Reporting with Data in R

Christian McDonald

2021-12-12

Contents

About this book

> NOTE: This book is in the middle of a rewrite. The original "beta" version used in spring 2019 is long gone in a commit history and the new version is being rewritten under a new URL. Some chapters need to be written or rewritten, as noted. This new version will become v1.0 and is being used for fall 2021.

Reporting with Data in R is a series of lessons and instructions used for courses in the School of Journalism and Media, Moody College of Communication at the University of Texas at Austin.

It is written in bookdown and the source is available on Github.

I'm a strong proponent of what I call Scripted Journalism, a method of committing data-centric journalism in a programmatic, repeatable and transparent way. There are a myriad of programming languages that further this, including Python (pandas using Jupyter) and JavaScript (Observable), but we'll be using R, RMarkdown and RStudio.

R is a super powerful, open-source programming language for data that is deep with features and an awesome community of users who build upon it. No matter the challenge before you in your data storytelling, there is probably a package available to help you solve that challenge. Probably more than one.

About my philosphy

There is always more than one way to do things in R. This book is a Tidyverse-oriented, opinionated collection of lessons intended to teach students new to programming and R for the expressed act of committing journalism. As a beginner course, I strive to make it as simple as possible, which means I may not go into detail about alternative (and possibly better) ways to accomplish tasks in favor of staying in the Tidyverse and reducing options to simplify understanding.

This is the second version of this book. The first "beta" version was used in Spring 2019, and it was my first time to introduce R to beginning students.

While the experience went well, there were pros and cons to using R in a beginning data class and I continue to experiment with material. I hope to use my experience in that first class to improve this edition.

After that Spring 2019 class I chose to use a different web-based tool — Workbench — which allowed for a similar scripted workflow but without the same level of coding. I loved Workbench, especially for beginning students, but the site is scheduled to close down in October 2021.

Tips on my writing style

I try to be consistent in the way I write documentation and lessons. I'm human, so sometimes break my own rules, but in general I keep the following in mind.

Things to do

I usually put things I want you to DO in ordered lists:

1. Do this thing.
2. Then do this thing.

Explanations are usually in text, like this very paragraph.

And sometimes I'll explain things in lists:

- This is the first thing I want you to know.
- This is the second. You don't have to DO these things, just know about them.

Notes, some important

I will use the blockquote syntax to set off irrelevant background:

> Markdown was developed by JOHN GRUBER, as outlined on his Daring Fireball blog.

But sometimes those asides are important. I usually indicate that:

> IMPORTANT: You really should learn how to use Rmarkdown as you will use it the whole semester, and hopefully for the rest of your life.

Copy code blocks

When you see R code in the instructions, you can roll your cursor over the right-corner and click on the copy icon to copy the code to your clipboard:

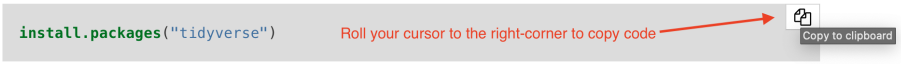


Figure 1: Copy to clipboard

You can then paste the code inside your R chunk.

That said, typing code yourself has many, many benefits. You learn better when you type yourself, make mistakes and have to fix them. **I encourage you to always type short code snippets.** Leave the copying to long ones.

About the author

I'm a career journalist who most recently served as Data and Projects Editor at the Austin American-Statesman before joining the University of Texas at Austin faculty full-time in Fall 2018 as an assistant professor of practice. I've taught data-related course at UT since 2013.

- My UT Github: utdata
- My Personal Github: critmcdonald
- Twitter: crit
- Email: christian.mcdonald@utexas.edu

License

Let's just say this is free on the internet and I don't make any money from it and you shouldn't either.

Other resources

This text stands upon the shoulders of giants and by design does not cover all aspects of using R. Here are some other useful books, tutorials and sites dedicated to R. There are other task-specific tutorials and articles sprinkled throughout the book in the Resources section of select chapters.

- R Journalism Examples, a companion piece of sorts to this book with example code to accomplish specific tasks. It is a work-in-progress, and quite nascent at that.
- R for Data Science by Hadley Wickham and Garrett Grolemund.
- The Tidyverse site, which has tons of documentation and help.
- The RStudio Cheatsheets.
- R Graphics Cookbook
- The R Graph Gallery another place to see examples.
- Practical R for Journalism by Sharon Machlis, an editor with PC World and related publications. Sharon is a longtime proponent of using R in journalism.
- Sports Data Analysis and Visualization and Data Journalism with R and the Tidyverse by Matt Waite, a professor at the University of Nebraska-Lincoln.
- R for Journalists site by Andrew Tran, a reporter at the Washington Post. A series of videos and tutorials on using R in a journalism setting.

Chapter 1

Install Party

Let's get this party started.

> NOTE: R and RStudio are already installed on lab computers.

1.1 Mac vs PC

I use a Mac in class and in my examples. I'm a big fan of using keyboard commands to do operations in any program, but I reference this from a Mac perspective. So if I say use *Cmd+S* or *Command+S* to save, that might be *Cntl+S* or *Control+S* on a PC. The letters may not be the same on a PC, but you can usually figure it out by look at menu items in RStudio to figure out the PC command.

We will install R and RStudio. It might take some time depending on your Internet connection.

If you are doing this on your own you might follow this tutorial. But below you'll find the basic steps.

1.2 Installing R

Our first task is to install the R programming language onto your computer.

- Go to the https://cloud.r-project.org/.
- Click on the link for your operating system.
- The following steps will differ slightly based on your operating system.

  - For Macs, you want the "latest package" unless you have an "M1" Mac (Nov. 2020 or newer), in which case choose the **arm64.pkg** version.
  - For Windows, you want the "base" package. You'll need to decide whether you want the 32- or 64-bit version. (Unless you've got a pretty old system, chances are you'll want 64-bit.)

Here's hoping it will be self explanatory after that.

You'll never "launch" R as a program in a traditional sense, but you need it on your computers. We'll use RStudio, which is next.

1.3 Installing RStudio

RStudio is an "integrated development environment" – or IDE – for programming in R. Basically, it's the program you will use when doing work for this class.

- Go to https://www.rstudio.com/download.
- Scroll down to the versions and find **RStudio Desktop** and click on the **Download** button.
- It should take you down the page to the version you need for your computer.
- Install it. Should be like installing any other program on your computer.

1.4 Class project folder

To keep things consistent and help with troubleshooting, I'd like you to save your work in the same location all the time.

- On both Mac and Windows, every user has a "Documents" folder. Open that folder. (If you don't know where it is, ask me to help you find it.)
- Create a new folder called "rwd." Use all lowercase letters.

When we create new "Projects," I want you to always save them in the `Documents/rwd` folder. This just keeps us all on the same page.

Chapter 2

Introduction to R

2.1 RStudio tour

When you launch RStudio, you'll get a screen that looks like this:

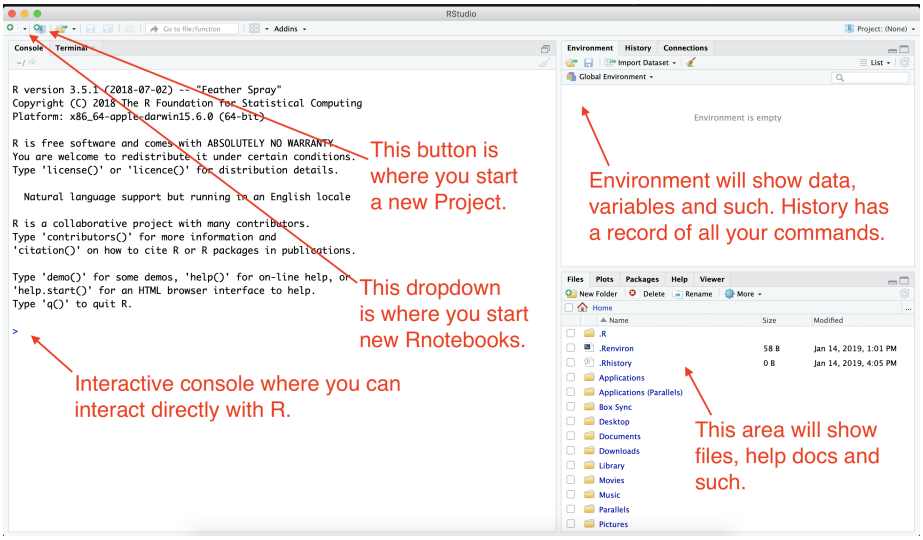


Figure 2.1: RStudio launch screen

2.2 Updating preferences

There is a preference in RStudio that I would like you to change. By default, the program wants to save a the state of your work (all the variables and such) when you close a project, but that is not good practice. We'll change that.

1. Go to the **RStudio** menu and choose **Preferences**
2. Under the **General** tab, uncheck the first four boxes.
3. On the option "Save Workspace to .Rdata on exit," change that to **Never**.
4. Click *OK* to close the box.

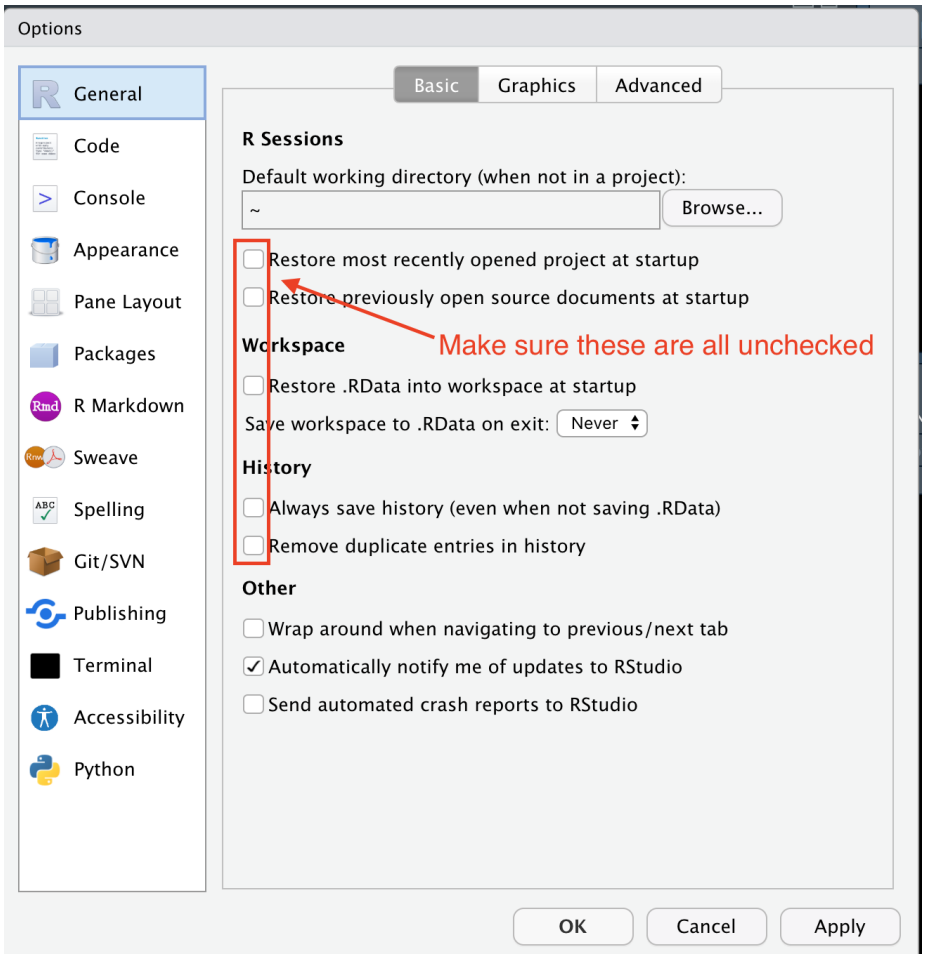


Figure 2.2: RStudio preferences

2.3 Starting a new Project

When we work in RStudio, we will create "Projects" to hold all the files related to one another. This sets the "working directory," which is a sort of home base for the project.

1. Click on the second button that has a green `+R` sign.
2. That brings up a box to create the project with several options. You want **New Directory** (unless you already have a Project directory, which you don't for this.)
3. For **Project Type**, choose **New Project**.
4. Next, for the **Directory name**, choose a new name for your project folder. For this project, use "firstname-first-project" but use YOUR firstname.
5. For the subdirectory, you want to use the **Browse** button to find your new `rwd` folder we created earlier.

I want you to be anal about naming your folders. It's a good programming habit.

- Use lowercase characters.
- Don't use spaces. Use dashes.
- For this class, start with your first name.

When you hit **Create Project**, your RStudio window will refresh and you'll see the `yourfirstname-first-project.Rproj` file in your Files list.

2.4 Using R Notebooks

For this class, we will almost always use RNotebooks. This format allows us to write text in between our blocks of code. The text is written in a language called RMarkdown, a juiced-up version of the common documentation syntax used by programmers, Markdown. We'll learn that in a moment.

2.4.1 Create your first notebook

1. Click on the button at the top-left of RStudio that has just the green `+` sign.
2. Choose the item **R Notebook**.

This will open a new file with some boilerplate R Markdown code.

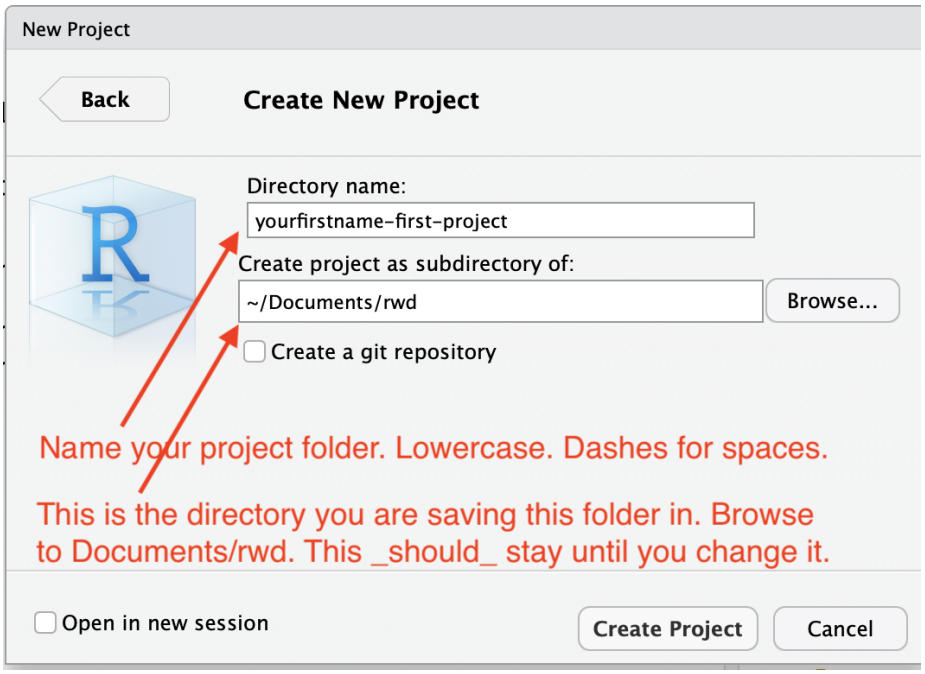


Figure 2.3: Rstudio project name, directory

1. At the top between the `---` marks, is the **metadata**. This is written using YAML, and what is inside are commands for the R Notebook. Don't sweat the YAML syntax too much right now, as we won't be editing it often.
2. Next, you'll see a couple of paragraphs of text that describes how to use an RNotebook. It is written in RMarkdown, and has some inline links and bold commands, which you will learn,
3. Then you will see an R code chunk that looks like the figure below.

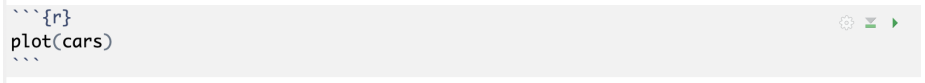


Figure 2.4: R code chunk

Let's take a closer look at this:

- The three back tick characters (the key found at the top left on your keyboard) followed by the `{r}` indicate that this is a chunk of R code. The last three back ticks say the code chunk is over.
- The `{r}` bit can have some parameters added to it. We'll get into that later.

- The line `plot(cars)` is R programming code. We'll see what those commands do in a bit.
- The green right-arrow to the far right is a play button to run the code that is inside the chunk.
- The green down-arrow and bar to the left of that runs all the code in the Notebook up to that point. That is useful as you make changes in your code and want to rerun what is above the chunk in question.

2.4.2 Save the .Rmd file

1. Do *Cmd+S* or hit the floppy disk icon to save the file.
2. It will ask you what you want to name this file. Call it `01-first-file.Rmd`.

When you do this, you may see another new file created in your Files directory. It's the pretty version of the notebook which we'll see in a minute.

In the metadata portion of the file, give your notebook a better title.

1. Replace "R Notebook" in the `title: "R Notebook"` code to be "Christian's first notebook," but use your name.

2.4.3 Run the notebook

There is only one chunk to run in this notebook, so:

1. Click on the green right-arrow to run the code. The keyboard command (from somewhere within the chunk) is *Cmd+Shift+Return*.

You should get something like this:

What you've done here is create a plot chart of a piece of sample data that is already inside R. (FWIW, It is the speed of cars and the distances taken to stop. Note that the data were recorded in the 1920s.)

But that wasn't a whole lot of code to see there is a relationship with speed vs stopping distance, eh?

This is a "base R" plot. We'll be using the tidyverse ggplot methods later in the semester.

2.4.4 A note about RMarkdown

We always want to annotate our code to explain what we are doing. To do that, we use a syntax called RMarkdown, which is an R-specific version of Markdown.

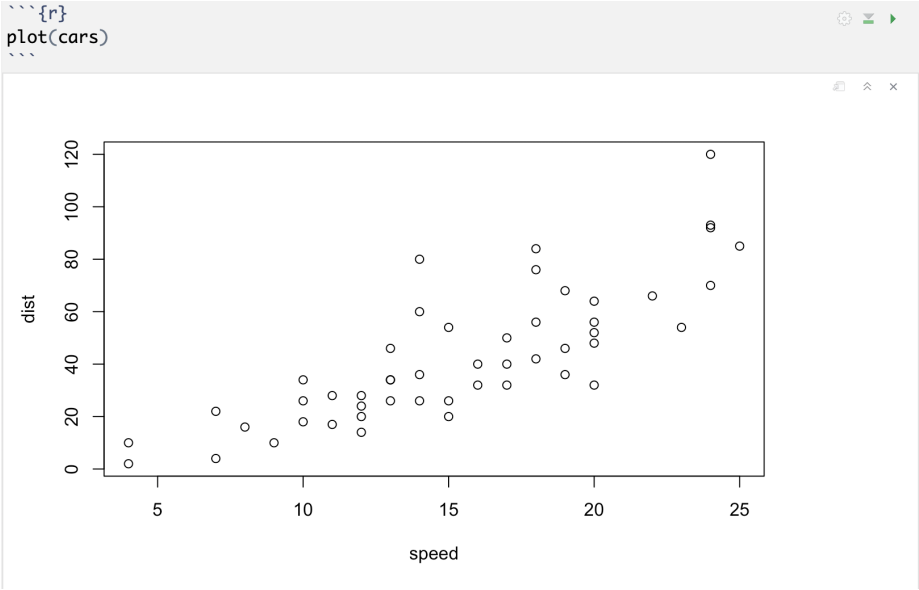


Figure 2.5: Cars plot

We use this syntax because it both makes sense in text but also makes a very pretty version in HTML when we "knit" our project. You can see how it to write RMarkdown here.

This entire book is written in RMarkdown.

Here is an example:

```
## My dating age

The following section details the [socially-acceptable maximum age of anyone you should

The math works like this:

- Take your age
- subtract 7
- Double the result
```

- The `##` line is a headline. Add more `###` and you get a smaller headline, like subheads.
- There is a full blank return between each element, including paragraphs of text.
- In the first paragraph we have embedded a hyperlink. We put the words we want to show inside square brackets and the URL in

parenthesis DIRECTLY after the closing square bracket: `[words to link](https://the_url.org)`.
- The `-` at the beginning of a line creates a bullet list. (You can also use `*`). Those lines need to be one after another without blank lines.

1. Go ahead and copy the code above and add it as text in the notebook so you can see it works later.

2.4.5 Adding new code chunks

The text after the chart describes how to insert a new code chunk. Let's do that.

1. Add a couple of returns before the paragraph of text about code chunks.
2. Use the keys *Cmd+Option+i* to add the chunk.
3. Your cursor will be inserted into the middle of the chunk. Type in this code in the space provided:

```
# update 54 to your age
age <- 54
(age - 7) * 2
```

```
## [1] 94
```

1. Change for "54" to your real age.
2. With your cursor somewhere in the code block, use the key command *Cmd+Shift+Return*, which is the key command to RUN ALL LINES of code chunk.

   NOTE: To run an individual line, use *Cmd+Return* while on that line.

Congratulations! The answer given at the bottom of that code chunk is the socially-acceptable maximum age of anyone you should date.

Throwing aside whether the formula is sound, let's break down the code.

- `# update 54 to your age` is a comment. It's a way to explain what is happening in the code without being considered part of the code. We create comments by starting with `#`. You can also add a comment at the end of a line.

- `age <- 54` is assigning a number (`54`) to an R object/variable called (`age`). A variable is a placeholder. It can hold numbers, text or even groups of numbers. Variables are key to programming because they allow you to change a value as you go along.
- The next part is simple math: `(age - 7) * 2` takes the value of `age` and subtracts `7`, then multiplies by `2`.
- When you run it, you get the result of the math equazion, `[1] 94` in my case. That means there was one observation, and the value was "94." For the record, my wife is *much* younger than that.

Now you can play with the number assigned to the age variable to test out different ages. Do that.

2.4.6 Practice adding code chunks

Now, on your own, add a similar section that calculates the **minimum** age of someone you should date, but using the formula `(age / 2) + 7`.

1. Add a RMarkdown headline and text describing what you are doing.
2. Create a code chunk that that calculates the formula based on your age.
3. Include a comment within the code block.

2.4.7 Preview the report

The rest of the boilerplate text here describes how you can *Preview* and *Knit* a notebook. Let's do that now.

- Press *Cmd+Shift+K* to open a Preview.

This will open a new window and show you the "pretty" notebook that we are building.

Preview is a little different than *Knit*, which runs all the code, then creates the new knitted HTML document. It's **Knit to HMTL** that you'll want to do before turning in your assignments. That is explained below.

2.4.8 The toolbar

One last thing to point out before we turn this in: The toolbar that runs across the top of the R Notebook file window. The image below explains some of the more useful tools, but you *REALLY* should learn and use keyboard commands when they are available.

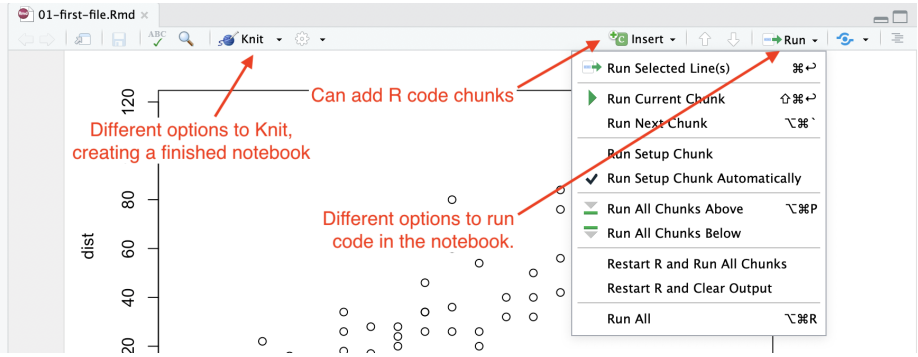


Figure 2.6: R Notebook toolbar

2.4.9 Knit the final workbook

1. Save your File with *Cmd+S*.
2. Click on the dropdown next to the **Run** menu item and choose *Restart R and Run All Chunks*. We do this to make sure everything still works.
3. Use the **Knit** button in the toolbar to choose **Knit to HTML**.

This will open your knitted file. Isn't it pretty?

2.5 Turning in our projects

If you now look in your Files pane, you'll see you have four files in our project. (Note the only one you actually edited was the `.Rmd` file.)

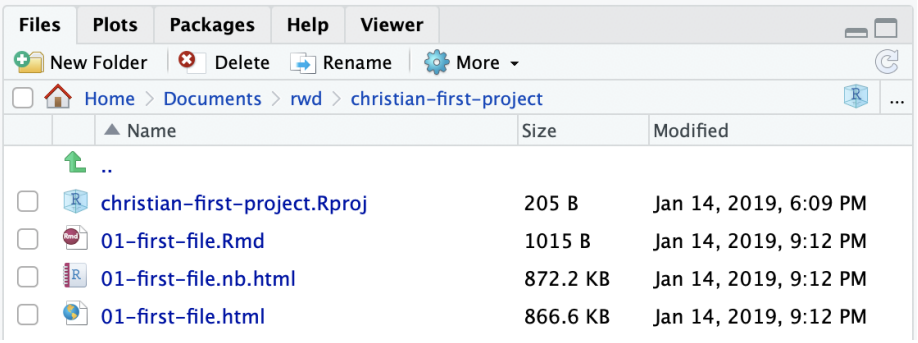


Figure 2.7: Files list

The best way to turn in all of those files into Canvas is to compress them into a single `.zip` file that you can upload to the assignment.

1. In your computer's Finder, open the `Documents/rwd` folder.
2. Follow the directions for your operating system linked below to create a compressed version of your `yourname-final-project` folder.
3. Compress files on a Mac.
4. Compress flies on Windows.
5. Upload the resulting `.zip` file to the assignment for this week in Canvas.

If you find you make changes to your R files after you've zipped your folder, you'll need to delete the `zip` file and compress it again.

Because we are building "repeatable" code, I'll be able to download your `.zip` files, uncompress them, and the re-run them to get the same results.

Well done! You've completed the first level and earned the *Beginner* badge.

Chapter 3

Summarize with count - import

> "If you're doing data analysis every day, the time it takes to learn a programming language pays off pretty quickly because you can automate more and more of what you do." –Hadley Wickham, chief scientist at RStudio

3.1 Learning goals of this lesson

- Practice organized project setup.
- Learn a little about data types available to R.
- Learn about R packages, how to install and import them.
- Learn how to download and import CSV files using the readr package.
- Introduce the Data Frame/Tibble.
- Introduce the tidyverse `%>%`.
- Learn how to modify data types (date) and `select()` columns.

We'll be exploring the Billboard Hot 100 charts along the way. Eventually you find the answers to a bunch of questions in this data and write about it.

3.2 Basic steps of this lesson

Before we get into our storytelling, we have to get our data and make sure it is in good shape for analysis. This is pretty standard for any new project. Here are the major steps we'll cover in detail for this lesson (and many more to come):

- Create your project structure
- Find the data and get it
- Import the data into your project
- Clean up data types and columns
- Export cleaned data for later analysis

3.3 Create a new project

We did this once Chapter 2, but here are the basic steps:

1. Launch RStudio
2. Make sure you don't have an existing project open. Use File > Close project if you do.
3. Use the `+R` button to create a **New Project** in a **New Directory**
4. Name the project `yourfirstname-billboard` and put it in your `~/Documents/rwd` folder.
5. Use the `+` button and use **R Notebook** to start a new notebook.
6. Change the title to "Billboard Hot 100 Import."
7. Delete the other boilerplate text.
8. Save the file as `01-import.Rmd`.

3.3.1 Describe the goals of the notebook

We'll add our first bit of RMarkdown just after the meta data to explain what we are doing. Add this text to your notebook:

```
## Goals of this notebook

Steps to prepare our data:

- Download the data
- Import into R
- Clean up data types and columns
- Export for next notebook
```

We want to start each notebook with a list like this so our future selves and others know what the heck we are trying to accomplish.

We will also write text like this for each new "section" or goal in the notebook.

3.3.2 The R Package environment

We have to back up from the step-by-step nature of this lesson and talk a little about the R programming language.

R is an open-source language, which means that other programmers can contribute to how it works. It is what makes R beautiful.

What happens is developers will find it difficult to do a certain task, so they will write an R "Package" of code that helps them with that task. They share that code with the community, and suddenly the R garage has an "ultimate set of tools" that would make Spicoli's dad proud.

One set of these tools is Hadley Wickham's Tidyverse, a set of packages for data science. These are the tools we will use most in this course. While not required reading, I highly recommend Wickham's book R for data science, which is free.

There are also a series of useful cheatsheets that can help you as you use the packages and functions from the tidyverse. We'll refer to these throughout the course.

3.3.3 Installing and using packages

There are two steps to using an R package:

- **Install the package** using `install.packages("package_name")`. You only have to do this once for each computer, so I usually do it using the R Console instead of in notebook.
- **Include the library** using `library(package_name)`. This has to be done for each Notebook or script that uses it, so it is usually one of the first things in the notebook.

> Note that you have to use "quotes" around the package name when you are installing, but you DON'T use quotes when you load the library.

We're going to install several packages we will use in this project. To do this, we are going to use the **Console**, which we haven't talked about much yet.

1. Use the image above to orient yourself to the R Console and Terminal.
2. In the Console, type in:

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

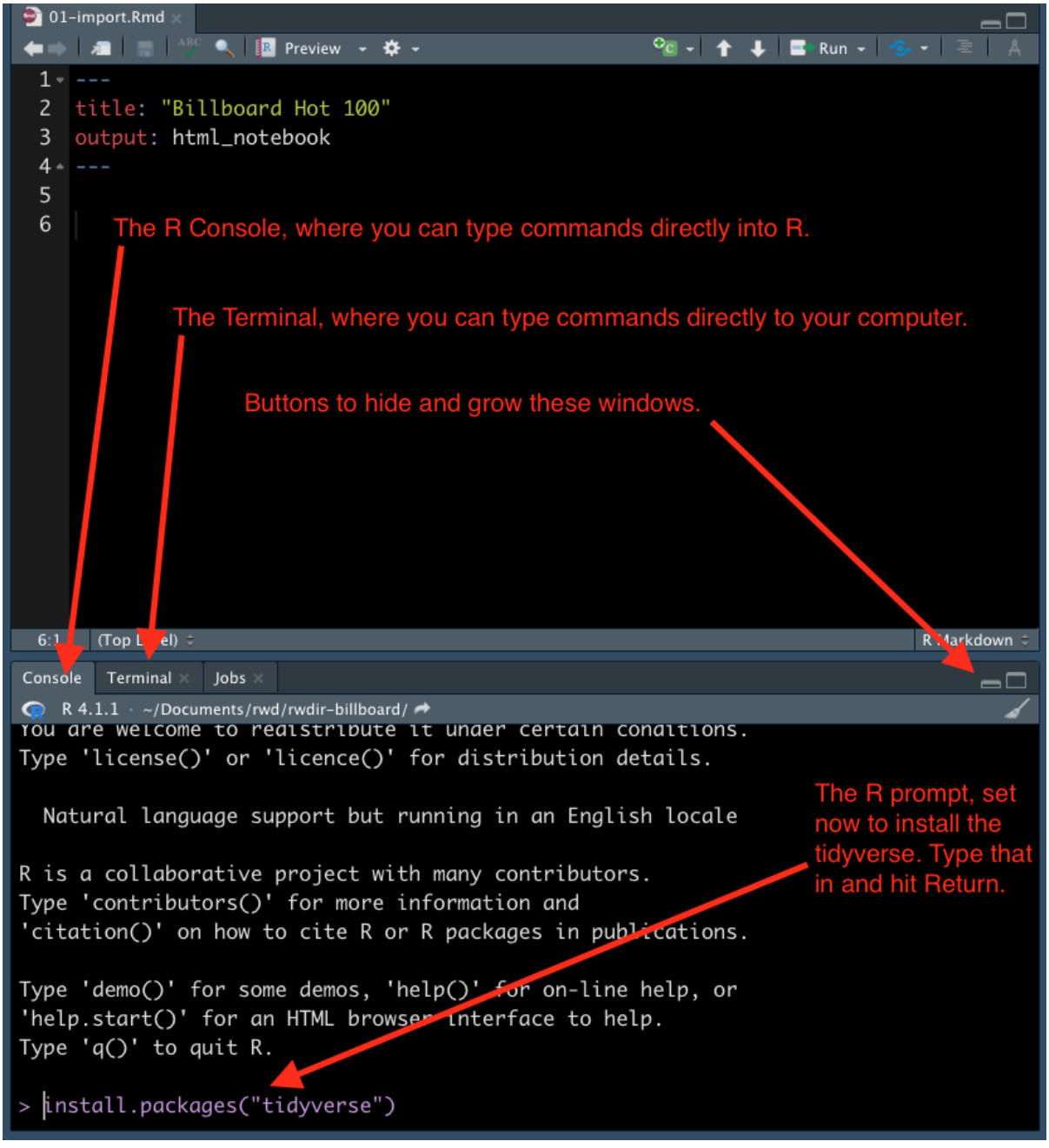


Figure 3.1: The Console and Terminal

As you type into the Console, you'll see some type-assist hints on what you need. You can use the arrow keys to select one and hit the *tab* key to complete that command, then enter the values you need. If it asks you to install "from source," type `Yes` and hit return.

You'll see a bunch of response in the Console.

1. We need two other packages as well, so also do:

```
install.packages("janitor")
install.packages("lubridate")
```

We'll use janitor to clean up our data column names, among other things. A good reference to learn more is the janitor vignette.

We'll use lubridate to fix some dates, which are a special complicated thing in programming. Lubridate is part of the tidyverse universe, but we have to install and load it separately.

You only have to install the packages once on your computer (though you have to load them into each new notebook, which is explained below).

3.3.4 Load the libraries

Next, we're going to tell our R Notebook to use these three libraries.

1. After the metadata at the top of your notebook, use *Cmd+option+i* to insert an R code chunk.
2. In that chunk, type in the two libraries and run the code block with *Cmd+Shift+Return*.

This is the code you need:

```
library(tidyverse)
library(janitor)
library(lubridate)
```

Your output will look something like this:

3.3.5 Create a directory for your data

I want you to create a folder called `data-raw` in your project folder. We are creating this folder because we want to keep a pristine version of our original data that we never change or overwrite. This is a basic data journalism principle: *Thou shalt not change raw data.*

In your Files pane at the bottom-right of Rstudio, there is a **New Folder** icon.

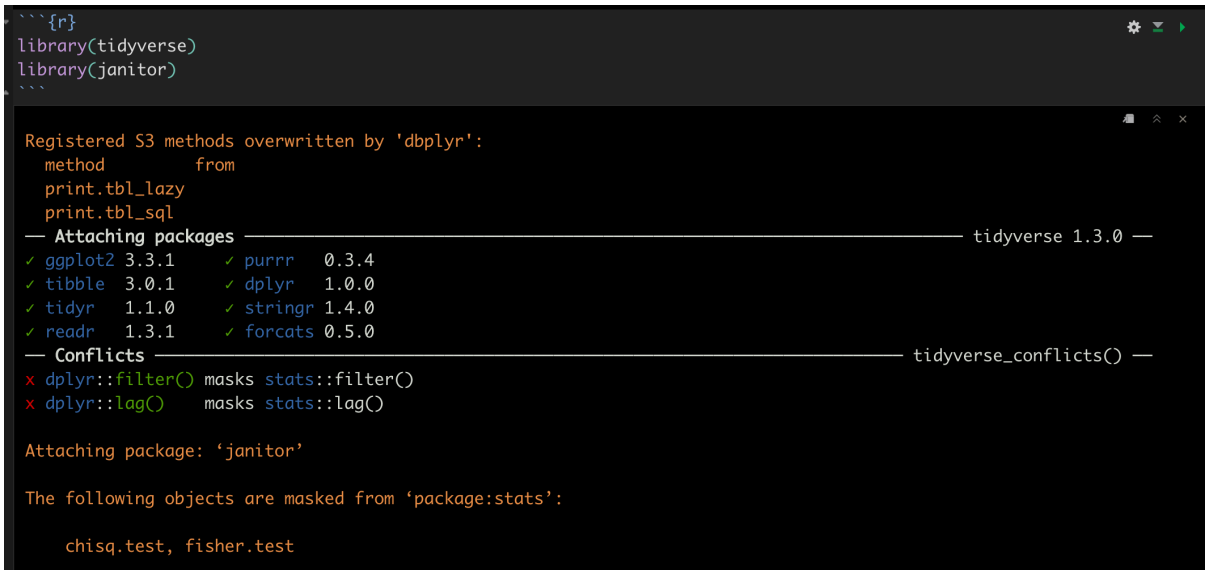


Figure 3.2: Libraries imported

1. Click on the **New Folder** icon.
2. Name your new folder `data-raw`. This is where we'll put raw data. We never write data to this folder.
3. Also create another new folder called `data-processed`. This is were we write data. We separate them so we don't accidentally overwrite raw data.

Once you've done that, they should show up in the file explorer in the Files pane. Click the refresh button if you don't see them. (The circlish thing at top right of the screenshot below. You might have to widen the pane to see it.)

Your `.Rproj` file name is likely different (and that s OK) and you can ignore the `.gitignore` I have there.

3.3.6 Let's get our data

Now that we have a folder for our data, we can download our data into it. The data was scraped and saved on data.world by Sean Miller, but you can just download my copy of the data using the `download.file` function in R.

For the purposes of this assignment, we will "source" the data as being from Billboard Media, as that is who initially provided it. I've worked with data fairly extensively, and it is sound.

1. Add a Markdown headline `## Downloading data` and on a new line text

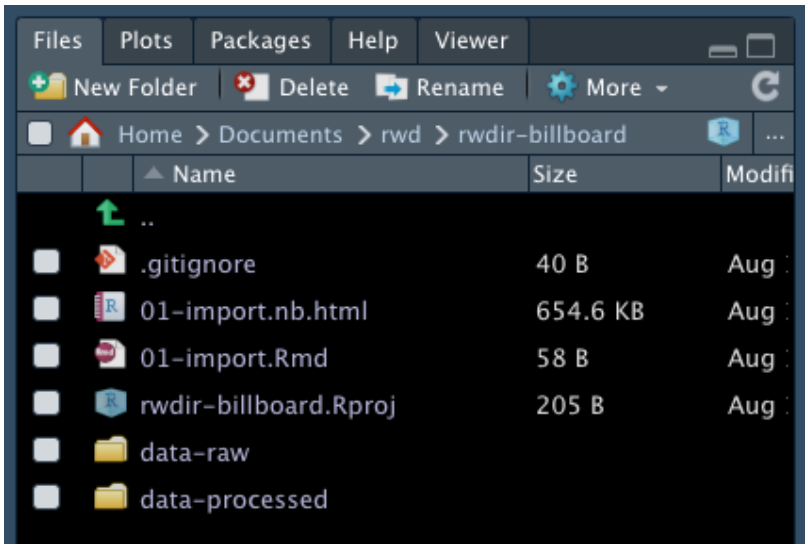


Figure 3.3: Directory made

that indicates you are downloading data. You would typically include a link and explain what it is, etc, often linking to the original source.

2. Create an R chunk and include the following (hint: use the copy icon at the top right):

```
# hot 100 download
download.file("https://github.com/utdata/rwd-billboard-data/blob/main/data-process/hot-100/hot100
```

This `download.file` function takes at least two arguments:

- The URL of the file you are downloading
- The path and name of where you want to save it.

Note those two arguments are in quotes. The path includes the folder name you are saving the file to, which we called `hot-stuff.csv`.

When you run this, it should save the file and then give you output similar to this:

```
trying URL 'https://github.com/utdata/rwd-billboard-data/blob/main/data-process/hot-100/hot100-or
Content type 'text/plain; charset=utf-8' length 45795374 bytes (43.7 MB)
==================================================
downloaded 43.7 MB
```

That's a pretty big file.

3.4 About data sources

Depending on the data source, importing can be brilliantly easy or a major pain in the rear. It all depends on how well-formatted is the data.

In this class, we will primarily use data from CSVs (Comma Separated Value), Excel files and APIs (Application Programming Interface).

- **CSVs** are a kind of lowest-common-denominator for data. Most any database or program can import or export them. It's just text with a **,** between each value.
- **Excel** files are good, but are often messy because humans get involved. They often have multiple header rows, columns used in multiple ways, notes added, etc. Just know you might have to clean them up before or after importing them.
- **APIs** are systems designed to respond to programming. In the data world, we often use the APIs by writing a query to ask a system to return a selection of data. By definition, the data is well structured. You can often determine the file type of the output as part of the API call, including ...
- **JSON** (or JavaScript Object Notation) is the data format preferred by JavaScript. R can read it, too. It is often the output format of APIs, and prevalent enough that you need to understand how it works. We'll get into that later in semester.

Don't get me wrong ... there are plenty of other data types and connections available through R, but those are the ones we'll deal with most in this book.

3.5 Our project data

Now that we've downloaded the data and talked about what data is, lets talk about our Billboard data specifically.

The data includes the Billboard's Weekly Hot 100 singles charts from its inception on 8/2/1958 through 2020. It is a modified version of data compiled by SEAN MILLER and posted on data.world. We are using a copy I have saved.

When you write about this data (and you will), you should source it as **the Billboard Hot 100 from Billboard Media**, since that is where the data comes from via an API.

3.6 Data dictionary

This data contains weekly Hot 100 singles chart from Billboard.com. **Each row of data represents a song and the corresponding position on that week's chart.** Included in each row are the following elements:

- Billboard Chart URL: Website for the chart
- WeekID: Basically the date
- Song name
- Performer name
- SongID: Concatenation of song & performer
- Current week on chart
- Instance: This is used to separate breaks on the chart for a given song. For example, an instance of 6 tells you that this is the sixth time this song has fallen off and then appeared on the chart
- Previous week position
- Peak Position: As of the current week
- Weeks on Chart: As of the current week

Let's import it so we can *see* the data.

3.7 Import the data

Since we are doing a new thing, we should note that with a Markdown headline and text.

1. Add a Markdown headline: `## Import data`
2. Add some text to explain that we are importing the Billboard Hot 100 data.
3. After your description, add a new code chunk (*Cmd+Option+i*).

We'll be using the `read_csv()` function from the tidyverse readr package, which is different from `read.csv` that comes with base R. `read_csv()` is mo betta.

Inside the function we put in the path to our data, inside quotes. If you start typing in that path and hit tab, it will complete the path. (Easier to show than explain).

1. Add the follow code into your chunk and run it.

```
read_csv("data-raw/hot-stuff.csv")
```

   Note the path is in quotes.

You get two results printed to your screen.

The first result called **"R Console"** shows what columns were imported and the data types. It's important to review these to make sure things happened

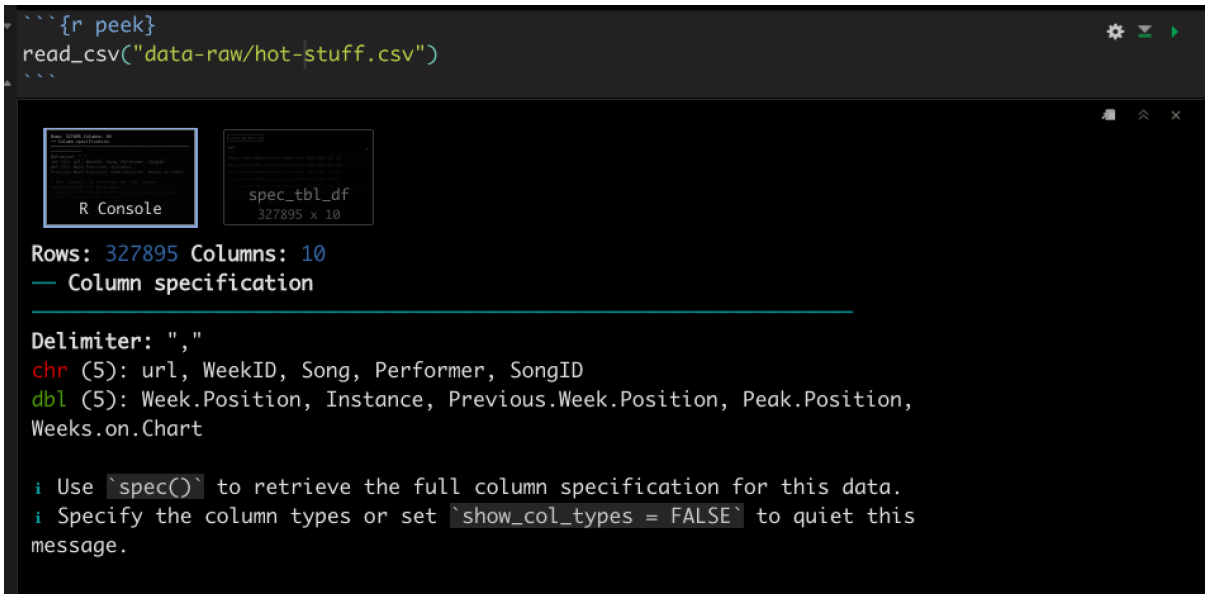


Figure 3.4: RConsole output

the way that expected. In this case it noted which columns came in as text (`chr`), or numbers (`dbl`).

Note: **Red** colored text in this output is NOT an indication of a problem.

The second result **spec_tbl_df** prints out the data like a table. The data object is a tibble, which is a fancy tidyverse version of a "data frame."

> I will use the term tibble and data frame interchangably. Think of tibbles and data frames like a well-structured spreadsheet. They are organized rows of data (called observations) with columns (called variables) where every column is a specific data type.

When we look at the data output into RStudio, there are several things to note:

- Below each column name is an indication of the data type. This is important.
- You can use the arrow icon on the right to page through the additional columns.
- You can use the paging numbers and controls at the bottom to page through the rows of data.
- The number of rows and columns is displayed.

Of special note here, we have only printed this data to the screen. We have not saved it in any way, but that is next.

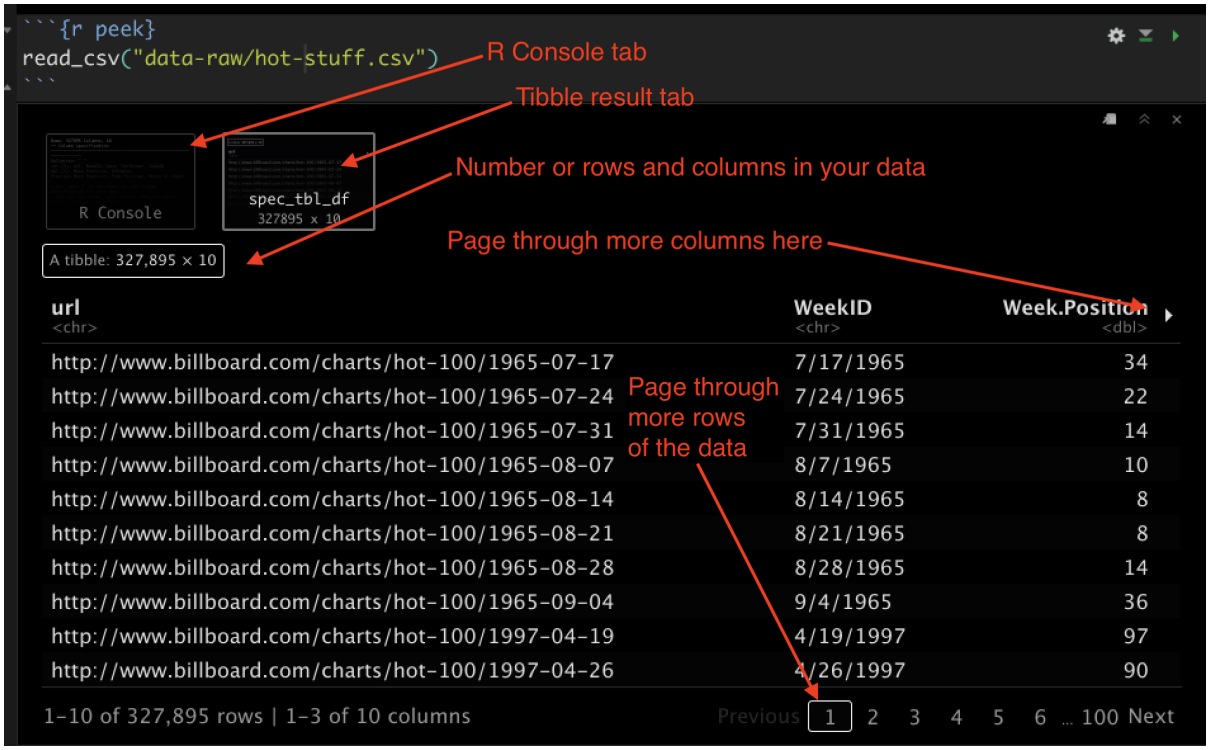


Figure 3.5: Data output

3.8 Assign our import to a tibble

As of right now, we've only printed the data to our screen. We haven't "saved" it at all. Next we need to assign it to an **R object** so it can be named thing in our project environment so we can reuse it. We don't want to re-import the data every time we use the data.

The syntax to create an object in R can seem weird at first, but the convention is to name the object first, then insert stuff into it. So, to create an object, the structure is this:

```
# this is pseudo code. don't run it.
new_object <- stuff_going_into_object
```

Let's make a object called `hot100` and fill it with our imported tibble.

1. Edit your existing code chunk to look like this. You can add the `<-` by using *Option+-* as in holding down the Option key and then pressing the hyphen:

```
hot100 <- read_csv("data-raw/hot-stuff.csv")
```

Run that chunk and several things happen:

- We no longer see the result of the data printed to the screen. That's because we created a tibble instead of printing it to the screen. You do get the RConsole output.
- In the **Environment** tab at the top-right of RStudio, you'll see the `hot100` object listed.
  - Click on the blue play button next to ratings and it will expand to show you a summary of the columns.
  - Click on the name and it will open a "View" of the data in another window, so you can look at it in spreadsheet form. You can even sort and filter it.
- Once you've looked at the data, close the data view with the little `x` next to the tab name.

3.8.1 Print a peek to the screen

Since we can't see the data after we assign it, let's print the object to the screen so we can refer to it.

1. Edit your import chunk to add the last two lines of this, including the one with the `#`:

```
hot100 <- read_csv("data-raw/hot100.csv")

# peek at the data
hot100
```

> You can use the green play button at the right of the chunk, or preferrably have your cursor inside the chunk and do *Cmd+Shift+Return* to run all lines. (*Cmd+Return* runs only the current line.)

This prints your saved tibble to the screen.

The line with the `#` is a comment *within* the code chunk. Commenting what your code is important to your future self, and sometimes we do that within the code chunk instead of markdown if it will be more clear.

3.8.2 Glimpse the data

There is another way to peek at the data that I prefer because it is more compact and shows you all the columns and data examples without scrolling: `glimpse()`.

1. In your existing chunk, edit the last line to add the `glimpse()` function as noted below.

I'm showing the return here as well. Afterward I'll explain the pipe: `%>%`.

```
hot100 <- read_csv("data-raw/hot-stuff.csv")

# peek at the data
hot100 %>% glimpse()
```

```
## Rows: 327,895
## Columns: 10
## $ url                   <chr> "http://www.billboard.com/charts/hot-100/1965-0~
## $ WeekID                <chr> "7/17/1965", "7/24/1965", "7/31/1965", "8/7/196~
## $ Week.Position         <dbl> 34, 22, 14, 10, 8, 8, 14, 36, 97, 90, 97, 97, 9~
## $ Song                  <chr> "Don't Just Stand There", "Don't Just Stand The~
## $ Performer             <chr> "Patty Duke", "Patty Duke", "Patty Duke", "Patt~
## $ SongID                <chr> "Don't Just Stand TherePatty Duke", "Don't Just~
## $ Instance              <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
```

```
## $ Previous.Week.Position <dbl> 45, 34, 22, 14, 10, 8, 8, 14, NA, 97, 90, 97, 9~
## $ Peak.Position          <dbl> 34, 22, 14, 10, 8, 8, 8, 8, 97, 90, 90, 90, 90,~
## $ Weeks.on.Chart         <dbl> 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4, 5, 6, 1, ~
```

Here you get the RConsole printout (hidden here for clarity), plus the glimpse shows there are 300,000+ rows and 10 columns in our data. Each column is then listed out with its data type and the first several values in that column.

3.8.3 About the pipe %>%

We need to break down this code a little: `hot100 %>% glimpse()`.

We are starting with the tibble `hot100`, but then we follow it with `%>%`, which is called a pipe. It is a tidyverse tool that allows us to take the **results** of an object or function and pass them into another function. Think of it like a sentence that says **"AND THEN" the next thing**.

It might look like there are no arguments inside `glimpse()`, but what we are actually doing is passing the `hot100` tibble into it.

You can't start a new line with a pipe. If you are breaking into multiple lines, but the `%>%` at the end.

> IMPORTANT: There is a keyboard command for the pipe `%>%`: **Cmd+Shift+m**. Learn that one.

3.8.4 What is clean data

The "Checking Your Data" section of this DataCamp tutorial has a good outline of what makes good data, but in general it should:

- Have a single header row with well-formed column names.
  - One column name for each column. No merged cells.
  - Short names are better than long ones.
  - Spaces in names make them harder to work with. Use and `_` or `.` between words. I prefer `_` and lowercase text.
- Remove notes or comments from the files.
- Each column should have the same kind of data: numbers vs words, etc.
- Each row should be a single thing called an "observation." The columns should describe attributes of that observation.

Data rarely comes clean like that. There can be many challenges importing and cleaning data. We'll face some of those challenges here. In our case our columns names could use help, and our field `WeekID` is not really a date, but text characters. We'll tackle those issues next.

3.9 Cleaning column names

So, given those notes above, we should clean up our column names. This is why we have included the janitor package, which includes a neat function called `clean_names()`

1. Edit the first line of your chunk to add a pipe and the clean_names function: `%>% clean_names()`

```
hot100 <- read_csv("data-raw/hot-stuff.csv") %>% clean_names()

# peek at the data
hot100 %>% glimpse()
```

```
## Rows: 327,895
## Columns: 10
## $ url                    <chr> "http://www.billboard.com/charts/hot-100/1965-0~
## $ week_id                <chr> "7/17/1965", "7/24/1965", "7/31/1965", "8/7/196~
## $ week_position          <dbl> 34, 22, 14, 10, 8, 8, 14, 36, 97, 90, 97, 97, 9~
## $ song                   <chr> "Don't Just Stand There", "Don't Just Stand The~
## $ performer              <chr> "Patty Duke", "Patty Duke", "Patty Duke", "Patt~
## $ song_id                <chr> "Don't Just Stand TherePatty Duke", "Don't Just~
## $ instance               <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
## $ previous_week_position <dbl> 45, 34, 22, 14, 10, 8, 8, 14, NA, 97, 90, 97, 9~
## $ peak_position          <dbl> 34, 22, 14, 10, 8, 8, 8, 8, 97, 90, 90, 90, 90,~
## $ weeks_on_chart         <dbl> 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4, 5, 6, 1, ~
```

This function has cleaned up your names, making them all lowercase and using _ instead of periods between words. Believe me when I say this is helpful later to auto-complete column names when you are writing code.

3.10 Fixing the date

Dates are a tricky datatype because you can do math with them. To use them properly in R we need to convert them from the text we have here to a *real date*.

Converting dates can be a pain, but the tidyverse universe has a package called lubridate that can help us with that.

Since we are doing something new, we want to start a new section in our notebook and explain what we are doing.

1. Add a headline: `## Fix our dates`.

2. Add some text that you are using lubridate to create a new column with a real date.
3. Add a new code chunk. Remember *Cmd+Option+i* will do that.

We are going to start by creating a new data frame that is the same as our current on, and then add a glimpse so we can see the results as we build upon it.

1. Add the following inside your code chunk.

```
# part we will build on
hot100_date <- hot100

# peek at the result
hot100_date %>% glimpse()
```

```
## Rows: 327,895
## Columns: 10
## $ url                   <chr> "http://www.billboard.com/charts/hot-100/1965-0~
## $ week_id               <chr> "7/17/1965", "7/24/1965", "7/31/1965", "8/7/196~
## $ week_position         <dbl> 34, 22, 14, 10, 8, 8, 14, 36, 97, 90, 97, 97, 9~
## $ song                  <chr> "Don't Just Stand There", "Don't Just Stand The~
## $ performer             <chr> "Patty Duke", "Patty Duke", "Patty Duke", "Patt~
## $ song_id               <chr> "Don't Just Stand TherePatty Duke", "Don't Just~
## $ instance              <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
## $ previous_week_position <dbl> 45, 34, 22, 14, 10, 8, 8, 14, NA, 97, 90, 97, 9~
## $ peak_position         <dbl> 34, 22, 14, 10, 8, 8, 8, 8, 97, 90, 90, 90, 90,~
## $ weeks_on_chart        <dbl> 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4, 5, 6, 1, ~
```

A refresher to break this down:

- I have a comment starting with `#` to explain the first part of the code
- We are taking the `hot100` object and pushing it into a new object called `hot100_date`.
- I have a blank line for clarity
- Another comment
- We glimpse the new `hot100_date` object so we can see changes as we work on it.

  To be clear, we haven't changed any data yet. We just created a new tibble like the old tibble.

3.10.1 Working with mutate()

We are going to replace our current date field `week_id` with a converted date. We use a dplyr function `mutate()` to do this, with some help from lubridate.

> dplyr is the tidyverse package of functions to manipulate data. We'll use it a lot. It is loaded with the `library(tidyverse)`.

Let's explain how mutate works first. Mutate changes every value in a column. We can either create a new column or overwrite an existing one.

Within the mutate function, we name the new thing first (confusing I know) and the set the value of that new thing.

```
# this is psuedo code. Don't run it.
data %>%
  mutate(
    newcol = oldcol
  )
```

That new value could be arrived at through math or any combination of other functions. In our case, we want to convert our old text-based date to a *real date*, and then assign it back to the "new" column, **but really we are overwriting the existing one**.

Some notes about the above:

- It might seem weird to list the new thing first when we are changing it, but that is how R works in this case. You'll see that pattern elsewhere, like we have already with assigning data into tibbles.
- We need to be careful when we overwrite data. In this case I feel comfortable doing so because we are creating a new tibble at the same time, so I still have my original data in my project.
- I strategically used returns to make the code more readable. This code would work the same if it were all on the same line, but writing it this way helps me understand it. RStudio will help you indent properly this as you type. (Easier to show than explain.)

1. Edit your chunk to add the changes below and run it. I **implore** you to *type* the changes so you see how RStudio helps you write it. Use tab completion, etc.

```
# part we will build on
hot100_date <- hot100 %>%
  mutate(
```

```
    week_id = mdy(week_id)
  )

# peek at the result
hot100_date %>% glimpse()
```

```
## Rows: 327,895
## Columns: 10
## $ url                    <chr> "http://www.billboard.com/charts/hot-100/1965-0~
## $ week_id                <date> 1965-07-17, 1965-07-24, 1965-07-31, 1965-08-07~
## $ week_position          <dbl> 34, 22, 14, 10, 8, 8, 14, 36, 97, 90, 97, 97, 9~
## $ song                   <chr> "Don't Just Stand There", "Don't Just Stand The~
## $ performer              <chr> "Patty Duke", "Patty Duke", "Patty Duke", "Patt~
## $ song_id                <chr> "Don't Just Stand TherePatty Duke", "Don't Just~
## $ instance               <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
## $ previous_week_position <dbl> 45, 34, 22, 14, 10, 8, 8, 14, NA, 97, 90, 97, 9~
## $ peak_position          <dbl> 34, 22, 14, 10, 8, 8, 8, 8, 97, 90, 90, 90, 90,~
## $ weeks_on_chart         <dbl> 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4, 5, 6, 1, ~
```

What we did there with your `mutate()` function was name our **new** column but we used the name of an existing one `week_id` so it will really **replace** the data. We replaced it with a lubridate function which I'll explain next.

Lubridate allows us to parse text and then turn it into a date if we supply the order of the date values in the original data.

- Our original date was something like "07/17/1965." That is month, followed by day, followed by year.
- The lubridate function `mdy()` converts that text into a *real* date, which properly shows as YYYY-MM-DD, or year then month then day. Lubridate is smart enough to figure out if you have `/` or `-` between your values in the original date.

If your original text is in a different date order, then you look up what function you need. I typically use the **cheatsheet** that you'll find on the lubridate page. You'll find them in the PARSE DATE-TIMES section.

3.11 Arrange the data

If you inspect our newish `week_id` in your glimpse return, you'll notice the first record starts in "1965-07-17" but our data goes back to 1958. We want to sort our data by the oldest records first using `arrange()`.

We will use the `%>%` and then the arrange function, feeding it our data (implied with the pipe) and the columns we wish to sort by.

1. Edit your chunk to the following to add the `arrange()` function:

```
# part we will build on
hot100_date <- hot100 %>%
  mutate(
    week_id = mdy(week_id)
  ) %>%
  arrange(week_id, week_position)

# peek at the result
hot100_date %>% glimpse()
```

```
## Rows: 327,895
## Columns: 10
## $ url                   <chr> "http://www.billboard.com/charts/hot-100/1958-0~
## $ week_id               <date> 1958-08-02, 1958-08-02, 1958-08-02, 1958-08-02~
## $ week_position         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~
## $ song                  <chr> "Poor Little Fool", "Patricia", "Splish Splash"~
## $ performer             <chr> "Ricky Nelson", "Perez Prado And His Orchestra"~
## $ song_id               <chr> "Poor Little FoolRicky Nelson", "PatriciaPerez ~
## $ instance              <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
## $ previous_week_position <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ peak_position         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~
## $ weeks_on_chart        <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
```

Now when you look at the glimpse, the first record is from "1958-08-02" and the first `week_position` is "1," which is the top of the chart.

Just to see this all clearly in table form, we'll print the top of the table to our screen so we can see it.

1. Add a line of text in your notebook explaining your are looking at the table.
2. Add a new code chunk and add the following.

   The result will look different in your notebook vs this book.

```
hot100_date %>% head(10)
```

```
## # A tibble: 10 x 10
##    url        week_id    week_position song    performer    song_id     instance
##    <chr>      <date>             <dbl> <chr>   <chr>        <chr>          <dbl>
##  1 http://ww~ 1958-08-02             1 Poor L~ Ricky Nelson Poor Littl~        1
##  2 http://ww~ 1958-08-02             2 Patric~ Perez Prado~ PatriciaPe~        1
```

```
##  3 http://ww~ 1958-08-02             3 Splish~ Bobby Darin  Splish Spl~        1
##  4 http://ww~ 1958-08-02             4 Hard H~ Elvis Presl~ Hard Heade~        1
##  5 http://ww~ 1958-08-02             5 When    Kalin Twins  WhenKalin ~        1
##  6 http://ww~ 1958-08-02             6 Rebel-~ Duane Eddy ~ Rebel-'rou~        1
##  7 http://ww~ 1958-08-02             7 Yakety~ The Coasters Yakety Yak~        1
##  8 http://ww~ 1958-08-02             8 My Tru~ Jack Scott   My True Lo~        1
##  9 http://ww~ 1958-08-02             9 Willie~ The Johnny ~ Willie And~        1
## 10 http://ww~ 1958-08-02            10 Fever   Peggy Lee    FeverPeggy~        1
## # ... with 3 more variables: previous_week_position <dbl>, peak_position <dbl>,
## #   weeks_on_chart <dbl>
```

This just prints the first 10 lines of the data.

1. Use the arrows to look at the other columns of the data (which you can't see in the book).

3.12 Selecting columns

We don't need all of these columns for our analysis, so we are going to **select** only the ones we need. This will make our exported data file smaller. To understand the concept, you can review the Select video in the Basic Data Journalism Functions series.

It boils down to this: We are selecting only the columns we need. In doing so, we will drop `url`, `song_id` and `instance`.

1. Add a Markdown headline: `## Selecting columns`.
2. Explain in text we are tightening the date to only the columns we need.
3. Add the code below and then I'll explain it.

```
hot100_tight <- hot100_date %>%
  select(
    -url,
    -song_id,
    -instance
    )

hot100_tight %>% glimpse()
```

```
## Rows: 327,895
## Columns: 7
## $ week_id               <date> 1958-08-02, 1958-08-02, 1958-08-02, 1958-08-02~
## $ week_position         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~
```

```
## $ song                   <chr> "Poor Little Fool", "Patricia", "Splish Splash"~
## $ performer              <chr> "Ricky Nelson", "Perez Prado And His Orchestra"~
## $ previous_week_position <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ peak_position          <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~
## $ weeks_on_chart         <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
```

In our code we:

- Name our new tibble
- Assign to a result of the hot100_date tibble
- In that tibble, we use the select statement to remove (using `-`) certain columns.

Alternately, we could just name the columns we want to keep without the `-` sign. But there were fewer to remove than keep.

> To be clear, there are other ways to use `select()` this that use less code, but I want to be as straigtforward as possible at this point. It's pretty powerful.

3.13 Exporting data

3.13.1 Single-notebook philosophy

I have a pretty strong opinion that you should be able to open any RNotebook in your project and run it from top to bottom without it breaking. In short, one notebook should not be dependent on the previous running of another open notebook.

This is why I had you name this notebook `01-import.Rmd` with a number 1 at the beginning. We'll number our notebooks in the order they should be run. It's an indication that before we can use the notebook `02-analysis` (next lesson!) that the `01-import.Rmd` notebook has to be run first.

> I use `01-` instead of just `1-` in case there are more than nine notebooks. I want them to appear in order in my files directory. I'm anal retentive.

So we will create an exported file from our first notebook that can be used in the second one. Once we create that file, the second notebook can be opened and run at any time.

Why make this so complicated?

The answer is **consistency**. When you follow the same structure with each project, you quickly know how to dive into that project at a later date. If everyone on your team uses the same structure, you can dive into your teammates code because you already know how it is organized. If we separate our importing/cleaning into it's own file to be used by many other notebooks, we can fix future cleaning problems in ONE place instead of many places.

One last example to belabor the point: It can save time. I've had import/cleaning notebooks that took 20 minutes to process. Imagine if I had to run that every time I wanted to rebuild my analysis notebook. Instead, the import notebook spits out clean file that can be imported in a fraction of that time.

This was all a long-winded way of saying we are going to export our data now.

3.13.2 Exporting as rds

We are able to pass cleaned data between notebooks because of a native R data format called `.rds`. When we export in this format it saves not only rows and columns, but also the data types. (If we exported as CSV, we would potentially have to re-fix the date or other data types when we imported again.)

We will use another readr function called `write_rds()` to create our file to pass along to the next notebook, saving the data into the `data-processed` folder we created earlier. We are separating it from our data-raw folder because "Thou shalt not change raw data" even by accident. By always writing data to this different folder, we help avoid accidentally overwriting our original data.

1. Create a Markdown headline `## Exports` and write a description that you are exporting files to .rds.
2. Add a new code chunk and add the following code:

```
hot100_tight %>%
   write_rds("data-processed/01-hot100.rds")
```

So, we are starting with the `hot100_tight` tibble that we saved earlier. We then pipe `%>%` the result of that into a new function `write_rds()`. In addition to the data, the function needs to where to save the file, so in quotes we give the path to where and what we want to call the file: `"data-processed/hot100.rds"`.

Remember, we are saving in data-processed because we never export into data-raw. We are naming the file starting with `01-` to indicate to our future selves that this output came from our first notebook. We then name it, and use the `.rds` extension.

3.14 Naming chunks

I didn't want to break our flow of work to explain this earlier, but I want you to name all your chunks so you can use a nice feature in RStudio to jump up and down your notebook.

Let me show you and example of why first. Look at the bottom of your window above the console and you'll see a dropdown window. Click on that.

Here is mine, but yours wlll be different:

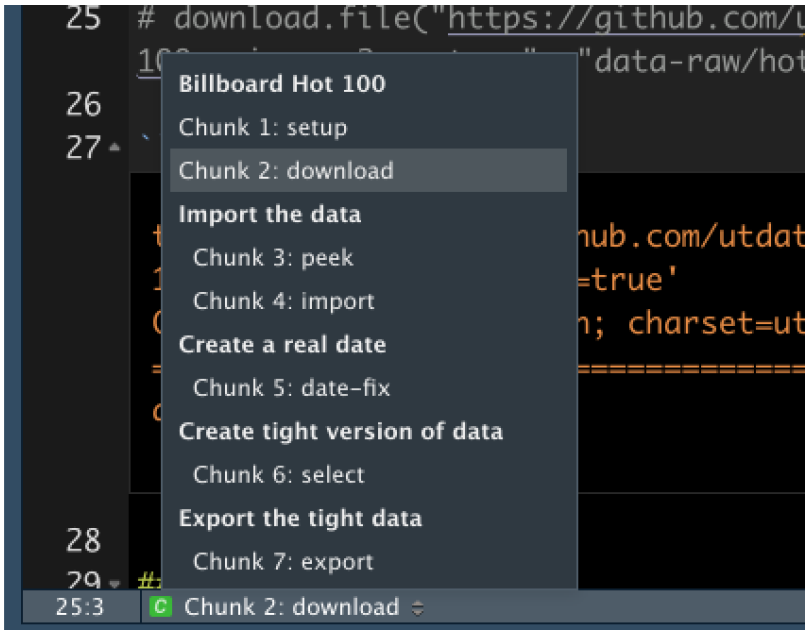


Figure 3.6: RStudio bookmarks

You'll notice that my chunks have names, but yours probably don't. It's pretty helpful have these names so you know what the chunk does. You can use this menu to skip up and down the notebook.

How to name a chunk? Well, I can't show you in code because it is not rendered in the book, but here is a picture:

See where I have `{r download}`? I named it that because that is what the chunk does.

- Chunk names can't have spaces. Use a single word or `-` or `_` between words.
- There are other configurations we can do here, but that is for later.

Figure 3.7: Named chunks

1. Go back through your notebook and name all your chunks.
2. Under the **Run** menu, choose *Restart R and run all chunks*.

Make sure that your Notebook ran all the way from top to bottom. The order of stuff in the notebook matters and you can make errors as you edit up and down the notebook. You **always** want to do this before you finish a notebook.

3.15 Knit your page

Lastly, we want to Knit your notebook so you can see the pretty HTML verison.

1. Next to the **Preview** menu in the notebook tool bar, click the little drop-down to see the knitting options.
2. Choose **Knit to HTML**.

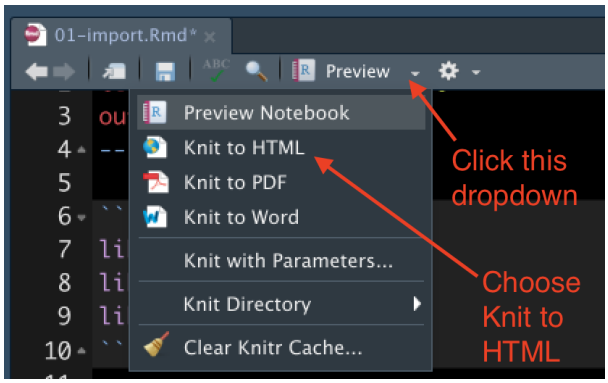


Figure 3.8: Knit to HTML

After you do this, the menu will probably change to just **Knit** and you can just click on it to knit again.

This will open a nice reader-friendly version of your notebook. You could send that file (called `01-import.html`) to your editor and they could open it in a web browser.

I use these knit files to publish my work on Github, but it is a bit more involved to do all that so we'll skip it at least for now.)

3.16 Review of what we've learned so far

Most of this lesson has been about importing and combining data, with some data mutating thrown in for fun. (OK, I have an odd sense of what fun is.) Importing data into R (or any data science program) can sometimes be quite challenging, depending on the circumstances. Here we were working with well-formed data, but we still used quite a few tools from the tidyverse universe like readr (read_csv, write_rds) and dplyr (select, mutate).

Here are the functions we used and what they do. Most are linked to documentation sites:

- `install.packages()` downloads an R package to your computer. Typically executed from within the Console and only once per computer. We installed the tidyverse, janitor and lubridate packages.
- `library()` loads a package. You need it for each package in each notebook, like `library(tidyverse)`.
- `read_csv()` imports a csv file. You want that one, not `read.csv`.
- `clean_names()` is a function in the janitor package that standardizes column names.
- `glimpse()` is a view of your data where you can see all of the column names, their data type and a few examples of the data.
- `head()` prints the first 6 rows of your data unless you specify a different integer within the function.
- `mutate()` changes data. You can create new columns or overwrite existing ones.
- `mdy()` is a lubridate function to convert text into a date. There are other functions for different date orders.
- `select()` selects columns in your tibble. You can list all the columns to keep, or use `-` to remove columns. There are many variations.
- `write_rds()` writes data out to a file in a format that preserves data types.

3.17 What's next

Importing data is just the first step of exploring a data set. We'll work through the next chapter before we turn in any work on this.

Please reach out to me if you have questions on what you've done so far. These are important skills you'll use on future projects.

Chapter 4

Summarize with count - analysis

This chapter continues the Billboard Hot 100 project. In the previous chapter we downloaded, imported and cleaned the data. We'll be working in the same project.

4.1 Goals of this lesson

- To use group by/summarize/arrange combination to count rows.
- To use filter to both focus data for summaries, and to logically end summary lists.
- Introduce the shortcut `count()` function, along with complex filters.

4.2 The questions we'll answer

Now that we have the Billboard Hot 100 charts data in our project it's time to find the answers to the following questions:

- Which performers had the most appearances on the Hot 100 chart at any position?
- Which performer/song combination has been on the charts the most number of weeks at any position?
- Which performer/song combination was No. 1 for the most number of weeks?
- Which performer had the most songs reach No. 1?

- Which performer had the most songs reach No. 1 in the most recent five years?
- Which performer had the most Top 10 hits overall?

  What are your guesses for the questions above? No peeking!

Before we can get into the analysis, we need to set up a new notebook.

4.3 Setting up an analysis notebook

At the end of the last notebook we exported our clean data as an `.rds` file. We'll now create a new notebook and import that data. It will be much easier this time.

1. If you don't already have it open, go ahead and open your Billboard project.
2. If your import notebook is still open, go ahead and close it.
3. Use the `+` menu to start a new **RNotebook*.
4. Update the title as "Billboard analysis" and then remove all the boilerplate text below the YAML metadata.
5. Save the file as `02-analysis.Rmd` in your project folder.
6. Check your Environment tab (top right) and make sure the Data pane is empty. We don't want to have any leftover data. If there is, then go under the **Run** menu and choose **Restart R and Clear Output**.

Since we are starting a new notebook, we need to set up a few things. First up we want to list our goals.

1. Add a headline and text describing the goals of this notebook. You are exploring the Billboard Hot 100 charts data.
2. Go ahead and copy all the questions outlined above into your notebook.
3. Start each line with a `-` or `*` followed by a space.
4. Now add a headline (two hashes) called Setup.
5. Add a chunk, also name it "setup" and add the tidyverse library.
6. Run the chunk to load the library.

```
library(tidyverse)
```

4.3.1 Import the data on your own

In this next part I want you to think about how you've did the import in the last notebook and I want you to:

1. Write a section to import the data using `read_rds()` and put it into a tibble called `hot100`.

Yes, it is true that we haven't talked about `read_rds()` yet but it works exactly the same way as `read_csv()`, so you should try to figure it out.

Here are some hints and guides:

- Start a new section with a headline and text to say what you are doing
- Don't forget to name your code chunk (this should all be getting familiar).
- `read_rds()` works the same was as `read_csv()`. The path *should* be `data-processed/01-hot100.rds` if you did the previous notebook properly.
- Remember the tibble needs to be named first and read data pushed into it.
- Add a glimpse to the chunk so you can refer to the data.

Try real hard first before clicking here for the answer

```
hot100 <- read_rds("data-processed/01-hot100.rds")

# peek at the data
hot100 %>% glimpse()
```

```
## Rows: 327,895
## Columns: 7
## $ week_id                <date> 1958-08-02, 1958-08-02, 1958-08-02, 1958-08-02~
## $ week_position          <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~
## $ song                   <chr> "Poor Little Fool", "Patricia", "Splish Splash"~
## $ performer              <chr> "Ricky Nelson", "Perez Prado And His Orchestra"~
## $ previous_week_position <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ peak_position          <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~
## $ weeks_on_chart         <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
```

4.4 Introducing dplyr

One of the packages within the tidyverse is dplyr. Dplyr allows us to transform our data frames in ways that let us explore the data and prepare it for visualizing. It's the R equivalent of common Excel functions like sort, filter and pivoting.

There is a cheatsheet on the dplyr that you might find useful.

We've used `select()`, `mutate()` and `arrange()` already, but we'll introduce more dplyr functions in this chapter.

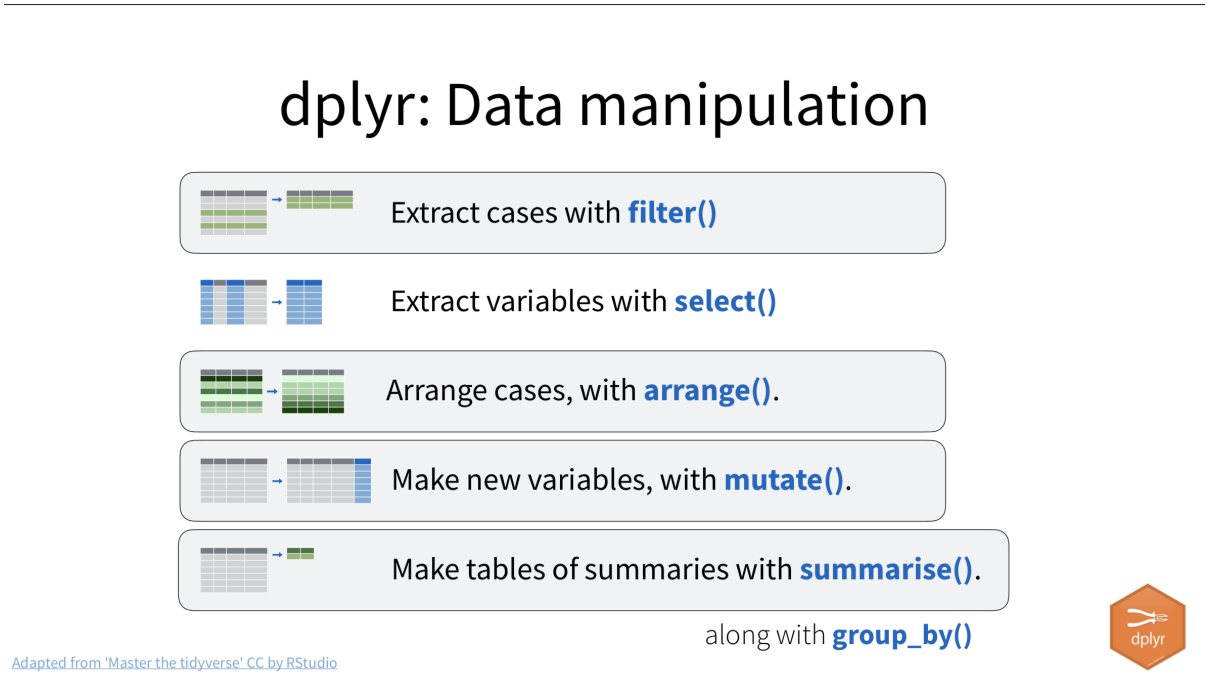


Figure 4.1: dplyr. Images courtesy Hadley and Charlotte Wickham

4.5 Most appearances

Our first question: Which performers had the most appearances on the Hot 100 chart at any position?

4.5.1 Group & Aggregate

Before we dive into the code, let's review this video about "Group and Aggregate" to get a handle on the concept.

Let's work through the logic of what we need to do to get our answer before I explain exactly how.

- Each row in the data is one song on the chart.
- Each of those rows has the `performer` which is the person(s) who performed it.
- To figure out how many times a performer is in the data, we need to count the rows with the same performer.

We'll use the tidyverse's version of Group and Aggregate to get this answer. It is actually two different functions within dplyr that often work together: `summarize()` and it's companion `group_by()`.

4.5.2 Summarize

> `summarize()` and `summarise()` are the same function, as R supports both the American and UK spelling of summarize. I don't care which you use.

We'll start with `summarize()` first because it can stand alone.

The `summarize()` function **computes tables *about* your data**. Our logic above has us wanting a "summary" of how many times certain performers appear in data, hence we use this function.

Here is an example in a different context:

Much like the `mutate()` function we used earlier, we list the name of the new column first, then assign to it the function we want to accomplish using `=`.

The example above is giving us two summaries: It is applying a function `mean()` (or average) on all the values in the `lifeExp` column, and then again with `min()`, the lowest life expectancy in the data.

Let me show you a similar example with our data answer this question:

Let's find the average "peak_position" of all songs on the charts through history:

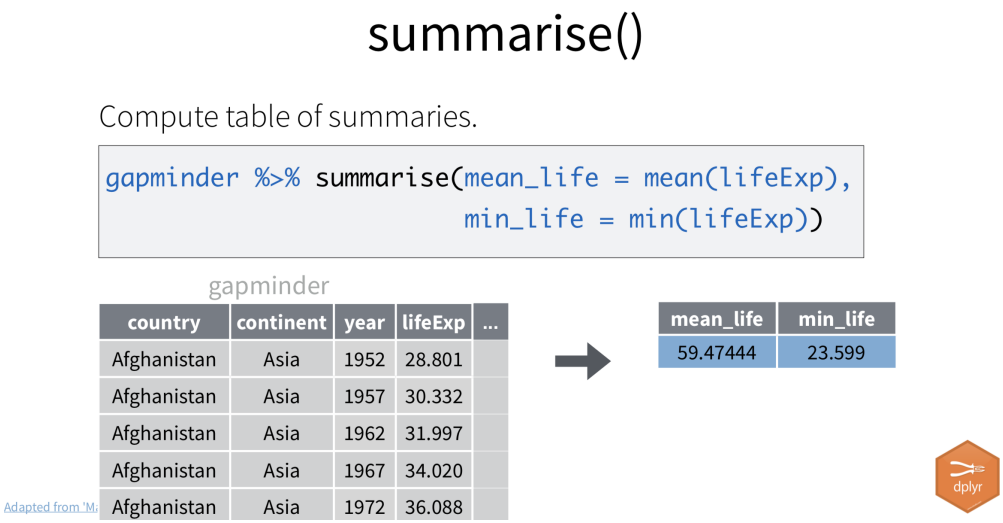


Figure 4.2: Learn about your data with summarize()

```
hot100 %>%
  summarize(mean_position = mean(peak_position))
```

```
## # A tibble: 1 x 1
##   mean_position
##           <dbl>
## 1          41.4
```

Meaning the average song on the charts tops out at No. 41.

> This is an admittedly simplistic view of the average `peak_position` since the same song will be listed multiple times with possibly new `peak_positions`, but hopefully you get the idea.

But in our case we want to **count** the number of rows, and there is a function for that: `n()`. (Think "number of observations or rows.")

Let's write the code and run it on our code, then I'll explain:

1. Set up a new section with a headline, text and empty code chunk.
2. Inside the code chunk, add the following:

```
hot100 %>%
  summarize(appearances = n())
```

```
## # A tibble: 1 x 1
##   appearances
##         <int>
## 1      327895
```

- We start with the tibble first and then pipe into `summarize()`.
- Within the function, we define our summary:
  - We name the new column "appearances" because that is a descriptive column name for our result.
  - We set that new column to count the **n**umber of rows.

Basically we are summarizing the total number of rows in the data.

AN ASIDE: I often break up the inside a `summarize()` into new lines so they are easier to read.

```
# you don't have to do this here, but know
# it is helpful when you have more than one summary
hot100 %>%
  summarize(
    appearances = n()
  )
```

But your are asking: Professor, we want to count the performers, right?

This is where `summarize()`'s close friend `group_by()` comes in.

4.5.3 Group by

Here is a weird thing about `group_by()`: It is always followed by another function. It really just pre-sorts data into groups so that whatever function is applied after happens within each individual group.

If add a `group_by()` on our performers before our summarize function, it will put all of the "Aerosmith" rows together, then all the "Bad Company" rows together, etc. and then we count the rows *within* those groups.

1. Modify your code block to add the group_by:

```
hot100 %>%
  group_by(performer) %>%
  summarize(appearances = n())
```

```
## # A tibble: 10,061 x 2
##    performer                                appearances
##    <chr>                                          <int>
##  1 "? (Question Mark) & The Mysterians"              33
##  2 "'N Sync"                                        172
##  3 "'N Sync & Gloria Estefan"                        20
##  4 "'N Sync Featuring Nelly"                         20
##  5 "'Til Tuesday"                                    53
##  6 "\"Groove\" Holmes"                               14
##  7 "\"Little\" Jimmy Dickens"                        10
##  8 "\"Pookie\" Hudson"                                1
##  9 "\"Weird Al\" Yankovic"                           91
## 10 "(+44)"                                            1
## # ... with 10,051 more rows
```

What we get in return is a **summarize**d table that shows all 10,000+ different performers that have been on the charts, and **n**umber of rows in which they appear in the data.

That's great, but who had the most?

4.5.4 Arrange the results

Remember in our import notebook when we had to sort the songs by date. We'll use the same `arrange()` function here, but we'll change the result to **desc**ending order, because journalists almost always want to know the *most* of something.

1. Add the pipe and arrange function below and run it, then I'll explain.

```
hot100 %>%
  group_by(performer) %>%
  summarize(appearances = n()) %>%
  arrange(appearances %>% desc())
```

```
## # A tibble: 10,061 x 2
##    performer     appearances
##    <chr>               <int>
##  1 Taylor Swift         1022
##  2 Elton John            889
```

```
##  3 Madonna               857
##  4 Kenny Chesney         758
##  5 Drake                 746
##  6 Tim McGraw            731
##  7 Keith Urban           673
##  8 Stevie Wonder         659
##  9 Rod Stewart           657
## 10 Mariah Carey          621
## # ... with 10,051 more rows
```

- We added the `arrange()` function and fed it the column of "appearances." If we left it with just that, then it would list the smallest values first.
- *Within the arrange function* we piped the "appearances" part into another function: `desc()` to change the order.

So if you read that line in English it would be "arrange by (appearances AND THEN descending order)."

You may also see this as `arrange(desc(appearances))`.

4.5.5 Get the top of the list

We've printed 10,000 rows of data into our notebook when we really only wanted the Top 10 or so. You might think it doesn't matter, but your knitted HTML file will store all that data and can make it a big file (like in megabytes), so I try to avoid that when I can.

We can use the `head()` command again to get our Top 10.

1. Pipe the result into `head()` function set to 10 rows.

```
hot100 %>%
  group_by(performer) %>%
  summarize(appearances = n()) %>%
  arrange(appearances %>% desc()) %>%
  head(10)
```

```
## # A tibble: 10 x 2
##    performer     appearances
##    <chr>               <int>
##  1 Taylor Swift         1022
##  2 Elton John            889
##  3 Madonna               857
##  4 Kenny Chesney         758
```

```
##  5 Drake                746
##  6 Tim McGraw           731
##  7 Keith Urban          673
##  8 Stevie Wonder        659
##  9 Rod Stewart          657
## 10 Mariah Carey         621
```

If I was to explain the code above in English, I would descibe it as this:

- We start with the hot100 data AND THEN
- we group the data by performer AND THEN
- we summarize it by counting the number of rows in each group, calling the count "appearances" AND THEN
- we arrange the result by appearances in descending order AND THEN
- we kept just the first 10 rows

Since we have our answer here and we're not using the result later, we don't need to create a new tibble or anything.

> AN IMPORTANT NOTE: The list we've created here is based on unique `performer` names, and as such considers collaborations separately. For instance, Drake is near the top of the list but those are only songs he performed alone and not the many, many collaborations he has had with other performers. So, songs by "Drake" are counted separately than "Drake featuring Future" and "Future featuring Drake." You'll need to make this clear when you write your data drop in a later assignment.

So, **Taylor Swift** ... is that who you guessed? A little history here, Swift past Elton John in the summer of 2019. Elton John has been around a long time, but Swift's popularity at a young age, plus changes in how Billboard counts plays in the modern era (like streaming) has rocketed her to the top. (Sorry, Rocket Man).

4.6 Performer/song with most appearances

Our quest here is this: **Which performer/song combination has been on the charts the most number of weeks at any position?**

This is very similar to our quest to find the artist with the most appearances, but we have to consider `performer` and `song` together because different artists can perform songs of the same name. For example, 17 different performers have a song called "Hold On" on the Hot 100 (at least through 2020).

1. Start a new section (headline, text describing goal and a new code chunk.)
2. Add the code below to the chunk and run it and then I'll outline it below.

```
hot100 %>% # start with the data, and then ...
  group_by(performer, song) %>% # group by performer and song, and then ..
  summarize(appearances = n()) %>% # name the column, then fill it with the number of rows ...
  arrange(appearances %>% desc()) # arrange by appearances in descending order
```

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

```
## # A tibble: 29,389 x 3
## # Groups:   performer [10,061]
##    performer                                song                    appearances
##    <chr>                                    <chr>                         <int>
##  1 Imagine Dragons                          Radioactive                      87
##  2 AWOLNATION                               Sail                             79
##  3 Jason Mraz                               I'm Yours                        76
##  4 The Weeknd                               Blinding Lights                  76
##  5 LeAnn Rimes                              How Do I Live                    69
##  6 LMFAO Featuring Lauren Bennett & GoonRock Party Rock Anthem               68
##  7 OneRepublic                              Counting Stars                   68
##  8 Adele                                    Rolling In The Deep              65
##  9 Jewel                                    Foolish Games/You Were~          65
## 10 Carrie Underwood                         Before He Cheats                 64
## # ... with 29,379 more rows
```

The logic is actually straightforward:

- We want to count combinations over two columns: `song, performer`. When you group_by more then one column, it will group rows where the values are the same in all columns. i.e. all rows with both "Rush" as a performer and *Tom Sawyer* as a song. Rows with "Rush" and *Red Barchetta* will be considered in a different group.
- With `summarize()`, we name the new column first (we chose `appearances`), then describe what should fill it. In this case we filled the column using the `n()`, which counts the number of rows in each group.
- Once you have a summary table, we sort it by appearances and set it to **desc**ending order, which puts the highest value on the top.

We will *often* use `group_by()`, `summarize()` and `arrange()` together, which is why I'll refer to this as the GSA trio. They are like three close friends that always want to be together.

4.6.1 Introducing filter()

I showed you `head()` in the previous quest and that is useful to quickly cut off a list, but it does so indiscriminately. In this case, if we use the default `head()` that retains six rows, it would cut right in the middle of a tie at 68 records. (at least with data through 2020). A better strategy is to cut off the list at a logical place using `filter()`. Let's dive into this new function:

Filtering is one of those Basic Data Journalism Functions:

The dplyr function `filter()` reduces the number of rows in our data based on one or more criteria.

The syntax works like this:

```
# this is psuedo code. don't run it
data %>%
  filter(variable comparison value)

# example
hot100 %>%
  filter(performer == "Judas Priest")
```

The `filter()` function typically works in this order:

- What is the variable (or column) you are searching in.
- What is the comparison you want to do. Equal to? Greater than?
- What is the observation (or value in the data) you are looking for?

Note the two equals signs `==` in our Judas Priest example above. It's important to use two of them when you are looking for "equals," as a single `=` will not work, as that means something else in R.

4.6.1.1 Comparisons: Logical tests

There are a number of these logical test for the comparison:

Operator	Definition
x < y	Less than
x > y	Greater than
x == y	Equal to
x <= y	Less than or equal to
x >= y	Greater than or equal to
x **!-** y	Not equal to

Operator	Definition
x **%in%** c(y,z)	In a group
is.na(x)	Is NA
!is.na(x)	Is not NA

Where you apply a filter matters. If we filter before group by/summarize/arrange (GSA) we are focusing the data before we summarize. If we filter after the GSA, we are affecting only the results of the summarize function, which is what we want to do here.

4.6.1.2 Filter to a logical cutoff

In this case, I want you to use filter *after* the GSA actions to include **only results with 65 or more appearances**.

1. Edit your current chunk to add a filter as noted in the example below. I'll explain it afte.

```
hot100 %>%
  group_by(performer, song) %>%
  summarize(appearances = n()) %>%
  arrange(appearances %>% desc()) %>%
  filter(appearances >= 65) # this is the new line
```

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

```
## # A tibble: 9 x 3
## # Groups:   performer [9]
##   performer                                song                     appearances
##   <chr>                                    <chr>                          <int>
## 1 Imagine Dragons                          Radioactive                       87
## 2 AWOLNATION                               Sail                              79
## 3 Jason Mraz                               I'm Yours                         76
## 4 The Weeknd                               Blinding Lights                   76
## 5 LeAnn Rimes                              How Do I Live                     69
## 6 LMFAO Featuring Lauren Bennett & GoonRock Party Rock Anthem                68
## 7 OneRepublic                              Counting Stars                    68
## 8 Adele                                    Rolling In The Deep               65
## 9 Jewel                                    Foolish Games/You Were ~          65
```

Let's break down that last line:

- `filter()` is the function.
- The first argument in the function is the column we are looking in, `appearances` in our case.
- We then provide a comparison operator `>=` to get "greater than or equal to."
- We then give the value to compare, `65` in our case.

4.7 Song/Performer with most weeks at No. 1

We introduced `filter()` in rhe last quest to limit the summary. For this quest you'll need to filter the data *before* the group by/summarize/arrange trio.

Let's review the quest: **Which performer/song combination was No. 1 for the most number of weeks?**

While this quest is very similar to the one above, it *really* helps to think about the logic of what you nneed and then build the query one line at a time to make each line works.

Let's talk through the logic:

- We are starting with our `hot100` data.
- Do we want to consider all the data? In this case, no: We only want songs that have a `week_position` of 1. This means we will **filter** before any summarizing.
- Then we want to count the number of rows with the same **performer** and **song** combinations. This means we need to `group_by` both `performer` and `song`.
- Since we are **counting row**, we need use `n()` as our summarize function, which counts the **number** or rows in each group.

So let's step through this with code:

1. Create a section with a headline, text and code chunk
2. Start with the `hot100` data and then pipe into `filter()`.
3. Within the filter, set the `week_position` to be `==` to `1`.
4. Run the result and check it

```
hot100 %>%
  filter(week_position == 1)
```

```
## # A tibble: 3,279 x 7
##    week_id    week_position song      performer   previous_week_p~ peak_position
##    <date>             <dbl> <chr>     <chr>                  <dbl>         <dbl>
```

```
##  1 1958-08-02             1 Poor Lit~ Ricky Nels~               NA             1
##  2 1958-08-09             1 Poor Lit~ Ricky Nels~                1             1
##  3 1958-08-16             1 Nel Blu ~ Domenico M~                2             1
##  4 1958-08-23             1 Little S~ The Elegan~                2             1
##  5 1958-08-30             1 Nel Blu ~ Domenico M~                2             1
##  6 1958-09-06             1 Nel Blu ~ Domenico M~                1             1
##  7 1958-09-13             1 Nel Blu ~ Domenico M~                1             1
##  8 1958-09-20             1 Nel Blu ~ Domenico M~                1             1
##  9 1958-09-27             1 It's All~ Tommy Edwa~                3             1
## 10 1958-10-04             1 It's All~ Tommy Edwa~                1             1
## # ... with 3,269 more rows, and 1 more variable: weeks_on_chart <dbl>
```

The result should show *only* songs with a `1` for `week_position`.

The rest of our logic is just like our last quest. We need to group by the `song` and `performer` and then `summarize` using `n()` to count the rows.

1. Edit your existing chunk to add the `group_by` and `summarize` functions. Name your new column `appearances` and set it to count the rows with `n()`.

   While I say to write and run your code one line at a time, `group_by()` won't actually show you any different results, so I usually write `group_by()` and `summarize()` together.

Try this on your own before you peek

```
hot100 %>%
  filter(week_position == 1) %>%
  group_by(performer, song) %>%
  summarize(appearances = n())
```

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

```
## # A tibble: 1,124 x 3
## # Groups:   performer [744]
##    performer                          song                              appearances
##    <chr>                              <chr>                                   <int>
##  1 ? (Question Mark) & The Mysterians 96 Tears                                    1
##  2 'N Sync                            It's Gonna Be Me                            2
##  3 24kGoldn Featuring iann dior       Mood                                        8
##  4 2Pac Featuring K-Ci And JoJo       How Do U Want It/California L~              2
##  5 50 Cent                            In Da Club                                  9
##  6 50 Cent Featuring Nate Dogg        21 Questions                                4
```

```
##  7 50 Cent Featuring Olivia          Candy Shop                          9
##  8 6ix9ine & Nicki Minaj             Trollz                              1
##  9 A Taste Of Honey                  Boogie Oogie Oogie                  3
## 10 a-ha                              Take On Me                          1
## # ... with 1,114 more rows
```

Look at your results to make sure you have the three columns you expect: performer, song and appearances.

This doesn't quite get us where we want because it is alphabetically by the perfomer. You need to **arrange** the data to show us the most appearances at the top.

1. Edit your chunk to add the `arrange()` function to sort by `appearances` in `desc()` order. This is just like our last quest.

Maybe check your last chunk on how you did this

```
hot100 %>%
  filter(week_position == 1) %>%
  group_by(performer, song) %>%
  summarize(appearances = n()) %>%
  arrange(appearances %>% desc())
```

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

```
## # A tibble: 1,124 x 3
## # Groups:   performer [744]
##    performer                                          song            appearances
##    <chr>                                              <chr>                 <int>
##  1 Lil Nas X Featuring Billy Ray Cyrus                Old Town Road            19
##  2 Luis Fonsi & Daddy Yankee Featuring Justin Bieber Despacito               16
##  3 Mariah Carey & Boyz II Men                         One Sweet Day            16
##  4 Boyz II Men                                        I'll Make Love~          14
##  5 Elton John                                         Candle In The ~          14
##  6 Los Del Rio                                        Macarena (Bays~          14
##  7 Mariah Carey                                       We Belong Toge~          14
##  8 Mark Ronson Featuring Bruno Mars                   Uptown Funk!             14
##  9 The Black Eyed Peas                                I Gotta Feeling          14
## 10 Whitney Houston                                    I Will Always ~          14
## # ... with 1,114 more rows
```

You have your answer now (you go, Lil Nas) but we are listing more than 1,000 rows. Let's cut this off at a logical place like we did in our last quest.

1. Use `filter()` to cut your summary off at `appearances` of 14 or greater.

You've done this before ... try it on your own!

```
hot100 %>%
  filter(week_position == 1) %>%
  group_by(performer, song) %>%
  summarize(appearances = n()) %>%
  arrange(appearances %>% desc()) %>%
  filter(appearances >= 14)
```

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

```
## # A tibble: 10 x 3
## # Groups:   performer [10]
##    performer                                          song            appearances
##    <chr>                                              <chr>                 <int>
##  1 Lil Nas X Featuring Billy Ray Cyrus                Old Town Road            19
##  2 Luis Fonsi & Daddy Yankee Featuring Justin Bieber Despacito                16
##  3 Mariah Carey & Boyz II Men                         One Sweet Day           16
##  4 Boyz II Men                                        I'll Make Love~         14
##  5 Elton John                                         Candle In The ~         14
##  6 Los Del Rio                                        Macarena (Bays~         14
##  7 Mariah Carey                                       We Belong Toge~         14
##  8 Mark Ronson Featuring Bruno Mars                   Uptown Funk!            14
##  9 The Black Eyed Peas                                I Gotta Feeling         14
## 10 Whitney Houston                                    I Will Always ~         14
```

Now you have the answers to the performer/song with the most weeks at No. 1 with a logical cutoff. If you add to the data, that logic will still hold and not cut off arbitrarily at a certain number of records.

4.8 Performer with most songs to reach No. 1

Our new quest is this: **Which performer had the most songs reach No. 1?** The answer might be easier to guess if you know music history, but perhaps not.

This sounds similar to our last quest, but there is a **distinct** difference. (That's a bad joke that will reveal itself here in a bit.)

Again, let's think through the logic of what we have to do to get our answer:

- We need to consider only No. 1 songs. (filter!)

- Because a song could be No. 1 for more than one week, we need to consider the same song/performer combination only once. (We'll introduce a new function for this.)
- Once we have all the unique No. 1 songs in a list, then we can group by **performer** and count how many times many times they are on the list.

Let's start by getting the No. 1 songs. You've did this in the last quest.

1. Create a new section with a headline, text and code chunk.
2. Start with the `hot100` data and filter it so you only have `week_position` of 1.

Try on your own. You got this!</summary.

```
hot100 %>%
  filter(week_position == 1)
```

```
## # A tibble: 3,279 x 7
##    week_id    week_position song      performer   previous_week_p~ peak_position
##    <date>             <dbl> <chr>     <chr>                  <dbl>         <dbl>
##  1 1958-08-02             1 Poor Lit~ Ricky Nels~               NA             1
##  2 1958-08-09             1 Poor Lit~ Ricky Nels~                1             1
##  3 1958-08-16             1 Nel Blu ~ Domenico M~                2             1
##  4 1958-08-23             1 Little S~ The Elegan~                2             1
##  5 1958-08-30             1 Nel Blu ~ Domenico M~                2             1
##  6 1958-09-06             1 Nel Blu ~ Domenico M~                1             1
##  7 1958-09-13             1 Nel Blu ~ Domenico M~                1             1
##  8 1958-09-20             1 Nel Blu ~ Domenico M~                1             1
##  9 1958-09-27             1 It's All~ Tommy Edwa~                3             1
## 10 1958-10-04             1 It's All~ Tommy Edwa~                1             1
## # ... with 3,269 more rows, and 1 more variable: weeks_on_chart <dbl>
```

Now look at the result. Note how "Poor Little Fool" shows up more than once? Other songs to as well. If we counted rows by `performer` now, we could count that song more than once. That's not what we want.

4.8.1 Using distinct()

Our next challenge in our logic is to show only unique performer/song combinations. We do this with `distinct()`.

We feed the `distinct()` function with the variables we want to consider together, in our case the `perfomer` and `song`. All other columns are dropped since including them would mess up their distinctness.

1. Add the distinct() function to your code chunk.

```
hot100 %>%
  filter(week_position == 1) %>%
  distinct(song, performer)
```

```
## # A tibble: 1,124 x 2
##    song                              performer
##    <chr>                             <chr>
##  1 Poor Little Fool                  Ricky Nelson
##  2 Nel Blu Dipinto Di Blu (Volaré) Domenico Modugno
##  3 Little Star                       The Elegants
##  4 It's All In The Game              Tommy Edwards
##  5 It's Only Make Believe            Conway Twitty
##  6 Tom Dooley                        The Kingston Trio
##  7 To Know Him, Is To Love Him       The Teddy Bears
##  8 The Chipmunk Song                 The Chipmunks With David Seville
##  9 Smoke Gets In Your Eyes           The Platters
## 10 Stagger Lee                       Lloyd Price
## # ... with 1,114 more rows
```

Now we have a list of just No. 1 songs!

4.8.2 Summarize the performers

Now that we have our list of No. 1 songs, we can "count" the number of times a performer is in the list to know how many No. 1 songs they have.

We'll again use the group_by/summarize combination for this, but we are only grouping by `performer` since that is what we are counting.

1. Edit your chunk to add a group_by on `performer` and then a `summarize()` to count the rows. Name the new column `no_hits`. Run it.
2. After you are sure the group_by/summarize runs, add an `arrange()` to show the `no1_hits` in descending order.

You've done this. Give it ago!

```
hot100 %>%
  filter(week_position == 1) %>%
  distinct(song, performer) %>%
  group_by(performer) %>%
  summarize(no1_hits = n()) %>%
  arrange(no1_hits %>% desc())
```

```
## # A tibble: 744 x 2
##    performer          no1_hits
##    <chr>                 <int>
##  1 The Beatles              19
##  2 Mariah Carey             16
##  3 Madonna                  12
##  4 Michael Jackson          11
##  5 Whitney Houston          11
##  6 The Supremes             10
##  7 Bee Gees                  9
##  8 The Rolling Stones        8
##  9 Janet Jackson             7
## 10 Stevie Wonder             7
## # ... with 734 more rows
```

4.8.3 Filter for a good cutoff

Like we did earlier, use a `filter()` after your arrange to cut the list off at a logical place.

1. Edit your chunk to filter the summary to show performer with `8` or more No. 1 hits.

You can do this. Really

```
hot100 %>%
  filter(week_position == 1) %>%
  distinct(song, performer) %>%
  group_by(performer) %>%
  summarize(no1_hits = n()) %>%
  arrange(no1_hits %>% desc()) %>%
  filter(no1_hits >= 8)
```

```
## # A tibble: 8 x 2
##   performer          no1_hits
##   <chr>                 <int>
## 1 The Beatles              19
## 2 Mariah Carey             16
## 3 Madonna                  12
## 4 Michael Jackson          11
## 5 Whitney Houston          11
## 6 The Supremes             10
## 7 Bee Gees                  9
## 8 The Rolling Stones        8
```

4.9 No. 1 hits in last five years

Which performer had the most songs reach No. 1 in the most recent five years?

Let's talk through the logic. This is very similar to the No. 1 hits above but with two differences:

- In addition to filtering for No. 1 songs, we also want to filter for songs in 2016-2020.
- We might need to adjust our last filter for a better "break point."

We haven't talked about filtering dates, so let me tell you this: You can use filter operations on dates just like you do any other text. This will give you rows *after* 2015.

```
filter(week_id > "2015-12-31")
```

But since we need this filter before our group, we can do this within the same filter function where we get the number one songs.

1. Create a new section (headline, text, chunk).
2. Build (from scratch) the same filter, group_by, summarize and arrange as above, but leave out the cut-off filter at the end. (We'll need to adjust that based on the results). Make sure it runs.
3. EDIT your filter to put a comma after `week_position == 1` and then add this filter: `week_id > "2015-12-31"`. Run the code.
4. Build a new cut-off filter at the end keep only rows with more than 1 `top_hits`.

No, really. Try it on your own first.

```
hot100 %>%
  filter(
    week_position == 1,
    week_id > "2015-12-31"
  ) %>%
  distinct(song, performer) %>%
  group_by(performer) %>%
  summarize(top_hits = n()) %>%
  arrange(top_hits %>% desc()) %>%
  filter(top_hits > 1)
```

```
## # A tibble: 10 x 2
##    performer      top_hits
```

```
##    <chr>             <int>
##  1 Drake                 5
##  2 Ariana Grande         3
##  3 Taylor Swift          3
##  4 BTS                   2
##  5 Cardi B               2
##  6 Ed Sheeran            2
##  7 Justin Bieber         2
##  8 Olivia Rodrigo        2
##  9 The Weeknd            2
## 10 Travis Scott          2
```

4.10 Top 10 hits overall

Which performer had the most Top 10 hits overall?

This one I want you to do on your own.

The logic is very similar to the "Most No. 1 hits" quest you did before, but you need to adjust your filter to find songs within position 1 through 10. Don't overthink it, but do recognize that the "top" of the charts are smaller numbers, not larger ones.

1. Make a new section
2. Describe what you are doing
3. Do it using the group_by/summarize method
4. Filter to cut off at a logical number or rows. (i.e., don't stop at a tie)

4.10.1 A shortcut: count()

You are going to think I'm a horrible person, but there is an easier way to do this . . .

We count stuff in data science (and journalism) all the time. Because of this tidyverse has a shortcut to group and count rows of data. I needed to show you the long way because a) we will use `group_by()` and `summarize()` with other math that isn't just counting rows, and b) you need to understand what is happening when you use `count()`, which is really just using group_by/summarize underneath.

The `count()` function takes the columns you want to group and then does the summarize on `n()` for you:

```
hot100 %>%
  count(performer)
```

```
## # A tibble: 10,061 x 2
##    performer                                n
##    <chr>                                <int>
##  1 "? (Question Mark) & The Mysterians"    33
##  2 "'N Sync"                              172
##  3 "'N Sync & Gloria Estefan"              20
##  4 "'N Sync Featuring Nelly"               20
##  5 "'Til Tuesday"                          53
##  6 "\"Groove\" Holmes"                     14
##  7 "\"Little\" Jimmy Dickens"              10
##  8 "\"Pookie\" Hudson"                      1
##  9 "\"Weird Al\" Yankovic"                 91
## 10 "(+44)"                                  1
## # ... with 10,051 more rows
```

To get the same pretty table you still have to rename the new column and reverse the sort, you just do it differently as arguments within the `count()` function. You can view the `count()` options here.

- Add this chunk to your notebook (with a note you are trying `count()`) so you have it to refer to.

```
hot100 %>%
  count(performer, name = "appearances", sort = TRUE) %>%
  filter(appearances > 600)
```

```
## # A tibble: 13 x 2
##    performer       appearances
##    <chr>                 <int>
##  1 Taylor Swift           1022
##  2 Elton John              889
##  3 Madonna                 857
##  4 Kenny Chesney           758
##  5 Drake                   746
##  6 Tim McGraw              731
##  7 Keith Urban             673
##  8 Stevie Wonder           659
##  9 Rod Stewart             657
## 10 Mariah Carey            621
## 11 Michael Jackson         611
## 12 Chicago                 607
## 13 Rascal Flatts           604
```

So you have to do the same things here as in our first quest, but when you just need a quick count to get an answer, then `count()` is brilliant.

> IMPORTANT: We concentrate on using group_by/summarize/arrange because it can do so much more than `count()`. Count can ONLY count rows. It can't do any other kind of math in summarize.

4.10.2 Complex filters

Don't do these, but you'll need them for reference later:

If you want filter data for multiple criteria, you can write two equations and combine with `&`. Only rows with both sides being true are returned.

```
# gives you only Poor Little Fool rows where song is No. 1, but not any other position
filter(song == "Poor Little Fool" & week_position == 1)
```

If you want an "or" filter, then you write two equations with a `|` between them.

> `|` is the *Shift* of the `\` key above Return on your keyboard. That `|` character is also sometimes called a "pipe," which gets confusing in R with `%>%`.)

```
# gives you Taylor or Drake songs
filter(performer == "Taylor Swift" | performer == "Drake")
```

If you have multiple criteria, you separate them with a comma `,`. Note I've also added returns to make it more readable.

```
# gives us rows with either Taylor Swift or Drake, but only those at No. 1
filter(
  performer == "Taylor Swift" | performer == "Drake",
  week_position == 1
)
```

4.11 Review of what we've learned

We introduced a number of new functions in this lesson, most of them from the dplyr package. Mostly we filtered and summarized our data. Here are the functions we introduced in this chapter, many with links to documentation:

- `filter()` returns only rows that meet logical criteria you specify.
- `summarize()` builds a summary table *about* your data. You can count rows `n()` or do math on numerical values, like `mean()`.

- `group_by()` is often used with `summarize()` to put data into groups before building a summary table based on the groups.
- `distinct()` returns rows based on unique values in columns you specify. i.e., it deduplicates data.
- `count()` is a shorthand for the group_by/summarize operation to count rows based on groups. You can name your summary columns and sort the data within the same function.

4.12 Turn in your project

1. Make sure everything runs properly (Restart R and Run All Chunks) and then Knit to HTML.
2. Zip the folder.
3. Upload to the Canvas assignment.

4.13 Soundtrack for this assignment

This lesson was constructed with the vibes of The Bright Light Social Hour. They've never had a song on the Hot 100 (at least not through 2020).

Chapter 5

Summarize with math - import

With our Billboard assignment, we went through some common data wrangling processes — importing data, cleaning it and querying it for answers. All of our answers involved counting numbers of rows and we did so with two methods: The summary trio: `group_by`, `summmarize` and `arrange` (which I dub GSA), and then the shortcut `count()` that allows us to do all of that in one line.

For this data story we need to leave `count` behind and stick with the summary trio GSA because now we must do different kinds of math within our summarize functions, mainly `sum()`.

5.1 About the story: Military surplus transfers

In June 2020, Buzzfeed published the story *Police Departments Have Received Hundreds Of Millions Of Dollars In Military Equipment Since Ferguson* about the amount of military equipment transferred to local law enforcement agencies since Michael Brown was killed in Ferguson, Missouri. After Brown's death there was a public outcry after "what appeared to be a massively disproportionate show of force during protests brought scrutiny to a federal program that transfers unused military equipment to local law enforcement." Reporter John Templon used data from the Law Enforcement Support Office for the update on the program and published his data analysis, which he did in Python.

You will analyze the same dataset focusing on some local police agencies and write a short data drop about transfers to those agencies.

5.1.1 The LESO program

The Defense Logistics Agency transfers surplus military equipment to local law enforcement through its Law Enforcement Support Office. You can find more information about the program here.

The agency updates the data quarterly and the data I've collected contains transfers through **June 30, 2021**. The original file is linked from the headline "ALASKA - WYOMING AND US TERRITORIES."

The data there comes in an Excel spreadsheet that has a new sheet for each state. I used R to pull the data from each sheet and combine it into a single data set and I'll cover the process I used in class, but you won't have to do that part.

I will supply a link to the combined data below.

5.1.2 About the data

There is no data dictionary or record layout included with the data but I have corresponded with the Defense Logistics Agency to get a decent understanding of what is included. Columns in bold are those we care about the most.

- sheet: Which sheet the data came from. This is an artifact from the data merging script.
- **state**: A two-letter designation for the state of the agency.
- **agency_name**: This is the agency that got the equipment.
- nsn: A special number that identifies the item. It is not germane to this specific assignment.
- **item_name**: The item transferred. Googling the names can sometimes yield more info on specific items.
- **quantity**: The number of the "units" the agency received.
- ui: Unit of measurement (item, kit, etc.)
- **acquisition_value**: a cost *per unit* for the item.
- demil_code: Another special code not germane to this assignment.
- demil_ic: Another special code not germane to this assignment.
- **ship_date**: The date the item(s) were sent to the agency.
- station_type: What kind of law enforcement agency made the request.

Here is a glimse of our main columns of interest, except for the date:

```
## Rows: 10
## Columns: 5
## $ state             <chr> "KY", "SC", "CA", "TX", "OH", "NC", "CA", "MI", "AZ"~
## $ agency_name       <chr> "MEADE COUNTY SHERIFF DEPT", "PROSPERITY POLICE DEPT~
## $ item_name         <chr> "GENERATOR SET,DIESEL ENGINE", "RIFLE,7.62 MILLIMETE~
```

```
## $ quantity          <dbl> 5, 1, 32, 1, 1, 1, 1, 1, 1, 1
## $ acquisition_value <dbl> 4623.09, 138.00, 16.91, 749.00, 749.00, 138.00, 499.~
```

Each row of data is a transfer of a particular type of item from the U.S. Department of Defense to a local law enforcement agency. The row includes the name of the item, the quantity, and the value ($) of a single unit.

What the data doesn't have is the **total value** of the items in the shipment. If there are 5 generators as noted in the first row above and the cost of each one is $4623.09, we have to multiply the `quantity` times the `acquisition_value` to get the total value of that equipment. We will do that as part of the assignment.

The local agencies really only pay the shipping costs for the item, *so you can't say they paid for the items*, so the **total value** you calculate is the "value" of the items, not their cost to the local agency.

5.2 The questions we will answer

All answers will be based on data from **Jan. 1, 2010** to present. In addition, we'll only consider **Texas** agencies as you answer the following.

- For each agency in Texas, find the summed **quantity** and summed **total value** of the equipment they received. (When I say "summed" that means we'll add together all the values in the column.)
  - Once you have the list, we'll think about what stands out and why?
- We'll take the list above, but filter that summary to show only the following local agencies:
  - AUSTIN POLICE DEPT
  - SAN MARCOS POLICE DEPT
  - TRAVIS COUNTY SHERIFFS OFFICE
  - UNIV OF TEXAS SYSTEM POLICE HI_ED
  - WILLIAMSON COUNTY SHERIFF'S OFFICE
- For each of the agencies above we'll use summarize to get the *summed* **quantity** and *summed* **total_value** of each **item** shipped to the agency. We'll create a summarized list for each agency so we can write about each one.
- You'll research some of the more interesting items the agencies received (i.e. Google the names) so you can include them in your data drop.

5.3 Create your project

We will build the same project structure that we did with the Billboard project. In fact, all our class projects will have this structure. Since we've done this before, some of the directions are less detailed.

1. With RStudio open, make sure you don't have a project open. Go to File > Close project.
2. Use the create project button (or File > New project) to create a new project in a "New Directory." Name the directory "yourname-military-surplus."
3. Create two folders: `data-raw` and `data-processed`.

5.4 Import/cleaning notebook

Again, like Billboard, we'll create a notebook specifically for downloading, cleaning and prepping our data.

1. Create your RNotebook.
2. Rename the title "Military Surplus import/clean."
3. Remove the rest of the boilerplate template.
4. Save the file and name it `01-import.Rmd`.

5.4.1 Add the goals of the notebook

1. In Markdown, add a headline noting these are notebook goals.
2. Add the goals below:

```
- Download the data
- Import the data
- Clean datatypes
- Remove unnecessary columns
- Create a total_value column
- Filter to Texas agencies
- Filter the date range (since Jan. 1 2010)
- Export the cleaned data
```

> NOTE: Most of these are pretty standard in a import/cleaning notebook. Filtering to Texas agencies is specific to this data set, but we would do all these other things in all projects.

5.4.2 Add a setup section

This is the section where we add our libraries and such. Again, every notebook has this section, though the packages may vary on need.

1. Add a headline and text about what we are doing: Our project setup.
2. Add a code chunk to load the libraries. You should only need `tidyverse` for this notebook because the data already has clean names (no need for janitor) and the dates will import correctly (no need for lubridate).

```
library(tidyverse)
```

5.4.3 Download the data

1. A new section means a new headline and description. Add it. It is good practice to describe and link to the data you will be using. You can use this:

```
The Defense Logistics Agency transfers surplus military equipment to local law enforcement throug
```

1. Use the `download.file()` function to download the date into your `data-raw` folder. Remember you need two arguments:

```
download.file("url_to_data", "path_to_folder/filename.csv")
```

- The data can be found at this url: `https://github.com/utdata/rwd-r-leso/blob/main/data-processed/le`
- It should be saved into your `data-raw` folder with a name for the file.

Once you've built your code chunk and run it, you should make sure the file downloaded into the correct place: in your `data-raw` folder.

You should be able to do this on your own. Really.

```
# You can comment the line below once you have the data
download.file("https://github.com/utdata/rwd-r-leso/blob/main/data-processed/leso.csv?raw=true",
              "data-raw/leso.csv")
```

5.4.4 Import the data

We are again working with a CSV, or comma-separated-values text file.

1. Add a new section: Headline, text if needed, code chunk.

I suggest you build the code chunk a bit at a time in this order:

1. Use `read_csv()` to read the file from our `data-raw` folder.
2. Edit that line to put the result into a tibble object using `<-`. Name your new tibble `leso`.
3. Print the tibble as a table to the screen again by putting the tibble object on a new line and running it. This allows you to see it in columnar form.

Try real hard first before clicking here for the answer. Note the book will also show the response.

```
# assigning the tibble
leso <- read_csv("data-raw/leso.csv")
```

```
## Rows: 129348 Columns: 12
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (7): state, agency_name, nsn, item_name, ui, demil_code, station_type
## dbl  (4): sheet, quantity, acquisition_value, demil_ic
## dttm (1): ship_date
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# printing the tibble
leso
```

```
## # A tibble: 129,348 x 12
##    sheet state agency_name   nsn    item_name    quantity ui    acquisition_val~
##    <dbl> <chr> <chr>         <chr>  <chr>           <dbl> <chr>            <dbl>
##  1     1 AL    ABBEVILLE PO~ 1005-~ MOUNT,RIFLE        10 Each              1626
##  2     1 AL    ABBEVILLE PO~ 1240-~ SIGHT,REFLEX        9 Each               333
##  3     1 AL    ABBEVILLE PO~ 1240-~ OPTICAL SIG~        1 Each               246.
##  4     1 AL    ABBEVILLE PO~ 1385-~ UNMANNED VE~        1 Each             10000
##  5     1 AL    ABBEVILLE PO~ 2320-~ TRUCK,UTILI~        1 Each             62627
##  6     1 AL    ABBEVILLE PO~ 2320-~ TRUCK,UTILI~        1 Each             62627
##  7     1 AL    ABBEVILLE PO~ 2355-~ MINE RESIST~        1 Each            658000
##  8     1 AL    ABBEVILLE PO~ 2540-~ BALLISTIC B~       10 Kit             15872.
##  9     1 AL    ABBEVILLE PO~ 5855-~ ILLUMINATOR~       10 Each               926
## 10     1 AL    ABBEVILLE PO~ 6760-~ CAMERA ROBOT        1 Each              1500
## # ... with 129,338 more rows, and 4 more variables: demil_code <chr>,
## #   demil_ic <dbl>, ship_date <dttm>, station_type <chr>
```

5.4.5 Glimpse the data

1. In a new block, print the tibble but pipe it into `glimpse()` so you can see all the column names.

```
leso %>% glimpse()
```

```
## Rows: 129,348
## Columns: 12
## $ sheet             <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ state             <chr> "AL", "AL", "AL", "AL", "AL", "AL", "AL", "AL", "AL"~
## $ agency_name       <chr> "ABBEVILLE POLICE DEPT", "ABBEVILLE POLICE DEPT", "A~
## $ nsn               <chr> "1005-01-587-7175", "1240-01-411-1265", "1240-DS-OPT~
## $ item_name         <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "OPTICAL SIGHTING AND~
## $ quantity          <dbl> 10, 9, 1, 1, 1, 1, 1, 10, 10, 1, 5, 10, 11, 10, 1, 3~
## $ ui                <chr> "Each", "Each", "Each", "Each", "Each", "Each", "Eac~
## $ acquisition_value <dbl> 1626.00, 333.00, 245.88, 10000.00, 62627.00, 62627.0~
## $ demil_code        <chr> "D", "D", "D", "Q", "C", "C", "C", "D", "D", "D", "D~
## $ demil_ic          <dbl> 1, 1, NA, 3, 1, 1, 1, 1, 1, 7, 1, 1, 1, 1, 1, NA, 1,~
## $ ship_date         <dttm> 2016-09-19, 2016-09-14, 2016-06-02, 2017-03-28, 201~
## $ station_type      <chr> "State", "State", "State", "State", "State", "State"~
```

5.4.5.1 Checking datatypes

Take a look at your glimpse returns. These are the things to watch for:

- Are your variable names (column names) clean? All lowercase with `_` separating words?
- Are dates saved in a date format? `ship_date` looks good at `<dttm>`, which means "datetime."
- Are your numbers really numbers? `acquisition_value` is the column we are most concerned about here, and it looks good.

This data set looks good (because I pre-prepared it fo you), but you always want to check and make corrections, like we did to fix the date in the Billboard assignment.

5.4.6 Remove unnecessary columns

Sometimes at this point in a project, you might not know what columns you need to keep and which you could do without. The nice thing about doing this with code in a notebook is we can always go back, make corrections and run our notebook again. In this case, I'm going to tell you which columns you can

remove so we have a tighter data set to work with. We'll also learn a cool trick with `select()`.

1. Start a new section with a headline, text to explain you are removing unneeded columns.
2. Add a code chunk and the following code. I'll explain it below.

```
leso_tight <- leso %>%
  select(
    -sheet,
    -nsn,
    -starts_with("demil")
  )

leso_tight %>% glimpse()
```

```
## Rows: 129,348
## Columns: 8
## $ state             <chr> "AL", "AL", "AL", "AL", "AL", "AL", "AL", "AL", "AL"~
## $ agency_name       <chr> "ABBEVILLE POLICE DEPT", "ABBEVILLE POLICE DEPT", "A~
## $ item_name         <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "OPTICAL SIGHTING AND~
## $ quantity          <dbl> 10, 9, 1, 1, 1, 1, 1, 10, 10, 1, 5, 10, 11, 10, 1, 3~
## $ ui                <chr> "Each", "Each", "Each", "Each", "Each", "Each", "Eac~
## $ acquisition_value <dbl> 1626.00, 333.00, 245.88, 10000.00, 62627.00, 62627.0~
## $ ship_date         <dttm> 2016-09-19, 2016-09-14, 2016-06-02, 2017-03-28, 201~
## $ station_type      <chr> "State", "State", "State", "State", "State", "State"~
```

We did a select like this with billboard, but note the third item within the `select()`:

`-starts_with("demil")`.

This removes both the `demil_code` and `demil_ic` columns in one move by finding all the columns that "start with 'demil'." The `-` before it negates (or removes) the columns.

There are other special operators you can use with select(), like: `ends_with()`, `contains()` and many more. Check out the docs on the select function.

So now we have a tibble called `leso_tight` that we will work with in the next section.

5.4.7 Create a total_value column

When we used `mutate()` to convert the date in the Billboard assignment, we were reassigning values in each row of a column back into the same column.

In this assignment, we will use `mutate()` to create a **new** column with new values based on a calculation (`quantity` multiplied by the `acquisition_value`) for each row. Let's review the concept first.

If you started with data like this:

item	item_count	item_value
Bread	2	1.5
Milk	1	2.75
Beer	3	9

And wanted to create a total value of each item in the table, you would use `mutate()`:

```
data %>%
  mutate(total_value = item_count * item_value)
```

And you would get a return like this, with your new `total_value` column added at the end:

item	item_count	item_value	total_value
Bread	2	1.5	3
Milk	1	2.75	2.75
Beer	3	9	27

Other math operators work as well: `+`, `-`, `*` and `/`.

So, now that we've talked about how it is done, I want you to:

1. Create a new section with headline, text and code chunk.
2. Use `mutate()` to create a new `total_value` column that multiplies `quantity` times `acquisition_value`.
3. Assign those results into a new tibble called `leso_total` so we can all be on the same page.
4. Glimpse the new tibble so you can check the results.

Try it on your own. You can figure it out!

```
leso_total <- leso_tight %>%
  mutate(
    total_value = quantity * acquisition_value
  )

leso_total %>% glimpse()
```

```
## Rows: 129,348
## Columns: 9
## $ state             <chr> "AL", "AL", "AL", "AL", "AL", "AL", "AL", "AL", "AL"~
## $ agency_name       <chr> "ABBEVILLE POLICE DEPT", "ABBEVILLE POLICE DEPT", "A~
## $ item_name         <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "OPTICAL SIGHTING AND~
## $ quantity          <dbl> 10, 9, 1, 1, 1, 1, 1, 10, 10, 1, 5, 10, 11, 10, 1, 3~
## $ ui                <chr> "Each", "Each", "Each", "Each", "Each", "Each", "Eac~
## $ acquisition_value <dbl> 1626.00, 333.00, 245.88, 10000.00, 62627.00, 62627.0~
## $ ship_date         <dttm> 2016-09-19, 2016-09-14, 2016-06-02, 2017-03-28, 201~
## $ station_type      <chr> "State", "State", "State", "State", "State", "State"~
## $ total_value       <dbl> 16260.00, 2997.00, 245.88, 10000.00, 62627.00, 62627~
```

Check that it worked!!. Use the glimpsed data to check the first item: For me, 10 * 1626.00 = 16260.00, which is correct!

Note that new columns are added at the end of the tibble. That is why I suggested you glimpse the data instead of printing the tibble so you can easily see results on one screen.

5.4.8 Filtering our data

You used `filter()` in the Billboard lesson to get No. 1 songs and to get a date range of data. We need to do something similar here to get only Texas data of a certain date range, but we'll build the filters one at a time so we can check the results.

5.4.8.1 Apply the TX filter

1. Create a new section with headlines and text that denote you are filtering the data to Texas and since Jan. 1, 2010
2. Create the code chunk and start your filter process using the `leso_total` tibble.
3. Use `filter()` on the `state` column to keep all rows with "TX."

Really, you got this.

```
leso_total %>%
  filter(
    state == "TX"
  )
```

```
## # A tibble: 8,684 x 9
##    state agency_name           item_name         quantity ui    acquisition_val~
##    <chr> <chr>                 <chr>                <dbl> <chr>            <dbl>
```

```
##  1 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  2 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  3 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  4 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  5 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  6 TX    ABERNATHY POLICE DEPT RIFLE,5.56 MILLI~        1 Each             749
##  7 TX    ABERNATHY POLICE DEPT RIFLE,5.56 MILLI~        1 Each             749
##  8 TX    ABERNATHY POLICE DEPT SIGHT,REFLEX             5 Each             333
##  9 TX    ABERNATHY POLICE DEPT TRUCK,UTILITY            1 Each           62627
## 10 TX    ABILENE POLICE DEPT   RIFLE,5.56 MILLI~        1 Each             499
## # ... with 8,674 more rows, and 3 more variables: ship_date <dttm>,
## #   station_type <chr>, total_value <dbl>
```

How do you know if it worked? Well the first column in the data is the `state` column, so they should all start with "TX." Also note you started with nearly 130k observations (rows), and there are only 8,600+ in Texas.

5.4.8.2 Add the date filter

1. Now, **EDIT THAT SAME CHUNK** to add a new part to your filter to also get rows with a `ship_date` of 2010-01-01 or later.

If you do this on your own, treat yourself to a cookie

```
leso_total %>%
  filter(
    state == "TX",
    ship_date >= "2010-01-01"
  )
```

```
## # A tibble: 7,407 x 9
##    state agency_name           item_name         quantity ui    acquisition_val~
##    <chr> <chr>                 <chr>                <dbl> <chr>            <dbl>
##  1 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  2 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  3 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  4 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  5 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .~        1 Each              58.7
##  6 TX    ABERNATHY POLICE DEPT RIFLE,5.56 MILLI~        1 Each             749
##  7 TX    ABERNATHY POLICE DEPT RIFLE,5.56 MILLI~        1 Each             749
##  8 TX    ABERNATHY POLICE DEPT SIGHT,REFLEX             5 Each             333
##  9 TX    ABERNATHY POLICE DEPT TRUCK,UTILITY            1 Each           62627
## 10 TX    ALLEN POLICE DEPT     SIGHT,REFLEX             1 Each             333
## # ... with 7,397 more rows, and 3 more variables: ship_date <dttm>,
## #   station_type <chr>, total_value <dbl>
```

5.4.8.3 Checking the results with summary()

How do you know this date filter worked? Well, we went from 8600+ rows to 7400+ rows, so we did something. You might look at the results table and click over to the `ship_date` columns so you can see some of the results, but you can't be sure the top row is the oldest. We could use an `arrange()` to test that, but I have another suggestion: `summary()`.

Now, `summary()` is different than `summarize()`, which we'll do plenty of in a mintue. The `summary()` function will show you some results about each column in your data, and when it is a number or date, it will give you some basic stats like min, max and median values.

1. Use the image below to add a `summary()` function to your filtering data chunk.
2. Once you've confirmed that the "Min." of `ship_date` is not older than 2010, then **REMOVE THE SUMMARY STATEMENT**.

If you leave the summary statement there when we create our updated tibble, then you'll "save" the summary and not the data.

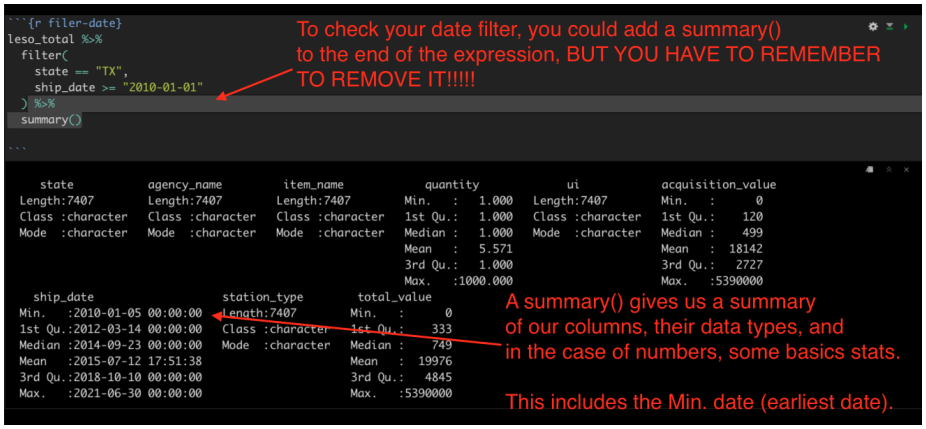


Figure 5.1: Summary function

5.4.8.4 Add filtered data to new tibble

Once you've checked and removed the summary, you can save your filtered data into a new tibble.

1. Edit the filtering chunk to put the results into a new tibble called `leso_filtered`.

Seriously? You were going to look?

```
leso_filtered <- leso_total %>%
  filter(
    state == "TX",
    ship_date >= "2010-01-01"
  )

leso_filtered %>% glimpse()
```

```
## Rows: 7,407
## Columns: 9
## $ state             <chr> "TX", "TX", "TX", "TX", "TX", "TX", "TX", "TX", "TX"~
## $ agency_name       <chr> "ABERNATHY POLICE DEPT", "ABERNATHY POLICE DEPT", "A~
## $ item_name         <chr> "PISTOL,CALIBER .45,AUTOMATIC", "PISTOL,CALIBER .45,~
## $ quantity          <dbl> 1, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ ui                <chr> "Each", "Each", "Each", "Each", "Each", "Each", "Eac~
## $ acquisition_value <dbl> 58.71, 58.71, 58.71, 58.71, 58.71, 749.00, 749.00, 3~
## $ ship_date         <dttm> 2011-11-03, 2011-11-03, 2011-11-03, 2011-11-03, 201~
## $ station_type      <chr> "State", "State", "State", "State", "State", "State"~
## $ total_value       <dbl> 58.71, 58.71, 58.71, 58.71, 58.71, 749.00, 749.00, 1~
```

5.4.9 Export cleaned data

Now that we have our data selected, mutated and filtered how we want it, we can export your `leso_filtered` tibble into an `.rds` file to use in our analysis notebook. If you recall, we use the `.rds` format because it will remember data types and such.

1. Create a new section with headline and text explaining that you are exporting the data.
2. Do it. The function you need is called `write_rds` and you need to give it a path/name that saves the file in the `data-processed` folder. Name it `01-leso-tx.rds` so you know it a) came from the first notebook b) is the Texas only data. **Well-formatted, descriptive file names are important to your future self and other colleagues**.

Try it

```
leso_filtered %>% write_rds("data-processed/01-leso-tx.rds")
```

5.5 Things we learned in this lesson

This chapter was similar to when we imported data for Billboard, but we did introduce a couple of new concepts:

- `starts_with()` can be used within a `select()` function to select columns with similar names. There are also `ends_with()` and `contains()` and others. See the documentation on Select.
- `summary()` gives you descriptive statistics about your tibble. We used it to check the "min" date, but you can also see averages (mean), max and medians.

Chapter 6

Summarize with math - analysis

In the last chapter, we covered the overall story about the LESO data ... that local law enforcement agencies can get surplus military equipment from the U.S. Department of Defense. We downloaded a pre-processed version of the data and filtered it to just Texas records over a specific time period (2010 to present), and used `mutate()` to create a new column calculated fron other variables in the data.

6.1 Learning goals of this lesson

In this chapter we will start querying the data using **summarize with math**, basically using summarize to add values in a column instead of counting rows, which we did with the Billboard assignment.

Our learning goals are:

- To use the combination of `group_by()`, `summarize()` and `arrange()` to add columns of data using `sum()`.
- To use different `group_by()` groupings in specific ways to get desired results.
- To practice using `filter()` on those summaries to better see certain results, including filtering with*in* a vector (or list of strings).
- We'll research and write about some of the findings, practicing data-centric ledes and sentences describing data.

6.2 Questions to answer

A reminder of what we are looking for: All answers are be based on data from **Jan. 1, 2010** to present for only consider **Texas** agencies. We did this filtering already.

- For each agency in Texas, find the summed **quantity** and summed **total value** of the equipment they received. (When I say "summed" that means we'll add together all the values in the column.)
  - Once you have the list, we'll think about what stands out and why?
- We'll take the list above, but filter that summary to show only the following local agencies:
  - AUSTIN POLICE DEPT
  - SAN MARCOS POLICE DEPT
  - TRAVIS COUNTY SHERIFFS OFFICE
  - UNIV OF TEXAS SYSTEM POLICE HI_ED
  - WILLIAMSON COUNTY SHERIFF'S OFFICE
- For each of the agencies above we'll use summarize to get the *summed* **quantity** and *summed* **total_value** of each **item** shipped to the agency. We'll create a summarized list for each agency so we can write about each one.
- You'll research some of the more interesting items the agencies received (i.e. Google the names) so you can include them in your data drop.

6.3 Set up the analysis notebook

Before we get into how to do this, let's set up our analysis notebook.

1. Make sure you have your military surplus project open in RStudio. If you have your import notebook open, close it and use Run > Restart R and Clear Output.
2. Create a new RNotebook and edit the title as "Military surplus analysis."
3. Remove the boilerplate text.
4. Create a setup section (headline, text and code chunk) that loads the tidyverse library.
5. Save the notebook at `02-analysis.Rmd`.

We've started each notebook like this, so you should be able to do this on your own now.

6.3.1 Load the data into a tibble

1. Next create an import section (headline, text and chunk) that loads the data from the previous notebook and save it into a tibble called `tx`.
2. Add a `glimpse()` of the data for your reference.

We did this in Billboard and you should be able to do it. You'll use `read_rds()` and find your data in your data-processed folder.

Remember your data is in data-processed

```
tx <- read_rds("data-processed/01-leso-tx.rds")

tx %>% glimpse()
```

```
## Rows: 7,407
## Columns: 9
## $ state             <chr> "TX", "TX", "TX", "TX", "TX", "TX", "TX", "TX", "TX"~
## $ agency_name       <chr> "ABERNATHY POLICE DEPT", "ABERNATHY POLICE DEPT", "A~
## $ item_name         <chr> "PISTOL,CALIBER .45,AUTOMATIC", "PISTOL,CALIBER .45,~
## $ quantity          <dbl> 1, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ ui                <chr> "Each", "Each", "Each", "Each", "Each", "Each", "Eac~
## $ acquisition_value <dbl> 58.71, 58.71, 58.71, 58.71, 58.71, 749.00, 749.00, 3~
## $ ship_date         <dttm> 2011-11-03, 2011-11-03, 2011-11-03, 2011-11-03, 201~
## $ station_type      <chr> "State", "State", "State", "State", "State", "State"~
## $ total_value       <dbl> 58.71, 58.71, 58.71, 58.71, 58.71, 749.00, 749.00, 1~
```

You should see the `tx` object in you Environment.

6.4 How to tackle summaries

As we get into the first quest, let's talk about "how" we do summaries.

When I am querying my data, I start by envisioning what the result should look like.

Let's take the first question: For each agency in Texas, find the summed **quantity** and summed **total value** of the equipment they received.

Let's break this down:

- "For each agency in Texas." For all the questions, we only want Texas agencies. We took care of this in nthe import book so TX agencies should already filtered.

- But the "For each agency" part tells me I need to **group_by** the `agency_name` so I can summarize totals within each agency.
- "find the summed **quantity** and summed **total_value**": Because I'm looking for a total (or `sum()` of columns) I need `summarize()`.

So I envision my result looking like this:

agency_name	summed_quantity	summed_total_value
AFAKE POLICE DEPT	6419	10825707.5
BFAKE SHERIFF'S OFFICE	381	3776291.52
CFAKE SHERIFF'S OFFICE	270	3464741.36
DFAKE POLICE DEPT	1082	3100420.57

The first columns in that summary will be our grouped values. This example is only grouping by one thing, `agency_name`. The other two columns are the summed values I'm looking to generate.

6.4.1 Summaries with math

We'll start with the **total_quantity**.

1. Add a new section (headline, text and chunk) that describes the first quest: For each agency in Texas, find the summed **quantity** and summed **total value** of the equipment they received.
2. Add the code below into the chunk and run it.

```
tx %>%
  group_by(agency_name) %>%
  summarize(
    sum_quantity = sum(quantity)
  )
```

```
## # A tibble: 357 x 2
##    agency_name                       sum_quantity
##    <chr>                                    <dbl>
##  1 ABERNATHY POLICE DEPT                       13
##  2 ALLEN POLICE DEPT                           11
##  3 ALVARADO ISD PD                              4
##  4 ALVIN POLICE DEPT                          539
##  5 ANDERSON COUNTY SHERIFFS OFFICE              8
##  6 ANDREWS COUNTY SHERIFF OFFICE               12
##  7 ANSON POLICE DEPT                            9
```

```
##  8 ANTHONY POLICE DEPT                       10
##  9 ARANSAS PASS POLICE DEPARTMENT            38
## 10 ARP POLICE DEPARTMENT                     18
## # ... with 347 more rows
```

Let's break this down a little.

- We start with the `tx` data, and then . . .
- We group by `agency_name`. This organizes our data (behind the scenes) so our summarize actions will happen *within each agency*. Now I normally say run your code one line at a time, but you would note be able to *see* the groupings, so I usually write `group_by()` and `summarize()` together.
- In `summarize()` we first name our new column: `sum_quantity`. We could call this whatever we want, but good practice is to name it what it is. We use good naming techniqes and split the words using `_`. I also use all lowercase characters.
- We set that column to equal `=` the **sum of all values in the `quantity` column**. `sum()` is the function, and we feed it the column we want to add together: `quantity`.
- I put the inside of the summarize function in its own line because we will add to it. I enhances readability. RStudio will help you with the indenting, etc.

If you look at the first line of the return, it is taking all the rows for the "ABERNATHY POLICE DEPT" and then adding together all the values in the `quantity` field.

If you wanted to test this (and it is a real good idea), you might look at the data from one of the values and check the math. Here are the Abernathy rows. I usually do these tests in a code chunk of their own, and sometimes I delete them after I'm sure it worked.

```
tx %>%
  filter(agency_name == "ABERNATHY POLICE DEPT")
```

```
## # A tibble: 9 x 9
##   state agency_name           item_name          quantity ui    acquisition_val~
##   <chr> <chr>                 <chr>                 <dbl> <chr>            <dbl>
## 1 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .4~        1 Each              58.7
## 2 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .4~        1 Each              58.7
## 3 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .4~        1 Each              58.7
## 4 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .4~        1 Each              58.7
## 5 TX    ABERNATHY POLICE DEPT PISTOL,CALIBER .4~        1 Each              58.7
## 6 TX    ABERNATHY POLICE DEPT RIFLE,5.56 MILLIM~        1 Each             749
## 7 TX    ABERNATHY POLICE DEPT RIFLE,5.56 MILLIM~        1 Each             749
```

```
## 8 TX    ABERNATHY POLICE DEPT SIGHT,REFLEX              5 Each            333
## 9 TX    ABERNATHY POLICE DEPT TRUCK,UTILITY             1 Each          62627
## # ... with 3 more variables: ship_date <dttm>, station_type <chr>,
## #   total_value <dbl>
```

If we look at the `quantity` column there and eyeball all the rows, we see there 8 rows with a value of "1," and one row with a value of "5." $8 + 5 = 13$, which matches our `sum_quantity` answer in our summary table. We're good!

6.4.2 Add the total_value

We don't have to stop at one summary. We can perform multiple summarize actions on the same or different columns within the same expression.

Edit your summary chunk to:

1. Add add a comma after the first summarize action.
2. Add the new expression to give us the `sum_total_value` and run it.

```
tx %>%
  group_by(agency_name) %>%
  summarize(
    sum_quantity = sum(quantity),
    sum_total_value = sum(total_value)
  )
```

```
## # A tibble: 357 x 3
##    agency_name                     sum_quantity sum_total_value
##    <chr>                                  <dbl>           <dbl>
##  1 ABERNATHY POLICE DEPT                     13          66084.
##  2 ALLEN POLICE DEPT                         11        1404024
##  3 ALVARADO ISD PD                            4            480
##  4 ALVIN POLICE DEPT                        539        2545240.
##  5 ANDERSON COUNTY SHERIFFS OFFICE            8         827891
##  6 ANDREWS COUNTY SHERIFF OFFICE             12           1476
##  7 ANSON POLICE DEPT                          9           5077
##  8 ANTHONY POLICE DEPT                       10           7490
##  9 ARANSAS PASS POLICE DEPARTMENT            38         571738
## 10 ARP POLICE DEPARTMENT                     18           5789.
## # ... with 347 more rows
```

6.4.3 Arrange the results

OK, this gives us our answers, but in alphabetical order. We want to arrange the data so it gives us the most `sum_total_value` in **desc**ending order.

1. EDIT your block to add an `arrange()` function below

```
tx %>%
  group_by(agency_name) %>%
  summarize(
    sum_quantity = sum(quantity),
    sum_total_value = sum(total_value)
  ) %>%
  arrange(sum_total_value %>% desc())
```

```
## # A tibble: 357 x 3
##    agency_name                       sum_quantity sum_total_value
##    <chr>                                    <dbl>           <dbl>
##  1 HOUSTON POLICE DEPT                       6419       10825708.
##  2 HARRIS COUNTY SHERIFF'S OFFICE             381        3776292.
##  3 DPS SWAT- TEXAS RANGERS                   1730        3520630.
##  4 JEFFERSON COUNTY SHERIFF'S OFFICE          270        3464741.
##  5 SAN MARCOS POLICE DEPT                    1082        3100421.
##  6 AUSTIN POLICE DEPT                        1458        2741021.
##  7 MILAM COUNTY SHERIFF DEPT                  125        2723192.
##  8 ALVIN POLICE DEPT                          539        2545240.
##  9 HARRIS COUNTY CONSTABLE PCT 3              293        2376945.
## 10 PARKS AND WILDLIFE DEPT                   5608        2325655.
## # ... with 347 more rows
```

6.4.4 Consider the results

Is there anything that sticks out in that list? It helps if you know a little bit about Texas cities and counties, but here are some thoughts to ponder:

- Houston is the largest city in the state (4th largest in the country). It makes sense that it tops the list. Same for Harris County or even the state police force. Austin being up there is also not crazy, as it's almost a million people.
- But what about San Marcos (63,220)? Or Milam County (24,770)? Those are way smaller cities and law enforcement agencies. They might be worth looking into.

Perhaps we should look some at the police agencies closest to us.

6.5 Looking a local agencies

Our next goal is this:

We'll take the summary above, but filter it to show only some local agencies of interest.

Since we are essentially taking an existing summary and adding more filtering to it, it makes sense to go back into that chunk and save it into a new object so we can reuse it.

1. EDIT your existing summary chunk to save it into a new tibble. Name it `tx_quants_totals` so we are all on the same page.
2. Add a new line that prints the result to the screen so you can still see it.

```
# adding the new tibble object in next line
tx_quants_totals <- tx %>%
  group_by(agency_name) %>%
  summarize(
    sum_quantity = sum(quantity),
    sum_total_value = sum(total_value)
  ) %>%
  arrange(sum_total_value %>% desc())

# peek at the result
tx_quants_totals
```

```
## # A tibble: 357 x 3
##    agency_name                        sum_quantity sum_total_value
##    <chr>                                     <dbl>           <dbl>
##  1 HOUSTON POLICE DEPT                        6419       10825708.
##  2 HARRIS COUNTY SHERIFF'S OFFICE              381        3776292.
##  3 DPS SWAT- TEXAS RANGERS                    1730        3520630.
##  4 JEFFERSON COUNTY SHERIFF'S OFFICE           270        3464741.
##  5 SAN MARCOS POLICE DEPT                     1082        3100421.
##  6 AUSTIN POLICE DEPT                         1458        2741021.
##  7 MILAM COUNTY SHERIFF DEPT                   125        2723192.
##  8 ALVIN POLICE DEPT                           539        2545240.
##  9 HARRIS COUNTY CONSTABLE PCT 3               293        2376945.
## 10 PARKS AND WILDLIFE DEPT                    5608        2325655.
## # ... with 347 more rows
```

The result is the same, but we can reuse the `tx_quants_totals` tibble.

6.5.1 Filtering within a vector

Let's talk through the filter concepts before you try it with this data.

When we talked about filtering with the Billboard project, we discussed using the | operator as an "OR" function. If we were to apply that logic here, it would look like this:

```
data %>%
  filter(column_name == "Text to find" | column_name == "More text to find")
```

That can get pretty unwieldy if you have more than a couple of things to look for.

There is another operator `%in%` where we can search for multiple items from a list. (This list of terms is officially called a vector, but whatever.) Think of it like this in plain English: *Filter* the *column* for things *in* this *list*.

```
data %>%
  filter(col_name %in% c("This string", "That string"))
```

We can take this a step further by saving the items in our list into an R object so we can reuse that list and not have to type out all the terms each time we use them.

```
list_of_strings <- c(
  "This string",
  "That string"
)

data %>%
  filter(col_name %in% list_of_strings)
```

6.5.2 Use the vector to build this filter

1. Create a new section (headline, text and chunk) and describe you are filtering the summed quantity/values for some select local agencies.
2. Create a saved vector list (like the list_of_strings above) of the five agencies we want to focus on. Call it `local_agencies`.
3. Start with the `tx_quants_totals` tibble you created for totals by agency and then use `filter()` and `%in%` to filter by your new `local_agencies` list.

These are the agencies:

```
AUSTIN POLICE DEPT
SAN MARCOS POLICE DEPT
```

```
TRAVIS COUNTY SHERIFFS OFFICE
UNIV OF TEXAS SYSTEM POLICE HI_ED
WILLIAMSON COUNTY SHERIFF'S OFFICE
```

> To be clear, in the interest of time I've done considerable work beforehand to figure out the exact names of these agencies. It helps that I'm familiar with local cities and counties so I used some creative filtering to find their "official" names in the data. I just don't want to get into how right now.

Use the example above to build with your data

```
local_agencies <- c(
  "AUSTIN POLICE DEPT",
  "SAN MARCOS POLICE DEPT",
  "TRAVIS COUNTY SHERIFFS OFFICE",
  "UNIV OF TEXAS SYSTEM POLICE HI_ED",
  "WILLIAMSON COUNTY SHERIFF'S OFFICE"
)

tx_quants_totals %>%
  filter(agency_name %in% local_agencies)
```

```
## # A tibble: 5 x 3
##   agency_name                          sum_quantity sum_total_value
##   <chr>                                       <dbl>           <dbl>
## 1 SAN MARCOS POLICE DEPT                       1082        3100421.
## 2 AUSTIN POLICE DEPT                           1458        2741021.
## 3 UNIV OF TEXAS SYSTEM POLICE HI_ED               3        1305000
## 4 TRAVIS COUNTY SHERIFFS OFFICE                 151         935354.
## 5 WILLIAMSON COUNTY SHERIFF'S OFFICE            210         431449.
```

6.6 Item quantities, totals for local agencies

Now that we have an overall idea of what local agencies are doing, let's dive a little deeper. It's time to figure out the specific items that they received.

Here is the quest: For each of the agencies above we'll use summarize to get the *summed* **quantity** and *summed* **total_value** of each **item** shipped to the agency. We'll create a summarized list for each agency so we can write about each one.

In some cases an agency might get the same item shipped to them at different times. For instance, APD has multiple rows of a single "ILLUMINATOR,INTEGRATED,SMALL ARM" shipped to them on the same date, and

at other times the quantity is combined as 30 items into a single row. We'll group our summarize by **item_name** so we can get the totals for both **quantity** and **total_value** for like items.

1. Create a new section (headline, text and first code chunk) and describe that you are finding the sums of each different item the agency has received since 2010.
2. Our first code chunk will start with the `tx` data, and then filter the results to just "AUSTIN POLICE DEPT."
3. Use `group_by` to group by `item_name`.
4. Use summarize to build the `summed_quantity` and `summed_total_value` columns.
5. Arrange the results so the most expensive items are at the top.

```
tx %>%
  filter(agency_name == "AUSTIN POLICE DEPT") %>%
  group_by(item_name) %>%
  summarize(
    summed_quantity = sum(quantity),
    summed_total_value = sum(total_value)
  ) %>%
  arrange(summed_total_value %>% desc())
```

```
## # A tibble: 46 x 3
##    item_name                                summed_quantity summed_total_val~
##    <chr>                                              <dbl>             <dbl>
##  1 HELICOPTER,FLIGHT TRAINER                              1            833400
##  2 IMAGE INTENSIFIER,NIGHT VISION                        85            467847.
##  3 SIGHT,THERMAL                                         29            442310
##  4 PACKBOT 510 WITH FASTAC REMOTELY CONTROLLE~            4            308000
##  5 SIGHT,REFLEX                                         420            144245.
##  6 ILLUMINATOR,INTEGRATED,SMALL ARMS                    135            122302
##  7 RECON SCOUT XT                                         8             92451.
##  8 RECON SCOUT XT,SPEC                                    6             81900
##  9 TEST SET,NIGHT VISION VIEWER                           2             56650
## 10 PICKUP                                                 1             26327
## # ... with 36 more rows
```

Please realize that this combines items that may have been shipped on any date our time period. If you want to learn more about *when* they got the items, you would have to build a new list of the data without grouping/summarizing.

6.6.1 Build the lists for other agencies

On your own . . .

1. Build a similar list for all the other local agencies. Basically you are just changing the filtering. You should end up with five chunks, each summarizing a different agency.

6.6.2 Google some interesting items

You'll want some more detail in your data drop about some of these specific items.

1. Do some Googling on some of these items of interest to learn more about them. I realize (and you should, too) that for a "real" story we would need to reach out to sources for more information, but you can get a general idea from what you find online for the writing assignment below.

6.7 Write a data drop

Once you've found answers to all the questions listed, you'll weave those into a writing assignment. Include this as a Microsoft Word document saved into your project folder along with your notebooks. (If you are a Google Docs fan, you can write there and then export as a Word doc.)

You will **not** be writing a full story ... we are just practicing writing a lede and "data sentences" about what you've found. You *do* need to source the data and briefly describe the program but this is not a fully-fleshed story. Just concentrate on how you would write the facts and attribution. You will want to refer back to Chapter 5.1 and the subchapters there for more information.

1. Write a data drop from the data of between four and six paragraphs. Be sure to include attribution about where the data came from.
2. You can pick the lede angle from any of the questions outlined above. Each additional paragraph should describe what you found from the data.
3. Use Microsoft Word and include it inside your stuffed project when you upload it to Canvas.

Here is a **partial** example to give you an idea of what I'm looking for. (These numbers may be old and you can't use this angle as your lede ;-)).

> The Jefferson County Sheriff's Office is flying high thanks to gifts of over $3.5 million worth of surplus U.S. Department of Defense equipment.
>
> Among the items transferred over the past decade to the department was a $923,000 helicopter in October 2016 and related parts the

following year, according to data from the Defense Logistics Agency data — the agency that handles the transfers.

The sheriff's office has received the fourth highest value of equipment among any law enforcement agency in Texas since August 2014 despite being a county of only 250,000 people.

6.8 What we learned in this chapter

- We used `sum()` within a `group_by()`/`summarize()` function to add values within a column.
- We used `summary()` to get descriptive statistics about our data, like the minimum and maximum values, or an average (mean).
- We learned how to use `c()` to **combine** a list of like values into a *vector*, and then used that vector to filter a column for valus `%in%` that vector.

Chapter 7

Intro to ggplot

7.1 Goals for this section

- An introduction to the Grammar of Graphics
- We'll make charts!

7.2 Introduction to ggplot

ggplot2 is the data visualization library within Hadley Wickham's tidyverse. It uses a concept called the Grammar of Graphics, the idea that you can build every graph from the same components: a data set, a coordinate system, and geoms – the visual marks that represent data points.

Even though the package is called `ggplot2`, the function to make graphs is just `ggplot()`. I will often just call everything `ggplot`.

7.2.1 What I like/dislike about ggplot

The ggplot system allows you to display data right in your notebook. It is really good at helping you find important things in your data that can inform reporting. It's an important tool in your R-based data journalism toolkit.

What ggplot is less good at is creating publishable graphics. Don't get me wrong ... you can do it, but nuances of the ggplot system take time to master at that level. There are other tools (like Datawrapper and Flourish) that can do that better, even without code. That said, there is a place for R in that workflow, too. We'll cover using Datawrapper in a later chapter.

7.2.2 The Grammar of Graphics

ggplot uses this concept of the Grammar of Graphics ... i.e., that you can use code to describe how to build a chart layer-by-layer.

With a hat tip to Matt Waite, we can describe the components of the Grammar of Graphics as:

- **data**: which data you are pulling from for the chart.
- **aesthetics**: describes how to apply specific data to the plot. What is on x axis, what is on y axis, for starters.
- **geometries**: the shape the data is going to take on the graph. lines, columns, points.
- **scales**: any transformations we might make on the data.
- **layers**: how we might layer multiple geometries over top of each other to reveal new information.
- **facets**: how we might graph many elements of the same data set in the same space.

What to remember here is this: for every graphic we start with the data, and then we build a chart from it one "layer" at a time.

The best way to learn this system is to do it and explain along the way.

7.3 Start a new project

1. Get into RStudio and make sure you don't have any other files or projects open.
2. Create a new project, name it `yourname-ggplot` and save it in your rwd folder.
3. No need to create our folder structure ... we won't need it here.
4. Start a new RMarkdown notebook and save it as `01-intro-ggplot.Rmd`.
5. Remove the boilerplate and create a setup section that loads `library(tidyverse)`, like we do with every notebook.

7.4 The layers of ggplot

> Much of this first plot explanation comes from Hadley Wickham's R for Data Science, with edits to fit the lesson here.

We're going to use a data set that is part of the tidyverse to explore how ggplot works.

1. Start a new section "First plot" and add a code chunk.
2. Add the code below and run it to see what the mpg dataset looks like.

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4           1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4           2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4           2    2008     4 auto~ f        21    30 p     comp~
##  5 audi         a4           2.8  1999     6 auto~ f        16    26 p     comp~
##  6 audi         a4           2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4           3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 quattro   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 quattro   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 quattro   2    2008     4 manu~ 4        20    28 p     comp~
## # ... with 224 more rows
```

The `mpg` data contains observations collected by the US Environmental Protection Agency on 38 models of cars. It's a data set embedded into the tidyverse for lessons like this one.

Among the variables in mpg are:

- `displ`, a car's engine size, in liters.
- `hwy`, a car's fuel efficiency on the highway, in miles per gallon (mpg).

With these two variables we can test the theory that cars with smaller engines (`displ`) get better gas mileage (`hwy`).

7.4.1 Build the base layer

With ggplot2, you begin a plot with the function`ggplot()`. `ggplot()` creates a coordinate system that you can add layers to. The first argument for `ggplot()` is the data set to use in the graph So `ggplot(mpg)` creates an empty graph with no axes or anything.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot. ggplot2 comes with many geom functions that each add a different type of layer to a plot.

Each geom function in ggplot2 takes a `mapping` argument. This defines how variables in your dataset are mapped to visual properties. The `mapping` argument is always paired with `aes()`, and the `x` and `y` arguments of `aes()` specify which variables to map to the x and y axes. ggplot2 looks for the mapped variables in the `data` argument, in this case, `mpg`.

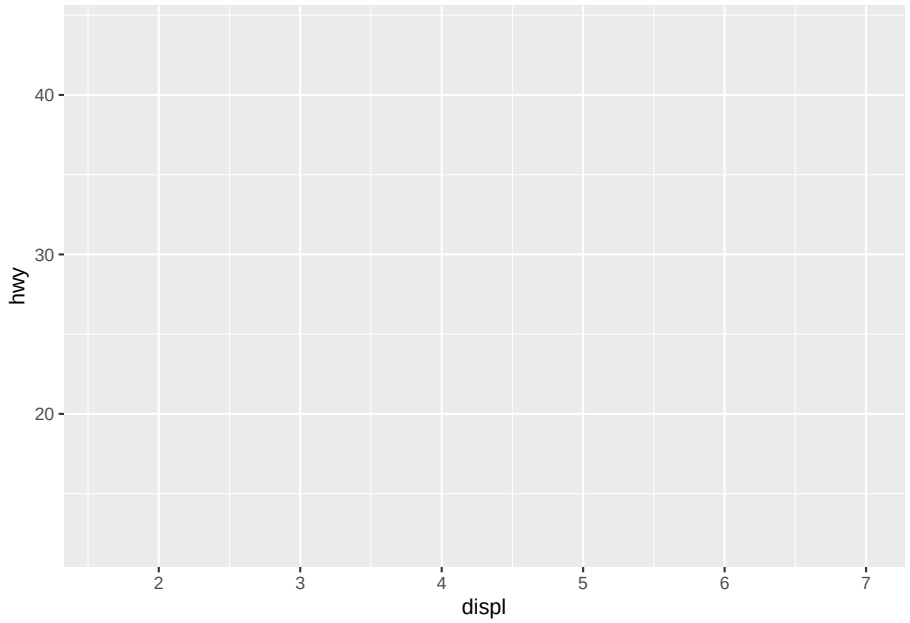
We can apply `aes()` mappings to the graph as a whole and/or to the individual geom layers.

Frankly, it is easier to show this than explain it. The code below sets up the grid and axes lines, but it hasn't placed any data on the plot.

Do this:

1. Add some text that you are building the mpg chart.
2. Add the code chunk below and run it.

```
ggplot(mpg, aes(x = displ, y = hwy))
```



Let's work through the code above:

- `ggplot()` is our function to make a chart.
- The first argument `ggplot()` needs is the data. It could be specified as `data = mpg` but we don't need the `data =` part as it is always the first item specified inside of (or piped into) `ggplot()`

- Next is the **aesthetics** or `aes()`. This is where tell ggplot what data to plot on the `x` and `y` axis. You might see this as `mapping = aes(<VALUES>)` but we can often get by without the `mapping =` part.

In our case we are applying these `aes()` to the entire chart. You'll see later we can also apply different `aes()` to specific geoms.

7.4.2 Layers can we add to our plots

We'll now add onto this base layer a number of things:

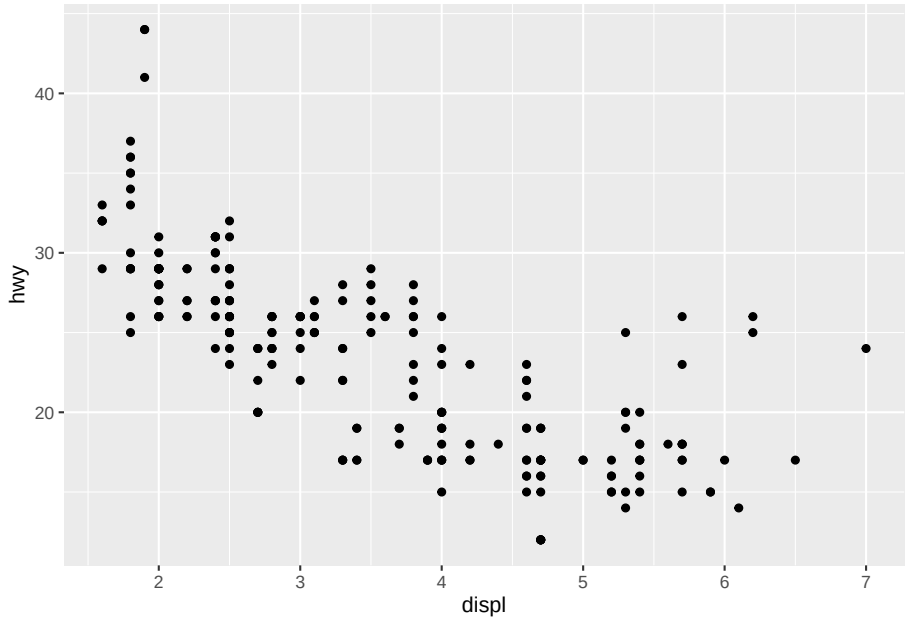
- **geometries** (or geoms as we call them) are the way we plot data on the base grid. There are many geoms, but here are a few common ones:
  - `geom_points()` adds dots onto the grid based on the data. Will will use these here to build a scatterplot graph.
  - `geom_line()` adds lines between data points on the grid. Basically a line chart.
  - `geom_col()` and `geom_bars()` adds bars to the grid based on values in the data. A bar chart. We'll use `geom_col()` later in this lesson but you can read about the difference between the two in a later chapter.
  - `geom_text()` adds labels based on values in the data.
- **labels** (or labs, since we use the `labs()` function for them) are a series of text-based items we can layer onto our plots like titles, bylines and axis names.
- **themes** change the visual styles of the grids and axis. There are several available within ggplot and many other from the R community.

We add layers onto the chart using the `+` at the end of a line. Think of the `+` as the `%>%` of ggplot.

7.4.3 Add geom_points

1. EDIT your plot chunk to add the `+` and a new line for `geom_point()`

```
ggplot(mpg, aes(x = displ, y = hwy)) + # don't forget the + at the end of this line
  geom_point() # the geom_point
```



The `geom_point()` function above is inheriting the `aes()` values from the line above it.

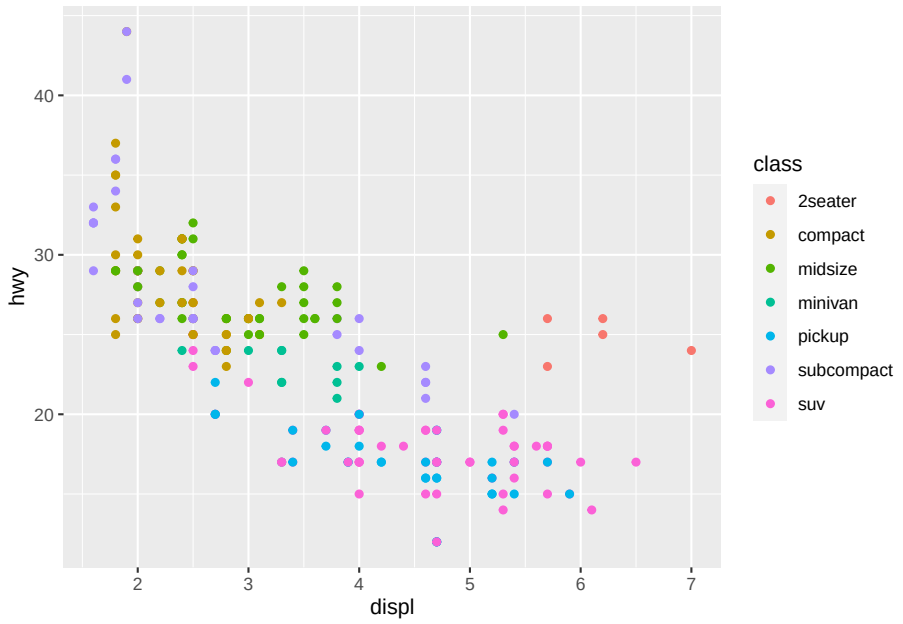
7.4.4 Adding other mappings

We can add aesthetics to either the plot as a whole (which we did with the x and y values above) and those will apply to all the geoms unless overwritten.

But we can also add aesthetics to specific geoms. We'll demonstrate this below.

1. Edit your `geom_point()` function to add a color mapping to the points with `aes(color = class)`. `color` is the type of aesthetic, and `class` is a column in the data.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) # this is the line you are editing
```



As you can see, the dots were given colors based on the values in the `class` column, and ggplot also added a legend to the graphic.

There are other aesthetics you can use.

1. Change the `color` aesthetic to one of these values and run it to see how it affects the chart: `alpha`, `size` and `shape`. (i.e., `alpha = class`.)
2. Once you've tried them, change it back to `color`.

OK, enough of the basics ... let's build a chart you *might* care about.

7.5 Let's build a bar chart

We'll build some charts from our first-day survey where you told me your favorite Disney Princess and favored flavor of ice cream.

We aren't going to create different notebooks or download the data to to your computer ... we're just doing to save it directly into a tibble.

1. Start a new section: Princess preference chart.
2. Use text to note that we'll use class data to build a chart.
3. Add the code below to get the data.

```
# read the data and fill the tibble: class
class <- read_csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vRnSAx9eBoOGdZ3pMLZ
```

```
## Rows: 34 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): name, princess, ice_cream
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# peek at the data
class
```

```
## # A tibble: 34 x 3
##    name     princess                      ice_cream
##    <chr>    <chr>                         <chr>
##  1 Addie    Rapunzel (Tangled)            Cookie Dough
##  2 Aisling  Pocahontas                     Mint Chocolate Chip
##  3 Alexis   Jasmine (Aladdin)             Cookie Dough
##  4 Ana      Belle (Beauty and the Beast)  Cookies and Cream
##  5 Andreana Mulan                         Strawberry
##  6 Angelica Tiana (Princess and the Frog) Strawberry
##  7 Ariana   Merida (Brave)                Coffee/Jamoca
##  8 Cecilia  Cinderella                    Mint Chocolate Chip
##  9 Chandle  Jasmine (Aladdin)             Cookie Dough
## 10 Chris    Cinderella                    Cookie Dough
## # ... with 24 more rows
```

7.5.1 Prepare the data

While there are ways for ggplot to calculate values from your data on the fly, I much prefer to first build a table of the values I want plotted on a chart.

Our goal here is to make a bar (or column) chart showing the number of votes for each princess from the data. So, we need to count the number of rows for each value ... our typical group_by/summarize/arrange process. I'm going to use the `count()` shortcut for our GSA here since we haven't used it much lately. I'm saving the summarized data into a new dataframe called `princess_data`. Follow along in your notebook:

1. Add a new section: Princess chart

2. Add text that you are creating a data frame to plot.
3. Add the code below to create that data.

```
princess_data <- class %>%
  count(princess, name = "votes", sort = TRUE)
  # this above line counts the princess rows, sets the name and sorts

# peek at the data
princess_data
```

```
## # A tibble: 10 x 2
##    princess                      votes
##    <chr>                         <int>
##  1 Mulan                             8
##  2 Jasmine (Aladdin)                 4
##  3 Pocahontas                        4
##  4 Aurora (Sleeping Beauty)          3
##  5 Belle (Beauty and the Beast)      3
##  6 Cinderella                        3
##  7 Rapunzel (Tangled)                3
##  8 Tiana (Princess and the Frog)     3
##  9 Merida (Brave)                    2
## 10 Ariel (Little Mermaid)            1
```

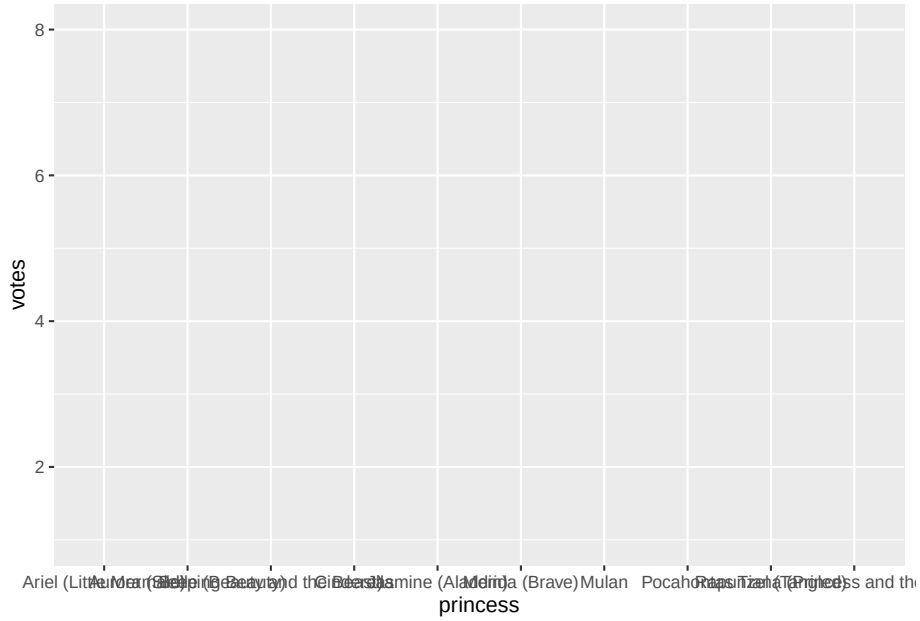
I hope you understand what we've done here. We're counting the number of rows for each princess.

7.5.2 Build our plot with geom_col

Our first goal is to build the first layer the plot ... basically tell ggplot what data we are using so it will build the grid to hold our plots.

1. Add some text noting that you'll now plot.
2. Add the following code chunk that starts the plot and run it.

```
ggplot(princess_data, aes(x = princess, y = votes)) # sets our x and y axes
```



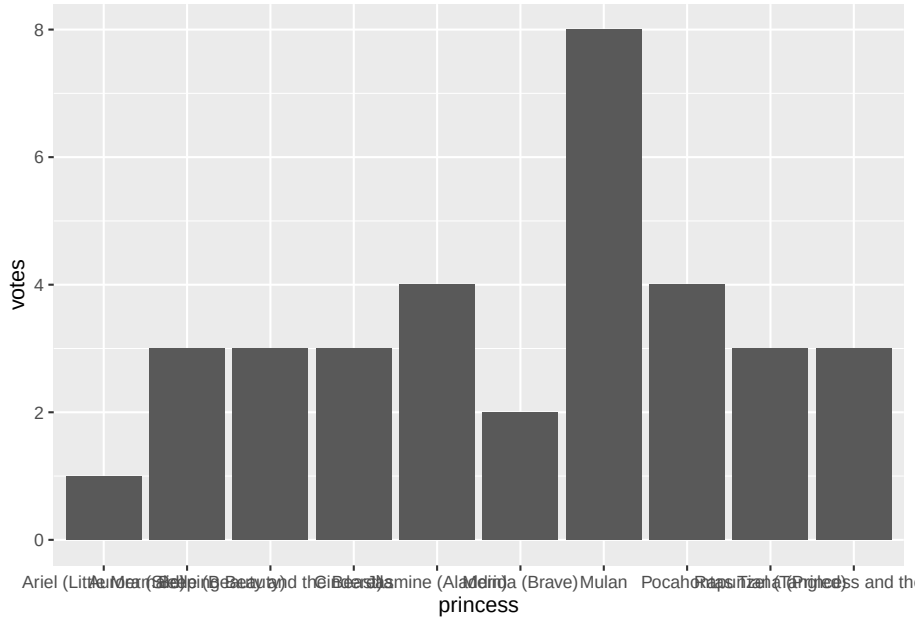
You'll see the grid and x/y axis of the data, but no geometries are applied yet. Don't worry yet now it looks ... we'll get there.

7.5.3 Add the geom_col layer

Now it is time to add our columns.

1. Edit the plot code to add the ggplot pipe `+` and on the next line add `geom_col()`.

```
ggplot(princess_data, aes(x = princess, y = votes)) + # don't forget the + on this lin
  geom_col() # adds the bars
```



This added our data to the plot, though there are a couple of issues:

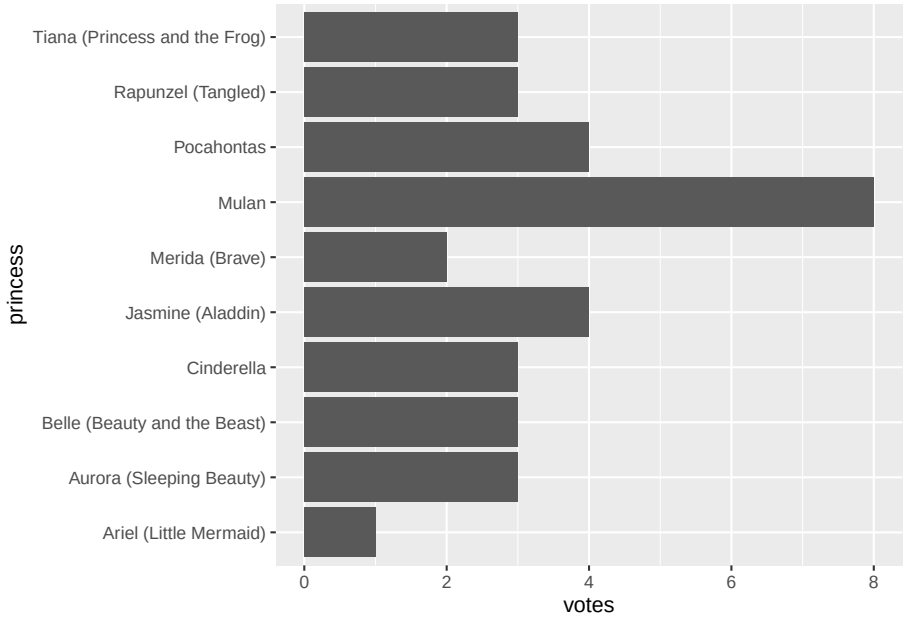
- We can't read the value names. We can fix this.
- The order of the bars is alphabetical instead of in vote order. Again, we can fix it.

7.5.4 Flip the axis

We can "flip" the axis to turn it sideways to read the labels. This can be a bit confusing later because the "x" axis is now going up/down.

1. Edit your plot chunk to add the ggplot pipe `+`and `coord_flip()` on the next line.

```
ggplot(princess_data, aes(x = princess, y = votes)) +
  geom_col() + # don't forget the +
  coord_flip() # flips the axis
```



7.5.5 Reorder the bars

The bars on our chart are in alphabetical order of the x axis (and reversed thanks to our flip.) We want to order the values based on the `votes` in the data.

> Complication alert: Categorical data can have factors, which are like an internal ordering system. Some categories, like months in a year, have an "order" that is not alphabetical.

We can reorder our categorical values in a plot by editing the `x` values in our `aes()` using `reorder()`. (There is a tidyverse function called `fct_reorder()` that works the same way.

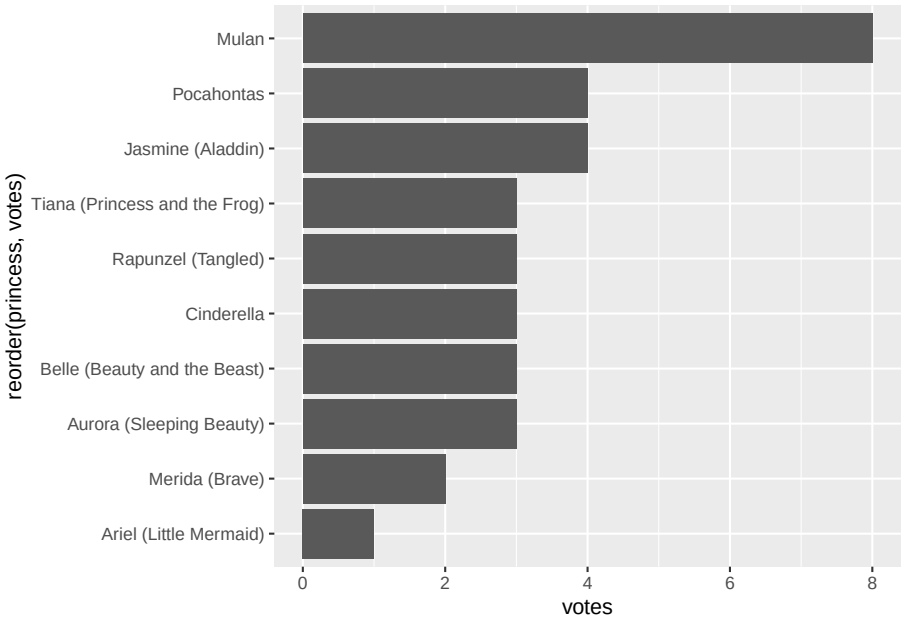
`reorder()` takes two arguments: The column to reorder, and the column to base that reorder on. It can happen in two different ways, and I'll be honest and say I don't know which is easier to comprehend.

- `x = reorder(princess, votes)` says "set the x axis as `princess`, but order as `votes`. OR ...
- `x = princess %>% reorder(votes)` says "set the x axis as `princess` *and then* reorder by `votes`.

They both work. Even though I'm a fan of the tidyverse `%>%` construct, I'm going with the first version.

1. Edit your chunk to reorder the bars.

```
ggplot(princess_data, aes(x = reorder(princess, votes), y = votes)) + # this is the line you edit
  geom_col() +
  coord_flip()
```

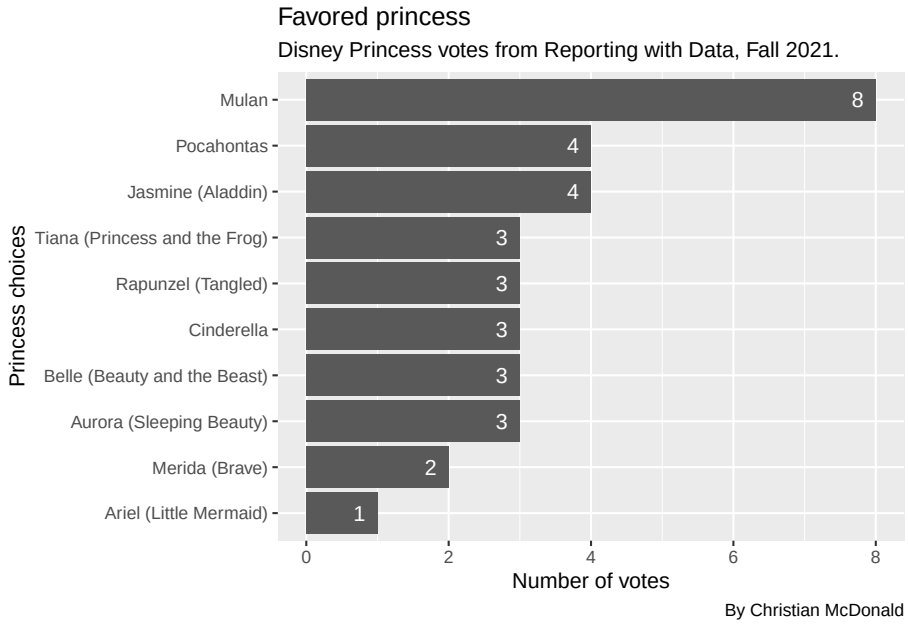


7.5.6 Add some titles, labels

Now we'll add a **layer** of labels to our chart using the the `labs()` function. You'll see we can add and change a number of things with `labs()`.

- To add the labels to the bars, we use a `geom_text()` because we are actually plotting them on the graph. The example below also changes the color of the text and moves the labels to inside the bar with `hjust` (or horizontal justification. `vjust` would move it up and down).
- The `labs()` function allows for labels "around" the chart. These are some standard values used.

```
ggplot(princess_data, aes(x = reorder(princess, votes), y = votes)) +
  geom_col() +
  coord_flip() + # don't forget +
  geom_text(aes(label = votes), hjust = 2, color = "white") + # plots votes text value
  # labs below has several settings
  labs(
    title = "Favored princess", # adds a title
    subtitle = "Disney Princess votes from Reporting with Data, Fall 2021.", # adds a
    caption = "By Christian McDonald", # adds the byline
    x = "Princess choices", # renames the x axis label (which is really y since it is
    y = "Number of votes" # renames the y axis label (which is really x since it is fl
  )
```



There you go! You've made a chart showing how our classes rated Disney Princesses.

7.6 On your own: Ice cream!

Now it is time for you to put these skills to work:

1. Build a chart about the favorite ice creams from RWD classes.

Some things to consider:

- You need a new section, etc.
- You're starting with the same `class` data
- You need to prepare the data based on `ice_cream`
- You need to build the chart

It's essentially the same process we used for the princess chart, but using `ice_cream` variable.

7.7 What we've learned

There is a ton, really.

- ggplot2 (which is really the `ggplot()` function) is the charting library for the tidyverse. This whole lesson was about it.

Here are some more references for ggplot:

- The ggplot2 documentation and ggplot2 cheatsheets.
- R for Data Science, Chap 3. Hadley Wickam dives right into plots in his book.
- R Graphics Cookbook has lots of example plots. Good to harvest code and see how to do things.
- The R Graph Gallery another place to see examples.

Chapter 8

Deeper into ggplot

In the last chapter you were introduced to ggplot2, the graphic function that is part of the tidyverse. With this chapter we'll walk through more ways to use ggplot. This will also serve as a reference for you. A huge hat tip to Jo Lukito. One of her lessons inspired most of this.

8.1 References

ggplot2 has a LOT to it and we'll cover only the basics. Here are some references you might use:

- ggplot cheatsheet
- R for Data Science
- R Graphics Cookbook
- The R Graph Gallery another place to see examples.
- ggplot2: Elegant graphics for Data Analysis

8.2 Learning goals for this chapter

Some things we'll touch on concerning ggplot:

- Prepare and build a line chart
- Using themes to change the look of our charts
- Adding/changing aesthetics in layers
- Facets (multiple charts from same data)
- Saving files
- Interactivity with Plotly

8.3 Set up your notebook

We'll use the same `yourname-ggplot` project we used in the last chapter, but start a new RNotebook.

1. Open your plot project.
2. Start a new RNotebook. Add the goals listed above.
3. Load the tidyverse package.

8.3.1 Let's get the data

> I hope to demonstrate in class the creation of this first plot. Otherwise you should be able to follow along in the screencast.

Again, we won't download the data ... we'll just import it and save it to a tibble. We are using data from a weekly project called #tidytuesday that the community uses to practice R. Perhaps in the near future we'll have our own #tidytuesday sessions!

1. Start a new section to indication you are importing the data
2. Note in text it is from #tidytuesday
3. Add the code chunk below, which will download a saved copy of the data.

```
kids_data <- read_rds("https://github.com/utdata/rwdir/blob/main/data-raw/kids-data.rds

# peek at the table
kids_data
```

```
## # A tibble: 23,460 x 6
##    state                variable  year      raw   inf_adj inf_adj_perchild
##    <chr>                <chr>    <dbl>    <dbl>     <dbl>            <dbl>
##  1 Alabama              PK12ed    1997  3271969  4665308.             3.93
##  2 Alaska               PK12ed    1997  1042311  1486170              7.55
##  3 Arizona              PK12ed    1997  3388165  4830986.             3.71
##  4 Arkansas             PK12ed    1997  1960613  2795523              3.89
##  5 California           PK12ed    1997 28708364 40933568              4.28
##  6 Colorado             PK12ed    1997  3332994  4752320.             4.38
##  7 Connecticut          PK12ed    1997  4014870  5724568.             6.70
##  8 Delaware             PK12ed    1997   776825  1107629.             5.63
##  9 District of Columbia PK12ed    1997   544051   775730.             6.11
## 10 Florida              PK12ed    1997 11498394 16394885              4.45
## # ... with 23,450 more rows
```

```
# glimpse it
kids_data %>% glimpse()
```

```
## Rows: 23,460
## Columns: 6
## $ state            <chr> "Alabama", "Alaska", "Arizona", "Arkansas", "Californ~
## $ variable         <chr> "PK12ed", "PK12ed", "PK12ed", "PK12ed", "PK12ed", "PK~
## $ year             <dbl> 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997,~
## $ raw              <dbl> 3271969, 1042311, 3388165, 1960613, 28708364, 3332994~
## $ inf_adj          <dbl> 4665308.5, 1486170.0, 4830985.5, 2795523.0, 40933568.~
## $ inf_adj_perchild <dbl> 3.929449, 7.548493, 3.706679, 3.891275, 4.282325, 4.3~
```

If you want to learn more about this dataset you can find information here. In short: "This dataset provides a comprehensive accounting of public spending on children from 1997 through 2016." Included is spending for higher education (the `highered` values in the column `variable`.)

We'll filter this data to get to our data of interest: How much does Texas (and some neighboring states) spend on higher education.

8.4 Make a line chart of the Texas data

Our first goal here is to plot the "inflation adjusted spending per child" for higher education in Texas.

8.4.1 Prepare the data

We need to filter our data to include just the `highered` values for Texas. We're going to save that filtered data into a new tibble.

1. Add a new section indicating we are building a chart to show higher education spending in Texas.
2. Note that we are preparing the data.
3. Add the chunk below and run it.

```
tx_hied <- kids_data %>%
  filter(
    variable == "highered",
    state == "Texas"
  )

# peek at the data
tx_hied
```

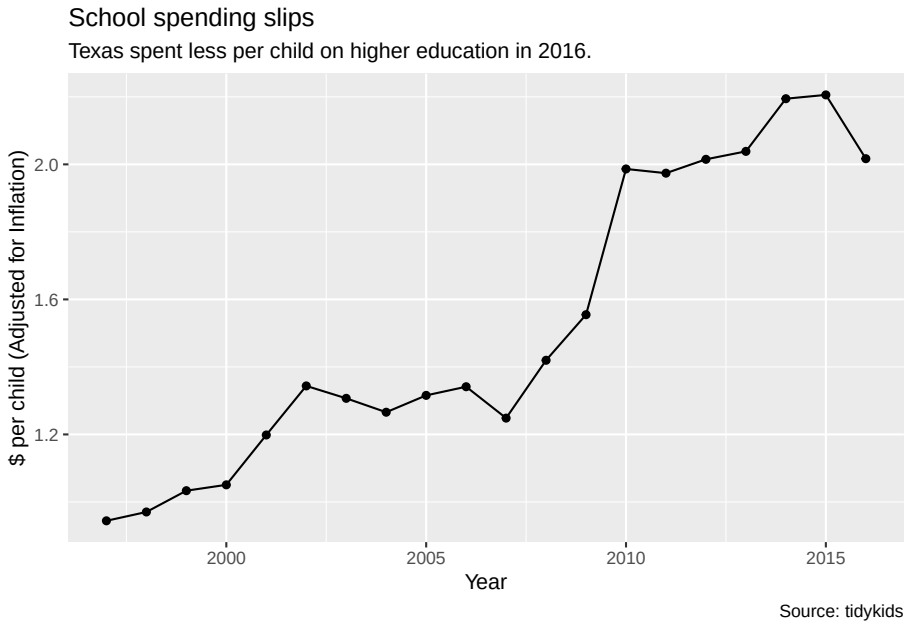
```
## # A tibble: 20 x 6
##    state variable  year       raw   inf_adj inf_adj_perchild
##    <chr> <chr>    <dbl>     <dbl>     <dbl>            <dbl>
##  1 Texas highered  1997  3940232   5618146.            0.944
##  2 Texas highered  1998  4185619   5895340.            0.970
##  3 Texas highered  1999  4578617   6368075             1.03
##  4 Texas highered  2000  4810358   6554888.            1.05
##  5 Texas highered  2001  5684406.  7564852             1.20
##  6 Texas highered  2002  6558453   8589044             1.34
##  7 Texas highered  2003  6584970.  8462055             1.31
##  8 Texas highered  2004  6611486   8290757             1.27
##  9 Texas highered  2005  7180804   8730524             1.32
## 10 Texas highered  2006  7744386   9119121             1.34
## 11 Texas highered  2007  7540724   8644579             1.25
## 12 Texas highered  2008  8914255  10011380             1.42
## 13 Texas highered  2009 10039289  11145217             1.55
## 14 Texas highered  2010 13097474  14413453             1.99
## 15 Texas highered  2011 13366868  14416946             1.97
## 16 Texas highered  2012 13999386  14828316             2.01
## 17 Texas highered  2013 14520493  15124855             2.04
## 18 Texas highered  2014 16101982  16470816             2.19
## 19 Texas highered  2015 16591235  16773450             2.21
## 20 Texas highered  2016 15507047  15507047             2.02
```

8.4.2 Plot the chart

I want you to create the plot here one step at a time so you can review how the layers are added.

1. Add and run the ggplot() line first (but without the `+`)
2. Then add the `+` and the `geom_point()` and run it.
3. Then add the `+` and `geom_line()` and run it.
4. When you add the `labs()` add all the lines at once and run it.

```
ggplot(tx_hied, aes(x = year, y = inf_adj_perchild)) + # we create our graph
  geom_point() + # adding the points
  geom_line() +  # adding the lines between points
  labs(
    title = "School spending slips",
    subtitle = "Texas spent less per child on higher education in 2016.",
    x = "Year", y = "$ per child (Adjusted for Inflation)",
    caption = "Source: tidykids"
  )
```




```
  # labs above add text layer on top of graph
```

We have a pretty decent chart showing `year` on our x axis and `inf_adj_perchild` (or inflation-adjusted spending per child) on our y axis.

8.4.3 Saving plots as an object

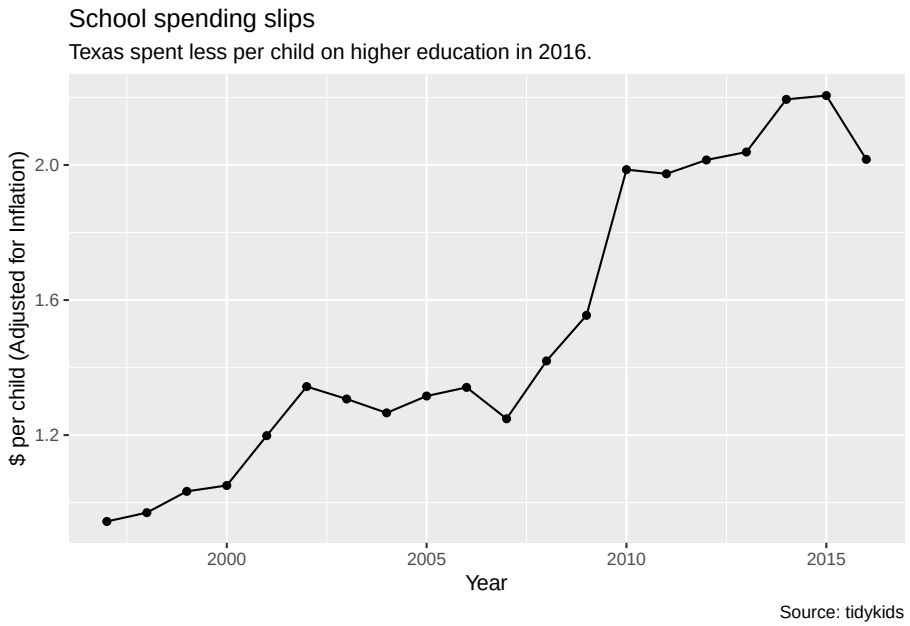
Sometimes it is helpful to push the results of a plot into an R object to "save" those configurations. You can continue to add layers after, but don't have to rebuild the basic chart each time. We'll do that here so we can explore themes next.

1. Edit your Texas plot chunk you made earlier to save it into an R object, and then call `tx_plot` after it so you can see it.

```
# the line below pushes the graph results into tx_plot
tx_plot <- ggplot(tx_hied, aes(x = year, y = inf_adj_perchild)) +
  geom_point() +
  geom_line() +
  labs(
    title = "School spending slips",
    subtitle = "Texas spent less per child on higher education in 2016.",
    x = "Year", y = "$ per child (Adjusted for Inflation)",
```

```
    caption = "Source: tidykids"
  )

# Since we saved the plot into an R object above, we have to call it again to see it.
# We save graphs like this so we can reuse them.
tx_plot
```



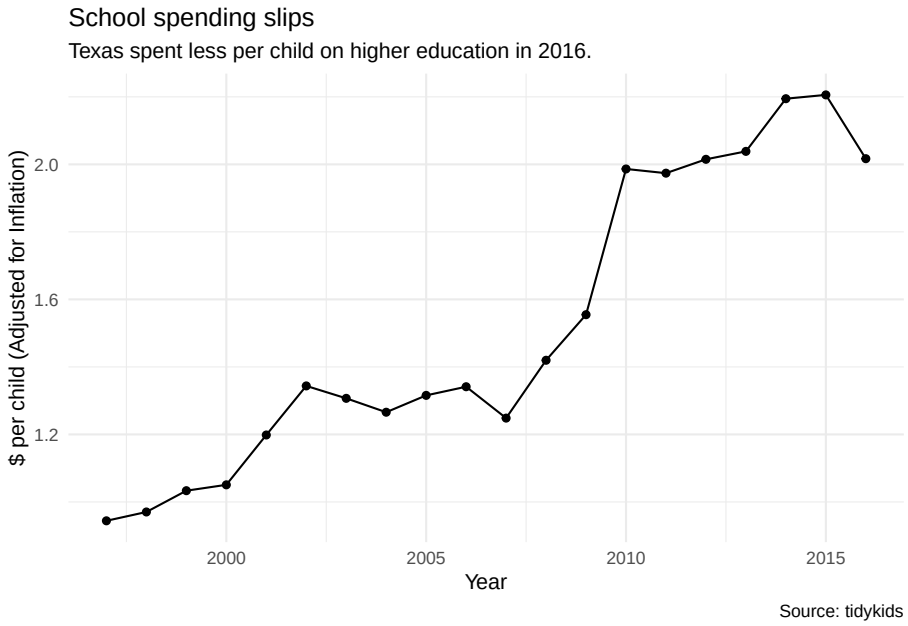
We can continue to build upon the `tx_plot` object like we do below with themes, but those changes won't be "saved" into the R environment unless you assign it to an R object.

8.5 Themes

The *look* of the graph is controlled by the theme. There are a number of preset themes you can use. Let's look at a couple.

1. Create a new section saying we'll explore themes
2. Add the chunk below and run it.

```
tx_plot +
  theme_minimal()
```



This takes our existing `tx_plot` and then applies the `theme_minimal()` look to it.

There are a number of themes built into ggplot, most are pretty simplistic.

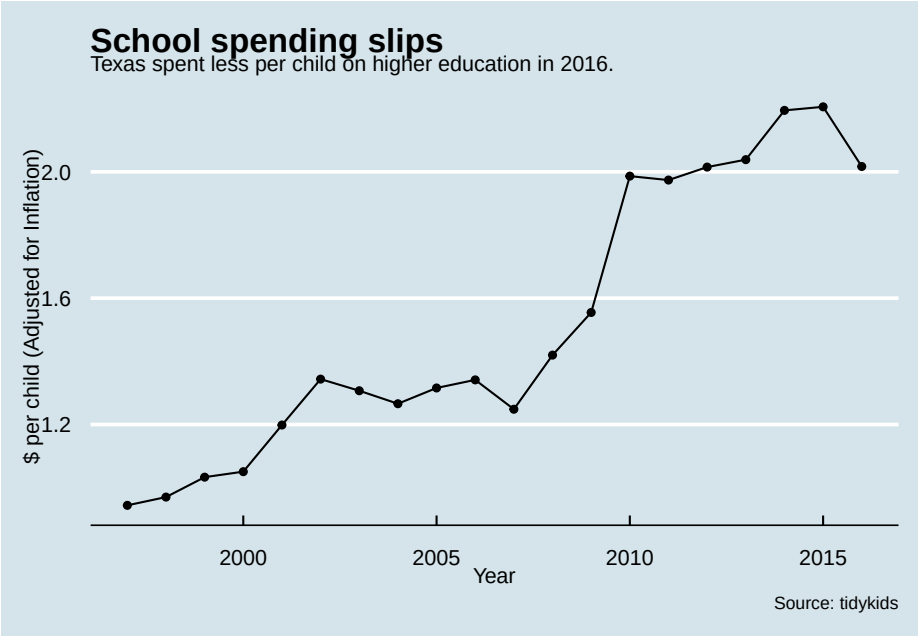
1. Edit your existing chunk to try different themes. Some you might try are `theme_classic()`, `theme_dark()` and `theme_void()`.

8.5.1 More with ggthemes

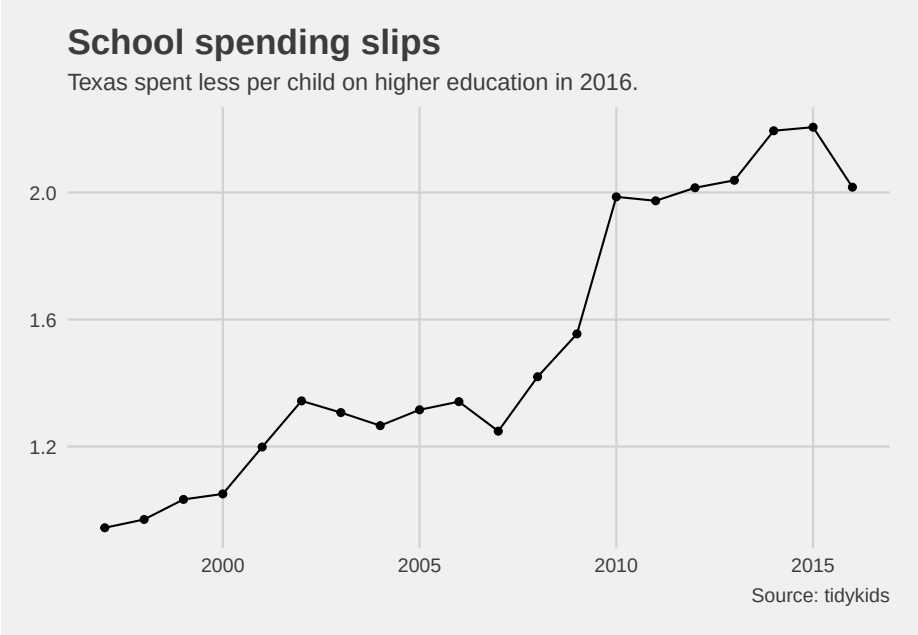
There are a number of other packages that build upon `ggplot2`, including `ggthemes`.

1. In your R console, install the ggthemes package: `install.packages("ggthemes")`
2. Add the `library(ggthemes)` at the top of your current chunk.
3. Update the theme line to view some of the others options noted below.

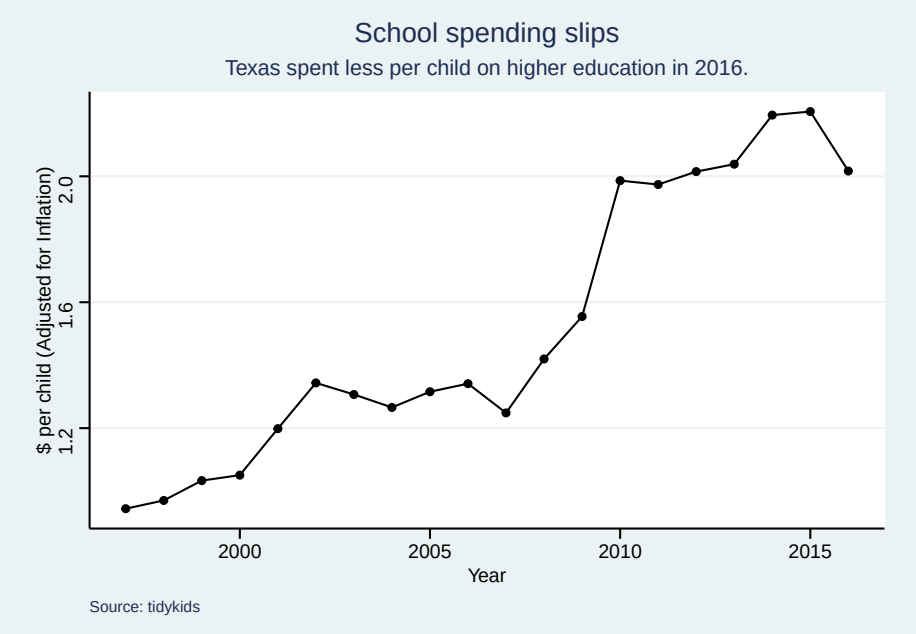
```
library(ggthemes)
tx_plot +
  theme_economist()
```




```
tx_plot +
  theme_fivethirtyeight()
```



```
tx_plot +
  theme_stata()
```



8.5.2 There is more to themes

There is also a `theme()` function that allows you individually adjust about every visual element on your plot.

We do a wee bit of that later.

8.6 Adding more information

OK, our Texas higher education spending is fine . . . but how does that compare to neighboring states? Let's work through building a new chart that shows all those steps.

8.6.1 Prepare the data

We need to go back to our original `kids_data` to get the additional states.

1. Start a new section that notes we are building a chart for five states.

2. Note that we'll first prepare the data.

```
five_hied <- kids_data %>%
  filter(
    variable == "highered",
    state %in% c("Texas", "Oklahoma", "Arkansas", "New Mexico", "Louisiana")
  )

five_hied
```

```
## # A tibble: 100 x 6
##    state      variable  year     raw  inf_adj inf_adj_perchild
##    <chr>      <chr>    <dbl>   <dbl>    <dbl>            <dbl>
##  1 Arkansas   highered  1997  457171  651853.            0.907
##  2 Louisiana  highered  1997  672364  958684.            0.731
##  3 New Mexico highered  1997  639409  911696.            1.68
##  4 Oklahoma   highered  1997  624053  889800.            0.942
##  5 Texas      highered  1997 3940232 5618146.            0.944
##  6 Arkansas   highered  1998  477757  672909.            0.930
##  7 Louisiana  highered  1998  747739 1053172.            0.805
##  8 New Mexico highered  1998  667738  940492.            1.74
##  9 Oklahoma   highered  1998  690234  972177.            1.02
## 10 Texas      highered  1998 4185619 5895340.            0.970
## # ... with 90 more rows
```

Note we used our `%in%` filter to get any state listed in `c()`.

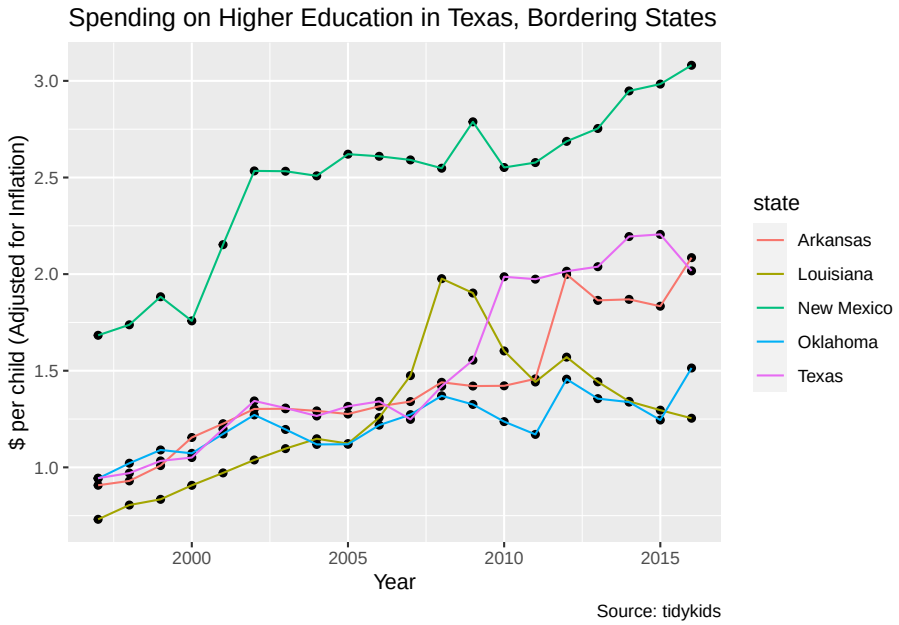
8.6.2 Plot multiple line chart

Let's add a different line for each state. To do this you would use the color aesthetic `aes()` in the `geom_line()` geom. Recall that geoms can have their own `aes()` variable information. This is especially useful for working with a third variable (like when making a stacked bar chart or line plot with multiple lines). Notice that the color aesthetic (meaning that it is in aes) takes a variable, not a color. You can learn how to change these colors here.

1. Add a note that we'll now build the chart.
2. Add the code chunk below and run it. Look through the comments so you understand it.

```
ggplot(five_hied, aes(x = year, y = inf_adj_perchild)) +
  geom_point() +
  geom_line(aes(color = state)) + # The aes selects a color for each state
```

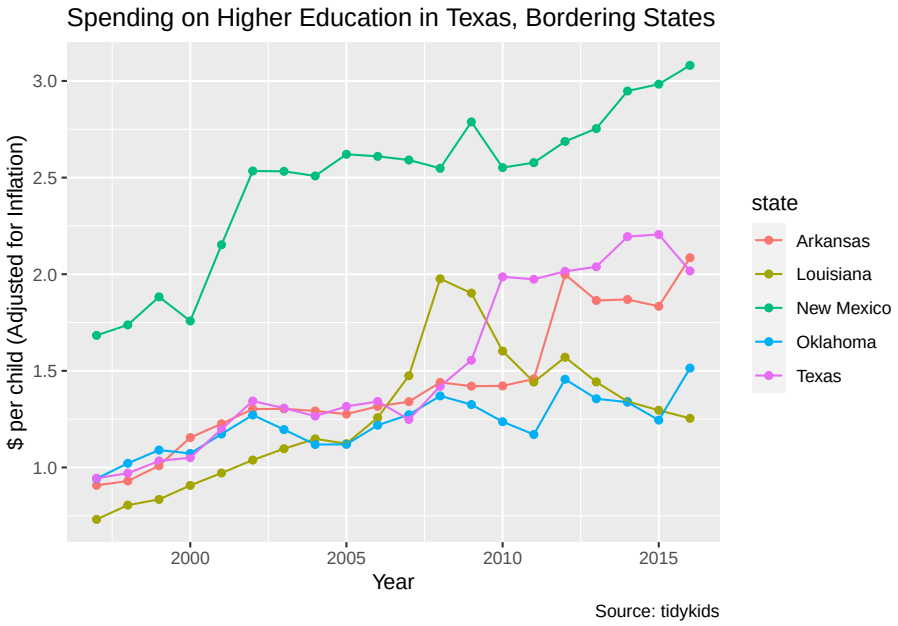
```
  labs(
    title = "Spending on Higher Education in Texas, Bordering States",
    x = "Year",
    y = "$ per child (Adjusted for Inflation)",
    caption = "Source: tidykids"
  )
```



Notice that R changes the color of the line, but not the point? This is because we only included the aesthetic in the `geom_line()` geom and not the `geom_point()` geom.

1. Edit your `geom_point()` to add `aes(color = state)`.

```
ggplot(five_hied, aes(x = year, y = inf_adj_perchild)) +
  geom_point(aes(color = state)) + # add the aes here
  geom_line(aes(color = state)) +
  labs(title = "Spending on Higher Education in Texas, Bordering States",
       x = "Year", y = "$ per child (Adjusted for Inflation)",
       caption = "Source: tidykids")
```



8.7 On your own: Line chart

I want you to make a line chart of preschool-to-high-school spending (the "PK12ed" value in the `variable` column) showing the inflation adjusted per-child spending (the `inf_adj_perchild` column) for the five states that border the Gulf of Mexico. This is very similar to the chart you just made, but with different values.

Some things to do/consider:

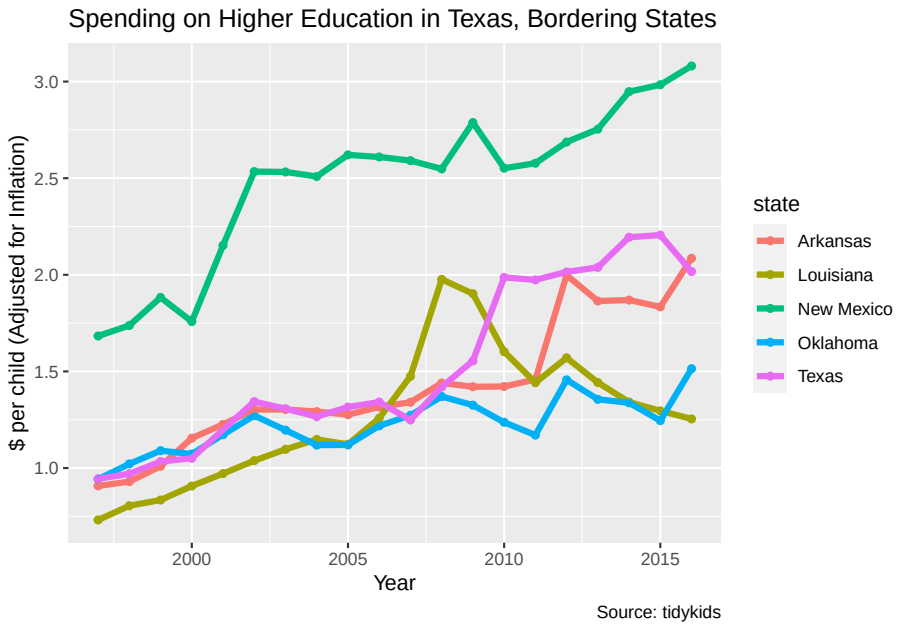
1. Do this in a new section and explain it.
2. You'll need to prepare the data just like we did above to get the right data points and the right states.
3. I really suggest you build both chunks (the data prep and the chart) one line at a time so you can see what each step adds.
4. Save the resulting plot into a new R object because we'll use it later.

8.8 Tour of some other adjustments

You don't have to add these examples below to your own notebook, but here are some examples of other things you can control.

8.8.1 Line width

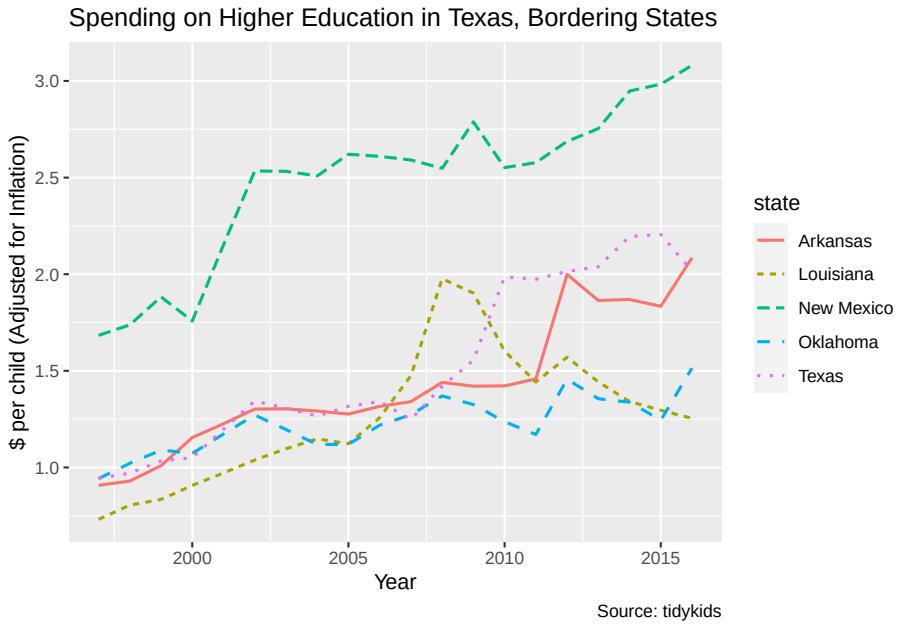
```
ggplot(five_hied, aes(x = year, y = inf_adj_perchild)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state), size = 1.5) + # added size here
  labs(title = "Spending on Higher Education in Texas, Bordering States",
       x = "Year", y = "$ per child (Adjusted for Inflation)",
       caption = "Source: tidykids")
```



8.8.2 Line type

This example removes the points and adds a `linetype = state` to the ggplot aesthetic. This gives each state a different type of line. We also set the color in the `geom_line()`

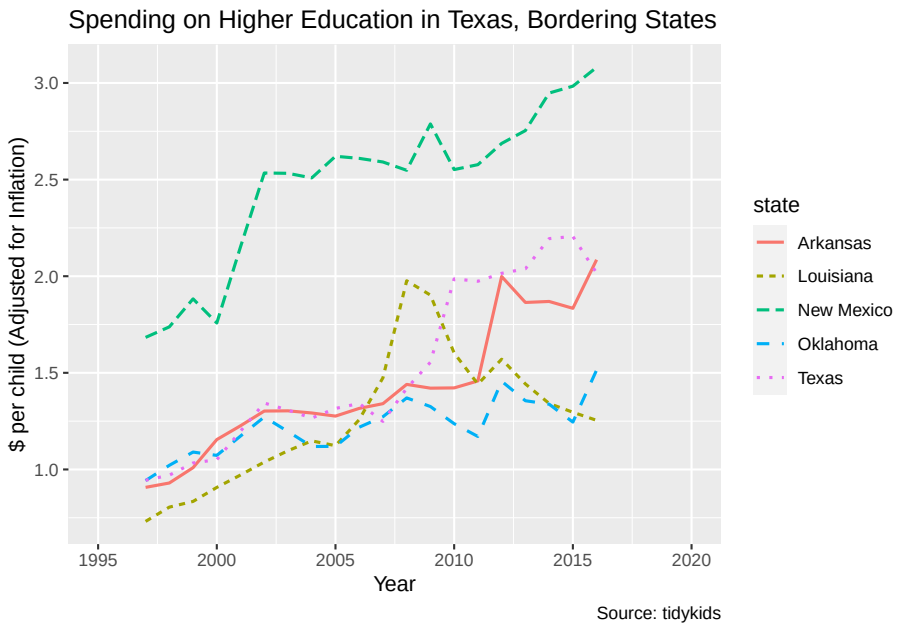
```
ggplot(five_hied, aes(x = year, y = inf_adj_perchild)) +
  geom_line(aes(color = state, linetype = state), size = .75) +
  labs(title = "Spending on Higher Education in Texas, Bordering States",
       x = "Year", y = "$ per child (Adjusted for Inflation)",
       caption = "Source: tidykids")
```



8.8.3 Adjust axis

`ggplot()` typically makes assumptions about scale. Sometimes, you may want to change it though (e.g., make them a little larger). There are a couple different ways to do this. The most straightforward may be `xlim()` and `ylim()`.

```
ggplot(five_hied, aes(x = year, y = inf_adj_perchild, linetype = state)) +
  geom_line(aes(color = state), size = .75) +
  xlim(1995, 2020) + # sets minimum and maximum values on axis
  labs(title = "Spending on Higher Education in Texas, Bordering States",
       x = "Year", y = "$ per child (Adjusted for Inflation)",
       caption = "Source: tidykids")
```



The function `xlim()` and `ylim()` are shortcuts for `scale_x_continuous()` and `scale_y_continuous()` which do more things.

8.9 Facets

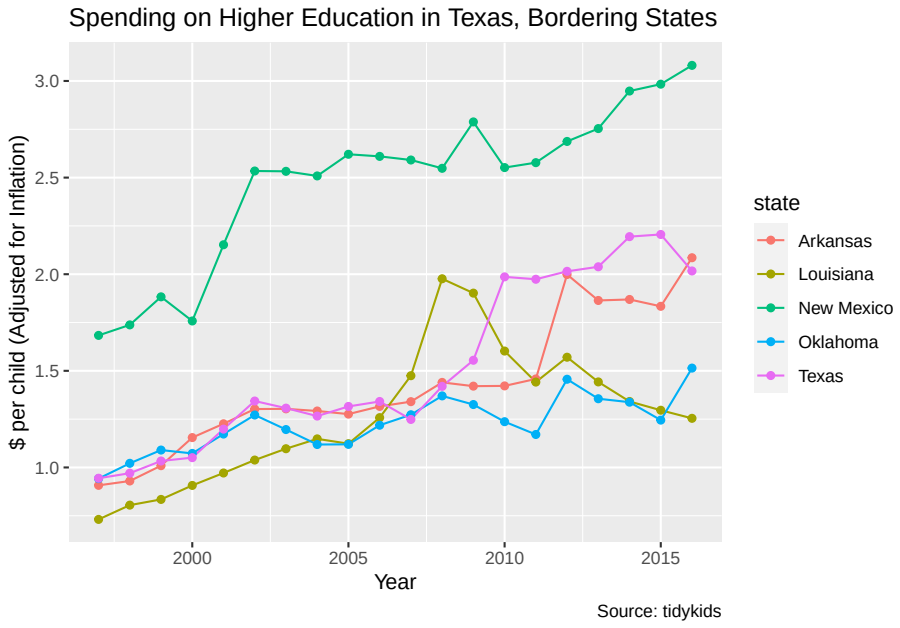
Facets are a way to make multiple graphs based on a variable in the data. There are two types, the `facet_wrap()` and the `facet_grid()`. There is a good explanation of these in R for Data Science.

We'll start by creating a base graph and then apply the facet.

1. Start a new section about facets
2. Add the code below to create your chart and view it.

```
five_plot <- ggplot(five_hied, aes(x = year,
                                   y = inf_adj_perchild)) +
  geom_line(aes(color = state)) +
  geom_point(aes(color = state)) +
  labs(title = "Spending on Higher Education in Texas, Bordering States",
       x = "Year", y = "$ per child (Adjusted for Inflation)",
       caption = "Source: tidykids")

five_plot
```

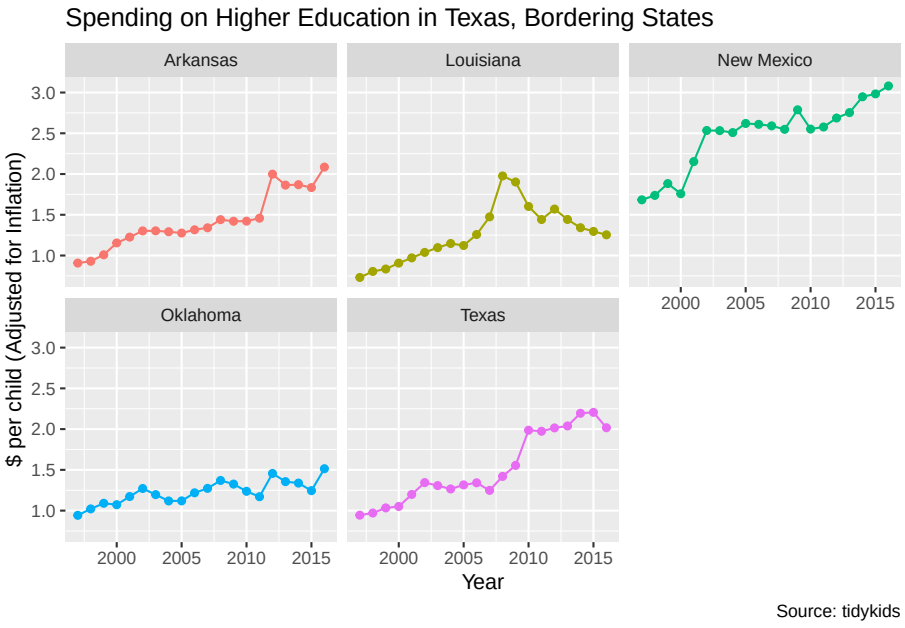


8.9.1 Facet wrap

The facet_wrap() splits your chart based on a single variable. You define which variable to split upon with ~ followed by the variable name.

1. Add a new chunk and create the facet wrap shown here.

```
five_plot +
  facet_wrap(~ state) +
  theme(legend.position = "none") # removes the legend. Try it without it!
```



A couple of notes about the above code:

- Note the comment in the code above where we used the `theme()` function to remove the legend.
- You can specify the number of rows or columns of the grouping by adjusting the facet_wrap() function: `facet_wrap(~ state, nrow = 2)` or `facet_wrap(~ state, ncol = 2)`. Try them!

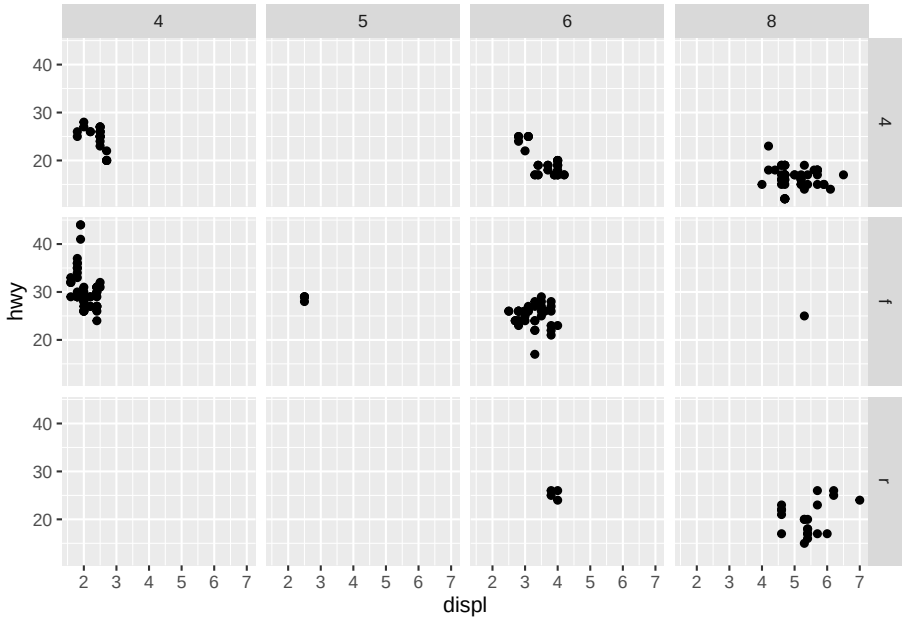
8.9.2 Facet grids

A `facet_grid()` allows you to plot on a combination of variables. We don't really have two numbers to compare in our higher education data so we'll show this with the `mpg` data we've used before.

1. Start a new section noting you'll try facet grid.
2. Add the chunk below and run it.

Explanations follow the chart.

```
ggplot(mpg) +
  geom_point(aes(x = displ, y = hwy)) + # add points to the chart
  facet_grid(drv ~ cyl) # splits into charts by drive train and cylinder
```



This chart is kinda hard to read, but let's try:

- Inside the mini charts, the best gas mileage is toward the top (from `hwy`) and the smaller engines are to the left (from `displ`.)
- The rows of charts are divided by drive train `drv`: four-wheel drive, front-wheel drive and rear-wheel drive.
- The columns of charts are divided by cylinders: like a 4-cylinder car vs 8-cylinder car.

This chart tells us that 4-cylinder, front-wheel drive cars with smaller engines get the best gas mileage. The blank charts mean that combination of values didn't exist in the data.

8.10 On your own: Facet wrap

1. Create a section about doing a facet wrap on your own.
2. Take the "On your own" plot that you made earlier (The school spending for Gulf states) and apply a `facet_wrap()` here. You were instructed to save the plot into an R object, so you should be able to use that.
3. Remove the legend since each mini chart is labeled.

8.11 Saving plots

To save plots as images, you can right-click plots that you make in RNotebooks. Or, you can use the export button in the Plot pane. Or (and this is a preferred strategy), you can save them using `ggsave()`. (Learn more here).

1. Use your Files pane to create a new folder called "images" so we can save our chart there.
2. Start a section on saving plots and add the following chunk.

```
ggsave("images/txplot.png", plot = tx_plot)
```

```
## Saving 6.5 x 4.5 in image
```

Using `ggsave` creates a higher-res image than other methods. It needs"

- The path and name of the image, in quotes
- the `plot =` variable to say which plot you are saving. (Your plot must already be saved into an R object for this method to work.)

8.12 Interactive plots

Want to make your plot interactive? You can use plotly's `ggplotly()` function to transform your graph into an interactive chart.

To use plotly, you'll want to install the plotly package, add the library, and then use the ggplotly() function:

1. In your R Console, run `install.packages("plotly")`. (You only have to do this once on your computer.)
2. Add a new section to note you are creating an interactive chart.
3. Add the code below and run it. Then play with the chart!

```
library(plotly)

tx_plot %>%
  ggplotly()
```

(We can't show the interactive version in this book.)

Now you have tool tips on your points when you hover over them.

The `ggplotly()` function is not perfect. Alternatively, you can use plotly's own syntax to build some quite interesting charts, but it's a whole new syntax to master.

8.13 What we learned

There is so much more to ggplot2 than what we've shown here, but these are the basics that should get you through the class. At the top of this chapter are a list of other resources to learn more.

Chapter 9

Tidy data

Data "shape" can be important when you are trying to work with and visualize data. In this chapter we'll discuss "tidy" data and how this style of organization helps us.

Slides by Hadley Wickham are used with permission from the author.

9.1 Goals for this section

- Explore what it means to have "tidy" data.
- Learn about and use `pivot_longer()`, `pivot_wider()` to make our data tidy.
- Use Skittles to explore shaping data.

9.2 The questions we'll answer

- Are candy colors evenly distributed within a package of Skittles? (The mean of candies by color over all packages)
- Plot a column chart showing the average number of colored candies among all packages using ggplot
- Plot the same data using Datawrapper.
- Bonus 1: Who got the most candies in their bag?
- Bonus 2: What is the average number of candy in a bag?

9.3 What is tidy data

"Tidy" data is well formatted so each variable is in a column, each observation is in a row and each value is a cell. Our first step in working with any data is to make sure we are "tidy."

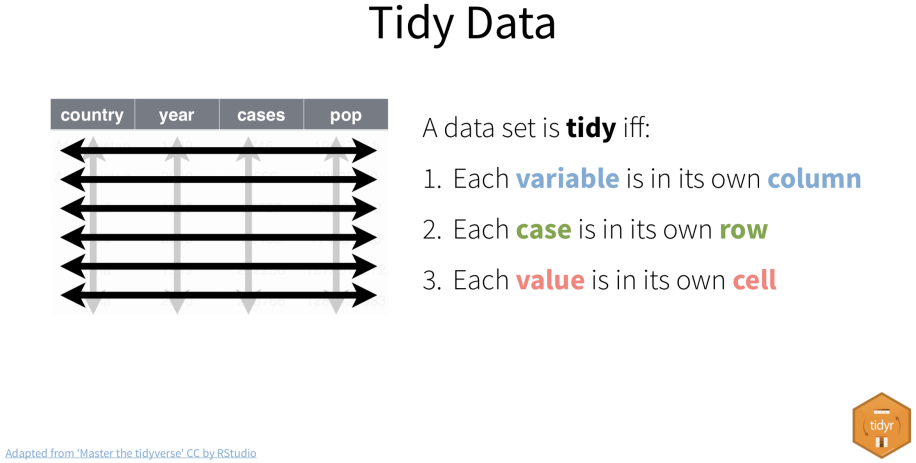


Figure 9.1: Tidy data definition

It's easiest to see the difference through examples. The data frame below is of tuberculosis reports from the World Health Organization.

- Each row is a set of observations (or case) from a single country for a single year.
- Each column describes a unique variable. The year, the number of cases and the population of the country at that time.

Table2 below isn't tidy. The **count** column contains two different type of values.

When our data is tidy, it is easy to manipulate. We can use functions like `mutate()` to calculate new values for each case.

9.4 Tidyr package

When our data is tidy, we can use the tidyr package to reshape the layout of our data to suit our needs. It gets loaded with `library(tidyverse)`.

In the figure below, the table on the left is "wide." There are are multiple year columns describing the same variable. It might be useful if we want to calculate

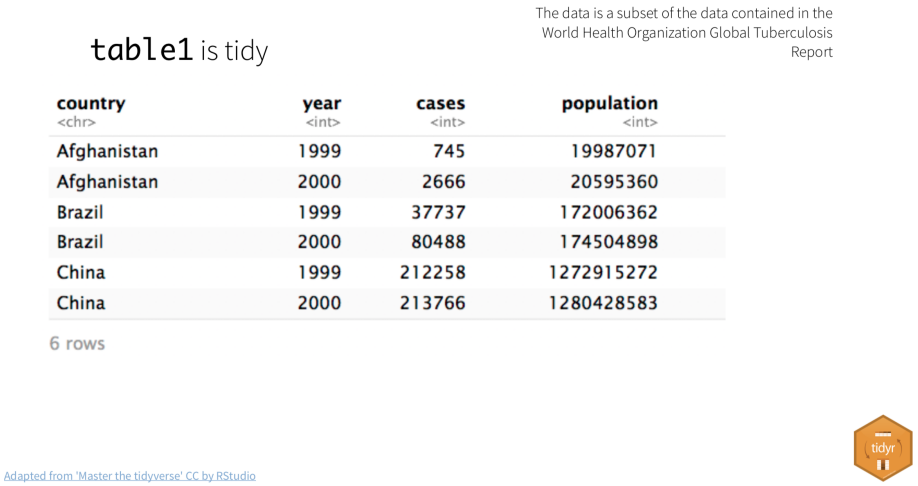


Figure 9.2: A tidy table

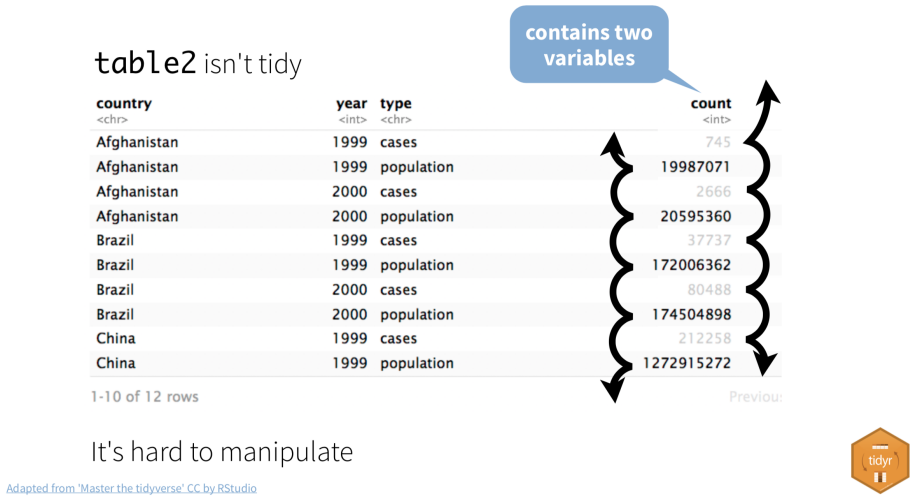


Figure 9.3: An untidy table

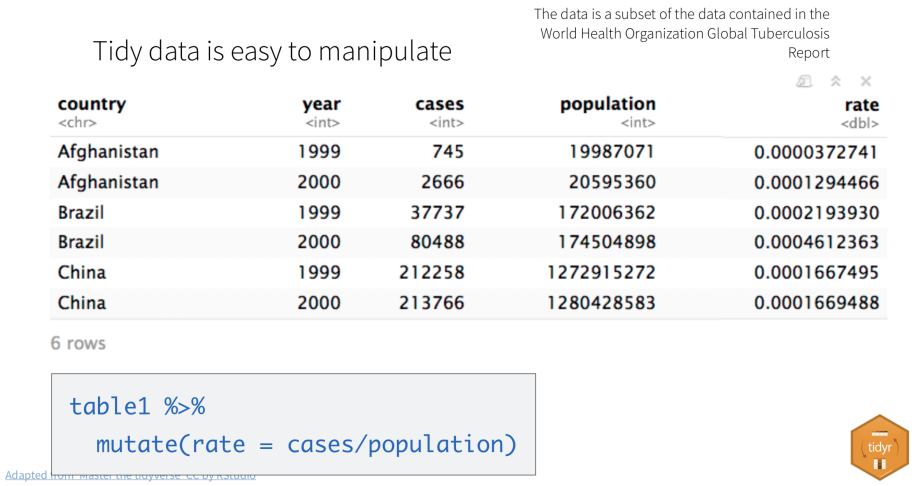


Figure 9.4: Manipulate a tidy table

the difference of the values for two different years. It's less useful if we want plot on a graphic because we don't have columns to map as X and Y values.

The table on the right is "long," in that each column describes a single variable. It is this shape we need when we want to plot values on a chart. We can then set our "Year" column as an X axis, our "n" column on our Y axis, and group by the "Country."

9.5 The tidyr verbs

The two functions we'll use most to reshape are data are:

- pivot_longer() "lengthens" data, increasing the number of rows and decreasing the number of columns.
- pivot_wider() "widens" data, increasing the number of columns and decreasing the number of rows.

Again, the best way to learn this is to present the problem and solve it with explanation.

9.6 Prepare our Skittles project

Start a new project to explore this subject.

Country	2011	2012	2013
FR	7000	6900	7000
DE	5800	6000	6200
US	15000	14000	13000

Country	Year	n
FR	2011	7000
DE	2011	5800
US	2011	15000
FR	2012	6900
DE	2012	6000
US	2012	14000
FR	2013	7000
DE	2013	6200
US	2013	13000

Figure 9.5: Wide vs long

1. Create a new project and call it: `yourname-skittles`
2. No need to create folders. We'll just load data directly into the notebook.
3. Start a new RNotebook and edit the headline
4. Create your setup block and load the libraries below.

```
library(tidyverse)
library(janitor)
library(lubridate)
```

9.6.1 Get the data

We'll just load this data directly from Google Sheets into this notebook.

1. Add a section that you are importing data.
2. Add this import chunk.

```
data <- read_csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vTxm9NxK67thlGjYOQBo_0JRvx2d137
```

```
## Rows: 124 Columns: 7

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Timestamp, Name
## dbl (5): Red, Green, Orange, Yellow, Purple
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# peek at the data
data %>% glimpse()
```

```
## Rows: 124
## Columns: 7
## $ timestamp <chr> "7/27/2020 18:21:19", "8/1/2020 16:04:56", "7/30/2020 14:56:~
## $ name      <chr> "Alora Jones", "Alyssa Hiarker", "Annie Patton", "Christian ~
## $ red       <dbl> 12, 13, 12, 9, 7, 10, 12, 11, 7, 18, 13, 11, 10, 14, 8, 8, 1~
## $ green     <dbl> 11, 15, 12, 10, 12, 14, 15, 5, 10, 9, 11, 13, 7, 12, 15, 10,~
## $ orange    <dbl> 12, 10, 8, 17, 13, 9, 12, 17, 10, 13, 11, 7, 12, 10, 20, 13,~
## $ yellow    <dbl> 9, 9, 10, 6, 11, 11, 10, 14, 21, 7, 11, 7, 15, 9, 4, 14, 10,~
## $ purple    <dbl> 15, 15, 18, 18, 17, 16, 12, 13, 14, 13, 12, 21, 16, 14, 10, ~
```

We cleaned the name on import. The `timestamp` is not a real date, so we need to fix that.

9.6.2 Fix the date

We're going to convert the `timestamp` and then turn it into a regular date.

1. Create a section and note you are fixing dates.
2. Add this chunk and run it. I'll explain it below.

```
skittles <- data %>%
  mutate(
    date_entered = mdy_hms(timestamp) %>% date()
  ) %>%
  select(-timestamp)

skittles %>% glimpse()
```

```
## Rows: 124
## Columns: 7
## $ name         <chr> "Alora Jones", "Alyssa Hiarker", "Annie Patton", "Christi~
## $ red          <dbl> 12, 13, 12, 9, 7, 10, 12, 11, 7, 18, 13, 11, 10, 14, 8, 8~
## $ green        <dbl> 11, 15, 12, 10, 12, 14, 15, 5, 10, 9, 11, 13, 7, 12, 15, ~
## $ orange       <dbl> 12, 10, 8, 17, 13, 9, 12, 17, 10, 13, 11, 7, 12, 10, 20, ~
## $ yellow       <dbl> 9, 9, 10, 6, 11, 11, 10, 14, 21, 7, 11, 7, 15, 9, 4, 14, ~
## $ purple       <dbl> 15, 15, 18, 18, 17, 16, 12, 13, 14, 13, 12, 21, 16, 14, 1~
## $ date_entered <date> 2020-07-27, 2020-08-01, 2020-07-30, 2020-06-23, 2020-07-~
```

Let's talk just a minute about what we've done here:

- We name our new tibble.
- We are filling that tibble starting with our imported data called `data`.
- We use mutate to create a new column `date_entered`, then fill it by first converting the text to an official timestamp datatype (which requires the lubridate function `mdy_hms()`), and then we extract just the date of that with `date()`.
- We then use `select()` to remove the old timestamp column.

9.6.3 Peek at the wide table

Let's look closer at this:

```
skittles %>% head()
```

```
## # A tibble: 6 x 7
##   name                red green orange yellow purple date_entered
##   <chr>             <dbl> <dbl>  <dbl>  <dbl>  <dbl> <date>
## 1 Alora Jones          12    11     12      9     15 2020-07-27
## 2 Alyssa Hiarker       13    15     10      9     15 2020-08-01
## 3 Annie Patton         12    12      8     10     18 2020-07-30
## 4 Christian McDonald    9    10     17      6     18 2020-06-23
## 5 Claudia Ng            7    12     13     11     17 2020-07-30
## 6 Cristina Pop         10    14      9     11     16 2020-07-22
```

This is not the worst example of data. It could be useful to create a "total" column, but there are better ways to do this with **long** data.

9.7 Pivot longer

What we want here is five rows for Alora Jones, with a column for "color" and a column for "candies."

The `pivot_longer()` function needs several arguments:

- Which columns do you want to pivot? For us, these are the color columns.
- What do you want to name the new column to describe the column names? For us we want to name this "color" since that's what those columns described.
- What do you want to name the new column to describe the values that were in the cells? For us we want to call this "candies" since these are the number of candies in each bag.

There are a number of ways we can describe which columns to pivot . . . anything in tidy-select works. You can see a bunch of examples here.

We are using a range, naming the first "red" and the last column "purple" with `:` in between. This only works because those columns are all together. We could also use `cols = !c(name, date_entered)` to say everything but those two columns.

1. Add a note that you are pivoting the data
2. Add the chunk below and run it

```
skittles_long <- skittles %>%
  pivot_longer(
    cols = red:purple, # sets which columns to pivot based on their names
    names_to = "color", # sets column name for color
    values_to = "candies" # sets column name for candies
  )

skittles_long %>% head()
```

```
## # A tibble: 6 x 4
##   name           date_entered color  candies
##   <chr>          <date>       <chr>    <dbl>
## 1 Alora Jones    2020-07-27   red         12
## 2 Alora Jones    2020-07-27   green       11
## 3 Alora Jones    2020-07-27   orange      12
## 4 Alora Jones    2020-07-27   yellow       9
## 5 Alora Jones    2020-07-27   purple      15
## 6 Alyssa Hiarker 2020-08-01   red         13
```

9.7.1 Average candies per color

To get the average number of candies per each color, we can use our `skittles_long` data and `group_by` color (which will consider all the **red** rows together, etc.) and use `summarize()` to get the mean.

This is something you should be able to do on your own, as it is very similar to the `sum()`s we did with military surplus, but you use `mean()` instead.

Save the resulting summary table into a new tibble called `skittles_avg`.

Try it on your own

```
skittles_avg <- skittles_long %>%
  group_by(color) %>%
  summarize(avg_candies = mean(candies))
```

```
skittles_avg
```

```
## # A tibble: 5 x 2
##   color  avg_candies
##   <chr>        <dbl>
## 1 green         11.2
## 2 orange        12.1
## 3 purple        12.0
## 4 red           11.6
## 5 yellow        11.5
```

9.7.2 Round the averages

Let's modify this summary to round the averages to tenths so they will plot nicely on our chart.'

The `round()` function needs the column to change, and then the number of digits past the decimal to include.

1. Edit your summary to include the mutate below.

```
skittles_avg <- skittles_long %>%
  group_by(color) %>%
  summarize(avg_candies = mean(candies)) %>%
  mutate(
    avg_candies = round(avg_candies, 1)
  )

skittles_avg
```

```
## # A tibble: 5 x 2
##   color  avg_candies
##   <chr>        <dbl>
## 1 green         11.2
## 2 orange        12.1
## 3 purple        12
## 4 red           11.6
## 5 yellow        11.5
```

BONUS POINT OPPORTUNITY: Using a similar method to rounding above, you can also capitalize the names of the colors. You don't *have* to do this, but I'll give you bonus points if you do:

- In your mutate, add a rule that updates `color` column using `str_to_title(color)`.

You can read more about converting the case of a string here. It's part of the stringr package, which is loaded with tidyverse.

9.7.3 On your own: Plot the averages

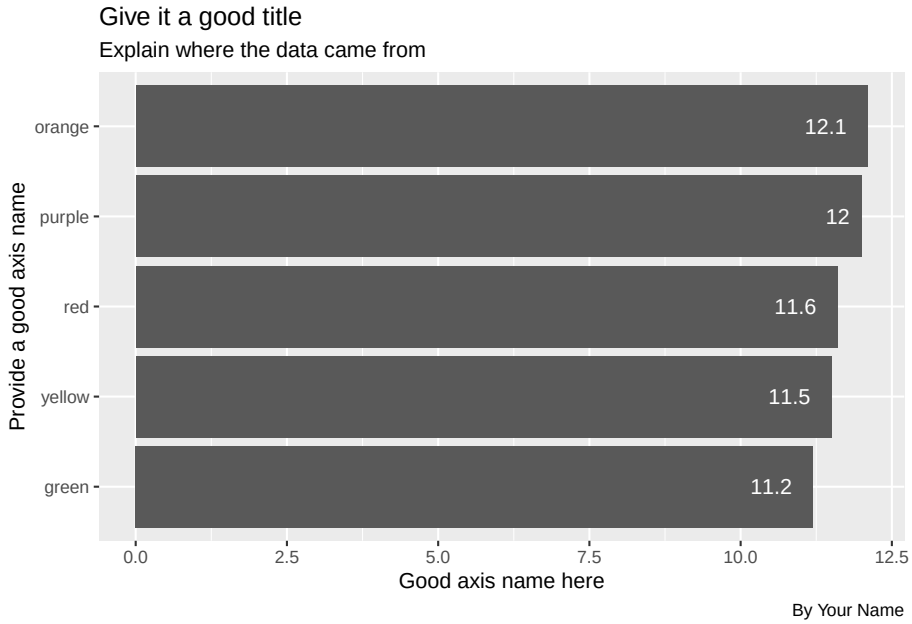
Now I want you to use ggplot to create a bar chart that shows the average number of candies in a bag. This is very similar to your plots of Disney Princesses and ice cream in Chapter 6.

1. Build a bar chart of averge color using ggplot.

Some things to consider:

- I want the bars to be ordered by the highest average on top.
- I want a good title, subtitle and byline, along with good axis names.
- Include the values on the bars
- Change the theme to something other than the default

Here is what it should look like, but with good text, etc. The numbers shownn here may vary depending on future updates to the data:



9.8 Using Datawrapper

There are some other great charting tools that journalists use. My favorite is Datawrapper and is free for the level you need it.

Datawrapper is so easy I don't even have to teach you how to use it. They have excellent tutorials.

What you do need is the data to plot, but you've already "shaped" it the way you need it. Your `skittles_avg` tibble is what you need.

Here are the steps I want you to follow:

9.8.1 Review how to make a bar chart

1. In a web browser, go to the Datawrapper Academy
2. Click on **Bar charts**
3. Choose **How to create a bar chart**

The first thing to note there is they show you what they expect the data to look like. Your `skittles_avg` tibble is just like this, but with Color and Candies.

You'll use these directions to create your charts so you might keep this open in its own tab.

9.8.2 Start a chart

1. In a new browser tab, go to datawrapper.de and click the big **Start creating** button.
2. Use the **Login/Sign Up** button along the top to create an account or log in if you have one.
3. The first screen you have is where you can **Upload data** or paste it into the window. We are going to paste the data, but we have to do some stuff in R to get it.

9.8.3 Get your candies data

We need to install a package called clipr.

1. In your R project in the R Console install clipr: `install.packages("clipr")`.
2. Start a section that says you are going to get data for Datawrapper.
3. Create a chunk with the following and run it.

```
library(clipr)
```

```
## Welcome to clipr. See ?write_clip for advisories on writing to the clipboard in R.
```

```
skittles_avg %>% write_clip(allow_non_interactive = TRUE)
```

You don't see anything happen, but all the data in `skittles_long` has been added to your clipboard. You have to have the `allow_non_interactive = TRUE` part to allow your page to knit.

9.8.4 Build the datawrapper graphic

1. Return to the browser where you are making the chart, but your cursor into the "Paste your copied data here . . . " window and paste.
2. Click **Proceed**.

You can now follow the Datawrapper Academy directions to finish your chart.

When you get to the Publish & Embed window, I want you to add that link to your R Notebook so I can find it for grading.

9.9 Bonus questions

More opportunities for bonus points on this assignment. These aren't plots, just data wrangling.

9.9.1 Most/least candies

Answer me this: Who got the most candies in their bag? Who got the least?

I want a well-structured section (headline, text) with two chunks, one for most and one for least.

9.9.2 Average total candies in a bag

Answer me this: What is the average number of candy in a bag?

Again, well-structured section and include the code.

Hint: You need a total number of candies per person before you can get an average.

9.10 Turn in your work

1. Make sure your notebook runs start-to-finish.
2. Knit the notebook
3. Stuff your project and turn it into the Skittles assignment in Canvas.

9.11 What we learned

- We learned what "tidy data" means and why it is important. It is the best shape for data wrangling and plotting.
- We learned about `pivot_longer()` and `pivot_wider()` and we used `pivot_longer()` on our Skittles data.
- Along the way we practiced a little lubridate conversion with `mdy_hms()` and extracted a date with `date()`.
- We also used `round()` to round off some numbers, and you might have used `str_to_title()` to change the case of the color values.

Chapter 10

Plotting for answers

For this chapter, we will use some data from the City of Austin data portal on animal intakes to the Austin Animal Center. You'll use that portal to download the data, prepare it for R, then answer some questions and make plots to show the answers.

Along the way we'll learn some stuff.

10.1 Goals of this lesson

- We'll use some string and date function to clean and parse some dates.
- We'll use `count()` to make some summaries. All of our "answers" are counting operations, so we'll practice using that shortcut instead of "GSA."
- We'll learn how to add commas to axis names in ggplot with the scales package.
- We'll use `recode()` to update some values in our data.

10.2 Questions we will answer

We'll tackle these after we clean our data, but so you know where we are going:

- Are animal intakes increasing or decreasing each year since 2016? Plot a column chart of intakes by year. Don't include 2021 since it is not a full year.
- Are there seasonal monthly trends in overall animal intakes? Plot intakes by year/month from 2016 to current, including 2021. (One long line or column chart.)

- Plot that same monthly trend, except with month as the x axis, with a new line for each year.
- Do certain types of animals drive seasonal trends? Use five full years of data (not 2021) to summarize intakes by animal type and month. Plot with month on the x axis with a line for each animal type.

10.3 Create your project

We'll be starting a new R project with our typical folder structure.

1. Create a new project. Call it `yourname-aac`.
2. Create your `data-raw` and `data-processed` folders.
3. Create a new R Notebook, title it "AAC Import/clean" and name the file `01-import.Rmd`
4. Add your setup section with the following libraries:

```
library(tidyverse)
library(janitor)
library(lubridate)
```

As always, create good Markdown sections, descriptions, add notes and name your R chunks so you have good bookmarks to work with.

10.4 Download the data

This time you have to go get your data online and then put the file into your `data-raw` folder yourself.

1. Go to https://data.austintexas.gov/
2. Search for "animal intakes." Find the link "Austin Animal Center Intakes" and click on it. (It may not be the first return, so be careful to get the right one.)

This page tells you about the data at the top. Note that records go back to October 2013. Further down the page there is a list of the "Columns in this Dataset" that kinda describes them, and you can see an example of the data at the bottom. Review those.

1. At the top-right of the page is an **Export** button. Click on that and choose **CSV**. This will save the file called `Austin_Animal_Center_Intakes.csv` to your computer's Downloads folder.

2. Find your Downloads folder and the file on your computer, and them move the csv file into your `data-raw` folder in your project folder.

Note that when you source this data in stories or charts, it comes from the Austin Animal Center, not the city's data portal. The portal is just the delivery method.

10.5 Import your data

Go back into RStudio in your `01-import.Rmd` file and import the data. You've done this many times now and should be able to do it on your own.

1. Don't forget to create a section with a headline, etc.
2. Work one line at a time. Use `read_csv()` to find your data and load it onto the screen.
3. Once that is using, add a `%>%` and use the `clean_names()` function to fix the column names.
4. Once that is all good, edit the chunk to save the imported data into a new tibble called `raw_data`.

You shouldn't need this.

```
raw_data <- read_csv("data-raw/Austin_Animal_Center_Intakes.csv") %>% clean_names()

# peek at the data
raw_data
```

```
## # A tibble: 132,214 x 12
##    animal_id name    date_time    month_year    found_location       intake_type
##    <chr>     <chr>   <chr>        <chr>         <chr>                <chr>
##  1 A786884   *Brock  01/03/2019 ~ 01/03/2019 0~ 2501 Magin Meadow D~ Stray
##  2 A706918   Belle   07/05/2015 ~ 07/05/2015 1~ 9409 Bluegrass Dr i~ Stray
##  3 A724273   Runster 04/14/2016 ~ 04/14/2016 0~ 2818 Palomino Trail~ Stray
##  4 A665644   <NA>    10/21/2013 ~ 10/21/2013 0~ Austin (TX)          Stray
##  5 A682524   Rio     06/29/2014 ~ 06/29/2014 1~ 800 Grove Blvd in A~ Stray
##  6 A743852   Odin    02/18/2017 ~ 02/18/2017 1~ Austin (TX)          Owner Surr~
##  7 A635072   Beowulf 04/16/2019 ~ 04/16/2019 0~ 415 East Mary Stree~ Public Ass~
##  8 A708452   Mumble  07/30/2015 ~ 07/30/2015 0~ Austin (TX)          Public Ass~
##  9 A818975   <NA>    06/18/2020 ~ 06/18/2020 0~ Braker Lane And Met~ Stray
## 10 A774147   <NA>    06/11/2018 ~ 06/11/2018 0~ 6600 Elm Creek in A~ Stray
## # ... with 132,204 more rows, and 6 more variables: intake_condition <chr>,
## #   animal_type <chr>, sex_upon_intake <chr>, age_upon_intake <chr>,
## #   breed <chr>, color <chr>
```

10.6 Fix the dates

Take a look at the `date_time` and `month_year` columns. They are both **timestamps** that include both the date and time. They imported as a **character** datatype `<chr>` and are in this format:

```
01/03/2019 04:19:00 PM
```

We won't be using the time for this exercise, so all we really need is the date. Lubridate doesn't have a conversion for this exact format (at least that I could find.) That's ok, we'll use some stringr functions to whip this into shape.

We can use a function called `str_sub()` to pluck out the date from this string. We'll create a new column using `mutate()` to do this. You've used mutate before.

`str_sub()` allows you to pluck any number of characters out of a string. We want the first 10 characters: "01/03/2019."

It takes three arguments:

- The column you are looking at? For us, this is the `date_time` column.
- What position do you want to start at? For us, we start at "1," the first character.
- How many characters do you want? For us, we want "10."

We use this inside a `mutate()` function to create a new column with the results of the `str_sub()` function.

1. Create a new section and note we are fixing the date.
2. Create a chunk, call your `raw_data` and pipe that into a `mutate()` function.
3. Inside your mutate, name your new column `intake_date`.
4. Set `intake_date` to `=` to `str_sub(date_time, 1, 10)`.

Try it, and then check the last column of the data that comes back to make sure you actually have a 10-character string like "01/03/2019."

Part of the answer

```
raw_data %>%
  mutate(
    intake_date = str_sub(date_time, 1, 10)
  )
```

```
## # A tibble: 132,214 x 13
##    animal_id name    date_time    month_year    found_location       intake_type
##    <chr>     <chr>   <chr>        <chr>         <chr>                <chr>
##  1 A786884   *Brock  01/03/2019 ~ 01/03/2019 0~ 2501 Magin Meadow D~ Stray
##  2 A706918   Belle   07/05/2015 ~ 07/05/2015 1~ 9409 Bluegrass Dr i~ Stray
##  3 A724273   Runster 04/14/2016 ~ 04/14/2016 0~ 2818 Palomino Trail~ Stray
##  4 A665644   <NA>    10/21/2013 ~ 10/21/2013 0~ Austin (TX)          Stray
##  5 A682524   Rio     06/29/2014 ~ 06/29/2014 1~ 800 Grove Blvd in A~ Stray
##  6 A743852   Odin    02/18/2017 ~ 02/18/2017 1~ Austin (TX)          Owner Surr~
##  7 A635072   Beowulf 04/16/2019 ~ 04/16/2019 0~ 415 East Mary Stree~ Public Ass~
##  8 A708452   Mumble  07/30/2015 ~ 07/30/2015 0~ Austin (TX)          Public Ass~
##  9 A818975   <NA>    06/18/2020 ~ 06/18/2020 0~ Braker Lane And Met~ Stray
## 10 A774147   <NA>    06/11/2018 ~ 06/11/2018 0~ 6600 Elm Creek in A~ Stray
## # ... with 132,204 more rows, and 7 more variables: intake_condition <chr>,
## #   animal_type <chr>, sex_upon_intake <chr>, age_upon_intake <chr>,
## #   breed <chr>, color <chr>, intake_date <chr>
```

10.6.1 Edit to convert to a real date

If you did the above correctly, you should have a column called `intake_date` as the last column, but it isn't actually a **date** yet, it is just characters that look like a date. We'll fix that now.

1. Edit your date-fix chunk to add another rule INSIDE your mutate.
2. The new column will still be `intake_date =` but now you'll set that to `mdy(intake_date)`
3. Run the chunk and make sure that your same last column `intake_date` says `<date>` right below the name. The order should now be `2019-01-03`.
4. Now that this all works, assign all this using `<-` into a tibble called `date_fix`.
5. Add a `glimpse()` of the date_fix tibble in the same chunk so you can eyeball the results.

this was simlar to converting the date in billboard

```
date_fix <- raw_data %>%
  mutate(
    intake_date = str_sub(date_time, 1, 10),
    intake_date = mdy(intake_date)
  )

date_fix %>% glimpse()
```

```
## Rows: 132,214
```

```
## Columns: 13
## $ animal_id        <chr> "A786884", "A706918", "A724273", "A665644", "A682524"~
## $ name             <chr> "*Brock", "Belle", "Runster", NA, "Rio", "Odin", "Beo~
## $ date_time        <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ month_year       <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ found_location   <chr> "2501 Magin Meadow Dr in Austin (TX)", "9409 Bluegras~
## $ intake_type      <chr> "Stray", "Stray", "Stray", "Stray", "Stray", "Owner S~
## $ intake_condition <chr> "Normal", "Normal", "Normal", "Sick", "Normal", "Norm~
## $ animal_type      <chr> "Dog", "Dog", "Dog", "Cat", "Dog", "Dog", "Dog", "Dog~
## $ sex_upon_intake  <chr> "Neutered Male", "Spayed Female", "Intact Male", "Int~
## $ age_upon_intake  <chr> "2 years", "8 years", "11 months", "4 weeks", "4 year~
## $ breed            <chr> "Beagle Mix", "English Springer Spaniel", "Basenji Mi~
## $ color            <chr> "Tricolor", "White/Liver", "Sable/White", "Calico", "~
## $ intake_date      <date> 2019-01-03, 2015-07-05, 2016-04-14, 2013-10-21, 2014~
```

Now that you can see the `date_time` and `intake_date` columns at once, check to make sure they converted correctly and you don't have any problems. Doublecheck the datatype for `intake_date`, which should be `<date>`.

10.7 Parse the date into helpful variables

Now that we have a good date to work with, we can use other lubridate functions to create some versions of the date that will help us down the road when we do summaries and plots.

> TBH, just diving into the data at this point you might not *know* you need these date parts yet until you try to create summaries and plots. If you find later that you need helpful columns like this, you can always come back to your import notebook, create and re-run it to get updated data. In the interest of time I'm front-loading the need based on experience.

We are going to create three variations of the date to help us later:

- A `yr` column with just the year, like `2019`.
- A `mo` column with the month, but using the name, like `Jan`.
- A `yrmo` column like `2019-01`

We'll do this in the same `mutate()` function, but we'll use different methods to do each one, which is a useful learning experience. We'll also use `select()` to reorder our columns to put these all at front of the tibble so we can see them.

We'll work through this out in the open so I can explain as we go along.

10.7.1 Extract the year

We can use `year()` from lubridate to pluck the YYYY value from `intake_date`. We'll use this to build our mutate.

1. Create a new section and note we are creating helpful date parts.
2. Add the following chunk so we can get started.

```
date_parts <- date_fix %>%
  mutate(
    yr = year(intake_date) # creates yr and fills it with YYYY
  )

# peek
date_parts %>% glimpse()
```

```
## Rows: 132,214
## Columns: 14
## $ animal_id        <chr> "A786884", "A706918", "A724273", "A665644", "A682524"~
## $ name             <chr> "*Brock", "Belle", "Runster", NA, "Rio", "Odin", "Beo~
## $ date_time        <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ month_year       <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ found_location   <chr> "2501 Magin Meadow Dr in Austin (TX)", "9409 Bluegras~
## $ intake_type      <chr> "Stray", "Stray", "Stray", "Stray", "Stray", "Owner S~
## $ intake_condition <chr> "Normal", "Normal", "Normal", "Sick", "Normal", "Norm~
## $ animal_type      <chr> "Dog", "Dog", "Dog", "Cat", "Dog", "Dog", "Dog", "Dog~
## $ sex_upon_intake  <chr> "Neutered Male", "Spayed Female", "Intact Male", "Int~
## $ age_upon_intake  <chr> "2 years", "8 years", "11 months", "4 weeks", "4 year~
## $ breed            <chr> "Beagle Mix", "English Springer Spaniel", "Basenji Mi~
## $ color            <chr> "Tricolor", "White/Liver", "Sable/White", "Calico", "~
## $ intake_date      <date> 2019-01-03, 2015-07-05, 2016-04-14, 2013-10-21, 2014~
## $ yr               <dbl> 2019, 2015, 2016, 2013, 2014, 2017, 2019, 2015, 2020,~
```

Look how I set up this chunk to work with it. I know that I'm going to be adding columns and checking values and it is a pain click to the end of the tibble each time to see the results. So what I've done is set this up to go into a new tibble called `date_parts` and then I glimpse that at the end so I can peek at the results. This allows me to look at the first couple of values in the glimpse to make sure I've done the work right. **I'll still be working one line at a time** as I edit the chunk further, but at least I can *see* what I'm doing.

Now, note we have a new column `yr` at that starts with "2019," which matches what is in `intake_date` (and even `date_time`). This is good.

Can you see how our mutate created the new `yr` column?

- We name the new column `yr`
- We fill that column with `year(intake_date)`, which plucks the year from that column.

10.7.2 Extract the month name

We'll edit the **same chunk** to do a similar action to get the name of the month in a new column. You'll see in a minute how we can choose to get the *name* of the month instead of the number.

1. Edit your chunk to add a new line to the mutate function. Don't forget the comma after the existing rule.
2. Add the new line as indicated below, then run it to see the results.

```
date_parts <- date_fix %>%
  mutate(
    yr = year(intake_date), # don't forget the comma
    mo = month(intake_date) # the new mutate rule to get month
  )

# peek
date_parts %>% glimpse()
```

```
## Rows: 132,214
## Columns: 15
## $ animal_id        <chr> "A786884", "A706918", "A724273", "A665644", "A682524"~
## $ name             <chr> "*Brock", "Belle", "Runster", NA, "Rio", "Odin", "Beo~
## $ date_time        <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ month_year       <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ found_location   <chr> "2501 Magin Meadow Dr in Austin (TX)", "9409 Bluegras~
## $ intake_type      <chr> "Stray", "Stray", "Stray", "Stray", "Stray", "Owner S~
## $ intake_condition <chr> "Normal", "Normal", "Normal", "Sick", "Normal", "Norm~
## $ animal_type      <chr> "Dog", "Dog", "Dog", "Cat", "Dog", "Dog", "Dog", "Dog~
## $ sex_upon_intake  <chr> "Neutered Male", "Spayed Female", "Intact Male", "Int~
## $ age_upon_intake  <chr> "2 years", "8 years", "11 months", "4 weeks", "4 year~
## $ breed            <chr> "Beagle Mix", "English Springer Spaniel", "Basenji Mi~
## $ color            <chr> "Tricolor", "White/Liver", "Sable/White", "Calico", "~
## $ intake_date      <date> 2019-01-03, 2015-07-05, 2016-04-14, 2013-10-21, 2014~
## $ yr               <dbl> 2019, 2015, 2016, 2013, 2014, 2017, 2019, 2015, 2020,~
## $ mo               <dbl> 1, 7, 4, 10, 6, 2, 4, 7, 6, 6, 8, 10, 7, 2, 3, 2, 11,~
```

What we get in return here is the *number* of the month: A "1" for January; a "7" for July, etc. What we really want is the *names* of the month to help us with plotting later.

1. Edit the mutate to add `, label = TRUE` within the `month()`.

```
date_parts <- date_fix %>%
  mutate(
    yr = year(intake_date),
    mo = month(intake_date, label = TRUE) # add the label argument
  )

# peek
date_parts %>% glimpse()
```

```
## Rows: 132,214
## Columns: 15
## $ animal_id        <chr> "A786884", "A706918", "A724273", "A665644", "A682524"~
## $ name             <chr> "*Brock", "Belle", "Runster", NA, "Rio", "Odin", "Beo~
## $ date_time        <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ month_year       <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ found_location   <chr> "2501 Magin Meadow Dr in Austin (TX)", "9409 Bluegras~
## $ intake_type      <chr> "Stray", "Stray", "Stray", "Stray", "Stray", "Owner S~
## $ intake_condition <chr> "Normal", "Normal", "Normal", "Sick", "Normal", "Norm~
## $ animal_type      <chr> "Dog", "Dog", "Dog", "Cat", "Dog", "Dog", "Dog", "Dog~
## $ sex_upon_intake  <chr> "Neutered Male", "Spayed Female", "Intact Male", "Int~
## $ age_upon_intake  <chr> "2 years", "8 years", "11 months", "4 weeks", "4 year~
## $ breed            <chr> "Beagle Mix", "English Springer Spaniel", "Basenji Mi~
## $ color            <chr> "Tricolor", "White/Liver", "Sable/White", "Calico", "~
## $ intake_date      <date> 2019-01-03, 2015-07-05, 2016-04-14, 2013-10-21, 2014~
## $ yr               <dbl> 2019, 2015, 2016, 2013, 2014, 2017, 2019, 2015, 2020,~
## $ mo               <ord> Jan, Jul, Apr, Oct, Jun, Feb, Apr, Jul, Jun, Jun, Aug~
```

How did I know to do that? I Googled get month name in lubridate. The first result took me to a lubridate.tidyverse.org page that explained how to do it. TBH, I was lucky to find the answer in the first result, but I do usually try official tidyverse pages first. I'm also used to reading their documentation.

You'll see there the default label for months is to abbreviate the month. If you wanted the long names you would also add and argument `abbr = FALSE`. You still need the `label` argument, too. We'll stick with the short names.

10.7.3 Extract YYYY-MM format

For our next challenge, we want to create a `yr_mo` column that has both the year and month of our date in YYYY-MM format.

TBH (I have a lot of those today), we could use `str_sub()` to do this, but then I wouldn't get to show you how to format dates in different ways.

For this one, I think we'll add the line first and then I'll explain it afterward.

1. Edit your mutate function to add yet another rule (don't forget the comma!)
2. Add the rule below, run it, then read about it below.
3. Check out the last row of your glimpse to make sure the column was created and in YYYY-MM format.

```
date_parts <- date_fix %>%
  mutate(
    yr = year(intake_date),
    mo = month(intake_date, label = TRUE),
    yr_mo = format(intake_date, "%Y-%m") # new rule to make YYYY-MM
  )

# peek
date_parts %>% glimpse()
```

```
## Rows: 132,214
## Columns: 16
## $ animal_id        <chr> "A786884", "A706918", "A724273", "A665644", "A682524"~
## $ name             <chr> "*Brock", "Belle", "Runster", NA, "Rio", "Odin", "Beo~
## $ date_time        <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ month_year       <chr> "01/03/2019 04:19:00 PM", "07/05/2015 12:59:00 PM", "~
## $ found_location   <chr> "2501 Magin Meadow Dr in Austin (TX)", "9409 Bluegras~
## $ intake_type      <chr> "Stray", "Stray", "Stray", "Stray", "Stray", "Owner S~
## $ intake_condition <chr> "Normal", "Normal", "Normal", "Sick", "Normal", "Norm~
## $ animal_type      <chr> "Dog", "Dog", "Dog", "Cat", "Dog", "Dog", "Dog", "Dog~
## $ sex_upon_intake  <chr> "Neutered Male", "Spayed Female", "Intact Male", "Int~
## $ age_upon_intake  <chr> "2 years", "8 years", "11 months", "4 weeks", "4 year~
## $ breed            <chr> "Beagle Mix", "English Springer Spaniel", "Basenji Mi~
## $ color            <chr> "Tricolor", "White/Liver", "Sable/White", "Calico", "~
## $ intake_date      <date> 2019-01-03, 2015-07-05, 2016-04-14, 2013-10-21, 2014~
## $ yr               <dbl> 2019, 2015, 2016, 2013, 2014, 2017, 2019, 2015, 2020,~
## $ mo               <ord> Jan, Jul, Apr, Oct, Jun, Feb, Apr, Jul, Jun, Jun, Aug~
## $ yr_mo            <chr> "2019-01", "2015-07", "2016-04", "2013-10", "2014-06"~
```

OK, for this mutate we used the `format()` function, which takes two arguments:

- The column we are pulling from: `intake_date` in our case.
- The format that we want that date in: `"%Y-%m"`.

How did I know to do that? I Googled it. Sound familiar? I don't remember exactly the terms, but I found this page that gave me enough to figure it out.

I didn't need the `as.Date()` part because I was already working with a date column.

The funky `%Y` or whatever is really just a code. The link above has a list of the common ones you might use, though there are many more. Here are those common formats:

Symbol	Meaning	Example
%d	day as a number (01-31)	01-31
%a	abbreviated weekday	Mon
%A	full weekday	Monday
%m	month (01-12)	01-12
%b	abbreviated month	Jan
%B	full month	January
%y	2-digit year	07
%Y	4-digit year	2007

10.7.4 Reorder and drop columns

OK, we have a mess of date columns all over our data now. We are done creating new ones, so let's reorganize our data to put the date columns we want at the beginning and drop some we don't need anymore.

1. Edit your chunk to pipe the equation we've been working into a `select()` statement.
2. Add that as indicated below. I'll explain after.

```
date_parts <- date_fix %>%
  mutate(
    yr = year(intake_date),
    mo = month(intake_date, label = TRUE),
    yr_mo = format(intake_date, "%Y-%m") # new rule to make YYYY-MM
  ) %>%
  select(intake_date, yr, mo, yr_mo, everything(), -date_time, -month_year)

# peek
date_parts %>% glimpse()
```

```
## Rows: 132,214
## Columns: 14
## $ intake_date      <date> 2019-01-03, 2015-07-05, 2016-04-14, 2013-10-21, 2014~
## $ yr               <dbl> 2019, 2015, 2016, 2013, 2014, 2017, 2019, 2015, 2020,~
## $ mo               <ord> Jan, Jul, Apr, Oct, Jun, Feb, Apr, Jul, Jun, Jun, Aug~
```

```
## $ yr_mo            <chr> "2019-01", "2015-07", "2016-04", "2013-10", "2014-06"~
## $ animal_id        <chr> "A786884", "A706918", "A724273", "A665644", "A682524"~
## $ name             <chr> "*Brock", "Belle", "Runster", NA, "Rio", "Odin", "Beo~
## $ found_location   <chr> "2501 Magin Meadow Dr in Austin (TX)", "9409 Bluegras~
## $ intake_type      <chr> "Stray", "Stray", "Stray", "Stray", "Stray", "Owner S~
## $ intake_condition <chr> "Normal", "Normal", "Normal", "Sick", "Normal", "Norm~
## $ animal_type      <chr> "Dog", "Dog", "Dog", "Cat", "Dog", "Dog", "Dog", "Dog~
## $ sex_upon_intake  <chr> "Neutered Male", "Spayed Female", "Intact Male", "Int~
## $ age_upon_intake  <chr> "2 years", "8 years", "11 months", "4 weeks", "4 year~
## $ breed            <chr> "Beagle Mix", "English Springer Spaniel", "Basenji Mi~
## $ color            <chr> "Tricolor", "White/Liver", "Sable/White", "Calico", "~
```

Alright, that is a helluva a select statement. The only *new* thing there is `everything()`, which selects (you guessed it) everything not already named. The selects come in order, so it works like this:

- select the intake_date, yr, mo, yr_mo
- select everything else
- remove date_time and month_year, which we don't need anymore

10.8 On your own: Filter dates

You've filtered data by date several times before, so you can do this on your own. It will affect all your work later, so you'll want to check your work. Here are the directions and some hints:

1. Our data goes back in time to 2013. We only want years 2016 and newer. Filter out the older data.
2. We don't want the partial month of October 2021 as it is not a full month. Filter out that data.
3. Save your filtered data into a new tibble.
4. Check your data to make sure it is right using `summary()`

10.9 On your own: Export your data

We're separating the import/cleaning from the rest of our analysis so we don't have to rerun that code all the time. We did the same with Billboard and Military Surplus assignments.

As such, you can handle this one on your own as well, with this guidance:

1. Export your data as an `.rds` file into your `data-processed` folder.

2. Name the file starting with `01-` so you can tell later where it came from. The rest of the name is up to you, but make sure it is in the right place and remember the name so you can open it in the next notebook.

10.10 Set up your analysis notebook

1. Set up a new notebook and call it `02-analysis.Rmd`
2. Create a setup section and load the following libraries:

```
library(tidyverse)
library(lubridate)
library(scales)
library(plotly)
library(clipr)
```

You *might* have to use `install.packages()` for the "scales" library, I'm not sure. You should have clipr from the last chapter.

For the love of your favorite deity (or none at all), please please please remember to run your libraries each time you open your notebook to work on it. That tripped up several folks over the past couple of weeks. **Restart R and Run All Chunks** is your friend.

10.11 On your own: Import your cleaned data

1. Yep, set up a new section
2. Import your data from the last notebook. You should be using `read_rds()`
3. Save the imported data into a tibble called `intakes` so we are all on the same page.

10.12 Question 1: Intakes by year

Our question is: **Are animal intakes increasing or decreasing each year since 2016? Plot a column chart of intakes by year. Don't include 2021 since it is not a full year.**

Remember that we break these plots into two parts:

- Summarize your data
- Plot your data

While I have not problem with you using group_by/summarize/arrange for this, this assignment is all about counting rows, so I think it is worth revisiting the `count()` function. Take a quick review of the count function from Bilboard.

With that in mind, write a summary that counts the number of rows for each year. Remember we have the `yr` column we created earlier.

1. Create a new section about "Intakes by year."
2. Build a summarize table that counts rows by year. Name your counted column `count_animals`.
3. Filter out any results from 2021 since that is not a full year.
4. Save the result into a tibble called `intakes_yr_data`.

Don't overthink it

```
intakes_yr_data <- intakes %>%
  count(yr, name = "count_animals") %>%
  filter(yr < 2021)

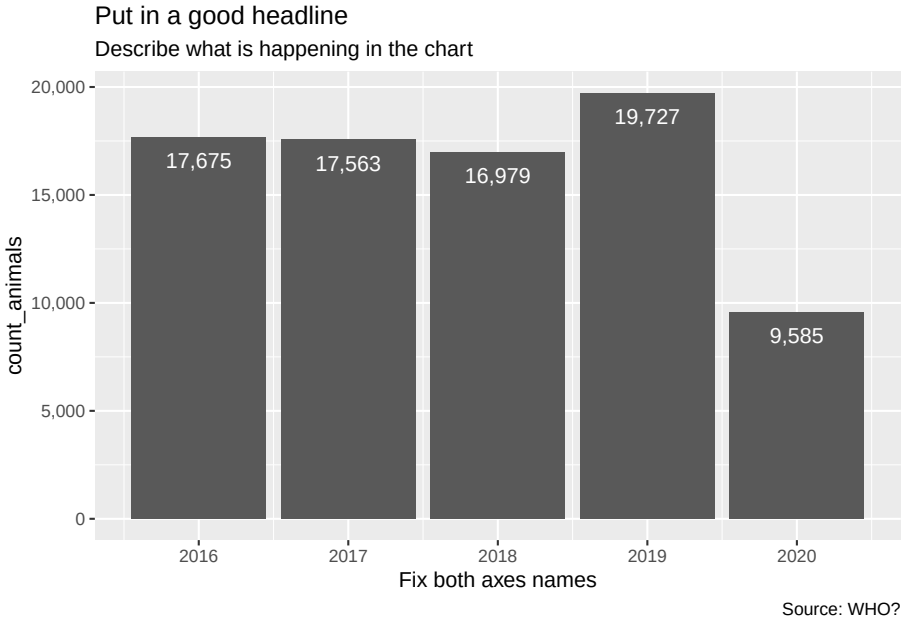
# peek at it
intakes_yr_data
```

```
## # A tibble: 5 x 2
##      yr count_animals
##   <dbl>         <int>
## 1  2016         17675
## 2  2017         17563
## 3  2018         16979
## 4  2019         19727
## 5  2020          9585
```

10.12.1 On your own: Column chart of intakes by year

You've built column charts before for princesses and ice cream. No need to flip the coordinates on this one. Here I want you to build a column chart based on the data you just made, and I want you to make it as complete as possible.

1. Plot a `geom_col` chart using `yr` on the x axis and `count_animals` on the y axis.
2. Include labs for title, subtitle, caption and good axis names
3. Again, no need to flip the axis. Year can stay along the bottom.



Once you've built the chart, what does it tell you? I'm sure you can guess what happened in 2020, but to be sure you would want to do some reporting and talk to the animal center to learn how they dealt with the pandemic.

10.12.2 Getting commas on numbers

One thing you might notice different about my example chart is I have commas in the values on the Y axis and within the text labels. It's a common need and there is a solution, and why we've installed and loaded the "scales" library:

1. Add the following as a new layer on your chart, but don't forget the `+` on the line above: `scale_y_continuous(labels = comma)`
2. Run that and make sure it works.

For the numbers as labels, in your `geom_text()` layer you can wrap your `aes(labels = count_animals)` in a comma() function.

1. In your geom_text() layer, edit the aes to this: `label = comma(count_animals)`.

10.13 Question 2: Intakes by month

Are there seasonal monthly trends in overall animal intakes? Plot intakes by month from 2016 to current, including 2021. (One long line or a column chart.)

To find this answer we'll plot as a single line chart and along the way we'll learn how to rotate our x axis labels.

10.13.1 Intakes by month as a line

We need our data first. To show every month on one chart we have to know how many animals were brought in by both year and month. This is why we created the `yr_mo` column earlier. This time we can keep the data from 2021.

1. Create a new section noting you are plotting by month as a line.
2. Use `count()` to summarize the data by the `yr_mo` column. Name your new column `count_animals`.
3. Assign the result to a new obljject using `<-` named `intakes_yrmo` so we are on the same page.

I'm too nice

```
intakes_yrmo <- intakes %>%
  count(yr_mo, name = "count_animals")

# peek
intakes_yrmo
```

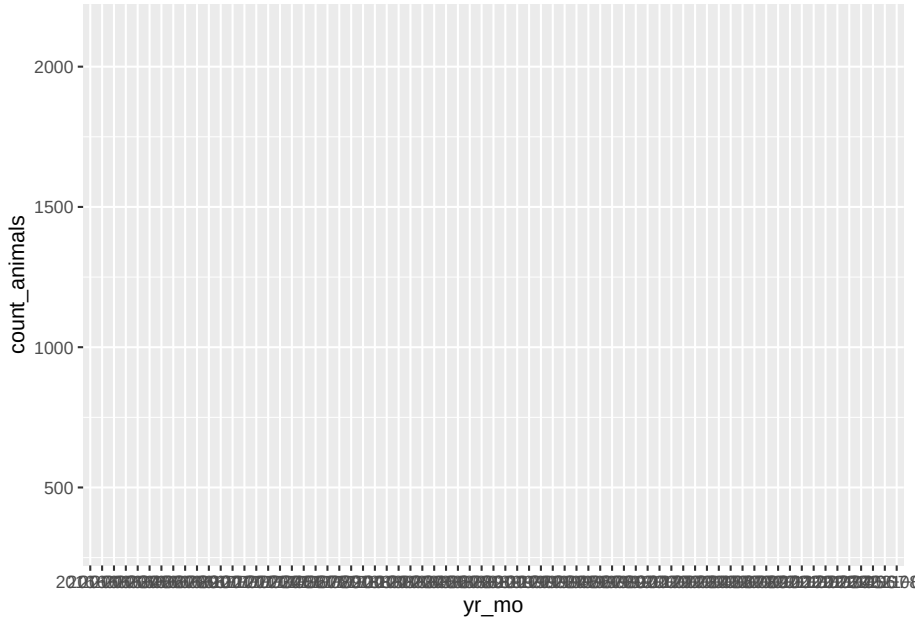
```
## # A tibble: 69 x 2
##    yr_mo   count_animals
##    <chr>           <int>
##  1 2016-01          1216
##  2 2016-02          1194
##  3 2016-03          1419
##  4 2016-04          1560
##  5 2016-05          2037
##  6 2016-06          1634
##  7 2016-07          1409
##  8 2016-08          1653
##  9 2016-09          1539
## 10 2016-10          1387
## # ... with 59 more rows
```

Now that you have the data, we'll try to plot this as a line. You might end up with this error:

"geom_path: Each group consists of only one observation. Do you need to adjust the group aesthetic?"

```
intakes_yrmo %>%
  ggplot(aes(x = yr_mo, y = count_animals)) +
  geom_line()
```

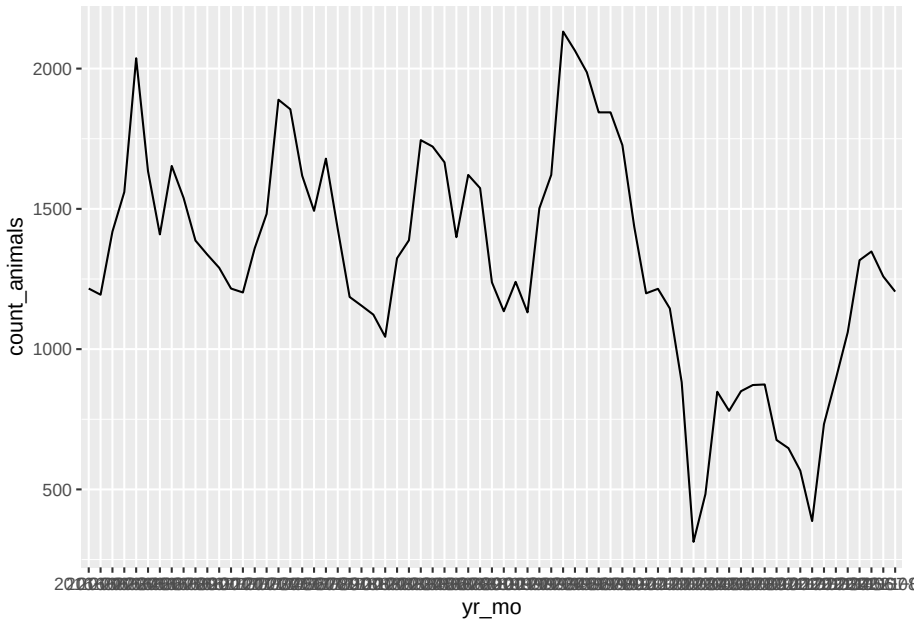
```
## geom_path: Each group consists of only one observation. Do you need to adjust
## the group aesthetic?
```



The `geom_line()` function expects you to have more than one line to plot, and when we don't have that (and often when we do) we have to specify the "group" or value to split the lines. When we have only one line we need to specify that:

1. Edit your `geom_line()` function to add `group = 1` inside it.

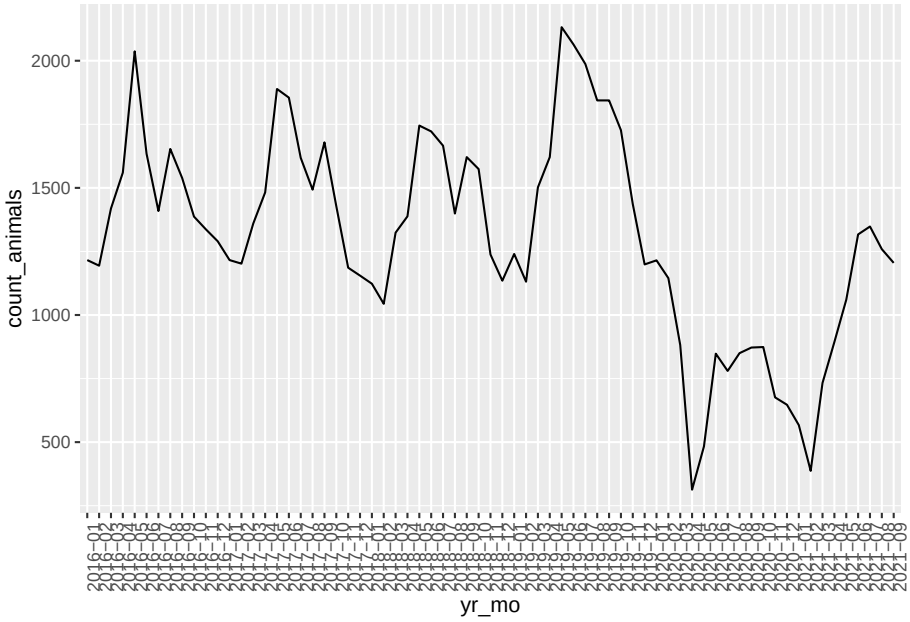
```
intakes_yrmo %>%
  ggplot(aes(x = yr_mo, y = count_animals)) +
  geom_line(group = 1) # group added here
```



OK, but we can't read the X axis names, and it wouldn't make sense to flip the whole line chart, so intead we'll flip the axis names. Remember that we can change just about *anything* in the theme, right?

1. Add the layer noted below to your plot. Don't forget the + on the previous line.

```
intakes_yrmo %>%
  ggplot(aes(x = yr_mo, y = count_animals)) +
  geom_line(group = 1) +
  theme(axis.text.x = element_text(angle = 90)) # flips the axis name
```



You can change the number there to get different angles, like `45` degrees.

How the heck did I know that? Can you guess? Yes, Google is your friend, searching for something like "rotate axis names in ggplot." To be honest, in this case I used a different method: I phoned a friend for help and Jo Lukito found this. She should've chided me with Let me Google that for you.

She also helped me with the next challenge.

10.13.2 Make the chart wider

Our chart is still too squished to read. For this one, the answer is in the R chunk settings so I can't show you in code with this book.

1. In your R chunk where you name the chunk like this: `{r chunk-name}`, edit it to say `{r chunk-name, fig.width = 7}` and then run it. Hopefully you have a more descriptive name chunk than chunk-name, though.

Unfortunately that change doesn't really show in this published book but it does help you in your own R Notebook.

10.13.3 So, what of this

We could plot this same data as a `geom_col()` but it would have the same issues. That said, we've proven two things here:

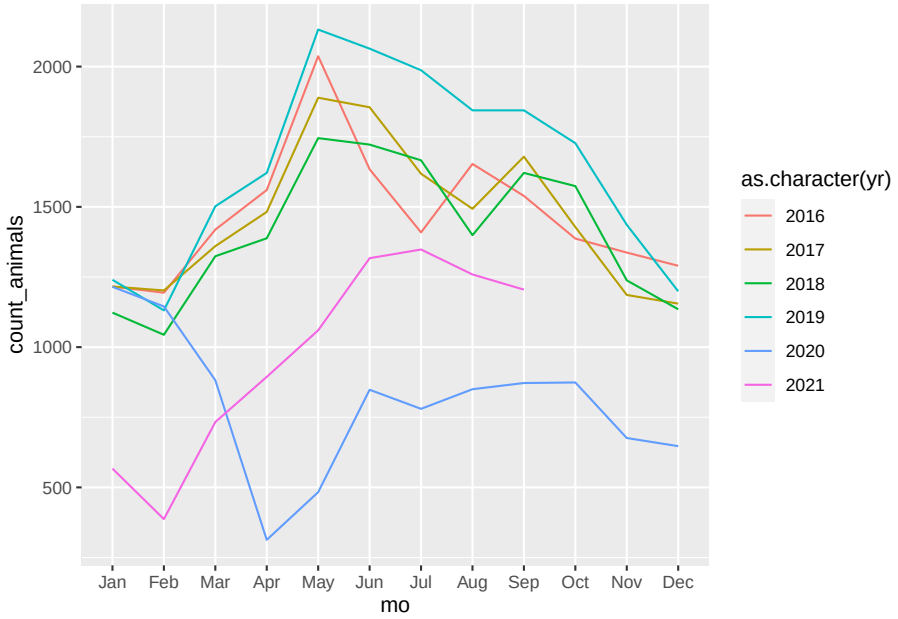
- There is definitely a seasonal trend to animal intakes. There seem to be surges each spring and early summer, between March and June,
- We've also proved this is a crappy way to see that. Let's find another way.

No need to clean up that line chart beyond what we've done, but keep it in your notebook.

10.14 Question 3: Intakes by month, split by year

Plot that same monthly trend, except with month as the x axis, with a new line for each year.

Let's visualize success so we know what we are going after. This is the basic plot before being cleaned up.



Here we can see each year by month all on the same scale and see that the lowest year by far was 2020, and 2021 is trending back up but doesn't yet reach the level of previous years.

- To make this chart you need the `mo` column that we created earlier so we can plot our months across the X axis. The brilliant thing we did was build that month from a real date so it retains its factor order of Jan, Feb, etc., instead of being alphabetical.

- But we also need the `yr` column so we can set the group and color of the lines by that year. So well be grouping by two values.
- We'll be counting rows as we have been and name them `count_animals`.

We'll do this one *together*, but you'll have to build a similar chart in a bit so pay attention.

10.14.1 Prepare the data with mo and yr

1. Start a new section that you are plotting intakes by month, split by year
2. Add the R chunk below

```
mo_yr_data <- intakes %>%
  count(mo, yr, name = "count_animals")

# peek
mo_yr_data
```

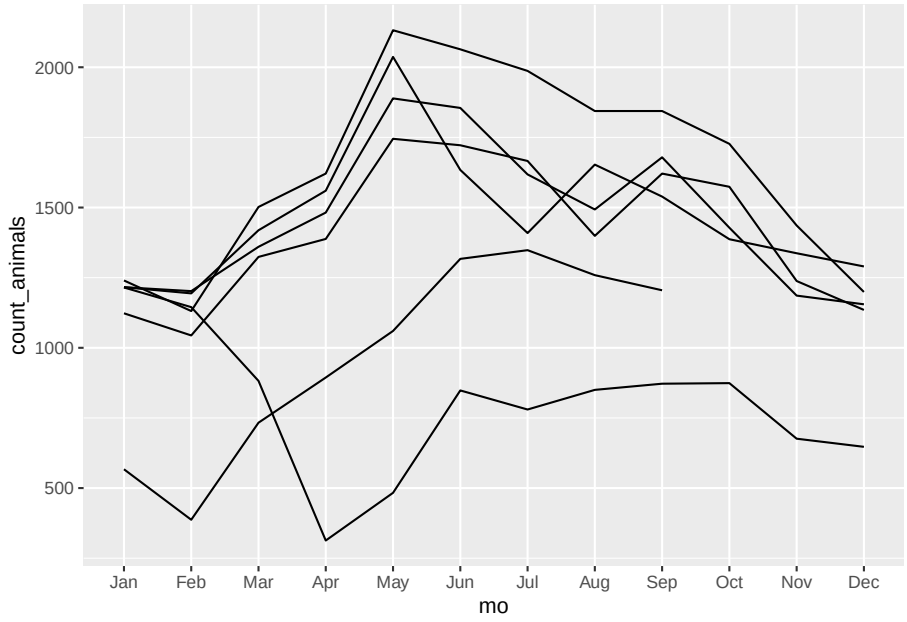
```
## # A tibble: 69 x 3
##    mo       yr count_animals
##    <ord> <dbl>         <int>
##  1 Jan    2016          1216
##  2 Jan    2017          1216
##  3 Jan    2018          1123
##  4 Jan    2019          1240
##  5 Jan    2020          1215
##  6 Jan    2021           567
##  7 Feb    2016          1194
##  8 Feb    2017          1202
##  9 Feb    2018          1044
## 10 Feb    2019          1131
## # ... with 59 more rows
```

10.14.2 Plot by month, split by year

So we have this month name `mo` to plot along the X axis and the `count_animals` to plot along the Y axis. We have `yr_mo` to group our lines.

1. Note that you are plotting now
2. Add the base chart as noted below.

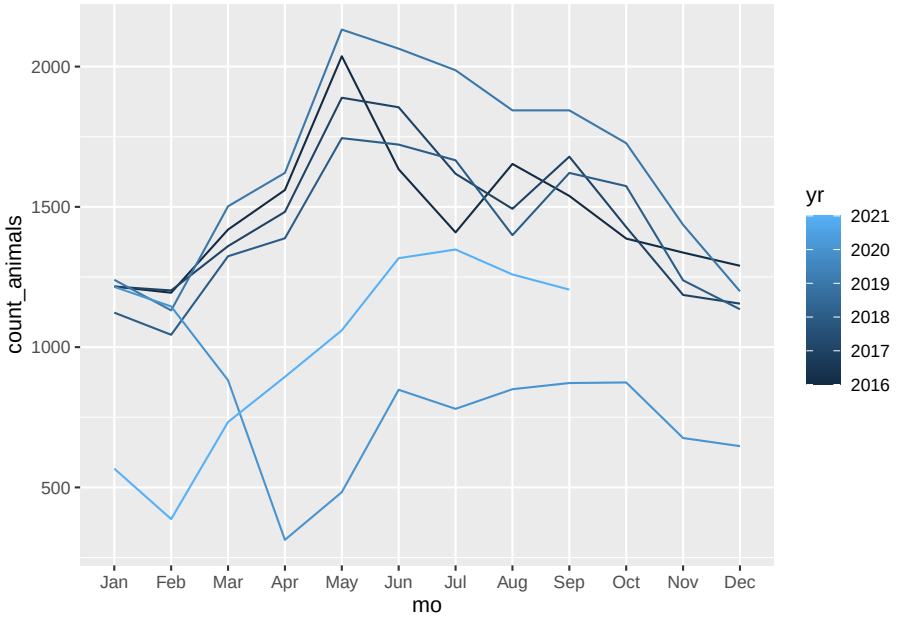
```
mo_yr_data %>%
  ggplot(aes(x = mo, y = count_animals)) + # sets our x,y
  geom_line(aes(group = yr)) # separates the lines
```



We are getting somewhere. We can see each year on the plot, but we can't tell them apart.

1. Inside the `aes()` for `geom_line()`, add `color = yr`.

```
mo_yr_data %>%
  ggplot(aes(x = mo, y = count_animals)) + # sets our x,y
  geom_line(aes(group = yr, color = yr)) # adds color to lines based on yr
```



Well, that is an interesting color choice for our years. Ggplot is using the `yr` value as a continuous number, and indeed if we look back up at our data we can see the column datatype is `<dbl>` which means it is a number. That happened way back when we created the `yr` column.

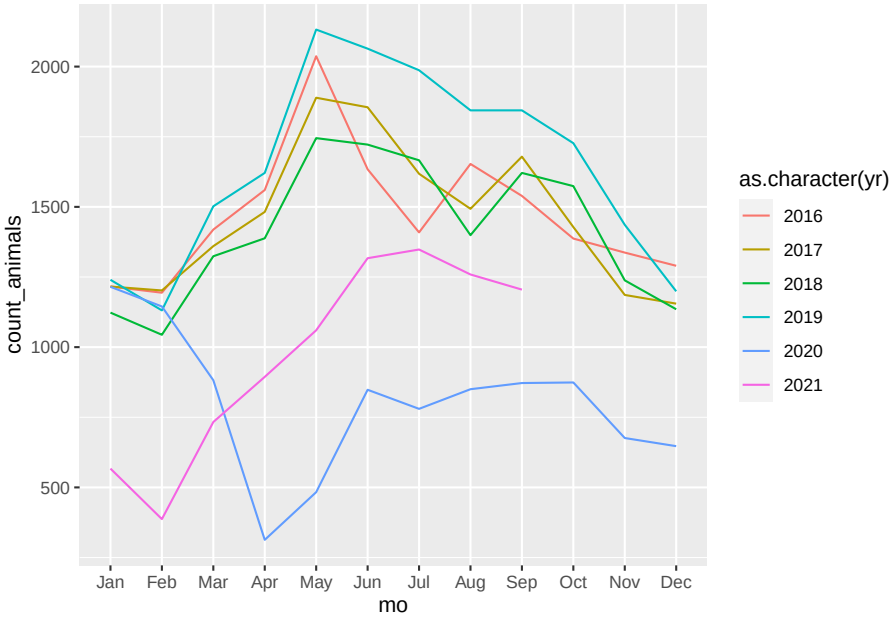
What to do? We could go back to the import/clean notebook and adjust that to make it a **character** datatype. Or we could go to our data prep step and mutate that column there. Or we could just fix it in the chart.

We'll just fix it in the chart.

10.14.3 Adjust the yr value to a character

1. In your plot code, find the `color = yr` part and change it to `color = as.character(yr)`

```
mo_yr_data %>%
  ggplot(aes(x = mo, y = count_animals)) +
  geom_line(aes(group = yr, color = as.character(yr))) # color = yr updated
```



What we've done is used the `as.character()` function to change the yr value from a number to text. Now ggplot thinks it is a **category** instead of a **continuous number**. If this came up again, we might go back to the import notebook and fix it, but this works for now.

But do note that also changed the label for our legend. We needed to fix that anyway and we can do that with `labs()` the same way we edit our x and y axes labels. Just use `color = "Year"` or whatever you want call it.

10.14.4 Clean up your intakes by month, split by year chart

1. Edit your plot code to do the following:

- Add a title and subtitle
- Add the source of our data in the caption
- Clean up your axis and color names

10.15 Question 4: Animal types by month

Do certain types of animals drive seasonal trends? Use five full years of data (not 2021) to summarize intakes by animal type and month. Plot with month on the x axis with a line for each animal type.

To envision success, just look at the chart you just made. Instead of having a line for each year, we want a line for each animal type.

Before we get there, let's take a look at those animal types.

10.15.1 Recode animal types

1. Create a section that you are plotting animal types by month
2. Create a quick `count()` on the column `animal_type`. I'll wait.

You need to do the code in your notebook, but here is the output:

```
## # A tibble: 5 x 2
##   animal_type     n
##   <chr>       <int>
## 1 Bird          513
## 2 Cat         34242
## 3 Dog         50662
## 4 Livestock      18
## 5 Other        4864
```

Note that there are *way* more Dogs and Cats than other animal types. We will create a new tibble that renames each row with Bird or Livestock as Other so they will all be in the same category.

We do this with `recode()` within a `mutate()`. You can read about recode here. If you read that you'll notice there is a weird thing in that it bucks the R trend of naming *new* things first and then filling them. Recode names the *old* thing first and then defines the new name. Oh, well. It's what we got.

Our recode will happen inside a `mutate()` function but let's talk about `recode()` first.

- The first argument is the column you are recoding: `recode(animal_type)`
- Next we give it a list of the values we want to change, then set it equal to the value we want it to be: `recode(animal_type, "Old Value" = "New Value")`. We can stack the changes inside recode.

1. Write a markdown note that you are going to recode `animal_type`.
2. Add the code block below, run it, then read the explanation.

```
intake_types <- intakes %>%
  filter(yr < 2021) %>% # removes 2021 because its partial yr
  mutate(
    animal_type = recode(animal_type, # first arg is the column to look at
```

```
      "Bird" = "Other", # old name, then new name
      "Livestock" = "Other" # old name, then new name
    )
  )

intake_types %>%
  count(animal_type)
```

```
## # A tibble: 3 x 2
##   animal_type     n
##   <chr>       <int>
## 1 Cat         30474
## 2 Dog         46222
## 3 Other        4833
```

Let's break this down:

- At the top We started with our data and assigned it to a new tibble. We don't want to change the original data.
- We start a mutate. We set it to change the original `animal_type` column with our recoded values.
- Within the `recode()` function we say with column we are chaning (`animal_type`) and then we note the old and new values, each in quotes.
- At the bottom we take the new tibble and pipe it into a `count(animal_type)` so we can check the results. Note that the *changed* data is still row-level data for each animal. We just piped that result into `count()` so we could check the recode changes. We didn't "save" that last count.

Like I did with the date parts earlier, when I built this I went ahead and started by assigning the original data to a new tibble and then printing that new tibble before I worked on the rest line-by-line.

Recode is a valuable method to clean data, but you have to be careful that you aren't overwritting your original data. I was comfortable doing so because we could easily eyeball the results to ake sure they were right, and we were saving the result into a new tibble anyway, so the original data is safe in `intakes`.

10.15.2 On your own: Count animal types by month

Now it is time for you to prep this data for our plot. Here is the logic: You are counting the rows by both the month `mo` and the `animal_type`. Remember to start with the 1intake_types‘ tibble you just created.

1. Write notes that you are creating the animal type summary by month

2. Write code that counts the rows based on month and animal type. Name your new column something that makes sense.

You should end up with a summary with three columns: mo, animal_type and whatever you named the count column.

10.15.3 On your own: Plot animal types by month

Remember this is very similar to your plot above where you made a line chart of counts split by year, but this time you are splitting by `animal_type`.

1. Write notes that you are plotting
2. Build the plot and include all the labs: title, subtitle, caption and fix the axis and color names.

One thing about describing this chart in your subtitle: You have to describe in your own words that this chart combines counts by each month over five years: 2016-2020. It's a nuance a reader needs to understand.

What did you learn about your plot? Is there a type of animal that drives that spring/summer surge? Tell me in text in your notebook how you might find out why that is so.

10.16 What we learned

- We learned how to download data from the City of Austin's data portal. We just scratched the surface there.
- We cleaned our date using `str_sub()` and extracted date parts with `year()` and `month()` and used `format()` to pull a date part in a specific way.
- We used `count()` to make some summaries. Remember this is the shortcut for group_by/summarize/arrange (GSA) that can only count rows and not do any other math. But it is damned handy.
- We prepped data and plotted to find answers to specific questions.
- We learned how to add commas to axis names in ggplot with the scales package using.
- We used `recode()` to update some values in our data.

10.17 Turn in your work

1. Make sure all your code runs fresh and knit it
2. Stuff the folder and submit it to canvas

10.18 What's next

I had planned on adding some more Datawrapper practice with this data which requires some pivoting, so keep this project around. We may revisit it.

Chapter 11

Census introduction

The U.S. Census Bureau has a wealth of data that can help journalists tell stories. This chapter is *not* a comprehensive guide on how to use it, but instead an introduction on some ways you can.

There is a recorded lecture in Canvas that covers different data programs within the census, so you should watch that before embarking on this. You do need some basic knowledge of those programs for this to make sense.

11.1 Goals of this chapter

- Introduce the official U.S. Census Bureau site data.census.gov.
- Explain API keys and set them up for census data.
- Introduce and explore the tidycensus package to pull census data into R.
- Introduce some mapping techniques using tidycensus.
- Note some other packages and methods to use census data.

11.2 Start a tidycensus project

1. Start a new project called `yourname-census`.
2. Start a new R Notebook and title it `broadband.Rmd`.
3. Install tidycensus **in your Console**: `install.packages("tidycensus")`
4. Create a setup section with the following libraries

```
library(tidyverse)
library(tidycensus)
library(scales)
```

11.3 Census data portal and API

Before we dive in, we need to do some setup for your machine.

The Census Bureau has a data portal data.census.gov where you can search for and download data, but the site is ... lacking. There is so much data it can be overwhelming at first. But once you've gained experience with the different programs and offerings it gets easier to find what you want. We'll use the official portal to find our data, but I also want to give a shoutout to CensusReporter, a tool built by journalists for journalists looking for finding census data. But for now, we'll use data.census.gov.

But what we are really here to learn is how to import Census Bureau data directly into R through using their Application Programming Interface, or API.

API's let programming languages like R talk to other systems and request responses, typically through a URL. Packages like tidycensus, which we'll explor here, help us do that behind the scenes. There are a number of packages that do this with the census API, and they each work a little differently to solve different challenges.

> Manually downloading census data is usually a multiple-step and multiple-decision process. An advantage to using the API is you can script that decision-making process for consistency. It's transparent and repeatable.

The Census requires a free API key to use their service. It's like your personal license, and should not be shared with others. We do NOT write our API key into our R Notebooks.

If you do not already have a Census API Key or have not installed in yet into your R environment, follow these steps:

11.3.1 Request a key

(I may ask you to do this before we get into this chapter. If I do, then you already have your key.)

1. Sign up for an API key here. You should get an email back within a few minutes that includes a long string of random numbers and letter and a link to activate the key. Do activate that key through the link in the email.

11.3.2 Install a key

2. **In your Console (NOT in Markdown)** enter the code below, **but replace your_key_here with the key you got in the email**. Keep the quotes there.

```
census_api_key("your_key_here", install = TRUE)
```

1. Next you'll need to reload the environment. **In your Console** do this:

```
readRenviron("~/.Renviron")
```

You can check to make sure it is working by running the following **in your Console**:

```
Sys.getenv("CENSUS_API_KEY")
```

Which should return your API key string. That just let's you know it is working.

> You only have to set up your API key **once on each machine**. Once installed, it gets automatically loaded when you restart R.

Here is more explanation on saving keys if you need it.

11.4 About the tidycensus package

Kyle Walker is a professor at TCU who developed the tidycensus package to return census data in tidyverse-ready tibbles with options to include spatial geometry to make maps. He is also writing a book Analyzing US Census Data.

Basically, it's brilliant.

There is excellent documentation on using tidycensus. In this chapter we'll walk through searching through data, fetching data and making a map.

11.4.1 Finding the data you want

To get data through tidycensus you need to know the "census program" and the specific "variables" or data points you want and there are thousands of these IDs across the different Census files. To be honest, I find this the hardest part of the process.

We can use a function in tidycensus called `load_variables` to help find what we need, but we must know what "program," the "year" and sometimes the "product" our data is in before we can use it to find our specific data.

I've found the best way to start this process is to find the data you want on data.census.gov first, which we'll walk through in a moment.

We'll start by using the American Community Survey (ACS) program. The survey comes in 5-year and 1-year versions. The 1-year version is more current, but is limited in geographies available. The 5-year version has more data, but it encompasses 5-years of answers. See the lecture for more on that. We'll uses the 5-year versions for . To access a 5-year data set, we use the last year of the data, so we use "2019" for the 2015-2019 5-Year ACS, which is the most recent release.

Just to make it more complicated (sorry), there are different "products" within the ACS: Detailed Tables, Subject Tables and Data Profiles.

Finding the variables is the hardest part of this exercise, I think.

11.5 Broadband access by county in Texas

Here is our first challenge: **What counties in Texas have the highest percentage of households with broadband access of any type.**

Here is how we go about finding what we need:

1. Go to http://data.census.gov.
2. In the search box there, search for "broadband access"
3. The first **Table** results should be "TYPES OF COMPUTERS AND INTERNET SUBSCRIPTIONS." Click on that table to open it.
4. Click on the dropdown at top right (indicated below) and change it to "2019 ACS **5-Year** Estimates Subject Table." (We use the 5-year version so we can get results for every county in Texas. The 1-year version would only have larger counties.)

If you have trouble finding the table, you can get it here.

The default view of this table shows us values for this data across the entire US. This particular table has both **Total** values (the **estimate**) and the the **Percent** values (which is a percentage of that variable vs **Total Households**, the "univerise" of our data shown on the first line of data.)

The values we want for this is the one I've underlined above with the red line ... it is the **Percent** of **households** with **Broadband of any type**.

Before we can ask for our data through the API, we need to find the **exact variable name** for this value, but that isn't shown here on the table (damnit.) But we'll use the information on this screen to find it.

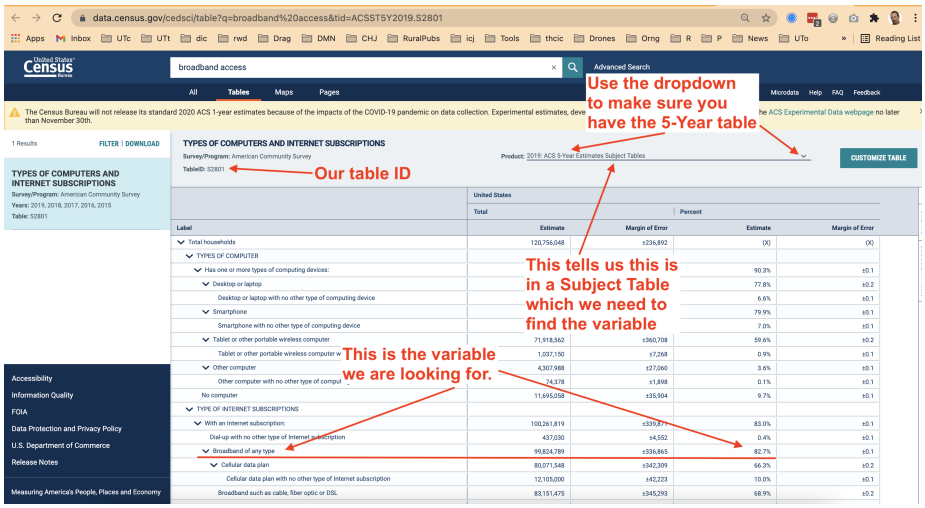


Figure 11.1: Broadband table

11.6 Using the tidycensus load_variable() function

Looking at my marked up screenshot above, I look for several things when I'm trying to get data:

- I look at the table name: `S2801`
- I look at the "Product" line in the result. It will say something like **"2019 ACS 5-Year Estimates Subject Table"**. It is the last two words in this line we are looking at to determine the program.

Here is a schematic of the difference.

Product	tidycensus value	Details
Detailed Tables	"acs1" or "acs5"	Tables about specific subjects or characteristics. Typically estimates and margin of errors only.
Subject Tables	"acs5/subject"	Topic-based collections. Has both estimates and percentages.

Product	tidycensus value	Details
Data Profiles	"acs5/profile"	Specific collections around social, economic, housing and demographic areas. Can have both estimates and percentages.

Here we'll use the `load_variables()` function again, but we have to specify that we are looking at **acs5/subject** tables since that is the program we are using.

1. Create a new section that you are finding our broadband variable.
2. Add the code below and run it.

```
v19_subject <- load_variables(2019, "acs5/subject", cache = TRUE)
```

- The first item `v19_subject` is the R object we are filling.
- We start with `load_variables()` and the first argument is the year of data that we want. Fo us this is: `2019`.
- The second argument is the profile type: `acs5/subject` for the subject tables.
- The third argument `cache = TRUE` saves this table to your computer so it doesn't download it each time you view it.

Once you have saved the table into an object, you'll see it listed in your Environment pane. Click on the little table icon on the right side to open it as a table:

Once you have the table open:

1. Click on the **Filter** box and put in our table id under **name**: `S2801`. That filters the results to just the table we want.
2. Under **label** type in "broadband," which will further filter the table just to rows that have that term.
3. We are looking for rows that start with "Estimate!!Percent" and we want the one that ends with "!!Broadband of any type." It happens the be the first Percent result if you haven't reordered the table accidently.
4. If you hover over the **label** column and leave your cursor there, you'll get a pop-up that shows you the value.
5. Once you find the row you need, you copy the **name** value for that row. The one we need is `S2801_C02_014`.
6. Copy that value and then add it as a Markdown note in your notebook so you have it saved there.

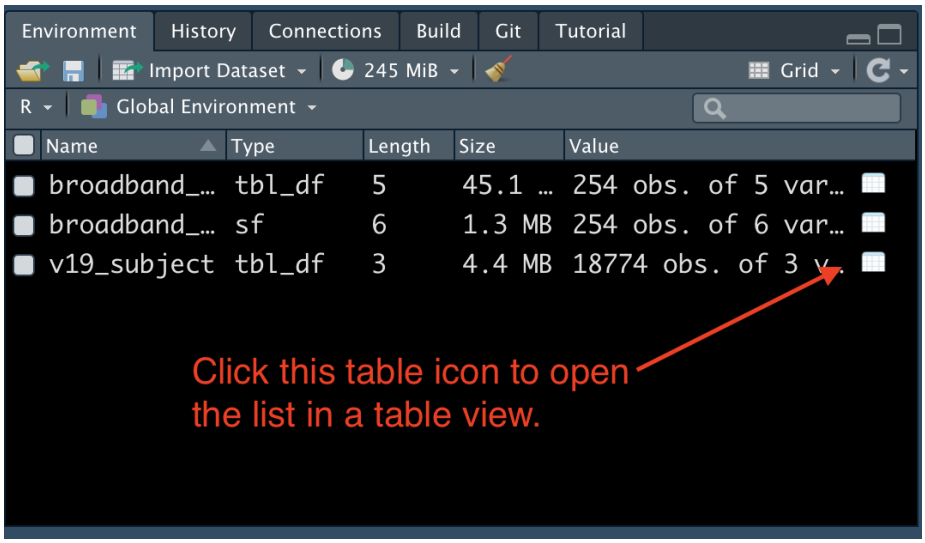


Figure 11.2: Open table

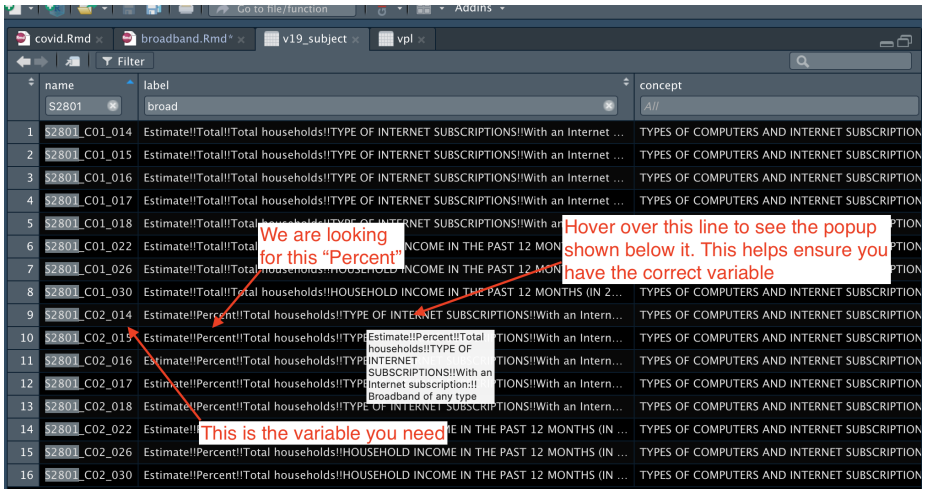


Figure 11.3: Broadband filter

I swear this is the hardest part of this. I wish there were an easier way, but I haven't found one. Believe me, I've asked around about it.

11.7 Use tidycensus to fetch our data

Now that we have the variable we are looking for, we'll grab that value for the U.S. so we can make sure we have the right thing.

To fetch data from the ACS program we use a the function `get_acs()`. It needs at least three things:

- The **year** of the dataset. For us this is `2019`.
- The variable (or **variables**) you seek. For us that is `S2801_C02_014`. We'll also name for clarity.
- The **geography** scope of the data. We'll first use `us` to check we have the right thing.

1. Start a new section and note you are getting the broadband percent data.
2. Add the following chunk and run it.

```
get_acs(
  year = 2019,
  variables = c(broadband = "S2801_C02_014"),
  geography = "us"
)
```

```
## Getting data from the 2015-2019 5-year ACS

## Using the ACS Subject Tables

## # A tibble: 1 x 5
##   GEOID NAME          variable  estimate   moe
##   <chr> <chr>         <chr>        <dbl> <dbl>
## 1 1     United States broadband     82.7   0.1
```

Now for a **very important step**:

1. Check the values you got back for **estimate** and compare it to the value you want from the table you got on data.census.gov.

The **estimate** should be "82.7," which is the percentage of households throughout the US that have broadband access of any type.

If you don't have the right number, then you don't have the correct variable. Make sure that is `S2801_C02_014`.

11.7.1 Get broadband rates by county in Texas

Now that we have confirmed we have the right data, we can get this for Texas counties.

1. Create a subsection or otherwise note you are getting broadband for Texas counties.
2. Add the chunk below and run it and then I'll explain after

```
get_acs(
  year = 2019,
  variables = c(broadband = "S2801_C02_014"),
  geography = "county", # we changed this to county
  state = "TX" # We've added state to filter to just Texas
)
```

```
## Getting data from the 2015-2019 5-year ACS

## Using the ACS Subject Tables

## # A tibble: 254 x 5
##    GEOID NAME                     variable  estimate   moe
##    <chr> <chr>                    <chr>        <dbl> <dbl>
##  1 48001 Anderson County, Texas   broadband     67.9   3
##  2 48003 Andrews County, Texas    broadband     86.2   3.2
##  3 48005 Angelina County, Texas   broadband     82.3   1.7
##  4 48007 Aransas County, Texas    broadband     81.8   3.6
##  5 48009 Archer County, Texas     broadband     75.4   3.4
##  6 48011 Armstrong County, Texas  broadband     81.6   4.8
##  7 48013 Atascosa County, Texas   broadband     80.2   2.5
##  8 48015 Austin County, Texas     broadband     74.3   2.8
##  9 48017 Bailey County, Texas     broadband     70.5   5.8
## 10 48019 Bandera County, Texas    broadband     81.8   3.1
## # ... with 244 more rows
```

Here is what we've done here:

- We have the same `get_acs()` function with the same **year** and **variables**.
- We changed the **geography** argument to "county" which will give us this same variable for *every* county in the country. But we don't want that so ...
- We added a **`state = "TX"`** argument to get just Texas counties.

Your list should start with Anderson County, Texas.

11.7.2 On your own: Arrange to get lowest/highest values

We're going to save the result above into a new R object and then use that to get the two lists: The counties with the most access, and those with the least.

1. **Edit** your chunk to save the `get_acs()` code into a new object called `broadband_tx`. Run it so the object is created.
2. Write a bit of markdown that says you are getting the counties with the lowest access.
3. Write a block that takes `broadband_tx` and arranges it by the `estimate`, which will put the lowest access at the top. Pipe that into `head(10)` to see just the top values.
4. Write a bit of markdown that says you are getting the *hightest* access and then a block to arrange by `estimate` in descening order. Pipe that into `head(10)` as well.

11.7.3 What you have here

If you were writing a story about broadband access, you could note the counties that have the least and most access.

Getting the data answer was easy once you have the variable, right?

Now to blow your mind a little.

11.8 Map broadband access by counties in Texas

Walker's tidycensus package can also bring in spatial files with a single additional line of code, which allows us to map data using ggplot's `geom_sf()` function.

1. Make a new section and note that you are *mapping* broadband access.
2. Add the code chunk below and run it.

   I'm suppressing the output here because it doesn't show well in the book.

```
broadband_tx_geo <- get_acs(
  year = 2019,
  variables = c(broadband = "S2801_C02_014"),
  geography = "county",
  state = "TX",
  geometry = TRUE # this is the only new line
```

```
)

broadband_tx_geo
```

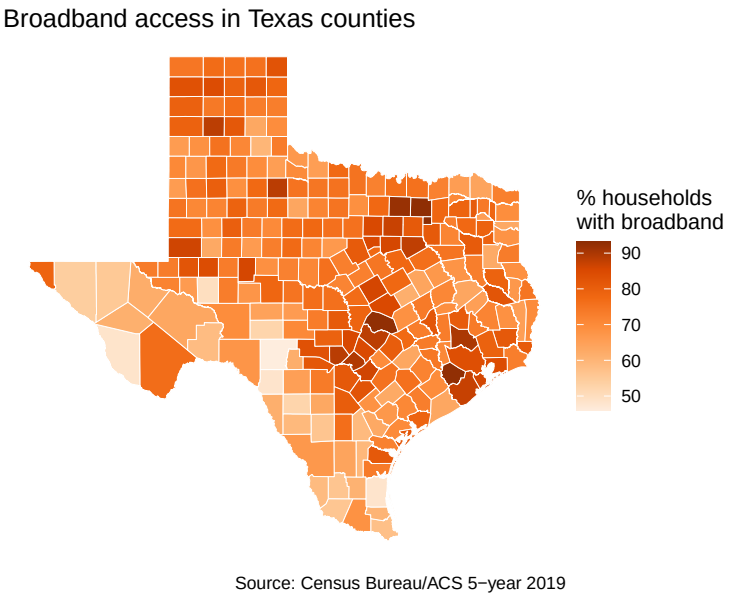
We've added ONE LINE of code here that gives us the geometric shapes for each county in our data. I saved this into a new tibble for clarity.

11.8.1 Plot the broadband map

We'll plot the map here, and it looks like a lot, so I'll explain each part afterward.

I *really* suggest that you build this one line or section at a time. You'll see me do that in the screencast (I hope).

```
ggplot(broadband_tx_geo) +
  geom_sf(aes(fill = estimate), color = "white", size = .1) +
  theme_void() +
  labs(
    title = "Broadband access in Texas counties",
    caption = "Source: Census Bureau/ACS 5-year 2019"
  ) +
  scale_fill_distiller(
    palette = "Oranges",
    direction = 1,
    name = "% households\nwith broadband"
  ) +
  theme(plot.margin = unit(c(10,0,10,0), "pt"))
```



Let's go through it line by line:

- We start with ggplot and fill it with our data.
- We add `geom_sf`, which stands for shapefile. We add an `aes()` of `fill = estimate`, which is the column in our data that has the median family income. At this point, we have a map! The rest is just improving it.
- In addition to the geom, I added `color = "white"` which changes the color of the lines around the counties. I also added `size - .1` to make them thinner.
- `theme_void()` removes the grid lines and axis names. It is pretty much used only with maps.
- We add our `labs()` which you should be familiar with now.
- Then we add `scale_fill_distiller()` with a bunch of settings. I'll be honest, I only know of this scale because of a some tutorial that I have since lost. But let's walk through the options used here:
  - `palette = "Oranges"` changes the colors since I didn't like the default blue. You could try some of these? I googled to find that.
  - `direction = 1` reverses the colors so higher numbers are darker instead of lighter. It makes more sense that way, at least to me.
  - `name = "% households\nwith broadband"` update the name on the legend. Note the weird `\n` in the middle with creates a **new line** with the legend so the name isn't so long.
- The last `theme(plot.margin)` line adds some margins around the plot since `theme_void()` removes all of that. I found that here. The number order is clockwise: top, right, bottom, left.

11.8.2 You did it!

So you have a map of the broadband access by county for Texas.

Getting the data from tidycensus into R is pretty sweet, but it does take a get getting used to. I can tell you it is still easier than downloading CSV data and filtering and selectiong for what you need. And the mapping is just fun.

11.9 Keep this project

We'll use this project in the next chapter, too.

11.10 What we've learned

- We used http://data.census.gov to find interesting data. There is way more to explore there.
- You set up your computer with a U.S. Census API key.
- We've explored the `tidycensus` package by Kyle Walker to fetch data using the Census API.
- You made a map!

> The rest of this chapter is just for your information. You don't need to do anything with it.

11.11 Other useful census-related packages

11.11.1 The censusapi package

Hannah Recht now of Kaiser Health News developed the censusapi package to pull data directly from the Census Bureau into R. See that site for more examples and documentation on use.

Below is an example code chunk from censusapi that pulls the median income estimate (`B19013_001E`) and margin of error (`B19013_001M`), but I'm also including the total population for the county with `B01003_001E`, which was not in the "B19013" table. This is another advantage to using the API, as we are pulling from multiple tables at once. To do this manually, we would have to search for and download two separate data sets and merge them.

```
tx_income <- getCensus(name = "acs/acs5", vintage = 2017,
    vars = c("NAME","B01003_001E", "B19013_001E", "B19013_001M"),
    region = "county:*", regionin = "state:48")
```

Which ends up looking like this:

state	county	NAME	B01003_001E	B19013_001E	B19013_001M
48	199	Hardin County, Texas	55993	56131	3351
48	207	Haskell County, Texas	5806	43529	6157
48	227	Howard County, Texas	36491	50855	2162

This gives us data in a different "shape" than the tidycensus package. It adds new variables as a column as "wide" data instead of new rows in "long" data, which is what tidycensus does.

This censusapi example I have goes through a use of censusapi and finishes out by pulling data form Kyle Walker's tigris to make a median income map.

If you already have the Census data, or perhaps data that is not from the census but has a county name or one of the other geographic code values, then you can use Walker's tigris package to get just the shapefiles. That censusapi notebook also includes use of the tigris package, if you want to see that.

11.12 Interactive maps with leaflet

You can also create interactive maps in R.

Here is a tutorial that walks through creating an interactive map of median income by census tract using the leaflet, mapview, tigris and acs packages. It's a pretty basic map best used for exploration, but it's pretty neat and not too hard to make.

11.13 More resources and examples

Some other resources not already mentioned:

- R Census guide
- Mapping Census Bureau Data in R with Choroplethr by package creator Ari Lamstein
- Using the R Package RankingProject to Make Simple Visualizations for Comparing Populations by former Census Bureau statistician Jerzy Wieczorek

- Baltimore Sun example story and code. Christine Zhang says "Sometimes I prefer the output of one over the other (censusapi vs tidycensus) which is why I alternate.
- Spatial Data Science with R Tutorial.
- Not a tutorial but but this post by Timo Grossenbacher is an explanation and inspiration on how far you can take R in mapping. Here is a version that uses U.S. Census Bureau data.

11.14 Using the data portal to download data

When you download a table from the data.census.gov portal, you get a stuffed archive with three files. Here is an example from a 5-year ACS data set for table B19013, which includes median income data:

- `ACSDT5Y2017.B19013_data_with_overlays_2019-04-20T000019.csv` is the data. It contains two header rows (arg!) with the first row being coded values for each column. The second row has long descriptions of what is in each column.
- `ACSDT5Y2017.B19013_metadata_2019-04-20T000019.csv` is a reference file that gives the code and description for each header in the data.
- `ACSDT5Y2017.B19013_table_title_2019-04-20T000019.txt` is a reference file with information about the table. If data is masked or missing, this file will explain the symbols used in the data to describe how and why.

The first part of the file names include the program, year and table the data comes from. At the end of the file name is the date and time the data was downloaded from the portal.

11.14.1 Importing downloaded data

When I import this data into R, I typically use the `read_csv()` function and skip the first, less-descriptive row. The second row becomes the headers, which are really long but explain the columns. I then rename them to something shorter. If you are only using selected columns, then you might use `select()` to get only those you need.

Here is an example:

```
tx_income <- read_csv("data-raw/ACSDT5Y2017.B19013_2019-04-20T000022/ACSDT5Y2017.B19013_data_with
  rename(
    median_income = `Estimate!!Median household income in the past 12 months (in 2017 inflation-a
    median_income_moe = `Margin of Error!!Median household income in the past 12 months (in 2017
  ) %>% clean_names()
```

Which yields this:

id	geographic_area_name	median_income	median_income_moe
0500000US48199	Hardin County, Texas	56131	3351
0500000US48207	Haskell County, Texas	43529	6157
0500000US48227	Howard County, Texas	50855	2162

11.14.2 Fields made to join with other data

Pay attention to fields named `id` or `geoid` or similar names as these are often fields meant to be joined to other tables. Many use parts of FIPS codes that define specific geographic areas, and allow you to match similar fields in multiple data sets.

This is especially important when it comes to mapping data, as these codes are how you join data to "shape files," which are a data representation of geographic shapes for mapping. While we won't go into a lot of detail about maps in this lesson, I've linked some examples below.

You may find you want to join data based on geography names, in which case you might need to use dplyr tools to split and normalize those terms so they match your other data set, like changing "Travis County, Texas" to just "Travis."

Chapter 12

Joining census data

In the last chapter we introduced U.S. Census Bureau's data.census.gov and used it to guide us in using their API with the tidycensus package. We used tidycensus to pull data and we mapped it.

In this chapter we'll use tidycensus again, but we'll use it to pull data that compliments other data (COVID cases), joining two data sets together to get a per-population rate.

12.1 Goals of the chapter

- Introduce the New York Times' collection of COVID-related data.
- Along the way we'll learn how to use `slice()` to get a subset of data.
- Use tidycensus to pull 2020 Decennial Census population data.
- Introduce the concept of "joining" data on a common key variable.
- Create COVID cases and death rates using the combination of these two data sets.
- Map the results.

12.2 Questions we'll answer

- Which counties in Texas have the highest/lowest cases-per-population rates for COVID.
- Which counties have the highest/lowest deaths-per-population rates for COVID.

12.3 Set up your notebook

We're going to use the same project from the last chapter, `yourname-census`, but start a new notebook.

1. Open your census project
2. Start a new R Notebook and name it `covid-rates.Rmd`.
3. Create a setup chunk with the following libraries

```
library(tidyverse)
library(tidycensus)
library(scales)
library(janitor)
```

12.4 New York Times COVID data

The New York Times has been collecting COVID-19 case data since the beginning of the pandemic, and they have been publishing it on Github for others to use. One of files they publish is cumulative case and death counts by county throughout the United States.

We will use this file, along with 2020 U.S. Census Bureau population data, to create cases-per-population and deaths-per-population rates for each county in Texas. Because counties have varying populations, we need a rate like this to compare them on an equal basis.

12.4.1 On your own: Import the NYT data

We won't bother to save the covid data to your hard drive ... we'll just import directly from the URL.

1. Create a new section and note you are download the NYT covid data
2. Start an R chunk and use `read_csv` to import the data from this url: `https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv` and save it into an R object called `covid`. (This is similar to how you imported the Skittles data.)
3. Print your new `covid` tibble to the screen. You can pipe it into `head()` to show just the top of it so we can see what it is.

There is more than 1.8 million rows of this data, so it might take a couple of seconds.

Let's tour the data real quick:

```
covid %>% head()
```

```
## # A tibble: 6 x 6
##   date       county    state      fips  cases deaths
##   <date>     <chr>     <chr>      <chr> <dbl>  <dbl>
## 1 2020-01-21 Snohomish Washington 53061     1      0
## 2 2020-01-22 Snohomish Washington 53061     1      0
## 3 2020-01-23 Snohomish Washington 53061     1      0
## 4 2020-01-24 Cook      Illinois   17031     1      0
## 5 2020-01-24 Snohomish Washington 53061     1      0
## 6 2020-01-25 Orange    California 06059     1      0
```

This is the same data that powers the NYT's Coronavirus tracker. They have a team of reporters that update the data every day.

There is a new row of data for each county, each day. The data includes the date of the update along with the county, state, a FIPS code (a special geographic designation) as well as the total cases and deaths from COVID up to that date.

Since the data is cumulative, we only want the most recent date for each county.

12.4.2 Slice the most recent data

Most of the concept in the next chunk you know already, but `slice()` is new so I'll do this all and then explain it.

1. Create a new section that notes you are getting the most recent Texas data by county.
2. Add the code chunk below (one line at a time!) and run it.

```
tx_covid <- covid %>%
  filter(state == "Texas") %>%
  group_by(county) %>%
  slice_max(date) %>% # our new function that gets the latest date
  filter(county != "Unknown") %>%
  select(-state)

tx_covid
```

```
## # A tibble: 254 x 5
## # Groups:   county [254]
##    date       county    fips  cases deaths
##    <date>     <chr>     <chr> <dbl>  <dbl>
##  1 2021-12-11 Anderson  48001  8149    208
```

```
##  2 2021-12-11 Andrews   48003  2963     65
##  3 2021-12-11 Angelina  48005 13920    422
##  4 2021-12-11 Aransas   48007  2655     67
##  5 2021-12-11 Archer    48009  1337     24
##  6 2021-12-11 Armstrong 48011   323      8
##  7 2021-12-11 Atascosa  48013  8838    209
##  8 2021-12-11 Austin    48015  3722     56
##  9 2021-12-11 Bailey    48017  1041     28
## 10 2021-12-11 Bandera   48019  2490     64
## # ... with 244 more rows
```

> IMPORTANT NOTE: The **date** values in your data should be more recent than what is shown here since you are downloading at a later date.

Let's break this down

- The first line creates a new R object `tx_covid` and fills it with the expression that follows, which is built from our raw `covid` data/
- The next line filters for Texas. Typical stuff for us.
- The next challenge is to find the most recent date for each county in Texas. There will be a row for each date for each county, and we want only the most recent date for each county. So, we do this by using `group_by(county)`so the next action will happen within the records for each county.
- `slice_max()` is our new function, which is related to the tidyverse `slice()` function, which lets you "subset" rows of data based on its position in your data. The documentation includes this: "`slice_min()` and `slice_max()` select rows with highest or lowest values of a variable." That is what we do here ... we use `slice_max()` to get the "highest" or latest `date` in our data. Since this follows `group_by(county)`, it finds the most recent date for *each* county.
- If you are running these one line at a time like I suggested, you'll notice there are 255 returns when there are *only* 254 counties in Texas. That is because there is a county value of **Unknown**, which we filter out using a standard filter.
- Lastly, we use `select()` to remove the `state` column since we don't need it anymore.

So now you have the latest COVID-19 case and death counts for each county in Texas.

12.5 Get county populations using tidycensus

We can't compare the counts of each county against each other because they have differing numbers of people. Let's put it like this using Oct. 15, 2021 numbers: Harris County – the state's largest county – has had 569,187 cases out of 4.7 million people. Loving County – the smallest county – has only 8 cases, but that is out of a total of 64 people in the county. Which is worse? And yes, only 64 people lived in Loving County when the 2020 census was taken on April 1, 2020.

So, let's use tidycensus to get the populations. Since the 2020 Decennial "Redistricting file" was released this year, we have really fresh and accurate population numbers.

12.5.1 Find the data on data.census.gov

As I said in the last chapter, the hardest part of using Census data is finding what you want. There is so much data it is hard to find or even know what you need.

From experience I know that in the first Decennial Redistricting data release, there is a **RACE** table that has a total population for each census geography. **This is the most accurate count we have of people in the U.S., but we only get it every 10 years.** You are just lucky to live at this time where this data is super fresh (relatively).

If you search for "Total Population" on http://data.census.gov you should get the **P1** table called **RACE** as the first return, but if not you can click here to get directly to it.

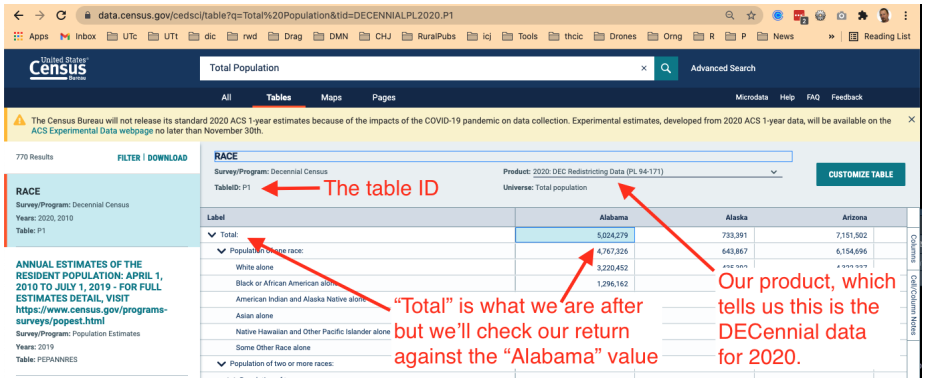


Figure 12.1: P1 table

12.5.2 Find the variable name

OK, now for the fun part. We can again use `load_variables()`, but the program we search for is `pl`.

> UNIMPORTANT ASIDE: That is an L in "pl," as in the PL 94-171, the law that requires the Census Bureau to produce this file. It's kinda confusing since we are also working with the P1 (one) table.

1. Start a new section and note you are getting the decennial population variable.
2. Run the code below and then open the table by double-clicking table icon for that object in your Environment pane.

```
pl_variables <- load_variables(2020, "pl", cache = TRUE)
```

This is how you can open the table:

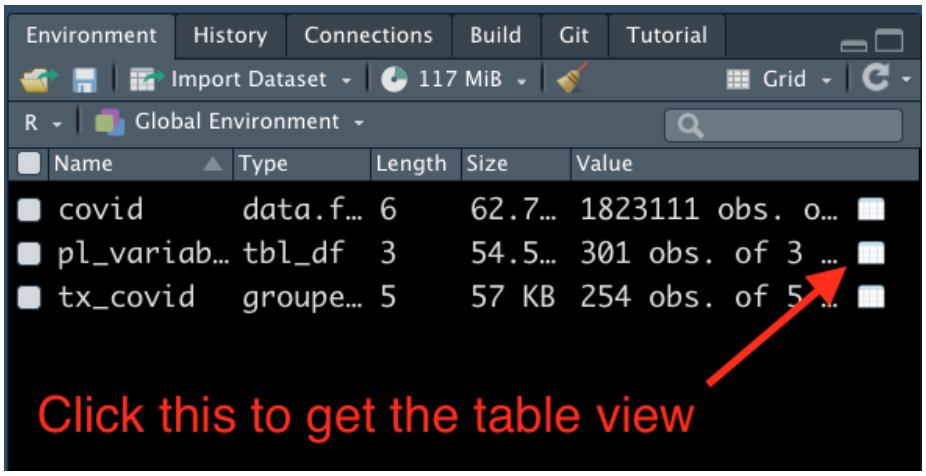


Figure 12.2: Open table

When we open that as a table view from your environment, we get lucky that we can see the value we are looking for: `P1_001N`.

1. Note the name of the variable in your markdown so you have it handy.

12.5.3 Fetch the population data

Getting data form the decennial census is similar to the ACS, just with a differently-named function: `get_decennial()`. You need at least the **year**, the **variables** and the **geography** for your return.

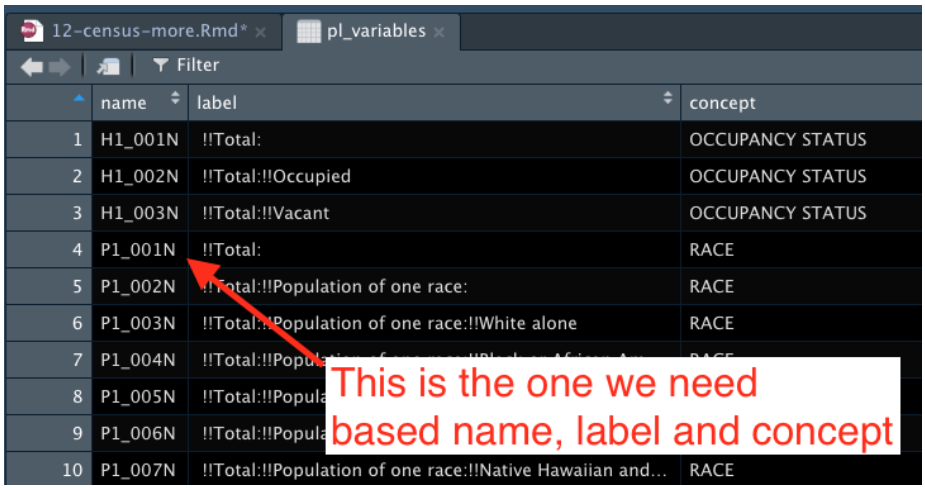


Figure 12.3: P1 value

Since it is Alabama that we see on the data.census.gov portal, we'll start by pulling state values to make sure our numbers match.

1. Start a subsection or otherwise note you are fetching a test of the variable. If you haven't already, you should include a link back to your table on data.census.gov.
2. Insert the code block below and run it. Might as well copy/paste all of it because you need all three arguments for it to work.

```
get_decennial(
  year = 2020,
  variables = "P1_001N",
  geography = "state"
)
```

```
## Getting data from the 2020 decennial Census

## Using the PL 94-171 Redistricting Data summary file

## Note: 2020 decennial Census data use differential privacy, a technique that
## introduces errors into data to preserve respondent confidentiality.
## i Small counts should be interpreted with caution.
## i See https://www.census.gov/library/fact-sheets/2021/protecting-the-confidentiality-of-the-20
## This message is displayed once per session.
```

```
## # A tibble: 52 x 4
##    GEOID NAME                  variable    value
##    <chr> <chr>                 <chr>       <dbl>
##  1 01    Alabama               P1_001N   5024279
##  2 02    Alaska                P1_001N    733391
##  3 04    Arizona               P1_001N   7151502
##  4 05    Arkansas              P1_001N   3011524
##  5 06    California            P1_001N  39538223
##  6 08    Colorado              P1_001N   5773714
##  7 09    Connecticut           P1_001N   3605944
##  8 10    Delaware              P1_001N    989948
##  9 11    District of Columbia  P1_001N    689545
## 10 16    Idaho                 P1_001N   1839106
## # ... with 42 more rows
```

Our return shows a value of 5024279 for Alabama, which matches the data portal, so we have the correct variable. Let's keep this here just for reference.

12.5.4 Get data for Texas counties

1. Start a new section and note you are getting the Texas county populations.
2. Build your data using the `get_decennial()` as noted below. Note you need the first three arguments before it wil work at all.

   I've suppressed the output here because it doesn't show well in the book.

```
tx_pop <- get_decennial(
    year = 2020,
    variables = "P1_001N",
    geography = "county",
    state = "TX",
    geometry = TRUE
  ) %>%
  clean_names() %>%
  arrange(name)

tx_pop
```

A couple of things to note here:

- We set the **geography** argument to "county" to get county results across the U.S.

- We use **state** argument set to "TX" to get just Texas counties.
- We use the `geometry = TRUE` argument to get our shapes for mapping.
- We pipe the result of all that into `clean_names()` because some of the column names are UPPERCASE. We don't *have* to do this, I'm just anal retentive.

12.6 About joins

OK, before we go further we need to talk about joining data.

There are several types of joins. We describe these as left vs right based on which table we reference first (which is the left one).

What joins do is match rows from multiple data sets that have the same "key value" and then append the data along those rows. Our two data sets have the same value in both, they just happend to be named different things.

Look at these two glimpses and note the **fips** and **geoid** fields between them.

```
tx_covid %>% glimpse()
```

```
## Rows: 254
## Columns: 5
## Groups: county [254]
## $ date   <date> 2021-12-11, 2021-12-11, 2021-12-11, 2021-12-11, 2021-12-11, 20~
## $ county <chr> "Anderson", "Andrews", "Angelina", "Aransas", "Archer", "Armstr~
## $ fips   <chr> "48001", "48003", "48005", "48007", "48009", "48011", "48013", ~
## $ cases  <dbl> 8149, 2963, 13920, 2655, 1337, 323, 8838, 3722, 1041, 2490, 146~
## $ deaths <dbl> 208, 65, 422, 67, 24, 8, 209, 56, 28, 64, 187, 24, 135, 745, 52~
```

```
tx_pop %>% glimpse()
```

```
## Rows: 254
## Columns: 5
## $ geoid    <chr> "48001", "48003", "48005", "48007", "48009", "48011", "48013"~
## $ name     <chr> "Anderson County, Texas", "Andrews County, Texas", "Angelina ~
## $ variable <chr> "P1_001N", "P1_001N", "P1_001N", "P1_001N", "P1_001N", "P1_00~
## $ value    <dbl> 57922, 18610, 86395, 23830, 8560, 1848, 48981, 30167, 6904, 2~
## $ geometry <MULTIPOLYGON [°]> MULTIPOLYGON (((-96.0648 31..., MULTIPOLYGON (((~
```

The `fips` and `geoid` columns are the same value under different names. This is the FIPS code for counties used in all kinds of government data. It's actually a combination of the **state** FIPS (48 for Texas) and the **county** designation (the

last three numbers). This creates a unique combination for every county in the U.S.

Knowing we have these two similar values, we can **Join** the rows that have the same value in both. Basically we will add the ***columns*** of one data set to the other based on that value.

In our case, we have the same 254 rows in each data set, but if we didn't we would need to decide what to do with data that DIDN'T match.

In the figure below, we can see which matching records are retained based on the type of join we use.

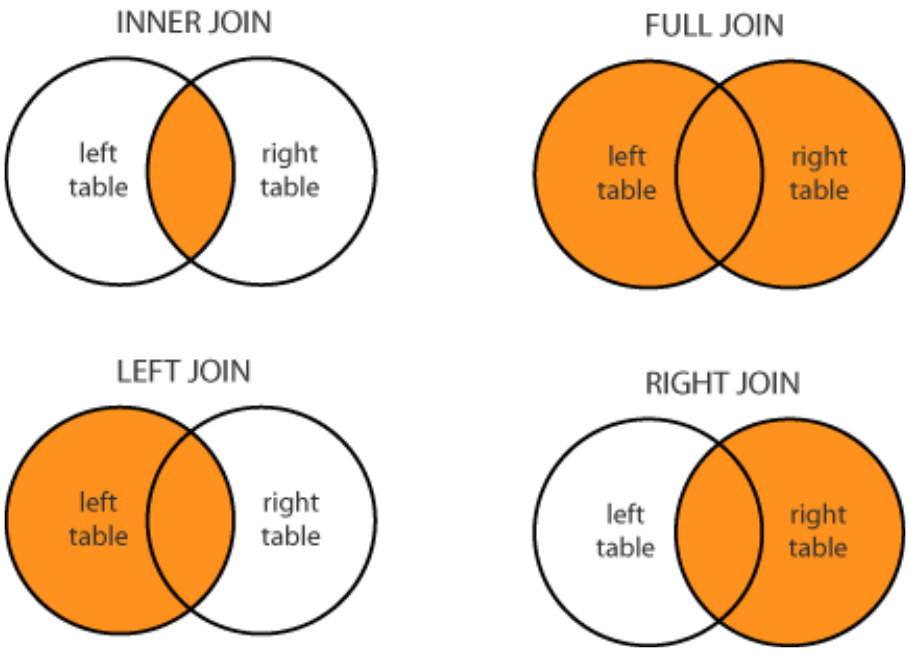


Figure 12.4: Types of joins

The join syntax works like this, with the * as the direction of the join:

```
new_dataframe <- *_join(first_df, second_df, by = field_name_to_join_on)
```

If the fields you are joining on are not named the same thing (as in our case), then you can define the relationships: `by = c("a" = "b")`.

For our purposes here we want to use an `inner_join()`. We have to **start with our population data first** for the mapping to work later. (This threw me for a loop when wrote this, so learn from my experience.) We'll then add our our covid data.

12.6.1 Join our data together

1. Start a new section and note we are joining our data.
2. Add the code below, run and review the results and the explanation below.

```
tx_joined <- inner_join(
  tx_pop, # first table
  tx_covid, # second table with columns we'll add
  by = c("geoid" = "fips") # note the joining columns
)

tx_joined
```

```
## Simple feature collection with 254 features and 8 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -106.6456 ymin: 25.83738 xmax: -93.50829 ymax: 36.5007
## Geodetic CRS:  NAD83
## # A tibble: 254 x 9
##    geoid name   variable value                  geometry date       county cases
##    <chr> <chr>  <chr>    <dbl>        <MULTIPOLYGON [°]> <date>     <chr>  <dbl>
##  1 48001 Ander~ P1_001N  57922 (((-96.0648 31.98066, -9~ 2021-12-11 Ander~  8149
##  2 48003 Andre~ P1_001N  18610 (((-103.0647 32.52219, -~ 2021-12-11 Andre~  2963
##  3 48005 Angel~ P1_001N  86395 (((-95.00488 31.42396, -~ 2021-12-11 Angel~ 13920
##  4 48007 Arans~ P1_001N  23830 (((-96.8229 28.16743, -9~ 2021-12-11 Arans~  2655
##  5 48009 Arche~ P1_001N   8560 (((-98.95382 33.49637, -~ 2021-12-11 Archer  1337
##  6 48011 Armst~ P1_001N   1848 (((-101.6294 34.75006, -~ 2021-12-11 Armst~   323
##  7 48013 Atasc~ P1_001N  48981 (((-98.80488 29.10702, -~ 2021-12-11 Atasc~  8838
##  8 48015 Austi~ P1_001N  30167 (((-96.62085 30.0452, -9~ 2021-12-11 Austin  3722
##  9 48017 Baile~ P1_001N   6904 (((-103.0469 33.8503, -1~ 2021-12-11 Bailey  1041
## 10 48019 Bande~ P1_001N  20851 (((-99.60332 29.74026, -~ 2021-12-11 Bande~  2490
## # ... with 244 more rows, and 1 more variable: deaths <dbl>
```

So, to break this down:

- Our new combined dataframe will be called `tx_joined`, which is filled from the result of . . .
- We start the `inner_join()` and then name our two tables.
- Since our "key" columns don't have the same name, we have to specify how to join them. We need to name these columns in the same order as we name the tables, hence `by = c("fips" = "geoid")`

Page through the result so you can see the new columns.

Here is glimpse of the new table:

```
tx_joined %>% glimpse()
```

```
## Rows: 254
## Columns: 9
## $ geoid    <chr> "48001", "48003", "48005", "48007", "48009", "48011", "48013"~
## $ name     <chr> "Anderson County, Texas", "Andrews County, Texas", "Angelina ~
## $ variable <chr> "P1_001N", "P1_001N", "P1_001N", "P1_001N", "P1_001N", "P1_00~
## $ value    <dbl> 57922, 18610, 86395, 23830, 8560, 1848, 48981, 30167, 6904, 2~
## $ geometry <MULTIPOLYGON [°]> MULTIPOLYGON (((-96.0648 31..., MULTIPOLYGON (((~
## $ date     <date> 2021-12-11, 2021-12-11, 2021-12-11, 2021-12-11, 2021-12-11, ~
## $ county   <chr> "Anderson", "Andrews", "Angelina", "Aransas", "Archer", "Arms~
## $ cases    <dbl> 8149, 2963, 13920, 2655, 1337, 323, 8838, 3722, 1041, 2490, 1~
## $ deaths   <dbl> 208, 65, 422, 67, 24, 8, 209, 56, 28, 64, 187, 24, 135, 745, ~
```

Now that we have our `cases`, `deaths` and `value` (or population) columns in the same table so we can do some math to create case and death rates.

12.6.2 Clean up column names, etc.

Let's rename our `value` column to a more descriptive `total_pop` and remove some other unneeded columns.

I don't think we've renamed columns using `rename()` yet so we'll do this together. Rename works similar to `mutate()` in that you name the new column first and then set it equal to the old one. We're also using `select()` which we've done before.

1. Create a new section noting you are renaming columns and other cleanup.
2. Add this chunk and run it.

```
tx_renamed <- tx_joined %>%
  rename(
    total_pop = value
  ) %>%
  select(-name, -variable)

tx_renamed %>% glimpse()
```

```
## Rows: 254
## Columns: 7
## $ geoid     <chr> "48001", "48003", "48005", "48007", "48009", "48011", "48013~
## $ total_pop <dbl> 57922, 18610, 86395, 23830, 8560, 1848, 48981, 30167, 6904, ~
## $ geometry  <MULTIPOLYGON [°]> MULTIPOLYGON (((-96.0648 31..., MULTIPOLYGON ((~
```

```
## $ date      <date> 2021-12-11, 2021-12-11, 2021-12-11, 2021-12-11, 2021-12-11,~
## $ county    <chr> "Anderson", "Andrews", "Angelina", "Aransas", "Archer", "Arm~
## $ cases     <dbl> 8149, 2963, 13920, 2655, 1337, 323, 8838, 3722, 1041, 2490, ~
## $ deaths    <dbl> 208, 65, 422, 67, 24, 8, 209, 56, 28, 64, 187, 24, 135, 745,~
```

12.7 Create our rate columns

Our whole goal here was to find the number of cases per population per county and we are finally at a place where we can. We do this using `mutate()` and we've done stuff like this in the past with the Military Surplus assignment.

We're actually going to find "cases per 1,000 people" because otherwise the numbers would be too small to comprehend. For a good explanation of why we need this, read Robert Niles' take on per capita rates. It is also covered on Page 16 of Numbers in the Newsroom by Sarah Cohen.

To get a per-1,000 rate, the math works like this: **(cases / (total_pop / 1000))**.

1. Start a new section noting you are creating your rates
2. Add the chunk and run it.

   I'm suppressing the output here because the book doesn't show it well.

```
tx_rates <- tx_renamed %>%
  mutate(
    cases_per_pop = (cases / (total_pop / 1000)) %>% round()
  )

tx_rates
```

You should be able to comprehend the `mutate()` function by now.

1. Go to the last column in the tibble that returns and make sure you have a new column named 'cases_perpop_ and that everything worked as you expected.

12.7.1 On your own: Get a deaths rate per 1,000

1. Edit your rates chunk to add an additional rule to calculate deaths per 1,000 people. It's pretty much the same but using `deaths`.

12.7.2 On your own: Find counties with highest rates

On your own, I want you to create **two** lists: One of the Top 10 counties with the highest cases-per-1,000 and another of the Top 10 counties with the highest deaths-per-1,000 cases. Don't overthink this ... it is a simple arrange, select and head combination. You **do not** need to save these into new object ... just print to the screen.

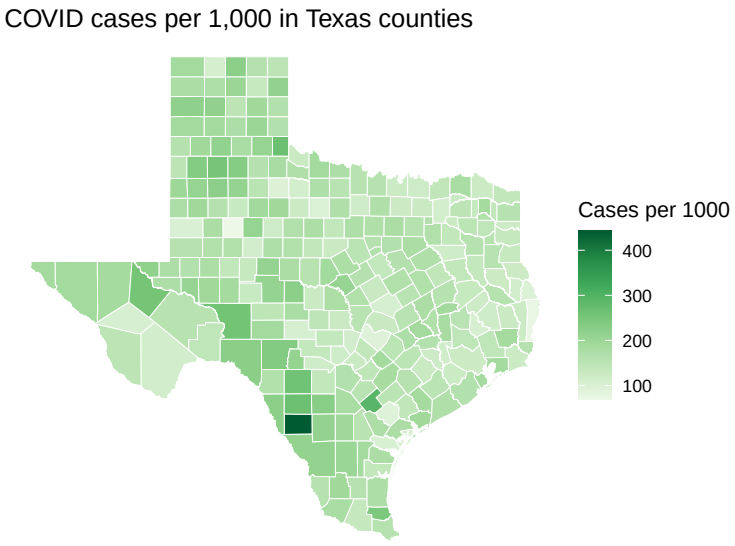
Use `select()` to show just the county, total_pop, cases/deaths and the appropriate rate for each view. **Note that the `geometry` column will *always* show**. You can't remove it ... that's just how it works.

12.8 Mapping the rates for Texas counties

We'll build a Texas map of the rates much like we did with the broadband rates in the last chapter. In fact it is pretty much the same code with updated variables and text.

1. Create a new section that you will map cases per 1,000
2. Add the chunk below and run it.
3. Update the labs to add a title, etc.

```
tx_rates %>%
  ggplot() +
  geom_sf(aes(fill = cases_per_pop), color = "white", size = .2) +
  theme_void() +
  labs(
    title = "COVID cases per 1,000 in Texas counties",
    caption = "Source: New York Times COVID, Census Bureau/2020 Decennial"
  ) +
  scale_fill_distiller(
    palette = "Greens",
    direction = 1,
    name = "Cases per 1000"
  ) +
  theme(plot.margin = unit(c(10,0,10,0), "pt"))
```



12.9 On your own: Map the deaths per 1,000

On your own, create a map like above but:

1. Use your deaths per 1,000 data.
2. Change the color
3. Update the titles, etc to be correct.

12.10 What we've learned

- We learned that the New York Times is awesome that they have collected COVID data and then shared it with the world.
- We used `group_by()` and `slice_max()` to get the most recent data for each county.
- We used tidycensus to get population data and geometry data for mapping.
- You made more maps.

12.11 Turn in your work

(There may be an extra-credit assignment that you can try if I can work it up in time. You should know by class time.)

1. Make sure your Markdown is in good shape, chunks named, etc.
2. Make sure everything runs and knits all the way through.
3. Zip your project and turn it into Canvas.

Chapter 13

Mastery - Mixed Beverages

With this assignment you will be given a dataset, recorded interviews and some story ideas and you will build a full story based on the collection.

13.1 The assignment outline

First, let's talk about the deliverables:

- You'll produce an 500-word data drop based on the data. This should be written as a news story like most of our other data drops. You need three or four "facts" from the data to build a good case, and then support those with quotes. Don't pad your story with an exposé on the drinking culture in Austin, just find more data facts and quotes to build a better story. (100 points)
- You'll turn in any analysis work you've done (your R project), regardless if it is used in the story. I want ALL of your work. (100 points)
- You'll produce at least one publishable chart to go with your story using ggplot or Datawrapper. This chart needs to have a proper headline, description, legend, annotations and such so that the chart can be understood outside the context of the story. Include an image and/or link in your story. (100 points)
- You will be assigned an editing partner from the class to work with. You will each do your own story (and must have different angles), but the idea is you have someone to regularly talk with about your analysis, edit your story and proof-read your charts. At the end of the first week you have an assignment where you tell me your editing partner and angle they are looking at, and then another later where you discuss how you helped each other.

13.2 About the interviews

I have two recorded interviews for you to work from, and the **links are in Canvas** on the main assignment called "Mixed Beverage project." Both interviews include discussion of alcohol sales, including dealing with the pandemic. When these were recorded in spring 2021 we were also using TABC violations data, so the interviews also cover selling to minors and such. Those parts aren't really germane to this assignment, but there is plenty to draw from.

- **Andy Kahn** is a bartender at The Mockingbird near campus. He worked previously as a manager at The Hole in the Wall.
- **Chad Womack** is the owner of The Dogwood, which has locations on W. 6th and in The Domain (as well as other cities). He co-owns them with his brother Brad (yes, that Brad Womack) and Jason Carrier through Carmack Concepts.

It is worth going through the interviews early on so you can tailor your analysis around what they talk about.

13.3 About the data

You'll be drawing from at least five years of data from the Mixed Beverage Gross Receipts for locations with an Austin local address. You should take a look at that page while you familiarize yourself with the data, which also serves as your data dictionary.

Each month, every Texas establishment that sells liquor-by-the-drink (restaurants, bars, stadiums, etc) has to report their sales to the state for tax purposes.

13.3.1 Important things to know

Each row of the data is the amount of total money brought in ***each month*** by an establishment, based on the Obligation End Date, which is always the last day of the reporting month. This means you can see trends by month, but not any time period smaller than that.

There are four values of money reported: Total Receipts, Wine Receipts, Beer Receipts, Liquor Receipts and Cover Charge Receipts. The beer, wine, liquor and cover charge values *should* add up to the Total Receipts, though I've found that is not always true. I would **stay away from any analysis on Cover Charges** as there must be some special rules of when they need to be reported that would need investigation to understand. The amounts are sales amounts (i.e., total amount of money bought in for that month for that category) and **not** profits or number of drinks sold.

Some other things to know:

- Sometimes names of taxpayers and locations are obscured. Hotel chains may have a company that serves all their hotels. A beverage company may hold license to sell in multiple locations. It is important to group by both Location Name AND Location Address to find specific locations. You can google the address to find out the real name of an establishment.
- Also be aware that some companies have more than one location with the same name, so again you should group by both Location Name and Location Address to get totals specific locations.
- Each Taxpayer Name has only one Taxpayer Number, but that company could own many establishments in many locations.
- For the most part we can ignore Responsibility Begin and End dates if you always group by Location Name and Location Address when looking at single locations. This is used when an establishment changes hands and has different owners within the same month.

13.4 Story ideas

Here is a list of questions you could ask that can lead to story ideas.

- Which establishments have sold the most alcohol over the past five years? How does that look on a monthly basis? (Be sure to watch for multiple locations of the same name and name changes of the same location.)
  - Perhaps take the same idea but for a smaller geographic area. The Drag (Guadalupe between 19th and 30th). West Campus (78705). The Domain area (78758). Downtown (78701). South Austin (78704). East Austin (78702) and north of there (78722). One thing to be aware of here is that ZIPs cross over city boundaries
  - You could compare sales between popular areas. Or sales over time in a certain area.
- How did COVID-19 affect sales in 2020 and how are they rebounding in 2021?
- Is the number of restaurants or bars growing in certain areas? Like in east or south Austin? Again, ZIP codes might be a good geography to hone in on.
- How do sales break down by alcohol type? Who sold the most beer last year? Wine? Again, could break down by geography.
- How do sales change over the seasons around West Campus? What do establishments do to deal with that?
- How much does March mean to sales of alcohol downtown during SXSW? (Like what percentage of their yearly sales?) What was the difference in 2020 and 2021 compared to 2019?

- What owner/company has sold the most alcohol? (Taxpayer name). Or who owns the most locations in the city?

There are certainly other ideas, but key are the fact you have date ranges, some geographic locations like addresses and ZIP codes and the beer/wine/liquor/total sales values to work from.

13.5 Downloading and cleaning the data

We'll get our data directly from the data.texas.gov portal using the Socrata API. You'll need to install the RSocrata package first, but you don't need an API key for this.

1. Go ahead and set up a new project. You'll have a separate import-cleaning notebook from your analysis notebook. Use good naming practices.
2. Install the RSocrata package in your console: `install.packages("RSocrata")`.
3. You'll need the following libraries:

```
library(tidyverse)
library(janitor)
library(RSocrata)
```

13.5.1 Set up the download url

The key to using RSocrata and the Socrata API is to build a URL that will get us the data we want. I figured this out by reading through their documentation but that is beyond the scope of this lesson. But I will explain it, more or less.

For this first part we are just building a flexible way to filter our data before downloading. I'll explain below.

```
mixbev_base_url = 'https://data.texas.gov/resource/fp9t-htqh.json?'
start_date = '2016-01-31'
end_date = '2021-08-31'
city = 'AUSTIN'

download_url <-  paste(
  mixbev_base_url,
  "$limit=100&", # comment out this line in your notebook
  "$where=obligation_end_date_yyyymmdd%20between%20",
  "'", start_date, "'",
  " and ",
```

```
  "'", end_date, "'",
  "&location_city=",
  "'", city, "'",
  sep = ""
)

download_url
```

```
## [1] "https://data.texas.gov/resource/fp9t-htqh.json?$limit=100&$where=obligation_end_date_yyyy
```

- The first several lines are creating variables for the base URL and dates ranges for the data. If we wanted data from a different date range or city, we could change those. It just provides flexibility and ease for updates.
- One thing about those dates: We are filtering the data to include five full years plus valid months in 2021. Since reporting lags on this by as much as two months, we don't have full reports for September or October 2021 so we are excluding them.
- The `download_url` object pieces together the parts of the url that we need to download the data. This url is an "endpoint" for the Socrata API for this dataset. It uses SoQL queries to select only the data we want. The `paste()` function is just putting together the pieces of the URL endpoint based on our variables. I print it out at the end so you can see what the finished URL looks like. You could copy/paste that endpoint (between the " ") into a browser to see what the data looks like.
- I have a `limit` line in here for testing. I just pull 1000 lines of the data instead of all of them. **You need to comment out this line in your notebook to get all the data.**

  REALLY, REALLY IMPORTANT NOTE: Hey, did you see that note about the `"$limit=100&"` line? You **must** comment out the limit to get all the data you need.

This example above could be repurposed to pull data from a different dataset on any agency's Socrata platform.

13.5.2 Download the data

Now that we have the url we can download the data into an R object. **This can take 30 seconds or more** as it a decent amount of data, about 75k rows. (Though I only show 100 here. Remember to comment that `limit` line out when you are ready to pull **all** of the data).

```
receipts_api <- read.socrata(download_url)

# look at the data
receipts_api %>% glimpse()
```

```
## Rows: 100
## Columns: 24
## $ taxpayer_number                      <chr> "32039395531", "12630963622", "174~
## $ taxpayer_name                        <chr> "CORNER WEST, LLC", "MAC ACQUISITI~
## $ taxpayer_address                     <chr> "715 W 6TH ST", "1855 BLAKE ST STE~
## $ taxpayer_city                        <chr> "AUSTIN", "DENVER", "MERRILLVILLE"~
## $ taxpayer_state                       <chr> "TX", "CO", "IN", "FL", "TX", "TX"~
## $ taxpayer_zip                         <chr> "78701", "80202", "46410", "32837"~
## $ taxpayer_county                      <chr> "227", "0", "0", "0", "227", "227"~
## $ location_number                      <chr> "1", "7", "39", "5", "3", "3", "2"~
## $ location_name                        <chr> "THE DOGWOOD", "ROMANO'S MACARONI ~
## $ location_address                     <chr> "715 W 6TH ST", "701 E STASSNEY LN~
## $ location_city                        <chr> "AUSTIN", "AUSTIN", "AUSTIN", "AUS~
## $ location_state                       <chr> "TX", "TX", "TX", "TX", "TX", "TX"~
## $ location_zip                         <chr> "78701", "78745", "78701", "78758"~
## $ location_county                      <chr> "227", "227", "227", "227", "227",~
## $ inside_outside_city_limits_code_y_n <chr> "Y", "Y", "Y", "Y", "Y", "Y", "Y",~
## $ tabc_permit_number                   <chr> "MB750447", "MB710753", "MB913070"~
## $ responsibility_begin_date_yyyymmdd  <dttm> 2010-06-15, 2008-12-18, 2015-07-0~
## $ obligation_end_date_yyyymmdd         <dttm> 2016-04-30, 2016-10-31, 2018-02-2~
## $ liquor_receipts                      <chr> "237013", "2658", "63669", "73418"~
## $ wine_receipts                        <chr> "5497", "8537", "41558", "20379", ~
## $ beer_receipts                        <chr> "83765", "1309", "20078", "105510"~
## $ cover_charge_receipts                <chr> "0", "0", "0", "0", "0", "0", "0",~
## $ total_receipts                       <chr> "326275", "12504", "125305", "1993~
## $ responsibility_end_date_yyyymmdd     <dttm> NA, 2018-02-14, NA, NA, NA, NA, N~
```

13.5.3 Fix some values

If you look at the data types of these columns you'll find that the `_receipts` columns are `<chr>` (which means characters) instead of `<dbl>` (which is numbers). We can't do math with text, so we need to fix that.

I'm going to give you an assist here because I found a pretty cool way to do this that I want to share.

Tidyverse has a function called type_convert() that helps change between text and number data types and it can help us here. It allows us to specify data types for specific columns.

1. Start a new section and note you are fixing the receipts columns.
2. Add this code block and run it.

```
receipts_converted <- receipts_api %>%
  type_convert(
    cols(
        .default = col_character(), # sets a default of character unless specified below
        liquor_receipts = col_double(),
        wine_receipts = col_double(),
        beer_receipts = col_double(),
        cover_charge_receipts = col_double(),
        total_receipts = col_double()
    )
  )

receipts_converted %>% glimpse()
```

```
## Rows: 100
## Columns: 24
## $ taxpayer_number                    <chr> "32039395531", "12630963622", "174~
## $ taxpayer_name                      <chr> "CORNER WEST, LLC", "MAC ACQUISITI~
## $ taxpayer_address                   <chr> "715 W 6TH ST", "1855 BLAKE ST STE~
## $ taxpayer_city                      <chr> "AUSTIN", "DENVER", "MERRILLVILLE"~
## $ taxpayer_state                     <chr> "TX", "CO", "IN", "FL", "TX", "TX"~
## $ taxpayer_zip                       <chr> "78701", "80202", "46410", "32837"~
## $ taxpayer_county                    <chr> "227", "0", "0", "0", "227", "227"~
## $ location_number                    <chr> "1", "7", "39", "5", "3", "3", "2"~
## $ location_name                      <chr> "THE DOGWOOD", "ROMANO'S MACARONI ~
## $ location_address                   <chr> "715 W 6TH ST", "701 E STASSNEY LN~
## $ location_city                      <chr> "AUSTIN", "AUSTIN", "AUSTIN", "AUS~
## $ location_state                     <chr> "TX", "TX", "TX", "TX", "TX", "TX"~
## $ location_zip                       <chr> "78701", "78745", "78701", "78758"~
## $ location_county                    <chr> "227", "227", "227", "227", "227",~
## $ inside_outside_city_limits_code_y_n <chr> "Y", "Y", "Y", "Y", "Y", "Y", "Y",~
## $ tabc_permit_number                 <chr> "MB750447", "MB710753", "MB913070"~
## $ responsibility_begin_date_yyyymmdd <dttm> 2010-06-15, 2008-12-18, 2015-07-0~
## $ obligation_end_date_yyyymmdd       <dttm> 2016-04-30, 2016-10-31, 2018-02-2~
## $ liquor_receipts                    <dbl> 237013, 2658, 63669, 73418, 203774~
## $ wine_receipts                      <dbl> 5497, 8537, 41558, 20379, 6527, 26~
## $ beer_receipts                      <dbl> 83765, 1309, 20078, 105510, 21446,~
## $ cover_charge_receipts              <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ total_receipts                     <dbl> 326275, 12504, 125305, 199307, 231~
## $ responsibility_end_date_yyyymmdd   <dttm> NA, 2018-02-14, NA, NA, NA, NA, N~
```

Let's break this down

- We start with a new R Object to fill (and a glimpse of it at the end).
- We take `receipts_api` and pipe into `type_convert()` and specify the `cols()` function there.
- The first line inside cols is `.default = col_character()` which tells type_convert to only change columns we specify after it. If we didn't include this then the `*_zip` and `*_county` columns would be converted to numbers and we don't want that. Especially the ZIP code, because that is NOT a number.
- The rest of the lines set the receipts columns to `col_double()` to make them numbers.

We could've done this using `mutate()` with `as.numeric()` and overwriting the columns, but I like this method and wanted to show it to you.

13.6 Export your data

This tibble `receipts_converted` is ready to be exported as an .rds file to use in your analysis notebook. I'll leave that up to you ;-).

13.7 How to tackle the analysis

The specifics will depend on what you are trying to learn from the data, but check out the next chapter "How to interview your data" for some general tips and concept reviews.

Chapter 14

How to interview a new dataset

For those unfamiliar with exploring data, starting the process can be paralyzing.

How do I explore when I don't know what I'm looking for? Where do I start?

Every situation is different, but there are some common techniques and some common sense that you can bring to every project.

14.1 Start by listing questions

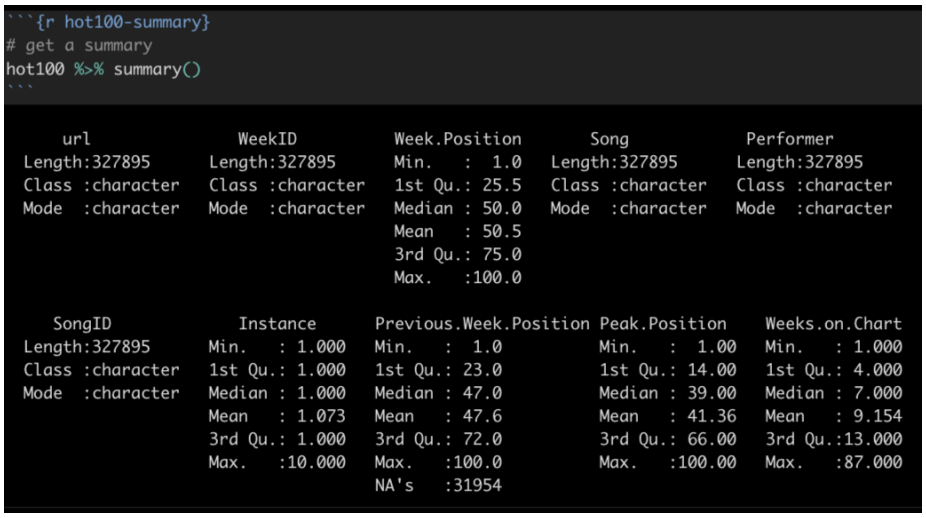
It's likely you've acquired data because you needed it to add context to a story or situation. Spend a little time at the beginning brainstorming as list of questions you want to answer. (You might ask a colleague to participate: the act of describing the data set will reveal questions for both of you.) I like to start my RNotebook with this list.

14.2 Understand your data

Before you start working on your data, make sure you understand what all the columns and values mean. Look at your data dictionary, or talk to the data owner to make sure you understand what you are working with.

To get a quick summary of all the values, you can use a function called `summary()` to give you some basic stats for all your data. Here is an example from the Billboard Hot 100 data we used in a class assignment.

A `summary()` will show you the data type for each column, and then for number values it will show you the min, max, median, mean and other stats.



```
```{r hot100-summary}
# get a summary
hot100 %>% summary()
```

      url   WeekID   Week.Position   Song   Performer
 Length:327895   Length:327895   Min.   : 1.0   Length:327895   Length:327895
 Class :character   Class :character   1st Qu.: 25.5   Class :character   Class :character
 Mode :character   Mode :character   Median : 50.0   Mode :character   Mode :character
                                       Mean   : 50.5
                                       3rd Qu.: 75.0
                                       Max.   :100.0

    SongID   Instance   Previous.Week.Position Peak.Position   Weeks.on.Chart
 Length:327895   Min.   : 1.000   Min.   : 1.0   Min.   : 1.00   Min.   : 1.000
 Class :character   1st Qu.: 1.000   1st Qu.: 23.0   1st Qu.: 14.00   1st Qu.: 4.000
 Mode :character   Median : 1.000   Median : 47.0   Median : 39.00   Median : 7.000
                    Mean   : 1.073   Mean   : 47.6   Mean   : 41.36   Mean   : 9.154
                    3rd Qu.: 1.000   3rd Qu.: 72.0   3rd Qu.: 66.00   3rd Qu.:13.000
                    Max.   :10.000   Max.   :100.0   Max.   :100.00   Max.   :87.000
                                     NA's   :31954
```

Figure 14.1: Summary of billboard data

14.3 Pay attention to the shape of your data

Is your data long or wide?

Wide data adds new observations as columns, with the headers describing the observation. Official reports and Excel files from agencies are often in this format:

Country	2018	2017
United States	20,494,050	19,390,604
China	13,407,398	12,237,700

Long data is where each row in the data is a single observation, and each column is an attribute that describes that observation. Data-centric languages and applications like R and Tableau typically prefer this format.

Country	Year	GDP
United States	2018	20,494,050
United States	2017	19,390,604
China	2018	13,407,398
China	2017	12,237,700

The shape of the data will determine how you go about analyzing it. They are both useful in different ways. Wide data allows you to calculate columns

to show changes. Visualization programs will sometimes want a long format to more easily categorize values based on the attributes.

You can pivot your data with `pivot_longer()` and `pivot_wider` to change the shape of your data.

14.4 Counting and aggregation

A large part of data analysis is counting and sorting, or filtering and then counting and sorting. Depending on the program you are using you may approach it differently but think of these concepts:

14.4.1 Counting rows based on a column

If you are just counting the number of rows based on the values within a column (or columns), then `count()` is the key. When you use `count()` like this, a new column called `n` is created to hold the count of the rows. You can rename `n` with the `name = "new_name"` argument, and you can change the sorting to descending order using the `sort = TRUE` argument.

In this example, we are counting the number of rows for each princess in our survey data, the arranging them in descending order.

```
survey %>%
  count(princess, name = "votes", sort = TRUE)
```

princess	votes
Mulan	14
Rapunzel (Tangled)	7
Jasmine (Aladdin)	6
Ariel (Little Mermaid)	5
Tiana (Princess and the Frog)	2
Aurora (Sleeping Beauty)	1
Belle (Beauty and the Beast)	1
Merida (Brave)	1
Snow White	1

14.4.2 Sum, mean and other aggregations

If you want to aggregate values in a column, like adding together values, or to find a mean or median, then you will want to use the GSA combination: `group_by()` on your columns of interest, then use `summarize()` to aggregate

the data in the manner you choose, like `sum()`, `mean()` or the number of rows `n()`. You can then use `arrange()` to order the result however you want.

Here is an example where we use `group_by` and `summarize()` to add together values in our mixed beverage data. In this case, we had multiple rows for each name/address group, but we wanted to add together `total_receipts()` for each group.

```
receipts %>%
  group_by(location_name, location_address) %>%
  summarize(
    total_sales = sum(total_receipts)
  ) %>%
  arrange(desc(total_sales))
```

location_name	location_address	total_sales
WLS BEVERAGE CO	110 E 2ND ST	35878211
RYAN SANDERS SPORTS	9201 CIRCUIT OF THE AMERICAS BLVD	20714630
W HOTEL AUSTIN	200 LAVACA ST	15435458
ROSE ROOM/ 77 DEGREE	11500 ROCK ROSE AVE	14726420
THE DOGWOOD DOMAIN	11420 ROCK ROSE AVE STE 700	14231072

The result will have all the columns you included in the group, plus the columns you create in your summarize statement. You can summarize more than one thing at a time, like the number of rows `numb_rows = n()` and average of the values `average = mean(column_name)`.

14.4.3 Creating columns to show difference

Sometimes you need to perform math on two columns to show the difference between them. Use `mutate()` to create the column and do the math. Here's a pseudo-code example:

```
new_or_reassigned_df <- df %>%
  mutate(
    new_col_name = (part_col / total_col) * 100
  )
```

14.5 Cleaning up categorical data

If you are going to count our summarize rows based on categorical data, you should make sure the values in that column are clean and free of typos and values that might better be combined.

Some strategies you might use:

- Create a `count()` of the column to show all the different values and how often they show up.
- You might want to use `mutate()` to create a new column and then update the values there. Or you might use `recode()` the set specific values to new values.

If you find you have hundreds of values to clean, then come see me. There are some other tools like OpenRefine that you can learn fairly quickly to help.

14.6 Time as a variable

If you have dates in your data, then you almost always want to see change over time for different variables.

- Summarize records by year or month as appropriate and create a Bar or Column chart to show how the number of records for each time period.
- Do you need to see how different categories of data have changed over time? Consider a line chart that shows those categories in different colors.
- If you have the same value for different time periods, do might want to see the change or percent change in those values. You can create a new column using `mutate()` to do the math and show the difference.

14.7 Explore the distributions in your data

We didn't talk about histograms in class, but sometimes you might want see the "distribution" of values in your data, i.e. how the values vary within the column. Are many of the values similar? A histogram can show this.

Here is an example of a histogram use wells data exploring the borehole_depth (how deep the well is). Each bar represents the number of wells broken down in 100ft depth increments (set with `binwidth=100`). So the first bar shows that most of the wells (more than 7000) are less than 100 feet deep.

```
wells %>%
  ggplot(aes(x = borehole_depth)) +
  geom_histogram(binwidth = 100)
```

While there are wells deeper than 1000 feet, they are so few they don't even show on the graphic.

You'll rarely use a histogram as a graphic with a story because they are more difficult to explain to readers. But they do help you to understand how much values differ within a column.

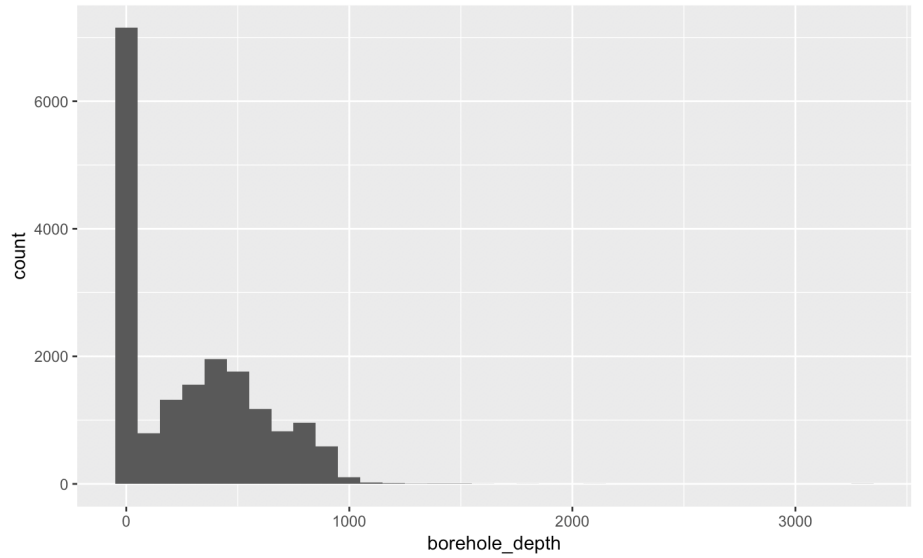


Figure 14.2: Borehole depth histogram

14.7.1 More on histograms

If you google around, you might see other ways to create a histogram, including `hist()` and `qplot()`. You might stick with the ggplot's `geom_histogram()` since you already are familiar with the syntax.

- Tutorial on histograms using ggplot from DataCamp.
- R Cookbook on histograms.

14.8 Same ideas using spreadsheets

Check out this resource by David Eads on the same topic, with some more specifics about Google Sheets.

Chapter 15

Verbs

NEEDS REVIEW

- Needs `pivot_longer()` and `pivot_wider()` instead of gather and spread.

An opinionated list of the most common Tidyverse and other R verbs used with data storytelling.

15.1 Import/Export

- `read_csv()` imports data from a CSV file. (It handles data types better than the base R `read.csv()`). Also `write_csv()` when you need export as CSV. **Example:** `read_csv("path/to/file.csv")`.
- `write_rds` to save a data frame as an `.rds` R data data file. This preserves all the data types. `read_rds()` to import R data. **Example:** `read_rds("path/to/file.rds")`.
- `readxl` is a package we didn't talk about, but it has read_excel() that allows you to import from an Excel file, including specified sheets.
- clean_names() from the `library(janitor)` package standardizes column names.

15.2 Data manipulation

- `select()` to select columns. **Example:** `select(col01, col02)` or `select(-excluded_col)`.

- `rename()` to rename a column. **Example:** `rename(new_name = old_name)`.
- `filter()` to filter rows of data. **Example:** `filter(column_name == "value")`.
  - See Relational Operators like `==`, `>`, `>=` etc.
  - See Logical operators like `&`, `|` etc.
  - See is.na tests if a value is missing.
- `distinct()` will filter rows down to the unique values of the columns given.
- `arrange()` sorts data based on values in a column. Use `desc()` to reverse the order. **Example:** `arrange(col_name %>% desc())`
- `mutate()` changes and existing column or creates a new one. **Example:** `mutate(new_col = (col01 / col02))`.
- `gather()` collapses columns into two, one a key and the other a value. Turns wide data into long. **Example:** `gather(key = "new_key_col_name", value = "new_val_col_name", 3:5)` will gather columns 3 through 5. Can also name columns to gather.
- `spread()` turns long data into wide by spreading into multiple columns based on as key.

15.3 Aggregation

- `count()` will count the number rows based on columns you feed it.
- `group_by()` and `summarize()` often come together. When you use `group_by()`, every function after it is broken down by that grouping. **Example:** `group_by(song, artist) %>% summarize(weeks = n(), top_chart_position = min(peak_position))`

15.4 Math

These are the function often used within `summarize()`:

- `n()` to count the number of rows. `n_distinct()` counts the unique values.
- `sum()` to add things together.
- `mean()` to get an average.
- `median()` to get the median.
- `min()` to get the smallest value. `max()` for the largest.
- `+`, `-`, `*`, `/` are math operators similar to a calculator.