

# **Reporting with Data in R**

Christian McDonald & Jo Lukito

2023-12-20

# Table of contents

<b>About this book</b>	<b>12</b>
Some words from Prof. McDonald . . . . .	12
Conventions and styles in this book . . . . .	13
Things to do . . . . .	13
Code blocks . . . . .	14
Notes, some important . . . . .	16
About the authors . . . . .	16
Christian McDonald . . . . .	16
Jo Lukito . . . . .	16
License . . . . .	17
Other resources . . . . .	17
<b>1 Install Party</b>	<b>18</b>
1.1 Mac vs PC . . . . .	18
1.2 RStudio vs posit.cloud . . . . .	18
1.3 Installing R . . . . .	19
1.4 Install Quarto . . . . .	19
1.5 Installing RStudio . . . . .	19
1.5.1 Getting “Git” errors on Macs . . . . .	20
1.6 Class project folder . . . . .	20
<b>2 Introduction to R</b>	<b>21</b>
2.1 About Quarto, R and scripted journalism . . . . .	21
2.2 RStudio tour . . . . .	23
2.3 Updating preferences . . . . .	24
2.4 The R Package environment . . . . .	27
2.4.1 How we use packages . . . . .	27
2.4.2 Install some packages . . . . .	27
2.5 Starting a new Quarto Website . . . . .	29
2.5.1 The files pane . . . . .	31
2.6 The Quarto document . . . . .	32
2.6.1 Render the document . . . . .	33
2.6.2 The metadata . . . . .	33
2.6.3 R code chunks . . . . .	34

2.7	Let's do some data analysis . . . . .	35
2.7.1	Adding new code chunks . . . . .	36
2.7.2	Practice adding code chunks . . . . .	37
2.7.3	The toolbar . . . . .	38
2.8	A quick look back at files . . . . .	38
2.9	Publish to quarto.pub . . . . .	39
2.10	On your own: Update the About page . . . . .	40
2.11	Turning in our projects . . . . .	40
2.11.1	Exporting for posit.cloud . . . . .	40
2.12	Review of what we've learned so far . . . . .	41
<b>I</b>	<b>Counting</b>	<b>42</b>
<b>3</b>	<b>Billboard Cleaning</b>	<b>43</b>
3.1	Learning goals of this lesson . . . . .	43
3.2	Basic steps of this lesson . . . . .	43
3.3	Create a new project . . . . .	44
3.3.1	Update index with project description . . . . .	44
3.3.2	Create directories for your data . . . . .	45
3.4	Create our cleaning notebook . . . . .	46
3.5	Update our <code>_quarto.yml</code> . . . . .	46
3.5.1	Describe the goals of the notebook . . . . .	48
3.6	The setup chunk . . . . .	48
3.6.1	Load the libraries . . . . .	49
3.6.2	About the libraries . . . . .	49
3.6.3	About the options . . . . .	50
3.7	About the Billboard Hot 100 . . . . .	50
3.7.1	Data dictionary . . . . .	51
3.7.2	Let's download our data . . . . .	51
3.7.3	Comment the download code . . . . .	52
3.8	Import the data . . . . .	53
3.8.1	Assign our import to an R object . . . . .	55
3.8.2	Print a peek to your R Notebook . . . . .	56
3.8.3	Glimpse the data . . . . .	57
3.8.4	About the pipe <code> &gt;</code> . . . . .	57
	A rabbit dives into a pipe . . . . .	58
3.9	Cleaning data . . . . .	58
3.9.1	Cleaning column names . . . . .	59
3.9.2	Fixing the date . . . . .	60
3.9.3	Arrange the data . . . . .	63
3.9.4	Selecting columns . . . . .	66

3.10 Exporting data . . . . .	67
3.10.1 Using multiple notebooks . . . . .	67
3.10.2 About paths . . . . .	69
3.11 Bookmarks . . . . .	69
3.12 Render and publish . . . . .	71
3.12.1 Publish to Quarto pub . . . . .	72
3.13 Review of what we've learned so far . . . . .	72
3.14 What's next . . . . .	73
<b>4 Billboard Analysis</b>	<b>74</b>
4.1 Goals of this lesson . . . . .	74
4.2 The questions we'll answer . . . . .	74
4.3 Setting up an analysis notebook . . . . .	75
4.3.1 Add your goals, setup . . . . .	75
4.3.2 Import the cleaned data . . . . .	76
4.4 Introducing dplyr . . . . .	78
4.5 Most appearances . . . . .	79
4.5.1 Group & Aggregate . . . . .	80
4.5.2 Summarize . . . . .	80
4.5.3 Group by . . . . .	82
4.5.4 Arrange the results . . . . .	83
4.5.5 Get the top of the list . . . . .	84
4.5.6 Data Takeaways: Describing your learnings . . . . .	85
4.5.7 Render your notebook . . . . .	86
4.6 Song with most appearances . . . . .	86
4.6.1 Introducing filter() . . . . .	88
4.6.2 Data Takeaway: Song appearances . . . . .	90
4.6.3 Render your second quest . . . . .	90
4.7 Song the longest at No. 1 . . . . .	90
4.7.1 Data Takeaway: Longest at No. 1 . . . . .	94
4.8 Performer with most No. 1 singles . . . . .	94
4.8.1 Using distinct() . . . . .	95
4.8.2 Summarize the performers . . . . .	96
4.8.3 Filter for a good cutoff . . . . .	96
4.8.4 The perils of collaborations . . . . .	97
4.8.5 Data Takeaway: Most No. 1 hits . . . . .	98
4.9 Most No. 1 hits in last five years . . . . .	98
4.9.1 Data Takeaway: No. 1 in past five years . . . . .	99
4.10 Complex filters . . . . .	100
4.10.1 Multiple truths . . . . .	100
4.10.2 Either is true . . . . .	100
4.10.3 Mixing criteria . . . . .	101
4.10.4 Search within a string . . . . .	102

4.11 On your own . . . . .	102
4.11.1 Most Top 10 hits . . . . .	103
4.11.2 Find something you want to know . . . . .	103
4.12 Summarizing by year . . . . .	103
4.13 Update your index summary . . . . .	104
4.14 Turn in your project . . . . .	104
4.15 Review of what we've learned . . . . .	105
4.16 Soundtrack for this assignment . . . . .	105
<b>II Summing</b>	<b>106</b>
<b>5 Military Surplus Cleaning</b>	<b>107</b>
5.1 About the story: Military surplus transfers . . . . .	107
5.1.1 The 1033 program . . . . .	107
5.1.2 About the data . . . . .	109
5.2 The questions we will answer . . . . .	111
5.3 Getting started: Create your project . . . . .	112
5.4 Cleaning notebook . . . . .	112
5.4.1 The goals of the notebook . . . . .	113
5.4.2 Add a setup section . . . . .	113
5.5 Download the data . . . . .	114
5.6 Import the data . . . . .	115
5.6.1 Glimpse the data . . . . .	116
5.7 Checking datatypes . . . . .	116
5.8 Remove unnecessary columns . . . . .	116
5.9 Create a total_value column . . . . .	117
5.9.1 Check that it worked!! . . . . .	119
5.10 Controlled vs. non-controlled . . . . .	121
5.11 Filtering our data . . . . .	123
5.11.1 Checking the results with summary() . . . . .	123
5.12 Export cleaned data . . . . .	124
5.13 Render and clean up your notebook . . . . .	125
5.14 Things we learned in this lesson . . . . .	125
<b>6 Military Surplus Analysis</b>	<b>126</b>
6.1 Learning goals of this lesson . . . . .	126
6.2 Questions to answer . . . . .	126
6.3 Set up the analysis notebook . . . . .	127
6.3.1 Load the data into a tibble . . . . .	128
6.4 Filter to data of interest . . . . .	128
6.4.1 Get the controlled items . . . . .	129
6.4.2 Filter for Texas . . . . .	130

6.5	Building summaries with math . . . . .	131
6.5.1	Summarize across the data . . . . .	132
6.5.2	Data Takeaway: Totals since 2010 . . . . .	133
6.5.3	NA values in a sum, mean and median . . . . .	133
6.6	Totals by agency . . . . .	134
6.6.1	Group_by, then summary with math . . . . .	135
6.6.2	Add the total_value . . . . .	136
6.6.3	Check the math . . . . .	136
6.6.4	Arrange the results . . . . .	137
6.6.5	Consider the results . . . . .	138
6.6.6	Data Takeaway: Texas totals . . . . .	138
6.7	Looking a local agencies . . . . .	138
6.7.1	Save our “by agency” list . . . . .	138
6.7.2	Filtering within a vector . . . . .	139
6.7.3	Create a vector to build this filter . . . . .	140
6.7.4	Data Takeaway: Local totals . . . . .	142
6.8	Types of items shipped to each agency . . . . .	143
6.8.1	Items for local agencies . . . . .	145
6.8.2	Research some interesting items . . . . .	146
6.8.3	Data Takeaway: Local items . . . . .	147
6.9	Update index summary . . . . .	147
6.10	Turn in your project . . . . .	148
6.11	What we learned in this chapter . . . . .	148

### III Visualization 149

<b>7</b>	<b>Intro to ggplot</b>	<b>150</b>
7.1	Goals for this section . . . . .	150
7.2	Introduction to ggplot . . . . .	150
7.2.1	What ggplot is best for . . . . .	151
7.2.2	The Grammar of Graphics . . . . .	151
7.3	Start a new project . . . . .	152
7.3.1	Install palmerpenguins . . . . .	152
7.4	The layers of ggplot . . . . .	154
7.4.1	Our goal chart . . . . .	155
7.4.2	1st: the data . . . . .	155
7.4.3	2nd: Map the data . . . . .	156
7.4.4	3rd: geometries . . . . .	157
7.4.5	Chart code review . . . . .	159
7.4.6	Global vs local aesthetics . . . . .	160
7.4.7	Labels . . . . .	164
7.4.8	An aside: Titles and subtitles . . . . .	165

7.4.9	Themes . . . . .	166
7.5	Let's build a bar chart . . . . .	167
7.5.1	Prep princess data . . . . .	169
7.5.2	Build our plot with geom_col . . . . .	170
7.5.3	Add the geom_col layer . . . . .	171
7.5.4	Flip the axes . . . . .	172
7.5.5	Reorder the bars . . . . .	173
7.5.6	Adding a geom_text layer . . . . .	174
7.5.7	Add some titles and more labels . . . . .	176
7.6	On your own: Ice cream! . . . . .	178
7.7	What we've learned . . . . .	179
<b>8</b>	<b>Deeper into ggplot</b>	<b>180</b>
8.1	Learning goals for this chapter . . . . .	180
8.2	References . . . . .	180
8.3	Set up your notebook . . . . .	181
8.3.1	Let's get the data . . . . .	181
8.4	Goal 1: Line chart . . . . .	182
8.4.1	Working through logic . . . . .	183
8.4.2	Wrangling the data . . . . .	184
8.4.3	Plot the chart . . . . .	185
8.4.4	Cleaning Up . . . . .	186
8.4.5	Saving plots as an object . . . . .	190
8.5	Themes . . . . .	192
8.5.1	More with ggthemes . . . . .	193
8.6	Goal 2: Multiple line chart . . . . .	195
8.6.1	Prepare the data . . . . .	196
8.6.2	Plot the chart . . . . .	197
8.7	On your own . . . . .	199
8.8	Tour of some other adjustments . . . . .	200
8.8.1	Line width . . . . .	200
8.8.2	Line type . . . . .	201
8.9	Facets . . . . .	203
8.9.1	Facet grids . . . . .	205
8.10	Interactive plots . . . . .	206
8.11	What we learned . . . . .	207
<b>9</b>	<b>Tidy data</b>	<b>208</b>
9.1	Goals for this section . . . . .	208
9.2	The questions we'll answer . . . . .	208
9.3	What is tidy data . . . . .	208
9.3.1	Wide vs long data . . . . .	211
9.3.2	Why we might want different shaped data . . . . .	212

9.4	The candy project . . . . .	212
9.4.1	Prepare our candy project . . . . .	213
9.4.2	Get the data . . . . .	213
9.4.3	Drop unneeded columns . . . . .	214
9.4.4	Peek at the wide table . . . . .	215
9.4.5	Where are we going with this data . . . . .	215
9.5	The tidy verbs . . . . .	216
9.6	Pivot longer . . . . .	217
9.6.1	Pivot our candy data longer . . . . .	218
9.6.2	Get average candies per color . . . . .	219
9.6.3	Round the averages . . . . .	219
9.6.4	On your own: Plot the averages . . . . .	220
9.7	Introducing Datawrapper . . . . .	221
9.7.1	Review how to make a bar chart . . . . .	222
9.7.2	Start a chart . . . . .	222
9.7.3	Export your data for Datawrapper . . . . .	222
9.7.4	Build the datawrapper graphic . . . . .	223
9.8	Pivot wider . . . . .	223
9.8.1	Pivot wider example . . . . .	224
9.8.2	Choosing what to flip . . . . .	225
9.8.3	Pivot wider on your own . . . . .	227
9.9	Bonus questions . . . . .	227
9.9.1	Most/least candies . . . . .	228
9.9.2	Average total candies in a bag . . . . .	228
9.10	Turn in your work . . . . .	228
9.11	What we learned . . . . .	228

## IV Bind & Join 229

<b>10 Denied Cleaning</b>	<b>230</b>	
10.1	Goals of the chapter . . . . .	230
10.2	About the story: Denied . . . . .	230
10.3	About the data . . . . .	231
10.3.1	Set up your project . . . . .	232
10.4	The DSTUD data . . . . .	232
10.4.1	How the files differ . . . . .	233
10.4.2	Importing multiple files at once . . . . .	235
10.4.3	Clean up DSTUD file . . . . .	238
10.5	The DREF data . . . . .	240
10.5.1	Import reference data . . . . .	241
10.5.2	Clean up DREF file . . . . .	241

10.6 About joins . . . . .	242
10.6.1 Joining our reference table . . . . .	244
10.7 Some cleanup: filter and select . . . . .	246
10.8 Create an audit benchmark column . . . . .	247
10.9 Export the data . . . . .	248
10.10 New functions we used . . . . .	249
<b>11 Denied Analysis</b>	<b>250</b>
11.1 Goals of this chapter . . . . .	250
11.2 Project setup . . . . .	250
11.3 Import cleaned data . . . . .	251
11.4 Make a searchable table . . . . .	251
11.4.1 Pivot wider . . . . .	252
11.4.2 Make a datatable . . . . .	252
11.4.3 Data takeaway: How has Austin done . . . . .	253
11.5 Choosing a chart to display data . . . . .	253
11.6 Plot yearly student percentage . . . . .	254
11.6.1 Build the statewide percentage by year data . . . . .	255
11.6.2 Build the statewide percentage chart . . . . .	256
11.6.3 Data takeaway: State percentage . . . . .	258
11.7 Districts by benchmark and year . . . . .	258
11.7.1 Summarize the audit flag data . . . . .	258
11.7.2 Build the audit flag chart . . . . .	259
11.7.3 Which chart is best? . . . . .	262
11.7.4 Data takeaway: District benchmarks . . . . .	262
11.8 Local districts . . . . .	262
11.8.1 On your own . . . . .	264
11.8.2 Data takeaway: The local districts . . . . .	264
11.9 Special Education change since 2015 . . . . .	264
11.9.1 Describing the count changes . . . . .	265
11.9.2 Describing percentage differences . . . . .	265
11.10 Build your data drop . . . . .	266
11.11 Turn in your project . . . . .	266
11.12 New functions used in this chapter . . . . .	267
<b>Appendices</b>	<b>268</b>
<b>A R Functions</b>	<b>268</b>
A.1 Import/Export . . . . .	268
A.2 Data manipulation . . . . .	268
A.3 Aggregation . . . . .	269
A.4 Math . . . . .	269

<b>B Using posit.cloud</b>	<b>270</b>
B.1 Create a web project from our template . . . . .	270
B.2 Exporting a project . . . . .	271
B.3 Share your project . . . . .	272
B.4 Building a web project from scratch . . . . .	272
B.4.1 Create your project . . . . .	272
B.4.2 Create the Quarto file . . . . .	272
B.4.3 Create your index file . . . . .	273
<b>C Grouping by dates</b>	<b>274</b>
C.1 Setting up . . . . .	274
C.2 Plucking date parts . . . . .	275
C.3 Grouping by a date part on the fly . . . . .	275
C.4 Making reusable date parts . . . . .	277
C.4.1 Let's make a year . . . . .	278
C.4.2 The magical month . . . . .	278
C.4.3 Floor dates . . . . .	279
<b>D A counting shortcut</b>	<b>282</b>
D.1 Setup and import . . . . .	282
D.2 Basic count . . . . .	282
D.2.1 Sort the results . . . . .	283
D.2.2 Name the new column . . . . .	284
D.2.3 Filter results as normal . . . . .	284
D.3 Grouping on multiple variables . . . . .	285
<b>E Using case_when</b>	<b>286</b>
E.1 Catching up with the data . . . . .	286
E.1.1 Categorization logic with case_when() . . . . .	287
<b>F Troubleshooting</b>	<b>291</b>
F.1 When things break . . . . .	291
F.2 Learning how things work . . . . .	292
F.2.1 Help docs . . . . .	292
F.2.2 Good Googling . . . . .	293
F.2.3 Tidyverse docs and cheatsheets . . . . .	293
F.3 R Frequently asked Questions . . . . .	293
F.3.1 Functions, objects and variables . . . . .	293
F.3.2 Some R code basics . . . . .	294
<b>G Exploring a new dataset</b>	<b>296</b>
G.1 Start by listing questions . . . . .	296
G.2 Understand your data . . . . .	296

G.3	Learn more through cleaning . . . . .	297
G.3.1	Cleaning up categorical data . . . . .	298
G.4	Pay attention to the shape of your data . . . . .	298
G.5	Counting and aggregation . . . . .	299
G.5.1	Counting rows based on a column . . . . .	299
G.5.2	Sum, mean and other aggregations . . . . .	300
G.5.3	Creating columns to show difference . . . . .	300
G.6	Time as a variable . . . . .	301
G.7	Explore the distributions in your data . . . . .	301
G.7.1	More on histograms . . . . .	302
G.8	Same ideas using spreadsheets . . . . .	303
<b>H</b>	<b>Chart production tips</b>	<b>304</b>
H.1	Titles, descriptions and annotations . . . . .	304
H.2	Other considerations . . . . .	305

# About this book



Note

**This work is under constant revision.** It is written using [Quarto](#) and the source is available on [Github](#).

This version has been revised for spring 2024.

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. The first two versions of the book were written by Associate Professor of Practice [Christian McDonald](#). Assistant Professor [Jo Lukito](#) began collaborating and teaching sections of the Reporting with Data in spring 2022. Anastasia Goodwin taught a section in fall 2023 and provided some valuable edits to this work.

This means the writing voice or point of view may change or be confusing at times. For instance, Prof. McDonald uses a Mac and Prof. Lukito uses a PC, so keyboard commands and screenshot examples may flip back and forth. We'll try to note the author and platform at the beginning of each chapter.

## Some words from Prof. McDonald

I'm a strong proponent of what I call Scripted Journalism, a method of committing data journalism that is programmatic, repeatable, transparent and annotated. There are a myriad of programming languages that further this, including Python ([pandas](#) using [Jupyter](#)) and JavaScript ([Observable](#)), but we'll be using [R](#), [Quarto](#) 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.

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, we strive to make it as simple as possible, which means we 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. We rarely discuss differences from base R; Tidyverse is our default.

Programming languages evolve constantly, and R has seen some significant changes in the past few years, many of them driven by Posit, the company that makes RStudio and maintains the [Tidyverse](#) packages.

- The introduction of [Quarto](#) in mid-2022. This modern implementation of RMarkdown is a major shift for this edition of the book. Every lesson and video will need to be updated, and we may not get to all of them. The good news is RMarkdown still works inside Quarto documents.
- The introduction of the base R pipe `|>` in 2021. Posit developers began using the `|>` in favor of the magrittr pipe `%>%` in 2022, and this book follows their lead. The two implementations work interchangeably and you'll see plenty of `%>%` in the wild.
- The use of YAML code chunk options. I first noticed this style when I started using Quarto in 2023. It might not matter much for this book as we don't use that many code chunk options in our assignments, but we'll see.

## Conventions and styles in this book

We will try to be consistent in the way we write documentation and lessons. But we are human, so sometimes we break our own rules, but in general keep the following in mind.

### Things to do

Things to DO are in ordered lists:

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

Explanations are usually in text, like this very paragraph.

Sometimes details will be explained 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.

## Code blocks

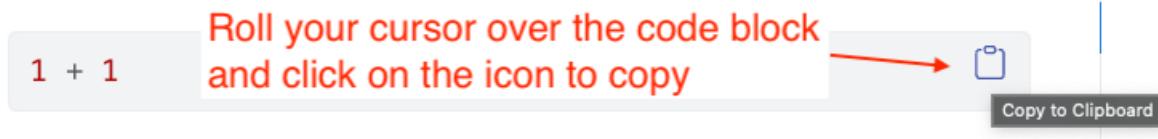
This book often runs the code that is shown, so you'll see the code and the result of that code below it.

```
1 + 1
```

```
[1] 2
```

### Copying 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 block content to your 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. **We encourage you to always type short code snippets.** Leave the copying to long ones.

### Hidden code

Sometimes we want to include code in the book but not display it so you can try the to write the code yourself first. When we do this, it will look like this:

```
1 + 1
```

```
[1] 2
```

If you click on the triangle or the words that follow, you'll reveal the code. Click again to hide it.

## Annotated code

Sometimes when we are explaining code it is helpful to match lines of code to the explanation about them, which we do through annotated code.

```
mtcars |>  
  head()
```

(1)  
(2)

- ① First we take the Motor Trend Car Road Tests data set AND THEN ...
- ② We pipe into the head() command, which gives us the “top” of the data.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

When there are annotations like this you have to be careful if you are copying code from the book. Either copy it one line at a time or use the copy icon noted above.

## Fenced code

Sometimes we need to show code chunk options that are added, like when explaining how to name chunks. In those cases, you may see the code chunk with all the tick marks, etc. like this:

```
```{r block-named}  
1 + 1  
~~~
```

[1] 2

or

```
```{r}  
#| label: block-named-yaml  
  
1 + 1  
~~~
```

[1] 2

You can still copy/paste these blocks, but you'll get the entire code block, not just the contents.

## Notes, some important

We will use callouts to set off a less important aside:

Markdown was developed by JOHN GRUBER, as outlined on his [Daring Fireball blog](#).

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

! Important

You really should learn how to use [Markdown](#) as you will use it the whole semester, and hopefully for the rest of your life.

## About the authors

### Christian McDonald

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. I've taught data-related course at UT since 2013. I also serve as the innovation director of the [Dallas Morning News Journalism Innovation Endowment](#).

- The UT Data Github: [utdata](#)
- Threads: [@critmcdonald](#) | Mastodon [crit](#) | Bluesky: @crit
- Email: [christian.mcdonald@utexas.edu](mailto:christian.mcdonald@utexas.edu)

### Jo Lukito

I'm an aspiring-journalist-turned-academic who studies journalism and digital media. To make a long story short (tl;dr): I trained to be a journalist as an undergraduate student, but just fell in love with **researching and supporting** journalism. I completed my Ph.D in 2020, and my dissertation focused on international trade reporting (which relies on plenty o' data). I also do a ton of social media research (especially in politics and disinformation), so if you're interested in the social media beat, I'm your gal!

- Prof. Jo Lukito's git: [jlukito](#)
- Twitter: [JosephineLukito](#)
- Email: [jlukito@utexas.edu](mailto:jlukito@utexas.edu)
- Website: <https://www.jlukito.com/>

## License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Let's just say this is free on the internet. We 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 for Data Science](#)
- The [Tidyverse](#) site, which has tons of documentation and help.
- The [RStudio Cheatsheets](#).
- [ggplot2: Elegant Graphics for Data Analysis](#)
- [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 and University of Texas alum. A series of videos and tutorials on using R in a journalism setting.

# 1 Install Party

This chapter was written by Prof. McDonald using macOS.

Let's get this party started.

## 1.1 Mac vs PC

We're a big fan of using keyboard commands to do operations in any program, but Prof. McDonald references this from a Mac perspective and Prof. Lukito references this from a PC perspective. So if we say use *Cmd+S* or *Command+S* to save, that translates to *Cntl+S* or *Control+S* on a PC. You can typically just switch *Cmd* (for Mac) and *Cntl* (for PC), but occasionally there are other differences. You can usually check menu items in RStudio to figure out the command your your computer.

**We will note the author and operating system at the top chapters so you have a frame of reference.**

## 1.2 RStudio vs posit.cloud

This book is written assuming the use of the RStudio IDE application, which is free and available for Macs, PC's and Linux. In cases where computers have trouble running R and Rstudio, it is possible to to use the online [posit.cloud](#) version of RStudio. If you are using posit.cloud for the Reporting with Data class, you'll likely hit a pay tier at some point. The Cloud Plus plan at \$5/mo is typically sufficient. You may find some specific posit.cloud instruction in this book, but we don't outline every difference.

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.

## 1.3 Installing R

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

1. Go to the <https://cloud.r-project.org/>.
2. Click on the link for your operating system.
3. The following steps will differ slightly based on your operating system.
  - For Macs, you'll need to know if you have an Apple or Intel chip. Go under the Apple menu to **About this Mac** and you should be able to see if you have *Apple* or *Intel*. You'll choose which download based on that.
  - 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.)

This should be pretty self explanatory: once you download the installation file, you should be able to run it on your respective computers to install R.

You'll never “launch” R as a program in a traditional sense, but you need it on your computers (it's mostly so that the computer can recognize R as a “language”). In all situations (in this class, and beyond), we'll use RStudio, which is next.

## 1.4 Install Quarto

Go to the [Quarto Getting Started page](#) and there should be a big blue button that links to the software for your computer. Follow the prompts to install.

## 1.5 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.

1. Go to <https://posit.co/downloads/>.
2. You've already done step 1 to install R. Find step 2.
3. There should be a big blue button to download for your computer.
4. Install it. This should be like installing any other program on your computer.

### **1.5.1 Getting “Git” errors on Macs**

If later during package installation you get errors that mention “git” or “xcode-select” then **say yes!** and do it. It might take some time. If it doesn’t finish, then try again the next time it comes up.

## **1.6 Class project folder**

To keep things consistent and help with troubleshooting, we recommend that you 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 us 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.

## 2 Introduction to R

This chapter was written by Prof. McDonald using macOS, with edits by Prof. Lukito.

### 2.1 About Quarto, R and scripted journalism

Before we dive into RStudio and programming and all that, I want to show you where we are heading, so you can “visualize success”. As I wrote in the book intro, I’m a believer in Scripted Journalism ... data journalism that is repeatable, transparent and annotated. As such, the whole purpose of this book is to train you **to create documents to share your work**.

The best way to explain this is to show you an example.

1. Go to this link in a new browser window: [Major League Soccer salaries](#).

This is a website with all the code from a data journalism project. If you click on the navigation link for [Cleaning](#) you can read where the data come from and see all the steps I went through – with code and explanation – to process the data so I could work with it. And in the [Analysis 2023](#) notebook you’ll see I set out with some questions for the data, and then I wrote the code to find my answers. Along with the way I wrote explanations of how and why I did what I did.

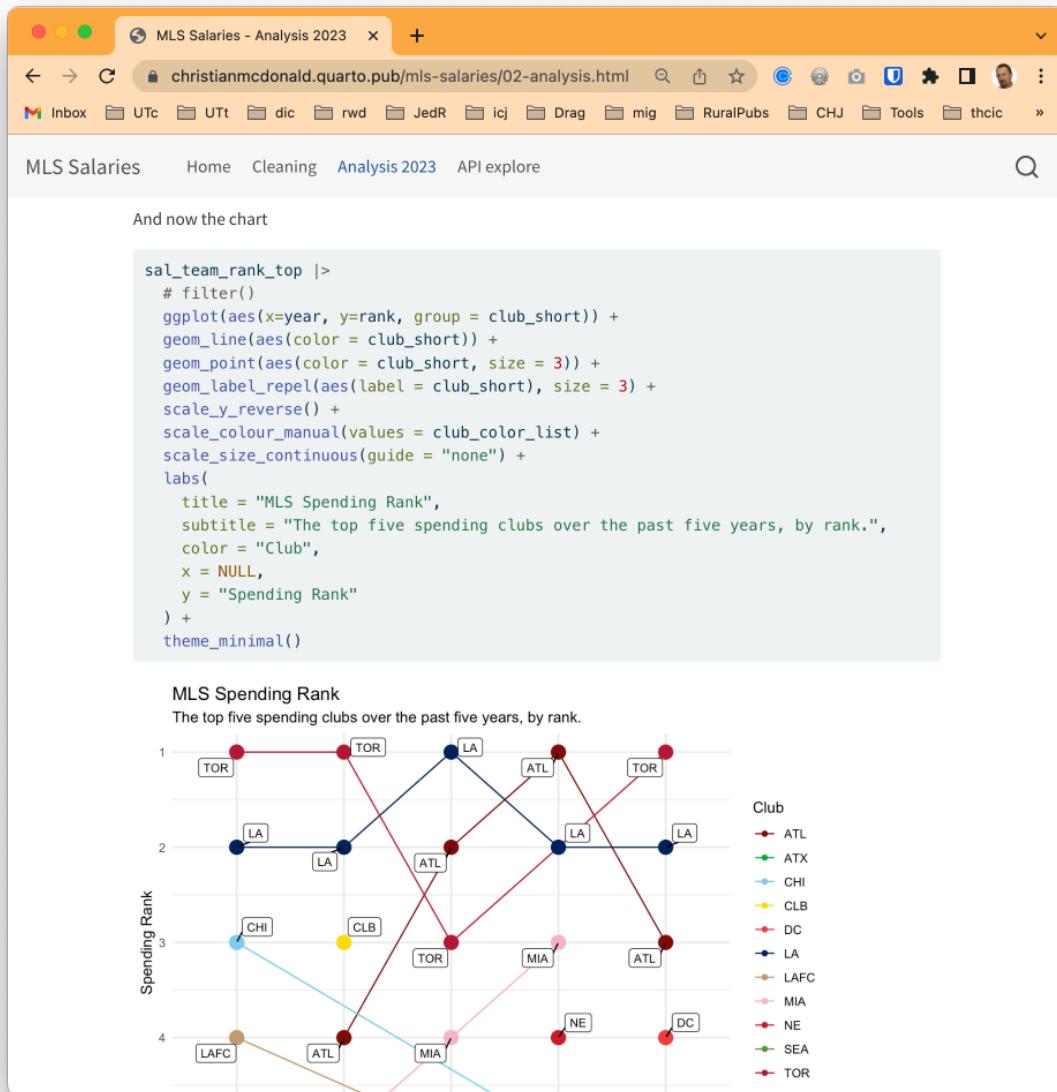


Figure 2.1: Quarto Pub page

This website was created using Quarto and R, and the tool I used to write it was the RStudio IDE. Here's the crazy part: **I didn't have to write any HTML code**, I just wrote my thoughts in text and my code in R. With a few short lines of configuration and a two-word command `quarto publish` I was able to publish my work on the Internet for free.

Keep this in mind:

- We can also easily publish the same work in other formats, like PDF, Word or even a slide show.
- We can also choose NOT to “publish” our work. We don’t *have* to share our work on the internet, we are just ready if we want to.

Creating shareable work is our goal **for every project**. Let’s get started.

## 2.2 RStudio tour

### ⚠ Warning

If you are using the online posit.cloud version of RStudio, then some steps have to be done differently, mainly in dealing with project creation and exporting. I’ll try to note when I’m aware of differences, but may not go into great detail in the book. Happy to do so in class.

In this case, you need go to your posit.cloud account and then use the blue **New Project** button to launch a new RStudio project, and then continue below.

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

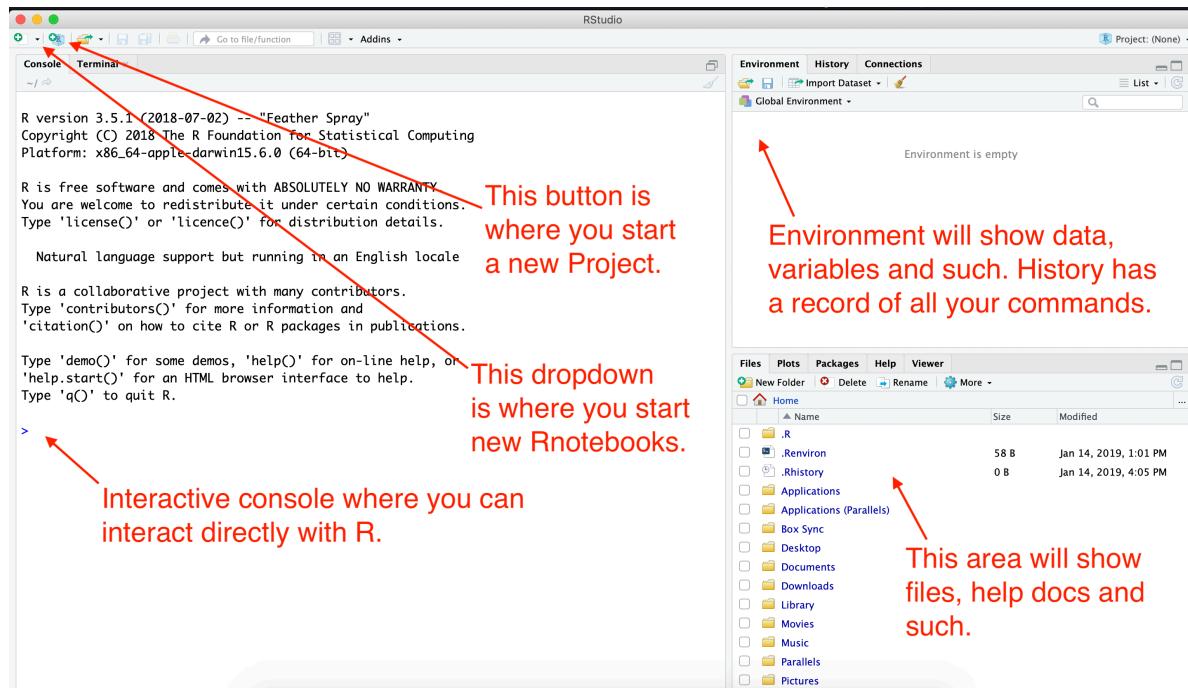


Figure 2.2: RStudio launch screen

## 2.3 Updating preferences

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

1. Go to the **Tools** menu and choose **Global Options**.
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 *Apply* to save the change (but don’t close the box yet).

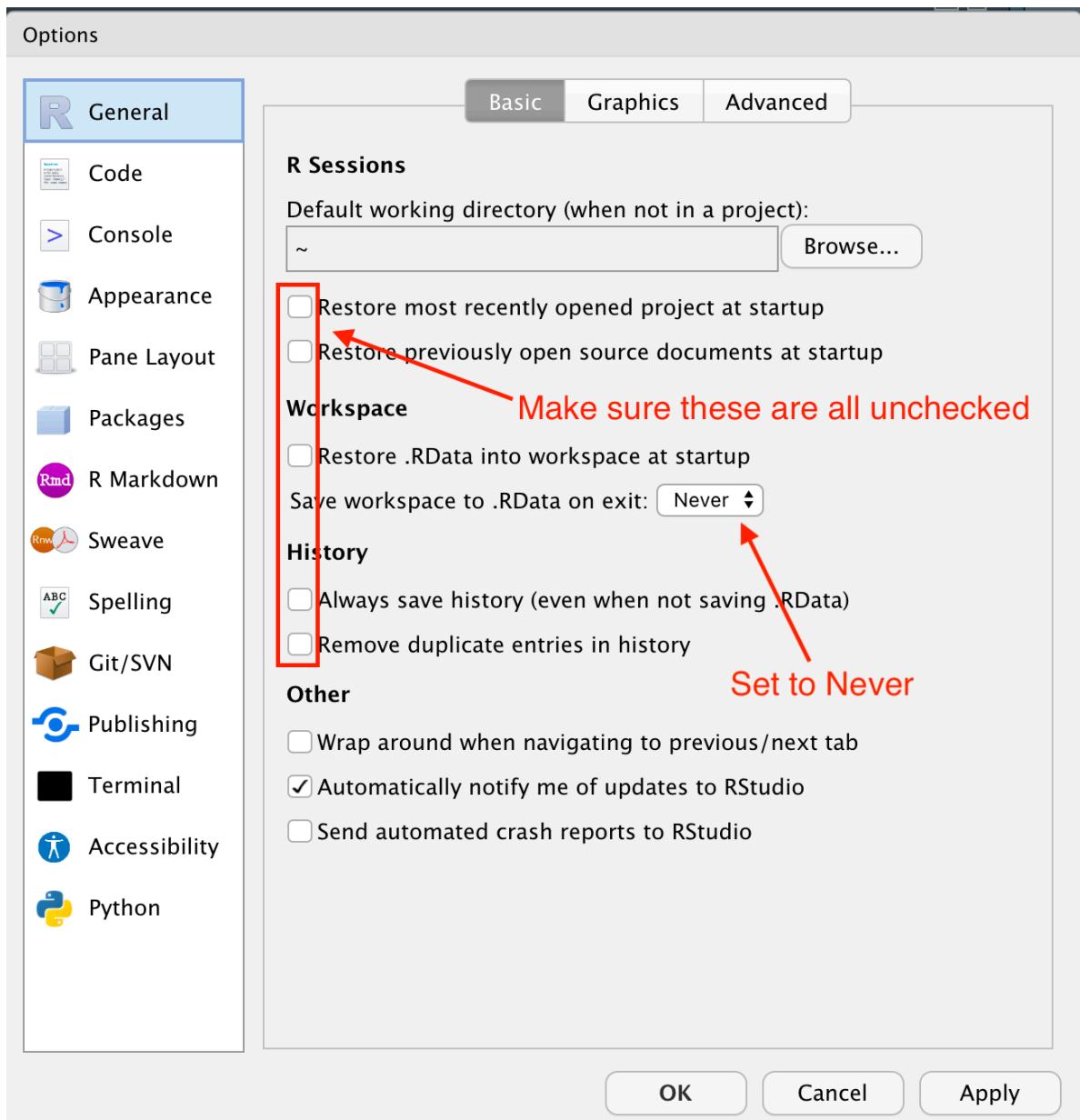


Figure 2.3: RStudio preferences

Next we will set some value in the **Code** pane.

1. On the left options, click on the **Code** pane.
2. Check the box for **Use native pipe operator**, |>.
3. Click **OK** to save and close the box.

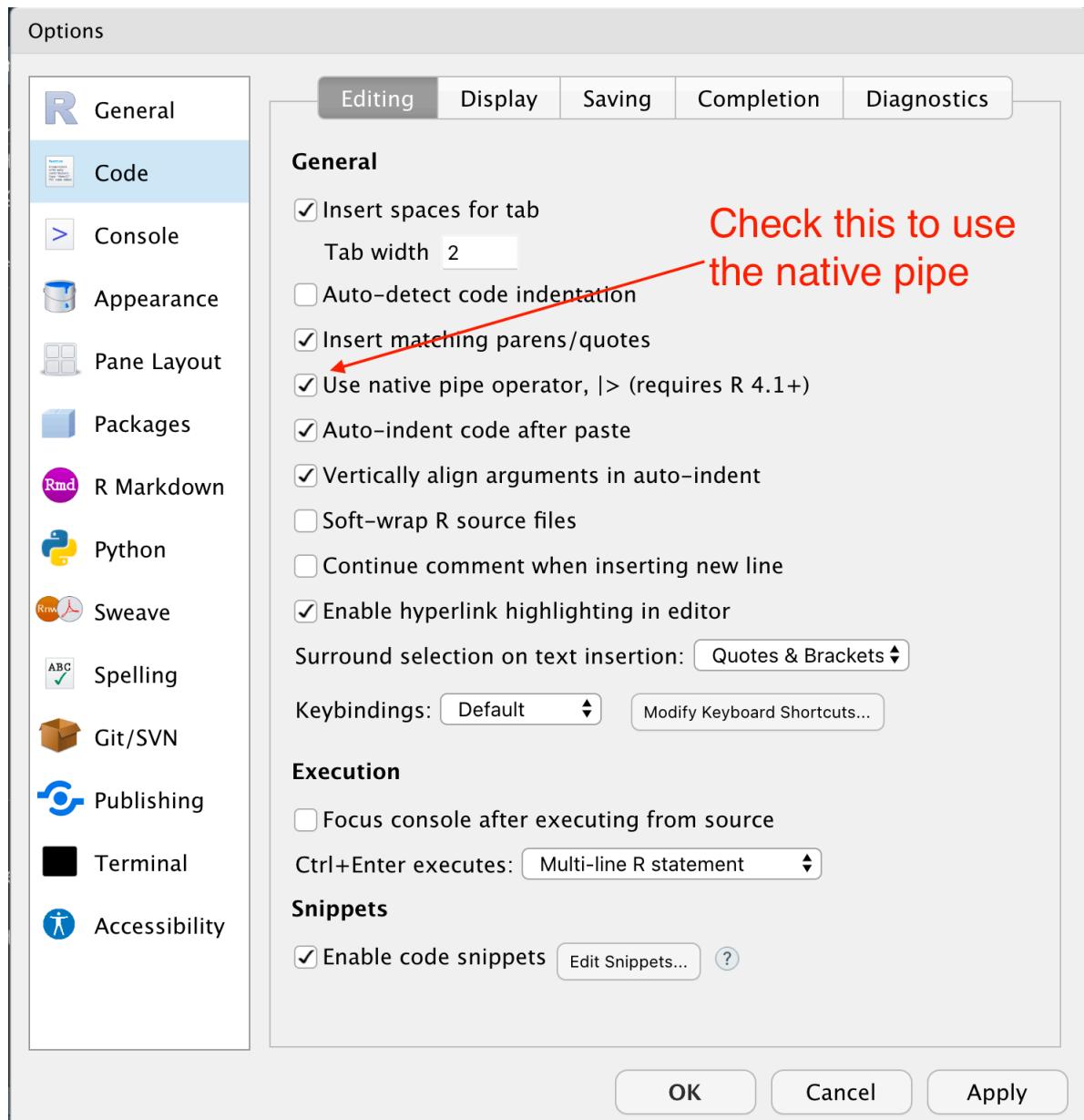


Figure 2.4: Native pipe preference

We'll get into why we did this part later.

## 2.4 The R Package environment

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 code that solves that problem and save it into an R “package” so they can use it later. 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 the [tidyverse](#) developed by Hadley Wickham and his team at Posit. It’s a set of R packages for data science that are designed to work together in similar ways. Prof. Lukito and I are worshipers of the tidyverse worldview and we’ll use these tools extensively. While not required reading, I highly recommend Wickham’s book [R for data science](#), which is free.

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

We will use these tidyverse packages extensively throughout the course. We’ll install some other packages later when we need them.

### 2.4.1 How we use packages

There are two steps to using an R package:

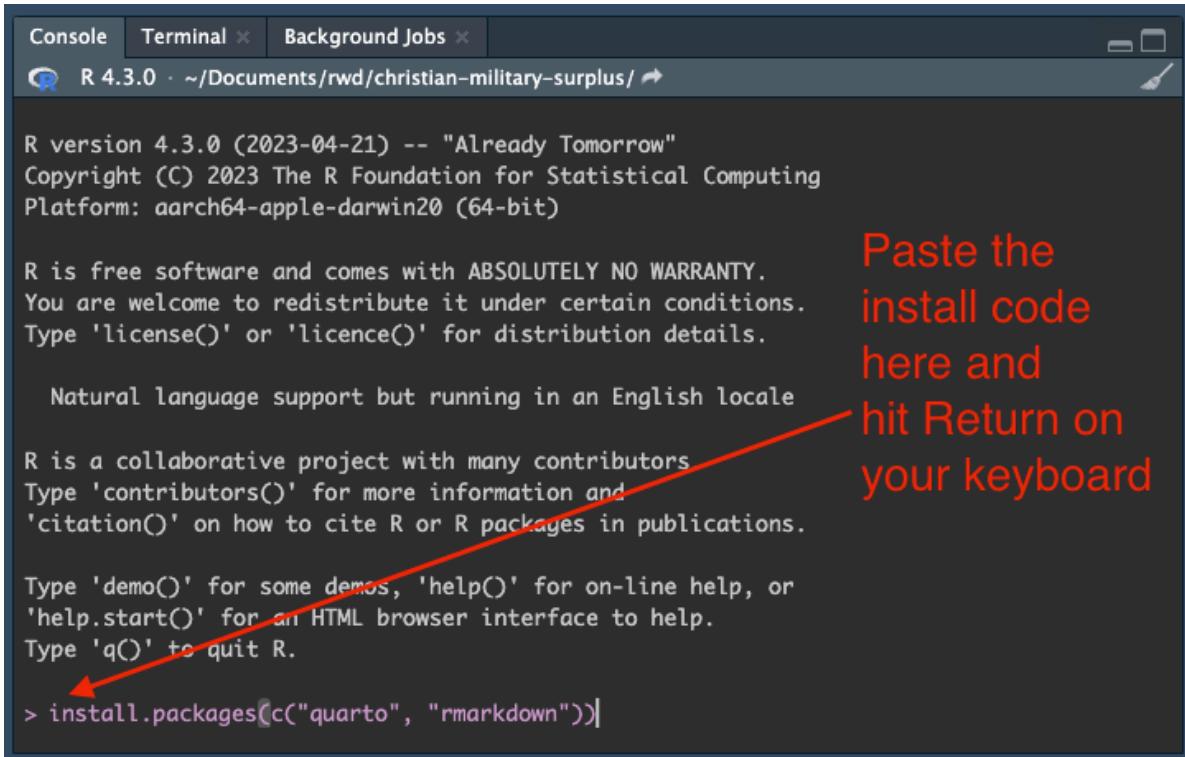
- **Install the package** onto your computer by using `install.packages("package_name")`. You only have to do this once for each computer, so I usually do it using the R Console instead of in a notebook.
- **Include the library** using `library(package_name)`. This has to be done for each notebook or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

#### Note

You use “quotes” around the package name when you are installing, but you DON’T need quotes when you load the library.

### 2.4.2 Install some packages

We need to install some packages before we can go further. To do this, we will use the **Console**, which we haven’t talked about much yet.



```

Console Terminal × Background Jobs ×
R 4.3.0 · ~/Documents/rwd/christian-military-surplus/ ↗

R version 4.3.0 (2023-04-21) -- "Already Tomorrow"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> install.packages(c("quarto", "rmarkdown"))

```

Paste the  
install code  
here and  
hit Return on  
your keyboard

Figure 2.5: The Console and Terminal

1. Use the image above to orient yourself to the R Console and Terminal.
2. Copy the code below and paste it into the Console, then hit **Return**.

```
install.packages(c("quarto", "rmarkdown", "tidyverse", "janitor"))
```

You'll see a bunch of response fly by in the Console. It's probably all fine unless it ends the last response with an error.

If you are using the RStudio IDE app on your computer, you only have to do this `install.packages()` move once. However, if you are using the online posit.cloud version of RStudio, you'll have to do this for **each new project** because each project is a new virtual computer. I've included a [posit.cloud cheatsheet here](#)

OK, we're done with all the computer setup. Let's get to work.

## 2.5 Starting a new Quarto Website

! Important

This is a case where posit.cloud differs greatly. See the [posit.cloud Appendix](#) to see how to build a new Quarto Website 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 hexagon sign.
2. That brings up a box to create the project with several options. You want **New Directory** in this case.
3. For **Project Type**, choose **Quarto Website**.
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 **rwd** folder we created earlier.

I want you to be anal retentive 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.

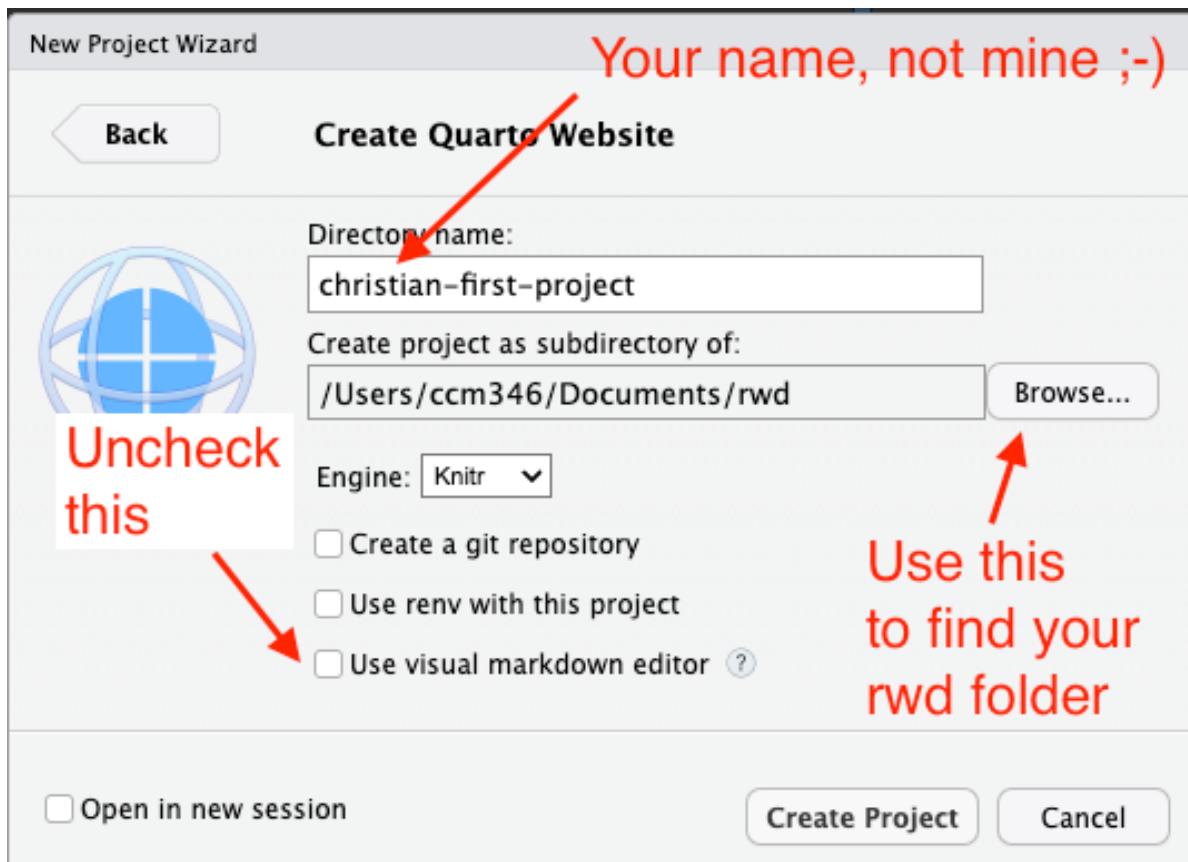


Figure 2.6: Quarto project, directory

When you hit **Create Project**, your RStudio window will refresh and two things will happen:

- You'll see the three files listed in your **Files** window.
- A new document window will open on the left.

### 2.5.1 The files pane

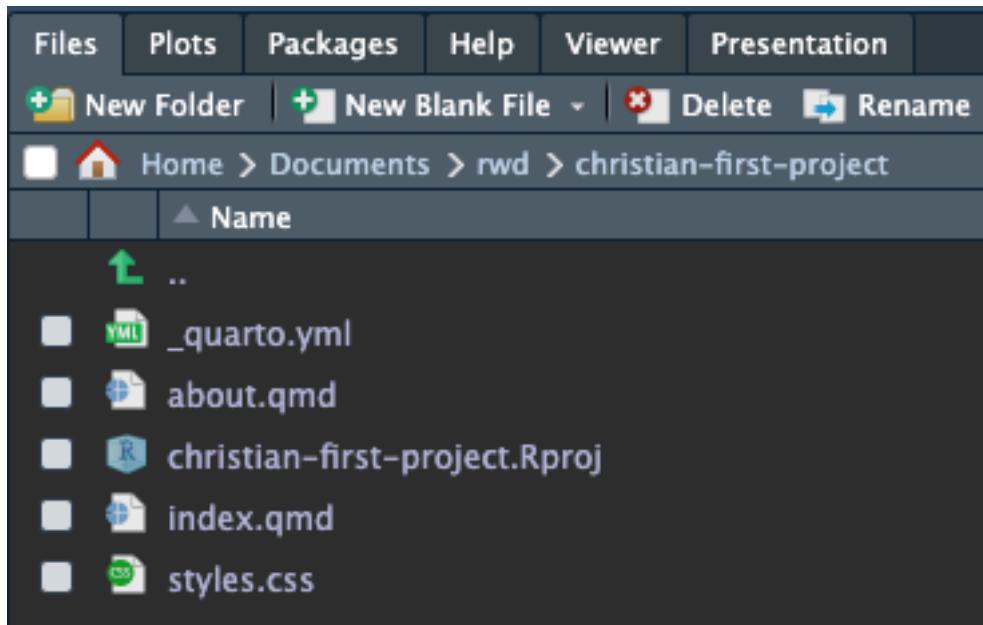


Figure 2.7: New Quarto Project

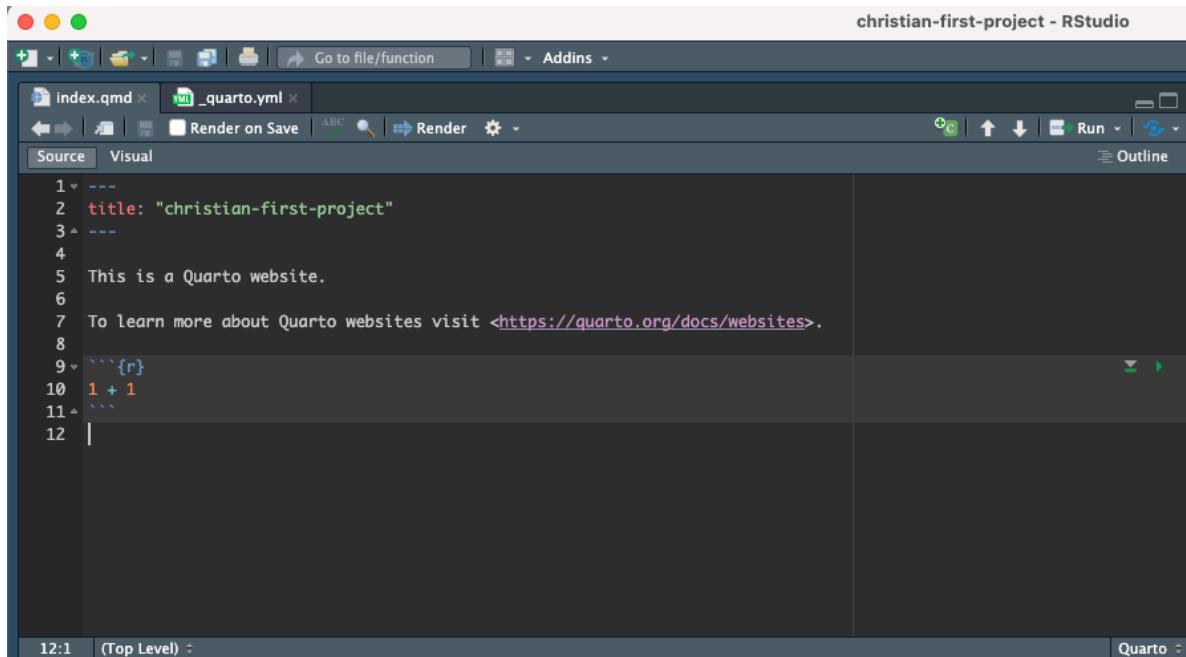
Let's walk through the files created and explain what they are in order of importance.

- The `_quarto.yml` file is a YAML configuration file for your project. This allows us to set publishing rules for our Quarto project. We might not get too much into all the options, but you can read more about it in the [Quarto Guide](#) if you like.
- The `index.qmd` file is a Quarto document that makes the “home page” of your website. In this book you’ll use the index to describe your project and record other important information related to it. Each Quarto file you create can become a new page in your website, and they all end in `.qmd`, which stands for Quarto Markdown. (We might also encounter and edit `.Rmd` files, which are very similar, but just a little less awesome.) The `about.qmd` file is another Quarto document created as a placeholder. We’ll usually rename/reuse that or delete it.
- The `christian-first-project.Rproj` file is your **Project** file. It sets the *working directory* or what is essentially “home base” for your project. When you go to open your project again, this is the file you will open.
- `styles.css` is file where you can assert extra control on your website. We won’t use it.

The big thing to remember is this: Most of our work is done in files with the `.qmd` suffix.

## 2.6 The Quarto document

The document that opened on the left is our Quarto document where we will do our work. The Quarto document is a cutting-edge way of authoring programming documents where the result can be output in a myriad of formats: As HTML, PDFs, slideshows, books, etc. In fact, this very book you are reading is written in R using Quarto.

A screenshot of the RStudio interface showing a new Quarto document. The title bar reads "christian-first-project - RStudio". The left pane shows the code editor with two tabs: "index.qmd" and "\_quarto.yml". The "index.qmd" tab contains the following R Markdown code:

```
1 ---  
2 title: "christian-first-project"  
3 ---  
4  
5 This is a Quarto website.  
6  
7 To learn more about Quarto websites visit <https://quarto.org/docs/websites>.  
8  
9 ````{r}  
10 1 + 1  
11 ````  
12 |
```

The "Quarto" tab is visible at the bottom right of the interface.

Figure 2.8: New Quarto document

I could write reams of text about the birth of Quarto and the evolution of what came before it, but you won't really care. Let's break this down to what you need to know:

- These documents allow us to weave together our **thoughts** and our **code**. We'll use **Markdown** to record our thoughts, and R to write our code.
- We *write* the document and we *run* the code to see the results, but all the while we are creating documents that we will be **rendered** to share with others. Our goal with these documents is to explain to an “audience” what we are doing with our data analysis. Often our most important audience is our future self.

Think of *Render* like exporting or publishing a pretty version of your work.

### 2.6.1 Render the document

Let's see what this basic document looks like when we Render it.

1. In the toolbar of the document you'll see a blue arrow with **Render** after it.
2. Click on that button or word.

What this should do is open up the Console below the document and you'll see a bunch of feedback, but on the right side of RStudio. You should end up with the **Viewer** pane that shows your document. Here is a tour of sorts:

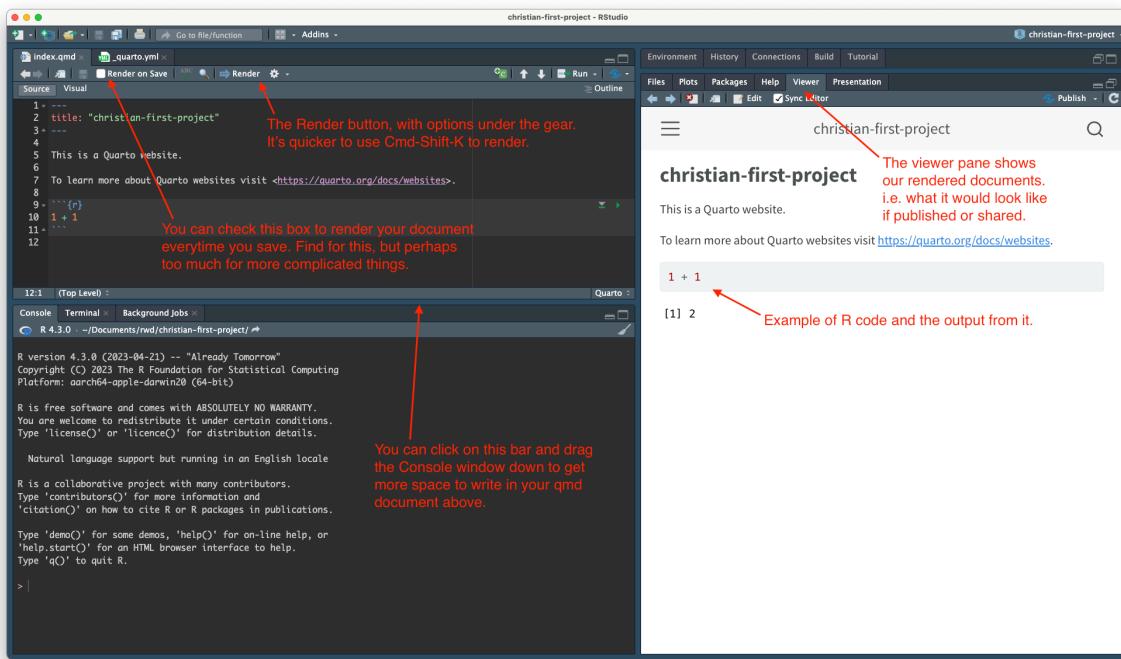


Figure 2.9: Render Quarto

For the posit.cloud users, this will launch in a new web browser window. RStudio users can do that if you want though a button in the Viewer toolbar.

### 2.6.2 The metadata

The top of our document has what we call the **metadata** of our document written in YAML. These are commands to control the output of our document. Right now it only has the title of our document.

```
---
```

```
title: "christian-first-project"
```

```
--
```

When we created our project, RStudio added the name of our folder here, **which kinda sucks** because that is not what the title is for. It should describe a title for your project like the top of a Microsoft Word or Google Docs document.

1. Edit the text inside the quotes to be more like a title you would find in a Word or Google Doc document, like this, but your your name:

```
---
```

```
title: "Christian's First Quarto project"
```

```
--
```

2. **Re-render** the document so you can see the updates. (You can click on the **Render** button or do **Cmd-Shift-K** to update it.)

### 2.6.3 R code chunks

The next bit of our document shows an R chunk:



```
9 ``{r}
10 1 + 1
11 ``
```

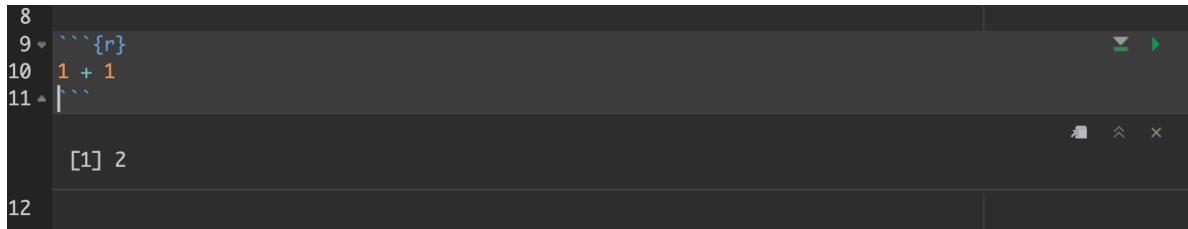
Figure 2.10: R chunk

This is where we write and execute our programming code. Let's break down parts of this:

- The three tick marks ` at the beginning and end indicate where the code chunk starts and ends.
- The {r} part notes that this is R code. (Quarto supports other languages like {python}, but we'll stick with R.)
- The 1 + 1 part is the code. In this case, it is some basic math. We do cooler stuff later.
- The green triangle that points to the right will **run all the code in this chunk**.
- The gray triangle that points down with the green bar will **run all the code above this chunk**.

Let's run this chunk of code to see what happens.

1. Click on the green triangle.



A screenshot of the RStudio interface. The code editor shows the following R code:

```
8
9 ~`{r}
10 1 + 1
11 ~`{r}
12
```

The output pane below the code editor shows the result of the code execution:

```
[1] 2
```

Figure 2.11: Code chunk has run

Doing this executes the code that is in the R block and it will print the result to your document below the chunk. You might have noticed something similar when you rendered your document earlier.

! Important

There are about five keyboard commands that I will *implore* you to learn. Here are the first three. Remember if you are on a PC use **Cntl** instead of **Cmd**.

- **Cmd+option+i** will insert a code block.
- **Cmd+Return** will run a single line (or selection) of code within a code block.
- **Cmd+Shift+Return** will run a whole code chunk.

## 2.7 Let's do some data analysis

Remember the goal of all our work here in this class is to explain to use data analysis to make sense of the world and then explain to others what we are doing. There are multiple parts to that:

- What are we thinking?
- Our code.
- Our thoughts about the results, if any.

Our goal with this notebook is to discover what is the [socially-acceptable age to date someone older or younger than ourselves](#).

1. Copy the text below. Note you can use the copy-clipboard button in the book if you roll your cursor over the code and click on the clipboard icon at the top right.
2. Paste the code at the bottom of your notebook.
3. Re-render your document to see what it looks like.

```
## My upper dating age
```

The following section details the [socially-acceptable maximum age of anyone you should date](https://the\_url.org).

The math works like this:

- Take your age
- subtract 7
- Double the result

Let's walk through this Markdown code. You might bookmark the [Markdown basics](#) so you can refer back to it as you learn.

- The ## line is a “Header 2” headline, meaning it is the second biggest. (The title is an H1.) Add more hashmarks ### and you get a smaller headline, like subheads, etc.
- There is a full blank return between each element, including paragraphs of text. The exception is the bullet list.
- 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](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.

### Note

I should note at this point there is a “Visual” editor where RStudio gives you more formatted look as your editing as it writes Markdown underneath the hood. I want you to use the “Source” editor so you can see and learn the underlying Markdown syntax. All my examples will be written in “Source” mode. After you pass this class, you can use “Visual” mode.

## 2.7.1 Adding new code chunks

Let's add a new R chunk to add the code that will calculate the maximum age of someone we should date.

1. Add a few blank lines after your Markdown text.
2. Use the keyboard command *Cmd+Option+i* to add an R chunk.
3. Your cursor will be inserted into the middle of the chunk. Type or copy this code in the space provided:

```
# update 57 to your actual age  
age <- 57  
(age - 7) * 2
```

```
[1] 100
```

1. Change the number 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.

Congratulations! The answer given at the bottom of that code chunk is the upper end age of someone socially acceptable for you to date.

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

- `# update 57 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 inside code chunks by starting with `#`. You can also add a comment at the end of a line. Comments will appear greyed out and *italicized* in your code chunks if formatted correctly.
- `age <- 57` is assigning a number (57) 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 below it the result of the math equation, `[1] 100` in my case. That means there was one observation [1], and the value was “100”. For the record, my wife is *much younger* than that. Perhaps this formula breaks down when you get older `-\_()_-/-`.

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

### 2.7.2 Practice adding code chunks

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

1. Add a Markdown headline and text describing what you are doing.
2. Use the keyboard command to create a new code chunk.
3. Include a comment within the code block that explains you are using the `age` variable already established.
4. Add the math statement `(age / 2) + 7` to the chunk.
5. Run the chunk and re-render your document.

Now you know the youngest a person should be that you date. FWIW, we don't recreate the assignment of the `age` variable since we already have one. A Quarto document is designed to run from top to bottom so that all the pieces work together.

### 2.7.3 The toolbar

One last thing to point out in the document window: The toolbar that runs across the top of the document window. The image below explains some of the more useful tools, but you *REALLY* should learn and use the keyboard commands instead.

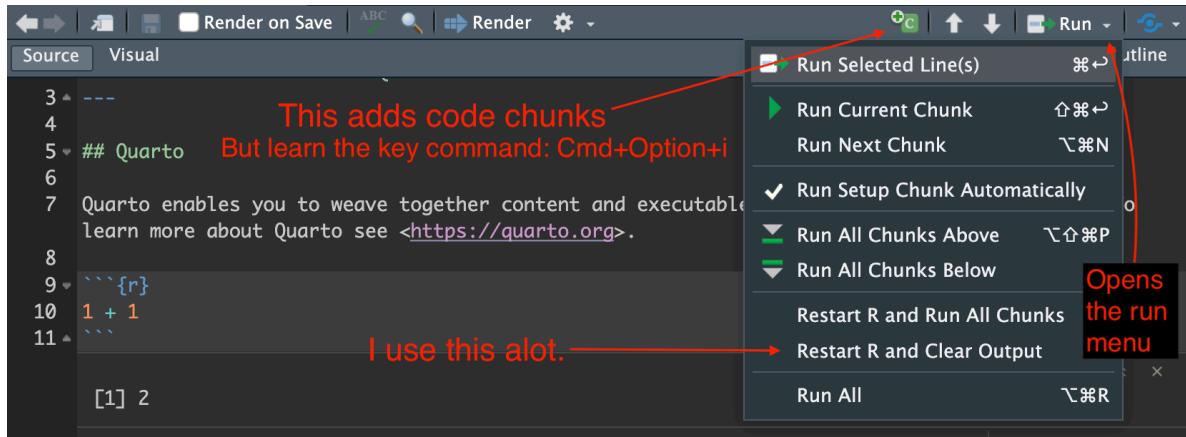


Figure 2.12: R Notebook toolbar

## 2.8 A quick look back at files

We've been concentrating on editing the Quarto document, but let's peek back at the Files pane to note some new files that were created.

1. In the window with the viewer, you'll notice several other tabs: Files, Plots, Packages, etc. Click on the **Files** tab.

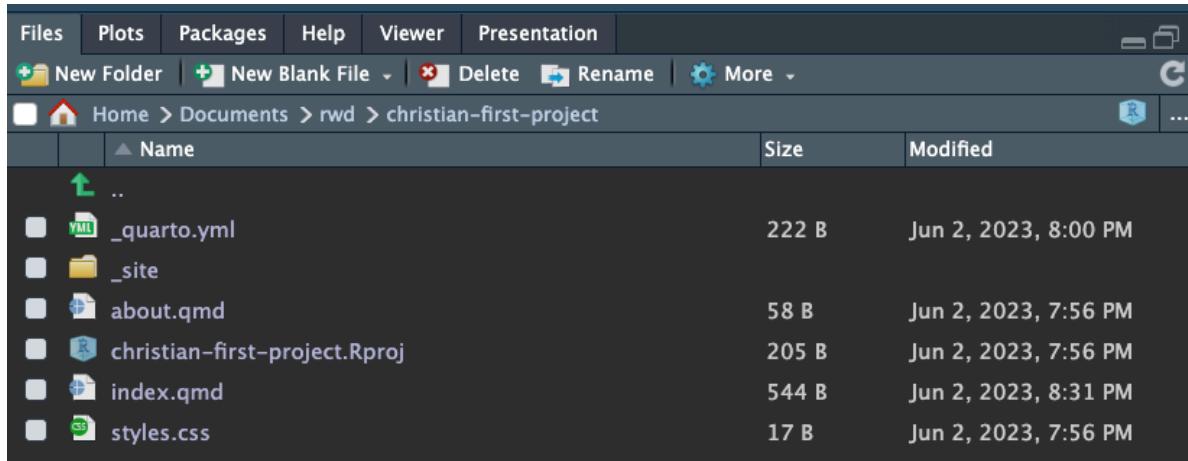


Figure 2.13: Quarto files list

Note there is a new folder there, `_site`. All your rendered versions go into the folder to make it easy to share or publish them.

With our next project, we'll create some other folders to store our data and such.

## 2.9 Publish to quarto.pub

Let's publish our work to the Internet!!!!

1. Go to [quartopub.com](https://quartopub.com) and sign up for a free account.
2. Come back to RStudio and down by the Console, click on the **Terminal**<sup>1</sup> pane.
3. Type in `quarto publish` and hit **Return** on your keyboard.

A bunch of stuff will happen in your Terminal pane. Here is a video showing this:

<https://www.youtube.com/watch?v=ctkKKgD1uQc>

A couple of things about this the video clip above:

- The first time you run this, you'll be asked to authenticate to quarto.pub, so it will be a little different.
- In the last bit of that video, a browser window opens with your website. That's cool!

You can learn more about [Quarto pub here](#), including how to configure your profile page.

---

<sup>1</sup>The Terminal is where you can send commands to your computer using text vs. pointing and clicking through “normal” actions. It’s super powerful and useful, but this is probably the only terminal command we’ll use.

## 2.10 On your own: Update the About page

Try some things on your own! Go into the `about.qmd` page and write some things about yourself, like your favorite hobbies. Use the [Markdown](#) guide to write headlines, lists and maybe even try an image? (You can use a URL from an image on the web.)

- Re-render your pages
- Re-publish your pages by going back to Terminal and using the `quarto publish` command.

## 2.11 Turning in our projects

### Note

This is a bit different for posit.cloud users. See below.

The best way to turn in all of those files into Canvas is to compress the project folder 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.
  - [Compress files on a Mac](#).
  - [Compress files on Windows](#).
3. 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 `zip` it again. Make sure you get the new version (or delete the old one first).

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

Well done!

### 2.11.1 Exporting for posit.cloud

You can export your project as a zip file from posit.cloud pretty easily. Just follow the directions on this screenshot:

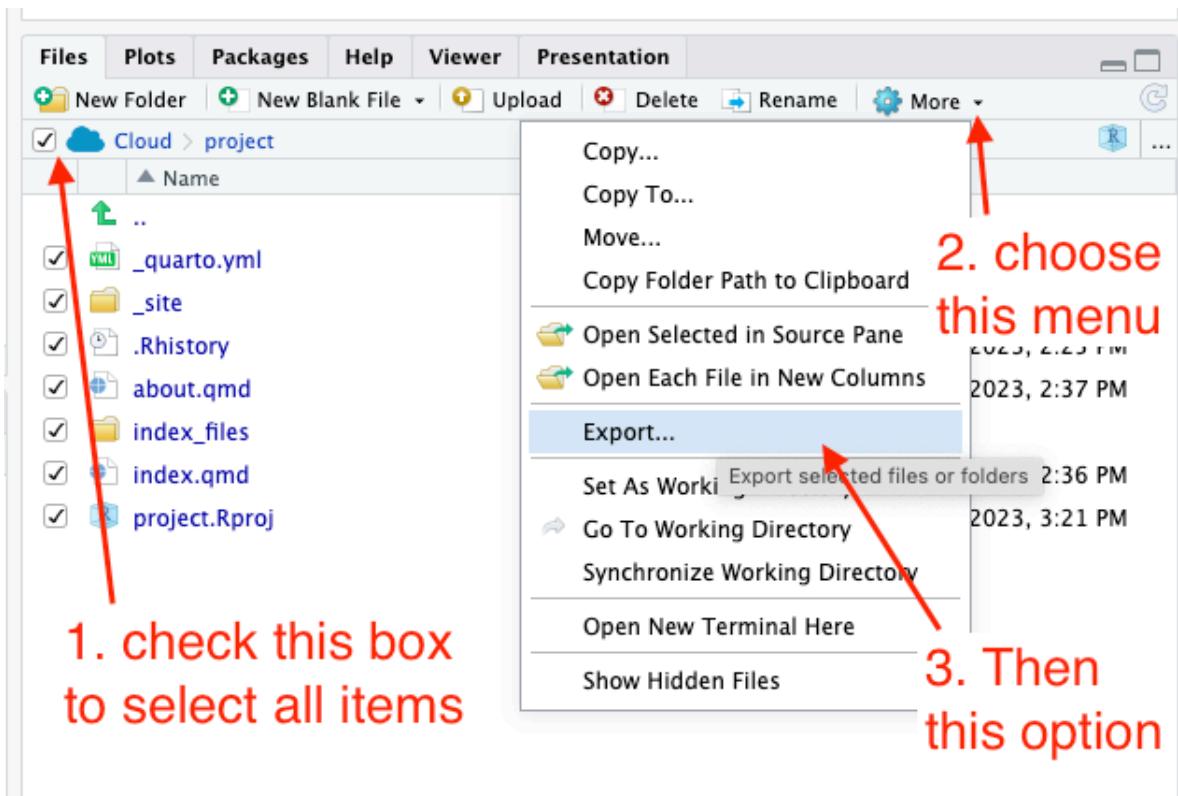


Figure 2.14: Export posit

Then submit the downloaded .zip file to Canvas.

## 2.12 Review of what we've learned so far

- You've learned about the RStudio IDE and how you write and render R code with it.
- You used `install.packages()` to download R packages to your computer. Typically executed from within the Console and only once per computer. We installed a lot of packages including the `tidyverse`.
- You created a Quarto Website and related Quarto documents with code on them.
- You published your work on Quarto Pub to share over the internet.

# **Part I**

# **Counting**

# 3 Billboard Cleaning

This lesson was written by Prof. McDonald and I use macOS.

“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 Posit

## 3.1 Learning goals of this lesson

- Practice organized project setup.
- Learn a little about data types available to R.
- Learn how to download and import CSV files using the `readr` package.
- Introduce the tibble/data frame.
- String functions together with the pipe: `|>` (or `%>%`).
- Learn how to modify data types (`date`) and `select()` columns.

Every project in this book is built around a data source that can birth acts of journalism. For this first one we’ll be exploring the Billboard Hot 100 music charts. You’ll use R skills to find the answer to a bunch of questions in the data and then write about it.

## 3.2 Basic steps of this lesson

Before we get into our storytelling, we have to set up our project, 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 (usually) get it onto on your computer
- Import the data into your project
- Clean up column names and data types
- Export cleaned data for later analysis

So this chapter is about collecting and cleaning data. We'll handle the analysis separately in another chapter. I usually break up projects this way (cleaning vs analyzing) to stay organized and to avoid repeating the same cleaning steps.

### 3.3 Create a new project

We started a new Quarto website in Chapter 2 so you got this! Here are the basic steps for you to follow:

1. Launch RStudio if you haven't already.
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 project in a **New Directory**. For project type, choose **Quarto Website**.
4. Name the project `yourfirstname-billboard` and put it in your `~/Documents/rwd` folder.

#### 3.3.1 Update index with project description

For our projects, we want to use the `index.qmd` file (and our resulting website home page) as a description of the project. This is so you, your editor and anyone you share this with can see what this is about.

To save time and such, I'll supply you with a short project description.

```
---
```

```
title: "Billboard Hot 100"
```

```
--
```

This project is an assignment in the Reporting with Data class at the University of Texas at

It is a study of [Billboard Hot 100] (<https://www.billboard.com/charts/hot-100/>) chart data as

1. Copy the text above and replace the contents of `index.qmd`.
2. Use the **Render** button or *Cmd-Shift-K* to render the page.

#### ! Important

You want each of your projects to have a description like this, along with details describing the source data, the goals of the project, the findings and perhaps links to stories and graphics published based on the data analysis. If you want to see some excellent examples of these summaries, check out the “Repo” links at [Buzzfeed News’ github repo](#). (RIP

Buzzfeed News.)

In the next chapter you'll return to this page to update it with your findings. For now we're done with this file so you can close it.

### 3.3.2 Create directories for your data

Next we are going to create some folders in our project. We'll use some standard names for consistency and we'll do this for every project we build.

The first folder is called `data-raw`. 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 core data journalism commandment: *Thou shalt not change original data*.

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

1. Click on the **New Folder** icon.
2. Name your new folder `data-raw`. This is where we'll put raw data that we don't want to overwrite.
3. Create an additional folder called `data-processed`. This is where we will write any data we create.

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 circular thing at top right of the screenshot below. You might have to widen the pane to see it.)

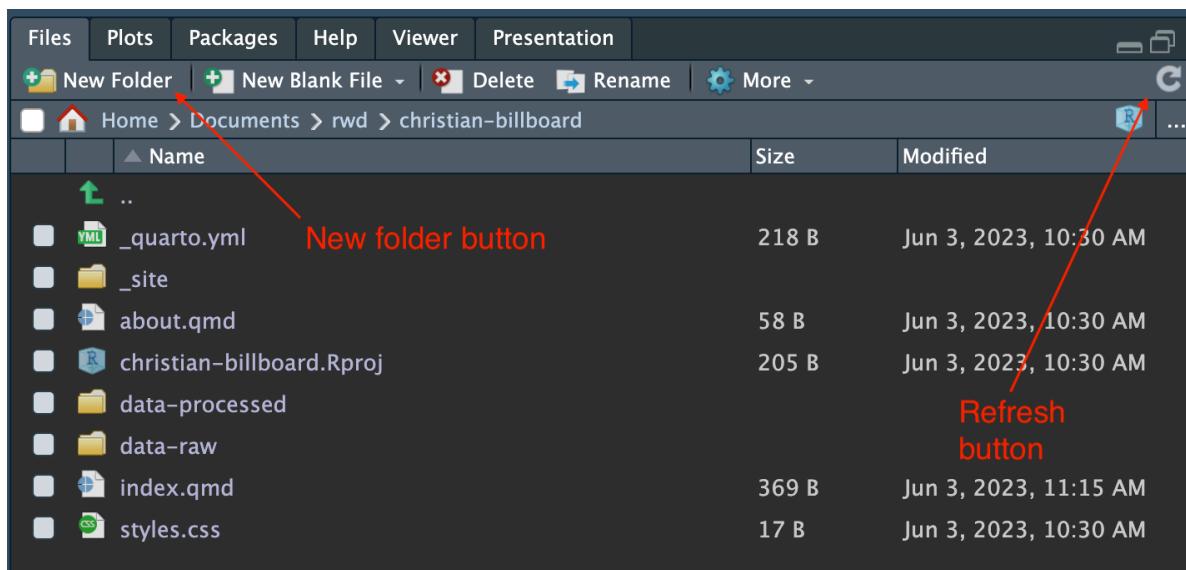


Figure 3.1: Directory made

Again, we'll do this with every new project we create:

- Create a Quarto website
- Update the index with information about the project
- Create `data-raw` and `data-processed` folders.

## 3.4 Create our cleaning notebook

We'll typically use at least three Quarto files in our projects:

- The `index.qmd` that describes the project.
- A “Cleaning” notebook where we prepare our data.
- An “Analysis” notebook where we pose and answer our data questions.

So let's create our “Cleaning” notebook.

1. Use the top left new document button ( a white box with a green + sign) and choose the **Quarto Document** to start a new notebook.
2. Set the title to “Cleaning”.
3. Make sure the **Editor** choice to use the visual editor is **NOT** checked. Everything else should be fine.
4. Click **Create**. This creates an Untitled document that we still need to save.
5. Save the new file (Do *Cmd-S* or look under the File menu or use the floppy disc icon in the tool bar.)
6. Name the file `01-cleaning.qmd`. It should already be set to save in the project folder.



Tip

We named this notebook starting with 01- because we will eventually end up with multiple notebooks that depend on each other and we will need to know the order to run them in the future.

## 3.5 Update our `_quarto.yml`

We've created our cleaning notebook and it will Render, but it won't show up in our website navigation yet until we specifically add it. The `_quarto.yml` config file controls various output options for the “website” that you are creating.

1. Open the `_quarto.yml` file.
2. On the line that now says – `about.qmd`, replace that text with – `01-cleaning.qmd`.

Let's talk a little about the part of the `_quarto.yml` file we edited:

```
website:
  title: "christian-billboard"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - 01-cleaning.qmd
```

Note the indents and dashes and such are important in YAML files. Under the “parent” line `website` we have two children, `title` and `navbar`. The `navbar` line has a child of `left` and that has two children - `href` and - `01-cleaning.qmd`.

When we added the `01-cleaning.qmd` file it automatically added that to the nav, and then used the title of the page as word linked in the navigation.

But for the index, we don’t want to use the title of the page. We have to define what we are doing with more detail, hence the `href` (meaning which page will be linked to) and the `text` to show in the navigation, *Home*.

While we are here, we should adjust some other configurations:

3. For the `title`: change the value to "Billboard Hot 100". This is the main website name that displays in the navigation bar.
4. At the bottom of the file on a new line, by itself and starting flush left, add `df-print: paged`. This makes data we output in our notebooks look more pretty.

Your file should look like this now:

```
project:
  type: website

website:
  title: "Billboard Hot 100"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - 01-cleaning.qmd

format:
  html:
    theme: cosmo
```

```
css: styles.css
toc: true

df-print: paged
```

5. Make sure you `_quarto.yml` file is saved.
6. Go back to your Cleaning notebook and Render it to make there are no errors and that you can see the page in the website navigation.

### 3.5.1 Describe the goals of the notebook

At the top of the cleaning file after the YAML metadata we'll want to explain the goals of the notebook and what we are doing.

1. Add this text to your notebook AFTER the metadata:

```
## Goals of this notebook

The steps we'll take to prepare our data:

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

We want to start each notebook with a list like this so our future selves and others know what we are trying to accomplish. It's not unusual to update this list as we work through the notebook.

We will also write Markdown like this to explain each new “section” or goal as we tackle them.

## 3.6 The setup chunk

In our previous chapter we installed several R packages onto our computer that we'll use in almost every lesson. There are others we'll install and use later.

But to use these packages in our notebook (and the functions they provide us) we have to load them using the `library()` function. We **always** have to declare these and convention dictates we put it near the top of a notebook so everyone understands what is needed to run our code.

### 3.6.1 Load the libraries

1. In your notebook after the goals listing, create a new “section” by adding a headline called “Setup” using Markdown, like this: `## Setup`.
2. After the headline use *Cmd+option+i* to insert an R code chunk.
3. Inside that chunk, I want you **to type** the code shown below.

I want you to **type the code** so you can see how RStudio helps you complete the code. It’s something you have to see or do yourself to understand how RStudio helps you type commands, but as you type RStudio will give you valid options you can scroll through and hit `tab` key to choose.

Here is a gif of me typing in the commands. I’m using keyboard commands like the up and down arrow to make selections, and the `tab` key to select them. This concept is called code completion.

Gif removed for printing

Here is the code:

```
```{r}
#| label: setup
#| message: false

library(tidyverse)
```
```

Warning: package 'ggplot2' was built under R version 4.3.1

```
```{r}
#| label: setup
#| message: false

library(janitor)
```
```

### 3.6.2 About the libraries

A little more about the two packages we are loading here. We use them a lot.

- The `tidyverse` package is actually a collection of packages for data science that are designed to work together. You can see the first time I run the chunk in the gif above that a bunch of libraries were loaded. By loading the whole tidyverse library we get `readr` functions for importing data, `dplyr` to manipulate data, `lubridate` to help work with dates, and `ggplot` to visualize data. [There are more](#).
- The [janitor package](#) is not maintained by the same folks at Posit, but it has a couple of useful tools to clean and view data I use a lot, including the `clean_names()` that we'll use here.

### 3.6.3 About the options

I then go back and add some lines at the top of the chunk called **execution options**. Let's talk about each line:

- We start with `#| label: setup` which is an execution option to “name” the code chunk. Labeling chunks is optional but useful.
  - The chunk label `setup` is special as it will be run first if it hasn’t already been run.
  - When we name a code chunk it creates a bookmark of sorts, which I’ll show you later.
- The `#| message: false` option suppress the message about all the packages loaded since we don’t need to display that in our notebook. I usually only add that for my setup chunk.

 Tip: good stopping point

We’ve just done a lot of work setting up our project, and we’re about to get into some important info about our the data we’ll be working with. This would be a good place to take a break if you need to. Seriously, go eat a cookie

## 3.7 About the Billboard Hot 100

The [Billboard Hot 100](#) singles charts has been the music industry’s standard record chart since its inception on Aug. 4th, 1958. The rankings, published by Billboard Magazine, are currently based on sales (physical and digital), radio play, and online streaming. The methods and policies of the chart have changed over time.

The data we will use was compiled by Prof. McDonald from a [number of sources](#). When you write about this data (and you will), you should source it as **the Billboard Hot 100 from Billboard Magazine**, since that is where it originally came from and they are the “owner” of the data.

### 3.7.1 Data dictionary

Take a look at the [current chart](#). Our data contains many (but not quite all) of the elements you see there. **Each row of data (or observation as they are known in R) represents a song and the corresponding position on that week's chart.** Included in each row are the following columns (a.k.a. variables):

- CHART DATE: The release date of the chart
- THIS WEEK: The current ranking as of the chart date
- TITLE: The song title
- PERFORMER: The performer of the song
- LAST WEEK: The ranking on the previous week's chart
- PEAK POS.: The peak rank the song has reached as of the chart date
- WKS ON CHART: The number of weeks the song has appeared as of the chart date

### 3.7.2 Let's download our data

Our data is stored on a code sharing website called Github, and it is formatted as a “comma-separated value” file, or `.csv`. That means it basically a text file where every line is a new row of data, and each field is separated by a comma.

Because it is on the internet and has a URL, we could import it directly into our project from there, but there are benefits to getting the file onto your computer first. The hosted file could change later in ways you can't control, and you would need Internet access to the file the next time you ran your notebook. If you download the file to keep your own copy, you have more control of your future.

Since this is a new “section” of our cleaning notebook, we'll note what we are doing and why in Markdown.

1. Add a Markdown headline `## Downloading data` and some text explaining you are downloading data. **Add a note about where the data comes from and include a link to the original source.** (This way others and future self will know where the data came from.)
2. Create an R chunk and copy/paste the following inside it. (Given the long URL, go ahead and use the copy icon at the top right of the chunk):

```
download.file(  
  "https://github.com/utdata/rwd-billboard-data/blob/main/data-out/hot100_assignment.csv?raw=true",  
  "data-raw/hot100_assignment.csv",  
  mode = "wb"  
)
```

Let's explain about this:

This `download.file()` code is a function, or a bit of code that has been written to do a specific task. This one (surprise!) downloads files from the internet. As a function, it has some “arguments”, which are expected pieces of information that the function needs to do its job. Sometimes we just enter the arguments in the order that the function expects them, like we do here. Other times we will give it the kind of argument like `mode = "w"`. You can search for documentation about functions in the Help tab in the bottom-right pane of RStudio.

In this case, we are supplying three arguments:

- The first is the URL of the file you are downloading. Note that this is in quotes.
- The second is the location where we want to save the file, and what want to name it. We call this a “path” and you can see it looks a lot like a URL. This path is in relation to the document we are in, so we are saying we want to put this file in the `data-raw/` folder with a name of `hot100_assignment.csv`, but it is really one path: `data-raw/hot100_assignment.csv`.
- The third argument `mode = "w"` gets into the weeds a little, but it helps Windows computers understand the file.

```
trying URL 'https://github.com/utdata/rwd-billboard-data/blob/main/data-out/hot100_assignment.csv'
Content type 'text/plain; charset=utf-8' length 18364249 bytes (17.5 MB)
=====
downloaded 17.5 MB
```

That's not a small file at 17 MB and 300,000 rows, but it's not a huge one, either.

You can check that this worked by going to your Files tab and clicking on the `data-raw` folder to go inside it and see if the file is there. To return out of the folder, click on the two dots to the right of the green up arrow.

gif removed for printing

### 3.7.3 Comment the download code

**!** Important

The Hot 100 data updates each week. We'll not download it again so our results don't change mid-analysis!

Now that we've downloaded the data to our computer, we don't need to run this line of code again unless we want to update our data from the source. We can “comment” the code to preserve it but keep it from running again if we re-run our notebook (and we will).

2. Above your code chunk, write in Markdown a note to your future self that you commented the download for now.
3. Inside your code chunk, highlight the lines of code that include the `download.file()` function, then go under the **Code** menu to **Comment/Uncomment Lines**. (Note the keyboard command there *Cmd-Shift-C*, as it is another useful one!)

Adding comments in programming is a common thing and every programming language has a way to do it. It is a way for the programmer to write notes to their future self, colleagues or — like in this case — comment out some code that you want to keep, but don't want to execute when the program is run.

We write most of our “explaining” text *outside* of code chunks using a syntax called [Markdown](#). Markdown is designed to be readable as written, but it is given pretty formatting when “printed” as a PDF or HTML file.

But, sometimes it makes more sense to explain something right where the code is being executed. If you are *inside a code chunk*, you start the comment with one or more hashes `#`. Any text on that line that follows won't be executed as code.

 Warning

Yes, it is confusing that in Markdown you use hashes `#` to make headlines, but in code chunks you use `#` to make comments. We're essentially writing in two languages in the same document.

## 3.8 Import the data

Now that we have the data on our computer, let's import it into our notebook so we can see it.

Since we are doing a new thing, we should again note what we are doing.

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 following code into your chunk and run it.

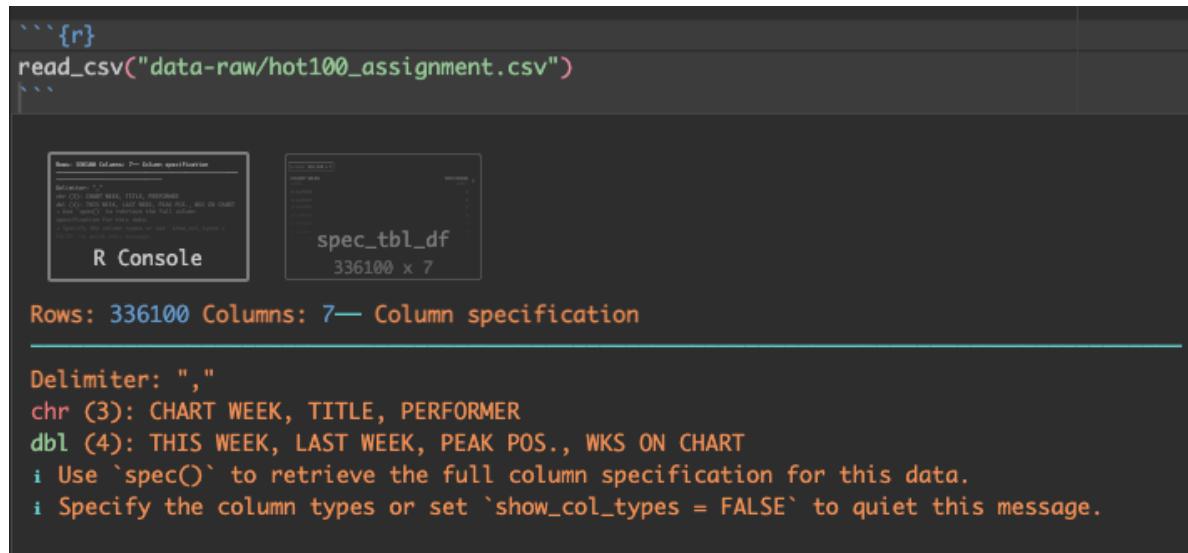
```
read_csv("data-raw/hot100_assignment.csv")
```

 Tip

Note the path to the file is in quotes!

You get two results printed to your notebook.

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 the way that expected. In this case it noted which columns came in as text (**chr**), or numbers (**dbl**). The red colored text in this output is NOT an indication of a problem.



A screenshot of an RStudio interface. On the left, the R Console window shows the command `read_csv("data-raw/hot100_assignment.csv")`. To its right is a `spec_tbl_df` object, which is a tibble with 336100 rows and 7 columns. Below the console, the output of the command is displayed:

```
Rows: 336100 Columns: 7— Column specification

Delimiter: ","
chr (3): CHART WEEK, TITLE, PERFORMER
dbl (4): THIS WEEK, LAST WEEK, PEAK POS., WKS ON CHART
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.
```

Figure 3.2: RConsole output

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”.

 Note

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.

```

```{r}
read_csv("data-raw/hot100_assignment.csv")
```

```

A tibble: 336,100 × 7

| CHART WEEK | THIS WEEK | TITLE                                      |
|------------|-----------|--------------------------------------------|
| 1/1/2022   |           | 1 All I Want For Christmas Is You          |
| 1/1/2022   |           | 2 Rockin' Around The Christmas Tree        |
| 1/1/2022   |           | 3 Jingle Bell Rock                         |
| 1/1/2022   |           | 4 A Holly Jolly Christmas                  |
| 1/1/2022   |           | 5 Easy On Me                               |
| 1/1/2022   |           | 6 It's The Most Wonderful Time Of The Year |
| 1/1/2022   |           | 7 Last Christmas                           |
| 1/1/2022   |           | 8 Feliz Navidad                            |
| 1/1/2022   |           | 9 Stay                                     |
| 1/1/2022   |           | 10 Sleigh Ride                             |

1–10 of 336,100 rows | 1–3 of 7 columns

Figure 3.3: Data output (IGNORE FILENAME)

When we look at the data output in 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.

At this point we have only printed this data to the screen. We have not saved it in any way, but that is next.

### 3.8.1 Assign our import to an R object

Now that we know how to find our data, we next need to assign it to an **R object** so it can be a named thing we can reuse. We don't want to re-import the data over and over each time we need it.

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 put it in your notebook.  
new_object <- stuff_going_into_object
```

Think of it like this: You must have a bucket before you can fill it with water. We “name” the bucket, then fill it with data. That bucket is then saved into our “environment”, meaning it is in memory where we can access it easily by calling its name.

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/hot100_assignment.csv")
```

Run that chunk and several things happen:

- We no longer see the result of the data in the notebook because we created an object instead of printing it.
- 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 `hot100` and it will expand to show you a summary of the columns.
  - Click on the name **hot100** 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.

### 3.8.2 Print a peek to your R Notebook

Since we can’t see the data after we assign it, let’s print the object to our notebook so we can refer to it.

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

```
# create the object, then fill it with data from the csv  
hot100 <- read_csv("data-raw/hot100_assignment.csv")  
  
# peek at the data  
hot100
```

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

### 3.8.3 Glimpse the data

There is another way to peek at the data that I use a lot 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/hot100_assignment.csv")

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

```
Rows: 341,300
Columns: 7
$ `CHART WEEK`    <chr> "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022"
$ `THIS WEEK`     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ~
$ TITLE           <chr> "All I Want For Christmas Is You", "Rockin' Around The ~
$ PERFORMER       <chr> "Mariah Carey", "Brenda Lee", "Bobby Helms", "Burl Ives~
$ `LAST WEEK`     <dbl> 1, 2, 4, 5, 3, 7, 9, 11, 6, 13, 15, 17, 18, 0, 8, 25, 1~
$ `PEAK POS.`     <dbl> 1, 2, 3, 4, 1, 5, 7, 6, 1, 10, 11, 8, 12, 14, 7, 16, 12~
$ `WKS ON CHART` <dbl> 50, 44, 41, 25, 11, 26, 24, 19, 24, 15, 31, 18, 14, 1, ~
```

The `glimpse` shows there are 300,000+ rows and 7 columns in our data. Each column is then listed out with its data type and the first several values in that column.

### 3.8.4 About the pipe `|>`

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

We are starting with the object `hot100`, but then we follow it with `|>`, which is called a pipe. The pipe is a construct that takes the **result** of an object or function and passes it into another function. Think of it like a sentence that says “**AND THEN**” the next thing.

Like this:

```
I woke up |>
got out of bed |>
dragged a comb across my head
```

You can't start a new line with a pipe. If you are breaking your code into multiple lines, then the `|>` needs to be at the end of a line and the next line should be indented so there is a visual clue it is related to line above it, like this:

```
hot100 |>  
  glimpse()
```

It might look like there are no arguments inside `glimpse()`, but what we are actually doing is passing the `hot100` tibble into it like this: `glimpse(hot100)`. For almost every function in R the first argument is “what data are you taking about?” The pipe allows us to say “hey, take the data we just mucked with (i.e., the code before the pipe) and use that in this function.”

### 💡 Tip

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

## A rabbit dives into a pipe

The concept of the pipe was first introduced by tidyverse developers in 2014 in a package called `magrittr`. They used the symbol `%>%` as the pipe. It was so well received the concept was written directly into base R in **2021**, but using the symbol `|>`. Hadley Wickham's 2022 rewriting of [R for Data Science](#) uses the base R pipe `|>` by default. You can configure which to use in RStudio.

(This switch to `|>` is quite recent so you might see `%>%` used in this book. **Assume `|>` and `%>%` are interchangeable.** There is A LOT of code in the wild using the `magrittr` pipe `%>%`, so you'll find many references on Stack Overflow and elsewhere.)

## 3.9 Cleaning data

Data is dirty. Usually because a human was involved at some point, and we humans are fallible.

Data problems are often revealed when importing so it is good practice to check for problems and fix them right away. We'll face some of those challenges in this project, but we should talk about what is good vs dirty data.

Good data should:

- Have a single header row with well-formed column names.
  - Descriptive names are better than not descriptive.

- Short names are better than long ones.
- Spaces in names make them harder to work with. Use and `_` or `.` between words.  
I prefer `_` and all lowercase text.
- Remove notes or comments from the files.
- Each column should have the same kind of data: numbers vs words, etc.

### 3.9.1 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/hot100_assignment.csv") %>% clean_names()
```

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

```
Rows: 341,300
Columns: 7
$ chart_week    <chr> "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022-
$ this_week     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17-
$ title         <chr> "All I Want For Christmas Is You", "Rockin' Around The Ch-
$ performer     <chr> "Mariah Carey", "Brenda Lee", "Bobby Helms", "Burl Ives", ~
$ last_week      <dbl> 1, 2, 4, 5, 3, 7, 9, 11, 6, 13, 15, 17, 18, 0, 8, 25, 19, ~
$ peak_pos       <dbl> 1, 2, 3, 4, 1, 5, 7, 6, 1, 10, 11, 8, 12, 14, 7, 16, 12, ~
$ wks_on_chart   <dbl> 50, 44, 41, 25, 11, 26, 24, 19, 24, 15, 31, 18, 14, 1, 49~
```

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 when you are writing code. It makes type-assist work better and you can now double-click on a column name to select all of it and copy and paste somewhere else. When you have spaces or dashes in an object you can't double-click on it to select all of it.

### 3.9.2 Fixing the date

Dates in programming are a tricky data type because they are represented essentially as the number of seconds before/after January 1, 1970. Yes, that's crazy, but it is also cool because that allows us to do math on them. So, to use our `chart_date` properly in R we need to convert it from the text into a real *date* datatype. (If you wish, you can [read more about why dates are tough in programming | PDF version.](#))

Converting text into dates can be challenging, but the tidyverse universe has a package called `lubridate` to ease the friction. (Get it?).

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

1. In Markdown 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 will be **changing or creating** our data, so we will create a **new object** to store it in. We do this so we can go back and reference the *unchanged* data if we need to. Because of this, we'll set up a chunk of code that allows us to peek at what is happening while we write our code. We'll do this kind of setup often when we are working out how to do something in code.

1. Add the following inside your code chunk.
2. Run the code, and then read through the annotations below.

#### Tip

This is our first use of annotated code with the circled numbers shown on the right edge of the code block. If you see these, the best way to use the code in the chunk is to a) type it yourself (preferred!), b) to highlight and copy a line without catching the circled number, or c) to use the “Copy to clipboard” icon that shows at the top-right of the code block when you put your cursor over it.

```
# part we will build upon  
hot100_date <- hot100  
  
# peek at the result  
hot100_date |> glimpse()
```

①  
②  
③  
④  
⑤

- ① I have a comment starting with `#` to explain the first part of the code.  
② We created a new object (or “bucket”) called `hot100_date` and we fill it with our `hot100` data. Yes, as of now they are exactly the same.

- ③ I leave a blank line for clarity ...
- ④ Then another comment ...
- ⑤ Then we glimpse the new `hot100_date` object so we can see changes as we work on it.

```
Rows: 341,300
Columns: 7
$ chart_week    <chr> "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022~
$ this_week     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17~
$ title         <chr> "All I Want For Christmas Is You", "Rockin' Around The Ch~
$ performer     <chr> "Mariah Carey", "Brenda Lee", "Bobby Helms", "Burl Ives", ~
$ last_week     <dbl> 1, 2, 4, 5, 3, 7, 9, 11, 6, 13, 15, 17, 18, 0, 8, 25, 19, ~
$ peak_pos      <dbl> 1, 2, 3, 4, 1, 5, 7, 6, 1, 10, 11, 8, 12, 14, 7, 16, 12, ~
$ wks_on_chart <dbl> 50, 44, 41, 25, 11, 26, 24, 19, 24, 15, 31, 18, 14, 1, 49~
```

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

### 3.9.2.1 Working with `mutate()`

We are going to use the text of our date field `chart_date` to create a new converted date. We will use the dplyr function `mutate()` to do this, with some help from lubridate.

 Note

`dplyr` is the tidyverse package of functions to manipulate data. We'll use functions from it a lot. Dplyr is loaded with the `library(tidyverse)` so you don't have to load it separately.

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

Within the `mutate` function, we name the new thing first (the bucket!) and then fill it with the new value.

```
# This is just explanatory psuedo code
# You don't need this in your notebook
data |>
  mutate(
    newcol = new_stuff_from_math_or_whatever
  )
```

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.

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 upon
hot100_date <- hot100 |>
  mutate(
    chart_date = mdy(chart_week)
  )

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

- ① At the end of the first line, we added the pipe `|>` because we are taking our `hot100` data **AND THEN** we will mutate it. (Remember *Cmd-shift-m* for the pipe!)
- ② Next, we start the `mutate()` function. If you use type assist and tab completion to type this in, your cursor will end up in the middle of the parenthesis. This allows you to then hit your **Return** key to split it into multiple lines with proper indenting. We do this so we can more clearly see what *inside* the `mutate ...` where the real action is going on here. It’s also possible to make multiple changes within the same `mutate`, and putting each one on their own line makes that more clear.
- ③ Inside the `mutate`, we first name our new column `chart_date` and then we set that equal to `mdy(chart_week)`, which is explained next.

```
Rows: 341,300
Columns: 8
$ chart_week   <chr> "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022", "1/1/2022~
$ this_week    <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17~
$ title        <chr> "All I Want For Christmas Is You", "Rockin' Around The Ch~
$ performer    <chr> "Mariah Carey", "Brenda Lee", "Bobby Helms", "Burl Ives", ~
$ last_week    <dbl> 1, 2, 4, 5, 3, 7, 9, 11, 6, 13, 15, 17, 18, 0, 8, 25, 19, ~
$ peak_pos     <dbl> 1, 2, 3, 4, 1, 5, 7, 6, 1, 10, 11, 8, 12, 14, 7, 16, 12, ~
$ wks_on_chart <dbl> 50, 44, 41, 25, 11, 26, 24, 19, 24, 15, 31, 18, 14, 1, 49~
$ chart_date   <date> 2022-01-01, 2022-01-01, 2022-01-01, 2022-01-01, 2022-01~~
```

The `mdy()` function is part of the `lubridate` package. Lubridate allows us to parse text and then turn it into a real date *if* we tell it the order of the date values in the original data.

- Our original date was something like “7/17/1965”. That is month, followed by day, followed by year.

- We use the lubridate function `mdy()` to say “that’s the order this text is in, now please convert this into a *real date*”, which properly shows as YYYY-MM-DD. Lubridate is smart enough to figure out if you have / or - between your values in the original text.

If your original text is in a different date order, then you look up what lubridate function you need to convert it. I typically use the **cheatsheet** in the [lubridate documentation](#). You’ll find them in the PARSE DATE-TIMES section.

### 💡 Tip

A note about the spacing and indenting of my code above: 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 as you type. (Easier to show than explain.)

#### 3.9.2.2 Check the result!

This new `chart_date` column is added as the LAST column of our data. After doing any kind of mutate you want to check the result to make sure you got the results you expected, which is why we didn’t just overwrite the original `chart_week` column. That’s also why we built our code this way with `glimpse()` so we can see example of our data from both the first and the last column. (We’ll rearrange all the columns in a bit once we are done cleaning everything.)

Check your glimpse returns ... did your dates convert correctly?

#### 3.9.3 Arrange the data

Just to be tidy, we want ensure our data is arranged to start with the oldest week and “top” of the chart and then work forward through time and rank.

i.e., let’s arrange this data so that the oldest data is at the top.

Sorting data is not a particularly difficult concept to grasp, but it is one of the [Basic Data Journalism Functions](#), so watch this video:

<https://vimeo.com/435910315>

In R, the sorting function we use is called `arrange()`.

We’ll build upon our existing code and use the pipe `|>` to push it into an `arrange()` function. Inside `arrange` we’ll feed it the columns we wish to sort by.

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

```

# part we will build upon
hot100_date <- hot100 |>
  mutate(
    chart_date = mdy(chart_week)
  ) |>
  arrange(chart_date, this_week)

# peek at the result
hot100_date |> glimpse()

```

- ① Add the pipe to the end of this line ...  
 ② ... and the arrange line

```

Rows: 341,300
Columns: 8
$ chart_week   <chr> "8/4/1958", "8/4/1958", "8/4/1958", "8/4/1958", "8/4/1958~"
$ this_week    <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17~
$ title        <chr> "Poor Little Fool", "Patricia", "Splish Splash", "Hard He~
$ performer    <chr> "Ricky Nelson", "Perez Prado And His Orchestra", "Bobby D~
$ last_week    <dbl> NA, N~
$ peak_pos     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17~
$ wks_on_chart <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ chart_date   <date> 1958-08-04, 1958-08-04, 1958-08-04, 1958-08-04, 1958-08~

```

Now when you look at the glimpse, the first record in the `chart_date` column is from “1958-08-04” and the first in the `this_week` 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 Markdown text in your notebook explaining your are looking at the table.
2. Add a new code chunk and add the following.

```
hot100_date |> head(10)
```

```

# A tibble: 10 x 8
  chart_week this_week title           performer last_week peak_pos wks_on_chart
  <chr>       <dbl> <chr>          <chr>      <dbl>      <dbl>      <dbl>
1 8/4/1958      1 Poor Little F~ Ricky Ne~      NA        1         1
2 8/4/1958      2 Patricia       Perez Pr~      NA        2         1
3 8/4/1958      3 Splish Splash  Bobby Da~      NA        3         1
4 8/4/1958      4 Hard Headed W~ Elvis Pr~      NA        4         1

```

```

5 8/4/1958      5 When          Kalin Tw~      NA      5      1
6 8/4/1958      6 Rebel-'rouser Duane Ed~      NA      6      1
7 8/4/1958      7 Yakety Yak    The Coas~      NA      7      1
8 8/4/1958      8 My True Love  Jack Sco~      NA      8      1
9 8/4/1958      9 Willie And Th~ The John~      NA      9      1
10 8/4/1958     10 Fever        Peggy Lee      NA     10      1
# i 1 more variable: chart_date <date>

```

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).

#### Note

It's OK that the `last_week` column has "NA" for those first rows because this is the first week ever for the chart. There was no `last_week`.

#### 3.9.3.1 Getting summary stats

Printing your data to the notebook can only tell you so much. Yes, you can arrange by different columns to see the maximum and minimum values, but it's hard to get an overall sense of your data that way when there is 300,000 rows like we have here. Luckily there is a nice function called `summary()` that gives you some summary statistics for each column.

1. Add some Markdown text that you'll print summary stats of your data.
2. Add a new R chunk and put the following in and run it

```
hot100_date |> summary()
```

| chart_week       | this_week      | title            | performer          |
|------------------|----------------|------------------|--------------------|
| Length:341300    | Min. : 1.0     | Length:341300    | Length:341300      |
| Class :character | 1st Qu.: 26.0  | Class :character | Class :character   |
| Mode :character  | Median : 51.0  | Mode :character  | Mode :character    |
|                  | Mean : 50.5    |                  |                    |
|                  | 3rd Qu.: 75.0  |                  |                    |
|                  | Max. :100.0    |                  |                    |
| last_week        | peak_pos       | wks_on_chart     | chart_date         |
| Min. : 0.00      | Min. : 1.00    | Min. : 1.000     | Min. :1958-08-04   |
| 1st Qu.: 23.00   | 1st Qu.: 13.00 | 1st Qu.: 4.000   | 1st Qu.:1974-12-14 |

```

Median : 47.00   Median : 38.00   Median : 7.000   Median :1991-04-20
Mean    : 47.29   Mean   : 40.69   Mean   : 9.288   Mean    :1991-04-19
3rd Qu.: 71.00   3rd Qu.: 65.00   3rd Qu.:13.000   3rd Qu.:2007-08-25
Max.    :100.00   Max.    :100.00   Max.    :91.000   Max.    :2023-12-30
NA's    :32460

```

These summary statistics can be informative for us. It is probably the easiest way to check what the newest and oldest dates are in your data (see the Min. and Max. returns for `chart_date`). You get an average (mean) and median for each number, too. You might notice potential problems in your data, like if we had a `this_week` number higher than “100” (we don’t).

### 3.9.3.2 Note the max date

Since this data updates each week, it is a good idea to note in your notebook what is the most recent chart date. You can find that as the `Max.` value for `chart_date` in your summary. You’ll need this value when you write your story.

### 3.9.4 Selecting columns

Now that we have the fixed date column, we don’t need the old `chart_week` version that is text. We’ll use this opportunity to discuss `select()`, which is another concept in our Basic Data Journalism Functions series, so watch this:

<https://vimeo.com/435910324>

In R, the workhorse of the select concept is the function called — you guessed it — `select()`. In short, the function allows us to choose a subset of our columns. We can either list the ones we want to keep or the ones we don’t want.

Like a lot of things in R, `select()` is both easy and complex. It’s really easy to just list the columns you want to keep. But `select()` can also be very powerful as you [learn more options](#).

Let’s just add one complexity: We’ll **rename some columns** as we list them.

1. Add a Markdown headline: `## Selecting columns`.
2. Explain in text we are dropping the text date column and renaming others.
3. Add the code below and then I’ll explain it. We again are setting this up to create a new object and view the changes.

```

hot100_clean <- hot100_date |>
  select(
    chart_date,
    current_rank = this_week,
    title,
    performer,
    previous_rank = last_week,
    peak_rank = peak_pos,
    wks_on_chart
  )

hot100_clean |> glimpse()

```

- ① Name our new object `hot100_clean` and fill it with the result of `hot100_date` and then include the modifications that follow.
- ② We start the `select()` function to list the columns to keep
- ③ We start with `chart_date` which is our *real date*. We just won't ever name the text version so we won't keep it.
- ④ When we get to our `this_week` column, we rename it to `current_rank`. We're doing this because we'll rename all the “ranking” columns with something that includes `_rank` at the end. (While this is good practice, the reasons get into the weeds).
- ⑤ Lastly, we glimpse the new object to check it.

```

Rows: 341,300
Columns: 7
$ chart_date    <date> 1958-08-04, 1958-08-04, 1958-08-04, 1958-08-
$ current_rank  <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1-
$ title         <chr> "Poor Little Fool", "Patricia", "Splish Splash", "Hard H-
$ performer     <chr> "Ricky Nelson", "Perez Prado And His Orchestra", "Bobby ~
$ previous_rank <dbl> NA, ~
$ peak_rank     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1-
$ wks_on_chart  <dbl> 1, ~

```

There are other ways to accomplish the same thing, but this works for us.

## 3.10 Exporting data

### 3.10.1 Using multiple notebooks

It is good practice to separate the cleaning of your data from your analysis. By doing all our cleaning at once and exporting it, we can use that cleaned data over and over in future

notebooks.

Because each notebook needs to be self-contained to render, we will export our cleaned data in a native R format called `.rds` (maybe stands for R data storage?). This format preserves all our data types so we won't have to reset or clean them.

 Note

This is one of the reasons I had you name this notebook `01-cleaning.qmd` with a `01` at the beginning, so we know to run this one before we can use the notebook `02-analysis.qmd` (next lesson!). 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 like that, which is a good trait for a data journalist.

One last thought to belabor the point: Separating your cleaning can save time. I've had 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 a 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.

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 avoiding our `data-raw` folder because **“Thou shalt not change original data”** even by accident.

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_clean |>  
  write_rds("data-processed/01-hot100.rds")
```

So, here we are starting with the `hot100_clean` 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 know where to save the file, so in quotes we give it the path to where we want to save the file: `"data-processed/01-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.

### 3.10.2 About paths

When I say “path to where we want to save the file” what I’m talking about is the folder structure on your computer. The frame of reference is where your notebook is stored, which is typically in your project folder. When we want to reference other files from within our notebook, we have to provide the **path** (or folder structure) to where that file is, including its name. We use / in the path to designate we are going inside a folder.

Here is the file structure inside your project folder:

```
01-cleaning.qmd
02-analysis.qmd
_quarto.yml
christian-billboard.Rproj
data-processed
  01-hot100.rds
data-raw
  hot100_assignment.csv
index.qmd
```

To get from `01-cleaning.qmd` we need to provide a **relative** path to where we are saving the file, including the folder name and the name of the file: `data-processed/01-hot100.rds`.

We’ll need to do a similar thing when import data in the next notebook.

## 3.11 Bookmarks

I didn’t want to break our flow of work to explain this earlier, but I want you show you a nice feature in RStudio to jump up and down your notebook.

1. 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 will be different:

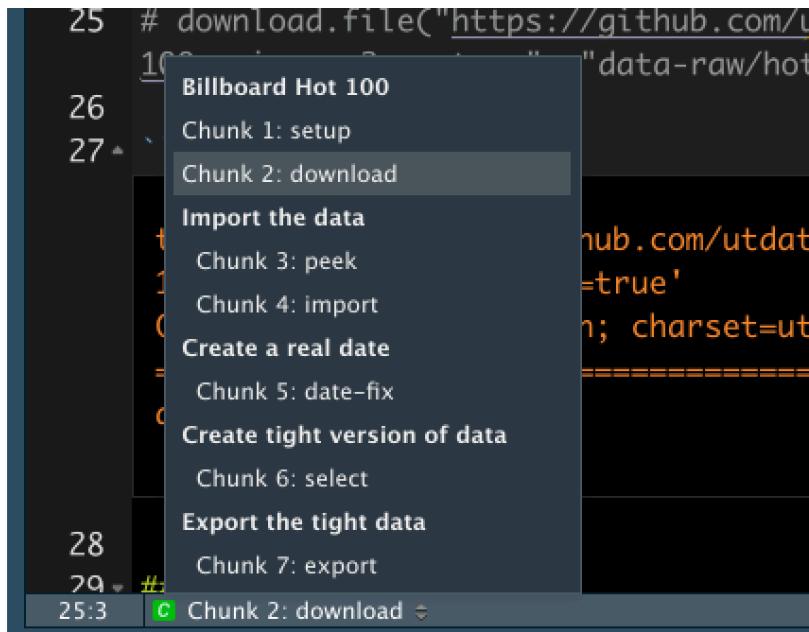


Figure 3.4: RStudio bookmarks

You'll notice that each Markdown headline you have is listed. My chunks also have names, but yours probably don't. It's optional to name chunks with a label but it helps you find them through the bookmarks. In addition, plots produced by the chunks will have useful names that make them easier to reference elsewhere, but that's later lesson in the semester.

Here is how you can name a chunk by using “[execution options](#)”, which can control your code output. (This example shows all of the R code chunk.)

```
```{r}
#| label: select-rows

hot100_clean |> select(title, performer) |> head()
```
```

```
# A tibble: 6 x 2
  title           performer
  <chr>          <chr>
1 Poor Little Fool Ricky Nelson
2 Patricia       Perez Prado And His Orchestra
3 Splish Splash  Bobby Darin
4 Hard Headed Woman Elvis Presley With The Jordanaires
5 When           Kalin Twins
6 Rebel-'rouser Duane Eddy His Twangy Guitar And The Rebels
```

Your chunk labels should be short but evocative and should not contain spaces. Hadley Wickham recommends using dashes – to separate words (instead of underscores \_) and avoiding other special characters in chunk labels.

You can read more execution options and chunks in [R for Data Science](#).

## 3.12 Render and publish

Lastly, I want you to render your notebook so you can see the pretty HTML version.

1. Click the Render button in the toolbar (or use *Cmd-Shift-k*).

This should open the HTML version of your page in the Viewer pane. Note there is also a button in that View toolbar that will instead open your site in your web browser.

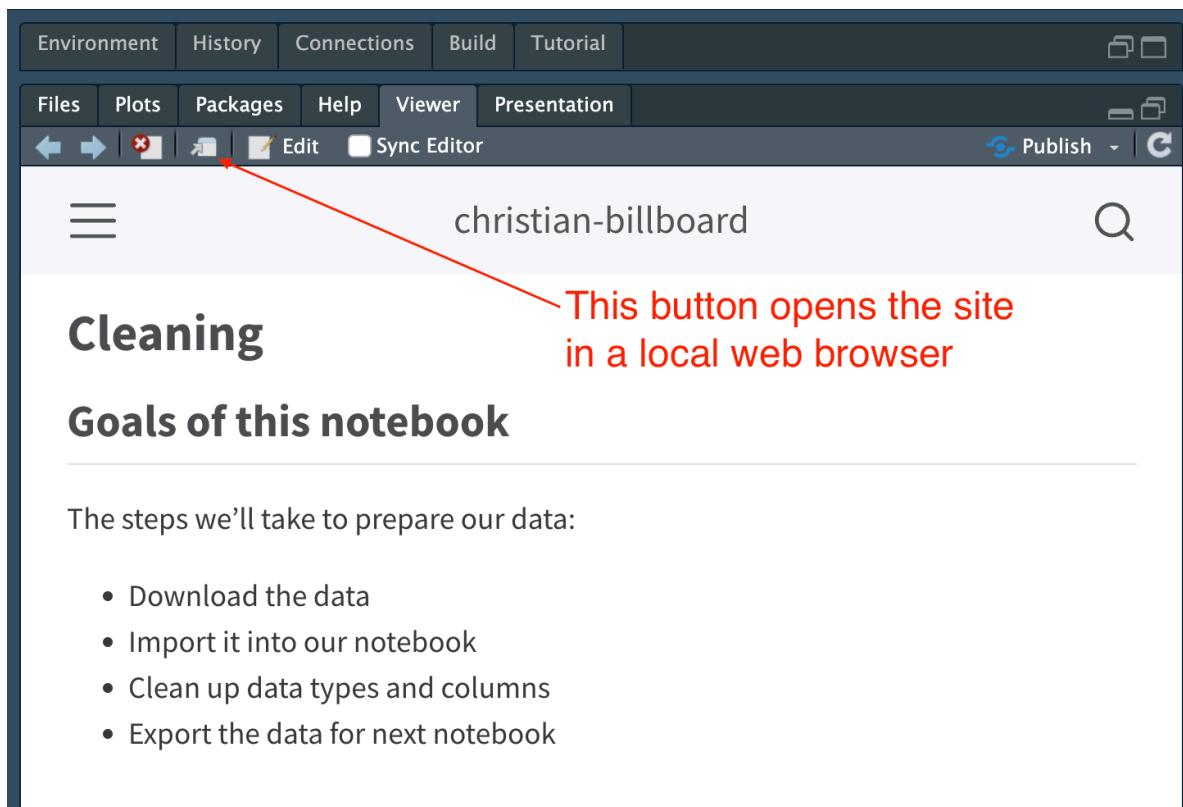


Figure 3.5: Render site

But note this version is only available on your computer. In a minute we'll publish this to Quarto Pub so you have an online version that we'll continue to update.

### 3.12.1 Publish to Quarto pub

Now that we have that in order, let's publish this to the internet. If you published your last project as instructed, it should remember your credentials.

1. In the bottom-left pane of RStudio, click on the pane **Terminal**.
2. Type in `quarto publish`.
3. The first option available (and perhaps the only) should be Quarto Pub. Hit **Return** to choose that.
4. It might know who you are already, but continue through the prompts.

You should end up with your browser open to your new Quarto Pub website.

<https://vimeo.com/833716452>

Well done.

## 3.13 Review of what we've learned so far

Most of this lesson has been about importing and cleaning data, which can sometimes be the most challenging part of a project. Here we were working with well-formed data, but we still used quite a few tools from the tidyverse 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:

- `library()` loads a package so we can use functions within it. We will use at least `library(tidyverse)` in every project.
- `read_csv()` imports a csv file. You want that function with the underscore, 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 differently 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 depending on the way your text is ordered.
- `select()` selects columns from your tibble. You can list all the columns to keep, or use `!` to remove columns. There are many variations.
- `summary()` gives you some quick summary statistics about your data like min, max, mean, median.
- `write_rds()` writes data out to a file in a format that preserves data types.

### **3.14 What's next**

This is part one of a two-chapter project. You might be asked to turn in your progress, or we might wait until both parts are done. Check your class assignments in Canvas.

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.

# 4 Billboard Analysis

This chapter is by Prof. McDonald and keyboard commands and screenshots are from macOS.

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

## 4.1 Goals of this lesson

- To use the group\_by/summarize/arrange combination to count rows of data
- To filter our data in two ways: to focus data before summarizing, and to logically cut summarized lists.
- We'll also cover some more complex filters using and/or logic
- Introduce the shortcut `count()` function

## 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 performer had the most appearances on the Hot 100 chart at any position?
- Which song (title & performer) has been on the charts the most?
- Which song (title & performer) 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 has had the most Top 10 hits overall?

**What are your guesses for the questions above? NO PEEKING!**

In each case we'll talk over the logic of finding the answer and the code to accomplish it.

Before we can get into the analysis, we want to set up a new notebook to separate our cleaning from our analysis.

### ! Important

The data outputs in this book will differ from what you get since the source data is updated every week. This is especially true in videos and gifs.

## 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 Quarto 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 Cleaning notebook is still open, go ahead and close it.
3. Use the `+` menu to start a new **Quarto Document**.
4. Set the title as "Analysis".
5. Save the file as `02-analysis.qmd` in your project folder.
6. Check your Environment tab (top right) and make sure the environment 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**.
7. Also go into your `_quarto.yml` file and add `02-analysis.qmd` on the line after `01-cleaning.qmd` so that this notebook will show up in your website navigation.

### 4.3.1 Add your goals, setup

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 we outlined above into your notebook.
3. Format those questions as a nice list. Start each line with a `-` or `*` followed by a space. There should be a blank line above and below the entire LIST but not between the items. List items should be on sequential lines ... and it is the only markdown item like that.
4. Now add another headline (two hashes) called Setup.
5. Add a chunk, add the option `#| label: setup` and add in the **tidyverse** library.
6. Run the chunk to load the libraries.

```
library(tidyverse)
```

### ! Important

Do you see the grey triangle above with the term ***Try to write the code on your own first*** next to it? That is code that I've written but hidden to give you a chance to write it yourself before you see it. If you click on the little triangle, it will turn and reveal the code!

#### 4.3.2 Import the cleaned data

We need to import the data that we cleaned and exported in the last notebook. It's just a couple of lines of code and you could write them all out and run it, but here is where I tell you for the first of a 1,000 times:

**WRITE ONE LINE OF CODE. RUN IT. CHECK THE RESULTS. REPEAT.**

Yes, sometimes for the sake of brevity I will give you multiple lines of code at once, but to understand what is actually going on you should add and run that code one line at a time.

We'll painstakingly walk through that process here to belabor the point. Here are our goals for this bit:

- Document that we are importing clean data
- Import our cleaned data
- Fill an R object with that data
- Glimpse the R Object so we can check it

I want you to:

1. Start a Markdown section with a headline noting you are importing the cleaned data.  
Add any other text notes you might need for yourself.
2. Add a new code chunk.
3. Inside your code chunk, TYPE IN this line of code:

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

This `read_rds()` function is just like the `read_csv()` function we used in our last notebook to read in data from our computer, except we are reading a different kind of file format: RDS. The argument (in quotes) is the path where our file is on our computer.

What did you expect will happen if you run the code?

**Now run that line of code.**

It should print the results of the data in your notebook. Did it?

If you got an error, you should read that error output for some clues about what happened. When using a `read_` function problems are usually about one of two things: It can't find the function (load the library tidyverse!) or it doesn't understand the path to the file (that you have the folder structure wrong or something misspelled).

 Note

CATCHING COMMON MISTAKES: In the appendices of this book there is a chapter on [Troubleshooting](#) that is worth bookmarking and reading.

Our next goal is to take all that data and put it into a new R object.

1. **Edit** your one line of code to assign the data to an object `hot100`. Remember we will add the new object FIRST (our bucket) and then use the `<-` operator to put the data inside it (the water).
2. Run it!

It should look like this:

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

What happened when you ran the code? Did you get an error? Did you get a result printed to the screen?

OK, that last one was a trick question. When you save data into an object it does NOT print it to your notebook, but it does save that object into your “Environment”, which means it is in memory and can be called and run at any time. You should see `hot100` listed in your Environment tab at the top right of RStudio.

Now, before I started this quest, I knew I wanted that data inside the `hot100` object and could've written it like that from the beginning. But I didn't because most “read” problems are in the path, so I wanted to make sure that function was written correctly first. Also, when I put the data into the object I can't see it until I print it out again, so I do it one piece at a time. You should, too.

So next we'll print the data out to our screen. We do that by just by calling the object.

1. **Edit** your code chunk to add a blank line then the object and run it, like this:

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

```
hot100
```

This should print out the data to your screen so you can see the first couple of columns and the first 10 rows of data.

But what we really need is to see all the column names, the data types and an example of the data. We can get that with our `glimpse()` function.

1. **Edit** the last line of the chunk to add the pipe and the `glimpse()` function
2. Run it.

Your code (and result) should look like this now:

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

```
Rows: 341,300  
Columns: 7  
$ chart_date    <date> 1958-08-04, 1958-08-04, 1958-08-04, 1958-08-04,  
$ current_rank  <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~  
$ title         <chr> "Poor Little Fool", "Patricia", "Splish Splash", "Hard H~  
$ performer     <chr> "Ricky Nelson", "Perez Prado And His Orchestra", "Bobby ~  
$ previous_rank <dbl> NA, ~  
$ peak_rank      <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~  
$ wks_on_chart   <dbl> 1, ~
```

Again, I could've written all that code at once before running it, as I've written code like that for many years. **I still write and run code one line at a time**, and you should, too. It is the easiest way to make sure your code is correct and find problems. So, for the second of a 1,000 times:

**WRITE ONE LINE OF CODE. RUN IT. CHECK THE RESULTS. REPEAT.**

**!** Dealing with pipes

If you are looking at my code and writing it into your notebook, you have to run each line of code BEFORE you add the pipe `|>` at the end of that line. A pipe `|>` must have a function directly following it on the same or next (or same) line to work.

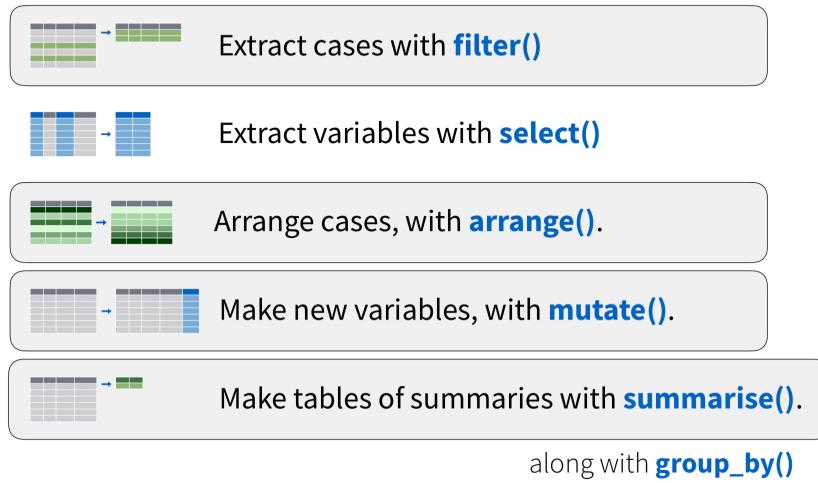
## 4.4 Introducing dplyr

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.

---

## dplyr: Data manipulation



Adapted from 'Master the tidyverse' CC by RStudio

along with `group_by()`



Figure 4.1: Images courtesy Hadley Wickham.

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

We will use these dplyr functions to find the answers to our questions.

### 4.5 Most appearances

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

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

Each row in the data is one song ranked on the chart. It includes a field with the name of the "performer" so we know who recorded it.

**So, to figure out how many times a performer is in the data, we need to count the number of 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: `group_by()` and `summarize()`

### 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.

<https://vimeo.com/435910349>

We'll dive deep into this next.

### 4.5.2 Summarize

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

The `summarize()` function **computes summary tables describing your data**. We're usually finding a single number to describe a column of data, like the "average" of numbers in column.

In our case we want a "summary" about the **number** of times a specific performer appears in data, hence we use `summarize()`.



Tip

**THEY BE THE ZAME:** `summarize()` and `summarise()` are the same function, as R supports both the American and UK spelling of summarize. They work the same and I don't care which you use.

Here is an example of `summarize()` in a different context:

# summarise()

Compute table of summaries.

```
gapminder %>% summarise(mean_life = mean(lifeExp),  
                           min_life = min(lifeExp))
```



Figure 4.2: Learn about your data with summarize()

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.

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

Again, in our case (as we work toward finding the performer with most appearances) we want to summarize the **number** of rows, and there is a function for that: `n()`. (Think “number of observations”.) Every row in the data is an appearance ... we just need to count how many rows have each performer.

But first, to show how this works, we’ll count *all* the rows in our data. Let’s write the code and run it, then I’ll explain:

1. Set up a new section with a Markdown headline, text and explain you are looking for most appearances.
2. Add a named code chunk and add the following:

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

- ① We start with the tibble first and then pipe into `summarize()`.
- ② Within the function, we define our summary: We name the new column “appearances” because that is a descriptive column name for our result, then we set that new column to count the **number** of rows.

```
# A tibble: 1 x 1
  appearances
  <int>
1       341300
```

Basically we are summarizing the total number of rows in the data. When this book was last updated there were **341300** rows.

But I bet you’re thinking: “Yo Professor, you said we want to count the number of times a *performer* has appeared, right?”

This is where we bring in a close friend to `summarize()` ... the `group_by()` function.

### 4.5.3 Group by

The `group_by()` function pre-sorts data into groups so that whatever function follows is applied *within* each of the groups. That means `group_by()` always has something following it, and that something is *usually* `summarize()`.

If we `group_by()` performer *before* we summarize/count our rows, it will put all of the rows with “Aerosmith” together, then all the “Bad Company” rows together, etc. and then it will count the rows *within* those groups ... first those for Aerosmith and then those for Bad Company.

1. **Modify your code block** to add the `group_by` line before `summarize`, then run it:

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

①

- ① Add the `group_by()` line before the `summarize`.

```
# A tibble: 10,742 x 2
  performer           appearances
  <chr>                  <int>
1 "\"Groove\" Holmes"      14
2 "\"Little\" Jimmy Dickens" 10
```

```

3 "\"Pookie\" Hudson"           1
4 "\"Weird Al\" Yankovic"      91
5 "$NOT & A$AP Rocky"          1
6 "'N Sync"                     172
7 "'N Sync & Gloria Estefan"   20
8 "'N Sync Featuring Nelly"    20
9 "'Til Tuesday"                53
10 "(+44)"                      1
# i 10,732 more rows

```

What we get in return is a **summarized** table that shows all 10,000+ different performers that have been on the charts, and the **number** 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 sorted all the rows by the oldest chart date? We'll use the same `arrange()` function here, but we'll change the result to **descending** order, because journalists almost always want to know the *most* of something.

1. Edit your chunk to add the pipe and `arrange` function below and run it, then I'll explain.

```

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

①

- ① 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 wrapped our column in another function: `desc()` to change the order to “descending”, or the most on the top.

```

# A tibble: 10,742 x 2
  performer     appearances
  <chr>            <int>
1 Taylor Swift      1386
2 Drake              896
3 Elton John         889
4 Madonna             857
5 Kenny Chesney       777
6 Tim McGraw          749
```

```
7 Keith Urban      674
8 Morgan Wallen   659
9 Stevie Wonder   659
10 Rod Stewart    657
# i 10,732 more rows
```

**i Note**

I sometimes pipe the column name into the `desc()` function like this:  
`arrange(appearances |> desc())`.

#### 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 (as in hard drive space), so I try to avoid that when I can.

We can use the `head()` command again to get our Top 10. It will give us a specified number of rows at the top of the table (with a default of six if we don't specify.) There is a corresponding `tail()` function to get the last rows.

1. **Edit your code** to pipe the result into `head()` function set to 10 rows.
2. Review the annotations below!

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

- ① We start with the hot100 data AND THEN
- ② We group by the data by performer AND THEN
- ③ We summarize it by counting the number of rows in each group, calling the new column "appearances" AND THEN
- ④ We arrange the result by appearances in descending order AND THEN
- ⑤ We display just the first 10 rows

```
# A tibble: 10 x 2
  performer     appearances
  <chr>           <int>
1 Taylor Swift      1386
```

|                 |     |
|-----------------|-----|
| 2 Drake         | 896 |
| 3 Elton John    | 889 |
| 4 Madonna       | 857 |
| 5 Kenny Chesney | 777 |
| 6 Tim McGraw    | 749 |
| 7 Keith Urban   | 674 |
| 8 Morgan Wallen | 659 |
| 9 Stevie Wonder | 659 |
| 10 Rod Stewart  | 657 |

Since we have our answer here and we're not using the result later, we don't need to create a new object or anything. Printing it to our notebook is sufficient.

So, **Taylor Swift** ... is that who you guessed? A little history here, Swift past Elton John in August 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). And it doesn't hurt that she is re-releasing her entire catalog (Taylor's version)!

#### ! Important

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 done with other performers. So, songs by "Drake" are counted separately than "Drake featuring Future" and even "Future featuring Drake". You'll need to make this clear when you write your data drop in a later assignment. If you include collaborations, Drake has something like 3,150 appearances compared to Taylor's 1,618.

#### 4.5.6 Data Takeaways: Describing your learnings

We've learned something here from the data and we should note it for our future selves in something we call a "data takeaway," which is a prose sentence explaining our finding. You'll do this for each "answer" you find in your data.

1. Add the following sentence after your code chunk

Data Takeaway: Taylor Swift has more appearances on the Billboard Hot 100 than any other artist.

Writing these sentences help you understand what you've found in your data. When you get toward the end of your project, you'll collect these takeaways in your `index.qmd` file.

#### 4.5.7 Render your notebook

Now that you have your first quest answers, let's celebrate by rendering your notebook so you can see all your pretty work.

1. In the toolbar of your notebook, click the **Render** button or use the *Cmd-Shift-k* keyboard command.

This will give you the website view of your page, and you *should* have the page listed in your site navigation if you did the setup properly at the beginning of this chapter.

### 4.6 Song with most appearances

Our next quest is this: **Which song (title & performer) has been on the charts the most?**

This is very similar to our quest to find the performer with the most appearances, but we have to consider both **title** and **performer** together because different artists can perform songs of the same name. For example, Adele's song *Hold On* entered the Hot 100 at 49 in December 2021, but at least 18 other performers have also had a song titled *Hold On* on the Hot 100.

Let's first talk through the logic of what we want to do and how it differed from our first quest:

- We want to count rows where BOTH the **title** and **performer** are the same. We can do this by putting both values in our **group\_by()** function, instead of just one thing. This will put all rows with both "Rush" as a performer AND *Tom Sawyer* as a title into the same group. Rows with "Rush" and *Red Barchetta* will be considered in a different group.
- Then we want to **summarize()** to count the number of rows in each group.
- Once we have a summary table, we'll sort it by appearances in **descending** order to put the highest value on the top.

Let's write the code.

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

#### WRITE ONE LINE OF CODE. RUN IT. CHECK IT. REPEAT

Remember, you write and run the line *BEFORE* you add the pipe `|>`! The annotations explain as if we were writing an English sentence.

```

hot100 |>
  group_by(performer, title) |>
  summarise(appearances = n()) |>
  arrange(desc(appearances))
```

(1)  
(2)  
(3)  
(4)

- ① Start with the `hot100` data AND THEN ...
- ② Group the data by both `performer` and `title` AND THEN ...
- ③ Summarize the data by first naming our new column “appearances” and setting that to the result of the `n()` function that counts the number of rows AND THEN ...
- ④ Arrange the data in descending order by the `appearances` column.

```

# A tibble: 31,044 x 3
# Groups:   performer [10,742]
  performer           title    appearances
  <chr>              <chr>        <int>
1 Glass Animals      Heat Waves     91
2 The Weeknd         Blinding Lights 90
3 Imagine Dragons    Radioactive    87
4 AWOLNATION         Sail          79
5 Jason Mraz         I'm Yours     76
6 LeAnn Rimes         How Do I Live 69
7 LMFAO Featuring Lauren Bennett & GoonRock Party Rock Anthem 68
8 OneRepublic        Counting Stars 68
9 Zach Bryan          Something In The Orange 66
10 Adele             Rolling In The Deep 65
# i 31,034 more rows
```

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 hang out together.

So, what was your guess or this question? A little bit of history for that answer ... Glass Animals' *Heat Waves* in 2022 overcame The Weeknd's *Blinding Lights*, which had overcome Imagine Dragon's *Radioactive* some time in 2021.

### Note

When you run the code above you might see a warning “**summarise() has grouped output by ‘performer’. You can override using the .groups argument.**” THIS IS NOT A PROBLEM. It is just R letting you know that the output of this remains grouped by the first item. Explaining it would break your brain right now. Don't worry about it. This is not the droid you are looking for.

### 4.6.1 Introducing filter()

I showed you `head()` in the previous quest and that was useful to show just a few records at the top of a list, but it does so indiscriminately. Note there are some songs here with the same number of weeks on the charts. If we used `head()` we might split that tie, leaving us with an incomplete answer. 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:

<https://vimeo.com/435910359>

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

The syntax works like this:

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

# example
hot100 |>
  filter(performer == "Taylor Swift")
```

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 Taylor Swift example above. It is important to use two of them when you are asking if a value is “true” or “equal to”, as a single `=` typically means you are assigning a value to something.

#### 4.6.1.1 Comparisons: Logical tests

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

| Operator              | Definition   |
|-----------------------|--------------|
| <code>x &lt; y</code> | Less than    |
| <code>x &gt; y</code> | Greater than |
| <code>x === y</code>  | Equal to     |

| Operator                   | Definition               |
|----------------------------|--------------------------|
| <code>x &lt;= y</code>     | Less than or equal to    |
| <code>x &gt;= y</code>     | Greater than or equal to |
| <code>x != y</code>        | Not equal to             |
| <code>x %in% c(y,z)</code> | In a group               |
| <code>is.na(x)</code>      | Is NA (missing values)   |
| <code>!is.na(x)</code>     | Is not NA                |

Where you apply a filter matters. If we want to consider only certain data when we are grouping/summarizing then we filter BEFORE the GSA. If we filter after the GSA, we affect 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 after.

```
hot100 |>
  group_by(performer, title) |>
  summarize(appearances = n()) |>
  arrange(appearances |> desc()) |>
  filter(appearances >= 65) ①
```

① `filter()` is the function. The first argument in the function is the column we are looking at – in our case the `appearances` column, which was created in the summarize line. We then provide a comparison operator `>=` to find values “greater than or equal to” 65.

```
# A tibble: 11 x 3
# Groups:   performer [11]
  performer           title      appearances
  <chr>              <chr>            <int>
  1 Glass Animals     Heat Waves        91
  2 The Weeknd        Blinding Lights    90
  3 Imagine Dragons   Radioactive       87
  4 AWOLNATION        Sail               79
  5 Jason Mraz        I'm Yours         76
  6 LeAnn Rimes        How Do I Live     69
```

|    |                                                             |                         |
|----|-------------------------------------------------------------|-------------------------|
| 7  | LMFAO Featuring Lauren Bennett & GoonRock Party Rock Anthem | 68                      |
| 8  | OneRepublic                                                 | Counting Stars          |
| 9  | Zach Bryan                                                  | Something In The Orange |
| 10 | Adele                                                       | Rolling In The Deep     |
| 11 | Jewel                                                       | Foolish Games/You Were~ |
|    |                                                             | 65                      |

The value we use to cut the list off – 65 in this case – is arbitrary. We just want to choose a value that makes common sense. 70 seemed too high because that would yield only five or so songs, and 60 was too low because there are lots of ties in the lower 60s.

#### 4.6.2 Data Takeaway: Song appearances

OK, time to write about what you've found.

1. Add this data takeaway after the code chunk above.

```
Data Takeaway: The song "Heat Wave" by Glass Animals has appeared 91 times on the Billboard
```

I'm providing these takeaways in this chapter as examples. You'll be asked to write your own in future projects.

#### 4.6.3 Render your second quest

This would be a good time to again **Render** your notebook so you can see what the output will look like. Look through that output, checking your headlines and such. Clean up any problems you might see.

### 4.7 Song the longest at No. 1

We introduced `filter()` in the last quest to limit the summary. For this quest you'll need to filter the data *before* the GSA trio.

Let's review the quest: **Which song (title & performer) 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 need and then build the query **one line at a time** to make each line work.

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 titles that have a `current_rank` of 1. This means we will `filter` before any summarizing.
- Then we want to count the number of rows with the same `performer` and `title` combinations. This means we need to `group_by` both `performer` and `title`.
- Since we are **counting rows**, we need use `n()` as our `summarize` function, which counts the `number` of 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 `current_rank` to be == to 1.
4. Run the result and check it

```
hot100 |>
  filter(current_rank == 1)
```

```
# A tibble: 3,413 x 7
  chart_date current_rank title   performer previous_rank peak_rank wks_on_chart
  <date>        <dbl> <chr>   <chr>           <dbl>      <dbl>       <dbl>
1 1958-08-04      1 Poor ~ Ricky Ne~        NA         1          1
2 1958-08-11      1 Poor ~ Ricky Ne~        1         1          2
3 1958-08-18      1 Nel B~ Domenico~       2         1          3
4 1958-08-25      1 Littl~ The Eleg~       2         1          4
5 1958-09-01      1 Nel B~ Domenico~       2         1          5
6 1958-09-08      1 Nel B~ Domenico~       1         1          6
7 1958-09-15      1 Nel B~ Domenico~       1         1          7
8 1958-09-22      1 Nel B~ Domenico~       1         1          8
9 1958-09-29      1 It's ~ Tommy Ed~       3         1          7
10 1958-10-06     1 It's ~ Tommy Ed~       1         1          8
# i 3,403 more rows
```

The result should show *only* titles with a 1 for `current_rank`.

The rest of our logic is just like our last quest. We need to group by the `title` 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()`.

```
hot100 |>
  filter(current_rank == 1) |>
  group_by(performer, title) |>
  summarize(appearances = n())
```

```

# A tibble: 1,161 x 3
# Groups:   performer [767]
  performer      title    appearances
  <chr>        <chr>        <int>
1 'N Sync     It's Gonna Be Me         2
2 24kGoldn Feat. iann dior       Mood          8
3 2Pac Feat. K-Ci And JoJo  How Do U Want It/California L~         2
4 50 Cent      In Da Club          9
5 50 Cent Feat. Nate Dogg    21 Questions        4
6 50 Cent Feat. Olivia      Candy Shop          9
7 6ix9ine & Nicki Minaj   Trollz            1
8 ? (Question Mark) & The Mysterians 96 Tears          1
9 A Taste Of Honey    Boogie Oogie Oogie        3
10 ABBA           Dancing Queen          1
# i 1,151 more rows

```

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

### Note

While **WRITING AND RUNNING ONE LINE AT A TIME** is still “[the Way](#)”, when you run a line with `group_by()` it won’t usually show different results without its buddy `summarize()`. So I usually write those two together, or I write the `summarize()` line first to make sure it works, then edit in the `group_by()` line to split the data before the summary.

Our result above doesn’t quite get us where we want because it lists the results alphabetically by the performer. 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.

```

hot100 |>
  filter(current_rank == 1) |>
  group_by(performer, title) |>
  summarize(appearances = n()) |>
  arrange(appearances |> desc())

```

```

# A tibble: 1,161 x 3
# Groups:   performer [767]

```

```

performer          title      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 Morgan Wallen                  Last Night     16
5 Harry Styles                 As It Was      15
6 Boyz II Men                  I'll Make Love~  14
7 Elton John                   Candle In The ~  14
8 Los Del Rio                  Macarena (Bays~  14
9 Mariah Carey                All I Want For~  14
10 Mariah Carey                We Belong Toge~  14
# i 1,151 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 15 or greater.

```

hot100 |>
  filter(current_rank == 1) |>
  group_by(performer, title) |>
  summarize(appearances = n()) |>
  arrange(appearances |> desc()) |>
  filter(appearances >= 15)

```

```

# A tibble: 5 x 3
# Groups:   performer [5]
  performer          title      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 Morgan Wallen                  Last Night     16
5 Harry Styles                 As It Was      15

```

Now you have the answers to the song with the most weeks at No. 1 with a logical cutoff. If you add to the data later, that logic will still hold and not cut off arbitrarily at a certain number of records. For instance, Mariah Carey's *All I Want for Christmas* will likely reach its 15th week at the top of the charts in 2024.

#### 4.7.1 Data Takeaway: Longest at No. 1

1. Add this data takeaway to your notebook:

```
The song "Old Town Road" by Lil Nas X Featuring Billy Ray Cyrus has spent more time at the top of the chart than any other song in history.
```

### 4.8 Performer with most No. 1 singles

Our next quest is this: **Which performer had the most titles reach No. 1?**

This sounds similar to our last quest, but there is a **distinct** difference. (That's a dad 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 title/performer combination only once. Another way to think of this is we need unique or **distinct** combinations of title/performer. (We'll introduce a new function to find this.)
- Once we have all the unique No. 1 songs in a list, then we can group by performer and count the **number** of 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 `current_rank` of 1.

```
hot100 |>  
  filter(current_rank == 1)
```

```
# A tibble: 3,413 x 7  
  chart_date current_rank title   performer previous_rank peak_rank wks_on_chart  
  <date>        <dbl> <chr>    <chr>      <dbl>      <dbl>       <dbl>  
1 1958-08-04      1 Poor ~ Ricky Ne~     NA         1          1          1  
2 1958-08-11      1 Poor ~ Ricky Ne~     1          1          1          2  
3 1958-08-18      1 Nel B~ Domenico~    2          1          1          3  
4 1958-08-25      1 Littl~ The Eleg~    2          1          1          4  
5 1958-09-01      1 Nel B~ Domenico~    2          1          1          5  
6 1958-09-08      1 Nel B~ Domenico~    1          1          1          6  
7 1958-09-15      1 Nel B~ Domenico~    1          1          1          7  
8 1958-09-22      1 Nel B~ Domenico~    1          1          1          8
```

```

9 1958-09-29          1 It's ~ Tommy Ed~          3          1          7
10 1958-10-06         1 It's ~ Tommy Ed~         1          1          8
# i 3,403 more rows

```

Now look at the result. Note how “Poor Little Fool” shows up more than once? Other songs do as well. If we counted rows by `performer` now, that would tell us the number of weeks they’ve had No. 1 songs, not how many *different* songs have made No. 1.

#### 4.8.1 Using `distinct()`

The next challenge in our logic is to show only unique performer/title 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 `title`. All other columns are dropped since including them would mess up their distinctness.

1. Edit your chunk to add the `distinct()` function to your code chunk.

```

hot100 |>
  filter(current_rank == 1) |>
  distinct(title, performer)

# A tibble: 1,161 x 2
  title                  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
# i 1,151 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 summarize 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/arrange combination for this, but we are only grouping by `performer` since that is the values 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 `no1_hits`. Run it.
2. After you are sure the group\_by/summarize runs, add an `arrange()` to show the `no1_hits` in descending order.

```
hot100 |>
  filter(current_rank == 1) |>
  distinct(title, performer) |>
  group_by(performer) |>
  summarize(no1_hits = n()) |>
  arrange(no1_hits |> desc())
```

```
# A tibble: 767 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 Taylor Swift      10
7 The Supremes      10
8 Bee Gees          9
9 The Rolling Stones  8
10 Janet Jackson     7
# i 757 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 performers with 10 or more No. 1 hits.

```

hot100 |>
  filter(current_rank == 1) |>
  distinct(title, performer) |>
  group_by(performer) |>
  summarize(no1_hits = n()) |>
  arrange(no1_hits |> desc()) |>
  filter(no1_hits >= 10)

```

```

# A tibble: 7 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 Taylor Swift      10
7 The Supremes     10

```

So, **The Beatles**. Was that your guess? Look closely at that list ... who has any chance of topping them?

#### 4.8.4 The perils of collaborations

If you are a student of music history, you might be scratching your head about now, thinking [the Beatles have 20 No. 1 hits!](#) You would be right, there is something up here.

The song *Get Back* by “The Beatles with Billy Preston” reached No. 1 on the Hot 100 on May 10, 1969. But that song does not show up with our other Beatles songs because those are listed as just “The Beatles”. This is the same issue I noted earlier about “Drake” and “Drake featuring Odd Future” or “Future featuring Drake” each being considered separate performers. They aren’t counted together.

How do we get around this? Well, we write accurately. We note our results consider collaborations separately. Yes, it’s true we could miss some things. It’s a good idea to double check your work, perhaps by searching the data for all distinct song titles that include “Beatles” in the performer field.

If the `performer` data were more consistent we might be able to extract each band/person, but it isn’t. Which of these would you consider separate? “Supremes & Temptations”, “Romeo & Juliet Soundtrack”, “Crosby, Stills & Nash”, “Delaney & Bonnie & Friends/Eric Clapton.” It’s a mess.

#### 4.8.5 Data Takeaway: Most No. 1 hits

So, that makes this takeaway a bit more nuanced. Add it to your notebook:

```
The Beatles have the most No. 1 hits on the Billboard Hot 100 charts with 19, but they also
```

### 4.9 Most 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 charts since 2019.
- We might need to adjust our last filter for a better “break point”.

There are a number of ways we could write a filter for the date, but we'll do so in a way that gets all the rows *after* the last day of 2018.

```
hot100 |>  
  filter(chart_date > "2018-12-31") |>  
  head() # added just to shorten our result
```

```
# A tibble: 6 x 7  
chart_date current_rank title   performer previous_rank peak_rank wks_on_chart  
<date>        <dbl> <chr>    <chr>        <dbl>      <dbl>       <dbl>  
1 2019-01-05      1 Thank ~ Ariana G~          1         1           8  
2 2019-01-05      2 Withou~ Halsey            2         2          12  
3 2019-01-05      3 All I ~ Mariah C~          7         3          30  
4 2019-01-05      4 Sicko ~ Travis S~          3         1          21  
5 2019-01-05      5 Sunflo~ Post Mal~            4         4          10  
6 2019-01-05      6 High H~ Panic! A~          6         5          21
```

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, one line at a time) the same filter, group\_by, summarize and arrange as above, but **leave out the cut-off filter** at the end. Make sure it runs.
3. **Edit your filter** to put a comma after `current_rank == 1` and then add this filter:  
`chart_date > "2017-12-31"`. Run the code.

4. Build a new cut-off filter at the end and keep only rows with more than 1 `top_hits`.

```
hot100 |>
  filter(
    current_rank == 1,
    chart_date > "2017-12-31" ①
  ) |>
  distinct(title, performer) |>
  group_by(performer) |>
  summarize(top_hits = n()) |>
  arrange(top_hits |> desc()) |>
  filter(top_hits > 1) ②
```

- ① This is where we add the new filter  
② This is the new cutoff since the earlier value wouldn't work

```
# A tibble: 11 x 2
  performer          top_hits
  <chr>              <int>
1 Taylor Swift        6
2 Drake                5
3 BTS                  4
4 Ariana Grande        3
5 Olivia Rodrigo        3
6 Harry Styles          2
7 Jack Harlow           2
8 Lizzo                 2
9 The Weeknd            2
10 The Weeknd & Ariana Grande  2
11 Travis Scott          2
```

Now you know who has the most No. 1 hits from 2019-2023. “*Are you not entertained?*”

#### 4.9.1 Data Takeaway: No. 1 in past five years

OK, here is the takeaway to add ...

```
Taylor Swift not only has the most appearances of all time in the Hot 100, she also has the m
```

## 4.10 Complex filters

You can combine filters in different ways to really target which rows to keep. For these I want you to play around a bit.

1. Copy each of the examples below into your notebook, **but change the title and/or performer** to someone you like.

### 4.10.1 Multiple truths

If you want to filter data for when **two or more things are true**, you can write two equations and combine with `&`. Only rows where both sides prove true are returned.

This shows when Poor Little Fool was No. 1, but not any other position.

```
hot100 |>
  filter(title == "Poor Little Fool" & current_rank == 1) |>
  select(chart_date:performer)
```

```
# A tibble: 2 x 4
  chart_date current_rank title           performer
  <date>        <dbl>   <chr>          <chr>
1 1958-08-04      1 Poor Little Fool Ricky Nelson
2 1958-08-11      1 Poor Little Fool Ricky Nelson
```

That `select()` line is capturing just the first four columns for us.

### 4.10.2 Either is true

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

This shows songs by Adam Sandler OR Alabama Shakes.

```
hot100 |>
  filter(performer == "Adam Sandler" | performer == "Alabama Shakes") |>
  select(chart_date:performer)
```

```
# A tibble: 4 x 4
chart_date current_rank title performer
<date>       <dbl> <chr>   <chr>
1 1999-01-02      80 The Chanukah Song Adam Sandler
2 1999-01-09      98 The Chanukah Song Adam Sandler
3 2013-03-02     100 Hold On Alabama Shakes
4 2013-03-09      93 Hold On Alabama Shakes
```

 Tip

The | is found as the *Shift* of the \ key above Return on your keyboard. That | character is also sometimes called a “pipe”, which gets confusing in R with |>.)

#### 4.10.3 Mixing criteria

If you have **multiple criteria**, you separate them with a comma ,. Note I’ve also added returns to make the code more readable and added distinct to songs only once.

This gives us rows with either Taylor Swift or Drake, but only those that reached No. 1 (and no collaborations.)

```
hot100 |>
  filter(
    performer == "Taylor Swift" | performer == "Drake",
    current_rank == 1
  ) |>
  distinct(current_rank, title, performer)
```

```
# A tibble: 15 x 3
current_rank title performer
<dbl> <chr>   <chr>
1          1 We Are Never Ever Getting Back Together Taylor Swift
2          1 Shake It Off Taylor Swift
3          1 Blank Space Taylor Swift
4          1 Look What You Made Me Do Taylor Swift
5          1 God's Plan Drake
6          1 Nice For What Drake
7          1 In My Feelings Drake
8          1 Toosie Slide Drake
9          1 Cardigan Taylor Swift
10         1 Willow Taylor Swift
```

|    |                                                       |              |
|----|-------------------------------------------------------|--------------|
| 11 | 1 What's Next                                         | Drake        |
| 12 | 1 All Too Well (Taylor's Version)                     | Taylor Swift |
| 13 | 1 Anti-Hero                                           | Taylor Swift |
| 14 | 1 Cruel Summer                                        | Taylor Swift |
| 15 | 1 Is It Over Now? (Taylor's Version) [From The Vault] | Taylor Swift |

#### 4.10.4 Search within a string

And if you want to search for text **within** the data, you can use `str_detect()` to look for specific characters within a value to filter rows. `str_detect()` needs two arguments: 1) What column to search in, and 2) what to search for “in quotes”. I also use `distinct()` here to show only unique title/performer combinations.

This shows songs where “2 Chainz” was among the performers.

```
hot100 |>
  filter(str_detect(performer, "2 Chainz")) |>
  distinct(title, performer)
```

| # A tibble: 40 x 2 |                                                                        |
|--------------------|------------------------------------------------------------------------|
|                    | title performer                                                        |
|                    | <chr> <chr>                                                            |
| 1                  | Mercy Kanye West, Big Sean, Pusha T, 2 Chainz                          |
| 2                  | Beez In The Trap Nicki Minaj Featuring 2 Chainz                        |
| 3                  | No Lie 2 Chainz Featuring Drake                                        |
| 4                  | Birthday Song 2 Chainz Featuring Kanye West                            |
| 5                  | Yuck! 2 Chainz Featuring Lil Wayne                                     |
| 6                  | Bandz A Make Her Dance Juicy J Featuring Lil Wayne & 2 Chainz          |
| 7                  | My Moment DJ Drama Featuring 2 Chainz, Meek Mill & Jeremih             |
| 8                  | F**kin Problems A\$AP Rocky Featuring Drake, 2 Chainz & Kendrick Lamar |
| 9                  | I'm Different 2 Chainz                                                 |
| 10                 | R.I.P. Young Jeezy Featuring 2 Chainz                                  |
|                    | # i 30 more rows                                                       |

You might be tempted to try “Drake” above, but there are so many other names that include Drake (like “Charlie Drake” and “Pete Drake”) that it’s too messy.

There are other variations and functions to help with filtering, but this is enough for now.

#### 4.11 On your own

I want you to do two things on your own using the skills we’ve covered above.

### **4.11.1 Most Top 10 hits**

**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 over think 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)
5. Write a Data Takeaway describing your finding

### **4.11.2 Find something you want to know**

Now I want you to find something else on your own. It doesn’t matter what it is. Just find something about a performer or song you like.

1. Start a new section with a Markdown headline
2. Use Markdown text to declare what you are looking for
3. Find it!
4. After your code, explain what functions you used and why (like what did they do for you)
5. Write a Data Takeaway describing your finding

## **4.12 Summarizing by year**

There is one more important concept I’d like you to learn and that you will need later, but this is already a long chapter and you have learned a lot of new stuff.

If you still have spare room in your brain now, go through the [Grouping by dates](#) chapter, which uses data from this project.

If your brain is mush, that’s cool. Just turn this project in. But know you’ll need to go through that chapter later when you are working on your mastery project.

## 4.13 Update your index summary

We have one, last major thing to do here before we finish, and that is to update our `index.qmd` file with a summary of what you've found.

1. Open your `index.qmd` file.
2. After the text that is here, add the lede, source graf and data takeaways noted below.
3. Then add your own data takeaways at the end.

### `## Summary`

Taylor Swift has had more solo appearances on the Billboard Hot 100 chart than any other artist.

The chart has been published weekly since August 1958, and this analysis includes data through

> \*You should update this value with that last chart date in your data.

Other findings from the data analysis include:

- Data Takeaway: The song "Heat Wave" by Glass Animals has appeared 91 times on the Hot 100,
- The song "Old Town Road" by Lil Nas X Featuring Billy Ray Cyrus has spent more time at the top of the chart than any other song.
- The Beatles have the most No. 1 hits with 19, but they also have a 20th top song -- "Get Back".
- Taylor Swift not only has the most appearances of all time in the Hot 100, she also has the most consecutive weeks at No. 1.
- ADD YOUR TOP 10 HITS
- ADD YOUR OWN FINDING

## 4.14 Turn in your project

1. Make sure everything runs and Renders properly.
2. Publish your changes to Quarto Pub and include the link to your project in your index notebook so I can bask in your glory.
3. Zip your project folder. (Or export to zip if you are using posit.cloud).
4. Upload to the Canvas assignment.

### ! Important

To be clear, it is your zipped project I am grading. The Quarto Pub link is for convenience.

## 4.15 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.
- `str_detect()` to search within strings.
- `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.

## 4.16 Soundtrack for this assignment

This lesson was constructed while listening to [The Bright Light Social Hour](#). They've not had a song on the Hot 100, at least not yet.

```
hot100 |>
  filter(str_detect(performer, "Bright Light"))

# A tibble: 0 x 7
# i 7 variables: chart_date <date>, current_rank <dbl>, title <chr>,
#   performer <chr>, previous_rank <dbl>, peak_rank <dbl>, wks_on_chart <dbl>
```

# **Part II**

# **Summing**

# 5 Military Surplus Cleaning

This chapter is by Prof. McDonald, who uses macOS.

## ⚠ Warning

Because of data updates, your answers may differ slightly from what is presented here.

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 using `group_by`, `summarize` and `arrange` (which I dub GSA) and we summarized with `n()` to count the number of rows within our groups.

For this data story we need the GSA summary trio again, but we will use math operations within our summarize functions, mainly `sum()` to add values together.

## 5.1 About the story: Military surplus transfers

After the 2014 death of Michael Brown and the unrest that followed, there was widespread criticism of Ferguson, Mo. police officers for their use of military-grade weapons and vehicles. There was also heightened scrutiny to a federal program that transfers unused military equipment to local law enforcement. Many news outlets, [including NPR](#) and [Buzzfeed News](#), did stories based on data from the “1033 program” handled through the [Law Enforcement Support Office](#). Buzzfeed News also did a [followup report](#) in 2020 where reporter John Templon published his [data analysis](#), which he did using Python.

You will analyze the same dataset to see how Central Texas police agencies have used the program and write a short data drop about transfers to those agencies.

### 5.1.1 The 1033 program

To work through this story we need to understand how this transfer program works. You can [read all about it here](#), but here is the gist:

In 1997, Congress approved the “1033 program” that allows the U.S. military to give “surplus” equipment that they no longer need to law enforcement agencies like city police forces. The program is run by the [Law Enforcement Support Office](#), which is part of the Defense Logistics Agency (which handles the global defense supply chain for all the branches of the military) within the Department of Defense. (The **program** is run by the **office** inside the **agency** that is part of the **department**.)

All kinds of equipment moves between the military and these agencies, from boots and water bottles to assault rifles and cargo planes. The local agency only pays for shipping the equipment, but that shipping cost isn’t listed in the data. What is in the data is the “original value” of the equipment in dollars, *but we can’t say the agency paid for it, because they didn’t.*

Property falls into two categories: controlled and non-controlled. **Controlled** property “consists of military items that are provided via a conditional transfer or ‘loan’ basis where title remains with DoD/DLA. This includes items such as small arms/personal weapons, demilitarized vehicles and aircraft and night vision equipment. This property always remains in the LESO property book because it still belongs to and is accountable to the Department of Defense. When a local law enforcement agency no longer wants the controlled property, it must be returned to Law Enforcement Support Office for proper disposition.” This is explained in the [LESO FAQ](#).

But most of the transfers to local agencies are for **non-controlled** property that can be sold to the general public, like boots and blankets. Those items are removed from the data after one year, unless it is deemed a special circumstance.

The agency releases data quarterly, but it is really a “snapshot in time” and not a complete history. Those non-controlled items transferred more than a year prior are missing, as are any controlled items returned to the feds.

### 5.1.2 About the data

The screenshot shows a Microsoft Excel spreadsheet titled "AllStatesAndTerritoriesQTR2FY22". The spreadsheet has a green header bar with tabs for Home, Insert, Draw, Page Layout, Formulas, Data, Review, View, Share, and Create and Share Adobe PDF. The main area displays a table with the following columns: A (State), B (Agency Name), C (NSN), D (Item Name), E (Quantity), F (UI), and G (Acquisition Value). The data includes entries for various police departments across different states like ALABAMA, ALASKA, ARIZONA, ARKANSAS, CALIFORNIA, COLORADO, CONNECTICUT, and others. The data shows items such as TRUCK,UTILITY, MOUNT,RIFLE, SIGHT,REFLEX, OPTICAL SIGHTING AND RANGING EQUIPMENT, MINI RESISTANT VEHICLE, ILLUMINATOR,INFRARED, CAMERA ROBOT, UNMANNED VEHICLE/GROUND, TRUCK,UTILITY, BALLISTIC BLANKET KIT, WEAPON PARTS, SIGHT,REFLEX, WEAPON PARTS, WEAPON PARTS, BOLT,BREACH, ILLUMINATOR,INFRARED, BARREL,SHOTGUN, BINOCULAR, and PIN,PIVOT. Quantities range from 1 to 10, and acquisition values range from \$5 to \$62,625.

| A  | B     | C                      | D                | E                                      | F        | G    |                 |
|----|-------|------------------------|------------------|----------------------------------------|----------|------|-----------------|
| 1  | State | Agency Name            | NSN              | Item Name                              | Quantity | UI   | Acquisition Val |
| 2  | AL    | ABBEVILLE POLICE DEPT  | 2320-01-371-9588 | TRUCK,UTILITY                          | 1        | Each | \$62,625        |
| 3  | AL    | ABBEVILLE POLICE DEPT  | 1005-01-587-7175 | MOUNT,RIFLE                            | 10       | Each | \$1,625         |
| 4  | AL    | ABBEVILLE POLICE DEPT  | 1240-01-411-1265 | SIGHT,REFLEX                           | 9        | Each | \$395           |
| 5  | AL    | ABBEVILLE POLICE DEPT  | 1240-DS-OPT-SIGH | OPTICAL SIGHTING AND RANGING EQUIPMENT | 1        | Each | \$245           |
| 6  | AL    | ABBEVILLE POLICE DEPT  | 2355-01-553-4634 | MINI RESISTANT VEHICLE                 | 1        | Each | \$658,000       |
| 7  | AL    | ABBEVILLE POLICE DEPT  | 5855-01-577-7174 | ILLUMINATOR,INFRARED                   | 10       | Each | \$1,125         |
| 8  | AL    | ABBEVILLE POLICE DEPT  | 6760-01-628-6105 | CAMERA ROBOT                           | 1        | Each | \$1,500         |
| 9  | AL    | ABBEVILLE POLICE DEPT  | 1385-01-574-4707 | UNMANNED VEHICLE/GROUND                | 1        | Each | \$10,000        |
| 10 | AL    | ABBEVILLE POLICE DEPT  | 2320-01-371-9584 | TRUCK,UTILITY                          | 1        | Each | \$62,625        |
| 11 | AL    | ABBEVILLE POLICE DEPT  | 2540-01-565-4700 | BALLISTIC BLANKET KIT                  | 10       | Kit  | \$15,871        |
| 12 | AL    | ADAMSVILLE POLICE DEPT | 1005-DS-SWE-PART | WEAPON PARTS                           | 1        | Each | \$50            |
| 13 | AL    | ADAMSVILLE POLICE DEPT | 1240-01-411-1265 | SIGHT,REFLEX                           | 12       | Each | \$395           |
| 14 | AL    | ADAMSVILLE POLICE DEPT | 1005-DS-SWE-PART | WEAPON PARTS                           | 1        | Each | \$435           |
| 15 | AL    | ADAMSVILLE POLICE DEPT | 1005-DS-SWE-PART | WEAPON PARTS                           | 1        | Each | \$245           |
| 16 | AL    | ADAMSVILLE POLICE DEPT | 1005-01-505-1035 | BOLT,BREACH                            | 1        | Each | \$60            |
| 17 | AL    | ADAMSVILLE POLICE DEPT | 5855-01-448-5464 | ILLUMINATOR,INFRARED                   | 12       | Each | \$355           |
| 18 | AL    | ADAMSVILLE POLICE DEPT | 1005-01-375-8167 | BARREL,SHOTGUN                         | 3        | Each | \$184           |
| 19 | AL    | ADAMSVILLE POLICE DEPT | 1240-01-361-1318 | BINOCLAR                               | 1        | Each | \$432           |
| 20 | AL    | ADAMSVILLE POLICE DEPT | 1005-00-992-6671 | PIN,PIVOT                              | 10       | Each | \$5             |

ALABAMA    ALASKA    ARIZONA    ARKANSAS    CALIFORNIA    COLORADO    CONN    +

Ready    100%

Figure 5.1: The raw data

The data comes in a spreadsheet that has a different tab for each state and territory. I've done some initial work on the original data that is beyond the scope of this class, so we'll use my copy of the data. **I will supply a link to the combined data below.**

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.

- **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:** The National Stock Number, a special ID that identifies the item in a government supplies database.
- **item\_name:** The item transferred. Googling the names can sometimes yield more info on specific items, or you can [search by nsn](#) for more info.
- **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**: Categories (as single letter key values) that indicate how the item should be disposed of. [Full details here](#).
- **demil\_ic**: Also part of the disposal categorization.
- **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 glimpse of a sample of the data:

```
Rows: 10
Columns: 12
$ sheet           <dbl> 17, 42, 5, 45, 36, 33, 5, 22, 3, 22
$ state            <chr> "KY", "SC", "CA", "TX", "OH", "NC", "CA", "MI", "AZ"~
$ agency_name      <chr> "MEADE COUNTY SHERIFF DEPT", "PROSPERITY POLICE DEPT~
$ nsn              <chr> "6115-01-435-1567", "1005-00-589-1271", "7520-01-519~
$ item_name        <chr> "GENERATOR SET,DIESEL ENGINE", "RIFLE,7.62 MILLIMETE~
$ quantity         <dbl> 5, 1, 32, 1, 1, 1, 1, 1, 1, 1
$ ui               <chr> "Each", "Each", "Dozen", "Each", "Each", "Each", "Ea~
$ acquisition_value <dbl> 4623.09, 138.00, 16.91, 749.00, 749.00, 138.00, 499.~
$ demil_code       <chr> "A", "D", "A", "D", "D", "D", "D", "D", "D"
$ demil_ic          <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1
$ ship_date         <dttm> 2021-03-22, 2007-08-07, 2020-10-29, 2014-11-18, 2014~
$ station_type      <chr> "State", "State", "State", "State", "State", "State"~
```

And a look at just some of the more important columns:

```
leso_sample |>
  select(agency_name, item_name, quantity, acquisition_value)
```

```
# A tibble: 10 x 4
  agency_name           item_name     quantity acquisition_value
  <chr>                  <chr>           <dbl>             <dbl>
1 MEADE COUNTY SHERIFF DEPT GENERATOR SET,DI~      5            4623.
2 PROSPERITY POLICE DEPT   RIFLE,7.62 MILLI~      1            138
3 KERN COUNTY SHERIFF OFFICE MARKER,TUBE TYPE    32            16.9
4 LEAGUE CITY POLICE DEPT RIFLE,5.56 MILLI~      1            749
5 TRUMBULL COUNTY SHERIFF'S OFFICE RIFLE,5.56 MILLI~    1            749
6 VALDESE POLICE DEPT     RIFLE,7.62 MILLI~      1            138
7 SACRAMENTO POLICE DEPT  RIFLE,5.56 MILLI~      1            499
8 BERRIEN COUNTY SHERIFF'S OFFICE RIFLE,5.56 MILLI~    1            499
9 LA PAZ COUNTY SHERIFF OFFICE RIFLE,5.56 MILLI~      1            499
10 MUSKEGON HEIGHTS POLICE DEPT RIFLE,5.56 MILLI~     1            749
```

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.

The data also doesn't have any variable indicating if an item is controlled or non-controlled, but I've corresponded with the Defense Logistics Agency to gain a pretty good understanding of how to isolate them based on the **demilitarization** codes.

These are things I learned about by talking to the agency and reading through documentation. This kind of reporting and understanding **ABOUT** your data is vital to avoid mistakes.

## 5.2 The questions we will answer

All answers should be based on **controlled** items given to **Texas agencies** from **Jan. 1, 2010 to present**.

- *How many total “controlled” items were transferred to Texas agencies, and what are they all worth?* We'll summarize all the controlled items only to get the total quantity and total value of everything.
- *How many total “controlled” items did each agency get and how much was it all worth?* Which agency got the most stuff?
  - *How about local police agencies?* I'll give you a list.
- *What specific “controlled” items did each agency get and how much were they worth?* Now we're looking at the kinds of items.
  - *What did local agencies get?*

You'll research some of the more interesting items the agencies received so you can include them in a short data drop.

## 5.3 Getting started: Create your project

### Note

If you are using posit.cloud, you'll want to refer to the [Using posit.cloud](#) chapter to create your project, using the material below as necessary.

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**.
3. For the project type, choose **Quarto Website**
4. Name the directory "yourname-military-surplus", but with your name.
5. Create two folders: **data-raw** and **data-processed**.
6. In the `index.qmd` file, replace the title value with "Military Surplus", as that is our project title.
7. Remove the boilerplate stuff after the YAML title and this bit about the project:

This project looks at how much U.S. military surplus equipment has been transferred to law enforcement agencies since 2001. This data is from the US Department of Defense's Defense Disposition Accountability System (DDAS) and includes information on surplus equipment such as vehicles, aircraft, ships, and weapons.

You'll come back to this at the end of the project to write your summary.

## 5.4 Cleaning notebook

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

1. Create a new **Quarto Document**.
2. For the title use "Cleaning".
3. Remove the rest of the boilerplate template.
4. Save the file and name it `01-cleaning.qmd`.
5. Open `_quarto.yml` and change the navigation link from `about.qmd` to `01-cleaning.qmd`.

### 5.4.1 The goals of the notebook

As noted before, I separate cleaning into a separate notebook so that each subsequent analysis notebook can take advantage of that work. It's the DRY principal in programming: Don't Repeat Yourself. Often I will be in an analysis notebook realizing that I have more cleaning to do, so I'll go back to the cleaning notebook, add it and rerun everything. Because I've worked with and researched this data, I'm aware of cleaning steps that a newcomer to the data might not be aware of at this point. To save you that time and heartache, we will take advantage of my experience and do all this cleaning work up front even though you haven't seen the "need" for it yet.

That's a long-winded opening to say let's add our notebook goals so you know what's going on here.

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

```
- Download the data
- Import the data
- Check datatypes
- Create a total_value variable
- Create a control_type variable
- Filter the date range (since Jan. 1 2010)
- Export the cleaned data
```

### 5.4.2 Add a setup section

This is the section in the notebook where we add our libraries and such. Again, every notebook has this section, though the packages used may vary on need. This time (and each time hence) we will add some "execution options" to our setup chunk.

1. Add a headline called "Setup" and text about what we are doing, adding our libraries.
2. Add your setup code chunk with the code below.

```
```{r}
#| label: setup
#| message: false

library(tidyverse)
```
```

Some reminders:

- The `#| label: setup` is special in that it will be run first no matter where we are in the notebook. Helpful to make sure your libraries are loaded.
- The `#| message: false` option silences the message about all the different packages in the tidyverse when the library is loaded. It just cleans up the notebook.

One other note about this ... we should only need the `tidyverse` library for this notebook because the data already has clean names (no need for `janitor`.)

## 5.5 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 the text below:

```
While the data we will use here if from Prof. McDonald, it is from the [Law Enforcement Suppo
```



Tip

Note the text above scrolls off the screen. Use the copy icon to select all the text.

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",
  mode = "wb"
)
```

What you need:

- The data can be found at this url below
- It should be saved into your `data-raw` folder with a name for the file.

```
https://github.com/utdata/rwd-r-leso/blob/main/data-processed/leso.csv?raw=true
```

```
# You can comment the lines 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",
  mode = "wb"
)
```

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.

 Tip

If you get an error about the path, you might make sure you created the `data-raw` folder first.

## 5.6 Import the data

It's time to import the data into the notebook. Like the Billboard project, we are again working with a CSV, or comma-separated-values text file.

1. Add a new Markdown section noting you are doing the import. Include a headline, text and a new 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.

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

The output you get will be the table, which looks sort of like this, but I'm just showing the first six lines:

```
# A tibble: 6 x 12
  sheet state agency_name      nsn item_name quantity ui    acquisition_value
  <dbl> <chr> <chr>        <chr> <chr>       <dbl> <chr>           <dbl>
1     1 AL   ABBEVILLE POLICE~ 1005~ MOUNT, RI~      10 Each          1626
2     1 AL   ABBEVILLE POLICE~ 1240~ SIGHT, RE~       9 Each           371
3     1 AL   ABBEVILLE POLICE~ 2320~ TRUCK, UT~       1 Each          62627
4     1 AL   ABBEVILLE POLICE~ 2540~ BALLISTI~       10 Kit           17265.
5     1 AL   ABBEVILLE POLICE~ 2320~ TRUCK, UT~       1 Each          62627
6     1 AL   ABBEVILLE POLICE~ 2355~ MINE RES~       1 Each          658000
# i 4 more variables: demil_code <chr>, demil_ic <dbl>, ship_date <dttm>,
#   station_type <chr>
```

### 5.6.1 Glimpse the data

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

```
leso |> glimpse()
```

```
Rows: 99,052
Columns: 12
$ sheet          <dbl> 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"~
$ agency_name    <chr> "ABBEVILLE POLICE DEPT", "ABBEVILLE POLICE DEPT", "A~
$ nsn             <chr> "1005-01-587-7175", "1240-01-411-1265", "2320-01-371~
$ item_name       <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~
$ quantity        <dbl> 10, 9, 1, 10, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ui              <chr> "Each", "Each", "Each", "Kit", "Each", "Each", "Each~
$ acquisition_value <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~
$ demil_code      <chr> "D", "D", "C", "D", "C", "Q", "D", "C", "D", "D~
$ demil_ic        <dbl> 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ ship_date       <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~
$ station_type    <chr> "State", "State", "State", "State", "State", "State"~
```

## 5.7 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 for you), but you always want to check and make corrections, like we did to fix the date in the Billboard assignment.

## 5.8 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, we will remove

the `sheet` variable since we don't need it. (It's an artifact of the processing I've done on the file.)

1. Start a new section with a headline and 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 |>          ①  
  select(!sheet)                 ②  
  
leso_tight |> glimpse()        ③
```

- ① We create a new tibble `leso_tight` and then fill it with `leso` data AND THEN ...
- ② We use `select` to remove the column called `sheet`. Within the `select()` function, we use `!` to “negate” the column called “sheet”.
- ③ We then glimpse the new data.

```
Rows: 99,052  
Columns: 11  
 $ state           <chr> "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", "2320-01-371~  
 $ item_name       <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~  
 $ quantity        <dbl> 10, 9, 1, 10, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, ~  
 $ ui               <chr> "Each", "Each", "Each", "Kit", "Each", "Each", "Each~  
 $ acquisition_value <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~  
 $ demil_code       <chr> "D", "D", "C", "D", "C", "Q", "D", "C", "D", "D~  
 $ demil_ic         <dbl> 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~  
 $ ship_date        <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~  
 $ station_type     <chr> "State", "State", "State", "State", "State", "State"~
```

If we wanted to remove more than one column — say both `sheet` and `station_type` — then we would use `!` negate in conjunction with the `c()` function to combine items into a new vector, like this: `select(!c(sheet, station_type))`. The use of `!` to remove things will come up in other places in R, as will using `c()` to combine things together.

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

## 5.9 Create a `total_value` column

During my reporting about this data I learned that the `acquisition_value` noted in the data is for a single “unit” of each item. If the shipment item was a rifle with a quantity of “5” and

`acquisition_value` of “200”, then each rifle is worth \$200, but the total shipment would be 5 times \$200, or \$1,000. That \$1000 total value is not listed in the data, so we need to add it.

Let’s walk through how to do that with a different example.

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

```
# Don't put this in your notebook. It's just explanation.  
data |>  
  mutate(total_value = item_count * item_value)
```

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.

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

### 5.9.1 Check that it worked!!

Note that new columns are added at the end of the tibble. Sometimes you can look through the glimpsed data to see if your mutate worked correctly, but that depends on the data.

If there isn't enough information visible, you can print some sample rows of the data to peek through it.

1. Add some text that you are checking the results.
  2. Add a new chunk and pipe `leso_total` into a new function `slice_sample(n = 10)`.

```
leso_total |>  
  slice_sample(n = 10)
```

① The argument  $n = 10$  is the number of rows to show. The default is a single row.

```
# A tibble: 10 x 12
  state agency_name nsn    item_name quantity ui      acquisition_value demil_code
  <chr> <chr>     <chr> <chr>       <dbl> <chr>           <dbl> <chr>
```

```

1 MS      JACKSON CO~ 1005~ RIFLE,5.~        1 Each       499   D
2 GA      CHAMBLEE P~ 1240~ MAGNIFIE~        10 Each      366.   D
3 NC      SPRING LAK~ 1005~ RIFLE,5.~        1 Each       499   D
4 GA      COLUMBUS P~ 1005~ RIFLE,5.~        1 Each       499   D
5 ID      FREMONT CO~ 1005~ PISTOL,C~        1 Each       58.7  D
6 CA      MARTINEZ P~ 1240~ SIGHT,RE~        1 Each       371   D
7 RI      LINCOLN PO~ 1005~ RIFLE,5.~        1 Each       499   D
8 MI      OAKLAND CO~ 5855~ ILLUMINA~        1 Each       941.   D
9 CA      KERN COUNT~ 8430~ BOOTS, M~        1 Pair       50    A
10 WV     STATE FIRE~ 1005~ RIFLE,5.~       1 Each       499   D
# i 4 more variables: demil_ic <dbl>, ship_date <dttm>, station_type <chr>,
#   total_value <dbl>

```

At this point you can page through the columns to check the math. If you want to see a different sample of data, just rerun the chunk as you get a new one each time. If you want a different number of lines, change the value inside the function.

But even with this, there are a lot of columns to page through. You can use add on `select()` to pluck out the columns of interest to check.

1. Edit your chunk's last line like this:

```

leso_total |>
  slice_sample(n = 10) |>
  select(agency_name, quantity, acquisition_value, total_value)

```

```

# A tibble: 10 x 4
  agency_name          quantity acquisition_value total_value
  <chr>                <dbl>            <dbl>        <dbl>
1 DNR DES MOINES           1              499         499
2 MONETT POLICE DEPT        1              108         108
3 RICHLAND POLICE DEPT       1              320         320
4 BRANTLEY COUNTY SHERIFFS OFFICE 3             1340.      4021.
5 CHEROKEE COUNTY SHERIFF OFFICE 1              138         138
6 CHICAGO POLICE DEPARTMENT    1              499         499
7 KERN COUNTY SHERIFF OFFICE    32              54        1728
8 MEIGS COUNTY SHERIFF'S DEPT     1              749         749
9 NORTH KINGSVILLE POLICE DEPT    1              371         371
10 SALINAS POLICE DEPT          1              499         499

```

This should allow you to check if the math worked. In one of my samples there was a record for “JONESBORO POLICE DEPARTMENT” that has a quantity of “6” and

`acquisition_value` of “114.00” and I got a `total_value` of “684.00”. So,  $6 * 114.00 = 684.00$ , which is correct!

Something to note about this process: In the first part of this chunk we are taking `leso_tight`, making some changes to it and then **saving** it into `leso_total`. We have two objects now, one before the changes and one after.

In the last part of the chunk, we are taking `leso_total` object and we are peeking at some rows and some columns, but we are **not saving** those changes into a new object. There is no `<-` operator here. We are just printing the results to the screen so we can *see* parts of the data more clearly.

## 5.10 Controlled vs. non-controlled

Again, by reading through the [documentation](#) about this data I learned about controlled vs non-controlled property. Basically non-controlled generic items like boots and blankets are removed from the data after one year, but controlled items like guns and airplanes remain on the list until they are returned to the military for disposal. We are only concerned with the controlled items.

There isn’t anything within the data that says it is “controlled” and really no clear indication in the documentation on how to tell what is what. So, I emailed the agency and asked them. Here is an edited version of their answer:

Property with the DEMIL codes A and Q6 are considered non-controlled general property and fall off the LESO property books after one year. All other Demil codes are considered controlled items and stay on the LESO property book until returned to DLA for disposition/disposal. However, there are some exceptions. For instance, aircraft are always controlled regardless of the demil code. Also, LESO has the discretion to keep items as controlled despite the demil code. This happens with some high value items. There isn’t a standard minimum value. It also may also depend on the type of property.

This actually took some back and forth to figure out, as I had noticed there were AIRPLANE, CARGO-TRANSPORT items in the data that were older than a year, along with some surveillance robots. That’s when they replied about the airplanes, but it turns out the robots were simply an error. **Data is dirty** when humans get involved; somebody coded it wrong.

These “DEMIL codes” they referenced are the `demil_code` and `demil_ic` columns in the data, so we can use those to mark which records are “non-controlled” (A and Q6) and then mark all the rest as “controlled”. We know of one exception – airplanes – which we need to mark controlled. We can’t do much with the “high value” items since there isn’t a specific maximum value to test. We’ll just have to note that in our story, something like “the agency has discretion

to designate as as controlled, such as high value items, but they are not categorized as such and may have been dropped from our analysis.”

But, we can catch most of them ... we just need to work through the logic. This is not an uncommon challenge in data science, so we have some tools to do this. Our workhorse here is the `case_when()` function, where we can make decisions based on values in our data.

In the interest of time and getting this done, I will just provide the code.

```
leso_control <- leso_total |>
  mutate(
    control_type = case_when(
      str_detect(item_name, "AIRPLANE") ~ TRUE,
      (demil_code == "A" | (demil_code == "Q" & demil_ic == 6)) ~ FALSE,
      TRUE ~ TRUE
    )
  )

leso_control |> glimpse()
```

```
Rows: 99,052
Columns: 13

$ 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", "2320-01-371~
$ item_name            <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~
$ quantity             <dbl> 10, 9, 1, 10, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ui                   <chr> "Each", "Each", "Each", "Kit", "Each", "Each", ~
$ acquisition_value    <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~
$ demil_code           <chr> "D", "D", "C", "D", "C", "C", "Q", "D", "C", "D", "D~
$ demil_ic              <dbl> 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ship_date            <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~
$ station_type          <chr> "State", "State", "State", "State", "State", "State"~
$ total_value           <dbl> 16260.0, 3339.0, 62627.0, 172647.1, 62627.0, 658000.~
$ control_type          <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, ~
```

If you want to learn more about the code in that block, read the Appendix chapter titled [Using case when](#).

## 5.11 Filtering our data

In the Billboard lesson you used `filter()` to get No. 1 songs and to get a date range of data. We need to do something similar here to get only data of a certain date range. We'll build the filters one at a time so we can save the resulting objects separately.

1. Create a new section with headlines and text that denote you are filtering the data to records since Jan. 1, 2010.
2. Create a chunk and filter your `leso_control` data to get rows with a `ship_date` later than 2010-01-01.
3. Save the result into a new tibble called `leso_dated`.
4. Print out the new tibble `leso_dated`.

```
leso_dated <- leso_control |>  
  filter(  
    ship_date >= "2010-01-01"  
)  
  
leso_dated
```

### 5.11.1 Checking the results with `summary()`

How do you know this date filter worked? Well, my data went from 99052 to 67852 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 minute. 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. Edit your chunk to not just print out the tibble, but pipe that into `summary()`, like this:

```
leso_dated |> summary()
```

```
state          agency_name        nsn          item_name  
Length:67852   Length:67852   Length:67852   Length:67852  
Class :character Class :character Class :character Class :character  
Mode  :character Mode  :character Mode  :character Mode  :character
```

```

      quantity          ui      acquisition_value    demil_code
Min.    : 0.00  Length:67852      Min.    :     0  Length:67852
1st Qu.: 1.00  Class :character  1st Qu.:    200  Class :character
Median : 1.00  Mode   :character  Median :    499  Mode   :character
Mean   : 5.66                               Mean   : 22002
3rd Qu.: 1.00                               3rd Qu.:  3578
Max.   :44000.00                           Max.   :22000000

      demil_ic      ship_date      station_type
Min.    :0.000  Min.    :2010-01-05 00:00:00.00  Length:67852
1st Qu.:1.000  1st Qu.:2012-03-07 00:00:00.00  Class :character
Median :1.000  Median :2014-08-27 00:00:00.00  Mode   :character
Mean   :1.436  Mean   :2015-08-15 23:52:06.43
3rd Qu.:1.000  3rd Qu.:2018-05-04 00:00:00.00
Max.   :7.000  Max.   :2023-09-29 07:00:14.00
NA's   :5880

      total_value      control_type
Min.    :     0  Mode :logical
1st Qu.: 371  FALSE:7918
Median : 749  TRUE :59934
Mean   : 24066
3rd Qu.: 6392
Max.   :22000000

```

You should be able to look at the **Min** value of `ship_date` in the summary to make sure there are not dates before 2010. You can also look at the **Max** value to see your latest `ship_date`. You should make a mental (and perhaps written) note about that max date.

We now have an object, `leso_dated` that has all the data since 2010. We'll export this for our next notebook.

## 5.12 Export cleaned data

Now that we have our data selected, mutated and filtered how we want it, we can export into `.rds` files to use in other notebooks. As you may recall from Billboard, 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 the file with 01- at the beginning so you know it came from the first notebook, and then use a short and descriptive name for the rest. No spaces. **Well-formatted, descriptive file names are important to your future self and other colleagues.**

Here is my version:

```
leso_dated |> write_rds("data-processed/01-leso-all.rds")
```

## 5.13 Render and clean up your notebook

You should Render your notebook and read it over carefully. Some things to look for:

- Make sure you have nice headlines for each of your sections.
- Make sure your navigation works for the Home and Cleaning pages.
- Flesh out your `index.qmd` file to make sure the basic assignment information is there.

## 5.14 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:

- `slice_sample(n = 10)` gives us 10 random rows from our data.
- `case_when()` allows you to categorize a new column based on logic within your data. There is more about this in the [this appendix](#).
- `summary()` gives you descriptive statistics about your tibble. We used it to check the “min” and “max” date, but you can also see averages (mean) and medians.

# 6 Military Surplus Analysis

This chapter is by Prof. McDonald, who uses macOS.

In the last chapter, we learned about the LESO data ... that local law enforcement agencies can get surplus military equipment from the U.S. Department of Defense. We downloaded a combined version of the data, modified it for our purposes (used `mutate()` to create a new column calculated from other variables in the data) and then filtered it to a specific time period.

Throughout this lesson I'll give you a chance to work out the code on your own, but then give you the answer and explain it. **You won't learn this if you just copy/paste.** You need more than code that runs; you need to understand what the code is doing.

## 6.1 Learning goals of this lesson

In this chapter we will start querying the data using **summarize with math**, basically adding 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 *in* a vector.
- We'll write Data about some of the findings, practicing data-centric ledes and sentences describing data.

## 6.2 Questions to answer

All answers should be based on **controlled** items given to **Texas agencies** from **Jan. 1, 2010 to present**.

- *How many total “controlled” items were transferred to Texas agencies, and what are they all worth?* We’ll summarize all the controlled items only to get the total quantity and total value of everything.
- *How many total “controlled” items did each agency get and how much was it all worth?* Which agency got the most stuff?
  - *How about local police agencies? What was the total quantity and value I’ll give you a list.*
- *What specific “controlled” items did each agency get and how much were they worth?* Now we’re looking at the kinds of items.
  - *What did local agencies get?*

You’ll research some of the more interesting items the agencies received 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 **Quarto Document** with the title “Texas Analysis”.
3. Save the notebook at `02-analysis.qmd`.
4. Update your `_quarto.yml` file to add the `02-analysis.qmd` to your navigation.
5. Remove the boilerplate text.
6. Create a setup section (headline, text and code chunk) that loads the tidyverse library. Include the same execution options we had in the cleaning notebook.

We’ve started each notebook like this, so you should be able to do this on your own now. That said, here is the setup chunk with options:

```
```{r}
#| label: setup
#| message: false
#| warning: false

library(tidyverse)
```
```

### 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 `leso`.
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.

```
leso <- read_rds("data-processed/01-leso-all.rds")
leso |> glimpse()
```

You should see the `leso` object in your Environment tab. You also have the glimpse like above so you an idea of what each variable is in the data.

As a review, here is what the glimpse looks like:

```
Rows: 67,852
Columns: 13
$ state           <chr> "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", "2320-01-371~
$ item_name       <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~
$ quantity        <dbl> 10, 9, 1, 10, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ui              <chr> "Each", "Each", "Each", "Kit", "Each", "Each", "Each~
$ acquisition_value <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~
$ demil_code      <chr> "D", "D", "C", "D", "C", "Q", "D", "C", "D", "D~
$ demil_ic        <dbl> 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ ship_date       <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~
$ station_type    <chr> "State", "State", "State", "State", "State", "State"~
$ total_value     <dbl> 16260.00, 3339.00, 62627.00, 172647.10, 62627.00, 65~
$ control_type    <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE~
```

## 6.4 Filter to data of interest

For this analysis we only want to look at “controlled” items, and only those items given to Texas agencies. We'll build these filters separately so we can save the data at each step.

### 6.4.1 Get the controlled items

For this analysis we want to focus on “controlled” items vs the more generic non-controlled items we learned about in the documentation. Let’s filter to capture just the controlled data for our analysis.

As you might recall from the Billboard project, the `filter()` function is our workhorse for focusing our data. In our import notebook we created our `control_type` column so we could do exactly this: Find only the rows of “controlled” items.

1. Start a new Markdown section and note you are getting controlled items.

Let’s first count how many of these items are controlled vs. non-controlled. This is much like counting songs by performer: We can use GSA and `n()` to count the number of rows for `control_type`, the column we created in the cleaning notebook.

2. Try to write a quick count of the `leso` data that uses `group_by()` on the `control_type` column and summarizes using `n()`.

```
leso |>
  group_by(control_type) |>
  summarise(number_items = n())
```

```
# A tibble: 2 x 2
  control_type number_items
  <lg1>           <int>
1 FALSE            7918
2 TRUE             59934
```

We can see from this result that there are 7918 items that are not controlled, and 59934 that are. When we write our filter, we should end up with 59934 observations, which is only the TRUE ones.

3. Start with your `leso` data, but then filter it to `control_type == TRUE`.
4. Save the result into a new tibble called `leso_c`.

```
leso_c <- leso |>
  filter(control_type == TRUE)
```

To test that this worked you could do the same GSA on `control_type` we used above, or even `glimpse()` which shows the number of rows.

If you really wanted to just count the number of rows, there is another function, `nrow()`.

```
leso_c |> nrow()
```

```
[1] 59934
```

At this point you have a new tibble called `leso_c` that has only the weapons and other controlled property, so now we can take a closer look at that data.

### 6.4.2 Filter for Texas

In this analysis we only want to look at Texas data so we'll make a new object with just that. Remember, you want to do each step and run it to make sure it is working.

1. Create a new section with headlines and text that denote you are filtering the data to Texas.
2. Create the code chunk and start your filter process using the `leso_c` tibble.
3. Use `filter()` on the `state` column to keep all rows with "TX".
4. Edit the chunk to put all that into a new object called `leso_c_tx`.
5. Print out the tibble.

```
leso_c_tx <- leso_c |>
  filter(
    state == "TX"
  )

leso_c_tx
```

```
# A tibble: 4,692 x 13
  state agency_name nsn item_name quantity ui acquisition_value demil_code
  <chr> <chr>      <chr> <chr>       <dbl> <chr>             <dbl> <chr>
1 TX    ABERNATHY ~ 1240~ SIGHT,RE~      5 Each            371 Q
2 TX    ABERNATHY ~ 2320~ TRUCK,UT~      1 Each           62627 C
3 TX    ALVARADO I~ 2310~ TRUCK,AM~      1 Each           96466 C
4 TX    ALVARADO I~ 1005~ RIFLE,5.~      1 Each            120 D
5 TX    ALVARADO I~ 1005~ RIFLE,5.~      1 Each            120 D
6 TX    ALVARADO I~ 1005~ RIFLE,5.~      1 Each            120 D
7 TX    ALVARADO I~ 1005~ RIFLE,5.~      1 Each            120 D
8 TX    ALVARADO I~ 1005~ SHOTGUN,~      1 Each            108 D
9 TX    ALVARADO P~ 1385~ UNMANNED~      1 Each           264117 Q
10 TX   ALVARADO P~ 2355~ MINE RES~     1 Each           733000 C
# i 4,682 more rows
```

```
# i 5 more variables: demil_ic <dbl>, ship_date <dttm>, station_type <chr>,
#   total_value <dbl>, control_type <lgl>
```

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”. We are also down to 4692 rows.

## 6.5 Building summaries with math

As we get into the first quest, let’s talk about “how” we go about these summaries.

When I am querying my data, I start by envisioning what the result should look like. Let’s take the first question: **How many total “controlled” items were transferred to Texas agencies, and what are they all worth?**

Let’s glimpse the data ...

```
leso_c_tx |> glimpse()
```

```
Rows: 4,692
Columns: 13
$ state          <chr> "TX", "TX", "TX", "TX", "TX", "TX", "TX", "TX"~
$ agency_name    <chr> "ABERNATHY POLICE DEPT", "ABERNATHY POLICE DEPT", "A~
$ nsn            <chr> "1240-01-540-3690", "2320-01-371-9584", "2310-01-111~
$ item_name      <chr> "SIGHT,REFLEX", "TRUCK,UTILITY", "TRUCK,AMBULANCE", ~
$ quantity       <dbl> 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1~
$ ui              <chr> "Each", "Each", "Each", "Each", "Each", "Each", "Eac~
$ acquisition_value <dbl> 371.00, 62627.00, 96466.00, 120.00, 120.00, 120.00, ~
$ demil_code     <chr> "Q", "C", "C", "D", "D", "D", "D", "Q", "C", "C"~
$ demil_ic       <dbl> 3, 1, 1, 1, 1, 1, 1, 3, 1, NA, 1, NA, 1, NA, NA, ~
$ ship_date      <dttm> 2016-02-02, 2016-03-07, 2020-12-03, 2011-09-13, 201~
$ station_type   <chr> "State", "State", "State", "State", "State", "State"~
$ total_value    <dbl> 1855.00, 62627.00, 96466.00, 120.00, 120.00, 120.00, ~
$ control_type   <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE~
```

... and then talk through the logic:

- “How many total controlled items” is how many individual things were transferred. We have this `quantity` column that has the number of items in each row, so if we want the total for the data set, we can add together all the values in that column. We do this within a `summarize()` but instead of counting rows using `n()`, we’ll use the function `sum(quantity)` which will add all the values in `quantity` column together.

- “... what are they all worth” is very similar, but we want to add together all those values in our `total_value` column. (Remember, we don’t want to use `acquisition_value` because that is the value of only ONE item, not the total for the row.)

### 6.5.1 Summarize across the data

So, let’s put this together with code.

1. Start a new Markdown section that you are getting total values.
2. Add a code chunk like below and run it.

```
leso_c_tx |>
  summarise(
    sum(quantity),
    sum(total_value)
  )
```

- ① We start with the `leso_c_tx` tibble of the controlled items then we pipe into `summarize()`.  
 ② Because we are going to add multiple things, I put them on separate lines just to make this more readable.

```
# A tibble: 1 x 2
`sum(quantity)` `sum(total_value)`
<dbl>             <dbl>
1       19578        125827483.
```

You’ll notice that the names of the columns are the function names. We can “name” our new columns just like we did in Billboard. We could call this whatever we want, but good practice is to name it what it is. We’ll use good naming techniques and split the words using `_`. I also use all lowercase characters.

3. Edit your chunk to add in the new column names and run it.

```
leso_c_tx |>
  summarise(
    summed_quantity = sum(quantity),
    summed_total_value = sum(total_value)
  )
```

- ① Note we added `summed_quantity =` on this line  
 ② and `summed_total_value =` on this line.

```
# A tibble: 1 x 2
  summed_quantity summed_total_value
            <dbl>                <dbl>
1          19578        125827483.
```

OK, from this we have learned something: The `summed_quantity` value there is the number of items all Texas law enforcement agencies have received, and the `summed_total_value` is how much they were worth.

### 6.5.2 Data Takeaway: Totals since 2010

1. Write a Data Takeaway sentence in your notebook summing up the totals in a way that you could drop into a story.

### 6.5.3 NA values in a sum, mean and median

We'll do summarize again below with grouping, but first we need to talk about `NA` values.

When we do math like this within summarize we need to take special note if our column has any blank values, called `NA`, as in “Not Available”. If there are, then you will get `NA` for the result. R will NOT do the math on the remaining values unless you tell it so. This is true not only for `sum()`, but also for `mean()` which gets an average, and for `median()` which finds the middle number in a column.

There is a way to get around this by including an argument within the mathy function: `sum(col_name, na.rm = TRUE)`.

I can show this with the `demil_ic` column which is a number datatype with some `NA` values. To be clear, the `demil_ic` variable isn't really designed to do math on it as it is really a category, but it will show what I'm talking about here.

```
# You don't have to add this to your notebook.
# I'm just explaining concepts
leso_c_tx |>
  summarise(
    dumb_sum = sum(demil_ic),
    less_dumb_sum = sum(demil_ic, na.rm = TRUE),
    dumb_avg = mean(demil_ic),
    less_dumb_avg = mean(demil_ic, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 4
  dumb_sum less_dumb_sum dumb_avg less_dumb_avg
  <dbl>       <dbl>     <dbl>       <dbl>
1      NA        6345      NA        1.39
```

So there you have examples of using `sum()` and `mean()` with and without `na.rm = TRUE`. OK, you've been warned.

## 6.6 Totals by agency

OK, your next question is this: **For each agency, how many things did they get and how much was it all worth?**

The key part of thinking about this logically is **For each agency**. That “for each” is a clue that we need `group_by()` for something. We basically need what we did above, but we first need to group\_by the `agency_name`.

Let's break this question down:

- “For each agency” tells me I need to `group_by` the `agency_name` so I can summarize totals within each agency.
- “how many total things” means how many items. Like before, we have the `quantity` variable, so we need to add all those together within summarize like we did above.
- “how much was it worth” is another sum, but this time we want to sum the `total_value` column

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.

### 6.6.1 Group\_by, then summary with math

We'll start with the `total_quantity`.

1. Add a new section (headline, text and chunk) that describes the second 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.

```
leso_c_tx |>  
  group_by(agency_name) |>  
  summarize(  
    summed_quantity = sum(quantity)  
)
```

- ① We start with the `leso_c_tx`, which is the “controlled” data, and then ...
- ② 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 not be able to *see* the groupings at this point, so I usually write `group_by()` and `summarize()` together.
- ③ In `summarize()` we first name our new column: `summed_quantity`, then 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 this 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.

```
# A tibble: 306 x 2  
  agency_name           summed_quantity  
  <chr>                  <dbl>  
1 ABERNATHY POLICE DEPT          6  
2 ALVARADO ISD PD K-12            6  
3 ALVARADO POLICE DEPT           2  
4 ALVIN POLICE DEPT             125  
5 ANDERSON COUNTY SHERIFFS OFFICE    7  
6 ANDREWS COUNTY SHERIFF OFFICE      12  
7 ANTHONY POLICE DEPT              10  
8 ARANSAS PASS POLICE DEPARTMENT     28  
9 ARCHER COUNTY SHERIFF OFFICE        3  
10 ATLANTA POLICE DEPT            233  
# i 296 more rows
```

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.

## 6.6.2 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 summed\_total\_value and run it.

```
leso_c_tx |>
  group_by(agency_name) |>
  summarize(
    summed_quantity = sum(quantity),
    summed_total_value = sum(total_value) ①
  )
```

① Here we add the summed\_total\_value.

```
# A tibble: 306 x 3
  agency_name          summed_quantity summed_total_value
  <chr>                  <dbl>                <dbl>
1 ABERNATHY POLICE DEPT      6                64482
2 ALVARADO ISD PD K-12       6                97054
3 ALVARADO POLICE DEPT       2                997117
4 ALVIN POLICE DEPT        125               445997.
5 ANDERSON COUNTY SHERIFFS OFFICE 7                733720
6 ANDREWS COUNTY SHERIFF OFFICE 12                1476
7 ANTHONY POLICE DEPT        10                7490
8 ARANSAS PASS POLICE DEPARTMENT 28               514746.
9 ARCHER COUNTY SHERIFF OFFICE 3                 1101000
10 ATLANTA POLICE DEPT       233               89150.
# i 296 more rows
```

## 6.6.3 Check the math

If you wanted to test this (and that 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 my logic works.

```
leso_c_tx |>
  filter(agency_name == "ABERNATHY POLICE DEPT") |>
  select(agency_name, quantity, total_value)
```

```
# A tibble: 2 x 3
  agency_name      quantity total_value
  <chr>            <dbl>      <dbl>
1 ABERNATHY POLICE DEPT     5        1855
2 ABERNATHY POLICE DEPT     1       62627
```

If we look at the `quantity` column there we can eyeball all the rows and add them on a calculator. Our answer should match our `summed_quantity` in the summary table.

If we add up the `total_value` rows then we should end up with the same number as `summed_total_value` above.

#### 6.6.4 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 `summed_total_value` in `descending` order.

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

```
leso_c_tx |>
  group_by(agency_name) |>
  summarize(
    summed_quantity = sum(quantity),
    summed_total_value = sum(total_value)
  ) |>
  arrange(summed_total_value |> desc())
```

①

① Adding the pipe and arrange here.

```
# A tibble: 306 x 3
  agency_name      summed_quantity summed_total_value
  <chr>            <dbl>          <dbl>
1 HOUSTON POLICE DEPT     2396        7307059.
2 JEFFERSON COUNTY SHERIFFS OFFICE   206        3420305.
3 SAN MARCOS POLICE DEPT      525        3238539.
4 DPS SWAT- TEXAS RANGERS     137        3030695
5 AUSTIN POLICE DEPT      1399        2721969.
6 HARRIS COUNTY CONSTABLE PCT 3    101        1953836.
7 MILAM COUNTY SHERIFF DEPT     78        1531325.
8 HARRIS COUNTY SHERIFF'S OFFICE   12        1527065
9 MARSHALL POLICE DEPT      64        1502568.
10 DEPT OF CRIM JUSTICE OIG      4        1446516
# i 296 more rows
```

So now we've sorted the results to put the highest `summed_total_value` at the top.

Remember, there are two ways we can set up that `arrange()` function in descending order:

- `arrange(summed_total_value |> desc())`
- `arrange(desc(summed_total_value))`

Both work and are correct. It really is your preference.

### 6.6.5 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, in fact). 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 we have almost a million people.
- But what about San Marcos (pop. 63,220)? Or Milam County (pop. 24,770)? Those are way smaller cities and law enforcement agencies. They might be worth looking into.

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

### 6.6.6 Data Takeaway: Texas totals

1. After your last code chunk here, write a Data Takeaway from what you see in this result.

## 6.7 Looking a local agencies

Our second quest had a second part: **How about local police agencies? What was the total quantity and value?**

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

### 6.7.1 Save our “by agency” list

Since we want to take an existing summary and add 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 the result into a new tibble. Name it `tx_agency_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.

```

tx_agency_totals <- leso_c_tx |>
  group_by(agency_name) |>
  summarize(
    summed_quantity = sum(quantity),
    summed_total_value = sum(total_value)
  ) |>
  arrange(summed_total_value |> desc())

tx_agency_totals

```

① Creating the new object `tx_agency_totals` to put this summary into.

② Here we print out the new object, which we can reuse later.

```

# A tibble: 306 x 3
  agency_name          summed_quantity summed_total_value
  <chr>                  <dbl>                <dbl>
1 HOUSTON POLICE DEPT      2396            7307059.
2 JEFFERSON COUNTY SHERIFFS OFFICE 206        3420305.
3 SAN MARCOS POLICE DEPT     525             3238539.
4 DPS SWAT- TEXAS RANGERS    137             3030695
5 AUSTIN POLICE DEPT       1399            2721969.
6 HARRIS COUNTY CONSTABLE PCT 3   101             1953836.
7 MILAM COUNTY SHERIFF DEPT    78              1531325.
8 HARRIS COUNTY SHERIFF'S OFFICE 12              1527065
9 MARSHALL POLICE DEPT       64              1502568.
10 DEPT OF CRIM JUSTICE OIG    4               1446516
# i 296 more rows

```

## 6.7.2 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:

```

# Just for explanation. Don't add to notebook.
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 items is officially called a “vector”.) Think of it like this in plain English: *Filter the column for things in this list.*

```
# Just for explanation. Don't add to notebook.  
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.

```
# Just for explanation. Don't add to notebook.
list_of_strings <- c(
  "This string",
  "That string"
)
data |>
  filter(col_name %in% list_of_strings)
```

This last version is the easiest to read and the most flexible to maintain. We can make adjustments to the list as needed.

### 6.7.3 Create a vector to build this filter

In the interest of time, I'm going to give you the list of local police agencies to filter on. To be clear, I did considerable work to figure out the exact names of these agencies. I consulted a different data set that lists all law enforcement agencies in Texas and then I used some creative filtering to find their "official" names in the leso data. It also helps that I'm familiar with local cities and counties so I can recognize the names. **I don't want to get sidetracked on that process here so I'll give you the list and show you how to use it.**

1. Create a new section (headline, text and chunk) and describe you are filtering the summed quantity/values for local agencies.
  2. Add the code below into that chunk.

```
local_agencies <- c(  
  "AUSTIN PARKS POLICE DEPT", #NI  
  "AUSTIN POLICE DEPT",  
  "BASTROP COUNTY SHERIFF'S OFFICE",  
  "BASTROP POLICE DEPT".
```

```

"BEE CAVE POLICE DEPT",
"BUDA POLICE DEPT",
"CALDWELL COUNTY SHERIFFS OFFICE",
"CEDAR PARK POLICE DEPT",
"ELGIN POLICE DEPARTMENT",
"FLORENCE POLICE DEPT", #NI
"GEOGETOWN POLICE DEPT",
"GRANGER POLICE DEPT", #NI
"HAYS CO CONSTABLE PRECINCT 4",
"HAYS COUNTY SHERIFFS OFFICE",
"HUTTO POLICE DEPT",
"JARRELL POLICE DEPT", #NI
"JONESTOWN POLICE DEPT", #NI
"KYLE POLICE DEPT",
"LAGO VISTA POLICE DEPT",
"LAKEWAY POLICE DEPT", #NI
"LEANDER POLICE DEPT",
"LIBERTY HILL POLICE DEPT", #NI
"LOCKHART POLICE DEPT",
"LULING POLICE DEPT",
"MANOR POLICE DEPT",
"MARITINDALE POLICE DEPT", #NI
"PFLUGERVILLE POLICE DEPT",
"ROLLINGWOOD POLICE DEPT", #NI
"SAN MARCOS POLICE DEPT",
"SMITHVILLE POLICE DEPT", #NI
"SUNSET VALLEY POLICE DEPT", #NI
"TAYLOR POLICE DEPT", #NI
"THRALL POLICE DEPT", #NI
# TEXAS STATE UNIVERSITY HI_ED
"TRAVIS COUNTY SHERIFFS OFFICE",
# TRAVIS CONSTABLE OFFICE,
# SOUTHWESTERN UNIVERSITY HI_ID
"WESTLAKE HILLS POLICE DEPT", #NI
"UNIV OF TEXAS SYSTEM POLICE HI_ED",
"WILLIAMSON COUNTY SHERIFF'S OFFICE"
)

tx_agency_totals |>
  filter(agency_name %in% local_agencies)

```

(3)

(4)

- ① We start by giving our list of agencies a name: `local_agencies`. This creates an R object

that we can reuse. We will need this list a number of times, and it makes sense to manage it in once place instead of each time we need it.

- ② Next we fill that object with a list of agency names. Again, I did some pre-work to figure out those names that we aren't covering here, and in some cases there is a comment `#NI` next to them. That is a note to myself that the particular agency is NOT INCLUDED in our data, which means I haven't confirmed the name spelling. It could be at a later date the Austin Parks department gets equipment, but they list their name as "CITY OF AUSTIN PARKS" instead of "AUSTIN PARKS POLICE DEPT" and it would not be filtered properly. This is another example that **your most important audience for your code is your future self.**
- ③ Now that our list is created, we can use it to filter our `tx_agency_totals` data. So, we start with that data, and then ...
- ④ We pipe into `filter()`, but inside our filter we don't set it == to a single thing, instead we say: `agency_name %in% local_agencies`, which says look inside that `agency_name` column and keep any row that has a value that is also in our `local_agencies` vector.

```
# A tibble: 17 x 3
  agency_name          summed_quantity summed_total_value
  <chr>                  <dbl>                <dbl>
1 SAN MARCOS POLICE DEPT      525            3238539.
2 AUSTIN POLICE DEPT        1399            2721969.
3 UNIV OF TEXAS SYSTEM POLICE HI_ED     3            1305000
4 LEANDER POLICE DEPT       212            1190263.
5 GEORGETOWN POLICE DEPT      41             1097977.
6 CALDWELL COUNTY SHERIFFS OFFICE    339            995833.
7 CEDAR PARK POLICE DEPT      106            971886.
8 BASTROP COUNTY SHERIFF'S OFFICE    266            719301.
9 HAYS COUNTY SHERIFFS OFFICE      384            456141.
10 WILLIAMSON COUNTY SHERIFF'S OFFICE   166            75326
11 LOCKHART POLICE DEPT        16             57259.
12 BEE CAVE POLICE DEPT       16             50360.
13 TRAVIS COUNTY SHERIFFS OFFICE     28             36302
14 HUTTO POLICE DEPT         90             13513.
15 BASTROP POLICE DEPT        10             4990
16 LULING POLICE DEPT        16             4700.
17 BUDA POLICE DEPT         16             1736.
```

#### 6.7.4 Data Takeaway: Local totals

This filters our list of agencies to just those in Central Texas. Do you see anything interesting there?

1. Write your Data Takeaway about what you've found.

## 6.8 Types of items shipped to each agency

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.

Our question is this: **What specific “controlled” items did each agency get and how much were they worth?**

In some cases an agency might get the same item shipped to them at different times. For instance, “AUSTIN POLICE DEPT” has multiple rows where they get a single “ILLUMINATOR,INTEGRATED,SMALL ARM” shipped to them on the same date, but then on other dates they have the same item but the quantity is 30. We want all of these “ILLUMINATOR,INTEGRATED,SMALL ARM” for the Austin police added together into a single record.

The logic works like this:

- Start with the controlled data, and then ...
- Group by the `agency_name` and `item_name`, which will group all the rows where those values are the same. All “AUSTIN POLICE DEPT” rows with “ILLUMINATOR,INTEGRATED,SMALL ARM” will be considered together, and then ...
- Summarize to sum the `quantity`, and then do the same for `total_value`.

The code for this is very similar to what we did above when we summaries agencies, except we are grouping by two things, the `agency_name` and the `item_name`. Let's do it:

1. Create a new section (headline, text and code chunk) and describe that you are finding the sums for each item that each agency has received since 2010.
2. Consult (or even copy) the code you wrote when you created the agency totals, but modify the `group_by()` to add the `item_name`, like this: `group_by(agency_name, item_name)`.
3. Be sure you rename your created R objects, too, perhaps to `tx_agency_item_totals`.

```
tx_agency_item_totals <- leso_c_tx |>  
  group_by(agency_name, item_name) |>  
  summarize(  
    summed_quantity = sum(quantity),  
    summed_total_value = sum(total_value)  
  ) |>  
  arrange(summed_total_value |> desc())  
  
tx_agency_item_totals
```

- ① Be sure to create a new object
- ② Here we add the second grouping variable

```
# A tibble: 1,339 x 4
# Groups:   agency_name [306]
  agency_name      item_name summed_quantity summed_total_value
  <chr>            <chr>           <dbl>                  <dbl>
1 HOUSTON POLICE DEPT AIRCRAFT~          1  5390000
2 DPS SWAT- TEXAS RANGERS MINE RES~        4  2611000
3 DEPT OF CRIM JUSTICE OIG TRUCK,CA~        4  1446516
4 UNIV OF TEXAS SYSTEM POLICE HI_~ MINE RES~       2  1228000
5 JEFFERSON COUNTY SHERIFFS OFFICE HELICOPT~       1  922704
6 BURKBURNETT POLICE DEPT MINE RES~        1  865000
7 CLEBURNE POLICE DEPT MINE RES~        1  865000
8 CUERO POLICE DEPT MINE RES~        1  865000
9 HARRIS COUNTY CONSTABLE PCT 3 MINE RES~        1  865000
10 MARSHALL POLICE DEPT MINE RES~       1  865000
# i 1,329 more rows
```

This reuse of code like this – copying the agency grouping code and editing it to add the item\_name value – is very common in coding, and there is nothing wrong with doing so as long as you are careful.

When you reuse code, review it carefully so you don't override things by accident. In this instance, our original code created an R object: `tx_agency_totals <- leso_c_tx |> ...` that holds the result of our functions, and we call that later to view it. If we reuse this code and don't update that object name, we'll reset the values inside that already-existing object, which was not our intent. We want to create a NEW thing `tx_agency_item_totals` so we can use that later, too. And if we don't update the “peek” at the object, we'll be looking at the old one instead of the new one.

### Note

With that last code chunk you might see a warning in your R Console: `summarise() has grouped output by 'agency_name'.` You can override using the `.groups` argument. This is not a problem, it's just a reminder that when we group by more than one thing, the first grouping is retained when future functions are applied to this result. It's more confusing than helpful, to be honest. Just know if we wanted to do further manipulation, we might need to use `ungroup()` first.

### 6.8.1 Items for local agencies

Just like we did for our agency totals, we want to filter this list of items to those sent to our local agencies. However, this time we've already created the list of local agencies so we don't have to redo that part ... we just need to filter by it.

1. Start a new section (headline, text) and explain that you are looking at items sent to local agencies.
2. Use `filter()` to focus the data just on or `local_agencies`.

```
tx_agency_item_totals |>
  filter(agency_name %in% local_agencies)
```

```
# A tibble: 149 x 4
# Groups:   agency_name [17]
  agency_name      item_name summed_quantity summed_total_value
  <chr>            <chr>           <dbl>              <dbl>
1 UNIV OF TEXAS SYSTEM POLICE HI_~ MINE RES~          2        1228000
2 AUSTIN POLICE DEPT             HELICOPT~         1        833400
3 CEDAR PARK POLICE DEPT       MINE RES~          1        733000
4 GEORGETOWN POLICE DEPT       MINE RES~          1        733000
5 LEANDER POLICE DEPT          MINE RES~          1        733000
6 SAN MARCOS POLICE DEPT      MINE RES~          1        733000
7 BASTROP COUNTY SHERIFF'S OFFICE MINE RES~          1        658000
8 AUSTIN POLICE DEPT           IMAGE IN~         85        471053.
9 SAN MARCOS POLICE DEPT      CAPABILI~         1        464109
10 SAN MARCOS POLICE DEPT      UNMANNED~        3        451684
# i 139 more rows
```

Because our original list arranged the data by the most expensive items, we can see that here. But it might be easier to rearrange the data by agency name first, then the most expensive items.

3. **EDIT your chunk** to add an `arrange` function.
4. Within the `arrange`, set it to `agency_name` first, then `summed_total_value` in descending order.

```
tx_agency_item_totals |>
  filter(agency_name %in% local_agencies) |>
  arrange(agency_name, desc(summed_total_value))
```

```

# A tibble: 149 x 4
# Groups: agency_name [17]
  agency_name      item_name      summed_quantity summed_total_value
  <chr>          <chr>           <dbl>            <dbl>
1 AUSTIN POLICE DEPT HELICOPTER,FLIGHT TRAI~        1            833400
2 AUSTIN POLICE DEPT IMAGE INTENSIFIER,NIGH~       85            471053.
3 AUSTIN POLICE DEPT SIGHT,THERMAL                 29            442310
4 AUSTIN POLICE DEPT PACKBOT 510 WITH FASTA~        4            308000
5 AUSTIN POLICE DEPT SIGHT,REFLEX                  420            161177.
6 AUSTIN POLICE DEPT ILLUMINATOR,INTEGRATED~       135            141470
7 AUSTIN POLICE DEPT RECON SCOUT XT                 8            92451.
8 AUSTIN POLICE DEPT TELESCOPE,STRAIGHT            40            57960
9 AUSTIN POLICE DEPT TEST SET,NIGHT VISION ~        2            55610
10 AUSTIN POLICE DEPT POWER SUPPLY ASSEMBLY         63            21120.
# i 139 more rows

```

### 6.8.2 Research some interesting items

You'll want a little more detail about some of these items for your data drop. 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 search online to learn enough to at least describe some of these items. There are a couple of ways to go about researching items.

1. Simply search for the items on Google, [like this](#).
2. Each item has a “National Stock Number,” which is an ID for a government database of supplies. You can search the data for the `nsn` value and then look up that value online.

Let do an example, looking up “ILLUMINATOR,INTEGRATED,SMALL ARMS”. First we filter the data to find the item. (I’m also using `select` so we can control the output and see it in this book, but you might not want that in your notebook so you can also check `ship_dates`, etc.)

```

leso_c_tx |>
  filter(
    item_name == "ILLUMINATOR,INTEGRATED,SMALL ARMS",
    agency_name == "AUSTIN POLICE DEPT"
  ) |>
  select(item_name, nsn)

```

```

# A tibble: 100 x 2
  item_name      nsn
  <chr>          <chr>

```

```
1 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-571-1258
2 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
3 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
4 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
5 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
6 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
7 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
8 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
9 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
10 ILLUMINATOR, INTEGRATED, SMALL ARMS 5855-01-534-5931
# i 90 more rows
```

It looks like most of these illuminators use this nsn: 5855-01-534-5931.

Now we go to the website <https://nationalstocknumber.info/> and plug that number into the search bar. We get a couple of returns, and if we click on one we [get this page](#). It gives us a description of the item:

“A device which is a combination of several lasers and white light illumination used to provide multiple capabilities for engaging targets and providing light. The device may contain a flashlight or other white light illumination source, an illuminator, infrared and stand alone aiming lasers/pointers. The device has the capability to mount on an individual weapon.”

Not every item is in the database, but it is worth checking.

### 6.8.3 Data Takeaway: Local items

Now that you've done a bit of research into some of these items, write prose-style data takeaways about items of interest.

## 6.9 Update index summary

I know there has been a lot of new code here, but learning the code is only half the battle. What have we learned about the data? It's time to summarize our findings.

1. Open your `index.qmd` file.
2. After the text that is here, pick the most interesting data fact you've found and write a story lede and source graf.
3. Follow that with the rest of your data takeaways, written as prose that you could drop into a story.

If you need a reminder of the style, look at the [Billboard example](#)

## 6.10 Turn in your project

1. Make sure everything runs and Renders properly.
2. Publish your changes to Quarto Pub and include the link to your project in your index notebook so I can bask in your glory.
3. Zip your project folder. (Or export to zip if you are using posit.cloud).
4. Upload to the Canvas assignment.

! Important

To be clear, it is your zipped project I am grading. The Quarto Pub link is for convenience.

## 6.11 What we learned in this chapter

- We used `sum()` within a `group_by()/summarize()` function to add values within a column. Sometimes `sum()` needs the argument `na.rm = TRUE`.
- 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 values `%in%` that vector.
- We used `nrow()` to count the number of rows in an object.

## **Part III**

# **Visualization**

# 7 Intro to ggplot

The chapter is by Prof. Lukito with some updates by Prof. McDonald. We don't remember who wrote what, so perhaps it's the McLuk version.

This week, we'll move on from summaries to talking about data visualizations ("data viz"), an essential skill for any data journalist. While there are a lot of different ways to make figures and graphs, this chapter will focus on one popular R package for data viz: `ggplot`.

## 7.1 Goals for this section

In this chapter, we'll learn the basics of **data visualization** using the Grammar of Graphics principles. We'll start with some smaller datasets to give you a sense of how the code works. And, in the next chapter, you'll apply this to a dataset we have already used in the class.

Our learning goals are:

- To learn about the Grammar of Graphics
- To make scatterplots
- To make bar charts

## 7.2 Introduction to ggplot

`ggplot2` is the data visualization library within Hadley Wickham's [tidyverse](#). It is a **beast** of a package because it supports a whole variety of different types of data visualizations, from [bar charts](#) and [line charts](#) to fancy [choropleth maps](#) and [animated figures](#).

Even though the package is called `ggplot2`, the function to make graphs is just `ggplot()`. So, for simplicity, we'll just call everything `ggplot`.

The `ggplot` package relies on a concept called the [Grammar of Graphics](#), hence the `gg` in `ggplot`. The basic logic of the Grammar of Graphics is that any graph you could ever want to build will need similar things: a data set, some information about the scales of your variables, and the type of figure or graph that you want to create. These various things can be "layered" on top of each other to create a visually pleasing plot.

Folks who have used Adobe creative programs (e.g., Photoshop, Illustrator, etc.) can think about it like laying an image: each layer in your image *should do something* to change the image. Likewise, each layer in a `ggplot` figure will add to the overall graph.

### 7.2.1 What `ggplot` is best for

Let me just start by saying that I'm a total `ggplot` geek. I'll talk about `ggplot` figures the way people talk about new TikTok trends. When producing figures and graphs in R, `ggplot` is the **absolute best approach** because you'll see the results right in your notebook. And, basic data visualizations are an absolutely essential skill for any data journalist: it helps you find important things in your data that you may ultimately report on. So, `ggplot` is important for any R-based data journalism project.

That being said, there are less complicated ways of creating publishable graphics. Tools like [Datawrapper](#) and [Flourish](#) can produce equally beautiful graphics without the code. So why learn `ggplot`? Because, (1) `ggplot` is super useful when you're just learning about the data and (2) to get good enough in `ggplot` to make publishable graphics, you have to practice, practice, practice. Yes, `ggplot` is a big package with lots of nuance. But the more you take the time to learn it, the more you will master it.

### 7.2.2 The Grammar of Graphics

This section was inspired by [Matt Waite](#) and the [BBC Visual Cookbook](#).

As noted above, the `gg` in `ggplot` stands for “Grammar of Graphics,” which is a fancy way of saying we’ll build our charts layer by layer. There are three main things you need to make a plot:

- **data:** You have to tell `ggplot` the name of the object with your data.
- **mapping:** Defines how variables in your dataset are mapped to visual properties (**aesthetics**) of your plot. i.e., which columns are on the x or y axis. We use the function `aes()` to describe these aesthetics.
- **geometries:** This how you describe the shape of your visualization, whether it’s lines, bars, points, or something else. We do this through different `geom_()` functions.

In addition to these three things, there are lots of helper layers we’ll learn about along the way, including:

- **themes** (“theme”): this is where you tell R the font you’d like to use, the background color, and other things you want to “pretty up” the data viz.
- **coord\_flip:** a special layer for flipping the chart

- **scales**: transforming the data to make the plot more read-able
- **labels** (“labs”): for making titles and labels
- **facets**: For graphing many elements of the same data set in the same space (one dataset, multiple figures)

This all may seem complicated now, but it'll make sense once we start putting together these layers together, one at a time. After all, the best way to learn any R package is to **do it**.

## 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 **Quarto Website** project, name it `yourname-ggplot` and save it in your `rwd` folder.
3. (No need for a folder structure, we'll do this all in one file.)
4. We'll use our `index.qmd` file for everything for this project, so update the title as “ggplot practice”, remove the boilerplate and create a setup section that loads `library(tidyverse)` and `library(janitor)`, like we do with every notebook.

The `ggplot` package is a part of `tidyverse`, so we don't need to do anything special there.

### 7.3.1 Install palmerpenguins

For this lesson, we will install a new R package, which will give us access to some data to work through some visualization techniques. I'd like you to meet the Palmer Penguins ...

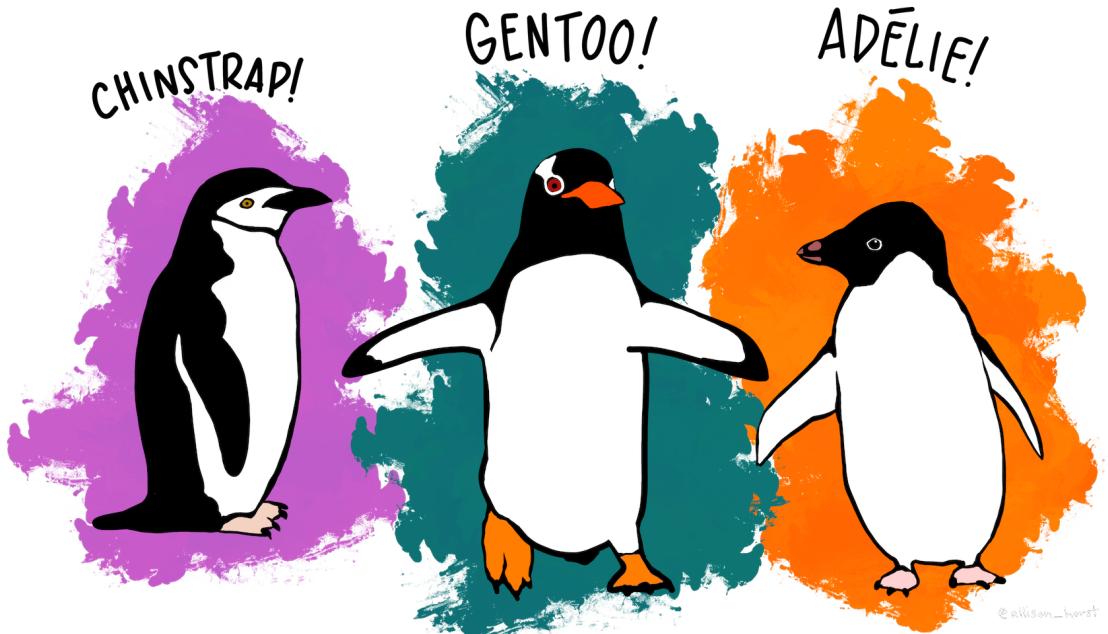


Figure 7.1: Artwork by @allison\_horst

The [palmerpenguins](#) package includes scientific measurements of penguins observed on islands in the Palmer Archipelago near Palmer Station, Antarctica. The package and the awesome artwork is maintained by Allison Horst. It's also a great one for data exploration & visualization so we'll use it to learn about ggplot.

Let's install and load it.

1. In your Console, run `install.packages("palmerpenguins")`.
2. In your setup section, add the library and rerun the chunk: `library(palmerpenguins)`

Remember you only have to **install** a package once on your computer, but you should load the **library** at the top of each notebook that uses it. (i.e., don't leave the `install.packages()` part in your notebook.)

## 7.4 The layers of ggplot

### i Note

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

Before we dive into `ggplot`, let's peek at our data.

1. Start a new section “First plot”
2. Add some text that you are studying pretty penguins.
3. Add a code chunk like below and run it to see what the `penguins` dataset looks like.

```
penguins
```

```
# A tibble: 344 x 8
  species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>        <dbl>         <dbl>          <int>        <int>
1 Adelie   Torgersen     39.1         18.7          181       3750
2 Adelie   Torgersen     39.5         17.4          186       3800
3 Adelie   Torgersen     40.3         18            195       3250
4 Adelie   Torgersen      NA           NA            NA        NA
5 Adelie   Torgersen     36.7         19.3          193       3450
6 Adelie   Torgersen     39.3         20.6          190       3650
7 Adelie   Torgersen     38.9         17.8          181       3625
8 Adelie   Torgersen     39.2         19.6          195       4675
9 Adelie   Torgersen     34.1         18.1          193       3475
10 Adelie  Torgersen      42           20.2          190       4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

As noted above, the data are measurements from a number of penguins, things like species, sex, bill length, etc.

Among the variables in `penguins` are:

- `flipper_length_mm`: length of a penguin’s flipper, in millimeters.
- `body_mass_g`: body mass of a penguin, in grams.
- `species`: a penguin’s species (Adelie, Chinstrap, or Gentoo).

With these variables, we’ll explore the relationship between flipper lengths and body masses of these penguins, taking into consideration the species of the penguin.

### 7.4.1 Our goal chart

Our goal is to recreate this chart:

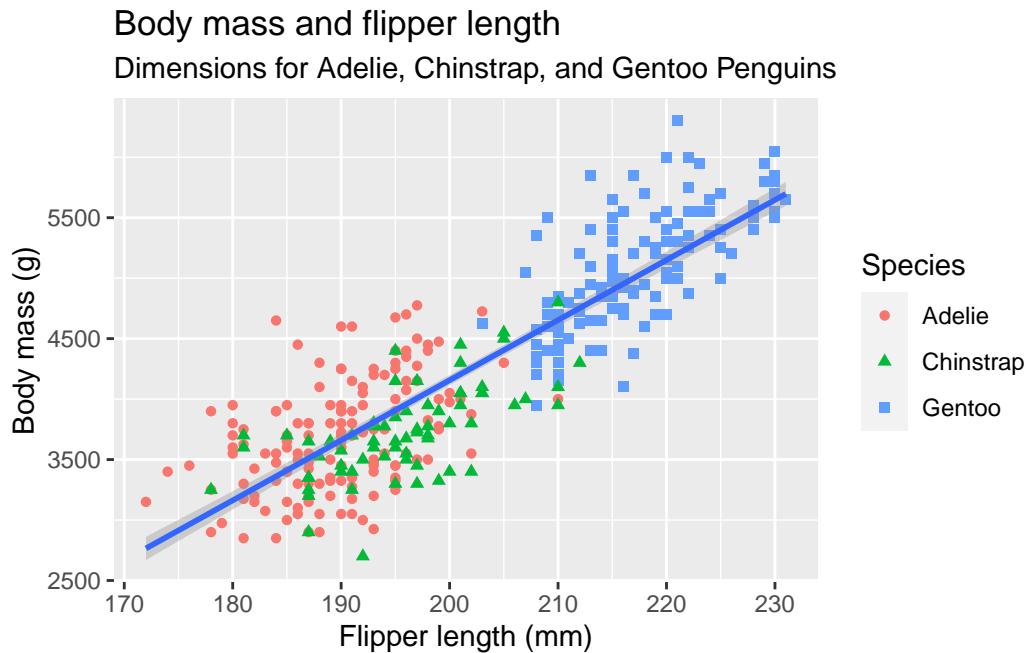


Figure 7.2: Our goal graphic

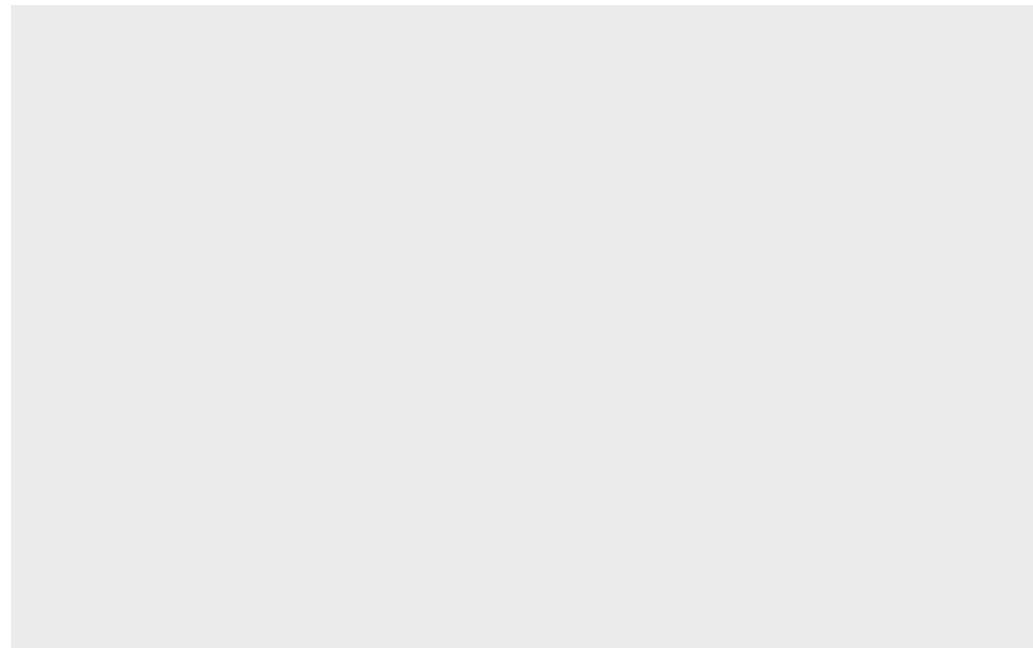
### 7.4.2 1st: the data

When working with the `ggplot2` package, you'll start nearly every figure with the `ggplot()` function. In the `ggplot()` function, you'll tell R what data you're using, and the coordinate system you want to build based on the data.

The first thing you'll want to do is tell `ggplot` the dataset you want to use (in this case, `penguins`). Let's do that now.

1. Edit your first plot section,
2. Make a new code chunk and add the code below.

```
ggplot(penguins)
```



This tells us... absolutely nothing! But that's not surprising: you haven't even told `ggplot` what variables you want to focus on or the way you want to visualize the data. To do that, you'll need a mapping argument.

#### 7.4.3 2nd: Map the data

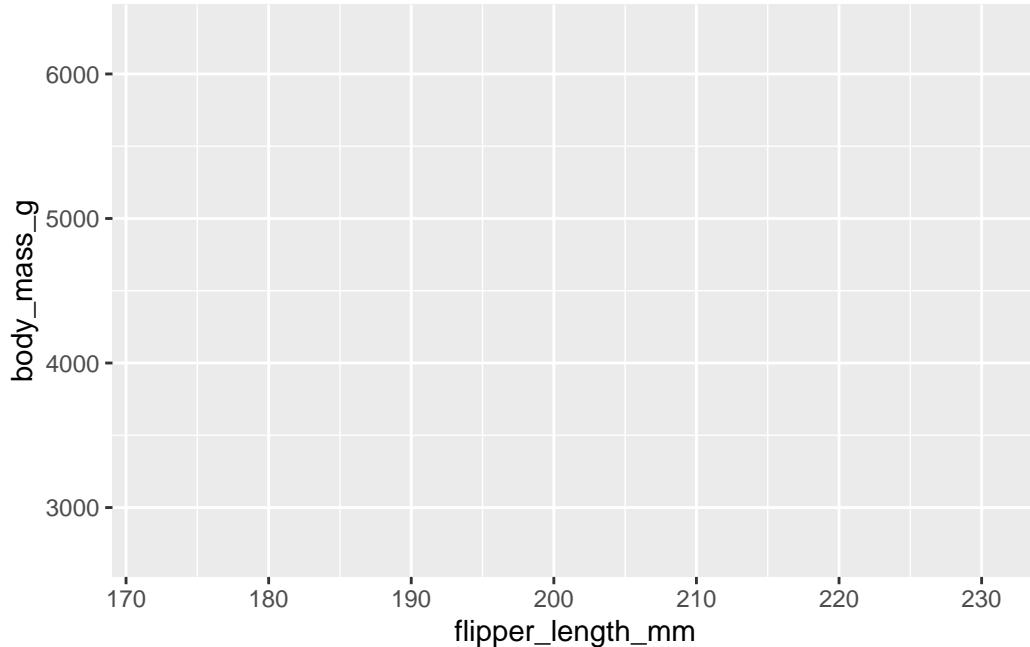
We use the `aes()` (short for “aesthetic”) to describe which data to plot and where. This is considered a *mapping* argument, because you use this argument to tell `ggplot` how you want to map your data.

The workhors of `aes()` is to indicate the variable to plot horizontally (the `x` axis) and vertically (the `y` axis). So your `aes()` argument will look something like this: `aes(x = some_variable, y = another_variable)`.

In our case, set our `flipper_length_mm` (the flipper length) for `x` and `body_mass_g` for `y`:

1. **Edit** your code to add the following line of code. Note I've rearranged the indenting, too.

```
ggplot(  
  penguins,  
  aes(x = flipper_length_mm, y = body_mass_g)  
)
```



We are getting closer! The plot knows what data we are using and the “range” of that data, so it has added tick marks and values along our `x` and `y` axes.

#### 7.4.4 3rd: geometries

Now that we know which data to apply, we have to make a choice on how we want to show the shape of that data on our plot, and we do that through **geometries** (or “geoms” as we call them.) We’ll talk about different ones in a minute, but we’ll start with plotting our data as points.

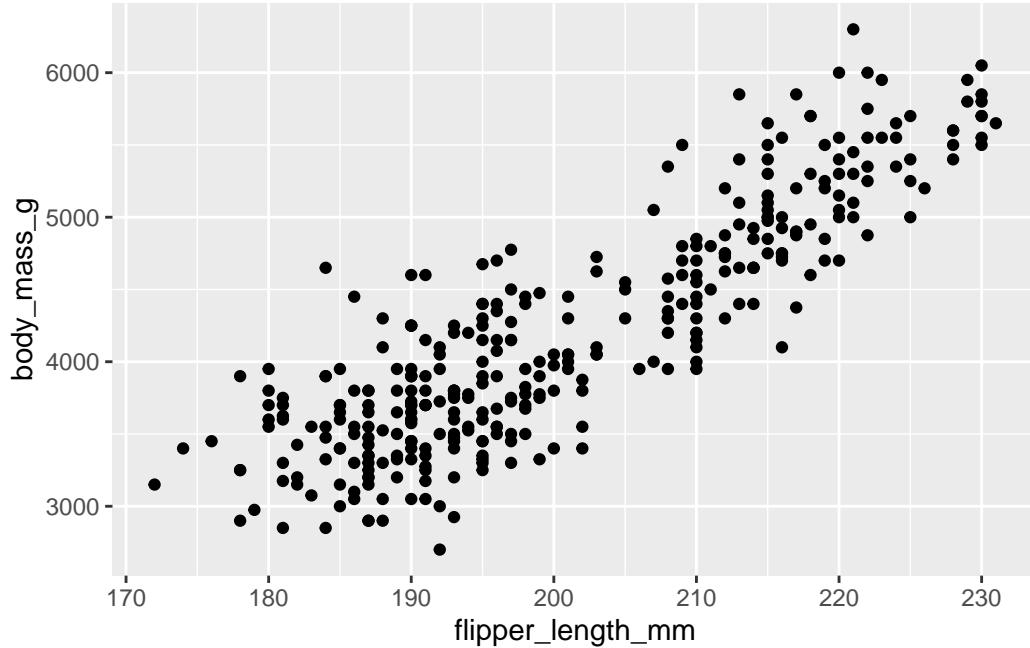
This is our first *added* layer on our chart, and therefore introduction of the `+` option within `ggplot`. We use `+` to add on layers of information in our grammar of graphics. It’s *sort of* like the pipe `|>` in the way it works, but we use `+` when adding `ggplot` layers.

1. **Edit** your chunk to add the `+` and the `geom_point()` functions as noted below.

(I *really* recommend you type the additions so you can see how RStudio helps fill the code.)

```
ggplot(
  penguins,
  aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point() ①  
②
```

- ① Don't forget the `+` here
- ② The `geom_point()` plotted dots based on the values set in our `aes()` function.



OK, now we are getting somewhere. We can **see** each penguin plotted on the chart, and we can generally see that as the birds get heavier (higher on vertical axis) their flippers are also longer (to the right on the horizontal axis.)

#### **i** Note

We do get this warning: `Warning: Removed 2 rows containing missing values geom_point()`. This is telling us there are two rows of data that don't have one of the two values, so they were dropped. Like R, ggplot2 subscribes to the philosophy that missing values should never silently go missing. As journalists, we just need to make sure we know why something is missing and note if it is important.

#### **7.4.4.1 Other geometries**

There are [many geoms](#), but here are a few common ones:

- `geom_point()` adds dots onto the grid based on the data. We used them above.
- `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.

#### 7.4.5 Chart code review

Let's review the different parts of this code. Note I've rewritten it a bit here to put some arguments on their own line so it is more readable.

```
ggplot(①
  penguins,
  aes(x = flipper_length_mm, y = body_mass_g)②
) +③
  geom_point()④
```

- ① `ggplot()` is the function we use to make a chart.
- ② Inside of `ggplot()` the first argument is the data, in this case the `penguins` data.
- ③ The second argument is the `aes()` function, where we apply our **aesthetics**. Aesthetics describe how we will paint our data onto the plot. Inside of this function, we set values for the `x` and `y` axis so we know WHERE to paint. There are some other aesthetic values we can paint with data, and we will.
- ④ Lastly we add on a layer (with `+` on the previous line) to add `geom_point()` to paint "points" on top of the plot based on the set aesthetics. (It's like [pointillism](#).)

#### ! Important

You might see this same code written different ways. Let's talk about why.

#### Verbose arguments

```
ggplot(data = penguins,
       mapping = aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

The above version adds `data =` and `mapping =` to the first two arguments. It is the official descriptive way to do this, and how Hadley Wickham describes it in [R for Data Science](#). But, since the first argument can assumed to be the data we often don't include it unless we are trying to differentiate with different data used later. Same goes for `mapping = ...` we don't specify it unless necessary.

## Piping into ggplot

```
penguins |>  
  ggplot(aes(x = flipper_length_mm, y = body_mass_g)) +  
  geom_point()
```

In this case we start with the data AND THEN pipes it into `ggplot()` as the first argument. Prof. McDonald often does it this way. The scoundrel.

### 7.4.6 Global vs local aesthetics

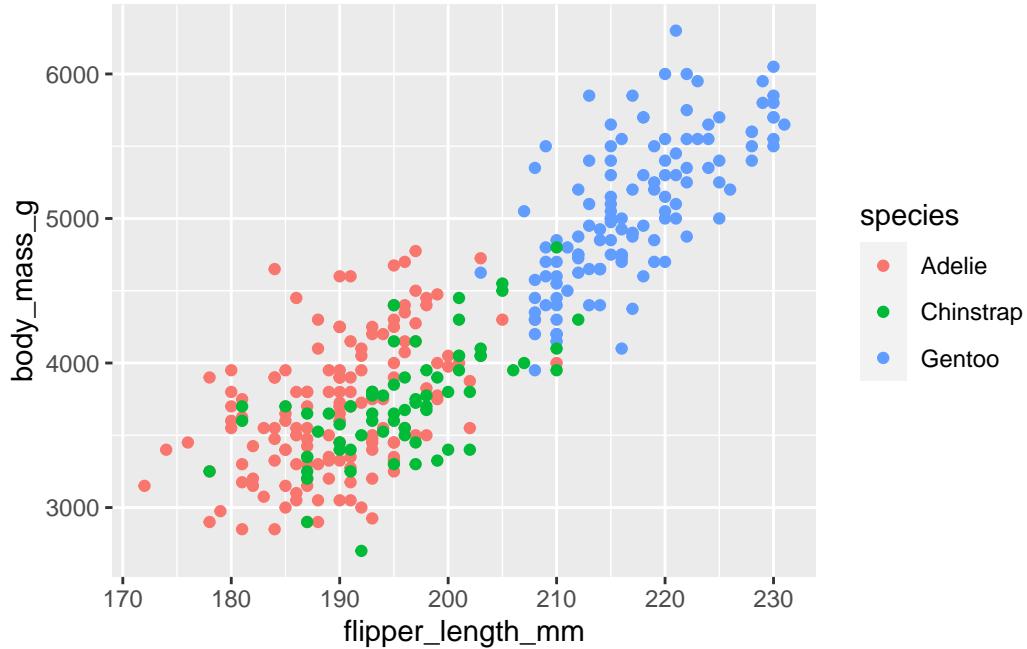
Our initial plot appears to show that flipper length is related to the weight of the penguin, but it's always a good idea to be skeptical of any apparent relationship between two variables and ask if there may be other variables that explain or change the nature of this apparent relationship. For example, does this relationship exist for all three species?

Let's color our points based on the species to see.

1. **Edit** your plot and add the color mapping to the `aes()` function, like below. **NOTE** I've also rearranged some of the code to make it more readable.

```
ggplot(  
  penguins,  
  aes(x = flipper_length_mm,  
      y = body_mass_g,  
      color = species))  
  +  
  geom_point()
```

- ① This is the line where you are adding color.

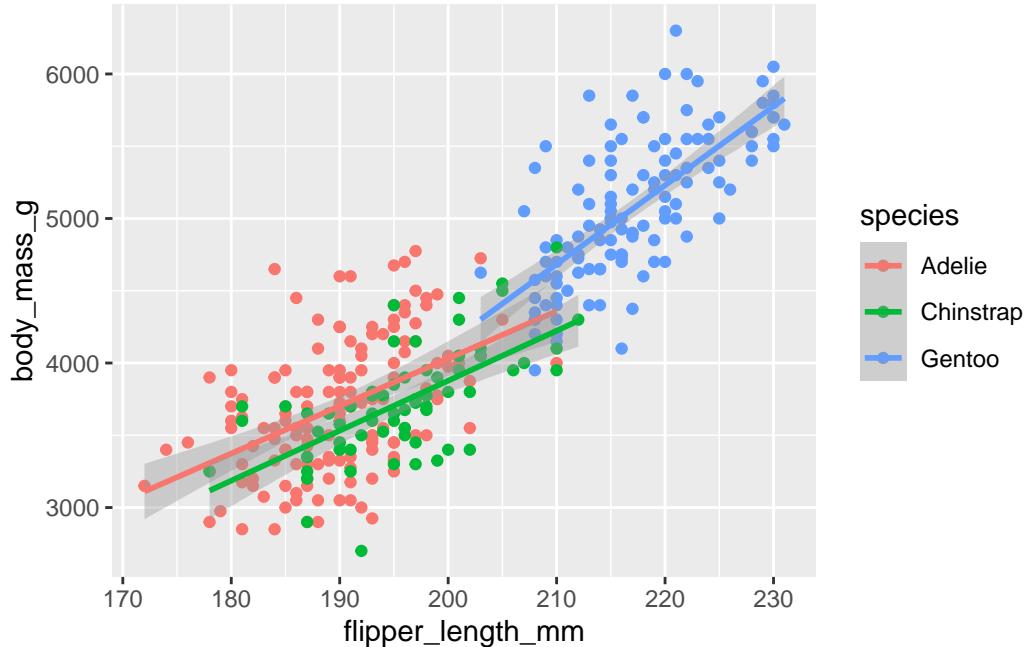


When a categorical variable like the `species` is mapped to an aesthetic, ggplot will assign a color to each unique value and add a legend so you can tell them apart.

Let's add another layer: a smooth curve displaying the relationship between body mass and flipper length. Since this is a new geometric object representing our data, we will add a new geom as a layer on top of our point geom: `geom_smooth()`. And we will specify that we want to draw the line of best fit based on a linear model with `method = "lm"`.

2. **Edit** your code to add on the last layer shown here:

```
ggplot(
  penguins,
  aes(
    x = flipper_length_mm,
    y = body_mass_g,
    color = species)
) +
  geom_point() +
  geom_smooth(method = "lm")
```



We've added the lines, but it doesn't look like our "goal" graphic we started with in Section 7.4.1, which only has one line instead of three separate lines for each of the penguin species.

When we set the `x`, `y` and `color` aesthetics in our code, we set them at the *global* level. All those characteristics apply to every added layer. However, each `geom` function can have its own aesthetics, setting them at the *local* level, applying to only that layer.

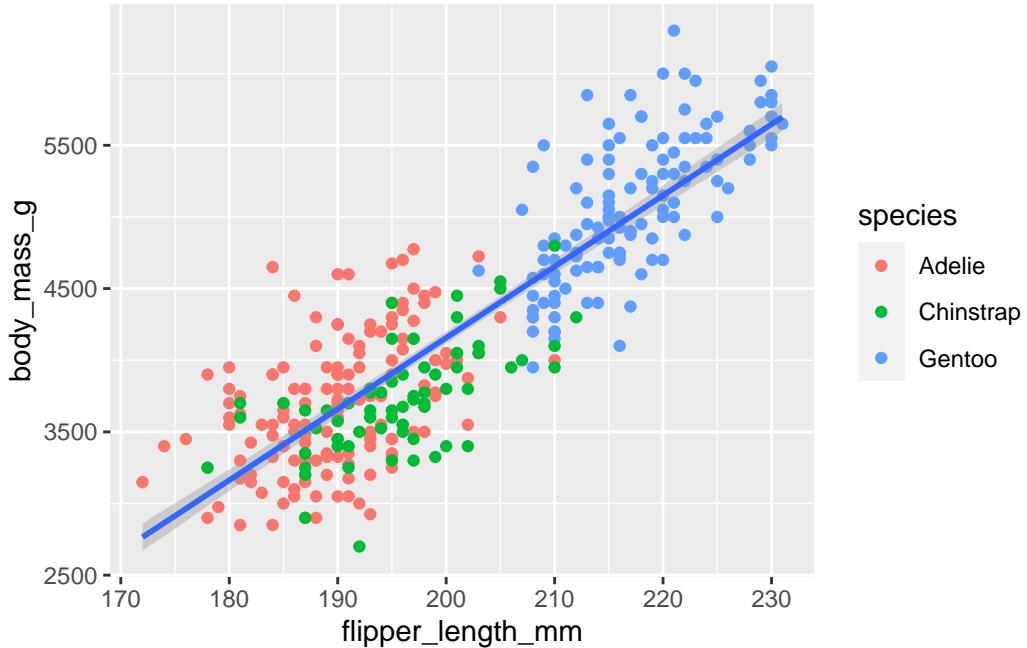
Let's show this by moving our color aesthetic from the global level to apply just at the point level.

3. Edit your code to move the `color = species` bit from the *global* aesthetics to a *local* aesthetic within `geom_point()`, as shown below.

```
ggplot(
  penguins,
  aes(x = flipper_length_mm,
      y = body_mass_g)
) +
  geom_point(aes(color = species)) + ①
  geom_smooth(method = "lm") ②
```

① This is the line where we remove the `color = species` function.

- ② This is where we add it back, creating an `aes()` function inside `geom_point()` so it only applies to that geom.

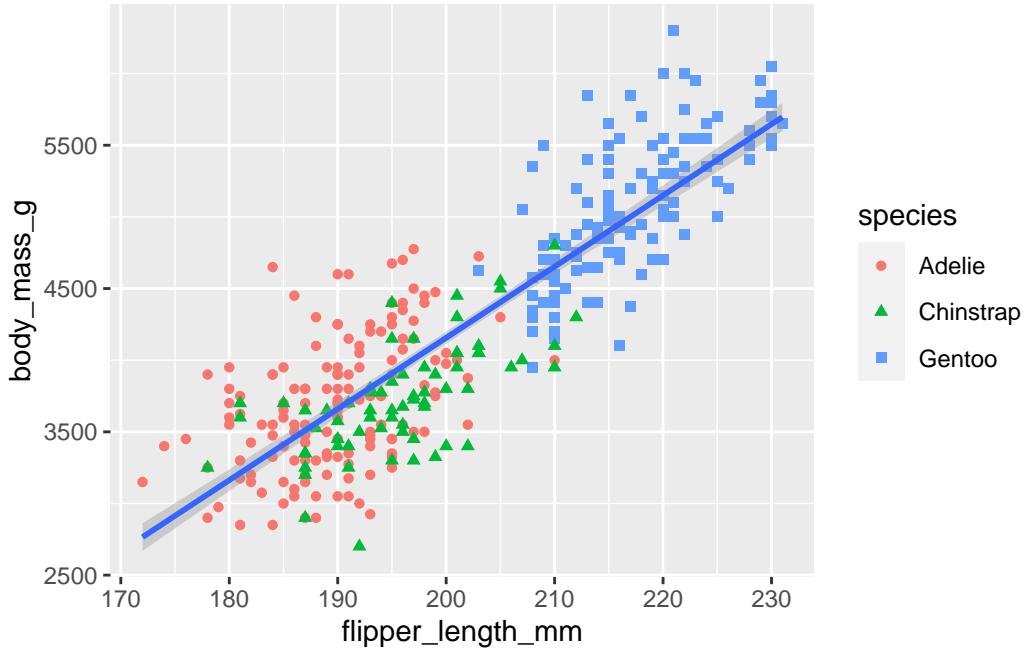


OK, this is looking pretty good, but it is not a great idea to represent information using only colors on a plot, as people perceive colors differently due to color blindness or other color vision differences. We can improve readability of this chart if we also map species based on the shape aesthetic.

4. Edit your code to add `shape = species` aesthetic to the `geom_point()` function, as shown below.

```
ggplot(
  penguins,
  aes(x = flipper_length_mm,
      y = body_mass_g)
) +
  geom_point(aes(color = species, shape = species)) + ①
  geom_smooth(method = "lm")
```

- ① We add the `shape = species` argument here inside the `aes()` function.



#### 7.4.7 Labels

In addition to geoms, you can adjust and add labels (text layers) to our plot.

**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.

Like `geom_point()` above, we'll use the `+` at the end of each line before we add another layer.

Let's add a labels layer so we can describe our chart to our readers. Note there are a number of arguments available in `labs()`, even more than those listed here.

1. Edit your code to add the `labs()` layer indicated below.

```
ggplot(
  penguins,
  aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(aes(color = species, shape = species)) +
  geom_smooth(method = "lm") +
  labs(
    title = "Body mass and flipper length",          ①
    subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo Penguins", ②
  ) ③
```

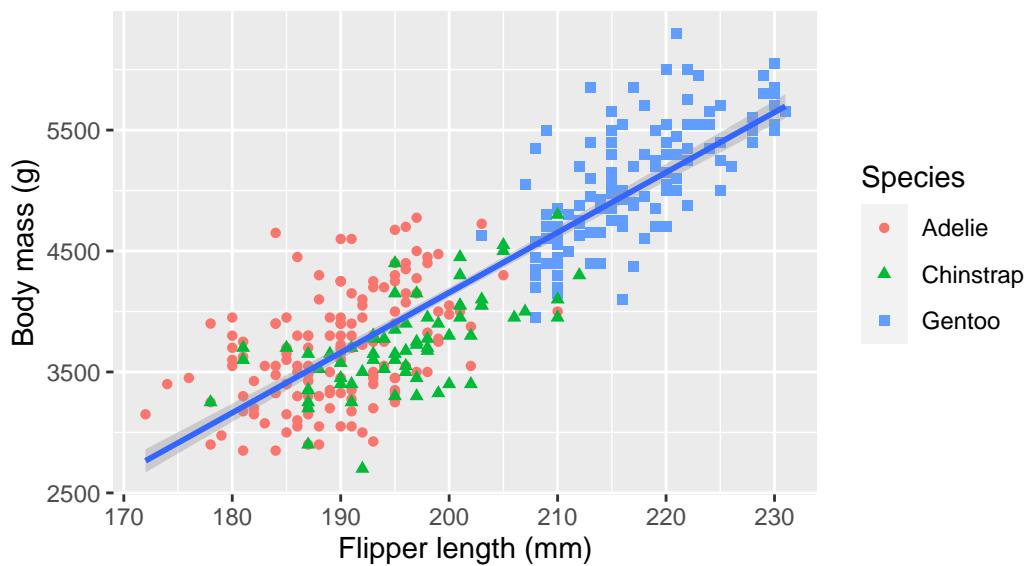
```

x = "Flipper length (mm)", y = "Body mass (g)",
color = "Species", shape = "Species"
)

```

- ① We start with the `labs()` function and open it up to write many arguments inside.
- ② `title` gives us a headline.
- ③ `subtitle` is a readout under the headline. We sometimes need to wrap the text in a `str_wrap()` function to keep it from running out of the chart.
- ④ The `x` axis already had a label on our chart by default, but it was the not-very-print-friendly name of the variable. Fine for us because we know what it is, but here we are replacing that with text more appropriate for a reader. We do the same with the `y` label.
- ⑤ The `color` and `shape` arguments are similar in that we are replacing the existing label above the legend, resetting those to capitalize “Species”. We need to change them both so they are the same. (You might experiment by removing the `shape` argument and re-running it to see what happens.)

**Body mass and flipper length**  
Dimensions for Adelie, Chinstrap, and Gentoo Penguins



#### 7.4.8 An aside: Titles and subtitles

Our example chart here is a scientific figure designed to show the relationship between the length of flippers to the body size of penguins, and that perhaps the relationship holds true among different species. It is not a good example of a journalistic chart.

We want our chart titles to be *MORE THAN A LABEL* ... they should communicate something and further the story we are trying to tell with the plot. They should have a verb! If we were telling a story here, we might try: **Flippers longer on heavier penguins.**

Our subtitles should provide the context necessary to understand what we are looking at and why. The combination of the title and subtitle should stand alone to tell enough of this story that we don't have to go elsewhere (like story) to understand it. A journalistic model might be something like this: **Dr. Kristen Gorman studied three species of penguins at the Palmer Long-Term Ecological Research station in Antarctica between 2007 and 2009, recording basic biographical measurements nearly 350 animals.**

#### 7.4.9 Themes

You can change just about *anything* on a ggplot chart if you know the function and arguments to describe it. The [cheatsheet](#) shows a lot of them, but it can get really dense.

**Themes** are collections of these visual changes saved into a single function. There are several available within ggplot and [many others from the R community](#).

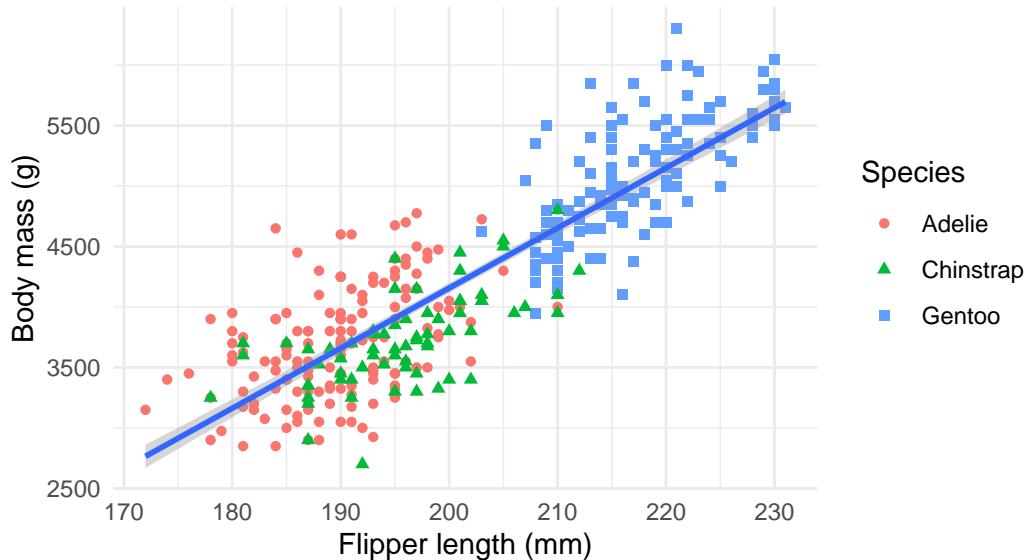
Let's show that here.

1. **Edit** your code chunk to add on the last + `theme_minimal()` function.

```
ggplot(  
  penguins,  
  aes(x = flipper_length_mm, y = body_mass_g)  
) +  
  geom_point(aes(color = species, shape = species)) +  
  geom_smooth(method = "lm") +  
  labs(  
    title = "Body mass and flipper length",  
    subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo Penguins",  
    x = "Flipper length (mm)", y = "Body mass (g)",  
    color = "Species", shape = "Species"  
) +  
  theme_minimal()
```

## Body mass and flipper length

### Dimensions for Adelie, Chinstrap, and Gentoo Penguins



This changes the background color and grid lines to make the data pop a little better. There are other themes that create more radical change, some in their own packages. We'll play more later.

OK, you've made your first ggplot chart! Are you ready to make another?

## 7.5 Let's build a bar chart

In our first week of class, we sent out a survey where you told us your favorite Disney Princess and favorite flavor of ice cream. Let's now play around with some of this data.

For this lesson, we're not going to create a different notebook or download the data to our computer. Instead, we're going to save the data directly into a tibble.

1. Start a new section: Importing class data
2. In the text, note that we are importing the chart data.
3. Add the code below to get the data.

```
class <- read_csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vQfwR6DBW5Qv605aEBFJ14V8i  
  clean_names()  
  
class
```

- ① We create a new object and then fill it by reading the data directly from the web.
- ② We lowercase the variable names using `clean_names()` from the janitor package.
- ③ Print the new object to peek at it.

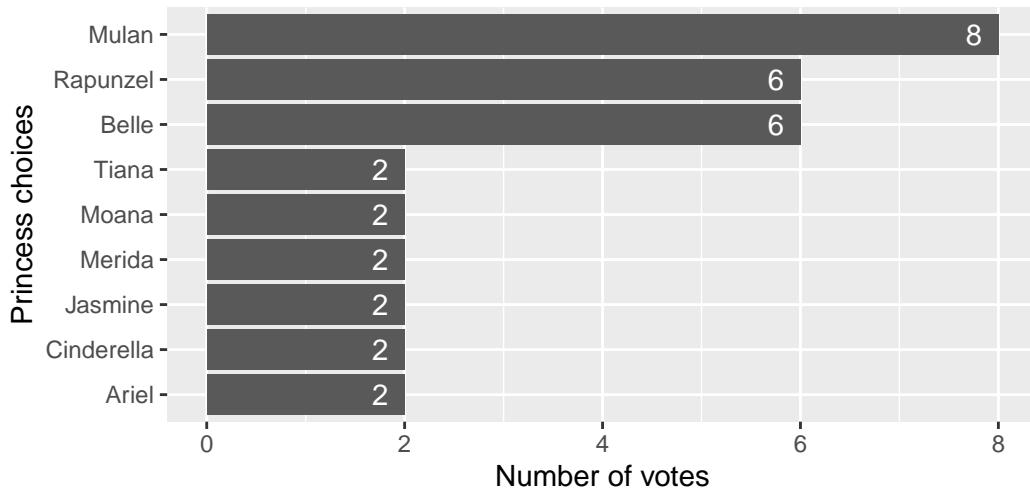
```
# A tibble: 32 x 3
  name      princess ice_cream
  <chr>    <chr>     <chr>
1 Alex      Belle     Strawberry
2 Amberlyn  Belle     Cookie Dough
3 Arianna   Jasmine  Mint Chocolate Chip
4 Bo        Rapunzel Mint Chocolate Chip
5 Campbell  Mulan    Chocolate
6 Dacia     Rapunzel Cookies and Cream
7 Ellie     Merida   Cookie Dough
8 Emma      Rapunzel Cookies and Cream
9 Evan      Mulan    Strawberry
10 Hannel   Mulan   Chocolate
# i 22 more rows
```

So, now, you should have the data in your environment.

And with this data, we want to build a chart like this:

### Pick of the princesses

Students in Reporting with Data each voted for their favorite Disney Princess.  
Some complained that Princess Leia was not an option.



### 7.5.1 Prep princess data

While there are ways for ggplot to calculate values from your data on the fly, I prefer to first build a table of the values I want, and then I will plot it on a chart. It's helpful to think of these steps as separate so you have a good workflow (clean the data, prepare the data in a table form, and then plot the data).

Today, our goal will be to make a bar chart, sometimes known as a column chart. This bar chart will show 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` (GSA) process.

For this lesson, I'll use the `count()` shortcut, since we haven't used it much. Next, I'll save the summarized data into a new dataframe called `princess_data`. Follow along in your notebook:

1. Add a section: Princess data
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)  
  
princess_data
```

- ① Start with a new object and start filling it with class data.  
② Here we use the `count()` function to count the number of rows based on the `princess` column. The argument `name =` just names the new columns something other than `n`, and the `sort =` argument sorts the data in descending order based on the counted column.  
③ Print it out so we can see it.

```
# A tibble: 9 x 2  
  princess   votes  
  <chr>     <int>  
1 Mulan        8  
2 Belle         6  
3 Rapunzel     6  
4 Ariel         2  
5 Cinderella    2  
6 Jasmine       2  
7 Merida        2  
8 Moana         2  
9 Tiana         2
```

At this point, y'all should be plenty familiar with these summary functions, and the output should be easy to interpret: we're just counting the number of rows for each princess.

Now that we have our table data, let's actually plot it.

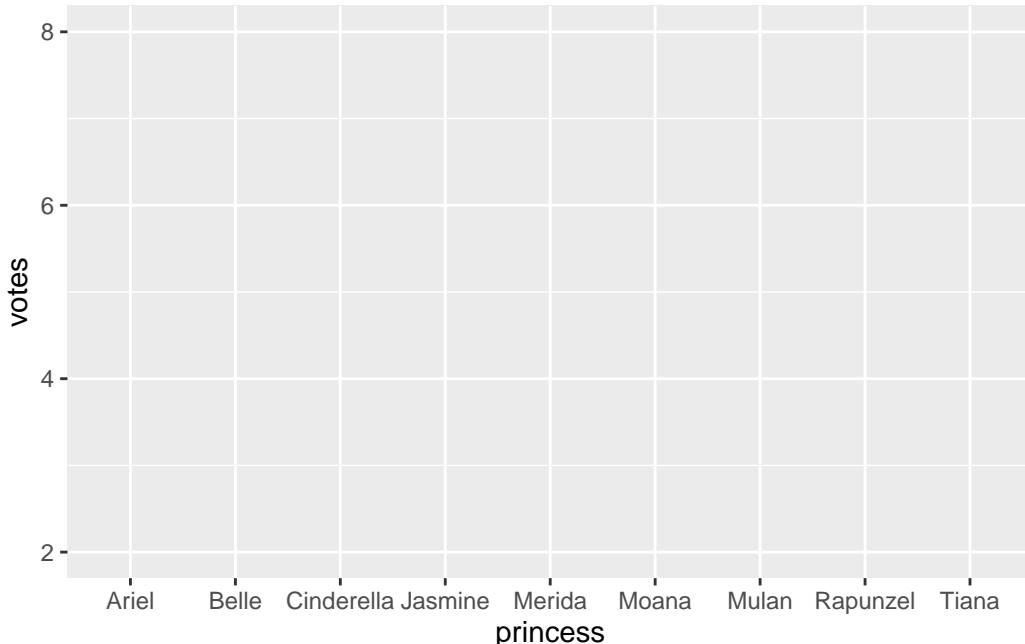
### 7.5.2 Build our plot with geom\_col

Like in the previous lesson, we'll start our plot by creating the first layer: the `ggplot()` function, which takes the data as its first argument and the `aes()` mapping layer as its second argument.

1. Add some text noting that you'll now plot.
2. Add the following code chunk, which is the first layer

```
ggplot(  
  princess_data,  
  aes(x = princess, y = votes))  
①
```

① sets our x and y axes values.



You'll see the grid and x/y axis of the data, but no geometries are applied yet, so you won't see any data. But remember, we're adding these all in layers.

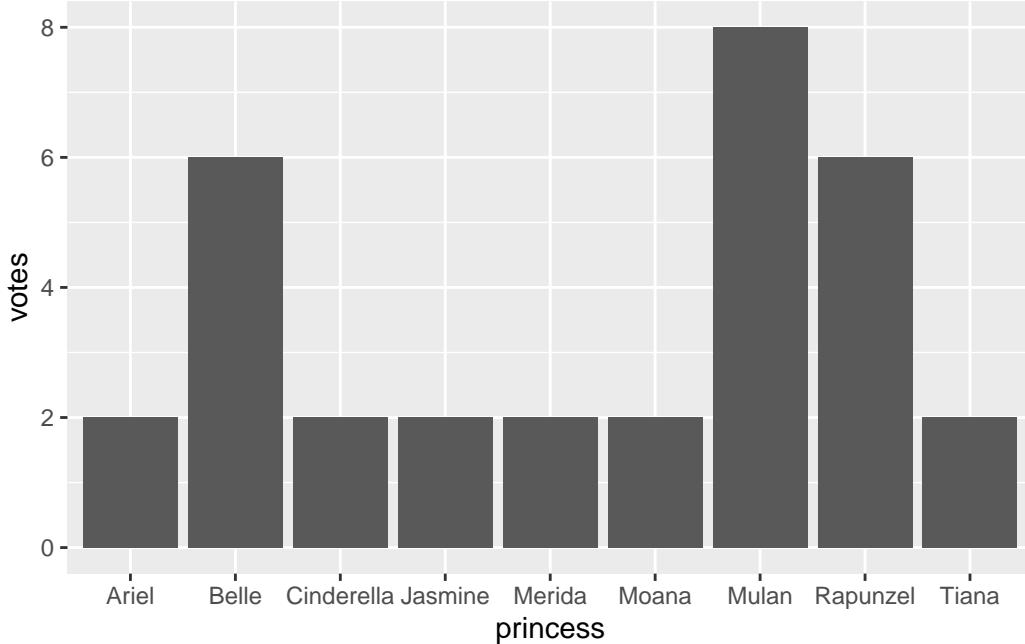
### 7.5.3 Add the geom\_col layer

Now it is time to add our columns. To do this, we'll use `geom_col()`. Similar to `geom_point()`, `geom_col()` adds a geometric layer that tells R how to display the data (in this case, with columns as opposed to points). Let's write this code now.

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)  
) +  
  geom_col()  
①  
②
```

- ① Don't forget to add the `+` at the end of the previous line
- ② The `geom_col()` function adds the bars based on the global aesthetics we've already set.



Our two-layer chart is getting somewhere now. We're able to see the data in the plot, but there are a couple issues:

- Depending on the width of our notebook, the princess names might collide with each other. 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 axes

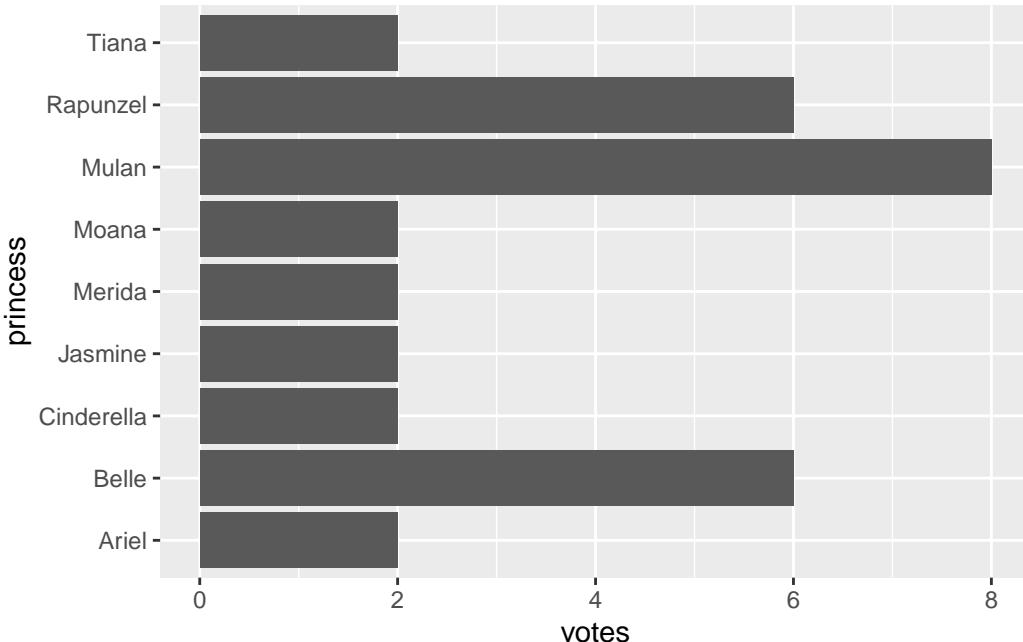
One way to fix the labels is to “flip” the axes, so the x axis becomes the y axis and vice versa. This is the equivalent of rotating the whole figure. When we do this, the axis will turn sideways, making it easier to read the labels. Worth noting: this can be a bit confusing later because the “x” axis is now going up/down (as opposed to left and right).

Let’s learn how to flip the axes now. We’ll do this by adding a new layer, `coord_flip()`, which is a special layer that flips the axes. Just like we added the previous `geom_col()` layer using `+`, we’ll do the same thing here. Let’s do that now.

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() +  
  coord_flip()  
①
```

- ① Adding `coord_flip()` swaps the x and y axes.



As you can see, rather than having vertical bars, we now have horizontal bars, and the names of each princess are fully displayed and read-able. Much better!

But the bars are still in an alphabetical order, as opposed to a vote order, so let's fix that now.

### 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.

#### i Note

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 don’t have that here, but know it is a thing.

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 for factors.)

`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.

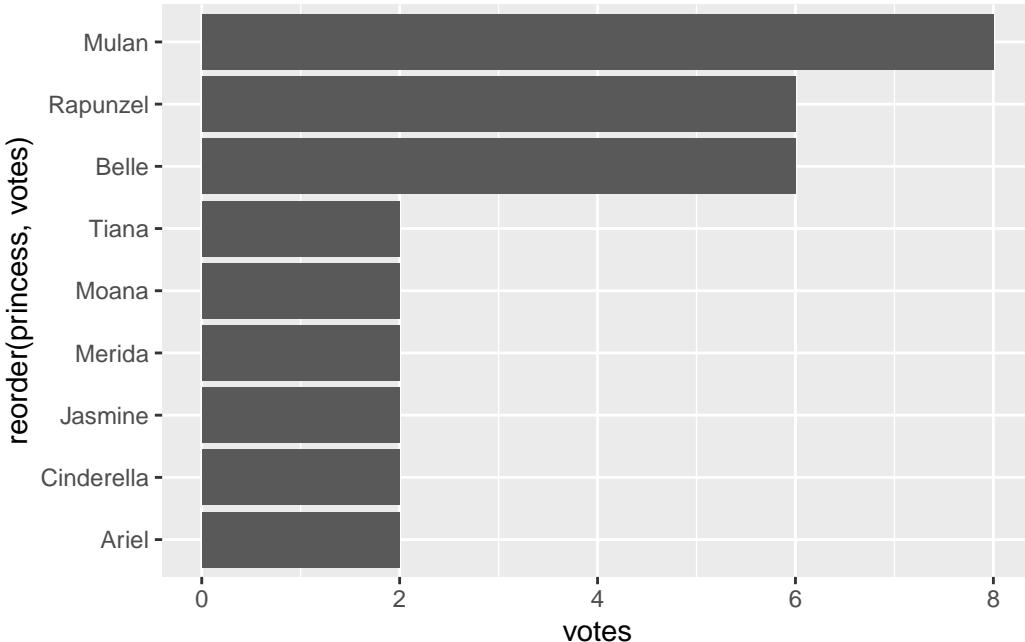
- `x = reorder(princess, votes)` says “we shall set `x` as reordered values of `princess` based on the order of `votes`. OR ...
- `x = princess |> reorder(votes)` says “set the `x` axis as `princess` and then reorder by `votes`.

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

1. Edit the first line of your chunk to reorder the bars.

```
ggplot(  
  princess_data,  
  aes(x = reorder(princess, votes), y = votes)) +  
  geom_col() +  
  coord_flip()
```

① This is the line where `reorder()` is added.



So now, our princess names are read-able, and the bars are organized in vote size. But what if we wanted to be clearer in our figure, so that we knew the exact number of votes for each princess? Let's learn how to add this information.

### 7.5.6 Adding a `geom_text` layer

Now, we're really starting to take advantage of the grammar of graphics by including more than one geometric layer. Specifically, we'll be using `geom_text()` to add some information to our bar charts.

As we mentioned previously, `geom` layers can take individual aesthetics (that build on top of the global aesthetics you put in the first layer). When using `geom_text()`, we'll include some local aesthetics using the `aes()` argument, to tell `ggplot` the label we'd like to add to the plot.

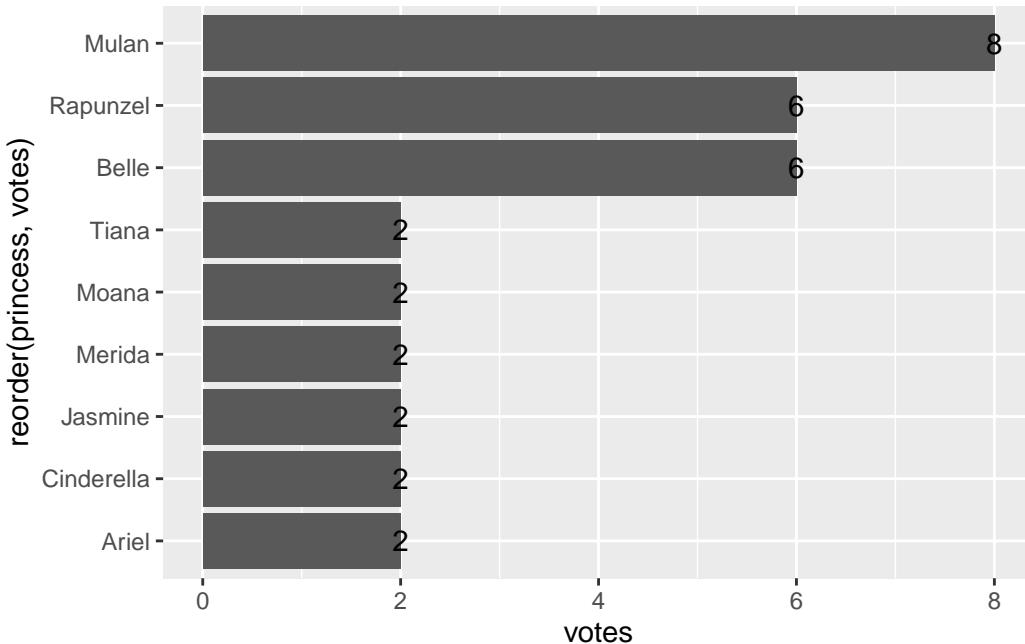
1. **Edit** your plot chunk to add the `+` and `geom_text()` layer on the end of your code.
2. Set the aesthetics of the `geom_text()` function to plot the labels onto the chart based on the number of votes: `aes(label = votes)`, as noted below.

```
ggplot(princess_data,
       aes(x = reorder(princess, votes), y = votes)
) +
  geom_col() +
```

```
coord_flip() +  
geom_text(aes(label = votes))
```

①

- ① This plots a text layer onto the chart based on both the position and values in `votes`.



Well that did... something. We've successfully added the numbers to this plot, but it's not very pretty. First, the number is plotted at the end of the bar, making it harder to read. So we'll want to **horizontally** adjust this by shifting the numbers a bit to the left. Second, black text is really hard to read against a dark grey background. So we'll change the text of the number to white.

We can make both of these edits directly in the `geom_text` layer.

1. **Edit** the last line of your plot chunk to add two new arguments.
2. The first argument you will add is `hjust`, which moves the text left. (`hjust` stands for horizontal justification. `vjust`, or vertical justification, would move it up and down).
3. The second argument you will add is `color`, which tells `ggplot` what the color of your text should be.

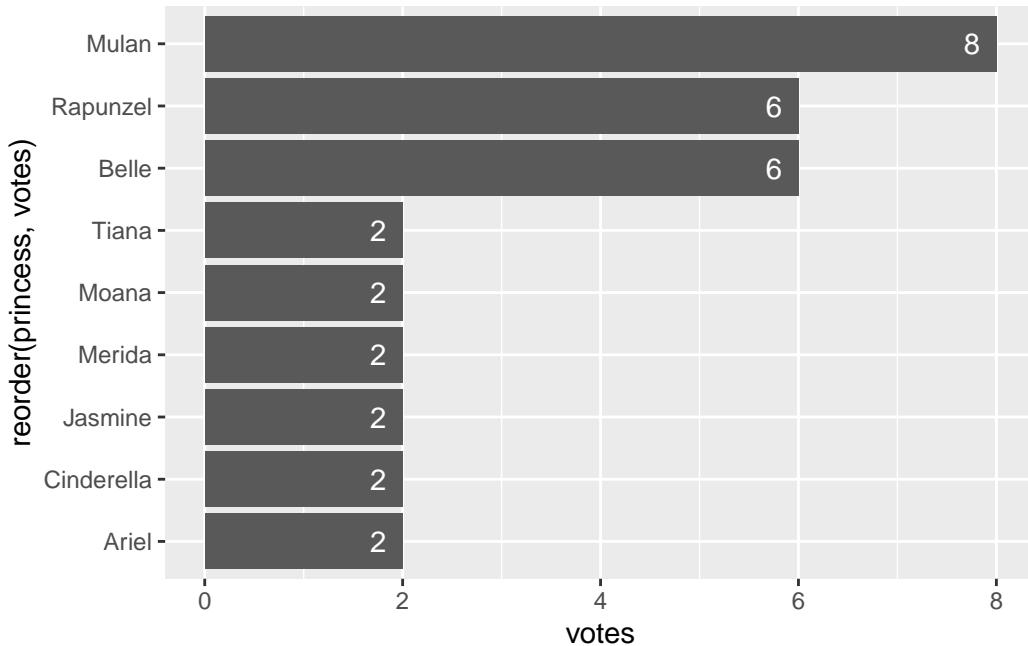
As a reminder, you should always separate your arguments within a function using commas (,).

```

ggplot(
  princess_data,
  aes(x = reorder(princess, votes), y = votes)
) +
  geom_col() +
  coord_flip() +
  geom_text(aes(label = votes), hjust = 2, color = "white") ①

```

- ① Note that both `hjust` and `color` are NOT inside the `aes()` function because we are not using the data to control them.



Great! But we're still not done. Even though we've added labels to each bar chart, we still haven't added a title, and the titles of our x and y axes are not great. So let's work on those now.

### 7.5.7 Add some titles and more labels

Now that we have a chart, with some information displayed in bars, flipped and arranged so we can see information, let's add to this by giving the chart some labels. We'll do this by adding a `layer` of labels to our chart using the `labs()` function. We can add and change a number of things with `labs()`, including creating a title, and changing the x and y axis titles.

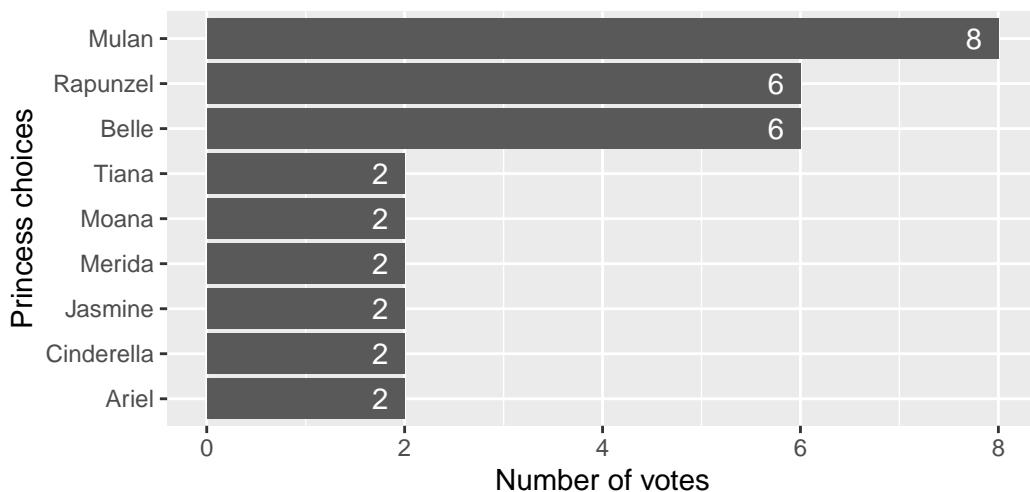
1. **Edit** the last line of your plot chunk to add the `ggplot` pipe `+` and `labs()` in the next line.
2. Add a title using the `title = argument`
3. Add a subtitle using the `subtitle = argument`. This is a great place to put information about your data (like when it was collected).
4. Add a caption using the `caption = argument`. Put your byline here!
5. Change the x and y axes titles using `x =` and `y =`.

```
ggplot(
  princess_data,
  aes(x = reorder(princess, votes), y = votes)
) +
  geom_col() +
  coord_flip() +
  geom_text(aes(label = votes), hjust = 2, color = "white") +
  labs(
    title = "Pick of the princesses",          ①
    subtitle = str_wrap("Students in Reporting with Data each voted for their favorite Disney princess"),
    caption = "By Jo Lukito",                  ③
    x = "Princess choices",                  ④
    y = "Number of votes"                   ⑤
  )
```

- ① A title to draw you in or communicate your goal.
- ② The subtitle is the place to explain what is needed to understand the chart. In this case we wrap the text in a `str_wrap()` function so the words don't run off the chart.
- ③ A `caption` is a good place to add your byline.
- ④ Because we use `coord_flip()` it is actually the `x` axis that is vertical.
- ⑤ Again, x and y are flipped.

## Pick of the princesses

Students in Reporting with Data each voted for their favorite Disney Princess. Some complained that Princess Leia was not an option.



By Jo Lukito

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` (which is the name of a variable in your `class` data frame)
- You need to build the chart, paying attention to axis

It's essentially the same process we used for the princess chart, but using `ice_cream` variable. That said, I really recommend you write all the code from scratch, ONE LINE AT A TIME, so you can soak in what each line does.

## 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.
- We also covered `reorder()`, which can reorder a variable based on the values in a different variable.

Here are some more references for ggplot:

- The [ggplot2 documentation](#) and [ggplot2 cheatsheets](#).
- [R for Data Science, Chap 3](#). Hadley Wickham dives right into plots in his book.
- [ggplot2: Elegant graphics for Data Analysis](#) by Wickham.
- [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.

# 8 Deeper into ggplot

The chapter is by Prof. Lukito, with edits from Prof. McDonald.

In the last chapter, you were introduced to [ggplot2](#), the `tidyverse` package that helps you build graphics, charts, and figures. In this chapter, we'll take your `ggplot` knowledge to the next level. We encourage you to treat this chapter as a reference.

## 8.1 Learning goals for this chapter

In this chapter, we will cover the following topics:

- How to prepare and build a line chart
- How to use themes to change the looks of a chart
- More about aesthetics in layers!
- Faceting, or making multiple charts from the same data
- How to make interactive plots with `Plotly`

## 8.2 References

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

- [The ggplot2 documentation](#) and [ggplot2 cheatsheets](#).
- [R for Data Science, Chap 3](#). Hadley Wickham dives right into plots in his book.
- [ggplot2: Elegant graphics for Data Analysis](#) by Wickham.
- [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.

## 8.3 Set up your notebook

This week, we'll return to our Military Surplus project, but add a new Quarto Document.

1. Open your `name-military-surplus` project.
2. Create a new Quarto Document.
3. Set the title as "Military Surplus figures"
4. Remove the rest of the boilerplate template.
5. Save the file and name it `03-ggplot.qmd`.
6. Create a setup chunk and load the `tidyverse` library. Also load a library called `scales`, though you shouldn't have to install it as it comes with the `tidyverse` package.

You'll also need to install the `scales` package and include that in your library.

7. In your **Console**, run `install.packages("scales")`
8. Then back in your notebook add `library(scales)` to your setup chunk.

### 8.3.1 Let's get the data

We'll be importing the cleaned data that has all the states.

1. Create a new section that notes you are importing
2. Use `read_rds()` to import the file and save it into an object called `leso`. The file should be at `data-processed/01-leso-all.rds` if you named it correctly in our [analysis](#) .
3. Glimpse the data so we can see the column names

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

```
Rows: 67,852
Columns: 13
$ state           <chr> "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", "2320-01-371~
$ item_name       <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~
$ quantity        <dbl> 10, 9, 1, 10, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ui              <chr> "Each", "Each", "Each", "Kit", "Each", "Each", "Each~
$ acquisition_value <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~
$ demil_code      <chr> "D", "D", "C", "D", "C", "Q", "D", "C", "D", "D~
$ demil_ic        <dbl> 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ship_date        <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~
```

```
$ station_type      <chr> "State", "State", "State", "State", "State", "State"~  
$ total_value       <dbl> 16260.00, 3339.00, 62627.00, 172647.10, 62627.00, 65~  
$ control_type      <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE~
```

Above we can see the columns were working with.

## 8.4 Goal 1: Line chart

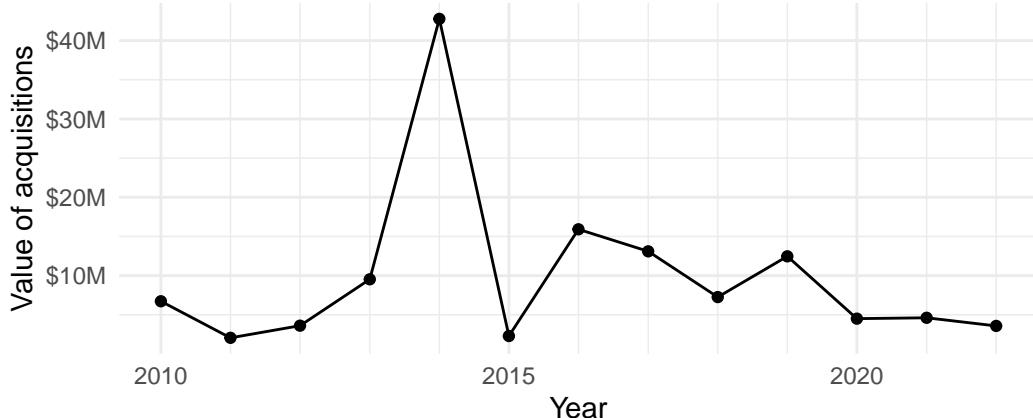
For our first chart here we are looking to plot the total acquisitions value for each year in the state of Texas. It can sometimes help to draw on paper the chart you are hoping to make so you can see how to format the data to make it.

In our case, we'll need to summarize our data by the year of the shipping date, then add together all the value of the equipment. Here is how we'll want to structure our data and the chart we'll make from it.

```
# A tibble: 13 x 2  
  year           yearly_value  
  <dttm>            <dbl>  
1 2010-01-01 00:00:00    6727288.  
2 2011-01-01 00:00:00    2059543.  
3 2012-01-01 00:00:00    3610790.  
4 2013-01-01 00:00:00    9531431.  
5 2014-01-01 00:00:00   42774914.  
6 2015-01-01 00:00:00    2292580.  
7 2016-01-01 00:00:00   15912672.  
8 2017-01-01 00:00:00   13099487.  
9 2018-01-01 00:00:00    7262359.  
10 2019-01-01 00:00:00   12462964.  
11 2020-01-01 00:00:00    4508671.  
12 2021-01-01 00:00:00   4619450.  
13 2022-01-01 00:00:00   3578707.
```

## Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal.



Source: Law Enforcement Support Office

But we don't have this data in the format, so we need to make it.

### 8.4.1 Working through logic

Data visualization is an iterative process: you may prepare the data, do the visualization, and then realize you want to prepare the data a different way. Remember that the process of coding can involve trial and error: you're often testing different things to see what works.

There are lots of choices we could make on *how* to do this, and I explored several different paths while making this chart. While I won't go through every possible scenario, I'll try to explain why I made the decisions I did. If I was doing this on my own, I might record these explorations and thoughts in my notebook so I could have a record of why I did things. It's not like other ways are wrong, they just have different pros and cons.

To kinda walk through this with the chart first:

- Along the X axis we want to plot each year. Since our data starts as individual transfers and their date, this is a clue that we need to group by the year of that shipping date.
- Along the Y axis we want to plot the value of all the equipment in Texas. That's a clue that we need to summarize our data to add up the value of the equipment.

The most exploration came around how I grouped this data by year. There are a myriad of ways to do this, like using `year()` to pluck the year from the shipping date to make a number like 2023 or to then convert that to text as "2023". I chose to create a "floor date" so I end

up with a real date because it gives me more flexibility with plotting. ggplot has some tools where I can format the date the way I want, though I admit I had to go learn about them to do this lesson!

The function `floor_date()` rounds a date *down* to a common value, but it remains a real date. If I convert both “2023-10-18” and “2023-10-28” to a floor\_date unit of “year” then they both end up as “2023-01-01”, the first day of the year. If I choose to set floor\_date as a unit of “month” then both dates end up as “2023-10-01”, the first day of the month in each year. It’s like we are rounding the date down to a specific unit. The key here is it remains a real date with properties like year, month, week and day. I’ll draw upon those properties later.

We are going to “create” this date as we group our data. You can read more about this in Appendices chapter [Grouping by dates](#).

### 8.4.2 Wrangling the data

Let’s filter to the data we want (Texas data before 2023) and group it by the year of the `ship_date` and summarize to sum the `total_value`.

1. Add a new section: Prepare Texas total values data
2. Add a new chunk and the code below. We’ll pick it apart after.

```
tx_values <- leso |>
  filter(
    state == "TX",
    ship_date < "2023-01-01"
  ) |>
  group_by(year = floor_date(ship_date, unit = "year")) |>
  summarize(early_value = sum(total_value))
```

```
tx_values
```

```
# A tibble: 13 x 2
  year                  early_value
  <dttm>                <dbl>
1 2010-01-01 00:00:00     6727288.
2 2011-01-01 00:00:00     2059543.
3 2012-01-01 00:00:00     3610790.
4 2013-01-01 00:00:00     9531431.
5 2014-01-01 00:00:00     42774914.
6 2015-01-01 00:00:00     2292580.
7 2016-01-01 00:00:00     15912672.
8 2017-01-01 00:00:00     13099487.
```

```

9 2018-01-01 00:00:00      7262359.
10 2019-01-01 00:00:00     12462964.
11 2020-01-01 00:00:00     4508671.
12 2021-01-01 00:00:00     4619450.
13 2022-01-01 00:00:00     3578707.

```

Let's walk through the code:

- We start by creating a new object to shove all this stuff into: `tx_values`. Then we start with the data ...
- You should be familiar enough with the filtering by now to understand what we are doing, but might not realize *why*. Filtering for just Texas rows makes sense because that is our focus, but you might not snap that the data for the most recent year is incomplete until you see the plot. It's important to recognize the date range of the data you are plotting so you don't compare a partial year to a complete one.
- Inside this `group_by()` we have the code that makes our floor date.
  - We start as we often do by naming the new column made by the group: `year`.
  - We then use the `floor_date()` function on the `ship_date` variable, setting the unit to “year”.
  - You can see the result of this in the return: Any row with a date within the year “2010” was given a date of “2010-01-01” and then summed together.
- The last line is the `summarize()` function where we add all the `total_values` together. We first named the column `yearly_value` since that is what we are creating.
- At the end, we print out the contents of the `tx_values` object we created.

### 8.4.3 Plot the chart

Alright, so now let's get ready to use the `ggplot()` function. I want you to create the plot here one step at a time so you can review how the layers are added.

In this new plot, we'll learn about a new `geom` layer: `geom_line()` (recall that in [the last chapter](#), we learned about `geom_point()` and `geom_col()`).

1. Start a new section. Note we are plotting Texas values.
2. Add and run the `ggplot()` line first (but without the `+`)
3. Then add the `+` and the `geom_point()` and run it.
4. Then add the `+` and `geom_line()` and run it.
5. Then add the `+` and `labs()`, and run everything.

```

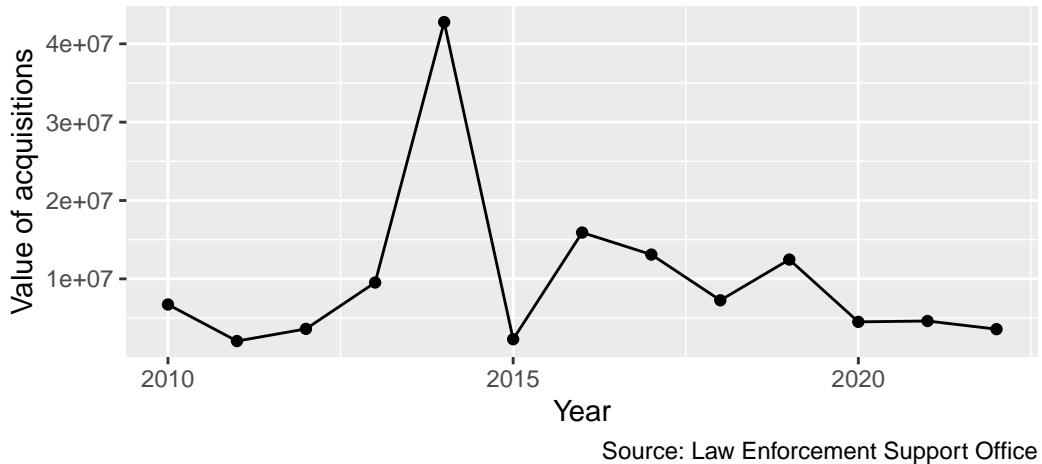
ggplot(tx_values, aes(x = year, y = yearly_value)) +
  geom_point() +
  geom_line() +
  labs(
    title = "Military surplus acquisitions in decline",
    subtitle = str_wrap("After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal."),
    x = "Year", y = "Value of acquisitions",
    caption = "Source: Law Enforcement Support Office"
  )

```

- ① we create the graph
- ② adding the points
- ③ adding the lines between the points
- ④ we add the labels. Note the `str_wrap()` function we've used with `subtitle =` which keeps the text from running outside the chart.

### Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal.



#### 8.4.4 Cleaning Up

Alright, so we have a working plot! But there's a couple things that are a bit ugly about this plot. First, I'm not digging the weird numbers on the side (what the heck does  $1e+07$  even mean?!). If we go back up and look at the output from `tx_values` we can see the `yearly_value` numbers are pretty large. These large numbers are causing R to read our numbers as "scientific

notation” (a math-y way of reading large numbers). For example, the total cost of supplies in 2014 was 42,949,729 (that’s the first spike in our figure, around the 4e + 07 mark). But what a pain to read!

There are a number of ways we could fix this. We could go back into the code block where we prepare the data and divide that value by “1000000” and call the values “per million”. Or, we can do what I did and use the `scales` package to “scale” the output for presentation without changing the underlying number.

Using functions from the `scales` package can be esoteric and unintuitive, but most of the things you would regularly need are covered in the [Position and Scales chapter of the ggplot2 book](#). It is from there that we learn about using a function `scale_y_continuous()`, the `label =` argument and the the `scale_dollar()` function.

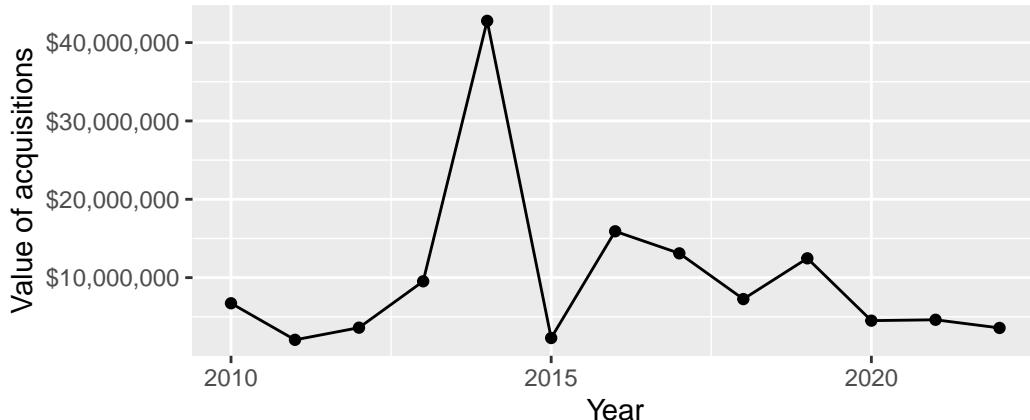
1. After the `geom_line()` line, add the `scale_y_continuous` function as indicated below.

```
ggplot(tx_values, aes(x = year, y = yearly_value)) +
  geom_point() +
  geom_line() +
  scale_y_continuous(labels = label_dollar()) +          ①
  labs(
    title = "Military surplus acquisitions in decline",
    subtitle = str_wrap("After a spike in 2014, the value of controlled military surplus equ",
    x = "Year", y = "Value of acquisitions",
    caption = "Source: Law Enforcement Support Office"
  )
```

① This is the added line for the scale

## Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies in recent years. Controlled equipment includes items like weapons returned to the military for disposal.



Source: Law Enforcement Support Office

I added this in the middle of the code because I like to keep the `labs()` at the bottom of the chart. It doesn't really matter. What it did was change the labels on the Y axis to use a currency format. Let's walk through it:

- The `scale_y_continuous()` function allows modification to the Y axis, and there is a companion for the X axis. The term “continuous” refers to the values being numbers. There are similar functions for categories (`scale_*_discrete()`) and dates (`scale_*_date()`). They all allow modification of things like the labels, grid lines and axis names. (The \* there is placeholder for x or y.)
- We use that function to set our “labels” with another function from the scales package, `label_dollar()`.

Again, the best resource for these type of things is the [Scales](#) chapters in the ggplot2 book.

To show just how much granularity you have with ggplot, let's make one more change to this chart. Those dollar values are still pretty large, and it might be nice to show them in millions of dollars, like "\$10M".

2. Edit the `label_dollar()` function to add this argument inside: `scale_cut = cut_long_scale()`.

```
ggplot(tx_values, aes(x = year, y = yearly_value)) +  
  geom_point() +  
  geom_line() +
```

```

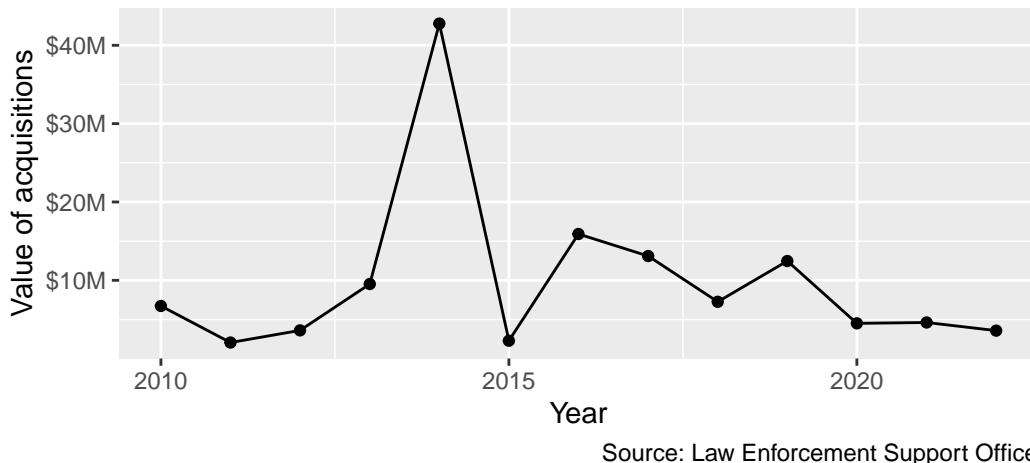
scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) + ①
labs(
  title = "Military surplus acquisitions in decline",
  subtitle = str_wrap("After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal."),
  x = "Year", y = "Value of acquisitions",
  caption = "Source: Law Enforcement Support Office"
)

```

- ① We're editing this line to add the `scale_cut` argument to `label_dollar()`.

### Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal.



Let's show one more change. Notice how the grid lines along the X axis don't fall along every year? We are getting our "breaks" set every five years, but our "minor breaks" (the fainter lines between the five year increments) sit in the middle of a year. That makes it a little harder to read each year, or even to understand each of those dots fall on a year. We can use the `scale_x_date()` function to change those by adjusting those minor breaks.

3. In the global `aes()` function, wrap the `date` value inside a function called `date()`. Currently our "date" is really a calendar date/time datatype called `<POSIXct>`. I've found our next function can be pretty picky about wanting a specific `<date>` datatype so we are adjusting it to avoid an error.
4. After the `scale_y` layer, add a new `scale_x_date()` later as indicated below.

```

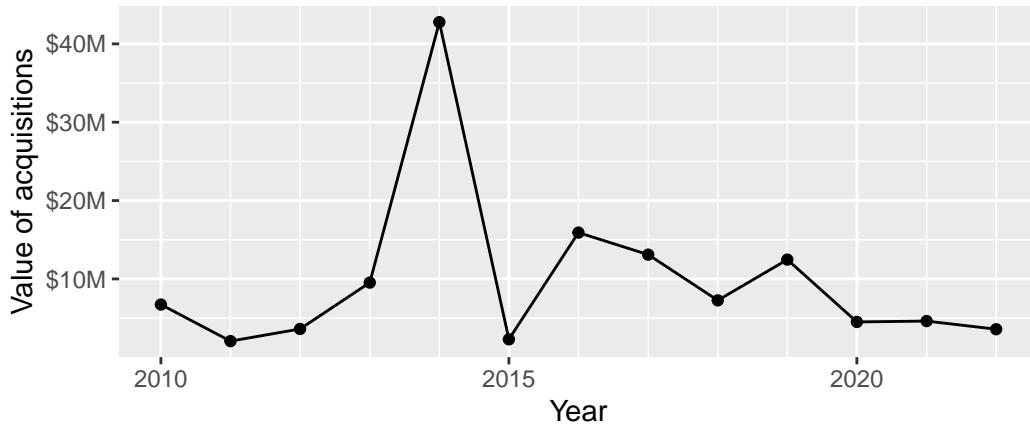
tx_values |>
  ggplot(aes(x = date(year), y = yearly_value)) + ①
  geom_point() +
  geom_line() +
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) + ②
  labs(
    title = "Military surplus acquisitions in decline",
    subtitle = str_wrap("After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal."),
    x = "Year", y = "Value of acquisitions",
    caption = "Source: Law Enforcement Support Office"
  )

```

- ① Surround “year” with `date()` function.
- ② This is the added later for `scale_x_date()`.

### Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal.



OK, now the fainter “minor breaks” lines fall on each year, making them a little easier for the eye to track. Again, I had to [do some digging](#) to figure that out.

#### 8.4.5 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 to this object, but you won’t need to rebuild the main

portions of the chart each time. We'll do that here so we can explore themes next.

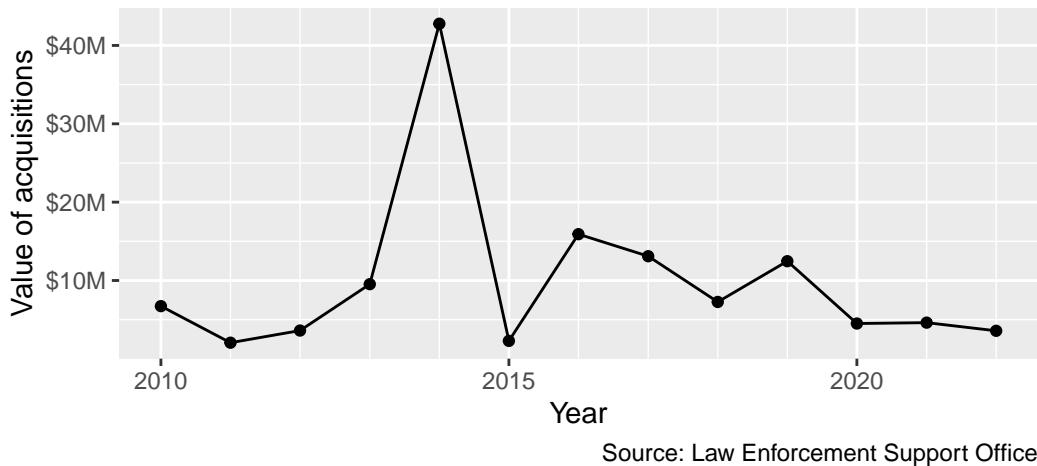
1. Edit your Texas plot chunk to save it into an R object called `tx_plot`.
2. Then call it after the code so you can see it.

```
tx_plot <- ggplot(tx_values, aes(x = date(year), y = yearly_value)) +      ①
  geom_point() +
  geom_line() +
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(
    title = "Military surplus acquisitions in decline",
    subtitle = str_wrap("After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal."),
    x = "Year", y = "Value of acquisitions",
    caption = "Source: Law Enforcement Support Office"
  )
  ②
tx_plot
```

- ① We edit this line to add the object at the beginning.  
② 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.

### Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal.



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

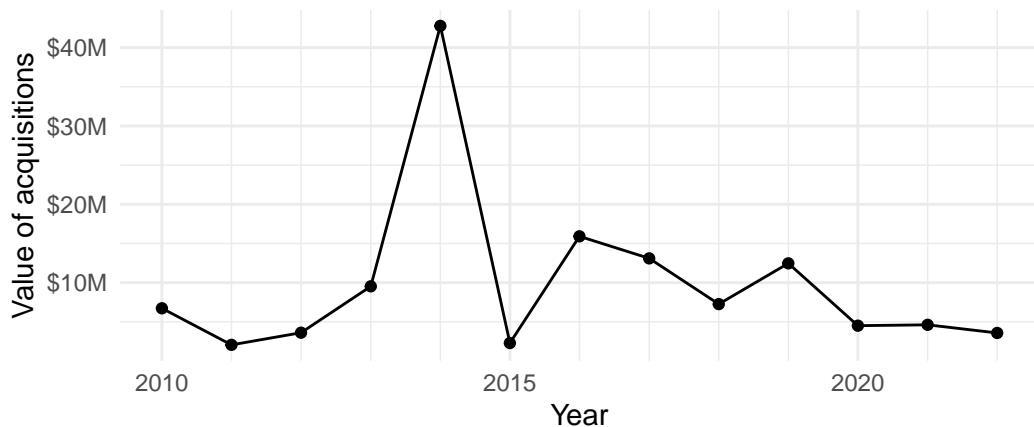
The *look* of the graph is controlled by the theme. There are a number of preset themes you can use. Let's look at a couple.

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

```
tx_plot +  
  theme_minimal()
```

### Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has declined in recent years. Controlled equipment includes items like weapons that have been returned to the military for disposal.



Source: Law Enforcement Support Office

This takes our existing `tx_plot` and then applies the `theme_minimal()` look to it. This is actually our "Goal graphic" that we were shooting for at the beginning of the lesson!

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

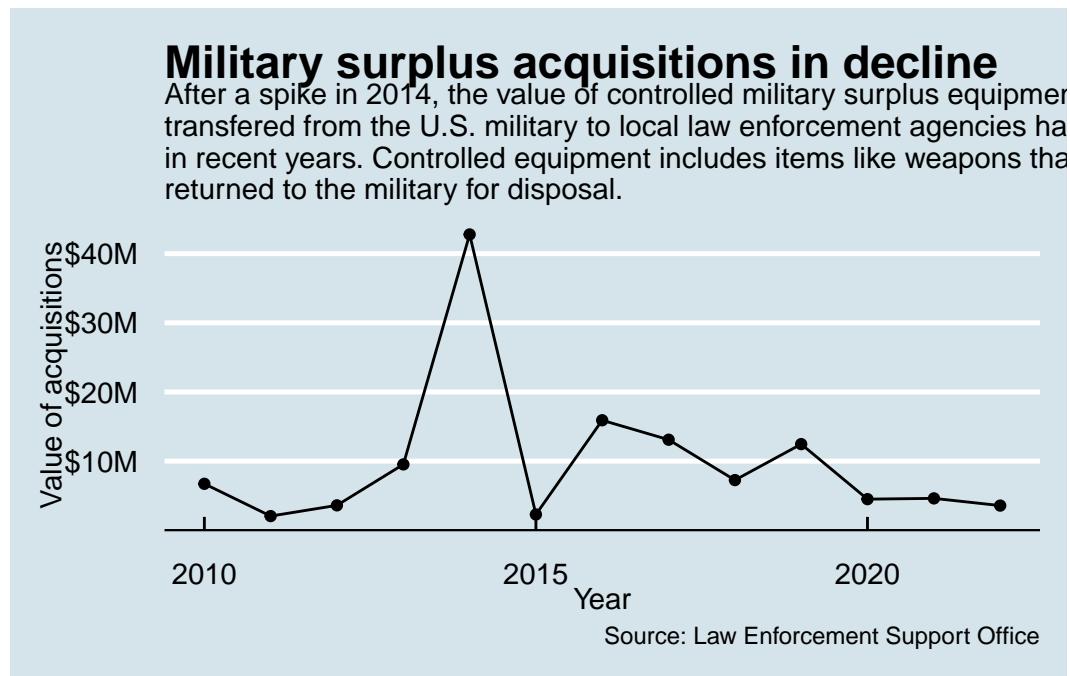
1. Edit your existing chunk to try different themes. Some you might try are `theme_classic()`, `theme_dark()` and `theme_void()`.

### 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 using the `install.packages()` function:  
`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()
```

## Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has dropped in recent years. Controlled equipment includes items like weapons returned to the military for disposal.

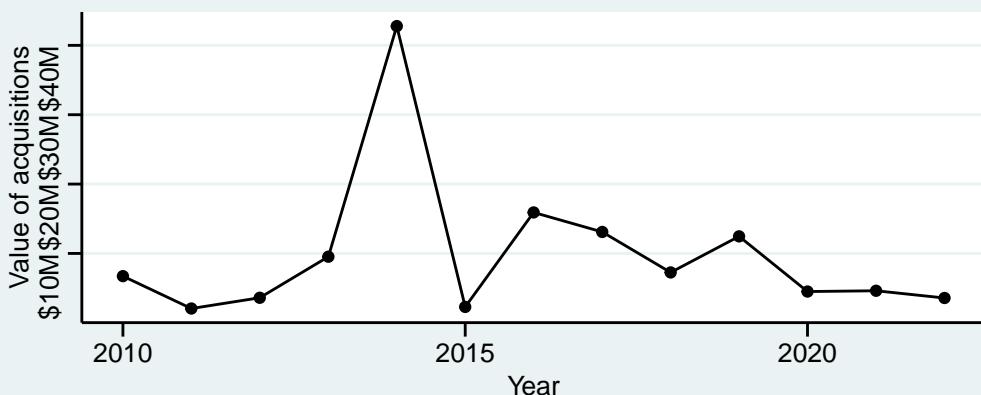


Source: Law Enforcement Support Office

```
tx_plot +  
  theme_stata()
```

## Military surplus acquisitions in decline

After a spike in 2014, the value of controlled military surplus equipment transferred from the U.S. military to local law enforcement agencies has dropped in recent years. Controlled equipment includes items like weapons that must be returned to the military for disposal.



Source: Law Enforcement Support Office

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 Goal 2: Multiple line chart

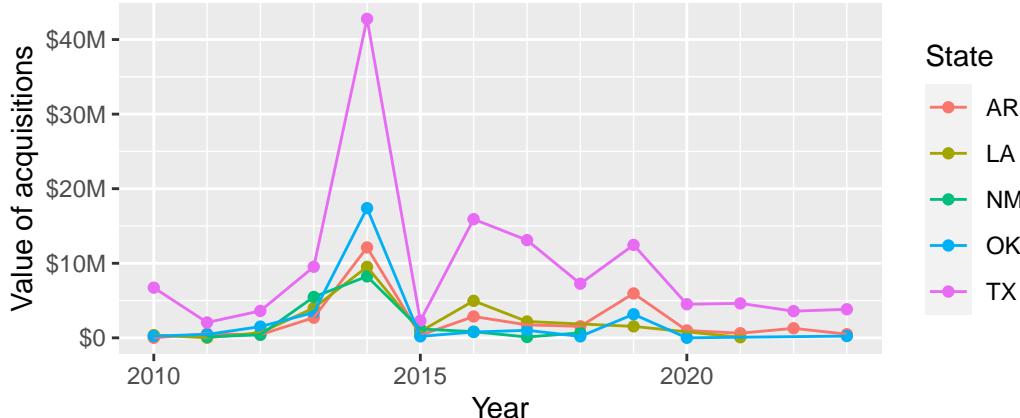
OK, our Texas military surplus information is fine ... but how does that compare to neighboring states? Let's work through building a new chart that shows all those steps. It's essentially the same chart as we did above, but adding the bordering states along with Texas.

Here is the data and image we are trying to make:

```
# A tibble: 58 x 3
# Groups:   state [5]
  state year     yearly_value
  <chr> <date>      <dbl>
1 AR    2010-01-01     1236
2 AR    2011-01-01   542429.
3 AR    2012-01-01   425292.
4 AR    2013-01-01  2698799.
5 AR    2014-01-01  12125078.
6 AR    2015-01-01  342453.
7 AR    2016-01-01  2866167
8 AR    2017-01-01  1736827.
9 AR    2018-01-01  1545877.
10 AR   2019-01-01  5953691.
# i 48 more rows
```

## Military surplus acquisitions track among Texas, bordering states

State and local law enforcement agencies can request surplus military equipment from the U.S. Department of Defence, paying only for the shipping. After 2010, Texas and bordering states have followed similar trends in requests since, with declining requests in recent years.



Source: Law Enforcement Support Office

### 8.6.1 Prepare the data

We need to go back to our original `leso` to get the additional states. But we also know now that we need to create a “`floor_date`” that has a `<date>` datatype. Let’s do that as we prepare the data instead of while we are grouping. I’ll sometimes go back into my cleaning notebook to add something like this, but we won’t go quite that far with this.

In the interest of time, here is all the code we are using, with annotations to explain each line.

```
region_data <- leso |>
  filter(state %in% c("TX", "OK", "AR", "NM", "LA")) |>
  mutate(
    year = floor_date(ship_date, unit = "year") |> date()
  ) |>
  group_by(state, year) |>
  summarize(yearly_value = sum(total_value))

region_data
```

- ① We create a new object to hold the data. We fill that starting with `leso`.
- ② We filter the `state` column for our list of five states. Note we are using `%in%` like we did in Chapter 6, but we’re doing it right inside the filter.

- ③ Here, inside a `mutate()`, we create the new variable `year` and fill it with the `floor_date` of our shipping date based on the year. We then turn that into a `<date>` datatype.
- ④ We group by `state` and `year`. This is much cleaner now that we've already created a `year` value.
- ⑤ Our summarize to total the costs.
- ⑥ And lastly, we print the new object so we can see it.

```
# A tibble: 58 x 3
# Groups:   state [5]
  state year     yearly_value
  <chr> <date>    <dbl>
1 AR    2010-01-01    1236
2 AR    2011-01-01   542429.
3 AR    2012-01-01   425292.
4 AR    2013-01-01  2698799.
5 AR    2014-01-01 12125078.
6 AR    2015-01-01  342453.
7 AR    2016-01-01  2866167
8 AR    2017-01-01  1736827.
9 AR    2018-01-01  1545877.
10 AR   2019-01-01  5953691.
# i 48 more rows
```

### 8.6.2 Plot the chart

For our next plot, we'll 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](#). For now, though, we'll use the `ggplot` default colors.

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(region_data, (1)
       aes(x = year, y = yearly_value)) + (2)
       geom_point() + (3)
       geom_line(aes(color = state)) + (4)
       scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) + (5)
       scale_x_date(minor_breaks = breaks_width("1 year")) + (6)
```

```

labs(title = "Military surplus acquisitions track among Texas, bordering states",
     subtitle = str_wrap("State and local law enforcement agencies can request surplus military equipment from the U.S. Department of Defence, paying only for the shipping. Ager 2010, Texas and bordering states have followed similar trends in requests since, with declining requests in recent years."),
     x = "Year", y = "Value of acquisitions",
     caption = "Source: Law Enforcement Support Office")

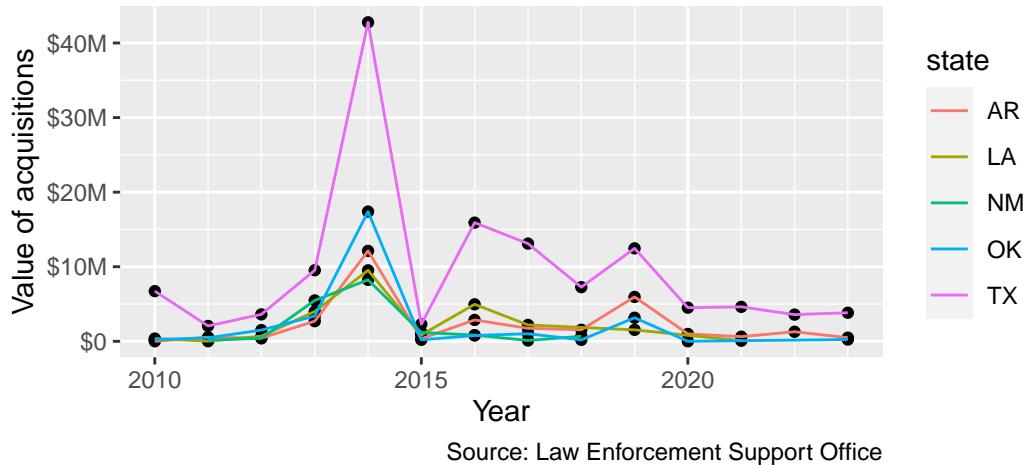
```

(7)

- ① Plotting the `region_data` object
- ② Set set the x and y axis
- ③ Adds the points
- ④ Adds the line, coloring them based on the `state` values
- ⑤ Fixes our labels on the y axis.
- ⑥ Fixes our minor breaks on the x axis.
- ⑦ Adding all our labels with `labs()` arguments.

## Military surplus acquisitions track among Texas, bordering states

State and local law enforcement agencies can request surplus military equipment from the U.S. Department of Defence, paying only for the shipping. After 2010, Texas and bordering states have followed similar trends in requests since, with declining requests in recent years.



Source: Law Enforcement Support Office

Adding this aesthetic not only added color to the lines, it added a legend so we can tell which state is which. If we had tried to build this plot without the added color aesthetic, it wouldn't plot right because it wouldn't know how to split the lines.

Notice that only the lines changed colors, and not the points? 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)`.
2. Also edit the `labs()` function to add `color = "State"`.

```

ggplot(region_data,
       aes(x = year, y = yearly_value)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state)) +
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(title = "Military surplus acquisitions track among Texas, bordering states",
       subtitle = str_wrap("State and local law enforcement agencies can request surplus military equipment from the U.S. Department of Defence, paying only for the shipping. After 2015, Texas and bordering states have followed similar trends in requests since with declining requests in recent years.", x = "Year", y = "Value of acquisitions",
       caption = "Source: Law Enforcement Support Office",
       color = "State")

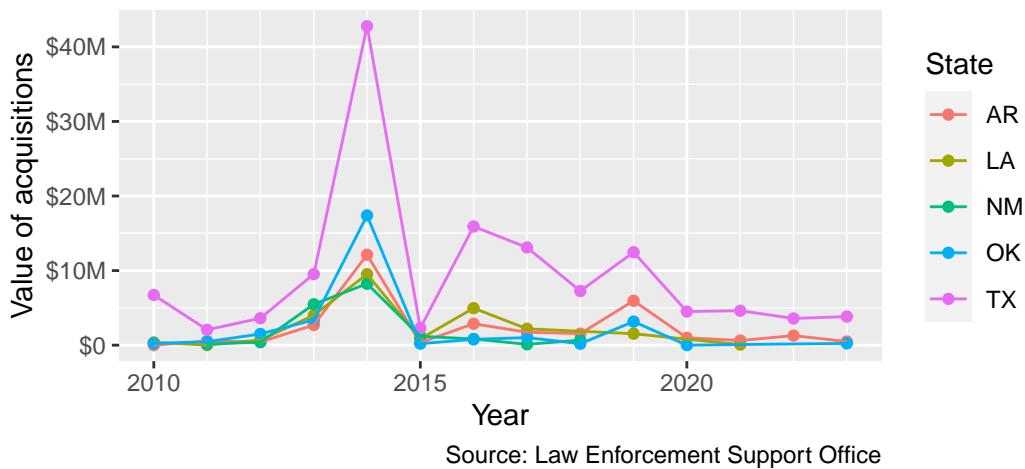
```

① Color aesthetic added here.

② Adding `color = "State"` here redraws the title above the color legend.

### Military surplus acquisitions track among Texas, bordering states

State and local law enforcement agencies can request surplus military equipment from the U.S. Department of Defence, paying only for the shipping. After 2015, Texas and bordering states have followed similar trends in requests since with declining requests in recent years.



OK, you have a line chart for reference!

## 8.7 On your own

I want you to make a line chart of military surplus acquisitions in three states that are **different** from the five we used above. 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.

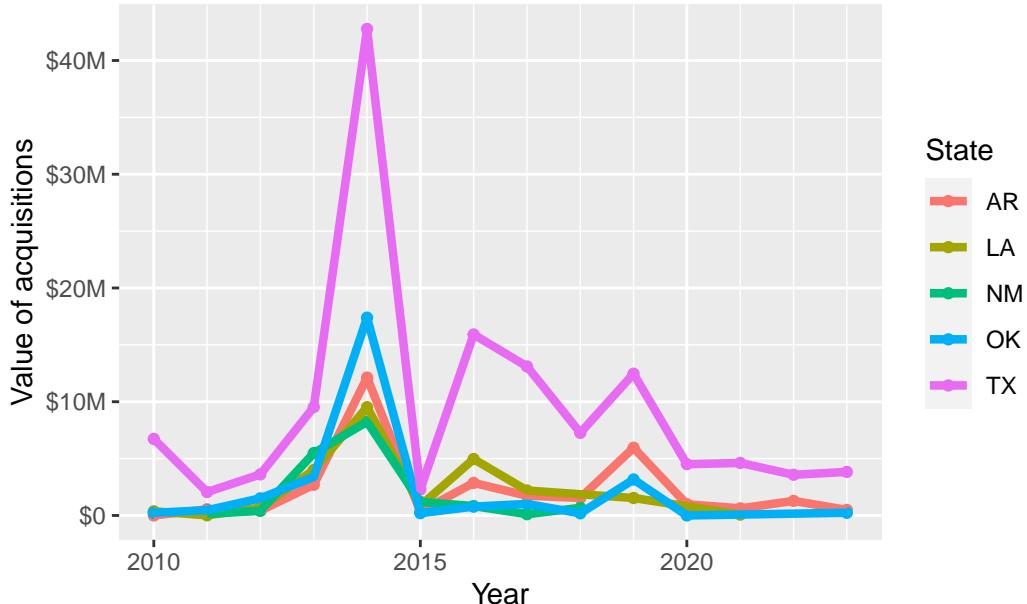
## 8.8 Tour of some other adjustments

These examples show how to tweak different parts of charts. Add them to your notebook, run it, then find the value that makes the change and adjust it with new values so you can see the difference.

### 8.8.1 Line width

```
ggplot(region_data,
       aes(x = year, y = yearly_value)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state), size = 1.5) + ①
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(x = "Year", y = "Value of acquisitions",
       caption = "Source: Law Enforcement Support Office",
       color = "State")
```

- ① You can set the line width for a geom, but note it is taking an inputted value, not one from the data. This is why it is outside the `aes()` function.



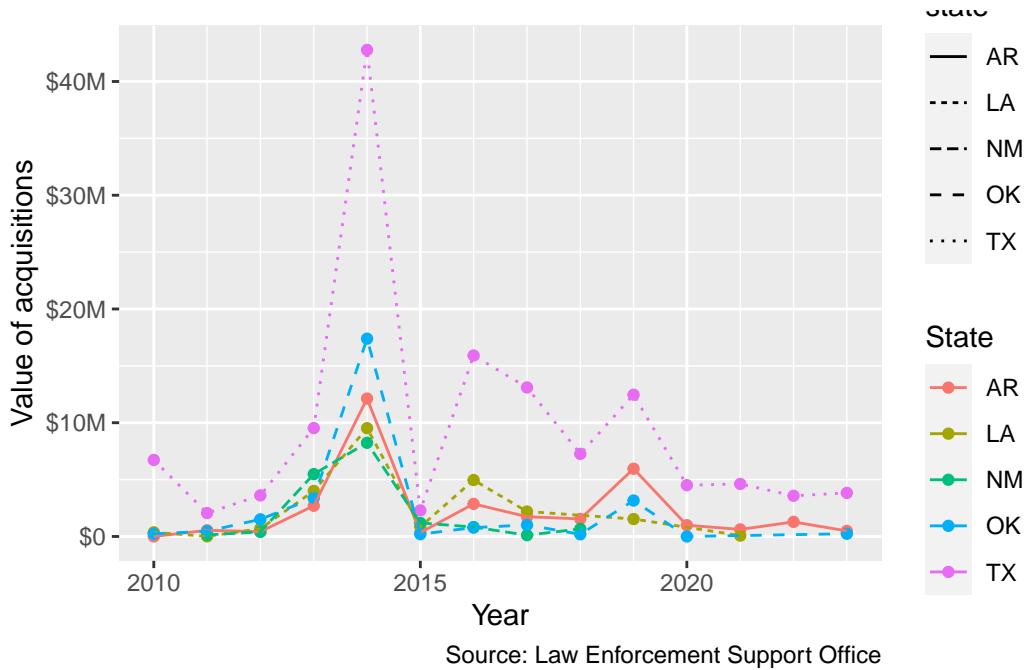
Source: Law Enforcement Support Office

### 8.8.2 Line type

This example adds a `linetype = state` to the ggplot aesthetic. This gives each state a different type of line.

```
ggplot(region_data,
       aes(x = year, y = yearly_value)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state, linetype = state), size = .5) + ①
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(x = "Year", y = "Value of acquisitions",
       caption = "Source: Law Enforcement Support Office",
       color = "State")
```

- ① We add the linetype here, but have to put it INSIDE the `aes()` now because we are “mapping” the type based on values in the `state` variable.



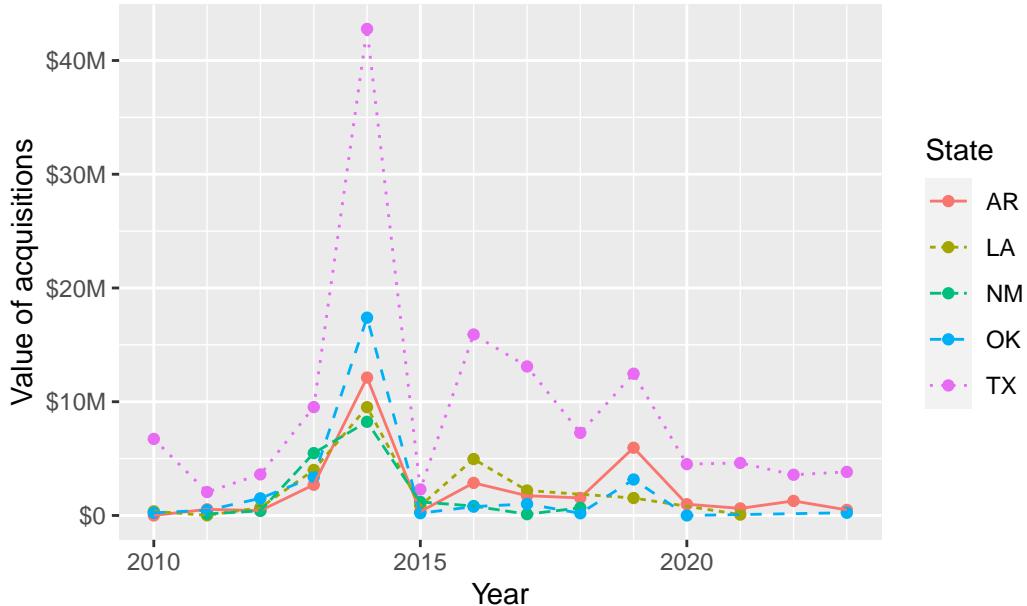
Source: Law Enforcement Support Office

But note that ggplot added a new legend because it takes the name from the variable “state”. We can fix this through overriding it in the `labs()` function.

```
ggplot(region_data,
       aes(x = year, y = yearly_value)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state, linetype = state), size = .5) +
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(x = "Year", y = "Value of acquisitions",
       caption = "Source: Law Enforcement Support Office",
       color = "State", linetype = "State")
```

①

- ① We add `linetype = "State"` to the `labs()` function, which would collapse everything into a single legend.



Source: Law Enforcement Support Office

## 8.9 Facets

Facets are a way to make multiple graphs based on a variable in the data, as it creates a bunch of mini charts you can compare among each other. 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 with the chart we have already, and then split it into facets by state.

1. Start a new section about facets
2. Add the code below to create your chart and view it. This is the same plot we've already created

```
ggplot(region_data,
       aes(x = year, y = yearly_value)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state, linetype = state), size = .5) +
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(title = "Military surplus acquisitions track among Texas, bordering states",
       subtitle = str_wrap("State and local law enforcement agencies can request surplus mil",
       x = "Year", y = "Value of acquisitions",
       caption = "Source: Law Enforcement Support Office",
```

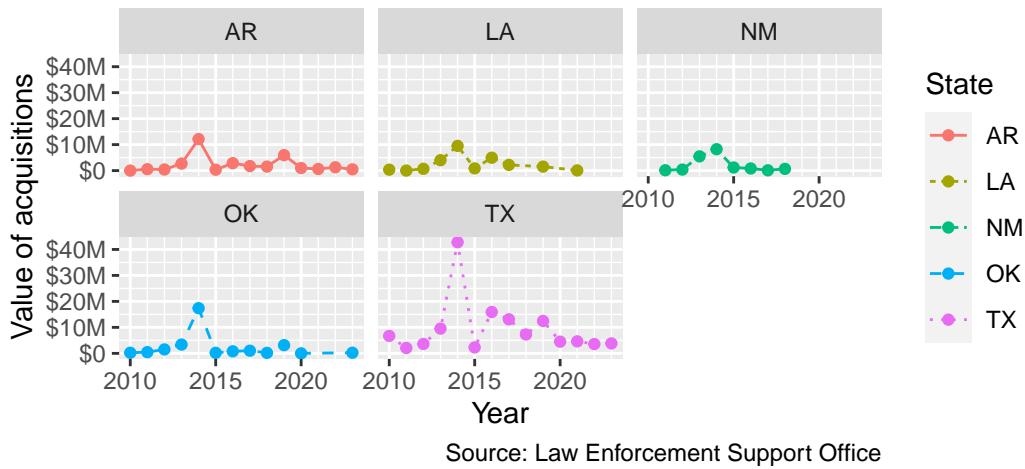
```
color = "State", linetype = "State") +
facet_wrap(~ state)
```

①

- This is the only line we added, setting a facet wrap to split on the state.

## Military surplus acquisitions track among Texas, bordering states

State and local law enforcement agencies can request surplus military equipment from the U.S. Department of Defence, paying only for the shipping. Agencies in Texas and bordering states have followed similar trends in requests since 2010, with declining requests in recent years.



Source: Law Enforcement Support Office

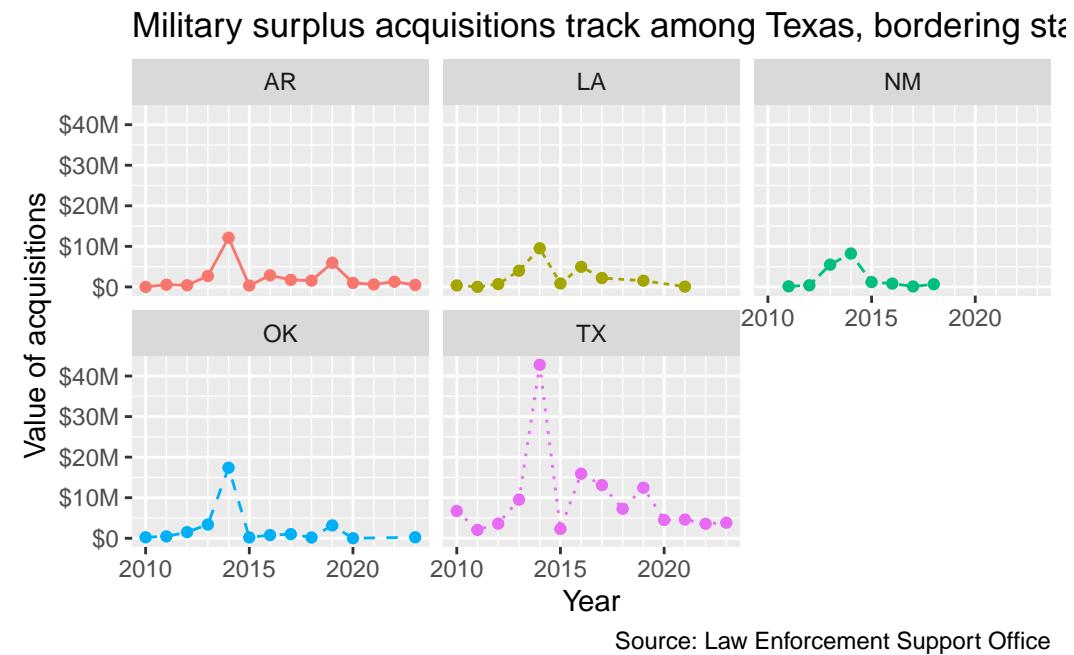
Since each state facet is labeled, we don't really need the color legend, so we can use the `theme()` function to override that.

- Add the last line here to remove the legend.

```
ggplot(region_data,
       aes(x = year, y = yearly_value)) +
  geom_point(aes(color = state)) +
  geom_line(aes(color = state, linetype = state), size = .5) +
  scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
  scale_x_date(minor_breaks = breaks_width("1 year")) +
  labs(title = "Military surplus acquisitions track among Texas, bordering states",
       x = "Year", y = "Value of acquisitions",
       caption = "Source: Law Enforcement Support Office",
       color = "State", linetype = "State") +
  facet_wrap(~ state) +
  theme(legend.position = "none")
```

①

- ① Here we set the legend.position to “none”, which is the answer to remove it. We could also set that to “bottom” or “left” to move it, if that was our desire. It is not, but you can try it if you like.



### 8.9.1 Facet grids

A `facet_grid()` allows you to plot on a combination of variables. We don’t really have enough variables to compare with our military surplus data but we can show this with the `penguins` data we used in the last chapter.

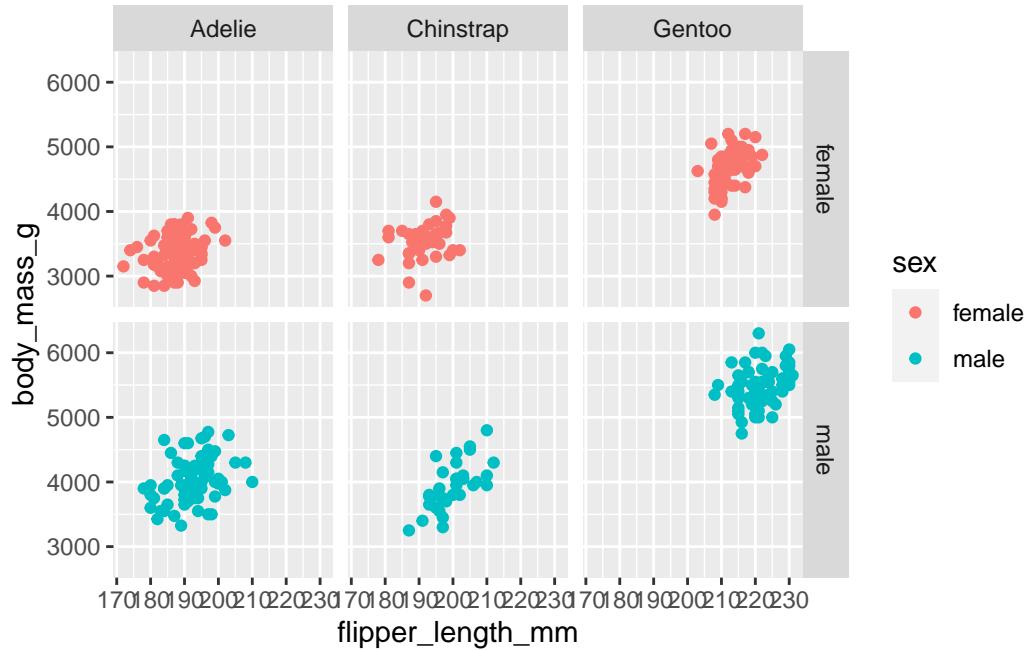
This chart compares the flipper length and body mass, but it does so for each species of bird by their gender.

```
library(palmerpenguins)

penguins |>
  drop_na() |> ①
  ggplot(aes(x = flipper_length_mm, y = body_mass_g)) + ②
  geom_point(aes(color = sex)) + ③
  facet_grid(sex ~ species) ④
```

- ① This drops any rows that are missing values, as some don’t have sex marked.  
 ② Our ggplot chart, much like we did in the last chapter, comparing flippers to weight.

- ③ We add the points and color them to distinguish the sex of the birds.  
 ④ We add the `facet_grid` that splits the chart by both sex and species.



## 8.10 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!

this has been suppressed for printing

```
library(plotly)

region_plot <- ggplot(region_data,
  aes(x = year, y = yearly_value)) +
  
```

①

```
geom_point(aes(color = state)) +
geom_line(aes(color = state)) +
scale_y_continuous(labels = label_dollar(scale_cut = cut_long_scale())) +
scale_x_date(minor_breaks = breaks_width("1 year")) +
labs(title = "Military police",
x = "Year", y = "Value of acquisitions",
caption = "Source: Law Enforcement Support Office",
color = "State")

region_plot |>
ggplotly()
```

(2)

- ① We saved our good plot into an object.  
② Then we pipe that object into the `ggplotly()` function, which makes it interactive.

Now you have tool tips on your points when you hover over them.

The `ggplotly()` function is not perfect. For instance, I've shortened the title to the point of uselessness because it ran on top of the controls. I'm sure that could be overcome with some effort. Alternatively, you can use `plotly`'s own syntax to build some quite interesting charts, but it's a whole [new syntax to master](#).

## 8.11 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.

# 9 Tidy data

This chapter was written by Prof. McDonald, who uses a Mac.

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 it helps us with both ggplot and other charting tools like [Datawrapper](#).

## 9.1 Goals for this section

- Explore what it means to have “tidy” data.
- Learn about and use `pivot_longer()`, `pivot_wider()` to shape our data for different purposes.
- Use candy data to practice shaping data.

## 9.2 The questions we'll answer

- Are candy colors evenly distributed within a standard package of M&M’s? (i.e., average per color in a standard package)
  - We’ll plot the result as a column chart to show the average number of colored candies. We’ll do it first in ggplot, then 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”.

# Tidy Data



A data set is **tidy** iff:

1. Each **variable** is in its own **column**
2. Each **case** is in its own **row**
3. Each **value** is in its own **cell**

Adapted from 'Master the tidyverse' CC by RStudio



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.

## table1 is tidy

The data is a subset of the data contained in the  
World Health Organization Global Tuberculosis  
Report

| country     | year | cases  | population |
|-------------|------|--------|------------|
| Afghanistan | 1999 | 745    | 19987071   |
| Afghanistan | 2000 | 2666   | 20595360   |
| Brazil      | 1999 | 37737  | 172006362  |
| Brazil      | 2000 | 80488  | 174504898  |
| China       | 1999 | 212258 | 1272915272 |
| China       | 2000 | 213766 | 1280428583 |

6 rows



Adapted from 'Master the tidyverse' CC by RStudio

Table2 below isn't tidy. The **count** column contains two different type of values.

## table2 isn't tidy

contains two variables

| country     | year | type       | count      |
|-------------|------|------------|------------|
| Afghanistan | 1999 | cases      | 745        |
| Afghanistan | 1999 | population | 19987071   |
| Afghanistan | 2000 | cases      | 2666       |
| Afghanistan | 2000 | population | 20595360   |
| Brazil      | 1999 | cases      | 37737      |
| Brazil      | 1999 | population | 172006362  |
| Brazil      | 2000 | cases      | 80488      |
| Brazil      | 2000 | population | 174504898  |
| China       | 1999 | cases      | 212258     |
| China       | 1999 | population | 1272915272 |

1-10 of 12 rows

Previous



It's hard to manipulate

Adapted from 'Master the tidyverse' CC by RStudio

When our data is tidy, it is easy to manipulate. We can use functions like `mutate()` to calculate new values for each case.

Tidy data is easy to manipulate

The data is a subset of the data contained in the  
World Health Organization Global Tuberculosis  
Report

| country     | year  | cases  | population | rate         |
|-------------|-------|--------|------------|--------------|
| <chr>       | <int> | <int>  | <int>      | <dbl>        |
| Afghanistan | 1999  | 745    | 19987071   | 0.0000372741 |
| Afghanistan | 2000  | 2666   | 20595360   | 0.0001294466 |
| Brazil      | 1999  | 37737  | 172006362  | 0.0002193930 |
| Brazil      | 2000  | 80488  | 174504898  | 0.0004612363 |
| China       | 1999  | 212258 | 1272915272 | 0.0001667495 |
| China       | 2000  | 213766 | 1280428583 | 0.0001669488 |

6 rows

```
table1 %>%  
  mutate(rate = cases/population)
```

Adapted from master the tidyverse - CC BY RStudio



When our data is tidy, we can use the `tidyverse` package to reshape the layout of our data to suit our needs. It gets loaded with `library(tidyverse)` so we won't need to load it separately.

### 9.3.1 Wide vs long data

In the figure below, the table on the left is “wide”. There are multiple year columns describing the same variable. It might be useful if we want to calculate 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 a single “year” column to map as an x or y axes.

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 in ggplot. We can then set our “Year” column as an x-axis, our “n” column on our y-axis, and group by the “Country”.

---

| 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 9.1: Wide vs long

**Neither shape is wrong**, they are just useful for different purposes. In fact, you'll find yourself pivoting the same data in different ways depending on your needs.

### 9.3.2 Why we might want different shaped data

There are a myriad of reasons why you might need to reshape your data. Performing calculations on row-level data might be easier if it is wide. Grouping and summarizing calculations might be easier when it is long. ggplot graphics like long data, while Datawrapper sometimes wants wide data to make the same chart.

I find myself shaping data back and forth depending on my needs.

## 9.4 The candy project

Let's visualize our goal here before we jump into the candy bowl.

Our first question is this: **Are there equal numbers of different-colored candies in a bag of M&M's?**

To visually communicate this, we'll build a chart similar to this one (though this one is about Skittles instead of M&M's.)

## But I like Green Skittles ...

Are Skittles candy colors equally distributed across multiple 61.5g packages? Students in Prof. Christian McDonald's Reporting with Data classes counted candy colors in 94 bags to find the average number per bag for each color.



Chart: Christian McDonald • Source: Class data • Created with Datawrapper

Figure 9.2: Skittles example

We'll build this chart in both ggplot and Datawrapper.

### 9.4.1 Prepare our candy project

We will use candy count data we've been collected in Reporting wth Data classes to explore this subject.

Start a new project.

1. Create a new project and call it: `yourname-candy`
2. No need to create data folders as we'll just load data directly into the notebook.
3. You can use your `index.qmd` file ... just change the title to "Tidy data: Candy".
4. Create your setup block and load the `tidyverse` and `janitor`.

Warning: package 'ggplot2' was built under R version 4.3.1

### 9.4.2 Get the data

We'll just load this data directly from Google Sheets into this notebook.

1. Add a Markdown section noting that you are importing data.
2. Add this import chunk and run it.

```
candy_raw <- read_csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vRCGayKLoY-52gKmEoP0j")

# peek at the data
candy_raw

# A tibble: 138 x 11
  timestamp first_name last_name candy_type box_code   red green orange yellow
  <chr>     <chr>     <chr>     <chr>     <chr>    <dbl> <dbl> <dbl> <dbl>
1 2/21/2022~ Christian McDonald Plain 140BSCL~     2     17    11     4
2 2/28/2022~ Andrew Logan Plain 140BSCL~     2      9    14     4
3 2/28/2022~ Gabby Ybarra Plain 140BSCL~     4     11    14     0
4 2/28/2022~ Veronica Apodaca Plain 140BSCL~     1     17    16     2
5 2/28/2022~ Cristela Jones Plain 140BSCL~     3     19    14     3
6 2/28/2022~ Marina Garcia Plain 140BSCL~     7     11    13     1
7 2/28/2022~ Samuel Stark Plain 140BSCL~     3      9    10     5
8 2/28/2022~ Kevin Malcolm ~ Plain 140BSCL~     4     15    13     6
9 2/28/2022~ Alexa Haverlah Plain 140BSCL~     5     15     6     2
10 2/28/2022~ Zacharia Washingt~ Plain 140BSCL~    1     15    14     1
# i 128 more rows
# i 2 more variables: blue <dbl>, brown <dbl>
```

This data comes from a Google Sheets document fed by a form that students have filled out, counting the colors of candies in a standard size bag of plain M&Ms.

#### 9.4.3 Drop unneeded columns

For this exercise we don't need the `timestamp`, `candy_type` or `box_code` columns. We'll drop them so we can keep things simple.

1. Create new section noting you'll drop unneeded columns.
2. Create an R chunk and use `select()` to remove the columns noted above and save the result into a new data frame called `candy`. Remember you can negate a list of columns like this: `!c(col1, col2)`.

You've done this in the past, so you should be able to do it on your own.

```
candy <- candy_raw |>
  select(!c(timestamp, candy_type, box_code))
```

#### 9.4.4 Peek at the wide table

Let's take look closer at this data:

```
candy |> head()
```

```
# A tibble: 6 x 8
  first_name last_name   red green orange yellow blue brown
  <chr>      <chr>     <dbl> <dbl>  <dbl>  <dbl> <dbl> <dbl>
1 Christian McDonald     2     17    11     4     16     4
2 Andrew     Logan        2      9    14     4     17     9
3 Gabby      Ybarra       4     11    14     0     19     7
4 Veronica   Apodaca     1     17    16     2     13     8
5 Cristela   Jones        3     19    14     3     13     3
6 Marina     Garcia       7     11    13     1     14     6
```

This is pretty well-formed data. This format would be useful to create a “total” column for each bag, but there are better ways to do this with **long** data. Same with getting our averages for each color.

#### 9.4.5 Where are we going with this data

Let's look at our plotting goal again:

### But I like Green Skittles ...

Are Skittles candy colors equally distributed across multiple 61.5g packages? Students in Prof. Christian McDonald's Reporting with Data classes counted candy colors in 94 bags to find the average number per bag for each color.

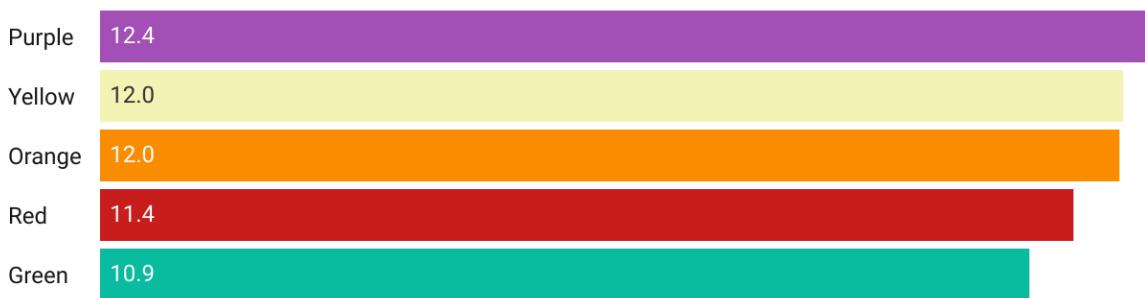


Chart: Christian McDonald • Source: Class data • Created with Datawrapper

Figure 9.3: Skittle example

In our chart, we want one axis to have the candy color and the other to have the average number of candy.

To plot a chart like this in ggplot or Datawrapper, the data needs to be the same shape, like this:

| Color  | Average |
|--------|---------|
| Green  | 10.9    |
| Orange | 12.0    |
| Purple | 12.4    |
| Red    | 11.4    |
| Yellow | 12      |

It will be easier to accomplish both of these tasks if our data were in the **long** format.

So, instead of this:

| first_name | last_name | red | green | orange | yellow | blue | brown |
|------------|-----------|-----|-------|--------|--------|------|-------|
| Christian  | McDonald  | 2   | 17    | 11     | 4      | 16   | 4     |

We want this:

| first_name | last_name | color  | candies |
|------------|-----------|--------|---------|
| Christian  | McDonald  | red    | 2       |
| Christian  | McDonald  | green  | 17      |
| Christian  | McDonald  | orange | 11      |
| Christian  | McDonald  | yellow | 4       |
| Christian  | McDonald  | blue   | 16      |
| Christian  | McDonald  | brown  | 4       |

## 9.5 The **tidyverse** verbs

The two functions we'll use to reshape are data are:

- `pivot_longer()` which “lengthens” data, increasing the number of rows and decreasing the number of columns.
- `pivot_wider()` which “widens” data, increasing the number of columns and decreasing the number of rows.

Again, the best way to learn this is to present a problem and then solve it with explanation.

## 9.6 Pivot longer

This visualization gives you an idea how `pivot_longer()` works.

Turn wide data into tall data with **pivot\_longer()**

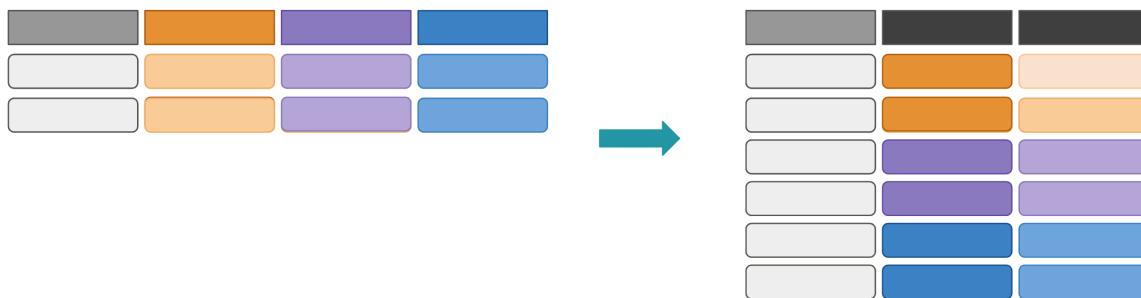


Figure 9.4: Pivot longer

Each column of data chosen (the colored ones) is turned into it's own row of data. Supporting data (the grey columns) are duplicated.

The `pivot_longer()` function needs several arguments: `cols=`, `names_to=` and `values_to=`. Below are two examples to pivot the example data shown above.

```
dataframe %>% pivot_longer(cols=2:4, names_to= "name", values_to = "name",)
```

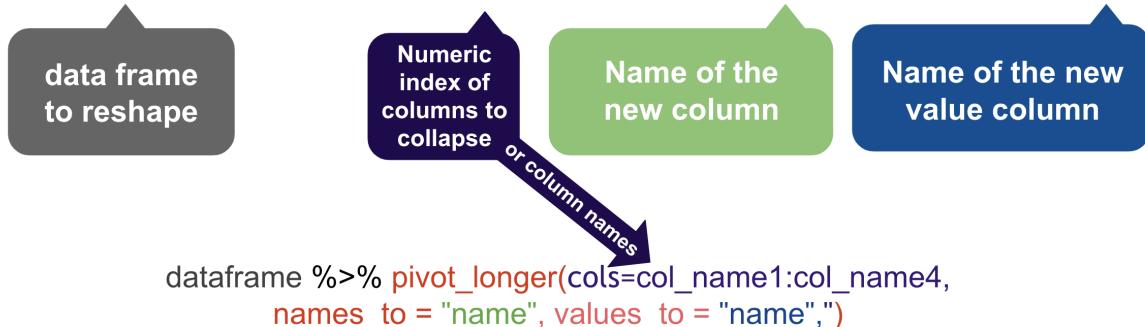


Figure 9.5: pivot\_longer code

- `cols=` is where you define a range of columns you want to pivot. *For our candy data we want the range red:brown.*
- `names_to=` allows you to name the new column filled by the column names. *For our candy data we want to name this “color” since that’s what those columns described.*
- `values_to=` allows you to name the new column filled with the cell data. *For us we want to call this “count\_candies” since these are the number of candies in each bag.*

There are a number of ways we can describe the `cols=` argument ... anything in `tidy-select` works. You can see a bunch of [examples here](#).

### 9.6.1 Pivot our candy data longer

What we want here is six rows for each person’s entry, with a column for “color” and a column for “count\_candies”.

We are using a range, naming the first “red” and the last column “brown” with `:` in between. This only works because those columns are all together. Otherwise we could list the all with `c(red, blue, green)` etc.

1. Add a note that you are pivoting the data
2. Add the chunk below and run it

```
candy_long <- candy |>
  pivot_longer(
    cols = red:brown,
    names_to = "color",
    values_to = "count_candies"
  )

candy_long |> head()
```

- ① Sets which columns to pivot based on their names.
- ② Sets the new column name for the former column names. This would default to “name” if we didn’t set it.
- ③ Sets the new column name for the former values. This would default to “value” if we didn’t set it.

```
# A tibble: 6 x 4
  first_name last_name color  count_candies
  <chr>      <chr>     <chr>        <dbl>
1 Christian  McDonald  red         2
2 Christian  McDonald  green       17
```

|   |           |          |        |    |
|---|-----------|----------|--------|----|
| 3 | Christian | McDonald | orange | 11 |
| 4 | Christian | McDonald | yellow | 4  |
| 5 | Christian | McDonald | blue   | 16 |
| 6 | Christian | McDonald | brown  | 4  |

### 9.6.2 Get average candies per color

To get the average number of candies per each color, we can use our `candy_long` data and `group_by` color (which will consider all the `red` rows together, etc.) and use `summarize()` to get the mean.

This is very similar to the `sum()`s we did with military surplus, but you use `mean()` instead. One thing you have to watch for with `mean()` is you might need the argument `na.rm = TRUE` if there are missing or zero values, since you can't divide by zero. We'll include that here in case there are bags that don't have any of a specific color.

Save the resulting summary table into a new tibble called `candy_avg`.

```
candy_avg <- candy_long |>
  group_by(color) |>
  summarize(avg_candies = mean(count_candies, na.rm = TRUE)) ①
candy_avg
```

① The `na.rm` hedges our bets in case some bags are missing a color.

```
# A tibble: 6 x 2
  color   avg_candies
  <chr>     <dbl>
1 blue      11.9
2 brown     6.22
3 green     11.9
4 orange    10.6
5 red       6.14
6 yellow    9.20
```

### 9.6.3 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 summarize function to add the `mutate()` function below.

```
candy_avg <- candy_long |>
  group_by(color) |>
  summarize(avg_candies = mean(count_candies, na.rm = TRUE)) |>
  mutate(
    avg_candies = round(avg_candies, 1)
  )

candy_avg
```

```
# A tibble: 6 x 2
  color   avg_candies
  <chr>     <dbl>
1 blue       11.9
2 brown      6.2
3 green      11.9
4 orange     10.6
5 red        6.1
6 yellow     9.2
```

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 a bonus point 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.

#### 9.6.4 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 by color in a bag. This is very similar to the [Disney Princesses bar chart in Intro to ggplot](#)).

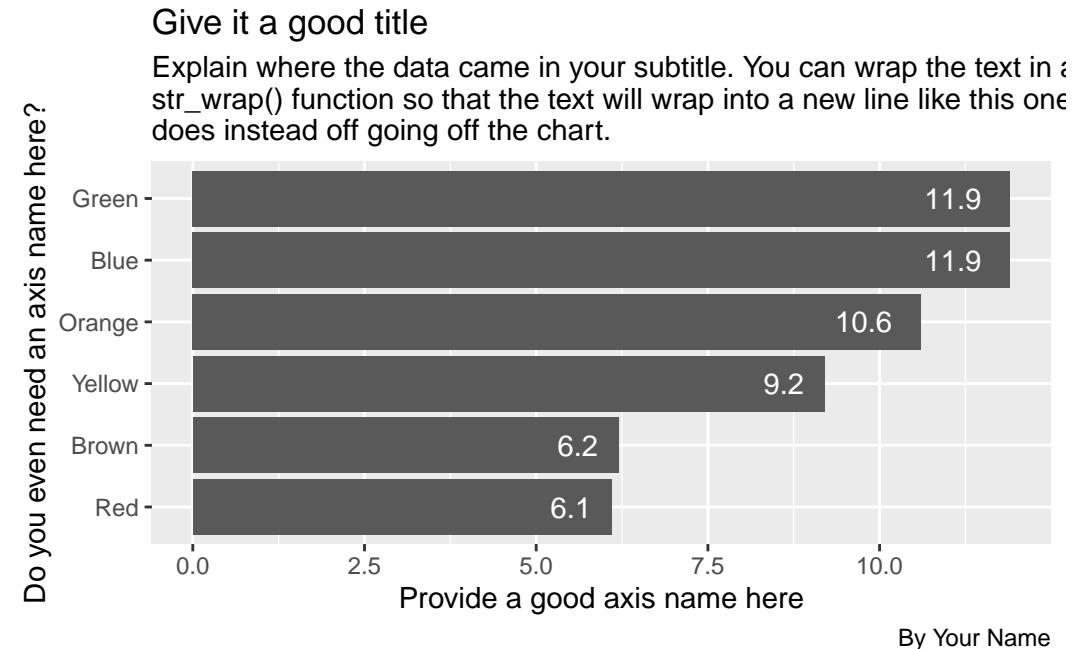
1. Build a bar chart of average color using ggplot.

Some things to consider:

- I want the bars to be ordered by the highest average on top.
- I want you to have a good title, subtitle and byline, along with good axes names. Make sure a reader has all the information they need to understand what you are communicating with the chart.

- Include the values on the bars.
- Change the theme to something other than the default.

Here is what it should more or less look like, but with good text, etc:



The numbers in the example above may not be up to date, so don't let that throw you.

## 9.7 Introducing Datawrapper

There are some other great charting tools that journalists use. My favorite is [Datawrapper](#) and it 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 `candy_avg` tibble is what you need.

Here are the steps I want you to follow:

### 9.7.1 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 `candy_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.

### 9.7.2 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 upload the data, but we have to write out the data to your computer first.

### 9.7.3 Export your data for Datawrapper

1. Go back to RStudio.
2. Create a new block and name it something about exporting
3. Take your data and pipe it into `write_csv()`. As an argument to `write_csv()`, give it a path and name of the file: We can write directly to our project folder as `candy_avg.csv`.

```
candy_avg |> write_csv("candy_avg.csv")
```

This will save your data file onto your computer in your project folder.

#### 9.7.3.1 posit.cloud export

If you are using posit.cloud, you will also need to **Export** your data from the cloud to get it onto your computer so you can import it into Datawrapper. Do this **only** if you are using the cloud version.

1. In the Files window (bottom right) go inside your `data-processed` folder.
2. Click on the checkbox next to the `candy_avg.csv` file.
3. Click on the **More** blue gear thing and choose **Export**.

4. You'll get a prompt to name the file (which you can keep the same) and then a **download** button. Click the button.
5. The file should go to your Downloads folder like anything else you download from the Internet.
6. When you go back to **Datawrapper** click on the **XLS/CSV upload** button and go find your file to import.

#### 9.7.4 Build the datawrapper graphic

1. Return to **Datawrapper** in your browser and click on the **XLS/CSV upload** button and go find your file to import it.
2. Once your data is either uploaded or copied into Datawrapper, you can click **Proceed** to go to the next step.

You can now follow the [Datawrapper Academy](#) directions to finish your chart.

When you get to the Publish & Embed window, I want you to click the **Publish Now** button and then add the resulting *Link to your visualization:* URL to your R Notebook so I can find it for grading.

## 9.8 Pivot wider

Now that you've pivoted data longer, I'd also like to show how you can pivot your table into a wide format. I'll sometimes do this to make an easier to read display or to shape data a specific way for Datawrapper or some other visualization software. Let's walk through the concept first.

As you can imagine, `pivot_wider()` does the opposite of `pivot_longer()`. When we pivot wider we move our data from a “long” format to a “wide” format. We create a new column based categories and values in the data.

The diagram illustrates the transformation of long data into wide data. On the left, a 'long' data frame is shown with columns X1, V1, and V2. The rows contain data for categories A and B across three variables X2, X3, and X4. An arrow points to the right, indicating the transformation. On the right, a 'wide' data frame is shown with columns X1, X2, X3, and X4. The data is now grouped by category A and B, with each variable having its own column.

| X1 | V1 | V2  |
|----|----|-----|
| A  | X2 | 1   |
| A  | X3 | 0.1 |
| A  | X4 | 10  |
| B  | X2 | 2   |
| B  | X3 | 0.2 |
| B  | X4 | 20  |

| X1 | X2 | X3  | X4 |
|----|----|-----|----|
| A  | 1  | 0.1 | 10 |
| B  | 2  | 0.2 | 20 |

Figure 9.6: Long to wide

`pivot_wider()` needs two arguments:

- `names_from` = lets us define from which column (or columns) we are pulling values from to create the new column names. In the example above, this would be “V1”.
- `values_from` = lets us say which column will be the values in the new arrangement. In the example above this would be “V2”.

So the code to flip the data above would be:

```
df |>
  pivot_wider(names_from = V1, values_from = V2)
```

### 9.8.1 Pivot wider example

Let's do this with our data, taking our “long” data and making it “wide” again.

Here is our “long” data:

```
candy_long
```

```
# A tibble: 828 x 4
  first_name last_name color  count_candies
  <chr>      <chr>    <chr>        <dbl>
1 Christian  McDonald red       2
2 Christian  McDonald green    17
3 Christian  McDonald orange   11
4 Christian  McDonald yellow  4
5 Christian  McDonald blue    16
6 Christian  McDonald brown   4
```

```

7 Andrew    Logan    red          2
8 Andrew    Logan    green        9
9 Andrew    Logan    orange       14
10 Andrew   Logan    yellow      4
# i 818 more rows

```

... and now we flip it back to having a column for each color:

```

candy_long |>
  pivot_wider(
    names_from = color,
    values_from = count_candies
  )

```

- ① Within `pivot_wider()` we need to tell it from which column to pull the new column headers.  
i.e., where do we get the “`names_from`”?
- ② The second thing it needs to know which variable has the data to fill these new columns.  
i.e., where do we get the “`values_from`”?

```

# A tibble: 138 x 8
  first_name last_name   red green orange yellow blue brown
  <chr>     <chr>     <dbl> <dbl>  <dbl>  <dbl> <dbl> <dbl>
1 Christian McDonald     2     17     11      4     16      4
2 Andrew    Logan        2      9     14      4     17      9
3 Gabby     Ybarra       4     11     14      0     19      7
4 Veronica  Apodaca     1     17     16      2     13      8
5 Cristela Jones         3     19     14      3     13      3
6 Marina    Garcia       7     11     13      1     14      6
7 Samuel    Stark        3      9     10      5     18      8
8 Kevin     Malcolm Jr   4     15     13      6     14      3
9 Alexa     Haverlah     5     15      6      2     21      5
10 Zacharia Washington    1     15     14      1     20      6
# i 128 more rows

```

Let's practice some more.

### 9.8.2 Choosing what to flip

Lets create a different look at our data so that we can spin it around in different ways.

When I buy candy for the class, we buy big boxes with many bags in it. There is a “box code” stamped on each package that denotes which box it was packaged in. We'll re-summarize

our data so we get the color averages within each box. I do some other cleanup here, too, to remove bags that came from the wild.

1. Create a new section and note you are pivoting wider
2. Add a new chunk with the code below and run it

See annotations below the code to explain what is going on, in case you are interested.

```
candy_box_color <- candy_raw |>  
  filter(box_code != "130FXWAC0336", box_code != "1524976SE") |>  
  select(!c(timestamp, candy_type)) |>  
  pivot_longer(cols = red:brown, names_to = "candy_color", values_to = "candy_count") |> ④  
  group_by(box_code, candy_color) |>  
  summarise(avg_candies = mean(candy_count) |> round(1)) ⑥
```

candy\_box\_color

- ① We create a new object to put everything in, then go back to the original data to get the box code.
- ② Removing some rows where there were only one or two bags. Not shown here is the research to find their values.
- ③ Removing the `timestamp` and `candy_type` variables.
- ④ Pivoting the data longer so we can group and summarize.
- ⑤ Group by the box and color before we ...
- ⑥ Summarize to get the average number of candies in each bag, within each box. We also round the result.

```
# A tibble: 36 x 3  
# Groups:   box_code [6]  
  box_code   candy_color avg_candies  
  <chr>     <chr>          <dbl>  
1 140BSCLV02 blue            14.7  
2 140BSCLV02 brown           6.3  
3 140BSCLV02 green           13.7  
4 140BSCLV02 orange          13.1  
5 140BSCLV02 red              3.1  
6 140BSCLV02 yellow          3.6  
7 227GTCLV02 blue            12.3  
8 227GTCLV02 brown           6.2  
9 227GTCLV02 green           13  
10 227GTCLV02 orange          7.1  
# i 26 more rows
```

OK, now you have a new object called `candy_box_color`.

Let's pivot this data to shows a column for each box, which means that is where we draw our "names\_from":

1. Add a note that you'll pivot our new data
2. Add the code below and run it

```
candy_box_color |>
  pivot_wider(names_from = box_code, values_from = avg_candies)
```

```
# A tibble: 6 x 7
  candy_color `140BSCLV02` `227GTCLV02` `233DRCIV02` `233DRCLV02` `250ARCLV02` 
  <chr>          <dbl>      <dbl>      <dbl>      <dbl>      <dbl>    
1 blue            14.7       12.3       10        7.7      10.1      
2 brown           6.3        6.2        5         6.8      6.6        
3 green           13.7       13         7        10.4      8.7        
4 orange          13.1       7.1        17       15.5      9.4        
5 red              3.1        6.6        3         6.8      8.7        
6 yellow          3.6        10.6       15       10.8      13.8      
# i 1 more variable: `259ARCLV00` <dbl>
```

### 9.8.3 Pivot wider on your own

Now I want you do apply the same `pivot_wider()` function to that same `candy_box_color` data, but to have a column for each color and a row for each box.

1. Start a new section and note this is pivot\_wider on your own.
2. Start with the `candy_box_color` data, and then ...
3. Use `pivot_wider()` to make the data shaped like this

| box_code   | blue | brown | green | orange | red | yellow |
|------------|------|-------|-------|--------|-----|--------|
| [box name] | #    | #     | #     | #      | #   | #      |

## 9.9 Bonus questions

More opportunities for bonus points on this assignment. These aren't plots, just data wrangling to find answers.

### **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 the most and one for the 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 Renders.
2. Publish it to Quarto Publish. Include the published link at the top of your notebook.
3. Stuff your project and turn it into the Candy 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 them to transpose our data.
- 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.

## **Part IV**

## **Bind & Join**

# 10 Denied Cleaning

This is the first of a two-chapter project. It is by Prof. McDonald, using macOS.

So far all of our lessons have come from a single data source, but that's not always the case in the real world. With this lesson we'll learn how to import multiple files at once and "bind" them, and to use "join" to enhance that data from a lookup table.

## 10.1 Goals of the chapter

- Import multiple files at once and bind them with `map()`
- Join data frames with `inner_join()`
- Use `str_remove()` to clean data
- Introduce `if_else()` for categorization

We'll use the results of this chapter in our next one.

## 10.2 About the story: Denied

In 2016, the Houston Chronicle published a multi-part series called Denied that outlined how a Texas Education Agency policy started in 2004 could force an audit of schools that have more than 8.5% of their students in special education programs. The story was a Pulitzer Prize finalist. Here's an excerpt:

Over a decade ago, the officials arbitrarily decided what percentage of students should get special education services — 8.5% — and since then they have forced school districts to comply by strictly auditing those serving too many kids.

Their efforts, which started in 2004 but have never been publicly announced or explained, have saved the Texas Education Agency billions of dollars but denied vital supports to children with autism, attention deficit hyperactivity disorder, dyslexia, epilepsy, mental illnesses, speech impediments, traumatic brain injuries, even blindness and deafness, a Houston Chronicle investigation has found.

More than a dozen teachers and administrators from across the state told the Chronicle they have delayed or denied special education to disabled students in order to stay below the 8.5 percent benchmark. They revealed a variety of methods, from putting kids into a cheaper alternative program known as “Section 504” to persuading parents to pull their children out of public school altogether.

Following the Chronicle’s reporting (along with other news orgs), the Texas Legislature in 2017 [unanimously banned](#) using a target or benchmark on how many students a district or charter school enrolls in special education.

We want to look into the result of this reporting based on three things:

- Has the percentage of special education students in Texas changed since the benchmarking policy was dropped?
- How many districts were above that arbitrary 8.5% benchmark before and after the changes?
- How have local districts changed?

To prepare for this:

1. Read [Part 1](#) of the original Denied series, at least to the heading “A special child.” Pay attention to parts about the “Performance-Based Monitoring Analysis System.”
2. Read [About this series](#)
3. Read [this followup](#) about the legislative changes.

## 10.3 About the data

Each year, the Texas Education Agency publishes the percentage of students in special education as part of their [Texas Academic Performance Reports](#). We can download a file that has the percentages for each district in the state.

There are some challenges, though:

- We have to download each year individually. There are nine years of data.
- There are no district names in the files, only a district ID. We can get a reference file, though.
- There are some differences in formatting for some files.

I will save you the hassle of going through the TAPR database to find and download the individual files by providing a starter project.

### **10.3.1 Set up your project**

With this project you'll start with a copy I have prepared for you. How you do that differs a little depending on the RStudio platform you are using.

#### **10.3.1.1 If you are using RStudio Desktop**

You will download a project that is already set up for you and then open it.

**Use the link below to download the project.**

[Download rwmdir-sped-template-main.zip](#)

1. Find that file on your computer and uncompress it.
2. Rename the project folder to **yourname-sped but use your name**.
3. Move the project folder to your **rwd** folder or wherever you've been saving your class projects.
4. In RStudio, choose File > New Project. Choose **EXISTING Directory** at the next step and then find the folder you saved. Use that to create your project.

#### **10.3.1.2 If you are a posit.cloud user**

1. From your posit.cloud account, go to [this shared project](#)
2. Click **Save a permanent copy** so you have your own version.
3. Rename the project **yourname-sped but use your name**.

## **10.4 The DSTUD data**

To track how percentages of special education students have changed over time, we will use data observations from the Texas Education Agency's Texas Academic Performance Reports. The TEA has a portal where you can downloaded bulk data by district or campus for each academic year.

While **our downloaded starter project has all the data we need**, you should go through this process for one of the years just so you see how it works.

The process goes like this:

- Start at the [Texas Academic Performance Reports](#) page
- Choose the school year of interest (choose 2012-13 to just look at the process)
- Find the **Data Download** page

- Choose the **TAPR Data in Excel (Rates Only)** option
- Click on **District Download**. Click **Continue** to move on.
- Choose the **Student Information** data set and choose **Continue**.
  - Choose the **Comma-Delimited (csv)** format
  - Choose **Student Enrollment: Counts/Percents**
  - Also choose **Students by Instructional Programs** (or **Student Membership by Program** or **Student Enrollment by Program** depending on the year. See note below.)
  - Click **Download**.

This downloads a file called `DSTUD.csv` or `DSTUD.dat`. As I downloaded them, I changed the name to include the year and `.csv` extension and put them in the `data-raw/` folder.

On the download page, there is a link to the data dictionary for the file called [Student Information Reference](#). As you'll see next, it proves important to understand the file as the data within reports can vary from year to year. There is a Glossary for each year, but in our case I'll guide you along.

#### 10.4.1 How the files differ

Let's look at one of those files. This is the first six lines of the 2012-13 version:

```
read_csv("data-raw/DSTUD_13.csv") |> head()

# A tibble: 6 x 34
  DISTRICT DPETALLC DPETASIC DPETASIP DPETBILC DPETBILP DPETBLAC DPETBLAP
  <chr>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 001902      595       4     0.7      5     0.8     34     5.7
2 001903     1236      10     0.8     16     1.3     54     4.4
3 001904      751       2     0.3     14     1.9     79    10.5
4 001906      406       2     0.5      4      1     24     5.9
5 001907     3288      33      1     400    12.2    924    28.1
6 001908     1619      10     0.6     73     4.5    268    16.6
# i 26 more variables: DPETDISC <dbl>, DPETDISP <chr>, DPETECOC <dbl>,
#   DPETECOP <dbl>, DPETGIFC <dbl>, DPETGIFP <dbl>, DPETHISC <dbl>,
#   DPETHISP <dbl>, DPETINDC <dbl>, DPETINDP <dbl>, DPETLEPC <dbl>,
#   DPETLEPP <dbl>, DPETNEDC <dbl>, DPETNEDP <dbl>, DPETPCIC <dbl>,
#   DPETPCIP <dbl>, DPETRSKC <dbl>, DPETRSKP <dbl>, DPETSPEC <dbl>,
#   DPETSPEP <dbl>, DPETTWOC <dbl>, DPETTWOP <dbl>, DPETVOCC <dbl>,
#   DPETVOCP <dbl>, DPETWHIC <dbl>, DPETWHIP <dbl>
```

As you can see by looking at that data, the [District Student Information Reference](#) and [Glossary](#) are pretty important so we can understand what these columns are.

The columns we want are:

- DISTRICT – District Number
- DPETALLC – District 2013 Student: All Students Count
- DPETSPEC – District 2013 Student: Special Ed Count
- DPETSPEP – District 2013 Student: Special Ed Percent

Luckily for us, those specific column names are the same in each of the nine data files.

Note how the DISTRICT values that start with zeros like this: 001902. We have to pay attention to not drop those leading zeros, and they are important later. Using `read_csv()` (instead of `read.csv()`) helps us preserve the zeros.

What is missing from this data? Well, the **name** of the district is missing, which is important. We'll solve in a bit.

In addition, the data structure of these files changed starting with the 2021-22 file:

```
read_csv("data-raw/DSTUD_22.csv") |> head()
```

```
# A tibble: 6 x 69
  DISTRICT DPEMSPEC DPEMSPEP DPEMSPET DPET504C DPET504P DPETALLC DPETASIC
  <chr>     <chr>    <chr>    <chr>      <dbl>    <dbl>    <dbl>    <dbl>
1 '001902'  15       16.3     92        52       9.1      574      3
2 '001903'  24       16.1     149       79       6.9      1150     2
3 '001904'  23       18.7     123       69       8.5      808      8
4 '001906'  5        13.5     37        31       9.1      342      1
5 '001907'  95      19.8     480      207      6.2      3360     24
6 '001908'  49       21.7     226      103      7.7      1332     6
# i 61 more variables: DPETASIP <dbl>, DPETATTC <dbl>, DPETATTB <dbl>,
#   DPETATTP <dbl>, DPETBILC <dbl>, DPETBILP <dbl>, DPETBLAC <dbl>,
#   DPETBLAP <dbl>, DPETDISC <dbl>, DPETDISP <dbl>, DPETDSLC <dbl>,
#   DPETDSLP <dbl>, DPETECOC <dbl>, DPETECOP <dbl>, DPETFEMC <dbl>,
#   DPETFEMP <dbl>, DPETFOSC <dbl>, DPETFOSP <dbl>, DPETG9XC <dbl>,
#   DPETGIFC <dbl>, DPETGIFP <dbl>, DPETHISC <dbl>, DPETHISP <dbl>,
#   DPETHOMC <dbl>, DPETHOMP <dbl>, DPETIMMC <dbl>, DPETIMMP <dbl>, ...
```

In the newer files the DISTRICT observations have a ' at the beginning: '001902. This is added to help preserve those leading zeros, but we will need to remove the ' so the values are the same structure as the other data files.

Also note there are 64 columns in this data vs the 34 in the 2013 file. The TEA has added different measurements through the years. That's OK, because we still only need the three columns of DISTRICT, DPETALLC, DPETSPEC and DPETSPEP for this analysis, so we'll specify only those columns as we import the data.

### 10.4.2 Importing multiple files at once

In our `raw-data/` folder we have 10 years of data from the 2012-2013 school year to the 2020-2022 school year. If all of our files were the same configuration (the same variables and data types) then we could just use `read_csv()` and feed it a list of files. Our files aren't the same, though, as some have additional columns.

The “long” way to do this would be importing each file individually, selecting the columns we need and then saving them as objects. We could then stack them on top of each other with the `bind_rows()` function.

```
data_01 |> bind_rows(data_02, data_03)
```

That would bind three tibbles of data if they were all the same.

But we can use some libraries available in tidyverse – purrr and readr – to import and bind together all the DSTUD files at one time after selecting just the variables we need from each file so they are the same.

We do this in three steps:

- We'll use `list.files()` to create a vector of the files we want
- We'll use `map()` to apply the `read_csv` function to each file and create a list tibbles
- We'll use `list_rbind()` to combine those tibbles into a single tibble

There are a couple of other functions we'll tuck in there along the way to help us.

#### 10.4.2.1 Building our files list

We'll use a function called `list.files()` to create a vector of the files we want.

1. If you haven't already, open the cleaning notebook in the project.
2. Run all the setup code.
3. Add a Markdown headline noting we are working on DSTUD data
4. Add text noting we are building our vector of files to import.
5. Run the code.

I explain the code below.

```
dstud_files <- list.files(          ①  
  "data-raw",                      ②  
  pattern = "DSTUD",                ③  
  full.names = TRUE                 ④  
)  
  
dstud_files                         ⑤
```

- ① The first line starts with the new object we are creating to hold our vector of data. It ends with our function, `list.files()`, which lists files in a directory as a vector.
- ② The first argument of `list.files()` is the path to where you want to look. We want to look in the `data-raw` folder. Without this argument, it would look in our main project folder.
- ③ The second optional argument is a regular expression (or regexp) search query to capture specific files based on the file name. We're looking for the text “DSTUD” in the file names so we don't capture the DREF file (which we'll deal with later).
- ④ The `full.names = TRUE` argument indicates that we want to keep not just the names of the files, but the full path to where the files are, which is what we need.
- ⑤ We're just printing out the resulting object to make sure we got what we wanted.

```
[1] "data-raw/DSTUD_13.csv" "data-raw/DSTUD_14.csv" "data-raw/DSTUD_15.csv"  
[4] "data-raw/DSTUD_16.csv" "data-raw/DSTUD_17.csv" "data-raw/DSTUD_18.csv"  
[7] "data-raw/DSTUD_19.csv" "data-raw/DSTUD_20.csv" "data-raw/DSTUD_21.csv"  
[10] "data-raw/DSTUD_22.csv"
```

#### 💡 About regular expressions

Regular expressions are a pattern-matching syntax used in many programming languages. Above we searched for a simple word, but regex is much more powerful than that. You can learn more about regular expressions in this [NICAR lesson](#) and how to use them in R in the [Regular expressions](#) chapter in R for Data Science.

Now we have an object that is a vector of all the file names. We'll loop through this list to import all the files.

#### 10.4.2.2 Map, import and bind

In the next section we'll add the file names to our vector to help us later, then use `map()` to walk through the list and apply `read_csv()` function to all our files. Then we'll bind together with `list_rbind()` and finish out normalizing the names.

There is a LOT going on in this bit, so note the code annotations for details.

1. Add some text noting that you are importing the DSTUD files
2. Add the code block below and run it.
3. Be sure to read through the explanations

```
dstud_raw <- dstud_files |>          ①
  set_names(basename) |>          ②
  map(          ③
    read_csv,          ④
    col_select = c(DISTRICT, DPETALLC, DPETSPEC, DPETSPEP)          ⑤
  ) |>          ⑥
  list_rbind(names_to = "source") |>
  clean_names()          ⑦

dstud_raw |> nrow()          ⑧
dstud_raw |> head()          ⑨
dstud_raw |> tail()          ⑩
```

- ① We create a new object to fill, then start with our list of file names.
- ② This line adds a name to each item in the `dstud_files` vector with the name of the file it came from. We'll use this later when we combine the files to know where the data came from.
- ③ We pipe into `map`, which will work on each file in turn, applying the functions and arguments inside this function.
- ④ The first argument is the function we are applying. We are using `read_csv()` since we are importing csv files.
- ⑤ This `col_select =` is an optional argument for `read_csv` that allows us to choose which columns we want as we read them in. After this argument we close the `map()` function, which results in a “list” of data frames, which we then pipe into ...
- ⑥ `list_rbind()` is a function that combines data frames from a list. We give this function the argument `names_to = "source"` to add a variable called `source` to each row of data with the name of the file it came from. This is why we added `set_names()` earlier because without it we wouldn't know which row came from which file. With this name, we can create a year column later.
- ⑦ We clean the names after importing them.
- ⑧ Here I'm just printing the number of rows in our data as a check that everything worked.
- ⑨ Here we are peeking at the top of the resulting object.
- ⑩ And then we peek at the bottom of the object.

```
[1] 12098
# A tibble: 6 x 5
```

```

source      district dpetallc dpetspec dpetspep
<chr>      <chr>    <dbl>     <dbl>     <dbl>
1 DSTUD_13.csv 001902      595      73    12.3
2 DSTUD_13.csv 001903     1236     113     9.1
3 DSTUD_13.csv 001904      751      81    10.8
4 DSTUD_13.csv 001906      406      45    11.1
5 DSTUD_13.csv 001907     3288     252     7.7
6 DSTUD_13.csv 001908     1619     151     9.3
# A tibble: 6 x 5
source      district dpetallc dpetspec dpetspep
<chr>      <chr>    <dbl>     <dbl>     <dbl>
1 DSTUD_22.csv '252901    2293     356    15.5
2 DSTUD_22.csv '252902    212      36     17
3 DSTUD_22.csv '252903    696     113    16.2
4 DSTUD_22.csv '253901    3284     435    13.2
5 DSTUD_22.csv '254901    1779     232     13
6 DSTUD_22.csv '254902    484      64    13.2

```

 The `col_select` argument vs `select()`

This `col_select =` argument we use above also works with a regular `read_csv()` function. It provides the same result as using `read_csv()` to import all the columns and then piping into a `select()` function to list the columns you want to keep or drop. It's yet another example showing multiple ways to accomplish the same result with R. Up to now I've preferred to use `read_csv() |> select()` because that allows you to see everything inside the data before choosing what to do with it. In our case above being able to pick just these few columns we need makes this much easier to deal with since the individual files have differences beyond some common columns.

### 10.4.3 Clean up DSTUD file

Look closely at the `district` column and notice the difference at the top and bottom of the file. See where the DSTUD\_22 files have a ' at the beginning? We need to take those out.

We also need to create a `year` variable from our `source` variable with the file name. We can pluck the numbers out of the file name and combine with text to make the year "2013", etc.

1. Add some text that you are cleaning the file to remove the apostrophe from `district` and to build a year variable out of `source`.
2. Copy the code below and run it.
3. Make sure you read the explanations so understand what is happening.

```

dstud <- dstud_raw |>
  mutate(
    district = str_remove(district, "'"),
    year = str_c("20", str_sub(source, 7, 8))
  ) |>
  select(!source)

dstud |> head()                                ⑤
dstud |> tail()                               ⑥

```

- ① We create our new object and fill with starting with the `dstud_raw` data we imported above.
- ② Within a `mutate` function, our first line uses `str_remove()` to pull out the apostrophe from the `district` variable. We are just reassigning it back to itself.
- ③ The second `mutate` operation is more complicated. We are setting `year` to equal a “string combination” (`the str_c()` function) that pieces together the text “20” with characters from the `source` variable based on their position, which happen to be the numbers of the year. To do that we use `str_sub()` where the first argument is variable to look at (`source`), the starting position (the 7th character) and the ending position (the 8th one). This only works because all the file names are structured the same way.
- ④ Here we remove the `source` column since we no longer need it.
- ⑤ We peek at the top of the file ...
- ⑥ ... and the bottom of the file to check our results.

```

# A tibble: 6 x 5
  district dpetallc dpetspec dpetspep year
  <chr>     <dbl>     <dbl>     <dbl> <chr>
1 001902      595       73     12.3 2013
2 001903     1236      113      9.1 2013
3 001904      751       81     10.8 2013
4 001906      406       45     11.1 2013
5 001907     3288      252      7.7 2013
6 001908     1619      151      9.3 2013

# A tibble: 6 x 5
  district dpetallc dpetspec dpetspep year
  <chr>     <dbl>     <dbl>     <dbl> <chr>
1 252901     2293      356     15.5 2022
2 252902      212       36      17   2022
3 252903      696       113     16.2 2022
4 253901     3284      435     13.2 2022
5 254901     1779      232      13   2022
6 254902      484       64     13.2 2022

```

OK, that seems like a lot, but there was a lot to learn there. This new data frame `dstud` is ready for us to use a little later.

## 10.5 The DREF data

We've noted that the `dstud` files don't have a **district name** within them, so we don't know what this district id means. Within the TAPR data download tool, there is another file called **District Reference** that has the district ID, names and other information we need. I downloaded the 2022 version as `DREF_22.csv`. Let's look at it:

```
read_csv("data-raw/DREF_22.csv") |> head()
```

```
# A tibble: 6 x 11
  DISTRICT CNTYNAME COUNTY DAD_POST DFLALTED DFLCHART DISTNAME D_RATING OUTCOME
  <chr>     <chr>    <chr>    <dbl> <chr>    <chr>    <chr>    <chr>    <chr>
1 '001902  ANDERSON '001        1 N       N      CAYUGA I~ A      Meets ~
2 '001903  ANDERSON '001        0 N       N      ELKHART ~ A      Meets ~
3 '001904  ANDERSON '001        0 N       N      FRANKSTO~ A      Meets ~
4 '001906  ANDERSON '001        1 N       N      NECHES I~ A      Meets ~
5 '001907  ANDERSON '001        0 N       N      PALESTIN~ B      Needs ~
6 '001908  ANDERSON '001        0 N       N      WESTWOOD~ B      Needs ~
# i 2 more variables: REGION <chr>, asvab_status <chr>
```

You'll see this file also has ' at the beginning of ID columns like `DISTRICT`, `COUNTY` etc. to preserve leading zeros.

Look at the [District Reference](#) documentation to understand the variables.

We want several columns out of this file:

- `DISTRICT` – We'll have to remove the ' that preserves the leading zeros.
- `CNTYNAME` – So we can focus on schools in different areas of the state
- `DISTNAME` – Which has our district name.
- `DFLCHART` – District 2022 Flag - Charter Operator (Y/N)
- `DFLALTED` – District 2022 Flag - Rated under AEA Procedures (Y/N)

We can use the `DISTRICT` variable to “join” the district name to all the other data.

We need the `DFLCHART` and `DFLALTED` fields so we can **filter out charter and alternative education schools**. For this analysis, we only want “traditional” public schools.

### 10.5.1 Import reference data

1. Add a new Markdown headline that you are working on the reference data.
2. Add text that you'll use `read_csv` to pull in the variables you need.
3. Add the code below, run it and read over the explanations.

```
dref_raw <- read_csv(  
  "data/raw/DREF_22.csv",  
  col_select = c(  
    DISTRICT,  
    CNTYNAME,  
    DISTNAME,  
    DFLCHART,  
    DFLALTED  
  )  
) |>  
clean_names()  
  
dref_raw |> head()
```

- ① The path to the files.  
② We use the `col_select =` argument to keep just the columns we want.

```
# A tibble: 6 x 5  
  district cntyname distname      dflchart dflaltered  
  <chr>     <chr>    <chr>      <chr>    <chr>  
1 '001902' ANDERSON CAYUGA ISD    N        N  
2 '001903' ANDERSON ELKHART ISD   N        N  
3 '001904' ANDERSON FRANKSTON ISD N        N  
4 '001906' ANDERSON NECHES ISD   N        N  
5 '001907' ANDERSON PALESTINE ISD N        N  
6 '001908' ANDERSON WESTWOOD ISD  N        N
```

### 10.5.2 Clean up DREF file

Now that we have our reference file, we need to clean up the apostrophes so we can join this data to our student data.

1. Add some text that you will clean up the apostrophes.
2. Add the code below, run it. Note the cleaning is similar to what we did with DSTUD.

```

dref <- dref_raw |>
  mutate(district = str_remove(district, ""))
dref |> head()

# A tibble: 6 x 5
  district cntyname distname      dflchart dflaltd
  <chr>     <chr>    <chr>        <chr>    <chr>
1 001902    ANDERSON CAYUGA ISD    N         N
2 001903    ANDERSON ELKHART ISD   N         N
3 001904    ANDERSON FRANKSTON ISD N         N
4 001906    ANDERSON NECHES ISD   N         N
5 001907    ANDERSON PALESTINE ISD N         N
6 001908    ANDERSON WESTWOOD ISD  N         N

```

Now we add this data to our `dstud` object.

## 10.6 About joins

OK, before we go further we need to talk about joining data. It's one of those [Basic Data Journalism Functions](#) ...

<https://vimeo.com/435910338?share=copy>

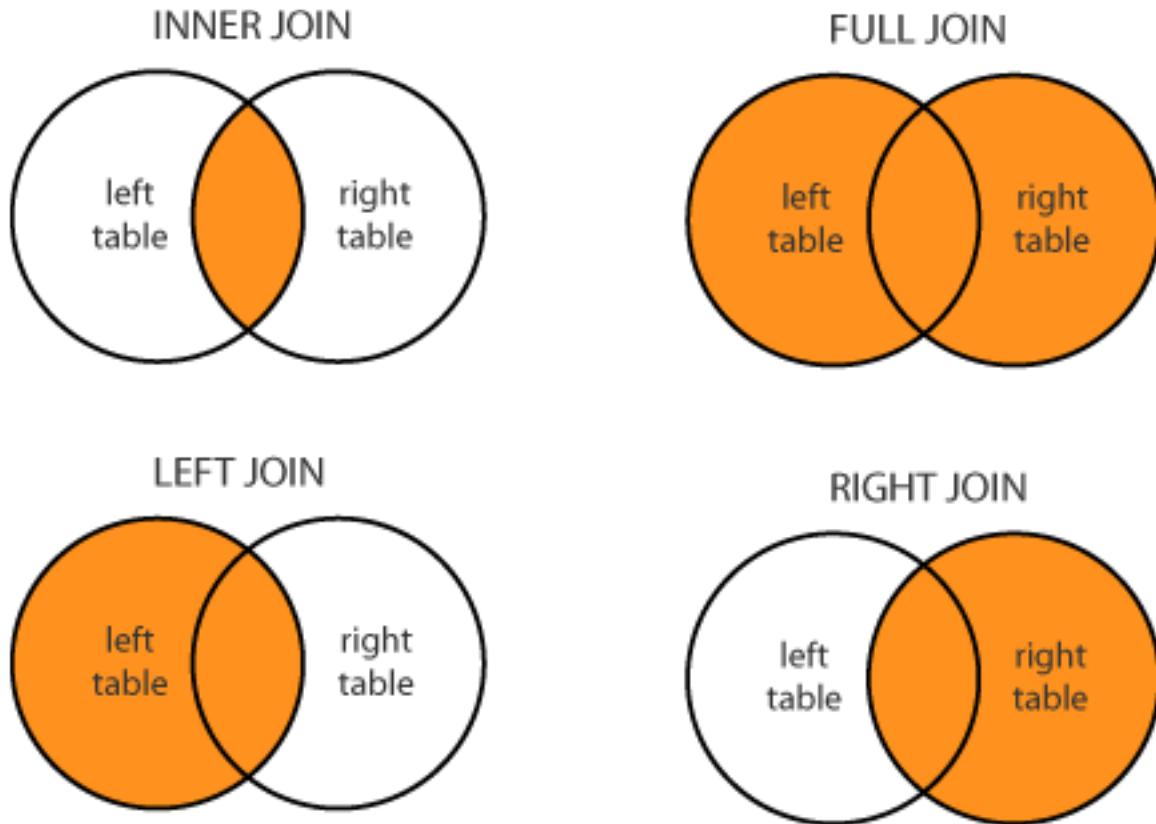
What joins do is match rows from two data sets that have a column of common values, like an ID or county name. (The `district` ID column in our case). Columns from the second data set will be added based on where the ID's match.

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). How much data you end up with depends on the “direction” of the join.

- An `inner_join` puts together columns from both tables where there are matching rows. If there are records in either table where the IDs don't match, they are dropped.
- A `left_join` will keep ALL the rows of your first table, then bring over columns from the second table where the IDs match. If there isn't a match in the second table, then new values will be blank in the new columns.
- A `right_join` is the opposite of that: You keep ALL the rows of the second table, but bring over only matching rows from the first.
- A `full_join` keeps all rows and columns from both tables, joining rows when they match.

Here are two common ways to think of this visually.

In the image below, The orange represents the data that remains after the join.



This next visual shows this as tables where only two rows “match” so you can see how non-matches are handled (the lighter color represents blank values). The functions listed there are the tidyverse versions of each join type.

`left_join()`



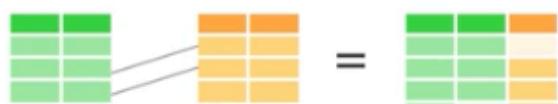
`right_join()`



`inner_join()`



`full_join()`



### 10.6.1 Joining our reference table

In our case we will start with the `dref` data and then use an `inner_join` to add all the yearly data values. We're doing it in this order so the `dref` values are listed first in our resulting table.

1. Start a new Markdown section and note we are joining the reference data
2. Add the chunk below and run it

```
sped_joined <- dref |>  
  inner_join(dstud, by = "district")  
  
sped_joined |> head()
```

- ① We are creating a new bucket `sped_joined` to save our data. We start with `dref` so those fields will be listed first in our result.
- ② We then pipe into `inner_join()` listing the `dstud` object, which will attach our `dref` data to our merged data when the ID matches in the `district` variable. The `by = "district"` argument ensures that we are matching based on the `district` column in both data sets.

| # A tibble: 6 x 9 |          |          |          |          |            |          |          |          |       |       |
|-------------------|----------|----------|----------|----------|------------|----------|----------|----------|-------|-------|
|                   | district | cntyname | distname | dflchart | dflaltered | dpetallc | dpetspec | dpetspep | year  |       |
|                   | <chr>    | <chr>    | <chr>    | <chr>    | <chr>      | <dbl>    | <dbl>    | <dbl>    | <chr> | <chr> |
| 1                 | 001902   | ANDERSON | CAYUGA   | I~ N     | N          | 595      | 73       | 12.3     | 2013  |       |
| 2                 | 001902   | ANDERSON | CAYUGA   | I~ N     | N          | 553      | 76       | 13.7     | 2014  |       |
| 3                 | 001902   | ANDERSON | CAYUGA   | I~ N     | N          | 577      | 76       | 13.2     | 2015  |       |
| 4                 | 001902   | ANDERSON | CAYUGA   | I~ N     | N          | 568      | 78       | 13.7     | 2016  |       |
| 5                 | 001902   | ANDERSON | CAYUGA   | I~ N     | N          | 576      | 82       | 14.2     | 2017  |       |
| 6                 | 001902   | ANDERSON | CAYUGA   | I~ N     | N          | 575      | 83       | 14.4     | 2018  |       |

We could've left out the `by` = argument and R would match columns of the same name, but it is best practice to specify your joining columns so it is clear what is happening. You wouldn't want to be surprised to join by other columns of the same name. If you wanted to specify join columns of different names it would look like this:

```
# don't use this ... it is just an example  
df1 |> inner_join(df2, by = c("df1_id" = "df2_id"))
```

You should also `glimpse()` your new data so you can see all the columns have been added.

1. Add text that you are peeking at all the columns.
  2. Add the code below and run it

```
sped_joined |> glimpse()
```

```
Rows: 11,882
Columns: 9
# district <chr> "001902", "001902", "001902", "001902", "001902", "001902", "~"
# cntyname <chr> "ANDERSON", "ANDERSON", "ANDERSON", "ANDERSON", "ANDERSON", "~~"
# distname <chr> "CAYUGA ISD", "CAYUGA ISD", "CAYUGA ISD", "CAYUGA ISD", "CAYU~"
# dflchart <chr> "N", "~~"
# dfalalted <chr> "N", "~~"
# dpetallc <dbl> 595, 553, 577, 568, 576, 575, 564, 557, 535, 574, 1236, 1207, ~
# dpetspec <dbl> 73, 76, 76, 78, 82, 83, 84, 82, 78, 84, 113, 107, 126, 144, 1~
# dpetspep <dbl> 12.3, 13.7, 13.2, 13.7, 14.2, 14.4, 14.9, 14.7, 14.6, 14.6, 9~
# year      <chr> "2013", "2014", "2015", "2016", "2017", "2018", "2019", "2020~
```

There are now **11882** rows in our joined data, fewer than what was in the original merged file because some districts (mostly charters) have closed and were not in our reference file. We are comparing only districts that have been open during this time period. For that matter, we don't want charter or alternative education districts at all, so we'll drop those next.

## 10.7 Some cleanup: filter and select

Filtering and selecting data is something we've done time and again, so you should be able to do this on your own.

You will next remove the charter and alternative education districts. This is a judgement call on our part to focus on just traditional public schools. We can always come back later and change if needed.

You'll also remove and rename columns to make them more descriptive.

1. Start a new markdown section and note you are cleaning up your data.
2. Create an R chunk and start with the `sped_joined` and then do all these things ...
3. Use `filter()` to keep rows where:
  - the `dflaltd` field is “N”
  - AND the `dflchart` field is “N”
4. Use `select()` to:
  - remove (or not include) the `dflaltd` and `dflchart` columns. (You can only do this AFTER you filter with them!)
5. Use `select()` or `rename()` to rename the following columns:
  - change `dpetallc` to `all_count`
  - change `dpetspec` to `sped_count`
  - change `dpetspep` to `sped_percent`
6. Make sure all your changes are saved into a new data frame called `sped_cleaned`.

I really, really suggest you don't try to write that all at once. Build it one line at a time so you can see the result as you build your code.

```
sped_cleaned <- sped_joined |>
  filter(dflaltd == "N" & dflchart == "N") |>
  select(
    district,
    distname,
    cntyname,
    year,
    all_count = dpetallc,
    sped_count = dpetspec,
    sped_percent = dptspep
  )
```

You should end up with **10204** rows and **7** variables.

## 10.8 Create an audit benchmark column

Part of this story is to note when a district is above the “8.5%” benchmark the TEA penalized them in their audit calculations. It would be useful to have a column that noted if a district was above or below that threshold so we could plot districts based on that flag. We’ll create this new column and introduce the logic of `if_else()`.

OK, our data looks like this:

```
sped_cleaned |> head()
```

```
# A tibble: 6 x 7
  district distname cntyname year all_count sped_count sped_percent
  <chr>     <chr>    <chr>   <chr>     <dbl>      <dbl>        <dbl>
1 001902    CAYUGA ISD ANDERSON 2013      595       73        12.3
2 001902    CAYUGA ISD ANDERSON 2014      553       76        13.7
3 001902    CAYUGA ISD ANDERSON 2015      577       76        13.2
4 001902    CAYUGA ISD ANDERSON 2016      568       78        13.7
5 001902    CAYUGA ISD ANDERSON 2017      576       82        14.2
6 001902    CAYUGA ISD ANDERSON 2018      575       83        14.4
```

We want to add a column called `audit_flag` that says **ABOVE** if the `sped_percent` is above “8.5”, and says **BELOW** if it isn’t. This is a simple true/false condition that is perfect for the `if_else()` function.

1. Add a new Markdown section and note that you are adding an audit flag column
2. Create an r chunk that and run it and I’ll explain after.

```
sped_flag <- sped_cleaned |>
  mutate(audit_flag = if_else(
    sped_percent > 8.5,
    "ABOVE",
    "BELOW"
  ))
```

(1)  
(2)  
(3)  
(4)  
(5)

```
# this pulls 10 random rows so I can check results
sped_flag |>
  sample_n(10) |>
  select(distname, sped_percent, audit_flag)
```

(6)  
(7)

- ① We’re making a new data frame called `sped_flag` and then starting with `sped_cleaned`.

- ② We use `mutate()` to create a new column and we name it `audit_flag`. We set the value of `audit_flag` to be the result of an `if_else()` function.
- ③ The `if_else` function takes three arguments and the first is a condition test (`sped_percent > 8.5` in our case). We are looking inside the `sped_percent` column to see if it is greater than “8.5”.
- ④ The second argument is what we want the result to be if the condition is true (the text “`ABOVE`” in our case.)
- ⑤ And the third argument is the result if the condition is NOT true (the text “`BELOW`” in our case).
- ⑥ Lastly we print out the new data `sped_cleaned` and pipe it into `sample_n()` which gives us a number of random rows from the data. I do this because the top of the data was all `TRUE` so I couldn’t see if the `mutate` worked properly or not. (Always check your calculations!!)
- ⑦ I’m using `select()` here to grab just the columns of interest for the check.

```
# A tibble: 10 x 3
  distname      sped_percent audit_flag
  <chr>          <dbl>     <chr>
1 CENTER ISD      8.4     BELOW
2 PENELOPE ISD    11.5    ABOVE
3 SOUTHSIDE ISD   10.4    ABOVE
4 DEL VALLE ISD   15.7    ABOVE
5 KRUM ISD        9.4     ABOVE
6 TOLAR ISD       6.8     BELOW
7 LIBERTY-EYLAU ISD 11.6    ABOVE
8 COTTON CENTER ISD 8.8     ABOVE
9 HALLSBURG ISD   8.1     BELOW
10 PONDER ISD      10      ABOVE
```

Be sure to look through the result to check the new `audit_flag` has values as you expect.

## 10.9 Export the data

This is something you’ve done a number of times as well, so I leave you to you:

1. Make a new section and note you are exporting the data
2. Export your `sped_flag` data using `write_rds()` and save it in your `data-processed` folder.

In the next chapter we’ll build an analysis notebook to find our answers!

## 10.10 New functions we used

- `list.files()` makes a list of file names in a folder. You can use the `pattern = argument` to search for specific files to include based on a regular expression.
- `map()` Loops through a vector or list and applies a function to each item. We used it with `read_csv()` to import multiple files and put them into a list.
- `list_rbind()` is a function to combine elements into a data frame by row-binding them. In other words, it stacks them on top of each other into one thing.
- `set_names()` provides a name to each element in a vector. We used it to add the “basename” of each file.
- `str_c()` is used to combine text.
- `str_sub()` allows us to extract part of a string based on its position.
- We used `inner_join()` to add columns from one data frame to another one based on a common key in both objects.
- `if_else()` provides specified results based on a true/false condition.

# 11 Denied Analysis

This is the second of of a two-chapter project. Now that we have assembled and cleaned our data, we need to answer these questions in our followup to the [Denied](#) series:

- Has the percentage of special education students in Texas changed since the benchmarking policy was dropped?
- How many districts were above that arbitrary 8.5% benchmark before and after the changes?
- How have local districts changed?

The idea here is to answer those questions through plotting. We'll use this chapter to walk through the process of using plots to explore and find answers in your data.

## 11.1 Goals of this chapter

- Introduce `datatables()` from the [DT package](#)
- Practice pivots to prepare data for plotting
- Practice plots to reveal insights in data
- There are wrap-up assignments that include writing, charts and this analysis

When a new concept is introduced, it's shown and explained here. However, there are also **on your own** parts where you apply concepts you have learned in previous chapters or assignments.

## 11.2 Project setup

1. Within the same project you've been working, create a new Quarto Notebook. You might call it `02-analysis.qmd`.
2. We will be using a new package so you'll need to install it. **Use your Console** to run `install.packages("DT")`.
3. Include the libraries below and run them.

```
library(tidyverse)
library(scales)
library(DT)
```

## 11.3 Import cleaned data

1. Create a section for your import
2. Import your cleaned data and call it `sped` if you want to follow along here.

You should know how to do all that and I don't know what you called your export file anyway.

But to recap, the data should look like this:

```
sped |> head()
```

```
# A tibble: 6 x 8
  district distname cntyname year  all_count sped_count sped_percent audit_flag
  <chr>     <chr>    <chr>    <chr>      <dbl>      <dbl>       <dbl> <chr>
1 001902    CAYUGA  I~ ANDERSON 2013      595        73      12.3 ABOVE
2 001902    CAYUGA  I~ ANDERSON 2014      553        76      13.7 ABOVE
3 001902    CAYUGA  I~ ANDERSON 2015      577        76      13.2 ABOVE
4 001902    CAYUGA  I~ ANDERSON 2016      568        78      13.7 ABOVE
5 001902    CAYUGA  I~ ANDERSON 2017      576        82      14.2 ABOVE
6 001902    CAYUGA  I~ ANDERSON 2018      575        83      14.4 ABOVE
```

## 11.4 Make a searchable table

Wouldn't it be nice to be able to see the percentage of special education students for each district for each year? The way our data is formatted now, that's pretty hard to see with our "long" data here.

We could use something more like this? (But with all the years)

| distname      | cntyname | 2013 | 2014 | 2015 | etc |
|---------------|----------|------|------|------|-----|
| CAYUGA ISD    | ANDERSON | 12.3 | 13.7 | 13.2 | ... |
| ELKHART ISD   | ANDERSON | 9.1  | 8.9  | 10.4 | ... |
| FRANKSTON ISD | ANDERSON | 10.8 | 9.7  | 9.7  | ... |
| NECHES ISD    | ANDERSON | 11.1 | 9    | 11.1 | ... |
| PALESTINE ISD | ANDERSON | 7.7  | 7.8  | 8.7  | ... |

| distname     | cntyname | 2013 | 2014 | 2015 | etc |
|--------------|----------|------|------|------|-----|
| WESTWOOD ISD | ANDERSON | 9.3  | 10   | 9.3  | ... |

And what if you could make that table searchable to find a district by name or county? That would be magic, right?

We can do this by first reshaping our data using `pivot_wider()` and then applying a function called `datatable()`.

### 11.4.1 Pivot wider

You used `pivot_wider()` with the candy data in Chapter 9, so you can look back at how that was done, but here are some hints:

1. Create a new section that notes you are creating a table of district percents.
2. First use `select()` to get just the columns you need: `distname`, `cntyname`, `year` and `sped_percent`.
3. Then use `pivot_wider()` to make a tibble like the one above. Remember that the `names_from` = argument wants to know which column you want to use to create the names of the new columns. The `values_from` = argument wants to know which column to pull the cell values from.
4. Save the result into a new tibble and call it `district_percents_data`

```
district_percents_data <- sped |>
  select(distname, cntyname, year, sped_percent) |>
  pivot_wider(names_from = year, values_from = sped_percent)
```

### 11.4.2 Make a datatable

Now comes the magic.

1. In a new R chunk take your `district_percents_data` and then pipe it into a function called `datatable()`

Suppressed for printing

```
district_percents_data |>
  datatable()
```

That's kinda brilliant, isn't it? You know can search by any value in the table.

### **11.4.3 Data takeaway: How has Austin done**

That will be useful tool for you when you are writing about specific districts.

1. Use the data table to find AUSTIN ISD and in your own words, write a data takeaway sentence describing the change over time.

You can see how that might be useful tool for writing about specific districts.

## **11.5 Choosing a chart to display data**

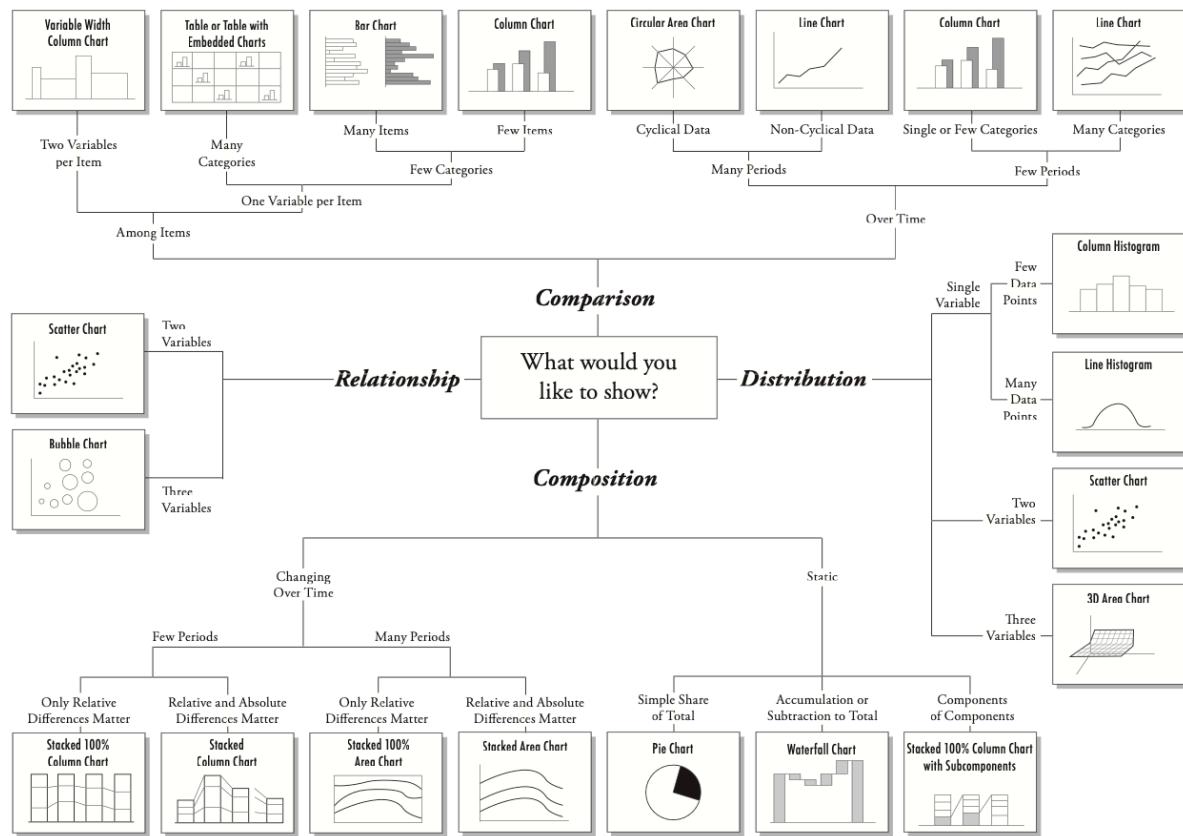
Our first question about this data was this: **Has the percentage of special education students in Texas changed since the benchmarking policy was dropped?**

Given the data we have, can we answer this?

Let's think about the charts that might be able to show two related variables like that. Choosing the chart type to display takes experimentation and exploration. Chapter 4 of Nathan Yau's *Data Points* book is an excellent look at which chart types help show different data.

This decision tree option might also help you think through it.

## Chart Suggestions—A Thought-Starter

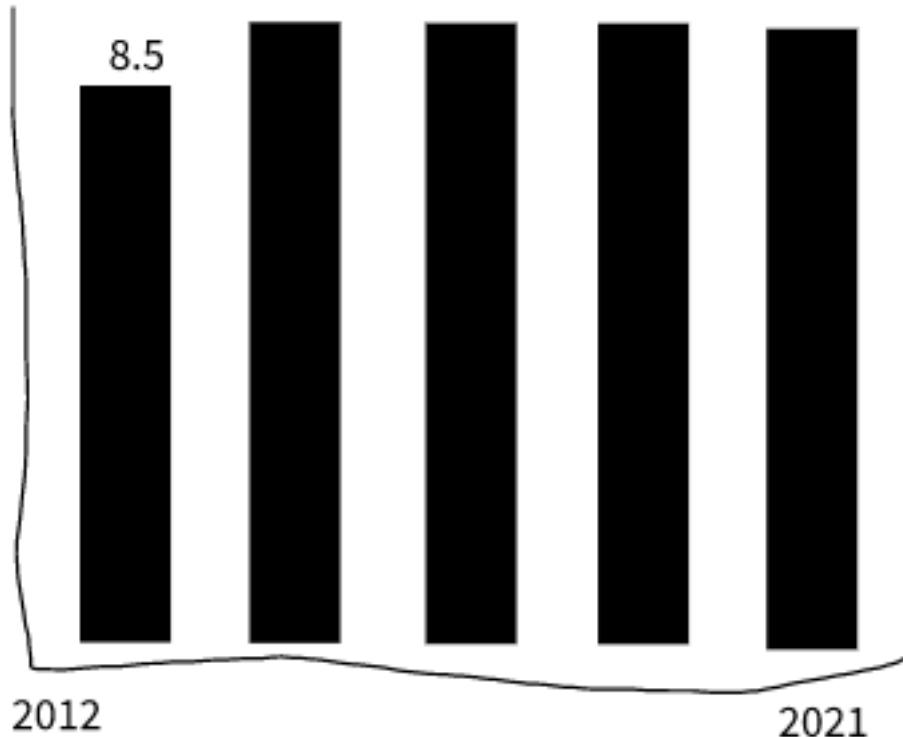


Lastly, another resource to consider this is the [ggplot cheatsheet](#), where it includes possible chart types based on the type of data we are comparing.

## 11.6 Plot yearly student percentage

I'll often do a hand drawing of a chart that might help me understand or communicate data. This helps me think about how to summarize and shape my data to get to that point.

For this question **Has the percentage of special education students in Texas changed since the benchmarking policy was dropped?**, we could show the percentage of special education students for each year. If we are to chart this, the x axis would be the year and the y axis would be the percentage for that year. Perhaps like this:



We have the percentage of students in special education for each district in each year. We *could* get an average of those percentages for each year, but that won't take into account the size of each district. Some districts have a single-digit number of students while others have hundreds of thousands of students.

But we also have the number of all students in a district with `all_count` and the number of special education students `sped_count`. With these, we can build a more accurate percentage across *all* the districts. This allows us to calculate our student groups by year.

### 11.6.1 Build the statewide percentage by year data

So, let's summarize our data. This is a “simple” `group_by()` and `summarize()` to get yearly totals, then a `mutate()` to build our percentage. I'll supply the logic so you can try writing it yourself.

1. Start a new notebook section and note you are getting yearly percentages
2. In an R chunk, start with your `sped` data.

3. Group by the `year` variable.
4. In summarize, create a `sum()` of the `all_count` variable. Call this `total_students`.
5. In the same summarize, create a `sum()` of the `sped_count` variable. You might call this `total_sped`.
6. Check your results at this point ... make sure it makes sense.
7. Next use `mutate()` to create a percentage called `sped_percent` from your `total_students` and `total_sped` summaries. The math for percentage is:  $(\text{part} / \text{whole}) * 100$ . You might want to round that resulting value to tenths as well.
8. I suggest you save all that into a new tibble called `yearly_percent` (because that's what I'm gonna do).

```
yearly_percent <- sped |>
  group_by(year) |>
  summarise(
    total_students = sum(all_count),
    total_sped = sum(sped_count)
  ) |>
  mutate(sped_percent = ((total_sped / total_students) * 100) |> round(1))

yearly_percent
```

|    | year  | total_students | total_sped | sped_percent |
|----|-------|----------------|------------|--------------|
|    | <chr> | <dbl>          | <dbl>      | <dbl>        |
| 1  | 2013  | 4870038        | 418360     | 8.6          |
| 2  | 2014  | 4929867        | 420847     | 8.5          |
| 3  | 2015  | 4984714        | 427377     | 8.6          |
| 4  | 2016  | 5034274        | 437487     | 8.7          |
| 5  | 2017  | 5070664        | 449303     | 8.9          |
| 6  | 2018  | 5088351        | 468097     | 9.2          |
| 7  | 2019  | 5099385        | 499240     | 9.8          |
| 8  | 2020  | 5142334        | 551718     | 10.7         |
| 9  | 2021  | 4993274        | 566484     | 11.3         |
| 10 | 2022  | 5025780        | 592457     | 11.8         |

### 11.6.2 Build the statewide percentage chart

And with that table, you can plot your chart using the `year` and `sped_percent`.

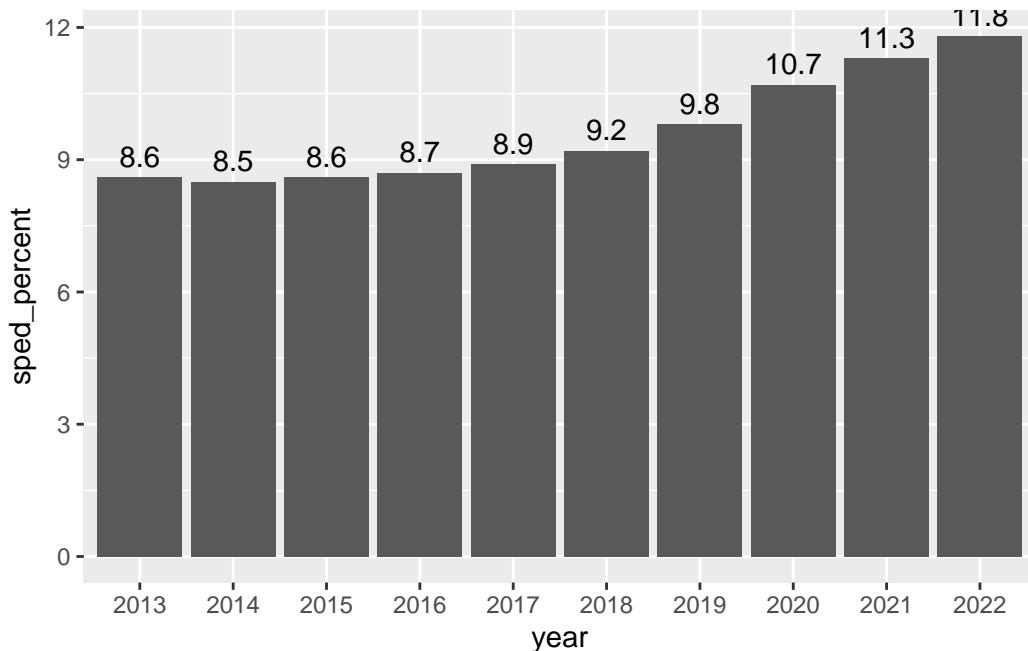
```

yearly_percent |>
  ggplot(aes(x = year, y = sped_percent)) +
  geom_col() +
  geom_text(aes(label = sped_percent, vjust = -.5))

```

(1)  
(2)  
(3)  
(4)

- ① I start with the data AND THEN pipe into ggplot.
- ② Within ggplot, set the aesthetics for the x and y axis.
- ③ Add `geom_col()` to build the chart. This is better than `geom_bar()` because it assumes one number and one categorical datum.
- ④ The `geom_text()` plots the labels on top of the bars based `sped_percent` but we need to adjust them with `vjust =` to move them above the bar.



This definitely shows us that the percentage of students in special education (in traditional public schools) has increased each year since 2017 when the benchmark was dropped and then outlawed by the legislature.

I would be careful about saying the increase is *because* of the changes (though that is likely true), but you can certainly say with authority that it has gone up, and interview other people to pontificate on the reasons why.

BTW, if you wanted to make that chart in [datawrapper](#), you could use the same data structure.

### 11.6.3 Data takeaway: State percentage

1. Write a data takeaway that describes how the statewide percentage of special education students has changed over time.

#### The COVID factor

Since we are staring at 2021 and 2022 numbers here, we have an opportunity to discuss how the “COVID era” will have a lasting affect on education data and many other data sources. The adjustments and adaptations that schools had to make changed our world and the data gathered during that time. While our world is always evolving, COVID affected some aspects of our society in profound and lasting ways, and we need to recognize that in our reporting.

In this case, as in many others you will come across in the wild, there isn’t much we can do other than to recognize it was a different time. If the COVID effect is strong enough, we might specifically note it in our charts or stories. In the end, this is the data we have from the TEA and there is no regression or alternative we can use to “normalize” our data to previous times.

## 11.7 Districts by benchmark and year

Our second question is this: **How many districts were above that arbitrary 8.5% benchmark before and after the changes?**

It was in anticipation for this that we built the `audit_flag` field in our data. With that we can count how many rows have the “ABOVE” or “BELOW” value. (In reality, it’s probably at this point we would discover that might be useful that field is and have to go back to the cleaning notebook to create it. I wanted to get that part out of the way so we can concentrate on the charts.)

To decide on which chart to build, we can go back to our chart decision workflow or consult the ggplot cheatsheet. Given we have the discrete values of `audit_flag` and `years`, and we want to plot how many districts are counted (which is a continuous value), we’re looking at a stacked or grouped column/bar chart or a line chart.

### 11.7.1 Summarize the audit flag data

Before we can build the chart, we need to summarize our data. The logic is this: We need to group our data by both the `year` and the `audit_flag`, and then count the number of rows for those values.

1. Start a new Markdown section and note you are counting districts by the audit flag.

2. Start with your original `sped` data.
3. Group your data by both `year` and `audit_flag`.
4. Summarize your data by counting `n()` the results. Name the variable `count_districts`.
5. Save the resulting data into a tibble called `flag_count_districts`. You might print that result out so you can refer to it.

```
flag_count_districts <- sped |>
  count(year, audit_flag, name = "count_districts")

# I used count()
```

The data should look like this:

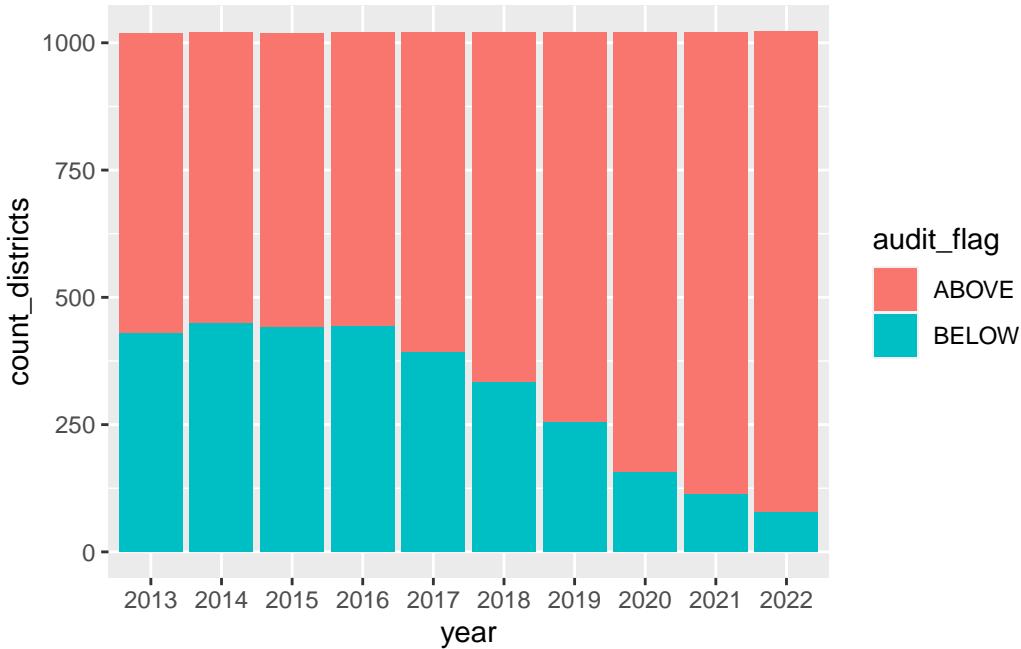
```
# A tibble: 20 x 3
  year   audit_flag count_districts
  <chr>  <chr>           <int>
1 2013   ABOVE          591
2 2013   BELOW          429
3 2014   ABOVE          571
4 2014   BELOW          449
5 2015   ABOVE          579
6 2015   BELOW          441
7 2016   ABOVE          578
8 2016   BELOW          442
9 2017   ABOVE          629
10 2017  BELOW          391
11 2018  ABOVE          688
12 2018  BELOW          332
13 2019  ABOVE          765
14 2019  BELOW          255
15 2020  ABOVE          865
16 2020  BELOW          156
17 2021  ABOVE          909
18 2021  BELOW          112
19 2022  ABOVE          945
20 2022  BELOW          77
```

### 11.7.2 Build the audit flag chart

Now that we have the data, we can build the ggplot column chart. The key new thing here is we are using a new aesthetic `fill` to apply colors based on the `audit_flag` column.

1. Add some notes you are building the first exploratory chart
2. Add an R chunk with the following:

```
flag_count_districts |>
  ggplot(aes(x = year, y = count_districts, fill = audit_flag)) +
  geom_col()
```



This is actually not a bad look because it does clearly show the number of districts below the 8.5% audit benchmark is dropping year by year, and the number of districts above is growing.

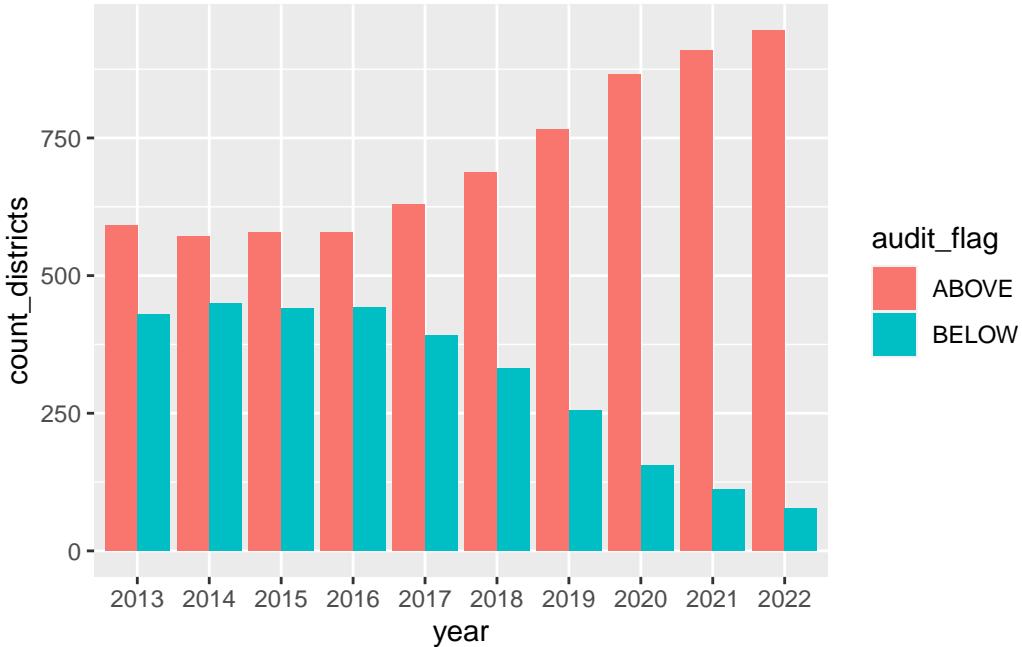
Leave that chart there for reference, but let's build a new one that is almost the same, but we'll adjust it to be a grouped column chart instead of stacked. The key difference is we are adding `position = "dodge"` to the column geom.

1. Note in Markdown text you are building the grouped column version
2. Add a new chunk and start with exactly what you have above.
3. Update the column geom to this: `geom_col(position = "dodge")`

```
flag_count_districts |>
  ggplot(aes(x = year, y = count_districts, fill = audit_flag)) +
  geom_col(position = "dodge")
```

①

① This is where we add `position = "dodge"` to set the bars side-by-side.



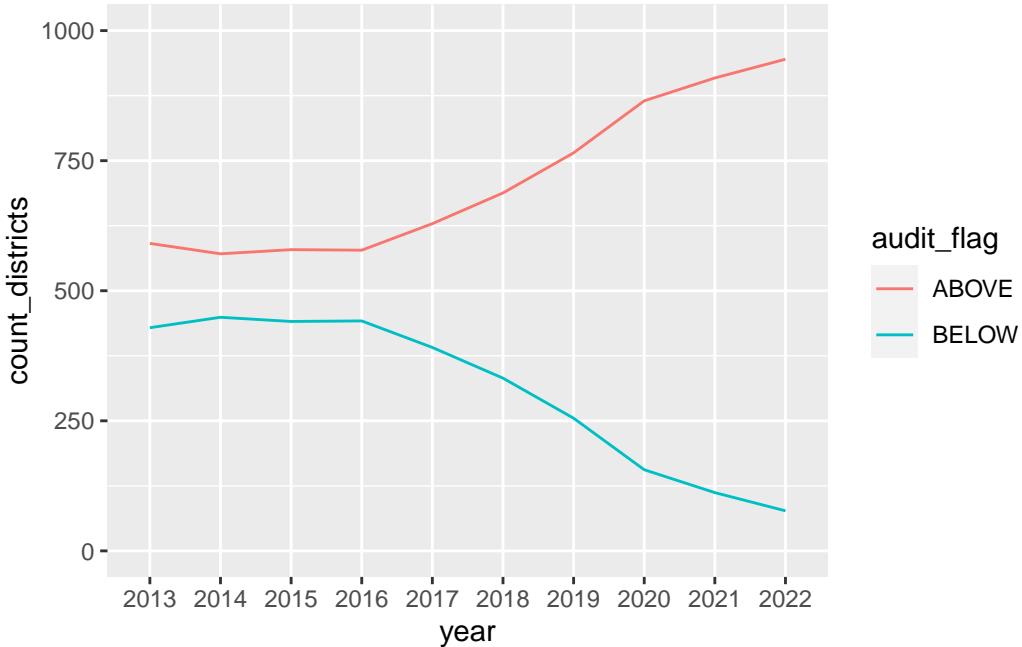
That's not bad at all ... it might be the winner.

Lastly, let's chart this data as a line chart to see if that looks any better or is easier to comprehend.

1. Note you are visualizing as a line
2. Add the chunk below.

```
flag_count_districts |>
  ggplot(aes(x = year, y = count_districts, group = audit_flag)) +
  geom_line(aes(color = audit_flag)) +
  ylim(0,1000) ①
```

① We added `ylim()` here because the default didn't start the y axis at zero. `ylim` is used here as a shortcut for `scale_y_continuous(limits = c(0,1000))`.



### 11.7.3 Which chart is best?

Remember, this is the question you are trying to answer: **How many districts were above that arbitrary 8.5% benchmark before and after the changes?**

Both the stacked bar chart and the grouped bar chart explain this concept. I like the grouped bars a little better because the stacked one kind of looks like all the bars are the same height, evoking 100% or something instead of the number of districts (which don't change that much.) I'm probably over thinking that, TBH. Sometimes you just gotta make a call.

### 11.7.4 Data takeaway: District benchmarks

1. Now that you've seen the same data in three ways, write a data takeaway the describe what you've learned.

## 11.8 Local districts

We have one last question: **How have local districts changed?** i.e., what are the percentages for districts in Bastrop, Hays, Travis and Williamson counties? We want to make sure none of these buck the overall trend.

You can certainly use the searchable table we made to get an idea of the number of districts and how the numbers have changed, but you can't *see* them there. Searching there does reveal there are too many districts to visualize them all at once. Maybe we can chart one county at a time.

Looking at the chart suggestions workflow, we are doing a comparison over time of many categories ... so our trusty line chart is the horse to hitch.

To make that line chart we think about our axes and groups: We need our x axis of `year`, y axis of the `sped_percent` and we need to group the lines by their district. Since we need a column that has each year, the *long* data we started with should suffice:

```
sped |> head()
```

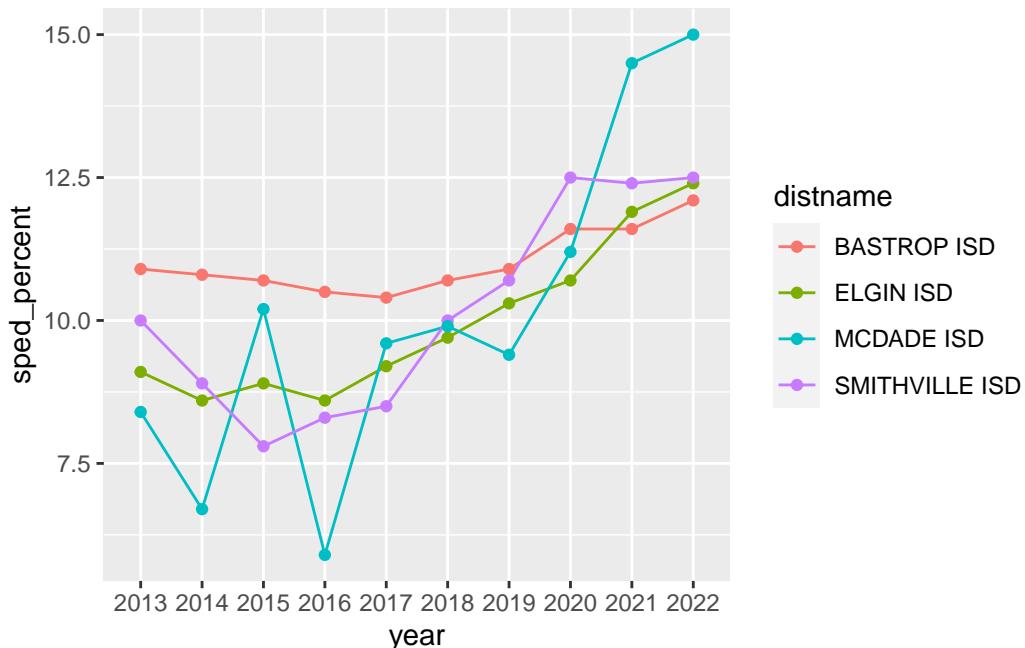
```
# A tibble: 6 x 8
  district distname cntyname year  all_count sped_count sped_percent audit_flag
  <chr>     <chr>    <chr>    <chr>      <dbl>       <dbl>        <dbl> <chr>
1 001902    CAYUGA   I~ ANDERSON 2013      595         73        12.3 ABOVE
2 001902    CAYUGA   I~ ANDERSON 2014      553         76        13.7 ABOVE
3 001902    CAYUGA   I~ ANDERSON 2015      577         76        13.2 ABOVE
4 001902    CAYUGA   I~ ANDERSON 2016      568         78        13.7 ABOVE
5 001902    CAYUGA   I~ ANDERSON 2017      576         82        14.2 ABOVE
6 001902    CAYUGA   I~ ANDERSON 2018      575         83        14.4 ABOVE
```

We just need to filter that down to a single county `cntyname == "BASTROP"` to show this. We can even do this all in one code block. Let's see if you can follow this logic and build Bastrop county for yourself.

1. Start a new section that you are looking at local districts
2. Start a new chunk with your original `sped` data
3. Filter it to have just rows for **BASTROP** county
4. Pipe that result into the `ggplot()` function
5. For the x axis, you are using `year`, for the y use `sped_percent` and for group use `distname`
6. Inside your `geom_line()` set the the color to the district: `aes(color = distname)`.
7. Do the same for a `geom_point()` layer.

It should look like this:

```
sped |>
  filter(cntyname == "BASTROP") |>
  ggplot(aes(x = year, y = sped_percent, group = distname)) +
  geom_line(aes(color = distname)) +
  geom_point(aes(color = distname))
```



### 11.8.1 On your own

- Now do the same for the other three counties, each in their own code chunk: Hays, Travis and Williamson.

These charts give you some reference for the local districts. You'll see the more districts there are within a county, the less effective the line chart becomes. But at least it gives you an idea of which districts are following the trend.

### 11.8.2 Data takeaway: The local districts

Write data takeaways about the three counties you've looked at.

## 11.9 Special Education change since 2015

We can see the changes in special education percentages in our chart, but how do we best describe in prose the change from before the law changes (2015) to our current year?

We have two values for each year to work with: The “Count” of special education students in each district, which is the actual number of students in the program; and the “Percentage” of students in special education out of the total in that district

To describe change from one year to the next you might review the Numbers in the Newsroom chapter on Measuring Change (p26). We'll use Austin ISD numbers from 2015 and 2022 in our examples:

```
# A tibble: 2 x 5
  distname   year all_count sped_count sped_percent
  <chr>     <chr>    <dbl>      <dbl>        <dbl>
1 AUSTIN ISD 2015     84191      8354       9.9
2 AUSTIN ISD 2022     71883      9396      13.1
```

### 11.9.1 Describing the count changes

- We can show the **simple difference** (or actual change) in the *count* of students from one year to the next. There were 8354 students in 2015 and 9396 in 2022:
  - New Count - Old Count = Simple Difference
  - Example: 9396 - 8354 = 1042
  - “Austin ISD served 1,000 more special education students in 2023 (9,396 students) compared to 2015 (8,354 students)”. (I rounded at the beginning for simplicity.)
- We can show the **percent change** in the *count* of students from one year to the next:
  - $((\text{New Count} - \text{Old Count}) / \text{Old Count}) * 100 = \text{Percent change}$
  - Example:  $((9396 - 8354) / 8354) * 100 = 12.5\%$
  - “The number of special education students served increased 12% from 8,354 in 2015 to 9,396 in 2022.” (Again, I rounded the percent change, but kept the actual counts. The counts could be rounded as well.)

This is accurate, but the percent change can be blown out of proportion when you are dealing with small number. In a small district, moving from 2 to 4 students is a 200% change.

### 11.9.2 Describing percentage differences

We also have the *percentage* of special education students in the school, which could be important. This is the share of students that are in the program compared to the total students in the school.

- We can find the **percentage point difference** from one year to the next using simple difference again, but we have to describe the change as the difference in percentage points:
  - New Percentage - Old Percentage = Percentage Point Difference
  - Example: 13.1% - 9.9% = 3.2 percentage points (NOT 3.2%).

- “The share of students in special education grew by 3.2 percentage points, from 9.9% in 2015 to 13.1% in 2022.”
- We can find the **percent change of share** from one year to the next, but we have to again be very specific about what we are talking about ... the growth (or decrease) of the *share* of students in special education.
  - $((\text{New Percentage} - \text{Old Percentage}) / \text{Old Percentage} * 100) = \text{Change in share of students}$
  - Example:  $((13.1 - 9.9) / 9.9) * 100 = 32.3$ .
  - “The share of students in special education grew by a third from 10% of students in 2015 to 13% of students in 2020.” This describes the growth in the share of students in the program, not the number of special education students overall. I also rounded the percentages.

Describing a “percentage point difference” to readers can be difficult, but perhaps less confusing than describing the “percent change of a percent”.

Great, so which do we use for this story? That depends on what you want to describe. Districts that have fewer special education students to begin with will show a more pronounced percent change with any fluctuation. Then again, a district that has a large percentage of students could be gaining a lot of students with a small percentage change. In the end, we might need to use all of these values to describe different kinds of school districts. We are talking about human beings, so perhaps the counts are important.

## 11.10 Build your data drop

Like you have with previous projects, update your index.qmd file to add the following:

1. Write a one to two sentence summary about the project as a whole. You can work from some of the material already there an in the cleaning notebook.
2. Write data-driven news lede based on what you think is the most important takeaway from this analysis.
3. Write a source paragraph that explains the source and range of the data used in the project.
4. Write a series of data takeaway sentences based on what you found.

## 11.11 Turn in your project

1. Make sure everything runs and Renders properly.
2. Publish your changes to Quarto Pub and include the link to your project in your index notebook so I can bask in your glory.

3. Zip your project folder. (Or export to zip if you are using posit.cloud).
4. Upload to the Canvas assignment.

**!** Important

To be clear, it is your zipped project I am grading. The Quarto Pub link is for convenience.

## 11.12 New functions used in this chapter

- We used `datatable()` to build an interactive, searchable table from our data. It's part of the `DT` packages.
- I used `count()` which is a shortcut for grouping and counting rows of data.

# A R Functions

An opinionated list of the most common data wrangling functions. It leans heavily into the [Tidyverse](#).

## A.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 use, but it has `read_excel()` that allows you to import from an Excel file, including specified sheets and ranges.
- `clean_names()` from the `library(janitor)` package standardizes column names.

## A.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 an existing column or creates a new one. **Example:** `mutate(new_col = (col01 / col02))`.
- `round()` is a base R function that can round a number to a set decimal point. Often used within a `mutate()` function.
- `recode()`, `if_else()` and `case_when()` are all functions that can be used with `mutate()` to create new categorizations with your data.

- `pivot_longer()` “lengthens” data, increasing the number of rows and decreasing the number of columns. **Example:** `pivot_longer(cols = 3:5, names_to = "new_key_col_name", values_to = "new_val_col_name")` will take the third through the fifth columns and turn each value into a new row of data. It will put them into two columns: The first column will have the name you give it in `names_to` and contain the old column name that corresponds to each value pivoted. The second column will have the name of whatever you set in `values_to` and will contain all the values from each of the columns.
- `pivot_wider()` is the opposite of `pivot_longer()`. **Example:** `pivot_wider(names_from = col_of_key_values, values_from = col_with_values)`. See the link.

## A.3 Aggregation

- `group_by()` and `summarize()` often come together. When you use `group_by()`, every function after it is broken down by that grouping. We often add `arrange()` to these, calling this our GSA functions. **Example:** `group_by(song, artist) %>% summarize(weeks = n(), top_chart_position = min(peak_position))`. To break or remove groupings, use `ungroup()`.
- `count()` is a shortcut for GSA that count the number rows based on variable groups you feed it.

## A.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.

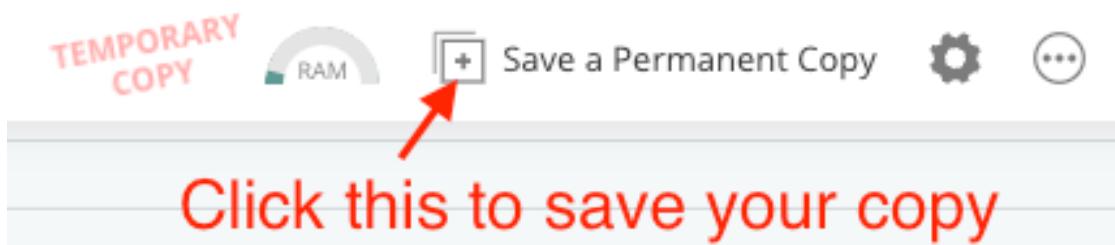
## B Using posit.cloud

If you are using the online version of RStudio called [posit.cloud](#) then there are some things that work differently than with the desktop version. This chapter is designed to help with that.

### B.1 Create a web project from our template

Since posit.cloud does not have an option to create a “Quarto Website” project, we have put together a template for you to start from.

1. Start posit.cloud if you haven't already
2. Click on this link: [RWD PositCloud Template](#)
3. At the top of the browser window, there is a button to save the template as your own project called **Save a Permanent Copy**.



4. Rename your project right away so you don't forget.



5. Update the `index.qmd` file with information about your project.
6. Update your `_quarto.yml` file as you wish.

The base packages should already be installed, so you should be good to add new Quarto Documents and Render them.

## B.2 Exporting a project

You can export your project as a `.zip` file to turn in to an assignment or share with others.

1. In the **Files** pane, click the box next to the *Cloud* icon to select all your files.
2. Under the **More** gear there is a dropdown. Click on that.
3. Choose **Export** from the More menu.

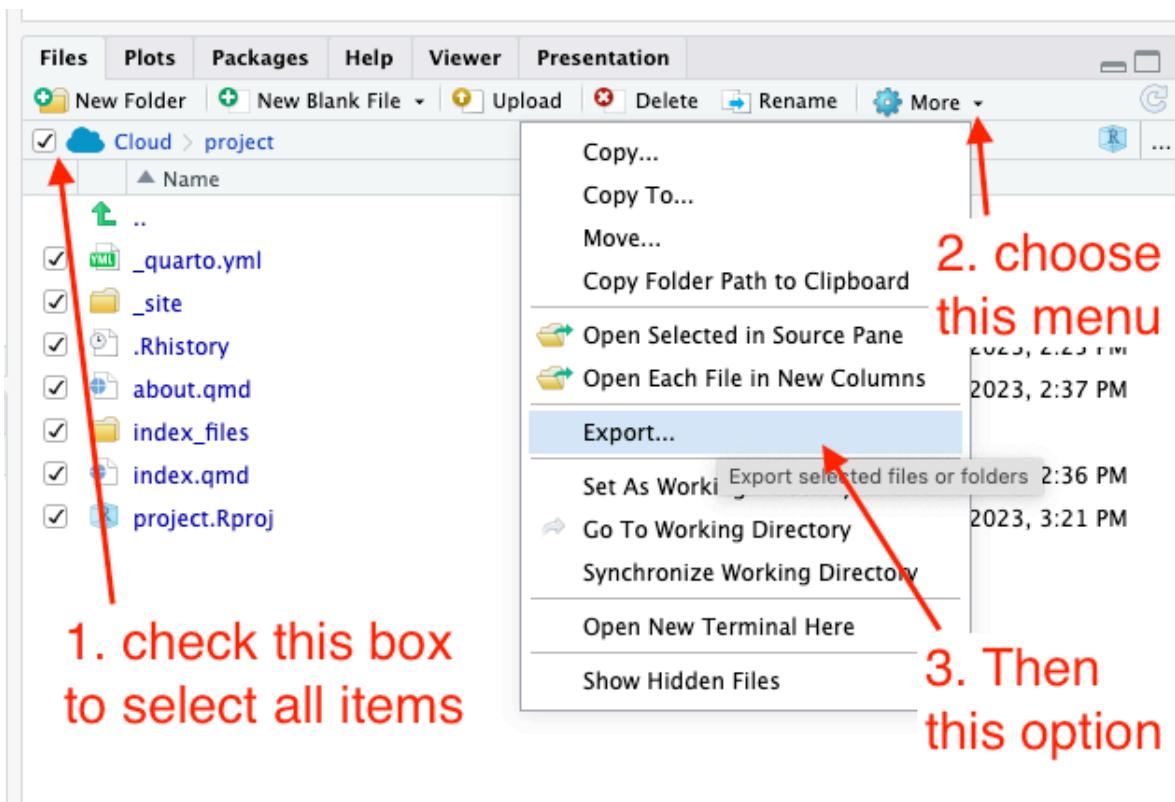


Figure B.1: Exporting a project

This should download all your files as a `.zip` file, which you can upload to Canvas.

## B.3 Share your project

It is possible to share your posit.cloud project with another user on the service (like your instructor) and they will get a copy of it.

You can find [directions for that here](#).

## B.4 Building a web project from scratch

If you can't or don't want to start with the template above (like you already have a project started), you can build your own Quarto Website.

### B.4.1 Create your project

1. Start posit.cloud if you haven't already
2. If you aren't in a project already, create one. Use the **New Project** button and choose **New RStudio project**.
3. In your Console, copy and paste this command and run it.

```
install.packages(c("quarto", "rmarkdown", "tidyverse", "janitor"))
```

It will take some time to run. Your internet connection will have an impact on the speed.

### B.4.2 Create the Quarto file

1. Use the new document toolbar button to create a **Text file**.
2. Paste in the code below.
3. Save the file and name it `_quarto.yml`.

The name must be exact.

```
project:  
  type: website  
  
website:  
  title: "Site name"  
  navbar:  
    left:  
      - href: index.qmd  
        text: Home
```

```
#- filename.qmd

format:
  html:
    theme: cosmo
    toc: true
    df-print: paged
    code-overflow: wrap
```

### B.4.3 Create your index file

1. Use the new document toolbar button to create a new **Quarto Document**
2. For the Title field, use your project name, like “Billboard project”
3. Uncheck the visual editor button.
4. Immediately save the file and name it `index.qmd`

At some point you’ll likely add a new file and want to replace `filename.qmd` with your filename and remove the `#` comment. You’ll can add other files there as you create them. This adds them to the website navigation. You can learn about website navigation in the [Quarto Guide](#).

**You should be good to go with a new project at this point.** You can Render your index to see what the site looks like at this point.

# C Grouping by dates

## Note

This lesson was pulled out of the Billboard project because it was getting really long, but you'll need this skill when you are working on your mastery assignments.

It is not uncommon in data journalism to count or sum records by year based on a date within your data. Or even by other date parts like month or week. There are some really nifty features within the `lubridate` package that make this pretty easy.

We'll run through some of those scenarios here using the Billboard Hot 100 data we used in Chapters 3 & 4. If you want to follow along, you can create a new Quarto Document in your Billboard project. Or you can just use this for reference.

## C.1 Setting up

We need to set up our notebook with libraries and data before we can talk specifics. We need to load both the `tidyverse` and `lubridate` libraries. Lubridate is installed with the `tidyverse` package, but you have to load it separately.

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

And we need our cleaned Billboard Hot 100 data. We'll glimpse the rows and check out date range so we can remember what we have.

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

```
Rows: 341,300
Columns: 7
$ chart_date    <date> 1958-08-04, 1958-08-04, 1958-08-04, 1958-08-
```

```
$ current_rank <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~  
$ title <chr> "Poor Little Fool", "Patricia", "Splish Splash", "Hard H~  
$ performer <chr> "Ricky Nelson", "Perez Prado And His Orchestra", "Bobby ~  
$ previous_rank <dbl> NA, ~  
$ peak_rank <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~  
$ wks_on_chart <dbl> 1, ~  
  
hot100$chart_date |> summary()
```

|              | Min.         | 1st Qu.      | Median       | Mean         | 3rd Qu.      | Max. |
|--------------|--------------|--------------|--------------|--------------|--------------|------|
| "1958-08-04" | "1974-12-14" | "1991-04-20" | "1991-04-19" | "2007-08-25" | "2023-12-30" |      |

## C.2 Plucking date parts

If you look at the [lubridate cheatsheet](#) under “GET AND SET DATE COMPONENTS” you’ll see functions to pluck out parts of a date, like `year()`.

If we have a date, like perhaps Taylor Swift’s birthday, we can pluck the year from it.

```
year("1989-12-13")
```

```
[1] 1989
```

## C.3 Grouping by a date part on the fly

Let’s show how this might be useful through an example question:

**Which performer has the most appearances on the chart in a given year?**

The logic works like this:

- Group all the records by `performer` AND the year of the `chart_date`
- Summarize and count the rows

```
hot100 |>  
  group_by(  
    year(chart_date),  
    performer  
  ) |>  
  summarize(appearances = n()) |>  
  arrange(desc(appearances))
```

- ① This is where we add the year to the group\_by, plucking it from the `chart_date` with the `year()` function.

```
# A tibble: 23,407 x 3
# Groups:   year(chart_date) [66]
`year(chart_date)` performer     appearances
<dbl> <chr>             <int>
1      2023 Morgan Wallen    327
2      1964 The Beatles     214
3      2023 Taylor Swift    208
4      2023 SZA              177
5      2021 Olivia Rodrigo   172
6      2018 Drake            168
7      2022 Bad Bunny        148
8      2019 Billie Eilish    145
9      2016 Drake            134
10     2015 The Weeknd       126
# i 23,397 more rows
```

It looks like Morgan Wallen has the most appearances and 2023 is only half over as of this running. When I ran this code last year The Beatles topped the list. There is definitely some kinda story here.

Anyway, notice how the year column name is kinda shite? We would not be able to easily reference that variable later, so we should rename that AS we make the group:

```
hot100 |>
  group_by(
    yr = year(chart_date), ①
    performer
  ) |>
  summarize(appearances = n()) |>
  arrange(desc(appearances))
```

- ① Added the “`yr =`” here

```
# A tibble: 23,407 x 3
# Groups:   yr [66]
      yr performer     appearances
      <dbl> <chr>             <int>
1 2023 Morgan Wallen    327
2 1964 The Beatles     214
```

```
3 2023 Taylor Swift          208
4 2023 SZA                  177
5 2021 Olivia Rodrigo       172
6 2018 Drake                 168
7 2022 Bad Bunny              148
8 2019 Billie Eilish          145
9 2016 Drake                 134
10 2015 The Weeknd            126
# i 23,397 more rows
```

It is a good practice to rename any grouping variable made from a function like that. FWIW, it would've worked if I called the new column `year`, but I named it `yr` so I'm less likely to confuse it with the function `year()`. It's a personal preference what to name the new column.

## C.4 Making reusable date parts

If you think you'll use a date parts more than once, then it makes sense to create a new columns and save them. You might make several date parts, but we'll start with only one.

### 💡 Tip

I usually go back to my cleaning notebook to add these once I recognize I need them, and then rerun everything.

I've created a random sample of data with only the `chart_date` and `title` columns just so it is easier to see what we are doing. You would normally work with the whole data frame! Here is our sample:

```
hot100_sample <- hot100 |>
  slice_sample(n = 6) |>
  select(chart_date, title)

hot100_sample

# A tibble: 6 x 2
  chart_date     title
  <date>        <chr>
1 2018-03-03 I Fall Apart
2 1963-08-31 Surf City
3 1997-10-04 How Do I Live
4 1978-07-29 Only The Good Die Young
```

```
5 1974-08-24 Rub It In  
6 1972-06-03 It's Going To Take Some Time
```

### C.4.1 Let's make a year

Here's how we do it:

```
hot100_sample |>  
  mutate(  
    yr = year(chart_date)  
  )  
①  
②
```

- ① We use `mutate` to create a new column. This is where it starts.

```
# A tibble: 6 x 3  
  chart_date title          yr  
  <date>     <chr>        <dbl>  
1 2018-03-03 I Fall Apart  2018  
2 1963-08-31 Surf City    1963  
3 1997-10-04 How Do I Live 1997  
4 1978-07-29 Only The Good Die Young 1978  
5 1974-08-24 Rub It In    1974  
6 1972-06-03 It's Going To Take Some Time 1972
```

- We name the new column `yr`, and then set the value to the `year()` of `chart_date`.

### C.4.2 The magical month

We can also pluck out the month of the date, which is pretty useful if you want to measure seasonality within a year, like hot days of summer, etc. The default `month()` function pulls the month as a number.

```
hot100_sample |>  
  mutate(  
    mo = month(chart_date)  
  )  
  
# A tibble: 6 x 3  
  chart_date title          mo  
  <date>     <chr>        <dbl>  
1 2018-03-03 I Fall Apart  3
```

|   |            |                              |    |
|---|------------|------------------------------|----|
| 2 | 1963-08-31 | Surf City                    | 8  |
| 3 | 1997-10-04 | How Do I Live                | 10 |
| 4 | 1978-07-29 | Only The Good Die Young      | 7  |
| 5 | 1974-08-24 | Rub It In                    | 8  |
| 6 | 1972-06-03 | It's Going To Take Some Time | 6  |

But there are some options within `month()` to give us month NAMES that are ordered as factors instead of alphabetical.

```
hot100_sample |>
  mutate(
    mo_label = month(chart_date, label = TRUE),
    mo_long = month(chart_date, label = TRUE, abbr = FALSE)
  ) |>
  arrange(mo_label)
```

| # A tibble: 6 x 4 | chart_date | title                        | mo_label | mo_long |
|-------------------|------------|------------------------------|----------|---------|
|                   | <date>     | <chr>                        | <ord>    | <ord>   |
| 1                 | 2018-03-03 | I Fall Apart                 | Mar      | March   |
| 2                 | 1972-06-03 | It's Going To Take Some Time | Jun      | June    |
| 3                 | 1978-07-29 | Only The Good Die Young      | Jul      | July    |
| 4                 | 1963-08-31 | Surf City                    | Aug      | August  |
| 5                 | 1974-08-24 | Rub It In                    | Aug      | August  |
| 6                 | 1997-10-04 | How Do I Live                | Oct      | October |

Note the datatype `<ord>` under the column `mo_label` and `mo_long`. That means this is an “ordered factor” and that when sorted by those labels it will list in MONTH order instead of alphabetical order, which is quite useful.

### C.4.3 Floor dates

Sometimes you want to count the number of records within a month and year, like all the songs in January 2000, then February 2000, etc. One way to do that is to create a `floor_date`, which gives you the “lowest” date within a certain unit like year or month. It’s easiest to show with our sample data:

```
hot100_sample |>
  mutate(
    fl_month = floor_date(chart_date, unit = "month"),
    fl_year = floor_date(chart_date, unit = "year")
  )
```

```

# A tibble: 6 x 4
  chart_date title                fl_month   fl_year
  <date>     <chr>              <date>     <date>
1 2018-03-03 I Fall Apart        2018-03-01 2018-01-01
2 1963-08-31 Surf City          1963-08-01 1963-01-01
3 1997-10-04 How Do I Live      1997-10-01 1997-01-01
4 1978-07-29 Only The Good Die Young 1978-07-01 1978-01-01
5 1974-08-24 Rub It In          1974-08-01 1974-01-01
6 1972-06-03 It's Going To Take Some Time 1972-06-01 1972-01-01

```

You can see the resulting new columns are real dates, but they are normalized:

- The `fl_month` gives you the first day of the month for that `chart_date`.
- The `fl_year` gives you the first day of the year for that `chart_date`.

Let's put this to use with an example. I'll create a `fl_month` on the fly to find **Recent appearances by Taylor Swift**. I'll also do the `year()` on the fly in my filter.

```

swift_month <- hot100 |>
  filter(
    performer == "Taylor Swift",
    chart_date >= "2020-07-01"
  ) |>
  group_by(
    fl_month = floor_date(chart_date, unit = "month")
  ) |>
  summarize(appearances = n())

swift_month

```

```

# A tibble: 32 x 2
  fl_month   appearances
  <date>       <int>
1 2020-08-01      28
2 2020-09-01       6
3 2020-10-01       5
4 2020-11-01       1
5 2020-12-01      12
6 2021-01-01       7
7 2021-02-01       5
8 2021-03-01       4
9 2021-04-01      12

```

```
10 2021-05-01          3  
# i 22 more rows
```

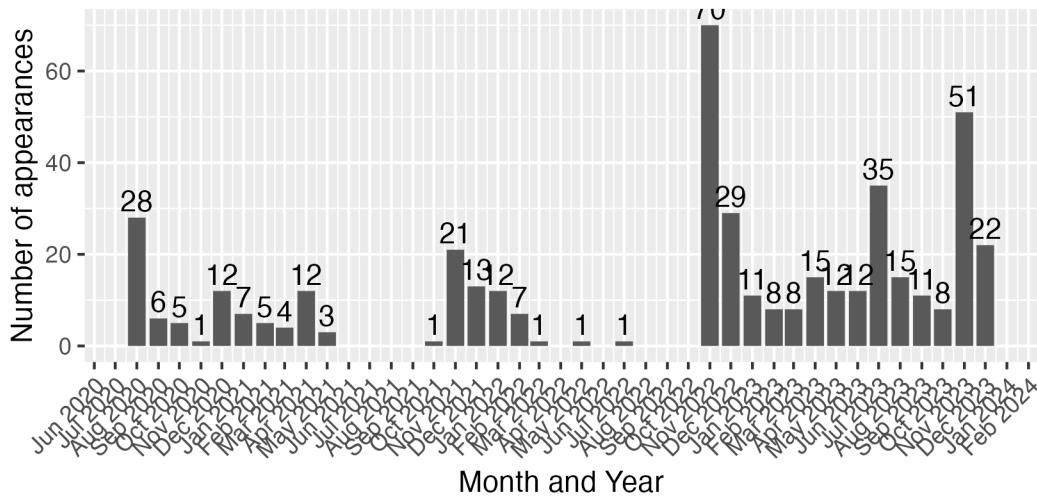
And chart it:

```
swift_month_plot <- swift_month |>  
  ggplot(aes(x = fl_month, y = appearances)) +  
  geom_col() +  
  geom_text(aes(label = appearances), vjust = -.3) +  
  scale_x_date(date_labels="%b %Y", date_breaks  ="1 month") +  
  guides(x =  guide_axis(angle = 45)) +  
  labs(  
    x = "Month and Year",  
    y = "Number of appearances",  
    title = "Swifts' \"Midnights\" drives most hits",  
    subtitle = str_wrap("While her most recent new album drove the most appearances on the Billb  
)  
  
ggsave("figures/swift_month_plot.png")
```

Saving 5.5 x 3.5 in image

## Swifts' "Midnights" drives most hits

While her most recent new album drove the most appearances on the Billboard Hot 100 within a month, each "Taylor's version" album since 'Folklore' has generated hits.



Can you guess when she [released her albums](#)?

# D A counting shortcut

We count stuff in data science (and journalism) all the time. In our Billboard project we used `summarize()` and the `n()` function to count rows because we needed to understand the concepts of group\_by, summarize and arrange. It is perfectly valid and the best way to explain and understand grouping and summarizing.

But in the interest of full disclosure, know that dplyr has a shortcut to group, count and arrange rows of data. The `count()` function takes the columns you want to group and then does the summarize on `n()` for you. We'll demonstrate them with the Billboard data. You can create a new notebook to try this, or just use this for reference.

We didn't start with `count()` because it can't do anything else other than count rows, and we needed to build our GSA knowledge to summarize all kinds of ways, like `sum()`, `mean()` or `slice()`.

## D.1 Setup and import

We don't normally put these together, but we're just setting up a quick demonstration

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

## D.2 Basic count

We're going to rework our first quest of the Billboard analysis:

**Which performer had the most appearances on the Hot 100 chart at any position?**

Our logic is we want to count the number of rows based on each performer. We do this by adding the variables we want to group as arguments to `count()`:

```
hot100 |>
  count(performer)

# A tibble: 10,742 x 2
  performer             n
  <chr>                 <int>
1 "\"Groove\" Holmes"     14
2 "\"Little\" Jimmy Dickens" 10
3 "\"Pookie\" Hudson"      1
4 "\"Weird Al\" Yankovic"   91
5 "$NOT & A$AP Rocky"       1
6 "'N Sync"                  172
7 "'N Sync & Gloria Estefan" 20
8 "'N Sync Featuring Nelly" 20
9 "'Til Tuesday"                53
10 "(+44)"                      1
# i 10,732 more rows
```

### D.2.1 Sort the results

If we want the highest counted row at the top (and we almost always do) then we can add an argument: `sort = TRUE`.

```
hot100 |>
  count(performer, sort = TRUE)

# A tibble: 10,742 x 2
  performer             n
  <chr>                 <int>
1 Taylor Swift        1386
2 Drake                  896
3 Elton John            889
4 Madonna                 857
5 Kenny Chesney          777
6 Tim McGraw              749
7 Keith Urban              674
8 Morgan Wallen           659
9 Stevie Wonder            659
10 Rod Stewart              657
# i 10,732 more rows
```

### D.2.2 Name the new column

Notice the counted table is called `n`. We can rename that with another argument, `name =` and give it the name we want in quotes.

```
hot100 |>  
  count(performer, sort = TRUE, name = "appearances")
```

```
# A tibble: 10,742 x 2  
  performer      appearances  
  <chr>             <int>  
1 Taylor Swift        1386  
2 Drake                 896  
3 Elton John            889  
4 Madonna                857  
5 Kenny Chesney          777  
6 Tim McGraw              749  
7 Keith Urban              674  
8 Morgan Wallen            659  
9 Stevie Wonder              659  
10 Rod Stewart               657  
# i 10,732 more rows
```

### D.2.3 Filter results as normal

To cut off the results, we just filter as we normally would.

```
hot100 |>  
  count(performer, sort = TRUE, name = "appearances") |>  
  filter(appearances > 650)
```

```
# A tibble: 10 x 2  
  performer      appearances  
  <chr>             <int>  
1 Taylor Swift        1386  
2 Drake                 896  
3 Elton John            889  
4 Madonna                857  
5 Kenny Chesney          777  
6 Tim McGraw              749  
7 Keith Urban              674
```

|                 |     |
|-----------------|-----|
| 8 Morgan Wallen | 659 |
| 9 Stevie Wonder | 659 |
| 10 Rod Stewart  | 657 |

So the code above does the same things here as we did in our first Billboard quest, but with fewer lines.

### D.3 Grouping on multiple variables

We can group on multiple variables by adding them. We'll show this with the second quest:

**Which song (title & performer) has been on the charts the most?**

```
hot100 |>
  count(
    title,
    performer,
    sort = TRUE,
    name = "appearances") |>
  filter(appearances >= 70)
```

```
# A tibble: 5 x 3
  title           performer      appearances
  <chr>          <chr>            <int>
1 Heat Waves     Glass Animals     91
2 Blinding Lights The Weeknd      90
3 Radioactive    Imagine Dragons   87
4 Sail           AWOLNATION       79
5 I'm Yours      Jason Mraz      76
```

## E Using `case_when`

In the Military Surplus Cleaning chapter we created a `control_type` variable, creating a category to designate if our items have to be returned to the Department of Defense for disposal. Refer back to that chapter about why we need to do this.

I skipped the detailed explanation of the code there because it is a more advanced tactic and would mire the flow of the lesson. In this appendix, I go into the details for those who might be interested.

You can also learn more about `case_when()` and other categorization techniques in the [Jedr Categorization Tutorial](#).

### E.1 Catching up with the data

I'll start here with the Military Surplus data after the shipment totals have been calculated.

```
Warning: package 'ggplot2' was built under R version 4.3.1
```

```
# A tibble: 6 x 12
  state agency_name  nsn    item_name quantity ui      acquisition_value demil_code
  <chr> <chr>       <chr> <chr>        <dbl> <chr>           <dbl> <chr>
1 AL    ABBEVILLE P~ 1005~ MOUNT,RI~      10 Each          1626   D
2 AL    ABBEVILLE P~ 1240~ SIGHT,RE~       9 Each           371    D
3 AL    ABBEVILLE P~ 2320~ TRUCK,UT~       1 Each          62627   C
4 AL    ABBEVILLE P~ 2540~ BALLISTI~      10 Kit          17265.  D
5 AL    ABBEVILLE P~ 2320~ TRUCK,UT~       1 Each          62627   C
6 AL    ABBEVILLE P~ 2355~ MINE RES~       1 Each          658000  C
# i 4 more variables: demil_ic <dbl>, ship_date <dttm>, station_type <chr>,
#   total_value <dbl>
```

Rows: 99,052

Columns: 12

```
$ 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", "2320-01-371~
$ item_name            <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~
$ quantity              <dbl> 10, 9, 1, 10, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ui                    <chr> "Each", "Each", "Each", "Kit", "Each", "Each", "Each~
$ acquisition_value     <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~
$ demil_code            <chr> "D", "D", "C", "D", "C", "Q", "D", "C", "D", "D~
$ demil_ic              <dbl> 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ ship_date              <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~
$ station_type          <chr> "State", "State", "State", "State", "State", "State"~
$ total_value            <dbl> 16260.0, 3339.0, 62627.0, 172647.1, 62627.0, 658000.~

```

### E.1.1 Categorization logic with `case_when()`

We will use the `mutate()` function to create a new column called `control_type`. We've used `mutate` before, but this time we will fill in values in the new column based on other data inside each row. `case_when()` allows us to create a test (or a number of tests) and then mark the new value based on the answer. Once new data has been written the function evaluates the next row, so we write the most specific rules first.

I usually approach this by thinking of the logic first, then writing some code, then testing it. Sometimes my logic is faulty and I have to try again, which is why we test the results. Know this could go on for many cycles.

Here is the basic logic:

- We want to create a new column to denote if the item is controlled. In that column we want it to be TRUE when an item is controlled, and FALSE when it is not.
- We know that items with “AIRPLANE” are always controlled, no matter their demil designations.
- Otherwise we know that items that have a `demil_code` of “A”, OR a `demil_code` of “Q” AND a `demil_id` of “6”, are non-controlled.
- Everything else is controlled.

I've noted this logic in a specific order for a reason: It's the order that we write the logic in the code based on how the function `case_when()` works. This process is powerful and can get complicated depending on the logic needed. This example is perhaps more complicated than I like to use when explaining this concept, but this is real data and we *need* this, so here we go.

Here is the code and annotations.

```

leso_control <- leso_total |>
  mutate(
    control_type = case_when(
      str_detect(item_name, "AIRPLANE") ~ TRUE,
      (demil_code == "A" | (demil_code == "Q" & demil_ic == 6)) ~ FALSE,
      TRUE ~ TRUE
    )
  )

leso_control |> glimpse()
```

- ① Our first line creates a new tibble `leso_control` and fills it with the result of the rest of our expression. We start with the `leso_total` tibble.
- ② We mutate the data and start with the name of new column: `control_type`. We are filling that column with the result of the `case_when()` function for each row. Within the `case_when()` we are making the determination if the item is controlled or not. The left side of the `~` is the test, and the right side of `~` is what we enter into the column if the test passes. But we have more than one test:
- ③ The first test is we use the `str_detect()` function to look inside the `item_name` field looking for the term “AIRPLANE”. If the test finds the term, then the `control_type` field gets a value of `TRUE` and we move to the next row. If not, it moves to the next rule to see if that is a match. (We could fill this column with any text or number we want, but we are using `TRUE` and `FALSE` because that is the most basic kind of data to keep. If the item is controlled, set the value is `TRUE`. If not, it should be set to `FALSE`.)
- ④ Our second rule has two complex tests and we want to mark the row `FALSE` if either are true. (Remember, this is based on what the DLA told me: items with A or Q6 are non-controlled.) Our `case_when()` logic first looks for the value “A” in the `demil_code` field. If it is yes, then it marks the row `FALSE`. If no it goes to the next part: Is there a “Q” in the `demil_code` field AND a “6” in the `demil_ic` field? Both “Q” and “6” have to be there to get marked as `FALSE`. If both fail, then we move to the next test.
- ⑤ The last test is our catch-all. If none of the other rules apply, then mark this row as `TRUE`, which means it is controlled. So our default in the end is to mark everything `TRUE` if any of the other rules don’t mark it first.
- ⑥ Lastly we glimpse at the data just so we can see the column was created.

```

Rows: 99,052
Columns: 13
$ state              <chr> "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", "2320-01-371~
$ item_name          <chr> "MOUNT,RIFLE", "SIGHT,REFLEX", "TRUCK,UTILITY", "BAL~
$ quantity           <dbl> 10, 9, 1, 10, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
```

```
$ ui <chr> "Each", "Each", "Each", "Kit", "Each", "Each", "Each-  
$ acquisition_value <dbl> 1626.00, 371.00, 62627.00, 17264.71, 62627.00, 65800~  
$ demil_code <chr> "D", "D", "C", "D", "C", "C", "Q", "D", "C", "D", "D~  
$ demil_ic <dbl> 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~  
$ ship_date <dttm> 2016-09-19, 2016-09-14, 2016-09-29, 2018-01-30, 201~  
$ station_type <chr> "State", "State", "State", "State", "State", "State"~  
$ total_value <dbl> 16260.0, 3339.0, 62627.0, 172647.1, 62627.0, 658000.~  
$ control_type <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE~
```

As I said, there was a lot of trial and error to figure that out, but I'll show some tests here to show that we did what we were intending.

This shows airplanes are marked as controlled with TRUE.

```
# showing the results and some columns that determined them
leso_control |>
  select(item_name, demil_code, demil_ic, control_type) |>
  filter(str_detect(item_name, "AIRPLANE"))
```

```
# A tibble: 12 x 4
  item_name          demil_code demil_ic control_type
  <chr>              <chr>      <dbl>    <lgl>
1 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
2 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
3 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
4 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
5 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
6 AIRPLANE,CARGO-TRANSPORT A             1 TRUE
7 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
8 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
9 AIRPLANE,CARGO-TRANSPORT Q             6 TRUE
10 AIRPLANE,CARGO-TRANSPORT Q            6 TRUE
11 AIRPLANE,CARGO-TRANSPORT Q            6 TRUE
12 AIRPLANE,FLIGHT T42A                 Q             3 TRUE
```

This shows how many items are marked TRUE vs FALSE for each demil\_code and demil\_ic combination. I used it to check that most A records were FALSE, along with Q6.

```
leso_control |>  
  count(demil_code, demil_ic, control_type, name = "cnt") |> ①  
  pivot_wider(names_from = control_type, values_from = cnt) ②
```

- ① This is the count shortcut using three variables. I name counted column `cnt`. It's hard to visualize that result here but it counts how many times each unique combination of `demil_code`, `demil_ic` and `control_type`.
- ② Here I use `pivot_wider()` to show the TRUE and FALSE counts on the same row. Just makes it easier to see.

```
# A tibble: 29 x 4
  demil_code demil_ic `FALSE` `TRUE`
  <chr>       <dbl>    <int>   <int>
1 A             0        1     NA
2 A             1      4391      1
3 A             4        1     NA
4 A             7      155     NA
5 A            NA      2770     NA
6 B             3        NA     20
7 B            NA        NA     86
8 C             1        NA    5316
9 C             4        NA      1
10 C            7        NA    221
# i 19 more rows
```

We're done with this extended explanation.

# F Troubleshooting

Programming in R (or any language) can be frustrating for beginners. Sometimes the expected does not happen and the unexpected does happen. Programming also breeds a constant state of learning because there are always new challenges and new solutions to find or adapt.

This troubleshooting guide is a collaboration and combination of other sources. Some major hat tips to [Andrew Tran](#) and [Nick HK](#) among many others.

Nick's [Common R Problems](#) is really worth reading.

## F.1 When things break

The first and best troubleshooting advice I can give it this:

**WRITE ONE LINE OF CODE. RUN IT. CHECK THE RESULTS. REPEAT.**

When you write and run code one line at a time, problems are easier to find.

Beyond that there are generally two categories of problems: You *wrote* “it” wrong, or you *used* “it” wrong. The first can sometimes be hard to see just like any typo.

Some tips:

1. **Read the error message** and look for words you recognize. You may not understand the error exactly, but words can be clues to what part of your code is wrong.
2. **Check the spelling** of variables, values and functions, especially if the error message says something like **object ‘wahtever’ not found**. If you are writing code that depends on matching strings and you are getting unexpected results, check those strings. Word case (as in Title case) can matter.
3. **Check the code** for balanced parenthesis and other punctuation problems. RStudio will show you matching parenthesis with highlighting, and will indicate problems with red X icons and red underlines in your code. Writing beautiful, well-indenting can sometimes help avoid and spot these types of errors.

Beyond that, here are some common things students come across:

- **There is no package called ‘packagename’:** You are trying to use a library that is named wrong or you don’t have installed. You can install packages with `install.packages('packagename')` where ‘packagename’ is replaced with the name of the package you actually want.
- **Forgetting to use quotation marks when they are needed:** `install.packages("gclus")` will work, while `install.packages(gclus)` will generate an error.
- **Could not find function ‘functionname’:** You either misspelled the function or you are missing a `library()` in your setup. It’s best practice to have every `library()` loaded in a setup chunk near the top of the notebook.
- **Using the wrong case:** `help()`, `Help()`, and `HELP()` are three different functions (and only the first one will work)
- **Forgetting to include the parentheses in a function call:** `help()` rather than `help`. Even if there are no options, you still need the `()`.
- **Using the \ in a path name on Windows**” R sees the backlash character as an escape character. `setwd("c:\mydata")` will generate an error. Use `setwd("c:/mydata")` or `setwd("c:\\mydata")` instead.

## F.2 Learning how things work

There are infinite ways to write code incorrectly or use a function improperly. Documentation and experience (sometimes of others) are key to these challenges.

### F.2.1 Help docs

One way to find documentation is through the built-in **Help** function within RStudio. If you look at the pane at the bottom-right of RStudio, you’ll see tabs for “Files”, “Plots”, “Packages” and “Help”. Click on the **Help** tab.

You can type in a function or part of a function and get a list of items. If you search for “count” and hit return you’ll get documentation on how to use it. It takes some getting used to in reading the docs, but the examples at the bottom are often useful.

There are also some Console commands to find things:

---

| Function                 | Action                                                  |
|--------------------------|---------------------------------------------------------|
| <code>help.start</code>  | General help                                            |
| <code>help("foo")</code> | Help on function foo (the quotation marks are optional) |
| or <code>?foo</code>     |                                                         |

---

| Function                        | Action                                                      |
|---------------------------------|-------------------------------------------------------------|
| <code>help.search("foo")</code> | Searches the help system for instances of the string foo    |
| <code>??foo</code>              |                                                             |
| <code>example("foo")</code>     | Examples of function foo (the quotation marks are optional) |

## F.2.2 Good Googling

Another way to get help is to Google for it, but there is an art to it especially since there are other data science languages and programs with similar terms as R. Some tips:

- Use “in R” in your search: *How to merge data frames in R*
- Use the name of the package if you know it: *Add labels with ggplot*
- Use “tidyverse” if appropriate: *convert text to date with tidyverse*

There are plenty of Stack Overflow answers along with different tutorials from blogs and such. It is a well-used language, so there are lots of answers to help. Too many, sometimes.

## F.2.3 Tidyverse docs and cheatsheets

It is worth becoming familiar with the [tidyverse](#) site. Click on each icon from the home page to learn what each package does. R is also big on [cheatsheets](#), which are printable pages that go through all the verbs. They can be a bit much at first, but useful once you use R more.

We’ll try to put together a list of other resources and tutorials. You can find some [I’ve collected already here](#).

# F.3 R Frequently asked Questions

## F.3.1 Functions, objects and variables

The names of things and how they are used are important in R, and can cause confusion. The term **date** could represent any number of things in R code depending on how you are using it, and that can be confusing. Knowing the difference between functions, objects and variables and how they are referenced in code helps.

- **Variables** are like the column names from a spreadsheet table. If you think of a data frame (or tibble in R) as a spreadsheet table, then when you reference a “variable” name, it is that column. A data frame of police calls might have a **date** column that has the date/time of the call in each row, like “2021-01-06 17:29:38”.

- **Functions** are collections of code that solve specific problems. In R, they are always followed by parenthesis, like this: `date()`, which is a function to pull only the date from a date/time variable. There are often arguments inside a function, like `date(date)` could be a function pull the date “2021-01-06” from a date/time variable called `date`. Knowing that functions always have parenthesis is a good clue to help figure out how something is being used.
- **Objects** are stored values in R. You can name objects anything you like, including unwise things like `date`, since that is already a function and maybe a variable.

That is all to prove it can be confusing. When you have a chance to name things, make good choices.

### F.3.1.1 Naming things

Be thoughtfully obvious about the names you choose for objects and variables.

- Avoid naming things with what could be a function name.
- Avoid spaces. Use an underscore `_` or dash `-` instead. I use underscores for naming things in code, but dashes for naming files or folders that could become URLs at some point.
- Be descriptive. Name things what they are, like `police_calls`.
- Be consistent. If you have multiple date variables like `open_date` and `close_date` then it is easy to know and `select()` them.

### F.3.2 Some R code basics

- `<-` is known as an “assignment operator” – it means “Make the object named to the left equal to the output of the code to the right”
- `=` makes an object equal to a value, which is similar to `<-` but used within a function.
- `==` tests whether the objects on either end are equal. This is often used in filtering data.
- `&` means AND, in Boolean logic
- `|` means OR, in Boolean logic
- `!` means NOT, in Boolean logic
- When referring to values entered as text, or to dates, put them in quote marks like this: `"United States"`, or `"2016-07-26"`. Numbers are not quoted
- When entering two or more values as a list, combine them using the function `c`, for combine, with the values separated by commas, for example: `c("2017-07-26", "2017-08-04")`
- As in a spreadsheet, you can specify a range of values with a colon, for example: `c(1:10)` creates a list of integers (whole numbers) from one to ten.
- Some common operators:
  - + – add, subtract

- \* / multiply, divide
  - > < greater than, less than
  - >= <= greater than or equal to, less than or equal to
  - != not equal to
- Handling null values:
    - Nulls are designated as NA
    - `is.na(x)` looks for nulls within variable x.
    - `!is.na(x)` looks for non-null values within variable x

Here, `is.na()` is a **function**.

# G Exploring 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.

## G.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 R notebooks with this list.

## G.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.

```

```{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	Length:327895
Class :character	Class :character	1st Qu.: 25.5	Class :character	Class :character	Class :character
Mode :character	Mode :character	Median : 50.0	Mode :character	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	Min. : 1.000
Class :character	1st Qu.: 1.000	1st Qu.: 23.0	1st Qu.: 14.00	1st Qu.: 4.000	1st Qu.: 4.000
Mode :character	Median : 1.000	Median : 47.0	Median : 39.00	Median : 7.000	Median : 7.000
	Mean : 1.073	Mean : 47.6	Mean : 41.36	Mean : 9.154	Mean : 9.154
	3rd Qu.: 1.000	3rd Qu.: 72.0	3rd Qu.: 66.00	3rd Qu.: 13.000	3rd Qu.: 13.000
	Max. :10.000	Max. :100.0	Max. :100.0	Max. :100.0	Max. :87.000
	NA's :31954				


```

Figure G.1: Summary of billboard data

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.

### G.3 Learn more through cleaning

The act of inspecting your data, cleaning data types and poking around for problems can help you learn more about it. Some tips to make sure your data is ready to analyze:

- Column names should be short, descriptive and clear. Save typing by making them all lowercase with parts separated by underscores. (You can use the `clean_names()` functions for this). Use patterns for related variables like `_date` or `_address` parts.
- Fix all your data types. Make sure numbers are numbers (can use `parse_number()` to fix) and dates into real dates (see [lubridate](#)) and maybe create date parts for grouping. Make sure number-oriented IDs are text if they are supposed to start with zero. (ZIP codes are like this ... not really a number, but category.)
- Remove the cruft of columns you don't need using `select()`. Make sure you understand what you are removing, but feel secure that you can come back and add them back if you find you need them.

### G.3.1 Cleaning up categorical data

If you are going to count or 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, perhaps using “`case_`” functions like `case_match()` or `case_when()` to rename specific values in your data.

If you find you have hundreds of values to clean, there are some other tools like [OpenRefine](#) you can learn fairly quickly to help.

## G.4 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.

## G.5 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:

### G.5.1 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     |

## G.5.2 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    | 9201 CIRCUIT OF THE AMERICAS | 20714630    |
| SPORTS          | BLVD                         |             |
| W HOTEL AUSTIN  | 200 LAVACA ST                | 15435458    |
| ROSE ROOM/ 77   | 11500 ROCK ROSE AVE          | 14726420    |
| DEGREE          |                              |             |
| THE DOGWOOD     | 11420 ROCK ROSE AVE STE 700  | 14231072    |
| DOMAIN          |                              |             |

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)`.

## G.5.3 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
  )
```

## G.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.

## G.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)
```

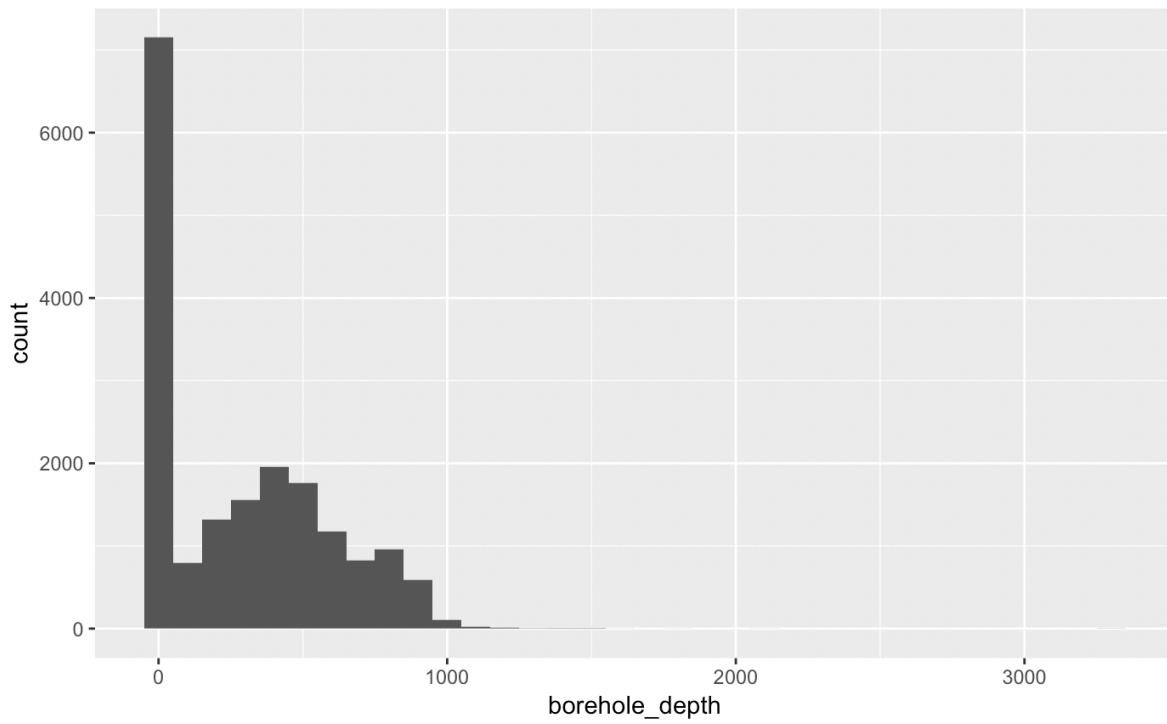


Figure G.2: Borehole depth histogram

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.

### G.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.

## G.8 Same ideas using spreadsheets

Check out [this resource by David Eads](#) on the same topic, with some more specifics about Google Sheets.

# H Chart production tips

By Christian McDonald

Building charts is a part of the data exploration process, and often the only audience is ourselves or our editors. We're just trying to learn something about our data. A title might help us quickly understand what we plotted, but we might not sweat every axis name and every detail.

But we can also build charts for others, like our readers or an audience that hasn't been mired in the data. It may run with a story or be part of a report. With these you are specifically trying to communicate a finding from your data in far more detail to someone who has far less context to begin with. Whether you are building this in ggplot, Datawrapper or some other tool, you must take more care to be thorough, accurate and communicative.

## H.1 Titles, descriptions and annotations

Chart titles and descriptions can be some of the most difficult writing you can do as a journalist. You don't want to describe the steps of your analysis nor say the obvious, but you do need to give the reader all the relevant detail needed to understand the chart. **Write titles, descriptions and annotations as if the chart stands alone, and a reader knows nothing before viewing it.**

Some tips paraphrased from Nathan Yau's *Data Points* book ...

- The title – typically larger and bolder fonts — sets the stage or describes what people should see or look for in the data. A descriptive title also helps. For example, “Rising Gas Prices” says more about the chart than just “Gas Prices.” By including “Rising”, it presents the conclusion immediately, and readers will look to the chart to verify and see the details. Saying just “Gas Prices” leaves the data interpretation to the readers and places them in the exploration phase.
- The description or lead-in text is used to prepare readers for what a chart shows, but in further detail. The text expands on what the title declares, where the data is from, how it was derived, **or what it means** (best charts do this, says @crit). Basically, it's information that might help others understand the data better but often doesn't directly point to the specific elements.

- To explain specifics points or areas, you can use lines and arrows as an annotation layer on top of a chart. This places descriptions directly in the context of the data so that a reader doesn't have to look outside a graph for additional information to fully understand what you show.

## H.2 Other considerations

- Proper data encodings and visual cues: Think about what you are trying to convey with the graphic and plot your data in a fashion that furthers that understanding. (See Nathan Yau's *Data Points*, Chap. 3.)
- Legends for encodings: If your plots include labels for categories on the chart, you may not need a separate legend, but be sure readers can distinguish items.
- Labels for axis: They help describe the value being plotted. In some obvious cases where the meaning is clear, like years, they may be dropped.
- Include unit values to further understanding: Sometimes the value can be added to the plot itself, other times grid lines may be enough.
- Annotations: Add explanations to the plot if they help readers understand nuance of what you are trying to convey.
- Source of the data: This is the source of the data, not the delivery method. (i.e., The Comptroller of Public Accounts, not data.texas.gov.)
- Your byline: Credit yourself and your publications.