Lecture 1: Introduction to R

James Sears*
AFRE 891 SS 24
Michigan State University

*Parts of these slides are adapted from **“Advanced Data Analytics”** by Nick Hagerty and **“Introduction to Data Science”** by Rafael A. Irizarry, used under **CC BY-NC-SA 4.0**.

Table of Contents

Reading/Using the Slides

Reading the Slides

As you'll soon see, I frequently use **color to emphasize text**.

One distinction: **Links**

- **Underlined orange** is reserved for **links**
- Whenever you see **underlined orange text** on other slides, click it to go to the referenced content

Using the Slides

I highly recommend that you **replicate code in the slides as I go**[†].

- Try to type the code yourself first, use the code written in slides if you fall behind
- Good habit to create a **reference script** with all the new methods/functions as you go

These slides are written in **R Markdown**, which we'll cover next week.

- Slides are rendered as a standalone **web page (.html)** or **pdf file**
- Source code as an interactive **R Markdown file (.Rmd)**

† A wild link appears. Click it!

Course Introduction

Introductions

Me

[James Sears](#)

searsja1@msu.edu

Assistant Professor (environmental and consumer behavioral economics)

You

A quick roundtable of

- Names
- Program/year
- Fields/interests
- Coding Background

Syllabus Highlights

(Read the whole thing here: **891**, **991**)

Why This Course(s)?

Because a huge chunk of what's important for doing modern (high-quality) empirical economic research **isn't taught in core classes**.

- How to find, clean, and wrangle datasets
- Working with spatial data
- Creating professional-quality data visualizations
- Recent empirical methods outside the scope of core econometrics
- Fundamentals of data science

In short: these are the skills that will **increase your research productivity** and expand your opportunities while on the **job market**.

In shorter: these are the **skills I wish I had** going into my original research work.

Grading

Component	Weight
6 × **Homework Assignments** (13.33% each)	80%
Final Project	20%

- **Homework assignments** focus on coding practice through applied practice, replication, simulation, and/or extension
  - Contents are differentiated according to your enrolled course version
- **Final project** relates to applying course methods + research supply chain
  - Different specifics for 891/991

Final Project: 891

The final project for students enrolled in 891 has **two options**:

Replication

- Identify a paper you are interested in replicating that aligns with our covered course content - and replicate it!
- Create an RMarkdown file that you can hand to a fellow student that fully walks through the replication process

Data Acquisition

- Identify a data source that's not in directly machine-readable format (i.e. `dta, rds, csv`, etc.)
- Use the course's data acquisition techniques (web scraping, APIs) to construct the dataset
- Create an RMarkdown file that documents the process and executes all code needed to obtain the dataset

Final Project: 891

Both project types involve

1. **Project Proposal**: brief summary of the paper you have selected and why, and how the replication package is organized (before Spring Break)
2. **Material Submission**: Fully documented R Markdown file walking through full replication process or conducting data acquisition (Week 15)
3. **Presentation**: 8-10 minute conference-style presentation sharing what you did with fellow students (Finals Week)
4. **Discussant**: work through assigned student's material submission and prepare 3 minute presentation on your experience (Finals Week)

Grading:

Component	Weight
Proposal	3%
Material Submission	7%
Presentation	7%
Discussant	3%

Final Project: 991

The final project for 991 is a **research project** that relates to course methods + your planned research portfolio

- Creation of new dataset using course data acquisition techniques
- Replication + extension of existing paper (using course methods)
- Use course methods to tackle desired research question

Final Project: 991

Project deliverables include

1. **Project Prospectus**: one-page prospectus outlining the proposed project and planned direction (before Spring Break)
2. **Replication Package**: Fully replication package containing all necessary data/code and documentation needed to replicate your project (Week 15)
3. **Presentation**: 8-10 minute conference-style presentation summarizing your project (Finals Week)
4. **Discussant**: work through assigned student's replication package and prepare 3 minute presentation on how it went/any issues (Finals Week)

Grading:

Component	**Weight**
Prospectus	3%
Replication Package	7%
Presentation	7%
Discussant	3%

Course Schedule (I)

Intro to Data Science (Weeks 1-2)

- Introduction and R Basics
- Version control with Git(hub) and Productivity Tools

Data Processing (Weeks 3-5)

- Data Wrangling
- Data Cleaning

Data Visualization (Week 6)

- Principles of Data Visualization
- Common Chart Types
- Custom Themes and Extending `ggplot2`

Course Schedule (II)

Data Acquisition (Weeks 7, 9A)†

- Finding and Acquiring Data
- Considerations for Administrative or PII Data
- Scraping Static Webpages
- Scraping Dynamic Webpages
- Respectful Web Scraping

Programming (Weeks 9B-10)

- Conditional Logic
- Function writing
- Indirection and Name Injection
- Vectorization
- Parallelization

† Week 8 is Spring Break!

Course Schedule (III)

Analysis (Weeks 11-12A)

- Fast Fixed Effects Regression and IV Analysis
- Causal Inference: Diff-in-Diff and Event Study Methods
- Synthetic Control Methods (canonical, synthetic DiD, Partially Pooled SCM)
- Producing Tables and Figures from Regression Output

Spatial Analysis (Weeks 12B-13)

- Intro to Geospatial Data
- Vector Data and Spatial Operations
- Raster Data and Integration
- Static and Interactive Mapping

Course Schedule (IV)

Machine Learning (Weeks 14-15)

- Fundamentals of Machine Learning
- Prediction Methods
- Classification Methods
- Model Selection and Regularization
- Regression Trees and Forest-Based Methods
- Machine Learning for Causal Inference

About R and RStudio

Why Are We Using R in This Course?

- It's free and open source
- It's widely used in industry
- It's widely used in academic research
- It has a large and active user community

Compared with Stata:

- More of a true programming language
- Steeper learning curve (takes more to start, ultimately more powerful)
- Faster for fixed effects models (often much faster)
- Many methods come to R earlier

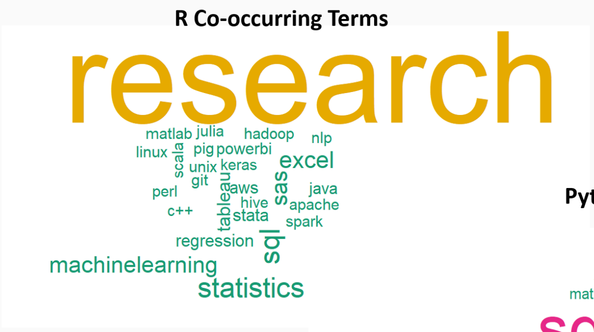
R vs. Python

R:

- Built for statistics and data analysis
- Better at econometrics and data visualization

Python:

- Built for general-purpose programming and software development
- Better at machine learning



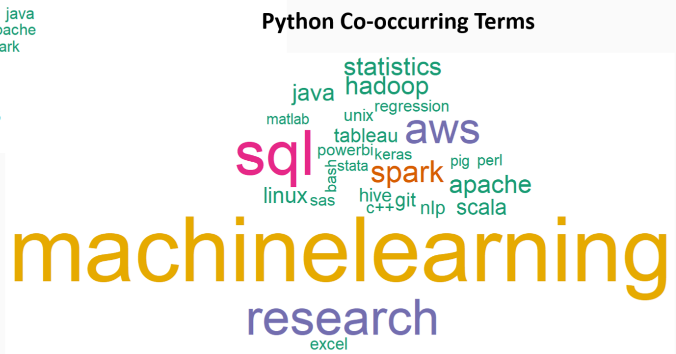


Image by Alex daSilva (source) is not included under the CC license.

R vs. Python

R:

- Built for statistics and data analysis
- Better at econometrics and data visualization

Python:

- Built for general-purpose programming and software development
- Better at machine learning

Most economists use **either Stata or R**

Many data scientists in industry use **both R and Python**

Rising competitor to both: **Julia**

R Is a Means, Not an End

- The goals of this course are **platform-agnostic**
  - It’s not about the syntax of specific packages
  - It’s about the concepts, logic, and thought processes underlying what we're doing and why
- Your eventual goal: **use the right tool for the job**
  - From this course, you'll have a good sense of whether **R** is the right tool
  - Or how to figure out if it is

R and Myself

- Personally, I use **R almost exclusively** because it gives me **one environment** for all steps of the research workflow
  - Cleaning/manipulating large datasets
  - Spatial data
  - Visualization
  - Econometric analysis
  - Web scraping
  - Machine learning
- While it might not be **the best** at all of these tasks, it's almost always **one of the best**
- Using different software for each task makes **reproducibility more difficult**

R and You

- Many of you will **know more than me** about these things! Please speak up and share if you
    - Find an error in the code/discussion
    - Know a better way of doing things
    - Have suggestions on improving the course

This is also the first implementation of concurrent course versions; I would love **any and all feedback** as we go!

R and RStudio

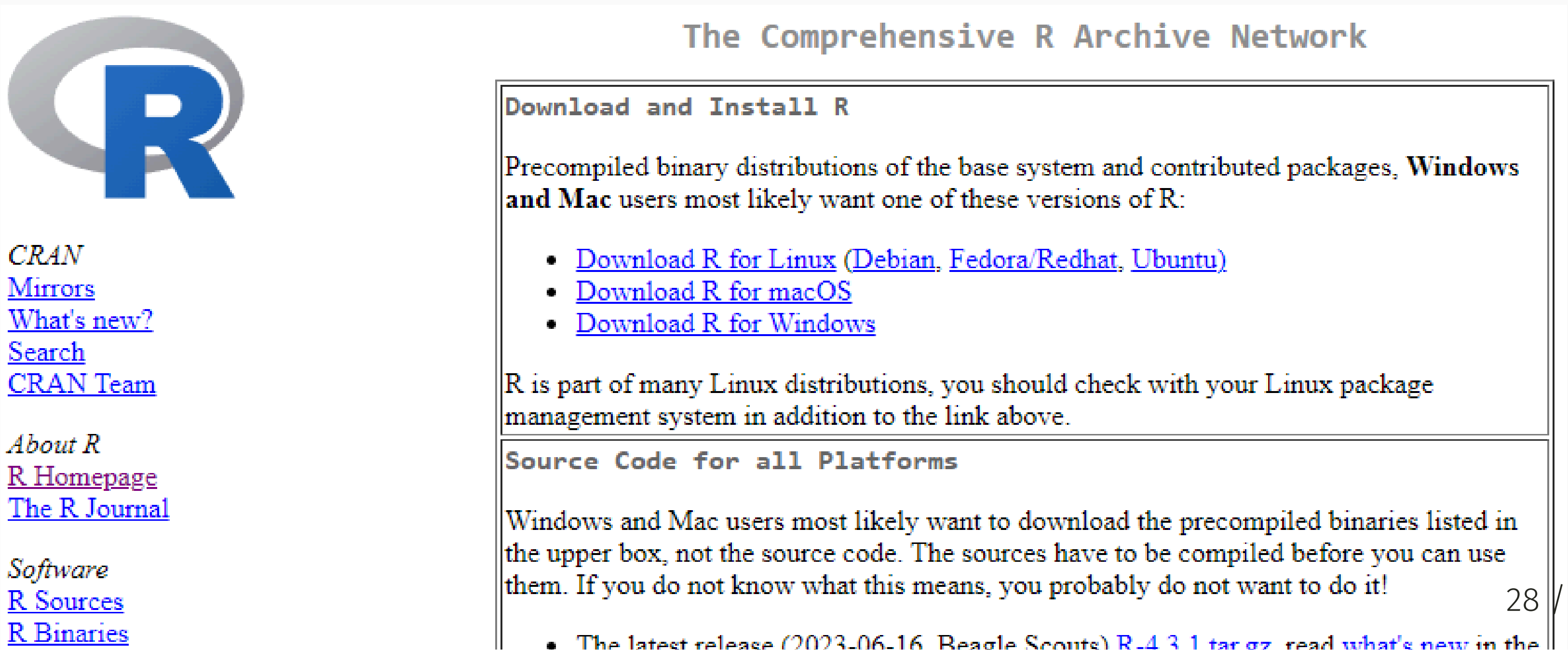
R vs. RStudio

- **R** is the **programming language**
- **RStudio** is the **environment** in which we use **R**
- While we could use **R** without **RStudio**, **RStudio** offers a lot of benefits

Installing R

To install **R**, go to the **R Project website**.

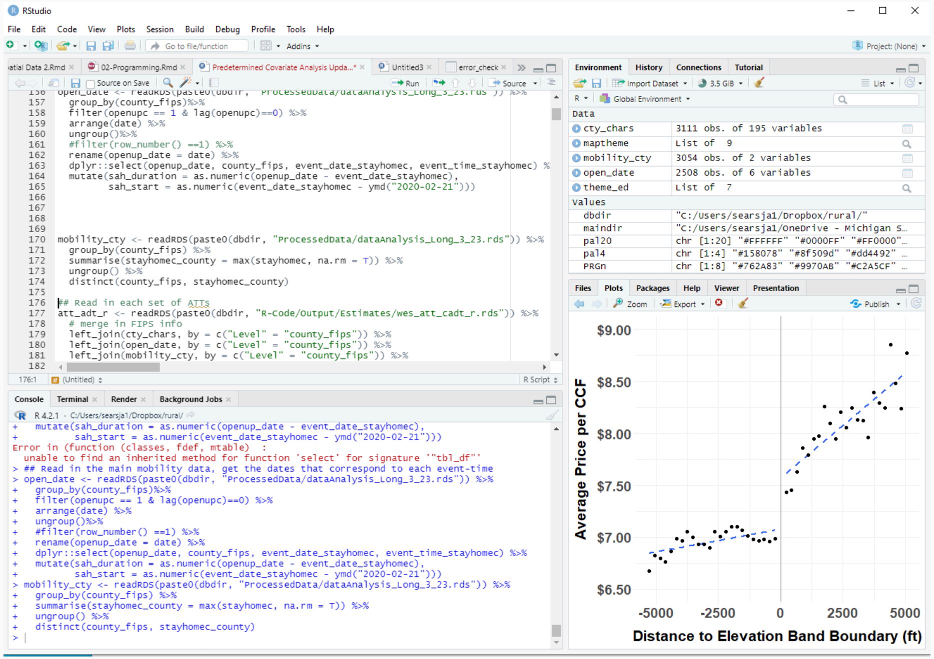
- **Windows:** "R For Windows > Base > Download R # for Windows"
- **Mac:** "R for (Mac) OS X > R-#.pkg"
- Where "#" is the current version number



RStudio

RStudio has a lot of features to make programming in **R more user friendly**

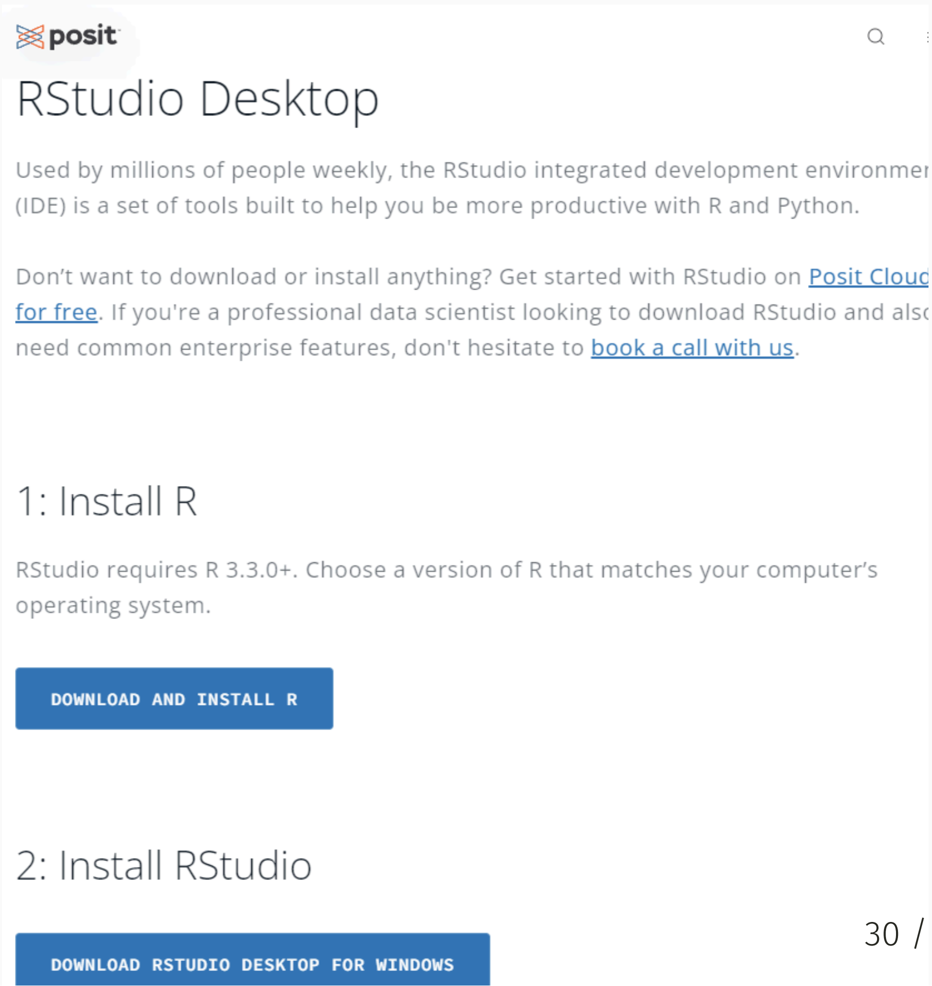
- Create and edit scripts
- View output and visualizations
- Navigate file structures
- See objects in memory



Installing RStudio

To install **RStudio**, go to the **RStudio Download Page**

- Scroll down, follow the link to install RStudio for your operating system.
- Correct file should be linked under **2. Install RStudio**
- Can scroll further down to the entire list and download the version for Windows or Mac.



Other Things

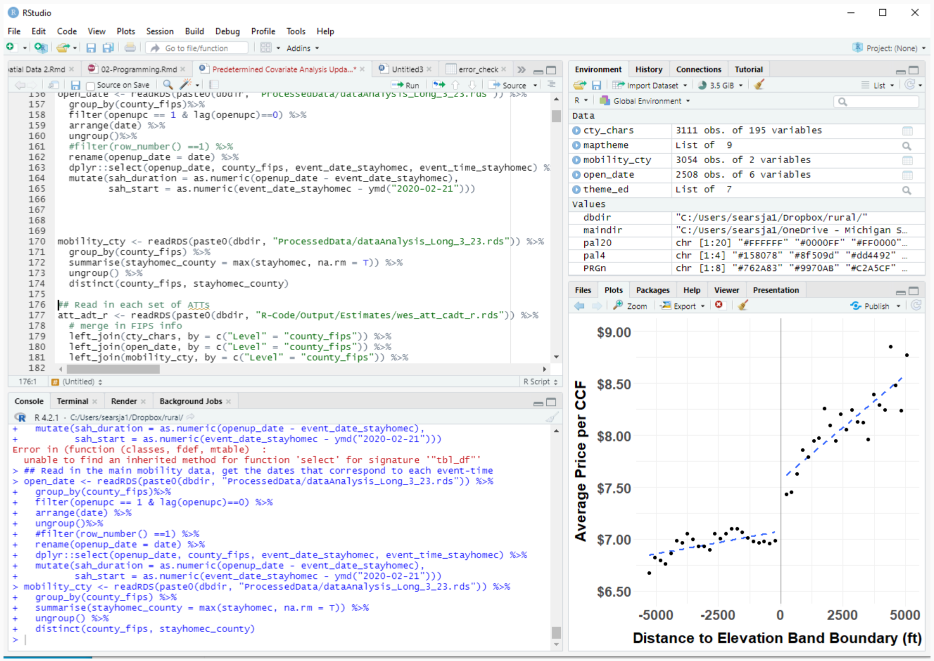
- Create an account on **GitHub** and register for a **student/educator discount**
    - You will soon receive an invitation to the course repo on GitHub, as well as **GitHub classroom**, which is how we'll disseminate and submit assignments, receive feedback and grading, etc.

- **Windows:** Install **Rtools**
- **Mac:** Configure/open your C++ toolchain (see **here**

RStudio

RStudio has **four main elements**

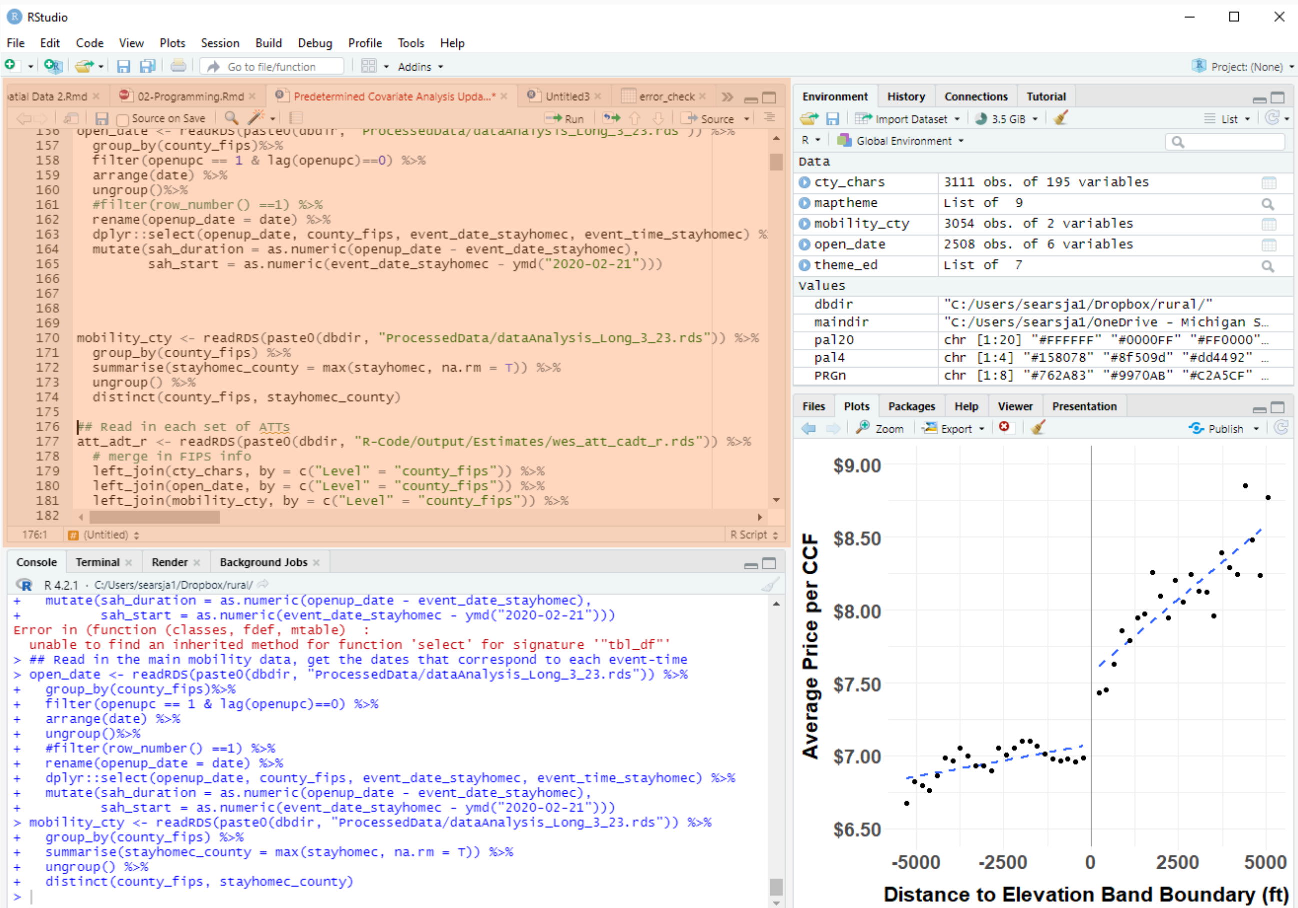
1. Script Window
2. Console
3. Environment
4. Files/Plots/Packages/Help Window



Getting Around RStudio: Script Window

Scripts are the **do file equivalent** in **R**:

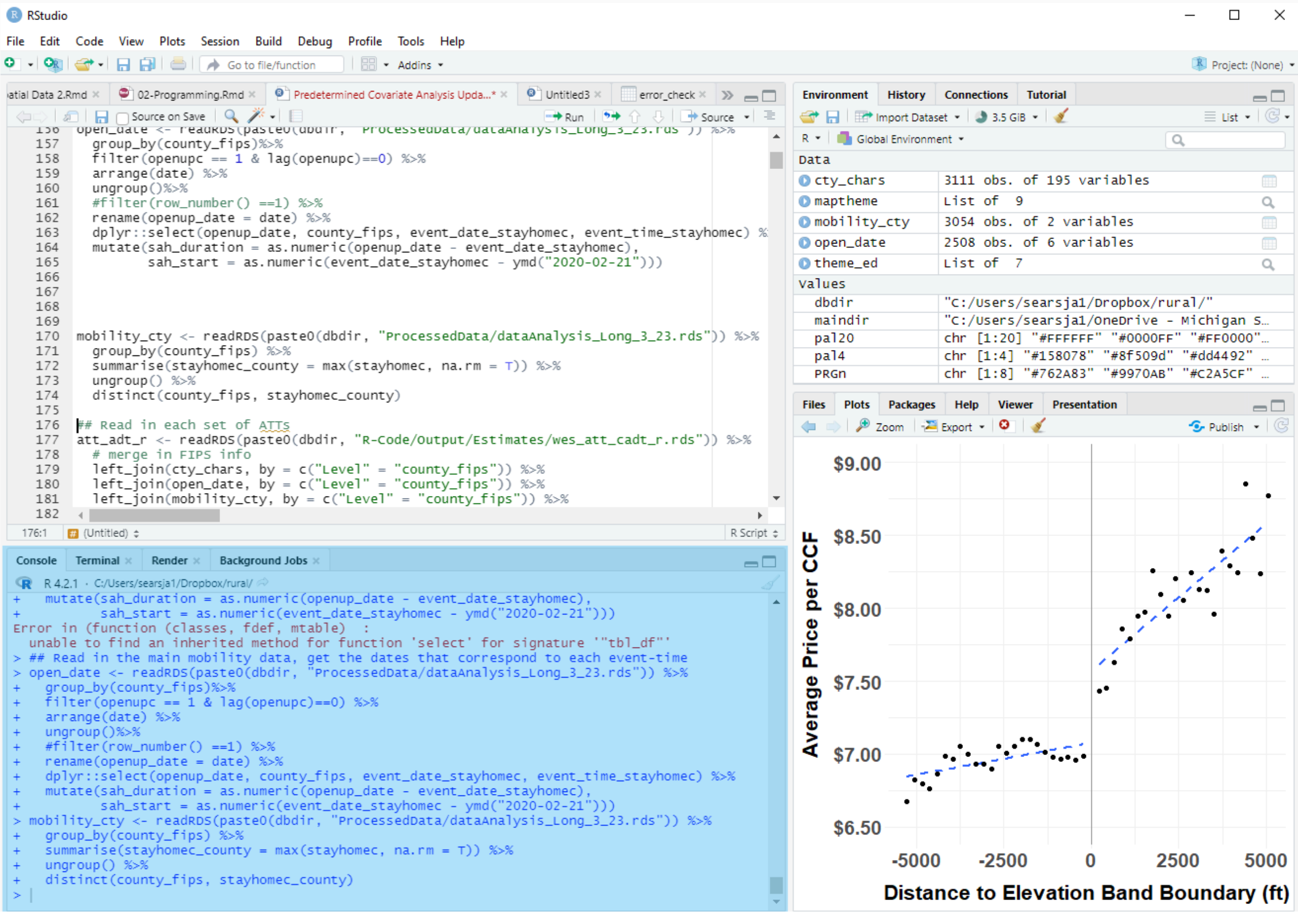
- Allow you to **write and save code**, flip through multiple scripts/objects



Getting Around RStudio: Console

Console is the **direct R interface**

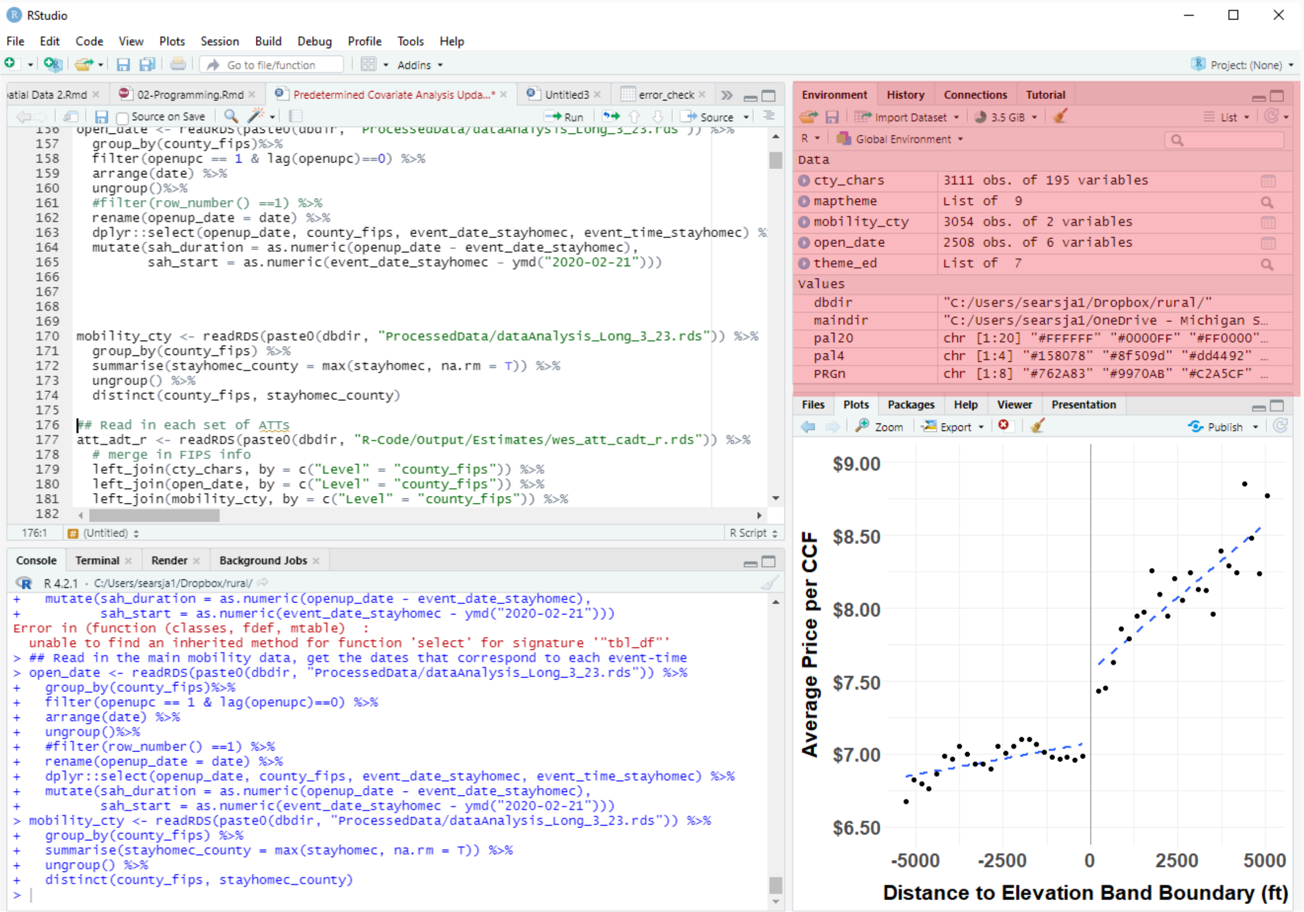
- View output or plug in code directly (use scripts!)



Getting Around RStudio: Environment

Environment shows you **everything currently loaded in R**

- Datasets, matrices, strings, functions, and other **objects**

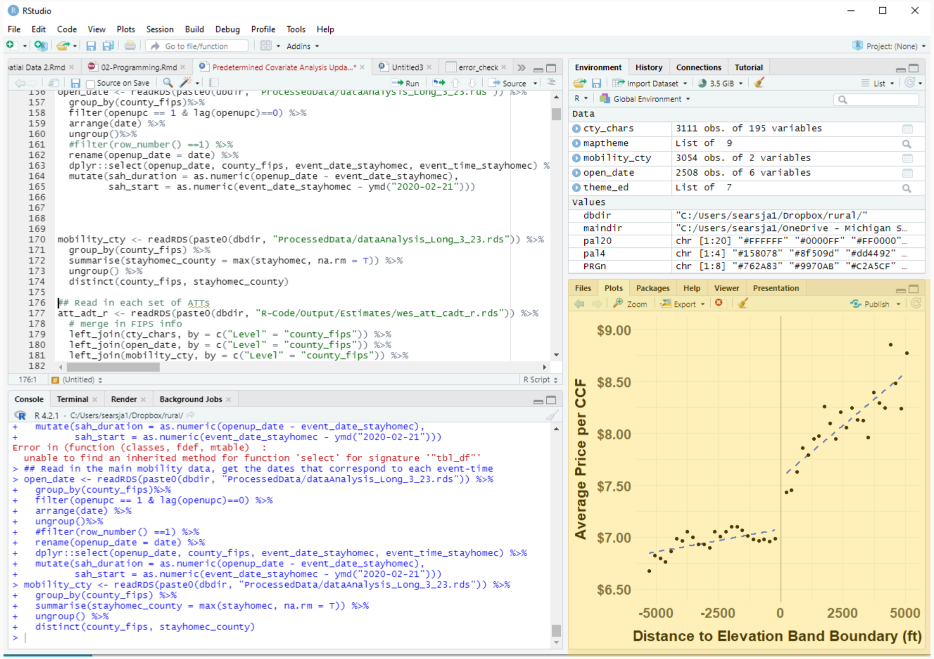


Getting Around RStudio: Files/Plots

Files/Plots/Packages/Help

shows you... **everything else**

- **Files** navigate file paths
- **Plots** view data visualizations
- **Packages** load/see packages
- **Help** get help with function syntax



Getting to Know RStudio

1. **Try out the console**
   - Use it as a calculator
   - Access previous commands
2. **Try a new script and save it**
3. **Set global options (Tools -> Options)**
   - Uncheck **"Restore .RData into workspace at start"**
   - Set **"Save workspace to .RData on exit"** to **"Never"**
4. **Keyboard shortcuts**

Time for Some Live Coding

Open a **new R script.**

As we go through examples, **retype everything yourself and run it line by line** (`Ctrl+Enter`)[1]. You'll learn more this way.

(Feel free to try out slight tweaks along the way, too.)

[1] For Mac users: `Cmd` = `Ctrl`. You can also use the `Run` button on the upper-right of the Script window to run code or view shortcuts.

Basic R Operators

Basic Arithmetic

You can use **R** like a **fancy, low-portability graphing calculator:**

```
1 + 2 - 3  # Addition/Subtraction
```

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

```
5 / 2  # Division
```

```
## [1] 2.5
```

```
4 * 3  # Multiplication
```

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

```
2 ^ 3  # Exponentiation
```

```
## [1] 8
```

Remember PEMDAS?

Parentheses **matter**!

```
2 + 4 * 1 ^ 3
```

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

```
(2 + 4 * 1) ^ 3
```

```
## [1] 216
```

```
2 + (4 * 1) ^ 3
```

```
## [1] 66
```

What feels like 95% of my coding errors are due to **unmatched parentheses**

This gets even more important as we use nested functions

Logical Evaluation

Logical operators follow **standard programming conventions**:

```
1 == 2 # == for equivalency (two equal signs)
```

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

```
1 > 2 # < and > work as expected
```

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

```
1 > 2 & 0.5 < 0.5 # The "&" means "and"
```

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

```
1 > 2 | 1 > 0.5 # The "|" means "or"
```

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

Negating Logic

Negate logical comparisons with `!` **and parentheses**

```r
1 != 2 # ! and = next to each other for neq
```

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

```r
!(1 > 2) # add parentheses around the condition you want to negate
```

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

```r
!(1 > 2 & 0.5 < 0.5) # can negate complex conditions
```

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

```r
!(1 > 2) | !(1 > 0.5) # or combine with other logical operators
```

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

Errors

What if we accidentally used `=` instead of `==`?

- `=` is reserved for **assignment**
- `==` is reserved for **equivalence**

```
1 = 1
```

```
## Error in 1 = 1: invalid (do_set) left-hand side to assignment
```

What should you do if you don't understand the error message?

- **Always read the error message!**

Commenting

Another good practice is to **comment code**

- Use the **pound sign** `#` to comment out everything behind it on a given line.

```
# Use it at the start of a line to add comments
4 > 3
```

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

```
# or to comment out parts of lines
4 + 5 - 23 # * 6798127347^38 yikes that would've been big
```

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

Commenting

Another good practice is to **comment code**

Widely accepted conventions:

- Put the comment **before** the code it refers to
- Use present tense

Script Chunks

You can use `# ----` or `#####` to add **collapsible chunks** to your scripts

```
# Use it Break up tasks (can put text in the middle) ----
4 > 3
```

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

```
#####
# or separate preamble from different sections
```

Click the **downward triangle** (left next to the line numbers) to **hide the lines between breaks**

Objects

Objects

R is an example of **object-oriented programming (OOP)**

- Everything is an **object**
- Everything has a **name**
- You do things with **functions**
- Functions come pre-written in **packages (i.e. "libraries")**
- You can (and should) **write your own functions too**

Understanding **objects** is the first key to using **R**.

Objects

We can store values for later by assigning them to **objects.**

We assign using one of two **assignment operators:**

- `<-` (the `<` followed by `-`),
- or `=`

```
price <- 149.99
tax <- 0.085
```

Here price "gets" the value `149.99`.

Choice of Assignment Operator

Why use `<-` instead of `=`?

Well, it turns out that it's a **lot more complicated of a question than it appears (warning: pedantic rabbit hole)**

My recommendation:

- Use `<-` to **assign objects to memory**
- Use `=` for **declaring function arguments**

Another reason: Google asks their developers to assign objects with `<-` in the **Google RGuide**, so if it's good enough for Google it's good enough for me.

Objects

To see the value of an object, just **type its name:**

```
price
```

```
## [1] 149.99
```

Notice that `price` and `tax` are now listed in your **Environment pane**.

Now, we can calculate the sales tax:

```
price * tax
```

```
## [1] 12.74915
```

Objects

We can assign a new value to `price` and recalculate the tax amount:

```
price = 90
price * tax
```

```
## [1] 7.65
```

Note that object names are **case sensitive:**

```
Price = 99.99
PRICE = 9.99
sales.tax = 0.05
sales_tax = 0.075
```

```
Price * sales.tax
```

```
## [1] 4.9995
```

Object Names

Rules for naming objects:

- Names can include numbers
- Names can include periods `.` and underscores `_`
- Names **must start with a letter**

```
sales_tax <- 0.075
10tax   <- 0.1
```

```
## Error in parse(text = input): <text>:2:3: unexpected symbol
## 1: sales_tax <- 0.075
## 2: 10tax
##       ^
```

Types of Objects

We'll see throughout the course that there are many types of objects in **R**, including

- **Single values** (numbers, characters; like `price`)
- **Vectors**
- **Matrices**
- **Data frames**
- **Arrays**
- **Factors**
- **Lists**

We'll dive into many of these in-depth during this course, but let's take a quick look at each

Vectors

A **vector** is an **ordered collection of numbers or character strings** indexed by 1, 2, ..., n, where n is the length of the vector.

```r
# for ordered integers, can use :
int_1_10 <- 1:10

# or the combine operator to manually combine elements
c(11, 13, 14, 200)
```

```
## [1]  11  13  14 200
```

Vectors

Other useful ways to create non-sequential vectors include

Creating a sequence with `seq()`

```
seq(0, 10, by = 2)
```

```
## [1]  0  2  4  6  8 10
```

Repeating elements with `rep()`

```
rep(1, 5)
```

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

```
rep(c(1,2,3), 2)
```

```
## [1] 1 2 3 1 2 3
```

Vectors

A vector with both numeric and text entries **converts everything to text:**

```
c(7, "Michigan", 19, "State")
```

```
## [1] "7"        "Michigan" "19"       "State"
```

Vectors

You can **name elements** in a vector (with or without quotations)

```
state_founding <- c(MI = 1837, IN = 1816, IL = 1818)
state_founding
```

```
##   MI   IN   IL
## 1837 1816 1818
```

```
state_founding == c("MI" = 1837, "IN" = 1816," IL" = 1818)
```

```
##   MI   IN   IL
## TRUE TRUE TRUE
```

Or by using the `names()` function

```
state_founding <- c(1837, 1816, 1818)
states <- c("MI", "IN", "IL")
names(state_founding) <- states
```

Matrices

A **matrix** is... well, a matrix.

- A **two-dimensional collection** of numeric or character values indexed by integer pairs (i,j)

```
xmat <- matrix(c(1,2,3,4,5,6),
               ncol = 3,
               nrow = 2)
xmat
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Matrices

Fill **across columns** instead by setting `byrow = FALSE`

```
matrix(c(1,2,3,4,5,6),
             ncol = 3,
             nrow = 2,
       byrow = FALSE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Matrices

Combine matrices on the rows with `rbind()`

- Must have matching number of columns

```
ymat <- matrix(c(7,8,9),
               ncol = 3,
               nrow = 1)

rbind(xmat, ymat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    7    8    9
```

Matrices

Combine matrices on the columns with `cbind()`

- Must have matching number of rows

```
zmat <- matrix(c(7,8,9,10,11,12,13,14),
               ncol = 4,
               nrow = 2)

cbind(xmat, zmat)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    3    5    7    9   11   13
## [2,]    2    4    6    8   10   12   14
```

Arithmetic with Vectors and Matrices

Arithmetic operators apply **element-wise** to vectors and matrices.

```
vec1 <- 1:4
vec2 <- 11:14
vec1 + vec2
```

```
## [1] 12 14 16 18
```

Arithmetic with Vectors and Matrices

Arithmetic operators apply **element-wise** to vectors and matrices.

```
mat1 <- matrix(2:7, nrow = 2)
mat2 <- matrix(seq(4,14, by = 2), nrow = 2)
mat2/mat1
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    2    2    2
```

Arithmetic with Vectors and Matrices

To perform **matrix multiplication**, use `%*%`

- Remember rules for dimensions!

```
mat1 <- matrix(2:7, nrow = 2) # 2 x 3 matrix
mat3 <- matrix(seq(4,14, by = 2), nrow = 3) # 3 x 2 matrix
mat1 %*% mat3 # results in 2x2 matrix
```

```
##      [,1] [,2]
## [1,]   80  152
## [2,]   98  188
```

Data Frames

data frames are the data science version of a matrix

- Each **column** is a **variable**
- Each **row** is an **observation**
- Variables can have **names**
- Columns can be of **different types**
- Should sound familiar to Stata folks

```
ex_df ← data.frame(state = c("MI", "IN", "WI", "IL"),
                   pop_m = c(10.04, 8.87, 5.91, 12.55),
                   is_michigan = c(T, F, F, F))
ex_df
```

```
##   state pop_m is_michigan
## 1    MI 10.04        TRUE
## 2    IN  8.87       FALSE
## 3    WI  5.91       FALSE
## 4    IL 12.55       FALSE
```

Accessing Element(s)

We can access specific element(s) of vectors with their **integer position** and **brackets** `[]`

```
# get the first element of int_1_10
int_1_10[1]
```

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

```
# get the third and fourth elements
int_1_10[3:4]
```

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

Accessing Element(s)

We can do the **same with matrices and data frames**, as well as access entire rows/columns

```
# retrieve the third and fourth columns elements of row two in zmat:
zmat[2,3:4]
```

```
## [1] 12 14
```

```
# Replace the entire second row of ex_df with Pennsylvania:
ex_df[2,] <- c("PA", 8.24, FALSE)

# take the sum of elements in the third and fourth columns of zmat:
sum(zmat[,3:4])
```

```
## [1] 50
```

Accessing Element(s)

If an object has names, you can also index using those!

```
state_founding["MI"]
```

```
##   MI
## 1837
```

```
ex_df["pop_m"]
```

```
##   pop_m
## 1 10.04
## 2  8.24
## 3  5.91
## 4 12.55
```

Accessing Element(s)

We can access **entire columns** in **dataframes** with `$`

- Can combine it with indexing (`[]`) too!

```
# get the "pop_m" column from ex_df:
ex_df$pop_m
```

```
## [1] "10.04" "8.24"  "5.91"  "12.55"
```

```
# which is equivalent to using integer indexing
ex_df[,2]
```

```
## [1] "10.04" "8.24"  "5.91"  "12.55"
```

```
# combine with indexing to get third row element of "state"
ex_df$state[3]
```

```
## [1] "WI"
```

Arrays

Arrays are **n-dimensional collections** of numeric/text elements, indexed by an n-tuple of integers - i.e. (i,j,k) for 3-dimensional array

```
ar4 <- array(1:24, dim = c(2,2,2,3))
ar4
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 1, 2
##
```

Arrays

Just like with matrices, we can use integer indexing to retrieve specific element(s) from an array

```
# get first element
ar4[1,1,1,1]
```

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

```
# get 2×2 matrix in "last" position
# (2/2 in third dimension, 3/3 in fourth)
ar4[,,2,3]
```

```
##      [,1] [,2]
## [1,]   21   23
## [2,]   22   24
```

Factors

A **factor** is a special kind of vector, where each element has an associated **level (i.e. character label)**

- Useful for storing categorical variables
- R will treat each level distinctly

```
edu <- factor(c("bach", "hs", "some_col", "phd", "bach", "hs"),
                     # specify the order
          levels = c("hs", "some_col", "bach", "phd"),
          # specify the text labels
          labels = c("High School", "Some College",
                     "Bachelors", "Doctorate"))
edu
```

```
## [1] Bachelors    High School  Some College Doctorate    Bachelors
## [6] High School
## Levels: High School Some College Bachelors Doctorate
```

Lists

Lists are an **ordered collection of objects** (that may be of different types)

```
# combine many of our previous objects into a list
ex_list <- list(
  ex_df,
  int_1_10,
  xmat
)
ex_list
```

```
## [[1]]
##   state pop_m is_michigan
## 1    MI 10.04        TRUE
## 2    PA  8.24       FALSE
## 3    WI  5.91       FALSE
## 4    IL 12.55       FALSE
##
## [[2]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
```

Lists

List objects are indexed with **double brackets** `[[]]`

```
# get the second list object (the vector)
ex_list[[2]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Lists

You can also **name list objects** and reference them with `$`

```
# combine many of our previous objects into a list
ex_list <- list(
  df = ex_df,
  vec = int_1_10,
  mat = xmat
)
ex_list$vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Checking Object Types

Unsure what type an object is? Use `class()` to check:

```
class(ex_list)
```

```
## [1] "list"
```

```
class(zmat)
```

```
## [1] "matrix" "array"
```

```
class(ex_df)
```

```
## [1] "data.frame"
```

Converting Types

Use the `as.type()` group of functions to **convert between types**

```
num <- 1:4
char <- as.character(num)
char
```

```
## [1] "1" "2" "3" "4"
```

```
as.numeric(char)
```

```
## [1] 1 2 3 4
```

NA

If a conversion **isn't obvious**, you'll get an **NA**

```r
as.numeric("AFRE 891")
```

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

In R. `NA` contains **no information**

```r
NA == NA
```

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

```r
NA + 0
```

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

```r
is.na(NA + 0)
```

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

Other Special Values

Other special values:

- Infinity (`Inf` or `-Inf`)
- Not a Number (`NaN`)

```r
1/0
```

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

```r
-1/0
```

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

```r
0/0
```

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

Practice

Practice

Time for some practice:

- Create a matrix named `mat40` with 4 rows and 10 columns containing the values 1:40
- Save the object `row4` to memory as a vector of the fourth row of `mat40`
- Use indexing to retrieve
  1. The element in the 3rd row and 6th column
  2. The 5-7th elements from the 2nd row
  3. The entire 6th column
- Using 1-3, create a list with each object named `obj_X`

Functions

Functions

In order to do more than arithmetic in **R** we'll use **functions**

```
log(5)
```

```
## [1] 1.609438
```

To see the **arguments** a function takes, look up its help file with `?fn`

```
?log
```

Some arguments are required, while others are optional. You can see that `base` is optional because it has a default value: `exp(1)`.

Function Arguments

If you type the arguments in the expected order, you **don't need to use argument names**:

```
log(5, 10)
```

```
## [1] 0.69897
```

But if you use argument names you can 1) put them in **any order**, and 2) protect yourself from **counting incorrectly**

```
log(x = 5, base =  exp(1))
```

```
## [1] 1.609438
```

```
log(base =  exp(1), x = 5)
```

```
## [1] 1.609438
```

Function Arguments

We can use **objects as arguments**

```
val <- exp(1)
log(x = 10, base = val)
```

```
## [1] 2.302585
```

or **nest functions directly**

```
log(max(c(price,5,10,20,50,3,11,20)))
```

```
## [1] 4.49981
```

Where to Find Functions

Many functions that we'll regularly use can be found either

1. Already loaded in **R**
2. In **packages**

Later in the course we'll learn about **writing your own functions**, which is something you can (and should) do!

Packages, Libraries, and Paths

Packages, Libraries, and Paths

While using **R** like a calculator is fun, the real advantages come out when we load **packages**

- **Packages** are nice curated bundles of functions and tools that let us transform R into a data-slaying kaiju
- Think Voltron (or insert other dated reference here)
- Base R loads in several packages by default (stats, utils)

```
# View loaded packages:
(.packages())
```

```
## [1] "fontawesome" "knitr"       "stats"       "graphics"    "grDevices"
## [6] "utils"       "datasets"    "methods"     "base"
```

Packages

To use another package, we must first **call it** with `library()`

```
# call the haven package with library()

library(haven)

# what's now loaded?
(.packages())
```

```
##  [1] "haven"       "fontawesome" "knitr"       "stats"       "graphics"
##  [6] "grDevices"   "utils"       "datasets"    "methods"     "base"
```

Packages

If we try and load a package that's **not installed** or **misspelled**, we get an error

```r
library(hevan)
```

```
## Error in library(hevan): there is no package called 'hevan'
```

Installing Packages

To install a package, we can use `install.packages()`

```
install.packages("tidyverse")
```

R will search for a package named "tidyverse" on **CRAN** and install it if found

Once installed, you can call the package.

Installing Packages

My preferred way: use the **pacman** package to load/install packages for you:

```
# first, install the pacman package
install.packages("pacman")
```

Then use the `p_load()` function[2] to load desired packages in a single call, and automatically install any that are missing

```
pacman::p_load(haven, tidyverse, fixest)
```

[2] You can run a function from an (installed) package without loading it by using the `PACKAGE::package_function()` syntax

Installing Packages

Sometimes packages aren't listed on CRAN and you'll have to install them **directly from Github repositories**

For example, if we want to use the `synthdid` package, we would first need to run

```
# use the install_github function from the remotes package
remotes::install_github("synth-inference/synthdid")

# can also use install_github() from the devtools package
```

Package Best Practices

It's a good idea to begin your scripts with a **Preamble**

- Add comments so you know what you're doing (helpful for revisiting a year later during revision requests...)
- Load packages (make peace with this, Stata users)
- Set file paths (assign main project path as an object)
- Change any options
- Store custom ggplot2 themes (we'll learn these later on)

Preamble Example: Notes and Packages

```r
##############################################################
##############       Culpable Consumption       ##############
##############          Update 11-2023          ##############
##############################################################
## Created: 11-23-2023
## Updated 11-23-2023
## Purpose: Run updated analyses for the "Culpable Consumption" Paper

# make sure to have installed pacman with
# install.packages("pacman") before
# running the below line:
pacman::p_load(fixest, lubridate, rdd, rdrobust,
showtext, tictoc, tidyverse)


# Read in strings as non-factors, turn off scientific notation
options(stringsAsFactors = F, scipen = 999)

# use showtext functionality to write text in plots
showtext_auto()
```

Preamble Example: ggplot2 Theme

```r
# ggplot theme
theme_ed <- theme(
# set text sizes/spacing
 axis.text=element_text(size=16, family = "lato"),
 axis.title.y = element_text(size=16, family = "lato", margin=margin(r=10
 axis.title.x = element_text(size=16, family = "lato", margin=margin(t=10
 plot.caption = element_text(hjust = 0, face = "italic"),
 plot.title=element_text(size=18, family = "lato"),
 legend.text=element_text(size=16, family = "lato"),
 legend.title=element_text(size=16, family = "lato"),
# custom ticks/gridlines/background
 axis.ticks = element_line(color = "grey95", linewidth = 0.3),
 panel.background = element_rect(fill = NA),
 panel.grid.major = element_line(color = "grey95", linewidth = 0.3),
 panel.grid.minor = element_line(color = "grey95", linewidth = 0.3),
# tweak legend position
 legend.position = "bottom"
  )
```

Preamble Example: Paths

```
# Set main folder path to object
# depending on computer I'm using
if (Sys.info()["nodename"] == "DESKTOP-SHT9660" ){
  onedir <- "C:/Users/james/OneDrive - Michigan State University/Research/
} else if (Sys.info()["nodename"] == "JAMES-DESKTOP" ){
  onedir <- "F:/OneDrive - Michigan State University/Research/Culpable Co
}else {
  onedir <- "C:/Users/searsja1/OneDrive - Michigan State University/Resea
}

# set main project path as working directory
setwd(onedir)
```

Run `Sys.info()` to see information about your system, including its name (`nodename` element)

Cleaning Up

While we're talking about best practices, we should talk about how to **clean up** your environment.

You can **remove an object** you're done with from memory using `rm()`

```
rm(state_founding)
```

To remove **multiple objects**,

- Nest `c()` within `rm()` to remove **specific objects**
- use `rm(list=ls())` to remove **literally everything**

Cleaning Up

After running intensive functions or having RStudio open for a while, or removing large objects from memory, it's also a good idea to **empty the trash**.

Use `gc()` (garbage collect) to manually[3] free up previously-allocated memory

```
gc()
```

[3] R does this automatically at various points, but it's often worth running yourself to make sure you have every bit of usable ram available

Cleaning Up

Eventually, it's a good idea to **start a new RStudio Session**.

- Your **environment** is transient; don't get too attached to it.
- Save your **script**, don't save the **environment**!
- Recreate smaller objects by re-running your script later
- Save out **large objects** that took lots of time/compute to create
- Exit RStudio when done working to return system resources

Tidyverse and Base R

Tidyverse and Base R

In R there are two main workflow approaches:

1. **Base R:** using the built-in R functionality and position indices
    - More pure "programming" approach
2. **tidyverse:** opinionated set of packages with organized grammar and data structures
    - Easily string together processing steps, produce much better visuals
        - Great free online book that teaches tidyverse: **R for Data Science**

Despite what you may read on the internet, there is no **one right way** to use **R**.

We're going to cover a **combination of both** in this class, but leaning in favor of tidyverse.

Interacting with Data Frames

Interacting with Data Frames

Data frames are the **main tidyverse object**, so let's get some practice with them.

To start, let's **load a dataset**[4] from the `dslabs` package.

```
pacman::p_load(dslabs)

# load the historic co2 dataset

data(historic_co2)
```

[4] We will learn shortly how to load and save datasets from file, too.

Data Frame Info

To learn more about a data frame, you can

- Examine its **structure** with `str()`

```
str(historic_co2)
```

```
## spc_tbl_ [694 × 3] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ year  : num [1:694] 1959 1960 1961 1962 1963 ...
##  $ co2   : num [1:694] 316 317 318 318 319 ...
##  $ source: chr [1:694] "Mauna Loa" "Mauna Loa" "Mauna Loa" "Mauna Loa" ...
```

Data Frame Info

To learn more about a data frame, you can

- Look at **column names** with `names()` or `colnames()`

```
names(historic_co2)
```

```
## [1] "year"   "co2"    "source"
```

```
colnames(historic_co2)
```

```
## [1] "year"   "co2"    "source"
```

Data Frame Info

To learn more about a data frame, you can

- Display **basic summary statistics** with `summary()`

```
summary(historic_co2)
```

```
##       year              co2            source
##  Min.   :-803182   Min.   :177.7   Length:694
##  1st Qu.:-470498   1st Qu.:206.7   Class :character
##  Median : -43278   Median :236.9   Mode  :character
##  Mean   :-219753   Mean   :245.9
##  3rd Qu.:  -8924   3rd Qu.:271.8
##  Max.   :   2018   Max.   :408.5
```

Data Frame Info

To learn more about a data frame, you can

- **Examine the first few rows** of data with `head(X, nrow)`

```
head(historic_co2)
```

```
## # A tibble: 6 × 3
##      year   co2 source
##     <dbl> <dbl> <chr>
## 1   1959   316. Mauna Loa
## 2   1960   317. Mauna Loa
## 3   1961   318. Mauna Loa
## 4   1962   318. Mauna Loa
## 5   1963   319. Mauna Loa
## 6   1964   320. Mauna Loa
```

Data Frame Info

To learn more about a data frame, you can

- Directly **inspect it** - either by clicking on it in the environment window or using `View()`

Subsetting with Logicals

It's often useful to **subset** a vector based on properties of another vector

i.e. Only take years with CO2 concentrations above the 98th percentile:

```
high <- historic_co2$co2 > quantile(historic_co2$co2, 0.98)
historic_co2$co2[high]
```

```
##  [1] 379.80 381.90 383.79 385.60 387.43 389.90 391.65 393.85 396.52 398.65
## [11] 400.83 404.24 406.55 408.52
```

How many years match this condition?

- `sum()` coerces T/F to 1/0

```
sum(high)
```

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

Subsetting with %in%

Another useful way to **subset on a range of values** is `%in%`.

```
2024 %in% historic_co2$year
```

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

Let's create a 2010s version of the data only using years 2000-2010:

```
co2_2000_2010 <- historic_co2[historic_co2$year %in% 2000:2010,]
co2_2000_2010$year
```

```
##  [1] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2001
```

Subsetting with which()

Alternatively we can subset with `which()`

- `which()` returns the position indices of elements that match a **logical condition**

```
which(historic_co2$year %in% 1920:1929)
```

```
## [1] 93 94 95 96
```

```
co2_1920s <- historic_co2[which(historic_co2$year %in% 1920:1929),]
```

Challenge

Let's practice interacting with data frames:

- Change the column name `co2` to `co2_ppm` (use `names/colnames` and indexing)
- Examine the year variable. How is it measured/reported?
- Add a new column named `co2_ppb` as the annual co2 concentration in parts per billion
  - Use arithmetic and indexing, or
  - Create as vector and add in with `cbind()` (column bind)

Table of Contents