Introduction to R

SOC 300: Social Research Methods

Daulton Selke

2025-09-03

Table of contents

Preface

Welcome to this introductory R tutorial for SOC 300!

Here you will find all the step-by-step instructions for completing our initial foray into R for quantitative analysis of social science data. We will begin by establishing some common ground in basic R operations and functionality. After we lay this foundation, we will progress through various data processing tasks—from importing and cleaning public data to visualizing and analyzing these data for consumption by interested stakeholders.

You will receive an R script file with the commands detailed here, so that you can easily run and manipulate them on your own device, but you will always be able to refer to this mini-textbook in the event that you would like to see everything in one place and refer to some more detailed documentation on various R operations.

Part I

Day 1: Getting Started

1 Background

Before we start exploring some of R's basic functionality, I'm going to set the stage a little on what R and R studio are, why we are using these tools in particular, and what we will need to know before we dig in.

1.1 What is R?

At it's core, R is a programming language. There's a lot to say about this from a computer science perspective, but, for our purposes, you can just think of R as a language with a very particular structure that's designed to tell our computers what to do.

There are all sorts of different programming languages out there, and they all offer certain benefits or cater to particular computing needs. Unlike some general purpose languages like C, C++, or Python, R is relatively specialized, and this is part of what makes it so useful for us. R is designed with statistical computing as a primary motivator, and now—roughly 30 years into its tenure—stands as one of the most widely adopted resources for statistical data science in the social sciences and beyond.

Though there can be a bit of a learning curve when getting used to R, we will focus on exactly the things that we need and build ourselves up slowly. Once you get used to it, R will allow you to perform incredibly complex statistical procedures with relative ease, and it can even help us with other related tasks like visualizing and presenting our analyses.

1.2 What is R Studio?

We are going to pair R with the software R Studio, which is what's known as an Integrated Development Environment (IDE). Programming languages can be leveraged in a number of different ways. You could run R commands entirely from a Windows command line or Mac terminal. But that would probably not be very ideal for us—not to mention sort of ethereal and frustrating for those of without any programming experience. IDEs provide user-friendly interfaces for working with programming languages, so that we can easily manage our code, quickly generate and view the output of our analyses, and generally keep track of what we are doing with R. There are lots of other IDEs out there, but R Studio is an ideal balance of ease and power, so it will serve as our IDE of choice.

The best way to think about R Studio's relationship to R is by framing it as an analogy with a desktop computer. R Studio is to R as a monitor is to a computer. R is the thing that's doing all the heavy lifting computationally, and R Studio is the thing that allows us to view and interact with R in a way that's simple and straightforward.

1.3 Acquiring R and R Studio

All of the CHASS computers (and likely most NC State computers) come with R and R Studio, so you do not need to download them, but you may find it convenient to work on R assignments using your own personal device, so I've provided some instructions below.

Note that you will want to install R first,

1.3.1 Downloading R

You can download R from the Comprehensive R Archive Network (CRAN).

When you click that link, you will arrive at CRAN's homepage. Navigate to the sidebar on the left, find 'Download' near the top, and then click 'CRAN'. This will take you to the 'mirrors' page. Mirrors are just different host locations for downloading the R installation files. This allows you to maximize download speed by choosing a nearby server, so scroll down to 'USA' and choose one of those (I usually opt for the Durham, NC mirror).

Unless you are very experienced with computers, you should download one of the options listed as 'pre-compiled binary distributions'. These will typically be the first options listed. Don't even worry about what that means if you're not familiar. Just choose the one that reflects your operating system (there are options for Windows, Mac, and Linux) That should download an R installer, and you can follow the directions to complete a default installation.

1.3.2 Downloading R Studio

R Studio is a little more straightforward to download. Just navigate to its homepage, scroll down a little, and you will find a big button that says 'Download R Studio for [your operating system]'. Go ahead and run the installer with the default settings.

2 R Fundamentals

In this first section, we are going to start from the ground up and start to familiarize ourselves with the way R works and what it expects from us. We will begin with the most basic building blocks of R data and work our way up to the data frame—the object that will be most relevant for us. While you won't generally need to build data frames from scratch within R for your own research, it's a good way to familiarize yourself with the structure of data in R. While we will start here with a rather simple data frame, all the principles you learn here will scale up as we start to work with much larger and more complex data frames.

> **Note**
>
> I'm writing this mini-book in a language called Quarto, which allows us to carry out a bunch of neat formatting tasks with little effort. One of the big perks is that Quarto can read and process R code, so I will show you all my R code here, and you will be able to directly view the output. If you want to follow along with your own script on your personal device, you should copy and paste the commands from this document and run them locally.

2.1 Basic Operations

First, we will start by exploring some of the basic characteristics of R.

R can be used as a simple calculator and will process both numbers and conventional mathematical operator symbols. You can run the commands below by placing your cursor at the beginning or end of the line and pressing CTRL+Enter (Windows) or Command+Return (Mac)

```
5+2
```

```
[1] 7
```

You should see the result displayed in the console below.

2.2 Storing Objects

R is especially helpful for allowing us to create and store objects that we can call and manipulate later. We can create names for these objects and then use R's 'assignment operator,' the `<-` symbol, to assign a value to our specified object name. Here, we'll assign the previous calculation to an object that we are calling `our_object`.

If you run this command on your own device, you should see `our_object` populate in the upper-right Environment window. This is where you can find all of the objects that you create in your R session. We can run the object itself, as well as combine it with other operations

```
our_object <- 5+2
```

There are some more baroque ways around this, but it's best to operate under the impression that object names cannot include spaces (or start with numbers). This kind of thing is common in some programming languages, so there are a couple stylistic conventions to address this. I tend to use what's called 'snake case,' which involves replacing spaces with underscores. There's also 'camel case,' where each word has the first letter capitalized, e.g. MyVariableName. I would settle on one that you like and be consistent with it.

```
our_object
```

```
[1] 7
```

```
our_object + 3
```

```
[1] 10
```

```
our_object * 100
```

```
[1] 700
```

2.3 A Note on Functions

R is also useful for its implementation of functions, which you can think of in the sense you likely learned in your math classes. Functions are defined procedures that take some input value, transform that value according to the procedure, and then output a new value.

R comes with a great deal of already defined functions, and we can use these to perform all sorts of helpful operations. You can call a function by indicating it's common name and

then placing it's required inputs between parentheses, e.g. `function_name(input)`.Note that function inputs are also often referred to as 'arguments'. We'll get a lot of mileage out of functions, and part of the initial learning curve of R will be related to getting used to the range of available functions and the syntax you must follow to call them.

Now, let's take a step back and think about some of our basic building blocks in R.

2.4 Vectors and R Data Types

You can think of vectors as ordered sets of values. We can use the `c()` function (short for 'combine') to create a vector made up of the values we provide. Let's make a few different vectors—each one will have 5 separate items in it, and we separate those items with commas. Note that when we want R to process something as text (and not a named object, number, or function), we put it in quotation marks.

```
num_vec <- c(1.2, 3.4, 5.6, 7.1, 2.8)

character_vec <- c("east", "west", "south", "south", "north")

logical_vec <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
```

Let's talk a bit about what we have here. Each of these vectors represents a **data type** in R, or, in other words, one of the basic ways in which R stores data. There are some more data types out there, but these are the most most relevant for us.

- ***Numeric Data:*** As the name suggests, this is the typical fashion in which numbers are stored in R. Numeric data encompasses both *continuous* values and *discrete* values. These are essentially numbers that can have decimal places vs. integers (whole numbers).
- ***Character Data:*** Character here refers to the idea of character strings. This is typically how R stores text data—as distinct strings of text. Note that, while numbers are typically processed as numeric by R, numbers can also become character data if you place them between quotation marks.
- ***Logical Data:*** In R syntax, upper-case 'true' and 'false' have fixed values and, when used without quotes, will refer to these pre-defined logical values. We probably won't use this data type much for analyses, but we will run into them in other places. They can be useful for sorting and searching through subsets of data, and we will also use logical values to turn certain procedures on or off in some functions.

Many R functions will respond differently to different data types, so it's important to keep these in mind when you need to troubleshoot errors.

Take the `mean()` function, for example. As the name implies, this function will return the arithmetic mean of a numeric vector. Let's give it the one we just made above:

```
mean(num_vec)
```

```
[1] 4.02
```

```
(1.2+3.4+5.6+7.1+2.8)/5
```

```
[1] 4.02
```

> **Note**
>
> Note that it gives the same response as if we had manually calculated it. Functions can make our lives a lot easier with larger amounts of data, but always make sure you're familiar with what's going on under the hood of any given function.

But, what happens when we run the following command?

```
mean(character_vec)
```

```
Warning in mean.default(character_vec): argument is not numeric or logical:
returning NA
```

```
[1] NA
```

It doesn't make any sense to take the mean of the cardinal directions, so it will throw a warning message. We need a variable that can be represented numerically. As we'll see, it's a good habit to make sure you know the data type of your variables before you begin your analysis.

Now that we've talked about some of these basic building blocks for data, let's talk about putting them together.

2.5 Data Frames

For the most part, we will be working with data frames. These are collections of data organized in rows and columns. In data science, it's generally preferable for data to take a particular shape wherein each row indicates a single observation, and each column represents a unique variable. This is called the 'tidy' data format.

2.5.1 Building a Data Frame

Let's use the vectors we created above to mock up a little data frame. We will imagine some variables that those vectors could represent. But first, let's make a couple more vectors.

Let's add a vector of participant IDs associated with imaginary people in our mock data set. In accordance with tidy data, each of our rows will then represent a unique person. The column vectors will represent the variables that we are measuring for each person. Lastly, the individual cells will represent the specific values measured for each variable.

For reasons that will become clear in the next section, we are also going to add one more character vector.

```
p_id_vec<-c("p1", "p2", "p3", "p4", "p5")

ordinal_vec<-c("small", "medium", "medium", "large", "medium")
```

Now, let's use a function to create a data frame and store it in a new object.

We can use `data.frame()` for this. `data.frame()` expects that we will give it some vectors, which it will then organize into columns. We could just give it the vectors, and it would take the vector names as column names, e.g.:

```
our_df <- data.frame(p_id_vec, num_vec, character_vec, ordinal_vec, logical_vec)
```

Or we could specify new variable names and use the = sign to associate them with the vector. We will go with this latter strategy because our current vector names do not translate well to variable names.

We'll imagine building a small data frame of dog owners and rename our vectors accordingly.

```
our_df<-data.frame(
  p_id = p_id_vec,
  dog_size = ordinal_vec,
  side_of_town = character_vec,
  food_per_day = num_vec,
  has_a_labrador = logical_vec
)
```

> **Tip**
>
> As a slight tangent, note that we can use line breaks to our advantage with longer strings of code. The above command is identical to the one below, but some find the line-break

strategy more intuitively readable. It's most important that your code works, so you don't have to organize it like that, but know that's an option

```
our_df <- data.frame(p_id = p_id_vec, dog_size = ordinal_vec, side_of_town = character_vec
```

Now our vectors make up meaningful variables in our mock data frame.

- `p_id` = An ID for each participant in our survey of dog owners
- `dog_size` = Owner's ranking of their dog's size
- `side_of_town` = Which part of town the owners reside
- `food_per_day` = The amount of food each owner feeds their dog daily (in ounces)
- `has_a_labrador` = true/false indicator for whether the owner has a lab or not

2.5.2 Examining our Data Frame

Take a look at our new data frame by clicking on the object in our Environment window at the upper right, or by running the command `View(our_df)`.

Once we have created a data frame, we can refer to individual variable vectors with the `$` operator in R

```
our_df$food_per_day
```

```
[1] 1.2 3.4 5.6 7.1 2.8
```

```
mean(our_df$food_per_day)
```

```
[1] 4.02
```

We can look at some basic characteristics of our variables with the `summary()` function. Note that it will return different information depending on the data type of the variable

```
summary(our_df)
```

```
    p_id              dog_size         side_of_town        food_per_day
 Length:5           Length:5           Length:5           Min.   :1.20
 Class :character   Class :character   Class :character   1st Qu.:2.80
 Mode  :character   Mode  :character   Mode  :character   Median :3.40
                                                          Mean   :4.02
```

```
                                                    3rd Qu.:5.60
                                                    Max.   :7.10
 has_a_labrador
 Mode :logical
 FALSE:3
 TRUE :2
```

Let's think about these for a second.

The summary of `has_a_labrador` makes sense. It's recognized as a logical vector and tells us the number of TRUEs and FALSEs

`food_per_day` works as well. We're dealing with a continuous variable that allows for decimal places, so it makes sense to take the mean and look at the range and distribution.

But how about `side_of_town`? What that summary tells us is that this variable is a character type (or class). 'Length' refers to the size of the vector. So, a vector containing 5 items would be a vector of length 5. But does it make sense for us to treat the `side_of_town` variable as 5 totally separate strings of characters?

```
summary(our_df$side_of_town)
```

```
   Length     Class      Mode
        5 character character
```

Not quite. When we have two entries of "south", for example, we want those responses to be grouped together and not treated as unique entries.

```
our_df$side_of_town
```

```
[1] "east"  "west"  "south" "south" "north"
```

For this, we will want another key R data type.

2.6 Factors

2.6.1 Unorderd Factors

Factors are often the best way to treat categorical variables (nominal or ordinal) in R. Factors are a certain kind of vector that can only contain a number of pre-defined values. Each of these pre-defined values is considered a 'level' of the factor. So, we want `side_of_town` to be a factor variable with 4 levels: east, west, south, and north.

We can turn this variable into a factor with R's `as.factor()` function.

```
our_df$side_of_town <- as.factor(our_df$side_of_town)
```

Check the `summary()` output again and notice how the output is reported now. Instead of simply listing that the vector contained 5 character strings, we can now see the different levels and the number of people who belong to each side of town.

```
summary(our_df)
```

```
     p_id              dog_size          side_of_town  food_per_day
 Length:5           Length:5           east :1      Min.   :1.20
 Class :character   Class :character   north:1      1st Qu.:2.80
 Mode  :character   Mode  :character   south:2      Median :3.40
                                       west :1      Mean   :4.02
                                                    3rd Qu.:5.60
                                                    Max.   :7.10

 has_a_labrador
 Mode :logical
 FALSE:3
 TRUE :2
```

2.6.2 Ordered Factors

Now, let's think about `dog_size`. This should clearly be a factor variable as well. But, unlike `food_per_day`, the levels of this variable have an apparent order, from small to large.

The `factor()` function allows us to turn a vector into a factor, as well as manually specify the levels. Additionally, we can activate a process in the function letting it know that we want the order to matter.

```
our_df$dog_size <- factor(
  our_df$dog_size,
  levels=c("small", "medium", "large"),
  ordered = TRUE
  )
```

Take a look back at the summary. Now, instead of 5 separate character strings, we can see the breakdown of how many people have a dog of a certain size.

```
summary(our_df)
```

```
     p_id              dog_size side_of_town  food_per_day  has_a_labrador
 Length:5           small :1   east :1      Min.   :1.20   Mode :logical
 Class :character   medium:3   north:1      1st Qu.:2.80   FALSE:3
 Mode  :character   large :1   south:2      Median :3.40   TRUE :2
                               west :1      Mean   :4.02
                                            3rd Qu.:5.60
                                            Max.   :7.10
```

Note that the `str()` command is also useful for quickly gleaning the various data types of variable columns within a data frame. It will show us our variable names, the data types, and then a preview of the first several values in each variable column.

We can also verify that `dog_size` has been successfully re-coded as an ordered factor.

```
str(our_df)
```

```
'data.frame':   5 obs. of  5 variables:
 $ p_id          : chr  "p1" "p2" "p3" "p4" ...
 $ dog_size      : Ord.factor w/ 3 levels "small"<"medium"<..: 1 2 2 3 2
 $ side_of_town  : Factor w/ 4 levels "east","north",..: 1 4 3 3 2
 $ food_per_day  : num  1.2 3.4 5.6 7.1 2.8
 $ has_a_labrador: logi  TRUE FALSE TRUE FALSE FALSE
```

There are cases where you will want to convert a column like `p_id` to a factor variable as well, but often we just need a variable like `p_id` to serve as a searchable index for individual observations, so we can leave it be for now.

This is all part of the process of data cleaning, where we make sure our data is structured in a fashion that's amenable to analysis. This re-coding of variables is an essential component, and we'll see plenty more tasks in this vein when we work with GSS data later on.

As we close this section, here is a figure to help you internalize the hierarchy of variable types based on the levels of measurement. The bottom level of the hierarchy (in green) reflects the R data type that is best aligned with a particular measurement level. Also recall that numeric data can either be interval or ratio, though we will generally treat these similarly.

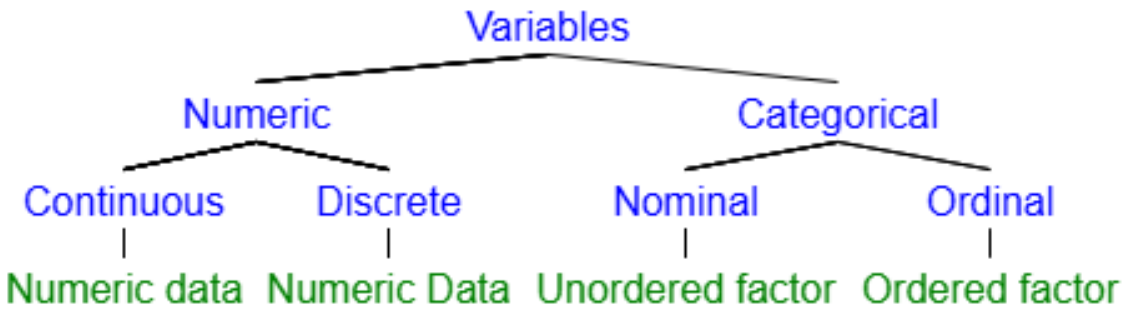


Figure 2.1: A hierarchy of variables and their corresponding R data types

For our last bit, let's learn a little about working with functions that don't come included in base R.

3 Packages and the Tidyverse

R's open-source culture has encouraged a rich ecosystem of custom functions designed by scientists and researchers in the R userbase. These come in the form of 'packages', which are suites of several related functions. For example, there are packages for conducting statistical tests, producing data visualizations, generating publication-ready tables, and all manner of other tasks.

3.1 Loading Packages

Let's try this out with one of the better known R packages–'tidyverse'. This is actually a collection of several packages with a variety of interrelated functions for 'tidying', visualizing, and analyzing data. We will focus on what we need from 'tidyverse', but, if you're curious, you can read more here: https://www.tidyverse.org/

If you're on a lab computer, this package may already be installed. Let's check by running the following command:

```
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   3.5.2     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.1.0
-- 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 beco
```

If you receive an error when you run this, you likely do not have the package installed on your system. This is also probably the case if you are on your personal device and only recently acquired R.

If you got an error, run the following command:

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

With a few exceptions, you will always install new packages in this fashion: install.packages("package_name")

After it's done installing, go back and run the library(tidyverse) command again. Note that you always need to do this for an added package. Whether you've had it for a while or just installed it, you need to load any outside package into your current session by placing its name in the library() function.

```
library(tidyverse)
```

3.2 Bringing in our Data

Let's try bringing in a data frame to play with a few tidyverse functions. We'll use the `load()` function to bring in a subset of the General Social Survey, which contains a few variables from the 2022 survey wave. Run the following command and select the file "our_gss.rda"

```
load(file.choose())
```

The `file.choose()` function will open up a file-explorer window that allows you to manually select an R data file to load in. We'll talk about some other ways to import data files using R syntax next time.

Go ahead and take a look at the data frame. Each GSS survey wave has about 600-700 variables in total, so I've plucked several and done a little pre-processing to get us a subset to work with. All the variables here have pretty straightforward names, but I'll note that `realrinc` is a clear outlier there. This is short for 'Real respondent's income' and reflects the respondent's income reported in exact dollar amounts. I'll put a summary here so you can take a look if you're not following along with your own script.

```
summary(our_gss)
```

```
      year            id              age           race          sex
 Min.   :2022   Min.   :   1.0   Min.   :18.00   black: 565   female:1897
 1st Qu.:2022   1st Qu.: 886.8   1st Qu.:34.00   other: 412   male  :1627
 Median :2022   Median :1772.5   Median :48.00   white:2514   NA's  :  20
 Mean   :2022   Mean   :1772.5   Mean   :49.18   NA's :  53
 3rd Qu.:2022   3rd Qu.:2658.2   3rd Qu.:64.00
 Max.   :2022   Max.   :3545.0   Max.   :89.00
```

```
                                  NA's   :208
   realrinc                          educ
 Min.   :   204.5   12th grade        :909
 1st Qu.:  8691.2   4 years of college:697
 Median : 18405.0   2 years of college:506
 Mean   : 27835.3   1 year of college :268
 3rd Qu.: 33742.5   6 years of college:210
 Max.   :141848.3   (Other)           :934
 NA's   :1554       NA's              : 20
                                partyid
 independent (neither, no response):835
 strong democrat                   :595
 not very strong democrat          :451
 strong republican                 :431
 independent, close to democrat    :400
 (Other)                           :797
 NA's                              : 35
```

3.3 Data Wrangling with Tidyverse

Let's use this subset to explore some tidyverse functionality. Tidyverse includes several functions for efficiently manipulating data frames in preparation for analyses. We will encounter a number of these throughout our time with R, but I want to briefly introduce a few key functions and operations that we will dig into more next time.

3.3.1 select()

It happens quite often that we have a data frame containing far more variables than we ultimately need for a given analysis. The `select()` function allows us to quickly subset data frames according to the variable columns we need.

```
sex_inc <- select(our_gss, id, sex, realrinc)
```

You should now have an object that contains all 3,544 observations, but includes only the 3 columns that we specified with `select()`.

```
summary(sex_inc)
```

```
       id              sex          realrinc
 Min.   :   1.0   female:1897   Min.   :   204.5
 1st Qu.: 886.8   male  :1627   1st Qu.:  8691.2
 Median :1772.5   NA's  :  20   Median : 18405.0
 Mean   :1772.5                 Mean   : 27835.3
 3rd Qu.:2658.2                 3rd Qu.: 33742.5
 Max.   :3545.0                 Max.   :141848.3
                                NA's   :1554
```

3.3.2 filter()

`filter()` functions similarly except that, instead of sub-setting by specific variables, it allows you to subset by specific values. So, let's take the `sex_inc` object we just created above. We now have this subset of three variables—id, sex, and income—but let's imagine we want to answer a question that's specific to women.

In order to do that, we need to *filter* the data to include only observations where the value of the variable `sex` is 'female'.

```
fem_inc <- filter(sex_inc, sex=="female")
```

Note that the `fem_inc` object still has 3 variables, but there are now roughly half the observations, suggesting that we have successfully filtered out the male observations.

```
summary(fem_inc)
```

```
       id            sex          realrinc
 Min.   :   1   female:1897   Min.   :   204.5
 1st Qu.: 879   male  :   0   1st Qu.:  7668.8
 Median :1783                 Median : 15337.5
 Mean   :1772                 Mean   : 22702.1
 3rd Qu.:2664                 3rd Qu.: 27607.5
 Max.   :3544                 Max.   :141848.3
                              NA's   :883
```

3.3.3 summarize()

As the name suggests, `summarize()` allows us to quickly summarize information across variables. It will give us a new data frame that reflects the particular summaries that we ask for, which can be very useful for quickly generating means within and across different variables.

Let's try another simple use case, and then I'll introduce a concept that will help us learn to chain these functions together.

```
mean_inc <- summarize(our_gss, "mean_inc"=mean(realrinc, na.rm=TRUE))
```

> **Note**
>
> You probably noticed the `na.rm = TRUE` input that I supplied for the above function. This is short for 'remove NAs', which we need to do when a variable has any NA values. If we don't, R will return an error, because it does not know to disregard NA values when calculating a column mean.

This gives us a new data frame that we called `mean_inc`. It should have 1 row and 1 column, and it just gives us the average income of a person in our GSS subset—about $28,000/year.

```
mean_inc
```

```
  mean_inc
1 27835.33
```

Now, this is not really all that impressive when we are asking for a broad summary like this. In fact, if all we wanted was to see the average income, we could get that more easily, e.g.

```
mean(our_gss$realrinc, na.rm = TRUE)
```

```
[1] 27835.33
```

The true power of `summarize()` comes from chaining it together with other tidyverse functions. However, in order to do that, we will need to learn about one more new R operation.

3.4 The Pipe

This one might be a little unintuitive, so don't worry if it doesn't immediately click. We will continue to get plenty of practice with it over the next couple of sessions.

The pipe operator looks like this: `|>`. What it does is take whatever is to the left of the symbol and 'pipe' it into the function on the right-hand side. That probably sounds a little strange, so let's see some examples.

We'll refer back to our `summarize()` command from above.

```
mean_inc <- summarize(our_gss, "mean_inc"=mean(realrinc, na.rm=TRUE))
```

This is equivalent to...

```
mean_inc <- our_gss |>
  summarize("mean_age" = mean(realrinc, na.rm=TRUE))
```

Notice that, in the first command, the first input that we give `summarize()` is the data frame that we want it to work with.

In the command featuring the pipe operator, we supply the data frame and then pipe it into `summarize()`. The real magic comes from chaining multiple pipes together. This will likely take a little practice to get used to, but it can become a very powerful tool in our R arsenal.

3.4.1 Putting It All Together

Let's illustrate with an example. I'll let you know what I want to do in plain English, and then I will execute that desire with multiple piped commands.

Ultimately, I want to see the mean income, but I want to see the mean broken down by sex rather than the mean of the entire data frame.

So, I want to take a **selection** of variables from `our_gss`. I want these variables to be **grouped by** `sex`. Finally, I want to see a **summary** of the mean according to this variable grouping.

```
sex_means <- our_gss |>
  select(id, sex, realrinc) |>
  group_by(sex) |>
  summarize("mean_inc" = mean(realrinc, na.rm=TRUE))
```

> **Note**
>
> Note that I've added one new function here, `group_by()`. We'll see this plenty more in other cases, and it will help us quite a bit when it comes to summary statistics. As the name suggests, it applies a grouping structure that can be leveraged with all kinds of other functions.

```
sex_means
```

```
# A tibble: 3 x 2
  sex    mean_inc
  <fct>     <dbl>
1 female   22702.
2 male     33191.
3 <NA>     11248.
```

Notice that this will give us a new data frame with the average income of `sex` for both `male` and `female`. However, we actually get a 3rd row for NAs, which is a 3rd category of the `sex` variable. At the level of the survey, it's important to keep track of this for a number of reasons, but we do not need them for our purposes. So, we can add a final pipe with the `drop_na()` function in order to drop the NA level of `sex`.

```
sex_means <- our_gss |>
  select(id, sex, realrinc) |>
  group_by(sex) |>
  summarize("mean_inc" = mean(realrinc, na.rm=TRUE)) |>
  drop_na(sex)

sex_means
```

```
# A tibble: 2 x 2
  sex    mean_inc
  <fct>     <dbl>
1 female   22702.
2 male     33191.
```

We will see plenty more on the tidyverse, so don't fret if these new functions are not completely clicking right away. We will keep building with these in the next unit and hopefully accumulate some muscle memory.

Part II

Day 2: Survey Data and Univariate Analysis

Part III

Day 3: Bivariate Analysis with the GSS

Part IV

Day 4: Multivariable Analysis and Elaboration

Part V

Day 5: Data Visualization and ggplot

Introduction

Up to this point, we have worked with pretty simple visualizations, just so we can quickly glean important information about our statistical models without spending too much time focusing on the aesthetics of the visuals.

In this lab, we will learn a bit more about R's graphical capability—especially through tidyverse's `ggplot`—which provides us with incredible customizability. We will learn how to fine-tune some of the visuals we have already worked with, and we will preview some other common visual styles that can manage with `ggplot`.

Visualization and Analytical Thinking

Before we start working with some of these new visual tools, I want to take an opportunity to stress the importance of visualization more generally. It's easy to see the process of presenting visuals as something somewhat superficial, but visualization can be critical for defining the kind of questions we can ask about our data.

For now, I'm going to obscure the code I'm using for this document. We will learn more about the kind of commands I used to generate the following figures, but I don't want anyone to get bogged down initially. I'll use these visuals to help impart an important lesson about data visualization's in the research process.

Thirteen Data Sets

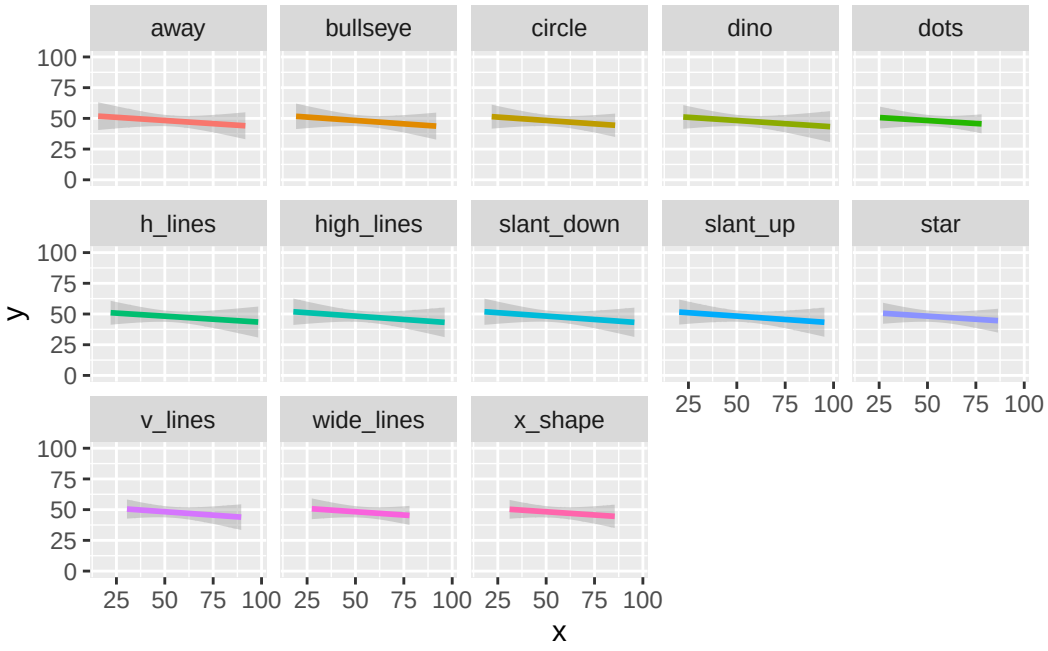
Let's take a look at a collection of thirteen different data sets. Each data set has 142 observations with 2 columns, labeled x & y.

I'll use some tidyverse commands to get some summary statistics for each of the data sets, including the mean of both variables and their standard deviations. Let's see what seems to distinguish some of these data sets from one another.

```
# A tibble: 13 x 4
   mean_x mean_y std_x std_y
    <dbl>  <dbl> <dbl> <dbl>
```

```
 1   54.3   47.8   16.8   26.9
 2   54.3   47.8   16.8   26.9
 3   54.3   47.8   16.8   26.9
 4   54.3   47.8   16.8   26.9
 5   54.3   47.8   16.8   26.9
 6   54.3   47.8   16.8   26.9
 7   54.3   47.8   16.8   26.9
 8   54.3   47.8   16.8   26.9
 9   54.3   47.8   16.8   26.9
10   54.3   47.8   16.8   26.9
11   54.3   47.8   16.8   26.9
12   54.3   47.8   16.8   26.9
13   54.3   47.8   16.8   26.9
```

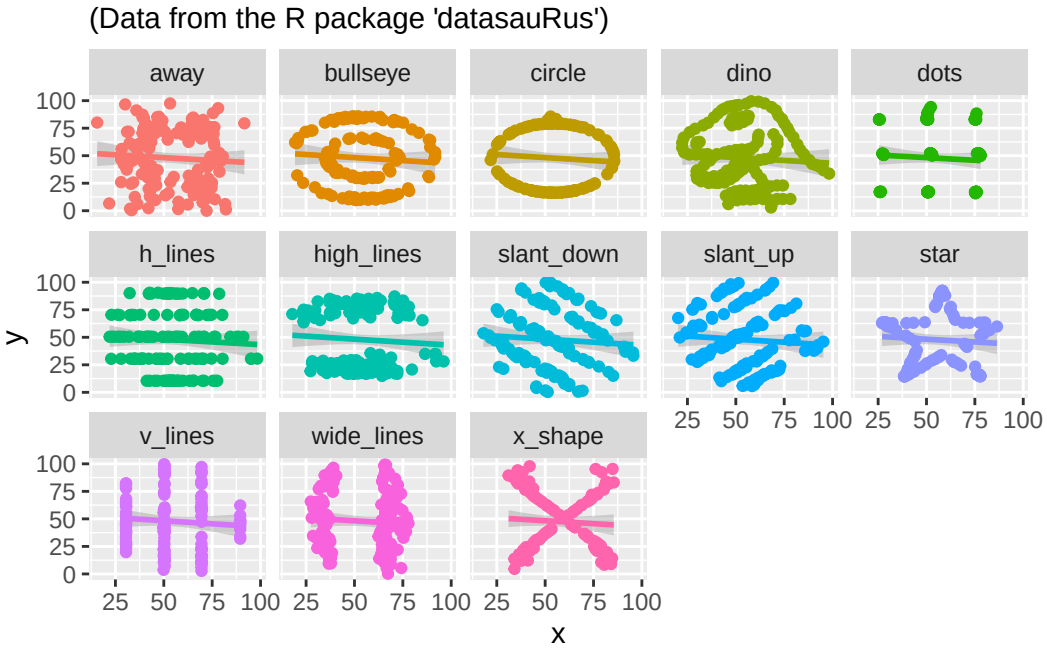
Well, there's not much we can say here. All the summary statistics are identical. Why don't we try modeling a linear relationship between the x and y variables. Maybe looking at the correlations will tell us something. I'll display the linear regression lines for each data set below.



Okay. This is not revealing much either. All the lines seem to have the same slope, which shows a (slight) negative relationship where y decreases as x increases. The correlations aren't revealing any notable distinctions.

But wait. One thing we can see here is that, while the correlations appear to be about the same, there are some differences in the ranges of values. Note that the regression lines don't extend across the same range of x-axis values in each data set. Maybe there is something here after all.

Let's just go ahead and plot the actual data over our regression lines.



Now there's some distinction!

This is a tongue in cheek data set known as the 'datasaurus dozen'. It's often used in intro statistical classes to help illustrate the importance of visualization. It's inspired by another conceptually similar data set known as 'Anscombe's quartet' which likewise stresses the role of plotting data in producing well informed analyses.

In Sum

So, take this as a showcase of the importance of visualizing your data. This isn't to discount summary statistics and other numeric description of data—those are still invaluable for us.

Rather, cases like Datasaurus or Anscombe's quartet highlight the necessity of understanding the shape of your data. This will determine the kind of questions you can ask with the data, as well as the kind of statistical tools you need to describe it.

For example, in the case we just examined, those x and y variables do not have any kind of clear linear relationship. In that case, tools like regression that assume linearity are not appropriate. Any relationship between the variables could only be explored through other statistical means.

So, making our figures and tables look aesthetically pleasing is indeed valuable in its own right, but don't underestimate the utility of good visualization for the analytic process itself.