Data Journalism with R and the Tidyverse

Code, data and visuals for storytellers

Matt Waite & Sarah Cohen (original authors); updated by Derek Willis and Sean Mu

2024-08-13

Table of contents

1 Introduction

If you were at all paying attention in pre-college science classes, you have probably seen this equation:

```
d = rt or distance = rate*time
```

In English, that says we can know how far something has traveled if we know how fast it's going and for how long. If we multiply the rate by the time, we'll get the distance.

If you remember just a bit about algebra, you know we can move these things around. If we know two of them, we can figure out the third. So, for instance, if we know the distance and we know the time, we can use algebra to divide the distance by the time to get the rate.

```
d/t = r or distance/time = rate
```

In 2012, the South Florida Sun Sentinel found a story in this formula.

People were dying on South Florida tollways in terrible car accidents. What made these different from other car fatal car accidents that happen every day in the US? Police officers driving way too fast were causing them.

But do police regularly speed on tollways or were there just a few random and fatal exceptions?

Thanks to Florida's public records laws, the Sun Sentinel got records from the toll transponders in police cars in south Florida. The transponders recorded when a car went through a given place. And then it would do it again. And again.

Given that those places are fixed – they're toll plazas – and they had the time it took to go from one toll plaza to another, they had the distance and the time.

It took high school algebra to find how fast police officers were driving. And the results were shocking.

Twenty percent of police officers had exceeded 90 miles per hour on toll roads. In a 13-month period, officers drove between 90 and 110 mph more than 5,000 times. And these were just instances found on toll roads. Not all roads have tolls.

The story was a stunning find, and the newspaper documented case after case of police officers violating the law and escaping punishment. And, in 2013, they won the Pulitzer Prize for Public Service.

All with simple high school algebra.

1.1 Modern data journalism

It's a single word in a single job description, but a Buzzfeed job posting in 2017 is another indicator in what could be a profound shift in how data journalism is both practiced and taught.

"We're looking for someone with a passion for news and a commitment to using data to find amazing, important stories — both quick hits and deeper analyses that drive conversations," the posting seeking a data journalist says. It goes on to list five things BuzzFeed is looking for: Excellent collaborator, clear writer, deep statistical understanding, knowledge of obtaining and restructuring data.

And then there's this:

"You should have a strong command of at least one toolset that (a) allows for filtering, joining, pivoting, and aggregating tabular data, and (b) enables reproducible workflows."

This is not the data journalism of 20 years ago. When it started, it was a small group of people in newsrooms using spreadsheets and databases. Data journalism now encompases programming for all kinds of purposes, product development, user interface design, data visualization and graphics on top of more traditional skills like analyzing data and writing stories.

In this book, you'll get a taste of modern data journalism through programming in R, a statistics language. You'll be challenged to think programmatically while thinking about a story you can tell to readers in a way that they'll want to read. They might seem like two different sides of the brain – mutually exclusive skills. They aren't. I'm confident you'll see programming is a creative endeavor and storytelling can be analytical.

Combining them together has the power to change policy, expose injustice and deeply inform.

1.2 Installations

This book is all in the R statistical language. To follow along, you'll go here and download and install both R the language and RStudio.

Going forward, you'll see passages like this:

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

That is code that you'll need to run in your RStudio. When you see that, you'll know what to do.

1.3 About this book

This book is the collection of class materials originally written for Matt Waite's Data Journalism class at the University of Nebraska-Lincoln's College of Journalism and Mass Communications. It has been substantially updated by Derek Willis and Sean Mussenden for data journalism classes at the University of Maryland Philip Merrill College of Journalism, with contributions from Sarah Cohen of Arizona State University.

There's some things you should know about it:

- It is free for students.
- The topics will remain the same but the text is going to be constantly tinkered with.
- What is the work of the authors is copyright Matt Waite 2020, Sarah Cohen 2022 and Derek Willis, Daniel Trielli and Sean Mussenden 2024.
- The text is Attribution-NonCommercial-ShareAlike 4.0 International Creative Commons licensed. That means you can share it and change it, but only if you share your changes with the same license and it cannot be used for commercial purposes. I'm not making money on this so you can't either.

- As such, the whole book – authored in Quarto – in its original form is open sourced on Github. Pull requests welcomed!

1.4 What we'll cover

- Public records and open data
- R Basics
- Replication
- Data basics and structures
- Aggregates
- Mutating
- Working with dates
- Filters
- Cleaning I: Data smells
- Cleaning II: Janitor
- Cleaning III: Open Refine

- Cleaning IV: Pulling Data from PDFs
- Joins
- Basic data scraping
- Getting data from APIs: Census
- Visualizing for reporting: Basics
- Visualizing for reporting: Publishing
- Geographic data basics
- Geographic queries
- Geographic visualization
- Text analysis basics
- Writing with and about data
- Data journalism ethics
- AI & Data journalism

2 Learn a new way to read

Getting started in data journalism often feels as if you've left the newsroom and entered the land of statistics, computer programming and data science. This chapter will help you start seeing data reporting in a new way, by learning how to study great works of the craft as a writer rather than a reader.

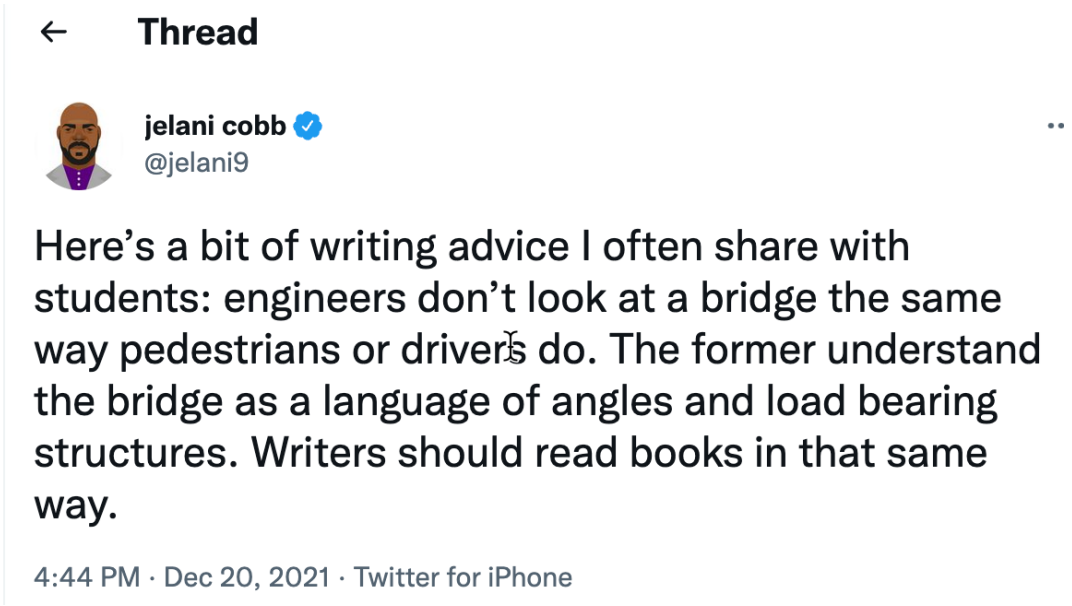


Figure 2.1: jelani cobb

Jelani Cobb tweeted, "an engineer doesn't look at a bridge the same way pedestrians or drivers do." They see it as a "language of angles and load bearing structures." We just see a bridge. While he was referring to long-form writing, reporting with data can also be learned by example – if you spend enough time with the examples.

Almost all good writers and reporters try to learn from exemplary work. I know more than one reporter who studies prize-winning journalism to hone their craft. This site will have plenty of examples, but you should stay on the lookout for others.

2.1 Read like a reporter

Try to approach data or empirical reporting as a reporter first, and a consumer second. The goal is to triangulate how the story was discovered, reported and constructed. You'll want to think about why *this* story, told this way, at this time, was considered newsworthy enough to publish when another approach on the same topic might not have been.

What were the questions?

In data journalism, we often start with a tip, or a hypothesis. Sometimes it's a simple question. Walt Bogdanich of The New York Times is renowned for seeing stories around every corner. Bogdanich has said that the prize-winning story "A Disability Epidemic Among a Railroad's Retirees" came from a simple question he had when railway workers went on strike over pension benefits – how much were they worth? The story led to an FBI investigation and arrests, along with pension reform at the largest commuter rail in the country.

The hypothesis for some stories might be more directed. In 2021, the Howard Center for Investigative Journalism at ASU published "Little victims everywhere", a set of stories on the lack of justice for survivors of child sexual assault on Native American reservations. That story came after previous reporters for the center analyzed data from the Justice Department showing that the FBI dropped most of the cases it investigated, and the Justice Department then only prosecuted about half of the matters referred to it by investigators. The hypothesis was that they were rarely pursued because federal prosecutors – usually focused on immigration, white collar crime and drugs – weren't as prepared to pursue violent crime in Indian Country.

When studying a data-driven investigation, try to imagine what the reporters were trying to prove or disprove, and what they used to do it. In journalism, we rely on a mixture of quantitative and qualitative methods. It's not enough to prove the "numbers" or have the statistical evidence. That is just the beginning of the story. We are supposed to ground-truth them with the stories of actual people and places.

Go beyond the numbers

It's easy to focus on the numbers or statistics that make up the key findings, or the reason for the story. Some reporters make the mistake of thinking all of the numbers came from the same place – a rarity in most long-form investigations. Instead, the sources have been woven together and are a mix of original research and research done by others. Try to pay attention to any sourcing done in the piece. Sometimes, it will tell you that the analysis was original. Other times it's more subtle.

But don't just look at the statistics being reported in the story. In many (most?) investigations, some of the key people, places or time elements come directly from a database.

When I was analyzing Paycheck Protection Program loan data for ProPublica, one fact hit me as I was looking at a handful of sketchy-looking records: a lot of them were from a single county in coastal New Jersey. It turned out to be a pretty good story.

Often, the place that a reporter visits is determined by examples found in data. In this story on rural development funds, all of the examples came from an analysis of the database. Once the data gave us a good lead, the reporters examined press releases and other easy-to-get sources before calling and visiting the recipients or towns.

2.2 Reading tips

You'll get better at reading investigations and data-driven work over time, but for now, remember to go beyond the obvious:

- Where might the reporters have found their key examples, and what made them good characters or illustrations of the larger issue? Could they have come from the data?
- What do you think came first – a narrative single example that was broadened by data (naively, qualitative method), or a big idea that was illustrated with characters (quantitative method)?
- What records were used? Were they public records, leaks, or proprietary data?
- What methods did they use? Did they do their own testing, use statistical analysis, or geographic methods? You won't always know, but look for a methodology section or a description alongside each story.
- How might you localize or adapt these methods to find your own stories?
- Pick out the key findings (usually in the nut graf or in a series of bullets after the opening chapter): are they controversial? How might they have been derived? What might have been the investigative hypothesis? Have they given critics their due and tried to falsify their own work?
- How effective is the writing and presentation of the story? What makes it compelling journalism rather than a dry study? How might you have done it differently? Is a video story better told in text, or would a text story have made a good documentary? Are the visual elements well integrated? Does the writing draw you in and keep you reading? Think about structure, story length, entry points and graphics all working together.
- Are you convinced? Are there holes or questions that didn't get addressed?

2.3 Analyze data for story, not study

As journalists we'll often be using data, social science methods and even interviewing differently than true experts. We're seeking stories, not studies. Recognizing news in data is one of the hardest skills for less experienced reporters new to data journalism. This list of potential newsworthy data points is adapted from Paul Bradshaw's "Data Journalism Heist".

LAST UPDATED ON 2015-06-10

- Compare the claims of powerful people and institutions against facts – the classic investigative approach.
- Report on *unexpected* highs and lows (of change, or of some other characteristic)

- Look for outliers – individual values that buck a trend seen in the rest
- Verify or bust some myths
- Find signs of distress, happiness or dishonesty or any other emotion.
- Uncover *new* or *under-reported* long-term trends.
- Find data suggesting your area is *the same* or *different* than most others of its kind.

Bradshaw also did a recent study of data journalism pieces: "Here are the angles journalists use most often to tell the stories in data", in Online Journalism Blog. I'm not sure I agree, only because he's looking mainly at visualizations rather than stories, but they're worth considering.

2.4 Exercises

- If you're a member of Investigative Reporters and Editors (and you should be, it's cheap for students), go to the site and find a recent prize-winning entry (usually text rather than broadcast). Get a copy of the IRE contest entry from the Resources page. Try to match up what the reporters said they did and how they did it with key portions of the story.
- The next time you find a good data source, try to find a story that references it. If your data is local, you might look for a story that used similar data elsewhere, such as 911 response times or overdose deaths. But many stories use federal datasets that can easily be localized. Look at a description of the dataset and then the story to see how the data might have been used.

3 Newsroom math

Statistics are people with the tears washed off

- Paul Brodeur

Jo Craven McGinty, then of The New York Times, used simple rates and ratios to discover that a 6-story brick New Jersey hospital was the most expensive in the nation. In 2012, Bayonne Medical Center "charged the highest amounts in the country for nearly one-quarter of the most common hospital treatments," the Times story said.

To do this story, McGinty only needed to know the number of the procedures reported to the government and the total amount each hospital charged. Dividing those to find an average price, then ranking the most common procedures, led to this surprising result.

3.1 Why numbers?

Using averages, percentages and percent change is the bread and butter of data journalism, leading to stories ranging from home price comparisons to school reports and crime trends. It may have been charming at one time for reporters to announce that they didn't "do" math, but no longer. Instead, it is now an announcement that the reporter can only do some of the job. You will never be able to tackle complicated, in-depth stories without reviewing basic math.

The good news is that most of the math and statistics you need in a newsroom isn't nearly as difficult as high school algebra. You learned it somewhere around the 4th grade. You then had a decade to forget it before deciding you didn't like math. But mastering this most basic arithmetic again is a requirement in the modern age.

In working with typical newsroom math, you will need to learn how to:

- Overcome your fear of numbers
- Integrate numbers into your reporting
- Routinely compute averages, differences and rates
- Simplify and select the right numbers for your story

While this chapter covers general tips, you can find specific instructions for typical newsroom math in this Appendix A

3.2 Overcoming your fear of math

When we learned to read, we got used to the idea that 26 letters in American English could be assembled into units that we understand without thinking – words, sentences, paragraphs and books. We never got the same comfort level with 10 digits, and neither did our audience.

Think of your own reaction to seeing a page of words. Now imagine it as a page of numbers.

Instead, picture the number "five". It's easy. It might be fingers or it might be a team on a basketball court. But it's simple to understand.

Now picture the number 275 million. It's hard. Unfortunately, 275 billion isn't much harder, even though it's magnitudes larger. (A million seconds goes by in about 11 days but you almost surely have not have been alive for a billion seconds – about 36 years.)

The easiest way to get used to some numbers is to learn ways to cut them down to size by calculating rates, ratios or percentages. In your analysis, keep an eye out for the simplest *accurate* way to characterize the numbers you want to use. "Characterize" is the important word here – it's not usually necessary to be overly precise so long as your story doesn't hinge on a nuanced reading of small differences. (And is anything that depends on that news? It may not be.)

Here's one example of putting huge numbers in perspective. Pay attention to what you really can picture - it's probably the $21 equivalent.

The Chicago hedge fund billionaire Kenneth C. Griffin, for example, earns about $68.5 million a month after taxes, according to court filings made by his wife in their divorce. In 2016, he had given a total of $300,000 to groups backing Republican presidential candidates. That is a huge sum on its face, yet is the equivalent of only $21.17 for a typical American household, according to Congressional Budget Office data on after-tax income. *"Buying Power", Nicholas Confessore, Sarah Cohen and Karen Yourish, The New York Times, October 2015*

Originally the reporters had written it even more simply, but editors found the facts so unbelievable that they wanted give readers a chance to do the math themselves. That's reasonable, but here's an even simpler way to say it: "earned nearly $1 billion after taxes…He has given $300,000 to groups backing candidates, the equivalent of a dinner at Olive Garden for the typical American family, based on Congressional Budget Office income data." (And yes, the reporter checked the price for an Olive Garden meal at the time for four people.)

3.3 Put math in its place

For journalists, numbers – or facts – make up the third leg of a stool supported by human stories or anecdotes, and insightful comment from experts. They serve us in three ways:

- ***As summaries***. Almost by definition, a number counts something, averages something, or otherwise summarizes something. Sometimes, it does a good job, as in the average height of Americans. Sometimes it does a terrible job, as in the average income of Americans. Try to find summaries that accurately characterize the real world.
- ***As opinions***. Sometimes it's an opinion derived after years of impartial study. Sometimes it's an opinion tinged with partisan or selective choices of facts. Use them accordingly.
- ***As guesses***. Sometimes it's a good guess, sometimes it's an off-the-cuff guess. And sometimes it's a hopeful guess. Even when everything is presumably counted many times, it's still a (very nearly accurate) guess. Yes, the "audits" of presidential election results in several states in 2021 found a handful of errors – not a meaningful number, but a few just the same.

Once you find the humanity in your numbers, by cutting them down to size and relegating them to their proper role, you'll find yourself less fearful. You'll be able to characterize what you've learned rather than numb your readers with every number in your notebook. You may even find that finding facts on your own is fun.

3.4 Going further

Tipsheets

- Steve Doig's "Math Crib Sheet"
- Appendix A: Common newsroom math, adapted from drafts of the book *Numbers in the Newsroom*, by Sarah Cohen.

Reading and viewing

- "Avoiding Numeric Novocain: Writing Well with Numbers," by Chip Scanlan, Poynter.com
- T. Christian Miller's "Writing the data-driven story"
- A viral Twitter thread:

3.5 Exercises

- Imagine that someone gave you $1 million and you could spend it on anything you want. Write down a list of things that would add up to about that amount. That should be easy. Now, imagine someone gave you $1 billion and you could spend it on whatever you want, but anything left over after a year had to be returned. How would you spend it? (You can give away money, but it can't be more than 50% of a charity's annual revenues. So you can't give 10 $100 million gifts!) See how far you get trying to spend it. A few homes, a few yachts, student loan repayments for all of your friends? You've hardly gotten started.

4 Defining "Data"

> data / de .tə/ :
>
> information in an electronic form that can be stored and used by a computer, or information, especially facts or numbers, collected to be examined and >considered and used to help decision-making
>
> – Cambridge Dictionary – sort of [1]

4.1 The birth of a dataset

Most journalism uses data collected for one purpose for something entirely different. Understanding its original uses – what matters to the people who collected it, and what doesn't – will profoundly affect its accuracy or usefulness.

Trace data and administrative records

In "The Art of Access", David Cullier and Charles N. Davis describe a process of tracking down the life and times of a dataset. Their purpose is to make sure they know how to request it from a government agency. The same idea applies to using data that we acquire elsewhere.

Understanding how and why data exists is crucial to understanding what you, as a reporter, might do with it.

Anything you can systematically search or analyze could be considered one piece of of data. As reporters, we usually deal with data that was created in the process of doing something else – conducting an inspection, delivering a tweet, or scoring a musical. In the sciences, this flotsam and jetsom that is left behind is called "digital trace data" if it was born digitally.

In journalism and in the social sciences, many of our data sources were born during some government process – a safety inspection, a traffic ticket, or the filing of a death certificate. These administrative records form the basis of much investigative reporting and they are often the subject of public records and FOIA requests. They were born as part of the government doing its job, without any thought given to how it might be used in another way. In the sciences, those are often called "administrative records".

[1] I flipped the order of these two definitions!

This trace data might be considered the first part of the definition above – information that can be stored and used.

Here's how Chris Bail from Duke University describes it.

Data collected and curated for analysis

Another kind of data is that which is compiled or collected specifically for the purpose of studying something. It might collected in the form of a survey or a poll, or it might be a system of sampling to measure pollution or weather. But it's there because the information has intrinsic value AS information.

The video suggests a hard line between trace data and custom data. In practice, it's not that clear. Many newsrooms may curate data published in other sources or in administrative records, such as the Washington Post's police shooting dataset. In other cases, the agencies we are covering get already-compiled data from state and local governments.

This type of data might be considered the second type in the definition – tabular information that is used for decision-making.

4.2 Granular and aggregated data

One of the hardest concepts for a lot of new data journalists is the idea of *granularity* of your data source. There are a lot of ways to think about this: individual items in a list vs. figures in a table; original records vs. compilations; granular data vs. statistics.

Generally, an investigative reporter is interested in getting data that is as close as possible to the most granular information that exists, at least on computer files. Here's an example, which might give you a little intuition about why it's so important to think this way:

When someone dies in the US, a standard death certificate is filled out by a series of officials - the attending physician, the institution where they died and even the funeral direcor.

Click on this link to see a blank version of the standard US death certificate form – notice the detail and the detailed instructions on how it is supposed to be filled out. [2]

A good reporter could imagine many stories coming out of these little boxes. Limiting yourself to just to COVID-19-related stories: You could profile the local doctor who signed the most COVID-19-related death certificates in their city, or examine the number of deaths that had

[2] You should do this whenever you get a dataset created from administrative records. That is, track down its origin and examine the pieces you were given and the pieces that were left out; look at what is written in free-form vs what is presented as a check box. You may need a copy of the template that an agency uses to collect the information, but many governments make these available on their websites or are willing to provide them without a fuss.

COVID as a contributing, but not underlying or immediate, cause of death. You could compare smoking rates in the city with the number of decedents whose tobacco use likely contributed to their death. Maybe you'd want to know how long patients suffered with the disease before they died. And you could map the deaths to find the block in your town most devastated by the virus.

Early in the pandemic, Coulter Jones and Jon Kamp examined the records from one of the few states that makes them public, and concluded that "Coronavirus Deaths were Likely Missed in Michigan, Death Certificates Suggest"

But you probably can't do that. The reason is that, in most states, death certificates are not public records and are treated as secrets. [3]. Instead, state and local governments provide limited statistics related to the deaths, usually by county, with no detail. That's the difference between granular data and aggregate data. Here are some of the typical (not universal) characteristics of each:

Granular	Aggregate
Intended for some purpose other than your work	Intended to be presented as is to the public
Many rows (records), few columns (variables)	Many columns (variables), few rows (records)
Requires a good understanding of the source	Explanatory notes usually come with the data
Easy to cross-reference and compile	Often impossible to repurpose
Has few numeric columns	May be almost entirely numerical
Is intended for use in a database	Is intended for use in a spreadsheet

We often have to consider the trade-offs. Granular data with the detail we need - especially when it involves personally identifiable information like names and addresses - can take months or years of negotiation over public records requests, even when the law allows it. It's often much easier to convince an agency to provide summarized or incomplete data. Don't balk at using it if it works for you. But understand that in the vast majority of cases, it's been summarized in a way that's lost information that could be important to your story.

4.3 Nouns

That brings us to one of the most important things you must find out about any data you begin to analyze: What "noun" does each row in a tabular dataset represent? In statistics, they might be called *observations* or *cases*. In data science, they're usually called *records*. Either

[3] See "Secrecy in Death Records: A call to action", by Megain Craig and Madeleine Davison, Journal of Civic Information, December 2020

way, every row must represent the same thing – a person, a place, a year, a water sample or a school. And you can't really do anything with it until you figure out what that is.

In 2015, Sarah Cohen did a story at The New York Times called "More Deportation Follow Minor Crimes, Records Show". The government had claimed it was only removing hardened criminals from the country, but our analysis of the data suggested that many of them were for minor infractions.

In writing the piece, they had to work around a problem in our data: the agency refused to provide them anything that would help us distinguish individuals from one another. All the reporters knew was that each row represented one deportation – not one person! Without a column, or *field* or a *variable* or an *attribute* for an individual – say, name and date of birth, or some scrambled version of an their DHS number – they had no way to even estimate how often people were deported multiple times. If you read the story, you'll see the very careful wording, except when they had reported out and spoken to people on the ground.

4.4 Further reading

- "Basic steps in working with data", the Data Journalism Handbook, Steve Doig, ASU Professor. He describes in this piece the problem of not knowing exactly how the data was compiled.
- "Counting the Infected" , Rob Gebelof on The Daily, July 8, 2020.
- "Spreadsheet thinking vs. Database thinking", by Robert Kosara, gets at the idea that looking at individual items is often a "database", and statistical compilations are often "spreadsheets".
- "Tidy Data", in the Journal of Statistical Software (linked here in a pre-print) by Hadley Wickham , is the quintessential article on describing what we think of as "clean" data. For our purposes, much of what he describes as "tidy" comes when we have individual, granular records – not statistical compilations. It's an academic article, but it has the underlying concepts that we'll be working with all year.

4.5 Exercises

- The next time you get a government statistical report, scour all of the footnotes to find some explanation of where the data came from. You'll be surprised how often they are compilations of administrative records - the government version of trace data.

5 Ethics in data journalism

This originally appeared on Open News in March 2013.

In 2009, a senior web editor asked me and another developer a question: could our development group build a new news application for Tampabay.com that displayed a gallery of mug shots? Stories about goofy crimes with strange mug shots were popular with readers. The vision, on the part of management, was a website that would display the mugshots collected every day from publicly available websites by two editors—well paid, professional editors with other responsibilities.

Newsrooms are many things. Alive. Filled with energy. Fueled by stress, coffee and profanity. But they are also idea factories. Day after day, ideas come from everywhere. From reporters on the beat. From editors reading random things. From who knows where. Some of them are brilliant. Some would never work. Most need more people and time than are available. And some are dumber than anyone cares to admit.

We thought this idea was nuts. Why would we pay someone, let alone an editor, to fetch mug shots from the Internet? Couldn't we do that with a scraper?

If only this were the most complex question we would face.

Because given enough time and enough creativity, scraping a mug shot website is easy. You need to recognize a pattern, parse some HTML and gather the pieces you need. At least that's how it should work. Police agencies that put mugs online usually buy software from a vendor. Apparently, those vendors enjoy making horrific, non-standard, broken-in-interesting-and-unique-ways HTML. You'll swear. A lot. But you'll grind it out. And that's part of the fun. Scraping isn't any fun with clean, semantic, valid HTML. And scraping mug shot websites, by that definition, is tons of fun.

The complexity comes when you realize the data you are dealing with represent real people's lives.

5.1 Problems

The first problem we faced, long before we actually had data, was that data has a life of its own. Because we were going to put this information in front of a big audience, Google was going to find it. That meant if we used our normal open door policy for the Googlebot, someone's mug

shot was going to be the first record in Google for their name, most likely. It would show up first because most people dont actively cultivate their name on the web for visibility in Google. It would show up first because we know how SEO works and they dont. It would show up first because our site would have more traffic than their site, and so Google would rank us higher.

And that record in Google would exist as long as the URL did. Longer when you consider the cached versions Google keeps.

That was a problem because here are the things we could not know:

- Was this person wrongly arrested?
- Was this person innocent?
- Were the charges dropped against this person?
- Did this person lie about any of their information?

5.2 The Googlebot

So it turned out to be very important to know the Googlebot. It's your friend … until it isn't. We went to our bosses and said words that no one had said to them before: we did not want Google to index these pages. In a news organization, the page view is the coin of the realm. It is — unfortunately — how many things are evaluated when the bosses ask if it was successful or not. So, with that in mind, Google is your friend. Google brings you traffic. Indeed, Google is your single largest referrer of traffic at a news organization, so you want to throw the doors open and make friends with the Googlebot.

But here we were, saying Google wasn't our friend and that we needed to keep the Googlebot out. And, thankfully, our bosses listened to our argument. They too didn't want to be the first result in Google for someone.

So, to make sure we were telling the Googlebot no, we used three lines of defense. We told it no in robots.txt and on individual pages as a meta tag, and we put the most interesting bits of data into a simple JavaScript wrapper that made it hard on the bot if the first two things failed.

The second solution had ramifications beyond the Googlebot. We decided that we were not trying to make a complete copy of the public record. That existed already. If you wanted to look at the actual public records, the sheriff's offices in the area had websites and they were the official keeper of the record. We were making browsing those images easy, but we were not the public record.

That freedom had two consequences: it meant our scrapers could, at a certain point and given a number of failures, just give up on getting a mug. Data entered by humans will be flawed. There will be mistakes. Because of that, our code would have to try and deal with that. Well,

there's an infinite number of ways people can mess things up, so we decided that since we were not going to be an exact copy of the public record, we could deal with the most common failures and dump the rest. During testing, we were getting well over 98% of mugs without having to spend our lives coding for every possible variation of typo.

The second consequence of the decision actually came from the newspapers lawyers. They asked a question that dumbfounded us: How long are you keeping mugs? We never thought about it. Storage was cheap. We just assumed we'd keep them all. But, why should we do that? If we're not a copy of the public record, we dont have to keep them. And, since we didnt know the result of each case, keeping them was really kind of pointless.

So, we asked around: How long does a misdemeanor case take to reach a judgement? The answer we got from various sources was about 60 days. From arrest to adjudication, it took about two months. So, at the 60 day mark, we deleted the data. We had no way of knowing if someone was guilty or innocent, so all of them had to go. We even called the script The Reaper.

We'd later learn that the practical impacts of this were nil. People looked at the day's mugs and moved on. The amount of traffic a mug got after the day of arrest was nearly zero.

5.3 Data Lifetimes

The life of your data matters. You have to ask yourself, Is it useful forever? Does it become harmful after a set time? We had to confront the real impact of deleting mugs after 60 days. People share them, potentially lengthening their lifetime long after they've fallen off the homepage. Delete them and that URL goes away.

We couldn't stop people from sharing links on social media—and indeed probably didn't want to stop them from doing it. Heck, we did it while we were building it. We kept IMing URLs to each other. And that's how we realized we had a problem. All our work to minimize the impact on someone wrongly accused of a crime could be damaged by someone sharing a link on Facebook or Twitter.

There's a difference between frictionless and unobstructed sharing and some reasonable constraints.

We couldn't stop people from posting a mug on Facebook, but we didn't have to make it easy and we didn't have to put that mug front and center. So we blocked Facebook from using the mug as the thumbnail image on a shared link. And, after 60 days, the URL to the mug will throw a 404 page not found error. Because it's gone.

We couldn't block Google from memorializing someone's arrest, only to let it live on forever on Facebook.

5.4 You Are a Data Provider

The last problem didn't come until months later. And it came in the middle of the night. Two months after we launched, my phone rang at 1 a.m. This is never a good thing. It was my fellow developer, Jeremy Bowers, now with NPR, calling me from a hotel in Washington DC where he was supposed to appear in a wedding the next day. Amazon, which we were using for image hosting, was alerting him that our bandwidth bills had tripled on that day. And our traffic hadn't changed.

What was going on?

After some digging, we found out that another developer had scraped our site—because we were so much easier to scrape than the Sheriff's office sites—and had built a game out of our data called Pick the Perp. There were two problems with this: 1. The game was going viral on Digg (when it was still a thing) and Reddit. It was getting huge traffic. 2. That developer had hotlinked our images. He/she was serving them from our S3 account, which meant we were bearing the costs. And they were going up exponentially by the minute.

What we didn't realize when we launched, and what we figured out after Pick the Perp, was that we had become data provider, in a sense. We had done the hard work of getting the data out of a website and we put it into neat, semantic, easily digestible HTML. If you were after a stream of mugshots, why go through all the hassle of scraping four different sheriff's office's horrible HTML when you could just come get ours easily?

Whoever built Pick the Perp, at least at the time, chose to use our site. But, in doing so, they also chose to hotlink images—use the URL of our S3 bucket, which cost us money—instead of hosting the images themselves.

That was a problem we hadn't considered. People hotlink images all the time. And, until those images are deleted from our system, they'll stay hotlinked somewhere.

Amazon's S3 has a system where you can attach a key to a file that expires after X period of time. In other words, the URL to your image only lasts 15 minutes, or an hour, or however long you decide, before it breaks. It gives you fine grained control over how long someone can use your image URL.

So at 3 a.m., after two hours of pulling our hair out, we figured out how to sync our image keys with our cache refreshes. So every 15 minutes, a url to an image expired and Pick the Perp came crashing down.

While the Pick the Perp example is an easy one—it's never cool to hotlink an image—it does raise an issue to consider. Because you are thinking carefully about how to build your app the right way doesn't mean someone else will. And it doesn't mean they won't just go take your data from your site. So how could you deal with that? Make the data available as a download? Create an API that uses your same ethical constructs? Terms of service? All have pros and cons and are worth talking about before going forward.

5.5 Ethical Data

We live in marvelous times. The web offers you no end of tools to make things on the web, to put data from here on there, to make information freely available. But, we're an optimistic lot. Developers want to believe that their software is being used only for good. And most people will use it for good. But, there are times where the data you're working with makes people uncomfortable. Indeed, much of journalism is about making people uncomfortable, publishing things that make people angry, or expose people who don't want to be exposed.

What I want you to think about, before you write a line of code, is what does it mean to put your data on the internet? What could happen, good and bad? What should you do to be responsible about it?

Because it can have consequences.

On Dec. 23, the Journal News in New York published a map of every legal gun permit holder in their home circulation county. It was a public record. They put it into Google Fusion Tables and Google dutifully geocoded the addresses. It was a short distance to publication from there.

Within days, angry gun owners had besieged the newspaper with complaints, saying the paper had given criminals directions to people's houses where they'd find valuable guns to steal. They said the paper had violated their privacy. One outraged gun owner assembled a list of the paper's staff, including their home addresses, telephone numbers, email addresses and other details. The paper hired armed security to stand watch at the paper.

By February, the New York state legislature removed handgun permits from the public record, citing the Journal News as the reason.

There's no end of arguments to be had about this, but the simple fact is this: The reason people were angry was because you could click on a dot on the map and see a name and an address. In Fusion Tables, removing that info window would take two clicks.

Because you can put data on the web does not mean you should put data on the web. And there's a difference between a record being "public" and "in front of a large audience."

So before you write the first line of code, ask these questions:

- This data is public, but is it widely available? And does making it widely available and easy to use change anything?
- Should this data be searchable in a search engine?
- Does this data expose information someone has a reasonable expectation that it would remain at least semi-private?
- Does this data change over time?
- Does this data expire?
- What is my strategy to update or delete data?

- How easy should it be to share this data on social media?
- How should I deal with other people who want this data? API? Bulk download?

Your answers to these questions will guide how you build your app. And hopefully, it'll guide you to better decisions about how to build an app with ethics in mind.

6 Public records

Public records are the lifeblood of investigative reporting. They carry their own philosophical framework, in a manner of speaking.

- Sunlight is the best disinfectant. Corruption hides in the shadows.
- You paid for it with your taxes. It should be yours (with exceptions).
- Journalism with a capital J is about holding the powerful accountable for their actions.

Keeping those things in mind as you navigate public records is helpful.

6.1 Federal law

Your access to public records and public meetings is a matter of the law. As a journalist, it is your job to know this law better than most lawyers. Which law applies depends on which branch of government you are asking. In addition to documents and other kinds of information, FOIA also provides access to structured datasets of the kind we'll use in this class.

The Federal Government is covered by the Freedom of Information Act, or FOIA. FOIA is not a universal term. Do not use it if you are not talking to a federal agency. FOIA is a beacon of openness to the world. FOIA is deeply flawed and frustrating.

Why?

- There is no real timetable with FOIA. Requests can take months, even years.
- As a journalist, you can ask that your request be expedited.
- Guess what? That requires review. More delays.
- Exemptions are broad. National security, personal privacy, often overused.
- Denied? You can appeal. More delays.

The law was enacted in 1966, but it's still poorly understood by most federal employees, if not outright flouted by political appointees. Lawsuits are common.

Post 9/11, the Bush administration rolled back many agency rules. Obama ordered a "presumption of openness" but followed it with some of the most restrictive policies ever seen. The Trump Administration, similar to the Obama administration, claims to be the most transparent administration, but has steadily removed records from open access and broadly denied access to records.

Result? FOIA is in trouble.

SPJ is a good resource.

6.2 State law

States are – generally – more open than the federal government. The distance between the government and the governed is smaller. Some states, like Florida and Texas, are very open. Others, like Virginia and Pennsylvania, are not. Maryland is somewhere in the middle.

These laws generally give you license to view – and obtain a copy of – a record held by a state or local government agency.

What is a public record? Generally speaking, public records are information stored on paper or in an electronic format held by a state or local government agency, but each state has its own list of types of records – called "exemptions" – that are not subject to disclosure.

If a record has both exempt and non-exempt information mixed in, most states require an agency to disclose it after removing the exempt information, a process called "redaction." Agencies aren't required to create a record in order to fill your request.

In some states but not all – the public information law (or related case law) explicitly dictates that extracting a slice of a database doesn't constitute creation of a record. Most states can charge you a reasonable fee for time spent retrieving or copying records, though many have provisions to waive those fees for journalists. Every state law operates on a different timeline. Some only require agencies respond in a "reasonable" time, but others spell out exactly how fast an agency must respond to you, and how fast they must turn over the record.

The Reporters Committee For Freedom of the Press has a good resource for learning the law in your state.

Please and thank you will get you more records than any lawyer or well-written request. Be nice. Be polite. And be persistent. Following up regularly to check on status of a request lets an agency know they can't ignore you (and some will try). Hunting for records is like any other kind of reporting – you have to do research. You have to ask questions. Ask them: What records do you keep? For how long?

When requesting data, you are going to scare the press office and you are going to confuse the agency lawyer. Request to have their data person on the phone.

A good source of info? Records retention schedules, often required by law or administrative rule at an agency. Here's an example from Maryland's Circuit Courts.

7 Setting Up Your Computer To Use GitHub

You will use your computer a LOT in this class. There is no way to avoid that. So you will need to get comfortable with using your computer in predictable and replicable ways. This is not the place for coming up with different ways to use your computer. Repetition is a good thing for this class.

It is technically possible to do most of the work for this class with a ChromeBook, but I don't recommend it. If you have an older Mac (pre-2017), you may need to install an older version of R, but you should come and see me. It will help if you have the latest version of your operating system installed.

7.1 Space, the Final Frontier

You will need to have a substantial amount of hard drive space to store the files for this class. If you are not in the practice of deleting files because "everything's in the cloud" anyway, you will need to clear out some space on your laptop. How much space? The more, the better, but let's go with at least 10GB.

7.2 GitHub

GitHub is a platform for managing and storing files, data and code built atop Git, a popular open source version control software. GitHub accounts are free and it's easy to get started. The one prerequisite is that you have Git installed on your local computer. There are installers for Mac, Windows and Linux. Install Git first.

7.2.1 How GitHub Works

Version control is based on the ideas that you want to keep track of changes you make to a collection of files and that multiple people can work together without getting in each other's way or having to do things in a set order. For individual users, it's great for making sure that you always have your work.

GitHub users work in what are known as repositories on their local computers and also *push* changes to a remote repository located on GitHub. That remote repository is key: if you lose your computer, you can fetch a version of your files from GitHub. If you want to work with someone else on the same files, you can each have a local copy, push changes to GitHub and then pull each others' changes back to your local computers.

So, like Microsoft Word's track changes, but with a remote backup and multiple editors.

7.2.2 Getting Started with GitHub

After installing Git and signing up for a GitHub account, download and install GitHub Desktop. It will have you sign into your GitHub account and then you'll have access to any existing repositories. If you don't have any, that's fine! You can make one locally.

GitHub has good documentation for working in the Desktop app, and while the emphasis in this book will be on using GitHub for version control, it also supports recording issues (read: problems or questions) with your files, contributing to projects that aren't yours and more.

7.2.3 A Place for Your Stuff

We all will be working in the same way, and every student will have a repository for class assignments. Go here and click the "Use this template" button and choose "Create a new repository". This will create a copy of the repository in your GitHub account. Give it a name that makes sense, like `data_journalism_2024_fall`. Your work will occur in that repository and you will submit links to individual documents. Pro tip: do not put assignment files outside your repository.

You will clone your repository from GitHub.com to your laptop. You can sync it to multiple machines as long as you have GitHub Desktop (or the command line version) installed. The iMacs in the classroom have both, so you can use them during class if needed.

You can store your repository anywhere on your computer, but my STRONG RECOMMENDATION is that you NOT store it in your Downloads or Desktop folders. Instead, make a new folder at the root (or home) directory and put your files there. Why? Because many folks have their Download and Desktop folders as part of their iCloud or OneNote setups, and Git DOES NOT LIKE THAT. Unless you want to spend time troubleshooting seemingly inexplicable problems, avoid those folders.

Also, maybe clean up those Download and Desktop folders.

7.2.4 Local vs. GitHub

GitHub works on a system of pulling and pushing changes to files between what you have on your computer and what is on your GitHub.com account. Here's how you should work:

When you are working *locally*, before you start you should pull from GitHub.com to make sure you have the latest version of your files. When you finish, you should save and add, commit and push any changes/additions to GitHub.com. You can add/commit/push as often as you like, and there's reason not to do it often.

The nature of Git and GitHub means that you can have a copy of your files on the Web and on your computer. You can work offline and then update your file when you connect to the Internet again. When submitting your assignments, you need to submit the full URL to your assignment's file from GitHub.com.

7.2.5 GitHub Tips

In your class repository you will see various files that begin with a period, such as .gitignore. These are essentially helper files that contain metadata or instructions that tell Git and RStudio what to do and not do. For example, .gitignore contains a list of files and file types that should NOT be under version control. Maybe you have a big data file (GitHub doesn't really deal well with files of more than 50MB) or you have files containing credentials that you don't want on the Internet. Those files can still be in your local folder, but they won't be added to your GitHub repository.

7.2.6 Advanced GitHub Use

Although our focus is on the GitHub Desktop app, you can use Git and GitHub from your computer's command line interface, and GitHub has a purpose-built command line client, too. GitHub can also serve as a publishing platform for many types of files, and entire websites are hosted on GitHub Pages.

8 R Basics

R is a programming language, one specifically geared toward data analysis.

Like all programming languages, it has certain built-in functions.

There are many ways you can write and execute R code. The first, and most basic, is the console, shown here as part of a software tool called RStudio (Desktop Open Source Edition) that we'll be using all semester.

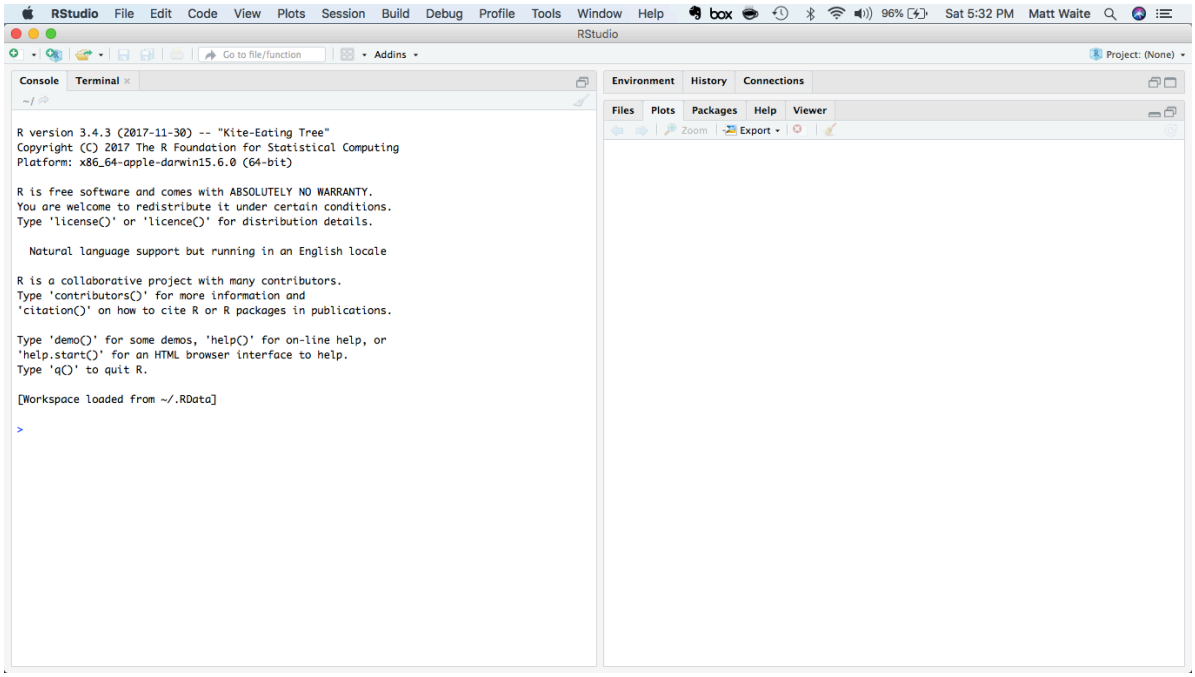


Figure 8.1: ready

Think of the console like talking directly to the R language engine that's busy working inside your computer. You use it send R commands, making sure to use the only language it understands, which is R. The R language engine processes those commands, and sends information back to you.

Using the console is direct, but it has some drawbacks and some quirks we'll get into later. Let's examine a basic example of how the console works.

If you load up R Studio, type 2+2 into the console and hit enter it will spit out the number 4, as displayed below.

```
2+2
```

```
[1] 4
```

It's not very complex, and you knew the answer before hand, but you get the idea. With R, we can compute things.

We can also store things for later use under a specific name. In programming languages, these are called **variables**. We can assign things to variables using this left-facing arrow: `<-`. The `<-` is a called an **assignment operator**.

If you load up R studio and type this code in the console…

```
number <- 2
```

…and then type this code, it will spit out the number 4, as show below.

```
number * number
```

```
[1] 4
```

We can have as many variables as we can name. We can even reuse them (but be careful you know you're doing that or you'll introduce errors).

If you load up R studio and type this code in the console…

```
firstnumber <- 1
secondnumber <- 2
```

…and then type this, it will split out the number 6, as shown below.

```
(firstnumber + secondnumber) * secondnumber
```

```
[1] 6
```

We can store anything in a variable. A whole table. A list of numbers. A single word. A whole book. All the books of the 18th century. Variables are really powerful. We'll explore them at length.

A quick note about the console: After this brief introduction, we won't spend much time in R Studio actually writing code directly into the console. Instead, we'll write code in fancied-up text files – interchangably called R Markdown or R Notebooks – as will be explained in the next chapter. But that code we write in those text files will still *execute* in the console, so it's good to know how it works.

8.1 About libraries

The real strength of any programming language is the external libraries (often called "packages") that power it. The base language can do a lot, but it's the external libraries that solve many specific problems – even making the base language easier to use.

With R, there are hundreds of free, useful libraries that make it easier to do data journalism, created by a community of thousands of R users in multiple fields who contribute to open-source coding projects.

For this class, we'll make use of several external libraries.

Most of them are part of a collection of libraries bundled into one "metapackage" called the Tidyverse that streamlines tasks like:

- Loading data into R. (We'll use the readr Tidyverse library)
- Cleaning and reshaping the data before analysis. (We'll use the the tidyr and dplyr Tidyverse libraries)
- Data analysis. (We'll use the dplyr Tidyverse library)
- Data visualization (We'll use the ggplot2 Tidyverse library)

To install packages, we use the function `install.packages()`.

You only need to install a library once, the first time you set up a new computer to do data journalism work. You never need to install it again, unless you want to update to a newer version of the package.

To install all of the Tidyverse libraries at once, the function is `install.packages('tidyverse')`. You can type it directly in the console.

To use the R Markdown files mentioned earlier, we also need to install a Tidyverse-related library that doesn't load as part of the core Tidyverse package. The package is called, conveniently, rmarkdown. The code to install that is `install.packages('rmarkdown')`

9 Data journalism in the age of replication

A single word in a single job ad for Buzzfeed News posted in 2017 offered an indication of a profound shift in how data journalism is both practiced and taught.

"We're looking for someone with a passion for news and a commitment to using data to find amazing, important stories — both quick hits and deeper analyses that drive conversations," the posting seeking a data journalist says. It goes on to list five things BuzzFeed is looking for: Excellent collaborator, clear writer, deep statistical understanding, knowledge of obtaining and restructuring data.

And then there's this:

"You should have a strong command of at least one toolset that (a) allows for filtering, joining, pivoting, and aggregating tabular data, and (b) enables reproducible workflows."

The word you're seeing more and more of? Reproducible. And it started in earnest in 2017 when data journalism crossed a major threshold in American journalism: It got it's own section in the Associated Press Stylebook.

"Data journalism has become a staple of reporting across beats and platforms," the Data Journalism section of the Stylebook opens. "The ability to analyze quantitative information and present conclusions in an engaging and accurate way is no longer the domain of specialists alone."

The AP's Data Journalism section discusses how to request data and in what format, guidelines for scraping data from websites with automation, the ethics of using leaked or hacked data and other topics long part of data journalism conference talks.

But the third page of the section contains perhaps the most profound commandment: **"As a general rule, all assertions in a story based on data analysis should be reproducible. The methodology description in the story or accompanying materials should provide a road map to replicate the analysis."**

Reproducible research – replication – is a cornerstone of scientific inquiry. Researchers across a range of academic disciplines use methods to find new knowledge and publish it in peer reviewed journals. And, when it works, other researchers take that knowledge and try it with their own samples in their own locations. Replication studies exist to take something from an "interesting finding" to a "theory" and beyond.

It doesn't always work.

Replication studies aren't funded at nearly the level as new research. And, to the alarm of many, scores of studies can't be replicated by others. Researchers across disciplines are finding that when their original studies are replicated, flaws are found, or the effects found aren't as strong as the original. Because of this, academics across a number of disciplines have written about a replication crisis in their respective fields, particularly psychology, social science and medical research.

In Chapter 1 of the New Precision Journalism, Phil Meyer wrote that "we journalists would be wrong less often if we adapted to our own use some of the research tools of the social scientists."

Meyer would go on to write about how computers pouring over datasets too large to crunch by hand had changed social science from a discipline with "a few data and a lot of interpretation" into a much more meaningful and powerful area of study. If journalists could become comfortable with data and some basic statistics, they too could harness this power.

"It used to be said that journalism is history in a hurry," Meyer wrote. "The argument of this book is that to cope with the acceleration of social change in today's world, journalism must become social science in a hurry."

He wrote that in 1971. It might as well have been yesterday.

Journalism doesn't have a history of replication, but the concerns about credibility are substantially greater. Trust in media is at an all time low and shows no signs of improving. While the politics of the day have quite a bit to do with this mistrust of media, being more transparent about what journalists do can't hurt.

The AP's commandment that "Thou must replicate your findings" could, if taken seriously by the news business, have substantial impacts on how data journalism gets done in newsrooms and how data journalism gets taught, both at professional conferences and universities.

How? Two ways.

- The predominant way that data journalism gets done in a newsroom is through simple tools like Microsoft Excel or Google Sheets. Those simple tools, on their own, lack significant logging functions that automatically keep track of steps a data journalist took to reach a given conclusion. That means journalists using those tools have to maintain separate, detailed logs of what they did so any analysis can be replicated.
- The predominant way that data journalism gets taught – both in professional settings and at most universities – doesn't deal with replication at all. The tools and the training stress "getting things done" – an entirely logical focus for a deadline driven business. The choices of tools – like spreadsheet programs – are made to get from data to story as quick as possible, without frightening away math and tech phobic students.

If the AP's replication rules are to be followed, journalism needs to become much more serious about the tools and techniques used to do data journalism. The days of "point and click" tools to do "quick and dirty" analysis that get published are dying. The days of formal methods using documented steps are here.

9.1 The stylebook

Troy Thibodeaux, previously the editor of the AP's data journalism team, said the stylebook entry started when the data team found themselves answering the same questions over and over. With a grant from the Knight Foundation, the team began to document their own standards and turn that into a stylebook section.

From the beginning, they had a fairly clear idea of what they wanted to do – think through a project and ask what the frequently asked questions are that came up. It was not going to be a soup-to-nuts guide to how to do a data project.

When the section came out, eyebrows went up on the replication parts, surprising Thibodeaux.

"From our perspective, this is a core value for us," he said. "Just for our own benefit, we need to be able to have someone give us a second set of eyes. We benefit from that every day. We catch things for each other."

Thibodeaux said the AP data team has two audiences when it comes to replication – they have the readers of the work, and members of the collective who may want to do their own work with the data.

"This is something that's essential to the way we work," he said. "And it's important in terms of transparency and credibility going forward. We thought it would be kind of unexceptionable."

9.2 Replication

Meyer said he was delighted to see replication up for discussion now, but warned that we shouldn't take it too far.

"Making the analysis replicable was something I worried about from the very beginning," he wrote in an email. So much so that in 1967, after publishing stories from his landmark survey after the Detroit riots, he shipped the data and backup materials about it to a social science data repository at the University of North Carolina.

And, in doing so, he opened the door to others replicating his results. One scholar attempted to find fault with Meyer's analysis by slicing the data ever thinner until the differences weren't significant – gaming the analysis to criticize the stories.

Meyer believes replication is vitally important, but doesn't believe it should take on the trappings of science replication, where newsrooms take their own samples or re-survey a community. That would be prohibitively expensive.

But journalists should be sharing their data and analysis steps. And it doesn't need to be complicated, he said.

"Replication is a theoretical standard, not a requirement that every investigator duplicate his or her own work for every project," he said. "Giving enough information in the report to enable another investigator to follow in your footsteps is enough. Just telling enough to make replication possible will build confidence."

But as simple as that sounds, it's not so simple. Ask social scientists.

Andrew Gelman, a professor of statistics and political science and director of the Applied Statistics Center at Columbia University, wrote in the journal CHANCE that difficulties with replication in empirical research are pervasive.

"When an outsider requests data from a published paper, the authors will typically not post or send their data files and code, but instead will point to their sources, so replicators have to figure out exactly what to do from there," Gelman wrote. "End-to-end replicability is not the norm, even among scholars who actively advocate for the principles of open science."

So goes science, so goes journalism.

Until a recent set of exceptions, journalists rarely shared data. The "nerd box" – a sidebar story that explains how a news organization did what they did – is a term that first appeared on NICAR-L, a email listserv of data journalists, in the 1990s.

It was a form born in print.

As newsrooms adapted to the internet, some news organizations began linking to their data sources if they were online. Often, the data used in stories were obtained through records requests. Sometimes, reporters created the data themselves.

Journalism, more explicitly than science, is a competitive business. There have been arguments that nerd boxes and downloadable links give too much away to competitors.

Enter the AP Stylebook.

The AP Stylebook argues explicitly for both internal and external replication. Externally, they argue that the **"methodology description in the story or accompanying materials should provide a road map to replicate the analysis"**, meaning someone else could do the replication post publication.

Internally, the AP Stylebook says: **"If at all possible, an editor or another reporter should attempt to reproduce the results of the analysis and confirm all findings before publication."**

There are two problems here.

First is that journalism, unlike science, has no history of replication. There is no "scientific method" for stories. There is no standard "research methods" class taught at every journalism school, at least not where it comes to writing stories. And, beyond that, journalism school isn't a requirement to get into the news business. In other words, journalism lacks the standards other disciplines have.

The second problem is, in many ways, worse: Except for the largest newsrooms, most news organizations lack editors who could replicate the analysis. Many don't have a second person who would know what to do.

Not having a second set of eyes in a newsroom is a problem, Thibodeaux acknowledges. Having a data journalism team "is an incredible luxury" at the AP, he said, and their rule is nothing goes on the wire without a second set of eyes.

Thibodeaux, for his part, wants to see fewer "lone nerds in the corner" – it's too much pressure. That person gets too much credibility from people who don't understand what they do, and they get too much blame when a mistake is made.

So what would replication look like in a newsroom? What does this mean for how newsrooms do data journalism on deadline?

9.3 Goodbye Excel?

For decades, Excel has been the gateway drug for data journalists, the Swiss Army knife of data tools, the "One Tool You Can't Live Without." Investigative Reporters and Editors, an organization that trains investigative journalists, have built large amounts of their curricula around Excel. Of the journalism schools that teach data journalism, most of them begin and end with spreadsheets.

The Stylebook says at a minimum, today's data journalists should keep a log that details:

- The source of the data, making sure to work on a copy of the data and not the original file.
- Data dictionaries or any other supporting documentation of the data.
- **"Description of all steps required to transform the data and perform the analysis."**

The trouble with Excel (or Google Sheets) is, unless you are keeping meticulous notes on what steps you are taking, there's no way to keep track. Many data journalists will copy and paste the values of a formula over the formula itself to prevent Excel from fouling up cell references when moving data around – a practical step that also cuts off another path to being able to replicate the results.

An increasing number of data journalists are switching to tools like analysis notebooks, which use languages like Python and R, to document their work. The notebooks, generally speaking, allow a data journalist to mix code and explanation in the same document.

Combined with online sharing tools like GitHub, analysis notebooks seem to solve the problem of replication. But the number using them is small compared to those using spreadsheets. Recent examples of news organizations using analysis notebooks include the Los Angeles Times, the New York Times, FiveThirtyEight, and Buzzfeed.

Peter Aldous, a data journalist at Buzzfeed, published a story about how the online news site used machine learning to find airplanes being used to spy on people in American cities. Published with the story is the code Aldous used to build his case.

"I think of it this way: As a journalist, I don't like to simply trust what people tell me. Sometimes sources lie. Sometimes they're just mistaken. So I like to verify what I'm told," he wrote in an email. "By the same token, why should someone reading one of my articles believe my conclusions, if I don't provide the evidence that explains how I reached them?"

The methodology document, associated code and source data took Aldous a few hours to create. The story, from the initial data work through the reporting required to make sense of it all, took a year. Aldous said there wasn't a discussion about if the methodology would be published because it was assumed – "it's written into our DNA at BuzzFeed News."

"My background is in science journalism, and before that (way back in the 1980s) in science," Aldous said. "In science, there's been a shift from descriptive methods sections to publishing data and analysis code for reproducible research. And I think we're seeing a similar shift in data journalism. Simply saying what you've done is not as powerful as providing the means for others to repeat and build on your work."

Thibodeaux said that what Buzzfeed and others do with analysis notebooks and code repositories that include their data is "lovely."

"That to me is the shining city on the hill," Thibodeaux said. "We're not going to get there, and I don't think we have to for every story and every use case, and I don't think it's necessarily practical for every person working with data to get to that point."

There's a wide spectrum of approaches that still gets journalists to the essence of what the stylebook is trying to do, Thibodeaux said. There are many tools, many strategies, and the AP isn't going to advocate for any single one of them, he said. They're just arguing for transparency and replicability, even if that means doing more work.

"There's a certain burden that comes with transparency," he said. "And I think we have to accept that burden."

The question, Thibodeaux said, is what is sufficient? What's enough transparency? What does someone need for replicability?

"Maybe we do have to set a higher standard – the more critical the analysis is to the story, and the more complex that analysis is, that's going to push the bar on what is a sufficient methodology statement," he said. "And it could end up being a whole code repo in order to just say, this isn't black magic, here's how we got it if you're so interested."

9.4 "Receptivity … is high"

Though written almost half a century ago, Meyer foresaw how data journalism was going to arrive in the newsroom.

"For the new methods to gain currency in journalism, two things must happen," he wrote. "Editors must feel the need strongly enough to develop the in-house capacity for systematic research … The second need, of course, is for the editors to be able to find the talent to fill this need."

Meyer optimistically wrote that journalism schools were prepared to provide that talent – they were not then, and only small handful are now – but students were unlikely to be drawn to these new skills if they didn't see a chance to use those skills in their careers.

It's taken 50 years, but we are now at this point.

"The potential for receptivity, especially among the younger generation of newspaper managers, is high," Meyer wrote.

9.5 Replication in notebooks

For our purposes in this book, replication requires two things from you, the student: What and why. What is this piece of code doing, and why are you doing that here and now? What lead you to this place? That you can copy and paste code from this book or the internet is not impressive. What is necessary for learning is that you know what a piece of code is doing a thing and why you want to do that thing here.

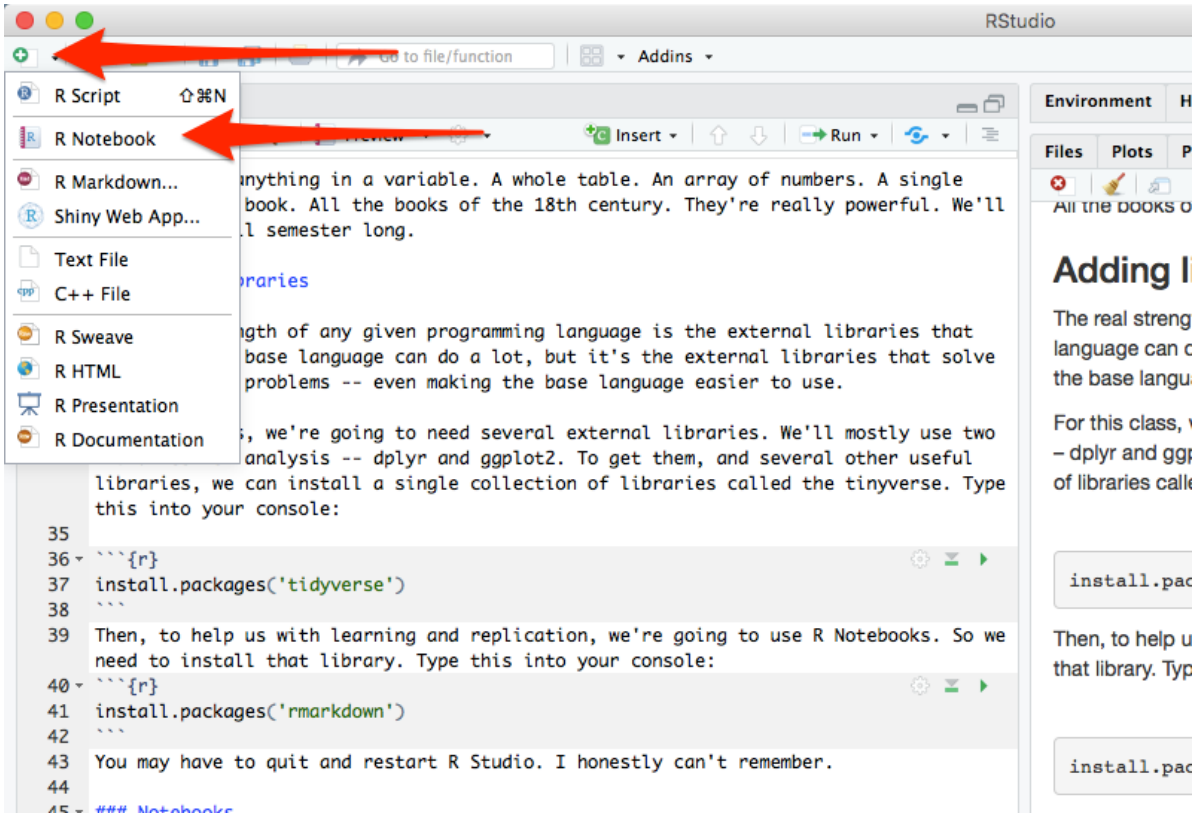
How will we replicate? We'll make use of special text files – R Markdown, also known as R Notebooks – that combine contextual text; the code we use to load, clean, analyze and visualize data; and the output of that code that allowed us to draw certain conclusions to use in stories.

In an R Notebook, there are two blocks: A block that uses markdown, which has no special notation, and a code block. The code blocks can run mulitple languages inside R Studio. There's R, of course, but we could also run Python, a general purpose scripting language; and SQL, or Structured Query Language, the language of databases.

For the rest of the class, we're going to be working in notebooks.

In notebooks, you will both run your code and explain each step, much as I am doing here in this online book. This entire book was produced with R markdown files.

To start a notebook in R Studio, you click on the green plus in the top left corner and go down to R Notebook.



In our first lab, we'll go through the process of editing a markdown notebook.

10 Aggregates

10.1 Libraries

R is a statistical programming language that is purpose-built for data analysis.

Base R does a lot, but there are a mountain of external libraries that do things to make R better/easier/more fully featured. We already installed the tidyverse – or you should have if you followed the instructions for the last assignment – which isn't exactly a library, but a collection of libraries. Together, they make up the Tidyverse. Individually, they are extraordinarily useful for what they do. We can load them all at once using the tidyverse name, or we can load them individually. Let's start with individually.

The two libraries we are going to need for this assignment are `readr` and `dplyr`. The library `readr` reads different types of data in. For this assignment, we're going to read in csv data or Comma Separated Values data. That's data that has a comma between each column of data.

Then we're going to use `dplyr` to analyze it.

To use a library, you need to import it. Good practice – one I'm going to insist on – is that you put all your library steps at the top of your notebooks.

That code looks like this:

```
library(readr)
```

To load them both, you need to do this:

```
library(readr)
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

But, because those two libraries – and several others that we're going to use over the course of this class – are so commonly used, there's a shortcut to loading all of the libraries we'll need:

```
library(tidyverse)
```

You can keep doing that for as many libraries as you need.

10.2 Importing data

The first thing we need to do is get some data to work with. We do that by reading it in. In our case, we're going to read a datatable from an "rds" file, which is a format for storing data with R. Later in the course, we'll more frequently work with a format called a CSV. A CSV is a stripped down version of a spreadsheet you might open in a program like Excel, in which each column is separated by a comma. RDS files are less common when getting data from other people. But reading in CSVs is less foolproof than reading in rds files, so for now we'll work with rds.

The rds file we're going to read in contains individual campaign contributions from Maryland donors via WinRed, an online fundraising platform used by conservatives. We'll be working with a slice of the data from earlier this year.

So step 1 is to import the data. The code to import the data looks like this:

```
maryland_winred_contributions <- read_rds("maryland_winred.rds")
```

Let's unpack that.

The first part – **maryland_winred_contributions** – is the name of a variable.

A **variable** is just a name that we'll use to refer to some more complex thing. In this case, the more complex thing is the data we're importing into R that will be stored as a **dataframe**, which is one way R stores data.

We can call this variable whatever we want. The variable name doesn't matter, technically. We could use any word. You could use your first name, if you like. Generally, though, we want to give variables names that are descriptive of the thing they refer to. Which is why we're calling this one **maryland_winred_contributions**. Variable names, by convention are one word all lower case (or two or more words connected by an underscore). You can end a variable with a number, but you can't start one with a number.

The `<-` bit, you'll recall from the basics, is the **variable assignment operator**. It's how we know we're assigning something to a word. Think of the arrow as saying "Take everything on the right of this arrow and stuff it into the thing on the left." So we're creating an empty vessel called **maryland_winred_contributions** and stuffing all this data into it.

read_rds() is a function, one that only works when we've loaded the tidyverse. A **function** is a little bit of computer code that takes in information and follows a series of pre-determined steps and spits it back out. A recipe to make pizza is a kind of function. We might call it **make_pizza()**.

The function does one thing. It takes a preset collection of ingredients – flour, water, oil, cheese, tomato, salt – and passes them through each step outlined in a recipe, in order. Things like: mix flour and water and oil, knead, let it sit, roll it out, put tomato sauce and cheese on it, bake it in an oven, then take it out.

The output of our **make pizza()** function is a finished pie.

We'll make use of a lot of pre-written functions from the tidyverse and other packages, and even write some of our own. Back to this line of code:

```
maryland_winred_contributions <- read_rds("maryland_winred.rds")
```

Inside of the **read_rds()** function, we've put the name of the file we want to load. Things we put inside of function, to customize what the function does, are called **arguments**.

The easiest thing to do, if you are confused about how to find your data, is to put your data in the same folder as as your notebook (you'll have to save that notebook first). If you do that, then you just need to put the name of the file in there (maryland_winred.rds). If you put your data in a folder called "data" that sits next to your data notebook, your function would instead look like this:

```
maryland_winred_contributions <- read_rds("data/maryland_winred.rds")
```

In this data set, each row represents an individual contribution to a federal political committee, typically a candidate's campaign account.

After loading the data, it's a good idea to get a sense of its shape. What does it look like? There are several ways we can examine it.

By looking in the R Studio environment window, we can see the number of rows (called "obs.", which is short for observations), and the number of columns(called variables). We can double click on the dataframe name in the environment window, and explore it like a spreadsheet.

There are several useful functions for getting a sense of the dataset right in our markdown document.

If we run `glimpse(maryland_winred_contributions)`, it will give us a list of the columns, the data type for each column and and the first few values for each column.

```
glimpse(maryland_winred_contributions)
```

```
Rows: 131,395
Columns: 24
$ linenumber       <chr> "SA11AI", "SA11AI", "SA11AI", "SA11AI", "SA11AI", "SA~
$ fec_committee_id <chr> "C00694323", "C00694323", "C00694323", "C00694323", "~
$ tran_id          <chr> "A000BA09B6F8D45FCBA7", "A0011063AFC5B47B2AE6", "A001~
$ flag_orgind      <chr> "IND", "IND", "IND", "IND", "IND", "IND", "IND", "IND~
$ org_name         <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ last_name        <chr> "Curro", "Mukai", "Smith", "SaylorJones", "Gillissen"~
$ first_name       <chr> "Peter", "Peggy", "Alan", "Jean", "Troy", "Bryan", "K~
$ middle_name      <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ prefix           <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ suffix           <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ address_one      <chr> "1902 Blakewood Ct", "729 Fox Bow Dr", "308 Troon Cir~
$ address_two      <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ city             <chr> "Fallston", "Bel Air", "Mount Airy", "Gaithersburg", ~
$ state            <chr> "MD", "MD", "MD", "MD", "MD", "MD", "MD", "MD", "MD",~
$ zip              <chr> "21047", "21014", "21771", "20877", "20695", "21228",~
$ prigen           <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ date             <date> 2024-04-04, 2024-04-04, 2024-04-02, 2024-04-05, 2024~
$ amount           <dbl> 5.21, 17.76, 26.03, 1.67, 5.21, 75.00, 0.99, 5.21, 2.~
$ aggregate_amount <dbl> 10.21, 1221.77, 3884.66, 23.38, 11.21, 218.00, 29.33,~
$ employer         <chr> "Woodfield outdoors", "RETIRED", "RETIRED", "RETIRED"~
$ occupation       <chr> "BUSINESS DEVELOPMENT", "RETIRED", "RETIRED", "RETIRE~
$ memo_code        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ memo_text        <chr> "Earmarked for TRUMP NATIONAL COMMITTEE JFC (C0087389~
$ cycle            <dbl> 2024, 2024, 2024, 2024, 2024, 2024, 2024, 2024, 2024,~
```

If we type `head(maryland_winred_contributions)`, it will print out the columns and the first six rows of data.

```
head(maryland_winred_contributions)
```

```
# A tibble: 6 x 24
  linenu~1 fec_c~2 tran_id flag_~3 org_n~4 last_~5 first~6 middl~7 prefix suffix
  <chr>    <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>
1 SA11AI   C00694~ A000BA~ IND     <NA>    Curro   Peter   NA      NA     NA
2 SA11AI   C00694~ A00110~ IND     <NA>    Mukai   Peggy   NA      NA     NA
3 SA11AI   C00694~ A0011E~ IND     <NA>    Smith   Alan    NA      NA     NA
4 SA11AI   C00694~ A001C3~ IND     <NA>    Saylor~ Jean    NA      NA     NA
```

```
5 SA11AI   C00694~ A00219~ IND     <NA>    Gillis~ Troy    NA      NA     NA
6 SA11AI   C00694~ A002A2~ IND     <NA>    Lally   Bryan   NA      NA     NA
# ... with 14 more variables: address_one <chr>, address_two <chr>, city <chr>,
#   state <chr>, zip <chr>, prigen <lgl>, date <date>, amount <dbl>,
#   aggregate_amount <dbl>, employer <chr>, occupation <chr>, memo_code <lgl>,
#   memo_text <chr>, cycle <dbl>, and abbreviated variable names 1: linenumber,
#   2: fec_committee_id, 3: flag_orgind, 4: org_name, 5: last_name,
#   6: first_name, 7: middle_name
```

We can also click on the data name in the R Studio environment window to explore it interactively.

10.3 Group by and count

So what if we wanted to know how many contributions went to each recipient?

To do that by hand, we'd have to take each of the 133,363 individual rows (or observations or records) and sort them into a pile. We'd put them in groups – one for each recipient – and then count them.

`dplyr` has a group by function in it that does just this. A massive amount of data analysis involves grouping like things together and then doing simple things like counting them, or averaging them together. So it's a good place to start.

So to do this, we'll take our dataset and we'll introduce a new operator: `|>`. The best way to read that operator, in my opinion, is to interpret that as "and then do this."

We're going to establish a pattern that will come up again and again throughout this book: `data |> function`. In English: take your data set and then do this specific action to it.

The first step of every analysis starts with the data being used. Then we apply functions to the data.

In our case, the pattern that you'll use many, many times is: `data |> group_by(COLUMN NAME) |> summarize(VARIABLE NAME = AGGREGATE FUNCTION(COLUMN NAME))`

In our dataset, the column with recipient information is called "memo_text"

Here's the code to count the number of contributions to each recipient:

```
maryland_winred_contributions |>
  group_by(memo_text) |>
  summarise(
    count_contribs = n()
  )
```

```
# A tibble: 481 x 2
   memo_text                                              count_contribs
   <chr>                                                           <int>
 1 Earmarked for AARON BEAN FOR CONGRESS (C00816983)                   1
 2 Earmarked for AARON DIMMOCK FOR CONGRESS (C00877225)                6
 3 Earmarked for ABE FOR ARIZONA (C00853986)                         178
 4 Earmarked for ADAM MORGAN FOR CONGRESS (C00857060)                 58
 5 Earmarked for ADRIAN SMITH FOR CONGRESS (C00412890)                 1
 6 Earmarked for ALABAMA FIRST PAC (C00821058)                         3
 7 Earmarked for ALAMO PAC (C00387464)                                 2
 8 Earmarked for ALASKANS FOR DAN SULLIVAN (C00570994)                 3
 9 Earmarked for ALASKANS FOR NICK BEGICH (C00792341)                  8
10 Earmarked for ALEX FOR NORTH DAKOTA (C00873927)                     2
# ... with 471 more rows
```

So let's walk through that.

We start with our dataset – `maryland_winred_contributions` – and then we tell it to group the data by a given field in the data. In this case, we wanted to group together all the recipients, signified by the field name `memo_text`, which you could get from using the glimpse() function. After we group the data, we need to count them up.

In dplyr, we use the `summarize()` function, which can do alot more than just count things.

Inside the parentheses in summarize, we set up the summaries we want. In this case, we just want a count of the number of loans for each county grouping. The line of code `count_contribs = n(),` says create a new field, called `count_contribs` and set it equal to `n()`. `n()` is a function that counts the number of rows or records in each group. Why the letter n? The letter n is a common symbol used to denote a count of something. The number of things (or rows or observations or records) in a dataset? Statisticians call it n. There are n number of contributions in this dataset.

When we run that, we get a list of recipients with a count next to them. But it's not in any order.

So we'll add another "and then do this" symbol – `|>` – and use a new function called `arrange()`. Arrange does what you think it does – it arranges data in order. By default, it's in ascending order – smallest to largest. But if we want to know the county with the most loans, we need to sort it in descending order. That looks like this:

```
maryland_winred_contributions |>
  group_by(memo_text) |>
  summarise(
    count_contribs = n()
```

```
  ) |>
  arrange(desc(count_contribs))
```

```
# A tibble: 481 x 2
   memo_text                                                        count~1
   <chr>                                                              <int>
 1 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C00873893)       41835
 2 Earmarked for TRUMP SAVE AMERICA JOINT FUNDRAISING COMMITTEE (C00770~   10812
 3 Earmarked for NRSC (C00027466)                                      9063
 4 Earmarked for HOGAN FOR MARYLAND INC. (C00869016)                   5754
 5 Earmarked for NRCC (C00075820)                                      4416
 6 Earmarked for TEAM SCALISE (C00750521)                              3184
 7 Earmarked for REPUBLICAN NATIONAL COMMITTEE (C00003418)             3063
 8 Earmarked for TRUMP NATIONAL COMMITTEE JFC (C00873893)              3042
 9 Earmarked for MIKE JOHNSON FOR LOUISIANA (C00608695)                2811
10 Earmarked for TEAM ELISE (C00830679)                                2389
# ... with 471 more rows, and abbreviated variable name 1: count_contribs
```

The Trump National Committee JFC Inc. has 41,835 contributions, more than any other recipient.

We can, if we want, group by more than one thing.

The WinRed data contains a column detailing the date of the contribution: "date".

We can group by "memo_text" and "date" to see how many contributions occurred on every date to every recipient. We'll sort by the count of contributions in descending order.

```
maryland_winred_contributions |>
  group_by(memo_text, date) |>
  summarise(
    count_contribs = n()
  ) |>
  arrange(desc(count_contribs))
```

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

```
# A tibble: 12,935 x 3
# Groups:   memo_text [481]
   memo_text                                                  date       count~1
   <chr>                                                      <date>       <int>
```

```
 1 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-05-31    5629
 2 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-05-30    4845
 3 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-06-01    1681
 4 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-06-27    1228
 5 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-06-02    1178
 6 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-06-03     939
 7 Earmarked for TRUMP SAVE AMERICA JOINT FUNDRAISING COMMIT~ 2024-05-31     863
 8 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-06-30     832
 9 Earmarked for TRUMP SAVE AMERICA JOINT FUNDRAISING COMMIT~ 2024-05-30     830
10 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C008738~ 2024-04-15     710
# ... with 12,925 more rows, and abbreviated variable name 1: count_contribs
```

Ok, now go and find out why contributions to Trump would be exponentially higher at the end of May.

10.4 Other summarization methods: summing, mean, median, min and max

In the last example, we grouped like records together and counted them, but there's so much more we can to summarize each group.

Let's say we wanted to know the total dollar amount of contributions to each recipient? For that, we could use the `sum()` function to add up all of the loan values in the column "amount". We put the column we want to total – "amount" – inside the sum() function `sum(amount)`. Note that we can simply add a new summarize function here, keeping our count_contribs field in our output table.

```
maryland_winred_contributions |>
  group_by(memo_text) |>
  summarise(
    count_contribs = n(),
    total_amount = sum(amount)
  ) |>
  arrange(desc(total_amount))
```

```
# A tibble: 481 x 3
   memo_text                                                  count~1 total~2
   <chr>                                                        <int>   <dbl>
 1 Earmarked for TRUMP NATIONAL COMMITTEE JFC, INC. (C00873893)   41835  1.74e6
 2 Earmarked for HOGAN FOR MARYLAND INC. (C00869016)               5754  4.79e5
```

```
 3 Earmarked for NRSC (C00027466)                                    9063  2.46e5
 4 Earmarked for TRUMP SAVE AMERICA JOINT FUNDRAISING COMMITTEE~   10812  1.97e5
 5 Earmarked for NRCC (C00075820)                                    4416  1.17e5
 6 Earmarked for PARROTT FOR CONGRESS (C00691931)                     504  1.15e5
 7 Earmarked for REPUBLICAN NATIONAL COMMITTEE (C00003418)           3063  8.90e4
 8 Earmarked for TRUMP NATIONAL COMMITTEE JFC (C00873893)            3042  7.11e4
 9 Earmarked for TEAM SCALISE (C00750521)                            3184  6.15e4
10 Earmarked for TED CRUZ FOR SENATE (C00492785)                     2187  3.33e4
# ... with 471 more rows, and abbreviated variable names 1: count_contribs,
#   2: total_amount
```

We can also calculate the average amount for each recipient – the mean – and the amount that sits at the midpoint of our data – the median.

```
maryland_winred_contributions |>
  group_by(memo_text) |>
  summarise(
    count_contribs = n(),
    total_amount = sum(amount),
    mean_amount = mean(amount),
    median_amount = median(amount)
  ) |>
  arrange(desc(count_contribs))
```

```
# A tibble: 481 x 5
   memo_text                                      count~1 total~2 mean_~3 media~4
   <chr>                                            <int>   <dbl>   <dbl>   <dbl>
 1 Earmarked for TRUMP NATIONAL COMMITTEE JFC, ~   41835  1.74e6   41.5    20.2
 2 Earmarked for TRUMP SAVE AMERICA JOINT FUNDR~   10812  1.97e5   18.3     5
 3 Earmarked for NRSC (C00027466)                   9063  2.46e5   27.1    10
 4 Earmarked for HOGAN FOR MARYLAND INC. (C0086~    5754  4.79e5   83.3    26.0
 5 Earmarked for NRCC (C00075820)                   4416  1.17e5   26.5    15
 6 Earmarked for TEAM SCALISE (C00750521)           3184  6.15e4   19.3    10
 7 Earmarked for REPUBLICAN NATIONAL COMMITTEE ~    3063  8.90e4   29.1    10
 8 Earmarked for TRUMP NATIONAL COMMITTEE JFC (~    3042  7.11e4   23.4     7.88
 9 Earmarked for MIKE JOHNSON FOR LOUISIANA (C0~    2811  2.53e4    9.01    2
10 Earmarked for TEAM ELISE (C00830679)             2389  3.13e4   13.1     4.75
# ... with 471 more rows, and abbreviated variable names 1: count_contribs,
#   2: total_amount, 3: mean_amount, 4: median_amount
```

We see something interesting here. The mean contribution amount is higher than the median amount in most cases, but the difference isn't huge. In some cases the mean gets skewed

by larger amounts. Examining both the median – which is less sensitive to extreme values – and the mean – which is more sensitive to extreme values – gives you a clearer picture of the composition of the data.

What about the highest and lowest amounts for each recipient? For that, we can use the `min()` and `max()` functions.

```
maryland_winred_contributions |>
  group_by(memo_text) |>
  summarise(
    count_contribs = n(),
    total_amount = sum(amount),
    mean_amount = mean(amount),
    median_amount = median(amount),
    min_amount = min(amount),
    max_amount = max(amount)
  ) |>
  arrange(desc(max_amount))
```

```
# A tibble: 481 x 7
   memo_text                     count~1 total~2 mean_~3 media~4 min_a~5 max_a~6
   <chr>                           <int>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 Earmarked for TRUMP 47 COMMI~       8  2.82e4  3520.   799.    200     10331.
 2 Earmarked for TRUMP NATIONAL~   41835  1.74e6    41.5   20.2     0.01   9702.
 3 Earmarked for HOGAN FOR MARY~    5754  4.79e5    83.3   26.0     1      6600
 4 Earmarked for HOVDE FOR WISC~     145  1.17e4    80.9    5.21    0.05   6600
 5 Earmarked for JOHN KENNEDY F~    1189  2.32e4    19.5    5       0.01   6600
 6 Earmarked for ROUNDS FOR SEN~       2  6.62e3  3312.  3312.     25      6600
 7 Earmarked for TIM SCOTT FOR ~     446  1.30e4    29.1    5       0.01   6500
 8 Earmarked for CAPITO FOR WES~       3  5.05e3  1683.    25      25      5000
 9 Earmarked for GUTHRIE VICTOR~       3  5.55e3  1851.   500      52.0    5000
10 Earmarked for MAX MILLER FOR~       3  6   e3  2000    500     500      5000
# ... with 471 more rows, and abbreviated variable names 1: count_contribs,
#   2: total_amount, 3: mean_amount, 4: median_amount, 5: min_amount,
#   6: max_amount
```

From this, we can see that some committees focus on small-dollar donors while others ask for (and get) larger amounts. This pattern isn't random: campaigns make choices about how they will raise money.

It would be interesting to see what the largest donation was. To do that, we could simply take our original data set and sort it from highest to lowest on the amount.

```
maryland_winred_contributions |>
  arrange(desc(amount))
```

```
# A tibble: 131,395 x 24
   linen~1 fec_c~2 tran_id flag_~3 org_n~4 last_~5 first~6 middl~7 prefix suffix
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>
 1 SA11AI  C00694~ AEB164~ IND     <NA>    Boland  John    NA      NA     NA
 2 SA11AI  C00694~ AED8E1~ IND     <NA>    Scully  Finbar  NA      NA     NA
 3 SA11AI  C00694~ A598DB~ IND     <NA>    Morgan  Kenneth NA      NA     NA
 4 SA11AI  C00694~ ABDDBD~ IND     <NA>    Jacobs~ Mark    NA      NA     NA
 5 SA11AI  C00694~ A6AE2F~ IND     <NA>    Jacobs~ Mark    NA      NA     NA
 6 SA11AI  C00694~ AA7420~ IND     <NA>    Jacobs~ Mark    NA      NA     NA
 7 SA11AI  C00694~ A05F19~ IND     <NA>    DETERM~ MARK    NA      NA     NA
 8 SA11AI  C00694~ AFB4D6~ IND     <NA>    Wheeler Rex     NA      NA     NA
 9 SA11AI  C00694~ A0490D~ IND     <NA>    Gallag~ Daniel  NA      NA     NA
10 SA11AI  C00694~ A2EF52~ IND     <NA>    Chaney  Bryan   NA      NA     NA
# ... with 131,385 more rows, 14 more variables: address_one <chr>,
#   address_two <chr>, city <chr>, state <chr>, zip <chr>, prigen <lgl>,
#   date <date>, amount <dbl>, aggregate_amount <dbl>, employer <chr>,
#   occupation <chr>, memo_code <lgl>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: org_name, 5: last_name, 6: first_name, 7: middle_name
```

Only two contributions of $10,000 or more. That's because the maximum contribution an individual can give for both a primary and a general election in this cycle is $3,300, but campaigns have to report whatever they are given, no matter how large (they likely will refund the difference).

11 Mutating data

Often the data you have will prompt questions that it doesn't immediately answer. The amount of donations or number of votes are great, but comparing absolute numbers to each other is only useful if you have a very small number. We need percentages!

To do that in R, we can use `dplyr` and `mutate` to calculate new metrics in a new field using existing fields of data. That's the essence of `mutate` - using the data you have to answer a new question.

So first we'll import the tidyverse so we can read in our data and begin to work with it.

```
library(tidyverse)
```

Now we'll import a dataset of county-level election results from Maryland's 2024 primary that is in the data folder in this chapter's pre-lab directory. We'll use this to explore ways to create new information from existing data.

```
primary_24 <- read_csv('data/maryland_primary_2024.csv')
```

```
Rows: 895 Columns: 11
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (5): county_name, office_name, office_district, candidate_name, party
dbl (6): early_voting, election_day, absentee, provisional, second_absentee,...

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.
```

First, let's add a column called `percent_election_day` for the percentage of votes that were cast on election day for each candidate result in a county. The code to calculate a percentage is pretty simple. Remember, with `summarize`, we used `n()` to count things. With `mutate`, we use very similar syntax to calculate a new value – a new column of data – using other values in our dataset.

If we look at what we got when we imported the data, you'll see there's `election_day` as the numerator, and we'll use `votes` as the denominator. We can simply things by only selecting a few columns.

```
primary_24 |>
  select(office_name, office_district, candidate_name, party, county_name, election_day, vote
  mutate(
  percent_election_day = election_day/votes
)
```

```
# A tibble: 895 x 8
   office_name   office_district candidate~1 party count~2 elect~3 votes perce~4
   <chr>         <chr>           <chr>       <chr> <chr>     <dbl> <dbl>   <dbl>
 1 U.S. Congress 06              Adrian Pet~ DEM   Allega~      10    12   0.833
 2 U.S. Congress 06              Altimont M~ DEM   Allega~       9    13   0.692
 3 U.S. Congress 06              April McCl~ DEM   Allega~     544  1428   0.381
 4 U.S. Congress 06              Ashwani Ja~ DEM   Allega~     125   228   0.548
 5 U.S. Congress 06              Brenda J. ~ REP   Allega~      53   120   0.442
 6 U.S. Congress 06              Chris Hyser REP   Allega~      99   190   0.521
 7 U.S. Congress 06              Dan Cox     REP   Allega~    1505  2130   0.707
 8 U.S. Congress 06              Destiny Dr~ DEM   Allega~      23    46   0.5
 9 U.S. Congress 06              Geoffrey G~ DEM   Allega~      12    37   0.324
10 U.S. Congress 06              George Glu~ DEM   Allega~      26    42   0.619
# ... with 885 more rows, and abbreviated variable names 1: candidate_name,
#   2: county_name, 3: election_day, 4: percent_election_day
```

Now we've got our `percent_election_day` column. But what do you see right away? Do those numbers look like we expect them to? No. They're a decimal expressed as a percentage. So let's fix that by multiplying by 100.

```
primary_24 |>
  select(office_name, office_district, candidate_name, party, county_name, election_day, vote
  mutate(
  percent_election_day = (election_day/votes)*100
)
```

```
# A tibble: 895 x 8
   office_name   office_district candidate~1 party count~2 elect~3 votes perce~4
   <chr>         <chr>           <chr>       <chr> <chr>     <dbl> <dbl>   <dbl>
 1 U.S. Congress 06              Adrian Pet~ DEM   Allega~      10    12    83.3
 2 U.S. Congress 06              Altimont M~ DEM   Allega~       9    13    69.2
 3 U.S. Congress 06              April McCl~ DEM   Allega~     544  1428    38.1
 4 U.S. Congress 06              Ashwani Ja~ DEM   Allega~     125   228    54.8
 5 U.S. Congress 06              Brenda J. ~ REP   Allega~      53   120    44.2
 6 U.S. Congress 06              Chris Hyser REP   Allega~      99   190    52.1
```

```
 7 U.S. Congress 06              Dan Cox     REP   Allega~    1505  2130    70.7
 8 U.S. Congress 06              Destiny Dr~ DEM   Allega~      23    46    50
 9 U.S. Congress 06              Geoffrey G~ DEM   Allega~      12    37    32.4
10 U.S. Congress 06              George Glu~ DEM   Allega~      26    42    61.9
# ... with 885 more rows, and abbreviated variable names 1: candidate_name,
#   2: county_name, 3: election_day, 4: percent_election_day
```

Now, does this ordering do anything for us? No. Let's fix that with arrange.

```
primary_24 |>
  select(office_name, office_district, candidate_name, party, county_name, election_day, vote
  mutate(
  percent_election_day = (election_day/votes)*100
)  |>
  arrange(desc(percent_election_day))
```

```
# A tibble: 895 x 8
   office_name   office_district candidate~1 party count~2 elect~3 votes perce~4
   <chr>         <chr>           <chr>       <chr> <chr>     <dbl> <dbl>   <dbl>
 1 U.S. Senator  <NA>            "Laban Y. ~ REP   Dorche~       5     5   100
 2 U.S. Senator  <NA>            "Laban Y. ~ REP   Kent C~       6     6   100
 3 U.S. Senator  <NA>            "John A. M~ REP   Somers~      33    39    84.6
 4 U.S. Congress 06              "Adrian Pe~ DEM   Allega~      10    12    83.3
 5 U.S. Senator  <NA>            "Moe H. Ba~ REP   Garret~      36    44    81.8
 6 U.S. Senator  <NA>            "Moe H. Ba~ REP   Allega~      33    41    80.5
 7 U.S. Congress 03              "Ray Bly"   REP   Carrol~      97   122    79.5
 8 U.S. Congress 06              "Kiambo \"~ DEM   Garret~      10    13    76.9
 9 U.S. Congress 03              "Robert J.~ REP   Carrol~     313   410    76.3
10 U.S. Congress 03              "Aisha Kha~ DEM   Carrol~      19    25    76
# ... with 885 more rows, and abbreviated variable names 1: candidate_name,
#   2: county_name, 3: election_day, 4: percent_election_day
```

So now we have results ordered by `percent_election_day` with the highest percentage first. To see the lowest percentage first, we can reverse that `arrange` function.

```
primary_24 |>
  select(office_name, office_district, candidate_name, party, county_name, election_day, vote
  mutate(
  percent_election_day = (election_day/votes)*100
)  |>
  arrange(percent_election_day)
```

```
# A tibble: 895 x 8
   office_name   office_district candidate~1 party count~2 elect~3 votes perce~4
   <chr>         <chr>           <chr>       <chr> <chr>     <dbl> <dbl>   <dbl>
 1 U.S. Congress 06              Mohammad S~ DEM   Garret~       0     2     0
 2 U.S. Congress 08              Jamie Rask~ DEM   Prince~       0     2     0
 3 U.S. Senator  <NA>            Steven Hen~ DEM   Talbot~       0     3     0
 4 U.S. Congress 06              Laurie-Ann~ DEM   Garret~       2    18    11.1
 5 U.S. Senator  <NA>            Brian E. F~ DEM   Dorche~       2    17    11.8
 6 U.S. Senator  <NA>            Brian E. F~ DEM   Talbot~       2    15    13.3
 7 U.S. Senator  <NA>            Marcellus ~ DEM   Talbot~       2    15    13.3
 8 U.S. Senator  <NA>            Brian E. F~ DEM   Garret~       1     7    14.3
 9 U.S. Senator  <NA>            Joseph Per~ DEM   Somers~       2    12    16.7
10 U.S. Senator  <NA>            Brian E. F~ DEM   Worces~       5    29    17.2
# ... with 885 more rows, and abbreviated variable names 1: candidate_name,
#   2: county_name, 3: election_day, 4: percent_election_day
```

Mutating can help us make different amounts easier to compare. We also can use filter to limit the data to meet certain conditions. For example, we could only look at the election day percentages of candidates who received at least 100 votes in a jurisdiction.

11.1 Another use of mutate

Mutate is also useful for standardizing data - for example, making different spellings of, say, cities into a single one.

Let's load some campaign contribution data - in this case Maryland donors to Republican committees via WinRed's online platform earlier this year - and take a look at the `city` column in our data.

```
maryland_cities <- read_csv("data/winred_md_cities.csv")
```

```
Rows: 469 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): city
dbl (2): count, sum

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.
```

You'll notice that there's a mix of styles: "Baltimore" and "BALTIMORE" for example. R will think those are two different cities, and that will mean that any aggregates we create based on city won't be accurate.

So how can we fix that? Mutate - it's not just for math! And a function called `str_to_upper` that will convert a character column into all uppercase. Now we can say exactly how many donations came from Baltimore (I mean, of course, BALTIMORE).

```
standardized_maryland_cities <- maryland_cities |>
  mutate(
    upper_city = str_to_upper(city)
)
```

Note that mutate doesn't literally combine similar records together - you'll still need to do another group_by and summarize block - but it does make your data more accurate. There are lots of potential uses for standardization - addresses, zip codes, anything that can be misspelled or abbreviated.

11.2 A more powerful use

Mutate is even more useful when combined with some additional functions. Let's focus on individual contributions from Maryland donors via WinRed; we'd like to group their donations by amount into one of four categories:

1. Under $100
2. $101-$499
3. $500-$1,499
4. $1,500-$2,999
5. More than $2,999

Mutate can make that happen by creating a new column and putting in a category value *based on the amount* of each record. First, let's load the individual contributions that we did in the previous chapter:

```
maryland_winred_contributions <- read_rds("data/maryland_winred.rds")

head(maryland_winred_contributions)
```

```
# A tibble: 6 x 24
  linenu~1 fec_c~2 tran_id flag_~3 org_n~4 last_~5 first~6 middl~7 prefix suffix
  <chr>    <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>
```

```
1 SA11AI   C00694~ A000BA~ IND     <NA>    Curro   Peter   NA      NA     NA
2 SA11AI   C00694~ A00110~ IND     <NA>    Mukai   Peggy   NA      NA     NA
3 SA11AI   C00694~ A0011E~ IND     <NA>    Smith   Alan    NA      NA     NA
4 SA11AI   C00694~ A001C3~ IND     <NA>    Saylor~ Jean    NA      NA     NA
5 SA11AI   C00694~ A00219~ IND     <NA>    Gillis~ Troy    NA      NA     NA
6 SA11AI   C00694~ A002A2~ IND     <NA>    Lally   Bryan   NA      NA     NA
# ... with 14 more variables: address_one <chr>, address_two <chr>, city <chr>,
#   state <chr>, zip <chr>, prigen <lgl>, date <date>, amount <dbl>,
#   aggregate_amount <dbl>, employer <chr>, occupation <chr>, memo_code <lgl>,
#   memo_text <chr>, cycle <dbl>, and abbreviated variable names 1: linenumber,
#   2: fec_committee_id, 3: flag_orgind, 4: org_name, 5: last_name,
#   6: first_name, 7: middle_name
```

Now that we've gotten a look, we can use `case_when` to give our new category column a value using some standard numeric logic:

```
maryland_winred_categories <- maryland_winred_contributions |>
  mutate(
    amount_category = case_when(
        amount < 100 ~ "Less than $100",
        amount >= 100 & amount < 500 ~ "Between $100 and $499",
        amount >= 500 & amount < 1500 ~ "Between $500 and $1499",
        amount >= 1500 & amount < 3000 ~ "Between $500 and $2999",
        amount >= 3000 ~ "$3,000 or more"
      )
  )
head(maryland_winred_categories)
```

```
# A tibble: 6 x 25
  linenu~1 fec_c~2 tran_id flag_~3 org_n~4 last_~5 first~6 middl~7 prefix suffix
  <chr>    <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>
1 SA11AI   C00694~ A000BA~ IND     <NA>    Curro   Peter   NA      NA     NA
2 SA11AI   C00694~ A00110~ IND     <NA>    Mukai   Peggy   NA      NA     NA
3 SA11AI   C00694~ A0011E~ IND     <NA>    Smith   Alan    NA      NA     NA
4 SA11AI   C00694~ A001C3~ IND     <NA>    Saylor~ Jean    NA      NA     NA
5 SA11AI   C00694~ A00219~ IND     <NA>    Gillis~ Troy    NA      NA     NA
6 SA11AI   C00694~ A002A2~ IND     <NA>    Lally   Bryan   NA      NA     NA
# ... with 15 more variables: address_one <chr>, address_two <chr>, city <chr>,
#   state <chr>, zip <chr>, prigen <lgl>, date <date>, amount <dbl>,
#   aggregate_amount <dbl>, employer <chr>, occupation <chr>, memo_code <lgl>,
#   memo_text <chr>, cycle <dbl>, amount_category <chr>, and abbreviated
#   variable names 1: linenumber, 2: fec_committee_id, 3: flag_orgind,
```

```
#   4: org_name, 5: last_name, 6: first_name, 7: middle_name
```

We can then use our new `amount_category` column in group_by statements to make summarizing easier:

```
maryland_winred_categories |>
  group_by(amount_category) |>
  summarize(total_amount = sum(amount)) |>
  arrange(desc(total_amount))
```

```
# A tibble: 5 x 2
  amount_category        total_amount
  <chr>                         <dbl>
1 Less than $100             1716339.
2 Between $100 and $499      1256984.
3 Between $500 and $1499      689883.
4 $3,000 or more              365885.
5 Between $500 and $2999      140922.
```

The largest category - by far - in dollar amount is the sub-$100 category, which makes sense for an online fundraising platform. Big little money.

Mutate is there to make your data more useful and to make it easier for you to ask more and better questions of it.

12 Working with dates

One of the most frustrating things in data is working with dates. Everyone has a different opinion on how to record them, and every software package on the planet has to sort it out. Dealing with it can be a little … confusing. And every dataset has something new to throw at you. So consider this an introduction.

First, there's the right way to display dates in data. Most of the rest of the world knows how to do this, but Americans aren't taught it. The correct way to display dates is the following format: YYYY-MM-DD, or 2024-09-15. Any date that looks different should be converted into that format when you're using R.

Luckily, this problem is so common that the Tidyverse has an entire library for dealing with it: lubridate.

We'll use a new library to solve most of the common problems before they start. If it's not already installed, just run `install.packages('lubridate')`

12.1 Making dates dates again

First, we'll import `tidyverse` like we always do and our newly-installed lubridate.

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

Let's start with a dataset of campaign expenses from Maryland political committees:

```
maryland_expenses <- read_csv("data/maryland_expenses.csv")
```

```
Rows: 5190 Columns: 13
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (12): expenditure_date, payee_name, address, payee_type, committee_name,...
dbl  (1): amount

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.
```

```
head(maryland_expenses)
```

```
# A tibble: 6 x 13
  expen~1 payee~2 address payee~3 amount commi~4 expen~5 expen~6 expen~7 expen~8
  <chr>   <chr>   <chr>   <chr>    <dbl> <chr>   <chr>   <chr>   <chr>   <chr>
1 3/1/22  Bank o~ 10 Lig~ Busine~   30.0 Gordon~ Other ~ Bank C~ <NA>    EFT
2 4/1/22  Bank o~ 10 Lig~ Busine~   30.0 Gordon~ Other ~ Bank C~ <NA>    EFT
3 5/1/22  Bank o~ 10 Lig~ Busine~   30.0 Gordon~ Other ~ Bank C~ <NA>    EFT
4 6/30/22 Meta    1 Hack~ Busine~   26.0 Hazel ~ Media   Online~ <NA>    Debit ~
5 6/13/22 Antiet~ 1000 W~ Busine~ 9506   Barr  ~ Media   TV      <NA>    Check
6 6/10/22 Meta    1 Hack~ Busine~   35.4 Ali  S~ Media   Online~ <NA>    Debit ~
# ... with 3 more variables: vendor <chr>, fundtype <chr>, comments <chr>, and
#   abbreviated variable names 1: expenditure_date, 2: payee_name,
#   3: payee_type, 4: committee_name, 5: expense_category, 6: expense_purpose,
#   7: expense_toward, 8: expense_method
```

Take a look at that first column, expenditure_date. It *looks* like a date, but see the `<chr` right below the column name? That means R thinks it's actually a character column. What we need to do is make it into an actual date column, which lubridate is very good at doing. It has a variety of functions that match the format of the data you have. In this case, the current format is `m/d/y`, and the lubridate function is called `mdy` that we can use with mutate:

```
maryland_expenses <- maryland_expenses |> mutate(expenditure_date=mdy(expenditure_date))

head(maryland_expenses)
```

```
# A tibble: 6 x 13
  expenditure_d~1 payee~2 address payee~3 amount commi~4 expen~5 expen~6 expen~7
  <date>          <chr>   <chr>   <chr>    <dbl> <chr>   <chr>   <chr>   <chr>
1 2022-03-01      Bank o~ 10 Lig~ Busine~   30.0 Gordon~ Other ~ Bank C~ <NA>
2 2022-04-01      Bank o~ 10 Lig~ Busine~   30.0 Gordon~ Other ~ Bank C~ <NA>
3 2022-05-01      Bank o~ 10 Lig~ Busine~   30.0 Gordon~ Other ~ Bank C~ <NA>
4 2022-06-30      Meta    1 Hack~ Busine~   26.0 Hazel ~ Media   Online~ <NA>
5 2022-06-13      Antiet~ 1000 W~ Busine~ 9506   Barr  ~ Media   TV      <NA>
6 2022-06-10      Meta    1 Hack~ Busine~   35.4 Ali  S~ Media   Online~ <NA>
# ... with 4 more variables: expense_method <chr>, vendor <chr>,
#   fundtype <chr>, comments <chr>, and abbreviated variable names
#   1: expenditure_date, 2: payee_name, 3: payee_type, 4: committee_name,
#   5: expense_category, 6: expense_purpose, 7: expense_toward
```

Now look at the expenditure_date column: R says it's a date column and it looks like we want it to: YYYY-MM-DD. Accept no substitutes.

Lubridate has functions for basically any type of character date format: mdy, ymd, even datetimes like ymd_hms.

That's less code and less weirdness, so that's good.

But to get clean data, I've installed a library and created a new field so I can now start to work with my dates. That seems like a lot, but don't think your data will always be perfect and you won't have to do these things.

Still, there's got to be a better way. And there is.

Fortunately, `readr` anticipates some date formatting and can automatically handle many of these issues (indeed it uses lubridate under the hood). When you are importing a CSV file, be sure to use `read_csv`, not `read.csv`.

Once again: use `read_csv`, not `read.csv` to import CSV data.

But you're not done with lubridate yet. It has some interesting pieces parts we'll use elsewhere.

For example, in spreadsheets you can extract portions of dates - a month, day or year - with formulas. You can do the same in R with lubridate. Let's say we wanted to add up the total amount spent in each month in our Maryland expenses data.

We could use formatting to create a Month field but that would group all the Aprils ever together. We could create a year and a month together, but that would give us an invalid date object and that would create problems later. Lubridate has something called a floor date that we can use.

So to follow along here, we're going to use mutate to create a month field, group by to lump them together, summarize to count them up and arrange to order them. We're just chaining things together.

```
maryland_expenses |>
  mutate(month = floor_date(expenditure_date, "month")) |>
  group_by(month) |>
  summarise(total_amount = sum(amount)) |>
  arrange(desc(total_amount))
```

```
# A tibble: 12 x 2
   month      total_amount
   <date>            <dbl>
 1 2022-06-01    11854494.
 2 2022-07-01     1807956.
```

```
 3 2022-05-01       94152.
 4 2022-01-01       18288.
 5 2022-03-01       18191.
 6 2022-04-01       11292.
 7 2022-02-01        5033.
 8 2020-07-01        1500
 9 2021-12-01         236.
10 2020-02-01          60
11 2021-03-01          20
12 NA                  NA
```

So the month of June 2022 had the most expenditures by far in this data. We'll be learning more about the calendar of campaigns and how it impacts campaign finance data.

13 Filters and selections

More often than not, we have more data than we want. Sometimes we need to be rid of that data. In `dplyr`, there's two ways to go about this: filtering and selecting.

Filtering creates a subset of the data based on criteria. All records where the amount is greater than 150,000. All records that match "College Park". Something like that. **Filtering works with rows – when we filter, we get fewer rows back than we start with.**

Selecting simply returns only the columns named. So if you only want to see city and amount, you select those fields. When you look at your data again, you'll have two columns. If you try to use one of your columns that you had before you used select, you'll get an error. **Selecting works with columns. You will have the same number of records when you are done, but fewer columns of data to work with.**

Now we'll import a dataset of county-level election results from Maryland's 2024 primary that is in the data folder in this chapter's pre-lab directory. It has results from all across the state, so one place to begin is by looking at individual jurisdictions - Maryland has 23 counties and one independent city, Baltimore. Let's start by loading tidyverse and reading in the Maryland data:

```
library(tidyverse)
```

```
primary_24 <- read_csv('data/maryland_primary_2024.csv')
```

```
Rows: 895 Columns: 11
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (5): county_name, office_name, office_district, candidate_name, party
dbl (6): early_voting, election_day, absentee, provisional, second_absentee,...

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.
```

The data we want to filter on is in `county_name`. So we're going to use filter and something called a comparison operator. We need to filter all records equal to "Prince George's". The

comparison operators in R, like most programming languages, are == for equal to, != for not equal to, > for greater than, >= for greater than or equal to and so on.

Be careful: = is not == and = is not "equal to". = is an assignment operator in most languages – how things get named.

```
prince_georges <- primary_24 |> filter(county_name == "Prince George's County")

head(prince_georges)
```

```
# A tibble: 6 x 11
  county~1 offic~2 offic~3 candi~4 party early~5 elect~6 absen~7 provi~8 secon~9
  <chr>    <chr>   <chr>   <chr>   <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 Prince ~ U.S. C~ 04      Emmett~ DEM       396    1851     511     184     666
2 Prince ~ U.S. C~ 04      Gabrie~ DEM       516    1936     458     230     821
3 Prince ~ U.S. C~ 04      George~ REP       449    1471     640      92     514
4 Prince ~ U.S. C~ 04      Glenn ~ DEM     10697   23889   11410    2048   16183
5 Prince ~ U.S. C~ 04      Joseph~ DEM       300    1774     554     183     614
6 Prince ~ U.S. C~ 05      Andrea~ DEM       640    1707     357     149     551
# ... with 1 more variable: votes <dbl>, and abbreviated variable names
#   1: county_name, 2: office_name, 3: office_district, 4: candidate_name,
#   5: early_voting, 6: election_day, 7: absentee, 8: provisional,
#   9: second_absentee
```

And just like that, we have just Prince George's results, which we can verify looking at the head, the first six rows.

We also have more data than we might want. For example, we may only want to work with the office, district, candidate name, party and votes.

To simplify our dataset, we can use select.

```
selected_prince_georges <- prince_georges |> select(office_name, office_district, candidate_n

head(selected_prince_georges)
```

```
# A tibble: 6 x 5
  office_name   office_district candidate_name    party votes
  <chr>         <chr>           <chr>             <chr> <dbl>
1 U.S. Congress 04              Emmett Johnson    DEM    3608
2 U.S. Congress 04              Gabriel Njinimbot DEM    3961
3 U.S. Congress 04              George McDermott  REP    3166
```

```
4 U.S. Congress 04              Glenn F. Ivey     DEM   64227
5 U.S. Congress 04              Joseph Gomes      DEM    3425
6 U.S. Congress 05              Andrea L. Crooms  DEM    3404
```

And now we only have five columns of data for whatever analysis we might want to do.

13.1 Combining filters

So let's say we wanted to see all the candidates for Senate and the number of votes each received in Prince George's County. We can do this a number of ways. The first is we can chain together a whole lot of filters.

```
prince_georges_senate <- primary_24 |> filter(county_name == "Prince George's County") |> fi

nrow(prince_georges_senate)
```

```
[1] 17
```

That gives us 17 candidates. But the code is repetitive, no? We can do better using boolean operators – AND and OR. In this case, AND is `&` and OR is `|`.

The difference? With AND, both things must be true to be included. With OR, any of those two things can be true and it will be included.

Here's the difference.

```
and_prince_georges <- primary_24 |> filter(county_name == "Prince George's County" & office_

nrow(and_prince_georges)
```

```
[1] 17
```

So AND gives us the same answer we got before. What does OR give us?

```
or_prince_georges <- primary_24 |> filter(county_name == "Prince George's County" | office_na

nrow(or_prince_georges)
```

```
[1] 428
```

So there's 428 rows that are EITHER in Prince George's OR are Senate results. OR is additive; AND is restrictive.

A general tip about using filter: it's easier to work your way towards the filter syntax you need rather than try and write it once and trust the result. Each time you modify your filter, check the results to see if they make sense. This adds a little time to your process but you'll thank yourself for doing it because it helps avoid mistakes.

14 Data Cleaning Part I: Data smells

Any time you are given a dataset from anyone, you should immediately be suspicious. Is this data what I think it is? Does it include what I expect? Is there anything I need to know about it? Will it produce the information I expect?

One of the first things you should do is give it the smell test.

Failure to give data the smell test can lead you to miss stories and get your butt kicked on a competitive story.

With data smells, we're trying to find common mistakes in data. For more on data smells, read the GitHub wiki post that started it all. Some common data smells are:

- Missing data or missing values
- Gaps in data
- Wrong type of data
- Outliers
- Sharp curves
- Conflicting information within a dataset
- Conflicting information across datasets
- Wrongly derived data
- Internal inconsistency
- External inconsistency
- Wrong spatial data
- Unusable data, including non-standard abbreviations, ambiguous data, extraneous data, inconsistent data

Not all of these data smells are detectable in code. You may have to ask people about the data. You may have to compare it to another dataset yourself. Does the agency that uses the data produce reports from the data? Does your analysis match those reports? That will expose wrongly derived data, or wrong units, or mistakes you made with inclusion or exclusion.

But with several of these data smells, we can do them first, before we do anything else.

We're going to examine three here as they apply to some precinct-level election results data: wrong type, missing data and gaps in data.

14.1 Wrong Type

First, let's look at **Wrong Type Of Data**.

We can sniff that out by looking at the output of `readr`.

Let's load the tidyverse.

```
# Remove scientific notation
options(scipen=999)
# Load the tidyverse
library(tidyverse)
```

Then let's load some precinct-level election results data from Texas for the 2020 general election.

This time, we're going to load the data in a CSV format, which stands for comma separated values and is essentially a fancy structured text file. Each column in the csv is separated – "delimited" – by a comma from the next column.

We're also going to introduce a new argument to our function that reads in the data, read_csv(), called "guess_max". As R reads in the csv file, it will attempt to make some calls on what "data type" to assign to each field: number, character, date, and so on. The "guess_max" argument says: look at the values in the whatever number of rows we specify before deciding which data type to assign. In this case, we'll pick 10.

```
# Load the data
texas_precinct_20 <- read_csv("data/tx_precinct_2020.csv", guess_max=10)
```

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,
e.g.:
  dat <- vroom(...)
  problems(dat)


Rows: 476915 Columns: 13
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (5): county, precinct, office, candidate, party
dbl (6): district, votes, absentee, election_day, early_voting, mail
lgl (2): provisional, limited

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.
```

Pay attention to the red warning that signals "one or more parsing issues." It advises us to run the problems() function to see what went wrong. Let's do that.

```
problems(texas_precinct_20)
```

```
# A tibble: 1,640 x 5
     row   col expected           actual file
   <int> <int> <chr>              <chr>  <chr>
 1 28450    12 1/0/T/F/TRUE/FALSE 20     ""
 2 28450    13 1/0/T/F/TRUE/FALSE 21     ""
 3 28451    12 1/0/T/F/TRUE/FALSE 15     ""
 4 28451    13 1/0/T/F/TRUE/FALSE 3      ""
 5 28465    12 1/0/T/F/TRUE/FALSE 20     ""
 6 28465    13 1/0/T/F/TRUE/FALSE 20     ""
 7 28466    12 1/0/T/F/TRUE/FALSE 15     ""
 8 28466    13 1/0/T/F/TRUE/FALSE 3      ""
 9 28472    12 1/0/T/F/TRUE/FALSE 22     ""
10 28472    13 1/0/T/F/TRUE/FALSE 17     ""
# ... with 1,630 more rows
```

It produces a table of all the parsing problems. It has 1,640 rows, which means we have that many problems. In almost every case here, the `readr` library has guessed that a given column was of a "logical" data type – True or False. It did it based on very limited information – only 1,000 rows. So, when it hit a value that looked like a date, or a character string, it didn't know what to do. So it just didn't read in that value correctly.

The easy way to fix this is to set the guess_max argument higher. It will take a little longer to load, but we'll use every single row in the data set to guess the column type – 476,915

```
texas_precinct_20 <- read_csv("data/tx_precinct_2020.csv", guess_max=476915)
```

```
Rows: 476915 Columns: 13
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (6): county, precinct, office, candidate, party, election_day
dbl (5): district, absentee, mail, provisional, limited
num (2): votes, early_voting

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.
```

This time, we got no parsing failures. And if we examine the data types `readr` assigned to each column using glimpse(), they generally make sense.

```
glimpse(texas_precinct_20)
```

```
Rows: 476,915
Columns: 13
$ county       <chr> "Newton", "Newton", "Newton", "Newton", "Newton", "Newton~
$ precinct     <chr> "Box 1", "Box 1", "Box 1", "Box 1", "Box 1", "Box 1", "Bo~
$ office       <chr> "Registered Voters", "Ballots Cast", "President", "Presid~
$ district     <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ candidate    <chr> NA, NA, "Donald J Trump", "Joseph R Biden", "Jo Jorgensen~
$ party        <chr> NA, NA, "REP", "DEM", "LBT", "GRE", NA, NA, NA, NA, NA, N~
$ votes        <dbl> 1003, 665, 557, 92, 6, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 553,~
$ absentee     <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ election_day <chr> NA, "141", "124", "11", "2", "2", "0", "0", "0", "0", "0"~
$ early_voting <dbl> NA, 524, 433, 81, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 430, 8~
$ mail         <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ provisional  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ limited      <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
```

Things that should be characters – like county, precinct, candidate – are characters (chr). Things that should be numbers (dbl) – like votes – are numbers.

There are some minor problems. The election_day column is a good example. It read in as a number (chr), even though there clearly are numbers in it judging from our initial inspection. Here's why: the original file has a single value in that column that is "5+".

```
texas_precinct_20 |> filter(election_day == "5+")
```

```
# A tibble: 1 x 13
  county   precinct office   distr~1 candi~2 party votes absen~3 elect~4 early~5
  <chr>    <chr>    <chr>      <dbl> <chr>   <chr> <dbl>   <dbl> <chr>     <dbl>
1 Anderson 11       Railroa~      NA Chryst~ DEM     141      NA 5+          136
# ... with 3 more variables: mail <dbl>, provisional <dbl>, limited <dbl>, and
#   abbreviated variable names 1: district, 2: candidate, 3: absentee,
#   4: election_day, 5: early_voting
```

Because this is just one result that's weird, we can fix it by comparing the other votes Castaneda received in Anderson to the county totals for her. The difference should be what that "5+"

value should be. I've done those calculations and it turns out that 49 is the actual likely value.

We can fix that pretty easily, by changing that value to "49" using `case_when` and then using `mutate` to make the entire column numeric.

```
texas_precinct_20 <- texas_precinct_20 |>
  mutate(election_day = case_when(
    election_day == '5+' ~ '49',
    TRUE ~ election_day
  ))

texas_precinct_20 <- texas_precinct_20 |> mutate(election_day = as.numeric(election_day))
```

When we glimpse() the dataframe again, it's been changed

```
glimpse(texas_precinct_20)
```

```
Rows: 476,915
Columns: 13
$ county       <chr> "Newton", "Newton", "Newton", "Newton", "Newton", "Newton~
$ precinct     <chr> "Box 1", "Box 1", "Box 1", "Box 1", "Box 1", "Box 1", "Bo~
$ office       <chr> "Registered Voters", "Ballots Cast", "President", "Presid~
$ district     <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ candidate    <chr> NA, NA, "Donald J Trump", "Joseph R Biden", "Jo Jorgensen~
$ party        <chr> NA, NA, "REP", "DEM", "LBT", "GRE", NA, NA, NA, NA, NA, N~
$ votes        <dbl> 1003, 665, 557, 92, 6, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 553,~
$ absentee     <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ election_day <dbl> NA, 141, 124, 11, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 123, 1~
$ early_voting <dbl> NA, 524, 433, 81, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 430, 8~
$ mail         <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ provisional  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ limited      <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
```

Now we've got numbers in the election_day column that we can add.

14.2 Missing Data

The second smell we can find in code is **missing data**.

We can do that by grouping and counting columns. In addition to identifying the presence of NA values, this method will also give us a sense of the distribution of values in those columns.

Let's start with the "mail" column, which represents the number of votes a candidate received in a precinct from ballots cast by mail. The following code groups by the mail column, counts the number in each group, and then sorts from highest to lowest. There are 402,345 NA values in this column. This is most of our rows, so that should give us some pause. Either counties didn't report votes by mail as a separate category or called it something else. This will impact how we can describe the data.

```
texas_precinct_20 |>
  group_by(mail) |>
  summarise(
    count=n()
  ) |>
  arrange(desc(count))
```

```
# A tibble: 30 x 2
    mail  count
   <dbl>  <int>
 1    NA 402345
 2     0  71589
 3     1   1708
 4     2    646
 5     3    277
 6     4    125
 7     5     79
 8     6     35
 9     7     21
10     8     21
# ... with 20 more rows
```

Now let's try the "provisional" column, which represents the number of accepted provisional votes cast. In this case, there are 135,073 NA values. The rest have different dollar amounts.

```
texas_precinct_20 |>
  group_by(provisional) |>
  summarise(
    count=n()
  ) |>
  arrange(desc(count))
```

```
# A tibble: 27 x 2
   provisional  count
         <dbl>  <int>
 1          NA 473381
 2           0   2836
 3           1    210
 4           2    105
 5           3     69
 6           5     53
 7           4     44
 8           6     44
 9          10     29
10           7     27
# ... with 17 more rows
```

The number of NA values - 473,381 - is even higher, which should give us confidence that most Texas counties did not report provisional votes at the precinct level. Like with mail votes, this helps define the constraints we have to work under with this data.

14.3 Gaps in data

Let's now look at **gaps in data**. It's been my experience that gaps in data often have to do with time, but there are other potential gaps, too. To illustrate those, we're going to introduce some voter registration data from Yadkin County, North Carolina. Let's load it and take a look:

```
yadkin_voters <- read_csv("data/yadkin_voters.csv")
```

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,
e.g.:
  dat <- vroom(...)
  problems(dat)
```

```
Rows: 28456 Columns: 67
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (37): county_desc, voter_reg_num, ncid, last_name, first_name, middle_n...
dbl   (9): county_id, zip_code, full_phone_number, birth_year, age_at_year_e...
lgl  (20): mail_addr3, mail_addr4, ward_abbrv, ward_desc, county_commiss_abb...
date  (1): registr_dt
```

```
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.
```

Each row represents a current or previously registered voter in the county, along with information about that person and the political jurisdictions they reside in. When we talk about gaps, often they indicate the administrative boundaries. Here's an example: let's find the most recent `registr_dt` in this dataset:

```
yadkin_voters |> arrange(desc(registr_dt))
```

```
# A tibble: 28,456 x 67
   count~1 count~2 voter~3 ncid  last_~4 first~5 middl~6 name_~7 statu~8 voter~9
     <dbl> <chr>   <chr>   <chr> <chr>   <chr>   <chr>   <chr>   <chr>   <chr>
 1      99 YADKIN  000000~ ER48~ FINCAN~ JOY     LYNN    <NA>    A       ACTIVE
 2      99 YADKIN  000000~ ER48~ LUCK    ERIN    EILEEN  <NA>    A       ACTIVE
 3      99 YADKIN  000000~ ER37~ OSBORNE EMILY   DANIEL~ <NA>    A       ACTIVE
 4      99 YADKIN  000000~ AC80~ TEMPLE  BRENDA  PUGH    <NA>    A       ACTIVE
 5      99 YADKIN  000000~ EN44~ EDWARDS ROCKY   ALLEN   <NA>    A       ACTIVE
 6      99 YADKIN  000000~ BN36~ HAWKINS COREY   ALAN    <NA>    A       ACTIVE
 7      99 YADKIN  000000~ BN27~ LING    ANDREW  STUART  <NA>    A       ACTIVE
 8      99 YADKIN  000000~ ER48~ SPICER  CHARLES STANLEY JR      A       ACTIVE
 9      99 YADKIN  000000~ CG17~ WHITAK~ SABRINA JOLEIG~ <NA>    A       ACTIVE
10      99 YADKIN  000000~ ER48~ GLEI    SALLY   COWIN   <NA>    A       ACTIVE
# ... with 28,446 more rows, 57 more variables: reason_cd <chr>,
#   voter_status_reason_desc <chr>, res_street_address <chr>,
#   res_city_desc <chr>, state_cd <chr>, zip_code <dbl>, mail_addr1 <chr>,
#   mail_addr2 <chr>, mail_addr3 <lgl>, mail_addr4 <lgl>, mail_city <chr>,
#   mail_state <chr>, mail_zipcode <chr>, full_phone_number <dbl>,
#   confidential_ind <chr>, registr_dt <date>, race_code <chr>,
#   ethnic_code <chr>, party_cd <chr>, gender_code <chr>, birth_year <dbl>, ...
```

It's July 14, 2022. That means that this dataset doesn't have any records newer than that, so if we were describing it we'd need to include that information.

What about the most recent `birth_year`?

```
yadkin_voters |> arrange(desc(birth_year))
```

```
# A tibble: 28,456 x 67
   count~1 count~2 voter~3 ncid  last_~4 first~5 middl~6 name_~7 statu~8 voter~9
```

```
     <dbl> <chr>   <chr>   <chr> <chr>   <chr>   <chr>   <chr>   <chr>   <chr>
 1      99 YADKIN  000000~ ER47~ ADAMS   BLAIN   NOAH    <NA>    A       ACTIVE
 2      99 YADKIN  000000~ ER47~ ALEXAN~ PERRY   NATHAN~ <NA>    A       ACTIVE
 3      99 YADKIN  000000~ ER48~ ALVARE~ KAREN   ESMERA~ <NA>    A       ACTIVE
 4      99 YADKIN  000000~ ER48~ ALVARE~ PABLO   <NA>    <NA>    A       ACTIVE
 5      99 YADKIN  000000~ ER47~ ARCADI~ JAZMIN  <NA>    <NA>    A       ACTIVE
 6      99 YADKIN  000000~ ER47~ ARELLA~ ALEX    <NA>    <NA>    A       ACTIVE
 7      99 YADKIN  000000~ ER47~ ARZATE~ ELIDETH <NA>    <NA>    A       ACTIVE
 8      99 YADKIN  000000~ ER47~ ARZATE~ JESSICA <NA>    <NA>    A       ACTIVE
 9      99 YADKIN  000000~ ER47~ BALL    ALEXIS  NYKOL   <NA>    A       ACTIVE
10      99 YADKIN  000000~ ER47~ BAUTIS~ ALEXAN~ ROSE    <NA>    A       ACTIVE
# ... with 28,446 more rows, 57 more variables: reason_cd <chr>,
#   voter_status_reason_desc <chr>, res_street_address <chr>,
#   res_city_desc <chr>, state_cd <chr>, zip_code <dbl>, mail_addr1 <chr>,
#   mail_addr2 <chr>, mail_addr3 <lgl>, mail_addr4 <lgl>, mail_city <chr>,
#   mail_state <chr>, mail_zipcode <chr>, full_phone_number <dbl>,
#   confidential_ind <chr>, registr_dt <date>, race_code <chr>,
#   ethnic_code <chr>, party_cd <chr>, gender_code <chr>, birth_year <dbl>, ...
```

Lots of 2004 records in there, which makes sense, since those folks are just becoming eligible to vote in North Carolina, where the minimum age is 18. In other words, we shouldn't see records in here where the "birth_year" is greater than 2004. If we do, we should ask some questions.

It's good to be aware of all gaps in data, but they don't always represent a problem.

14.4 Suspicious Outliers

Any time you are going to focus on a column for analysis, you should check for suspicious values. Are there any unusually large values or unusually small values? Are there any values that should not exist in the data?

Finally, let's first look at "registr_dt" again, so we can see if there's any missing months, or huge differences in the number of registrations by month. If we're going to work with dates, we should have `lubridate` handy for `floor_date`.

```
library(lubridate)
```

The `floor_date` function will allow us to group by month, instead of a single day.

```
yadkin_voters |>
  mutate(registration_month = floor_date(registr_dt, "month")) |>
  group_by(registration_month) |>
   summarise(
    count=n()
  ) |>
  arrange(registration_month)
```

```
# A tibble: 634 x 2
   registration_month count
   <date>             <int>
 1 1900-01-01            12
 2 1900-05-01             1
 3 1933-02-01             1
 4 1949-04-01             1
 5 1949-12-01             1
 6 1951-04-01             1
 7 1955-06-01             1
 8 1956-05-01             1
 9 1956-10-01             1
10 1960-04-01             1
# ... with 624 more rows
```

So, uh, if this data is accurate, then we have 13 registered voters who are more than 120 years old in Yadkin County. What's the most likely explanation for this? Some data systems have placeholder values when certain information isn't known or available. The next oldest registration month is from 1933, which seems plausible.

15 Data Cleaning Part II: Janitor

The bane of every data analyst's existence is data cleaning.

Every developer, every data system, every agency, the all have opinions about how data gets collected. Some decisions make sense from the outside. Some decisions are based entirely on internal politics: who is creating the data, how they are creating it, why they are creating it. Is it automated? Is it manual? Are data normalized? Are there free form fields where users can just type into or does the system restrict them to choices?

Your journalistic questions – what you want the data to tell you – is almost never part of that equation.

So cleaning data is the process of fixing issues in your data so you can answer the questions you want to answer. Data cleaning is a critical step that you can't skip past. A standard metric is that 80 percent of the time working with data will be spent cleaning and verifying data, and 20 percent the more exciting parts like analysis and visualization.

The tidyverse has a lot of built-in tools for data cleaning. We're also going to make use of a new library, called `janitor` that has a bunch of great functions for cleaning data. Let's load those now.

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

Now let's load a tiny slice of our Maryland WinRed contributions. To make the cleaning demonstration in this chapter easier, this dataset only has 14 rows, all from Conowingo, Maryland.

```
conowingo <- read_rds("data/conowingo.rds")
```

Let's glimpse it to get a sense of it, to examine the column data types and possible values.

```
glimpse(conowingo)
```

```
Rows: 14
Columns: 21
$ `1_linenumber`   <chr> "SA11AI", "SA11AI", "SA11AI", "SA11AI", "SA11AI", "SA~
$ fec_committee_id <chr> "C00694323", "C00694323", "C00694323", "C00694323", "~
$ tran_id          <chr> "AC8C7155172624C54909", "A61211F59EBD8451C8A8", "A9C8~
$ flag_orgind      <chr> "IND", "IND", "IND", "IND", "IND", "IND", "IND", "IND~
$ LAST_NAME        <chr> "Pabis", "Garvey", "Huddleston", "Huddleston", "Buona~
$ first_name       <chr> "Sherry", "Brian", "Deborah", "Deborah", "Wm", "Debor~
$ middle_name      <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA
$ prefix           <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA
$ suffix           <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA
$ `address one`    <chr> "144 Bill Leight Rd", "116 Finnegans Pl", "118 Merry ~
$ address_two      <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA
$ city             <chr> "Conowingo", "Conowing", "conowingo", "Conowingo", "C~
$ state            <chr> "MD", "MD", "MD", "MD", "MD", "MD", "MD", "MD", "MD",~
$ zip              <chr> "21918", "21918", "21918", "21918", "21918", "21918-2~
$ date             <date> 2022-04-16, 2022-06-24, 2022-06-15, 2022-06-18, 2022-~
$ amount           <chr> "45", "35", "25", "25", "25", "25", "15", "10", "5.76~
$ aggregate_amount <dbl> 180, 35, 45, 95, 41, 70, 15, 41, 41, 41, 41, 41, 41, ~
$ employer         <chr> "SELF-EMPLOYED", "US ARMY", "RETIRED", "RETIRED", "JR~
$ occupation       <chr> "SOFTWARE ENGINEER", "INFORMATION REQUESTED", "RETIR~
$ memo_text        <chr> "Earmarked for SAVE AMERICA JOINT FUNDRAISING COMMITT~
$ cycle            <dbl> 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022,~
```

And let's examine the full data set.

```
conowingo
```

```
# A tibble: 14 x 21
   1_lin~1 fec_c~2 tran_id flag_~3 LAST_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
```

```
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
14 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <chr>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: `1_linenumber`, 2: fec_committee_id,
#   3: flag_orgind, 4: LAST_NAME, 5: first_name, 6: middle_name,
#   7: `address one`
```

There are a number of issues with this data set that might get in the way of asking questions and receiving accurate answers. They are:

- The column headers are inconsistently styled (note: I've purposely dirtied these up, which is why they look different than previous versions of this data we've loaded). The first column "1_linenumber" starts with a number. The "NAME" column is all caps, while the rest are lowercase. And "address one" has a space in it. Those problems will make them hard to analyze, to refer to in functions we write.
- The amount column is stored as a character, not a number. If we try to do math to it – say, calculate the average loan size – it won't work.
- There's a fully duplicated row – a common problem in data sets. The first row is exactly the same as the second.
- The city field has five different forms – including misspellings – of Arnold. If we wanted to group and count the number of loans in Arnold, this inconsistency would not let us do that correctly.
- The zip field mixes five digit ZIP codes and nine digit ZIP codes. If we wanted to group and count the number of loans in a given ZIP code, this inconsistency would not let us do that correctly.
- The street address field is inconsistent. It has multiple variations of Merry Knoll Lane.

Let's get cleaning. Our goal will be to build up one block of code that does all the necessary cleaning in order to answer this question: what is the total amount of contributions from Conowingo, MD in ZIP code 21918?

15.1 Cleaning headers

One of the first places we can start with cleaning data is cleaning the column names (or headers).

Every system has their own way of recording headers, and every developer has their own thoughts of what a good idea is within it. R is most happy when headers are lower case, without special characters.

If column headers start with a number, or have a space in between two words, you have to set them off with backticks when using them in a function. Generally speaking, we want one word (or words separated by an underscore), all lowercase, that don't start with numbers.

The `janitor` library makes fixing headers trivially simple with the function `clean_names()`

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names()

# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 14 x 21
   x1_li~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
14 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <chr>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: x1_linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

This function changed `LAST_NAME` to `last_name`. It put an underscore in `address_one` to get rid of the space. And it changed `1_linenumber` to `x1_linenumber`. That last one was an improvement – it no longer starts with a number – but it's still kind of clunky.

We can use a tidyverse function `rename()` to fix that. Let's just call it `linenumber`. NOTE: when using `rename()`, the *new* name comes first.

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber)

# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 14 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
14 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <chr>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

15.2 Changing data types

Right now, the amount column is stored as a character. Do you see the little `<chr>` under the amount column in the table above? If we wanted to do math to it, we'd get an error, like so.

```
# cleaning function
total_conowingo <- cleaned_conowingo |>
  summarise(total_amount = sum(amount))
```

```
Error in `summarise()`:
i In argument: `total_amount = sum(amount)`.
Caused by error in `sum()`:
! invalid 'type' (character) of argument
```

```
# display the cleaned dataset
total_conowingo
```

```
Error in eval(expr, envir, enclos): object 'total_conowingo' not found
```

We got an "invalid 'type' (character)" error. So let's fix that using the mutate() function in concert with as.numeric(). We'll reuse the same column name, so it overwrites it.

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount))


# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 14 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
14 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
```

```
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

Notice that the amount has been converted to a `<dbl>`, which is short for double, a number format. When we attempt to add up all of the amounts to create a total, this time it works fine.

```
# cleaning function
total_conowingo <- cleaned_conowingo |>
  summarise(total_amount = sum(amount))

# display the cleaned dataset
total_conowingo
```

```
# A tibble: 1 x 1
  total_amount
         <dbl>
1          226
```

15.3 Duplicates

One of the most difficult problems to fix in data is duplicate records in the data. They can creep in with bad joins, bad data entry practices, mistakes – all kinds of reasons. A duplicated record isn't always there because of an error, but you need to know if it's there before making that determination.

So the question is, do we have any records repeated?

Here we'll use a function called `get_dupes` from the janitor library to check for fully repeated records in our cleaned data set.

```
cleaned_conowingo |>
  get_dupes()
```

```
No variable names specified - using all columns.
```

```
# A tibble: 2 x 22
  linenu~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
  <chr>    <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
1 SA11AI   C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
2 SA11AI   C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
# ... with 12 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>,
#   dupe_count <int>, and abbreviated variable names 1: linenumber,
#   2: fec_committee_id, 3: flag_orgind, 4: last_name, 5: first_name,
#   6: middle_name, 7: address_one
```

In this case, a contribution by Derrick Hamilton in our table is fully duplicated. Every field is identical in each.

We can fix this by adding the function `distinct()` to our cleaning script. This will keep only one copy of each unique record in our table

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount)) |>
  distinct()


# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 13 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
```

```
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

15.4 Cleaning strings

The rest of the problems with this data set all have to do with inconsistent format of values in a few of the columns. To fix these problems, we're going to make use of mutate() and a new function, `case_when()` in concert with "string functions" – special functions that allow us to clean up columns stored as character strings. The tidyverse package `stringr` has lots of useful string functions, more than we'll learn in this chapter.

Let's start by cleaning up the zip field. Remember, three of rows had a five-digit ZIP code, while two had a nine-digit ZIP code, separated by a hyphen.

We're going to write code that tells R to keep the first five digits on the left, and get rid of anything after that by using `mutate()` in concert with `str_sub()`, from the `stringr` package.

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount)) |>
  distinct() |>
  mutate(zip = str_sub(zip, start=1L, end=5L))


# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 13 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
```

```
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA      NA      118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA      NA      553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

Let's break down this line of code. It says: take the value in each zip column and extract the first character on the left (1L) through the fifth character on the left (5L), and then use that five-digit zip to overwrite the zip column.

We'll use a different set of functions to standardize how we standardize the different flavors of the word "Conowingo" in the city column. Let's start by changing every value to title case – first letter uppercase, subsequent letters lowercase – using the `str_to_title()` function from `stringr`.

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount)) |>
  distinct() |>
  mutate(zip = str_sub(zip, start=1L, end=5L)) |>
  mutate(city = str_to_title(city))


# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 13 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
```

```
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

That was enough to standardize two values (CONOWINGO and conowingo). The only ones that remain are the two clear misspellings (Conowing and Conowingoo). To fix those, we're going to do some manual editing. And for that, we're going to use `case_when()`, a function that let's us say if a value meets a certain condition, then change it, and if it doesn't, don't change it.

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount)) |>
  distinct() |>
  mutate(zip = str_sub(zip, start=1L, end=5L)) |>
  mutate(city = str_to_title(city)) |>
  mutate(city = case_when(
    city == "Conowing" ~ "Conowingo",
    TRUE ~ city
  ))

# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 13 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
```

```
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA      NA      144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA      NA      116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA      NA      118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA      NA      118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA      NA      118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA      NA      553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA      NA      84 Cle~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

This is a little complex, so let's break it down.

What the code above says, in English, is this: Look at all the values in the city column. If the value is "Conowing", then (that's what the "~" means, then) replace it with the word "Conowingo". If it's anything other than that (that's what "TRUE" means, otherwise), then keep the existing value in that column.

We could fix "Conowingoo" by adding another line inside that function, that looks identical: `city == "Conowingoo" ~ "Conowingo"`. Like so.

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount)) |>
  distinct() |>
  mutate(zip = str_sub(zip, start=1L, end=5L)) |>
  mutate(city = str_to_title(city)) |>
  mutate(city = case_when(
    city == "Conowing" ~ "Conowingo",
    city == "Conowingoo" ~ "Conowingo",
    TRUE ~ city
  ))
```

```
# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 13 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

Instead of specifying the exact value, we can also solve the problem by using something more generalizable, using a function called str_detect(), which allows us to search parts of words.

The second line of our case_when() function below now says, in English: look in the city column. If you find that one of the values starts with "Conowing" (the "^" symbol means "starts with"), then (the tilde ~ means then) change it to "Conowingo".

```
# cleaning function
cleaned_conowingo <- conowingo |>
  clean_names() |>
  rename(linenumber = x1_linenumber) |>
  mutate(amount = as.numeric(amount)) |>
  distinct() |>
  mutate(zip = str_sub(zip, start=1L, end=5L)) |>
  mutate(city = str_to_title(city)) |>
  mutate(city = case_when(
```

```
    str_detect(city,"^Conowing") ~ "Conowingo",
    TRUE ~ city
  ))


# display the cleaned dataset
cleaned_conowingo
```

```
# A tibble: 13 x 21
   linen~1 fec_c~2 tran_id flag_~3 last_~4 first~5 middl~6 prefix suffix addre~7
   <chr>   <chr>   <chr>   <chr>   <chr>   <chr>   <lgl>   <lgl>  <lgl>  <chr>
 1 SA11AI  C00694~ AC8C71~ IND     Pabis   Sherry  NA      NA     NA     144 Bi~
 2 SA11AI  C00694~ A61211~ IND     Garvey  Brian   NA      NA     NA     116 Fi~
 3 SA11AI  C00694~ A9C895~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 4 SA11AI  C00694~ ABAA82~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 5 SA11AI  C00694~ A45519~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 6 SA11AI  C00694~ A33987~ IND     Huddle~ Deborah NA      NA     NA     118 Me~
 7 SA11AI  C00694~ AE7C6D~ IND     Hamilt~ Derrick NA      NA     NA     553 Be~
 8 SA11AI  C00694~ A07D7A~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
 9 SA11AI  C00694~ A29033~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
10 SA11AI  C00694~ AD6C11~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
11 SA11AI  C00694~ A99BBC~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
12 SA11AI  C00694~ A142B2~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
13 SA11AI  C00694~ AF8E4C~ IND     Buonau~ Wm      NA      NA     NA     84 Cle~
# ... with 11 more variables: address_two <lgl>, city <chr>, state <chr>,
#   zip <chr>, date <date>, amount <dbl>, aggregate_amount <dbl>,
#   employer <chr>, occupation <chr>, memo_text <chr>, cycle <dbl>, and
#   abbreviated variable names 1: linenumber, 2: fec_committee_id,
#   3: flag_orgind, 4: last_name, 5: first_name, 6: middle_name, 7: address_one
```

By using str_detect(city,"^Conowing"), we pick up any values that start with "Conowing", so it would change all four of these. If we used city == "Conowing", it would only pick up one.

Lastly, there's the issue with inconsistent spelling of Merry Knoll Lane in the street address column. Do we need to clean this?

Remember the motivating question that's driving us to do this cleaning: what is the total amount of contributions from Conowingo, MD in ZIP code 21918?

We don't need the street_address field to answer that question. So we're not going to bother cleaning it.

That's a good approach for the future. A good rule of thumb is that you should only spend time cleaning fields that are critical to the specific analysis you want to do.

16 Data Cleaning Part III: Open Refine

Gather 'round kids and let me tell you a tale. Back in the previous century, Los Angeles Times journalists Sara Fritz and Dwight Morris wanted to answer this seemingly simple question: what do political campaigns spend their money on?

While campaigns are required to list a purpose of each expenditure, the problem is that they can choose what words to use. There's no standard dictionary or drop-down menu to choose from. Want to call that donut purchase "Food"? Sure. What about "Supplies for volunteers"? Works for me. How about "Meals"? Mom might disagree, but the FEC won't.

In order to answer their initial question, the reporters had to standardize their data. In other words, all food-related purchases had to be labeled "Food". All travel expenses had to be "Travel". It took them months - many months - to do this for every federal candidate.

I tell you this because if they had Open Refine, it would have taken them a week or two, not months.

I did data standardization before Open Refine, and every time I think about it, I get mad.

Fortunately (unfortunately?) several columns in the campaign finance data we'll work with are flawed in the same way that the LA Times' data was, so we can do this work in a better, faster way.

We're going to explore two ways into Open Refine: Through R, and through Open Refine itself.

16.1 Refinr, Open Refine in R

What is Open Refine?

Open Refine is a software program that has tools – algorithms – that find small differences in text and helps you fix them quickly. How Open Refine finds those small differences is through something called clustering. The algorithms behind clustering are not exclusive to Open Refine, so they can be used elsewhere.

Enter `refinr`, a package that contains the same clustering algorithms as Open Refine but all within R. Go ahead and install it if you haven't already by opening the console and running `install.packages("refinr")`. Then we can load libraries as we do.

```
library(tidyverse)
library(refinr)
library(janitor)
```

Let's load some campaign expenditure data focused on food-related expenses in Washington, D.C. Essentially, where campaigns spend their money on D.C. restaurants.

Now let's try and group and count the number of expenditures by recipient. To make it a bit more manageable, let's use another string function from `stringr` and filter for recipients that start with the uppercase "W" or lowercase "w" using the function `str_detect()` with a regular expression.

The filter function in the codeblock below says: look in the city column, and pluck out any value that starts with (the "^" symbol means "starts with") a lowercase "w" OR (the vertical "|", called a pipe, means OR) an uppercase "W".

```
dc_food |>
  group_by(recipient_name) |>
  summarise(
    count=n()
  ) |>
  filter(str_detect(recipient_name, '^w|^W')) |>
  arrange(recipient_name)
```

```
# A tibble: 50 x 2
   recipient_name      count
   <chr>               <int>
 1 W HOTELS                3
 2 W. MILLAR & CO          1
 3 W. MILLAR & CO.         5
 4 WA METRO                1
 5 WAGSHAL'S DELI          1
 6 WALMART SUPERCENTER    13
 7 WALTER'S SPORTS BAR     1
 8 WALTERS                 1
 9 WALTERS SPORT           1
10 WALTERS SPORTS          1
# ... with 40 more rows
```

There are several problems in this data that will prevent proper grouping and summarizing. We've learned several functions to do this manually.

By using the Open Refine package for R, `refinr`, our hope is that it can identify and standardize the data with a little more ease.

The first merging technique that's part of the `refinr` package we'll try is the `key_collision_merge`.

The key collision merge function takes each string and extracts the key parts of it. It then puts every key in a bin based on the keys matching.

One rule you should follow when using this is: **do not overwrite your original fields**. Always work on a copy. If you overwrite your original field, how will you know if it did the right thing? How can you compare it to your original data? To follow this, I'm going to mutate a new field called clean_city and put the results of key collision merge there.

```
cleaned_dc_food <- dc_food |>
  mutate(recipient_clean=key_collision_merge(recipient_name)) |>
  select(recipient_name, recipient_clean, everything()) |>
  arrange(recipient_clean)

cleaned_dc_food
```

```
# A tibble: 13,108 x 50
   recipient_n~1 recip~2 commi~3 commi~4 repor~5 repor~6 image~7 line_~8 trans~9
   <chr>         <chr>   <chr>   <chr>     <dbl> <chr>     <dbl> <chr>   <chr>
 1 &PIZZA        &PIZZA  C00003~ REPUBL~    2022 M4      2.02e17 21B     SB21B-~
 2 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2021 M3      2.02e17 21B     SB21B-~
 3 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2021 M6      2.02e17 21B     SB21B-~
 4 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2021 M12     2.02e17 21B     SB21B-~
 5 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2022 M3      2.02e17 21B     SB21B-~
 6 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2022 M3      2.02e17 21B     SB21B-~
 7 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2021 M10     2.02e17 21B     SB21B-~
 8 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2022 M4      2.02e17 21B     SB21B-~
 9 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2022 M5      2.02e17 21B     SB21B-~
10 1 WEST DUPONT 1 WEST~ C00003~ REPUBL~    2022 M5      2.02e17 21B     SB21B-~
# ... with 13,098 more rows, 41 more variables: file_number <dbl>,
#   entity_type <chr>, entity_type_desc <chr>,
#   unused_recipient_committee_id <chr>, recipient_committee_id <chr>,
#   recipient_state <chr>, beneficiary_committee_name <chr>,
#   national_committee_nonfederal_account <lgl>, disbursement_type <lgl>,
#   disbursement_type_description <lgl>, disbursement_description <chr>,
#   memo_code <chr>, memo_code_full <lgl>, disbursement_date <chr>, ...
```

To examine changes `refinr` made, let's examine the changes it made to cities that start with the letter "W".

```
cleaned_dc_food |>
  group_by(recipient_name, recipient_clean) |>
  summarise(
    count=n()
  ) |>
  filter(str_detect(recipient_clean, "^w|^W")) |>
  arrange(recipient_clean)
```

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

```
# A tibble: 52 x 3
# Groups:   recipient_name [52]
   recipient_name      recipient_clean     count
   <chr>               <chr>               <int>
 1 W HOTELS            W HOTELS                3
 2 W. MILLAR & CO      W. MILLAR & CO.         1
 3 W. MILLAR & CO.     W. MILLAR & CO.         5
 4 WA METRO            WA METRO                1
 5 WAGSHAL'S DELI      WAGSHAL'S DELI          1
 6 WALMART SUPERCENTER WALMART SUPERCENTER    13
 7 WALTERS             WALTERS                 1
 8 WALTERS SPORT       WALTERS SPORT           1
 9 WALTERS SPORTS      WALTERS SPORTS          1
10 WALTER'S SPORTS BAR WALTERS SPORTS BAR      1
# ... with 42 more rows
```

You can see several changes on the second page of results, including that refinr made "PIZZA, WE THE" into "WE THE PIZZA" which is pretty smart. Other potential changes, grouping together "WASHINGTON NATIONALS" and "WASHINGTON NATIONALS BASEBALL CLUB", didn't happen. Key collision will do well with different cases, but all of our records are upper case.

There's another merging algorithim that's part of refinr that works a bit differently, called `n_gram_merge()`. Let's try applying that one.

```
cleaned_dc_food <- dc_food |>
  mutate(recipient_clean=n_gram_merge(recipient_name)) |>
  select(recipient_name, recipient_clean, everything())

cleaned_dc_food
```

```
# A tibble: 13,108 x 50
   recipient_n~1 recip~2 commi~3 commi~4 repor~5 repor~6 image~7 line_~8 trans~9
   <chr>         <chr>   <chr>   <chr>     <dbl> <chr>     <dbl> <chr>   <chr>
 1 ST. REGIS HO~ ST. RE~ C00658~ MARK G~    2021 YE      2.02e17 17      SB17.I~
 2 HARKER, GRAY  HARKER~ C00536~ JOBS, ~    2021 YE      2.02e17 21B     SB21B.~
 3 REPUBLICAN N~ REPUBL~ C00055~ NY REP~    2021 M2      2.02e17 21B     B35A42~
 4 JOE'S SEAFOO~ JOE'S ~ C00551~ OORAH!~    2021 M4      2.02e17 21B     SB21B.~
 5 NATIONAL DEM~ NATION~ C00391~ JIM CO~    2021 Q1      2.02e17 17      BA7DD5~
 6 NATIONAL DEM~ NATION~ C00454~ MARCIA~    2021 Q1      2.02e17 17      SB17.2~
 7 CAFE MILANO   CAFE M~ C00391~ JIM CO~    2021 Q1      2.02e17 17      B68639~
 8 CAPITOL HILL~ CAPITO~ C00383~ CONAWA~    2021 Q1      2.02e17 17      BA078C~
 9 CAPITOL HILL~ CAPITO~ C00725~ CLIFF ~    2021 Q1      2.02e17 17      17883
10 CAPITOL HILL~ CAPITO~ C00383~ CONAWA~    2021 Q1      2.02e17 17      B51CE8~
# ... with 13,098 more rows, 41 more variables: file_number <dbl>,
#   entity_type <chr>, entity_type_desc <chr>,
#   unused_recipient_committee_id <chr>, recipient_committee_id <chr>,
#   recipient_state <chr>, beneficiary_committee_name <chr>,
#   national_committee_nonfederal_account <lgl>, disbursement_type <lgl>,
#   disbursement_type_description <lgl>, disbursement_description <chr>,
#   memo_code <chr>, memo_code_full <lgl>, disbursement_date <chr>, ...
```

To examine changes `refinr` made with this algorithm, let's again look at recipients starting with W. We see there wasn't a substantial change from the previous method.

```
cleaned_dc_food |>
  group_by(recipient_name, recipient_clean) |>
  summarise(
    count=n()
  ) |>
  filter(str_detect(recipient_clean, "^w|^W")) |>
  arrange(recipient_clean)
```

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

```
# A tibble: 51 x 3
# Groups:   recipient_name [51]
   recipient_name      recipient_clean     count
   <chr>               <chr>               <int>
 1 W HOTELS            W HOTELS                3
 2 W. MILLAR & CO      W. MILLAR & CO.         1
```

```
 3 W. MILLAR & CO.      W. MILLAR & CO.          5
 4 WA METRO             WA METRO                 1
 5 WAGSHAL'S DELI       WAGSHAL'S DELI           1
 6 WALMART SUPERCENTER  WALMART SUPERCENTER     13
 7 WALTERS              WALTERS                  1
 8 WALTERS SPORT        WALTERS SPORT            1
 9 WALTERS SPORTS       WALTERS SPORT            1
10 WALTER'S SPORTS BAR  WALTERS SPORTS BAR       1
# ... with 41 more rows
```

This method also made some good changes, but not in every case. No single method will be perfect and often a combination is necessary.

That's how you use the Open Refine r package, refinr.

Now let's upload the data to the interactive version of OpenRefine, which really shines at this task.

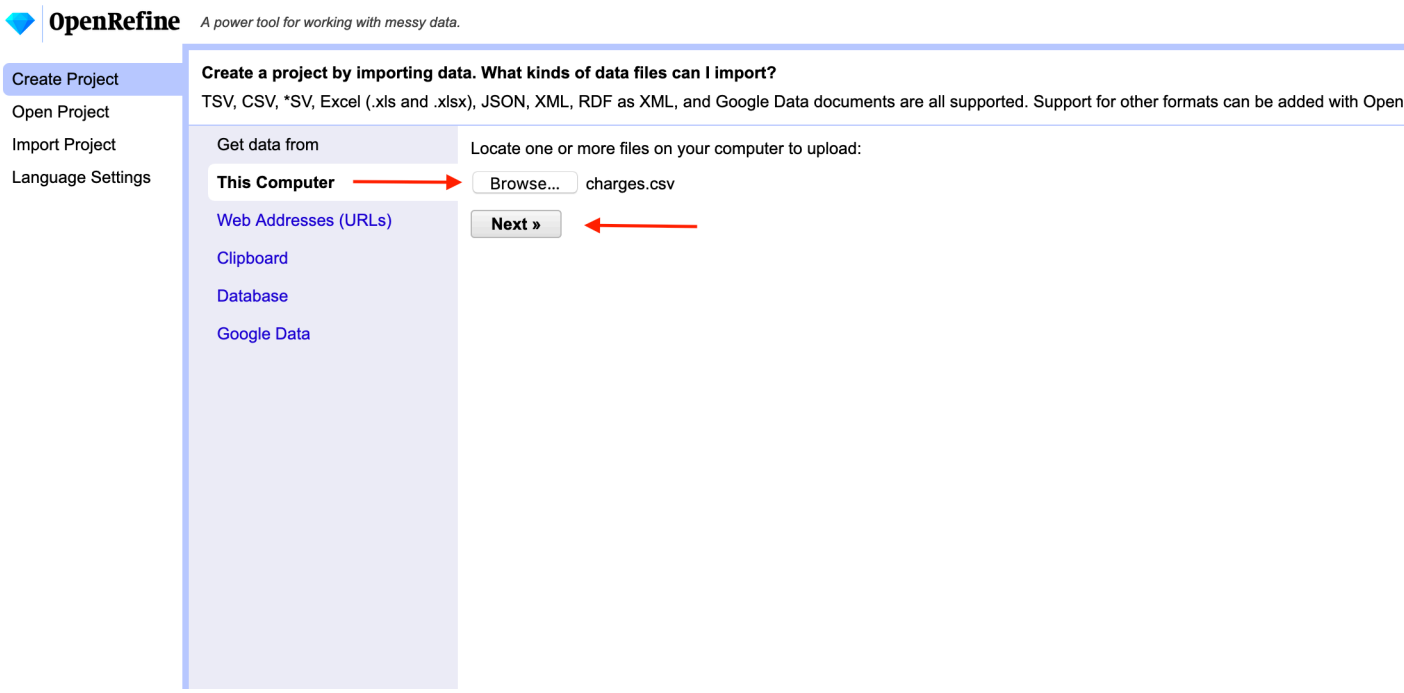
16.2 Manually cleaning data with Open Refine

Open Refine is free software. You should download and install it; the most recent version is 3.6.0. Refinr is great for quick things on smaller datasets that you can check to make sure it's not up to any mischief.

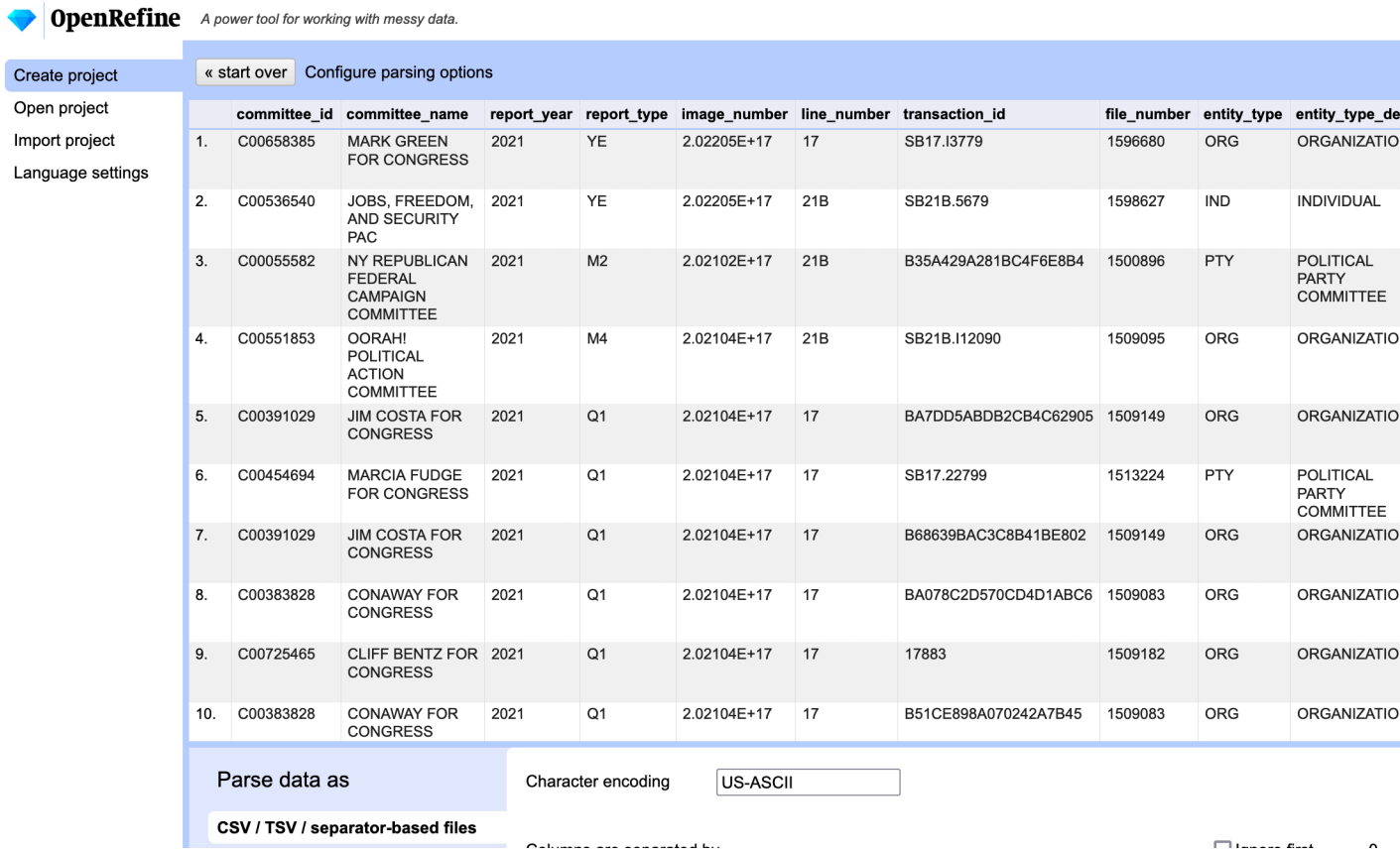
For bigger datasets, Open Refine is the way to go. And it has a lot more tools than refinr does (by design).

After you install it, run it. (If you are on a Mac it might tell you that it can't run the program. Go to System Preferences -> Security & Privacy -> General and click "Open Anyway".) Open Refine works in the browser, and the app spins up a small web server visible only on your computer to interact with it. A browser will pop up automatically.

You first have to import your data into a project. Click the choose files button and upload a csv of the DC food-related expenditures.



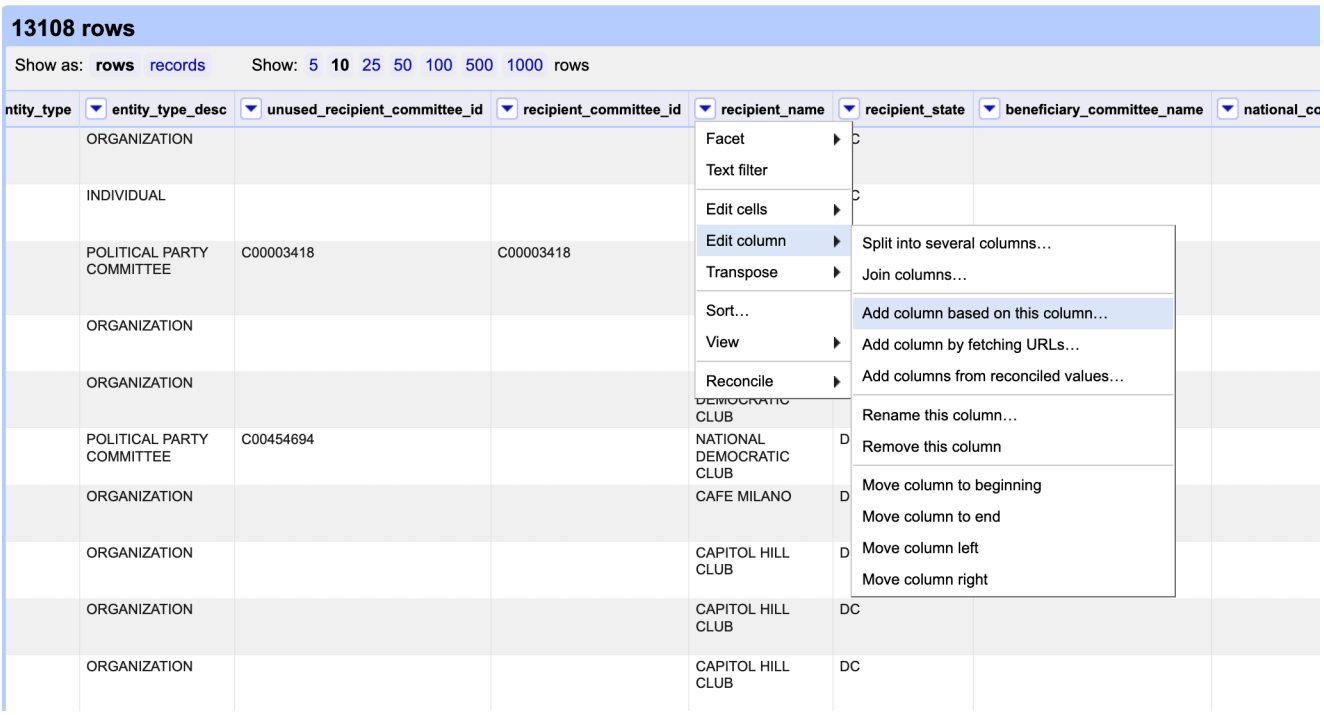
After your data is loaded into the app, you'll get a screen to look over what the data looks like. On the top right corner, you'll see a button to create the project. Click that.



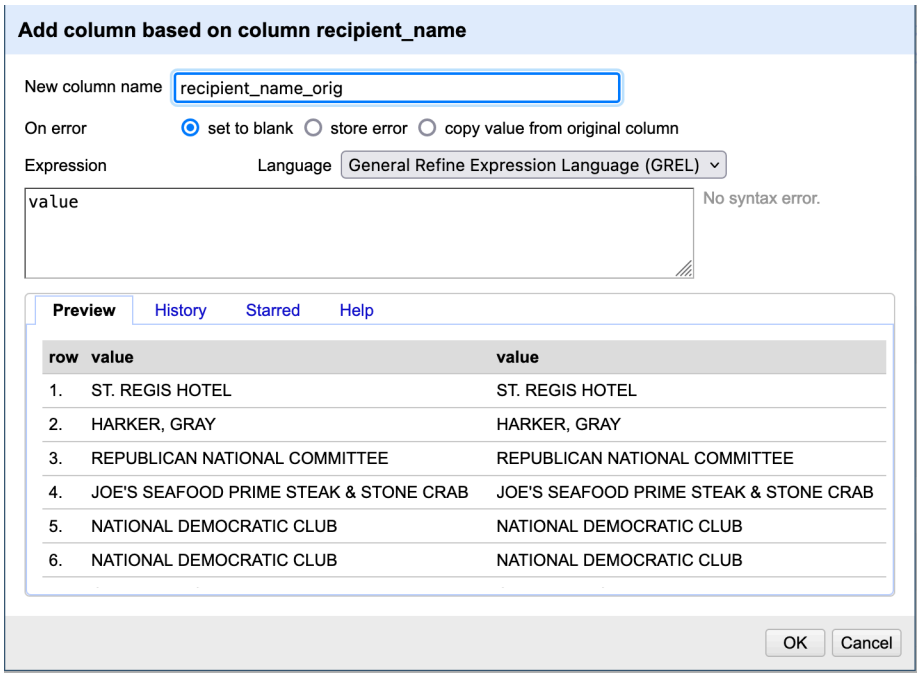
Open Refine has many, many tools. We're going to use one piece of it, as a tool for data cleaning. To learn how to use it, we're going to clean the "recipient_name" field.

First, let's make a copy of the original recipient_name column so that we can preserve the original data while cleaning the new one.

Click the dropdown arrow next to the recipient_name column, choose "edit column" > "Add column based on this column":

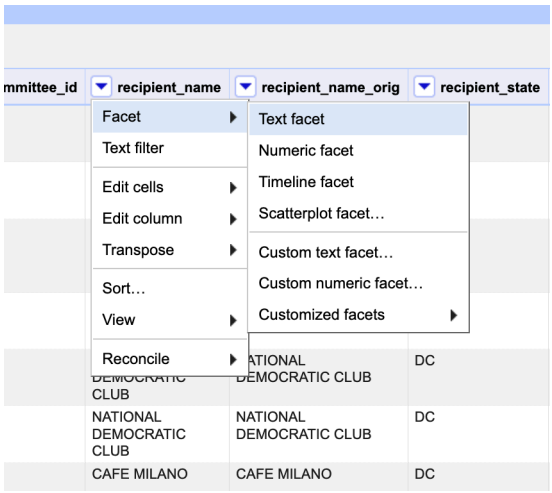


On the window that pops up, type “recipient_name_orig” in the “new column name” field. Then hit the OK button.



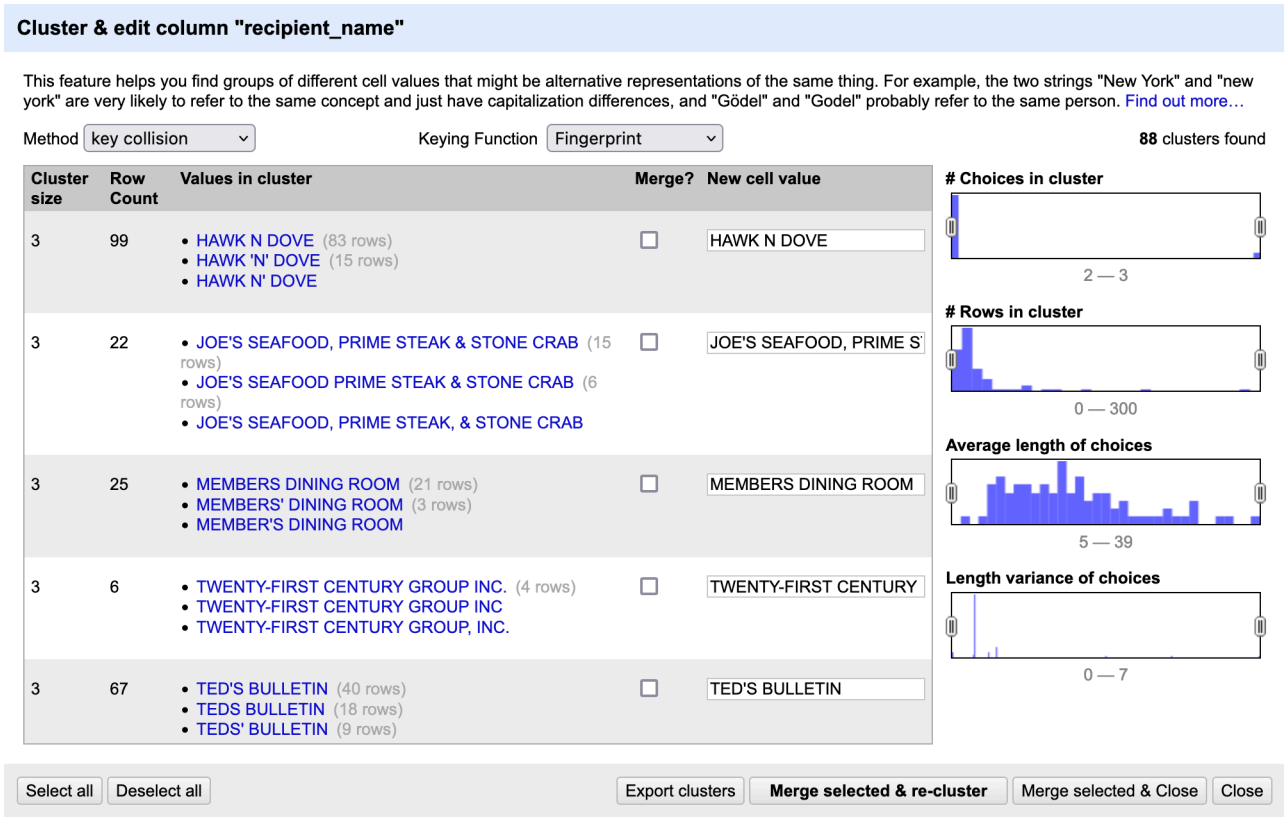
Now, let’s get to work cleaning the recipient_name column.

Next to the recipient_name field name, click the down arrow, then facet, then text facet.



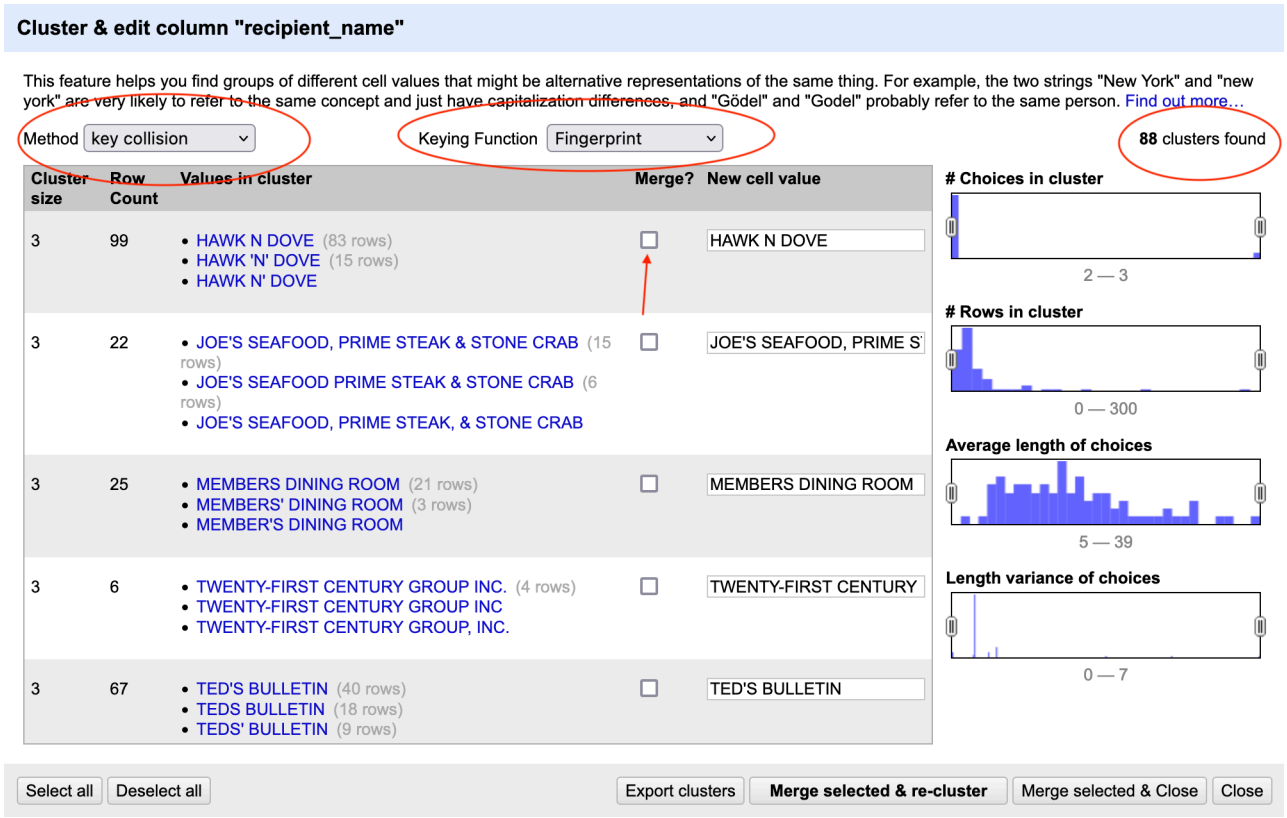
After that, a new box will appear on the left. It tells us how many unique recipient_names there are: 1,655. And, there's a button on the right of the box that says Cluster.

Click the cluster button. A new window will pop up, a tool to help us identify things that need to be cleaned, and quickly clean them.



The default "method" used is a clustering algorithim called "key collision", using the fingerprint function. This is the same method we used with the refinr package above.

At the top, you'll see which method was used, and how many clusters that algorithm identified. There are several different methods, each of which work slightly differently and produce different results.



Then, below that, you can see what those clusters are. Right away, we can see how useful this program is. It identified 99 rows that have some variation on "Hawk N Dove" in the recipient_name field. It proposed changing them all to "HAWK N DOVE".

Using human judgment, you can say if you agree with the cluster. If you do, click the "merge" checkbox. When it merges, the new result will be what it says in New Cell Value. Most often, that's the row with the most common result. You also can manually edit the "New Cell Value" if you want it to be something else:

Now begins the fun part: You have to look at all 88 clusters found and decide if they are indeed valid. The key collision method is very good, and very conservative. You'll find that most of them are usually valid.

Be careful! If you merge two things that aren't supposed to be together, it will change your data in a way that could lead to inaccurate results.

When you're done, click Merge Selected and Re-Cluster.

If any new clusters come up, evaluate them. Repeat until either no clusters come up or the clusters that do come up are ones you reject.

Now. Try a new method, maybe the "nearest neighbor levenshtein" method. Notice that it finds even more clusters - 167 - using a slightly different approach.

Rinse and repeat.

You'll keep doing this, and if the dataset is reasonably clean, you'll find the end.

When you're finished cleaning, click "Merge Selected & Close".

Then, export the data as a csv so you can load it back into R.

rows

rows records Show: 5 **10** 25 50 100 500 1000 rows

file_number	entity_type	entity_type_desc	unused_recipient_committee_id	recipient_committee_id	recipient_name	recipient_name_orig	recipient_state	beneficia
1596680	ORG	ORGANIZATION			ST. REGIS HOTEL	ST. REGIS HOTEL	DC	
1598627	IND	INDIVIDUAL			HARKER, GRAY	HARKER, GRAY	DC	
1500896	PTY	POLITICAL PARTY COMMITTEE	C00003418	C00003418	REPUBLICAN NATIONAL COMMITTEE	REPUBLICAN NATIONAL COMMITTEE	DC	REPUBLICAN COMMITTEE
1509095	ORG	ORGANIZATION			JOE'S SEAFOOD PRIME STEAK & STONE CRAB	JOE'S SEAFOOD PRIME STEAK & STONE CRAB	DC	
1509149	ORG	ORGANIZATION			NATIONAL DEMOCRATIC CLUB	NATIONAL DEMOCRATIC CLUB	DC	
1513224	PTY	POLITICAL PARTY COMMITTEE	C00454694		NATIONAL DEMOCRATIC CLUB	NATIONAL DEMOCRATIC CLUB	DC	MARCIA FUD CONGRESS
1509149	ORG	ORGANIZATION			CAFE MILANO	CAFE MILANO	DC	
1509083	ORG	ORGANIZATION			CAPITOL HILL CLUB	CAPITOL HILL CLUB	DC	
1509182	ORG	ORGANIZATION			CAPITOL HILL CLUB	CAPITOL HILL CLUB	DC	
1509083	ORG	ORGANIZATION			CAPITOL HILL CLUB	CAPITOL HILL CLUB	DC	

A question for all data analysts – if the dataset is bad enough, can it ever be cleaned?

There's no single definitive answer. You have to find it yourself.

17 Combining and joining

Often, as data journalists, we're looking at data across time or at data stored in multiple tables. And to do that, we need to often need to merge that data together.

Depending on what we have, we may just need to stack data on top of each other to make new data. If we have 2019 data and 2018 data and we want that to be one file, we stack them. If we have a dataset of cows in counties and a dataset of populations in county, we're going to join those two together on the county – the common element.

Let's explore.

17.1 Combining data (stacking)

Let's say that we have Maryland county voter registration data from five different elections in five different files. They have the same record layout and the same number of counties (plus Baltimore City). We can combine them into a single dataframe.

Let's do what we need to import them properly. I've merged it all into one step for each of the datasets.

```
library(tidyverse)
```

```
county_voters_2016 <- read_csv("data/county_voters_2016.csv")
```

```
Rows: 25 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): COUNTY
dbl (7): YEAR, DEM, REP, LIB, UNA, OTH, TOTAL

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.
```

```
county_voters_2018 <- read_csv("data/county_voters_2018.csv")
```

```
Rows: 25 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): COUNTY
dbl (7): YEAR, DEM, REP, LIB, UNA, OTH, TOTAL

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.
```

```
county_voters_2020 <- read_csv("data/county_voters_2020.csv")
```

```
Rows: 25 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): COUNTY
dbl (7): YEAR, DEM, REP, LIB, OTH, UNA, TOTAL

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.
```

```
county_voters_2022 <- read_csv("data/county_voters_2022.csv")
```

```
Rows: 25 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): COUNTY
dbl (7): YEAR, DEM, REP, LIB, OTH, UNA, TOTAL

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.
```

```
county_voters_2024 <- read_csv("data/county_voters_2024.csv")
```

```
Rows: 25 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): COUNTY
```

```
dbl (7): YEAR, DEM, REP, LIB, UNA, OTH, TOTAL

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.
```

All of these datasets have the same number of columns, all with the same names, so if we want to merge them together to compare them over time, we need to stack them together. The verb here, in R, is `bind_rows`. You tell the function what you want to combine and it does it, assuming that you've got column names in common containing identically formatted data.

Since we have five dataframes, we're going to need to pass them as a list, meaning they'll be enclosed inside the `list` function.

```
county_voters_combined <- bind_rows(list(county_voters_2016, county_voters_2018, county_voter
```

And boom, like that, we have 125 rows of data together instead of five dataframes. Now we can ask more interesting questions like how a county's registration patterns have changed over time.

There are plenty of uses for `bind_rows`: any regularly updated data that comes in the same format like crime reports or award recipients or player game statistics. Or election results.

17.2 Joining data

More complicated is when you have two separate tables that are connected by a common element or elements. But there's a verb for that, too: `join`.

Let's start by reading in some Maryland 2020 county population data:

```
maryland_population <- read_csv('data/maryland_population_2020.csv')
```

```
Rows: 24 Columns: 2
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): COUNTY
dbl (1): POP2020

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.
```

One of the columns we have is called `county`, which is what we have in our county_voters_2020 dataframe.

To put the Maryland population data and voter registration data together, we need to use something called a join. There are different kinds of joins. It's better if you think of two tables sitting next to each other. A `left_join` takes all the records from the left table and only the records that match in the right one. A `right_join` does the same thing. An `inner_join` takes only the records where they are equal. There's one other join – a `full_join` which returns all rows of both, regardless of if there's a match – but I've never once had a use for a full join.

In the best-case scenario, the two tables we want to join share a common column. In this case, both of our tables have a column called `county` that has the same characteristics: values in both look identical, including how they distinguish Baltimore City from Baltimore County. This is important, because joins work on *exact matches*.

We can do this join multiple ways and get a similar result. We can put the population file on the left and the registration data on the right and use a left join to get them all together. And we use `by=` to join by the correct column. I'm going to count the rows at the end. The reason I'm doing this is important: **Rule 1 in joining data is having an idea of what you are expecting to get**. So with a left join with population on the left, I have 24 rows, so I expect to get 24 rows when I'm done.

```
maryland_population |> left_join(county_voters_2020, by="COUNTY") |> nrow()
```

```
[1] 24
```

Remove the nrow and run it again for yourself. By default, `dplyr` will do a "natural" join, where it'll match all the matching columns in both tables. So if we take out the by, it'll use all the common columns between the tables. That may not be right in every instance but let's try it. If it works, we should get 24 rows.

```
maryland_population |> left_join(county_voters_2020)
```

```
Joining with `by = join_by(COUNTY)`
```

```
# A tibble: 24 x 9
   COUNTY           POP2020  YEAR    DEM    REP   LIB   OTH    UNA  TOTAL
   <chr>              <dbl> <dbl>  <dbl>  <dbl> <dbl> <dbl>  <dbl>  <dbl>
 1 Allegany           68106  2020  12820  22530   204   508   7674  43736
 2 Anne Arundel      588261  2020 174494 135457  1922  3581  90162 405616
 3 Baltimore City    585708  2020 311610  30163   951  4511  52450 399685
 4 Baltimore County  854535  2020 313870 142534  2227  7201 100576 566408
```

```
 5 Calvert            92783  2020  24587  28181   332   706  14178  67984
 6 Caroline           33293  2020   6629  10039    86   215   4208  21177
 7 Carroll           172891  2020  33662  63967   670  1292  25770 125361
 8 Cecil             103725  2020  21601  30880   341   887  15110  68819
 9 Charles           166617  2020  72416  24711   349   977  19849 118302
10 Dorchester         32531  2020   9848   8730    78   183   3348  22187
# ... with 14 more rows
```

Since we only have one column in common between the two tables, the join only used that column. And we got the same answer. If we had more columns in common, you could see in your results columns with .X after them - that's a sign of duplicative columns between two tables, and you may decide you don't need both moving forward.

Let's save our joined data to a new dataframe, but this time let's remove the select function so we don't limit the columns to just three.

```
maryland_population_with_voters <- maryland_population |> left_join(county_voters_2020)
```

```
Joining with `by = join_by(COUNTY)`
```

Now, with our joined data, we can answer questions in a more useful way. But joins can do even more than just bring data together; they can include additional data to enable you to ask more sophisticated questions. Right now we have registered voters and total population. But we can do more.

Let's try adding more Maryland demographic data to the mix. Using a file describing the 18-and-over population (from which eligible voters come) from the state's data catalog, we can read it into R:

```
maryland_demographics <- read_csv('data/maryland_demographics.csv')
```

```
Rows: 24 Columns: 11
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (1): NAME
dbl (10): GEOCODE, pop_18_over, pop_one_race, pop_white, pop_black, pop_nati...

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.
```

Again, we can use a `left_join` to make our demographic data available. This time we'll need to specify the two fields to join because they do not have identical names. We'll use `COUNTY` from our population data and `NAME` from the demographic data, and the order matters - the first column is from the dataframe you name *first*.

```
maryland_population_with_voters_and_demographics <- maryland_population_with_voters |> left_
```

Now we've got population data and demographic data by county. That means we can draw from both datasets in asking our questions. For example, we could see the counties with the highest 18+ Black population as a percentage of all population 18 and over and also the percentage of Democrats in that county.

We can get this by using mutate and arrange:

```
maryland_population_with_voters_and_demographics |>
  mutate(pct_black_18_plus = (pop_black/pop_18_over)*100, pct_dems = (DEM/TOTAL)*100) |>
  arrange(desc(pct_black_18_plus)) |>
  select(COUNTY, pct_black_18_plus, pct_dems)
```

```
# A tibble: 24 x 3
   COUNTY           pct_black_18_plus pct_dems
   <chr>                        <dbl>    <dbl>
 1 Prince George's               60.9     78.3
 2 Baltimore City                56.3     78.0
 3 Charles                       48.2     61.2
 4 Somerset                      39.0     41.8
 5 Baltimore County              28.8     55.4
 6 Dorchester                    26.2     44.4
 7 Wicomico                      25.6     42.3
 8 Howard                        18.7     52.4
 9 Montgomery                    18.1     61.0
10 Anne Arundel                  17.4     43.0
# ... with 14 more rows
```

If you know Maryland political demographics, this result isn't too surprising, but Somerset County - the state's 2nd smallest in terms of population - stands out for its Black population, which is a greater percentage than Baltimore County and Montgomery County.

Let's change that to look at Asian population:

```
maryland_population_with_voters_and_demographics |>
  mutate(pct_asian_18_plus = (pop_asian/pop_18_over)*100, pct_dems = (DEM/TOTAL)*100) |>
  arrange(desc(pct_asian_18_plus)) |>
  select(COUNTY, pct_asian_18_plus, pct_dems)
```

```
# A tibble: 24 x 3
   COUNTY           pct_asian_18_plus pct_dems
   <chr>                        <dbl>    <dbl>
 1 Howard                       19.4      52.4
 2 Montgomery                   16.0      61.0
 3 Baltimore County              6.34     55.4
 4 Frederick                     4.88     38.9
 5 Prince George's               4.68     78.3
 6 Anne Arundel                  4.52     43.0
 7 Baltimore City                4.17     78.0
 8 Charles                       3.55     61.2
 9 Harford                       3.15     35.4
10 St. Mary's                    3.13     35.7
# ... with 14 more rows
```

Here, Howard and Montgomery County stand out in terms of the percentage of Asian population 18 and over. The jurisdictions with the highest percentage of Democrats - Prince George's and Baltimore City - have small Asian populations.

Sometimes joins look like they should work but don't. Often this is due to the two columns you're joining on having different data types: joining a column to a column, for example. Let's walk through an example of that using some demographic data by zip code.

```
maryland_zcta <- read_csv('data/maryland_zcta.csv')
```

```
Rows: 468 Columns: 40
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (5): FIRST_CLAS, FIRST_MTFC, FIRST_FUNC, REPORT_2_P, REPORT_9_P
dbl (35): OBJECTID_1, ZCTA5CE10, FIRST_STAT, FIRST_GEOI, ZCTA5N, STATE, AREA...

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.
```

```
glimpse(maryland_zcta)
```

```
Rows: 468
Columns: 40
$ OBJECTID_1 <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ~
$ ZCTA5CE10  <dbl> 20601, 20602, 20603, 20606, 20607, 20608, 20609, 20611, 206~
$ FIRST_STAT <dbl> 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,~
$ FIRST_GEOI <dbl> 2420601, 2420602, 2420603, 2420606, 2420607, 2420608, 24206~
$ FIRST_CLAS <chr> "B5", "B5", "B5", "B5", "B5", "B5", "B5", "B5", "B5", "B5",~
$ FIRST_MTFC <chr> "G6350", "G6350", "G6350", "G6350", "G6350", "G6350", "G635~
$ FIRST_FUNC <chr> "S", "S", "S", "S", "S", "S", "S", "S", "S", "S", "S", "S",~
$ ZCTA5N     <dbl> 20601, 20602, 20603, 20606, 20607, 20608, 20609, 20611, 206~
$ STATE      <dbl> 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,~
$ AREALAND   <dbl> 115635266, 35830723, 44239637, 7501011, 54357590, 45583064,~
$ AREAWATR   <dbl> 387684, 352762, 219356, 1248760, 448221, 5330329, 6602735, ~
$ POP100     <dbl> 24156, 24955, 28967, 431, 9802, 919, 1120, 1078, 261, 11860~
$ HU100      <dbl> 8722, 9736, 10317, 230, 3504, 426, 554, 413, 142, 4424, 204~
$ NHW        <dbl> 9785, 8466, 9625, 377, 2165, 438, 1009, 798, 245, 4044, 352~
$ NHB        <dbl> 11146, 13054, 15025, 45, 6321, 453, 82, 215, 12, 6786, 32, ~
$ NHAI       <dbl> 155, 116, 98, 1, 33, 5, 2, 5, 0, 106, 2, 32, 3, 4, 38, 8, 1~
$ NHA        <dbl> 880, 731, 1446, 4, 560, 2, 1, 10, 0, 186, 3, 165, 5, 1, 402~
$ NHNH       <dbl> 11, 15, 24, 0, 3, 0, 1, 0, 0, 4, 1, 2, 0, 0, 4, 1, 0, 3, 1,~
$ NHO        <dbl> 48, 58, 65, 0, 6, 0, 0, 0, 0, 8, 0, 1, 0, 3, 5, 8, 0, 5, 10~
$ NHT        <dbl> 849, 999, 1091, 0, 234, 9, 15, 33, 1, 321, 13, 213, 14, 4, ~
$ HISP       <dbl> 1282, 1516, 1593, 4, 480, 12, 10, 17, 3, 405, 2, 244, 9, 7,~
$ PNHW       <dbl> 40.5, 33.9, 33.2, 87.5, 22.1, 47.7, 90.1, 74.0, 93.9, 34.1,~
$ PNHB       <dbl> 46.1, 52.3, 51.9, 10.4, 64.5, 49.3, 7.3, 19.9, 4.6, 57.2, 7~
$ PNHAI      <dbl> 0.6, 0.5, 0.3, 0.2, 0.3, 0.5, 0.2, 0.5, 0.0, 0.9, 0.5, 0.5,~
$ PNHA       <dbl> 3.6, 2.9, 5.0, 0.9, 5.7, 0.2, 0.1, 0.9, 0.0, 1.6, 0.7, 2.8,~
$ PNHNH      <dbl> 0.0, 0.1, 0.1, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0, 0.2, 0.0,~
$ PNHO       <dbl> 0.2, 0.2, 0.2, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0,~
$ PNHT       <dbl> 3.5, 4.0, 3.8, 0.0, 2.4, 1.0, 1.3, 3.1, 0.4, 2.7, 3.2, 3.6,~
$ PHISP      <dbl> 5.3, 6.1, 5.5, 0.9, 4.9, 1.3, 0.9, 1.6, 1.1, 3.4, 0.5, 4.2,~
$ POP65_     <dbl> 1922, 1964, 1400, 108, 847, 173, 271, 129, 54, 1372, 73, 55~
$ PCTPOP65_  <dbl> 8.0, 7.9, 4.8, 25.1, 8.6, 18.8, 24.2, 12.0, 20.7, 11.6, 18.~
$ MEDAGE     <dbl> 37.3, 32.6, 34.5, 49.1, 40.9, 46.6, 47.6, 44.3, 47.3, 40.8,~
$ VACNS      <dbl> 376, 769, 531, 15, 172, 39, 32, 22, 14, 249, 18, 158, 8, 18~
$ PVACNS     <dbl> 4.3, 7.9, 5.1, 6.5, 4.9, 9.2, 5.8, 5.3, 9.9, 5.6, 8.8, 7.2,~
$ PHOWN      <dbl> 71.1, 59.7, 73.8, 49.7, 83.1, 60.4, 44.8, 63.8, 38.3, 73.9,~
$ PWOMORT    <dbl> 11.2, 9.0, 4.7, 39.3, 10.3, 28.2, 38.7, 21.8, 43.9, 17.4, 2~
$ PRENT      <dbl> 19.9, 34.4, 22.6, 18.1, 7.4, 15.9, 27.0, 18.3, 31.7, 10.5, ~
$ PLT18SP    <dbl> 30.4, 43.6, 29.9, 31.2, 22.1, 14.1, 28.9, 24.5, 43.9, 26.7,~
```

```
$ REPORT_2_P <chr> "http://mdpgis.mdp.state.md.us/Census2010/PDF/00_SF1DP_2Pro~
$ REPORT_9_P <chr> "http://mdpgis.mdp.state.md.us/census2010/PDF/00_SF1_9PROFI~
```

You can see that ZCTA5N, the column representing the Zip Code Tabulation Area, is a numeric column. But should it be? Do we ever want to know the average zip code in Maryland? Zip codes and ZCTAs look like numbers but really are character columns. Let's change that so that we can be sure to join them correctly with other data where the zip codes are not numbers. We'll use `mutate`:

```
maryland_zcta <- maryland_zcta |> mutate(across(ZCTA5N, as.character))
```

What's happening here is that we're telling R to take all of the values in the ZCTA5N column and make them "as.character". If we wanted to change a column to numeric, we'd do "as.numeric". When you join two dataframes, the join columns *must* be the same datatype.

Joining datasets allows you to expand the range and sophistication of questions you're able to ask. It is one of the most powerful tools in a journalist's toolkit.

18 Cleaning Data Part IV: PDFs

The next circle of Hell on the Dante's Inferno of Data Journalism is the PDF. Governments everywhere love the PDF and publish all kinds of records in a PDF. The problem is a PDF isn't a data format – it's a middle finger, saying I've Got Your Accountability Right Here, Pal.

It's so ridiculous that there's a constellation of tools that do nothing more than try to harvest tables out of PDFs. There are online services like CometDocs where you can upload your PDF and point and click your way into an Excel file. There are mobile device apps that take a picture of a table and convert it into a spreadsheet. One of the best is a tool called Tabula. It was build by journalists for journalists.

There is a version of Tabula that will run inside of R – a library called Tabulizer – but the truth is I'm having the hardest time installing it on my machine, which leads me to believe that trying to install it across a classroom of various machines would be disastrous. The standalone version works just fine, and it provides a useful way for you to see what's actually going on.

Unfortunately, harvesting tables from PDFs with Tabula sometimes is an exercise in getting your hopes up, only to have them dashed. We'll start with an example. First, let's load the tidyverse and janitor.

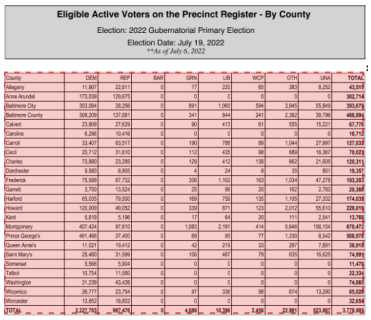
```
library(tidyverse)
library(janitor)
```

18.1 Easy does it

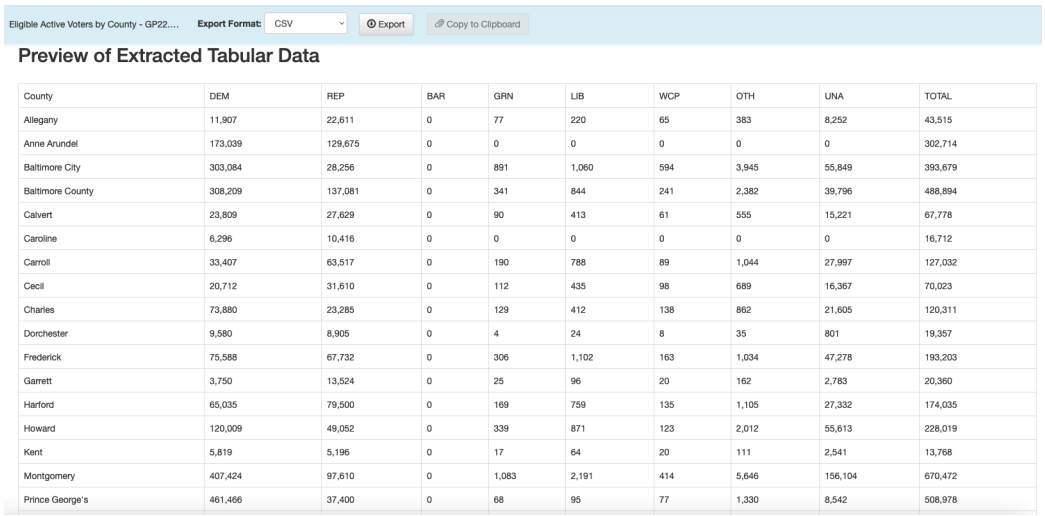
Tabula works best when tables in PDFs are clearly defined and have nicely-formatted information. Here's a perfect example: active voters by county in Maryland.

Download and install Tabula. Tabula works much the same way as Open Refine does – it works in the browser by spinning up a small webserver in your computer.

When Tabula opens, you click browse to find the PDF on your computer somewhere, and then click import. After it imports, click autodetect tables. You'll see red boxes appear around what Tabula believes are the tables. You'll see it does a pretty good job at this.



Now you can hit the green "Preview & Export Extracted Data" button on the top right. You should see something very like this:



County	DEM	REP	BAR	GRN	LIB	WCP	OTH	UNA	TOTAL
Allegany	11,907	22,611	0	77	220	65	383	8,252	43,515
Anne Arundel	173,039	129,675	0	0	0	0	0	0	302,714
Baltimore City	303,084	28,256	0	891	1,060	594	3,945	55,849	393,679
Baltimore County	308,209	137,081	0	341	844	241	2,382	39,796	488,894
Calvert	23,809	27,629	0	90	413	61	555	15,221	67,778
Caroline	6,296	10,416	0	0	0	0	0	0	16,712
Carroll	33,407	63,517	0	190	788	89	1,044	27,997	127,032
Cecil	20,712	31,610	0	112	435	98	689	16,367	70,023
Charles	73,880	23,285	0	129	412	138	862	21,605	120,311
Dorchester	9,580	8,905	0	4	24	8	35	801	19,357
Frederick	75,588	67,732	0	306	1,102	163	1,034	47,278	193,203
Garrett	3,750	13,524	0	25	96	20	162	2,783	20,360
Harford	65,035	79,500	0	169	759	135	1,105	27,332	174,035
Howard	120,009	49,052	0	339	871	123	2,012	55,613	228,019
Kent	5,819	5,196	0	17	64	20	111	2,541	13,768
Montgomery	407,424	97,610	0	1,083	2,191	414	5,646	156,104	670,472
Prince George's	461,466	37,400	0	68	95	77	1,330	8,542	508,978

You can now export that extracted table to a CSV file using the "Export" button. And then we can read it into R:

```
voters_by_county <- read_csv("data/tabula-Eligible Active Voters by County - GG22.csv")
```

```
Rows: 25 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): County
dbl (1): BAR
num (8): DEM, REP, GRN, LIB, WCP, OTH, UNA, TOTAL

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.
```

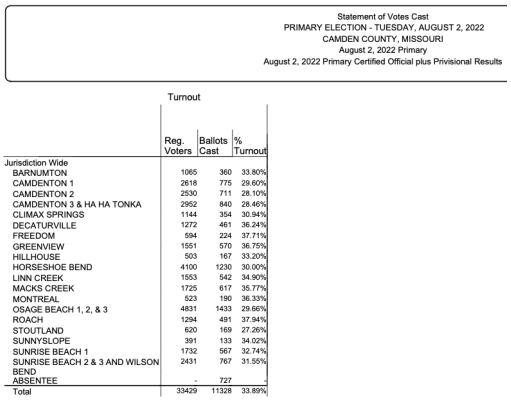
```
voters_by_county
```

```
# A tibble: 25 x 10
   County              DEM    REP   BAR   GRN   LIB   WCP   OTH    UNA  TOTAL
   <chr>             <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
 1 Allegany          11793  22732     0    76   223    73   382   8337  43616
 2 Anne Arundel     173922 129893     0   632  2257   434  2951  96403 406492
 3 Baltimore City   303620  28211     0   908  1080   666  3984  56665 395134
 4 Baltimore County 309297 137378     0   898  2493   687  5921 106789 563463
 5 Calvert           23779  27912     0    94   410    69   548  15169  67981
 6 Caroline           6250  10539     0    34   108    40   160   4454  21585
 7 Carroll           33572  63771     0   191   797   102  1033  28139 127605
 8 Cecil             20666  31961     0   111   447   104   678  16360  70327
 9 Charles           74373  23334     0   129   425   143   864  21819 121087
10 Dorchester         9608   8965     0    22   110    33   191   3745  22674
# ... with 15 more rows
```

Boom - we're good to go.

18.2 When it looks good, but needs a little fixing

Here's a slightly more involved PDF, 2022 Missouri primary election precinct results from Camden County.



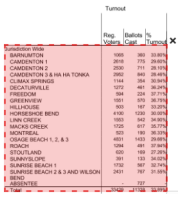
Statement of Votes Cast
PRIMARY ELECTION - TUESDAY, AUGUST 2, 2022
CAMDEN COUNTY, MISSOURI
August 2, 2022 Primary
August 2, 2022 Primary Certified Official plus Privisional Results

Turnout

	Reg. Voters	Ballots Cast	% Turnout
Jurisdiction Wide			
BARNUMTON	1065	360	33.80%
CAMDENTON 1	2618	775	29.60%
CAMDENTON 2	2530	711	28.10%
CAMDENTON 3 & HA HA TONKA	2952	840	28.46%
CLIMAX SPRINGS	1144	354	30.94%
DECATURVILLE	1272	461	36.24%
FREEDOM	594	224	37.71%
GREENVIEW	1551	570	36.75%
HILLHOUSE	503	167	33.20%
HORSESHOE BEND	4100	1230	30.00%
LINN CREEK	1553	542	34.90%
MACKS CREEK	1725	617	35.77%
MONTREAL	523	190	36.33%
OSAGE BEACH 1, 2, & 3	4831	1433	29.66%
ROACH	1294	491	37.94%
STOUTLAND	620	169	27.26%
SUNNYSLOPE	391	133	34.02%
SUNRISE BEACH 1	1732	567	32.74%
SUNRISE BEACH 2 & 3 AND WILSON BEND	2431	767	31.55%
ABSENTEE	-	727	-
Total	33429	11328	33.89%

Looks like a spreadsheet, right? Save that PDF file to your computer in a place where you'll remember it (like a Downloads folder).

Now let's repeat the steps we did to import the PDF into Tabula and autodetect the tables. It should look like this:



This is pretty good, but we want to capture the second line of the headers - `Voters`, `Cast` and `Turnout` - and for our purposes we only want the first page of the PDF for now. So hit "Clear All Selections" button at the top and let's draw a box around what we want. Using your cursor, click and drag a box across the first page so it looks like this:

Now you can hit the green "Preview & Export Extracted Data" button on the top right. Using the "Stream" method, you should see something very like this:

Preview of Extracted Tabular Data

	Voters	Cast	Turnout
Jurisdiction Wide			
BARNUMTON	1065	360	33.80%
CAMDENTON 1	2618	775	29.60%
CAMDENTON 2	2530	711	28.10%
CAMDENTON 3 & HA HA TONKA	2952	840	28.46%
CLIMAX SPRINGS	1144	354	30.94%
DECATURVILLE	1272	461	36.24%
FREEDOM	594	224	37.71%
GREENVIEW	1551	570	36.75%
HILLHOUSE	503	167	33.20%
HORSESHOE BEND	4100	1230	30.00%
LINN CREEK	1553	542	34.90%
MACKS CREEK	1725	617	35.77%
MONTREAL	523	190	36.33%
OSAGE BEACH 1, 2, & 3	4831	1433	29.66%
ROACH	1294	491	37.94%

You can now export that extracted table to a CSV file using the "Export" button. And then we can read it into R and clean up the column names and some other things:

```
camden_2022 <- read_csv("data/tabula-camden_county_election_results.csv") |> clean_names()
```

```
New names:
Rows: 23 Columns: 4
-- Column specification
-------------------------------------------------------- Delimiter: "," chr
(3): ...1, Voters, Turnout dbl (1): Cast
```

```
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.
* `` -> `...1`
```

```
camden_2022
```

```
# A tibble: 23 x 4
   x1                        voters  cast turnout
   <chr>                     <chr>  <dbl> <chr>
 1 Jurisdiction Wide         <NA>      NA <NA>
 2 BARNUMTON                 1065     360 33.80%
 3 CAMDENTON 1               2618     775 29.60%
 4 CAMDENTON 2               2530     711 28.10%
 5 CAMDENTON 3 & HA HA TONKA 2952     840 28.46%
 6 CLIMAX SPRINGS            1144     354 30.94%
 7 DECATURVILLE              1272     461 36.24%
 8 FREEDOM                   594      224 37.71%
 9 GREENVIEW                 1551     570 36.75%
10 HILLHOUSE                 503      167 33.20%
# ... with 13 more rows
```

18.3 Cleaning up the data in R

The good news is that we have data we don't have to retype. The bad news is, it's hardly in importable shape. We have a few things to fix. Some of the datatypes are wrong, we've got a column to rename and two rows with NAs, plus a precinct name that wraps over to the next line. Let's start by re-importing it and calling `mutate` to fix the column & datatype issues, plus that precinct name:

```
camden_2022 <- read_csv("data/tabula-camden_county_election_results.csv") |> clean_names()
```

```
New names:
Rows: 23 Columns: 4
-- Column specification
-------------------------------------------------------- Delimiter: "," chr
(3): ...1, Voters, Turnout dbl (1): Cast
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.
* `` -> `...1`
```

```
camden_2022 <- camden_2022 |>
  rename(precinct = x1) |>
  mutate(voters = as.numeric(voters),
         turnout = as.numeric(str_replace(turnout, '%',''))
         ) |>
  mutate(precinct = case_when(
    precinct == 'SUNRISE BEACH 2 & 3 AND WILSON' ~ 'SUNRISE BEACH 2 & 3 AND WILSON BEND',
    TRUE ~ precinct
  )
)
```

```
Warning: There were 2 warnings in `mutate()`.
The first warning was:
i In argument: `voters = as.numeric(voters)`.
Caused by warning:
! NAs introduced by coercion
i Run `dplyr::last_dplyr_warnings()` to see the 1 remaining warning.
```

```
camden_2022
```

```
# A tibble: 23 x 4
   precinct                  voters  cast turnout
   <chr>                      <dbl> <dbl>   <dbl>
 1 Jurisdiction Wide             NA    NA    NA
 2 BARNUMTON                   1065   360    33.8
 3 CAMDENTON 1                 2618   775    29.6
 4 CAMDENTON 2                 2530   711    28.1
 5 CAMDENTON 3 & HA HA TONKA   2952   840    28.5
 6 CLIMAX SPRINGS              1144   354    30.9
 7 DECATURVILLE                1272   461    36.2
 8 FREEDOM                      594   224    37.7
 9 GREENVIEW                   1551   570    36.8
10 HILLHOUSE                    503   167    33.2
# ... with 13 more rows
```

Ok, now we have numbers. Next we'll get rid of the rows where `cast` is NA (why not `voters`?). To do that, we'll use the inverse of the `is.na` function by placing an exclamation point before it (you can read that filter as "where cast is NOT NA").

```
camden_2022 <- camden_2022 |> filter(!is.na(cast))

camden_2022
```

```
# A tibble: 21 x 4
   precinct                  voters  cast turnout
   <chr>                      <dbl> <dbl>   <dbl>
 1 BARNUMTON                   1065   360    33.8
 2 CAMDENTON 1                 2618   775    29.6
 3 CAMDENTON 2                 2530   711    28.1
 4 CAMDENTON 3 & HA HA TONKA   2952   840    28.5
 5 CLIMAX SPRINGS              1144   354    30.9
 6 DECATURVILLE                1272   461    36.2
 7 FREEDOM                      594   224    37.7
 8 GREENVIEW                   1551   570    36.8
 9 HILLHOUSE                    503   167    33.2
10 HORSESHOE BEND              4100  1230    30
# ... with 11 more rows
```

All things considered, that was pretty easy. Many - most? - electronic PDFs aren't so easy to parse. Sometimes you'll need to open the exported CSV file and clean things up before importing into R. Other times you'll be able to do that cleaning in R itself.

Here's the sad truth: THIS IS PRETTY GOOD. It sure beats typing it out. And since many government processes don't change all that much, you can save the code to process subsequent versions of PDFs.

19 Scraping data with Rvest

Sometimes, governments put data online on a page or in a searchable database. And when you ask them for a copy of the data underneath the website, they say no.

Why? Because they have a website. That's it. That's their reason. They say they don't have to give you the data because they've already given you the data, never mind that they haven't given to you in a form you can actually load into R with ease.

Lucky for us, there's a way for us to write code to get data even when an agency hasn't made it easy: webscraping.

One of the most powerful tools you can learn as a data journalist is how to scrape data from the web. Scraping is the process of programming a computer to act like a human that opens a web browser, goes to a website, ingests the HTML from that website into R and turns it into data.

The degree of difficulty here goes from "Easy" to "So Hard You Want To Throw Your Laptop Out A Window." And the curve between the two can be steep. You can learn how to scrape "Easy" in a day. The hard ones take a little more time, but it's often well worth the effort because it lets you get stories you couldn't get without it.

In this chapter, we'll show you an easy one. And in the next chapter, we'll so you a moderately harder one.

Let's start easy.

We're going to use a library called `rvest`, which you can install it the same way we've done all installs: go to the console and `install.packages("rvest")`.

Like so many R package names, rvest is a bad pun. You're supposed to read it to sound like "harvest", as in "harvesting" information from a website the same way you'd harvest crops in a field.

We'll load these packages first:

```
library(rvest)
library(tidyverse)
library(janitor)
```

For this example, we're going to work on loading a simple table of data from the Maryland State Board of Elections. This is a table of unofficial county-level election results for the Attorney General's race from the November general election.

Let's suppose we can't find a table like that for download, but we do see a version on the SBOE website at this URL: https://elections.maryland.gov/elections/2022/general_results/gen_detail_results_2022_3_1.html.

(2074 of 2074 election day precincts reported)

Jurisdiction	Michael Anthony Peroutka Republican	Anthony G. Brown Democratic	Other Write-ins
Allegany	14,382	6,994	16
Anne Arundel	91,718	121,096	433
Baltimore City	14,594	128,282	288
Baltimore County	100,480	170,858	467
Calvert	20,275	16,647	41
Caroline	7,128	3,475	13
Carroll	44,195	26,928	112
Cecil	20,750	11,844	35
Charles	16,042	37,945	55
Dorchester	6,615	4,784	16
Frederick	48,214	56,647	112

We could get this table into R with the following manual steps: highlighting the text, copying it into Excel, saving it as a csv, and reading it into R. Or, we could write a few lines of webscraping code to have R do that for us!

In this simple example, it's probably faster to do it manually than have R do it for us. But during the time when ballots are being counted, this table is likely to change, and we don't want to keep doing manual repetitive tasks.

Why would we ever write code to grab a single table? There's several reasons:

1. Our methods are transparent. If a colleague wants to run our code from scratch to factcheck our work, they don't need to repeat the manual steps, which are harder to document than writing code.
2. Let's suppose we wanted to grab the same table every day, to monitor for changes. Writing a script once, and pressing a single button every day is going to be much more efficient than doing this manually every day.
3. If we're doing it manually, we're more likely to make a mistake, like maybe failing to copy every row from the whole table.
4. It's good practice to prepare us to do more complex scraping jobs. As we'll see in the next chapter, if we ever want to grab the same table from hundreds of pages, writing code is much faster and easier than going to a hundred different pages ourselves and downloading data.

So, to scrape, the first thing we need to do is start with the URL. Let's store it as an object called ag_url.

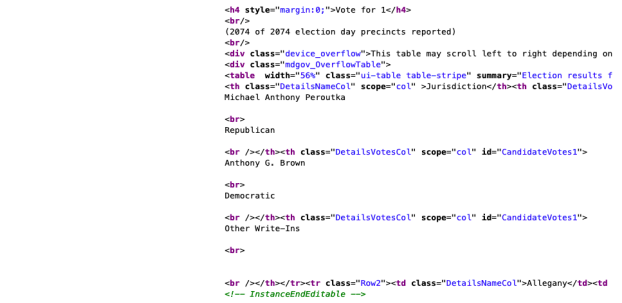
```
ag_url <- "https://elections.maryland.gov/elections/2022/general_results/gen_detail_results_
```

When we go to the web page, we can see a nicely-designed page that contains our information.

But what we really care about, for our purposes, is the html code that creates that page.

In our web browser, if we right-click anywhere on the page and select "view source" from the popup menu, we can see the source code. Or you can just copy this into Google Chrome: view-source:https://elections.maryland.gov/elections/2022/general_results/gen_detail_results_2022_3_1.html.

Here's a picture of what some of the source code looks like.



We'll use those HTML tags – things like `<table>` and `<tr>` – to grab the info we need.

Okay, step 1.

Let's write a bit of code to tell R to go to the URL for the page and ingest all of that HTML code. In the code below, we're starting with our URL and using the read_html() function from rvest to ingest all of the page html, storing it as an object called results.

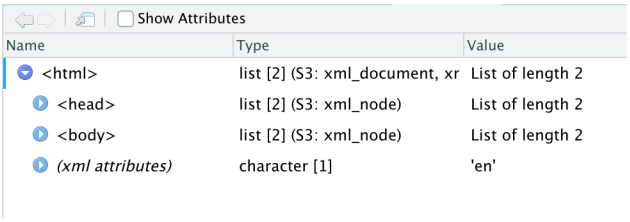
```
# read in the html
results <- ag_url |>
  read_html()

# display the html below
results
```

```
{html_document}
<html lang="en" manifest="manifest.appcache">
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
[2] <body>\r\n    <div class="container">  \r\n        <div class="skipNav">\ ...
```

If you're running this code in R Studio, in our environment window at right, you'll see results as a "list of 2".

This is not a dataframe, it's a different type of data structure a "nested list."

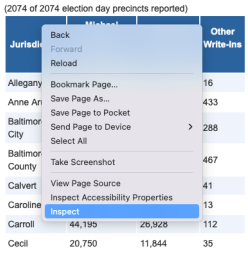
If we click on the name "results" in our environment window, we can see that it's pulled in the html and shown us the general page structure. Nested within the `<html>` tag is the `<head>` and `<body>`, the two fundamental sections of most web pages. We're going to pull information out of the `<body>` tag in a bit.



Now, our task is to just pull out the section of the html that contains the information we need.

But which part do we need from that mess of html code? To figure that out, we can go back to the page in a web browser like chrome, and use built in developer tools to "inspect" the html code underlying the page.

On the page, find the data we want to grab – "Vote for 1" - and right click on the word "Jurisdiction" in the column header of the table. That will bring up a dropdown menu. Select "Inspect", which will pop up a window called the "element inspector" that shows us where different elements on the page are located, what html tags created those elements, and other info.



The entire table that we want of results is actually contained inside an html `<table>`. It has a `<tbody>` that contains one row `<tr>` per county.

Because it's inside of a table, and not some other kind of element (like a `<div>`), rvest has a special function for easily extracting and converting html tables, called html_table(). This function extracts all the html tables on the page, but this page only has one so we're good.

```
# read in the html and extract all the tables
results <- ag_url |>
  read_html() |>
  html_table()

# show the dataframe
```

```
results
```

```
[[1]]
# A tibble: 25 x 4
   Jurisdiction     Michael Anthony Peroutka\r\n\r\n\r\nRepubl~1 Antho~2 Other~3
   <chr>            <chr>                                       <chr>   <chr>
 1 Allegany         14,382                                      6,994   16
 2 Anne Arundel     91,718                                      121,096 433
 3 Baltimore City   14,594                                      128,282 288
 4 Baltimore County 100,480                                     170,858 467
 5 Calvert          20,275                                      16,647  41
 6 Caroline         7,128                                       3,475   13
 7 Carroll          44,195                                      26,928  112
 8 Cecil            20,750                                      11,844  35
 9 Charles          16,042                                      37,945  55
10 Dorchester       6,615                                       4,784   16
# ... with 15 more rows, and abbreviated variable names
#   1: `Michael Anthony Peroutka\r\n\r\n\r\nRepublican`,
#   2: `Anthony G. Brown\r\n\r\n\r\nDemocratic`, 3: `Other Write-Ins`
```

In the environment window at right, look at results Note that it's now a "list of 1".

This gets a little complicated, but what you're seeing here is a nested list that contains one data frame – also called tibbles – one for each table that exists on the web page we scraped.

So, all we need to do now is to store that single dataframe as an object. We can do that with this code, which says "keep only the first dataframe from our nested list."

```
# Read in all html from table, store all tables on page as nested list of dataframes.
results <- ag_url |>
  read_html() |>
  html_table()

# Just keep the first dataframe in our list

results <- results[[1]]

# show the dataframe

results
```

```
# A tibble: 25 x 4
   Jurisdiction     Michael Anthony Peroutka\r\n\r\n\r\nRepubl~1 Antho~2 Other~3
   <chr>            <chr>                                       <chr>   <chr>
 1 Allegany         14,382                                      6,994   16
 2 Anne Arundel     91,718                                      121,096 433
 3 Baltimore City   14,594                                      128,282 288
 4 Baltimore County 100,480                                     170,858 467
 5 Calvert          20,275                                      16,647  41
 6 Caroline         7,128                                       3,475   13
 7 Carroll          44,195                                      26,928  112
 8 Cecil            20,750                                      11,844  35
 9 Charles          16,042                                      37,945  55
10 Dorchester       6,615                                       4,784   16
# ... with 15 more rows, and abbreviated variable names
#   1: `Michael Anthony Peroutka\r\n\r\n\r\nRepublican`,
#   2: `Anthony G. Brown\r\n\r\n\r\nDemocratic`, 3: `Other Write-Ins`
```

We now have a proper dataframe, albeit with some lengthy column headers.

From here, we can do a little cleaning. First we'll use clean_names() to lower the column names. Then use rename() to replace the candidate column names with simpler versions. We can just use the column positions instead of writing out the full names, which is nice.

Then let's use slice() to remove the last row – row number 25 – which contains totals and percentages that we don't need. Finally, we'll make sure the vote tallies are numbers using mutate and `gsub()`, which we use to replace all the commas with nothing.

```
# Read in all html from table, get the HTML table.
results <- ag_url |>
  read_html() |>
  html_table()

# Standardize column headers, remove last row

results <- results[[1]] |>
  clean_names() |>
  rename(peroutka = 2, brown = 3, write_ins = 4) |>
  slice(-25) |>
  mutate(peroutka = as.numeric(gsub(",","", peroutka))) |>
  mutate(brown = as.numeric(gsub(",","", brown))) |>
  mutate(write_ins = as.numeric(gsub(",","", write_ins)))

# show the dataframe
results
```

```
# A tibble: 24 x 4
   jurisdiction     peroutka  brown write_ins
   <chr>               <dbl>  <dbl>     <dbl>
 1 Allegany            14382   6994        16
 2 Anne Arundel        91718 121096       433
 3 Baltimore City      14594 128282       288
 4 Baltimore County   100480 170858       467
 5 Calvert             20275  16647        41
 6 Caroline             7128   3475        13
 7 Carroll             44195  26928       112
 8 Cecil               20750  11844        35
 9 Charles             16042  37945        55
10 Dorchester           6615   4784        16
# ... with 14 more rows
```

And there we go. We now have a nice tidy dataframe of Maryland attorney general election results that we could ask some questions of.

20 Intro to APIs: The Census

There is truly an astonishing amount of data collected by the US Census Bureau. First, there's the Census that most people know – the every 10 year census. That's the one mandated by the Constitution where the government attempts to count every person in the US. It's a mind-boggling feat to even try, and billions get spent on it. That data is used first for determining how many representatives each state gets in Congress. From there, the Census gets used to divide up billions of dollars of federal spending.

To answer the questions the government needs to do that, a ton of data gets collected. That, unfortunately, means the Census is exceedingly complicated to work with. The good news is, the Census has an API – an application programming interface. What that means is we can get data directly through the Census Bureau via calls over the internet.

Let's demonstrate.

We're going to use a library called `tidycensus` which makes calls to the Census API in a very tidy way, and gives you back tidy data. That means we don't have to go through the process of importing the data from a file. I can't tell you how amazing this is, speaking from experience. The documentation for this library is here. Another R library for working with Census APIs (there is more than one) is this one from Hannah Recht, a journalist with Kaiser Health News.

First we need to install `tidycensus` using the console: `install.packages("tidycensus", dependencies = TRUE)`. You also should install the `sf` and `rgdal` packages.

```
library(tidyverse)
library(tidycensus)
```

To use the API, you need an API key. To get that, you need to apply for an API key with the Census Bureau. It takes a few minutes and you need to activate your key via email. Once you have your key, you need to set that for this session. Just FYI: Your key is your key. Do not share it around.

```
census_api_key("YOUR KEY HERE", install=TRUE)
```

The two main functions in tidycensus are `get_decennial`, which retrieves data from the 2000 and 2010 Censuses (and soon the 2020 Census), and `get_acs`, which pulls data from the

American Community Survey, a between-Censuses annual survey that provides estimates, not hard counts, but asks more detailed questions. If you're new to Census data, there's a very good set of slides from Kyle Walker, the creator of tidycensus, and he's working on a book that you can read for free online.

It's important to keep in mind that Census data represents people - you, your neighbors and total strangers. It also requires some level of definitions, especially about race & ethnicity, that may or may not match how you define yourself or how others define themselves.

So to give you some idea of how complicated the data is, let's pull up just one file from the decennial Census. We'll use Summary File 1, or SF1. That has the major population and housing stuff.

```
sf1 <- load_variables(2010, "sf1", cache = TRUE)

sf1
```

```
# A tibble: 8,959 x 3
   name    label                                    concept
   <chr>   <chr>                                    <chr>
 1 H001001 Total                                    HOUSING UNITS
 2 H002001 Total                                    URBAN AND RURAL
 3 H002002 Total!!Urban                             URBAN AND RURAL
 4 H002003 Total!!Urban!!Inside urbanized areas     URBAN AND RURAL
 5 H002004 Total!!Urban!!Inside urban clusters      URBAN AND RURAL
 6 H002005 Total!!Rural                             URBAN AND RURAL
 7 H002006 Total!!Not defined for this file         URBAN AND RURAL
 8 H003001 Total                                    OCCUPANCY STATUS
 9 H003002 Total!!Occupied                          OCCUPANCY STATUS
10 H003003 Total!!Vacant                            OCCUPANCY STATUS
# ... with 8,949 more rows
```

Note: There are thousands of variables in SF1. That's not a typo. Open it in your environment by double clicking. As you scroll down, you'll get an idea of what you've got to choose from.

If you think that's crazy, try the SF3 file from 2000.

```
sf3 <- load_variables(2000, "sf3", cache = TRUE)

sf3
```

```
# A tibble: 16,520 x 3
   name    label                                          concept
```

```
   <chr>   <chr>                                              <chr>
 1 H001001 Total                                              HOUSING UNITS [1]
 2 H002001 Total                                              UNWEIGHTED SAMPLE HOUSIN~
 3 H002002 Total!!Occupied                                    UNWEIGHTED SAMPLE HOUSIN~
 4 H002003 Total!!Vacant                                      UNWEIGHTED SAMPLE HOUSIN~
 5 H003001 Total                                              100-PERCENT COUNT OF HOU~
 6 H004001 Percent of occupied housing units in sample        PERCENT OF HOUSING UNITS~
 7 H004002 Percent of vacant housing units in sample          PERCENT OF HOUSING UNITS~
 8 H005001 Total                                              URBAN AND RURAL [7]
 9 H005002 Total!!Urban                                       URBAN AND RURAL [7]
10 H005003 Total!!Urban!!Inside urbanized areas               URBAN AND RURAL [7]
# ... with 16,510 more rows
```

Yes. That's more than 16,000 variables to choose from. I told you. Astonishing.

So let's try to answer a question using the Census. What is the fastest growing state since 2000?

To answer this, we need to pull the total population by state in each of the decennial census. Here's 2000.

```
p00 <- get_decennial(geography = "state", variables = "P001001", year = 2000)
```

Now 2010.

```
p10 <- get_decennial(geography = "state", variables = "P001001", year = 2010)
```

Let's take a peek at 2010.

```
p10
```

As you can see, we have a GEOID, NAME, then variable and value. Variable and value are going to be the same. Because those are named the same thing, to merge them together, we need to rename them.

```
p10 |> select(GEOID, NAME, value) |> rename(Population2010=value) -> p2010

p00 |> select(GEOID, NAME, value) |> rename(Population2000=value) -> p2000
```

Now we join the data together.

```
alldata <- p2000 |> inner_join(p2010)
```

And now we calculate the percent change.

```
alldata |> mutate(change = ((Population2010-Population2000)/Population2000)*100) |> arrange(c
```

And just like that: Nevada.

You may be asking: hey, wasn't there a 2020 Census? Where's that data? The answer is that it's coming, slowly - the Census Bureau has a schedule of releases.

20.1 The ACS

In 2010, the Census Bureau replaced SF3 with the American Community Survey. The Good News is that the data would be updated on a rolling basis. The bad news is that it's more complicated because it's more like survey data with a large sample. That means there's margins of error and confidence intervals to worry about. By default, using `get_acs` fetches data from the 5-year estimates (currently 2018-2022), but you can specify 1-year estimates for jurisdictions with at least 65,000 people (many counties and cities).

Here's an example using the 5-year ACS estimates:

What is Maryland's richest county?

We can measure this by median household income. That variable is `B19013_001`, so we can get that data like this (I'm narrowing it to the top 20 for simplicity):

```
md <- get_acs(geography = "county",
              variables = c(medincome = "B19013_001"),
              state = "MD",
              year = 2022)
```

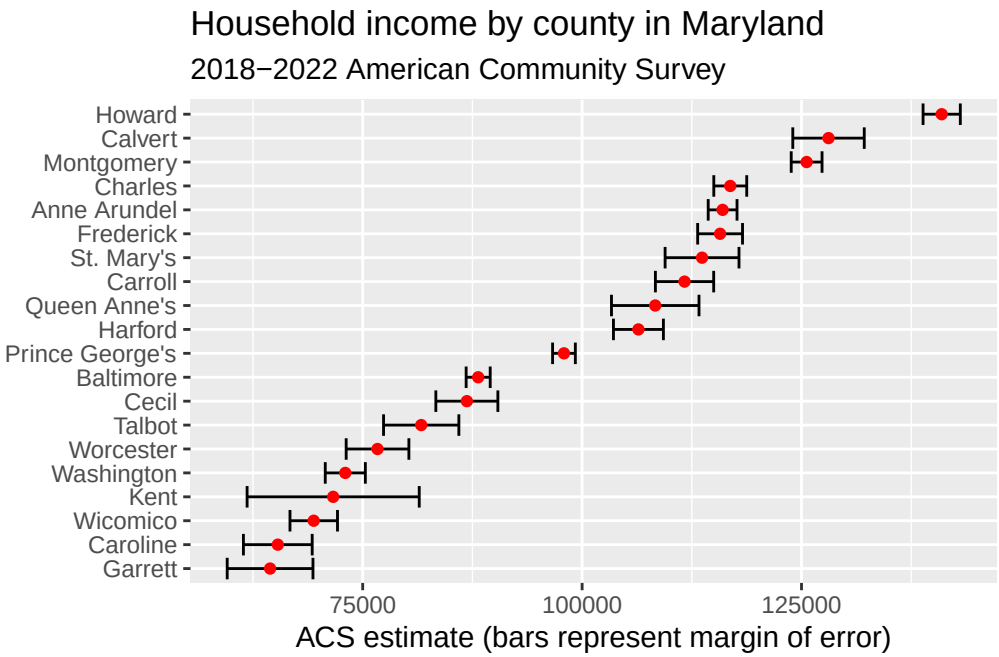
```
Getting data from the 2018-2022 5-year ACS
```

```
md <- md |> arrange(desc(estimate)) |> top_n(20, estimate)

md
```

```
# A tibble: 20 x 5
   GEOID NAME                                variable  estimate   moe
   <chr> <chr>                               <chr>        <dbl> <dbl>
 1 24027 Howard County, Maryland             medincome   140971  2122
 2 24009 Calvert County, Maryland            medincome   128078  4071
 3 24031 Montgomery County, Maryland         medincome   125583  1757
 4 24017 Charles County, Maryland            medincome   116882  1877
 5 24003 Anne Arundel County, Maryland       medincome   116009  1642
 6 24021 Frederick County, Maryland          medincome   115724  2555
 7 24037 St. Mary's County, Maryland         medincome   113668  4210
 8 24013 Carroll County, Maryland            medincome   111672  3322
 9 24035 Queen Anne's County, Maryland       medincome   108332  4986
10 24025 Harford County, Maryland            medincome   106417  2845
11 24033 Prince George's County, Maryland    medincome    97935  1296
12 24005 Baltimore County, Maryland          medincome    88157  1370
13 24015 Cecil County, Maryland              medincome    86869  3534
14 24041 Talbot County, Maryland             medincome    81667  4291
15 24047 Worcester County, Maryland          medincome    76689  3573
16 24043 Washington County, Maryland         medincome    73017  2283
17 24029 Kent County, Maryland               medincome    71635  9808
18 24045 Wicomico County, Maryland           medincome    69421  2707
19 24011 Caroline County, Maryland           medincome    65326  3919
20 24023 Garrett County, Maryland            medincome    64447  4886
```

Howard, Calvert, Montgomery, Anne Arundel, Charles. What do they all have in common? Lots of suburban flight from DC and Baltimore. But do the margins of error let us say one county is richer than the other. We can find this out visually using error bars. Don't worry much about the code here – we'll cover that soon enough.

```
md |>
  mutate(NAME = gsub(" County, Maryland", "", NAME)) |>
  ggplot(aes(x = estimate, y = reorder(NAME, estimate))) +
  geom_errorbarh(aes(xmin = estimate - moe, xmax = estimate + moe)) +
  geom_point(color = "red") +
  labs(title = "Household income by county in Maryland",
       subtitle = "2018-2022 American Community Survey",
       y = "",
       x = "ACS estimate (bars represent margin of error)")
```



As you can see, some of the error bars are quite wide. Some are narrow. But if the bars overlap, it means the difference between the two counties is within the margin of error, and the differences aren't statistically significant. So is the difference between Calvert and Montgomery significant? Nope. Is the difference between Howard and everyone else significant? Yes it is.

Let's ask another question of the ACS – did any counties lose income from the time of the global financial crisis to the current 5-year window?

Let's re-label our first household income data.

```
md22 <- get_acs(geography = "county",
                variables = c(medincome = "B19013_001"),
                state = "MD",
                year = 2022)
```

```
Getting data from the 2018-2022 5-year ACS
```

And now we grab the 2010 median household income.

```
md10 <- get_acs(geography = "county",
                variables = c(medincome = "B19013_001"),
                state = "MD",
                year = 2010)
```

```
Getting data from the 2006-2010 5-year ACS
```

What I'm going to do next is a lot, but each step is simple. I'm going to join the data together, so each county has one line of data. Then I'm going to rename some fields that repeat. Then I'm going to calculate the minimium and maximum value of the estimate using the margin of error. That'll help me later. After that, I'm going to calculate a perent change and sort it by that change.

```
md10 |>
  inner_join(md22, by=c("GEOID", "NAME")) |>
  rename(estimate2010=estimate.x, estimate2022=estimate.y) |>
  mutate(min2010 = estimate2010-moe.x, max2010 = estimate2010+moe.x, min2020 = estimate2022-m
  select(-variable.x, -variable.y, -moe.x, -moe.y) |>
  mutate(change = ((estimate2022-estimate2010)/estimate2010)*100) |>
  arrange(change)
```

```
# A tibble: 24 x 9
   GEOID NAME             estim~1 estim~2 min2010 max2010 min2020 max2020 change
   <chr> <chr>              <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>  <dbl>
 1 24011 Caroline County~   58799   65326   56740   60858   61407   69245   11.1
 2 24039 Somerset County~   42443   52149   39092   45794   45533   58765   22.9
 3 24019 Dorchester Coun~   45151   57490   43470   46832   54445   60535   27.3
 4 24041 Talbot County, ~   63017   81667   60081   65953   77376   85958   29.6
 5 24017 Charles County,~   88825  116882   87268   90382  115005  118759   31.6
 6 24035 Queen Anne's Co~   81096  108332   78068   84124  103346  113318   33.6
 7 24015 Cecil County, M~   64886   86869   63014   66758   83335   90403   33.9
 8 24031 Montgomery Coun~   93373  125583   92535   94211  123826  127340   34.5
 9 24027 Howard County, ~  103273  140971  101654  104892  138849  143093   36.5
10 24045 Wicomico County~   50752   69421   49313   52191   66714   72128   36.8
# ... with 14 more rows, and abbreviated variable names 1: estimate2010,
#   2: estimate2022
```

So according to this, Caroline County had the smallest change between 2010 and 2022, while all other jurisdictions saw percentage increases of at least 20 percent.

But did they?

Look at the min and max values for both. Is the change statistically significant?

The ACS data has lots of variables, just like the decennial Census does. To browse them, you can do this:

```
v22 <- load_variables(2022, "acs5", cache=TRUE)
```

And then view v20 to see what kinds of variables are available via the API.

20.2 "Wide" Results

Although one of the chief strengths of tidycensus is that it offers a, well, tidy display of Census data, it also has the ability to view multiple variables spread across columns. This can be useful for creating percentages and comparing multiple variables.

20.3 Sorting Results

You'll notice that we've used `arrange` to sort the results of tidycensus functions, although that's done after we create a new variable to hold the data. There's another way to use `arrange` that you should know about, one that you can use for exploratory analysis. An example using median household income from 2022:

```
md22 <- get_acs(geography = "county",
              variables = c(medincome = "B19013_001"),
              state = "MD",
              year = 2022)
```

```
Getting data from the 2018-2022 5-year ACS
```

```
arrange(md22, desc(estimate))
```

```
# A tibble: 24 x 5
   GEOID NAME                           variable  estimate   moe
   <chr> <chr>                          <chr>        <dbl> <dbl>
 1 24027 Howard County, Maryland        medincome   140971  2122
 2 24009 Calvert County, Maryland       medincome   128078  4071
 3 24031 Montgomery County, Maryland    medincome   125583  1757
 4 24017 Charles County, Maryland       medincome   116882  1877
 5 24003 Anne Arundel County, Maryland  medincome   116009  1642
 6 24021 Frederick County, Maryland     medincome   115724  2555
 7 24037 St. Mary's County, Maryland    medincome   113668  4210
 8 24013 Carroll County, Maryland       medincome   111672  3322
 9 24035 Queen Anne's County, Maryland  medincome   108332  4986
```

```
10 24025 Harford County, Maryland      medincome   106417  2845
# ... with 14 more rows
```

In this case we don't save the sorted results to a variable, we can just see the output in the console.

21 Visualizing your data for reporting

Visualizing data is becoming a much greater part of journalism. Large news organizations are creating graphics desks that create complex visuals with data to inform the public about important events.

To do it well is a course on its own. And not every story needs a feat of programming and art. Sometimes, you can help yourself and your story by just creating a quick chart, which helps you see patterns in the data that wouldn't otherwise surface.

Good news: one of the best libraries for visualizing data is in the tidyverse and it's pretty simple to make simple charts quickly with just a little bit of code. It's called ggplot2.

Let's revisit some data we've used in the past and turn it into charts. First, let's load libraries. When we load the tidyverse, we get ggplot2.

```
library(tidyverse)
```

The dataset we'll use is voter registration data by county in Maryland from 2020 and September 2022. Let's load it.

```
md_voters <- read_csv("data/maryland_voters_2020_2022.csv")
```

```
Rows: 25 Columns: 20
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (1): County
dbl (19): FIPS, DEM_2020, REP_2020, BAR_2020, GRN_2020, LIB_2020, WCP_2020, ...

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.
```

21.1 Bar charts

The first kind of chart we'll create is a simple bar chart.

It's a chart designed to show differences between things – the magnitude of one thing, compared to the next thing, and the next, and the next.

So if we have thing, like a county, or a state, or a group name, and then a count of that group, we can make a bar chart.

So what does the chart of the top 10 maryland counties with the biggest change in registered voters from 2020 to 2022 look like?

First, we'll create a dataframe of those top 10, called maryland_top_counties.

```
maryland_top_counties <- md_voters |>
  arrange(desc(TOTAL_DIFF)) |>
  select(County, TOTAL_DIFF) |>
  head(10)

maryland_top_counties
```

```
# A tibble: 10 x 2
   County           TOTAL_DIFF
   <chr>                 <dbl>
 1 Frederick             11590
 2 Montgomery             8333
 3 Howard                 5719
 4 Anne Arundel           5052
 5 Harford                4316
 6 Carroll                3908
 7 Charles                3778
 8 Cecil                  2282
 9 Baltimore County       2236
10 Saint Mary's           2180
```

Now let's create a bar chart using ggplot.

With ggplot, the first thing we'll always do is draw a blank canvas that will house our chart. We start with our dataframe name, and then (%>%) we invoke the ggplot() function to make that blank canvas. All this does is make a gray box, the blank canvas that will hold our chart.

```
maryland_top_counties |>
  ggplot()
```


Next we need to tell ggplot what kind of chart to make.

In ggplot, we work with two key concepts called geometries (abbreviated frequently as geom) and aesthetics (abbreviated as aes).

Geometries are the shape that the data will take; think of line charts, bar charts, scatterplots, histograms, pie charts and other common graphics forms.

Asesthetics help ggplot know what component of our data to visualize – why we'll visualize values from one column instead of another.

In a bar chart, we first pass in the data to the geometry, then set the aesthetic.

In the codeblock below, we've added a new function, geom_bar().

Using geom_bar() – as opposed to geom_line() – says we're making a bar chart.

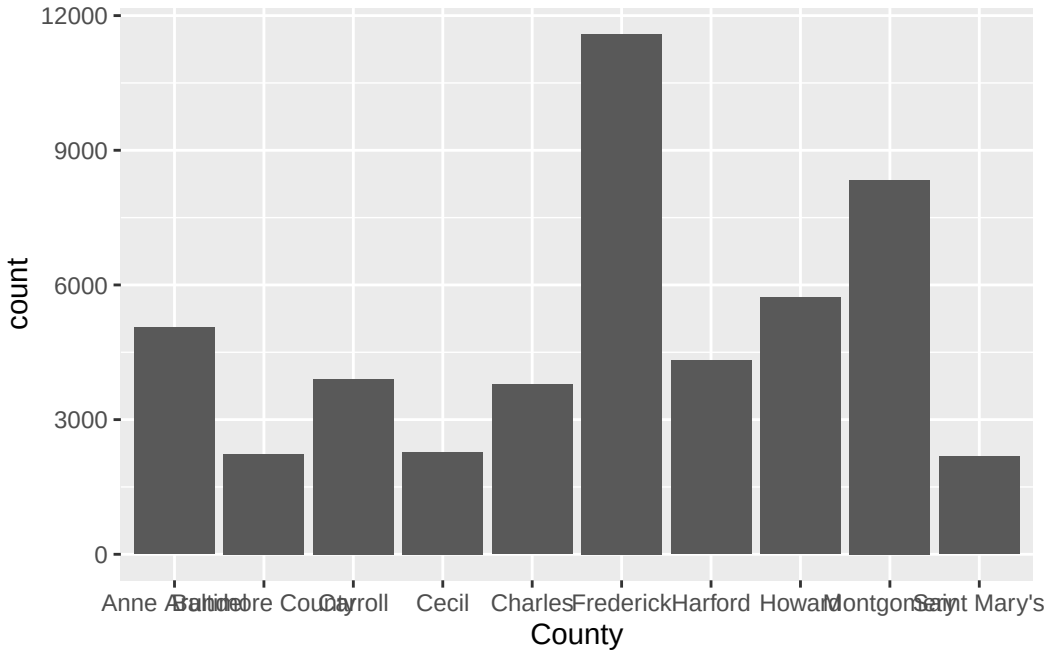
Inside of that function, the asthetic, aes, says which columns to use in drawing the chart.

We're setting the values on the x axis (horizontal) to be the name of the county. We set weight to total loans, and it uses that value to "weight" or set the height of each bar.

One quirk here with ggplot.

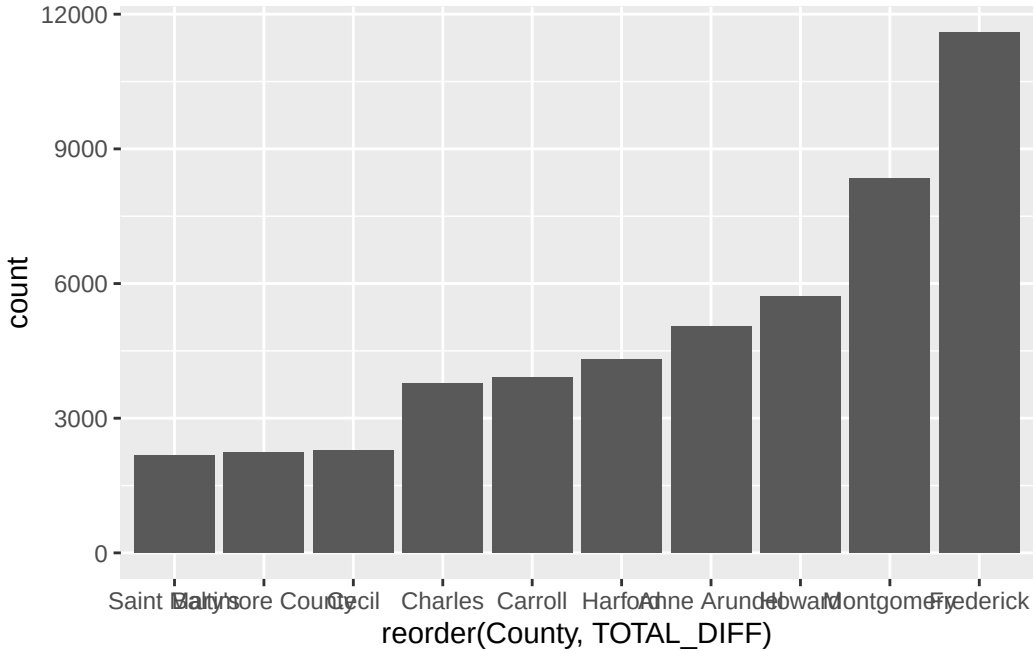
After we've invoked the ggplot() function, you'll notice we're using a + symbol. It means the same thing as %>% – "and then do this". It's just a quirk of ggplot() that after you invoke the ggplot() function, you use + instead of %>%. It makes no sense to me either, just something to live with.

```
maryland_top_counties |>
  ggplot() +
  geom_bar(aes(x=County, weight=TOTAL_DIFF))
```



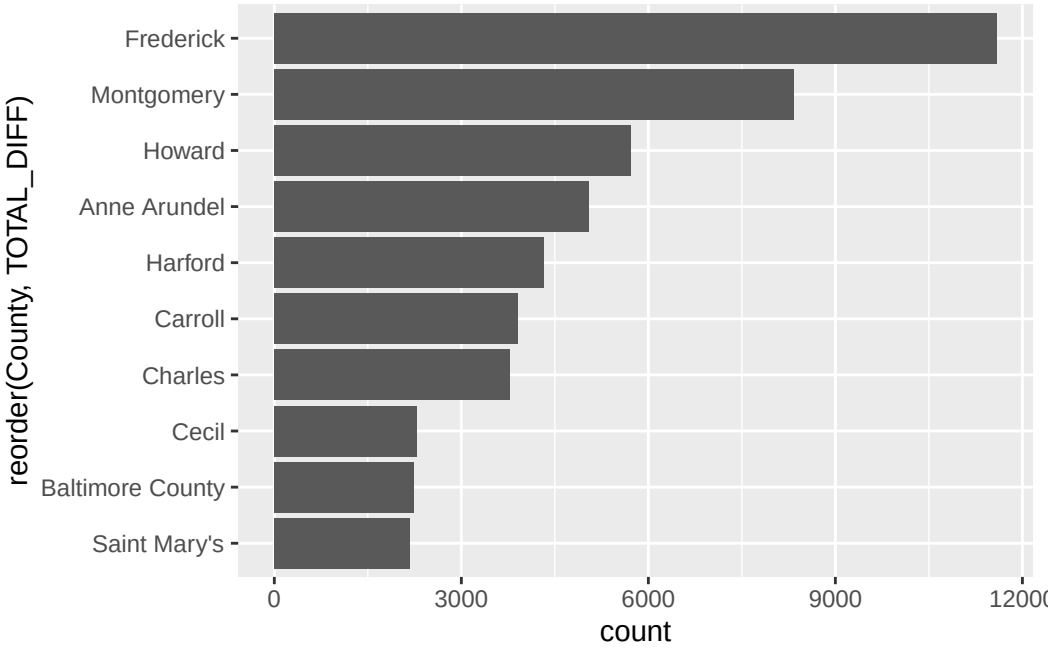
This is a very basic chart. But it's hard to derive much meaning from this chart, because the counties aren't ordered from highest to lowest by total_loans. We can fix that by using the reorder() function to do just that:

```
maryland_top_counties |>
  ggplot() +
  geom_bar(aes(x=reorder(County,TOTAL_DIFF), weight=TOTAL_DIFF))
```



This is a little more useful. But the bottom is kind of a mess, with overlapping names. We can fix that by flipping it from a vertical bar chart (also called a column chart) to a horizontal one. coord_flip() does that for you.

```
maryland_top_counties |>
  ggplot() +
  geom_bar(aes(x=reorder(County,TOTAL_DIFF), weight=TOTAL_DIFF)) +
  coord_flip()
```



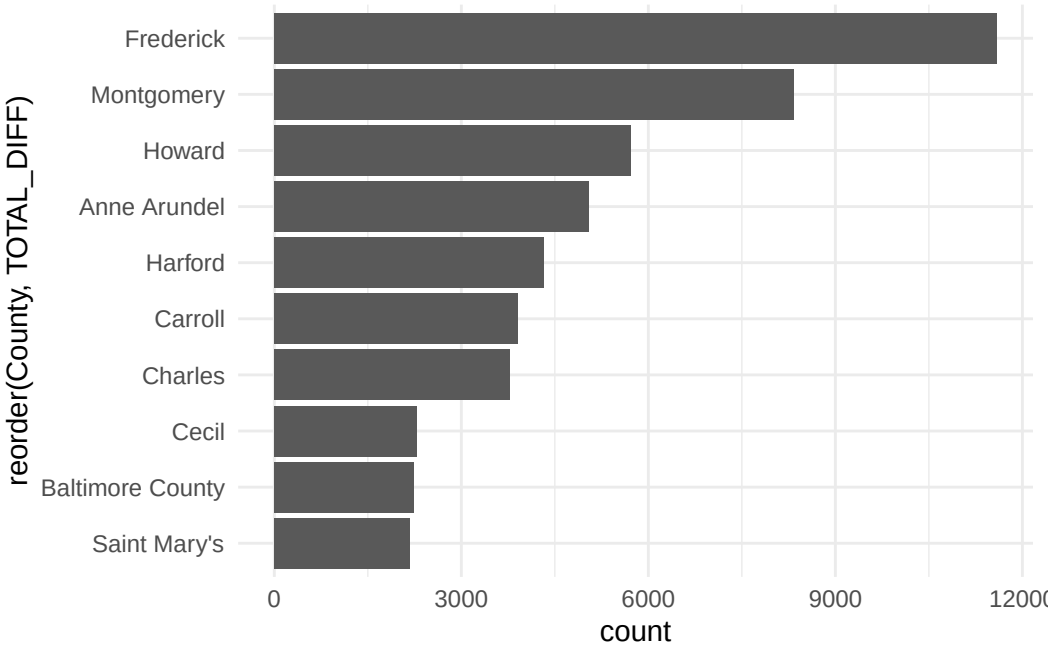
Is this art? No. Does it quickly tell you something meaningful? It does.

We're mainly going to use these charts to help us in reporting, so style isn't that important.

But it's worth mentioning that we can pretty up these charts for publication, if we wanted to, with some more code. To style the chart, we can change or even modify the "theme", a kind of skin that makes the chart look better.

It's kind of like applying CSS to html. Here I'm changing the theme slightly to remove the gray background with one of ggplot's built in themes, theme_minimal()

```
maryland_top_counties |>
  ggplot() +
  geom_bar(aes(x=reorder(County,TOTAL_DIFF), weight=TOTAL_DIFF)) +
  coord_flip() +
  theme_minimal()
```



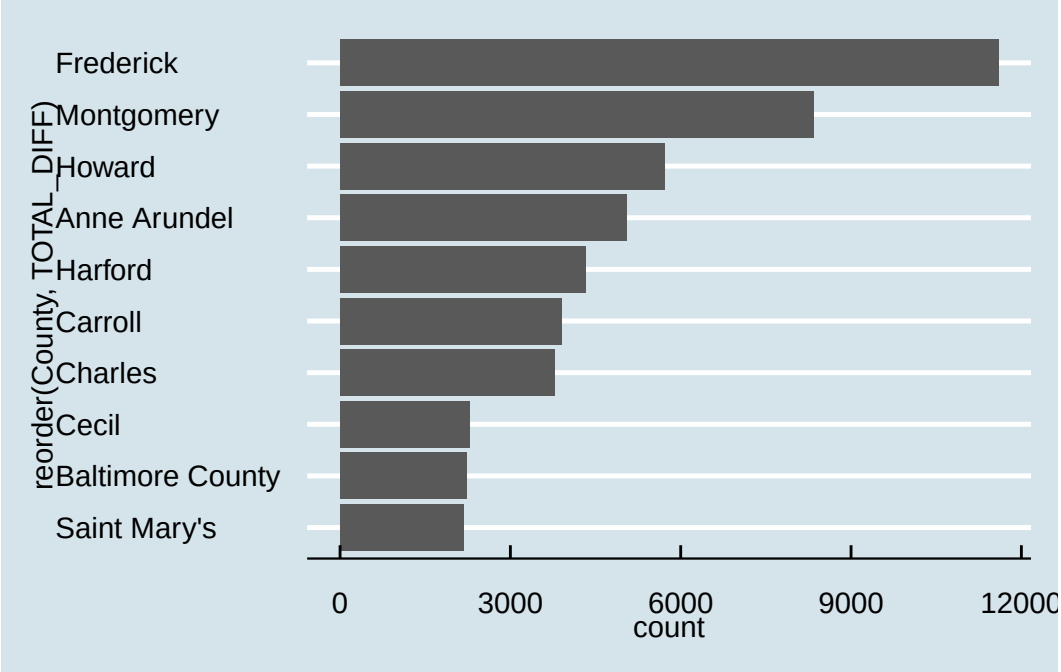
The ggplot universe is pretty big, and lots of people have made and released cool themes for you to use. Want to make your graphics look kind of like The Economist's graphics? There's a theme for that.

First, you have to install and load a package that contains lots of extra themes, called ggthemes.

```
#install.packages('ggthemes')
library(ggthemes)
```

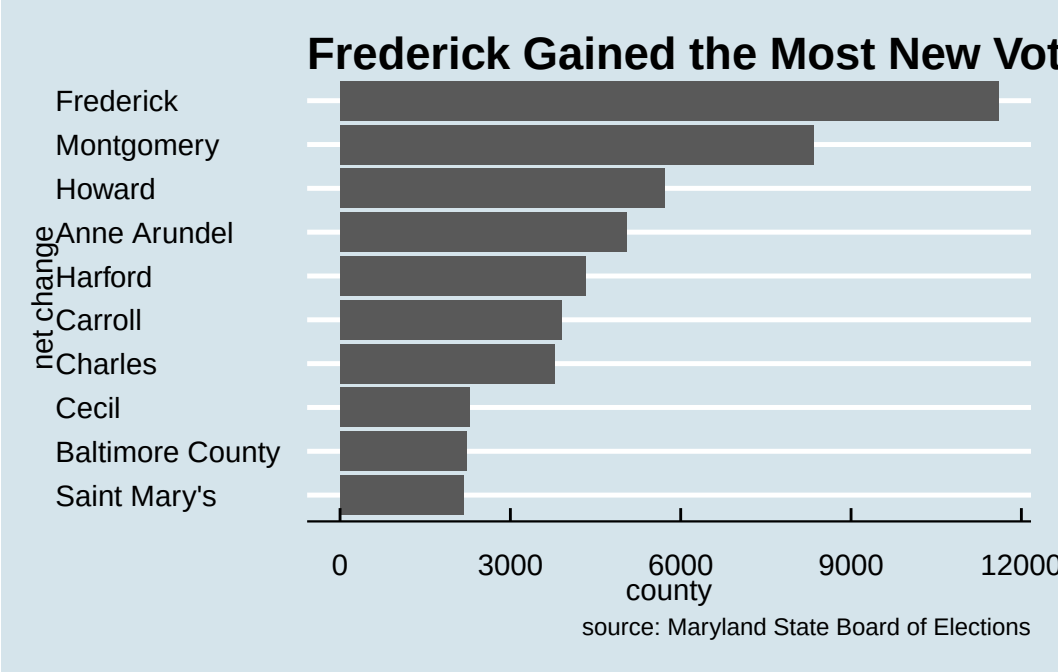
And now we'll apply the economist theme from that package with theme_economist()

```
maryland_top_counties |>
  ggplot() +
  geom_bar(aes(x=reorder(County,TOTAL_DIFF), weight=TOTAL_DIFF)) +
  coord_flip() +
  theme_economist()
```



Those axis titles are kind of a mess. Let's change "count" on the x axis to "net change" and change "reorder(County,TOTAL_DIFF)" to "county". And while we're at it, let's add a basic title and a source as a caption. We'll use a new function, labs(), which is short for labels.

```
maryland_top_counties |>
  ggplot() +
  geom_bar(aes(x=reorder(County,TOTAL_DIFF), weight=TOTAL_DIFF)) +
  coord_flip() +
  theme_economist() +
  labs(
    title="Frederick Gained the Most New Voters",
    x = "net change",
    y = "county",
    caption = "source: Maryland State Board of Elections"

  )
```



Viola. Not super pretty, but good enough to show an editor to help them understand the conclusions you reached with your data analysis.

21.2 Line charts

Let's look at how to make another common chart type that will help you understand patterns in your data.

Line charts can show change over time. It works much the same as a bar chart, code wise, but instead of a weight, it uses a y.

So, let's load some WinRed contribution data we've previously used and create a dataframe with a count of contributions for each date in our data.

```
md_winred <- read_rds("data/maryland_winred.rds")
```

Then we'll make our aggregate dataframe:

```
md_winred_by_date <- md_winred |>
  group_by(date) |>
  summarise(
    total_contributions=n()
```

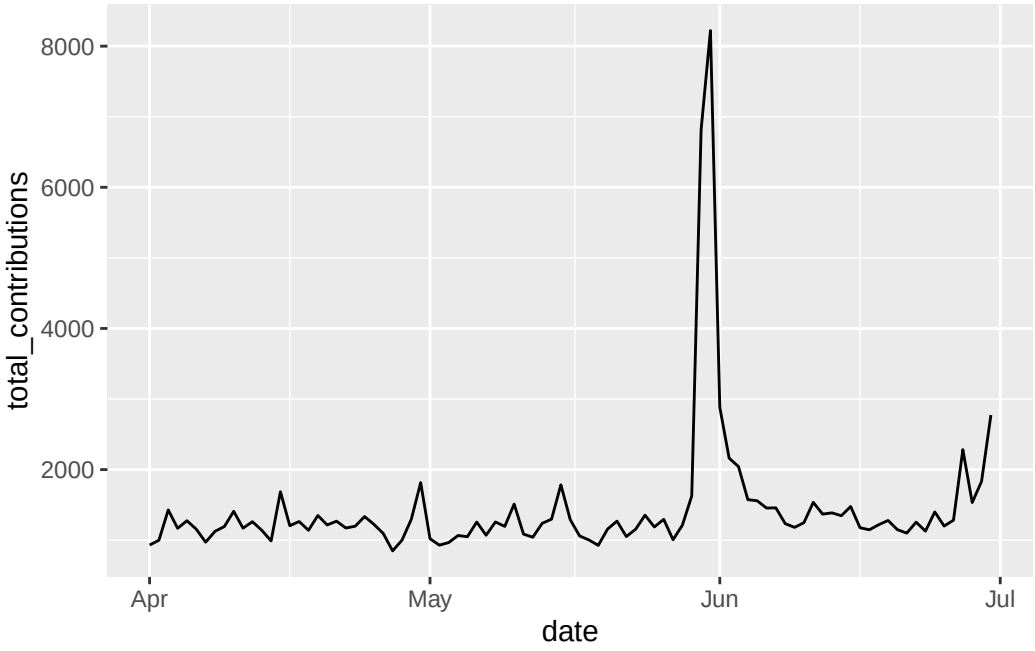
```
  )

md_winred_by_date
```

```
# A tibble: 91 x 2
   date       total_contributions
   <date>                   <int>
 1 2024-04-01                 931
 2 2024-04-02                 998
 3 2024-04-03                1429
 4 2024-04-04                1168
 5 2024-04-05                1275
 6 2024-04-06                1155
 7 2024-04-07                 971
 8 2024-04-08                1125
 9 2024-04-09                1193
10 2024-04-10                1409
# ... with 81 more rows
```

And now let's make a line chart to look for patterns in this data.

We'll put the date on the x axis and total contributions on the y axis.

```
md_winred_by_date |>
  ggplot() +
  geom_line(aes(x=date, y=total_contributions))
```



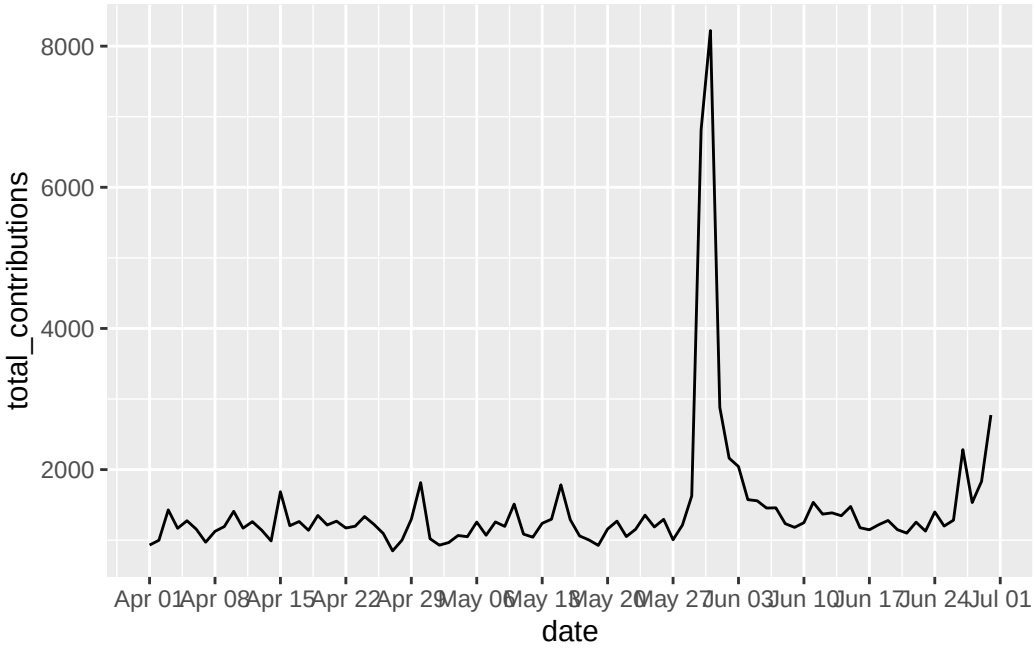
It's not super pretty, but there's a pattern here: the number of contributions fluctuates between 400 and 800 a day for most of this period, and then jumps way up at the end of June. We've learned that the end of June is the end of a reporting period, and donors respond to deadlines.

Right now, it's kind of hard to see specifics, though. When did some of those smaller spikes and troughs happen?

We can't really tell. So let's modify the x axis to have one tick mark and label per month. We can do that with a function called scale_x_date().

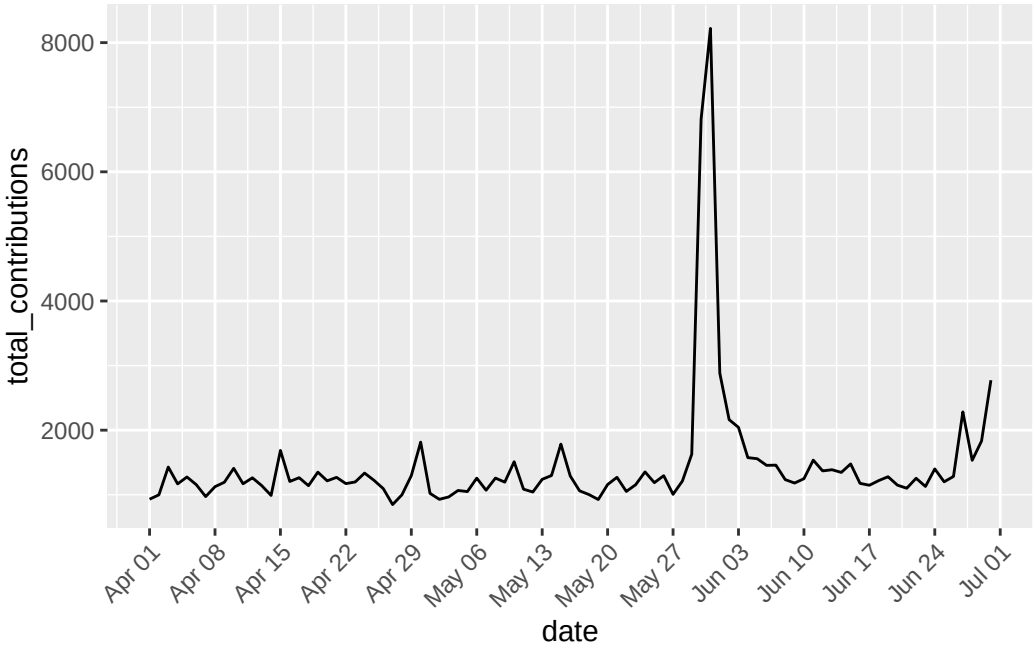
We'll set the date_breaks to appear for every week; if we wanted every month, we'd say date_breaks = "1 month". We can set the date to appear as month abbreviated name (%b) and day (%d).

```
md_winred_by_date |>
  ggplot() +
  geom_line(aes(x=date, y=total_contributions)) +
  scale_x_date(date_breaks = "1 week", date_labels = "%b %d")
```



Those are a little hard to read, so we can turn them 45 degrees to remove the overlap using the theme() function for styling. With "axis.text.x = element_text(angle = 45, hjust=1)" we're saying, turn the date labels 45 degrees.

```
md_winred_by_date |>
  ggplot() +
  geom_line(aes(x=date, y=total_contributions)) +
  scale_x_date(date_breaks = "1 week", date_labels = "%b %d") +
  theme(
    axis.text.x = element_text(angle = 45,  hjust=1)
  )
```



Again, this isn't as pretty as we could make it. But by charting this, we can quickly see a pattern that can help guide our reporting. Hmm, what happened on May 29th again?

We're just scratching the surface of what ggplot can do, and chart types. There's so much more you can do, so many other chart types you can make. But the basics we've shown here will get you started.

22 Visualizing your data for publication

Doing data visualization well, and at professional level, takes time, skill and practice to perfect. Understanding it and doing it at a complex level is an entire class on it's own. It uses some of the same skills here – grouping, filtering, calculating – but then takes that data and turns it into data pictures.

But simple stuff – and even some slightly complicated stuff – can be done with tools made for people who aren't data viz pros.

The tool we're going to use is called Datawrapper.

First, let's get some data and work with it. Let's use a simple CSV of total votes cast for four Maryland Republicans who ran for statewide office in the 2022 general election. Let's look at it.

```
library(tidyverse)
```

```
md_gop_cands <- read_csv("data/md_gop_cands.csv")
```

```
head(md_gop_cands)
```

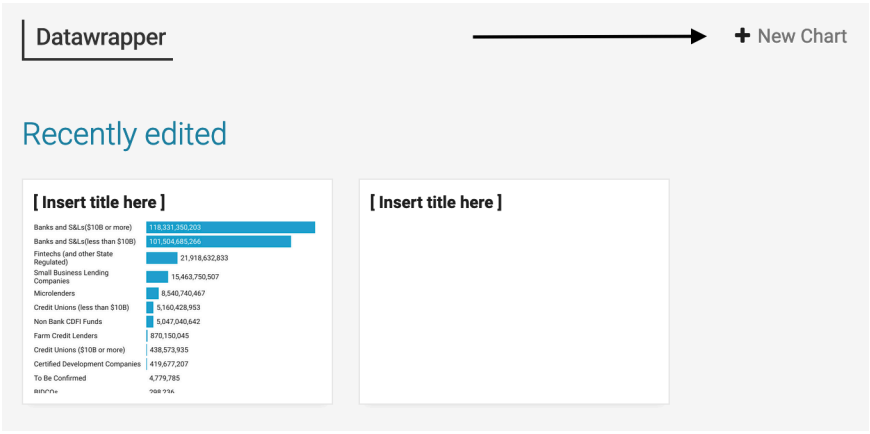
```
# A tibble: 4 x 5
  Candidate        `Early Voting` `Election Day`  Mail  Votes
  <chr>                     <dbl>          <dbl> <dbl>  <dbl>
1 Michael Peroutka         131551         451685 24669 606557
2 Dan Cox                  123287         424837 22112 569450
3 Barry Glassman           142726         484892 28656 655379
4 Chris Chaffee            130458         447330 23561 600543
```

22.1 Datawrapper

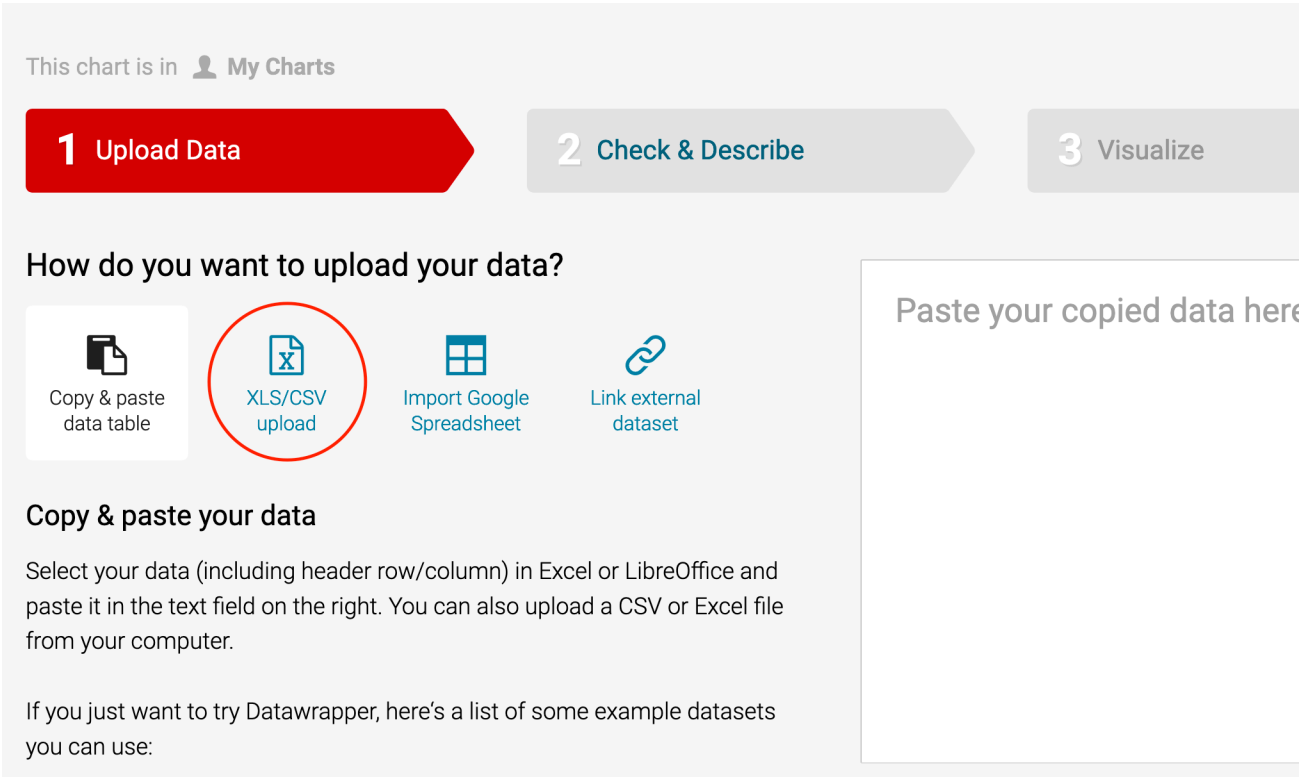
Making charts in Datawrapper is preposterously simple, which is the point. There are dozens of chart types, and dozens of options. To get from a csv to a chart to publication is very, very easy.

First, go to datawrapper.de and sign up for an account. It's free.

Once logged in, you'll click on New Chart.

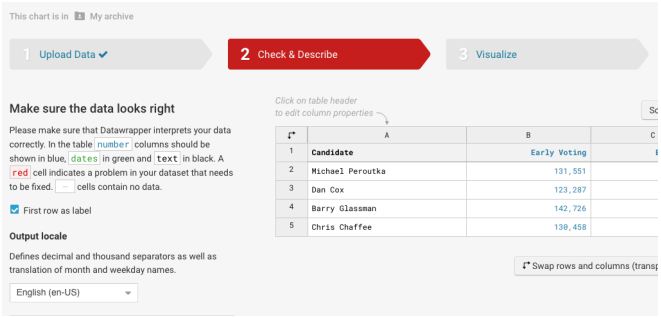


The first thing we'll do is upload our CSV. Click on XLS/CSV and upload the file.



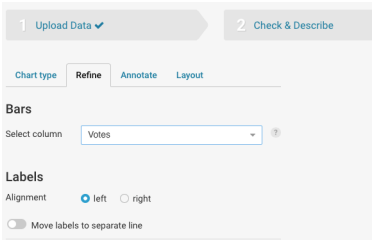
Next up is to check and see what Datawrappper did with our data when we uploaded it. As you can see from the text on the left, if it's blue, it's a number. If it's green, it's a date. If it's

black, it's text. Red means there's a problem. This data is very clean, so it imports cleanly. Click on the "Proceed" button.



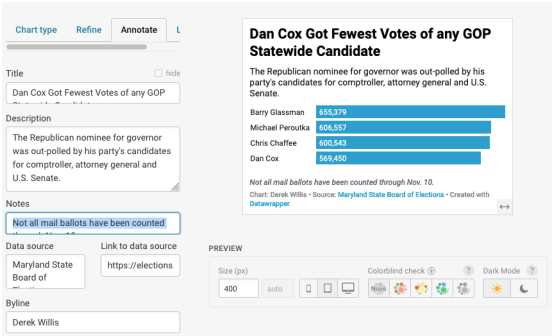
Now we make a chart. Bar chart comes up by default, which is good, because with totals, that's what we have.

Click on Refine. The first option we want to change is column we're using for the bars, which defaults to "Early Voting". Let's make it "Votes. Let's also choose to sort the bars so that the largest value (and bar appears first). We do that by clicking on the"Sort bars" button.

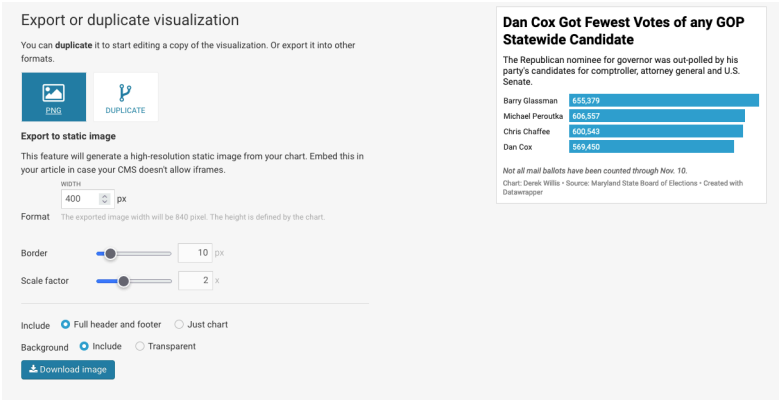


Now we need to annotate our charts. Every chart needs a title, a source line and a credit line. Most need chatter (called description here). Click on the "Annotate" tab to get started.

Really think about the title and description: the title is like a headline and the description is provides some additional context. Another way to think about it: the title is the most important lesson from the graphic, and the description could be the next most important lesson or could provide more context to the title.



To publish, we click the "Publish & Embed" tab. Some publication systems allow for the embedding of HTML into a post or a story. Some don't. The only way to know is to ask someone at your publication. Every publication system on the planet, though, can publish an image. So there's always a way to export your chart as a PNG file, which you can upload like any photo.



22.1.1 Making a Map

Let's create a choropleth map - one that shows variations between the percentage of votes received by Wes Moore across Maryland counties. We'll read that in from the data folder.

```
md_gov_county <- read_csv("data/md_gov_county.csv")
```

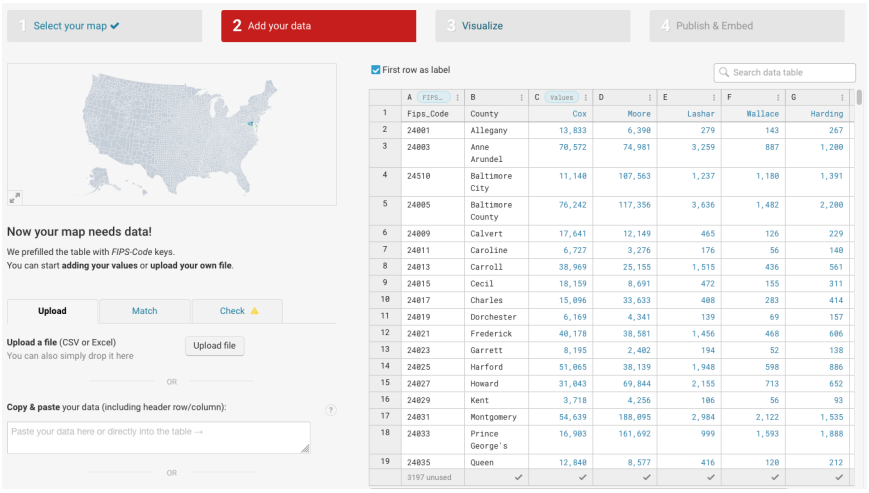
In order to make a map, we need to be able to tell Datawrapper that a certain column contains geographic information (besides the name of the county). The easiest way to do that for U.S. maps is to use something called a FIPS Code. You should read about them so you understand what they are, and think of them as a unique identifier for some geographical entity like a state or county. Our md_gov_county dataframe has a FIPS code for each county, but if you ever need one for a county, this is a solved problem thanks to the Tigris library that we used in pre_lab 9.

We'll need to write code to add columns showing the total number of votes for each county and the percentage of votes received by Wes Moore in each county, then replace the CSV file in the data folder with it

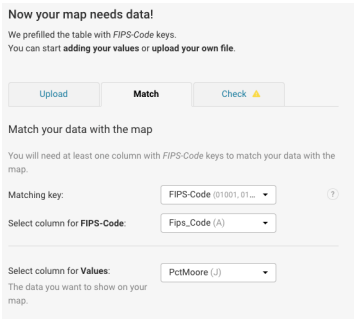
```
md_gov_county <- md_gov_county |>
  mutate(Total = Cox + Moore + Lashar + Wallace + Harding + `Write-ins`) |>
  mutate(PctMoore = Moore/Total * 100)

write_csv(md_gov_county, "data/md_gov_county_with_percent.csv")
```

Go back to Datawrapper and click on “New Map”. Click on “Choropleth map” and then choose “USA » Counties (2022)” for the map base and click the Proceed button.

Now we can upload the `md_gov_county_with_percent.csv` file we just saved using the Upload File button. It should look like the following image:



We’ll need to make sure that Datawrapper understands what the data is and where the FIPS code is. Click on the “Match” tab and make sure that yours looks like the image below:



Click the “Proceed” button (you should have to click it twice, since the first time it will tell you that there’s no data for 3,197 counties - the rest of the U.S.). That will take you to the Visualize tab.

You’ll see that the map currently is of the whole nation, and we only have Maryland data. Let’s fix that.

Look for “Hide regions without data” under Appearance, and click the slider icon to enable that feature. You should see a map zoomed into Maryland with some counties in various colors.

But it’s a little rough visually, so let’s clean that up.

Look for the “Show color legend” label and add a caption for the legend, which is the horizontal bar under the title. Then click on the “Annotate” tab to add a title, description, data source and byline. The title should represent the headline, while the description should be a longer phrase that tells people what they are looking at.

That’s better, but check out the tooltip by hovering over a county. It’s not super helpful. Let’s change the tooltip behavior to show the county name and a better-formatted number.

Click the “Customize tooltips” button so it expands down. Change {{ fips_code }} to {{ county }} and {{ pctmoore }} to {{ FORMAT(pctmoore, “00.0%”)}}

Experiment with the “Show labels” options to see if you can add county labels to your map.

Ok, that looks better. Let’s publish!

Click the “Proceed” button until you get to the “Publish & Embed” tab, then click “Publish Now”.

23 Geographic data basics

Up to now, we've been looking at patterns in data for what is more than this, or what's the middle look like. We've calculated metrics like percentages, or looked at how data changes over time.

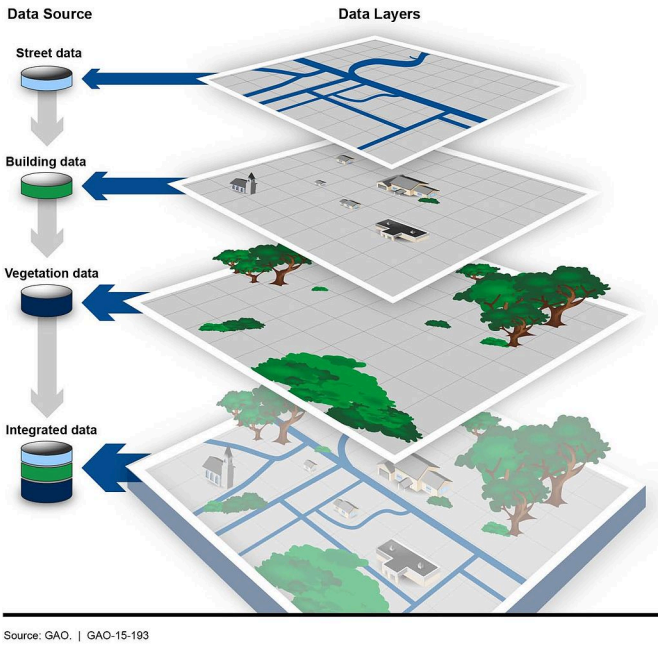
Another way we can look at the data is geographically. Is there a spatial pattern to our data? Can we learn anything by using distance as a metric? What if we merge non-geographic data into geographic data?

The bad news is that there isn't a One Library To Rule Them All when it comes to geo queries in R. But there's one emerging, called Simple Features, that is very good.

Go to the console and install it with `install.packages("sf")`

To understand geographic queries, you have to get a few things in your head first:

1. Your query is using planar space. Usually that's some kind of projection of the world. If you're lucky, your data is projected, and the software will handle projection differences under the hood without you knowing anything about it.
2. Projections are cartographers making opinionated decisions about what the world should look like when you take a spheroid – the earth isn't perfectly round – and flatten it. Believe it or not, every state in the US has their own geographic projection. There's dozens upon dozens of them.
3. Geographic queries work in layers. In most geographic applications, you'll have multiple layers. You'll have a boundary file, and a river file, and a road file, and a flood file and combined together they make the map. But you have to think in layers.
4. See 1. With layers, they're all joined together by the planar space. So you don't need to join one to the other like we did earlier – the space has done that. So you can query how many X are within the boundaries on layer Y. And it's the plane that holds them together.



23.1 Importing and viewing data

Let's start with the absolute basics of geographic data: loading and viewing. Load libraries as usual.

```
library(tidyverse)
library(sf)
library(janitor)
```

First: an aside on geographic data. There are many formats for geographic data, but data type you'll see the most is called the shapefile. It comes from a company named ERSI, which created the most widely used GIS software in the world. For years, they were the only game in town, really, and the shapefile became ubiquitous, especially so in government and utilities.

So more often than not, you'll be dealing with a shapefile. But a shapefile isn't just a single file – it's a collection of files that combined make up all the data that allow you to use it. There's a .shp file – that's the main file that pulls it all together – but it's important to note if your shapefiles has a .prj file, which indicates that the projection is specified.

You also might be working with a GeoDatabase, or a .gdb file. That's a slightly different, more compact version of a Shapefile.

The data we're going to be working with is a GeoDatabase from the Prince George's County Department of Planning that contains information about the county's election precincts.

Similar to `readr`, the `sf` library has functions to read geographic data. In this case, we're going to use `st_read` to read in our hospitals data. And then glimpse it to look at the columns.

```
pg_precincts <- st_read("data/Election_Precinct_2022_Py.gdb")
```

```
Reading layer `Election_Precinct_2022_Py' from data source
  `/Users/dwillis/code/datajournalismbook-elections/data/Election_Precinct_2022_Py.gdb'
  using driver `OpenFileGDB'
Simple feature collection with 356 features and 9 fields
Geometry type: GEOMETRY
Dimension:     XY
Bounding box:  xmin: 1290345 ymin: 317969.7 xmax: 1406390 ymax: 533363.1
Projected CRS: NAD83(2011) / Maryland (ftUS)
```

```
glimpse(pg_precincts)
```

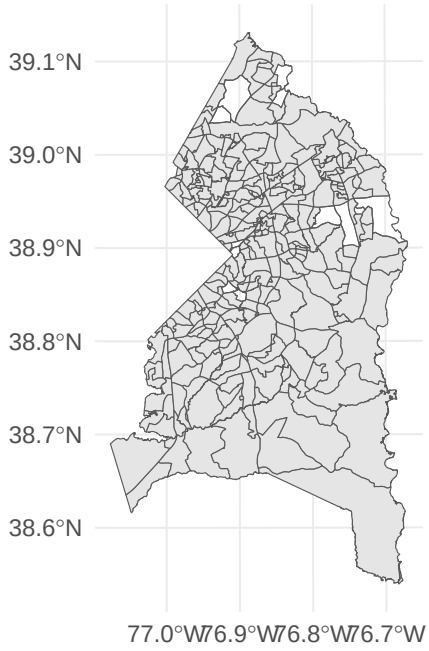
```
Rows: 356
Columns: 10
$ PRECINCT_ID   <chr> "003-004", "010-009", "012-002", "013-023", "013-006", "~
$ PRECINCT_NAME <chr> "PERRYWOOD ELEMENTARY SCHOOL", "OAKLANDS ELEMENTARY SCHO~
$ CONGRESS      <chr> "5", "4", "4", "5", "5", "5", "5", "4", "4", "5", "5", "~
$ LEGIS         <chr> "25", "21", "26", "24", "25", "23", "25", "25", "26", "2~
$ COUNCIL       <chr> "6", "1", "8", "6", "6", "9", "9", "9", "8", "9", "6", "~
$ SCHOOL        <chr> "7", "1", "8", "6", "6", "9", "9", "9", "8", "9", "6", "~
$ POLLING_ID    <chr> "003-004", "010-009", "012-002", "013-023", "013-006", "~
$ Shape_Length  <dbl> 57210.064, 32694.336, 18569.900, 9904.106, 26469.667, 32~
$ Shape_Area    <dbl> 120064600, 41053127, 16555067, 6072400, 33437592, 347735~
$ Shape         <GEOMETRY [US_survey_foot]> MULTIPOLYGON (((1378044 433..., MU~
```

This looks like a normal dataframe, and mostly it is. We have one row per precinct, and each column is some feature of that precinct: the ID, name and more. What sets this data apart from other dataframes we've used is the last column, "Shape", which is of a new data type. It's not a character or a number, it's a "Multipolygon", which is composed of multiple longitude and latitude values. When we plot these on a grid of latitude and longitude, it will draw those shapes on a map.

Let's look at these precincts. We have 356 of them, according to this data.

But where in Prince George's County are these places? We can simply plot them on a longitude-latitude grid using ggplot and geom_sf.

```
pg_precincts |>
  ggplot() +
  geom_sf() +
  theme_minimal()
```



Each shape is a precinct, with the boundaries plotted according to its degrees of longitude and latitude.

If you know anything about Prince George's, you can kinda pick out the geographic context here. To the west is the District of Columbia, for example. College Park is near the top. But this map is not exactly ideal. It would help to have a state and county map layered underneath of it, to help make sense of the spatial nature of this data.

This is where layering becomes more clear. First, we want to go out and get another shapefile, this one showing Maryland county outlines.

Instead of loading it from our local machine, like we did above, we're going to use a package to directly download it from the U.S. Census. The package is called `tigris` and it's developed by the same person who made `tidycensus`.

In the console, install tigris with `install.packages('tigris')`

Then load it:

```
library(tigris)
```

```
To enable caching of data, set `options(tigris_use_cache = TRUE)`
in your R script or .Rprofile.
```

Now, let's use the counties() function from tigris to pull down a shapefile of all U.S. counties.

```
counties <- counties()

glimpse(counties)
```

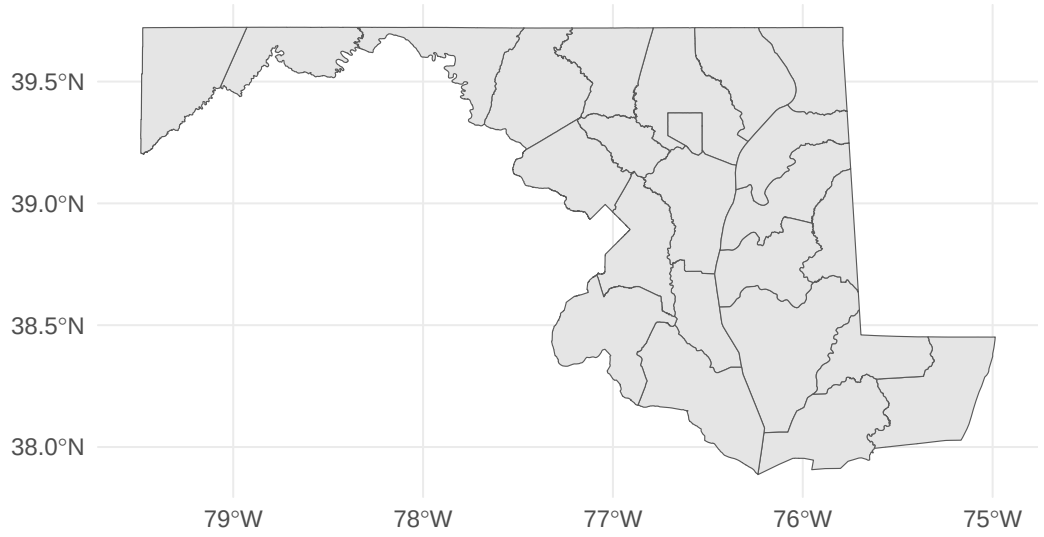
This looks pretty similar to our Census blocks shapefile, in that it looked mostly like a normal dataframe with the exception of the new geometry column.

This county shapefile has all 3233 U.S. counties. We only want the Maryland counties, so we're going to filter the data to only keep Maryland counties. There is no STATE column, but there is a STATEFP column, with each number representing a state. Maryland's FP number is 24.

```
md_counties <- counties |>
  filter(STATEFP == "24")
```

To see what this looks like, let's plot it out with ggplot. We can pretty clearly see the shapes of Maryland counties.

```
md_counties |>
  ggplot() +
  geom_sf() +
  theme_minimal()
```



Hey, look, it's Maryland! Of course, we just need Baltimore City, so let's get that:

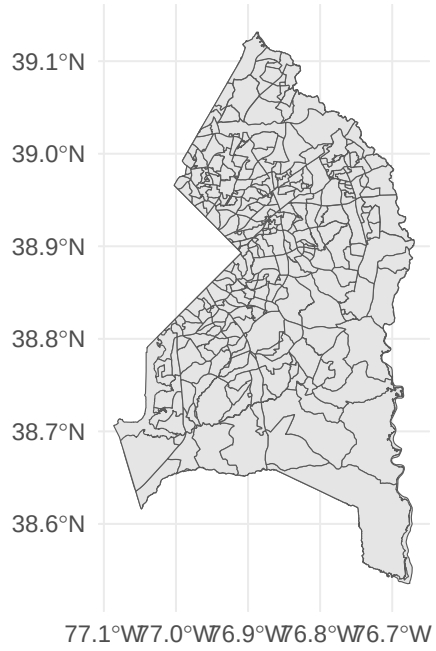
```
baltimore_city <- md_counties |>
  filter(COUNTYFP == "510")
```

With this county map, we can layer our places data.

Something to note: The layers are rendered in the order they appear. So the first geom_sf is rendered first. The second geom_sf is rendered ON TOP OF the first one.

We're also going to change things up a bit to put the datasets we want to display INSIDE of the geom_sf() function, instead of starting with a dataframe. We have two to plot now, so it's easier this way.

```
ggplot() +
  geom_sf(data=md_counties |> filter(COUNTYFP == "033")) +
  geom_sf(data=pg_precincts) +
  theme_minimal()
```



Notice the subtle differences at the boundaries?

Let's dive back into Prince George's precincts and see what more we can find out about them. It would be useful to know, for example, what turnout was like for the July primary election. We can use the state's data to determine this.

```
primary_22 <- read_csv("data/Official by Party and Precinct.csv") |> clean_names()
```

```
Rows: 14665 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (5): LBE, CONGRESSIONAL_DISTRICT_CODE, LEGISLATIVE_DISTRICT_CODE, PRECIN...
dbl (5): POLLS, EARLY_VOING, ABSENTEE, PROVISIONAL, ELIGIBLE_VOTERS

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.
```

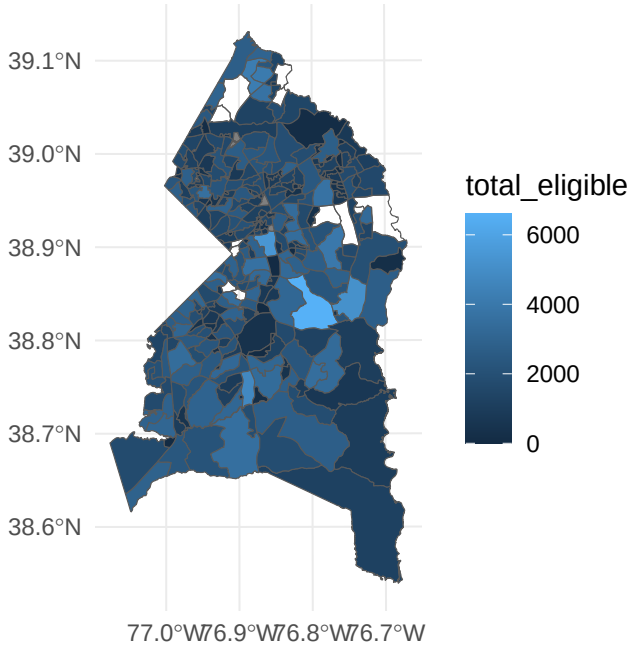
```
pg_turnout <- primary_22 |>
  filter(lbe == "Prince George's") |>
  group_by(precinct) |>
  summarise(total_polls = sum(polls), total_early = sum(early_voing), total_absentee = sum(ab

View(pg_turnout)
```

Now we can join the precincts to the turnout data.

```
pg_precincts_with_turnout <- pg_precincts |> left_join(pg_turnout, by=c("PRECINCT_ID"="preci
```

Now we can use color to distinguish precincts from each other. Let's use the total eligible voters to start with, and we'll use a color scale that will help us see the differences:

```
ggplot() +
  geom_sf(data=pg_precincts_with_turnout, aes(fill=total_eligible)) +
  scale_colour_viridis_b(option="magma") +
  theme_minimal()
```



With these changes, what else can we make out here? First, you can pretty easily spot our "ghost precincts" - they are the ones in white, where there are no eligible voters. But you also can see that there's some pretty big variation among the number of eligible voters per precinct across the county, with some very large ones in the middle.

24 Geographic analysis

In the previous chapter, we looked at election precincts in Prince George's County to show a bit of a pattern regarding concentration of the precincts with the most and 0 eligible voters. Let's go little further and look at voters statewide.

First, let's load the libraries we'll need. We're also going to load tidycensus and set an API key for tidycensus.

```
library(tidyverse)
library(sf)
library(janitor)
library(tidycensus)
census_api_key("549950d36c22ff16455fe196bbbd01d63cfbe6cf")
```

For the rest of this chapter, we're going to work on building a map that will help us gain insight into geographic patterns in voter registration by county in Maryland. Our question: by examining the number of Democrats/Republicans/Unaffiliated voters per 100,000 people in each county, what regional geographic patterns can we identify?

We've got voters by county, so let's load that and take a look:

```
voters_by_county <- read_csv("data/eligible_voters.csv")
```

```
Rows: 24 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): County
dbl (9): DEM, REP, BAR, GRN, LIB, WCP, OTH, UNA, TOTAL

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.
```

```
voters_by_county |> arrange(desc(TOTAL))
```

```
# A tibble: 24 x 10
   County              DEM    REP   BAR   GRN   LIB   WCP   OTH    UNA  TOTAL
   <chr>             <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
 1 Montgomery       409636  97505     0  1113  2227   446  5468 156978 673373
 2 Prince George's  462097  37739     0   789  1313   593 10733  85143 598407
 3 Baltimore County 309297 137378     0   898  2493   687  5921 106789 563463
 4 Anne Arundel     173922 129893     0   632  2257   434  2951  96403 406492
 5 Baltimore City   303620  28211     0   908  1080   666  3984  56665 395134
 6 Howard           120954  49011     0   351   894   134  1981  56199 229524
 7 Frederick         76712  68334     0   318  1147   179  1060  48035 195785
 8 Harford           65127  80005     0   238  1182   202  1628  40778 189160
 9 Carroll           33572  63771     0   191   797   102  1033  28139 127605
10 Charles           74373  23334     0   129   425   143   864  21819 121087
# ... with 14 more rows
```

So, what do we see here? Montgomery County has the most, followed by Prince George's & Baltimore County. Checks out.

Next, we'll go out and get population data for each county from tidycensus. The variable for total population is B01001_001.

```
md_county_population <- get_acs(geography = "county",
              variables = c(population = "B01001_001"),
              state = "MD")
```

```
Getting data from the 2017-2021 5-year ACS
```

```
md_county_population
```

```
# A tibble: 24 x 5
   GEOID NAME                            variable   estimate   moe
   <chr> <chr>                           <chr>         <dbl> <dbl>
 1 24001 Allegany County, Maryland       population    68684    NA
 2 24003 Anne Arundel County, Maryland   population   584064    NA
 3 24005 Baltimore County, Maryland      population   850702    NA
 4 24009 Calvert County, Maryland        population    92515    NA
 5 24011 Caroline County, Maryland       population    33234    NA
 6 24013 Carroll County, Maryland        population   172148    NA
 7 24015 Cecil County, Maryland          population   103370    NA
 8 24017 Charles County, Maryland        population   165209    NA
 9 24019 Dorchester County, Maryland     population    32486    NA
10 24021 Frederick County, Maryland      population   267498    NA
# ... with 14 more rows
```

Ultimately, we're going to join this county population table with our voters by county table, and then calculate a voters per 50,000 people statistic. But remember, we then want to visualize this data by drawing a county map that helps us pick out trends. Thinking ahead, we know we'll need a county map shapefile. Fortunately, we can pull this geometry information right from tidycensus at the same time that we pull in the population data by adding "geometry = TRUE" to our get_acs function.

```
md_county_population <- get_acs(geography = "county",
              variables = c(population = "B01001_001"),
              state = "MD",
              geometry = TRUE)
```

```
Getting data from the 2017-2021 5-year ACS
```

```
Downloading feature geometry from the Census website.  To cache shapefiles for use in future
```

```
  |
  |                                                                      |   0%
  |
  |                                                                      |   1%
  |
  |=                                                                     |   1%
  |
  |=                                                                     |   2%
  |
  |==                                                                    |   2%
  |
  |==                                                                    |   3%
  |
  |===                                                                   |   4%
  |
  |===                                                                   |   5%
  |
  |====                                                                  |   5%
  |
  |====                                                                  |   6%
  |
  |=====                                                                 |   7%
  |
  |=====                                                                 |   8%
```

```
|
|======                                                                |   8%
|
|========                                                              |  11%
|
|========                                                              |  12%
|
|=========                                                             |  13%
|
|==========                                                            |  14%
|
|============                                                          |  18%
|
|=============                                                         |  19%
|
|==============                                                        |  20%
|
|==============                                                        |  21%
|
|===============                                                       |  21%
|
|===============                                                       |  22%
|
|================                                                      |  23%
|
|=================                                                     |  24%
|
|=================                                                     |  25%
|
|==================                                                    |  25%
|
|==================                                                    |  26%
|
|====================                                                  |  27%
|
|====================                                                  |  28%
|
|====================                                                  |  28%
|
|====================                                                  |  29%
|
|====================                                                  |  29%
|
```

```
|=====================                                                 |  31%
|
|======================                                                |  31%
|
|======================                                                |  32%
|
|======================                                                |  33%
|
|======================                                                |  34%
|
|=======================                                               |  35%
|
|=======================                                               |  36%
|
|=======================                                               |  37%
|
|=======================                                               |  38%
|
|========================                                              |  39%
|
|========================                                              |  40%
|
|========================                                              |  41%
|
|========================                                              |  42%
|
|========================                                              |  42%
|
|=====================================                                 |  55%
|
|=====================================                                 |  56%
|
|==========================================                            |  64%
|
|==========================================                            |  65%
|
|==========================================                            |  65%
|
|==========================================                            |  66%
|
|=====================================================                 |  80%
|
|============================================================          |  89%
```

```
  |
  |=============================================================        |  89%
  |
  |==============================================================       |  90%
  |
  |===============================================================      |  91%
  |
  |================================================================     |  92%
  |
  |=================================================================    |  92%
  |
  |======================================================================| 100%
```

```
md_county_population
```

```
Simple feature collection with 24 features and 5 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -79.48765 ymin: 37.91172 xmax: -75.04894 ymax: 39.72304
Geodetic CRS:  NAD83
First 10 features:
   GEOID                              NAME   variable estimate moe
1  24047       Worcester County, Maryland population    52322  NA
2  24003    Anne Arundel County, Maryland population   584064  NA
3  24033 Prince George's County, Maryland population   957767  NA
4  24025         Harford County, Maryland population   259162  NA
5  24015           Cecil County, Maryland population   103370  NA
6  24011        Caroline County, Maryland population    33234  NA
7  24023         Garrett County, Maryland population    28955  NA
8  24029            Kent County, Maryland population    19335  NA
9  24041          Talbot County, Maryland population    37510  NA
10 24045        Wicomico County, Maryland population   103223  NA
                         geometry
1  MULTIPOLYGON (((-75.66061 3...
2  MULTIPOLYGON (((-76.83849 3...
3  MULTIPOLYGON (((-77.07995 3...
4  MULTIPOLYGON (((-76.0921 39...
5  MULTIPOLYGON (((-76.23326 3...
6  MULTIPOLYGON (((-76.01505 3...
7  MULTIPOLYGON (((-79.48765 3...
8  MULTIPOLYGON (((-76.27737 3...
9  MULTIPOLYGON (((-76.34647 3...
```

```
10 MULTIPOLYGON (((-75.92033 3...
```

We now have a new column, geometry, that contains the "MULTIPOLYGON" data that will draw an outline of each county when we go to draw a map.

The next step will be to join our population data to our voter data on the county column.

But there's a problem. The column in our population data that has county names is called "NAME", and it has the full name of the county spelled out in title case – first word capitalized and has "County" and "Maryland" in it. The voter data just has the name of the county. For example, the population data has "Anne Arundel County, Maryland" and the voter data has "Anne Arundel".

```
md_county_population
```

```
Simple feature collection with 24 features and 5 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -79.48765 ymin: 37.91172 xmax: -75.04894 ymax: 39.72304
Geodetic CRS:  NAD83
First 10 features:
   GEOID                              NAME   variable estimate moe
1  24047       Worcester County, Maryland population    52322  NA
2  24003    Anne Arundel County, Maryland population   584064  NA
3  24033 Prince George's County, Maryland population   957767  NA
4  24025         Harford County, Maryland population   259162  NA
5  24015           Cecil County, Maryland population   103370  NA
6  24011        Caroline County, Maryland population    33234  NA
7  24023         Garrett County, Maryland population    28955  NA
8  24029            Kent County, Maryland population    19335  NA
9  24041          Talbot County, Maryland population    37510  NA
10 24045        Wicomico County, Maryland population   103223  NA
                         geometry
1  MULTIPOLYGON (((-75.66061 3...
2  MULTIPOLYGON (((-76.83849 3...
3  MULTIPOLYGON (((-77.07995 3...
4  MULTIPOLYGON (((-76.0921 39...
5  MULTIPOLYGON (((-76.23326 3...
6  MULTIPOLYGON (((-76.01505 3...
7  MULTIPOLYGON (((-79.48765 3...
8  MULTIPOLYGON (((-76.27737 3...
9  MULTIPOLYGON (((-76.34647 3...
10 MULTIPOLYGON (((-75.92033 3...
```

```
voters_by_county
```

```
# A tibble: 24 x 10
   County              DEM    REP   BAR   GRN   LIB   WCP   OTH    UNA  TOTAL
   <chr>             <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
 1 Allegany          11793  22732     0    76   223    73   382   8337  43616
 2 Anne Arundel     173922 129893     0   632  2257   434  2951  96403 406492
 3 Baltimore City   303620  28211     0   908  1080   666  3984  56665 395134
 4 Baltimore County 309297 137378     0   898  2493   687  5921 106789 563463
 5 Calvert           23779  27912     0    94   410    69   548  15169  67981
 6 Caroline           6250  10539     0    34   108    40   160   4454  21585
 7 Carroll           33572  63771     0   191   797   102  1033  28139 127605
 8 Cecil             20666  31961     0   111   447   104   678  16360  70327
 9 Charles           74373  23334     0   129   425   143   864  21819 121087
10 Dorchester         9608   8965     0    22   110    33   191   3745  22674
# ... with 14 more rows
```

If they're going to join properly, we need to clean one of them up to make it match the other.

Let's clean the population table. We're going to rename the "NAME" column to "County", then remove ", Maryland" and "County" and make the county titlecase. Next we'll remove any white spaces after that first cleaning step that, if left in, would prevent a proper join. We're also going to rename the column that contains the population information from "estimate" to "population" and select only the county name and the population columns, along with the geometry. That leaves us with this tidy table.

```
md_county_population <- md_county_population |>
  rename(County = NAME) |>
  mutate(County = str_to_title(str_remove_all(County,", Maryland|County"))) |>
  mutate(County = str_trim(County,side="both")) |>
  rename(population = estimate) |>
  select(County, population, geometry)

md_county_population
```

```
Simple feature collection with 24 features and 2 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -79.48765 ymin: 37.91172 xmax: -75.04894 ymax: 39.72304
Geodetic CRS:  NAD83
First 10 features:
```

```
            County population                       geometry
1        Worcester      52322 MULTIPOLYGON (((-75.66061 3...
2     Anne Arundel     584064 MULTIPOLYGON (((-76.83849 3...
3  Prince George's     957767 MULTIPOLYGON (((-77.07995 3...
4          Harford     259162 MULTIPOLYGON (((-76.0921 39...
5            Cecil     103370 MULTIPOLYGON (((-76.23326 3...
6         Caroline      33234 MULTIPOLYGON (((-76.01505 3...
7          Garrett      28955 MULTIPOLYGON (((-79.48765 3...
8             Kent      19335 MULTIPOLYGON (((-76.27737 3...
9           Talbot      37510 MULTIPOLYGON (((-76.34647 3...
10        Wicomico     103223 MULTIPOLYGON (((-75.92033 3...
```

Now we can join them.

```
md_voters_per_10k <- md_county_population |>
  left_join(voters_by_county)
```

```
Joining with `by = join_by(County)`
```

```
md_voters_per_10k
```

```
Simple feature collection with 24 features and 11 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -79.48765 ymin: 37.91172 xmax: -75.04894 ymax: 39.72304
Geodetic CRS:  NAD83
First 10 features:
            County population    DEM    REP BAR GRN  LIB WCP   OTH   UNA  TOTAL
1        Worcester      52322  13864  19122   0  56  212  36   454  8372  42116
2     Anne Arundel     584064 173922 129893   0 632 2257 434  2951 96403 406492
3  Prince George's     957767 462097  37739   0 789 1313 593 10733 85143 598407
4          Harford     259162  65127  80005   0 238 1182 202  1628 40778 189160
5            Cecil     103370  20666  31961   0 111  447 104   678 16360  70327
6         Caroline      33234   6250  10539   0  34  108  40   160  4454  21585
7          Garrett      28955   3719  13584   0  24  101  20   157  2791  20396
8             Kent      19335   5865   5221   0  20   66  22   110  2565  13869
9           Talbot      37510  10803  11698   0  39  139  20   219  5582  28500
10        Wicomico     103223  26749  23897   0 105  347 106   667 13406  65277
                         geometry
1  MULTIPOLYGON (((-75.66061 3...
2  MULTIPOLYGON (((-76.83849 3...
```

```
3  MULTIPOLYGON (((-77.07995 3...
4  MULTIPOLYGON (((-76.0921 39...
5  MULTIPOLYGON (((-76.23326 3...
6  MULTIPOLYGON (((-76.01505 3...
7  MULTIPOLYGON (((-79.48765 3...
8  MULTIPOLYGON (((-76.27737 3...
9  MULTIPOLYGON (((-76.34647 3...
10 MULTIPOLYGON (((-75.92033 3...
```

Hang on - there's at least one county with NA values - St. Mary's, which is spelled "Saint Mary's" in the voter dataframe. And Baltimore County didn't match, either. Let's fix that using if_else, which allows us to conditionally mutate:

```
md_county_population <- md_county_population |>
  mutate(County = if_else(County == "St. Mary's", "Saint Mary's", County)) |>
  mutate(County = if_else(County == "Baltimore", "Baltimore County", County))
```

Our final step before visualization, let's calculate the number of voters per 10,000 population for each county and sort from highest to lowest to see what trends we can identify just from the table.

```
md_voters_per_10k <- md_county_population |>
  left_join(voters_by_county) |>
  mutate(voters_per_10k = TOTAL/population*10000) |>
  arrange(desc(voters_per_10k))
```

```
Joining with `by = join_by(County)`
```

```
md_voters_per_10k
```

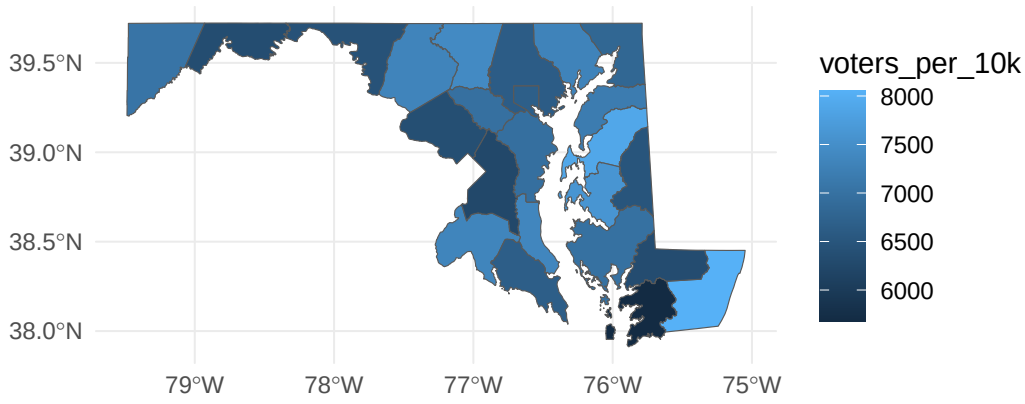
```
Simple feature collection with 24 features and 12 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -79.48765 ymin: 37.91172 xmax: -75.04894 ymax: 39.72304
Geodetic CRS:  NAD83
First 10 features:
         County population   DEM   REP BAR GRN  LIB WCP  OTH   UNA  TOTAL
1     Worcester      52322 13864 19122   0  56  212  36  454  8372  42116
2  Queen Anne's      49702 11102 19579   0  44  223  35  295  7916  39194
3        Talbot      37510 10803 11698   0  39  139  20  219  5582  28500
```

```
4       Carroll     172148 33572 63771   0 191  797 102 1033 28139 127605
5       Calvert      92515 23779 27912   0  94  410  69  548 15169  67981
6       Charles     165209 74373 23334   0 129  425 143  864 21819 121087
7     Frederick     267498 76712 68334   0 318 1147 179 1060 48035 195785
8       Harford     259162 65127 80005   0 238 1182 202 1628 40778 189160
9          Kent      19335  5865  5221   0  20   66  22  110  2565  13869
10      Garrett      28955  3719 13584   0  24  101  20  157  2791  20396
   voters_per_10k                       geometry
1        8049.386 MULTIPOLYGON (((-75.66061 3...
2        7885.799 MULTIPOLYGON (((-76.24918 3...
3        7597.974 MULTIPOLYGON (((-76.34647 3...
4        7412.517 MULTIPOLYGON (((-77.31151 3...
5        7348.106 MULTIPOLYGON (((-76.70121 3...
6        7329.322 MULTIPOLYGON (((-77.27382 3...
7        7319.120 MULTIPOLYGON (((-77.67716 3...
8        7298.910 MULTIPOLYGON (((-76.0921 39...
9        7173.002 MULTIPOLYGON (((-76.27737 3...
10       7044.034 MULTIPOLYGON (((-79.48765 3...
```

Let's take a look at the result of this table. There are some surprising ones at the top, some of Maryland's smallest counties! Worcester, Queen Anne's, Talbot may not have that many voters, but they also don't have a lot of people.

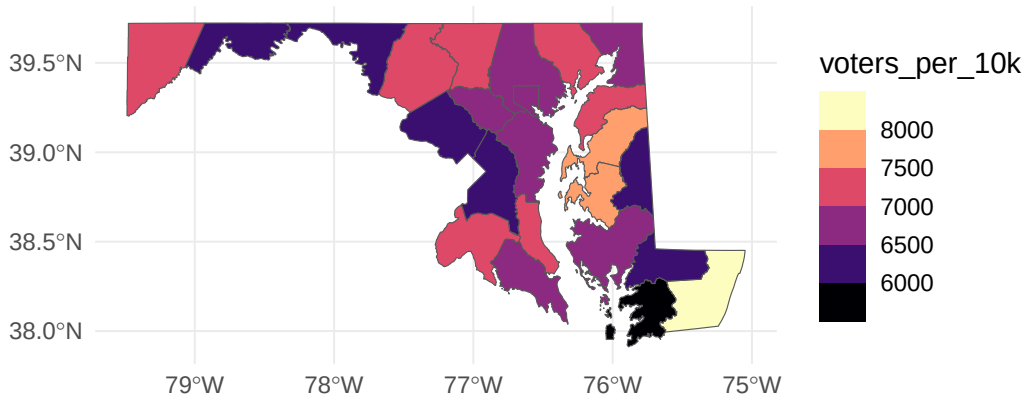
Okay, now let's visualize. We're going to build a choropleth map, with the color of each county – the fill – set according to the number of voters per 10K on a color gradient.

```
ggplot() +
  geom_sf(data=md_voters_per_10k, aes(fill=voters_per_10k)) +
  theme_minimal()
```



This map is okay, but the color scale makes it hard to draw fine-grained differences. Let's try applying the magma color scale we learned in the last chapter.

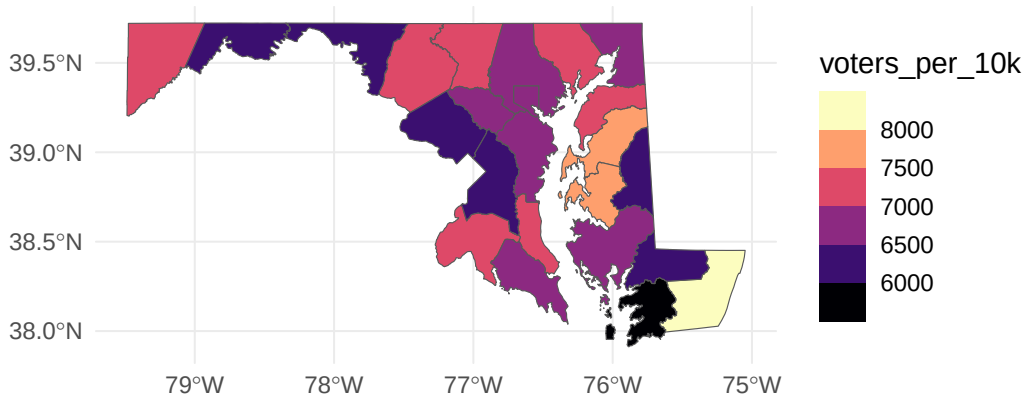
```
ggplot() +
  geom_sf(data=md_voters_per_10k, aes(fill=voters_per_10k)) +
  theme_minimal() +
  scale_fill_viridis_b(option="magma")
```



The highest ranking counties stand out nicely in this version, but it's still hard to make out fine-grained differences between other counties.

So let's change the color scale to a "log" scale, which will help us see those differences a bit more clearly.

```
ggplot() +
  geom_sf(data=md_voters_per_10k, aes(fill=voters_per_10k)) +
  theme_minimal() +
  scale_fill_viridis_b(option="magma",trans = "log")
```

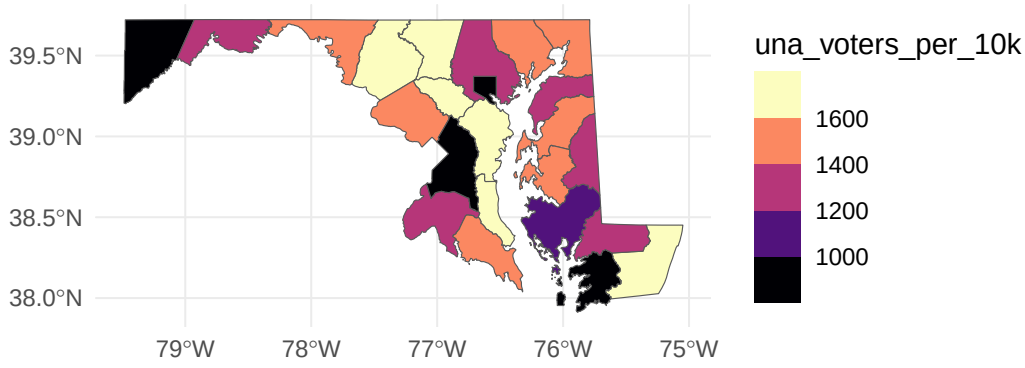


Let's repeat that for Unaffiliated voters:

```
md_voters_per_10k <- md_voters_per_10k |>
  mutate(una_voters_per_10k = UNA/population*10000)
```

And then map it:

```
ggplot() +
  geom_sf(data=md_voters_per_10k, aes(fill=una_voters_per_10k)) +
  theme_minimal() +
  scale_fill_viridis_b(option="magma",trans = "log")
```



What regional patterns do you see, especially on the ends of the scale?

25 AI and Data Journalism

The first thing to know about the large language models that have attracted so much attention, money and coverage is this: they are not fact machines.

But they are - mostly - very useful for people who write code and for those trying to work through complex problems. That's you. At its core, what a large language model does is predict the next word in a phrase or sentence. They are probabilistic prediction machines based on a huge set of training data. This chapter goes through some tasks and examples using LLMs.

25.1 Setup

We'll be using a service called Groq for the examples here. You should sign up for a free account and create an API key. Make sure you copy that key. We'll also need to install an R package to handle the responses:

```
# install.packages("devtools")
devtools::install_github("heurekalabsco/axolotr")
```

```
Skipping install of 'axolotr' from a github remote, the SHA1 (0a78e158) has not changed since
  Use `force = TRUE` to force installation
```

Then we can load that library and, using your API key, setup your credentials:

```
library(axolotr)

create_credentials(GROQ_API_KEY = "YOUR API KEY HERE")
```

```
Credentials updated successfully. Please restart your R session for changes to take effect.
```

See that "Please restart your R session for changes to take effect."? Go ahead and do that; you'll need to rerun the `library()` function above.

Let's make sure that worked. We'll be using the Llama 3.1 model released by Meta.

```
groq_response <- axolotr::ask(
  prompt = "Give me five names for a pet lemur",
  model = "llama-3.1-8b-instant"
)
```

```
Error in Groq API call:
```

```
groq_response
```

```
NULL
```

I guess you're getting a lemur?

25.2 Three Uses of AI in Data Journalism

There are at least three good uses of AI in data journalism:

- turning unstructured information into data
- helping with code debugging and explanation
- brainstorming about strategies for data analysis and visualization

If you've tried to use a large language model to actually do data analysis, it *can* work, but often the results can be frustrating. Think of AI as a potentially useful assistant for the work you're doing. If you have a clear idea of the question you want to ask or the direction you want to go, they can help. If you don't have a clear idea or question, they probably will be less helpful. Let's go over a quick example of each use.

25.2.1 Turning Unstructured Information into Data

News organizations are sitting on a trove of valuable raw materials - the words, images, audio and video that they produce every day. We can (hopefully) search it, but search doesn't always deliver meaning, let alone elevate patterns. For that, often it helps to turn that information into structured data. Let's look at an example involving my friend Tyson Evans, who recently celebrated his 10th wedding anniversary. You can read about his wedding in The New York Times.

This announcement is a story, but it's also data - or it should be.

Gabriela Herman, Tyson Evans

Share full article

June 22, 2014

Gabriela Nunes Herman and Tyson Charles Evans were married Saturday at the home of their friends Marcy Gringlas and Joel Greenberg in Chilmark, Mass. Rachel Been, a friend of the couple who received a one-day solemnization certificate from Massachusetts, officiated.

The bride, 33, will continue to use her name professionally. She is a Brooklyn-based freelance photographer for magazines and newspapers. She graduated from Wesleyan University in Middletown, Conn.

She is a daughter of Dr. Talia N. Herman of Brookline, Mass., and Jeffrey N. Herman of Cambridge, Mass. The bride's father is a lawyer and the executive vice president of DecisionQuest, a national trial consulting firm in Boston. Her mother is a senior primary care internist at Harvard Vanguard Medical Associates, a practice in Boston.

What if we could extract those highlighted portions of the text into, say, a CSV file? That's something that LLMs are pretty good at. Let's give it a shot using the full text of that announcement:

```
text = "Gabriela Nunes Herman and Tyson Charles Evans were married Saturday at the home of th

evans_response <- axolotr::ask(
  prompt = paste("Given the following text, extract information into a CSV file with the foll
  model = "llama-3.1-8b-instant"
)
```

```
Error in Groq API call:
```

```
evans_response
```

```
NULL
```

A brief word about that "no yapping" bit; it's a way to tell your friendly LLM to cut down on the chattiness in its response. What we care about is the data, not the narrative. And look at the results: without even providing an example or saying that the text described a wedding, the LLM did a solid job. Now imagine if you could do this with hundreds or thousands of similar announcements. You've just built a database.

25.2.2 Helping with Code Debugging and Explanation

When you're writing code and run into error messages, you should read them. But if they do not make sense to you, you can ask an LLM to do some translation, which is another great use case for AI. As with any debugging exercise, you should provide some context, things like "Using R and the tidyverse …" and describing what you're trying to do, but you also can ask LLMs to explain an error message in a different way. Here's an example:

```
debug_response <- axolotr::ask(
  prompt = "Explain the following R error message using brief, simple language and suggest a
  model = "llama-3.1-8b-instant"
)
```

```
Error in Groq API call:
```

```
debug_response
```

```
NULL
```

The trouble is that if you run that several times, it will give you slightly different answers. Not fact machines. But you should be able to try some of the suggested solutions and see if any of them work. An even better use could be to pass in working code that you're not fully understanding and ask the LLM to explain it to you.

25.2.3 Brainstorming about Strategies for Data Analysis and Visualization

Let's say that you have some data that you want to interview, but aren't sure how to proceed. LLMs can provide some direction, but you may not want to follow their directions exactly. You shouldn't accept their judgments uncritically; you'll still need to think for yourself. Here's an example of how that might go:

```
idea_response <- axolotr::ask(
  prompt = "I have a CSV file of daily data on campus police incidents, including the type o
  model = "llama-3.1-8b-instant"
)
```

```
Error in Groq API call:
```

```
idea_response
```

```
NULL
```

Note that the column names may not match your data; the LLM is making predictions about your data, so you could provide the column names. As this story from The Pudding makes clear, the potential for using LLMs to not just assist with but perform data analysis is real. What will make the difference is how much context you can provide and how clear your ideas and questions are. You still have to do the work.

26 An intro to text analysis

Throughout this course, we've been focused on finding information in structured data. We've learned a lot of techniques to do that, and we've learned how the creative mixing and matching of those skills can find new insights.

What happens when the insights are in unstructured data? Like a block of text?

Turning unstructured text into data to analyze is a whole course in and of itself – and one worth taking if you've got the credit hours – but some simple stuff is in the grasp of basic data analysis.

To do this, we'll need a new library – tidytext, which you can guess by the name plays very nicely with the tidyverse. So install it with `install.packages("tidytext")` and we'll get rolling.

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

Here's the question we're going to go after: what words or phrases appear most in Maryland Sen. Ben Cardin's press releases?

To answer this question, we'll use the text of Cardin's most recent 999 press releases.

Let's read in this data and examine it:

```
releases <- read_rds("data/cardin_releases.rds")
```

We can see what it looks like with head:

```
head(releases)
```

```
# A tibble: 6 x 4
  date       title                                                 url   text
  <date>     <chr>                                                 <chr> <chr>
```

```
1 2022-12-01 Cardin, Trone Lead Effort to Review Health Impacts of ~ http~ " -
~
2 2022-12-01 Cardin Votes to Avert Rail Strike, Will Continue to Ad~ http~ " -
~
3 2022-12-01 Cardin Introduces Legislation to Reauthorize and Impro~ http~ "-
U~
4 2022-12-01 Cardin Marks World AIDS Day 2022                       http~ " -
~
5 2022-12-01 Wyden, Crapo and Bipartisan Senate Finance Committee M~ http~ " -
~
6 2022-12-01 Cardin, Wicker Lead Legislation to Designate Wagner Gr~ http~ " -
~
```

What we want to do is to make the `text` column easier to analyze. Let's say we want to find out the most commonly used words. We'll want to remove URLs from the text of the releases since they aren't actual words. Let's use mutate to make that happen:

```
releases <- releases |>
  mutate(text = gsub("http.*","", text))
```

If you are trying to create a list of unique words, R will treat differences in capitalization as unique and also will include punctuation by default, even using its `unique` function:

```
a_list_of_words <- c("Dog", "dog", "dog", "cat", "cat", ",")
unique(a_list_of_words)
```

```
[1] "Dog" "dog" "cat" ","
```

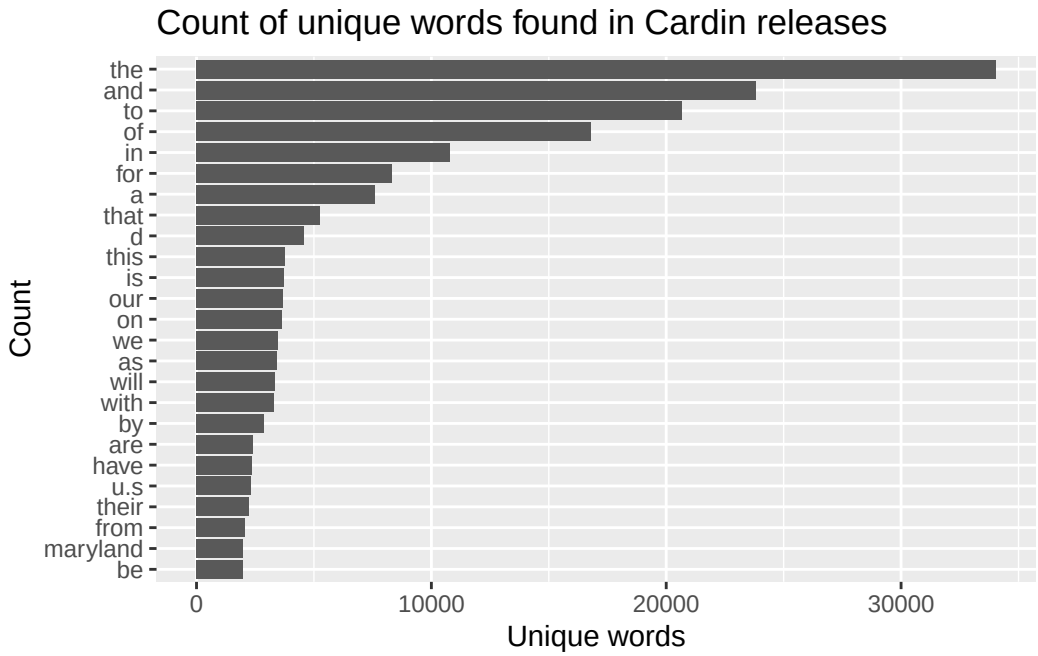
Fortunately, this is a solved problem with tidytext, which has a function called `unnest_tokens` that will convert the text to lowercase and remove all punctuation. The way that `unnest_tokens` works is that we tell it what we want to call the field we're creating with this breaking apart, then we tell it what we're breaking apart – what field has all the text in it. For us, that's the `text` column:

```
unique_words <- releases |> select(text) |>
  unnest_tokens(word, text)
View(unique_words)
```

Now we can look at the top words in this dataset. Let's limit ourselves to making a plot of the top 25 words:

```
unique_words |>
  count(word, sort = TRUE) |>
  top_n(25) |>
  mutate(word = reorder(word, n)) |>
  ggplot(aes(x = word, y = n)) +
  geom_col() +
  xlab(NULL) +
  coord_flip() +
      labs(x = "Count",
      y = "Unique words",
      title = "Count of unique words found in Cardin releases")
```

```
Selecting by n
```



Well, that's a bit underwhelming - a lot of very common (and short) words. This also is a solved problem in working with text data, and words like "a" and "the" are known as "stop words". In most cases you'll want to remove them from your analysis since they are so common. Tidytext provides a dataframe of them that we'll load, and then we'll add some of our own.

```
data("stop_words")

stop_words <- stop_words |>
```

```
  add_row(word = "ben") |>
  add_row(word = "cardin") |>
  add_row(word = "senator") |>
  add_row(word = "senators") |>
  add_row(word = "maryland") |>
  add_row(word = 'federal') |>
  add_row(word = 'u.s') |>
  add_row(word = 'md') |>
  add_row(word = 'senate') |>
  add_row(word = "hollen") |>
  add_row(word = "van") |>
  add_row(word = "chris")
```

Then we're going to use a function we haven't used yet called an `anti_join`, which filters out any matches. So we'll `anti_join` the stop words and get a list of words that aren't stop words.

From there, we can get a simple word frequency by just grouping them together and counting them. We can borrow the percent code from above to get a percent of the words our top 10 words represent.

```
unique_words |>
  anti_join(stop_words) |>
  group_by(word) |>
  tally(sort=TRUE) |>
  mutate(percent = (n/sum(n))*100) |>
  top_n(10)
```

```
Joining with `by = join_by(word)`
Selecting by percent
```

```
# A tibble: 10 x 3
   word            n percent
   <chr>       <int>   <dbl>
 1 health       1804   0.576
 2 support      1509   0.481
 3 act          1488   0.475
 4 funding      1457   0.465
 5 program      1297   0.414
 6 project      1255   0.400
 7 communities  1189   0.379
```

```
 8 community    1015   0.324
 9 public       1014   0.323
10 covid        1008   0.322
```

Those seem like more relevant unique words. Now, here's where we can start to do more interesting and meaningful analysis. Let's create two dataframes of unique words based on time: one for all of 2021 and the other for all of 2022:

```
unique_words_2021 <- releases |>
  filter(date < '2022-01-01') |>
  select(text) |>
  unnest_tokens(word, text)

unique_words_2022 <- releases |>
  filter(date >= '2022-01-01') |>
  select(text) |>
  unnest_tokens(word, text)
```

Then we can create top 10 lists for both of them and compare:

```
unique_words_2021 |>
  anti_join(stop_words) |>
  group_by(word) |>
  tally(sort=TRUE) |>
  mutate(percent = (n/sum(n))*100) |>
  top_n(10)
```

```
Joining with `by = join_by(word)`
Selecting by percent
```

```
# A tibble: 10 x 3
   word            n percent
   <chr>       <int>   <dbl>
 1 health       1177   0.608
 2 act          1025   0.530
 3 funding       962   0.497
 4 support       902   0.466
 5 covid         807   0.417
 6 19            806   0.417
 7 communities   767   0.396
 8 program       736   0.380
```

```
 9 american      708   0.366
10 pandemic      708   0.366
```

```
unique_words_2022 |>
  anti_join(stop_words) |>
  group_by(word) |>
  tally(sort=TRUE) |>
  mutate(percent = (n/sum(n))*100) |>
  top_n(10)
```

```
Joining with `by = join_by(word)`
Selecting by percent
```

```
# A tibble: 10 x 3
   word            n percent
   <chr>       <int>   <dbl>
 1 project       834   0.695
 2 health        627   0.523
 3 support       607   0.506
 4 program       561   0.468
 5 funding       495   0.413
 6 act           463   0.386
 7 community     453   0.378
 8 communities   422   0.352
 9 funds         421   0.351
10 baltimore     416   0.347
```

In the 2021 top 10 list, "covid" and "pandemic" appear, which makes sense, while the 2022 doesn't reference the same topic aside from the more generic "health".

26.1 Going beyond a single word

The next step in text analysis is using `ngrams`. An `ngram` is any combination of words that you specify. Two word ngrams are called bigrams (bi-grams). Three would be trigrams. And so forth.

The code to make ngrams is similar to what we did above, but involves some more twists.

So this block is is going to do the following:

1. Use the releases data we created above, and filter for pre-2022 releases.

2. Unnest the tokens again, but instead we're going to create a field called bigram, break apart summary, but we're going to specify the tokens in this case are ngrams of 2.
3. We're going to make things easier to read and split bigrams into word1 and word2.
4. We're going to filter out stopwords again, but this time we're going to do it in both word1 and word2 using a slightly different filtering method.
5. Because of some weirdness in calculating the percentage, we're going to put bigram back together again, now that the stop words are gone.
6. We'll then group by, count and create a percent just like we did above.
7. We'll then use top_n to give us the top 10 bigrams.

```
releases |>
  filter(date < '2022-01-01') |>
  unnest_tokens(bigram, text, token = "ngrams", n = 2) |>
  separate(bigram, c("word1", "word2"), sep = " ") |>
  filter(!word1 %in% stop_words$word) |>
  filter(!word2 %in% stop_words$word) |>
  mutate(bigram = paste(word1, word2, sep=" ")) |>
  group_by(bigram) |>
  tally(sort=TRUE) |>
  mutate(percent = (n/sum(n))*100) |>
  top_n(10)
```

```
Selecting by percent
```

```
# A tibble: 10 x 3
   bigram              n percent
   <chr>           <int>   <dbl>
 1 covid 19          786   0.856
 2 human rights      319   0.347
 3 19 pandemic       288   0.314
 4 health care       224   0.244
 5 chesapeake bay    190   0.207
 6 american rescue   166   0.181
 7 rescue plan       166   0.181
 8 public health     157   0.171
 9 john sarbanes     133   0.145
10 kweisi mfume      133   0.145
```

And we already have a different, more nuanced result. Health was among the top single words, and we can see that "health care", "human rights" and "chesapeake bay" are among the top 2-word phrases. What about after 2021?

```
releases |>
  filter(date >= '2022-01-01') |>
  unnest_tokens(bigram, text, token = "ngrams", n = 2) |>
  separate(bigram, c("word1", "word2"), sep = " ") |>
  filter(!word1 %in% stop_words$word) |>
  filter(!word2 %in% stop_words$word) |>
  mutate(bigram = paste(word1, word2, sep=" ")) |>
  group_by(bigram) |>
  tally(sort=TRUE) |>
  mutate(percent = (n/sum(n))*100) |>
  top_n(10)
```

```
Selecting by percent
```

```
# A tibble: 10 x 3
   bigram                n percent
   <chr>             <int>   <dbl>
 1 project location    262   0.447
 2 covid 19            167   0.285
 3 health care         140   0.239
 4 chesapeake bay      139   0.237
 5 human rights        119   0.203
 6 amount included     113   0.193
 7 baltimore city      100   0.171
 8 prince george's      98   0.167
 9 mental health        94   0.160
10 location baltimore   85   0.145
```

While "covid 19" is still a top phrase, it's not *the* leading phrase any longer. "Mental health" makes an appearance, too. You'll notice that the percentages are very small; that's not irrelevant but in some cases it's the differences in patterns that's more important.

There are some potential challenges to doing an analysis. For one, there are variations of words that could probably be standardized - maybe using OpenRefine - that would give us cleaner results. There might be some words among our list of stop words that actually are meaningful in this context.

26.2 Sentiment Analysis

Another popular use of text analysis is to measure the sentiment of a word - whether it expresses a positive or negative idea - and tidytext has built-in tools to make that possible.

We use word counts like we've already calculated and bring in a dataframe of words (called a lexicon) along with their sentiments using a function called `get_sentiments`. The most common dataframe is called "bing" which has nothing to do with the Microsoft search engine. Let's load it:

```
bing <- get_sentiments("bing")

bing_word_counts_2021 <- unique_words_2021 |>
  inner_join(bing) |>
  count(word, sentiment, sort = TRUE)
```

```
Joining with `by = join_by(word)`
```

```
bing_word_counts_2022 <- unique_words_2022 |>
  inner_join(bing) |>
  count(word, sentiment, sort = TRUE)
```

```
Joining with `by = join_by(word)`
```

```
View(bing_word_counts_2021)
View(bing_word_counts_2022)
```

Gauging the sentiment of a word can be heavily dependent on the context, and as with other types of text analysis sometimes larger patterns are more meaningful than individual results. But the potential with text analysis is vast: knowing what words and phrases that public officials employ can be a way to evaluate their priorities, cohesiveness and tactics for persuading voters and their colleagues. And those words and phrases are data.

27 Basic Stats: Linear Regression and The T-Test

A month into the Covid-19 pandemic, in April 2020, Reveal, an investigative reporting outfit, wrote a story based on original data analysis showing that a disproportionate share of PPP loans were going to states that Donald Trump won in 2016. In North Dakota, a state that gave a higher share of its vote to Trump than all but three states, 58 percent of small businesses got PPP loans. In Democratic-leaning New York, which was hit hard in the pandemic's first wave, only 18 percent of small businesses received loans. They wrote:

"Reveal's analysis found that businesses in states that Trump won in 2016 received a far greater share of the small-business relief funds than those won by his Democratic rival, Hillary Clinton. Eight of the top 10 recipient states – ranked according to the proportion of each state's businesses that received funding – went to Trump in 2016. Meanwhile, seven of the bottom 10 states, where the lowest proportion of businesses received funding, went to Clinton. Taken together, 32% of businesses in states that Trump won got Paycheck Protection Program dollars, we found, compared with 22% of businesses in states that went to Clinton."

It continued: "The figures were so stark that they sparked concerns of political interference. Rep. Jackie Speier, a California Democrat who serves on the House Oversight and Reform Committee, said the data raise questions about whether stimulus dollars were deliberately funneled to states that voted for Trump and have Republican governors."

The story didn't present any evidence of political meddling. Instead, it offered the results of several lines of data analysis that attempted to answer this central question: did red states get a bigger slice of the PPP pie than blue states?

Mostly, it used basic descriptive statistics, calculating rates, ranking states and computing averages. But the data set it used also presents an opportunity to use two slightly more advanced statistical analysis methods to look for patterns: linear regression, to examine relationships, and a t.test, to confirm the statistical validity of an average between two groups. So, let's do that here.

First, let's load libraries. We're going to load janitor, the tidyverse and a new package, corrr, which will help us do linear regression a bit easier than base R.

```
library(janitor)
library(tidyverse)
library(corrr)
```

Now let's load the data we'll be using. It has five fields:

- state_name
- vote_2016: whether Trump or Clinton won the state's electoral vote.
- pct_trump: the percentage of the vote Trump received in the state.
- businesses_receiving_ppe_pct: the percentage of the state's small businesses that received a PPP loan.
- ppe_amount_per_employee: the average amount of money provided by PPP per small business employee in the state.

```
reveal_data <- read_rds("data/reveal_data.rds")

reveal_data
```

```
# A tibble: 51 x 5
   state_name   vote_2016 pct_trump businesses_receiving_ppe_pct ppe_amount_pe~1
   <chr>        <chr>         <dbl>                        <dbl>            <dbl>
 1 North Dakota Trump          63.0                           58             7928
 2 Nebraska     Trump          58.8                           56             7244
 3 South Dakota Trump          61.5                           53             6541
 4 Oklahoma     Trump          65.3                           50             6499
 5 Mississippi  Trump          57.9                           49             5674
 6 Iowa         Trump          51.2                           48             6642
 7 Kansas       Trump          56.6                           47             7087
 8 Hawaii       Clinton        29.4                           47             7417
 9 Maine        Clinton        43.5                           45             6617
10 Arkansas     Trump          60.6                           44             5549
# ... with 41 more rows, and abbreviated variable name
#   1: ppe_amount_per_employee
```

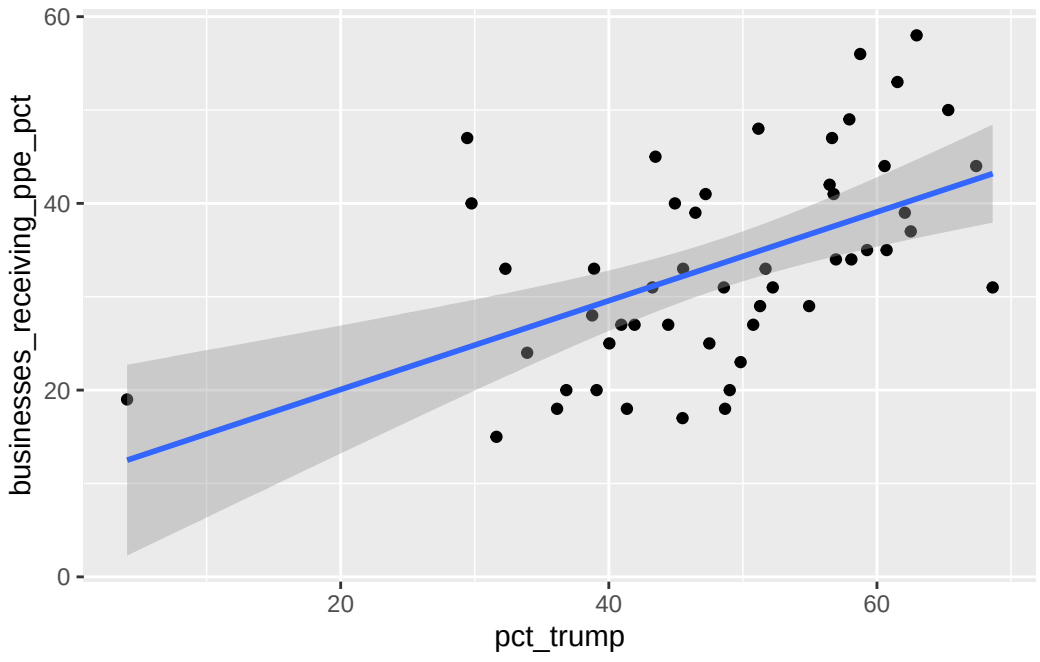
28 Linear Regression

Let's start with this question: did small businesses in states that voted more strongly for Trump get loans at higher rate than small businesses in Democratic states? We can answer it by examining the relationship or correlation between two variables, pct_trump and businesses_receiving_ppe_pct. How much do they move in tandem? Do states with more Trump support see bigger average PPP loans? Do extra Trumpy states get even more? Do super blue states get the least?

Let's start by plotting them to get a sense of the pattern.

```
reveal_data |>
  ggplot() +
  geom_point(aes(x=pct_trump,y=businesses_receiving_ppe_pct)) +
  geom_smooth(aes(x=pct_trump,y=businesses_receiving_ppe_pct), method="lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

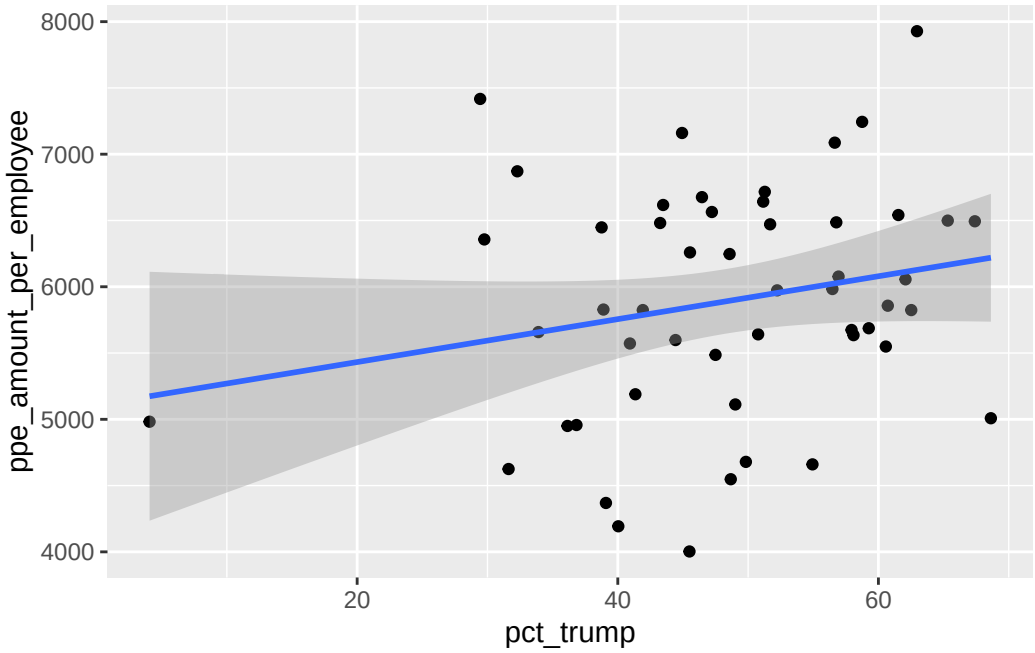


It's a bit messy, but we can see something of a pattern here in the blob of dots. Generally, the dots are moving from the lower left (less Trumpy states that got loans at a lower rate) to upper right (red states that got loans at a higher rate). The blue "line of best fit" shows the general direction of the relationship.

Let's test another variable, the average amount of money provided by PPP per small business employee in the state.

```
reveal_data |>
  ggplot() +
  geom_point(aes(x=pct_trump,y=ppe_amount_per_employee)) +
  geom_smooth(aes(x=pct_trump,y=ppe_amount_per_employee), method="lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



This one is a bit messier. There may be a slight upward slope in this blob of dots, but it's not quite as apparent. It seems less certain that there's a relationship between these two variables.

We can be a bit more precise by calculating a statistic called the correlation coefficient, also called "r". r is a value between 1 and -1. An r of 1 indicates a strong positive correlation.

An increase in air temperature and air conditioning use at home is strongly-positively correlated: the hotter it gets, the more we have to use air conditioning. If we were to plot those two variables, we might not get 1, but we'd get close to it.

An r of -1 indicates a strong negative correlation. An increase in temperature and home heating use is strongly negatively correlated: the hotter it gets, the less heat we use indoors. We might not hit -1, but we'd probably get close to it.

A correlation of 0 indicates no relationship.

All r values will fall somewhere on this scale, and how to interpret them isn't always straightforward. They're best used to give general guidance when exploring patterns.

We can calculate r with a function from the corrr package called "correlate()". First, we remove the non-numeric values from our reveal_data (state name and a binary vote_2016 column), then we correlate.

```
reveal_data |>
  select(-state_name, -vote_2016) |>
  correlate() |>
  select(term, pct_trump)
```

```
Correlation computed with
* Method: 'pearson'
* Missing treated using: 'pairwise.complete.obs'
```

```
# A tibble: 3 x 2
  term                          pct_trump
  <chr>                             <dbl>
1 pct_trump                        NA
2 businesses_receiving_ppe_pct      0.522
3 ppe_amount_per_employee           0.221
```

```
#glimpse(reveal_data)
```

The table this function produces generally confirms our interpretation of the two graphs above. The relationship between a state's pct_trump and ppe_amount_per employee is positive, but at .22 (on a scale of -1 to 1), the relationship isn't particularly strong. That's why the second graphic above was messier than the first.

The relationship between businesses in a state receiving ppe and the state's Trump vote is a bit stronger, if still moderate, .52 (on a scale of -1 to 1). Is this finding statistically valid? We can get a general sense of that by calculating the p-value of this correlation, a test of statistical significance. For that, we can use the cor.test function.

```
cor.test(reveal_data$pct_trump, reveal_data$businesses_receiving_ppe_pct)
```

```
    Pearson's product-moment correlation

data:  reveal_data$pct_trump and reveal_data$businesses_receiving_ppe_pct
t = 4.2818, df = 49, p-value = 8.607e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.2875734 0.6971386
sample estimates:
      cor
0.5218038
```

This output is quite a bit uglier, but for our purposes there are two key pieces of information from this chunk of unfamiliar words. First, it shows the correlation calculated above: r 0.5218. Two, it shows the p-value, which is 0.00008607. That's very low, as far as p-values go, which indicates that there's a very slim chance that our finding is a statistical aberration.

Now let's test the other one, the relationship between the pct_trump and the ppe_amount_per_employee.

```
cor.test(reveal_data$pct_trump, reveal_data$ppe_amount_per_employee)
```

```
    Pearson's product-moment correlation

data:  reveal_data$pct_trump and reveal_data$ppe_amount_per_employee
t = 1.5872, df = 49, p-value = 0.1189
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.05798429  0.46818515
sample estimates:
     cor
0.221133
```

Again, it shows our r value of .22, which was weaker. And the p-value here is a much larger 0.12. That indicates a higher chance of our finding being a statistical aberration, high enough that I wouldn't rely on its validity.

$p < .05$ is accepted in many scientific disciplines – and by many data journalists – as the cutoff for statistical significance. But there's heated debate about that level, and some academics question whether p-values should be relied on so heavily.

And to be clear, a low p-value does not prove that we've found what we set out to find. There's nothing on this graph or in the regression model output that proves that Trump's administration tipped the scales in favor of states that voted for it. It's entirely possible that there's some other variable – or variables – not considered here that explain this pattern.

All we know is that we've identified a potentially promising pattern, worthy of additional reporting and analysis to flesh out.

29 T-tests

Let's suppose we want to ask a related set of questions: did Trump states get higher ppp loan amounts per employee than states won by Clinton? Or did a larger percentage of businesses in states won by Trump receive, on average, a higher rate of PPP loans on average than states won by Clinton.

We can do this because, in our data, we have a column with two possible categorical values, Clinton or Trump, for each state.

We could just calculate the averages like we're used to doing.

```
reveal_data |>
  group_by(vote_2016) |>
  summarise(
    mean_ppp_amount_per_employee = mean(ppe_amount_per_employee),
    mean_businesses_receiving_ppe_pct = mean(businesses_receiving_ppe_pct)
  )
```

```
# A tibble: 2 x 3
  vote_2016 mean_ppp_amount_per_employee mean_businesses_receiving_ppe_pct
  <chr>                            <dbl>                              <dbl>
1 Clinton                          5704.                               28.2
2 Trump                            6021.                               37.2
```

Examining this, it appears that in both categories there's a difference.

The average amount of ppp loans per employee in Clinton states is smaller than Trump states (6,000 to 5,700). And the average percentage of businesses that got loans in Trump states was larger – 37% – than Clinton states – 28%. Should we report these as meaningful findings?

A t-test can help us answer that question. It can tell us where there's a statistically significant difference between the means of two groups. Have we found a real difference, or have we chanced upon a statistical aberration? Let's see by calculating it for the average loan amount.

```
t.test(ppe_amount_per_employee ~ vote_2016, data = reveal_data)
```

```
    Welch Two Sample t-test

data:  ppe_amount_per_employee by vote_2016
t = -1.2223, df = 36.089, p-value = 0.2295
alternative hypothesis: true difference in means between group Clinton and group Trump is not
95 percent confidence interval:
 -843.7901  209.1329
sample estimates:
mean in group Clinton   mean in group Trump
             5703.571              6020.900
```

We see our two means, for Trump and Clinton, the same as we calculated above. The t-value is approximately 1, the p-value here is .2295, both of which should which should give us pause that we've identified something meaningful. More on t-tests here

Let's try the percentage of businesses getting ppp loans.

```
t.test(businesses_receiving_ppe_pct ~ vote_2016, data = reveal_data)
```

```
    Welch Two Sample t-test

data:  businesses_receiving_ppe_pct by vote_2016
t = -3.182, df = 45.266, p-value = 0.002643
alternative hypothesis: true difference in means between group Clinton and group Trump is not
95 percent confidence interval:
 -14.68807  -3.30241
sample estimates:
mean in group Clinton   mean in group Trump
             28.23810              37.23333
```

This is a bit more promising. T is much stronger – about 3 – and the p-value is .002. Both of these should give us assurance that we've found something statistically meaningful. Again, this doesn't prove that Trump is stacking the deck for states. It just suggests there's a pattern worth following up on.

30 Writing with numbers

The number one sin of all early career data journalist is to get really, really, really attached to the analysis you've done and include every number you find.

Don't do that.

Numbers tell you what. Numbers rarely tell you why. What question has driven most people since they were three years old? Why. The very first thing to do is realize that is the purpose of reporting. You've done the analysis to determine the what. Now go do the reporting to do the why. Or as an old editor of mine used to say "Now go do that reporting shit you do."

The trick to writing a numbers story is to frame your story around people. Sometimes, your lead can be a number, if that number is compelling. Often, your lead is a person, a person who is one of the numbers you are writing about.

Tell their story. Briefly. Then, let us hear from them. Let them speak about what it is you are writing about.

Then come the numbers.

30.1 How to write about numbers without overwhelming with numbers.

Writing complex stories is often a battle against that complexity. You don't want to overwhelm. You want to simplify where you can. The first place you can do that is only use exact numbers where an exact number is called for.

Where you can, do the following:

- Using ratios instead of percents
- Often, it's better to put it in counts of 10. 6 of 10, 4 of 10. It's easy to translate that from a percentage to a ratio.
- But be careful when your number is 45 percent. Is that 4 in 10 or 5 in 10?
- If a ratio doesn't make sense, round. There's 287,401 people in Lincoln, according to the Census Bureau. It's easier, and no less accurate, to say there's more than 287,000 people in Lincoln.

A critical question your writing should answer: As compared to what?

How does this compare to the average? The state? The nation? The top? The bottom?

One of the most damning numbers in the series of stories Craig Pittman and I wrote that became the book Paving Paradise was comparing approvals and denials.

We were looking at the US Army Corps of Engineers and their permitting program. We were able to get a dataset of just a few years of permits that was relatively clean. From that, we were able to count the number of times the corps had said yes to a developer to wipe out wetlands the law protected and how many times they said no.

They said yes 12,000 times. They said no once.

That one time? Someone wanted to build an eco-lodge in the Everglades. Literally. Almost every acre of the property was wetlands. So in order to build it, the developer would have to fill in the very thing they were going to try to bring people into. The corps said no.

30.2 When exact numbers matter

Sometimes ratios and rounding are not appropriate.

This is being written in the days of the coronavirus. Case counts are where an exact number is called for. You don't say that there are more than 70 cases in Lancaster County on the day this was written. You specify. It's 75.

You don't say almost 30 deaths. It's 28.

Where this also comes into play is any time there are deaths: Do not round bodies.

30.3 An example

Read this story from USA Today and the Arizona Republic. Notice first that the top sets up a conflict: People say one thing, and that thing is not true.

> No one could have anticipated such a catastrophe, people said. The fire's speed was unprecedented, the ferocity unimaginable, the devastation unpredictable.
>
> Those declarations were simply untrue. Though the toll may be impossible to predict, worst-case fires are a historic and inevitable fact.

The first voice you hear? An expert who studies wildfires.

> Phillip Levin, a researcher at the University of Washington and lead scientist for the Nature Conservancy in Washington, puts it this way: "Fire is natural. But the disaster happens because people didn't know to leave, or couldn't leave. It didn't have to happen."

Then notice how they take what is a complex analysis using geographic information systems, raster analysis, the merging of multiple different datasets together and show that it's quite simple – the averaging together of pixels on a 1-5 scale.

Then, the compare what they found to a truly massive fire: The Paradise fire that burned 19,000 structures.

> Across the West, 526 small communities — more than 10 percent of all places — rank higher.

And that is how it's done. Simplify, round, ratios: simple metrics, powerful results.