Statistical Programming for the Social Sciences Using R

Wesley Stubenbord

2024-02-11

Contents

Introduction

Welcome! This is the companion website for *Statistical Programming for the Social Sciences Using R*, taught at the Sciences Po Reims campus for the Spring 2024 term.

This website will contain the relevant tutorials for each week's lesson as well as other resources that you may find helpful throughout the course. The syllabus, assignment submission portals, and other files can be found on the course Moodle site.

Chapter 1

An Introduction to R

To get started, you will need to install two things:

1. R, a programming language
2. RStudio, a software program that helps you program in R
    - This type of software program is called an IDE, an Integrated Development Environment

You don't necessarily need RStudio to program in R, but it makes life much easier and it is what we'll be using throughout the course.

1.1 Installing R

To install R, go to https://cran.irsn.fr/index.html, select the appropriate operating system, and follow the instructions.

For example, if you have a Mac, you will click on "Download R for macOS," followed by the "R-4.3.2-arm64.pkg" link beneath the "Latest release" header.

If you have a PC running Windows, you will click on "Download R for Windows" followed by "install R for the first time" and "Download R-4.3.2 for Windows."

In either case, your browser will start downloading an executable installation file which you will then need to run to install R.

CAUTION - A couple of things you may need to watch out for:

- If you are using an older laptop (> 10 years old), you may need to download a different version of R or RStudio. If in doubt, read the instructions

on the download page and refer to your operating system version to find the right version.

- If you have very little hard drive space on your computer, you may need to clear some space before you install RStudio. The latest RStudio version requires 215 MB and you will likely need some additional space for other software and data we will be using in the course later on. Around 2 GB should suffice.

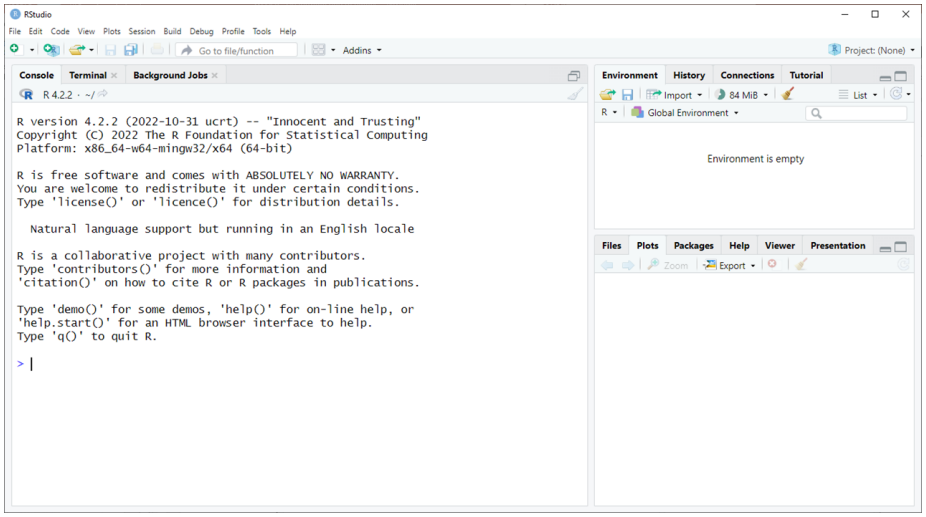
1.2 Installing RStudio

Once you've installed R, go to https://posit.co/download/rstudio-desktop/.

Posit (a company formerly known as RStudio) offers RStudio Desktop free of charge. Posit also offers a cloud-hosted version of the software (called Posit Cloud) which has both free and paid tiers. If you have trouble running RStudio Desktop on your computer, you may wish to consider using a Posit Cloud account, as described in the course syllabus.

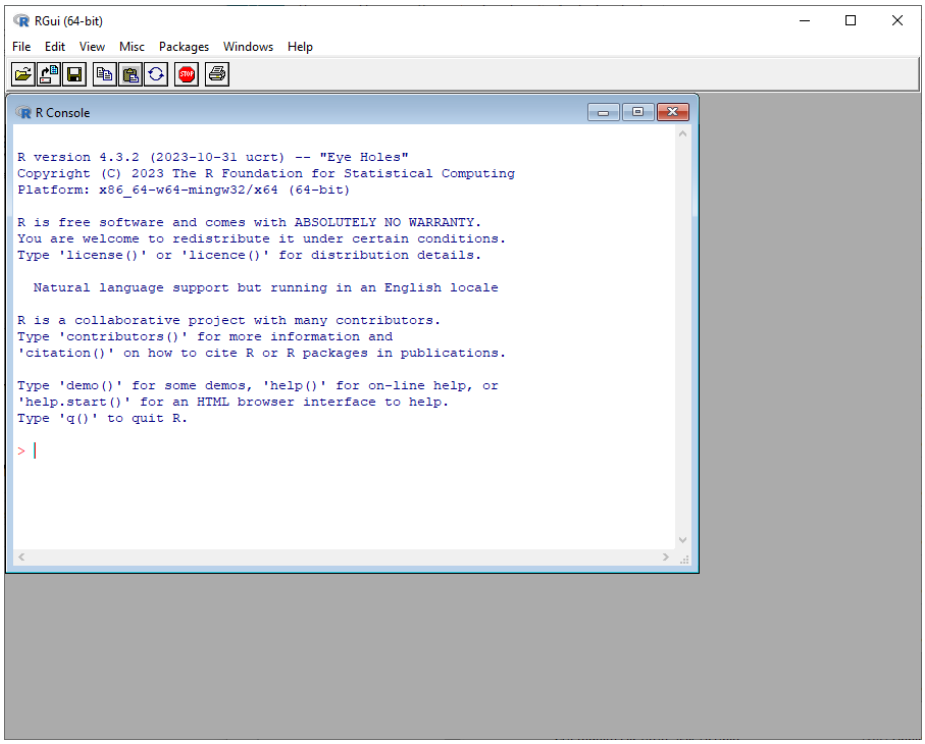
Step 1 is complete, you've already installed R. On the landing page linked above, you'll find different versions of RStudio according to your computer's operating system. Select the operating system that corresponds to your particular case (Windows, MacOS, or Linux), download the installer, and then run the installation file from your computer and follow the on-screen steps.

If all goes well, your screen should look something like this once you have RStudio correctly installed and running:



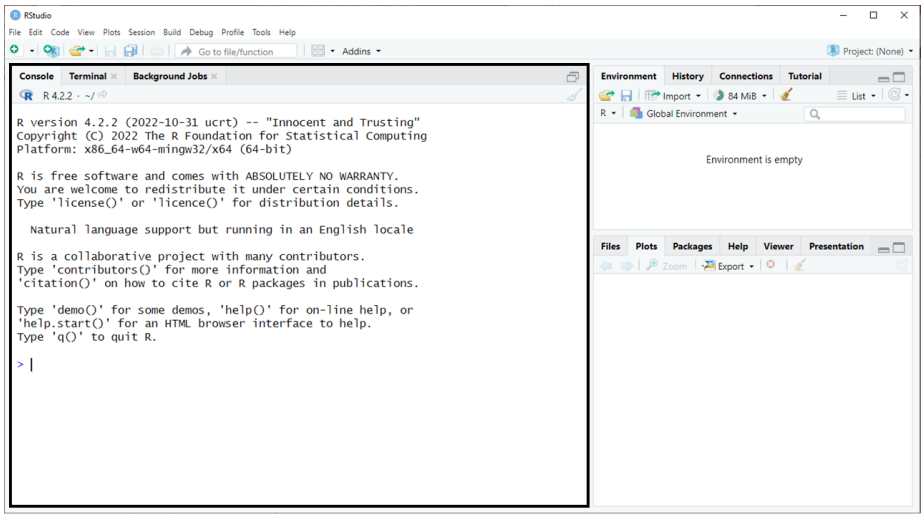
If your screen looks like the image below, it means that you've accidentally opened RGui, a basic graphical user interface included with R, and not RStudio.

We're always going to be working in RStudio for this class, so close out of RGui and open RStudio instead.



1.3 Using the Console

Now the fun begins. The RStudio window you've opened consists of a few different parts. The most important of these right now is the console pane (highlighted below).



The console allows you to interact with your computer using `R`. So, for example, if I want to use my computer as an over-sized calculator, I can type in the following `R` code in the console:

```
1+1
```

What happens when you press `Enter` on your keyboard? You get something like this:

```
1+1
```

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

You've provided an **input**, `1+1`, and received an **output**, `2`. In other words, using the language of `R`, you've told your computer to add one plus one and your computer has correctly interpreted your command and returned an answer, two. When your computer does not know how to interpret a command, usually because you've made a mistake, you will receive an error message as the output instead. Identifying errors and being able to correct them is an essential skill for a programmer.

One more note about outputs: the first number in brackets next to your output, `[1]`, indicates the number of the line of output. This is especially helpful when you are running code that generates multiple lines of output. We will see some examples of this later on.

For now, try entering a few more inputs, such as:

1. `10/3`

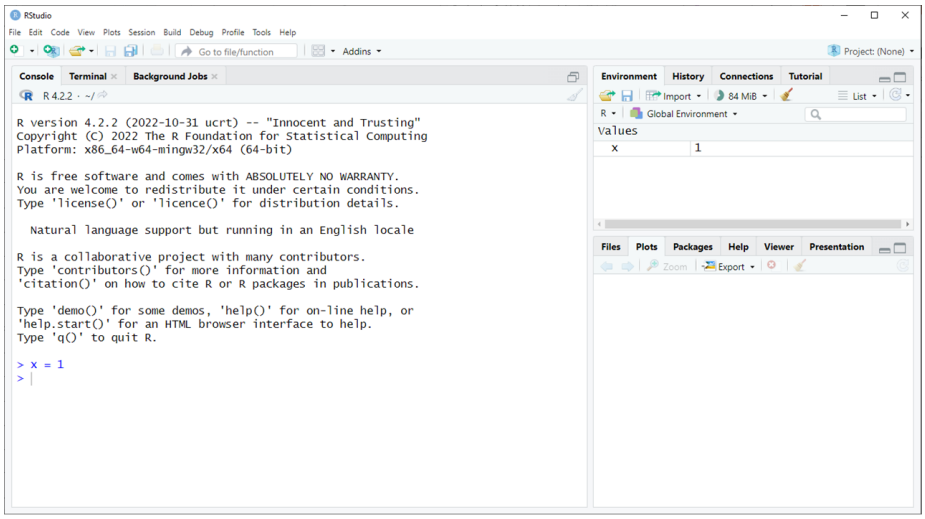
2. `(10/3) + 1`
3. `(10/3) + 1 + (5.111)\^2`

`R` is able to handle basic math operations with ease. What about other operations? Can you work with variables in `R`?

Try typing this in the console:

```
x = 1
```

What happens when you press `Enter`?



You may have noticed that there is no output. But, that doesn't mean nothing has happened. In fact, something has happened. You've stored a value, `1`, in a variable, `x`, somewhere in your computer's memory or in what we might call the environment. You don't receive an output, but RStudio reminds you of your new variable's existence via the Environment pane in the top right.

We can recall the value we input into our variable, `x`, by entering the variable name in the console:

```
x
```

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

See! Your computer remembers what you stored in your environment.

Try the following:

1. Can you assign a new value to your variable x?

2. Can you perform math operations on a variable (e.g., x*2)?
3. Can you create a new variable, y, and use it in math operations with x (e.g., x * y)?
4. Can you change the type of variable? What if, for example, I don't want x = 1, but I want x equal to the word "apple"?

1.4 Calculations with Variables

If you've made it this far, well done! Here's another thing you can try. Enter the following in the console:

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

You'll notice that we're using a different operator here. It's a less than symbol (<) followed by a dash (-). This is called an **assignment operator** and it has the same function as the equals sign (=). You can use either, I just so happen to prefer the way this one looks.

What happens when you press `Enter`? You have created a **vector**. Like other types of variables in `R`, a vector is an **object**. A vector holds a set of values of the same type. In this case, the object x contains a set of numbers, 1:5.

We can do all sorts of things with vectors and other objects in `R`. We can, for example, find the sum of a vector.

```
sum(x)
```

```
## [1] 15
```

How did we get an output of 15? 1+2+3+4+5 = 15. We can also find the mean of a vector.

```
mean(x)
```

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

And, we can perform other fun operations. Try the following:

1. Can you find the median of the vector x? What about the mode?[1]
2. What happens when you multiply a vector?
3. Can you create a new vector which consists of a set of letters?

[1] As you will have found, `mode()` doesn't calculate a statistic here (although even if it had, you still wouldn't have gotten a meaningful statistic for this particular vector). Some functions are easy to guess, like `median()` , but others can be false cognates just like in a spoken language. We'll talk more about finding the purpose of a function in the next chapter.

1.5 Saving Your Work

As you've started to see, working with a scripting language like `R` is quite different from working with software like Microsoft Excel or Google Sheets. You work interactively with data using code rather than by changing values directly in a user interface. No more clicking on cells to change values, now you change them programmatically.

One of the great advantages of interacting with data in this way, particularly for the social sciences, is that it allows us to see all of the steps you've taken to produce your analysis and repeat them. We don't have to take your word for how you've calculated something. We can see it and use it ourselves to produce the same thing.

This means that we leave our source data alone and write the code that produces the analysis. As with any good recipe, we want the code you write to be clear and easy to follow so that we can come back to it and understand what we did. We'll say more about how to do this later on.

There are a couple of different ways of saving your code:

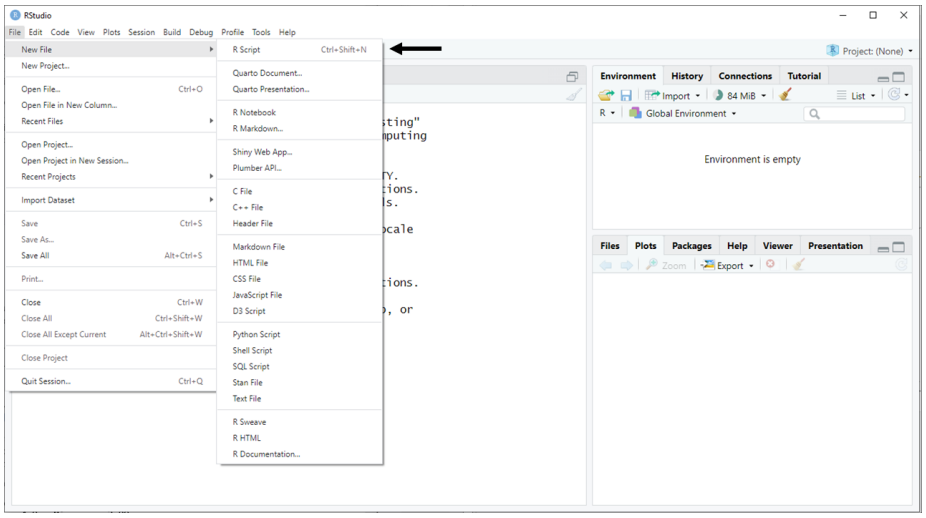
1. In an R Script, a simple text file ending in a `.r` extension

2. In an R Markdown file, an interactive format that allows you to see your code and the results together in the same file

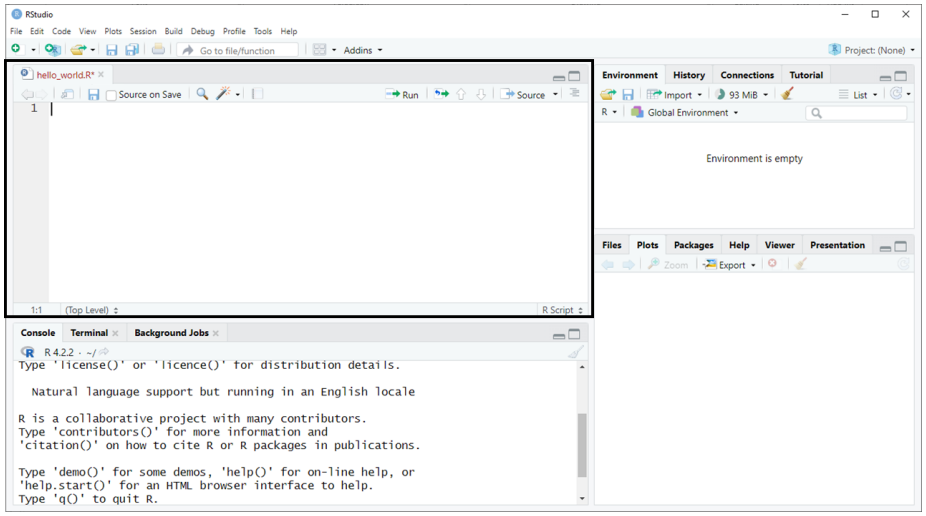
We're going to start with an R Script file and try out R Markdown later on.

1.6 Creating and Saving an R Script

To create an R Script file in RStudio, go to File > New File > R Script.

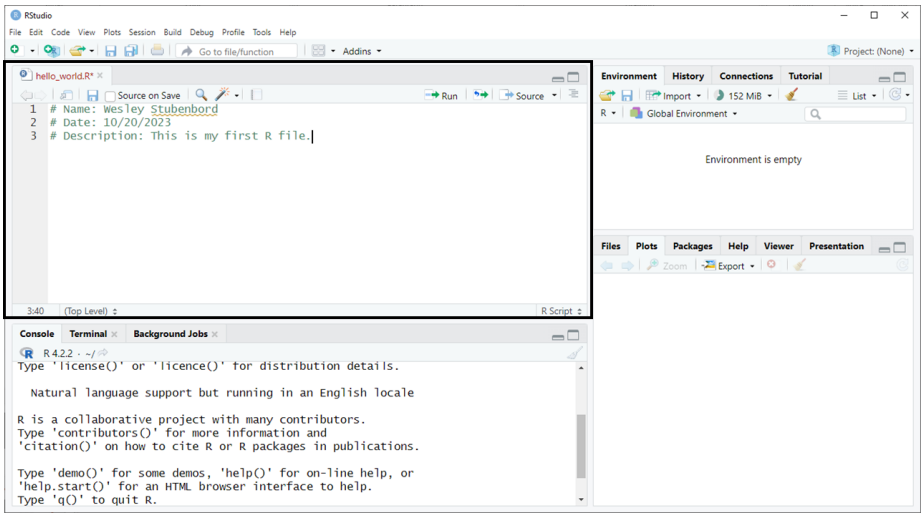


You should now have a window open in RStudio which looks like this:



You can enter comments in your R Script file using a hash tag (#) at the beginning of each comment line. A hash tag lets R know that this line should not be run as code. Its purpose is to tell us what is happening in a particular section of the code.

I like to start by adding my name, the date, and a description to each file I use. I'll ask that you use a header for each R file you submit for this class as well.



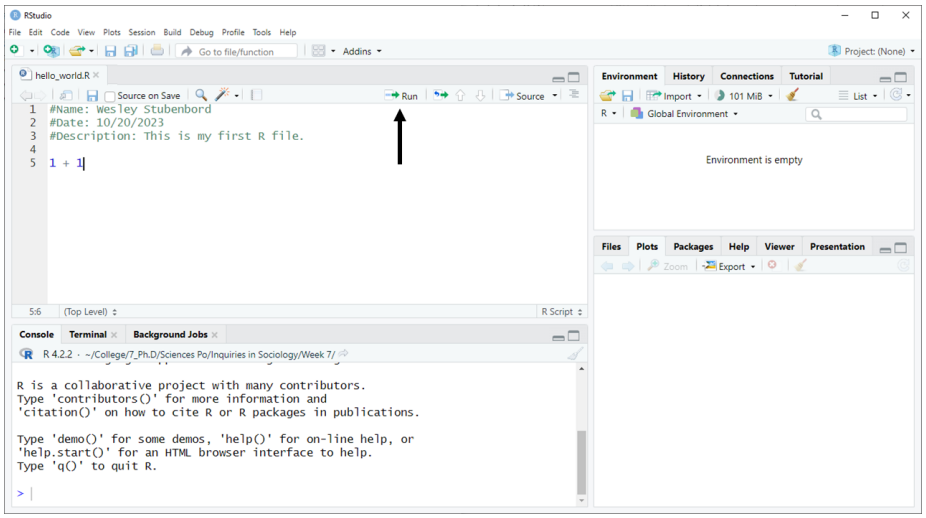
Now, save your R Script somewhere on your computer. Go to File > Save As, then choose a safe place on your computer to store it (I recommend creating a folder for this course), give your file a name, and press save. I called mine "hello_world".

1.7 Interacting in an R Script

Interacting in an R Script is slightly different from interacting with the console. Now when you type in code and hit `Enter`, it will not execute the code, it just creates a new line in your file.

To run code in a script in RStudio, you can either:

1. Select the lines you wish to run with your cursor and then press `Ctrl` + `Enter`
2. Or, put your cursor on the line you wish to run and click the `Run` button in the upper-right of the R Script pane



The first option allows you to run multiple lines at a time. The second runs only the line you are currently on. The results of your code will appear in the console pane below your R Script file when run successfully.

After you finish modifying your R Script file, you can save it and close out of RStudio. The next time you wish to access your saved code, you can open your R Script file and it will be exactly as you left it.

1.8 Summary

Let's briefly recap what you have learned in this lesson. So far you've learned:

- The difference between `R` and RStudio
- How to interact with the console
- How to create and store values in variables using an assignment operator
- What a vector is and how to create one
- How to use basic functions like `sum()` and `mean()` to perform calculations
- How to make comments using the `#` symbol
- How to create and save R Script files

Chapter 2

Working with Data in R

Before we can get to the nitty-gritty of working with real data, we need to familiarize ourselves with a few more essential concepts.

2.1 Functions

Last class, we assigned a vector to a variable like this:

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

Where `my_vector` is an object and $1, 2, 3, 4, 5, 6$ is the set of values assigned to it. When you run this code in your console (or in a script file), your new variable and its assigned values are stored in short-term memory and appear in the Environment pane of RStudio.

When we assigned a single value to another variable, however, as in:

```
x <- 1
```

or,

```
first_name = 'Wesley'
```

we didn't use `c()`. So, what exactly is `c()`?

Like `sum()` or `mean()`, `c()` is a **function**. Functions play an important role in all programming languages. They are snippets of code, often hidden in the background, that allow us to accomplish specific tasks, like adding up all of

the numbers in a vector, taking the mean, or creating a vector. In R, `c()` is a function which *c*ombines values into a vector or list.

Functions give us the ability to recall previously written code to perform the same task over again. Why re-write code every time you need to use it, after all, when you could use the same code you used last time? Instead of copying and pasting code, we can put it in a function, save it somewhere, and call it when we need it.

2.1.1 Calling a Function

When we want to use a function, or 'call it' as we will sometimes say, we type in the name of the function, enclose **arguments** in a set of parentheses, and run the command. The general form looks something like this:

```
function([arg1], [arg2], ...)
```

2.1.2 Using Arguments in a Function

In some cases, you may just have one argument for a function, as when you want to use the `sum()` function to add the elements of a vector:

```
sum(my_vector)
```

```
## [1] 21
```

In other cases, you can have multiple arguments:

```
sum(my_vector, my_vector)
```

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

Arguments can be required or optional and the number of arguments and the order in which they are input depends on the specific function you are using and what you are trying to accomplish. The `sum()` function, for instance, returns the sum of all values given as arguments.

Arguments can also be used to specify options for a function. Take a look at the example below:

```
sum(my_vector, NA, my_vector)
```

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

Here we are using the `sum()` function to add `my_vector` twice, as in the previous example, but now with a missing value (`NA`). Because the sum of two vectors plus a missing value is unknown, we get an unknown value (`NA`) as the output.

If we want the `sum()` function to ignore the unknown value, we can provide it with an additional, named argument which tells it to ignore `NA`. We can specify this by adding `, na.rm = TRUE` to our function call. See what happens below:

```
sum(my_vector, NA, my_vector, na.rm = TRUE)
```

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

We're back to an answer of 42. The `sum()` function ignored the missing value, as we specified, and added the two vectors.

All functions have named arguments and an ordering to them. If you omit the name of an argument in your function call, the function processes them according to their default order. It is generally a good habit to specify argument names, as in the example below where the 'x' argument takes the object you are trying to sum, but it is not entirely necessary for simple functions.

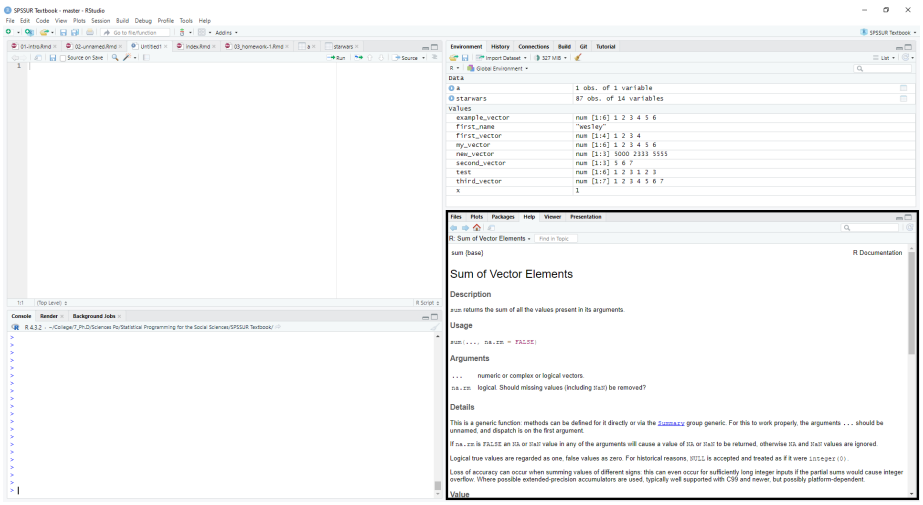
```
sum(x = my_vector)
```

```
## [1] 21
```

2.1.3 Getting Help with Functions

As you progress in `R`, you will learn many different functions and it can be difficult to keep track of all of the different arguments. Whenever you want to know more about what a function does or what arguments it takes, simply type `?function_name` into the RStudio console and you will get some useful documentation in the Help pane located in the lower-right of your RStudio window.

```
?sum
```



Check Your Understanding:

Let's take a quick pause to make sure we understand what we just learned.

1. Create a vector of three numbers and assign it to a variable called `first_vector`. Now use the `mean()` function to find the average of `first_vector`.
2. Using the `c()` function, create another vector called `second_vector` which includes the values of `first_vector` and an `NA` value. Try it on your own first, then click this footnote to see the answer.[1]
3. Using the `na.rm = TRUE` argument, calculate the mean of `second_vector`.

2.2 Packages

One of the great benefits of `R` is its power and flexibility. We've seen how functions provide us with the ability to reuse code, but functions are common to any programming language or statistical software.

It may sound cliché, but what makes `R` special is its community. `R` is a free and open-source software, which means that anyone can use or contribute to it. If you develop a new statistical method, for instance, you can write the code necessary to implement it and share it with others.

Base `R`, which you installed last class, comes with a number of built-in functions like `mean()`, `sum()`, `range()`, and `var()` . But, `R` users working out of the goodness of their hearts have developed many other functions that accomplish

[1] `second_vector <- c(first_vector, NA)`

an array of tasks, from making aesthetically-pleasing visualizations to executing complex machine learning algorithms.

These functions are put together into what are called **packages**, which can be easily installed and loaded into `R`. Packages can also contain data and other compiled code.

2.2.1 Installing Packages

We're going to use the `install.packages()` function to install one such package, called **tidyverse**.

```
install.packages('tidyverse')
```

Once you've run this command in your RStudio console, you will have downloaded the tidyverse and saved it to your **library**. The library is simply where your packages are stored.

Tidyverse is actually a set of packages, including `dplyr` and `ggplot2`, all of which are useful for data analysis in `R`. We'll be using the tidyverse throughout this course and you will find that it's the most commonly used set of packages for data analysis in `R`.

2.2.2 Loading Libraries

Whenever you start an `R` session and want to use a package, you have to be sure to load it. Loading a package makes sure that your computer knows what functions and data are inside, so that you can call them at will.

To load an `R` package, you can use the `library()` function, like this:

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

Now, that you've loaded tidyverse, you can access it's special functions like `mutate()` or its data sets, like `starwars`. Try entering `starwars` in your console after you've loaded the tidyverse. What's inside?

2.3 Loading Data

Great, you know what a function is, you have the tidyverse installed, and you've seen that data can be contained in packages, which are easy to install and load.

2.3.1 Using Data from Packages

Let's install and load another package, so that we can take a look at some more data.

```
install.packages('socviz')
```

```
library(socviz)
```

The `socviz` package accompanies a textbook called ***Data Visualization*** written by Kieran Healy, a Professor of Sociology at Duke University, and it contains some interesting datasets including election data from the 2016 U.S. presidential election. This dataset is stored in an object titled `election`. Once you have `socviz` installed and loaded, you can get a preview of its contents by entering the name of the object:

```
election
```

```
## # A tibble: 51 x 22
##    state     st     fips total_vote vote_margin winner party pct_margin r_points
##    <chr>     <chr> <dbl>      <dbl>       <dbl> <chr>  <chr>      <dbl>    <dbl>
##  1 Alabama   AL        1    2123372      588708 Trump  Repu~     0.277     27.7
##  2 Alaska    AK        2     318608       46933 Trump  Repu~     0.147     14.7
##  3 Arizona   AZ        4    2604657       91234 Trump  Repu~     0.035      3.5
##  4 Arkansas  AR        5    1130635      304378 Trump  Repu~     0.269     26.9
##  5 Californ~ CA        6   14237893     4269978 Clint~ Demo~     0.300    -30.0
##  6 Colorado  CO        8    2780247      136386 Clint~ Demo~     0.0491    -4.91
##  7 Connecti~ CT        9    1644920      224357 Clint~ Demo~     0.136    -13.6
##  8 Delaware  DE       10     443814       50476 Clint~ Demo~     0.114    -11.4
##  9 District~ DC       11     311268      270107 Clint~ Demo~     0.868    -86.8
## 10 Florida   FL       12    9502747      112911 Trump  Repu~     0.0119     1.19
## # i 41 more rows
## # i 13 more variables: d_points <dbl>, pct_clinton <dbl>, pct_trump <dbl>,
## #   pct_johnson <dbl>, pct_other <dbl>, clinton_vote <dbl>, trump_vote <dbl>,
## #   johnson_vote <dbl>, other_vote <dbl>, ev_dem <dbl>, ev_rep <dbl>,
## #   ev_oth <dbl>, census <chr>
```

For ease of use, we're going to store a copy of this data in a new object in our environment called `election_2016`.

```
election_2016 <- election
```

Now, we can play around with it. In addition to getting a preview of the data by entering the name of our object in the console, we can also access it through the Environment pane of our RStudio window. Click on `election_2016` and you will see the full dataset.

Just like in a spreadsheet, you can scroll through the full set of columns and rows. Remember, of course, that you cannot edit values in this view tab. This is by design. If we want to make changes to the data or perform calculations, we need to do so programmatically.

2.4 Data Types and Data Structures

This seems about as good a point as any to talk about the different types of data you will be working with in R.

2.4.1 Data Types

There are six different basic data types in `R`. The most important for our purposes are:

- **character**: letters such as 'a' or sets of letters such as 'apple'
- **numeric**: numbers such as 1, 1.1 or 23
- **logical**: the boolean values, TRUE and FALSE

The other types are integers (which can only hold integers and take the form `1L`), complex (as in complex numbers with an imaginary component, 1+2i), and raw (raw data in the form of bytes). You have already used the previous three and we will never use the latter three.

If you wish to check the data type of an object, you can use the `class()` function.

```
class(my_vector)
```

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

2.4.2 Data Structures

There are many different data structures in `R`. You've already become familiar with one, vectors, a set of values of the same type. Other types of data structures include:

- **list**: a set of values of different types
- **factor:** an ordered set of values, often used to define categories in categorical variables
- **data frame**: a two-dimensional table consisting of rows and columns similar to a spreadsheet
- **tibble:** a special version of a data frame from the *tidyverse*, intended to keep your data nice and tidy

Note that data structures are usually subsettable, which means that you can access elements of them. Observe:

```
my_list <- c('a', 'b', 'c', 2)
my_list[2]
```

```
## [1] "b"
```

In the example above, we've called an element of the list, `my_list`, using an index number in a set of brackets. Since we entered the index, `2`, inside brackets next to our list name, we received the second element of the list, the character `b`. We can also modify elements of a list in the same way.

Let's say that I now want to change 'b', the second element of `my_list`, to the word 'blueberry':

```
my_list[2] <- 'blueberry'
my_list
```

```
## [1] "a"         "blueberry" "c"         "2"
```

Easy enough. Now try it out yourself:

1. Create a vector with three elements: "sociology", "economics", and "psychology"
2. Call each of them individually.
3. Change the value of the second element to the value of the first element.
4. Change the value of the third element to the value of the first element.

Be sure to do the last two programmatically rather than by re-typing the initial values.

2.5 Using Functions with Data

Back to the elections data. We have our 2016 U.S. Presidential Election data stored in a **tibble** called `election_2016`.

If I want to output a single column from the data, I can do so by typing in the name of the data followed by the `$` symbol, also known as the **subset operator**, and the name of the column.

```
election_2016$state
```

```
##  [1] "Alabama"              "Alaska"               "Arizona"
##  [4] "Arkansas"             "California"           "Colorado"
##  [7] "Connecticut"          "Delaware"             "District of Columbia"
## [10] "Florida"              "Georgia"              "Hawaii"
## [13] "Idaho"                "Illinois"             "Indiana"
## [16] "Iowa"                 "Kansas"               "Kentucky"
## [19] "Louisiana"            "Maine"                "Maryland"
## [22] "Massachusetts"        "Michigan"             "Minnesota"
## [25] "Mississippi"          "Missouri"             "Montana"
## [28] "Nebraska"             "Nevada"               "New Hampshire"
## [31] "New Jersey"           "New Mexico"           "New York"
## [34] "North Carolina"       "North Dakota"         "Ohio"
## [37] "Oklahoma"             "Oregon"               "Pennsylvania"
## [40] "Rhode Island"         "South Carolina"       "South Dakota"
## [43] "Tennessee"            "Texas"                "Utah"
## [46] "Vermont"              "Virginia"             "Washington"
## [49] "West Virginia"        "Wisconsin"            "Wyoming"
```

If I want to perform a calculation on a column, I can pass the column as an argument to a function like so:

```
# Sum the total number of votes cast in the 2016 Presidential election.
sum(election_2016$total_vote, na.rm = TRUE)
```

```
## [1] 137125484
```

The `na.rm` argument isn't strictly necessary in this case (since there are no missing or unknown values), but it's good to remember that it's there when you need it.

For the remainder of today's session, I'd like you play around with this data. In particular:

1. Identify the variable type for the `ST`, `pct_johnson`, and `winner` columns.

2. Calculate the mean `vote_margin` across the states.
3. Use the `table()` function to count the number of states won by each presidential candidate.
4. Create a variable which contains the total number of votes received by Hillary Clinton (contained in the column `clinton_vote`) and a variable containing the total number of votes received by Donald Trump (`trump_vote`). Take the difference of the two.
5. Create a variable containing the total number of electoral votes received by Hillary Clinton (contained in `ev_dem`) and another containing the total number received by Donald Trump (`ev_rep`). Take the difference of the two.
6. Try using the `plot(x=, y=)` function to plot a couple of numeric columns.

Homework 1

Due Date: Tuesday, 13 February by 23:59:59

Submission Instructions: Submit your completed R script file to Moodle.

This homework will be relatively short and straight-forward. The goal is to ease you into `R` now so that you are ready to complete some of the more complex data analysis that will take place later.

1. Create an `R` script and save it with an appropriate name. Add a header to your R script file in the format below.

```
# Name: [first_name] [last_name]
# Date: [date]
# Description: [brief description of the file]

# Question 2:
```

2. In your `R` script file, load the `tidyverse` package.
3. Create a vector with the following set of numbers: $30, 60, 90, 120, 150$.
    a. Multiply the vector by 2. In a brief comment, tell me what the result was.
    b. Take the vector and divide it by 3. Tell me what the result was in a brief comment.
    c. Multiply the vector by itself. Tell me what the result was in a brief comment.
    d. Return the third element of the vector.
    e. Replace the second element of the vector with a missing value (`NA`).
    f. Sum the vector, excluding the missing value. In a comment, write the answer.
4. Using the `socviz` package (see section 2.3 of the course textbook), load the `election` dataset into a new object called `elec`.

a. Find the total popular vote received by Gary Johnson using the `johnson_vote` variable.

b. Find the total popular vote received by 'Other' candidates using the `other_vote` variable.

c. In a comment answer the following question: who received more votes, Gary Johnson or "other" candidates? By how much?

d. Use the `sum()` function on the `state` variable. In a brief comment, explain why this didn't work and what the error message is telling you.

Chapter 3

Summarizing Data with `dplyr`

In the previous chapter, you learned how to load data from a package, how to access a column from a tibble using the subset operator `$`, and how to use basic functions to answer questions like: what was the total number of votes cast in the 2016 U.S. presidential election?

We've had a couple strokes of luck so far, however. Namely, our data has been nice and tidy and our questions haven't required us to poke around in our data to get the answers we are interested in. This brings us to *data wrangling* - the art and science of manipulating, distilling, or cajoling data into a format that allows you to find the answers you are seeking.[1]

For this lesson, we are going to maintain the illusion of neat and tidy data and focus on learning the tools necessary to dig into a data set: in particular, `dplyr` and the **pipe operator**. In the next lesson, our luck will unfortunately run out and we will be confronted with the harsh reality of unseemly data.[2]

3.1 Basic Description with Base `R`

Let's use an example to get us started. Last class, you toyed around with the 2016 U.S. presidential election data from the `socviz` package, a helpful

[1] Data wrangling, if it can be defined, is as an expansive topic. We'll focus on summarizing data today.

[2] Sadly, almost all data you encounter out in the wild will be very unseemly for one reason or another. But, maybe after taking this course and ascending the ranks of government/business/academia, you too will become an evangelical for orderly data and help bring peace to a world of mismanaged data.

collection of data sets and other goodies assembled by Kieran Healy.[3]

We'll use another data set from the same package in a moment, but, for now, let's start again with the `election` data. We're also going to re-load our new best friend, the *tidyverse*.

```
library(socviz)
library(tidyverse)
```

Libraries loaded. Remember, once you have the packages installed, you don't have to do it again. So no need to include `install.packages()` in your script going forward.[4]

We'll load the data into an object in our environment. This time, I'm going to use a shorter name for the tibble to spare myself future pain. Longer names mean more to retype later.

```
elec_2016 <- election
```

Just like last time, we can do basic calculations on columns. We can even throw in a few new functions for good measure:

```
# table() gives a contingency table for character variables.
# Here it's giving the number of states (plus D.C.) won by each candidate.
table(elec_2016$winner)
```

```
##
## Clinton   Trump
##      21      30
```

```
# Wrapping prop.table() around a contingency table gives relative frequencies.
# i.e., Hillary Clinton won 41.2% of states (plus Washington D.C.) or 21/51.
prop.table(table(elec_2016$winner))
```

```
##
##   Clinton     Trump
## 0.4117647 0.5882353
```

[3] The `socviz` package serves as an accompaniment to Healy's textbook, *Data Visualization*, which is highly recommended.

[4] Anytime you do install packages, do it directly in the console. If someone needs to run your code, they should see the `library()` calls in the beginning of your code (after your header) and will know whether they need to install additional packages or not.

Alternatively, instead of using the `library()` function, you can always use the `require()` function, which has the benefit of both loading packages if you do have them and installing them if you don't.

```
# summary() gives us a nice 5-number summary for numeric variables.
# Here we see the min, max, median, mean, and quartiles for the pop. vote margin.
summary(elec_2016$vote_margin)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     2736   96382  212030  383997  522207 4269978
```

But, what if I want to do something more specific?

What if, for instance, I really want to know how much of the popular vote third party Libertarian candidate Gary Johnson won across the different regions of the United States? Here, we need special functions from dplyr and the pipe operator.

```
# An illustrative example - no need to try this just yet
elec_2016 %>%
  group_by(census) %>%
  summarize(total = sum(johnson_vote))
```

```
## # A tibble: 4 x 2
##   census      total
##   <chr>       <dbl>
## 1 Midwest   1203062
## 2 Northeast  676192
## 3 South     1370056
## 4 West      1239925
```

3.2 The Pipe Operator

The **pipe operator** is a very handy tool indeed. It is a specialized operator that comes from the `maggritr` package, which itself is contained in the tidyverse.

It looks like this: `%>%`. But, it can also look like this: `|>`. There isn't much of a difference between the two, so you can use whichever you prefer as long as you are consistent.

The pipe operator has a straightforward function: it allows you to combine a series of steps into a single line of code. And, it does this in a way that makes your code very legible. Whenever you see the pipe operator, you should read it as though it is saying, "And then [do this]."

So in the previous example I gave you might read it as:

```
elec_2016 %>%                              # Take the election data AND THEN
  group_by(census) %>%                     # group it by census region AND THEN
  summarize(total = sum(johnson_vote))     # sum up the Johnson vote.
```

Note a couple of things here:

1. The pipe operator always goes at the end of each line, followed by a new line
2. The pipe operator never goes at the end of the command

The first is a convention to make code more readable and the second is a requirement. If you leave a pipe operator at the end of your statement, `R` will search for the missing code and then give you an unhappy error when you try to run more code. Don't leave a pipe operator hanging.

3.3 Functions from `dplyr`

`dplyr` (pronounced dee-ply-R) is a handy set of tools for working with tabular data. It's one of the packages in tidyverse (along with `ggplot2`, `tidyr`, `tibble`, `readr`, and a few others), so you don't have to load it separately.

`dplyr` has a handful of special functions:

- `group_by()`, which groups data together at the level we desire (such as states by census region in our example).
- `filter()`, which gives us the rows corresponding to some criteria entered as an argument
- `select()`, which selects columns
- `summarize()`, which performs calculations
- `mutate()`, which creates new columns (e.g., variables)

3.4 Glimpsing GSS Data

Let's load another data set from `socviz`. This one is called `gss_sm` and contains a nice, clean extract from the 2016 General Social Survey.

```
gss <- gss_sm
```

The General Social Survey is a nationally representative biennial survey of U.S. adults on sociological topics produced out of the National Opinion Research Center (NORC) at the University of Chicago since 1972.

Take a quick look at the data. You can use `glimpse()`, another `dplyr` function, to get a good look at it and you can look at it visually using `view()`. Typing in `?gss_sm` (the original name of the data set from the package) will tell you what the variables are.

```
glimpse(gss)
```

As you might notice, there's a wealth of data in here. You might also notice that the data is at the *individual-level*. In this data, each row represents a individual respondent and each column is a variable (or their response to a question).

3.5 Selecting Columns

Maybe I want to narrow in and look at just a few variables. I can use the `select()` function to do this.

```
gss %>%
  select(id, sex, religion)
```

```
## # A tibble: 2,867 x 3
##       id sex    religion
##    <dbl> <fct>  <fct>
##  1     1 Male   None
##  2     2 Male   None
##  3     3 Male   Catholic
##  4     4 Female Catholic
##  5     5 Female None
##  6     6 Female None
##  7     7 Male   None
##  8     8 Female Catholic
##  9     9 Male   Protestant
## 10    10 Male   None
## # i 2,857 more rows
```

It returned a tibble with just the three variables I wanted to look at. I can always save a copy of this output if I want to, by storing it in a new object:

```
gender_rel <- gss %>%
  select(id, sex, religion)
```

And, if I decided I don't want this copy, I can always get rid of it using the `rm()` function.

```
rm(gender_rel)
```

3.6 Grouping and Summarizing

Let's say we want to summarize our respondents by religious affiliation.

To do this, we first have to tell the computer how we are going to group the data. Grouping doesn't change the data, but it prepares `R` to interpret it according to our groups. We're going to group by the `religion` variable.

Next, we have to tell the computer how to `summarize()` the groups. We're going to count the respondents using `n()`. `n()` just counts the rows within each group.

```
gss %>%
  group_by(religion) %>%      # Group by religion
  summarize(total = n())      # Create a total by counting the rows
```

```
## # A tibble: 6 x 2
##   religion   total
##   <fct>      <int>
## 1 Protestant  1371
## 2 Catholic     649
## 3 Jewish        51
## 4 None         619
## 5 Other        159
## 6 <NA>          18
```

As you can see, `summarize()` needs you to provide a new column name and a measurement. In the example above, we gave the column the name `total` and asked it count the total number of respondents by group using the `n()` function.

Again, if we wanted to, we could save a copy of this summary in a new table as in the command below. Our original table will always be untouched unless we save over it (e.g., `gss <- gss %>% ...`).

```
relig <- gss %>%
  group_by(religion) %>%
  summarize(total = n())
```

We can also group by two columns, such that we could find religious affiliation by sex, for example:

```
gss %>%
  group_by(religion, sex) %>%
  summarize(total = n())
```

```
## `summarise()` has grouped output by 'religion'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 12 x 3
## # Groups:   religion [6]
##    religion   sex    total
##    <fct>      <fct>  <int>
##  1 Protestant Male     559
##  2 Protestant Female   812
##  3 Catholic   Male     287
##  4 Catholic   Female   362
##  5 Jewish     Male      22
##  6 Jewish     Female    29
##  7 None       Male     339
##  8 None       Female   280
##  9 Other      Male      58
## 10 Other      Female   101
## 11 <NA>       Male      11
## 12 <NA>       Female     7
```

The ordering of the groups matters. Because religion came first in our `group()` function, our results will show us the number of protestants who are male and the number of protestants who are female. For a count, this is the same thing as the reverse (e.g., the number of protestants who are male is the number of males who are protestant), but for frequencies this is not the case and the order does matter.

3.7 Calculating with `mutate()`

Let's add a frequency column and a percentage column so that we can get a sense of whether there the gender composition within religious groups.

```
gss %>%
  group_by(religion, sex) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
## `summarise()` has grouped output by 'religion'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 12 x 5
## # Groups:   religion [6]
##    religion   sex    total  freq   pct
##    <fct>      <fct>  <int> <dbl> <dbl>
##  1 Protestant Male     559 0.408  40.8
##  2 Protestant Female   812 0.592  59.2
##  3 Catholic   Male     287 0.442  44.2
##  4 Catholic   Female   362 0.558  55.8
##  5 Jewish     Male      22 0.431  43.1
##  6 Jewish     Female    29 0.569  56.9
##  7 None       Male     339 0.548  54.8
##  8 None       Female   280 0.452  45.2
##  9 Other      Male      58 0.365  36.5
## 10 Other      Female   101 0.635  63.5
## 11 <NA>       Male      11 0.611  61.1
## 12 <NA>       Female     7 0.389  38.9
```

Notice, we used the same code as before, but we added a `mutate()` function which created two new columns `freq` and `pct`. We also told the `mutate()` function how to calculate the columns. `freq`, we said, should be the respondents in each group divided by the sum of the groups (or `freq = total / sum(total)`) and `pct` should be the frequency multiplied by 100 and rounded to the first decimal place, using the `round()` helper function. In doing so, we find that among Protestant respondents, 40.8% are male and 59.2% are female.

Calculating relative frequencies can be a bit of a beast, but the general form is always the same.

3.8 Filtering

What if we only wish to see the Protestant results of our last query? We can add a `filter()` function at the end.

```
gss %>%
  group_by(religion, sex) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1)) %>%
  filter(religion == "Protestant")
```

```
## `summarise()` has grouped output by 'religion'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 2 x 5
## # Groups:   religion [1]
##   religion   sex    total  freq   pct
##   <fct>      <fct>  <int> <dbl> <dbl>
## 1 Protestant Male     559 0.408  40.8
## 2 Protestant Female   812 0.592  59.2
```

Usually, you'll want to use a `filter()` function at the beginning of your query. It can save some code later on.

Here's another example of `filter()`. This time, I'm only interested in religious affiliation among holders of graduate degrees.

```
gss %>%
  filter(degree == 'Graduate') %>%
  group_by(religion) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
## # A tibble: 6 x 4
##   religion   total    freq   pct
##   <fct>      <int>   <dbl> <dbl>
## 1 Protestant   126 0.396    39.6
## 2 Catholic      63 0.198    19.8
## 3 Jewish        15 0.0472    4.7
## 4 None          82 0.258    25.8
## 5 Other         31 0.0975    9.7
## 6 <NA>           1 0.00314   0.3
```

And, so here I see that 39.6% of graduate-degree holding respondents were Protestant and 25.8% had no religious affiliation. Later on, we'll learn how to turn this sort of thing into a graph.

3.9 All Together Now

What if I want to do something like find all survey respondents who are Protestant, voted for Obama in the 2012 U.S. Presidential election, and have children? And I'd like to know their relative frequency across regions of the U.S.?

```
gss %>%
  filter(religion == "Protestant") %>%
  filter(obama == 1) %>%
  filter(childs > 0) %>%
  group_by(region) %>%
  summarize(total = n()) %>%
  mutate(freq = round(total / sum(total),4),
         pct = round((freq*100), 1))
```

```
## # A tibble: 9 x 4
##   region          total   freq   pct
##   <fct>           <int>  <dbl> <dbl>
## 1 New England        15 0.0409   4.1
## 2 Middle Atlantic    27 0.0736   7.4
## 3 E. Nor. Central    79 0.215   21.5
## 4 W. Nor. Central    19 0.0518   5.2
## 5 South Atlantic    107 0.292   29.2
## 6 E. Sou. Central    31 0.0845   8.5
## 7 W. Sou. Central    40 0.109   10.9
## 8 Mountain           24 0.0654   6.5
## 9 Pacific            25 0.0681   6.8
```

Now, I can rest easy knowing that I can find the percentage of 2012-Obama supporting Protestants with children who reside in the South Atlantic census region (29.2%).

3.10 Some Data Exploration

Use the remainder of class time today to explore the gss_sm dataset. Try summarizing a few different variables according to different groups