Session 1: Introduction to R

Jennifer Lin

2022-03-30

Welcome to MMSS 211-3, or more specifically, welcome to the R section! In the next few weeks, we will cover a lot of ground on one of the most exciting statistical softwares that is available for computational social sciences[1]. This quarter, we will learn to wrangle data and do all sorts of neat analyses. For for today, I am focusing on the basics of R. If you have taken a computer science class in the past, some of this might translate well for you. If you have not, that is OK too. I assume no prior knowledge in programming. Our goals today are threefold:

[1] Python can beg to differ.

1. Become familiar with the "lay of the land" of R and R Studio.
2. Understand the basic data structures for statistical computing and how to translate this theory to practice in R.
3. Introduce techniques to organize code and to troubleshoot issues.

Before we dive into the material, I want to provide a brief overview for how I plan to conduct these sessions going forward.

- Before class, I will post, to Canvas, the following:
  - Slides (PDF)
  - Handout (PDF)
  - Code (R Script and R Markdown)
- *Slides* are these current slides in PDF format
- *Handouts* are the narration to the slides that provide more detail about each topic
- *Code* files are for you to use during class sessions. You can choose between the R Script or R Markdown format. Both contain the same information.

In my opinion, the best way to learn R is by actively writing code. It often involves taking code that works and adopting it to your needs. This involve a lot of trial and error but with each line of code you break comes a new opportunity to build your R skills to be stronger than before. Therefore, for these sessions, I will try to integrate as many different datasets and perspectives as I am able so that you can find something that fits your interests. For the exercises, I leverage a "choose your own adventure" technique where you can pick the variables that interest you to explore. I cannot promise that I will cater to all interests, and this is harder to do especially in the early sessions. There, I am more interested in your ability to follow along. As we progress through the quarter, you will get more opportunities to pick and choose variables to practice the skills.

What is R

Now, to the fun part. What is R? When you launch R on your computer's terminal, the R program itself, or R studio, you get the following message:

```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (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.
```

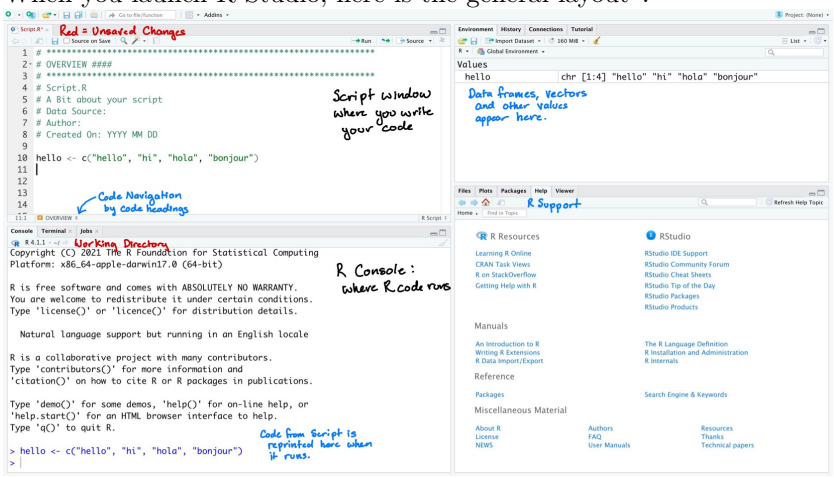
While most of us ignore this opening monologue (it is quite alright to do so), this message tells us a lot about what R generally is. Here is just a few lines that highlight the uniqueness of R. Specifically, R is…

- `free software and comes with ABSOLUTELY NO WARRANTY.` – R is free, since forever ago and likely will stay that way for a long time to come.
- `a collaborative project with many contributors.` – R is composed of a bunch of packages that people write and contribute to the cloud for all to use.

Difference between R and R Studio

You have likely written a paper for a class. When you write a paper, you can either engage with your computer's software to compile that paper yourself, or use one of the professionally developed software such as Microsoft Word, Google Docs or Pages. Inherently, each of these softwares use a similar underlying engine to convert the text you type to appear on the page and format it as you command. For us, R and R Studio operate similarly. R is the underlying software that does the math and computations while R Studio is the interface that we can use to help compile the code and view the results.

When you launch R Studio, here is the general layout[2].



[2] All of these are customizable, but by default, R sets it up as so.

On the top left corner is the R script. Here is where you should write all the code. R scripts have a `.R` extension and is saved as a plain text file. The title is on the top of the panel, where a red title indicated that you have unsaved changes. Directly to the bottom, you have the R console. This is the terminal that directly engages with R. When you run your code from the script, by hitting Command+ENTER (for Macs) and Control+ENTER (for PCs), a copy of the command you ran will print again in the console, followed by the output. IT IS CRUCIAL that you NEVER write code in the console because things written there is never saved. You will want to save your code so that you, or someone else, can reproduce your results. The upper right hand corner is the Environment window, which stores and shows all the objects you have created in your current R session. DO NOT save your environment. Finally, on the bottom right hand corner is a set of handy tabs that contains the function help window, plot viewer, file finder and other nifty tools.

Using R Scripts

When you engage with R, you will primarily do so using R Scripts. In this space, you can write code and insert comments to tell you and your readers what each line of code does. You can insert comments using the **#**. Any line that does not start with a **#** is treated as code. You can insert headings and subheadings in R scripts by using at least 4 number signs after the heading text. Newer R Studio versions also provide code navigation tools to help you browse through your headings to easily locate your code.

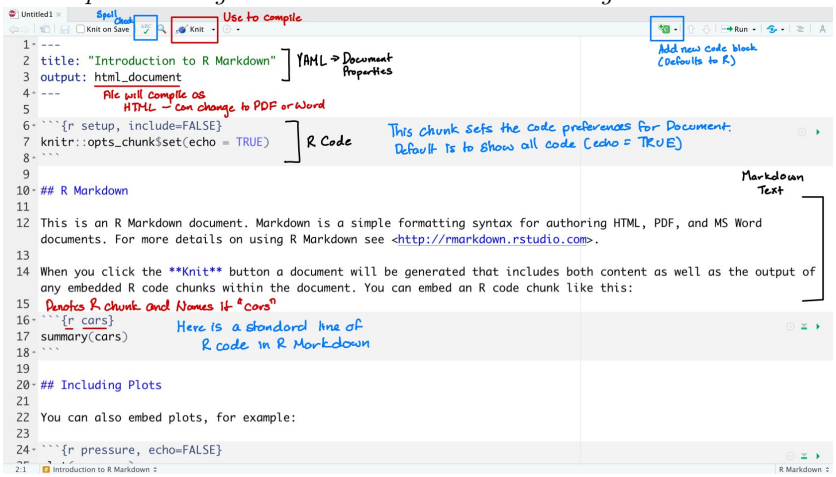
I have involved a document on Canvas about best practices for writing code. I would strongly recommend reading through it and thinking about how you want to approach coding each time you write R code. Everyone should develop a style of code writing that is neat, thorough and clear. I am in no rush for you to develop a system yet. However, here are my bare minimum expectations for good code:

1. ALWAYS comment your code.
2. Organize your code so readers know where one chunk ends and another begins.
3. Include proper indentation of code lines to show where a new chunk starts and where a continuation of code exists.
4. Limit the length of each code line so readers are not scrolling horizontally endlessly to follow your script.

Your future self will thank you for a clean code script!

Introduction to Markdown

This is a bare-bones introduction to Markdown. On Canvas, I posted a more comprehensive guide. I would recommend reading that.



Markdown is a plain text document that renders into formatted rich text using markup signals. Like an R script, it saves as a plain text document. Unlike an R script, you can compile the document to a PDF, Word or HTML document. A typical R markdown file includes fields for you to specify document properties (YAML), include marked up markdown text and R code. First, I will discuss the basics of markdown before discussing what makes R markdown so special.

In markdown, you type in the commands to tell the program to make something a heading, bold, italicize and so on. To insert headings, simply use the `#`

```
# Heading 1

## Heading 2

### Heading 3
```

To bold or italicize, use asterisks. You can **BOLD** text by inserting two asterisks before and after `**text that should be bolded**`. You can *italicize* text by inserting one asterisk before and after `*text that should be italicized*`.

We can also render math in markdown using LaTeX syntax. Inline math equations need to be enclosed in dollar signs such that $ax^2 + bx + c = 0$ is typed as `$ax^2 + bx + c = 0$` and

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

as

```
$$
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
$$
```

As you notice, one dollar sign provides us math in a line, whereas using two dollar signs provides math in a chunk on the center of the page. Also, using a back tick gives us verbatim text. Enclose text inline with one bark tick to show inline code `` `(some code here)` `` and use three back ticks to show code in a chunk.

````
```
Some code here
```
````

R markdown

R Markdown is a special kind of markdown because it allows you to integrate R code. Markdown files are often saved as `.md` and only al-

low for the markdown that we previously discussed. R Markdown documents allow you to integrate R code and the file is saved as a `.Rmd` file. You can compile R Markdown files to PDF (requires a LaTeX distribution installed), HTML, and Word (need Microsoft Word). The documents compile using the `knitr` package. The "Knit" button is on top of the R Markdown document window. Be careful as this will run all of your code each time. So if you still have broken code, your document will not compile.

The unique part of R markdown is the code chunks. Here, you can insert and run R code. To initiate a chunk, simply use three back ticks, along with a `{r}` such that:

```
```{r}
1 + 1
```
```

In each of these R chunks, you can include instructions to tell R Markdown whether to include the code in the document or not. Everything defaults to `TRUE`. There are many of these options, but a few of the most used ones include:

- `echo = TRUE` shows and evaluates code
- `eval = TRUE` evaluates code, FALSE does not
- `include = TRUE` includes the code in the output
- `warning = FALSE` suppresses warnings
- `message = FALSE` suppresses messages

For example, a code chunk like the following will tell R to run the code without including messages or warnings. It will tell R to show the code block in the output and run the code because `echo` and `eval` are not explicitly stated.

```
```{r, warning=FALSE, message=FALSE}
library(dplyr)
library(ggplot2)
library(tidyr)
```
```

Programming in R

Now that we have discussed scripts and markdown, we are ready to do some actual R programming. This will be the bulk of the sessions going forward.

To program in R, you will first need a basic understanding of the way R operates. R is an **object oriented programming language**. In essence, this means that objects are the central focus in R and

that everything in R is an object. So what are objects? These are functions, data structures, packages and anything, in general, that you can point an arrow to. And you literally point arrows to things in R. The *assignment arrow* (`<-`) is equivalent to an equal sign and it allows you to tell R that an object is equal to what follows the arrow.

R is a language, and as such, it has its own set of grammar rules. In English, a sentence is written as `noun + verb + other things`. In R, it is something similar. To run a function, commands often follow the format `Verb(Noun, Adjective)`. That is, in more colloquial terms, `Do Something(To What, How So)`. Formally, this is represented as `function(data, arguments)`. For example, if I want to calculate a mean, I would do the following:

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

mean(data, na.rm = TRUE)
```

Here, the verb is `mean()`, the noun is `data`, and the adjective is `na.rm`, which signals removing NAs.

Packages and Libraries

Earlier, we discussed how "`R is a collaborative project with many contributors.`", which implies that R has many add ons that people around the world have developed and made available for others to use. These add ons are known as **packages** and they are loaded using the `library()` function.

Packages in R are essentially likes apps on a phone or computer. They are a bundle of tools that help you achieve a certain goal – like wrangle data or draw graphs. There is an entire universe of R packages stored on the Comprehensive R Archive Network (CRAN), which houses most packages available to R users. Others might have packages that they do not publish. These can be found on GitHub and installed using `devtools`.

On Smartphone	In R
Download app	`install.packages("app_name")`
Open app	`library(app_name)`

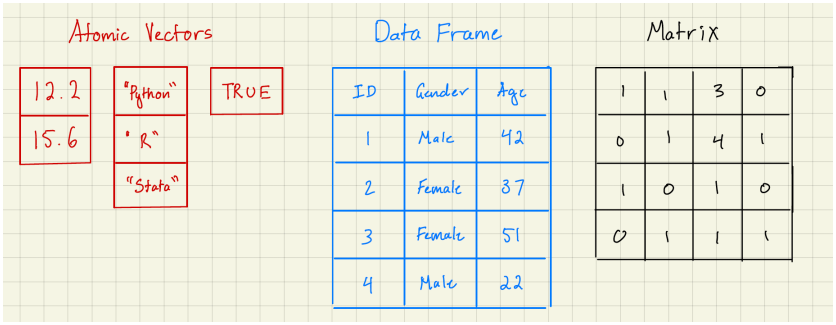
Data Structures in R

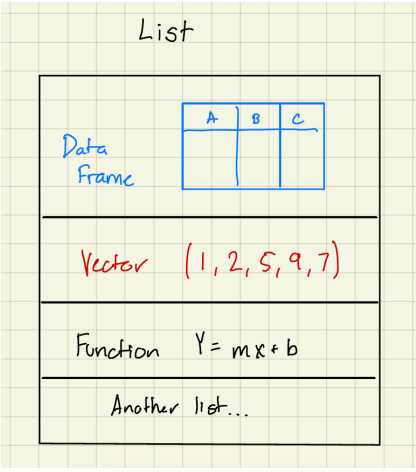
R can accommodate many types of data.

Type	Example
Character	`"dog", "cat", "fish"`
Double/Numeric	`-1, 2.8, 3, 4.5, 5, 6.4`
Integer	`1, 2, 3, -4, 5`
Logical	`TRUE, FALSE`
Complex	`1+4i`

In additional to the types of data described in the table above, you can combine these to form data frames, matrices or lists.

- *Vector*: A single strand of data.
- *Data Frame*: A collection of vectors where each row is usually an observation of an individual or other unit of analysis
- *Matrix*: An array of numbers
- *List*: A collection of data structures – data frames, functions, vectors, matrices and so on





Vectors

Vectors are a collection of values that are tied to an index that starts with 1*[In Python, this starts with 0.]. You can create vectors using

the function `c()` (concatenate). Put all vector elements within `c()`. To access aspects of a vector, simply use the square brackets `[]`A and include the index number within it. For example, here is a vector of statistics programs. I want to access the second element in the vector.

```
stats_programs <- c("Python", "R", "Stata", "SPSS", "SAS")

stats_programs[2]
```

```
## [1] "R"
```

Using the fact that we can locate items in a vector using the index number in the square brackets, we can easily extract values for a vector. For example, I am creating a vector of the first 12 numbers in the famous Fibonacci sequence.

```
fibonacci <- c(
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144)
```

I want to create a vector with only the first 10 numbers in this sequence. I can pull them out like so[3]:

[3] Notice that the colon (`:`) symbolizes "through". It works like a dash in plain English.

```
fibonacci[1:10]
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

Similarly, I can get the first, third and tenth number in this sequence, using `c()` that combines a number of things together for us.

```
fibonacci[c(1, 3, 10)]
```

```
## [1]  1  2 55
```

For numeric vectors, we can easily do some math to the entire vector. Any operation applied to the vector applies to all the units within it. For example, let's add one to each number in the Fibonacci sequence:

```
fibonacci + 1
```

```
##  [1]   2   2   3   4   6   9  14  22  35  56  90 145
```

Now, let's multiply everything by 4

```
fibonacci * 4
```

```
##  [1]   4   4   8  12  20  32  52  84 136 220 356 576
```

We can also edit the values in the vector. However, if you want to do this, BE CAREFUL! NEVER Write over the original data – actions cannot be undone!

You can edit a value using the index position number. For example, if the second number in the sequence should be 100, I can change it as follows. Notice that I create a new vector first so that I not writing over my original data.

```
new_fibonacci <- fibonacci
new_fibonacci[2] <- 100
```

We can now compare the outcomes

```
fibonacci
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144
```

```
new_fibonacci
```

```
##  [1]   1 100   2   3   5   8  13  21  34  55  89 144
```

Data Frames

Data frames are a collection of vectors, each column often with its own name and each row often representing an unique observation or unit of analysis.

let's use Baas R to create a small data frame of state names. First, I will create three different vectors that represent the state abbreviation, FIPS code and full name, respectively.

```
state_abbv <- c("AL", "AK", "AZ", "AR", "CA")
state_fips <- c(1, 2, 4, 5, 6)
state_names <- c("Alabama", "Alaska", "Arizona", "Arkansas", "California")
```

The `data.frame()` function works like `c()` in that it combines vectors together and calls it a data frame.

```
states <- data.frame(
  state_abbv,
  state_fips,
  state_names
)
```

We can also use the `tidyverse` packages to help us. Here, this function would be `tibble()`.

```
library(dplyr)
```

```
state_data <- tibble(
  state_abbv,
  state_fips,
  state_names
)
```

Here is the outcome:

state_abbv	state_fips	state_names
AL	1	Alabama
AK	2	Alaska
AZ	4	Arizona
AR	5	Arkansas
CA	6	California

When we work with data in R, we can either load our own data or call data from packages. When you do research, it will likely be the former, and if you are taking R classes, it will be the latter. For the most part, I will ask you to load in data, but for today, we will run it both ways.

The `tidycensus` R package has a dataset called `fips_codes`. This is a reference table that provides every state and county with their Federal Information Processing Standards (FIPS) codes that identifies the county. This is useful for GIS tools and working with census data in general. We can easily call it using `data()` after loading the package[4].

[4] Note that there is no need to install the package, though it is quite a neat one. I have included versions of this data in the folder that we will use to read in the data later on.

```
library(tidycensus)

data(fips_codes)
```

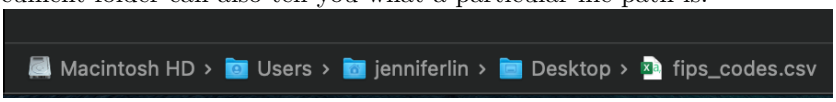
Here is what the first 6 rows of this data frame looks like.

state	state_code	state_name	county_code	county
AL	01	Alabama	001	Autauga County
AL	01	Alabama	003	Baldwin County
AL	01	Alabama	005	Barbour County
AL	01	Alabama	007	Bibb County
AL	01	Alabama	009	Blount County
AL	01	Alabama	011	Bullock County

We can suppose that we did not have access to this package and read the data in ourselves. When you work in R for research, this is often the step you need to take. Data would often not come pretty and packaged for you and may be presented in different formats. For today, I will demonstrate some of the more common ones, including the native `.RData` file made for R, plain text Comma Separated Values (CSV), Stata's `.dta`, and the JavaScript Object Notation (JSON). I

have included these four versions of the same data in the materials for today.

Before we load any external data, it is useful to have a quick discussion about file paths. File paths are a series of folder names and file name at the end that tell your computer where to locate a file. Your document folder can also tell you what a particular file path is.



You can use `file.choose()` to help locate a file but remember to insert the file path into the code when you have it so that your work can be reproduced.

If you put an R script and a data file in the same folder and open R from that script, you should be able to skip this step. Otherwise, you need to tell R where your data file is.

Loading External Data

In the accompanying materials, I am including four different files: `.csv`, `.RData`, `.json`, and `.dta`. We will look at how to read each of these file types in.

First, we start off with the CSV, which is likely the most common data type you will use. There are two ways to read in a CSV, using `read.csv()` in base R or using `readr::read_csv()`[5].

Let's take a look at the base R method.

[5] The `::` is notation for referencing functions within packages. Here, the format is `package::function()`. So `read_csv()` is a function in the `readr` package.

```
fips_csv <- read.csv("fips_codes.csv")
```

X	state	state_code	state_name	county_code	county
1	AL	1	Alabama	1	Autauga County
2	AL	1	Alabama	3	Baldwin County
3	AL	1	Alabama	5	Barbour County
4	AL	1	Alabama	7	Bibb County
5	AL	1	Alabama	9	Blount County
6	AL	1	Alabama	11	Bullock County

Notice how R automatically appends a row index. If we do not want that, we can use `readr::read_csv()`. Here, nothing gets appended.

```
library(readr)
fips_csv <- read_csv("fips_codes.csv")
```

...1	state	state_code	state_name	county_code	county
1	AL	01	Alabama	001	Autauga County
2	AL	01	Alabama	003	Baldwin County
3	AL	01	Alabama	005	Barbour County
4	AL	01	Alabama	007	Bibb County
5	AL	01	Alabama	009	Blount County
6	AL	01	Alabama	011	Bullock County

Another common file type is the `.RData`, which is essentially an R environment saved as a data file. Some people like these because they are rather lightweight and can be useful for large datasets. To read these in, you can use `load()`.

```
load("fips_codes.RData")
```

state	state_code	state_name	county_code	county
AL	01	Alabama	001	Autauga County
AL	01	Alabama	003	Baldwin County
AL	01	Alabama	005	Barbour County
AL	01	Alabama	007	Bibb County
AL	01	Alabama	009	Blount County
AL	01	Alabama	011	Bullock County

For the Python users among us, a favorite data storage method is the JavaScript Object Notation (JSON) data type. If you are working with APIs or other larger datasets, this will also be a common way the data are stored. To parse these data, you need the `rjson` package, which provides all sorts of tools for working with JSON files in R. We will need the `fromJSON()` function and the pipe from `dplyr`[6], and closing out was `as.data.frame()`. JSON files have a tendency to read in as a list and we need to convert that out from the list structure so we can work with it.

[6] More on this in a few weeks, but this function tells R to do something first then something else immediately after.

```
library(rjson)

fips_json <- fromJSON(file = "fips_codes.json") %>%
  as.data.frame()
```

state	state_code	state_name	county_code	county
AL	01	Alabama	001	Autauga County
AL	01	Alabama	003	Baldwin County
AL	01	Alabama	005	Barbour County
AL	01	Alabama	007	Bibb County
AL	01	Alabama	009	Blount County
AL	01	Alabama	011	Bullock County

Finally, since R is not the only statistical software that is out there, you may have co-authors who prefer things like SPSS, SAS, or Stata. Some data types may also be stored primarily in these file types. As a result, the `haven` package provides a great set of tools to help you read and write these files. For this section, I will demonstrate using Stata, and specifically the `.dta` file type.

```
library(haven)
```

```
fips_stata <- read_dta("fips_codes.dta")
```

state	state_code	state_name	county_code	county
AL	01	Alabama	001	Autauga County
AL	01	Alabama	003	Baldwin County
AL	01	Alabama	005	Barbour County
AL	01	Alabama	007	Bibb County
AL	01	Alabama	009	Blount County
AL	01	Alabama	011	Bullock County

From all of the data read ins, notice how we get the same data frame each time. This is a good thing. Your data should render the same way no matter what file type you use. If you are storing a dataset into different file types and loading one in gives you an error, it is likely that you made a mistake in reading in the data or writing the file to begin.

Accessing Variables

In R, you can easily access any given variable in a dataframe by referencing the dataframe object and then the variable, separated by a dollar sign. The format is `dataframe$variable` where the `dataframe` is the name of the dataframe of interest and the `variable` is the variable name of interest. Remember that all things in R are objects! So, if we want to get the variable that contains the state abbreviations in the `fips_codes` dataset, we will simply do:

```
fips_stata$state
```

Troubleshooting R Code

I know the content that we covered today was quite a lot. But hopefully, with practice, this will prove to be only an easy first step. The truth about learning R in the long run is that there are many trials and errors that you will need to encounter in order for you to have beautiful and functional code. That is why I find it fitting to close

the section with a brief discussion about ways to troubleshoot your R code.

For this quarter, I will be very happy to help you debug any issues. However, if and when you approach me, I will ask you about all the things you have tried. This is not because I want to brush you away. Rather, I want to help you develop habits to help yourself in the long term. By practicing troubleshooting skills early on, you can develop yourself to become an R expert in the future, without the need to rely on someone else. It is ultimately all about building intuition and having this intuition will pay dividends in the long run.

So, here are my recommended steps in troubleshooting R code.

Step 1: Check your Code

The majority of code issues in R arise because of a typo in your code. Forgetting to add a `+` or a `%>%` can mean that your code will not run. If you forget to close a parenthesis when you opened one previously, that can also break your code. One of the most challenging parts here is keeping track of your open brackets and ensuring that you close each one[7]. Typos can also matter when it comes to object references. R is rather case sensitive. So while you know what object you are referring to, R might not unless it is spelled and cased correctly.

[7] Believe me when I say that this is the majority of my R issues.

Step 2: Look at the R Help Page

For each package and function, authors write a series of help files that you can access to understand how the function works, know the arguments and see examples. You can access these help pages with `?package::function()`.

Step 3: Google!

Google is your best friend when it comes to debugging R issues. Here are some helpful tips that I use when I am trying to fix my R issues.

1. Draw (on paper) your desired end result and find words around that
2. Use these words to craft a Google search
3. See what Stack Overflow has to say
4. If nothing useful comes up, take advantage of related searches

Step 4: Schedule a Meeting with Me

I am always happy to help! If everything else fails, come talk to me!

Email me at jenniferlin2025@u.northwestern.edu with questions or come by my Office Hours!