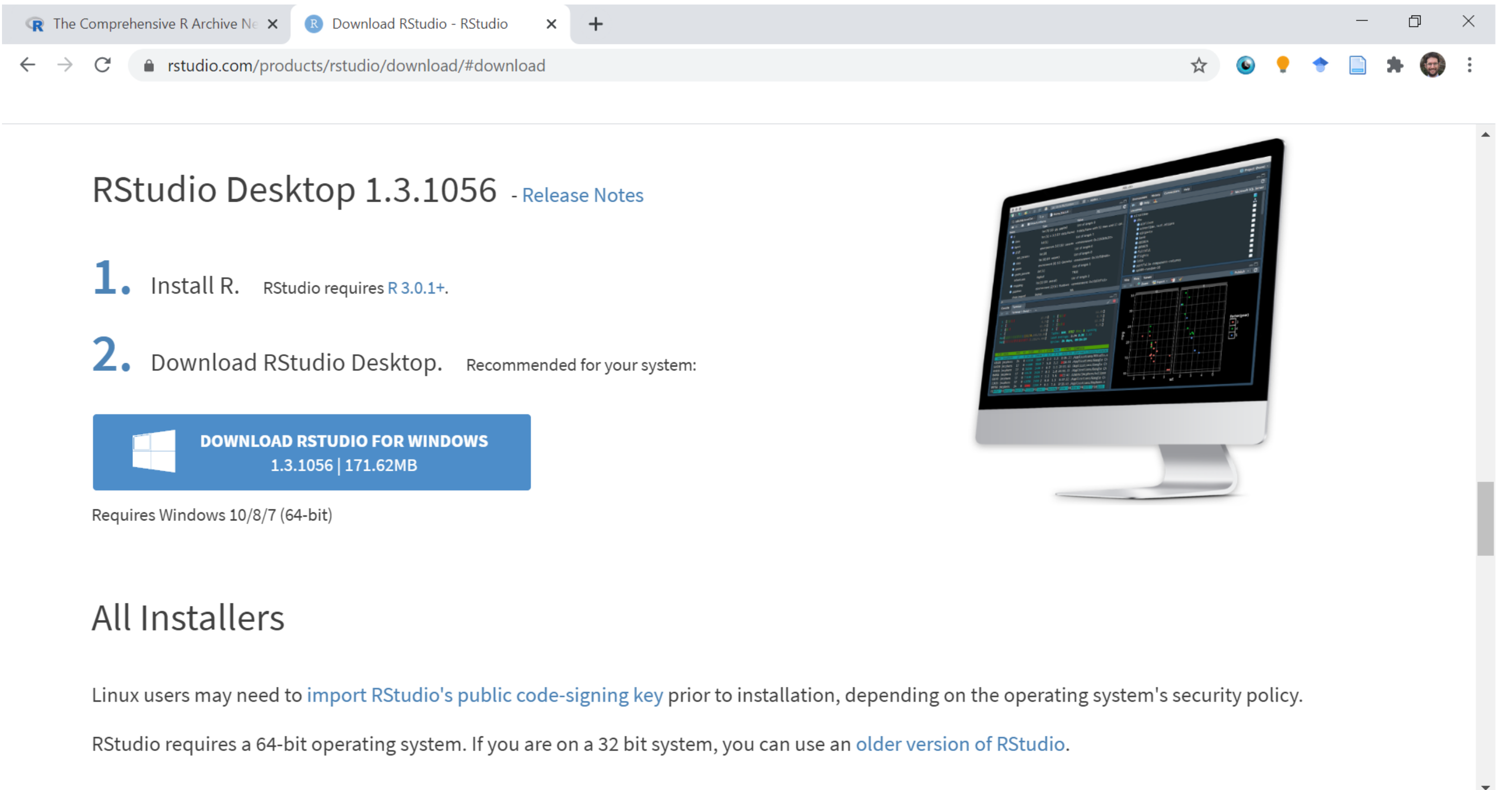
You could stop there, and work with R in this nice black terminal box…



…or we could use an IDE (integrated development environment) called RStudio.

Download RStudio

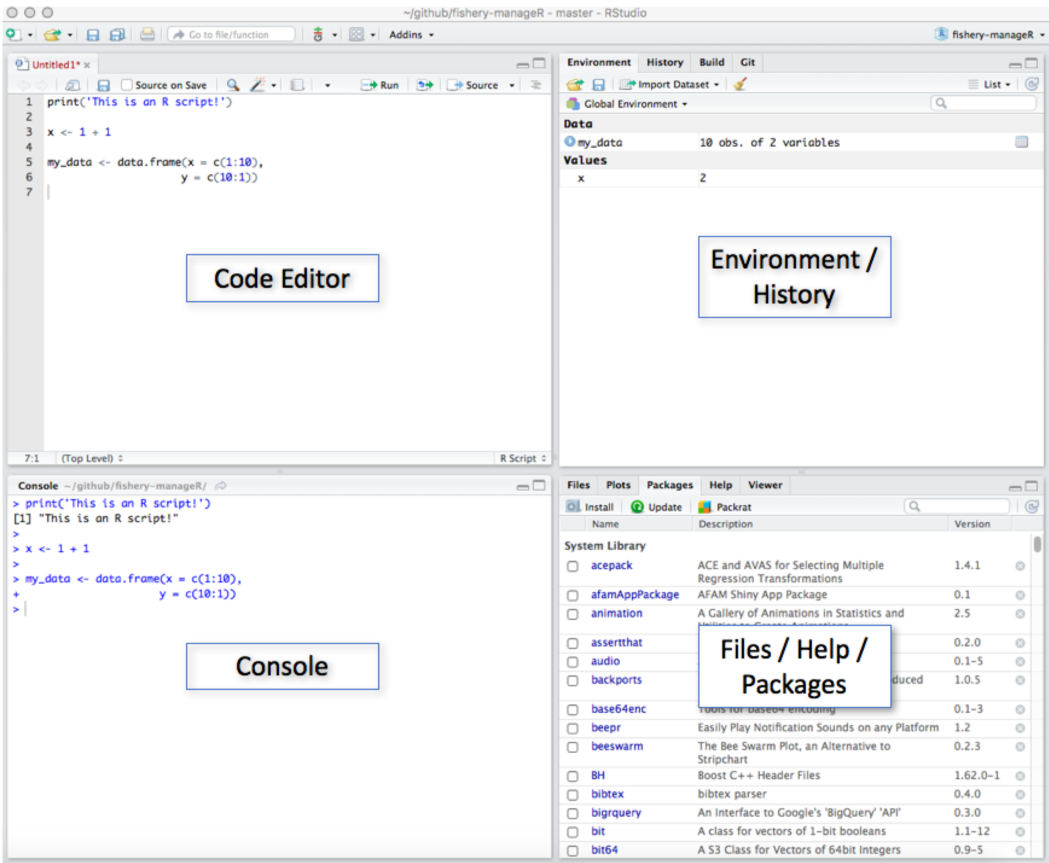
Here:



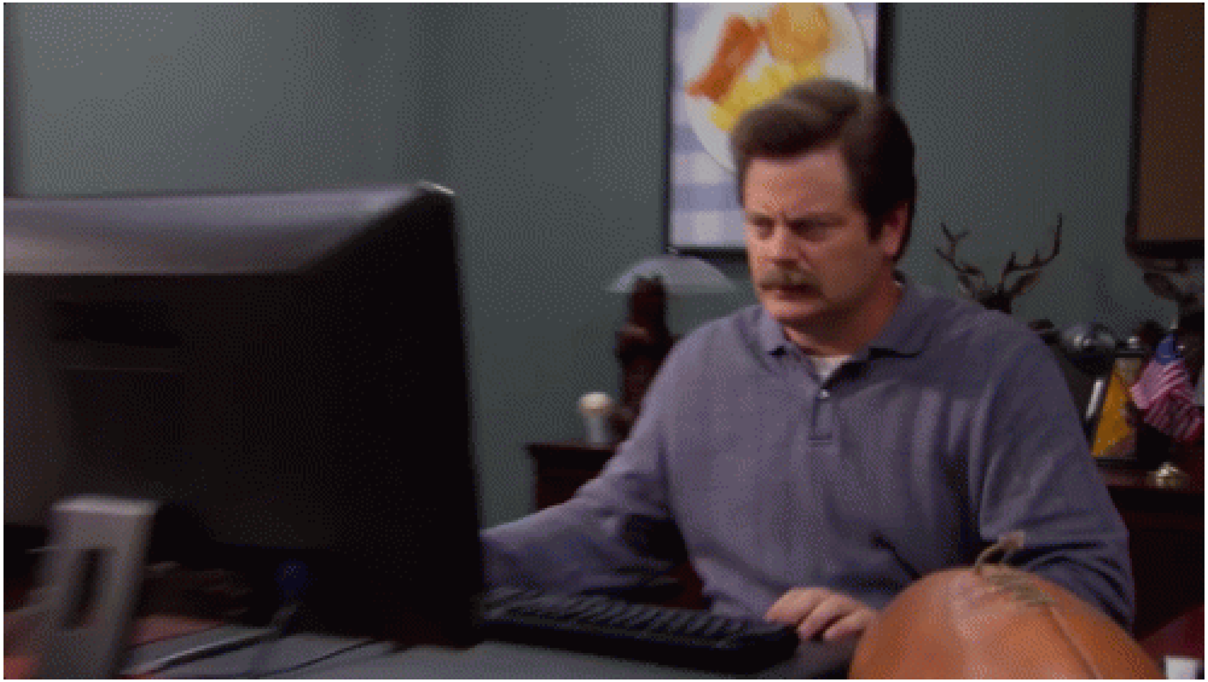
Navigating RStudio

Navigating RStudio

Open RStudio, and you should see *this*:



R Will Be Frustrating



R Will Be Frustrating

Computers are great at a lot of things, like:

- Multiplying large numbers
- Following instructions perfectly
- Never getting tired or sleepy

Computers are *not-so-great* at other things, like:

- Guessing what you really want

> "Like all programming languages, R does exactly what you tell it to, rather than exactly what you want it to…It is as if one had an endlessly energetic, powerful, but also extremely literal-minded robot to order around."
>
> -Kieran Healy

R Will Be Frustrating

But the good news is that **you're not alone**.

No matter what problem you're facing, I guarantee that someone has already asked a question about it on one of these websites:

stack overflow

Google

R is a Fancy Calculator

R is a Fancy Calculator

Type `2+2` into the console and press `Enter`.

```
2 + 2
```

```
[1] 4
```

I basically use R whenever I need a calculator now.

(I Kon-Maried my Texas Instruments TI-83. Thank you, old friend.)

R is a Fancy Calculator

Subtraction, Multiplication, and Division

```r
2 - 2
```

```
[1] 0
```

```r
2 * 2
```

```
[1] 4
```

```r
2 / 2
```

```
[1] 1
```

Note: whenever you see a **code block** like the ones above, please follow along by typing the code into your R Console! If you don't get the same result as me, stop me and we'll discuss.

R is a Fancy Calculator

Exponentiation

$2^4 = 2 \times 2 \times 2 \times 2 = ?$

```r
2^4
```

```
[1] 16
```

Now you try. Compute 3^6.

```r
3^6
```

```
[1] 729
```

R is a Fancy Calculator

Fractional Exponents

Remember what happens when you raise a number to a fractional power? Say, $9^{\frac{1}{2}}$?

```
9^0.5
```

```
[1] 3
```

$9^{\frac{1}{2}} = \sqrt{9}$

R is a Fancy Calculator

Fractional Exponents

What is $(4^3)^{\frac{1}{3}}$?

```
(4^3)^(1/3)
```

```
[1] 4
```

General Rule: When you have nested exponents like that, you can just multiply the exponents.

```
(2^3)^6
```

```
[1] 262144
```

```
2^18
```

```
[1] 262144
```

R is a Fancy Calculator

Hey, remember **logarithms**?

Subtraction reverses addition.

$2 + 3 - 3 = 2$

Division reverses multiplication.

$2 \times 3 \div 3 = 2$

Logarithms reverse exponentiation.

$\log_3(3^2) = 2$

The subscript is the "base" of the logarithm. When you see $\log_x(y)$, think "how many times do I have to multiply x by itself to get y?"

R is a Fancy Calculator

What is $\log_{10}(100)$?

```
log(100, base = 10)
```

```
[1] 2
```

Because $10^2 = 100$.

What is $\log_2(100)$?

```
log(100, base = 2)
```

```
[1] 6.643856
```

Because $2^6 = 64$ and $2^7 = 128$, so you need to multiply 2 by itself roughly 6.6 times in order to get 100.

Weird. I know.

R is a Fancy Calculator

The Nice Thing About Logarithms

Logarithms turn multiplication into addition:

$$\log(ab) = \log(a) + \log(b)$$

And they turn exponentiation into multiplication:

$$\log(a^b) = b \log(a)$$

This comes in handy a lot! Basically anytime you see a "log transformation" in political science, we're taking advantage of this fact to make the math easier.

R is a Fancy Calculator

Compute `log(342 * 702)` and `log(342) + log(702)`.

You should get:

```r
log(342 * 702)
```

```
[1] 12.38874
```

```r
log(342) + log(702)
```

```
[1] 12.38874
```

R is a Fancy Calculator

Now try `log(342 ^ 702)` and `702 * log(342)`.

You should get:

```r
log(342 ^ 702)
```

```
[1] Inf
```

```r
702 * log(342)
```

```
[1] 4096.037
```

The first one completely broke R because 342^{702} is an unimaginably large number that it can't keep in memory. So, it's useful to know your logarithm rules sometimes.

R is a Fancy Calculator

Oh Hey

What base does R use when you don't give it a base for the `log` calculation?

Answer: It uses base $e \approx 2.718$, also known as **Euler's Number**. The logarithm with base e is called the "natural logarithm", also denoted $\ln x$.

That's Leonhard Euler. Don't let the weird towel hat fool you. He was a towering 18th century mathematical genius.

R is a Fancy Calculator

R respects the **Order of Operations**.

- Parentheses, Exponents, Multiplication, Division, Addition, then Subtraction
- (**P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally)

Try $\frac{2(3+2)^2-2}{3}$

$$\frac{2(3+2)^2-2}{3} = \frac{2(5)^2-2}{3} = \frac{2(25)-2}{3} = \frac{50-2}{3} = \frac{48}{3} = 16$$

Or, in R:

```
(2*(3+2)^2 - 2)/3
```

```
[1] 16
```

R is a Fancy Calculator

Finally, R can evaluate **logical expressions**.

```
2 < 3
```

```
[1] TRUE
```

```
2 > 3
```

```
[1] FALSE
```

```
2*2 < 4*5
```

```
[1] TRUE
```

```
3*4 == 2*6
```

```
[1] TRUE
```

Objects

Objects

One of the features that makes R such a *fancy* calculator is the ability to save **objects**.

You can assign any value to an object like this:

```r
x <- 2
```

The left arrow (`<-`) tells R that we want to assign the value 2 to an object named `x`.

Now when you type `x` into the console, R will output the value 2.

```r
x
```

```
[1] 2
```

Objects

You can manipulate objects just like we manipulated numbers.

```
2 * x
```

```
[1] 4
```

```
2 - x^2 * 5
```

```
[1] -18
```

Objects

You can give objects whatever names you like, with a few restrictions:

- No spaces
- No special mathematical characters, like `-`, `+`, `*`, `/`
- Variable names can include numbers, but must *start* with a letter.

Use good names for your objects!

```r
georgia_population <- 10620000
```

This is a bad name:

```r
vargp <- 10620000
```

I personally use underscores (`_`) if I want to name an object something with multiple words.

Objects

Now you try! Create an object, name it whatever you want, and assign it a number.

Then create a second object that is two times your first object.

```
x <- 10

x_twice <- 2*x

x
```

```
[1] 10
```

```
x_twice
```

```
[1] 20
```

Objects

Objects can also be **characters** instead of numbers.

```
my_name <- "Joe"

my_name
```

```
[1] "Joe"
```

They can also be **booleans**, which is computer-speak for "true or false".

```
i_am_joe <- (my_name == "Joe")
two_plus_two_is_five <- (2 + 2 == 5)

i_am_joe
```

```
[1] TRUE
```

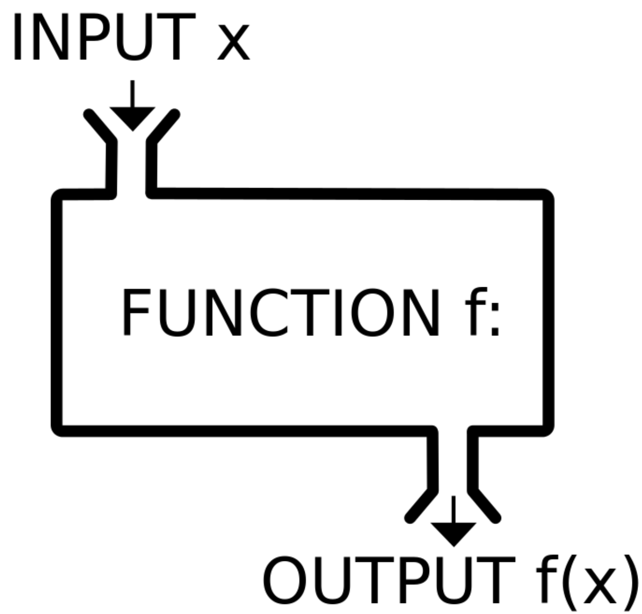
```
two_plus_two_is_five
```

```
[1] FALSE
```

Functions

Functions

A **function** transforms one or more inputs into an output.



$f(x) = 2x + 2$

$f(2) = ?$

Functions

You've already seen some built-in R functions, for example:

```
log(x, base)
```

The `log()` function takes two inputs, a number `x` and a value for `base`, and outputs the logarithm.

R functions almost always come in this form: a name followed by one or more inputs in parentheses, separated by commas.

Functions

What does the `round()` function do? What about the `paste()` function?

```
round(3.4)
```

```
[1] 3
```

```
round(3.6)
```

```
[1] 4
```

```
paste("Hi", "my name is", my_name)
```

```
[1] "Hi my name is Joe"
```

There are *lots* of built-in and user-created functions for R. If you ever want to know what a function does, type `?` + the name of the function in the console.

```
?log
```

Functions

You can also make your own functions, like this:

```
my_function <- function(x, y){
  return(2*x + y^2 - 2)
}
```

```
my_function(x = 4, y = 6)
```

```
[1] 42
```

This is **super** helpful when writing lots of repetitious code. More on this in a few weeks.

Now you try!

Write a function called `round_the_log` which takes a number, finds the natural logarithm, and rounds it to the nearest whole number.

Functions

Now you try!

Write a function called `round_the_log` which inputs a number `x`, and returns the natural logarithm rounded to the nearest whole number.

```
round_the_log <- function(x){
  return(round(log(x)))
}
```

```
round_the_log(42)
```

```
[1] 4
```

Some Useful RStudio Tips

Some Useful RStudio Tips

- If you want to learn more about a function, type `?` and then the function name into the console. For example `?log`.
- To repeat a previous console command, tap the `Up Arrow`.
- Code Completion: When you start typing a command, RStudio will try to guess what you want, and will give you a list of suggestions. Tap `Tab` to auto-complete.
- More Tips And Tricks, most of which will be more useful after we've learned a bit more.

We're finally ready to work with...

Data!

Data

1. Vectors

A **vector** is a collection of values arranged in some order. Vectors are the building blocks of datasets.

For example, this is the vector $\mathbf{x}$.

$$\mathbf{x} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$

We build vectors in R using the function `c()`, which is short for **concatenate**.

```
x <- c(1,3,4)

x
```

```
[1] 1 3 4
```

Data

1. Vectors

You can retrieve the elements of a vector using brackets `[]`.

```r
x
```

```
[1] 1 3 4
```

```r
x[1]
```

```
[1] 1
```

```r
x[2]
```

```
[1] 3
```

```r
x[2:3]
```

```
[1] 3 4
```

Data

1. Vectors

If you've never seen vectors before, don't worry. They basically operate how you would expect a bunch of numbers squished together to operate.

Adding Vectors

$$\begin{bmatrix}1\\3\\4\end{bmatrix} + \begin{bmatrix}3\\2\\2\end{bmatrix} = \begin{bmatrix}4\\5\\6\end{bmatrix}$$

```r
x <- c(1,3,4)
y <- c(3,2,2)

x + y
```

```
[1] 4 5 6
```

Scalar Multiplication

$$2 \times \begin{bmatrix}1\\3\\4\end{bmatrix} = \begin{bmatrix}2\\6\\8\end{bmatrix}$$

```r
x <- c(1,3,4)

2*x
```

```
[1] 2 6 8
```

Data

1. Vectors

Many R functions are **vectorized**. If you input a vector, it evaluates the function for each element of the vector.

```
x <- c(3.4, 3.5, 3.6)

round(x)
```

```
[1] 3 4 4
```

```
log(x)
```

```
[1] 1.223775 1.252763 1.280934
```

```
round_the_log(x)
```

```
[1] 1 1 1
```

Data

1. Vectors

Other functions input a vector and return a single value.

Suppose you have a vector x representing the number of Democratic votes cast in a collection of counties.

```
democratic_votes <- c(1562, 2890, 49134, 18, 901)
```

If you want to know how many entries are in the vector, use `length()`.

```
length(democratic_votes)
```

```
[1] 5
```

Data

1. Vectors

To get the **minimum** and **maximum** values in a vector, use the `min()` and `max()` functions.

```
min(democratic_votes)
```

```
[1] 18
```

```
max(democratic_votes)
```

```
[1] 49134
```

Data

1. Vectors

Maybe you want to take the sum across all counties:

$$\sum_i x_i$$

Use the `sum()` function.

```
sum(democratic_votes)
```

```
[1] 54505
```

Data

1. Vectors

Maybe you want the **mean** (average) value, the sum divided by the length.

$$\bar{x} = \frac{\sum_i x_i}{n}$$

Use the `mean()` function.

```
mean(democratic_votes)
```

```
[1] 10901
```

Now try to take the `sum()` divided by the `length()`. Do you get the same result?

```
sum(democratic_votes) / length(democratic_votes)
```

```
[1] 10901
```

Data

1. Vectors

The **median** is the value right in the middle of a vector sorted from least to greatest.

```
democratic_votes
```

```
[1]  1562  2890 49134    18   901
```

```
median(democratic_votes)
```

```
[1] 1562
```

```
sort(democratic_votes)
```

```
[1]    18   901  1562  2890 49134
```

Data

1. Vectors

The **variance** of a vector measures how far values tend to be from the mean. It is a measure of *dispersion*.

Variance equals the average *squared* distance from the mean.

$$\sigma_x^2 = \frac{\sum_i (x_i - \bar{x})^2}{n}$$

You can compute variance in R with the `var()` function.

```
var(democratic_votes)
```

```
[1] 457898755
```

Data

1. Vectors

Which of these vectors has a smaller variance?

```
x <- c(1, 1, 1, 1, 1, 1)

y <- c(1, 2, 3, 4, 5, 6)
```

```
var(x)
```

```
[1] 0
```

```
var(y)
```

```
[1] 3.5
```

Data

1. Vectors

The square root of variance is the **standard deviation**.

$$\sigma_x = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{n}}$$

```
sd(democratic_votes)
```

```
[1] 21398.57
```

Data

2. Matrices

A **matrix** (the plural is **matrices**) is a bunch of vectors smushed together.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3.2 & 1 & 7 \\ 9 & 8 & 42 \\ 1 & 12 & \pi \end{bmatrix}$$

Each row is a vector and so is each column.

Data

3. Playing with matrices

To create a matrix in R, we **bind** together a set of vectors using the `rbind()` or `cbind()` functions.

```
A <- rbind(c(1, 2, 3),
           c(2, 1, 3),
           c(42, 1, pi))

A
```

```
     [,1] [,2]     [,3]
[1,]    1    2 3.000000
[2,]    2    1 3.000000
[3,]   42    1 3.141593
```

Now you try. Create the matrix `A` again, but this time use `cbind()` to paste the columns together.

Data

3. Playing with matrices

You can retrieve entries in the matrix like so:

```
A[1,2]
```

```
[1] 2
```

```
A[3,3]
```

```
[1] 3.141593
```

```
A[3,]
```

```
[1] 42.000000  1.000000  3.141593
```

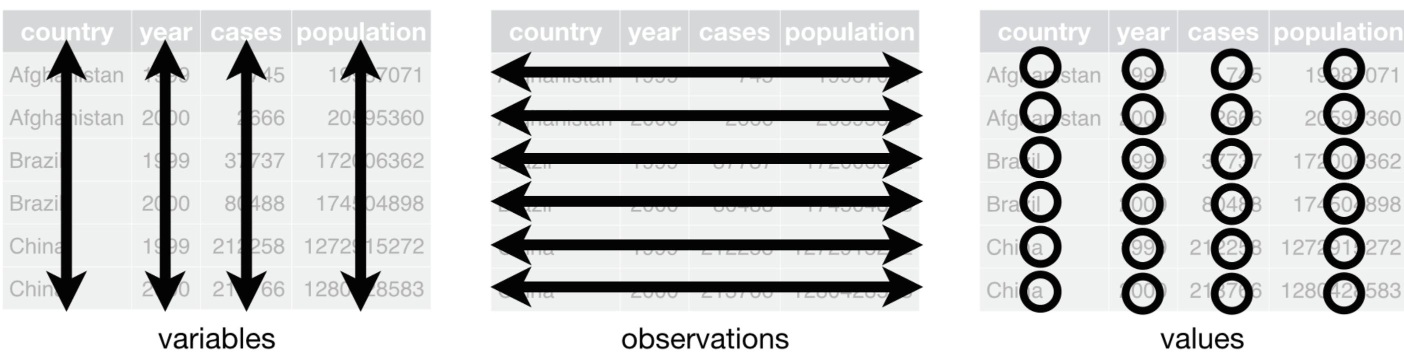
```
A[,2]
```

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

Data

4. Tidy Data

Tidy data is a rectangular matrix, arranged like this:



Unit of Observation: What does each row represent?

- Could be countries, people, "country-years", etc. Just keep it consistent!

Data

4. Tidy Data

In R, we represent data using a special object called a `data.frame`. (Or a `tibble`, which I'll show you next week.)

```
height <- c(5.3, 5.8, 4.2)
gender <- c('male', 'male', 'female')
party <- c('R', 'D', 'D')

data <- data.frame(height, gender, party)

data
```

```
  height gender party
1    5.3   male     R
2    5.8   male     D
3    4.2 female     D
```

Data

4. Tidy Data

You can access columns in a data frame by using `$` + the variable name.

```
data$height
```

```
[1] 5.3 5.8 4.2
```

```
data$party
```

```
[1] R D D
Levels: D R
```

And then you can manipulate it just like any other vector.

```
mean(data$height)
```

```
[1] 5.1
```

Now it's time for a lesson I like to call...

USE THE CODE EDITOR. SERIOUSLY.

USE THE CODE EDITOR. SERIOUSLY.

We've been having a lot of fun typing commands into the console. It's fast. It's easy. It's *addictive*.

NEVER ACTUALLY DO YOUR WORK THERE.

Why?

Because go ahead and close your RStudio window. Then reopen it.

Go ahead, I'll wait.

AAAAAAAAAAAAAAAAAAAAHHHHHHHHHHH WHAT HAPPENED TO MY WORK??? IT'S GONE! HOW WILL I EVER REMEMBER WHAT I DID TO GET MY RESULT????

Don't let this sort of thing happen to you. Use the **Code Editor**.

Writing R Scripts

Writing R Scripts

An **R script** is a series of commands that you can save, modify, and re-run whenever you like.

```
x <- c(1, 3, 4)

y <- mean(x)

z <- median(x)

y < z
```

```
[1] TRUE
```

The four lines of code above:

- Creates a vector called `x`
- Compute the mean and median of `x`
- Tells you if the mean is smaller than the median

Copy all that into the Code Editor.

Writing R Scripts

- To run the entire script, type `Ctrl + Shift + Enter`
- To run a single line in the script, click on the line and type `Ctrl + Enter`
- To save a script, type `Ctrl + S`. The file extension is `.R`

Commenting Your Code

Just plain code can be hard to read. If someone else (or a future version of you) tries to read this, it may not be obvious what the code does or why.

```
x <- c(1, 3, 4)

y <- mean(x)

z <- median(x)

y < z
```

Commenting Your Code

Comments make code easier to read. Insert a comment with `#`.

```
# This script creates a vector of data and returns TRUE if the mean 
# Author: Joe Ornstein
# Version: 1.0
# Date: August 26, 2020

# Create the vector
x <- c(1, 3, 4)

# Compute the mean
y <- mean(x)

# Compute the median
z <- median(x)

# Compare the mean and median with a logical expression
y < z
```

Now You Try!

Create, comment, and save an R script that does the following:

1. Create an object called `GDP` and assign it a vector of numbers (whatever numbers you like).

2. Create an object called `population` and assign it a vector of numbers. Make sure it is the same length as `GDP`.

3. Create an object called `GDP_per_capita`, which is GDP divided by population.

4. Put all three vectors into a `data.frame` called `data`.

5. Compute the mean and standard deviation of GDP per capita.

Okay. That's it for today.

Next week, we take our new data skills and make some **graphs**.