

Statistical Programming for the Social Sciences Using R

Wesley Stubenbord

2024-02-16

Contents

Introduction	5
Resources	5
1 An Introduction to R	7
1.1 Installing R	7
1.2 Installing RStudio	8
1.3 Using the Console	10
1.4 Calculations with Variables	12
1.5 Saving Your Work	13
1.6 Creating and Saving an R Script	14
1.7 Interacting in an R Script	15
1.8 Summary	16
2 Working with Data in R	17
2.1 Functions	17
2.2 Packages	20
2.3 Loading Data	22
2.4 Data Types and Data Structures	23
2.5 Using Functions with Data	25
Homework 1	27

3 Summarizing Data with dplyr	29
3.1 Basic Description with Base R	29
3.2 The Pipe Operator	31
3.3 Functions from dplyr	32
3.4 Glimpsing GSS Data	33
3.5 Selecting Columns	34
3.6 Grouping and Summarizing	35
3.7 Calculating with mutate()	38
3.8 How R Reads Functions	40
3.9 Filtering	40
3.10 Conditional Filtering	42
3.11 Fancy Tables with kable()	44
3.12 Another Example	44
3.13 Practice Exploring Data	45
Homework 2	47

Introduction

Welcome! This is the companion website for *Statistical Programming for the Social Sciences Using R*, taught at the Sciences Po Reims campus in the Spring 2024 term. The course is a broad introduction to the general programming skills required for data analysis in the social sciences.

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

Resources

There are a variety of R resources out there, many of which have been helpful in developing this course. If you would like to dig deeper into a topic or simply want to read other explanations of the concepts discussed in this course, here is a list of useful resources:

- ***R for Data Science, 2e*** by Hadley Wickham ([link](#)): A free introductory textbook on how to use the many features of R to make sense of data, written by the creator of the tidyverse package.
- ***Data Visualization*** by Kieran Healy ([link to companion site](#)): An introductory textbook on how to make practical and beautiful data visualizations in R. Healy has a distinctive and appealing visual style. He also happens to be an accomplished sociologist, known especially for his disdain of nuanced theory and his work related to markets and morals, and so his examples deal especially with social issues and phenomena.
- ***Introduction to Econometrics with R*** by Florian Oswald and colleagues at SciencesPo ([companion textbook and slides](#)): you may very well be taking this course, because you couldn't get into this other Sciences Po course. For this, I am both sorry, because you are missing out, and not sorry, because it keeps me gainfully employed and gives me an excuse to write this textbook. Fortunately for you, the more statistics-heavy course

developed by Professor Oswald and colleagues is freely-available online. If you'd like to see material which is more heavily-weighted towards applied statistical methods (especially applied economics) rather than general data analysis skills, take a look at their extremely well put together course and rue the day you woke up late for course sign-ups.

Chapter 1

An Introduction to R

If you are taking this course, you probably don't need an explanation of why R is useful or why it may be in your best interest to take this course.

So I won't beleaguer the point: yes, R will make you fabulously rich; yes, it will help you make new friends; and yes, it will allow you to escape your own mortality. Or, at the very least, it will allow you to do some interesting things with data, which is nearly as nice.

To get started with R, 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 known as an Integrated Development Environment (IDE)

The truth of the matter is that you don't necessarily need RStudio to program in R, but it certainly makes life easier. The difference between programming in R and programming in R using RStudio is akin to the difference between driving a Fiat Panda and driving a Porsche. Both will get you to the same place, but one is more likely to be an enjoyable experience. We will be using RStudio throughout the course as a result.

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” then “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 be sufficient.

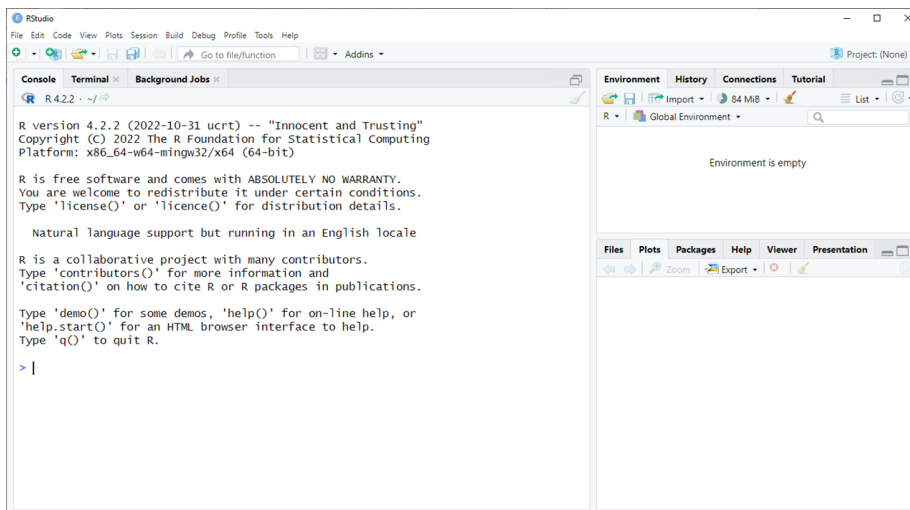
1.2 Installing RStudio

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

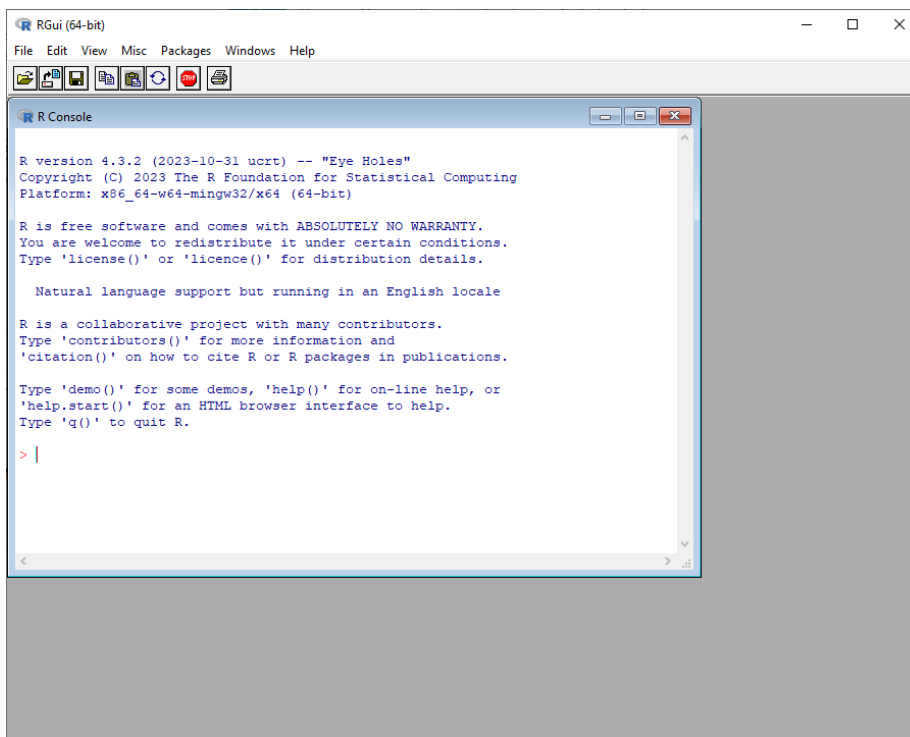
Posit (a company formerly known as RStudio) offers **RStudio Desktop** free of charge. Posit also offers a cloud-hosted version of the software (called Posit Cloud) which has both free and paid tiers. If you have trouble running RStudio Desktop on your computer, you may wish to consider using a Posit 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. Follow the on-screen steps to complete the set-up.

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

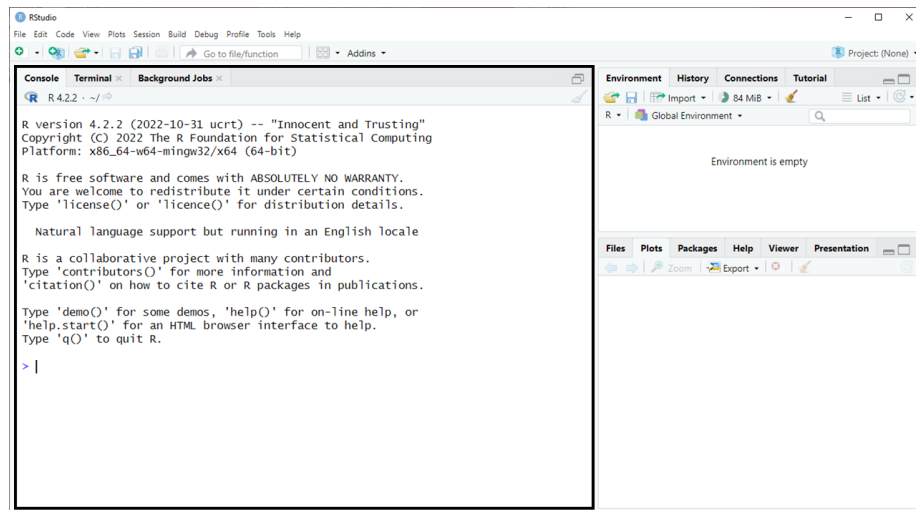


If your screen looks like the image below, it means that you've accidentally opened RGui, a basic graphical user interface included with R, and not RStudio. We're always going to be working with RStudio for this class, so close out of RGui and open RStudio instead.



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 with a black square 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* (or output) 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:

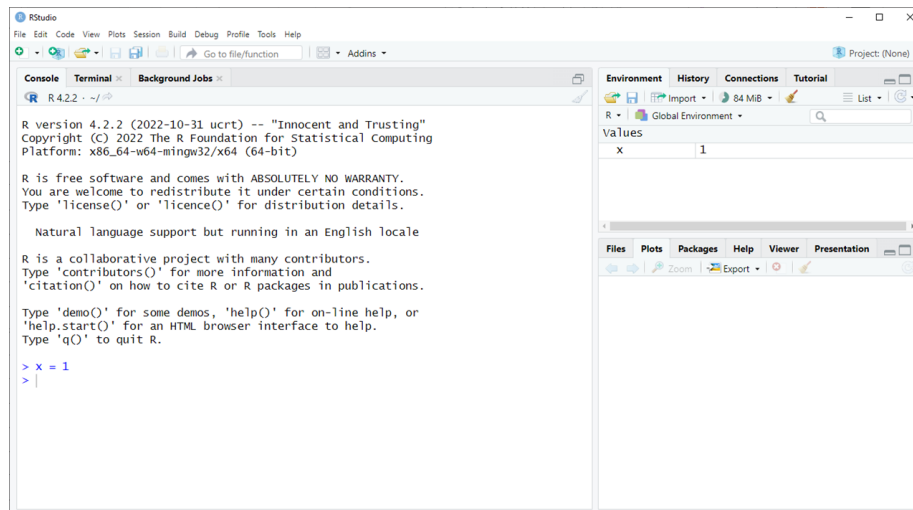
1. $10/3$
2. $(10/3) + 1$
3. $(10/3) + 1 + (5.111)\backslash^2$

As you can see, R is able to handle basic math operations with ease. What about other operations? Can you work with variables in R, for example?

Try typing this in the console:

```
x = 1
```

What happens when you press **Enter**?



You may have noticed that unlike before 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 is a type of object which holds a set of values of the same type. In this case, the object `x` contains a set of numbers: `{1, 2, 3, 4, 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?

1.5 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 anyone can come back to it and understand what you did. We’ll say more about how to do this later on.

There are a couple of different ways to save your code:

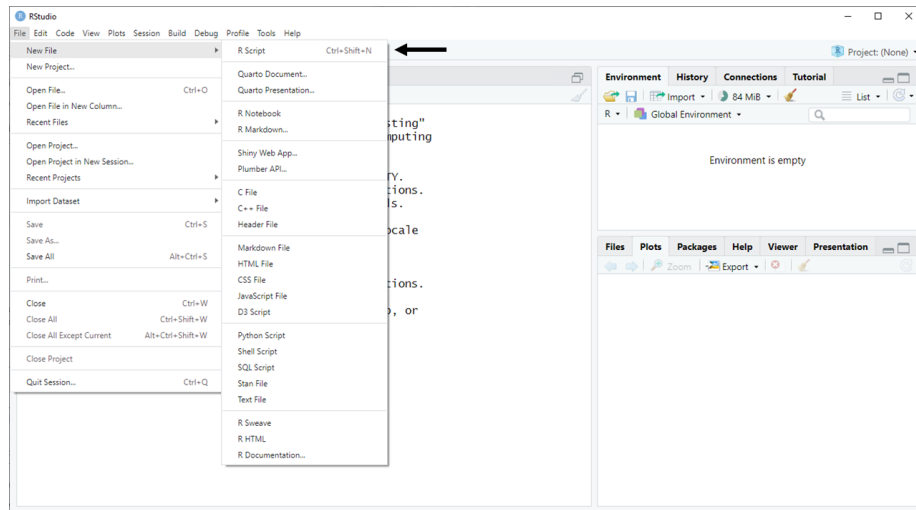
1. In an R Script, a simple text file ending in a `.r` extension
2. In an R Markdown file, an interactive format that allows you to see your code and the results together in the same file

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

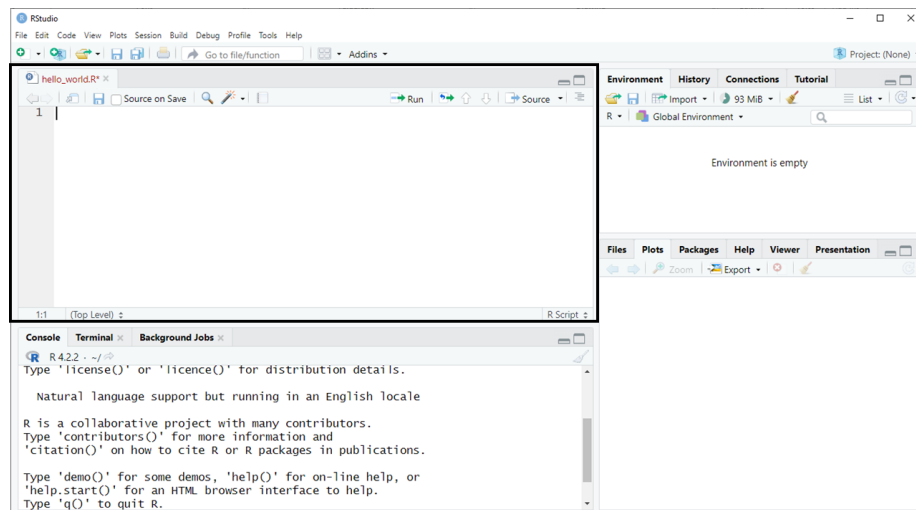
¹As you will have found, `mode()` doesn’t calculate a statistic here (although even if it had, you still wouldn’t have gotten a meaningful statistic for this particular vector). Some functions are easy to guess, like `median()`, but others can be false cognates just like in a spoken language. We’ll talk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

1.6 Creating and Saving an R Script

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

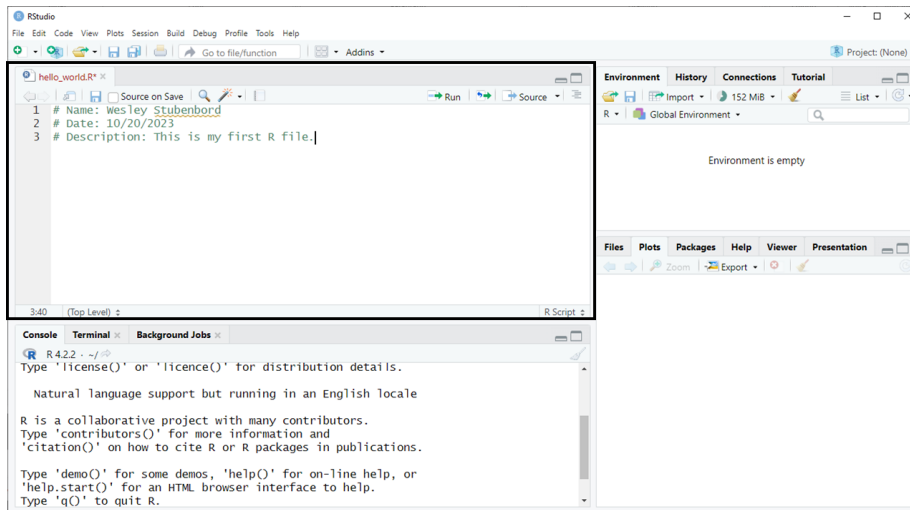


You should now have a window open in RStudio which looks like this:



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.



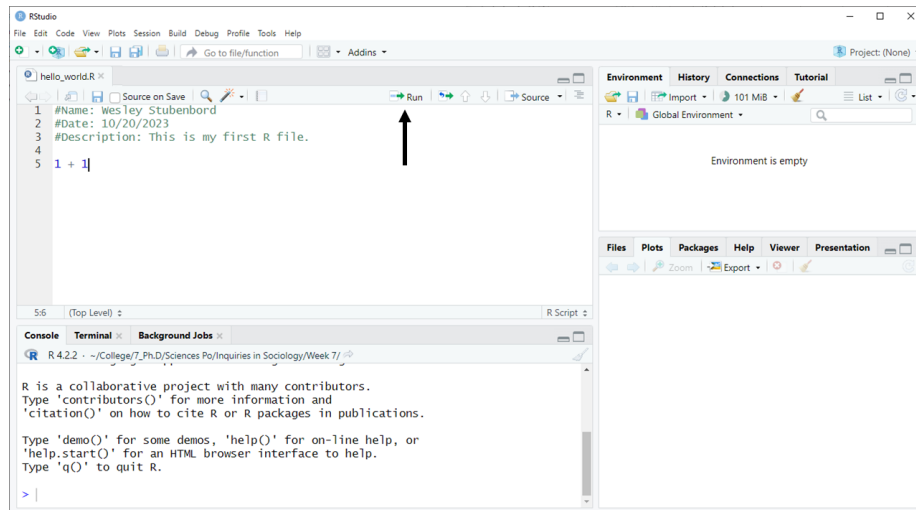
Now, save your R Script somewhere on your computer. Go to File > Save As, then choose a safe place on your computer to store it (I recommend creating a folder for this course), give your file a name, and press save. I called mine “hello_world”.

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 your code will be exactly as you left it.

1.8 Summary

Let's briefly recap what you 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 objects 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 combines 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.

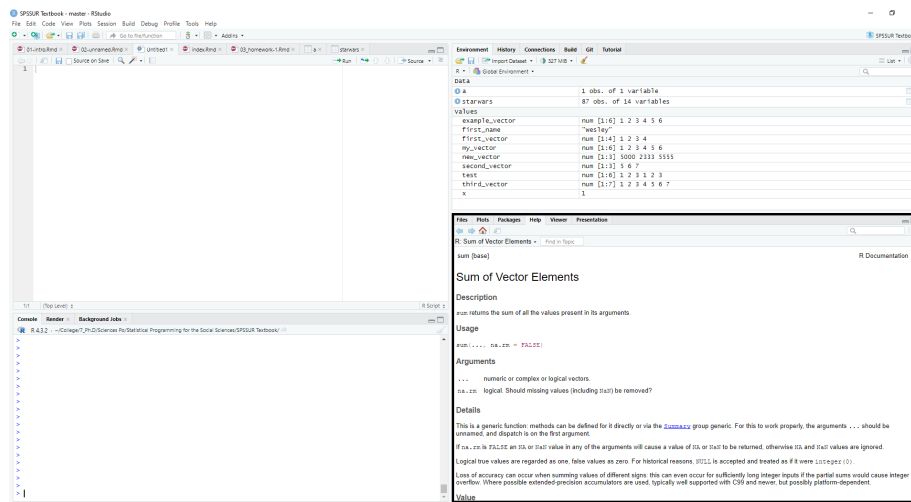
```
sum(x = my_vector)
```

```
## [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 its 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. Our data has been nice and tidy and our questions haven’t really required us to poke around in our data to find 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.

For this lesson, we are going to continue to maintain the illusion of neat and tidy data and focus on learning the tools necessary to dig deeper into a data set: in particular, **dplyr** and the **pipe operator**. In the next lesson, our luck will run out and we will be confronted with the harsh reality of unseemly data.¹

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 collection of data sets and other goodies developed by Kieran Healy.²

¹Sadly, 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 to make the world a tidier place.

²The **socviz** package serves as an accompaniment to Healy’s textbook, *Data Visualization*, which is highly recommended.

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

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

Libraries loaded. Remember, once you have the packages installed, you don't need to do it again. So, don't include `install.packages()` in your scripts going forward.³

We'll load the data into an object in our environment. This time, we'll use a slightly shorter name for the tibble to spare ourselves some future misery. Longer names mean more to retype later.

```
elec_2016 <- election
```

Just like last time, we can do basic calculations on columns. We can use the same principles to try out a few new functions:

```
# table() gives a contingency table for character variables.
# Here's 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% (21/51) of states (plus Washington D.C.).
prop.table(table(elec_2016$winner))
```

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

³Anytime you 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. RStudio also has a helpful auto-prompt feature that will inform you of missing packages.

Instead of the `library()` function, you can also use the `require()` function, which has the benefit of both loading packages if you have them already installed 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 we want to do something more specific?

What if we 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 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
```

We'll learn how to create frequency tables like this and more in a moment.

3.2 The Pipe Operator



The **pipe operator** is a handy tool indeed. It is a specialized operator that comes from the `magrittr` 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.⁴

The pipe operator has a straightforward function: it allows you to combine a series of steps into a single command. And, it does this in a way that makes your code 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 provided, 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 the 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 unfriendly 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 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 some desired level (e.g., states by census region)
- **filter()**, which gives us the rows corresponding to the criteria entered as an argument
- **select()**, which selects columns from the original data

⁴If you are interested, see here for more on the differences: <https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe>

- `summarize()` or `summarise()`, which performs calculations⁵
- `mutate()`, which creates new columns (or variables)
- `arrange()`, which sorts the row order by column values

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 by 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 sense of what's inside and you can inspect it visually using `view()`. Typing in `?gss_sm` (the original name of the data set from the package) will tell you what variables the data contains.⁶

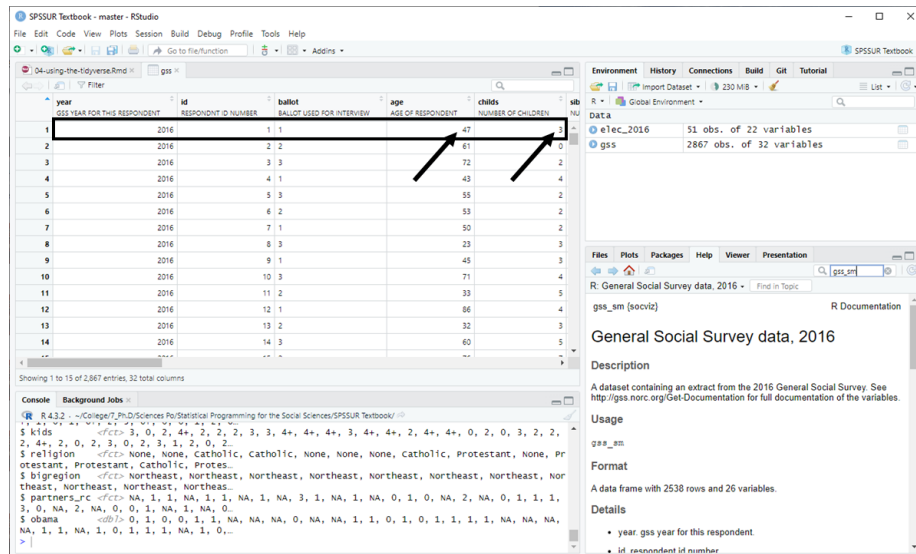
```
view(gss)
glimpse(gss)
```

There's a wealth of data in here. You may have also noticed that the data here is at the *individual-level*. Each row represents an individual respondent (identified by the `id` variable) and each column consists of a variable (in this case, a coded response to a survey question).

If we click on our data in the environment pane, we can see that the first data row corresponds to respondent #1 who is 47 years old and has 3 children:

⁵`summarize()` and `summarise()` are the same function, just two different spellings, the choice of which depends on who you've learned English from.

⁶You won't always be able to get documentation on a data set by using the help function, unfortunately. But, in this case, it works because `socviz` comes with documentation that was downloaded when you installed the package. Note that you must refer to the data in your help query by its original name (`?gss_sm` not `?gss`).



3.5 Selecting Columns

There are a lot of variables, 32 of them, in fact. Maybe we want to narrow in and look at just a few of them, like: `id`, `sex`, and `religion`. We can use the `select()` function to do this.

```
gss %>%                                     # Take the GSS data AND THEN
  select(id, sex, religion)                 # take just the ID, sex, and religion columns.
```

```
## # 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
```

In the code above, we told R that we wanted to take the GSS data and then only the `id`, `sex`, and `religion` variables. The `select` function output a new

tibble containing only those three variables that were entered as arguments. The number of rows or observations, 2,867, is the same as in the original data.

We can now save a copy of this new tibble by assigning it to a new object. Let's call this new object `gender_relig`.

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

Now we have a new object containing our new tibble. If you inspect this new tibble and then decide that you don't need or want it anymore, you can always get rid of it using the `rm()` function.⁷

```
view(gender_relig)
rm(gender_relig)
```

3.6 Grouping and Summarizing

Let's say we want to get a table which shows the number of respondents by religious affiliation. There are other ways of doing this, but we're going to use `dplyr` and the pipe operator.

To do this, we first have to tell `R` how we would like to group the data. Grouping doesn't visibly change the data, but it prepares `R` to interpret our next commands according to the groups we specify. We're going to group by the `religion` variable which contains the respondent's religious affiliation.

```
gss %>%
  group_by(religion)
```

```
## # A tibble: 2,867 x 32
## # Groups:   religion [6]
##   year  id ballot age childs sibs degree race sex region income16
##   <dbl> <dbl> <labelled> <dbl> <dbl> <labe> <fct> <fct> <fct> <fct> <fct>
## 1 2016   1 1      47     3 2    Bache~ White Male New E~ $170000~
## 2 2016   2 2      61     0 3    High ~ White Male New E~ $50000 ~
## 3 2016   3 3      72     2 3    Bache~ White Male New E~ $75000 ~
## 4 2016   4 1      43     4 3    High ~ White Fema~ New E~ $170000~
## 5 2016   5 3      55     2 2    Gradu~ White Fema~ New E~ $170000~
## 6 2016   6 2      53     2 2    Junio~ White Fema~ New E~ $60000 ~
## 7 2016   7 1      50     2 2    High ~ White Male New E~ $170000~
```

⁷Using the `rm()` function can help keep your environment a bit more orderly, but it isn't always necessary since your environment will be cleared out each time you close RStudio anyways.

```
## 8 2016      8 3          23      3 6      High ~ Other Fema~ Middl~ $30000 ~
## 9 2016      9 1          45      3 5      High ~ Black Male Middl~ $60000 ~
## 10 2016     10 3          71      4 1      Junio~ White Male Middl~ $60000 ~
## # i 2,857 more rows
## # i 21 more variables: relig <fct>, marital <fct>, padeg <fct>, madeg <fct>,
## #   partyid <fct>, polviews <fct>, happy <fct>, partners <fct>, grass <fct>,
## #   zodiac <fct>, pres12 <labelled>, wtssall <dbl>, income_rc <fct>,
## #   agegrp <fct>, ageq <fct>, siblings <fct>, kids <fct>, religion <fct>,
## #   bigregion <fct>, partners_rc <fct>, obama <dbl>
```

As you can see, our data doesn't appear to have changed in the output above. We still have 32 variables and 2,867 observations. But, we do actually get a helpful note at the top our output that says, **Groups: religion[6]**. Our observations have been successfully grouped according to the six religious affiliations in our data.

Next, we have to add another line to our pipe function which specifies how we want to `summarize()` the groups. We want it to count up our rows, so we'll use the `n()` function. The `n()` function just counts the number of rows in a data frame.⁸ We have to tell `summarize()` where we want to store these values, so we'll put them in a new variable called `total`.

```
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, we provided `summarize()` with a new column name, `total`, and a measurement, `n()`. Now, we have the total number of respondents for each group (religious affiliation). The pipe operator allowed us to combine the `group()` and `summarize()` functions together in sequence so that we got the analysis we wanted..

If we want, we can save a copy of our new tibble in another object, as in the command below. The original data object in our environment (i.e., `gss`) will

⁸There are other options for counting the number of rows, like the `count()` or `tally()` functions, but I won't use them here.

always be untouched unless we intentionally re-write it (i.e., `gss <- gss %>% ...`).

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

Another quick example. Let's say we want to see the count of our 2016 GSS respondents by sex:

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

```
## # A tibble: 2 x 2
##   sex      total
##   <fct>   <int>
## 1 Male     1276
## 2 Female   1591
```

In this example, we took the GSS data *and then* grouped it by `sex` *and then* summarized it by creating a `total` which contains a count of the number of rows.

In this case, because we have tidy data (more on this in future lessons), the number of rows corresponds to the number of respondents who took the 2016 GSS. We can see that 1,276 of our respondents were male and 1,591 were female.

3.6.1 Grouping by Two Variables

We can also create the equivalent of what is called a two-way contingency table by grouping with two variables at the same time. We can use this to 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
```

In the table above, we can now identify the number of protestants who are male and the number of protestants who are female.

3.6.2 The Order of `group()` arguments

It is worth noting that the ordering of groups as arguments in the `group()` function sometimes matters (i.e., `group_by(religion, sex)` as opposed to `group_by(sex, religion)`).

Because religion came first in our argument order, our results show us the number of protestants who are male and the number of protestants who are female. But we could have very easily shown the number of males who are protestant and the number of females who are protestant.

For a count, it does not matter. The number of protestants who are male is the same as the number of males who are protestant. But, when we start looking at relative frequencies and percentages, the order does matter. You'll get a sense for this in a moment.

3.7 Calculating with `mutate()`

Is there an equivalent proportion of males and females among protestants in the GSS? Let's add a relative frequency column to find out.

```
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 here, but now we've added another step, a `mutate()` function to create two new columns, `freq` (relative frequency) and `pct` (percentage).

We previously calculated the `total` or the number of observations for each sub-group (e.g., protestants who are males, protestants who are females, etc.). The `mutate()` function takes the `total` we calculated in the previous step and uses it to calculate first the relative frequency and then the percentage for each sub-group.

To calculate the relative frequency, we used `freq = total / sum(total)` or in plain English “create a new value called `freq` and then calculate this value by taking the number of observations for each sub-group (`total`) and then dividing it by the sum of the totals for all sub-groups (`sum(total)`).”

For the religious group protestant, we have two sub-groups, male and female, and so the frequency for males protestants is calculated as $559 / (559 + 812)$, which equals 0.408, or exactly what you see in the first row in the frequency column in our new tibble. Similarly, the frequency for female protestants would be $812 / (559 + 812)$ or 0.592 or what you see in the frequency column in the second row of our new tibble.

What about the percentage or `pct`? In the second argument of our `mutate()` function, we told R to take the `freq` we calculated in the previous step, multiply it by 100 (to make it a percentage), and then round it to the first decimal place using the `round()` function. 0.408, the relative frequency of male protestants, therefore becomes 40.8%.

As you can see, calculating relative frequencies and percentages using `dplyr` and the pipe function can be a bit of a beast. The good news is that the general form is always the same and so you'll be able to re-use the code often.

3.8 How R Reads Functions

In the previous examples, you may have noticed a bunch of *nested functions*, which is when a function is used as an argument inside another functions, e.g., `summarize(total = n())`. It's worth pausing for a moment to think about how R reads code, since you will be using these types of constructions quite often.

Functions are always read inside out, so a nested function will always evaluate the inner-most function first. Pipe operations, on the other hand, are always read from left-to-right or top-to-bottom (if you're breaking up your code using new lines, as you should be). The two commands below evaluate in the same way, but R reads them in a slightly different ordering.

```
# Inside-out evaluation
sum(c(1,2,3))           # A vector, {1,2,3} is created first AND THEN summed
```

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

```
# Left-to-right/top-to-bottom (sequential) evaluation
c(1,2,3) %>%           # A vector is created AND THEN
  sum()                # it is summed
```

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

3.9 Filtering

Back to the data. What if we only wanted to see the protestant results for our previous examples? We can use a `filter()` function.

```
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
```

In a filter function, you use logical and comparison operators (see the slides from Session 3 if you'd like a refresher) to define the criteria for your new tibble. In this case, we want only the observations for which the `religion` variable is equal to "Protestant".

R is case-sensitive and so if the values in your data are "protestant", for example, you won't see those results in the tibble output here.

Here's another example using `filter()`. Usually, you will want to use the `filter()` function at the beginning of your query. 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
```

Now, we 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 nice graph.

```
# What happens if I use a lower-case 'g' in 'Graduate' instead?
gss %>%
  filter(degree == 'graduate') %>%
  group_by(religion) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
## # A tibble: 0 x 4
## # i 4 variables: religion <fct>, total <int>, freq <dbl>, pct <dbl>
```

3.10 Conditional Filtering

What if we want to filter our respondents for multiple degree types? We want to see in our table of religious affiliation, for example, only people who have a bachelor's degree **or** a graduate degree.

For these types of queries, we can use other logical operators in our `filter()` criteria. Here, specifically, we'll use `|` which stands for 'or'.

```
gss %>%
  filter(degree == 'Graduate' | degree == 'Bachelor') %>%
  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   367  0.430    43
## 2 Catholic    193  0.226   22.6
## 3 Jewish       27  0.0316    3.2
## 4 None        204  0.239   23.9
## 5 Other        59  0.0691    6.9
## 6 <NA>         4  0.00468    0.5
```

Now our results include only college graduates and graduate degree holders. If we want to see them broken out separately after we have filtered, all we need to do is change `group_by(religion)` to `group_by(religion, degree)`.

What if we want to filter our observations for all individuals with less than a bachelor's degree? We can create a vector with our specific criteria and then use it in our filter argument. Look at this:

```
filter_criteria <- c('Lt High School', 'High School', 'Junior College')
```

```
gss %>%
  filter(degree %in% filter_criteria) %>%
  group_by(religion, degree) %>%
  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: 17 x 5
## # Groups:   religion [6]
##   religion degree      total   freq   pct
##   <fct>    <fct>    <int> <dbl> <dbl>
## 1 Protestant Lt High School   155 0.155  15.5
## 2 Protestant High School    742 0.740   74
## 3 Protestant Junior College  106 0.106  10.6
## 4 Catholic   Lt High School   100 0.220   22
## 5 Catholic   High School    322 0.708  70.8
## 6 Catholic   Junior College   33 0.0725  7.3
## 7 Jewish     Lt High School    1 0.0417  4.2
## 8 Jewish     High School     17 0.708  70.8
## 9 Jewish     Junior College    6 0.25    25
## 10 None      Lt High School    62 0.150   15
## 11 None      High School    298 0.722  72.2
## 12 None      Junior College   53 0.128  12.8
## 13 Other     Lt High School   10 0.1     10
## 14 Other     High School     73 0.73    73
## 15 Other     Junior College   17 0.17    17
## 16 <NA>      High School     9 0.9     90
## 17 <NA>      Junior College    1 0.1     10
```

We’ve first created a vector, called `filter_criteria`, with all of the degree-levels we want to include in our data (we’ve left out ‘Graduate’ and ‘Bachelor’). Then, we’ve changed the filter criteria to say, “Take all respondents who have a degree listed in our vector, `filter_criteria`.” In code, we write this as: `filter(degree %in% filter_criteria)`.

3.10.1 The `%in%` Operator

`%in%` is a special logical operator that checks to see whether the values you are specifying are contained in an object. If the value is contained in the object, your computer will return `TRUE` and if not, it will return `FALSE`. This is especially useful for `filter()` since `filter()` selects rows based on whether they meet a criteria (`TRUE`) or not (`FALSE`).

Here’s a simple example of how this operator works in general:

```
1 %in% c(1,2,3,4,5)
```

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

```
6 %in% c(1,2,3,4,5)
```

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

3.11 Fancy Tables with `kable()`

If we want to make a summary table look a little bit nicer, we can add the `knitr::kable()` function to the end of our query to produce a more polished looking table.

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

religion	total	freq	pct
Protestant	126	0.3962264	39.6
Catholic	63	0.1981132	19.8
Jewish	15	0.0471698	4.7
None	82	0.2578616	25.8
Other	31	0.0974843	9.7
NA	1	0.0031447	0.3

The `::` operator here tells R to pull the `kable()` function from the `knitr` package (which is located in the `tidyverse`). This is useful when there are multiple functions with the same name in different packages.

You can also add additional code to your `kable()` function to customize the look of your table (see here for examples).

3.12 Another Example

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

Here's a brief example:

```
gss %>%
  filter(religion == "Protestant" | religion == "Catholic") %>%
  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      33 0.0602    6
## 2 Middle Atlantic  57 0.104   10.4
## 3 E. Nor. Central 119 0.217   21.7
## 4 W. Nor. Central  36 0.0657    6.6
## 5 South Atlantic 121 0.221   22.1
## 6 E. Sou. Central  35 0.0639    6.4
## 7 W. Sou. Central  57 0.104   10.4
## 8 Mountain       35 0.0639    6.4
## 9 Pacific         55 0.100   10
```

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

3.13 Practice Exploring Data

You can see here that the `dplyr` functions provide an enormous amount of flexibility and power. R, like other programming languages, is also very sensitive to mistakes in syntax or spelling: a missing comma in a set of function arguments, a hanging pipe operator, a misspelled filter criteria, or an erroneous object name can all cause output errors. Check your code carefully, take a deep breath, and try again. You'll get the hang of it in no time.

Use the remainder of class time today to explore the `gss_sm` data. Try summarizing different variables according to different groupings. Try using other measures like `mean()` or `sd()` to summarize numeric variables (like the number of children).

If you are feeling overwhelmed at the moment - don't despair, we're going to continue practicing these skills throughout the rest of the course.

Homework 2

Due Date: Wednesday, 28 February by 23:59:59

Submission Instructions: Submit your completed R script file to the corresponding Moodle assignment. Your file should contain a header and the file name should be formatted according to the guidelines discussed in class and posted in the course slides.

Be sure to show your work. This means that your answer to each question should show the code you used to obtain the answer. You do not need to show other steps that may have been taken to get the answer (e.g., trial and error), only the code that provides the answer.

Question #1:

Part A:

Using the `socviz` package, load the GSS data into an object called `gss`. Use the `table()` function and subset operator to find the number of respondents by `agegrp`.

In a comment, identify the number of respondents between the ages of 35-45.

Part B:

Next, use the `dplyr` functions to output a tibble which summarizes the number of respondents by `agegrp`.

In a comment, identify the number of respondents between the ages of 45-55.

Part C:

Output another tibble which summarizes respondents by `agegrp` and whether they voted for `obama` in the 2012 U.S. presidential election. In this tibble, include both the count (call it `total`), relative frequency (`freq`), and percentage (`pct`). You do not have to round the percentages, but can if you want.

In a comment, identify the number of respondents between the ages of 18-35 who voted for Obama. In a separate comment, identify the number of respondents between the ages of 18-35 that have missing values for the `obama` variable.

Question #2:

Again, using the GSS data from the `socviz` package, create a tibble called `marit_happy` which summarizes the happiness of GSS respondents by marital status. Filter your data so that it shows only respondents who are “Married” or “Never Married”.

In a comment consisting of a few brief sentences, compare the reported happiness of U.S. adults who are married as opposed to those who have never been married.

Question #3:

Also using the same GSS data and `dplyr`, output a tibble with: (1) the number of respondents by `degree` (i.e., the respondent’s highest degree level) and (2) the mean number of children by `degree`. Hint: you may need to use an `na.rm` argument somewhere.

In a brief comment, identify the relationship between degree-level and number of children among U.S. adults.