R Structures, Syntax and Functions

Jennifer Lin

2022-09-11

Welcome to Graduate School! And more specifically, welcome to the lovely world of R! 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 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.

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.

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...

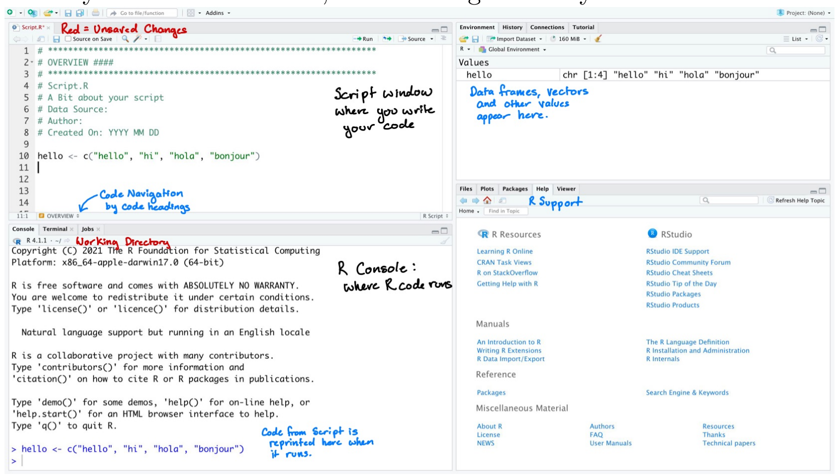
- `free software and comes with ABSOLUTELY NO WARRANTY.` – R is free, since forever ago and likely will stay that way for a long time to come.
- `a collaborative project with many contributors.` – R is composed of a bunch of packages that people write and contribute to the cloud for all to use.

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.



```
# ****************************************************************
# OVERVIEW ####
# ****************************************************************
# Script.R
# A Bit about your script
# Data Source:
# Author:
# Created On: YYYY MM DD

hello <- c("hello", "hi", "hola", "bonjour")
```

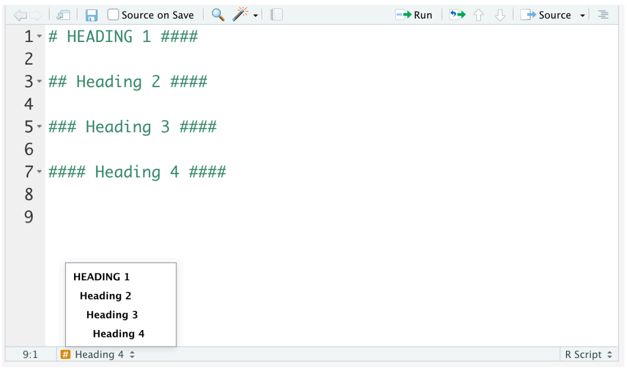
```
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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.

> hello <- c("hello", "hi", "hola", "bonjour")
>
```

```
Values
  hello          chr [1:4] "hello" "hi" "hola" "bonjour"
```

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.



```
# HEADING 1 ####

## Heading 2 ####

### Heading 3 ####

#### Heading 4 ####
```

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!

Programming in R

Now that we have discussed scripts, we are ready to do some actual R programming.

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)`

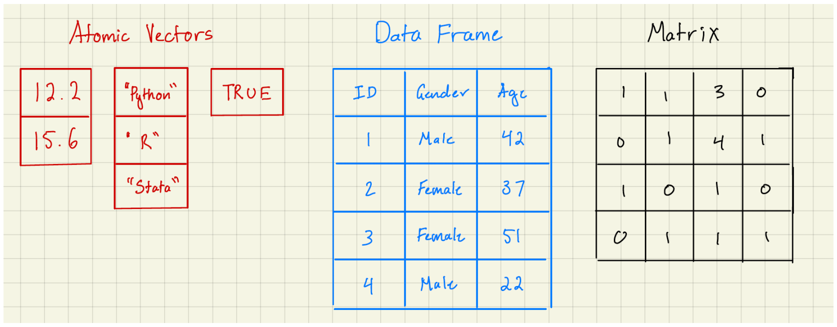
Data Structures in R

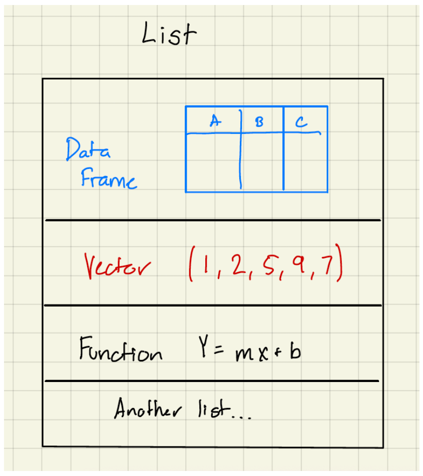
R can accommodate many types of data.

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.[3] 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.

[3] In Python, this starts with 0.

```
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[4]:

[4] 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[5].

[5] 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.

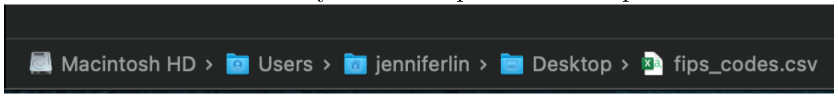
```
library(tidycensus)

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.


Macintosh HD › Users › jenniferlin › Desktop › fips_codes.csv

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()`[6].

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

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

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

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`[7], and closing out was `as.data.frame()`. JSON files have a tendency to read

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

in as a list and we need to convert that out from the list structure so we can work with it.

```r
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.

```r
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.

Working with Data

Now that you have seen how to work with R in a theoretical sense, let's take a look at some actual data that you might use at some point in your careers. The American National Elections Studies (ANES) is a nationally representative survey of adults in the US and is fielded each presidential election year. The ANES contains many items about political attitudes and behaviors. One of the things that it is most known for is the feeling thermometer, which is a measure of warmth towards an attitude object. This can be a person, group or ideology.

I created a clean version of this dataset to focus on the feeling thermometers. These data are from the 2020 wave.

Let's go ahead and read in the data using `read.csv()`:

```
ANES <- read.csv("ANES_2020.csv")
```

As mentioned previously, there are many feeling thermometer variables, here is a quick guide to all of these variables:

Variable	Label
FT_DemParty	Democrat Party
FT_RepParty	Republican Party
FT_Biden_pre	Joe Biden (PRE)
FT_Trump_pre	Donald Trump (PRE)
FT_Biden_post	Joe Biden (POST)
FT_Trump_post	Donald Trump (POST)
FT_Fauci	Anthony Fauci
FT_fundamentalist	Christian Fundamentalists
FT_Feminists	Feminists
FT_Liberals	Liberals
FT_laborUnions	Labor Unions
FT_bigBusiness	Big Businesses
FT_conservatives	Conservatives
FT_SCOTUS	Supreme Court
FT_gaymen	Gays/Lesbians
FT_Congress	Congress
FT_Muslims	Muslims
FT_Christians	Christians
FT_Jewish	Jews
FT_police	Police
FT_Transpeople	Transgender People
FT_Scientist	Scientists
FT_BLM	Black Lives Matter
FT_journalists	Journalists
FT_NATO	NATO
FT_UN	United Nations
FT_NRA	NRA

FT_Socialist	Socialists
FT_capitalist	Capitalists
FT_FBI	FBI
FT_ICE	ICE
FT_MeToo	#MeToo Movement
FT_rural	Rural Americans
FT_Parent	Planned Parenthood
FT_WHO	World Health Organization
FT_CDC	Centers for Disease Control

Accessing Variables in a Data Frame

The data that we loaded is a **data frame** and it is the same as the data frames that we discussed previously with the state FIPS codes. 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!

For example, to pull out the variable for party ID from the data above, you can use

```
PARTY <- ANES$PARTY
```

The `PARTY` variable also happens to be the second variable in our dataset. How do I know this? You can find out by viewing the dataset, using:

```
View(ANES)
```

I do not like the inclusion of `View()` commands in R scripts as it can take up a lot of run time, especially with large datasets. If this is something you want to do, pursue it on your own but do not send it to co-authors.

An alternative way to find out the sequence of variables is with `names()`

```
names(ANES)
```

```
##  [1] "X"                "PARTY"            "educ"
##  [4] "race"             "Voted_2020"       "gender"
##  [7] "FT_DemParty"      "FT_RepParty"      "FT_Biden_pre"
## [10] "FT_Trump_pre"     "FT_Biden_post"    "FT_Trump_post"
## [13] "FT_Fauci"         "FT_fundamentalist" "FT_Feminists"
## [16] "FT_Liberals"      "FT_laborUnions"   "FT_bigBusiness"
## [19] "FT_conservatives" "FT_SCOTUS"        "FT_gaymen"
## [22] "FT_Congress"       "FT_Muslims"        "FT_Christians"
## [25] "FT_Jewish"         "FT_police"         "FT_Transpeople"
## [28] "FT_Scientist"      "FT_BLM"            "FT_journalists"
## [31] "FT_NATO"           "FT_UN"             "FT_NRA"
## [34] "FT_Socialist"      "FT_capitalist"     "FT_FBI"
## [37] "FT_ICE"            "FT_MeToo"          "FT_rural"
## [40] "FT_Parent"         "FT_WHO"            "FT_CDC"
```

Here, we see that `PARTY` is the second variable. We can access it with the `$` or with matrix notation, which follows the format: `df[rows, cols]`.[8]

[8] `df` is the shorthand for data frame in R coding. It often symbolizes a generic data frame name.

You can do this using the name of the variable, in quotes.

```
PARTY <- ANES[, "PARTY"]
```

You can also do this with the position:

```
PARTY <- ANES[, 2]
```

Recall from earlier demonstrations with the Fibonacci sequence that you can call a value of a vector with its index. You do this with `vector[pos]`. You can do this with data frames too. However, instead of getting a single number, you get a data frame with one column.

```
PARTY <- ANES[2]
```

If you choose to select multiple columns, you can use `c()` as previously discussed.

```
DEMOGRAPHICS <- ANES[c(2:6)]
```

This provides the same results:

```
DEMOGRAPHICS <- ANES[, c(2:6)]
```

Let's put this all together. Notice that `grepl()` gives us names based on a pattern, called a "RegEx", or a Regular Expression. You can format a regular expression based on a common variable naming convention, or with the kinds of characters that are in the name, such as numbers, upper and lowercase letters, and symbols.[9]

[9] More on RegEx in the `stringr` R package and cheatsheet: `https://evoldyn.gitlab.io/evomics-2018/ref-sheets/R_strings.pdf`

```
Parties <- ANES[c(grepl("Party", names(ANES)))]
```

Now, if you want to be extra fancy, you can use the command `get()` to get particular variables. You can do it for a vector already in the R Environment, as follows:

```
get("PARTY")
```

Notice that this is doable because `PARTY` is a vector that we created after getting the `PARTY` variable from the ANES dataset.

You can also do it for a variable in a data frame, as follows:

```
get("gender", ANES)
```

Here, the second argument specifies where you are getting it from. By default, it is just from your environment. Notice, too, that the name of the variable of interest is in quotes. This is because anything *not* in quotes is treated as an object in the Environment. Things in quotes are character strings. These are useful to match variable names of interest. Since the variable is not an object (i.e. data frame, list, vector) in the environment, it *needs* to be in quotes. Understanding this will be useful in functions later on.

Variable Structure and Characteristics

Before you can do any sort of cleaning or statistics with your variables, you need to know what kind of variables they are. This is to say that you need to explore the dimensions of your dataset and get familiar with all the variables that you are working with.

Data Frame Dimensions

The inbuilt function, `dim()`, allows you to get a sense of the dimensions of your data.

```
dim(ANES)
```

```
## [1] 8280   42
```

The first element in this vector is the number of rows and the second is the number of columns. Notice how it should match the dimensions in the Environment window.

Structure of Data Frame and Types of Variables

Now that we have the dimensions of our data established, let's take a look at the structure of our data. Here, we are answering the question: "what kind of data are we working with?"

One way to do this is to use `str()`. This will tell you the kind of variable you are working with and give you the first few observations for each variable.

```
str(ANES)
```

```
## 'data.frame':    8280 obs. of  42 variables:
```

```
##  $ X                 : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ PARTY             : chr  "Republican" "Independent" "Democrat" "Republican" ...
##  $ educ              : chr  "Bachelor's Degree" "Some College" "High School" "Some College" ...
##  $ race              : chr  "Hispanic" "Asian" "White" "Asian" ...
##  $ Voted_2020        : logi  FALSE TRUE TRUE TRUE TRUE TRUE ...
##  $ gender            : chr  "Male" "Female" "Female" "Male" ...
##  $ FT_DemParty       : int  0 0 60 50 20 85 15 50 NA 60 ...
##  $ FT_RepParty       : int  85 50 0 70 70 15 75 100 NA 50 ...
##  $ FT_Biden_pre      : int  0 0 65 70 15 85 50 50 85 85 ...
##  $ FT_Trump_pre      : int  100 0 0 15 85 0 75 100 0 0 ...
##  $ FT_Biden_post     : int  0 15 85 100 0 100 0 50 85 70 ...
##  $ FT_Trump_post     : int  100 15 0 60 90 0 70 0 0 30 ...
##  $ FT_Fauci          : int  50 50 100 100 0 85 0 NA 30 50 ...
##  $ FT_fundamentalist: int  40 50 0 70 50 50 60 NA NA 50 ...
##  $ FT_Feminists      : int  65 100 75 70 30 60 60 100 50 50 ...
##  $ FT_Liberals       : int  30 0 75 70 10 70 0 NA 30 50 ...
##  $ FT_laborUnions    : int  30 70 75 70 50 50 50 0 30 50 ...
##  $ FT_bigBusiness    : int  70 50 0 85 0 40 50 0 50 15 ...
##  $ FT_conservatives : int  85 15 0 70 60 40 60 NA 50 50 ...
##  $ FT_SCOTUS         : int  100 50 25 85 60 60 70 50 50 50 ...
##  $ FT_gaymen         : int  85 100 100 70 50 70 60 NA 50 50 ...
##  $ FT_Congress       : int  40 15 0 100 10 85 50 50 50 40 ...
##  $ FT_Muslims        : int  85 90 100 70 10 50 60 NA 0 50 ...
##  $ FT_Christians     : int  85 100 30 70 50 85 60 100 50 50 ...
##  $ FT_Jewish         : int  100 100 100 70 90 100 60 50 50 50 ...
##  $ FT_police         : int  85 90 40 100 70 70 60 100 60 70 ...
##  $ FT_Transpeople    : int  70 90 100 70 50 70 70 50 50 50 ...
##  $ FT_Scientist      : int  100 70 100 85 60 85 85 NA 60 50 ...
##  $ FT_BLM            : int  0 70 80 50 0 70 0 NA 60 50 ...
##  $ FT_journalists    : int  40 15 75 85 50 85 60 50 50 50 ...
##  $ FT_NATO           : int  20 50 80 60 50 50 70 NA NA NA ...
##  $ FT_UN             : int  15 50 75 100 10 50 85 NA 60 50 ...
##  $ FT_NRA            : int  50 50 0 NA 70 15 85 50 NA 50 ...
##  $ FT_Socialist      : int  0 40 75 NA 0 60 0 NA NA 50 ...
##  $ FT_capitalist     : int  60 50 10 NA 60 30 70 NA NA 0 ...
##  $ FT_FBI            : int  50 50 50 85 20 70 79 100 60 50 ...
##  $ FT_ICE            : int  100 50 0 85 95 40 70 0 60 0 ...
##  $ FT_MeToo          : int  40 60 80 NA 0 50 70 NA NA NA ...
##  $ FT_rural          : int  100 60 75 NA 90 85 70 50 NA 50 ...
##  $ FT_Parent         : int  40 100 80 70 10 60 70 100 60 70 ...
##  $ FT_WHO            : int  0 50 80 100 0 70 70 50 NA 6 ...
##  $ FT_CDC            : int  15 50 80 100 10 70 85 100 70 70 ...
```

Notice that the output for this is very long. It does provide helpful information but if you have a large dataset, it might not be as helpful. Sometimes, we only care about the type of a specific variable. In that case, we can use `class()` or `typeof()` to find out what the type of the variable is.

Here is a demonstration using `class()`

```
class(ANES$PARTY)
```

```
## [1] "character"
```

```
class(ANES$FT_SCOTUS)
```

```
## [1] "integer"
```

And here is the demonstration using `typeof()`.[10]

[10] For Python users, `typeof()` might be more reminiscent of `type()` in Python.

```
typeof(ANES$PARTY)
```

```
## [1] "character"
```

```
typeof(ANES$FT_SCOTUS)
```

```
## [1] "integer"
```

Variable Type Conversions

Sometimes, the variable that is in the data set is not necessarily the variable that we want. Therefore, we might encounter situations where we need to convert it.

From the previous section, we saw that `PARTY` is a character variable. This means that the variable levels of “Democrat”, “Republican” and “Independent” are just words. They don’t have any grouping significance. We can call it a `factor` variable using `as.factor()` to give the levels some kind of significance, which will serve us well when we need to make plots.

```
PARTY <- as.factor(ANES$PARTY)

class(PARTY)
```

```
## [1] "factor"
```

```
SCOTUS <- as.numeric(ANES$FT_SCOTUS)

class(SCOTUS)
```

```
## [1] "numeric"
```

Sometimes, you might get the following error: `'list' object cannot be coerced to type 'double'`. In this case, use `unlist()` to convert the vector out of a list before putting it into the desired numeric form. This happens most often with integer class variables.

Basic Data Wrangling

Recoding Variables

In the future, you will learn of something called `dplyr`, which is a package that is based in the `tidyverse` series of packages. It is designed to help with data wrangling and contains a series of intuitively named functions that make this task a bit easier. However, before you can get to the easy stuff, you must gain a basic understanding of how data wrangling works in the back end. In this section, we will address a few of these tools.

Let's start by taking the `gender` variable and create a dummy where Female is TRUE and all else is FALSE.

```
ANES$FEMALE[ANES$gender == "Female"] <- TRUE
ANES$FEMALE[ANES$gender != "Female"] <- FALSE
```

Let's break this apart:

1. Notice that this code follows a specific formula: `df$NEW[df$old + Operator + old value] <- New value`
2. The operator in the recode process above is part of a few commonly used operation symbols as detailed in the table below.

Operator	Description
==	exactly equal to
<	less than
<=	less than or equal to
`%in%`	is in this list
!=	not equal to
`is.na()`	NA values

Now, let's try something with numeric values. Taking one of the feeling thermometers, let's recode anything higher than 51 as warm and anything below 49 as cold. Let's call 50 neutral. For the example, I will use feelings towards the Supreme Court.

```
ANES$SCOTUS_Cat[ANES$FT_SCOTUS >= 51] <- "Warm"
ANES$SCOTUS_Cat[ANES$FT_SCOTUS == 50] <- "Neutral"
ANES$SCOTUS_Cat[ANES$FT_SCOTUS <= 49] <- "Cold"
```

Viewing a Distribution of a Variable

Once you have completed the recoding process, you might be, no doubt, wondering if you recoded the variables correctly or what the distribution might look like. There are 2 commands in base R that will help you with that. These are `table()` and `prop.table()`.

`table()` Shows a raw distribution of counts

`prop.table()` Shows proportions. Takes a table object and can specify to go across the row (1), or down the column (2)

Here is a simple demonstration with `table()` and the categorical feeling thermometer we created earlier.

```
table(ANES$SCOTUS_Cat)
```

```
##
##    Cold Neutral    Warm
##    1553    1454    4360
```

Here is a demonstration of `prop.table()`.

```
prop.table(table(ANES$SCOTUS_Cat))
```

```
##
##      Cold   Neutral      Warm
## 0.2108049 0.1973666 0.5918284
```

Notice that, in the above, I used a nested function of `prop.table()` before `table()`. This is because `prop.table()` takes a table object. This is the same as:

```
tab <- table(ANES$SCOTUS_Cat)
prop.table(tab)
```

```
##
##      Cold   Neutral      Warm
## 0.2108049 0.1973666 0.5918284
```

Alternativel, we can just use `sum()` to find the counts on one level of a categorical variable.

```
sum(ANES$SCOTUS_Cat == "Cold", na.rm = TRUE)
```

```
## [1] 1553
```

Here, we indicate the variable that we want to count, the exact (`==`) value that we want to count and `na.rm = TRUE`, which signals that we want to remove NAs.

Summary Statistics in R

Now that we have seen how you can recode variables in base R, we can do simple computations

For the examples, we are going to take the feeling thermometer for the Supreme Court and calculate some basic statistics.

First, we can compute the **mean**:

```
mean(ANES$FT_SCOTUS)
```

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

Well, that did not work as anticipated. Why not? Let's look at the documentation for mean.

```
?mean()
```

From the documentation, we see the following:

```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

where `x` is a numeric or logical vector. The default for the argument `na.rm` is `FALSE`. `na.rm` is short for NA Remove. This takes out the NAs from the math. Let's see how this helps our mean computation from above.

```
mean(ANES$FT_SCOTUS, na.rm = TRUE)
```

```
## [1] 60.65834
```

Ah. Much Better!

We can do this with other functions to compute simple descriptive statistics. Let's look at the median.

```
median(ANES$FT_SCOTUS, na.rm = TRUE)
```

```
## [1] 60
```

We can also get quartiles:

```
quantile(ANES$FT_SCOTUS, na.rm = TRUE)
```

```
##   0%  25%  50%  75% 100%
##    0   50   60   75  100
```

We know the range is 0 to 100 because that is the bounds set by the survey, but just in case we did not,

```
range(ANES$FT_SCOTUS, na.rm = TRUE)
```

```
## [1]   0 100
```

We can compute the variance with `var()`

```
var(ANES$FT_SCOTUS, na.rm = TRUE)
```

```
## [1] 476.6355
```

And since the standard deviation (sd) is the square root of the variance:[11]

[11] $sd = \sqrt{var}$

```
sqrt(var(ANES$FT_SCOTUS, na.rm = TRUE))
```

```
## [1] 21.83198
```

Or, I guess, more simply:

```
sd(ANES$FT_SCOTUS, na.rm = TRUE)
```

```
## [1] 21.83198
```

Programming Central: For Loops and Custom Functions

Unlike Python, base R has more intuitive built in functions. Like Python, R can allow you to build your own functions, loop through analyses and conduct analyses if conditions are met.

For Loops

Suppose you have many of the same things that you need to do, like calculating the mean of many similarly structured variables or sampling randomly many, many times. It makes no sense to write the code for these computations all n times. Here is where **for loops** are very helpful.

Before going into the loops, let's revisit an earlier topic on getting the dimensions of a data frame. When we use functions like `gim()`, it gives us numbers for rows and columns. You can pull the number of rows and columns using the same logic behind pulling items from a vector. For example, here is the number of rows for the ANES data. Bear in mind that the `dim()` function returns

```
# Number of rows
dim(ANES)[1]
```

```
## [1] 8280
```

```
# Number of columns
dim(ANES)[2]
```

```
## [1] 44
```

But this is not always the most intuitive. There are better functions for this, including `ncol()` and `nrow()`. As the name suggests, `ncol()` gives us the number of columns of a data frame and `nrow()` gives us the number of rows.

What about vectors? Sometimes, you might need to loop through a vector. The functions described above will not work on a vector. For vectors, the `length()` function is what you need.

For example, consider the following vector:

```
age <- c(12, 34, 53, 24, 15)
```

There are 5 elements. Let's see if `length()` agrees.

```
length(age)
```

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

Perfect. Now, we are ready to think about loops.

In a loop, we are iterating over a designated vector and it takes on the following form.

```
for (variable in vector) {
  functions to loop
}
```

Here, `variable` is an indicator for what position you are in your vector. The most common input for `variable` is `i`. Hut, you can use anything. You need to make sure that the loop references the `i` or whatever name you use or your code will break.

Here is a more tangible example. I want to print a statement telling me what iteration I am on 10 times. The `variable` is `i` and the vector is from 1 to 10, denoted as `1:10`.[12]

[12] In R, if you want to generate a consecutive vector from x to y, you can use `x:y`. For example, generating a vector of numbers from 1 to 50 consecutively is `1:50`.

Here, to print the statement, I use the function `print()`, which prints something in the console. `paste()` joins things together. Here, I am having `paste()` join the character vector "This is Iteration" and the variable `i` denoting the position that we are in.

```
for (i in 1:10) {
  print(paste("This is Iteration", i))
}
```

```
## [1] "This is Iteration 1"
## [1] "This is Iteration 2"
## [1] "This is Iteration 3"
## [1] "This is Iteration 4"
## [1] "This is Iteration 5"
## [1] "This is Iteration 6"
## [1] "This is Iteration 7"
## [1] "This is Iteration 8"
## [1] "This is Iteration 9"
## [1] "This is Iteration 10"
```

Ok. This is great, but we can't use it. Why? Because the results are printed in the console and not saved as an object. Also because it is printing as 10 vectors, each of 1 item. Not helpful.

To make the output more helpful, we need to create empty buckets to hold our output. What these buckets are depends on what we need to store. And this is based on the outcome of the loop. Therefore, before creating the empty objects, think about what the outcome of your loop is. Are you generating one number or line per loop? You likely need a vector. Are you looping with regressions? Then you need a list.

So, how do we create these buckets? Let's start with a vector. We can create an empty vector using `c()`.

```
vector <- c()
```

If we want a vector with a specified number of units to fill, we can do

```
vector <- vector("numeric", 100)
```

Notice that the first argument specifies what kind of vector we want and the second specifies how many spaces we want.

To create empty data frames, do

```
df <- data.frame()
```

To make a list, it's

```
list <- list()
```

Ok. Now we are ready to redo our for loop and make it more meaningful.

```
iterations <- c()

for (i in 1:10) {
  iterations[i] <- paste("This is Iteration", i)
}
```

To make sure that the loop worked correctly, let's look at the `iterations` vector:

```
iterations
```

```
##  [1] "This is Iteration 1"  "This is Iteration 2"  "This is Iteration 3"
##  [4] "This is Iteration 4"  "This is Iteration 5"  "This is Iteration 6"
##  [7] "This is Iteration 7"  "This is Iteration 8"  "This is Iteration 9"
## [10] "This is Iteration 10"
```

GREAT! And that is a for loop. You can most definitely add more elements to it using the statistical functions that we talked about earlier or adopt new elements using functions from packages, or custom functions.

Custom Functions

As you become more advanced in R, you will learn about its many packages developed by users worldwide. Eventually, you, too, can develop your own R packages to conduct analyses needed for your research. But all that packages are, is simply a collection of custom functions that does things that base R can do in a more succinct way. For example. instead of doing `data <- df[c(1:4)]` to select variables, the `dplyr` package allows you to do this using `select()`.

Functions help alleviate the number of times you need to copy and paste the same long chunk of code over and over.[13] They can simplify your code as a result, making things easier to follow for your readers.

[13] **Pro tip:** If you are copying the same code at least 3 times, it is probably a good idea to write a function.

Here is the general format for a function.

```
function_name <- function(args){
  commands
}
```

To demonstrate this in the simplest terms, lets consider the following vector of hypothetical dollar values that five different people are paid per hour[14]:

[14] I know the rates are dismal but whatever. This is for demonstration purposes only

```
pay_per_hr <- c(8.5, 9, 8, 10, 9.5)
```

Now suppose that everyone is going to get a raise of 10% of what they had originally. We need to calculate the amount for the raise and then add it to the original amount.[15] This is a two step process so it is probably best that we write a function.

[15] Yes, I know doing 1.1 times the original gives us the same answer but I am demonstrating why functions are useful.

```
raise <- function(vec){
  out <- c()
  for (i in 1:length(vec)) {
    inc <- 0.1*vec[i]
    out[i] <- inc + vec[i]
  }
  return(out)
}
```

In this function, I am iterating through the length of a vector (the pay per hour vector) to get an outcome (`out`) that is 10% more than the original amount. Since the computations work on only one element in each vector, I need to use a for loop. The input is a vector (shortened as `vec`) and the output is another vector (shown as `out`). To specify what we want a function to spit out, we use the `return()` function. Here is that function in practice. Notice that to use a function, we can do `function_name(input)`. Our function name above was `raise()` and we are inputting a vector (`pay_per_hour`):

```
new_pay <- raise(pay_per_hr)
new_pay
```

```
## [1]  9.35  9.90  8.80 11.00 10.45
```

The output looks good! Bear in mind that this works because functions take objects from the environment. Any other part of an object in the environment will need to be retrieved using the methods to access them as specified earlier. For example, if we want to get a column from a data frame and pass that into a function, we need to supply the column as a character string in quotes rather than an object, without quotes. Let me demonstrate what I mean.

First, let's create a data frame with this new pay per hour rate and the number of hours each person works in a given day.

```
pay_df <- data.frame(
  hour_rate = new_pay,
  day_hours = c(10, 12, 9, 8, 10)
)
```

Now, I am going to demonstrate a simpler function that likely does not need to be written as a function. However, in this demonstration, the main point is to show you how to pull variables from an inputted data frame, along with a specified column name, and use it for computation.[16] As mentioned earlier, functions look for objects in the environment as inputs, which can be data frames or vectors or lists. Any other thing that is a part of an object, like a column name, needs to be supplied as a character vector.[17]

[16] The actual computation here is simple, but you can imagine universes where it can be harder and manipulations like these are necessary.

[17] If you want to pass a number into a function, you can simply put the number or write it as a vector of one element and pass that vector into the function.

In this function, I am going to cut everyone's work time by 2 hours and generate a new vector of hours that reflect this new work time. I am going to combine this vector with the original data frame and return the data frame as the output of the function.

To do this, I

1. Create a function that takes two arguments: The data frame as an object and a column name that is inserted as a character string.
2. Call the variable from the data frame into the function as a temporary vector (`temp`). Recall that, to call a column by name from a data frame can be done using `df[, "name"]` where name is a character.
3. Create a blank vector to store the output.
4. Run a for loop to subtract 2 hours from each person.
5. Join the new vector to the original data frame using `cbind()` (column bind[18] – joins two objects by columns – so if you have two data frames with 10 rows each, but one has 5 columns and the other has 6, you will get a new data frame with 11 columns and 10 rows.[19])
6. Return a data frame using `return()`

[18] The cousin function is `rbind()` or row bind, which joins two data frames with the same column names by rows. So if you have 2 data frames with the same names and one has 10 rows and the other with 20, you get a new data frame with the same number of columns but with 30 rows.

[19] This only works when there are the same number of rows/elements in the data frames/vectors that you want to join!

```
reduce_hour <- function(df, colname){
  temp = df[, colname]
  new_hr = c()
  for (i in 1:length(temp)) {
    new_hr[i] = temp[i] - 2
  }
  df = cbind(df, new_hr)
  return(df)
}
```

Now, let's apply this function. Notice that the arguments are presented in the order that I specified in the function and that the data frame name has no quotes but the column name has quotes.

```
pay_df2 <- reduce_hour(pay_df, "day_hours")
pay_df2
```

```
##   hour_rate day_hours new_hr
## 1      9.35        10      8
## 2      9.90        12     10
## 3      8.80         9      7
## 4     11.00         8      6
## 5     10.45        10      8
```

Perfect!

If Else Statements

Finally, we come to the last popular component of R programming[20] – the If Else statement. The If Else statement applies a function to an element of a vector if a condition is met, and if not, it goes to check

[20] There are other things like `while` statements that I will not cover.

the next condition, which, if it is met, it will run the function and so on, based on how many `if` conditions you have. Generally speaking, the If Else statement follows the following form:

```
if (condition is true) {
  do this
} else {
  do this instead
}
```

There are variations to this. For example, you can only have one if condition, where if something is true, it will apply a command but if it is not true, that it would be skipped. Here is what that might look like:

```
if (condition is true) {
  do this
}
```

You can also have nested If Else statements, where you have multiple if statements after the "else" of the previous if statement.[21]

[21] In Python, this is equivalent to `elseif`.

```
if (condition is true) {
  do this
} else if (condition is true) {
  do this instead
} else if (condition is true) {
  then do this
} else {
  when everything fails, do this
}
```

Let's look at an example. Suppose I create a vector of 1 element and I want R to print statements based on the value in the vector. If the value is positive, print "Positive Number". If it is negative, print "Negative Number" and if it is anything else, then print the value of the vector, stating that "X is VALUE."[22] Let's look at this statement in practice.

[22] `paste()` joins two character/numeric strings together with a space in the middle. The cousin, `paste0()` does this without a space. You can specify a desired separating character in `paste()` under the `sep =` argument. Currently, it defaults to a blank space.

```
x = -5

if (x > 0) {
  print("Positive Number")
} else if (x < 0) {
  print("Negative Number")
} else {
  print(paste("x is", x))
}
```

```
## [1] "Negative Number"
```

Notice that `x = -5` fails the first if, but passes the second, so it prints "Negative Number".

In this example, we notice that the If Else statement can only take on one element of a vector. If we want it to iterate through many elements in a vector, we will need the for loop. Let's look back to the `pay_df2` data frame from before, when we generated the new number of hours works for people. Now suppose that we want to make sure that everyone works 8 hour days from their original work times (`day_hours` variable). We will create a new variable that reflects the number of hours they currently deviate from the 8 hour workday. We can put this in a function. But we can also just write an If Else statement.

In the code below, I

1. Create a for loop that iterates through the length of the `day_hours` variable in `pay_df2`.
2. Start the If Else statement. Notice that, since I am storing the output in a variable in the data frame, I do not need to create an empty vector.
3. The first if condition is for people who work more than 8 hours. Since I want an absolute value of how many work hours they have that deviates from 8, I need to break the greater than, less than, and equal to conditions into three If statements
4. If the person does not work more than 8 hours, but they work less than 8 hours, the second If statement addresses their condition.
5. Finally, for all other cases (or equals to 8), put 0 since there is no deviation.

```
for (i in 1:length(pay_df2$day_hours)) {
  if (pay_df2$day_hours[i] > 8) {
    pay_df2$desc_hr[i] = pay_df2$day_hours[i] - 8
  } else if (pay_df2$day_hours[i] < 8) {
    pay_df2$desc_hr[i] = 8 - pay_df2$day_hours[i]
  } else {
    pay_df2$desc_hr[i] = 0
  }
}

pay_df2
```

```
##   hour_rate day_hours new_hr desc_hr
## 1      9.35        10      8       2
## 2      9.90        12     10       4
## 3      8.80         9      7       1
## 4     11.00          8       6        0
## 5     10.45         10       8        2
```

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.

I am generally very happy to help anyone debug R code. 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[23]. 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.

[23] 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.

Exercises

These exercises are doable with ONLY the code presented in this handout. You CANNOT use `dplyr` if you are already familiar with that. It will require you to be quite creative with how you place the pieces together.

Exercise 1

For this exercise, you are going to write some code that will allow you to loop through the ANES dataset and calculate the mean for each of the feeling thermometer variables.[24] Your solution should be a data frame with the name of the variable on one column and the mean on a second column.

[24] Remember that all the feeling thermometer variables start with `FT_`.

Hint: This exercise should take you 4 steps, with each step being 2 or 3 lines of code, at maximum. If you find that you are copying and pasting the same function 36 times, you are likely doing this wrong. Not to add more pressure, but there is a correct answer for how to do this. I will accept slight variations but nothing too crazy that deviates from what I am looking for. If you have a truly creative solution that I have never thought of, I will also accept it so long as it follows the rules. Good Luck!!!

Exercise 2

Find out the number of people who feel warm (over 51), neutral (at 50) or cold (less than 49) to each of the feeling thermometers in the ANES Data. The result should be a data frame with the variable name as one column, a column for number of cold, number of neutral, and number of warm.

Hint: Like Exercise 1, if you are copying something 36 times, it is likely wrong. We are looking at 5 major steps here. Good luck!