Introduction to R for Quantitative Social Science Research

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.

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.

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.

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.

Why R?

Ultimately, R is just one of several different options we could have gone with for a course like this. STATA, SPSS, Python, and even MS Excel are used with regularity for quantitative analysis in academia and industry. However, R has a few advantages that make it well-suited for us.

While software like Excel, SPSS, and STATA arguably have more accessible, user-friendly interfaces, the upper limit of their capabilities is far lower than R. On the other end of the spectrum, Python is a little overkill for our purposes. While it has some excellent resources for data science, it's also used for a wider variety of programming tasks related to web- and software development. I once heard a computational sociologist joke that using Python for something that can be done in R is like using a nuke when all you need is a hammer. While R is not quite as expansive as Python, it's specialization in statistical computing provides a helpful balance for us when it comes to our goals for the course.

Another big perk of R is that it is completely free. This is true for Python as well, but SPSS, STATA, and Excel all require the purchase of a license, and some of them can get very pricey. You have access to all of these programs as an NC State student, but you will be able to use R regardless of whether you are on an NC State computer or even enrolled in the university at all. Relatedly, R is completely open-source—you can view it's source code, and even modify it for your own purposes. This makes R incredibly customizable, and this open-source culture has brought about a dedicated community of researchers and data scientists who regularly contribute new functional add-ons to R.

Lastly, R is quite marketable as a technical skill. Especially for those who want to go on to do research of any kind, experience with R will likely be seen as a plus. I've been on the lookout for various research, teaching, and industry jobs as I get ready to enter the job market, and I see calls for R as a required or preferenced skill all the time.

In sum, R provides us with the ideal balance of computing power and feasibility while helping keep our pockets full and giving us some skills that translate well beyond the course.

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,

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.

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.

Part I

Day 1: Getting Started

1 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.

1.1 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 in your script file and pressing CTRL+Enter (Windows) or Command+Return (Mac)

```r
5+2
```

```
[1] 7
```

You should see the result displayed in the console below.

1.2 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

```r
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.

```r
our_object
```

```
[1] 7
```

```r
our_object + 3
```

```
[1] 10
```

```r
our_object * 100
```

```
[1] 700
```

1.3 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.

1.4 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.

```r
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:

```r
mean(num_vec)
```

```
[1] 4.02
```

Note

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

```
[1] 4.02
```

Observe that `mean()` 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?

```r
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.

1.5 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.

1.5.1 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.

```r
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.:

```r
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.

```r
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

1.5.2 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.

1.6 Factors

1.6.1 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        
```

1.6.2 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.

```r
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.

```r
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.

```r
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.

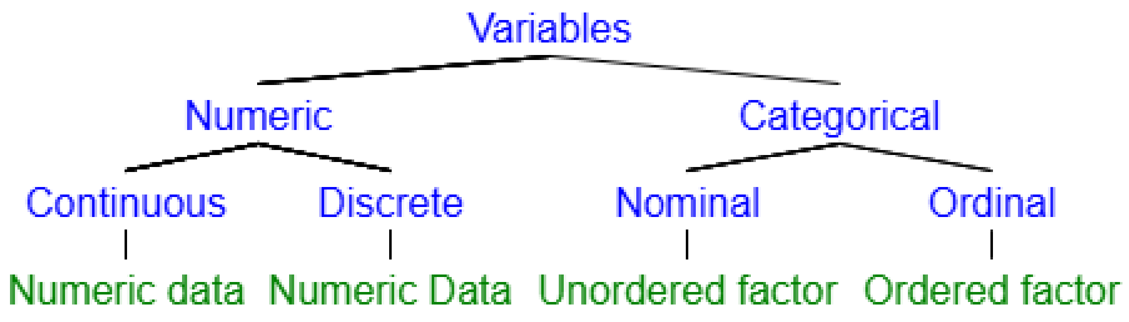


Figure 1.1: A hierarchy of variables and their corresponding R data types

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

2 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.

2.1 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:

```r
library(tidyverse)
```

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:

```r
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.

```r
library(tidyverse)
```

2.2 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"

```r
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.

```r
summary(our_gss)
```

```
      year            id               age            race            sex      
 Min.   :2022   1      :   1   Min.   :18.00   white:2514   female:1897  
 1st Qu.:2022   2      :   1   1st Qu.:34.00   other: 412   male  :1627  
 Median :2022   3      :   1   Median :48.00   black: 565   NA's  :  20  
 Mean   :2022   4      :   1   Mean   :49.18   NA's :  53                
 3rd Qu.:2022   5      :   1   3rd Qu.:64.00                             
 Max.   :2022   6      :   1   Max.   :89.00                             
                (Other):3538   NA's   :208                               
    realrinc                                   partyid    
 Min.   :   204.5   independent (neither, no response):835  
 1st Qu.:  8691.2   strong democrat                   :595  
 Median : 18405.0   not very strong democrat          :451  
 Mean   : 27835.3   strong republican                 :431  
 3rd Qu.: 33742.5   independent, close to democrat    :400  
 Max.   :141848.3   (Other)                           :797  
 NA's   :1554       NA's                              : 35  
          happy                   marital                                attend
 not too happy: 799   divorced     : 608   never                      :1149
 pretty happy :1942   married      :1468   about once or twice a year: 464
 very happy   : 779   never married:1095   every week                 : 441
 NA's         :  24   separated    : 103   less than once a year      : 416
                      widowed      : 255   several times a year       : 346
                      NA's         :  15   (Other)                    : 693
                                           NA's                       :  35
    cappun
 favor :2013
 oppose:1327
 NA's  : 204
```

2.3 Data Wrangling with Tidyverse

Let's use this subset to explore some tidyverse functionality. One of the packages included in the tidyverse is `dplyr`, which 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 `dplyr` functions and operations that we will dig into more next time.

2.3.1 select()

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

This function takes a data frame as its first input, and all following inputs are the variable columns that you want to keep

```r
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()`.

```r
summary(sex_inc)
```

```
       id           sex          realrinc       
 1      :   1   female:1897   Min.   :   204.5  
 2      :   1   male  :1627   1st Qu.:  8691.2  
 3      :   1   NA's  :  20   Median : 18405.0  
 4      :   1                 Mean   : 27835.3  
 5      :   1                 3rd Qu.: 33742.5  
 6      :   1                 Max.   :141848.3  
 (Other):3538                 NA's   :1554      
```

2.3.2 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       
 1      :   1   female:1897   Min.   :   204.5  
 3      :   1   male  :   0   1st Qu.:  7668.8  
 4      :   1                 Median : 15337.5  
 7      :   1                 Mean   : 22702.1  
 9      :   1                 3rd Qu.: 27607.5  
 10     :   1                 Max.   :141848.3  
 (Other):1891                 NA's   :883       
```

2.3.3 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 summaries that we ask for, which can be very useful for quickly generating descriptive statistics. We will use this to get the mean income value for our data frame.

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

ℹ 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 screw up, because it does not know to disregard NA values when calculating a column mean unless we tell it to.

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. I'll show you that in a moment, but let's take a look at one more helpful tidyverse function.

2.3.4 group_by()

Often when we're using a function like `summarize()`, we want to get summaries for all kinds of different subgroups within our data set. For example, we may want the mean for each value of `sex` or `partyid`, rather than for all people in the data frame. We can do this with `group_by`.

This function may seem a little unusual when used in isolation, because it does not seem to do much on the surface.

```
our_gss <- group_by(our_gss, partyid)
```

When you run that function, you will not generate any new objects, and you will not notice anything different about the data frame.

What it does is overlay a grouping structure on the data frame, which will in turn affect how other tidyverse functions operate.

Compare the output of `summarize()` run on this grouped version of our data frame with the use of `summarize()` above.

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

mean_inc
```

```
# A tibble: 9 x 2
  partyid                            mean_inc
  <fct>                                 <dbl>
1 strong democrat                      30677.
2 independent (neither, no response)   21570.
3 not very strong republican           29101.
4 not very strong democrat             31743.
5 independent, close to democrat       27916.
6 other party                          23891.
7 independent, close to republican     26825.
8 strong republican                    32376.
9 <NA>                                 16922.
```

We ran the same `summarize()` command as before, but now it reflects the grouping structure that we imposed.

2.4 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.

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

This is equivalent to…

```r
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.

2.4.1 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` and `partyid`.

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

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

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

Note

We can use `drop_na()` to do as the function's name suggests. When we learned about `group_by()` above, you may have noticed that a mean was reported for an `NA` category within the `partyid` variable. Any time you notice this and want your summaries to exclude these NA categories, just include that variable as an input to `drop_na()`.

```r
sexpol_means
```

```
# A tibble: 16 x 3
# Groups:   sex [2]
   sex    partyid                             mean_inc
   <fct>  <fct>                                  <dbl>
 1 female strong democrat                       30111.
 2 female independent (neither, no response)   17923.
 3 female not very strong republican            22448.
 4 female not very strong democrat              24300.
 5 female independent, close to democrat        19961.
 6 female other party                           26595.
 7 female independent, close to republican      20095.
 8 female strong republican                     21584.
 9 male   strong democrat                       31600.
10 male   independent (neither, no response)   25474.
11 male   not very strong republican            34544.
12 male   not very strong democrat              41233.
13 male   independent, close to democrat        36947.
14 male   other party                           22450.
15 male   independent, close to republican      31393.
16 male   strong republican                     40210.
```

So, using `dplyr`, we can quickly subset and manipulate data frames in just a few lines of relatively straightforward code. Here we have all the means for each value of `sex` and `partyid`, which would have been a tedious task had we calculated them all manually.

We will see plenty more on the tidyverse, so don't fret if you don't feel completely confident with these yet. It takes practice getting used to Rs peculiarities. 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

3 Recoding Variables

Today's venture concerns univariate analysis, i.e. the quantitative description of a single variable. Before we do that, however, we need to familiarize ourselves with some data-cleaning procedures.

3.1 Background

Recoding a variable involves manipulating the underlying structure of our variable such that we can use it for analysis. We did a little recoding during the last unit when we converted character vectors into factor variables. This allowed us to align R data types with the appropriate level of measurement.

There are also occasions when we need a variable to be translated from one level of measurement to another. For example, we may want to convert a ratio variable for "number of years of education" into an ordinal variable reflecting categories like "less than high school", "high school diploma", "Associates degree", and so on.

We may also want to collapse the categories of ordinal variables for some analyses. Consider a variable with a Likert-scale agreement rating, where you responses like "strongly agree," "moderately agree," "slightly agree," and so forth. You may decide to collapse these categories into categories of "Agree" and "Disagree".

We will get some practice doing this sort of thing, which is an essential component of responsible analysis. Additionally, our next unit on bivariate analysis will require us to work with categorical variables in particular, so we need to be capable of converting any numeric variables.

3.2 Converting Numeric to Categorical

We will start by recoding `age`—a ratio variable—into an ordinal variable reflecting age groupings. The same strategies we use here will work for any numeric variable.

3.2.1 Setting up our workspace

As usual, let's make sure we load in `tidyverse` along with our GSS data.

```r
library(tidyverse)
load("our_gss.rda")
```

Let's double check the structure of our data frame.

```r
str(our_gss)
```

```
'data.frame':	3544 obs. of  11 variables:
 $ year    : num  2022 2022 2022 2022 2022 ...
 $ id      : Factor w/ 4149 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ age     : num  72 80 57 23 62 27 20 47 31 72 ...
 $ race    : Factor w/ 3 levels "white","other",..: 1 1 1 1 1 1 2 1 1 NA ...
 $ sex     : Factor w/ 2 levels "female","male": 1 2 1 1 2 2 1 2 1 1 ...
 $ realrinc: num  40900 NA 18405 2250 NA ...
 $ partyid : Factor w/ 8 levels "strong democrat",..: 1 2 3 1 2 4 5 1 5 1 ...
 $ happy   : Ord.factor w/ 3 levels "not too happy"<..: 1 1 1 1 2 2 2 2 2 2 ...
 $ marital : Factor w/ 5 levels "divorced","married",..: 1 2 1 3 3 3 3 2 3 3 ...
 $ attend  : Factor w/ 9 levels "never","less than once a year",..: 3 3 1 5 3 1 2 2 1 8 ...
 $ cappun  : Factor w/ 2 levels "favor","oppose": 2 1 1 2 2 2 2 2 1 NA ...
```

Note

You might notice the 'int' category, which is short for 'integer'. This is a subtype of numeric data in R. Variables that are exclusively whole numbers are often recorded in this way, but we can work with them in R just like we can other sorts of numbers

3.2.2 The 'age' variable

We can take a look at all the values of age (along with the # of respondents in each age category) using the `count()` function.

```r
count(our_gss, age)
```

However, I'm not going to display those results here, as it will be a particularly long table of values (with 70+ different ages). It's fine to run it—it won't crash R or anything—but it will it clutter up this page. Let's take this as a good opportunity to take advantage of the fact that we are using one of the better funded and well-organized surveys in all of social sciences. As such, there's extensive documentation about all of the variables measured for the GSS. Go ahead and take a look at the age variable via the GSS Data Explorer, which allows us to search for unique variables and view their response values, the specific question(s) that was asked on the survey, and several other variable characteristics.

The responses range from 18 - 89 (in addition to a few categories for non-response). However, note that there's something unique about value 89. It's not just 89 years of age, but 89 & older. This isn't a real issue for our purposes, but take this as encouragement to interface with the codebook of any publicly available data you use. There's some imprecision at the upper end of this variable, and that might not be obvious without referencing the codebook.

For the purposes of this exercise, let's go ahead and turn age into a simple categorical variable with 3 levels—older, middle age, and younger. I'm going to choose the range somewhat arbitrarily for now. We can use univariate analysis to inform our decision about how to break up a numeric variable, so we will revisit this idea again later on.

- **Younger** = 18 - 35
- **Middle Age** = 36 - 49
- **Older** = 50 and up

At the end of the day, what we need to do is 1.) create a new variable column 2.) populate that column with ordinal labels that correspond with each respondent's numeric age interval.

3.2.3 New columns with mutate()

First, let's consider the `mutate()` function. This function takes a data frame and appends a new variable column. This new variable is the result of some calculation applied to an existing variable column.

Let's look at an application fo `mutate()` to get a feel for it. Now, this wouldn't be the best idea for a couple reasons, but, as an illustration, let's say we wanted to convert our yearly income values to an hourly wage (assuming 40 hrs/week).

`mutate()` takes a data frame as its input, and then we provide the name of our new variable column(s) along with the calculation for this new variable. Below, I use `mutate()` to create a new column called `hr_wage`. Then, I tell R that the `hr_wage` variable should be calculated by taking each person's income value and diving that by 52 weeks in a year, and then 40 hours in a week.

```r
# Without the pipe operator
our_gss <- mutate(
  our_gss,
  hr_wage = (realrinc/52)/40
  )
```

```
# With the pipe operator
our_gss <- our_gss |>
  mutate(
    hr_wage = (realrinc/52)/40
  )
```

Take a look at your new data frame object. I'll show a summary here to verify that we got our new column.

```
summary(our_gss)
```

```
      year            id              age              race           sex
 Min.   :2022   1      :   1   Min.   :18.00   white:2514   female:1897
 1st Qu.:2022   2      :   1   1st Qu.:34.00   other: 412   male  :1627
 Median :2022   3      :   1   Median :48.00   black: 565   NA's  :  20
 Mean   :2022   4      :   1   Mean   :49.18   NA's :  53
 3rd Qu.:2022   5      :   1   3rd Qu.:64.00
 Max.   :2022   6      :   1   Max.   :89.00
                (Other):3538   NA's   :208
    realrinc                                   partyid
 Min.   :   204.5   independent (neither, no response):835
 1st Qu.:  8691.2   strong democrat                   :595
 Median : 18405.0   not very strong democrat          :451
 Mean   : 27835.3   strong republican                 :431
 3rd Qu.: 33742.5   independent, close to democrat    :400
 Max.   :141848.3   (Other)                           :797
 NA's   :1554       NA's                              : 35
           happy             marital                               attend
 not too happy: 799   divorced     : 608   never                      :1149
 pretty happy :1942   married      :1468   about once or twice a year: 464
 very happy   : 779   never married:1095   every week                 : 441
 NA's         :  24   separated    : 103   less than once a year      : 416
                      widowed      : 255   several times a year       : 346
                      NA's         :  15   (Other)                    : 693
                                           NA's                       :  35
    cappun        hr_wage
 favor :2013   Min.   : 0.09832
 oppose:1327   1st Qu.: 4.17849
 NA's  : 204   Median : 8.84856  
               Mean   :13.38237  
               3rd Qu.:16.22236  
               Max.   :68.19631  
               NA's   :1554      
```

So, we can use `mutate()` to add a new column that contains our recoded variable. We just need a way to specify a calculation that takes into account the specific intervals we want for our ordinal labels. For this, we need one more function.

3.2.4 Custom intervals with cut()

We can use the `cut()` function to specify the intervals we want for our age groupings, and then we will combine it with `mutate()` to generate our recoded age variable. Specifically, `cut()` takes our intervals and turns each of them into a level in a new factor variable.

But first, I want to give a little context on interval notation in mathematics. It will help us all be a little more precise when we talk about ranges, and it will also help us understand an input we need to provide for `cut()`.

3.2.4.1 An aside on intervals

In mathematic terms, an interval is the set of all numbers in between two specified end points. In formal notation, these are indicated with the two endpoints placed in brackets, e.g [1,5] as the interval of 1 through 5.

There are some technical terms to describe whether or not we want to include the endpoints when we are talking about a particular interval.

- **Closed interval**: This is when both end points are included in the interval. Closedness is indicated with square brackets, so, the closed interval of 1 through 5 would be written just like I have above—[1,5]. This indicates any number x where $1 \leq x \leq 5$
- **Open interval**: This is when neither endpoint is included in the interval. In interval notation, openness is indicated with parentheses rather than square brackets, so the open interval of 1 through 5 would be written as (1,5). This interval would indicate any number x where $1 < x < 5$
- **Left-open interval**: This is when the left-hand endpoint is not included, but the right-hand endpoint is. This would be written as (1,5] in interval notation, and that interval would indicate any number x where $1 < x \leq 5$

- **Right-open interval**: When the right-hand endpoint isn't included but the left endpoint is, you have a right-open interval. This would be written as [1,5) in interval notation and would indicate any number x where $1 \leq x < 5$.

We will be working with the right-open interval format, and we can specify that in `cut()`.

Now, let's return to our task at hand.

3.2.4.2 Inputs for cut()

We'll go ahead and work with cut directly in `mutate()`, as its going to be the calculation that we are providing for the new column we generate with `mutate()`.

As a reminder, here are the intervals we need:

- **Younger** = 18 - 35
- **Middle Age** = 36 - 49
- **Older** = 50 and up

The following code may look a little chaotic at first glance, but it's really the same sort of thing that we did with `mutate()` above. It's just that `cut()` is a little bit more involved of a calculation than the simple division we did in our example.

Just like above, we are applying `mutate()` to `our_gss`, and we are giving the name of our new variable column (`age_ord`, in this case) as the first input. Then, for the calculation, we give the `cut()` function and the arguments it needs.

cut() and its inputs

- The variable for which we want to specify intervals (`age`)
- **breaks**: This is where we indicate the intervals. The first number we give is the low end of our lowest age group (18). The second number is the low end of our middle age group (36). The third number, as you probably guessed, is the low end of our highest age group. The last value should reflect our upper limit. In this case, I use the value `Inf`, which is short for 'infinity'. This essentially tells R that the last category can include any values higher than the previous number we entered.
- **include.lowest**: Putting `TRUE` here tells R that, in each interval, we want the lowest value to be included. If we don't do this, then our 'younger' age grouping would be 19 - 35 rather than 18 - 35. In other words, setting this to `TRUE` indicates a left-closed interval, and `FALSE` indicates a left-open interval.
- **right**: This is the input for specifying whether we want this to be a right-closed interval, and it takes a logical value (`TRUE` or `FALSE`). We want a right-open interval, so we will set this to `FALSE`.

- **labels**: R will actually default to formal interval notation for the names of each level of our new factor variable, so it would be [18,36), [36,50), [50, Inf). However, we can provide a character vector to specify custom labels for these new factor levels. In the event that you have an ordinal variable, make sure that you always specify these labels in ascending order. In this case, that would be Younger < Middle Age < Older.
- **ordered_result**: This takes a logical value and, as the name suggests, indicates whether we want the factor to be ordered or not. In our case, there is a clear progression in terms of age, so we need to set this to `TRUE`.

```
our_gss <- our_gss |>
  mutate(
    age_ord = cut(
      age,
      breaks = c(18, 36, 50, Inf),
      include.lowest = TRUE,
      right = FALSE,
      labels = c("Younger", "Middle Age", "Older"),
      ordered_result = TRUE
    )
  )
```

Go ahead and take a look at `our_gss`, and we should now have an additional variable column. I'll use the `str()` function here so we can confirm that our factor was ordered and added to the data frame.

```
str(our_gss)
```

```
'data.frame':	3544 obs. of  13 variables:
 $ year    : num  2022 2022 2022 2022 2022 ...
 $ id      : Factor w/ 4149 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ age     : num  72 80 57 23 62 27 20 47 31 72 ...
 $ race    : Factor w/ 3 levels "white","other",..: 1 1 1 1 1 1 2 1 1 NA ...
 $ sex     : Factor w/ 2 levels "female","male": 1 2 1 1 2 2 1 2 1 1 ...
 $ realrinc: num  40900 NA 18405 2250 NA ...
 $ partyid : Factor w/ 8 levels "strong democrat",..: 1 2 3 1 2 4 5 1 5 1 ...
 $ happy   : Ord.factor w/ 3 levels "not too happy"<..: 1 1 1 1 2 2 2 2 2 2 ...
 $ marital : Factor w/ 5 levels "divorced","married",..: 1 2 1 3 3 3 3 2 3 3 ...
 $ attend  : Factor w/ 9 levels "never","less than once a year",..: 3 3 1 5 3 1 2 2 1 8 ...
 $ cappun  : Factor w/ 2 levels "favor","oppose": 2 1 1 2 2 2 2 2 1 NA ...
 $ hr_wage : num  19.66 NA 8.85 1.08 NA ...
 $ age_ord : Ord.factor w/ 3 levels "Younger"<"Middle Age"<..: 3 3 3 1 3 1 1 2 1 3 ...
```

3.3 Restructuring Categorical Variables

Now, let's go ahead and tackle the other recoding situation I mentioned up at the top—collapsing the levels of categorical variables. We'll work with `partyid` here, but the logic of this process will apply to any categorical variable.

3.3.1 Collapsing categories

I encourage you to take a look at `partyid` in the GSS Data Explorer, like we did for 'age'..

Because `partyid` is a categorical variable, it has far fewer unique values than a ratio variable like `age`, so we can go ahead and take a look at all of its levels with the `count()` function.

> **Note**
>
> Observe that `count(our_gss, partyid)` basically provides the same information as `summary(our_gss$partyid)` when the variable is a factor. The only real difference is that `count()` organizes the info into a tidy data frame. In other words, you can use either function to take a look at factor variable like this.

```
count(our_gss, partyid)
```

```
                          partyid   n
1                 strong democrat 595
2 independent (neither, no response) 835
3          not very strong republican 361
4            not very strong democrat 451
5       independent, close to democrat 400
6                     other party 106
7     independent, close to republican 330
8               strong republican 431
9                            <NA>  35
```

Let's go ahead and collapse these into categories of "Democrat", "Independent", "Other Party", and "Republican". We'll use `mutate()` to make a new variable column called `partyid_recoded`. For the calculation of our new variable, we can use the `fct_collapse()` function. This function allows us to first give the name of a new factor level, and then we can give a character vector of the names of levels that want to be collapses into a single category.

So, to collapse our 2 Democrat levels into a single factor level, we would give the following input for `fct_collapse()`:

```
"Democrat" = c("strong democrat", "not very strong democrat")
```

And then repeat for each new factor level.

```
our_gss <- our_gss |>
  mutate(
    partyid_recoded=fct_collapse(partyid,
"Democrat" = c("strong democrat", "not very strong democrat"),
"Republican" = c("strong republican","not very strong republican"),
"Independent" = c("independent, close to democrat", "independent (neither, no response)", "i
"Other Party" = c("other party")
))
```

Now, let's take a look at our new variable.

```
count(our_gss, partyid_recoded)
```

```
  partyid_recoded    n
1        Democrat 1046
2     Independent 1565
3      Republican  792
4     Other Party  106
5            <NA>   35
```

3.3.2 Excluding levels

You may also want to work with an existing categorical variable, but only focus on certain values. In this case, we can create a recoded variable and simply code the levels we are uninterested in as `NA`.

We can use the `fct_recode()` function for this. For the first input, we will give it the factor-variable column that we want to recode. Then, we will let it know that we want to set the levels of "Other Party" and "Independent" to `NULL`. This will convert them to NAs and allow us to easily exclude them from our analyses. We will use `fct_recode()` within `mutate()` so we can create another recoded version of `partyid_recoded`. We'll call this one `dem_rep` to distinguish it from the original `partyid` and our first recoded version.

```
our_gss <- our_gss |>
  mutate(dem_rep = fct_recode(
    partyid_recoded,
    NULL = "Other Party",
    NULL = "Independent"))
```

Let's check to see that it worked.

```
count(our_gss, dem_rep)
```

```
     dem_rep    n
1   Democrat 1046
2 Republican  792
3       <NA> 1706
```

Now that we have gotten all this pre-processing stuff out of the way, let's go ahead and dig into some univariate analysis.

4 Univariate: Categorical

Before we start examining the relationships between multiple variables, we have to look under the hood of our variables individually and make sure we have a good sense of what they are like.

We want to know the frequency distribution of the response values, the measures of central tendency, and the dispersion. In other words, we want to see how many respondents chose each response value, which response values are most typical for each variable, and the extent to which the response values vary.

Luckily for us, R has plenty of tools for generating these quantitative summaries, as well as visualizing them in the form of barplots, histograms, and other common styles for displaying data.

This tutorial will cover each of these elements, and I will highlight any differences specific to certain levels of measurement along the way.

4.1 Frequency Distributions

Let's start with categorical variables. We will start with frequency distributions by way of frequency tables and bar plots. Then, we will talk about measures of tendency and observe when they are appropriate for categorical data.

4.1.1 Frequency tables

Frequency tables are the most effective way to report the frequency distributions for categorical variables. There are actually a ton of different ways to produce this kind of table in R, but we are going to use some convenient functions from the `janitor` package, so let's go ahead and load that in.

If you are on a lab computer, try to load it with `library()` first. If you get an error, go ahead and install it with `install.packages()` and then load it in.

While we're at it, we'll also bring in `tidyverse` and load in our GSS subset.

```r
library(janitor)
library(tidyverse)
load("our_gss.rda")
```

Let's go ahead and work with one of our `partyid` recodes from last time. We'll use `partyid_recoded`, where we retained all the party choices but collapsed the categories of degree (e.g. strong democrat + not very strong democrat = Democrat).

We can use a simple command with `tabyl()` from the `janitor` package to produce our frequency tables. Let's take a look at some output, and I'll go through a couple details and mention some further specifications we can make.

All we need to do is give our data frame as the first argument, and then our particular variable column as the second argument.

```r
# Without pipe operator
tabyl(our_gss, partyid_recoded)
```

```r
# With pipe operator
our_gss |>
  tabyl(partyid_recoded)
```

```
 partyid_recoded    n     percent valid_percent
        Democrat 1046 0.295146727    0.29809062
     Independent 1565 0.441591422    0.44599601
      Republican  792 0.223476298    0.22570533
     Other Party  106 0.029909707    0.03020804
            <NA>   35 0.009875847            NA
```

Now, we have a frequency table for `partyid_recoded`!

Each row is a level of `partyid_recoded`. The `n` column refers to the sample size, so, for example, 1,565 respondents indicated that they were Independents.

We then have two different columns for percentages. The `percent` column is the proportion of all respondents that chose each response value—while including NA values. Often, we want to get a proportion for only the sample of those who actually answered the question. This is what the `valid_percent` column indicates and is what we are often looking for.

Let's break it down a little for clarity.

The total number of respondents is the sum of the 'n' column.

1565 + 1046 + 792 + 106 + 35 = 3,544

Note that this is also the number of observations in our data frame

There are also 35 observations coded as NA, so the total number of *valid* respondents is 3,509.

Now, I'll give the calculations for the `Independent` row.

```r
# I'm going to define a few objects here for simplicity.

all_respondents <- 3544

all_minus_na <- (3544 - 35)

independents <- 1565
```

```r
# Percent column

independents / all_respondents
```

```
[1] 0.4415914
```

```r
# Valid percent column

independents / all_minus_na
```

```
[1] 0.445996
```

In this case, there really aren't that many NA values, so it does not change too much. In general, we will often discard NA categories in analyses we do for the course, but they give us important information about the survey and can be useful for some questions, so it's important to keep track of them nonetheless.

Now, let's clean a couple things up with the `adorn_()` functions.

Let's add a row for totals, turn the proportions into percentages, and round the decimals. We can use the pipe operator to quickly leverage a few different `adorn_()` functions.

```r
our_gss |>
  tabyl(partyid_recoded) |>
  adorn_totals(where = "row") |>
  adorn_pct_formatting() |>
  adorn_rounding(digits = 2)
```

```
 partyid_recoded    n percent valid_percent
        Democrat 1046   29.5%         29.8%
     Independent 1565   44.2%         44.6%
      Republican  792   22.3%         22.6%
     Other Party  106    3.0%          3.0%
            <NA>   35    1.0%             -
           Total 3544  100.0%        100.0%
```

We don't need to supply any input for `adorn_pct_formatting()`—which changes the proportions to percentages—but a couple of these functions do require inputs.

For `adorn_totals()`, we give "row" for the `where` input, which tells the function that we want the totals column to appear as a row.

For `adorn_rounding()`, we give an input for `digits`, which is the number of digits we want after the decimal point. We'll keep 2 and thereby round to the hundredth.

Lastly, if we want, we can also make the column names a little nicer. It may not be obvious because we've just been displaying this table rather than storing it as an object, but it's actually a data frame, so we can edit the column names directly.

Let's save it as an object so that's easier to see.

```r
our_table <- our_gss |>
  tabyl(partyid_recoded) |>
  adorn_totals(where = "row") |>
  adorn_pct_formatting() |>
  adorn_rounding(digits = 2)
```

We can then use the `colnames()` function. Let's supply our table data frame as an input to get a feel for it.

```r
colnames(our_table)
```

```
[1] "partyid_recoded" "n"               "percent"         "valid_percent"
```

It lists out the character vector of the 4 column names, and we can actually just write over them by supplying our desired column names as a character vector. If we assign that character vector to `colnames(our_table)`, it will overwrite the column names. Make sure the labels are in the same order as they appear in the data frame.

```
colnames(our_table) <- c(
  "Political Party",
  "Frequency",
  "Percent",
  "Valid Percent"
  )

our_table
```

```
 Political Party Frequency Percent Valid Percent
        Democrat      1046   29.5%         29.8%
     Independent      1565   44.2%         44.6%
      Republican       792   22.3%         22.6%
     Other Party       106    3.0%          3.0%
            <NA>        35    1.0%             -
           Total      3544  100.0%        100.0%
```

Excellent! We'll learn how to make this even nicer a bit later in the semester, but this looks pretty good for now.

Let's look at one last visualization technique for categorical variables.

4.1.2 Bar plots

For categorical data, bar plots are often the way to go. These typically have the response values on the x axis, and the count or percentage of each response value is tracked on the Y axis.

Much like with tables, we'll learn some fancier ways to do this later in the semester. But, in the meanwhile, base R has some great plotting functions that can be used very easily without much specification. They're not the prettiest out of the box, but they will do everything we need.

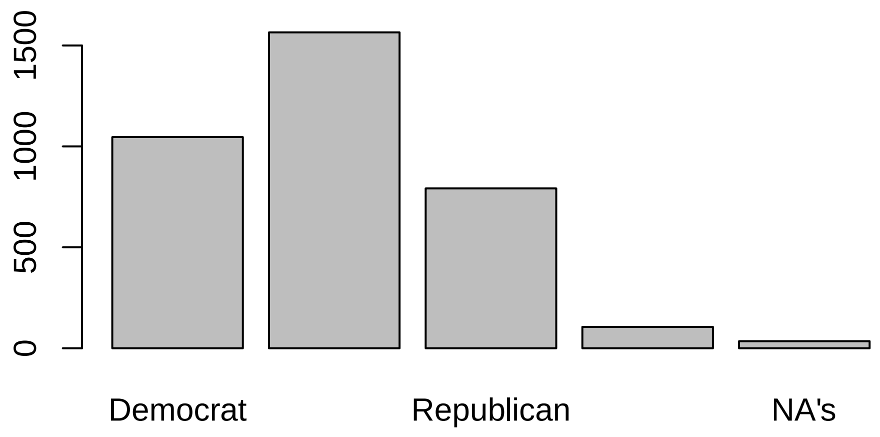
For `barplot()`, we just need the variable column that we are trying to plot. But, there's one thing to remember when you are trying to get a barplot this way—you have to provide the result of `summary(my_variable)` rather than the variable column directly.

I'll show you a couple ways to do that, which all do the exact same thing.

```
# No pipe operator; summary() nested inside barplot()

barplot(summary(our_gss$partyid_recoded))
```

```
# With pipe operator
our_gss$partyid_recoded |>
  summary() |>
  barplot()
```

```
# Make separate object for the summary() output

my_summary <- summary(our_gss$partyid_recoded)

barplot(my_summary)
```

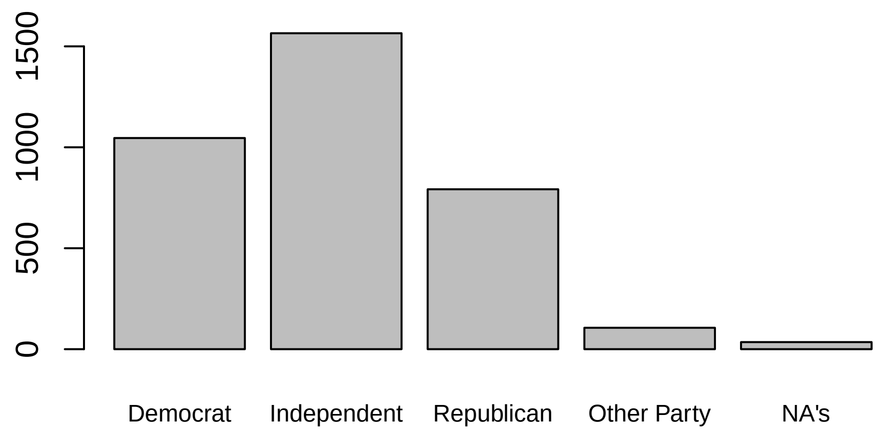


Now, as I mentioned, we won't spend too much time adjusting these plots for now, but there are a couple things we can do to make this a little better to look at for now.

For one, our response labels (Democrat, Republican, Independent, etc) Actually look pretty good in the display here, but it's not uncommon for these to get cut off if we have several response categories or the response names are especially long. So, I'll show you how to reduce the font size as one potential fix for that.

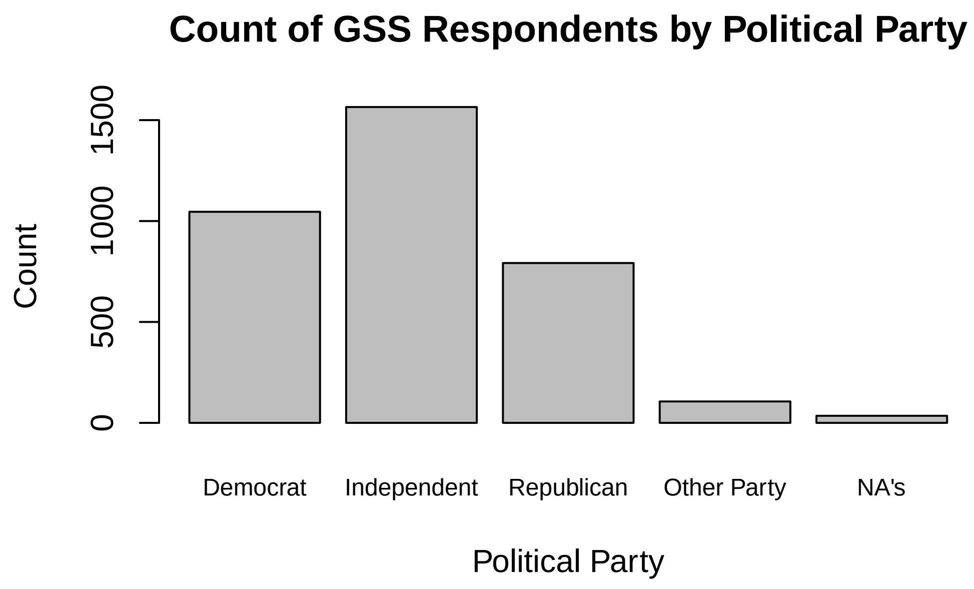
We can add the `cex.names` argument to `barplot()`. This argument takes a number, and the number should reflect an intended ratio of the default font size for our value labels on the x-axis. So, for example, I would enter '2' if I wanted the font to be twice as big. For our purposes, I want to reduce the font a bit, so we'll enter '.75' to reduce the font size by 1/4.

```r
our_gss$partyid_recoded |>
  summary() |>
  barplot(cex.names = .75)
```



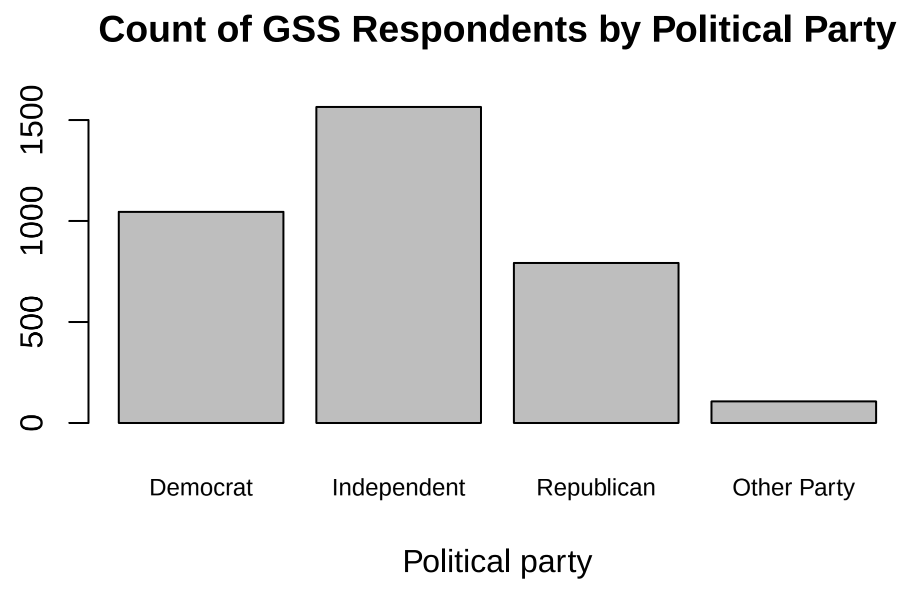
We can also add some simple descriptive information for the plot, such as a title and labels for the x and y axis. We can do so within `barplot()`

```r
our_gss$partyid_recoded |>
  summary() |>
  barplot(
    cex.names = .75,
    main = "Count of GSS Respondents by Political Party",
    xlab = "Political Party",
    ylab = "Count"
  )
```



As a final point, I'll also note that you can pass `partyid_recoded` into the function `na.omit()` before `summary()` if you ever want to produce a barplot that excludes NA responses. `na.omit()` will remove any NA values from the vector we provide it.

```
our_gss$partyid_recoded |>
  na.omit() |>
  summary() |>
  barplot(
    cex.names = .75,
    main = "Count of GSS Respondents by Political Party",
    xlab = "Political party"
  )
```



4.2 Measures of Central Tendency

Now, let's talk about measures of central tendency. Because there is no mathematically meaningful relationship between the different levels of a categorical variable, our options for reporting measures of central tendency are a little more limited.

The classic central tendency measures are the mode, the median, and the mean. For categorical variables, it is not ever meaningful to take the mean. As we saw back on our first day with R, you will get a warning message if you try to run `mean()` on a character or factor variable.

Our options are limited to the mode or the median, depending on the distribution of responses.

4.2.1 Nominal variables

In the case nominal variables, there is no semblance of meaningful arrangement to the categories. In this case, we can only report the mode.

Often times, we can just glean this from the frequency tables, or even just the `summary()` output. Let's stick with `partyid_recoded` as an example.

```r
summary(our_gss$partyid_recoded)
```

```
   Democrat Independent  Republican Other Party        NA's 
       1046        1565         792         106          35 
```

Remember that the mode is just the value with the most occurrences. So, the mode here is 'Independent'.

This is likely sufficient for the variables we will work with, but you may also want a method that's a little more exacting and does not depend on visually scanning for the mode.

R does not actually have a built-in `mode()` function like it does for `mean()` and `median()`. Well, it does have `mode()`, but it evaluates the 'mode' of the data, which, in this case, is actually the 'storage mode', or the data type. This is not particularly helpful for us, but we can use another convenient package called `DescTools`.

If you are on a lab computer, go ahead and try loading it with `library()` first. If it does not load, then install it with `install.packages("DescTools")` and then load it in.

```r
library(DescTools)
```

Then, we can just use the `Mode()` function within DescTools. Make sure you capitalize the 'M', as this is what distinguishes the DescTools function from the base R function.

Also, be sure to include `na.rm = TRUE`, as R will not know what to do with the NA values when calculating the mode, so we need to tell it to disregard them.

```r
Mode(our_gss$partyid_recoded, na.rm = TRUE)
```

```
[1] Independent
attr(,"freq")
[1] 1565
Levels: Democrat Independent Republican Other Party
```

So, this also gives us our mode, and is a bit more precise a way of doing so—especially when we have a bunch of possible response values.

Note that it also tells us how many responses were associated with that value (1565).

Now, let's talk about ordinal variables, where we have another consideration.

4.2.2 Ordinal variables

While ordinal variables are still categorical and thus unamenable to mathematical analysis, they do have a meaningful order. This means that it's possible for us to take the median value of an ordinal variable.

However, we can only do so when the variable is normally distributed. If there is significant skew in the distribution of responses, then we will need to report the mode for an ordinal variable.

Let's refresh our memory on distributions before we take some central tendency measures of ordinal variables in the GSS.

4.2.2.0.1 Distribution assumptions

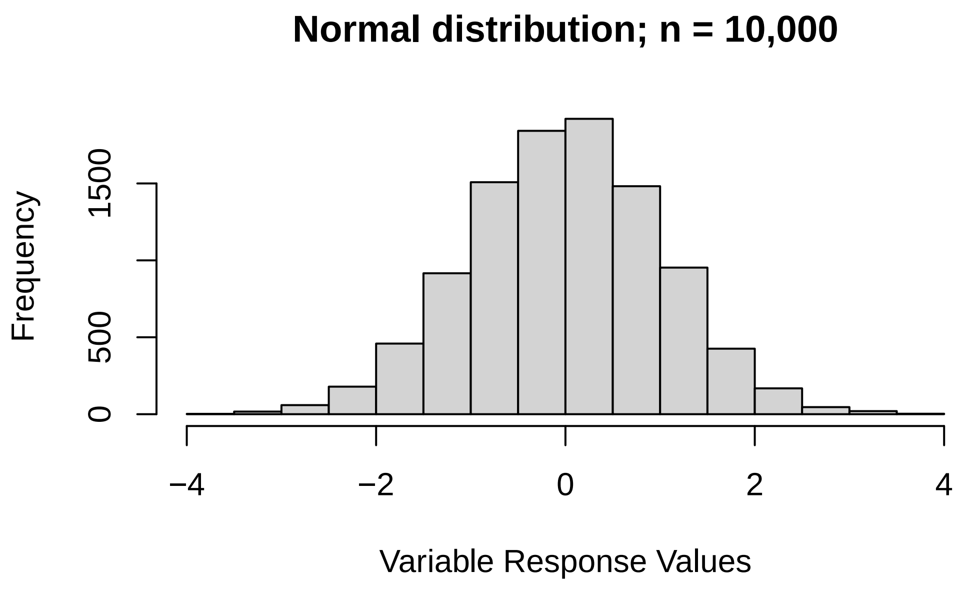
If the responses of an ordinal variable are normally distributed, we can take the median.

In the ideal form of the normal distribution, you have a mean of 0 and a standard deviation of 1, where roughly 68% of values are within 1 standard deviation in either direction of the mean, and roughly 95% of values are within 2 standard deviations in either direction.

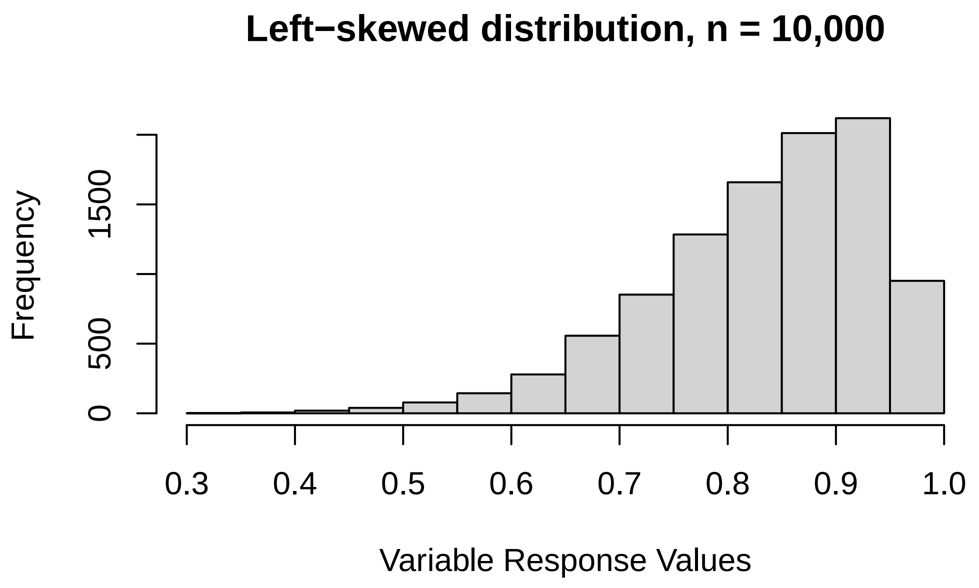
However, in practice, our distributions are unlikely to reflect the ideal normal distribution. But, if they are roughly normal, we can work with that.

I'll display a few distributions here as a reminder, starting with the classic normal distribution. If you produce a barplot and find the data looks generally like this—where the majority of variables are concentrated around the middle of the distribution—then you can report the median response rather than the mode.

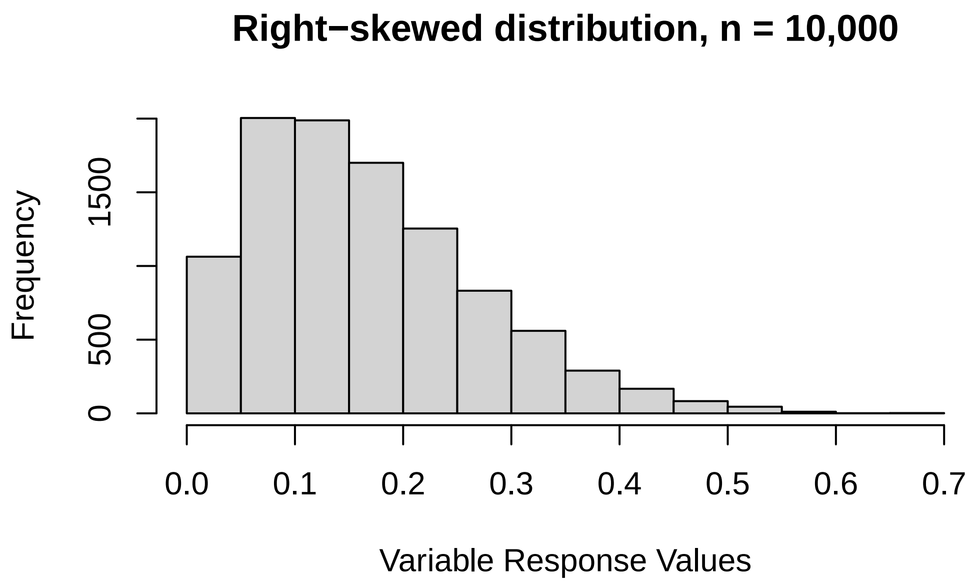
Here's the normal distribution



Here's a left-skewed distribution. This is when there's a prominent 'tail' on most of the left-hand side of the distribution. In other words, when most of the values are concentrated on the right-hand side of the distribution (the upper end), then we have a left-skew.



Lastly, here's a right-skewed distribution. As you can probably guess from the description of a left-skewed distribution, a right-skew occurs when we have a long tail through most of the right-hand side of the distribution. In other words, most values are concentrated on the left-hand side.

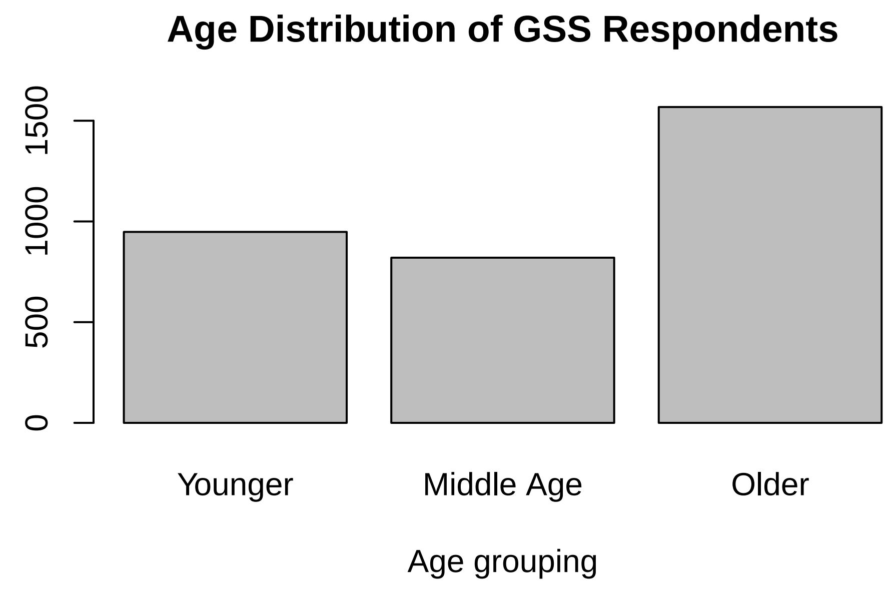


So, if the distribution of your variable looks (mostly) like the normal distribution, go ahead and take the median of any ordinal variable.

Let's go ahead and practice doing that with the `age_ord` variable we created last time.

First, we'll verify that the distribution allows for reporting the median.

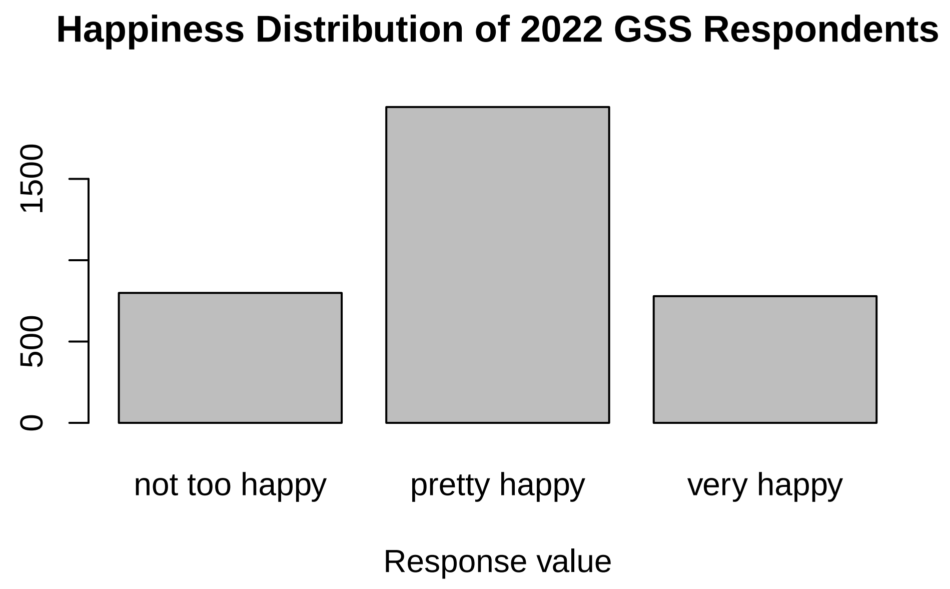
```
our_gss$age_ord |>
  na.omit() |>
  summary() |>
  barplot(
    main = "Age Distribution of GSS Respondents",
    xlab = "Age grouping"
  )
```



Well, hold on now. In the case of `age_ord`, it looks like we'd actually want to take the mode, because it's a bit left-skewed.

Let's try the `happy` variable. This is an ordinal variable in our subset that we haven't used yet. We'll see if its normally distributed.

```
our_gss$happy |>
 na.omit() |>
 summary() |>
 barplot(
   main = "Happiness Distribution of 2022 GSS Respondents",
   xlab = "Response value"
 )
```



Perfect. The distribution is exactly what we want to see. Most of the variables are concentrated at the midpoint of the distribution, with tails on either side. So, as it's roughly normally distributed, `happy` will serve as a good example of a case where taking the median of an ordinal variable is appropriate.

Now, the median response value is fairly apparent in this case, but I'll show you a method that you can use to calculate the median of any ordinal variable you might come across.

While R has a built-in `median()` function, it actually only works with numeric data. Fear not, however, as we can also use the `DescTools` again.

In order to distinguish itself from the default `median()` function, the DescTools package uses a capitalized 'M'. So, make sure you use `Median()` when you are working with an ordinal variable.

We also need to assure that NAs are not included for the calculation. As we have seen a few times, R will not know what to do with these if we don't tell it to exclude them.

```
Median(our_gss$happy, na.rm = TRUE)
```

```
[1] pretty happy
Levels: not too happy < pretty happy < very happy
```

Perfect! Looks like our median is 'pretty happy'.

And now we have all we need to report a univariate analysis of a categorical variable in this course:

- A frequency table
- A bar plot
- A median or mode

For now, it's just important that you can accurately identify the median or mode of a categorical value. We'll talk about ways to implement it in the tables a little later on when we focus on publication-ready tables and figures.

Now, let's talk about presenting a univariate analysis of numeric data.

5 Univariate: Numeric

While we also want to report on the frequency distribution and central tendency of numeric variables, there are some differences in the way we go about that relative to the techniques we just learned about for categorical variables.

Additionally, we will need to report on some measures of dispersion that can only be reported for numeric data, such as the standard deviation.

This section will walk us through the process of calculating and reporting necessary elements of a univariate analysis of numeric data.

5.1 Frequency Distributions

As we saw when we were recoding `age` earlier in the unit, a frequency table is not really appropriate for numeric data. When there are upwards of 71 different response values—as we have with `age`—a table would be both spatially unwieldy and difficult to interpret.

So, let's start with a style of plotting data that is essentially the barplot of numeric data.

First, let's go ahead and set up our environment by loading in our data and `tidyverse` per usual.

```r
library(tidyverse)
load("our_gss.rda")
```

5.1.1 Histograms

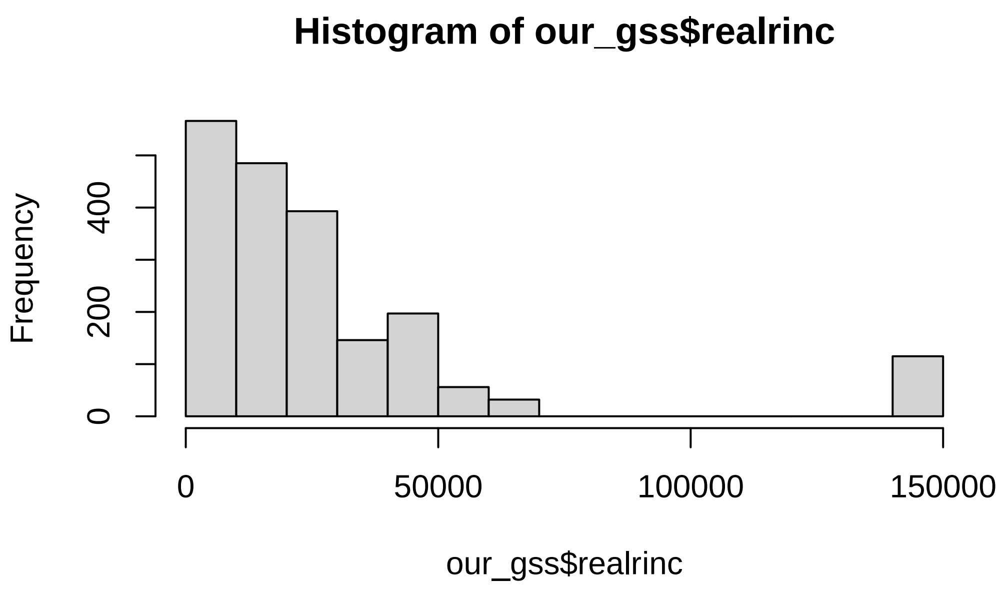
A histogram is the result of taking a numeric variable, slicing it up into equal, ordinal intervals, and then plotting the frequency of each interval.

There are plenty of other ways for plotting numeric data—some of which we will see later when we focus on visualization—but for now, this is a simple and effective way to get a sense of the distribution of a numeric variable. This is especially important when we need to decide on an appropriate measure of central tendency.

We can create these easily using the `hist()` function, which is in the same family of base R plotting functions as `barplot()`, so much of what we learned for that function will also apply with `hist()`. This function will automatically apply a commonly used method called Sturges's rule to divide our numeric variable into equal bins, so we do not have to specify anything in that regard.

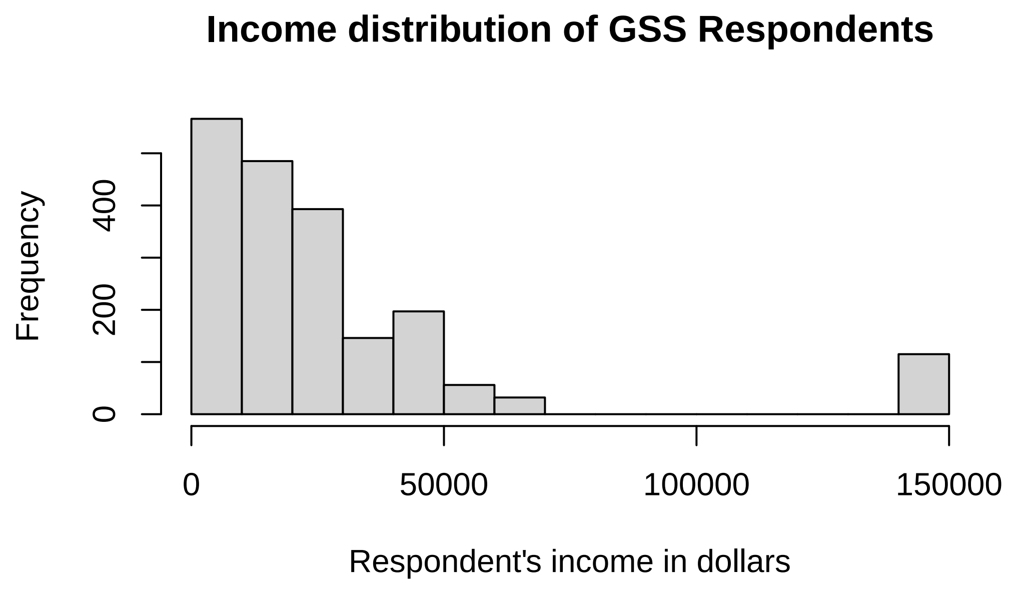
All we need to do is give the `hist()` function our numeric variable. Let's work with the `realrinc` variable for this example.

```
hist(our_gss$realrinc)
```



Much like we did with `barplot()`, we can clean this up a little by adding a title and a better label for the x axis

```
hist(
  our_gss$realrinc,
  main = "Income distribution of GSS Respondents",
  xlab = "Respondent's income in dollars",
  )
```



Now we have our histogram!

However, this visualization does suggest that we need to do some thinking about the appropriate measure of central tendency for `realrinc`.

5.2 Central Tendency

For numeric data, any of the 3 common measures of central tendency can be reported. However, the preferred measure for a given variable depends on its distribution.

The mean is the gold standard for numeric data, because it takes into account every single data point in its calculation, and is thus the most comprehensive index of central tendency.

However, the mean can be misleading in some cases. The distribution should be (basically) normal in order for you to report the mean, and you will want to report the median in most other cases.

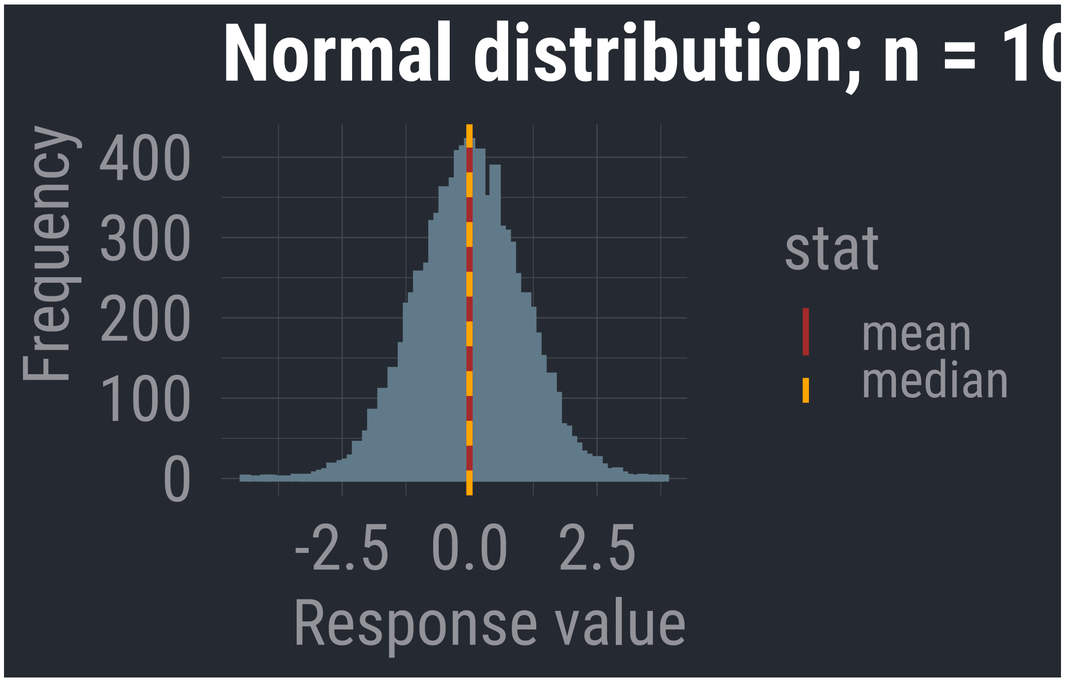
I will first cover the choice of mean or median by revisiting some basics of distributions, and then I will address the rare circumstance where the mode is appropriate.

5.2.1 Mean or Median?

In most cases, we will be making a decision between the mean and the median. When the distribution is normal, you should report the mean—otherwise, report the median. We'll see one exception to this later, but that's generally our task.

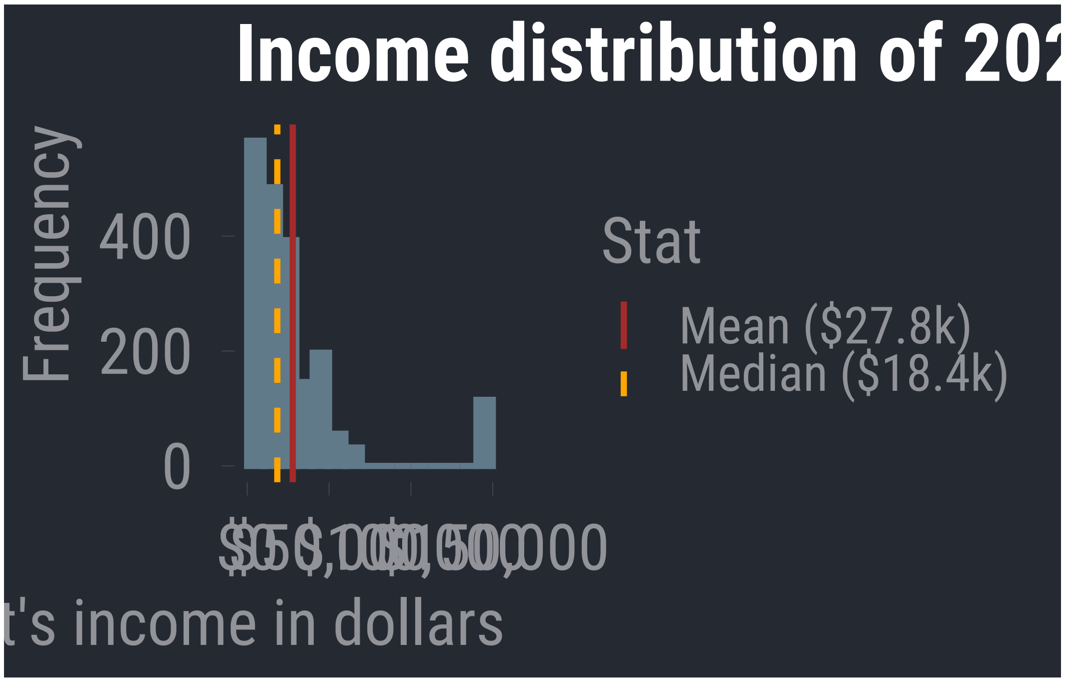
We saw some typical distributions in our last section on uivariate analysis of categorical data. We'll return to some discussion of those basic distribution types but add a little more context for numeric data that will help us see the impact of making the wrong decision.

I'm going to display a normal distribution, but this time, I'll draw a vertical line indicating the location of the mean and median.



In this case, they appear to be overlapping. In fact, in a completely normal distribution, all three measures of central tendency will be equal. Though, remember that real-world data will rarely exhibit perfect normality. As long as it's approximately normal, we can default to the mean because it accounts for every data point in the variable's distribution.

Now, let's look back at `realrinc`.



In the case of `realrinc`, we definitely do not have much normality going on. For one, there are some apparent outlier cases. Much of our data seems to be concentrated within about $0 - $75,000, but then we have a bunch of values at roughly double the maximum of that narrower range. And even within the range where most of our responses are clustered, we have some clear right-skew.

In general, it's ideal that the mean includes every data point, but this also makes it susceptible to non-normal distributions. As we can see, the mean will be pulled in the direction of the skew.

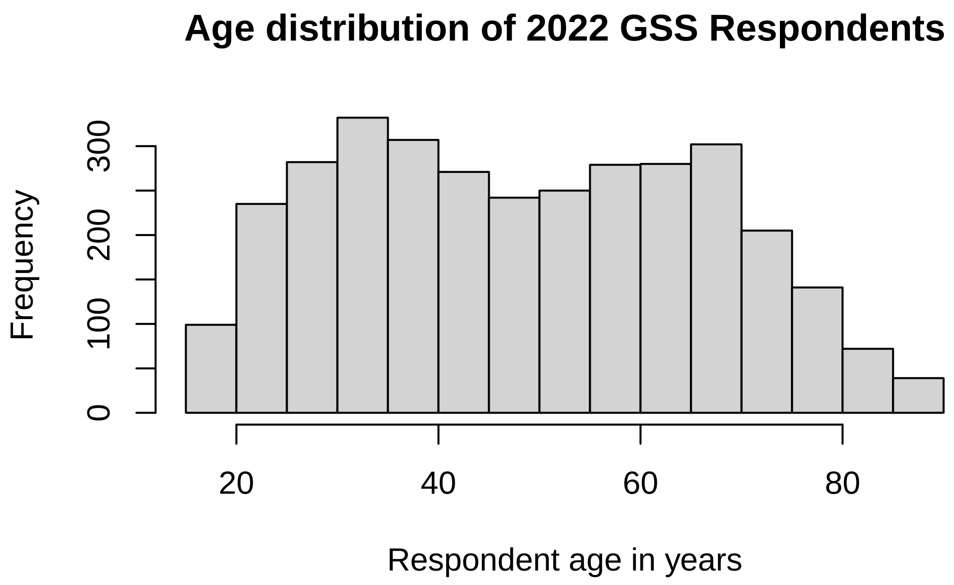
In this case, it becomes an advantage that the median does not take every data point into account. This makes it more resistant to skew, so, in any case where your numeric variable contains clear outliers and/or exhibits notable skew in either direction, you should report the median rather than the mean.

5.2.2 The Mode

By and large, the mode is more appropriate for categorical data. But there are some circumstances where it makes sense to report the mode for numeric data.

Let's take a look at the original `age` variable to see an example.

```r
hist(
  our_gss$age,
  main = "Age distribution of 2022 GSS Respondents",
  xlab = "Respondent age in years"
)
```



This isn't quite a normal distribution, but it's not exactly skewed in either direction either. And there are no apparent outliers.

What we have here is a **bimodal** distribution.

This is what happens when we have multiple peaks in a distribution—two, in this case. We have seen several different distributions so far, but they all had only one peak in the distribution. They were differentiated on the basis of that single peak's location in the distribution. In the case of a multimodal distribution, we want to report on any notable peak.

This may be a peculiarity of the 2022 survey wave, because we wouldn't necessarily expect this, but it looks like there are two distinct central tendencies for `age`. In this case, we want to capture the two distinct peaks, meaning we need to calculate two modes.

Thankfully, there's a convenient function for this, but we need to install a new package, so let's install and load in `multimode`.

```r
install.packages("multimode")
library(multimode)
```

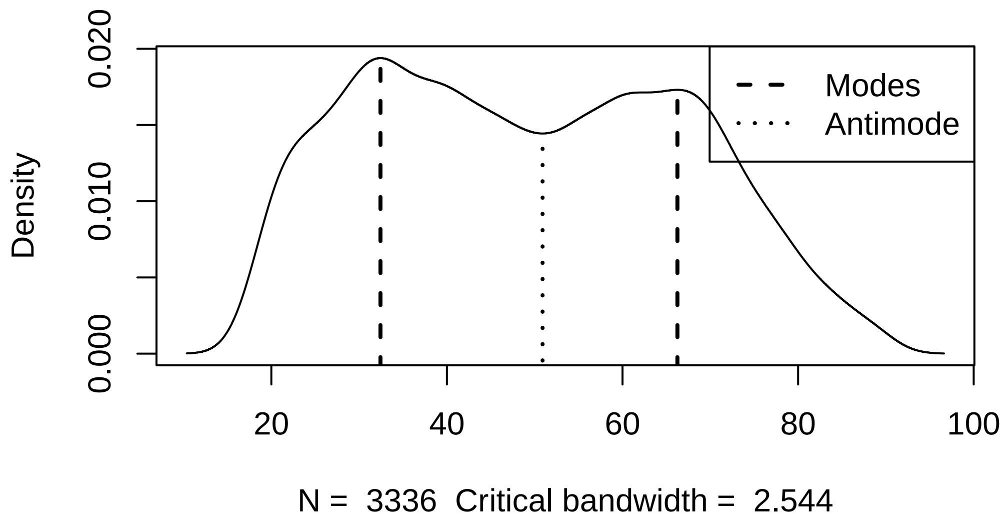
Then, we just need to provide a few arguments to the `locmodes()` function.

First, we give the variable for which we want to calculate multiple modes.

Then we give an input for `mod0`, which asks us how many modes the function should be looking for. This should be based on visual inspection of the data. We have two peaks in our distribution, so we will enter '2' for this input.

Lastly, we will set `display` to 'TRUE', which will show us a plot of the estimated modes superimposed onto the frequency distribution of the variable. This will let us evaluate whether the modes estimated by the function are plausible. They should match up with the visible peaks in the frequency distribution.

```r
locmodes(
  our_gss$age,
  mod0 = 2,
  display = TRUE
)
```



```
Estimated location
Modes:  32.4301  66.25356
Antimode: 50.89506

Estimated value of the density
Modes: 0.01938966  0.01730832
Antimode: 0.01444287

Critical bandwidth: 2.543659
```

We only really need to pay attention to the modes that it gives us (32.43 and 66.25), but it looks like this function spotted these values pretty effectively.

For a bimodal distribution like this, we can report both modes for the measure of central tendency. I don't expect we will run into too much of this, but go ahead and deal with it like this in the event that you do.

5.3 Dispersion

Now, let's talk about dispersion. This is the last major element of a univariate analysis of numeric data. Actually calculating this information is quite simple in R, so we will practice with some of our GSS variables and then talk about putting all of this information together.

5.3.1 Range

This is perhaps the simplest measure of dispersion and comprises the minimum and maximum values of the distribution.

R has a built in `range()` function, so we can simply provide one of our variables as the input. We'll work with `realrinc` again. As we have done before, we will also need to provide `na.rm = TRUE`, as this variable column includes NAs and will confuse R otherwise.

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

```
[1]    204.5 141848.3
```

At this point, I'll also offer a reminder about the `summary()` function, which we can use to see a variety of information about the dispersion (in an addition to the central tendency).

```r
summary(our_gss$realrinc)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's 
   204.5   8691.2  18405.0  27835.3  33742.5 141848.3     1554 
```

For a numeric variable, `summary()` will show us the minimum & maximum, as well as the mean & median, and the 1st & 3rd quartiles.

We can actually think of most of these elements in terms of percentiles.

- The minimum value is the 0th percentile of the distribution
- The 1st quartile is the 25th percentile.
- The median is the 50th percentile
- The 3rd quartile is the 75th percentile
- And the maximum is the 100th percentile.

`summary()` is great for quickly assessing some of these descriptive statistics, but it's a little less convenient for exporting this information into something like a table for our own univariate analysis.

So, I'll also highlight the `min()` and `max()` functions, which will come in hand for us shortly. They work simply enough—we just need to provide our variable column, and they will output the minimum and maximum, respectively.

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

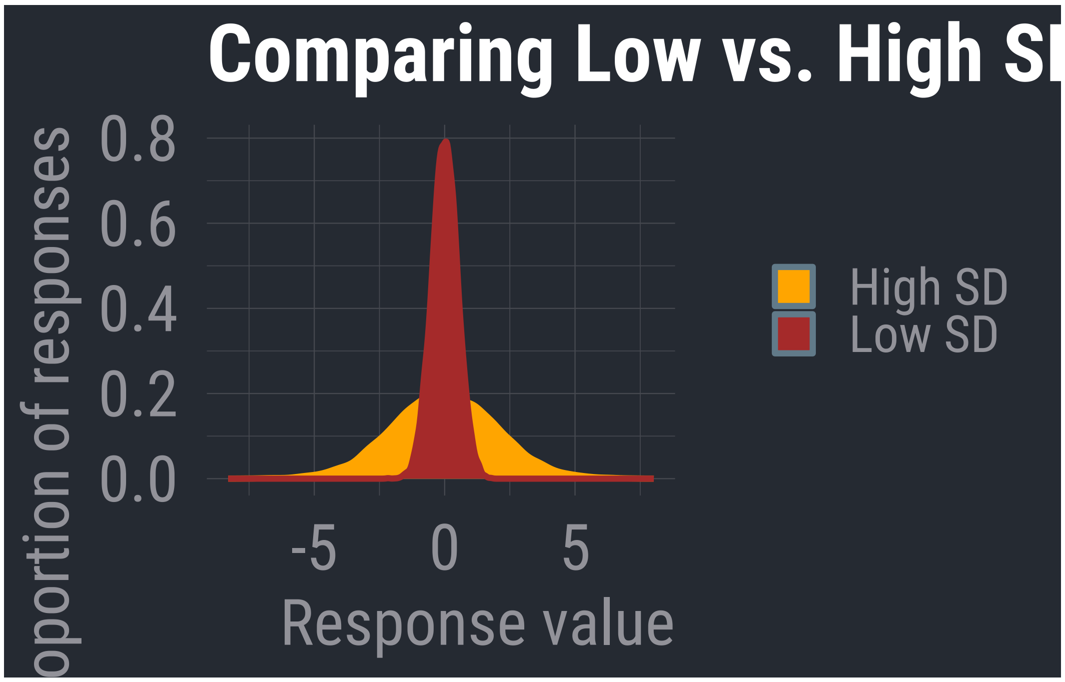
```
[1] 204.5
```

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

```
[1] 141848.3
```

5.3.2 Standard Deviation

This is one of the more commonly reported dispersion metrics for numeric data. The standard deviation is a measure of how much the average value varies from the mean. In plainer terms, it's a measure that tells us how spread out the distribution is.



We can grab the standard deviation quite easily with the `sd()` function

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

```
[1] 31962.34
```

Now, let's talk about putting all this together in our reporting of a univariate analysis for numeric data.

5.4 Putting It All Together

Now that we have worked through how to calculate all of these key statistics with R, let's revisit some of the first `tidyverse` functions we learned about back on the first day.

We can use a combination of `select()` and `summarize` to quickly compile several bits of important information, such as the range, the central tendency, and the standard deviation.

Let's do this for `realrinc`.

```r
our_gss |>
  select(realrinc) |>
  summarize(
    "Minimum" = min(realrinc, na.rm = TRUE),
    "Median" = median(realrinc, na.rm = TRUE),
    "Maximum" = max(realrinc, na.rm = TRUE),
    "SD" = sd(realrinc, na.rm = TRUE)
  )
```

```
  Minimum Median  Maximum       SD
1   204.5  18405 141848.3 31962.34
```

Note

You might notice that there's an attribute for each column that looks like `<dbl>`. This is short for 'double', which is itself shorthand for 'double-precision floating-point format'. This is in reference to the underlying data type that R uses to store continuous numeric values. You really don't need to know anything more about double-precision floating point than that for our purposes, but I mention it here because you might run into or 'double' in R. When you do, just think 'numeric data'.

If you have a numeric variable where the mean is more appropriate, you can just swap `median()` with `mean()` in the code template above. Be sure to also change the column name as well.

And that should do us! For any univariate analysis you report in this course, you will just need to produce

- a histogram displaying the variable's frequency distribution
- a table like the one above (with the appropriate measure of central tendency).

Part III

Day 3: Bivariate Analysis with the GSS

6 Bivariate: Categorical

Now that we know how to peek under the hood of our variables individually, it's time to start looking at them in combination.

We will start with bivariate analysis, or the association between two variables.

Let's begin with the procedure for categorical variables. We will start with the basic display of two categorical variables and then get some practice with the chi-square test, which is a common statistical test that assumes both variables are categorical.

We will learn to recognize associations in the contingency table, and then we'll use the chi-square test to estimate the likelihood that the relationship we find is genuine or merely an artifact of random chance.

6.1 Contingency Tables

Also known as cross-tabulations—or crosstabs for short—contingency tables are quite similar to frequency tables. Rather than showing the distribution of a single variable, they show us the distribution of our dependent variable across the levels of our independent variable. In technical terms, we can call this the conjoint distribution of two variables.

Let's dig in with an example. We'll keep working with our GSS subset, so let's go ahead and set up our workspace by loading in our packages & data. We will need both `tidyverse` and `janitor` for this tutorial.

```
library(tidyverse)
library(janitor)
load("our_gss.rda")
```

6.1.1 Creating our table

We'll set up a simple hypothesis to illustrate. Let's say that, based on theory, we predict that a person's likelihood of supporting the death penalty will vary according to their political affiliation.

Now, there's enough literature on capital punishment and political affiliation that we would probably be justified in making a more specific hypothesis about the predicted direction of the association, but let's just say we simply predict that rates of approval for the death penalty will vary across categories of Democrat and Republican.

Hypotheses are where we turn our research questions into very specific expectations about the data. If certain theoretical expectations are true, then we should find evidence in the data for the relationships suggested by theory. This is where we predict what the distribution of variables should look like in the event that these theoretical expectations are true.

And remember that our hypotheses are generally going to be tested against a null hypothesis, which assumes that whatever relationship you predict is not the case.

Null hypothesis:

$H_0 = There\ will\ be\ no\ difference\ in\ rates\ of\ death\ penalty\ approval\ among\ Democrats\ and\ Republicans$

Alternative hypothesis:

$$H_1 = Democrats\ and\ Republicans\ will\ have\ different\ rates\ of\ death\ penalty\ approval$$

In sum, the alternative hypothesis is that the proportion of approval for the death penalty will be different across political parties. The null hypothesis is that there will be no such difference. Though the peculiarities of many statistical tests differ, this general idea is true for a great number of statistical procedures. You will estimate the distribution of values that would be the case under the null hypothesis and then evaluate the statistical significance of the extent to which your data deviates from a null distribution.

We'll work with the `dem_rep` variable that we created when learning about variable re-coding, as well as the `cappun` variable.

Thankfully, we can use a familiar function in `tabyl()`, which we used to create our frequency table. We can simply add a second variable to produce a crosstab, though we will need to specify a couple of things so that the variables are properly formatted in terms of rows and columns.

If you did not save `our_gss.rda` after creating `dem_rep` in a previous exercise, you can go ahead and recreate it with the following code:

```
our_gss <- our_gss |>
  mutate(
    partyid_recoded=fct_collapse(partyid,
"Democrat" = c("strong democrat", "not very strong democrat"),
"Republican" = c("strong republican","not very strong republican"),
"Independent" = c("independent, close to democrat", "independent (neither, no response)", "i
"Other Party" = c("other party")
))

our_gss <- our_gss |>
  mutate(dem_rep = fct_recode(
    partyid_recoded,
    NULL="Other Party",
    NULL="Independent"))
```

And then we can make a simple cross-tab like so:

```
our_gss |>
  drop_na(dem_rep, cappun) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep
  )
```

```
 cappun Democrat Republican
  favor      426        633
 oppose      569        121
```

Warning

Whenever you make a two-way crosstab like this, make sure that you always give your dependent variable as `var1` and your independent variable as `var2`. This will make sure your dependent variable appears as rows and your independent variable as columns, which makes everything much easier to parse for our purposes.

For each cell, the count of that particular category intersection is displayed. So, for example, row 1 tells us that of those who indicated that they favor the death penalty, 426 are Democrats and 633 are Republicans.

Now, let's clean it up a little bit with the `adorn_()` functions like we did before.

First, let's add a `totals` row at the bottom, which will tell us how many respondents are in each category of the independent variable. Note that we always need to add `adorn_totals()` before all other `adorn_()` functions. It requires the raw counts as input, and these will be transformed by most other `adorn_()` functions, so we will get an error if we try to run `adorn_totals()` afterwards.

We'll give one input to `adorn_totals()`, which will be `where = "row"`, to indicate that we want the totals to appear as a row.

```r
our_gss |>
  drop_na(cappun, dem_rep) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep) |>
  adorn_totals(where = "row")
```

```
 cappun Democrat Republican
  favor      426        633
 oppose      569        121
  Total      995        754
```

Now, let's convert counts to proportions. Somewhat misleadingly, this is done with the `adorn_percentages()` function, and then we can add the `adorn_pct_formatting()` function to turn them into percentages proper.

```r
our_gss |>
  drop_na(cappun, dem_rep) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep) |>
  adorn_totals(where = "row") |>
  adorn_percentages() |>
  adorn_pct_formatting()
```

```
 cappun Democrat Republican
  favor    40.2%      59.8%
 oppose    82.5%      17.5%
  Total    56.9%      43.1%
```

While percentages help give important context, it's important that we still complement the percentages with the raw numbers. Thankfully, we can add those back in alongside the percentages with `adorn_ns()`

```r
our_gss |>
  drop_na(cappun, dem_rep) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep) |>
  adorn_totals(where = "row") |>
  adorn_percentages() |>
  adorn_pct_formatting() |>
  adorn_ns(position = "front")
```

```
 cappun    Democrat  Republican
  favor 426 (40.2%) 633 (59.8%)
 oppose 569 (82.5%) 121 (17.5%)
  Total 995 (56.9%) 754 (43.1%)
```

Great. For our final adornment, let's add some better labels for the variables. We'll use `adorn_title()` for this, and we'll supply `placement = top` to let the function know we want the labels on top of each variable column. Then we can provide custom label names with `row_name =` and `col_name =`.

```
our_gss |>
  drop_na(cappun, dem_rep) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep) |>
  adorn_totals(where = "row") |>
  adorn_percentages() |>
  adorn_pct_formatting() |>
  adorn_ns(position = "rear") |>
  adorn_title(
    placement = "top",
    row_name = "Death Penalty Attitude",
    col_name = "Political Party")
```

```
                        Political Party
 Death Penalty Attitude        Democrat  Republican
                  favor     40.2% (426) 59.8                            (633)
                 oppose     82.5% (569) 17.5                            (121)
                  Total     56.9% (995) 43.1                            (754)
```

Great! Like our other visualizations, we'll clean this up a bit more later on, but this will do for now.

6.1.2 Reading our table

Now that have our two-way crosstab, let's go ahead and think about what this means for our hypothesis.

	Political Party	
Death Penalty Attitude	Democrat	Republican
favor	40.2% (426)	59.8% (633)
oppose	82.5% (569)	17.5% (121)
Total	56.9% (995)	43.1% (754)

Because our dependent variable makes up the rows, we can simply read across each row to get a sense of the patterns.

Reading across the 'approve' row, we can see that about 40% of those who favor the death penalty are Democrats, whereas ~60% of those who favor the death penalty are Republicans. The contrast is sharper for those who oppose the death penalty, where 82.5% are Democrats and 17.5% are Republicans. The imbalance there is a consequence of the recoding strategy, as the original `partyid` variable also includes Independents and an Other Party category. Nonetheless, a general pattern clearly emerges: Democrats are more likely to oppose the death penalty than Republicans.

But how can we feel good about whether this association is likely to be reflective of something real, and not just an artifact of our particular sample or simply random chance? For this, we need to employ a statistical test.

Specifically, we will leverage the Chi-square test. This is used when both of the variables we're interested in are categorical.

6.2 Chi-square Test

Let's have a little refresher on what goes into the Chi-square test and the corresponding Chi-square value, which serves as a key indicator of the statistical significance for our findings.

6.2.1 How the test works

Given the sample size of our data, the number of variables we are comparing, and the number of categories within those variables, we can calculate an **expected** distribution of those variables for the case where the variables are completely unrelated. This would give us a contingency table where our sample size is distributed across cells in a random fashion.

The Chi-square test involves estimating the extent to which the cell counts we actually **observe** in our contingency table are significantly different from the cell counts we **expect** if the variables are unrelated.

$$\chi^2 = \sum \frac{(o-e)^2}{e}$$

This is the formal equation for the Chi-square test (slightly modified for interpretive simplicity). You will almost never need to calculate this by hand, but I will explain what the equation means in order to help us get a conceptual understanding of the test.

On the left-hand side of the equation, we have χ^2, which is the symbol for Chi-square—Chi being in reference to the letter of the Greek alphabet.

On the right-hand side, we have $\frac{(o-e)^2}{e}$, which is the heart of the calculation

The $\sum$ operator tells us that we will be taking a sum. So, we need to calculate $\frac{(o-e)^2}{e}$ for each cell in the contingency table, and then add all these values together.

- o: observed valued
- e: expected value

For each cell, we take the count value that we find in our contingency table, subtract the count we would expect if there's no relationship between the variables, square that difference, and then divide by the expected value.

So, the observed value is easy enough—it's just the value we actually see in our contingency table.

The observed value of the cell in the first row of the first column is 426.

Now, this raises the question of how exactly we calculate the expected values.

$$ExpectedCount = \frac{(RowTotal) \cdot (ColumnTotal)}{GrandTotal}$$

In other words, for each cell, you take the sum of the values in that row, then take the sum of the values in that column, and then multiply these together. Then, you divide that value by the grand total.

Let's work through an example from our table.

	Political Party		
Death Penalty Attitude	Democrat	Republican	Total
favor	40.2% (426)	59.8% (633)	100.0% (1,059)
oppose	82.5% (569)	17.5% (121)	100.0% (690)

	Political Party		
Total	56.9% (995)	43.1% (754)	100.0% (1,749)

The expected count for Democrats who favor the death penalty would be:

$$\frac{(426 + 633) \cdot (426 + 569)}{1749}$$

```
[1] 602.4614
```

You can get a sense right here that we may indeed be picking up on something in our crosstab. Democrats appear to oppose the death penalty more so than Republicans, based on the raw frequencies. Now we can see that our observed value for Democrats who favor the death penalty (426) is quite a bit lower than what we would expect if that variable were unrelated to political affiliation (602.46). This is promising, but there's a couple more things we need to do.

This calculation for the expected counts is actually downstream of a principle from probability theory that describes the joint probability of independent events:

$$P(A\&B) = P(A) \cdot P(B)$$

What this means is that, if two variables are independent, then the probability of any combination of these variables' response values occurring simultaneously is equivalent to the product of each value's individual probability.

Let's put this in context of our data:

$$P(Democrat\&Favor) = P(Democrat) \cdot P(Favor)$$

If the two variables are unrelated, we expect that the probability of both being a democrat and favoring the death penalty is equal to the probability of being a democrat multiplied by the probability of favoring the death penalty.

```
# Total of our sample
grand_total <- 1749

# Number of democrats
num_dems <- 995

# Number of people who favor the death penalty
num_favs <- 1059
```

```
# Probability of being a democrat
dem_prob <- num_dems/grand_total

# Probability of favoring the death penalty
fav_prob <- num_favs/grand_total

# Expected probability of Favor/Democrat
exp_dem_fav <- dem_prob * fav_prob

# Expected count of Favor/Democrat
exp_dem_fav * grand_total
```

```
[1] 602.4614
```

There it is worked out in a little more detail. The $ExpectedCount = \frac{(RowTotal)\cdot(ColumnTotal)}{GrandTotal}$ equation is written to output the counts directly, but hopefully that helps clarify what exactly the concept of 'expected' counts is getting at.

Now, all we'd need to do is plug these expected values into this formula:

$$\chi^2 = \sum \frac{(o-e)^2}{e}$$

I'll calculate the Chi-square value of the cell for Democrats who favor the death penalty as an example.

```
exp_val <- 602.4614

obs_val <- 426

chi_numerator <- (obs_val - exp_val)^2

chi_denominator <- exp_val

chi_numerator/chi_denominator
```

```
[1] 51.68568
```

Then we do this for the three other cells in our table, add these 4 values together, and voila—we have our Chi-square statistic.

I won't go through the trouble here of manually calculating all the others, but know that our value is 302.18.

Then you can use a look-up table for the corresponding p-values of Chi-squared statistics. You will need to know your degrees of freedom, which, for the Chi-square test, is:

$$df = (RowNumbers - 1) \cdot (ColumnNumbers - 1)$$

So, for our 2x2 contingency table, this would be:

```r
total_rows <- 2
total_columns <- 2

# Degrees of freedom
(total_rows - 1) * (total_columns - 1)
```

```
[1] 1
```

Here's an example of a Chi-square look-up table from the University of Sussex.

All you have to do is take your Chi-squared value & your degrees of freedom and then locate the column corresponding to the alpha level you are hoping for. In most cases, this will be an alpha of 0.05, which is standard across the social sciences. We want a p-value below this alpha level for statistical significance, which means that we want less than a 5% chance of a false positive (i.e. returning a statistically significant result when there actually is no association).

With an alpha of 0.05 and 1 degree of freedom, we need our Chi-squared value to be greater than 3.84 in order to claim statistical significance. In our case, our value of 302.18 is well above this, so we do indeed have a statistically significant finding.

Finally, let's learn a much simpler way to do all of this.

6.2.2 The chisq.test() function

I went about working through that example to make sure we all have a good idea of what's going on underneath the hood of the Chi-squared test.

Thankfully for all of us, you will rarely if ever need to actually go about manually calculating all of this stuff. As usual, R has a nifty function that we can run directly on our contingency table—the `chisq.test()` function—which will automatically run all of these calculations and output a corresponding p-value. We just need to provide the function a contingency table as input, and the output of `tabyl()` will work for us here. I'm simply going to borrow some code from above and then add one more pipe operator to pass the table into the `chisq.test()` function.

Caution

Note here that I am running this on a bare-bones table without any of the `adorn_()` functions that we have used to dress up our `tabyl()` output. Because those functions add a bunch more information beyond the simple frequency counts, this will throw off `chisq.test()`, which will not know what to do with the extra info (e.g. the percents & totals row). Make sure you leave out any adornments when running the Chi-square test.

```r
our_gss |>
  drop_na(cappun, dem_rep) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep) |>
  chisq.test()
```

```

	Pearson's Chi-squared test with Yates' continuity correction

data:  tabyl(drop_na(our_gss, cappun, dem_rep), var1 = cappun, var2 = dem_rep)
X-squared = 302.18, df = 1, p-value < 2.2e-16
```

This gives us our Chi-squared value (302.18), our degrees of freedom (1), and a corresponding p-value. Note that this is in scientific notation, but it's an extraordinarily small number—far below our 0.05 alpha threshold. I'll display it in full so you can see for yourself

```r
format(2.2e-16, scientific = FALSE)
```

```
[1] "0.00000000000000022"
```

So, 2.2e-16 indicates the number 22 following 16 zeros (including one before the decimal place).

In the case that your p-value is this small, it's common convention to simply report that it is < 0.001.

One other nifty thing we can do with `chisq.test()` is store it as an object.

```r
our_chisq <- our_gss |>
  drop_na(cappun, dem_rep) |>
  tabyl(
    var1 = cappun,
    var2 = dem_rep) |>
  chisq.test()
```

If we look inside this object, we'll see a bunch of the information used to calculate these various statistics. Notably, it has the table of both observed and expected counts, so we can easily compare the two.

```
our_chisq$expected
```

```
 cappun Democrat Republican
  favor 602.4614   456.5386
 oppose 392.5386   297.4614
```

```
our_chisq$observed
```

```
 cappun Democrat Republican
  favor      426        633
 oppose      569        121
```

As you can see, far fewer Democrats favor the death penalty than would be expected in the case that this is independent from one's political affiliation. Similarly, many more Republicans favor the death penalty than we would expect in the conjoint distribution of two independent variables. Now that we have assessed this difference with the Chi-square test, we can feel confident that the association originally suggested by our reading of the contingency tables—that Democrats are more likely to oppose the death penalty than Republicans—is statistically significant.

So, how can we interpret this in terms of our hypotheses?

Null hypothesis:

$$H_0 = There\ will\ be\ no\ difference\ in\ rates\ of\ death\ penalty\ approval\ among\ Democrats\ and\ Republicans$$

Alternative hypothesis:

$$H_1 = Democrats\ and\ Republicans\ will\ have\ different\ rates\ of\ death\ penalty\ approval$$

With a Chi-squared value of 302.18 at 1 degree of freedom, we can reject the null hypotheses that these two variables are unrelated. Instead, we find support for our hypothesis that political affiliation is associated with death-penalty attitudes. Our Chi-square value is associated with a p-value of <0.001, which is statistically significant at an alpha threshold of 0.05. This suggests that it is exceedingly unlikely the relationship we observe in our cross-tabulation is the result of random chance or peculiarities of our sample.

The conjoint distribution of `cappun` and `dem_rep` shows us that Democrats are more likely to oppose the death penalty than Republicans. Of those who oppose the death penalty, 82.5% are Democrats and only 17.5% are Republicans. On the other hand, of those who favor the death penalty, only 40.2% are Democrats, whereas 59.8% are Republicans.

Part IV

Day 4: Making Better Visualizations

7 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`.

7.1 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.

7.2 Thirteen Data Sets

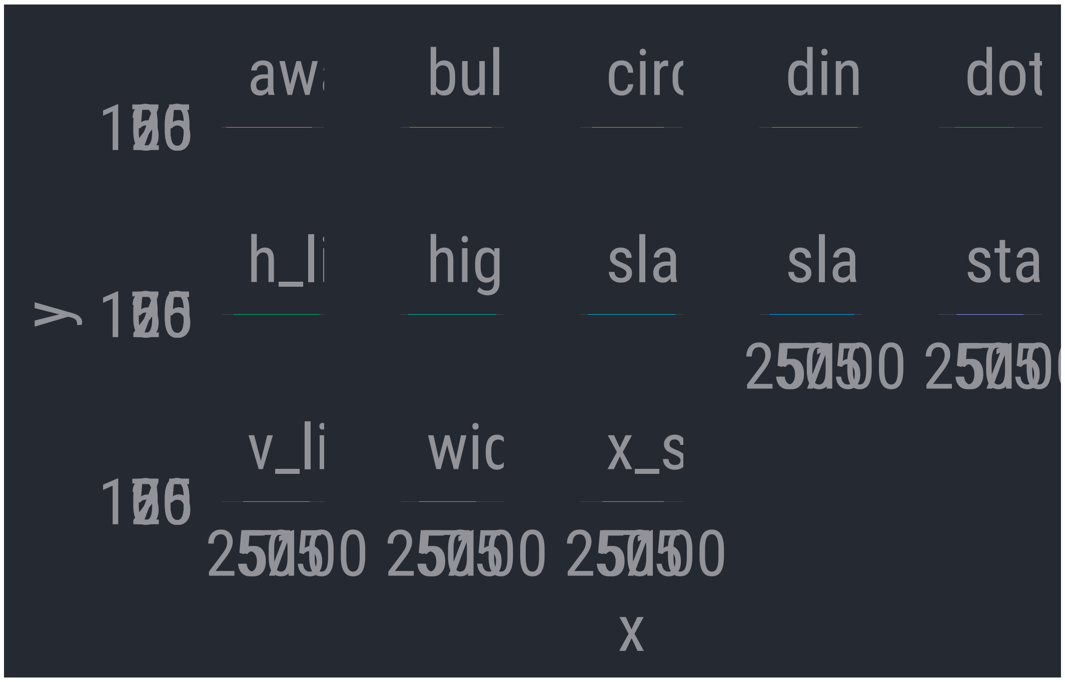
Let's take a look at a collection of thirteen different data sets. Each data set has 142 observations with 2 columns, labeled x & y.

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.

Mean (x)	Mean (y)	SD (x)	SD (y)
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9

Mean (x)	Mean (y)	SD (x)	SD (y)
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9
54.3	47.8	16.8	26.9

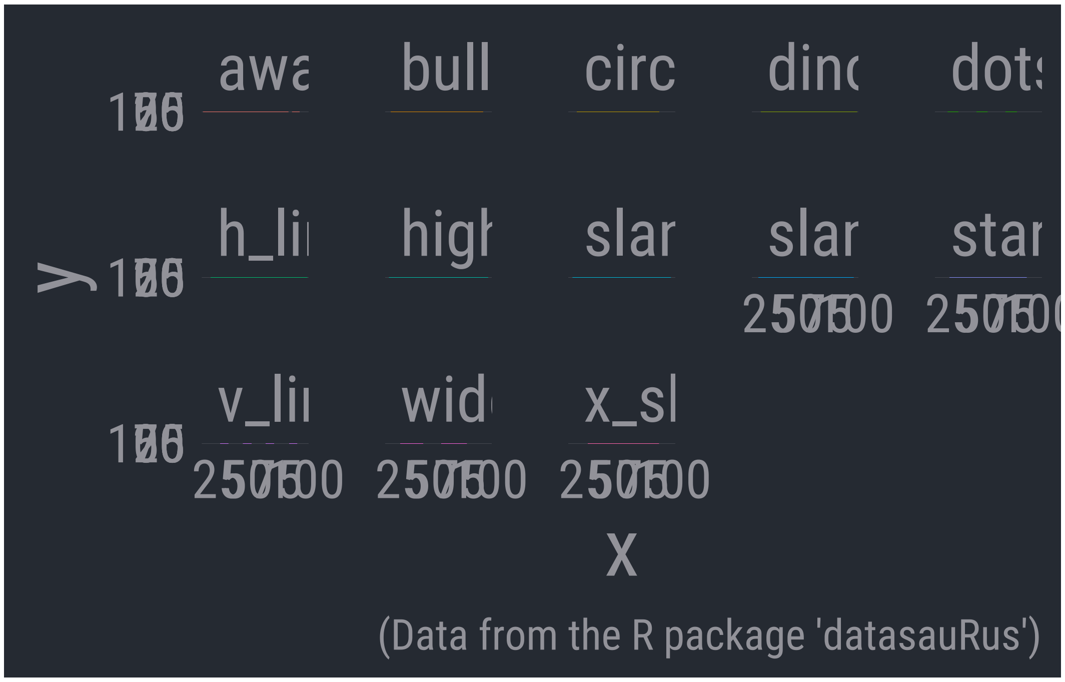
Well, there's not much we can say here. All the summary statistics are identical. Why don't we try modeling a linear relationship between the x and y variables. Maybe looking at the correlations will tell us something. I'll display the linear regression lines for each data set below.



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.



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.

7.3 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 standard OLS 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.

8 Plotting and Counting

In this last portion of our in-class lab sessions on quantitative analysis, we'll learn how to make some familiar visuals in new and exciting ways. For the most part, this will involve learning the graphing package of tidyverse, commonly known as `ggplot`, as well as `gtsummary`, a helpful package for producing tables. There are a couple of things to get used to with ggplot, but once you get a handle on its syntax, it will offer simple and extensive customizability that translates across a wide expanse of use cases. Even some of the graphics that we've seen in empirical papers for class have been produced using ggplot!

8.1 Setting up our workspace

Because ggplot comes with tidyverse, we just need to load in `tidyverse` like we've often done. We're also going to need `janitor`, and, as usual, we'll also pull in our GSS data.

```
library(tidyverse)
library(janitor)
load("our_gss.rda")
```

8.2 Figures

Specific to categorical variables, we learned about two key plot-types: histograms and barplots. We'll start our foray into `ggplot` by using it to recreate some figures in these styles.

8.2.1 Barplots & Basics of ggplot

The structure of figures made with ggplot is quite regular. The name 'ggplot' is actually a reference to the idea of a **g**rammar of **g**raphics, and though there are some specifics across the various plot types, we will offer ggplot mostly the same kind of information regardless.

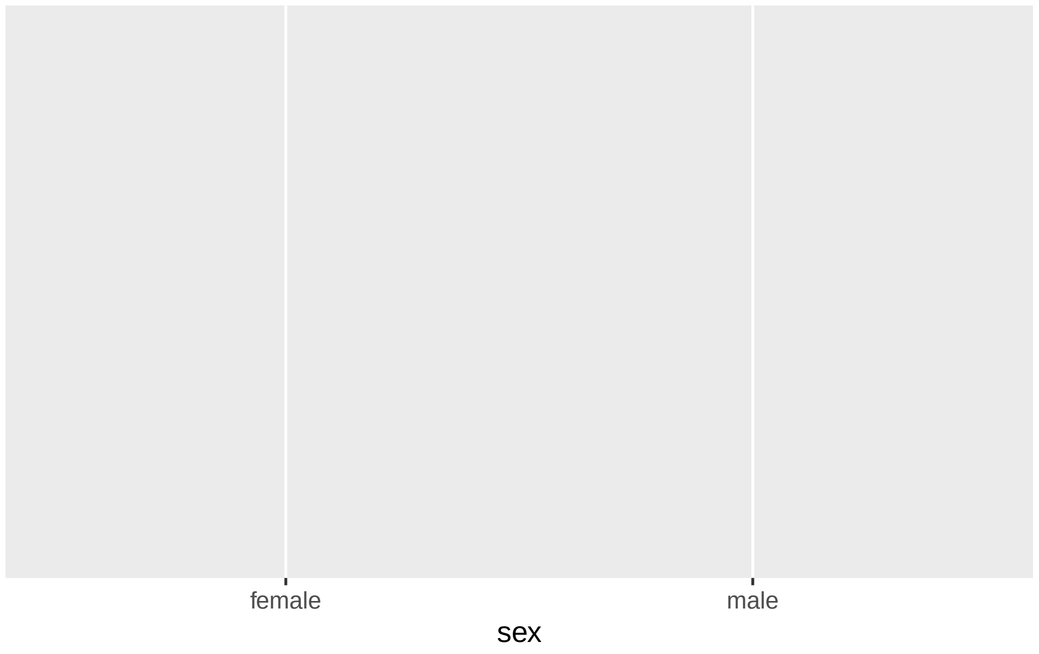
First, ggplot expects that we will give it a dataframe. This is good because it also means that ggplot plays well with the pipe operator.

Next, ggplot needs input for your **aesthetic mappings**, shortened to `aes()` within the function. This might sound a little hoighty toighty, but it simply refers to the specification of some key parameters for our figure. We want our data to be visually mapped in a certain sort of way, and this is where we set that up.

The simplest aesthetic mappings we can provide are the variables we want associated with the x & y columns. Let's try with `sex` as an example. Because we are making a barplot, our y-axis will reflect the frequency of the values of the y-axis.

We'll need to manually calculate this and add it to our data frame, which is thankfully simple enough.

```r
our_gss |>
  drop_na(sex) |>
  ggplot(
    aes(x = sex))
```



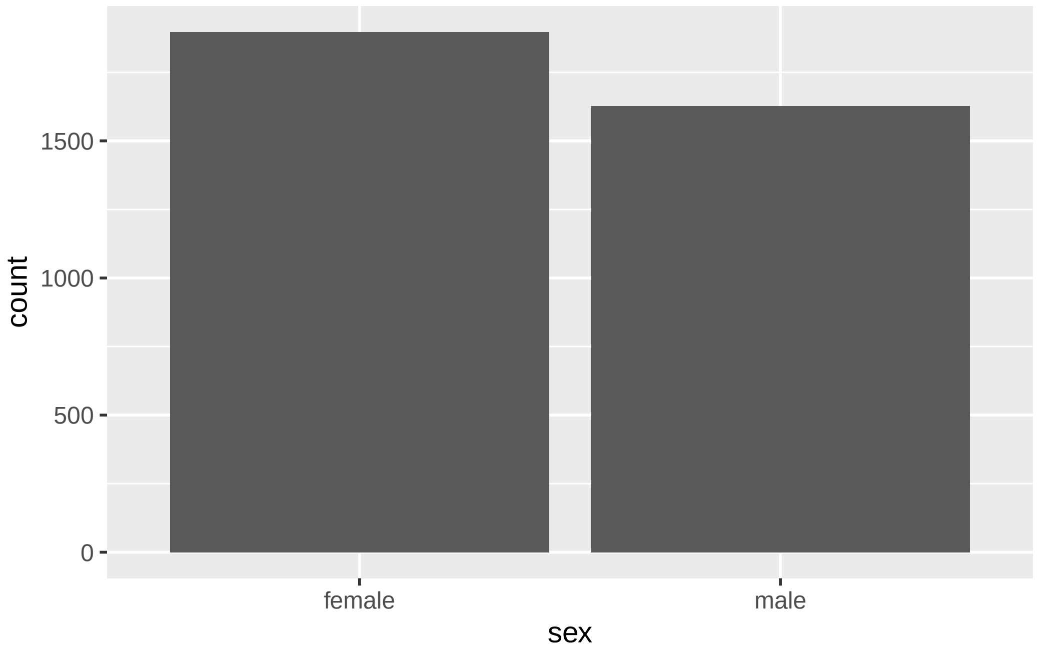
Now, you might be asking yourself: what's the deal? We only see the labels of the x-axis there.

The syntax of ggplot is often explained by making an analogy with paintings. You start by setting up a basic canvas with your `aes()` specifications, and then you add on different layers reflecting the shape of your data and various customizations to its appearance.

The first layer we typically add is a **geometric object** specification—shorted to `geom` in ggplot language. You can think of this as the place where you tell ggplot what kind of figure you want. There are a great many geoms, and you can read more about the full extent of them here.

In our case, we want a barplot, so we can add `geom_bar()` as an additional layer. We can simply add each new layer after a '+' operator

```
our_gss |>
  drop_na(sex) |>
  ggplot(
    aes(x = sex)) +
      geom_bar()
```

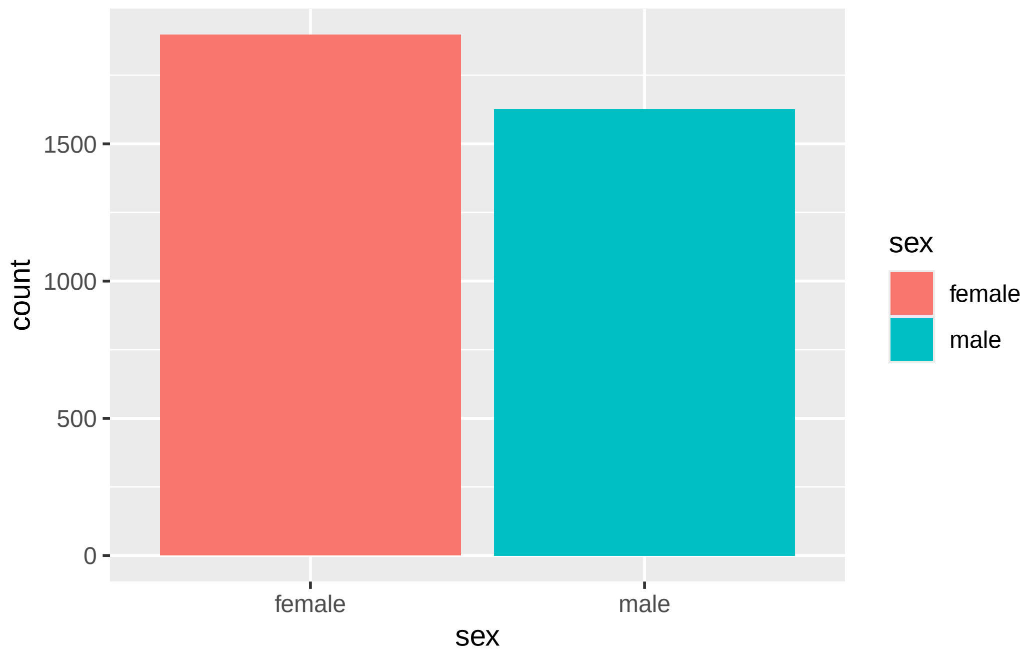


Voila! There's a pretty decent barplot. There are other kinds of statistical summaries we can take advantage of in ggplot, but the default behavior of `geom_bar()` will calculate the summary counts of the values of the x variable.

Now we tweak a couple things to make things a bit nicer and illustrate some of the perks of ggplot's customizability.

Let's add a little color. We can indicate this back up at the `aes()` layer. This works by adding a fill aesthetic, and then we specify a variable. What we do there is tell R that we want the fill-color of the bars to vary on the basis of that variable. So, each bar will be filled with a different color for each response category of `sex` in our case.

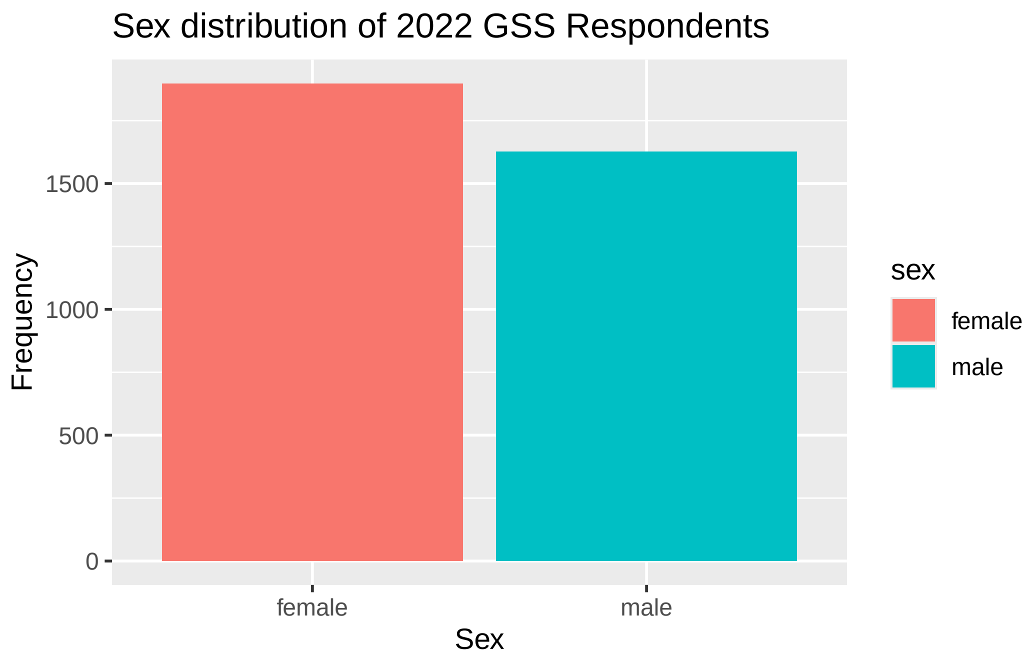
```r
our_gss |>
  drop_na(sex) |>
  ggplot(
    aes(x = sex, fill = sex)) +
      geom_bar()
```



Neat! This looks much better, and we'll play around with this more later, but now let's see how to add some axis & legend labels along with a title. The legend label actually works out fine as it is with `sex`, but the shorthand variable names often do not translate well as legend titles (imagine if it were 'realrinc', for example). So, I'll show you how to customize it just in case.

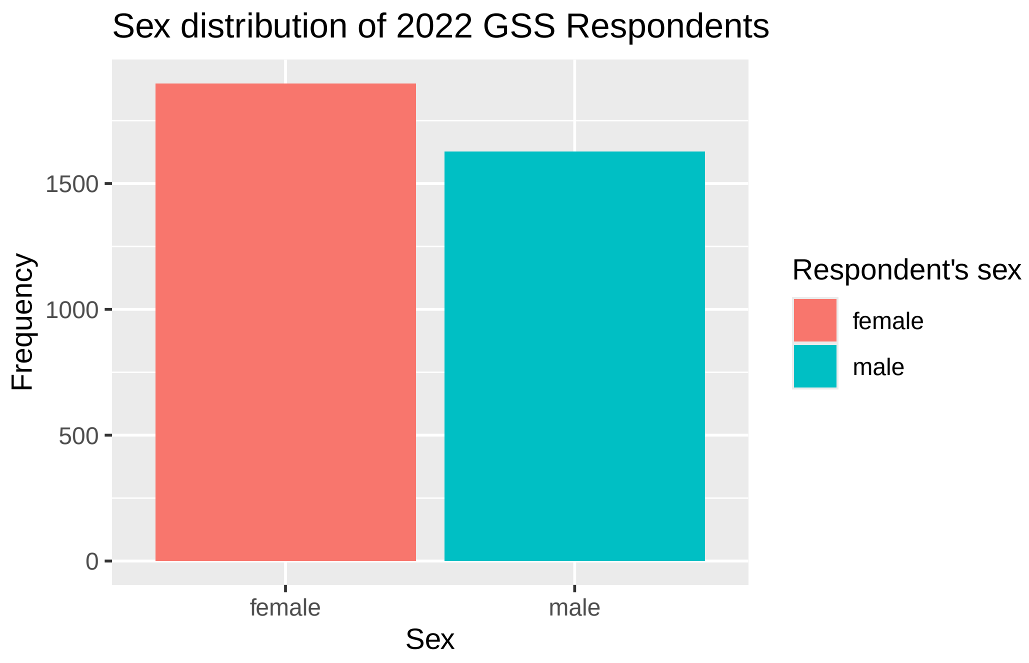
For the axis labels and title, we can add the `labs` layer

```r
our_gss |>
  drop_na(sex) |>
  ggplot(
    aes(x = sex, fill = sex)) +
      geom_bar() +
  labs(
    title = "Sex distribution of 2022 GSS Respondents",
    x = "Sex",
    y = "Frequency")
```



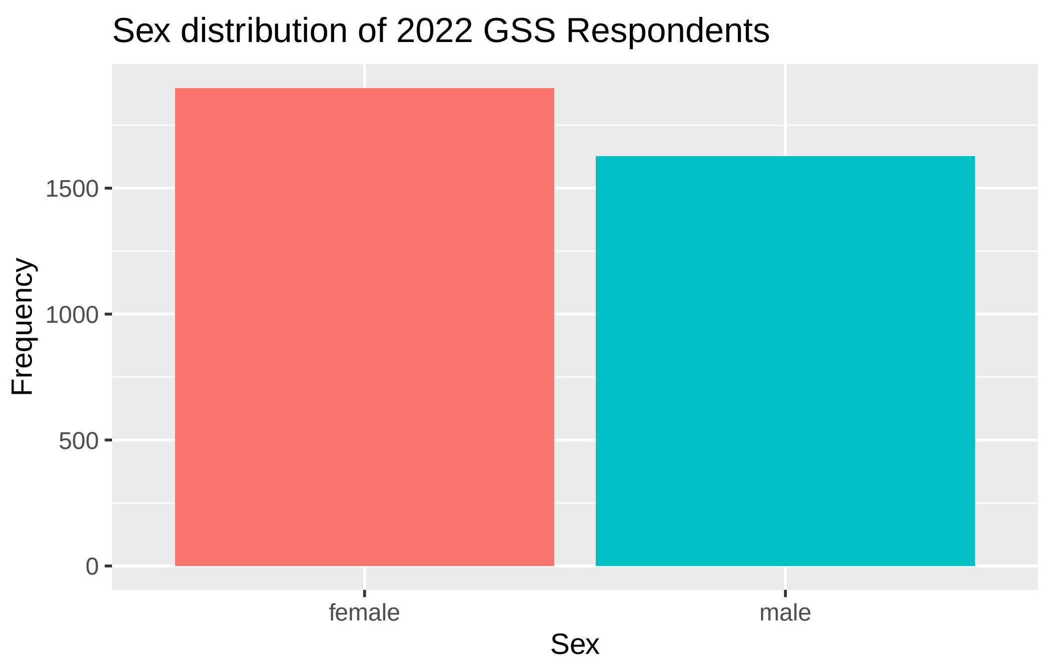
Then we can add the `guides` layer to customize the legend title for the `fill` aesthetic.

```r
our_gss |>
  drop_na(sex) |>
  ggplot(
    aes(x = sex, fill = sex)) +
  geom_bar() +
  labs(
    title = "Sex distribution of 2022 GSS Respondents",
    x = "Sex",
    y = "Frequency") +
  guides(fill = guide_legend(title="Respondent's sex"))
```



Lastly on this point, I'll also show you how to get rid of the legend entirely. You'll often want to customize the legend in the fashion I laid out above, but we actually don't really need it in this case. It's redundant with the x-axis labels. So, you can always get rid of a legend by adding the `guides()` layer and simply setting `fill` to 'none'.

```
our_gss |>
  drop_na(sex) |>
  ggplot(
    aes(x = sex, fill = sex)) +
  geom_bar() +
  labs(
    title = "Sex distribution of 2022 GSS Respondents",
    x = "Sex",
    y = "Frequency") +
  guides(fill = "none")
```



8.2.2 Bivariate Barplots

The `fill` aesthetic actually allows us to quite easily create a bivariate version of the barplot.

Let's consider our bivariate analysis from a previous session: `cappun` and `dem_rep`.

If you need to recreate `dem_rep` from `partyid`, go ahead and run the following code:

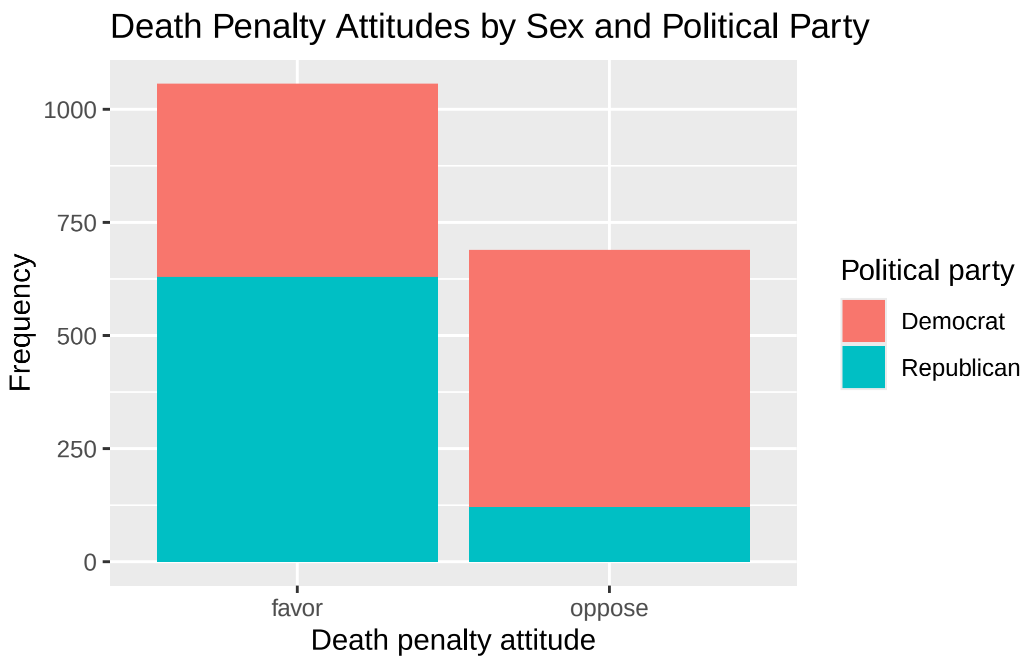
```
our_gss <- our_gss |>
  mutate(
    partyid_recoded=fct_collapse(partyid,
"Democrat" = c("strong democrat", "not very strong democrat"),
"Republican" = c("strong republican","not very strong republican"),
"Independent" = c("independent, close to democrat", "independent (neither, no response)", "i
"Other Party" = c("other party")
)) |>
  mutate(dem_rep = fct_recode(
    partyid_recoded,
    NULL="Other Party",
    NULL="Independent"))
```

As a reminder, here is the cross-tab for that conjoint distribution:

	Political Party	
Death Penalty Attitude	Democrat	Republican
favor	40.2% (426)	59.8% (633)
oppose	82.5% (569)	17.5% (121)
Total	56.9% (995)	43.1% (754)

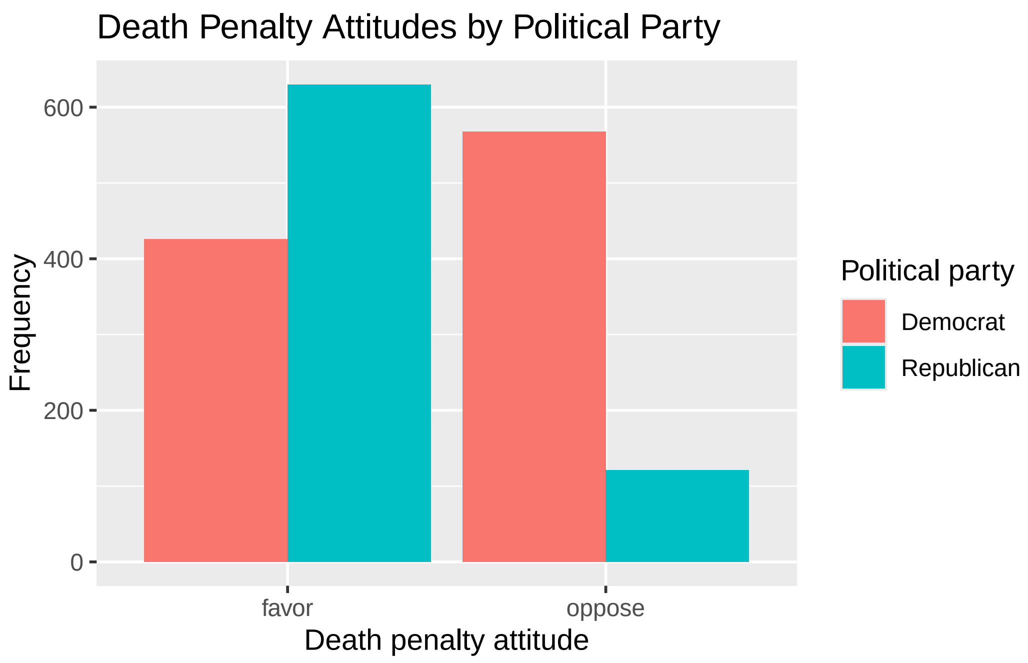
Now, we'll use our same barplot code from above, but this time we will set our x variable to `cappun` and swap out the `fill` aesthetic to `dem_rep`. This means we want the fill-color of the bars to vary on the basis of the categories in `dem_rep`. I'll also add `cappun` and `dem_rep` to `drop_na()` and update the labels & title.

```r
our_gss |>
  drop_na(sex, cappun, dem_rep) |>
  ggplot(
    aes(x = cappun, fill = dem_rep)) +
  geom_bar() +
  labs(
    title = "Death Penalty Attitudes by Sex and Political Party",
    x = "Death penalty attitude",
    y = "Frequency") +
  guides(fill = guide_legend(title="Political party"))
```



Good stuff! This is a stacked barplot and is a common way to display multivariate categorical data. I personally find it a little easier to intrepret when the columns are side-by-side rather than stacked on top of each other, so I'll show you how to do that too. This just requires setting a `position` mapping within `geom_bar()`. We set the position to 'dodge', which might sound a little counter-intuitive, which is because it actually has a more general purpose. The idea is that you want overlapping objects to 'dodge' one another, which will cause them to appear side by side rather than on top of one another. So, by doing so, we will un-nest our stacked bar charts and have each bar appear adjacent to one another.

```
our_gss |>
  drop_na(sex, cappun, dem_rep) |>
  ggplot(
    aes(x = cappun, fill = dem_rep)) +
  geom_bar(position = "dodge") +
  labs(
    title = "Death Penalty Attitudes by Political Party",
    x = "Death penalty attitude",
    y = "Frequency") +
  guides(fill = guide_legend(title="Political party"))
```



Not too shabby. This is a great way to complement the presentation of a bivariate analysis, and it's also great for quickly getting a sense of the relationship among variables of interest.

Characteristic	**N = 3,340**[1]
cappun	
favor	2,013 (60%)
oppose	1,327 (40%)

[1]n (%)

8.3 Tables

Now, let's learn a way to get some pretty snazzy frequency tables and cross-tabs using the `gtsummary` package, which is designed specifically for quickly producing publication-ready tables.

I'll load it in below. If you need to install it, you can run `install.packages("gtsummary")`

```r
library(gtsummary)
```

8.3.1 Frequency tables

The function we'll get the most mileage out of here is called `tbl_summary()`. It really does an incredible amount of stuff right out of the box, but it also offers a great deal of customizability. I'll note that `gtsummary()` can be a little idiosyncratic compared to some of the functions we've learned about already, and you can really get into the weeds with it. But I'll try to focus us here on just the things from which we can really get a lot of value.

It takes a data frame as it's first input, so, like ggplot, it plays well with the pipe operator.

We'll make a simple frequency table for univariate analysis of `cappun`. I'll first pass `cappun` through `drop_na()`, and then we'll pipe the output of that right into `tbl_summary()`. To get started, we just need to provide the variable we want to include, which has a straightforward enough input.

```r
our_gss |>
  drop_na(cappun) |>
  tbl_summary(
    include = cappun)
```

Excellent. This has some good detail right away. It automatically shows counts and percentages, and also gives the grand total while adding a footnote that this is what the percentages refer to. There are at least two things we'll want to customize.

Variable	**N = 3,340**[1]
cappun	
favor	2,013 (60%)
oppose	1,327 (40%)

[1]n (%)

Variable	**N = 3,340**[1]
cappun	
favor	2,013 (60%)
oppose	1,327 (40%)

[1]n (%)

For one, you might want to change 'Characteristic' to something like 'Variable.' We can do that like so:

```
our_gss |>
  drop_na(cappun) |>
  tbl_summary(
    include = cappun) |>
  modify_header(label ~ "Variable")
```

Note the use of the tilde symbol ~ here. Without getting too into the weeds, this is one of the idiosyncratic things about `gtsummary` I alluded to above. For our purposes, just observe that we need to do that here and don't worry too much about it.

However, observe that 'Variable' replaces 'Characteristic', but it's no longer bold. Thankfully, `gtsummary` has some font formatting conventions for us to take advantage of. If we simply place the quoted text between two asterisk (*), the text will be bolded

```
our_gss |>
  drop_na(cappun) |>
  tbl_summary(
    include = cappun) |>
  modify_header(label ~ "**Variable**")
```

Good. Now we'll want a better variable label than 'cappun'. Let's change that up.

Variable	**N = 3,340**[1]
Death Penalty Attitude	
favor	2,013 (60%)
oppose	1,327 (40%)

[1]n (%)

```r
our_gss |>
  drop_na(cappun) |>
  tbl_summary(
    include = cappun,
    label = list(
      cappun = "Death Penalty Attitude")) |>
  modify_header(label ~ "**Variable**")
```

Within the `tbl_summary()` function, we just need to add a 'label' input. Then you can specify your variable name and give a character vector for the label that we want. Because the 'label' input is designed to label multiple variables at once, it expects that you will give it a list of things. So, even if you're only working with one variable, you will need to put the label inside of `list()`.

8.3.2 Contingency tables

We can also make a quick and easy contingency table for bivariate analysis. Let's re-create our `cappun` x `dem_rep` table.

There's a nifty function called `tbl_cross()` which basically does the same thing as `tbl_summary()` but is more specific to contingency tables.

Remember to always provide your dependent variable to the 'rows' input.

```r
our_gss |>
  drop_na(cappun, dem_rep) |>
  tbl_cross(
    row = cappun,
    col = dem_rep,
    percent = "row",
    margin = c("column", "row"),
    label = list(
      cappun = "Death Penalty Attitude",
```

	Political Party		
	Democrat	Republican	Total
Death Penalty Attitude			
favor	426 (40%)	633 (60%)	1,059 (100%)
oppose	569 (82%)	121 (18%)	690 (100%)
Total	995 (57%)	754 (43%)	1,749 (100%)

```
      dem_rep = "Political Party"
    ))
```

So, for `tbl_cross()` we provide our row variable and our column variable. If we set `percent = "row"`, we will get percentages for each cell that will total across the rows—this is what we want. For `margin = c("column", "row")`, we're telling the function that we want totals for both columns and rows. Lastly, we give some better labels for our variables.

8.3.3 Chi-squared test

Now, a really neat feature of `tbl_summary()` is that it will automatically detect the type of variables that we have, and it will perform an appropriate statistical test. There's a lot of customization that you can do here, but this will work great out of the box for us.

All we have to do is pipe the command we just produced into `add_p()`, which will automatically calculate a p-value using a relevant statistical test.

```
our_gss |>
  drop_na(cappun, dem_rep) |>
  tbl_cross(
    row = cappun,
    col = dem_rep,
    percent = "row",
    margin = c("column", "row"),
    label = list(
      cappun = "Death Penalty Attitude",
      dem_rep = "Political Party"
    )) |>
  add_p()
```

Very neat! This automatically recognized that we have categorical variables, ran a chi-squared test, and reported the p-value. Note that it doesn't give the actual chi-squared value by default, but that information is in there. We can manually add a column like so:

	Political Party			
	Democrat	Republican	Total	p-value[1]
Death Penalty Attitude				<0.001
favor	426 (40%)	633 (60%)	1,059 (100%)	
oppose	569 (82%)	121 (18%)	690 (100%)	
Total	995 (57%)	754 (43%)	1,749 (100%)	

[1]Pearson's Chi-squared test

	Political Party				
	Democrat	Republican	Total	Chi-square value[1]	p-value[1]
Death Penalty Attitude				304	<0.001
favor	426 (40%)	633 (60%)	1,059 (100%)		
oppose	569 (82%)	121 (18%)	690 (100%)		
Total	995 (57%)	754 (43%)	1,749 (100%)		

[1]Pearson's Chi-squared test

```r
our_gss |>
  drop_na(cappun, dem_rep) |>
  tbl_cross(
    row = cappun,
    col = dem_rep,
    percent = "row",
    margin = c("column", "row"),
    label = list(
      cappun = "Death Penalty Attitude",
      dem_rep = "Political Party"
    )) |>
  add_p() |>
  modify_header(statistic ~ "Chi-square value")
```

Good stuff. Lots of great detail here, so this is pretty much good to go.

8.3.4 Elaboration Model

Lastly, this also gives us an easy way to automatically produce a 3-way cross-tab for multi-variate analysis. We'll need to change a couple of things around, but this can be done without

	female						male
	Democrat	Republican	Total	p-value[1]	Democrat	Republican	
Death Penalty Attitude				<0.001			
favor	244 (47%)	276 (53%)	520 (100%)		182 (34%)	354 (66%)	5
oppose	350 (83%)	71 (17%)	421 (100%)		218 (81%)	50 (19%)	2
Total	594 (63%)	347 (37%)	941 (100%)		400 (50%)	404 (50%)	8

[1]Pearson's Chi-squared test

too much work in `gtsummary`.

In our set up here, we have `cappun` as our dependent variable and `dem_rep` as our independent variable. Let's add `sex` as a test variable, and see whether sex has any effect on the relationship we have observed between death penalty attitudes and political party.

For this, we need the `tbl_strata()` function.

```r
our_gss |>
  select(cappun, sex, dem_rep) |>
  drop_na(cappun, sex, dem_rep) |>
  tbl_strata(
    strata = sex,
    .tbl_fun =
      ~ .x |>
    tbl_cross(
    row = cappun,
    col = dem_rep,
    percent = "row",
    margin = c("column", "row"),
    label = list(
      cappun = "Death Penalty Attitude",
      dem_rep = "Political Party"
    )) |>
    add_p()
)
```

This is starting to sprawl a little, but it's basically the same command as before, except that now we have put our `tbl_cross()` command inside of the `tbl_strata()`.

`tbl_strata`, first takes the 'strata' input, which is where we indicate our test variable. It's the variable by which we want to *stratify* our data. The effect of `strata = sex` is somewhat similar to `group_by()` from tidyverse—it groups the data by our test variable.

`.tbl_fun = ~.x` probably looks a little strange. This gets a little into the weeds, so don't worry about it too much, but I'll give a little context.

.tbl_fun is shorthand for "table function", and `.x` is an R-specific placeholder for function inputs. Essentially, we are saying "group the data by our `strata` variable, and then pass that stratified data into another function of my choosing."

Then, we can just give our `tbl_cross()` function from before, and this will be calculated according to the grouping structure of the test variable we provide.

Now, as an exercise, let's think about what we have here.

We have our original relationship predicting death-penalty attitudes based on political affiliation. But now we can see this relationship for both men and women. The female group has 941 respondents and the male group has 804.

If we look at the percentages for both the male & female groups, they are relatively similar and the basic relationship is about the same: Democrats are more likely to oppose the death penalty and less likely to favor the death penalty, regardless of sex. Not only this, but we can see that a chi-squared test has been run for both the male and female subgroups. Both of these tests returned statistically significant p-values, as we can see in the table.

So, this would be a case of replication. We have replication when our zero-order relationship remains statistically significant after introducing our test variable, and the nature of the relationship does not change. In this case, we observed a statistically significant relationship between political affiliation and death-penalty attitudes. After we added `sex` as a test variable, this original relationship remained statistically significant across the two response categories of `sex`. If the percentages had been markedly different across males and females, we would say that this was specification, as that would suggest that—even if political affiliation is associated with death penalty attitudes—the relationship is different for men and women.