

Tidyverse

Skills for Data Science

in R

Carrie Wright

* Shannon E. Ellis

Stephanie C. Hicks

* Roger D. Peng



Tidyverse Skills for Data Science in R

Carrie Wright, Shannon Ellis, Stephanie Hicks and Roger D. Peng

This book is for sale at <http://leanpub.com/tidyverseskillsdascience>

This version was published on 2021-09-03



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 - 2021 Carrie Wright, Shannon Ellis, Stephanie Hicks and Roger D. Peng

Also By These Authors

Books by Roger D. Peng

[R Programming for Data Science](#)

[The Art of Data Science](#)

[Exploratory Data Analysis with R](#)

[Executive Data Science](#)

[Report Writing for Data Science in R](#)

[Advanced Statistical Computing](#)

[The Data Science Salon](#)

[Conversations On Data Science](#)

[Mastering Software Development in R](#)

[Essays on Data Analysis](#)

Books by Shannon Ellis

[How To Develop A Leanpub Course With R](#)

Contents

1.	Introduction to the Tidyverse	1
	About This Course	2
	Tidy Data	2
	From Non-Tidy -> Tidy	18
	The Data Science Life Cycle	22
	The Tidyverse Ecosystem	23
	Data Science Project Organization	32
	Data Science Workflows	47
	Case Studies	47
2.	Importing Data in the Tidyverse	50
	About This Course	50
	Tibbles	50
	Spreadsheets	56
	CSVs	67
	TSVs	72
	Delimited Files	72
	Exporting Data from R	73
	JSON	74
	XML	78
	Databases	79
	Web Scraping	99
	APIs	106
	Foreign Formats	118
	Images	119
	googledrive	125
	Case Studies	127
3.	Wrangling Data in the Tidyverse	144

CONTENTS

About This Course	144
Tidy Data Review	145
Reshaping Data	145
Data Wrangling	156
Working With Factors	204
Working With Dates and Times	218
Working With Strings	227
Working With Text	256
Functional Programming	268
Exploratory Data Analysis	278
Case Studies	291
4. Visualizing Data in the Tidyverse	328
About This Course	328
Data Visualization Background	328
Plot Types	331
Making Good Plots	340
Plot Generation Process	350
ggplot2: Basics	352
ggplot2: Customization	386
Tables	443
ggplot2: Extensions	450
Case Studies	505
5. Modeling Data in the Tidyverse	552
About This Course	552
The Purpose of Data Science	552
Types of Data Science Questions	553
Data Needs	554
Descriptive and Exploratory Analysis	559
Inference	596
Linear Modeling	607
Multiple Linear Regression	640
Beyond Linear Regression	645
More Statistical Tests	650
Hypothesis Testing	650
Prediction Modeling	655
The <code>tidymodels</code> Ecosystem	676

CONTENTS

Case Studies	713
Summary of <code>tidymodels</code>	771
About the Authors	774

1. Introduction to the Tidyverse

The **data science life cycle** begins with a question that can be answered with data and ends with an answer to that question. However, there are a lot of steps that happen after a question has been generated and before arriving at an answer. After generating their specific question, data scientists have to determine what data will be useful, import the data, tidy the data into a format that is easy to work with, explore the data, generate insightful visualizations, carry out the analysis, and communicate their findings. Throughout this process, it is often said that [50-80% of a data scientist's time is spent wrangling data](#). It can be hard work to read the data in and get data into the format you need to ultimately answer the question. As a result, conceptual frameworks and software packages to make these steps easier have been developed.

Within the R community, R packages that have been developed for this very purpose are often referred to as the **Tidyverse**. According to their website, the [tidyverse](#) is “an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.” There are currently about a dozen packages that make up the official tidyverse; however, there are dozens of tidyverse-adjacent packages that follow this philosophy, grammar, and data structures and work well with the official tidyverse packages. It is this whole set of packages that we have set out to teach in this book.

In this course, we set out to introduce the conceptual framework behind tidy data and introduce the tidyverse and tidyverse-adjacent packages that we’ll be teaching throughout this book. Mastery of these fundamental concepts and familiarity with what can be accomplished using the tidyverse will be critical throughout the more technical courses ahead. So, be sure you are familiar with the vocabulary provided and have a clear understanding of the tidy data principles introduced here before moving forward.

In this book we assume familiarity with the R programming language. If you are not yet familiar with R, we suggest you first complete [R Programming](#) before returning to complete this book. However, if you have some familiarity with R and want to learn how to work more efficiently with data, then you’ve come to the right place!

About This Course

This course introduces a powerful set of data science tools known as the Tidyverse. The Tidyverse has revolutionized the way in which data scientists do almost every aspect of their job. We will cover the simple idea of “tidy data” and how this idea serves to organize data for analysis and modeling. We will also cover how non-tidy data can be transformed to tidy data, the data science project life cycle, and the ecosystem of Tidyverse R packages that can be used to execute a data science project.

If you are new to data science, the Tidyverse ecosystem of R packages is an excellent way to learn the different aspects of the data science pipeline, from importing the data, tidying the data into a format that is easy to work with, exploring and visualizing the data, and fitting machine learning models. If you are already experienced in data science, the Tidyverse provides a power system for streamlining your workflow in a coherent manner that can easily connect with other data science tools.

In this course it is important that you be familiar with the R programming language. If you are not yet familiar with R, we suggest you first complete [R Programming](#) before returning to complete this course.

Tidy Data

Before we can discuss all the ways in which R makes it easy to work with tidy data, we have to first be sure we know what tidy data are. Tidy datasets, by design, are easier to manipulate, model, and visualize because the tidy data principles that we’ll discuss in this course impose a general framework and a consistent set of rules on data. In fact, a well-known quote from Hadley Wickham is that “tidy datasets are all alike but every messy dataset is messy in its own way.” Utilizing a consistent tidy data format allows for tools to be built that work well within this framework, ultimately simplifying the data wrangling, visualization, and analysis processes. By starting with data that are already in a tidy format *or* by spending the time at the beginning of a project to get data into a tidy format, the remaining steps of your data science project will be easier.

Data Terminology

Before we move on, let’s discuss what is meant by **dataset**, **observations**, **variables**, and **types**, all of which are used to explain the principles of tidy data.

Dataset

A *dataset* is a collection of values. These are often numbers and strings, and are stored in a variety of ways. However, every value in a dataset belongs to a *variable* and an *observation*.

Variables

Variables in a dataset are the different categories of data that will be collected. They are the different pieces of information that can be collected or measured on each observation. Here, we see there are 7 different variables: ID, LastName, FirstName, Sex, City, State, and Occupation. The names for variables are put in the first row of the spreadsheet.

There are 7 different **variables** in this spreadsheet.

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher

Variables

Observations

The measurements taken from a person for each variable are called *observations*. Observations in a tidy dataset are stored in a single row, with each observation being put in the appropriate column for each variable.

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher

For each variable, we see there
are 4 different observations.

Observations

Types

Often, data are collected for the same individuals from multiple sources. For example, when you go to the doctor's office, you fill out a survey about yourself. That would count as one type of data. The measurements a doctor collects at your visit, however, would be a different type of data.

Demographic Survey Data

Two different types of data

Doctor's Office Measurements Data

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Height_inches	Weight_lbs	Insulin	Glucose
2	1004	Smith	Jane	65	180	0.60	163
3	4587	Nayef	Mohammed	75	215	1.46	150
4	1727	Doe	Janice	62	124	0.72	177
5	6879	Jordan	Alex	77	160	1.23	205

Types

Principles of Tidy Data

In Hadley Wickham's 2014 paper titled "[Tidy Data](#)", he explains:

Tidy datasets are easy to manipulate, model and visualize, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table.

These points are known as the tidy data principles. Here, we'll break down each one to ensure that we are all on the same page going forward.

1. Each variable you measure should be in a single column

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher



Principle #1 of Tidy Data

2. Every observation of a variable should be in a different row

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher



Principle #2 of Tidy Data

3. There should be one spreadsheet for each type of data

Demographic Survey Data

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher

Doctor's Office Measurements Data

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Height_inches	Weight_lbs	Insulin	Glucose
2	1004	Smith	Jane	65	180	0.60	163
3	4587	Nayef	Mohammed	75	215	1.46	150
4	1727	Doe	Janice	62	124	0.72	177
5	6879	Jordan	Alex	77	160	1.23	205

Principle #3 of Tidy Data

4. If you have multiple spreadsheets, they should include a column in each spreadsheet with the same column label that **allows them to be joined or merged**

Demographic Survey Data

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher

Doctor's Office Measurements Data

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Height_inches	Weight_lbs	Insulin	Glucose
2	1004	Smith	Jane	65	180	0.60	163
3	4587	Nayef	Mohammed	75	215	1.46	150
4	1727	Doe	Janice	62	124	0.72	177
5	6879	Jordan	Alex	77	160	1.23	205

Principle #4 of Tidy Data

Tidy Data Are Rectangular

When it comes to thinking about tidy data, remember that tidy data are rectangular data. The data should be a rectangle with each variable in a separate column and each entry in a different row. All cells should contain some text, so that the spreadsheet looks like a rectangle with something in every cell.

Tidy data = rectangular data

A

	A	B	C	D	E
1	id	sex	glucose	insulin	triglyc
2	101	Male	134.1	0.60	273.4
3	102	Female	120.0	1.18	243.6
4	103	Male	124.8	1.23	297.6
5	104	Male	83.1	1.16	142.4
6	105	Male	105.2	0.73	215.7

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints* 5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

Tidy Data = rectangular data

So, if you're working with a dataset and attempting to tidy it, if you don't have a rectangle at the end of the process, you likely have more work to do before it's truly in a tidy data format.

Tidy Data Benefits

There are a number of benefits to working within a tidy data framework:

1. Tidy data have a *consistent data structure* - This eliminates the *many* different ways in which data can be stored. By imposing a uniform data structure, the cognitive load imposed on the analyst is minimized for each new project.
2. Tidy data *foster tool development* - Software that all work within the tidy data framework can all work well with one another, even when developed by different individuals, ultimately increasing the variety and scope of tools available, without requiring analysts to learn an entirely new mental model with each new tool.
3. Tidy data require only a *small set of tools to be learned* - When using a consistent data format, only a small set of tools is required and these tools can be reused from one project to the next.

4. Tidy data allow for *datasets to be combined* - Data are often stored in multiple tables or in different locations. By getting each table into a tidy format, combining across tables or sources becomes trivial.

Rules for Storing Tidy Data

In addition to the four tidy data principles, there are a number of rules to follow when entering data to be stored, or when re-organizing untidy data that you have already been given for a project into a tidy format. They are rules that will help make data analysis and visualization easier down the road. They were formalized in a paper called “[Data organization in spreadsheets](#)”, written by two prominent data scientists, [Karl Broman](#) and [Kara Woo](#). In this paper, in addition to ensuring that the data are tidy, they suggest following these guidelines when entering data into spreadsheets:

1. Be consistent
2. Choose good names for things
3. Write dates as YYYY-MM-DD
4. No empty cells
5. Put just one thing in a cell
6. Don’t use font color or highlighting as data
7. Save the data as plain text files

We’ll go through each of these to make sure we’re all clear on what a great tidy spreadsheet looks like.

Be consistent

Being consistent in data entry and throughout an analysis is key. It minimizes confusion and makes analysis simpler. For example, here we see sex is coded as “female” or “male.” Those are the only two ways in which sex was entered into the data. This is an example of consistent data entry. You want to avoid sometimes coding a female’s sex as “female” and then entering it as “F” in other cases. Simply, you want to pick a way to code sex and stick to it.

With regard to entering a person’s sex, we were talking about how to code observations for a specific variable; however, consistency also matters when you’re choosing how to name a variable. If you use the variable name “ID” in one spreadsheet, use the same variable name

("ID") in the next spreadsheet. Do not change it to "id" (capitalization matters!) or "identifier" or anything else in the next spreadsheet. Be consistent!

Consistency matters across every step of the analysis. Name your files in a consistent format. Always code dates in a consistent format (discussed further below). Avoid extra spaces in cells. Being careful and consistent in data entry will be incredibly helpful when you get to the point of analyzing your data.

1. Be Consistent!

	A	B	C	D	E	F	G
1	ID	lastName	FirstName	Sex	City	State	Occupation
2	1004	Smith	Jane	female	Frederick	MD	Welder
3	4587	Nayef	Mohammed	male	Upper Darby	PA	Nurse
4	1727	Doe	Janice	female	San Diego	CA	Doctor
5	6879	Jordan	Alex	male	Birmingham	AL	Teacher

Keep exactly the same variable names across spreadsheets.

In these data, sex is always specified as "female" or "male." Pick a way to code your variables and stick to it.

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints* 5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

Be Consistent!

Choose good names for things

Choosing good variable names is important. Generally, avoid spaces in variable names and file names. You'll see why this is important as we learn more about programming, but for now, know that "Doctor Visit 1" is not a good file name. "doctor_visit_v1" is much better. Stick to using underscores instead of spaces or any other symbol when possible. The same thing goes for variable names. "FirstName" is a good variable name while "First Name" with a space in the middle of it is not.

Additionally, make sure that file and variable names are as short as possible while still being meaningful. "F1" is short, but it doesn't really tell you anything about what is in that file. "doctor_visit_v1" is a more meaningful file name. We know now that this spreadsheet

contains information about a doctor's visit. 'v1' specifies version 1 allowing for updates to this file later which would create a new file "doctor_visit_v2."

2. Choose good names for things

	Do this...	Not This!
Avoid Extra Spaces	'male'	'male '
Use underscores not spaces	doctor_visit_v1	Doctor Visit 1
Choose meaningful names	doctor_visit_v1	"F1"

○

Choose good names

Write dates as YYYY-MM-DD

When entering dates, there is a global [ISO 8601](#) standard. Dates should be encoded YYYY-MM-DD. For example if you want to specify that a measurement was taken on February 27th, 2018, you would type 2018-02-27. YYYY refers to the year, 2018. MM refers to the month of February, 02. And DD refers to the day of the month, 27. This standard is used for dates for two main reason. First, it avoids confusion when sharing data across different countries, where date conventions can differ. By all using ISO 8601 standard conventions, there is less room for error in interpretation of dates. Secondly, spreadsheet software often mishandles dates and assumes that non-date information are actually dates and vice versa. By encoding dates as YYYY-MM-DD, this confusion is minimized.

3. Write dates as YYYY-MM-DD

	Do this...	Not This!
Use 'ISO 8601' standard	2018-02-27	2/27 or 2_27_2018 or Feb 27

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints* 5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

YYYY-MM-DD

No empty cells

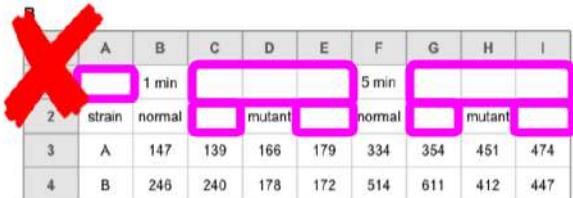
Simply, fill in every cell. If the data is unknown for that cell, put NA. Without information in each cell, the analyst is often left guessing. In the spreadsheets below, on the left, is the analyst to assume that the empty cells should use the date from the cell above? Or are we to assume that the date for that measurement is unknown? Fill in the date if it is known or type 'NA' if it is not. That will clear up the need for any guessing on behalf of the analyst. On the spreadsheet to the right, the first two rows have a lot of empty cells. This is problematic for the analysis. This spreadsheet does not follow the rules for tidy data. There is not a single variable per column with a single entry per row. These data would have to be reformatted before they could be used in analysis.

4. No empty cells

A



	A	B	C
1	id	date	glucose
2	101	2015-06-14	149.3
3	102		95.3
4	103	2015-06-18	97.5
5	104		117.0
6	105		108.0
7	106	2015-06-20	149.0
8	107		169.4



	A	B	C	D	E	F	G	H	I
1			1 min			5 min			
2	strain	normal		mutant		normal		mutant	
3	A	147	139	166	179	334	354	451	474
4	B	246	240	178	172	514	611	412	447

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints* 5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

No empty cells

Put just one thing in a cell

Sometimes people are tempted to include a number and a unit in a single cell. For weight, someone may *want* to put ‘165 lbs’ in that cell. Avoid this temptation! Keep numbers and units separate. In this case, put one piece of information in the cell (the person’s weight) and either put the unit in a separate column, or better yet, make the variable name `weight_lbs`. That clears everything up for the analyst and avoids a number and a unit from both being put in a single cell. As analysts, we prefer weight information to be in number form if we want to make calculations or figures. This is facilitated by the first column called “Weight_lbs” because it will be read into R as a numeric object. The second column called “Weight”, however, will be read into R as a character object because of the “lbs”, which makes our desired tasks more difficult.

5. Put just one thing in a cell



	A	B	C
1	Weight_lbs		
2	180		180 lbs
3	215		215 lbs
4	124		124 lbs
-			

One thing per cell

Don't use font color or highlighting as data

Avoid the temptation to highlight particular cells with a color to specify something about the data. Instead, add another column to convey that information. In the example below, 1.1 looks like an incorrect value for an individual's glucose measure. Instead of highlighting the value in red, create a new variable. Here, on the right, this column has been named 'outlier'. Including 'TRUE' for this individual suggests that this individual may be an outlier to the data analyst. Doing it in this way ensures that this information will not be lost. Using font color or highlighting however can easily be lost in data processing, as you will see in future lessons.

6. Don't use font color or highlighting as data

A

	A	B	C
1	id	date	glucose
2	101	2015-06-14	149.3
3	102	2015-06-14	95.3
4	103	2015-06-18	97.5
5	104	2015-06-18	1.1
6	105	2015-06-18	108.0
7	106	2015-06-20	149.0
8	107	2015-06-20	169.4

B

	A	B	C	D
1	id	date	glucose	outlier
2	101	2015-06-14	149.3	FALSE
3	102	2015-06-14	95.3	FALSE
4	103	2015-06-18	97.5	FALSE
5	104	2015-06-18	1.1	TRUE
6	105	2015-06-18	108.0	FALSE
7	106	2015-06-20	149.0	FALSE
8	107	2015-06-20	169.4	FALSE

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints*5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

No highlighting or font color

Save the data as plain text files

The following lessons will go into detail about which file formats are ideal for saving data, such as text files (.txt) and comma-delimited files (.csv). These file formats can easily be opened and will never require special software, ensuring that they will be usable no matter what computer or analyst is looking at the data.

Summary

The data entry guidelines discussed in and a few additional rules have been summarized below and are [available online for reference](#).

When...	Be sure to...	So Do this...	Avoid this...	Why?
Naming variables (aka assigning column headers)	Use meaningful variable names	'AgeAtDiagnosis'	'ADx'	'ADx' is an unclear and uninformative abbreviation
Naming variables	Avoid spacing in column headers	'AgeAtDiagnosis'	'Age A t Diagnosis'	Spacing in variable names makes the analyst's life more difficult
Naming variables	Use consistent capitalization	'AgeAtDiagnosis'	Using both 'AgeAtDiagnosis' and 'ageatdiagnosis'	Using consistent column names across tables/spreadsheets simplifies any merging the statistician may have to do.
Naming variables	Avoid using separators, but if it's necessary, use an underscore ('_')	'IGF1' (or 'IGF_1')	'IGF,1', 'IGF-1', 'IGF/1', 'IGF,1'	Separators (commas, periods, hyphens, slashes, spaces etc.) often have different meanings in coding languages than they do in text. Avoiding them avoids error.
Coding variables	Avoid unnecessary spaces	'male'	'male'	That extra space after 'male' makes it different from 'male' without a space.
Coding variables	Be consistent!	'male'	'Male', 'male', and 'M'	In the eyes of the statistician, 'Male', 'male', and 'M' could be incorrectly perceived as three different values.
Coding variables	Be careful of spelling errors	'male'	'males'	That extra 's' makes these two different categories.
Coding date and time	Use ISO 8601 coding	'YYYY-MM-DD'	'MM/DD/YY' and 'Month Day, Year'	Consistency simplifies the analyst's life, and YYYY-MM-DD will not be misconstrued if opened in Excel.
Coding missing data	Not leave any cells blank and use a consistent value	'NA'	'\N', 'N', red-highlighted blank cells, '...', ...	Each cell should be filled with a consistent value. Pick a way to denote missingness (ideally NA) and stick with it. Avoid using numbers or punctuation to denote missing data.
Entering data	Stick to text and numbers	Convey all information with direct text/numerical entry	Using cell color highlighting or font color to convey information	Your analyst may not use the same platform for analysis as you used for data entry, so avoiding font color and cell highlighting will minimize issues.
Generating an Excel file	Save the data in an appropriate format	Use one worksheet per table and save as CSV or text files	Multiple worksheets	Statisticians require this format to import your data onto other platforms.
Entering Data	Avoid entering unnecessary lines of text at the start	Start your first row with variable names	Adding lines of text	This violates the rules of tidy data and makes processing more difficult. Include this information in the "Code book" instead.
Opening files in Excel	Know and avoid its pitfalls	Consistently include one value per cell and be careful of date and time data.	Using macros, splitting cells, and merging cells	These formats are not amenable to data analysis on other platforms.

Ellis SE, Leek JT (2017) How to share data for collaboration. *PeerJ Preprints* 5:e3139v5 <https://doi.org/10.7287/peerj.preprints.3139v5>

Naming Guidelines

From Non-Tidy → Tidy

The reason it's important to discuss what tidy data are and what they look like is because out in the world, most data are untidy. If you are not the one entering the data but are instead handed the data from someone else to do a project, more often than not, those data will be untidy. Untidy data are often referred to simply as messy data. In order to work with these data easily, you'll have to get them into a tidy data format. This means you'll have to fully recognize untidy data and understand how to get data into a tidy format.

The following common problems seen in messy datasets again come from [Hadley Wickham's paper on tidy data](#). After briefly reviewing what each common problem is, we will then take a look at a few messy datasets. We'll finally touch on the concepts of tidying untidy data, but we won't actually do any practice *yet*. That's coming soon!

Common problems with messy datasets

1. Column headers are values but should be variable names.

2. A single column has multiple variables.
3. Variables have been entered in both rows and columns.
4. Multiple “types” of data are in the same spreadsheet.
5. A single observation is stored across multiple spreadsheets.

Examples of untidy data

To see some of these messy datasets, let's explore three different sources of messy data.

Examples from Data Organization in Spreadsheets

In each of these examples, we see the principles of tidy data being broken. Each variable is not a unique column. There are empty cells all over the place. The data are not rectangular. Data formatted in these messy ways are likely to cause problems during analysis.

A						
	A	B	C	D	E	F
1						
2		101	102	103	104	105
3	sex	Male	Female	Male	Male	Male
4						
5		101	102	103	104	105
6	glucose	134.1	120.0	124.8	83.1	105.2
7						
8		101	102	103	104	105
9	insulin	0.60	1.18	1.23	1.16	0.73

B						
	A	B	C	D	E	F
1	1MIN					
2			Normal			Mutant
3	B6	146.6	138.6	155.6	166	179.3
4	BTBR	245.7	240	243.1	177.8	171.6
5						186.9
6	5MIN					
7			Normal			Mutant
8	B6	333.6	353.6	408.8	450.6	474.4
9	BTBR	514.4	610.6	567.9	412.1	447.4
						423.8
						448.5

C						
	A	B	C	D	E	F
1						
2	Date	11/3/14				
3	Days on diet:	126				
4	Mouse #	43				
5	sex	f				
6	experiment	values		mean	SD	
7	control	0.196	0.191	1.081	0.49	0.52
8	treatment A	7.414	1.468	2.254	3.71	3.23
9	treatment B	9.811	9.259	11.298	10.12	1.05
10						
11	fold change	values		mean	SD	
12	treatment A	15.26	3.02	4.64	7.54	6.65
13	treatment B	20.19	19.05	23.24	20.83	2.17

D						
	A	B	C	D	E	F
1	GTT date	GTT weight	time	glucose mg/dl	insulin ng/ml	
2	321	2/9/15	24.5	0	99.2	lo off curve
3				5	349.3	0.205
4				15	286.1	0.129
5				30	312	0.176
6				60	99.9	0.122
7				120	217.9	0.251
8	322	2/9/15	18.9	0	185.8	0.251
9				5	297.4	2.228
10				15	439	2.078
11				30	362.3	0.775
12				60	232.7	0.5
13				120	260.7	0.523
14	323	2/9/15	24.7	0	196.5	0.191
15				5	530.6	off curve lo

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints* 5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

Examples from Data Organization in Spreadsheets

For a specific example, [Miles McBain](#), a data scientist from Brisbane, Australia set out to analyze Australian survey data on Same Sex marriage. Before he could do the analysis, however, he had a lot of tidying to do. He annotated all the ways in which the data were untidy, including the use of commas in numerical data entry, blank cells, junk at the top of

the spreadsheet, and merged cells. All of these would have stopped him from being able to analyze the data had he not taken the time to first tidy the data. Luckily, he wrote a [Medium piece](#) including all the steps he took to tidy the data.

Inspired by Miles' work, Sharla Gelfand decided to tackle a messy dataset from Toronto's open data. She similarly outlined all the ways in which the data were messy including: names and addresses across multiple cells in the spreadsheet, merged column headings, and lots of blank cells. She has also included the details of how she cleaned these data [in a blog post](#). While the details of the code may not make sense yet, it will shortly as you get more comfortable with the programming language, R.

Tidying untidy data

There are a number of actions you can take on a dataset to tidy the data depending on the problem. These include: filtering, transforming, modifying variables, aggregating the data, and sorting the order of the observations. There are functions to accomplish each of these actions in R. While we'll get to the details of the code in a few lessons, it's important at this point to be able to identify untidy data and to determine what needs to be done in order to get those data into a tidy format. Specifically, we will focus on a single messy dataset. This is dataset D from the 'Data Organization in Spreadsheets' example of messy data provided above. We note that there are blank cells and that the data are not rectangular.

D

	A	B	C	D	E	F
1		GTT date	GTT weight	time	glucose mg/dl	insulin ng/ml
2	321	2/9/15	24.5	0	99.2	lo off curve
3				5	349.3	0.205
4				15	286.1	0.129
5				30	312	0.175
6				60	99.9	0.122
7				120	217.9	lo off curve
8	322	2/9/15	18.9	0	185.8	0.251
9				5	297.4	2.228
10				15	439	2.078
11				30	362.3	0.775
12				60	232.7	0.5
13				120	260.7	0.523
14	323	2/9/15	24.7	0	198.5	0.151
15				5	530.6	off curve lo

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints* 5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

Messy dataset

To address this, these data can be split into two different spreadsheets, one for each type of data. Spreadsheet A includes information about each sample. Spreadsheet B includes measurements for each sample over time. Note that both spreadsheets have an ‘id’ column so that the data can be merged if necessary during analysis. The ‘note’ column does have some missing data. Filling in these blank cells with ‘NA’ would fully tidy these data. We note that sometimes a single spreadsheet becomes two spreadsheets during the tidying process. This is OK as long as there is a consistent variable name that links the two spreadsheets!

A				B					
	A	B	C		A	B	C	D	E
1	id	GTT date	GTT weight		1	id	GTT time	glucose mg/dl	insulin ng/ml
2	321	2/9/15	24.5		2	321	0	99.2	NA
3	322	2/9/15	18.9		3	321	5	349.3	0.205
4	323	2/9/15	24.7		4	321	15	286.1	0.129
					5	321	30	312	0.175
					6	321	60	99.9	0.122
					7	321	120	217.9	NA
					8	322	0	185.8	0.251
					9	322	5	297.4	2.228
					10	322	15	439	2.078
					11	322	30	362.3	0.775
					12	322	60	232.7	0.5
					13	322	120	260.7	0.523
					14	323	0	198.5	0.151
					15	323	5	530.6	NA
									insulin below curve

Broman KW, Woo KH. (2017) Data organization in spreadsheets. *PeerJ Preprints*5:e3183v1 <https://doi.org/10.7287/peerj.preprints.3183v1>

Tidy version of the messy dataset

The Data Science Life Cycle

Now that we have an understanding of what tidy data are, it's important to put them in context of the data science life cycle. We mentioned this briefly earlier, but the data science life cycle starts with a question and then uses data to answer that question. The focus of this book is mastering all the steps in between formulating a question and finding an answer.

Others have set out to design charts to explain all the steps in between asking and answering question. They are all similar but have different aspects of the process they highlight and/or on which they focus. These have been summarized in [A First Course on Data Science](#).

Regardless of which life cycle chart you like best, when it comes down to answering a data science question, **importing**, **tidying**, **visualizing**, and **analyzing** the data are important parts of the process. It's these four parts of the pipeline that we'll cover throughout this book.

The Tidyverse Ecosystem

With a solid understanding of tidy data and how tidy data fit into the data science life cycle, we'll take a bit of time to introduce you to the tidyverse and tidyverse-adjacent packages that we'll be teaching and using throughout this book. Taken together, these packages make up what we're referring to as the **tidyverse ecosystem**. The purpose for the rest of this course is not for you to understand *how* to use each of these packages (that's coming soon!), but rather to help you familiarize yourself with which packages fit into which part of the data science life cycle.

Note that we will describe the official tidyverse packages, as well as other packages that are tidyverse-adjacent. This means they follow the same conventions as the official tidyverse packages and work well within the tidy framework and structure of data analysis.

Reading Data into R

After identifying a question that can be answered using data, there are *many* different ways in which the data you'll want to use may be stored. Sometimes information is stored within an Excel spreadsheet. Other times, the data are in a table on a website that needs to be scraped. Or, in a CSV file. Each of these types of data files has their own structure, but R can work with all of them. To do so, however, requires becoming familiar with a few different packages. Here, we'll discuss these packages briefly. In later courses in the book we'll get into the details of what characterizes each file type and how to use each packages to read data into R.

`tibble`

While not technically a package that helps read data into R, `tibble` is a package that reimagines the familiar R `data.frame`. It is a way to store information in columns and rows, but does so in a way that addresses problems earlier in the pipeline.

From the [tibble website](#):

A tibble, or `tbl_df`, is a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not. Tibbles are `data.frames` that are lazy and surly: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). This forces you to confront problems earlier, typically leading

to cleaner, more expressive code. Tibbles also have an enhanced `print()` method which makes them easier to use with large datasets containing complex objects.

In fact, when working with data using the tidyverse, you'll get very comfortable working with tibbles.

`readr`

`readr` is a package that users of the tidyverse use all the time. It helps read rectangular data into R. If you find yourself working with CSV files frequently, then you'll find yourself using `readr` regularly.

According to the [readr website](#):

The goal of `readr` is to provide a fast and friendly way to read rectangular data (like csv, tsv, and fwf). It is designed to flexibly parse many types of data found in the wild, while still cleanly failing when data unexpectedly changes. If you are new to `readr`, the best place to start is the data import chapter in R for data science.

`googlesheets4`

`googlesheets4` is a brilliant tidyverse-adjacent package that allows users to “access and manage Google spreadsheets from R.” As more and more data is saved in the cloud (rather than on local computers), packages like `googlesheets4` become invaluable. If you store data on Google Sheets or work with people who do, this package will be a lifesaver.

`readxl`

Another package for working with tabular data is `readxl`, which is specifically designed to move data from Excel into R. If many of your data files have the .xls or .xlsx extension, familiarizing yourself with this package will be helpful.

From the [readxl website](#):

The `readxl` package makes it easy to get data out of Excel and into R. Compared to many of the existing packages (e.g. `gdata`, `xlsx`, `xlsReadWrite`) `readxl` has no external dependencies, so it's easy to install and use on all operating systems. It is designed to work with tabular data.

googledrive

Similar to `googlesheets4`, but for interacting with file on Google Drive from R (rather than just Google Sheets), the tidyverse-adjacent `googledrive` package is an important package for working with data in R.

haven

If you are a statistician (or work with statisticians), you've likely heard of the statistical packages SPSS, Stata, and SAS. Each of these has data formats for working with data that are compatible only with their platform. However, there is an R package that allows you to use `read` stored in these formats into R. For this, you'll need `haven`.

jsonlite & xml2

Data stored on and retrieved from the Internet are often stored in one of the two most common semi-structured data formats: JSON or XML. We'll discuss the details of these when we discuss how to use `jsonlite` and `xml2`, which allow data in the JSON and XML formats, respectively, to be read into R. `jsonlite` helps extensively when working with Application Programming Interfaces (APIs) and `xml2` is incredibly helpful when working with HTML.

rvest

If you are hoping to scrape data from a website, `rvest` is a package with which you'll want to become familiar. It allows you to scrape information directly from web pages on the Internet.

httr

Companies that share data with users often do so using Application Programming Interfaces or APIs. We've mentioned that these data are often stored in the JSON format, requiring packages like `jsonlite` to work with the data. However, to retrieve the data in the first place, you'll use `httr`. This package helps you interact with modern web APIs.

Data Tidying

There are *loads* of ways in which data and information are stored on computers and the Internet. We've reviewed that there are a number of packages that you'll have to use depending on the type of data you need to get into R. However, once the data are in R,

the next goal is to tidy the data. This process is often referred to as data wrangling or data tidying. Regardless of what you call it, there are a number of packages that will help you take the untidy data you just read into R and convert it into the flexible and usable tidy data format.

dplyr

The most critical package for wrangling data is `dplyr`. Its release completely transformed the way many R users write R code and work with data, greatly simplifying the process.

According to the [dplyr website](#):

`dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges.

`dplyr` is built around five primary verbs (`mutate`, `select`, `filter`, `summarize`, and `arrange`) that help make the data wrangling process simpler. This book will cover these verbs among other functionality within the `dplyr` package.

tidyverse

Like `dplyr`, `tidyverse` is a package with the primary goal of helping users take their untidy data and make it tidy.

According to the [tidyverse website](#):

The goal of `tidyverse` is to help you create tidy data. Tidy data is data where: each variable is in a column, each observation is a row, and each value is a cell. Tidy data describes a standard way of storing data that is used wherever possible throughout the `tidyverse`. If you ensure that your data is tidy, you'll spend less time fighting with the tools and more time working on your analysis.

janitor

In addition to `dplyr` and `tidyverse`, a common `tidyverse`-adjacent package used to clean dirty data and make users life easier while doing so is `janitor`.

According to the [janitor website](#):

`janitor` has simple functions for examining and cleaning dirty data. It was built with beginning and intermediate R users in mind and is optimized for user-friendliness. Advanced R users can already do everything covered here, but with `janitor` they can do it faster and save their thinking for the fun stuff.

`forcats`

R is known for its ability to work with categorical data (called factors); however, they have historically been more of a necessary evil than a joy to work with. Due to the frustratingly hard nature of working with factors in R, the `forcats` package developers set out to make working with categorical data simpler.

According to the [forcats website](#):

The goal of the `forcats` package is to provide a suite of tools that solve common problems with factors, including changing the order of levels or the values.

`stringr`

Similar to `forcats`, but for strings, the `stringr` package makes common tasks simple and streamlined. Working with this package becomes easier with some knowledge of regular expressions, which we'll cover in this book.

According to the [stringr website](#):

Strings are not glamorous, high-profile components of R, but they do play a big role in many data cleaning and preparation tasks. The `stringr` package provides a cohesive set of functions designed to make working with strings as easy as possible.

`lubridate`

The final common package dedicated to working with a specific type of variable is `lubridate` (a tidyverse-adjacent package), which makes working with dates and times simpler. Working with dates and times has historically been difficult due to the nature of our calendar, with its varying number of days per month and days per year, and due to time zones, which can make working with times infuriating. The `lubridate` developers aimed to make working with these types of data simpler.

According to the [lubridate website](#):

Date-time data can be frustrating to work with in R. R commands for date-times are generally unintuitive and change depending on the type of date-time object being used. Moreover, the methods we use with date-times must be robust to time zones, leap days, daylight savings times, and other time related quirks, and R lacks these capabilities in some situations. Lubridate makes it easier to do the things R does with date-times and possible to do the things R does not.

`glue`

The `glue` tidyverse-adjacent package makes working with interpreted string literals simpler. We'll discuss this package in detail in this specialization.

`skimr`

After you've got your data into a tidy format and all of your variable types have been cleaned, the next step is often summarizing your data. If you've used the `summary()` function in R before, you're going to love `skimr`, which summarizes entire data frames for you in a tidy manner.

According to the `skimr` tidyverse-adjacent package:

`skimr` provides a frictionless approach to summary statistics which conforms to the principle of least surprise, displaying summary statistics the user can skim quickly to understand their data.

`tidytext`

While working with factors, numbers, and small strings is common in R, longer texts have historically been analyzed using approaches outside of R. However, once the tidyverse-adjacent package `tidytext` was developed, R had a tidy approach to analyzing text data, such as novels, news stories, and speeches.

According to the `tidytext` website:

In this package, we provide functions and supporting data sets to allow conversion of text to and from tidy formats, and to switch seamlessly between tidy tools and existing text mining packages.

purrr

The final package we'll discuss here for data tidying is `purrr`, a package for working with functions and vectors in a tidy format. If you find yourself writing for loops to iterate through data frames, then `purrr` will save you a ton of time!

According to the [purrr website](#):

`purrr` enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. If you've never heard of FP before, the best place to start is the family of `map()` functions which allow you to replace many for loops with code that is both more succinct and easier to read.

Data Visualization

Data Visualization is a critical piece of any data science project. Once you have your data in a tidy format, you'll first explore your data, often generating a number of basic plots to get a better understanding of your dataset. Then, once you've carried out your analysis, creating a few detailed and well-designed visualizations to communicate your findings is necessary. Fortunately, there are a number of helpful packages to create visualizations.

ggplot2

The most critical package when it comes to plot generation in data visualization is `ggplot2`, a package that allows you to quickly create plots and meticulously customize them depending on your needs.

According to the [ggplot2 website](#):

`ggplot2` is a system for declaratively creating graphics, based on [The Grammar of Graphics](#). You provide the data, tell `ggplot2` how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

This package will be covered in a great amount of detail in this book, largely due to the fact that you'll find yourself using it all the time. Having a strong foundation of how to use `ggplot2` is incredibly important for any data science project.

kableExtra

While the ability to make beautiful and informative plots is essential, tables can be incredibly effective vehicles for the conveyance of information. Customizing plots can be done using the tidyverse-adjacent `kableExtra` package, which is built on top of the `knitr()` function from the `kable` package, which generates basic tables. `kableExtra` allows complex and detailed tables to be built using a `ggplot2`-inspired syntax.

ggrepel

Due to the popularity of `ggplot2`, there are a number of tidyverse-adjacent packages built on top of and within the `ggplot2` framework. `ggrepel` is one of these packages that “provides geoms for `ggplot2` to repel overlapping text labels.”

cowplot

`cowplot` is another tidyverse-adjacent package that helps you to polish and put finishing touches on your plots.

According to the `cowplot` developers:

The `cowplot` package provides various features that help with creating publication-quality figures, such as a set of themes, functions to align plots and arrange them into complex compound figures, and functions that make it easy to annotate plots and mix plots with images.

patchwork

The `patchwork` package (also tidyverse-adjacent) is similar to `cowplot` and is an excellent option for combining multiple plots together.

ggridge

Beyond static images, there are times when we want to display changes over time or other visualizations that require animation. The `ggridge` package (also tidyverse-adjacent) enables animation on top of `ggplot2` plots.

According to the `ggridge` website:

`ggridge` extends the grammar of graphics as implemented by `ggplot2` to include the description of animation. It does this by providing a range of new grammar classes that can be added to the plot object in order to customize how it should change with time.

Data Modeling

Once data have been read in, tidied, and explored, the last step to answering your question and before communicating your findings is data modeling. In this step, you're carrying out an analysis to answer your question of interest. There are a number of helpful suites of R packages.

The `tidymodels` ecosystem

When it comes to predictive analyses and machine learning, there is a suite of packages called `tidymodels`.

The `tidymodels` ecosystem contains packages for data splitting, preprocessing, model fitting, performance assessment, and more.

The great advantage is that it allows for users to use predictive algorithms that were written across dozens of different R packages and makes them all usable with a standard syntax.

Inferential packages: `broom` and `infer`

When carrying out inferential analyses the `tidymodels` suite of tidyverse-adjacent packages is essential.

The `broom` package which is part of the `tidymodels` suite takes statistical analysis objects from R (think the output of your `lm()` function) and converts them into a tidy data format. This makes obtaining the information you're most interested in from your statistical models much simpler.

Similarly, `infer` sets out to perform statistical inference using a standard statistical grammar. Historically, the syntax varied widely from one statistical test to the next in R. The `infer` package sets out to standardize the syntax, regardless of the test.

Tidyverts: `tsibble`, `feasts`, and `fable`

While many datasets are like a snapshot in time - survey data collected once or contact information from a business - time series data are unique. Time series datasets represent

changes over time and require computational approaches that are unique to this fact. A suite of tidyverse-adjacent packages called [tidyverts](#) have been developed to enable and simplify tidy time series analyses in R. Among these are `tsibble`, `feasts`, and `fable`.

A `tsibble` is the time-series version of a tibble in that it provides the data.frame-like structure most useful for carrying out tidy time series analyses.

According to the [tsibble website](#):

The *tsibble* package provides a data infrastructure for tidy temporal data with wrangling tools. Adhering to the tidy data principles, *tsibble* is an explicit data- and model-oriented object.

The `feasts` package is most helpful when it comes to the modeling step in time series analyses.

According to the [feasts website](#):

`Feasts` provides a collection of tools for the analysis of time series data. The package name is an acronym comprising of its key features: Feature Extraction And Statistics for Time Series.

The `fable` package is most helpful when it comes to the modeling step in forecasting analyses.

According to the [fable website](#):

The R package `fable` provides a collection of commonly used univariate and multivariate time series forecasting models including exponential smoothing via state space models and automatic ARIMA modelling. These models work within the `fable` framework, which provides the tools to evaluate, visualize, and combine models in a workflow consistent with the tidyverse.

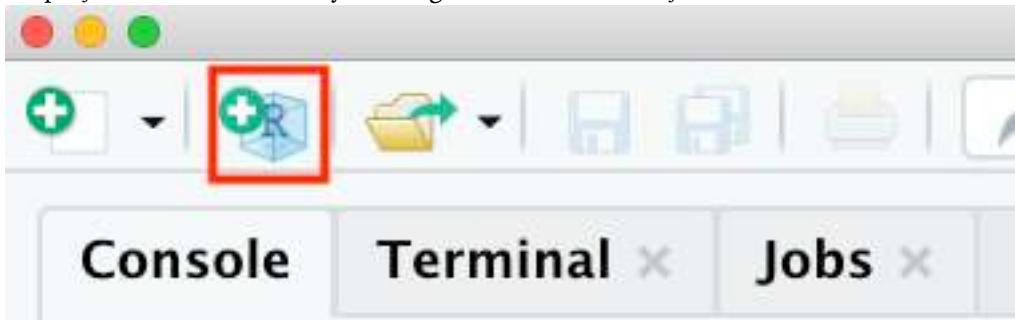
Data Science Project Organization

Data science projects vary quite a lot so it can be difficult to give universal rules for how they should be organized. However, there are a few ways to organize projects that are commonly useful. In particular, almost all projects have to deal with files of various sorts—data files, code files, output files, etc. This section talks about how files work and how projects can be organized and customized.

RStudio Projects

Creating [RStudio projects](#) can be very helpful for organizing your code and any other files that you may want to use in one location together.

New projects can be created by clicking on the RStudio Projects button in RStudio:



Creating RStudio Projects

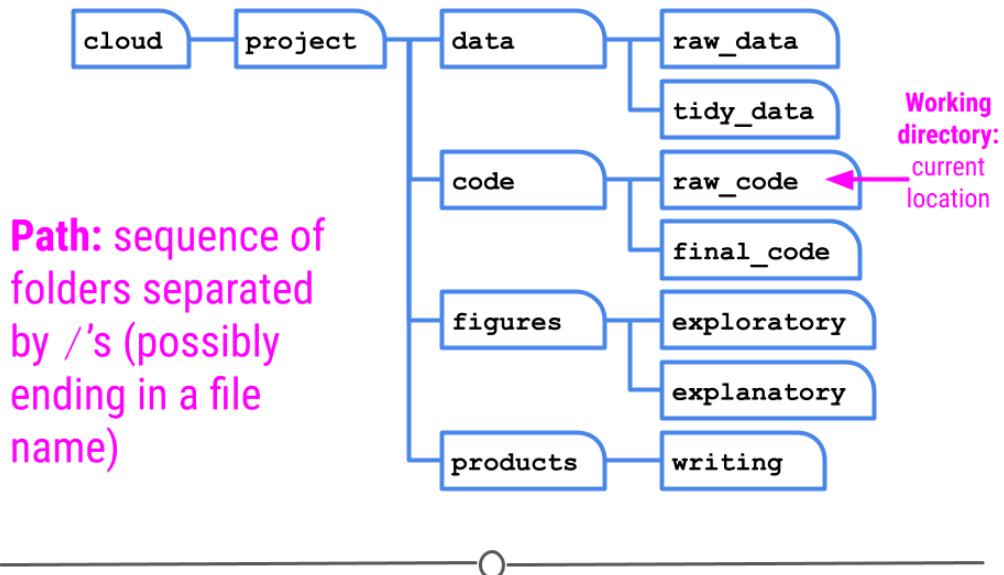
This will create a special file with the .Rproj extension. This file tells RStudio to identify the directory containing the .Rproj file as the main directory for that R Project. A new session of RStudio will be started when a user opens an R project from this main directory. The previous state including settings of that project will be maintained from one time to the next. The files that were open the last time the user worked on the project will automatically be opened again. Other packages like the `here` package will also recognize the .Rproj file to make analyses easier for the user. We will explain how this package works in the next section.

File Paths

Since we're assuming R knowledge in this course, you're likely familiar with file paths. However, this information will be critical at a number of points throughout this course, so we want to quickly review relative and absolute file paths briefly before moving on.

In future courses, whenever you write code and run commands within R to work with data, or whenever you use Git commands to document updates to your files, you will be working *in a particular location*. To know your location within a file system is to know exactly what folder you are in right now. The folder that you are in right now is called the **working directory**. Your current working directory in the Terminal may be different from your current working directory in R and may yet even be different from the folder shown in the Files pane of RStudio. The focus of this lesson is on the Terminal, so we will not discuss working directories within R until the next lesson.

Knowing your working directory is important because if you want to tell Terminal to perform some actions on files in other folders, you will need to be able to tell the Terminal where that folder is. That is, you will need to be able to specify a **path** to that folder. A path is a string that specifies a sequence of folders to traverse in order to end up at a final destination. The end of a path string may be a file name if you are creating a path to a file. If you are creating a path to a folder, the path string will end with the destination folder. In a path, folder names are separated by forward slashes /. For example, let's say that your current working directory in the Terminal is the `raw_code` directory, and you wish to navigate to the `exploratory` subfolder within the `figures` folder to see the graphics you've created.

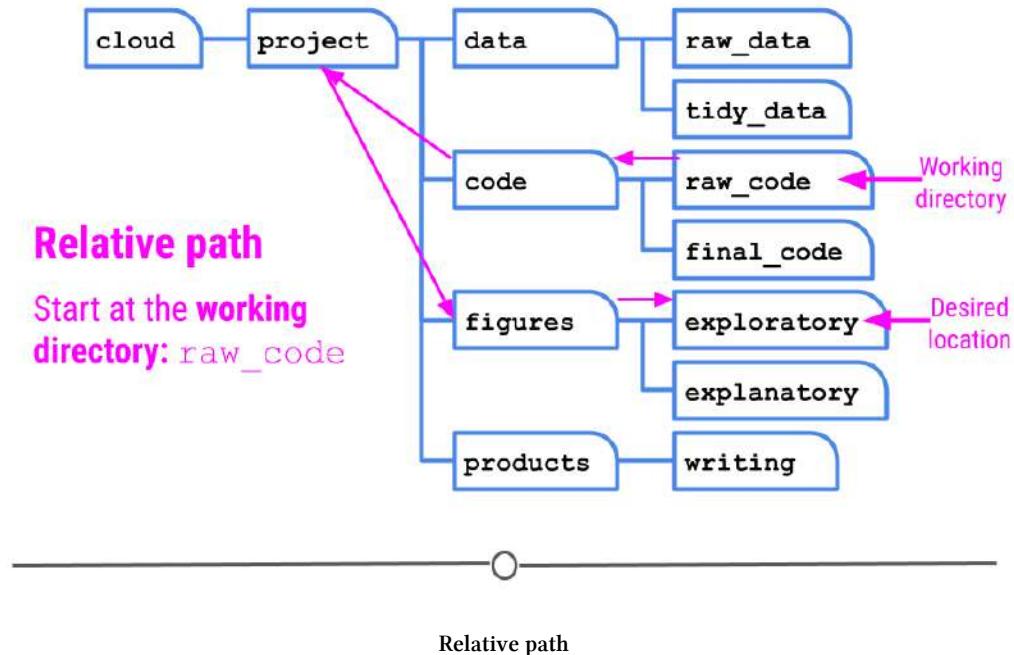


Definitions: working directory and path

There are two types of paths that can lead to that location: **relative** and **absolute**.

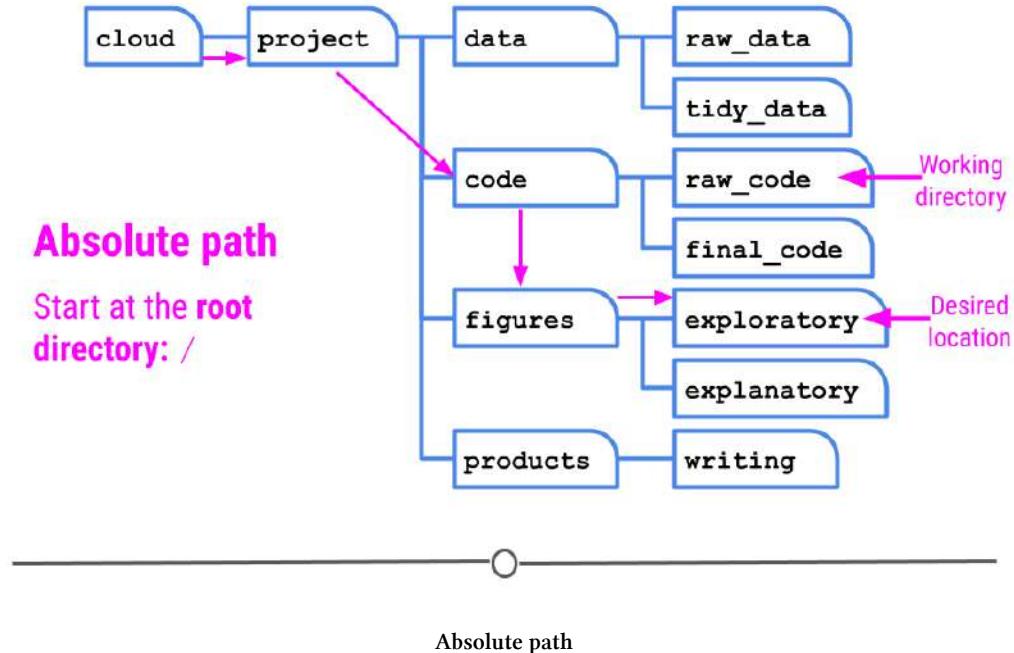
Relative Paths

The first is called a **relative path** which gives a path to the destination folder based on where you are right now (in `raw_code`).]



Absolute Paths

Alternatively, you can specify an **absolute path**. An absolute path starts at the **root directory** of a file system. The root directory does not have a name like other folders do. It is specified with a single forward slash / and is special in that it cannot be contained within other folders. In the current file system, the root directory contains a folder called `cloud`, which contains a folder called `project`.



Path Summary

Analogy: The root directory is analogous to a town square, a universal starting location. The desired destination location might be the baker. An absolute path specifies how to reach the bakery starting from this universal starting location of the town square. However, if we are in the library right now, a relative path would specify how to reach the bakery from our current location in the library. This could be pretty different than the path from the town square.

Imagine that your friend plans to go from the town square, then to the library, and finally to the bakery. In this analogy, the town square represents the root directory, a universal starting location. If your friend is currently at the library and asks you for directions, you would likely tell them how to go from the library (where they are) to the bakery (where they want to go). This represents a **relative path** – taking you from where you are currently to where you want to be. Alternatively, you could give them directions from the Town square, to the library, and then to the bakery. This represents an **absolute path**, directions that will always work in this town, no matter where you are currently, but that contain extra information given where your friend is currently.

	In the Terminal	In town
Root directory	/	Town square
Working directory	raw_code	Library
Destination	exploratory	Bakery

Absolute path: “We’re at the library but pretend you are at the town square. Turn left, then right.”

Relative path: “Turn right, then right again.”

Directions and paths analogy

To summarize once more, **relative paths** give a path to the destination folder based on where you are right now (your current working directory) while **absolute paths** give a path to the destination folder based on the root directory of a file system.

The `here` package

With most things in R, there’s a package to help minimize the pain of working with and setting file paths within an R Project - the `here` package.

As a reminder, to get started using the `here` package (or any R package!), it first has to be installed (using the `install.packages()` function) and then loaded in (using the `library()` function). Note that the package name in the `install.packages()` function has to be in quotes but for `library()` it doesn’t have to. The code to copy and paste into your R console is below:

```
install.packages("here")
library(here)
```

`here` is a package specifically designed to help you deal with file organization when you’re coding. This package allows you to define in which folder all your relative paths should

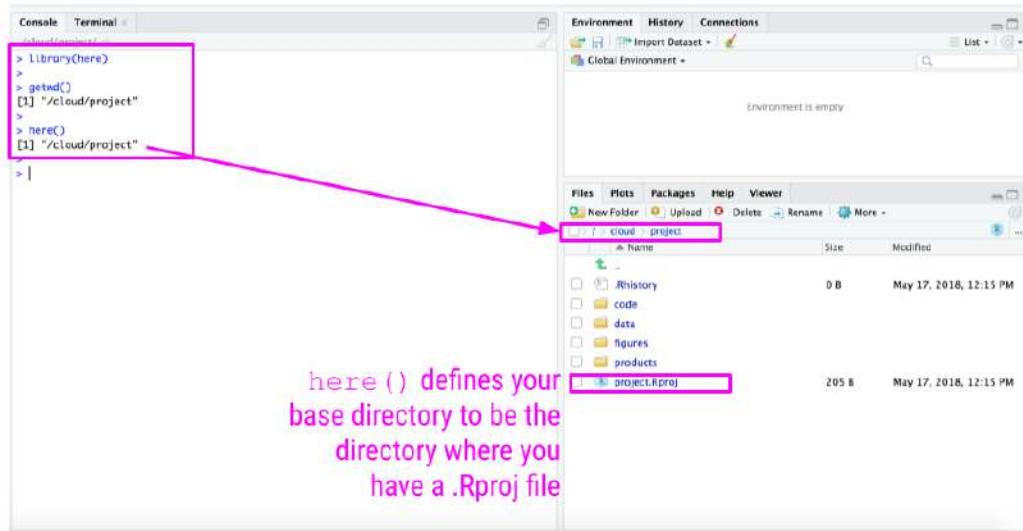
begin within a project. The path of this folder will be displayed after typing `library(here)` or `here()`.

Setting your project directory

After installing and loading the `here` package, to set your project directory using `here()`, you'll simply type the command `here()`. You'll notice that the output of `here()` and `getwd()` in this case is the same; however, what they're doing behind the scenes is different.

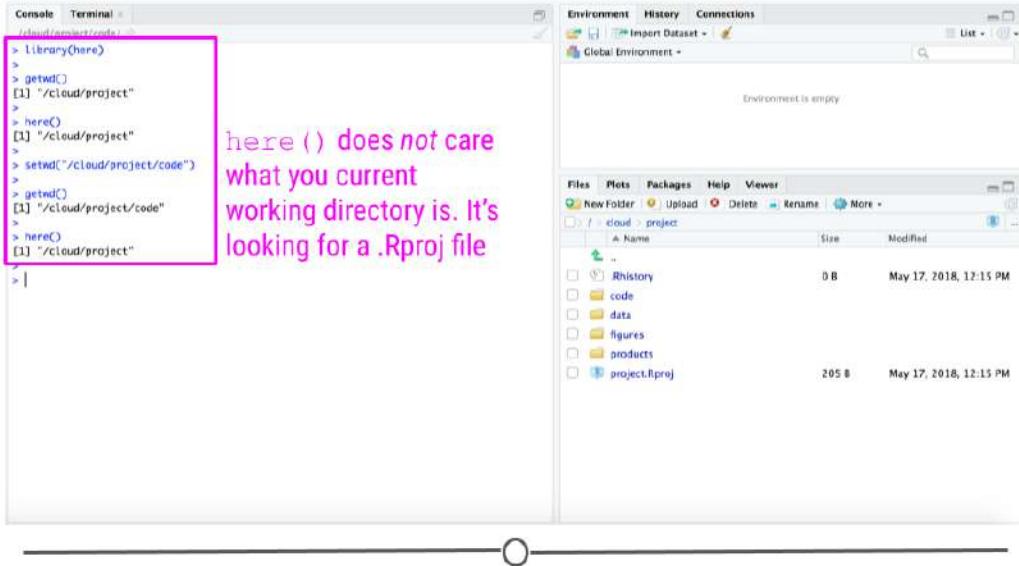
- `getwd()` - shows the directory you are in currently
- `here()` - sets the directory to use for all future relative paths

The `here()` function is what you want to use to set your project directory so that you can use it for future relative paths in your code. While in this case it *also* happened to be in the same directory you were in, it doesn't have to be this way. The `here()` function looks to see if you have a `.Rproj` file in your project. It then sets your base directory to whichever directory that file is located.



here sets your project directory for future reference using `here()`

So, if we were to change our current directory and re-type `here()` in the Console, you'll note that the output from `here()` does not change because it's still looking for the directory where `.Rproj` is.



`here()` does not care what your current working directory is

Note: In cases where there is no `.Rproj` file, `here()` will look for files other than a `.Rproj` file. You can read about those cases in the fine print [here](#). But for most of your purposes, `here()` will behave as we just discussed.

Get files paths using `here()`

After setting your project folder using `here()`, R will then know what folder to use to define any and all other paths within this project.

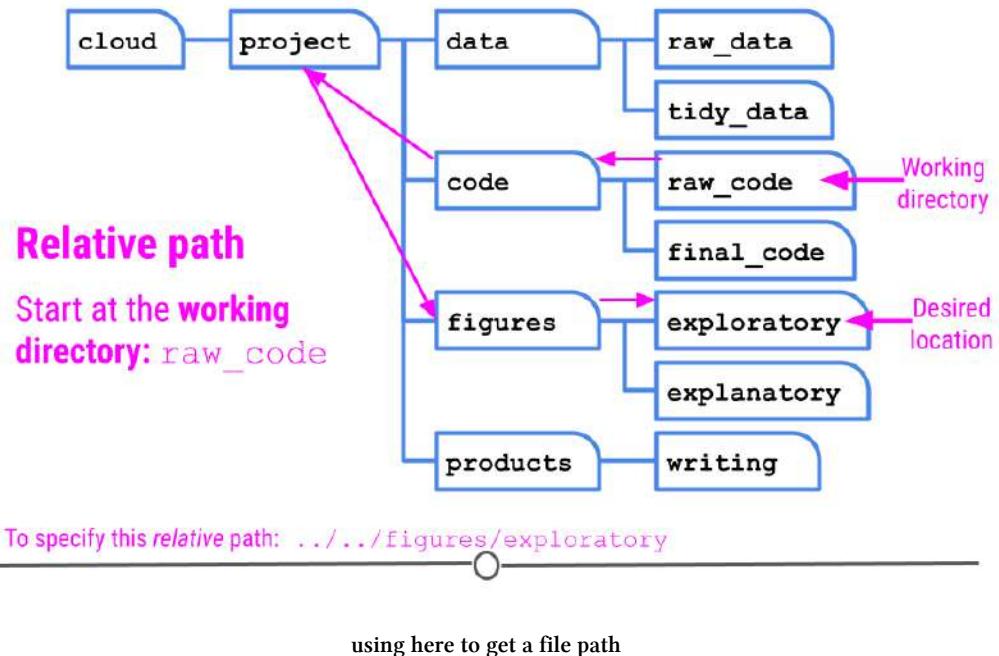
For example, if you wanted to include a path to a file named “`intro_code.R`” in your `raw_code` directory (which is in your `code` directory), you would simply specify that in your code like this:

```
here("code", "raw_code", "intro_code.R")
```

This code says start from where I've already defined the project starts (`here()`), then look in the folders `code` and then `raw_code`, for the file “`intro_code.R`.”

The syntax is simplified when using `here()`. Each subdirectory or file in the path is in quotes and simply separated by commas within the `here()` function, and voila you have the relative path you're looking for relative to `here()`.

The output from this code includes the correct file path to this file, just as you wanted!



Save and load files using `here()`

Using the `here` package, files within the project can be saved or loaded by simply typing `here` (to replace the path to the project directory) and typing any subdirectories like in this example, where we want to save data to the `raw_data` directory within the `data` directory of the project:

```
save(introcode_data_object, file = here::here("data", "raw_data", "intro_code_o\
bject.rda"))
```

Or if we want to load this data:

```
load(here::here("data", "raw_data", "intro_code_object.rda"))
```

Remember that the `::` notation indicates that we are using a function of a particular package. So the `here` package is indicated on the left of the `::` and the `here()` function is indicated on the right.

Where you should use this

You should use `here()` to set the base project directory for each data science project you do. And, you should use relative paths using `here()` throughout your code any time you want to refer to a different directory or sub-directory within your project using the syntax we just discussed.

The utility of the `here` package

You may be asking, “Why do I need the `here` package?”, and you may be thinking, “I can just write the paths myself or use the `setwd()` function”.

[Jenny Bryan](#), has several [reasons](#). We highly recommend checking them out.

In our own words:

- 1) It makes your project work more easily on someone else’s machine - there is no need to change the working directory path with the `setwd()` function. Everything will be relative to the `.Rproj` containing directory.
- 2) It allows for your project to work more easily for you on your own machine - say your working directory was set to “`/Users/somebody/2019_data_projects/that_big_project`” and you decide you want to copy your scripts to be in “`/Users/somebody/2020_data_projects/that_big_project_again`” to update a project for the next year with new data. You would need to update all of your scripts each time you set your working directory or load or save files. In some cases that will happen many times in a single project!
- 3) It saves time. Sometimes our directory structures are complicated and we can easily make a typo. The `here` package makes writing paths easier!

File Naming

Having discussed the overall file structure and the `here` package, it’s now important to spend a bit of time talking about file naming. It may not be the most *interesting* topic on its surface, but naming files well can save future you a lot of time and make collaboration with others a lot easier on everyone involved. By having descriptive and consistent file names, you’ll save yourself a lot of headaches.

Good File Names

One of the most frustrating things when looking at someone's data science project is to see a set of files that are completely scattered with names that don't make any sense. It also makes it harder to search for files if they don't have a consistent naming scheme.

One of the best organized file naming systems is due to [Jenny Bryan](#) who gives three key principles of file naming for data science projects. Files should be:

- Machine readable
- Human readable
- Nicely ordered

If your files have these three characteristics, then it will be easy for you to search for them (machine readable), easy for you to understand what is in the files (human readable), and easy for you to glance at a whole folder and understand the organization (be nicely ordered). We'll now discuss some concrete rules that will help you achieve these goals.

Machine readable files

The machine we are talking about when we say "machine readable" is a computer. To make a file machine readable means to make it easy for you to use the machine to search for it. Let's see which one of the following examples are good example of machine readable files and which are not.

- Avoid spaces: *2013 my report.md* is a not good name but *2013_my_report.md* is.
- Avoid punctuation: *malik's_report.md* is not a good name but *maliks_report.md* is.
- Avoid accented characters: *01_zoë_report.md* is not a good name but *01_zoe_report.md* is.
- Avoid case sensitivity: *AdamHooverReport.md* is not a good name but *adam-hoover-report.md* is.
- Use delimiters: *executivereportpepsi1.md* is not a good name but *executive_report-pepsi_v1.md* is.

So to wrap up, spaces, punctuations, and periods should be avoided but underscores and dashes are recommended. You should always use lowercase since you don't have to later remember if the name of the file contained lowercase or uppercase letters. Another suggestion is the use of delimiters (hyphens or underscores) instead of combining all the

words together. The use of delimiters makes it easier to look for files on your computer or in R and to extract information from the file names like the image below.

```
2018_jan_sales_cust001_prod001.md
2017_mar_sales_cust001_prod001.md
2016_may_sales_cust001_prod008.md
2017_jan_sales_cust120_prod007.md
2015_oct_sales_cust034_prod001.md
2015_oct_sales_cust034_prod002.md
```

Year	Month	Type	Customer ID	Product ID
2018	jan	sales	001	001
2017	mar	sales	001	001
2016	may	sales	001	008
2017	jan	sales	120	007
2015	oct	sales	034	001
2015	oct	sales	034	002

Extracting information from properly named files

Human readable files

What does it mean for a file to be human readable? A file name is human readable if the name tells you something informative about the content of the file. For instance, the name `analysis.R` does not tell you what is in the file especially if you do multiple analyses at the same time. Is this analysis about the project you did for your client X or your client Y? Is it preliminary analysis or your final analysis? A better name maybe would be `2017-exploratory_analysis_crime.R`. The ordering of the information is mostly up to you but make sure the ordering makes sense. For better browsing of your files, it is better to use the dates and numbers in the beginning of the file name.

Be nicely ordered

By using dates, you can sort your files based on chronological order. Dates are preferred to be in the ISO8601 format. In the United States we mainly use the mm-dd-yyyy format. If we use this format for naming files, files will be first sorted based on month, then day, then

year. However for browsing purposes it is better to sort files based on year, then month, and then day and, therefore, the yyyy-mm-dd format, also known as the ISO8601 format, is better. Therefore, `2017-05-21-analysis-cust001.R` is preferred to `05-21-2017-analysis-cust001.R`.

If dates are not relevant for your file naming, put something numeric first. For instance if you're dealing with multiple reports, you can add a reportxxx to the beginning of the file name so you can easily sort files by the report number.

`Report01_cust12_prod03.md`
`Report02_cust12_prod03.md`
`Report03_cust12_prod03.md`
`Report04_cust12_prod03.md`

Using numbers for ordering files

In addition to making sure your files can be nicely ordered, always left-pad numbers with zeros. That is first set a max number of digits for your numbers determined by how many files you will potentially have. So if you may not have more than 1000 files you can choose three digits. If not more than a hundred you can choose two digits and so on. Once you know the number of digits, left-pad numbers with zeros to satisfy the number of digits you determined in the first step. In other words, if you're using three digits, instead of writing 1 write 001 and instead of writing 17 write 017.

Bad Naming	Good Naming
1_consumer_analysis.md	001_consumer_analysis.md
reg_1_company_data.R	reg_01_company_data.R

Left padding numbers with zeros

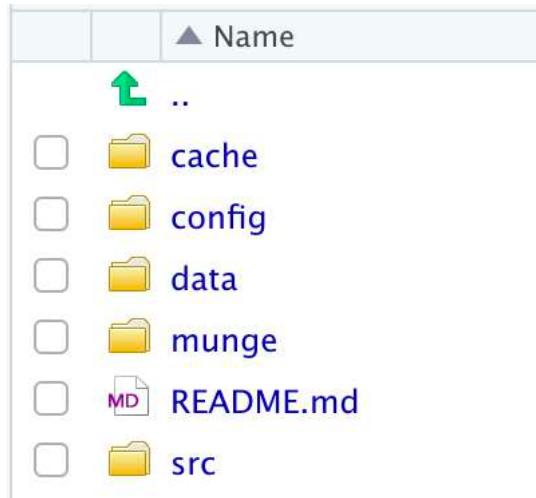
Project Template: Everything In Its Place

When organizing a data science project all of your files need to be placed somewhere on your computer. While there is no universal layout for how to do this, most projects have some aspects in common. The `ProjectTemplate` package codifies some of these defaults into a usable file hierarchy so that you can immediately start placing files in the right place.

Here, we load the `ProjectTemplate` package and create a new project called `data_analysis` under the current working directory using the `minimal` template.

```
library(ProjectTemplate)
create.project(project.name = "data_analysis",
              template = "minimal")
```

The `create.project()` function creates a directory called `data_analysis` for this project. Inside that directory are the following sub-directories (which we can view in the RStudio File browser):



Project Template Directory Layout

Inside each directory is a README file that contains a brief description of what kinds of files should go in this directory. If you do not need all of these directories, it is okay to leave them empty for now.

The `data` directory is most straightforward as it holds any data files (in any variety of formats) that will be needed for the project. The `munge` directory contains R code files that pre-process the data you will eventually use in any analysis. Any results from pre-processing can be stored in the `cache` directory if needed (for example, if the pre-processing takes a long time). Finally, the `src` directory contains R code for data analysis, such as fitting statistical models, computing summary statistics, or creating plots.

A benefit of the `ProjectTemplate` package is that it allows for a lot of customization. So if there is file/directory structure that you commonly use for your projects, then you can create a custom template that can be called everytime you call `create.project()`. Custom templates can be created with the `create.template()` function. For example, you might always want to have a directory called `plots` for saving plots made as part of the data analysis.

The `config` directory can contain configuration information for your project, such as any packages that need to be loaded for your code to work. We will not go into the details of this directory for now, but suffice it to say that there are many ways to customize your project. Lastly, the `load.project()` function can be used to “setup” your project each time you open it. For example, if you always need to execute some code or load some packages, calling `load.project()` with the right `config` settings will allow you to automate this process.

Data Science Workflows

The phrase “data science workflow” describes the method or steps by which a data scientist might evaluate data to perform a data analysis from start to finish. In this course we will provide examples of workflows with real data. In general, workflows involve the following steps:

- Identifying a question of interest - determining if it is feasible
- Identifying data to answer that question
- Importing that data into a programming language such as R
- Cleaning / wrangling / and tiding the data
- Exploratory data analysis to get to know the data
- Data analysis to look for associations in the data
- Generation of data visualizations to demonstrate findings
- Communication of your analysis and findings

We will demonstrate potential ways of organizing a workflow using real data from the [Open Case Studies](#) project.

Case Studies

Throughout this book, we’re going to make use of a number of case studies from [Open Case Studies](#) to demonstrate the concepts introduced in the course. We will generally make use of the same case studies throughout the book, providing continuity to allow you focus on the concepts and skills being taught (rather than the context) while working with interesting data. These case studies aim to address a public-health question and all of them use real data.

Case Study #1: Health Expenditures

The material for this first case study comes from the following:

Kuo, Pei-Lun and Jager, Leah and Taub, Margaret and Hicks, Stephanie. (2019, February 14). opencasestudies/ocs-healthexpenditure: Exploring Health Expenditure using State-level data in the United States (Version v1.0.0). Zenodo. <http://doi.org/10.5281/zenodo.2565307>

Health policy in the United States of America is complicated, and several forms of health care coverage exist, including that of federal government-led health care programs and that

of private insurance companies. We are interested in getting sense of the health expenditure, including health care coverage and health care spending, across the United States. More specifically, the questions are:

1. Is there a relationship between health care coverage and health care spending in the United States?
2. How does the spending distribution change across geographic regions in the United States?
3. Does the relationship between health care coverage and health care spending in the United States change from 2013 to 2014?

Data: health care

The two datasets used in this case study come from the [Henry J Kaiser Family Foundation \(KFF\)](#) and include:

- [Health Insurance Coverage of the Total Population](#) - Includes years 2013-2016
- [Health Care Expenditures by State of Residence \(in millions\)](#) - Includes years 1991-2014

Case Study #2: Firearms

The material for the second case study comes from the following:

Stephens, Alexandra and Jager, Leah and Taub, Margaret and Hicks, Stephanie. (2019, February 14). opencasestudies/ocs-police-shootings-firearm-legislation: Firearm Legislation and Fatal Police Shootings in the United States (Version v1.0.0). Zenodo. <http://doi.org/10.5281/zenodo.2565249>

In the United States, firearm laws differ by state. Additionally, [police shootings are frequently in the news](#). Understanding the relationship between firearm laws and police shootings is of public health interest.

A recent study set out “[to examine whether stricter firearm legislation is associated with rates of fatal police shootings](#)”. We’ll use the state-level data from this study about firearm legislation and fatal police shootings in this case study.

Question

The main question from this case study is: At the state-level, what is the relationship between firearm legislation strength and annual rate of fatal police shootings?

Data

To accomplish this in this case study, we'll use data from a *number* of different sources:

- [The Brady Campaign State Scorecard 2015](#): numerical scores for firearm legislation in each state.
- [The Counted](#): Persons killed by police in the US. The Counted project started because “the US government has no comprehensive record of the number of people killed by law enforcement.”
- [US Census](#): Population characteristics by state.
- [FBI’s Uniform Crime Report](#).
- [Unemployment rate data](#).
- [US Census 2010 Land Area](#).
- Education data for 2010 via the [US Census education table editor](#).
- Household firearm ownership rates - by using the percentage of firearm suicides to all suicides as a proxy (as this was used in the above [referenced study](#)) that we are trying to replicate. This data is downloaded from the [CDC’s Web-Based Injury Statistics Query and Reporting System](#).

2. Importing Data in the Tidyverse

Data are stored in all sorts of different file formats and structures. In this course, we'll discuss each of these common formats and discuss how to get them into R so you can start working with them!

About This Course

Getting data into your statistical analysis system can be one of the most challenging parts of any data science project. Data must be imported and harmonized into a coherent format before any insights can be obtained. You will learn how to get data into R from commonly used formats and how to harmonize different kinds of datasets from different sources. If you work in an organization where different departments collect data using different systems and different storage formats, then this course will provide essential tools for bringing those datasets together and making sense of the wealth of information in your organization.

This course introduces the Tidyverse tools for importing data into R so that it can be prepared for analysis, visualization, and modeling. Common data formats are introduced, including delimited files, spreadsheets, and relational databases. We will also introduce techniques for obtaining data from the web, such as web scraping and getting data from web APIs.

In this book we assume familiarity with the R programming language. If you are not yet familiar with R, we suggest you first complete [R Programming](#) before returning to complete this course.

Tibbles

Before we can discuss any particular file format, let's discuss the end goal - the **tibble!** If you've been using R for a while, you're likely familiar with the `data.frame`. It's best to think of tibbles as an updated and stylish version of the `data.frame`. And, tibbles are what tidyverse packages work with most seamlessly. Now, that doesn't mean tidyverse packages *require* tibbles. In fact, they still work with `data.frames`, but the more you work with tidyverse and tidyverse-adjacent packages, the more you'll see the advantages of using tibbles.

Before we go any further, tibbles *are* data frames, but they have some new bells and whistles to make your life easier.

How tibbles differ from data.frame

There are a number of differences between tibbles and data.frames. To see a full vignette about tibbles and how they differ from data.frame, you'll want to execute `vignette("tibble")` and read through that vignette. However, we'll summarize some of the most important points here:

- **Input type remains unchanged** - `data.frame` is notorious for treating strings as factors; this will not happen with tibbles
- **Variable names remain unchanged** - In base R, creating `data.frames` will remove spaces from names, converting them to periods or add “x” before numeric column names. Creating tibbles will not change variable (column) names.
- **There are no `row.names()` for a tibble** - Tidy data requires that variables be stored in a consistent way, removing the need for row names.
- **Tibbles print first ten rows and columns that fit on one screen** - Printing a tibble to screen will never print the entire huge data frame out. By default, it just shows what fits to your screen.

Creating a tibble

The `tibble` package is part of the `tidyverse` and can thus be loaded in (once installed) using:

```
library(tidyverse)
```

```
as_tibble()
```

Since many packages use the historical `data.frame` from base R, you'll often find yourself in the situation that you have a `data.frame` and want to convert that `data.frame` to a tibble. To do so, the `as_tibble()` function is exactly what you're looking for.

For example, the `trees` dataset is a `data.frame` that's available in base R. This dataset stores the diameter, height, and volume for Black Cherry Trees. To convert this `data.frame` to a tibble you would use the following:

```
as_tibble(trees)
# A tibble: 31 × 3
  Girth Height Volume
  <dbl>   <dbl>   <dbl>
1     8.3     70    10.3
2     8.6     65    10.3
3     8.8     63    10.2
4    10.5     72    16.4
5    10.7     81    18.8
6    10.8     83    19.7
7     11      66    15.6
8     11      75    18.2
9   11.1     80    22.6
10   11.2     75    19.9
# ... with 21 more rows
```

Note in the above example and as mentioned earlier, that tibbles, by default, only print the first ten rows to screen. If you were to print `trees` to screen, all 31 rows would be displayed. When working with large `data.frames`, this default behavior can be incredibly frustrating. Using tibbles removes this frustration because of the default settings for tibble printing.

Additionally, you'll note that the type of the variable is printed for each variable in the tibble. This helpful feature is another added bonus of tibbles relative to `data.frame`.

If you *do* want to see more rows from the tibble, there are a few options! First, the `View()` function in RStudio is incredibly helpful. The input to this function is the `data.frame` or tibble you'd like to see. Specifically, `View(trees)` would provide you, the viewer, with a scrollable view (in a new tab) of the complete dataset.

A second option is the fact that `print()` enables you to specify how many rows and columns you'd like to display. Here, we again display the `trees` `data.frame` as a tibble but specify that we'd only like to see 5 rows. The `width = Inf` argument specifies that we'd like to see all the possible columns. Here, there are only 3, but for larger datasets, this can be helpful to specify.

```

as_tibble(trees) %>%
  print(n = 5, width = Inf)
# A tibble: 31 × 3
   Girth Height Volume
   <dbl>   <dbl>   <dbl>
1     8.3     70    10.3
2     8.6     65    10.3
3     8.8     63    10.2
4    10.5     72    16.4
5    10.7     81    18.8
# ... with 26 more rows

```

Other options for viewing your tibbles are the `slice_*` functions of the `dplyr` package.

The `slice_sample()` function of the `dplyr` package will allow you to see a sample of random rows in random order. The number of rows to show is specified by the `n` argument. This can be useful if you don't want to print the entire tibble, but you want to get a greater sense of the values. This is a good option for data analysis reports, where printing the entire tibble would not be appropriate if the tibble is quite large.

slice_sample(trees, n = 10)			
	Girth	Height	Volume
1	17.5	82	55.7
2	16.0	72	38.3
3	17.3	81	55.4
4	8.8	63	10.2
5	11.1	80	22.6
6	10.7	81	18.8
7	11.0	66	15.6
8	14.0	78	34.5
9	20.6	87	77.0
10	8.3	70	10.3

You can also use `slice_head()` or `slice_tail()` to take a look at the top rows or bottom rows of your tibble. Again the number of rows can be specified with the `n` argument.

This will show the first 5 rows.

```
slice_head(trees, n = 5)
Girth Height Volume
1 8.3     70   10.3
2 8.6     65   10.3
3 8.8     63   10.2
4 10.5    72   16.4
5 10.7    81   18.8
```

This will show the last 5 rows.

```
slice_tail(trees, n = 5)
Girth Height Volume
1 17.5    82   55.7
2 17.9    80   58.3
3 18.0    80   51.5
4 18.0    80   51.0
5 20.6    87   77.0
```

tibble()

Alternatively, you can create a tibble on the fly by using `tibble()` and specifying the information you'd like stored in each column. Note that if you provide a single value, this value will be repeated across all rows of the tibble. This is referred to as “recycling inputs of length 1.”

In the example here, we see that the column `c` will contain the value ‘1’ across all rows.

```
tibble(
  a = 1:5,
  b = 6:10,
  c = 1,
  z = (a + b)^2 + c
)
# A tibble: 5 × 4
  a     b     c     z
  <int> <int> <dbl> <dbl>
1     1     6     1     50
2     2     7     1     82
```

```
3     3     8     1   122
4     4     9     1   170
5     5    10     1   226
```

The `tibble()` function allows you to quickly generate tibbles and even allows you to reference columns within the tibble you're creating, as seen in column z of the example above.

We also noted previously that tibbles can have column names that are not allowed in `data.frame`. In this example, we see that to utilize a nontraditional variable name, you surround the column name with backticks. Note that to refer to such columns in other tidyverse packages, you'll continue to use backticks surrounding the variable name.

```
tibble(
  `two words` = 1:5,
  `12` = "numeric",
  `:)` = "smile",
)
# A tibble: 5 × 3
  `two words` `12`    `:)`
  <int> <chr>  <chr>
1      1 numeric smile
2      2 numeric smile
3      3 numeric smile
4      4 numeric smile
5      5 numeric smile
```

Subsetting

Subsetting tibbles also differs slightly from how subsetting occurs with `data.frame`. When it comes to tibbles, `[]` can subset by name or position; `$` only subsets by name. For example:

```
df <- tibble(  
  a = 1:5,  
  b = 6:10,  
  c = 1,  
  z = (a + b)^2 + c  
)  
  
# Extract by name using $ or []]  
df$z  
[1] 50 82 122 170 226  
df[["z"]]  
[1] 50 82 122 170 226  
  
# Extract by position requires []]  
df[[4]]  
[1] 50 82 122 170 226
```

Having now discussed tibbles, which are the type of object most tidyverse and tidyverse-adjacent packages work best with, we now know the goal. In many cases, tibbles are ultimately what we want to work with in R. However, data are stored in many different formats outside of R. We'll spend the rest of this course discussing those formats and talking about how to get those data into R so that you can start the process of working with and analyzing these data in R.

Spreadsheets

Spreadsheets are an incredibly common format in which data are stored. If you've ever worked in Microsoft Excel or Google Sheets, you've worked with spreadsheets. By definition, spreadsheets require that information be stored in a grid utilizing rows and columns.

Excel files

Microsoft Excel files, which typically have the file extension .xls or .xlsx, store information in a workbook. Each workbook is made up of one or more spreadsheet. Within these spreadsheets, information is stored in the format of values and formatting (colors, conditional formatting, font size, etc.). While this may be a format you've worked with before and are familiar, we note that Excel files can only be viewed in specific pieces of software (like

Microsoft Excel), and thus are generally less flexible than many of the other formats we'll discuss in this course. Additionally, Excel has certain defaults that make working with Excel data difficult outside of Excel. For example, Excel has a habit of aggressively changing data types. For example if you type 1/2, to mean 0.5 or one-half, Excel assumes that this is a date and converts this information to January 2nd. If you are unfamiliar with these defaults, your spreadsheet can sometimes store information other than what you or whoever entered the data into the Excel spreadsheet may have intended. Thus, it's important to understand the quirks of how Excel handles data. Nevertheless, many people *do* save their data in Excel, so it's important to know how to work with them in R.

Reading Excel files into R

Reading spreadsheets from Excel into R is made possible thanks to the `readxl` package. This is not a core tidyverse package, so you'll need to install and load the package in before use:

```
##install.packages("readxl")
library(readxl)
```

The function `read_excel()` is particularly helpful whenever you want read an Excel file into your R Environment. The only required argument of this function is the path to the Excel file on your computer. In the following example, `read_excel()` would look for the file “filename.xlsx” in your current working directory. If the file were located somewhere else on your computer, you would have to provide the *path* to that file.

```
# read Excel file into R
df_excel <- read_excel("filename.xlsx")
```

Within the `readxl` package there are a number of example datasets that we can use to demonstrate the packages functionality. To read the example dataset in, we'll use the `readxl_example()` function.

```
# read example file into R
example <- readxl_example("datasets.xlsx")
df <- read_excel(example)
df
# A tibble: 150 × 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
        <dbl>      <dbl>       <dbl>      <dbl>   <chr>
1         5.1        3.5        1.4       0.2 setosa
2         4.9        3.0        1.4       0.2 setosa
3         4.7        3.2        1.3       0.2 setosa
4         4.6        3.1        1.5       0.2 setosa
5         5.0        3.6        1.4       0.2 setosa
6         5.4        3.9        1.7       0.4 setosa
7         4.6        3.4        1.4       0.3 setosa
8         5.0        3.4        1.5       0.2 setosa
9         4.4        2.9        1.4       0.2 setosa
10        4.9        3.1        1.5       0.1 setosa
# ... with 140 more rows
```

Note that the information stored in `df` is a tibble. This will be a common theme throughout the packages used in these courses.

Further, by default, `read_excel()` converts blank cells to missing data (NA). This behavior can be changed by specifying the `na` argument within this function. There are a number of additional helpful arguments within this function. They can all be seen using `?read_excel`, but we'll highlight a few here:

- `sheet` - argument specifies the name of the sheet from the workbook you'd like to read in (string) or the integer of the sheet from the workbook.
- `col_names` - specifies whether the first row of the spreadsheet should be used as column names (default: TRUE). Additionally, if a character vector is passed, this will rename the columns explicitly at time of import.
- `skip` - specifies the number of rows to skip before reading information from the file into R. Often blank rows or information about the data are stored at the top of the spreadsheet that you want R to ignore.

For example, we are able to change the column names directly by passing a character string to the `col_names` argument:

```
# specify column names on import
read_excel(example, col_names = LETTERS[1:5])
# A tibble: 151 × 5
  A       B       C       D       E
  <chr>   <chr>   <chr>   <chr>   <chr>
1 Sepal.Length Sepal.Width Petal.Length Petal.Width Species
2 5.1         3.5      1.4      0.2      setosa
3 4.9         3        1.4      0.2      setosa
4 4.7         3.2      1.3      0.2      setosa
5 4.6         3.1      1.5      0.2      setosa
6 5           3.6      1.4      0.2      setosa
7 5.4         3.9      1.7      0.4      setosa
8 4.6         3.4      1.4      0.3      setosa
9 5           3.4      1.5      0.2      setosa
10 4.4        2.9     1.4      0.2      setosa
# ... with 141 more rows
```

To take this a step further let's discuss one of the lesser-known arguments of the `read_excel()` function: `.name_repair`. This argument allows for further fine-tuning and handling of column names.

The default for this argument is `.name_repair = "unique"`. This checks to make sure that each column of the imported file has a unique name. If TRUE, `readxl` leaves them as is, as you see in the example here:

```
# read example file into R using .name_repair default
read_excel(
  readxl_example("deaths.xlsx"),
  range = "arts!A5:F8",
  .name_repair = "unique"
)
# A tibble: 3 × 6
  Name      Profession    Age `Has kids` `Date of birth` `Date of death` \
  <chr>     <chr>      <dbl> <lgl>      <dttm>      <dttm>      \
1 David Bowie  musician     69 TRUE      1947-01-08 00:00:00 2016-01-10 00:00\00:00
2 Carrie Fisher actor      60 TRUE      1956-10-21 00:00:00 2016-12-27 00:00\00:00
```

```
0:00
3 Chuck Berry    musician      90 TRUE          1926-10-18 00:00:00 2017-03-18 00:00\0:00
```

Another option for this argument is `.name_repair = "universal"`. This ensures that column names don't contain any forbidden characters or reserved words. It's often a good idea to use this option if you plan to use these data with other packages downstream. This ensures that all the column names will work, regardless of the R package being used.

```
# require use of universal naming conventions
read_excel(
  readxl_example("deaths.xlsx"),
  range = "arts!A5:F8",
  .name_repair = "universal"
)
New names:
* `Has kids` -> Has.kids
* `Date of birth` -> Date.of.birth
* `Date of death` -> Date.of.death
# A tibble: 3 × 6
  Name       Profession   Age Has.kids Date.of.birth     Date.of.death \ 
  <chr>      <chr>      <dbl> <lgl>    <dttm>        <dttm>        \ 
1 David Bowie  musician      69 TRUE      1947-01-08 00:00:00 2016-01-10 00:00:\00
2 Carrie Fisher actor       60 TRUE      1956-10-21 00:00:00 2016-12-27 00:00:\00
3 Chuck Berry  musician      90 TRUE      1926-10-18 00:00:00 2017-03-18 00:00:\00
```

Note that when using `.name_repair = "universal"`, you'll get a readout about which column names have been changed. Here you see that column names with a space in them have been changed to periods for word separation.

Aside from these options, functions can be passed to `.name_repair`. For example, if you want all of your column names to be uppercase, you would use the following:

```
# pass function for column naming
read_excel(
  readxl_example("deaths.xlsx"),
  range = "arts!A5:F8",
  .name_repair = toupper
)
# A tibble: 3 × 6
  NAME      PROFESSION    AGE `HAS KIDS` `DATE OF BIRTH` `DATE OF DEATH`\\
  <chr>     <chr>       <dbl> <lgl>        <dttm>          <dttm>          \\
1 David Bowie   musician     69 TRUE        1947-01-08 00:00:00 2016-01-10 00:0\\
0:00
2 Carrie Fisher actor       60 TRUE        1956-10-21 00:00:00 2016-12-27 00:0\\
0:00
3 Chuck Berry   musician     90 TRUE        1926-10-18 00:00:00 2017-03-18 00:0\\
0:00
```

Notice that the function is passed directly to the argument. It does not have quotes around it, as we want this to be interpreted as the `toupper()` function.

Here we've really only focused on a single function (`read_excel()`) from the `readxl` package. This is because some of the best packages do a single thing and do that single thing well. The `readxl` package has a single, slick function that covers most of what you'll need when reading in files from Excel. That is not to say that the package doesn't have other useful functions (it does!), but this function will cover your needs most of the time.

Google Sheets

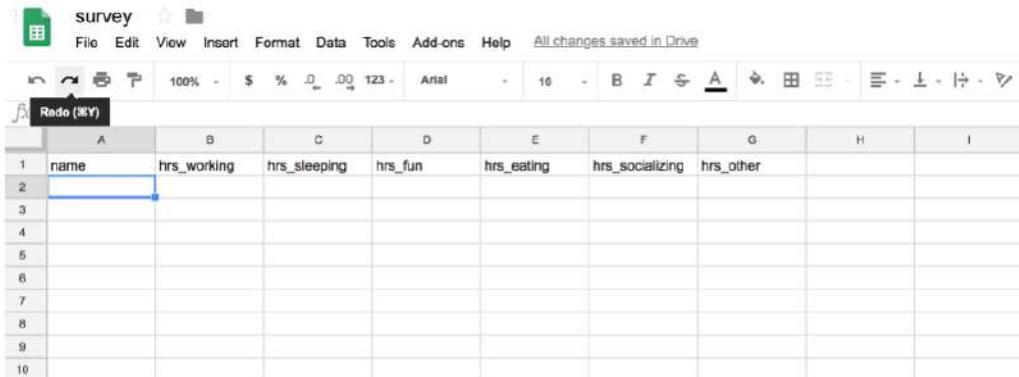
Similar to Microsoft Excel, Google Sheets is another place in which spreadsheet information is stored. Google Sheets also stores information in spreadsheets within workbooks. Like Excel, it allows for cell formatting and has defaults during data entry that *could* get you into trouble if you're not familiar with the program.

Unlike Excel files, however, Google Sheets live on the Internet, rather than your computer. This makes sharing and updating Google Sheets among people working on the same project much quicker. This also makes the process for reading them into R slightly different. Accordingly, it requires the use of a different, but also very helpful package, `googlesheets4`!

As Google Sheets are stored on the Internet and not on your computer, the `googlesheets4` package does not require you to download the file to your computer before reading it into

R. Instead, it reads the data into R directly from Google Sheets. Note that if the data hosted on Google Sheets changes, every time the file is read into R, the most updated version of the file will be utilized. This can be very helpful if you're collecting data over time; however, it could lead to unexpected changes in results if you're not aware that the data in the Google Sheet is changing.

To see exactly what we mean, let's look at a specific example. Imagine you've sent out a survey to your friends asking about how they spend their day. Let's say you're mostly interested in knowing the hours spent on work, leisure, sleep, eating, socializing, and other activities. So in your Google Sheet you add these six items as columns and one column asking for your friends names. To collect this data, you then share the link with your friends, giving them the ability to edit the Google Sheet.



The screenshot shows a Google Sheets document titled "survey". The spreadsheet has a header row with columns labeled "name", "hrs_working", "hrs_sleeping", "hrs_fun", "hrs_eating", "hrs_socializing", and "hrs_other". Rows 1 through 10 are visible, with row 2 being the active row as indicated by the blue selection bar under the "hrs_working" column. The rest of the rows are empty. The top menu bar includes File, Edit, View, Insert, Format, Data, Tools, Add-ons, Help, and "All changes saved in Drive". The toolbar below the menu includes icons for file, print, search, zoom, and various styling options like font, size, and alignment.

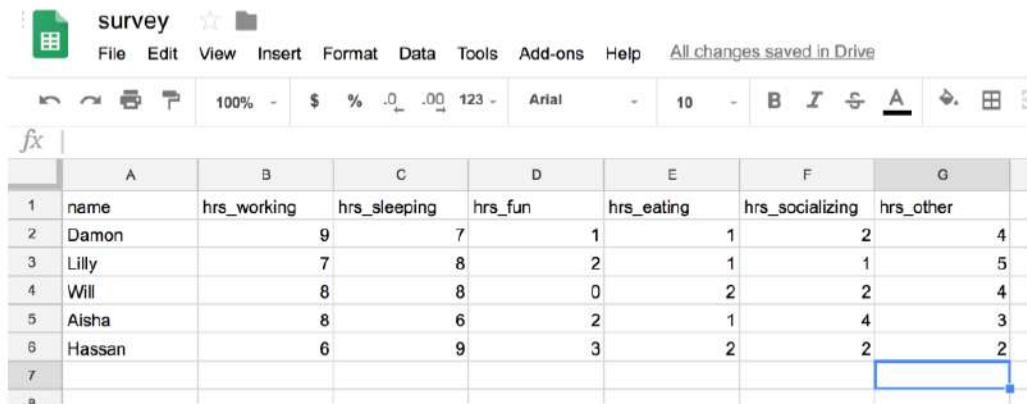
	A	B	C	D	E	F	G	H	I
1	name	hrs_working	hrs_sleeping	hrs_fun	hrs_eating	hrs_socializing	hrs_other		
2									
3									
4									
5									
6									
7									
8									
9									
10									

Survey Google Sheets

Your friends will then one-by-one complete the survey. And, because it's a Google Sheet, everyone will be able to update the Google Sheet, regardless of whether or not someone else is also looking at the Sheet at the same time. As they do, you'll be able to pull the data and import it to R for analysis at any point. You won't have to wait for everyone to respond. You'll be able to analyze the results in real-time by directly reading it into R from Google Sheets, avoiding the need to download it each time you do so.

In other words, every time you import the data from the Google Sheets link using the

googlesheets4 package, the most updated data will be imported. Let's say, after waiting for a week, your friends' data look something like this:



	A	B	C	D	E	F	G
1	name	hrs_working	hrs_sleeping	hrs_fun	hrs_eating	hrs_socializing	hrs_other
2	Damon	9	7	1	1	2	4
3	Lilly	7	8	2	1	1	5
4	Will	8	8	0	2	2	4
5	Aisha	8	6	2	1	4	3
6	Hassan	6	9	3	2	2	2
7							

Survey Data

You'd be able to analyze these updated data using R and the `googlesheets4` package!

In fact, let's have you do that right now! Click on [this link](#) to see these data!

The `googlesheets4` package

The `googlesheets4` package allows R users to take advantage of the Google Sheets Application Programming Interface (API). Very generally, APIs allow different applications to communicate with one another. In this case, Google has released an API that allows other software to communicate with Google Sheets and retrieve data and information directly from Google Sheets. The `googlesheets4` package enables R users (you!) to easily access the Google Sheets API and retrieve your Google Sheets data.

Using this package is the best and easiest way to analyze and edit Google Sheets data in R. In addition to the ability of pulling data, you can also edit a Google Sheet or create new sheets.

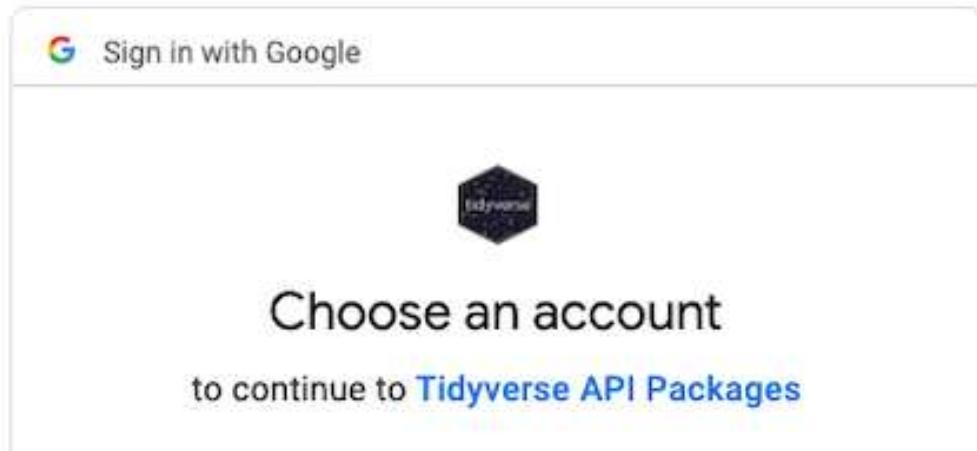
The `googlesheets4` package is tidyverse-adjacent, so it requires its own installation. It also requires that you load it into R before it can be used.

Getting Started with `googlesheets4`

```
#install.packages("googlesheets4")
# load package
library(googlesheets4)
```

Now, let's get to importing your survey data into R. Every time you start a new session, you need to authenticate the use of the `googlesheets4` package with your Google account. This is a great feature as it ensures that you want to allow access to your Google Sheets and allows the Google Sheets API to make sure that you should have access to the files you're going to try to access.

The command `gs4_auth()` will open a new page in your browser that asks you which Google account you'd like to have access to. Click on the appropriate Google user to provide `googlesheets4` access to the Google Sheets API.



After you click "ALLOW", giving permission for the `googlesheets4` package to connect to your Google account, you will likely be shown a screen where you will be asked to copy an authentication code. Copy this authentication code and paste it into R.

 Sign in with Google



Tidyverse API Packages wants to access your Google Account

This will allow **Tidyverse API Packages** to:

- See, edit, create, and delete your spreadsheets in  Google Drive

Make sure you trust Tidyverse API Packages

You may be sharing sensitive info with this site or app. Learn about how Tidyverse API Packages will handle your data by reviewing its **privacy policies**. You can always see or remove access in your **Google Account**.

[Learn about the risks](#)

[Cancel](#) [Allow](#)

Navigating `googlesheets4::gs4_find()`

Once authenticated, you can use the command `gs4_find()` to **list** all your worksheets on Google Sheets as a table. Note that this will ask for authorization of the `googledrive` package. We will discuss more about `googledrive` later.

```
> googlesheets4::gs4_find()
# A tibble: 55 x 3
  name          id      drive_resource
  <chr>        <chr>   <list>
  1 Example     2cdw-678dSPLfdID__Lit8eEFZPasdebglGwHk <named list [33]>
  # ... with 54 more rows, and 1 more variable:
  #   gs4_find() <dbl>
```

Reading data in using `googlesheets::gs_read()`

In order to ultimately access the information a specific Google Sheet, you can use the `read_sheets()` function by typing in the id listed for your Google Sheet of interest when using `gs4_find()`.

```
# read Google Sheet into R with id
read_sheet("2cdw-678dSPLfdID__Lit8eEFZPasdebglGwHk")
# note this is a fake id
```

You can also navigate to your own sheets or to other people's sheets using a URL. For example, paste [https://docs.google.com/spreadsheets/d/1FN7VVKzJJyifZFY5POdz_LaGTBYaC4SLBX9vyDnbY/] in your web browser or click [here](#). We will now read this into R using the URL:

```
# read Google Sheet into R with URL
survey_sheet <- read_sheet("https://docs.google.com/spreadsheets/d/1FN7VVKzJJyifZFY5POdz_LaGTBYaC4SLBX9vyDnbY/")
```

Note that we assign the information stored in this Google Sheet to the object `survey_sheet` so that we can use it again shortly.

```
> survey_sheet <- read_sheet("https://docs.google.com/spreadsheets/d/1FN7VVKzJJyifZFY5POdz_LaGTBYaC4SLBX9vyDnbY/")
Reading from "survey"
Range "Sheet1"
> survey_sheet
# A tibble: 5 x 7
  name    hrs_working hrs_sleeping hrs_fun hrs_eating hrs_socializing hrs_other
  <chr>       <dbl>        <dbl>      <dbl>      <dbl>        <dbl>        <dbl>
1 Damon        9            7          1          1          2            4
2 Lilly         7            8          2          1          1            5
3 Will          8            8          0          2          2            4
4 Aisha         8            6          2          1          4            3
5 Hassan        6            9          3          2          2            2
```

Sheet successfully read into R

Note that by default the data on the first sheet will be read into R. If you wanted the data on a particular sheet you could specify with the sheet argument, like so:

```
# read specific Google Sheet into R with URL
survey_sheet <- read_sheet("https://docs.google.com/spreadsheets/d/1FN7VVKzJJyi\fZFY5P0dz_La1GTBYaC4SLB-X9vyDnbY/", sheet = 2)
```

If the sheet was named something in particular you would use this instead of the number 2.

Here you can see that there is more data on sheet 2:

```
> survey_sheet <- read_sheet("https://docs.google.com/spreadsheets/d/1FN7VVKzJJyi\fZFY5P0dz_La1GTBYaC4SLB-X9vyDnbY/", sheet = 2)
Reading from "survey"
Range "'Sheet2'"
> survey_sheet
# A tibble: 28 x 7
   name    hrs_working hrs_sleeping hrs_fun hrs_eating hrs_socializing hrs_other
   <chr>      <dbl>        <dbl>     <dbl>     <dbl>        <dbl>       <dbl>
 1 Damon       9          7          1          1          2          4
 2 Lilly        7          8          2          1          1          5
 3 Will         8          8          2          2          2          4
 4 Aisha        8          6          2          1          4          3
 5 Hassan       6          9          3          2          2          2
 6 Me           10         8          2          2          1          1
 7 Ethan        7          3          2          6          1          5
 8 Emma         6          3          7          2          1          5
 9 Sofia        1          5          6          7          2          3
10 Oliver       1          3          2          5          7          6
# ... with 18 more rows
```

Specific Sheet successfully read into R

There are other additional (optional) arguments to `read_sheet()`, some are similar to those in `read_csv()` and `read_excel()`, while others are more specific to reading in Google Sheets:

- `skip = 1`: will skip the first row of the Google Sheet
- `col_names = FALSE`: specifies that the first row is not column names
- `range = "A1:G5"`: specifies the range of cells that we like to import is A1 to G5
- `n_max = 100`: specifies the maximum number of rows that we want to import is 100

In summary, to read in data from a Google Sheet in `googlesheets4`, you must first know the **id**, the **name** or the **URL** of the Google Sheet and have access to it.

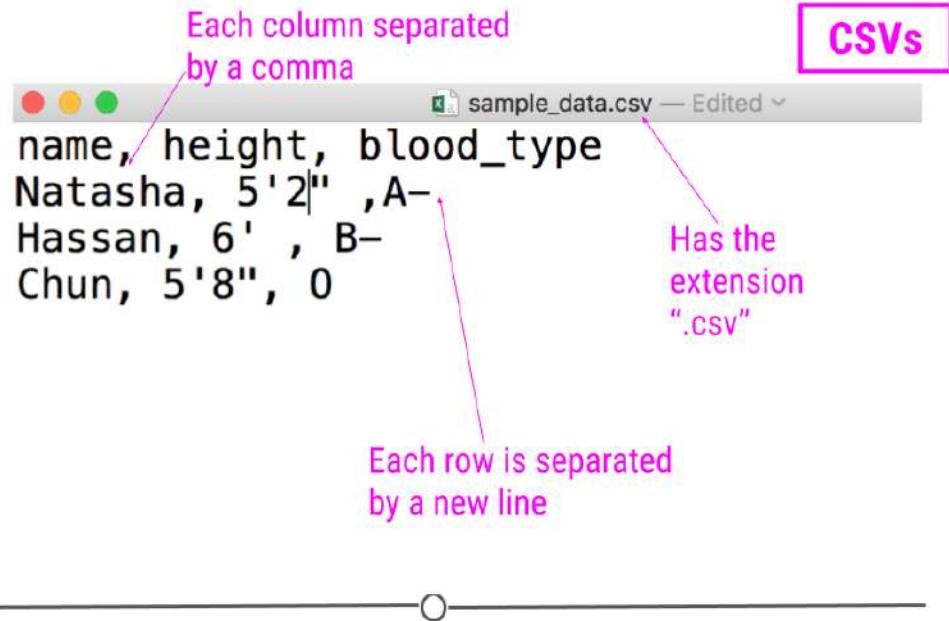
See <https://googlesheets4.tidyverse.org/reference/index.html> for a list of additional functions in the `googlesheets4` package.

CSVs

Like Excel Spreadsheets and Google Sheets, **Comma-separated values (CSV)** files allow us to store tabular data; however, it does this in a much simple format. CSVs are **plain-text** files,

which means that all the important information in the file is represented by text (where text is numbers, letters, and symbols you can type on your keyboard). This means that there are no workbooks or metadata making it difficult to open these files. CSVs are flexible files and are thus the preferred storage method for tabular data for many data scientists.

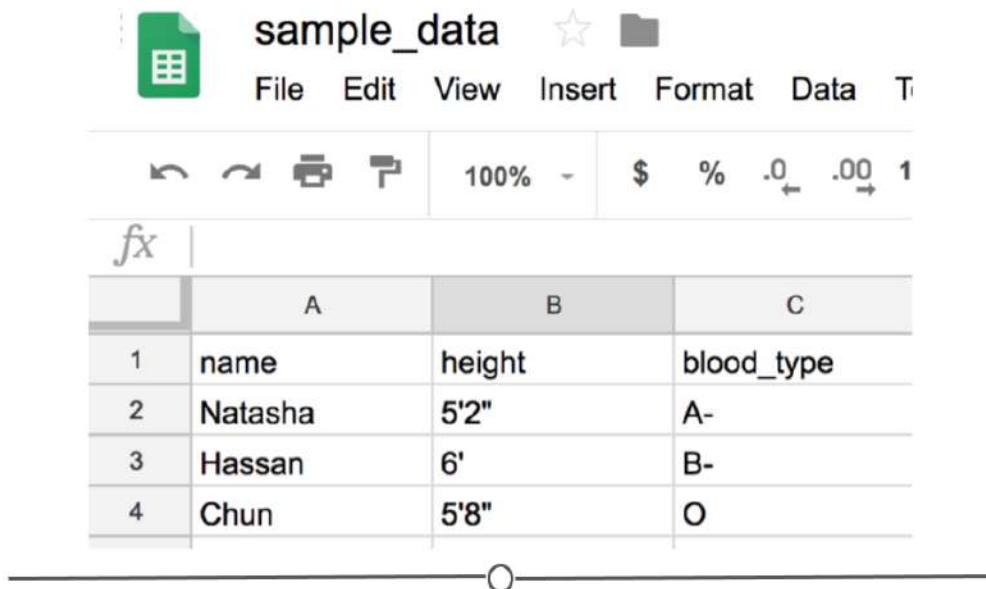
For example, consider a dataset that includes information about the heights and blood types of three individuals. You could make a table that has three columns (names, heights, and blood types) and three rows (one for each person) in Google Docs or Microsoft Word. However, there is a better way of storing this data in plain text without needing to put them in table format. CSVs are a perfect way to store these data. In the CSV format, the values of each column for each person in the data are separated by commas and each row (each person in our case) is separated by a new line. This means your data would be stored in the following format:



sample CSV

Notice that CSV files have a .csv extension at the end. You can see this above at the top of the file. One of the advantages of CSV files is their *simplicity*. Because of this, they are one of the most common file formats used to store tabular data. Additionally, because they are plain text, they are compatible with *many* different types of software. CSVs can be read by most programs. Specifically, for our purposes, these files can be easily read into R (or Google Sheets, or Excel), where they can be better understood by the human eye. Here you see the

same CSV opened in Google Sheets, where it's more easily interpretable by the human eye:



The screenshot shows a Google Sheets interface with a green icon in the top-left corner. The title bar reads "sample_data" with a star and folder icon. The menu bar includes File, Edit, View, Insert, Format, Data, and Tools. Below the menu is a toolbar with icons for back, forward, print, and search, followed by zoom controls (100%, %), currency (\$), percentage (%), and numerical input (.0, .00, 1). The main area displays a table with four rows and three columns. The columns are labeled A, B, and C. The first row contains column headers: "name", "height", and "blood_type". The second row contains values: "Natasha", "5'2\"", and "A-". The third row contains values: "Hassan", "6'", and "B-". The fourth row contains values: "Chun", "5'8\"", and "O".

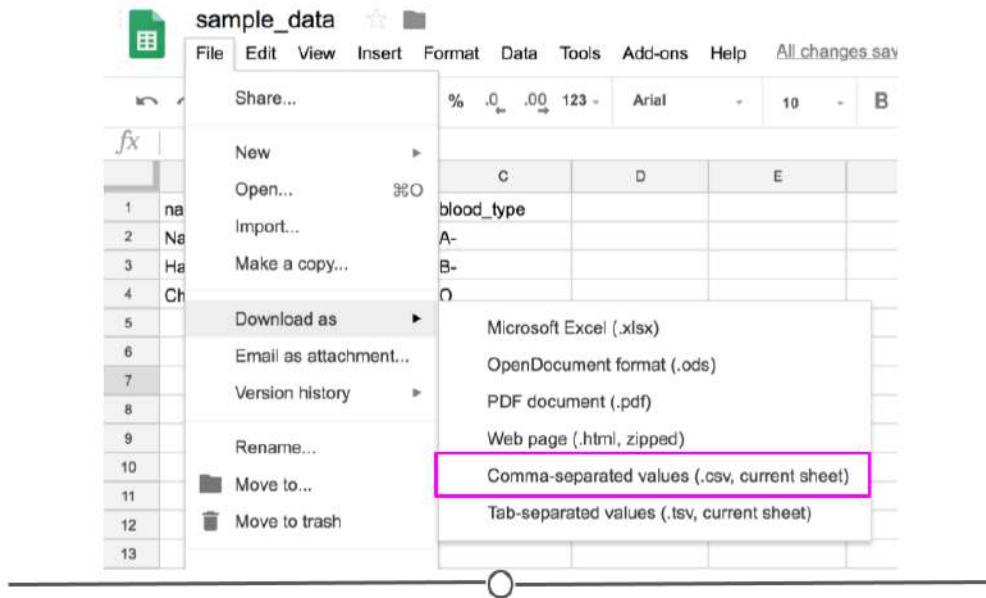
	A	B	C
1	name	height	blood_type
2	Natasha	5'2"	A-
3	Hassan	6'	B-
4	Chun	5'8"	O

CSV opened in Google Sheets

As with any file type, CSVs do have their limitations. Specifically, CSV files are best used for data that have a consistent number of variables across observations. In our example, there are three variables for each observation: “name”, “height”, and “blood_type”. If, however, you had eye color and weight for the second observation, but not for the other rows, you’d have a different number of variables for the second observation than the other two. This type of data is not best suited for CSVs (although NA values could be used to make the data rectangular). Whenever you have information with the same number of variables across all observations, CSVs are a good bet!

Downloading CSV files

If you entered the same values used above into Google Sheets first and wanted to download this file as a CSV to read into R, you would enter the values in Google Sheets and then click on “File” and then “Download as” and choose “Comma-separated values (.csv, current sheet)”. The dataset that you created will be downloaded as a CSV file on your computer. Make sure you know the location of your file (if on a Chromebook, this will be in your “Downloads” folder).



Download as CSV file

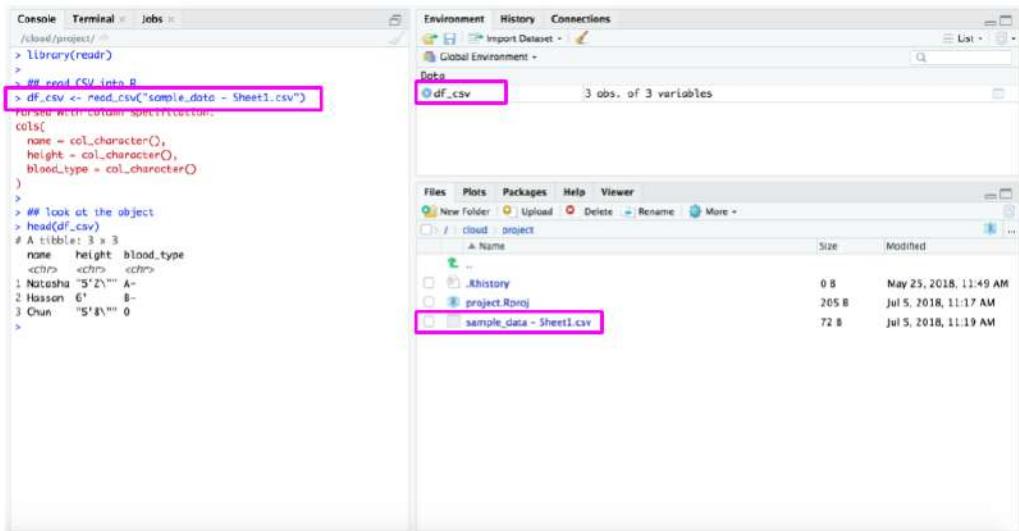
Reading CSVs into R

Now that you have a CSV file, let's discuss how to get it into R! The best way to accomplish this is using the function `read_csv()` from the `readr` package. (Note, if you haven't installed the `readr` package, you'll have to do that first.) Inside the parentheses of the function, write the name of the file in quotes, including the file extension (`.csv`). Make sure you type the exact file name. Save the imported data in a data frame called `df_csv`. Your data will now be imported into R environment. If you use the command `head(df_csv)` you will see the first several rows of your imported data frame:

```
## install.packages("readr")
library(readr)

## read CSV into R
df_csv <- read_csv("sample_data - Sheet1.csv")

## look at the object
head(df_csv)
```



read_csv()

Above, you see the simplest way to import a CSV file. However, as with many functions, there are other arguments that you can set to specify how to import your specific CSV file, a few of which are listed below. However, as usual, to see all the arguments for this function, use `?read_csv` within R.

- `col_names = FALSE` to specify that the first row does NOT contain column names.
- `skip = 2` will skip the first 2 rows. You can set the number to any number you want. This is helpful if there is additional information in the first few rows of your data frame that are not actually part of the table.

- `n_max = 100` will only read in the first 100 rows. You can set the number to any number you want. This is helpful if you’re not sure how big a file is and just want to see part of it.

By default, `read_csv()` converts blank cells to missing data (`NA`).

We have introduced the function `read_csv` here and recommend that you use it, as it is the simplest and fastest way to read CSV files into R. However, we note that there *is* a function `read.csv()` which is available by default in R. You will likely see this function in others’ code, so we just want to make sure you’re aware of it.

TSVs

Another common form of data is text files that usually come in the form of TXT or TSV file formats. Like CSVs, text files are simple, plain-text files; however, rather than columns being separated by commas, they are separated by tabs (represented by “\t” in plain-text). Like CSVs, they don’t allow text formatting (i.e. text colors in cells) and are able to be opened on many different software platforms. This makes them good candidates for storing data.

Reading TSVs into R

The process for reading these files into R is similar to what you’ve seen so far. We’ll again use the `readr` package, but we’ll instead use the `read_tsv()` function.

```
## read TSV into R
df_tsv <- read_tsv("sample_data - Sheet1.tsv")

## look at the object
head(df_tsv)
```

Delimited Files

Sometimes, tab-separated files are saved with the `.txt` file extension. TXT files can store tabular data, but they can also store simple text. Thus, while TSV is the more appropriate extension for tabular data that are tab-separated, you’ll often run into tabular data that

individuals have saved as a TXT file. In these cases, you'll want to use the more generic `read_delim()` function from `readr`.

Google Sheets does not allow tab-separated files to be downloaded with the .txt file extension (since .tsv is more appropriate); however, if you were to have a file “sample_data.txt” uploaded into R, you could use the following code to read it into your R Environment, where “\t” specifies that the file is tab-delimited.

Reading Delimited Files into R

```
## read TXT into R
df_txt <- read_delim("sample_data.txt", delim = "\t")

## look at the object
head(df_txt)
```

This function allows you to specify how the file you're reading is in delimited. This means, rather than R knowing by default whether or not the data are comma- or tab- separated, you'll have to specify it within the argument `delim` in the function.

The `read_delim()` function is a more generic version of `read_csv()`. What this means is that you *could* use `read_delim()` to read in a CSV file. You would just need to specify that the file was comma-delimited if you were to use that function.

Exporting Data from R

The last topic of this lesson is about how to export data from R. So far we learned about reading data into R. However, sometimes you would like to share *your* data with others and need to export your data from R to some format that your collaborators can see.

As discussed above, CSV format is a good candidate because of its simplicity and compatibility. Let's say you have a data frame in the R environment that you would like to export as a CSV. To do so, you could use `write_csv()` from the `readr` package.

Since we've already created a data frame named `df_csv`, we can export it to a CSV file using the following code. After typing this command, a new CSV file called `my_csv_file.csv` will appear in the Files section of RStudio (if you are using it).

```
write_csv(df_csv, path = "my_csv_file.csv")
```

You could similarly save your data as a TSV file using the function `write_tsv()` function.

We'll finally note that there are default R functions `write.csv()` and `write.table()` that accomplish similar goals. You may see these in others' code; however, we recommend sticking to the intuitive and quick `readr` functions discussed in this lesson.

JSON

All of the file formats we've discussed so far (tibbles, CSVs, Excel Spreadsheets, and Google Sheets) are various ways to store what is known as tabular data, data where information is stored in rows and columns. To review, when data are stored in a tidy format, variables are stored in **columns** and each observation is stored in a different **row**. The values for each observation is stored in its respective **cell**. These *rules* for tabular data help define the **structure** of the file. Storing information in rows and columns, however, is not the only way to store data.

Alternatively, JSON (JavaScript Object Notation) data are *nested* and *hierarchical*. JSON is a very commonly-used text-based way to send information between a browser and a server. It is easy for humans to read and to write. JSON data adhere to certain rules in how they are structured. For simplicity, JSON format requires objects to be comprised of **key-value pairs**. For example, in the case of: `{"Name": "Isabela"}`, "Name" would be a key, "Isabela" would be a value, and together they would be a key-value pair. Let's take a look at how JSON data looks in R. This means that key-pairs can be organized into different levels (hierarchical) with some levels of information being stored *within* other levels (nested).

Using a snippet of JSON data here, we see a portion of JSON data from Yelp looking at the **attributes** of a restaurant. Within **attributes**, there are four nested categories: **Take-out**, **Wi-Fi**, **Drive-Thru**, and **Good For**. In the hierarchy, **attributes** is at the top, while these four categories are within **attributes**. Within one of these attributes **Good For**, we see another level within the hierarchy. In this third level we see a number of other categories nested within **Good For**. This should give you a slightly better idea of how JSON data are structured.

```

    "attributes": {
        "Take-out": true, A key-value pair
        "Wi-Fi": "free",
        "Drive-Thru": true,
        "Good For": {
            "dessert": false,
            "latenight": false,
            "lunch": false,
            "dinner": false,
            "breakfast": false,
            "brunch": false
        },
    }

```

<https://blog.exploratory.io/working-with-json-data-in-very-simple-way-ad7ebcc0bb89>

JSON data are hierarchical and nested

To get a sense of what JSON data look like in R, take a peak at this minimal example:

```

## generate a JSON object
json <-
'[{"Name" : "Woody", "Age" : 40, "Occupation" : "Sherriff"}, 
 {"Name" : "Buzz Lightyear", "Age" : 34, "Occupation" : "Space Ranger"}, 
 {"Name" : "Andy", "Occupation" : "Toy Owner"}]'

## take a look
json
[1] "[\n  {"Name" : \"Woody\", \"Age\" : 40, \"Occupation\" : \"Sherriff\"}, \n  {"Name" : \"Buzz Lightyear\", \"Age\" : 34, \"Occupation\" : \"Space Rang\\er\"}, \n  {"Name" : \"Andy\", \"Occupation\" : \"Toy Owner\"\n}]"

```

Here, we've stored information about Toy Story characters, their age, and their occupation in an object called json.

In this format, we cannot easily work with the data within R; however, the `jsonlite` package can help us. Using the defaults of the function `fromJSON()`, `jsonlite` will take the data from JSON array format and helpfully return a data frame.

```
#install.packages("jsonlite")
library(jsonlite)

## take JSON object and convert to a data frame
mydf <- fromJSON(json)

## take a look
mydf
      Name Age   Occupation
1     Woody 40     Sheriff
2 Buzz Lightyear 34 Space Ranger
3     Andy  NA     Toy Owner

## take JSON object and convert to a
## data frame
mydf <- fromJSON(json)

## take a look
mydf      > mydf
      Name Age   Occupation
1     Woody 40     Sheriff
2 Buzz Lightyear 34 Space Ranger
3     Andy  NA     Toy Owner
```

`fromJSON()`

Data frames can also be returned to their original JSON format using the function: `toJSON()`.

```
## take JSON object and convert to a data frame
json <- toJSON(mydf)
json
[{"Name": "Woody", "Age": 40, "Occupation": "Sherriff"}, {"Name": "Buzz Lightyear", "Age": 34, "Occupation": "Space Ranger"}, {"Name": "Andy", "Occupation": "Toy Owner"}]

## take JSON object and convert to a
data frame
json <- toJSON(mydf)

> json
[{"Name": "Woody", "Age": 40, "Occupation": "Sherriff"}, {"Name": "Buzz Lightyear", "Age": 34, "Occupati
on": "Space Ranger"}, {"Name": "Andy", "Occupation": "Toy Owner"}]
```



toJSON()

While this gives us an idea of how to work with JSON formatted data in R, we haven't yet discussed how to read a JSON file into R. When you have data in the JSON format (file extension: .json), you'll use the `read_json()` function, which helpfully looks very similar to the other `read_` functions we've discussed so far:

```
# read JSON file into R
read_json("json_file.json")

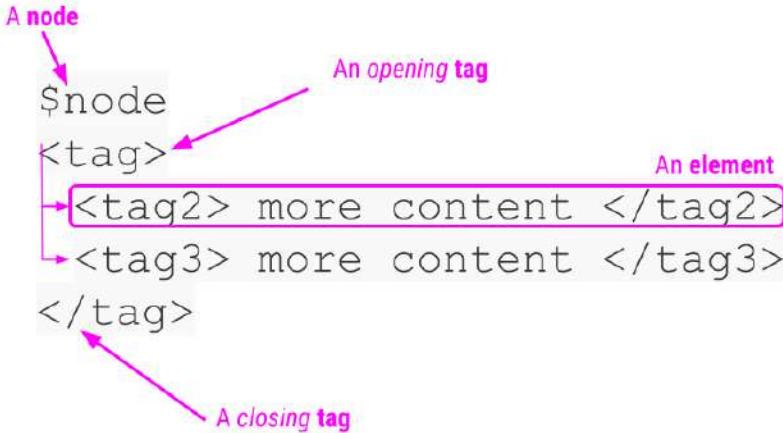
# read JSON file into R and
# simplifies nested lists into vectors and data frames
read_json("json_file.json", simplifyVector = TRUE)
```

Note in our examples here that by default, `read_json()` reads the data in while retaining

the JSON format. However, if you would like to simplify the information into a `data.frame`, you'll want to specify the argument, `simplifyVector = TRUE`.

XML

Extensible Markup Language (XML), is another human- and machine-readable language that is used frequently by web services and APIs. However, instead of being based on key-value pairs, XML relies on **nodes**, **tags**, and **elements**. The author defines these *tags* to specify what information is included in each *element* of the XML document and allows for elements to be nested within one another. The **nodes** define the hierarchical structure of the XML (which means that XML is hierarchical and nested like JSON)!



XML format relies on nodes, tags, and elements

XML accomplishes the same goal as JSON, but it just does it in a different format. Thus, the two formats are commonly used for similar purposes – sharing information on the web; however, because the format in which they do this is different, a different R package is needed to process XML data. This package is called `xml2`.

We will look into the `xml2` package a bit more when we look at importing html files.

```
# read XML file into R  
read_xml("xml_file.xml")
```

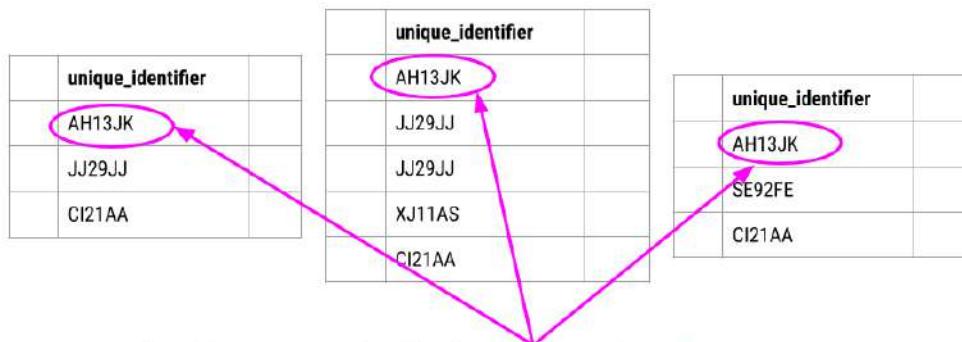
Databases

So far we've discussed reading in data that exist in a single file, like a CSV file or a Google Sheet. However, there will be many cases where the data for your project will be stored across a number of different tables that are all related to one another. In this lesson, we'll discuss what relational data are, why you would want to store data in this way, and how to work with these types of data into R.

Relational Data

Relational data can be thought of as information being stored across many tables, with each table being related to all the other tables. Each table is linked to every other table by a set of **unique identifiers**.

Three tables of information



entries are *related* to one another by their **unique identifier**

Relational data are related by unique identifiers

To better understand this, let's consider a toy example. Consider a town where you have a number of different restaurants. In one table you have information about these restaurants including, where they are located and what type of food they serve. You then have a second table where information about health and safety inspections is stored. Each inspection is a different row and the date of the inspection, the inspector, and the safety rating are stored in this table. Finally, you have a third table. This third table contains information pulled from an API, regarding the number of stars given to each restaurant, as rated by people online. Each table contains different bits of information; however, there is a common column `id` in each of the tables. This allows the information to be linked between the tables. The restaurant with the `id` "JJ29JJ" in the restaurant table would refer to the same restaurant with the `id` "JJ29JJ" in the health inspections table, and so on. The values in this `id` column are known as `unique identifiers` because they uniquely identify each restaurant. No two restaurants will have the same `id`, and the same restaurant will always have the same `id`, no matter what table you're looking at. The fact that these tables have unique identifiers connecting each table to all the other tables makes this example what we call **relational data**.

restaurant				health inspections					rating		
name	id	address	type	name	id	inspection_date	inspector	score	name	id	stars
Taco Stand	AH13JK	1 Main St.	Mexican	Taco Stand	AH13JK	2018-08-21	Sheila	97	Taco Stand	AH13JK	4.9
Pho Place	JJ29JJ	192 Street Rd.	Vietnamese	Pho Place	JJ29JJ	2018-03-12	D'eonte	98	Pho Place	JJ29JJ	4.8
Taco Stand	XJ11AS	18 W. East St.	Fusion	Pho Place	JJ29JJ	2018-01-02	Monica	66	Taco Stand	XJ11AS	4.2
Pizza Heaven	CI21AA	711 K Ave.	Italian	Taco Stand	XJ11AS	2018-12-16	Mark	43	Pizza Heaven	CI21AA	4.7
				Pizza Heaven	CI21AA	2018-08-21	Anh	99			

Unique identifiers help link entries across tables

Why relational data?

Storing data in this way has a number of advantages; however, the three most important are:

1. Efficient Data Storage
2. Avoids Ambiguity
3. Privacy

Efficient Data Storage - By storing each bit of information in a separate table, you limit the need to repeat information. Taking our example above, imagine if we included everything in a single table. This means that for each inspection, we would copy and paste the restaurant's address, type, and number of stars every time the facility is inspected. If a restaurant were inspected 15 times, this same information would be unnecessarily copy and pasted in each row! To avoid this, we simply separate out the information into different tables and relate them by their unique identifiers.

Avoids Ambiguity - Take a look at the first table: "restaurant" here. You may notice there are two different restaurants named "Taco Stand." However, looking more closely, they have a different id *and* a different address. They're even different types of restaurants. So, despite having the same name, they actually are two different restaurants. The unique identifier makes this immediately clear!

restaurant					health inspections					rating		
name	id	address	type		name	id	inspection_date	inspector	score	name	id	stars
Taco Stand	AH13JK	1 Main St.	Mexican		Taco Stand	AH13JK	2018-08-21	Sheila	97			
Pho Place	JJ29JJ	192 Street Rd.	Vietnamese		Pho Place	JJ29JJ	2018-03-12	D'eonte	98			
Taco Stand	XJ11AS	18 W. East St.	Fusion		Pho Place	JJ29JJ	2018-01-02	Monica	66			
Pizza Heaven	CI21AA	711 K Ave.	Italian		Taco Stand	XJ11AS	2018-12-16	Mark	43			
					Pizza Heaven	CI21AA	2018-08-21	Anh	99			

Two different restaurants with the same name!

Unique identifiers in relational data avoid ambiguity

Privacy - In using relational data, if there is ever information that is private and only some people should have access to, using this system simplifies that. You can restrict access to some of the data to ensure only those who should have access are able to access the data.

Relational Databases: SQL

Now that we have an idea of what relational data are, let's spend a second talking about how relational data are stored. Relational data are stored in databases. The most common database is **SQLite**. In order to work with data in databases, there has to be a way to **query** or search the database for the information you're interested in. **SQL** queries search through SQLite databases and return the information you ask for in your query.

For example, a query of the above example may look to obtain information about any restaurant that was inspected after July 1st of 2018. One would then use SQL commands to carry out this query and return the information requested.

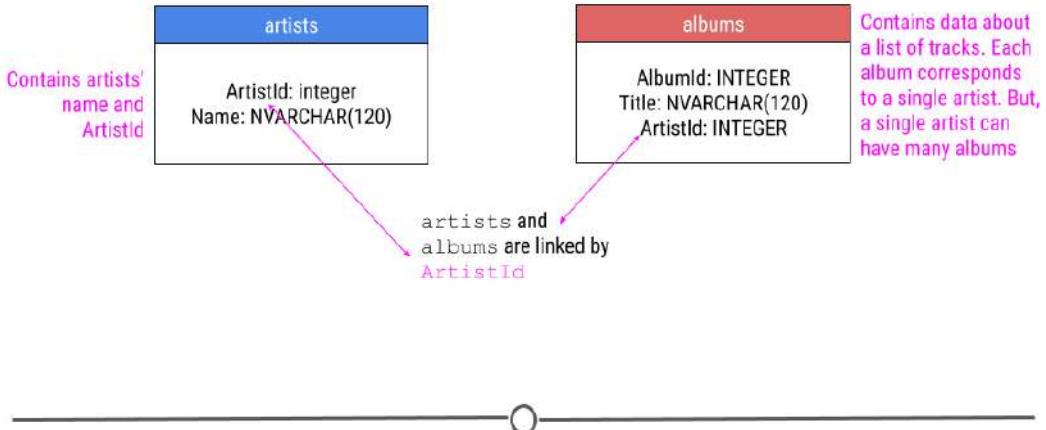
While we won't be discussing how to write SQL commands in-depth here, we *will* be discussing how to use the R package **RSQLite** to connect to an SQLite database using **RSQLite** and how to work with relational data using **dplyr** and **dbplyr**.

Connecting to Databases: **RSQLite**

To better understand databases and how to work with relational data, let's just start working with data from a database! The data we'll be using are from a database with relational data: **company.db**. The database includes tables with data that represents a digital media store. The data includes information generally related to media, artists, artists' work, and those who purchase artists' work (customers). You can download the database file [here](#):

- [company.db.zip](#)

You will need to unzip the file before using it. The original version of this database can be downloaded [here](#). For our purposes though, we're only going to only describe two of the tables we'll be using in our example in this lesson. We're going to be looking at data in the **artists** and **albums** tables, which both have the column **ArtistId**.



relationship between two tables in the company database

Without any more details, let's get to it! Here you'll see the code to install and load the RSQLite package.

You'll then load the `company.db` sample database, connect to the database, and first obtain a list the tables in the database. Before you begin, make sure that the file `company.db` is in your current working directory (you can check by calling the `ls()` function).

```
## install and load packages
## this may take a minute or two
# install.packages("RSQLite")
library(RSQLite)

## Specify driver
sqlite <- dbDriver("SQLite")

## Connect to Database
db <- dbConnect(sqlite, "company.db")

## List tables in database
dbListTables(db)
[1] "albums"          "artists"         "customers"        "employees"
[5] "genres"          "invoice_items"   "invoices"         "media_types"
[9] "playlist_track" "playlists"       "sqlite_sequence" "sqlite_stat1"
```

```
[13] "tracks"
```

The output from `dbListTables()` will include 13 tables. Among them will be the two tables we're going to work through in our example: `artists`, and `albums`.

The two tables we'll
work with throughout
this lesson!

```
> dbListTables(db)
[1] "albums"
[5] "genres"
[9] "playlist_track"
[13] "tracks"
[1] "artists"
[5] "invoice_items"
[9] "playlists"
[13] "customers"
[17] "invoices"
[21] "sqlite_sequence"
[25] "employees"
[29] "media_types"
[33] "sqlite_stat1"
```

output from `dbListTables(db)`

In this example, we're downloading a database and working with the data locally. However, more often, when working with SQLite databases, you'll be connecting remotely. Using the `RSSQLite` package is particularly helpful in this case because it allows you to connect to and query the database from R without reading all the data in. This is helpful in the case of very large databases, where you'll want to avoid copying all the data and will instead want to only work with the parts of the database you need.

Working with Relational Data: `dplyr` & `dbplyr`

To access these tables within R, we'll have to install the packages `dbplyr`, which enables us to access the parts of the database we're going to be working with. The `dbplyr` package allows you to use the same functions that you learned about and will learn about when working with `dplyr`; however, it allows you to use these functions with a *database*. While `dbplyr` has to be loaded to work with databases, you likely won't notice that you're using it beyond that. Otherwise, you'll just work with the files as if you were working with `dplyr` functions!

After installing and loading `dbplyr`, we'll be able to use the helpful `tbl()` function to extract the two tables we're interested in working with!

```
## install and load packages
# install.packages("dbplyr")
library(dbplyr)
library(dplyr)

## get two tables
albums <- tbl(db, "albums")
artists <- tbl(db, "artists")
```

Mutating Joins

Mutating joins allow you to take two different tables and combine the variables from both tables. This requires that each table have a column relating the tables to one another (i.e. a unique identifier). This unique identifier is used to match observations between the tables.

However, when combining tables, there are a number of different ways in which the tables can be joined:

- Inner Join - only keep observations found in *both* x *and* y
- Left Join - keep all observations in x
- Right Join - keep all observations in y
- Full Join - keep *any* observations in x *or* y

Let's break down exactly what we mean by this using just a small toy example from the `artists` and `albums` tables from the company database. Here you see three rows from the `artists` table and four rows from the `albums` table.

artists	
ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith

albums		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
6	Jagged Little Pill	4

○

small parts of the `albums` and `artist` tables

Inner Join

When talking about inner joins, we are only going to keep an observation if it is found in all of the tables we're combining. Here, we're combining the tables based on the `ArtistId` column. In our dummy example, there are only two artists that are found in *both* tables. These are highlighted in green and will be the rows used to join the two tables. Then, once the inner join happens, only these artists' data will be included after the inner join.

Inner Join

artists	
ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith

albums		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
6	Jagged Little Pill	4



inner join output will include any observation found in both tables

In our toy example, when doing an `inner_join()`, data from any observation found in all the tables being joined are included in the output. Here, ArtistIds “1” and “2” are in both the `artists` and `albums` tables. Thus, those will be the only `ArtistIds` in the output from the inner join.

And, since it’s a mutating join, our new table will have information from both tables! We now have `ArtistId`, `Name`, `AlbumId`, *and* `Title` in a single table! We’ve joined the two tables, based on the column `ArtistId`!

Inner Join: include any row in both tables

artists	
ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith

albums		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
6	Jagged Little Pill	4

`inner_join()`

ArtistId	Name	AlbumId	Title
1	AC/DC	1	For Those About To Rock We Salute You
2	Accept	2	Balls to the Wall
2	Accept	3	Restless and Wild



inner join includes observations found in both tables

Throughout this lesson we will use the coloring you see here to explain the joins, so we want to explain it explicitly here. Green cells are cells that will be used to make the merge happen and will be included in the resulting merged table. Blue cells are information that comes from the `artists` table that will be included after the merge. Red cells are pieces of information that come from the `albums` table that will be included after the merge. Finally, cells that are left white in the `artists` or `albums` table are cells that will not be included in the merge while cells that are white *after* the merge are NAs that have been added as a result of the merge.

Now, to run this for our tables from the database, rather than just for a few rows in our toy example, you would do the following:

```
## do inner join
inner <- inner_join(artists, albums, by = "ArtistId")

## look at output as a tibble
as_tibble(inner)
# A tibble: 347 × 4
  ArtistId Name          AlbumId Title
    <int> <chr>        <int> <chr>
1       1 AC/DC           1 For Those About To Rock We Salute You
2       2 Accept           2 Balls to the Wall
3       2 Accept           3 Restless and Wild
4       1 AC/DC           4 Let There Be Rock
5       3 Aerosmith        5 Big Ones
6       4 Alanis Morissette 6 Jagged Little Pill
7       5 Alice In Chains   7 Facelift
8       6 Antônio Carlos Jobim 8 Warner 25 Anos
9       7 Apocalyptica      9 Plays Metallica By Four Cellos
10      8 Audioslave         10 Audioslave
# ... with 337 more rows
```

Left Join

For a left join, all rows in the first table specified will be included in the output. Any row in the second table that is *not* in the first table will not be included.

In our toy example this means that ArtistIDs 1, 2, and 3 will be included in the output; however, ArtistID 4 will not.

Left Join

artists	
ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith

albums		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
6	Jagged Little Pill	4

left join will include all observations found in the first table specified

Thus, our output will again include all the columns from both tables combined into a single table; however, for ArtistId 3, there will be NAs for AlbumId and Title. NAs will be filled in for any observations in the first table specified that are missing in the second table.

Left Join: include all rows in first table



left join will fill in NAs

Now, to run this for our tables from the database, rather than just for a few rows in our toy example, you would do the following:

```
## do left join
left <- left_join(artists, albums, by = "ArtistId")

## look at output as a tibble
as_tibble(left)
# A tibble: 418 × 4
  ArtistId Name          AlbumId Title
    <int> <chr>        <int> <chr>
1       1 AC/DC           1 For Those About To Rock We Salute You
2       1 AC/DC           4 Let There Be Rock
3       2 Accept           2 Balls to the Wall
4       2 Accept           3 Restless and Wild
5       3 Aerosmith        5 Big Ones
6       4 Alanis Morissette 6 Jagged Little Pill
7       5 Alice In Chains   7 Facelift
8       6 Antônio Carlos Jobim 8 Warner 25 Anos
```

```
9      6 Antônio Carlos Jobim      34 Chill: Brazil (Disc 2)
10     7 Apocalyptica          9 Plays Metallica By Four Cellos
# ... with 408 more rows
```

Right Join

Right Join is similar to what we just discussed; however, in the output from a right join, all rows in the final table specified are included in the output. NAs will be included for any observations found in the last specified table but not in the other tables.

In our toy example, that means, information about ArtistIDs 1, 2, and 4 will be included.

Right Join

artists		albums	
ArtistId	Name	AlbumId	ArtistId
1	AC/DC	1	1
2	Accept	2	2
3	Aerosmith	3	2
		6	4

○

right join will include all observations found in the last table specified

Again, in our toy example, we see that `right_join()` combines the information across tables; however, in this case, ArtistId 4 is included, but Name is an NA, as this information was not in the `artists` table for this artist.

Right Join: include all rows in 2nd table

artists	
ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith

albums		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
6	Jagged Little Pill	4

right_join()

ArtistId	Name	AlbumId	Title
1	AC/DC	1	For Those About To Rock We Salute You
2	Accept	2	Balls to the Wall
2	Accept	3	Restless and Wild
4	NA	6	Jagged Little Pill



right join will fill in NAs

Now, to run this for our tables from the database, you would have to do something *slightly* different than what you saw above. Note in the code below that we have to change the class of the tables from the database into tibbles before doing the join. This is because SQL does not currently support right or full joins, but dplyr does. Thus, we first have to be sure the data are a class that dplyr can work with using `as_tibble()`. Other than that, the code below is similar to what you've seen already:

```
## do right join
right <- right_join(as_tibble(artists), as_tibble(albums), by = "ArtistId")

## look at output as a tibble
as_tibble(right)
# A tibble: 347 × 4
  ArtistId Name          AlbumId Title
    <int> <chr>        <int> <chr>
1       1 AC/DC           1 For Those About To Rock We Salute You
2       1 AC/DC           4 Let There Be Rock
3       2 Accept           2 Balls to the Wall
4       2 Accept           3 Restless and Wild
```

```

5      3 Aerosmith          5 Big Ones
6      4 Alanis Morissette  6 Jagged Little Pill
7      5 Alice In Chains    7 Facelift
8      6 Antônio Carlos Jobim 8 Warner 25 Anos
9      6 Antônio Carlos Jobim 34 Chill: Brazil (Disc 2)
10     7 Apocalyptica       9 Plays Metallica By Four Cellos
# ... with 337 more rows

```

While the output may look similar to the output from `left_join()`, you'll note that there are a different number of rows due to how the join was done. The fact that 347 rows are present with the right join and 418 were present after the left join suggests that there are artists in the artists table without albums in the albums table.

Full Join

Finally, a full join will take every observation from every table and include it in the output.

Full Join

artists	
ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith

albums		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
6	Jagged Little Pill	4

○

full join will include any observation found in either table

Thus, in our toy example, this join produces five rows, including all the observations from either table. NAs are filled in when data are missing for an observation.

Full Join: include any row in either table



full join will fill in NAs

As you saw in the last example, to carry out a full join, we have to again specify that the objects are tibbles before being able to carry out the join:

```
## do right join
full <- full_join(as_tibble(artists), as_tibble(albums), by = "ArtistId")

## look at output as a tibble
as_tibble(full)
# A tibble: 418 × 4
  ArtistId Name      AlbumId Title
    <int> <chr>     <int> <chr>
1       1 AC/DC      1 For Those About To Rock We Salute You
2       1 AC/DC      4 Let There Be Rock
3       2 Accept     2 Balls to the Wall
4       2 Accept     3 Restless and Wild
5       3 Aerosmith  5 Big Ones
6       4 Alanis Morissette 6 Jagged Little Pill
7       5 Alice In Chains 7 Facelift
8       6 Antônio Carlos Jobim 8 Warner 25 Anos
```

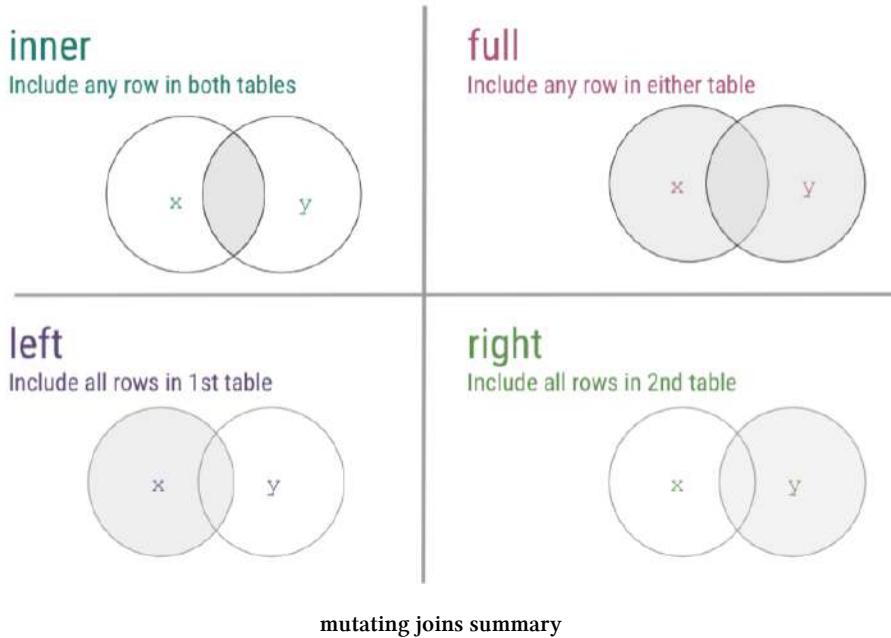
```

9          6 Antônio Carlos Jobim      34 Chill: Brazil (Disc 2)
10         7 Apocalyptica           9 Plays Metallica By Four Cellos
# ... with 408 more rows

```

Mutating Joins Summary

Now that we've walked through a number of examples of mutating joins, cases where you're combining information across tables, we just want to take a second to summarize the four types of joins discussed using a visual frequently used to explain the most common mutating joins where each circle represents a different table and the gray shading on the Venn diagrams indicates which observations will be included after the join.



To see a visual representation of this, there is a great resource on GitHub, where these joins are illustrated, so feel free to check out [this link](#) from Garrick Aden-Buie animating joins within relational data.

Filtering Joins

While we discussed mutating joins in detail, we're just going to mention the ability to carry out filtering joins. While mutating joins combined variables across tables, **filtering joins**

affect the observations, not the variables. This *still* requires a unique identifier to match the observations between tables.

Filtering joins keep observations in one table based on the observations present in a second table. Specifically:

- `semi_join(x, y)` : keeps all observations in `x` with a match in `y`.
 - `anti_join(x, y)` : keeps observations in `x` that do *NOT* have a match in `y`.

In our toy example, if the join `semi_join(artists, albums)` were run, this would keep rows of `artists` where the `ArtistID` in `artist` was also in the `albums` table.

Semi Join

Alternatively, `anti_join(artists, albums)` would output the rows of `artists` whose `ArtistId` was *NOT* found in the `albums` table.

Anti Join

Note that in the case of filtering joins, the number of variables in the table *after* the join does not change. While **mutating joins** merged the tables creating a resulting table with more columns, with **filtering joins** we're simply filtering the observations in one table based on the values in a second table.

How to Connect to a Database Online

As mentioned briefly above, most often when working with databases, you won't be downloading the entire database. Instead, you'll connect to a server somewhere else where the data live and query the data (search for the parts you need) from R.

For example, in this lesson we downloaded the entire `company` database, but only ended up using `artists` and `albums`. In the future, instead of downloading *all* the data, you'll just connect to the database and work with the parts you need.

This will require connecting to the database with host, user, and password. This information will be provided by the database's owners, but the syntax for entering this information into R to connect to the database would look something like what you see here:

```
## This code is an example only
con <- DBI::dbConnect(RMySQL::MySQL(),
  host = "database.host.com",
  user = "janeeverydaydoe",
  password = rstudioapi::askForPassword("database_password"))
)
```

While not being discussed in detail here, it's important to know that connecting to remote databases from R is possible and that this allows you to query the database without reading *all* the data from the database into R.

Web Scraping

We've mentioned previously that there is a lot of data on the Internet, which probably comes at no surprise given the vast amount of information on the Internet. Sometimes these data are in a nice CSV format that we can quickly pull from the Internet. Sometimes, the data are spread across a web page, and it's our job to "scrape" that information from the webpage and get it into a usable format. Knowing first that this is possible within R and second, having some idea of where to start is an important start to beginning to get data from the Internet.

We'll walk through three R packages in this lesson to help get you started in getting data from the Internet. Let's transition a little bit to talking about how to pull pieces of data from a website, when the data aren't (yet!) in the format that we want them.

Say you wanted to start a company but did not know exactly what people you would need. We could go to the websites of a bunch of companies similar to the company you hope to start and pull off all the names and titles of the people working there. You then compare the titles across companies and voila, you've got a better idea of who you'll need at your new company.

You could imagine that while this information may be helpful to have, getting it manually would be a pain. Navigating to each site individually, finding the information, copying and pasting each name. That sounds awful! Thankfully, there's a way to scrape the web from R directly!

A very helpful package `rvest` can help us do this. It gets its name from the word "harvest." The idea here is you'll use this package to "harvest" information from websites! However, as you may imagine, this is less straightforward than pulling data that are already formatted the way you want them (as we did previously), since we'll have to do some extra work to get everything in order.

rvest Basics

When `rvest` is given a webpage (URL) as input, an `rvest` function reads in the HTML code from the webpage. HTML is the language websites use to display everything you see on the website. Generally, all HTML documents require each webpage to have a similar structure. This structure is specified by using different **tags**. For example, a header at the top of your webpage would use a specific tag. Website links would use a different tag. These different tags help to specify how the website should appear. The `rvest` package takes advantage of these tags to help you extract the parts of the webpage you're most interested in. So let's see exactly how to do that all of this with an example.

The screenshot shows a web page titled "Web Scraping". A pink arrow points from the text "Different parts of the webpage have different HTML tags" to the heading "R Packages : Data". Another pink arrow points from the same text to the first row of a table. The table lists five R packages used for web scraping, their purposes, and URLs. The URL at the bottom of the page is http://jhubdatascience.org/stable_website/webscrape.html.

Package	Purpose	URL
<code>rvest</code>	Web Scraping	https://rvest.tidyverse.org/
<code>httr</code>	Working with APIs	https://httr.r-lib.org/
<code>dbplyr</code>	Working with databases (SQL)	https://dbplyr.tidyverse.org/
<code>jsonlite</code>	Parsing JSON	https://github.com/jeroen/jsonlite
<code>googlesheets</code>	Working with Google Sheets	https://github.com/jennybc/googlesheets

Different tags are used to specify different parts of a website

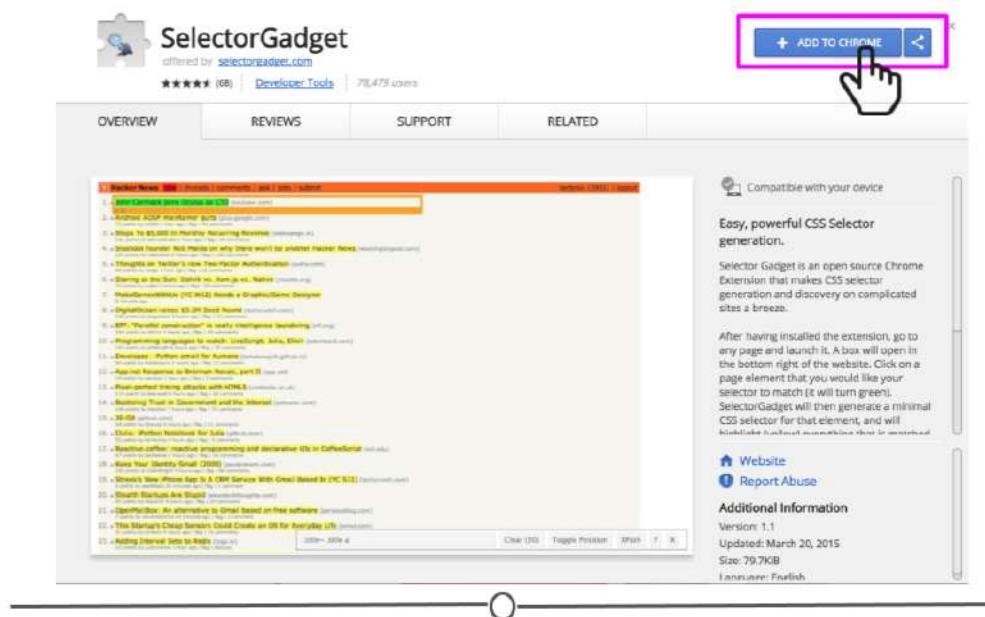
SelectorGadget

To use `rvest`, there is a tool that will make your life *a lot* easier. It's called SelectorGadget. It's a “javascript bookmarklet.” What this means for us is that we'll be able to go to a webpage, turn on SelectorGadget, and help figure out how to appropriately specify what components from the webpage we want to extract using `rvest`.

To get started using SelectorGadget, you'll have to enable the Chrome Extension.

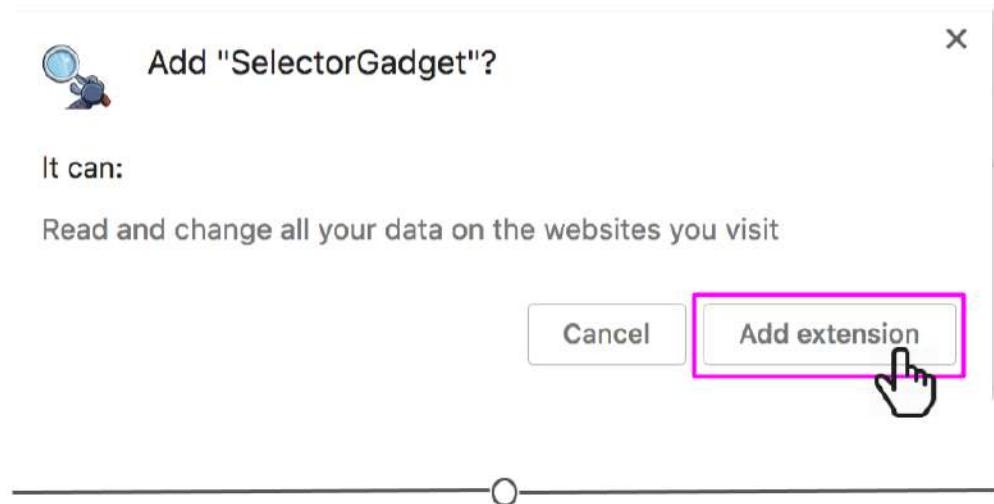
To enable SelectorGadget using Google Chrome:

1. Click [here](#) to open up the SelectorGadget Chrome Extension
2. Click “ADD TO CHROME”



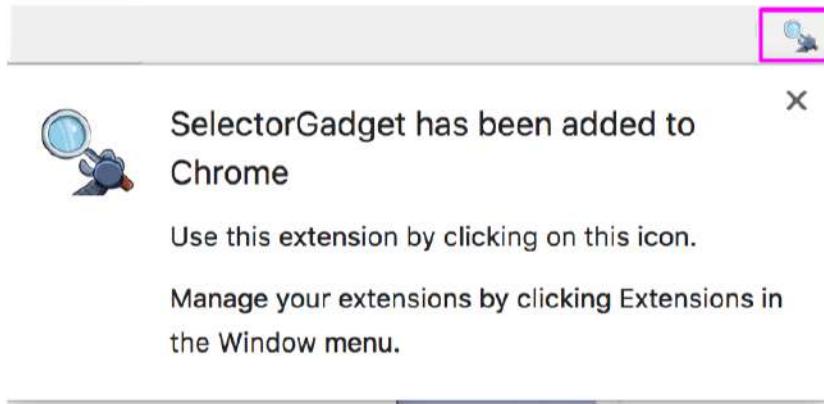
ADD TO CHROME

1. Click “Add extension”



Add extension

1. SelectorGadget's icon will now be visible to the right of the web address bar within Google Chrome. You will click on this to use SelectorGadget in the example below.



SelectorGadget icon

Web Scraping Example

Similar to the example above, what if you were interested in knowing a few recommended R packages for working with data? Sure, you could go to a whole bunch of websites and Google and copy and paste each one into a Google Sheet and have the information. But, that's not very fun!

Alternatively, you could write and run a few lines of code and get all the information that way! We'll do that in the example below.

Using SelectorGadget

To use SelectorGadget, navigate to the webpage we're interested in scraping: <http://jhudatascience.org/static/website/webscrape.html> and toggle SelectorGadget by clicking on the SelectorGadget icon. A menu at the bottom-right of your web page should appear.

Not Secure | jhudatascience.org/stable_website/webscrape.html

Home Web Scraping FAQs

Web Scraping

Web Scraping is an incredibly helpful tool when gathering and analyzing data from the Internet. However, one thing to keep in mind is the fact that websites are not static. Information gets added to and removed from them; they can be completely re-designed, and sometimes they completely disappear! For this reason it's important to record when you web scraped your information and store the data in a stable format (i.e. CSV, JSON, or XML) after you do your web scraping. This way you don't risk losing the information if the website changes *and* you're only doing the web scraping once. You want to minimize the number of times you scrape information.

Also, we've made this website due to stability issues. This website has been generated for the lesson you're currently working on! By using this website, we know that the content is stable and won't change...unless we decide to change it.

R Packages : Data

Package	Purpose	URL
<code>rvest</code>	Web Scraping	https://rvest.tidyverse.org/
<code>httr</code>	Working with APIs	https://httr.lib.org/
<code>dbplyr</code>	Working with databases (SQL)	https://dbplyr.tidyverse.org/
<code>jsonlite</code>	Parsing JSON	https://github.com/jason/jsonlite
<code>googlesheets</code>	Working with Google Sheets	https://github.com/jennybc/googlesheets

No valid path found. Clear Toggle Position XPath ? X

http://jhudatascience.org/stable_website/webscrape.html

SelectorGadget icon on webpage of interest

Now that SelectorGadget has been toggled, as you mouse over the page, colored boxes should appear. We'll click on the name of the first package to start to tell SelectorGadget which component of the webpage we're interested in.

Web Scraping

Web Scraping is an incredibly helpful tool when gathering and analyzing data from the Internet. However, one thing to keep in mind is the fact that websites are *not* static. Information gets added to and removed from them, they can be completely re-designed, and sometimes they completely disappear! For this reason it's important to record when you web scraped your information and store the data in a stable format (i.e. CSV, JSON, or XML) after you do your web scraping. This way you don't risk losing the information if the website changes *and* you're only doing the web scraping once. You want to minimize the number of times you scrape information.

Also, we've made this website due to stability issues. This website has been generated for the lesson you're currently working on! By using this website, we know that the content is static and won't change...unless we decide to change it.

R Packages : Data

Package	Purpose	URL
rvest	Web Scraping	https://rvest.tidyverse.org/
httr	Working with APIs	https://http-tidyverse.org/
dbplyr	Working with databases (SQL)	https://dbplyr.tidyverse.org/
jsonlite	Parsing JSON	https://github.com/r-lib/jsonlite
googlesheets	Working with Google Sheets	https://github.com/tidyverse/googlesheets

Text you'll use to specify which part of the webpage you'd like to scrape

strong

Clear (S) Toggle Position XPath ? X

http://jhudatascience.org/stable_website/webscrape.html

SelectorGadget selects `strong` on webpage of interest

An orange box will appear around the component of the webpage you've clicked. Other components of the webpage that SelectorGadget has deemed similar to what you've clicked will be highlighted. And, text will show up in the menu at the bottom of the page letting you know what you should use in `rvest` to specify the part of the webpage you're most interested in extracting.

Here, we see with that SelectorGadget has highlighted the package names and nothing else! Perfect. That's just what we wanted. Now we know how to specify this element in `rvest`!

Using `rvest`

Now we're ready to use `rvest`'s functions. First, we'll use `read_html()` (which actually comes from the `xml2` package) to read in the HTML from our webpage of interest.

We'll then use `html_nodes()` to specify which parts of the webpage we want to extract. Within this function we specify "strong", as that's what SelectorGadget told us to specify to "harvest" the information we're interested in.

Finally `html_text()` extracts the text from the tag we've specified, giving us that list of packages we wanted to see!

```
## load package
# install.packages("rvest")
library(rvest) # this loads the xml2 package too!

Attaching package: 'rvest'
The following object is masked from 'package:readr':
guess_encoding

## provide URL
packages <- read_html("http://jhudatascience.org/stable_website/webscrape.html" \
) # the function is from xml2

## Get Packages
packages %>%
  html_nodes("strong") %>%
  html_text()
[1] "rvest"          "httr"           "dbplyr"         "jsonlite"       "googlesheets"
```

With just a few lines of code we have the information we were looking for!

A final note: SelectorGadget

SelectorGadget selected what we were interested in on the first click in the example above. However, there will be times when it makes its guess and highlights more than what you want to extract. In those cases, after the initial click, click on any one of the items currently highlighted that you don't want included in your selection. SelectorGadget will mark that part of the webpage in red and update the menu at the bottom with the appropriate text. To see an example of this, watch this short video [here](#).

APIs

Application Programming Interfaces (APIs) are, in the most general sense, software that allow different web-based applications to communicate with one another over the Internet. Modern APIs conform to a number of standards. This means that many different applications are using the same approach, so a single package in R is able to take advantage of this and

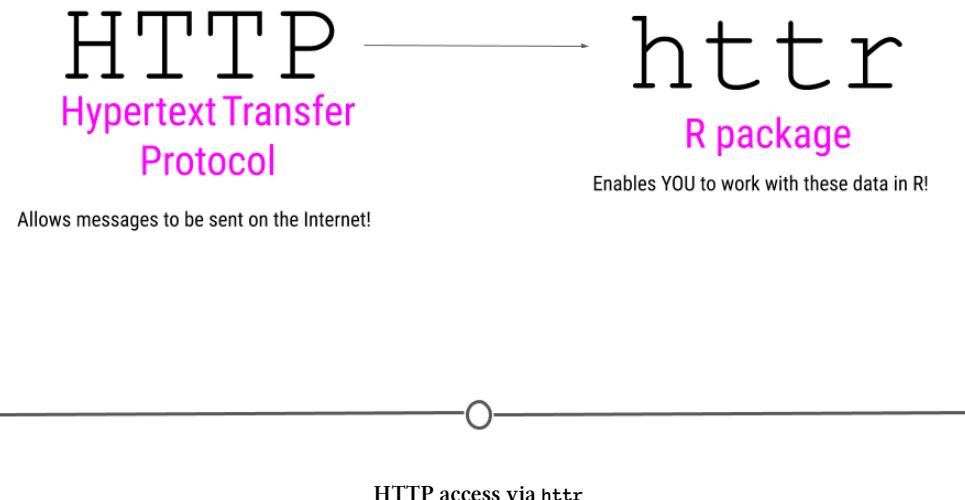
communicate with many different applications, as long as the application’s API adheres to this generally agreed upon set of “rules”.

The R package that we’ll be using to acquire data and take advantage of this is called `httr`. This package name suggests that this is an “R” package for “HTTP”. So, we know what R is, but what about HTTP?

You’ve probably seen HTTP before at the start of web addresses, (ie `http://www.gmail.com`), so you may have some intuition that HTTP has something to do with the Internet, which is absolutely correct! HTTP stands for Hypertext Transfer Protocol. In the broadest sense, HTTP transactions allow for messages to be sent between two points on the Internet. You, on your computer can request something from a web page, and the protocol (HTTP) allows you to connect with that webpage’s server, do something, and then return you whatever it is you asked for.

Working with a web API is similar to accessing a website in many ways. When you type a URL (ie `www.google.com`) into your browser, information is sent from your computer to your browser. Your browser then interprets what you’re asking for and displays the website you’ve requested. Web APIs work similarly. You **request** some information from the API and the API sends back a **response**.

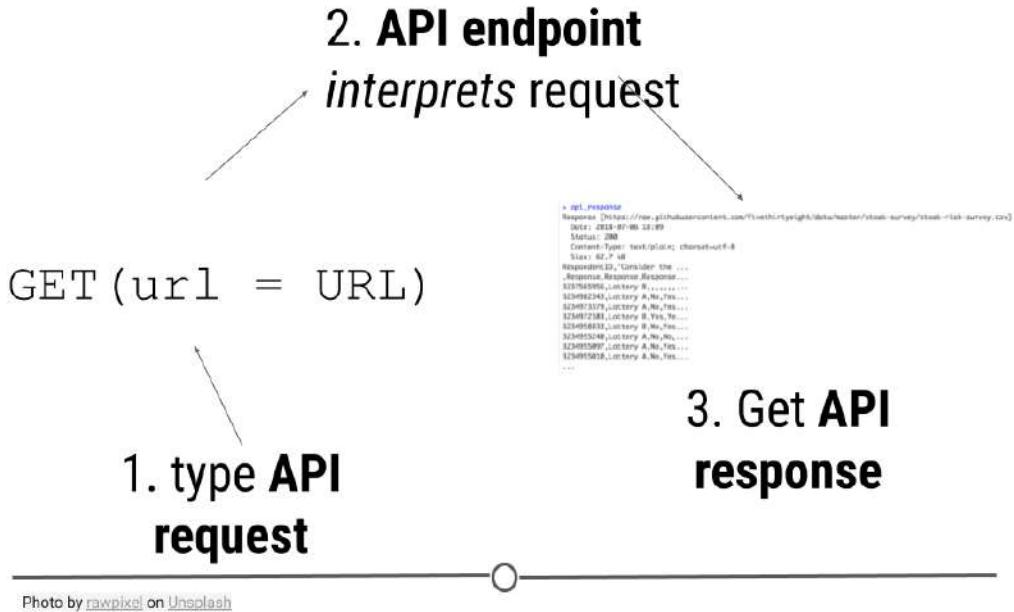
The `httr` package will help you carry out these types of requests within R. Let’s stop talking about it, and see an actual example!



Getting Data: `httr`

HTTP is based on a number of important verbs: `GET()`, `HEAD()`, `PATCH()`, `PUT()`, `DELETE()`, and `POST()`. For the purposes of retrieving data from the Internet, you may be able to guess which verb will be the most important for our purposes! `GET()` will allow us to *fetch* a resource that already exists. We'll specify a URL to tell `GET()` where to go look for what we want. While we'll only highlight `GET()` in this lesson, for full understanding of the many other HTTP verbs and capabilities of `httr`, refer to the additional resources provided at the end of this lesson.

`GET()` will access the API, provide the API with the necessary information to request the data we want, and retrieve some output.



API requests are made to an API endpoint to get an API response

Example 1: GitHub's API

The example is based on a wonderful [blogpost](#) from [Tyler Clavelle](#). In this example, we'll use will take advantage of GitHub's API, because it's accessible to anyone. Other APIs, while often freely-accessible, require credentials, called an **API key**. We'll talk about those later, but let's just get started using [GitHub's API](#) now!

API Endpoint

The URL you're requesting information from is known as the **API endpoint**. The documentation from GitHub's API explains what information can be obtained from their API endpoint: <https://api.github.com>. That's the base endpoint, but if you wanted to access a particular individual's GitHub repositories, you would want to modify this base endpoint to: <https://api.github.com/users/username/repos>, where you would replace *username* with your GitHub username.

API request: GET()

Now that we know what our API endpoint is, we're ready to make our **API request** using `GET()`.

The goal of this request is to obtain information about what repositories are available in *your* GitHub account. To use the example below, you'll want to change the username `janeeverydaydoe` to your GitHub username.

```
## load package
library(httr)
library(dplyr)

## Save GitHub username as variable
username <- 'janeeverydaydoe'

## Save base endpoint as variable
url_git <- 'https://api.github.com/'

## Construct API request
api_response <- GET(url = paste0(url_git, 'users/', username, '/repos'))
```

Note: In the code above, you see the function `paste0()`. This function concatenates (links together) each the pieces within the parentheses, where each piece is separated by a comma. This provides `GET()` with the URL we want to use as our endpoints!

```
## load package
library(httr)
library(dplyr)

## Save GitHub username as variable
username <- 'janeeverydaydoe'

## Save base endpoint as variable
url_git <- 'https://api.github.com/' API endpoint

## Construct API request
api_response <- GET(url = paste0(url_git,
'users/', username, '/repos')) API request
```

httr code to access GitHub

API response: content()

Let's first take a look at what other variables are available within the `api_response` object:

```
## See variables in response
names(api_response)
[1] "url"          "status_code"   "headers"      "all_headers"  "cookies"
[6] "content"      "date"         "times"        "request"     "handle"
```

```
## See variables in response
names(api_response)
```



```
> names(api_response)
[1] "url"          "status_code"   "headers"      "all_headers"  "cookies"     "content"
[7] "date"         "times"        "request"     "handle"
```

What API request from
httr returns



httr response

While we see ten different variables within `api_response`, we should probably first make sure that the request to GitHub's API was successful. We can do this by checking the status code of the request, where “200” means that everything worked properly:

```
## Check Status Code of request
api_response$status_code
[1] 200
```

But, to be honest, we aren't really interested in just knowing the request worked. We actually want to see what information is contained on our GitHub account.

To do so we'll take advantage of `httr`'s `content()` function, which as its name suggests, extracts the contents from an API request.

```
## Extract content from API response
repo_content <- content(api_response)
```

```

## Check Status Code of request
api_response$status_code

## Extract content from API response
repo_content <- content(api_response)

> api_response$status_code
[1] 200
'200' means request was successful!

> repo_content <- content(api_response)
content() extracts contents from API request

```

httr status code and content()

You can see here that the length of `repo_content` in our case is 6 by looking at the Environment tab. This is because the GitHub account `janeeverydaydoe` had six repositories at the time of this API call. We can get some information about each repo by running the function below:

```

## function to get name and URL for each repo
lapply(repo_content, function(x) {
  df <- data_frame(repo = x$name,
                    address = x$html_url)}) %>%
  bind_rows()
Warning: `data_frame()` was deprecated in tibble 1.1.0.
Please use `tibble()` instead.
# A tibble: 8 × 2
  repo                  address
  <chr>                <chr>
1 cbds                 https://github.com/JaneEverydayDoe/cbds
2 first_project         https://github.com/JaneEverydayDoe/first_pro...
3 gcd                  https://github.com/JaneEverydayDoe/gcd
4 hello-world          https://github.com/JaneEverydayDoe/hello-wor...

```

```
5 janeeverydaydoe.github.com      https://github.com/JaneEverydayDoe/janeevery...
6 my_first_project                https://github.com/JaneEverydayDoe/my_first...
7 newproject                      https://github.com/JaneEverydayDoe/newproject
8 Temporary_add_to_version_control https://github.com/JaneEverydayDoe/Temporary...

## function to get name and URL for each repo
lapply(repo_content, function(x) {
  df <- data_frame(repo = x$name,
                    address = x$html_url)) %>%
  bind_rows()

> lapply(repo_content, function(x) {
+   df <- data_frame(repo = x$name,
+                     address = x$html_url)}) %>%
+   bind_rows()
# A tibble: 6 x 2
  repo                  address
  <chr>                <chr>
1 first_project        https://github.com/JaneEverydayDoe/first_project
2 hello-world          https://github.com/JaneEverydayDoe/hello-world
3 janeeverydaydoe.github.com https://github.com/JaneEverydayDoe/janeeverydaydoe.g..
4 my_first_project     https://github.com/JaneEverydayDoe/my_first_project
5 newproject           https://github.com/JaneEverydayDoe/newproject
6 Temporary_add_to_version_control https://github.com/JaneEverydayDoe/Temporary_add_to_...
```



output from API request

Here, we've pulled out the name and URL of each repository in Jane Doe's account; however, there is *a lot* more information in the `repo_content` object. To see how to extract more information, check out the rest of Tyler's wonderful post [here](#).

Example 2: Obtaining a CSV

This same approach can be used to download datasets directly from the web. The data for this example are available for download from this link: data.fivethirtyeight.com, but are also hosted on GitHub [here](#), and we will want to use the specific URL for this file: <https://raw.githubusercontent.com/fivethirtyeight/data/master/steak-survey/steak-risk-survey.csv> in our `GET()` request.

The screenshot shows a GitHub repository page for the 'fivethirtyeight / data' organization. The 'steak-survey' folder is selected. The README files contain the following content:

```
Steak Survey

This folder contains data behind the stories:
• How Americans Like Their Steak.
• How Americans Order Their Steak.
```

<https://github.com/fivethirtyeight/data/tree/master/steak-survey>

steak-survey on GitHub

To do so, we would do the following:

```
## Make API request
api_response <- GET(url = "https://raw.githubusercontent.com/fivethirtyeight/data/master/steak-survey/steak-risk-survey.csv")

## Extract content from API response
df_steak <- content(api_response, type="text/csv")
```

```

> ## Make API request
> api_response <- GET(url = "https://raw.githubusercontent.com/fivethirtyeight/data/master/steak-survey/steak-risk-survey.csv")
>
> ## Extract content from API response
> df_stea<- content(api_response, type="text/csv")  

No encoding supplied: defaulting to UTF-8. content() extracts information
Parsed with column specification:  

cols(
  RespondentID = col_double(),
  `Consider the following hypothetical situations: <br>In Lottery A, you have a 50% chance of success, with a payout of $100. <br>In Lottery B, you have a 90% chance of success, with a pay out of $20. <br><br>Assuming you have $10 to bet, would you play Lottery A or Lottery B?` = co
l_character(),
  `Do you ever smoke cigarettes?` = col_character(),
  `Do you ever drink alcohol?` = col_character(),
  `Do you ever gamble?` = col_character(),
  `Have you ever been skydiving?` = col_character(),
  `Do you ever drive above the speed limit?` = col_character(),
  `Have you ever cheated on your significant other?` = col_character(),
  `Do you eat steak?` = col_character(),
  `How do you like your steak prepared?` = col_character(),
  Gender = col_character(),
  Age = col_character(),
  `Household Income` = col_character(),
  Education = col_character(),
  `Location (Census Region)` = col_character()
)

```

GET() steak-survey CSV

Here, we again specify our url within `GET()` followed by use of the helpful `content()` function from `httr` to obtain the CSV from the `api_response` object. The `df_stea`k includes the data from the CSV directly from the GitHub API, without having to download the data first!

`read_csv()` from a URL

Before going any further, we'll note that these data are in the CSV format and that the `read_csv()` function can read CSVs directly from a URL:

```
#use readr to read in CSV from a URL
df <- read_csv("https://raw.githubusercontent.com/fivethirtyeight/data/master/s\teak-survey/steak-risk-survey.csv")
```

As this is a simpler approach than the previous example, you'll want to use this approach when reading CSVs from URL. However, you won't always have data in the CSV format, so we wanted to be sure to demonstrate how to use `httr` when obtaining information from URLs using HTTP methods.

API keys

Not all API's are as "open" as GitHub's. For example, if you ran the code for the first example above exactly as it was written (and didn't change the GitHub username), you would have gotten information about the repos in janeeverydaydoe's GitHub account. Because it is a fully-open API, you're able to retrieve information about not only your GitHub account, but also other users' **public** GitHub activity. This makes good sense because sharing code among public repositories is an important part of GitHub.

Alternatively, while Google also has an API (or rather, *many* API's), they aren't quite as open. This makes good sense. There is no reason why one should have access to the files on someone else's Google Drive account. Controlling whose files one can access through Google's API is an important privacy feature.

In these cases, what is known as a key is required to gain access to the API. **API keys** are obtained from the website's API site (ie, for Google's APIs, you would start [here](#)). Once acquired, these keys should **never be shared on the Internet**. There is a reason they're required, after all. So, be sure to **never push a key to GitHub or share it publicly**. (If you do ever accidentally share a key on the Internet, return to the API and disable the key immediately.)

For example, to access the Twitter API, you would obtain your key and necessary tokens from [Twitter's API](#) and replace the text in the `key`, `secret`, `token` and `token_secret` arguments below. This would authenticate you to use Twitter's API to acquire information about your home timeline.

```
myapp = oauth_app("twitter",
                  key = "yourConsumerKeyHere",
                  secret = "yourConsumerSecretHere")
sig = sign_oauth1.0(myapp,
                    token = "yourTokenHere",
                    token_secret = "yourTokenSecretHere")
homeTL = GET("https://api.twitter.com/1.1/statuses/home_timeline.json", sig)
```

Foreign Formats

haven

Perhaps you or your collaborators use other types of statistical software such as [SAS](#), [SPSS](#), and [Stata](#). Files supported by these software packages can be imported into R and exported from R using the `haven` package.

As an example, we will first write files for each of these software packages and then read them. The data needs to be in a data frame format and spaces and punctuation in variable names will cause issues. We will use the Toy Story character data frame that we created earlier for this example. Note that we are using the `here` package that was described in the introduction to save our files in a directory called `data` which is a subdirectory of the directory in which the `.Rproj` file is located.

```
#install.packages("haven")
library(haven)

## SAS
write_sas(data = mydf, path = here::here("data", "mydf.sas7bdat"))
# read_sas() reads .sas7bdat and .sas7bcat files
sas_mydf <- read_sas(here::here("data", "mydf.sas7bdat"))
sas_mydf
# A tibble: 3 × 3
  Name      Age Occupation
  <chr>    <dbl> <chr>
1 Woody      40 Sheriff
2 Buzz Lightyear  34 Space Ranger
3 Andy        NA Toy Owner
# use read_xpt() to read SAS transport files (version 5 and 8)
```

We can also write the data frame to SPSS format.

```
## SPSS
write_sav(data = mydf, path = here::here("data", "mydf.sav"))
# use to read_sav() to read .sav files
sav_mydf <- read_sav(here::here("data", "mydf.sav"))
sav_mydf
# A tibble: 3 × 3
  Name          Age Occupation
  <chr>        <dbl> <chr>
1 Woody         40   Sheriff
2 Buzz Lightyear 34   Space Ranger
3 Andy          NA   Toy Owner
# use read_por() to read older .por files
```

Stata format is also supported.

```
## Stata
write_dta(data = mydf, path = here::here("data", "mydf.dta"))
# use to read_dta() to read .dta files
dta_mydf <- read_dta(here::here("data", "mydf.dta"))
dta_mydf
# A tibble: 3 × 3
  Name          Age Occupation
  <chr>        <dbl> <chr>
1 Woody         40   Sheriff
2 Buzz Lightyear 34   Space Ranger
3 Andy          NA   Toy Owner
```

When exporting and importing to and from all foreign statistical formats it's important to realize that the conversion will generally be less than perfect. For simple data frames with numerical data, the conversion should work well. However, when there are a lot of missing data, or different types of data that perhaps a given statistical software package may not recognize, it's always important to check the output to make sure it contains all of the information that you expected.

Images

Only a few decades ago, analyzing a large dataset of images was not feasible for most researchers. Many didn't even think of images as data. But, there is so much we can get

from analyzing image data. Although we will not study images processing techniques in this lesson, let's look at one example that gives us an idea of how image data can be used.

Within Google Maps there is a Street View feature that allows panoramic views from positions along many streets in the world. One of the things you may notice if you're looking around on Google Maps' street view is that for many streets in the world you do not only see houses; you are also able to see cars.



Source: Google

Google Maps street view

Some 50 million images of cars from over 200 cities were used by researchers to detect the make, model, body type, and age of the cars in each neighborhood. They were able to take unstructured image data and compile a structured dataset! These same researchers then pulled together a structured dataset from the Census and the 2008 elections of demographic information (such as race and income), and voting history in these same neighborhoods.



Source: Demography with deep learning and street view, Gebru et al.

Data used from Google Maps street view to predict demographics of a neighborhood

Using these two datasets (the Google Street view car image data and the demographic data), researchers used a technique known as **machine learning** to build an algorithm that could, from the images of cars in a neighborhood, predict the demographics (race, income, etc) and how that area is likely to vote. Comparing these two sets of data, they were able to accurately estimate income, race, education, and voting patterns at the zip code level from the Google Street view images.

Reading Images in R

Like with text, there are packages in R that will help you carry out analysis of images. In particular, `magick` is particularly helpful for advanced image processing within R, allowing you to process, edit, and manipulate images within R. Like JSON and XML, where there is more than one file format for a similar task, there are also a number of different image file formats. The `magick` package is capable of working with many different types of images, including PNG, JPEG, and TIFF. The `magick` package has a particularly helpful [vignette](#) where you can learn the ins and outs of working with images using `magick`'s functionality. Their documentation will discuss how to read image data into R, how to edit images, and even how to add images to your R plots!



https://cran.r-project.org/web/packages/magick/vignettes/intro.html#drawing_and_graphics

magick package's example of adding an image to a plot

A really useful manipulation that one can perform is text extraction from images. Typically this works best with images that have text, where the text is not angled and is in a conventional font.

We will show how to do this using a couple of tidyverse package hex stickers. This will involve using the `image_read()` function to import the image and the `image_ocr()` function to extract the text.

```
# install package
#install.packages("magick")
# load package
library(magick)

Linking to ImageMagick 7.1.0.2
Enabled features: fontconfig, freetype, ghostscript, heic, lcms, webp
Disabled features: cairo, fftw, pango, raw, rsvg, x11
Using 16 threads

img1 <- image_read("https://ggplot2.tidyverse.org/logo.png")
img2 <- image_read("https://pbs.twimg.com/media/D5bccHZWkAQuPqS.png")
#show the image
```

```
print(img1)
# A tibble: 1 × 7
  format width height colorspace matte filesize density
  <chr>   <int>  <int>  <chr>      <lgl>    <int>  <chr>
1 PNG       240     278  sRGB      TRUE     38516 +85x+85
```



plot of chunk unnamed-chunk-50

```
print(img2)
# A tibble: 1 × 7
  format width height colorspace matte filesize density
  <chr>   <int>  <int>  <chr>      <lgl>    <int>  <chr>
1 PNG       864     864  sRGB      TRUE     54056 +72x+72
```



plot of chunk unnamed-chunk-50

```
#concatenate and print text
cat(image_ocr(img1))
ggplot2
cat(image_ocr(img2))
parsnip
```

Great! We extracted the text!

googledrive

Another really helpful package is the `googledrive` package which enables users to interact with their Google Drive directly from R. This package, unlike the `googlesheets4` package, also allows for users to interact with other file types besides Google Sheets. It however does not allow for as many modifications of these files like the `googlesheets4` package allows for Google Sheets files.

Using `googledrive` requires having an established Google Drive account. You will be asked to authorize access for the package to interact with your Google Drive.

Finding files in your drive can be done using `drive_find()`.

```
# install.packages("googledrive")
# load package
library("googledrive")
# Files can be found based on file name words like this:
drive_find(pattern = "tidyverse")
# Files can be found based on file type like this:
drive_find(type = "googlesheets")
# Files that have specific types of visibility can be found like this:
files <- drive_find(q = c("visibility = 'anyoneWithLink'"))
```

Files can be viewed from your default browser by using the `drive_browse()` function:

```
drive_browse("tidyverse")
```

Files can be uploaded to your Google Drive using the `drive_upload()` function.

```
drive_upload(here::here("tidyverse.txt"))
```

If we wanted this file to be converted and saved as a Google file, then we would do the following based on the type of Goggle document desired. Only certain types of files can be converted to each type of Google file.

```
drive_upload(here::here("tidyverse.txt"), type = "document")
drive_upload(here::here("tidyverse.csv"), type = "spreadsheet")
drive_upload(here::here("tidyverse.pptx"), type = "presentation")
```

Files can be downloaded using the `drive_download()` function. Google file types need to be converted to a conventional file type. For example one might save a Google Sheet file to a CSV file. This would download a file called `tidyverse.csv` to your project directory. This file could then be used in an analysis.

```
drive_download("tidyverse", type = "csv")
tidyverse_data <- readr::read_csv(here("tidyverse.csv"))
```

Files can be moved to trash using the `drive_trash()` function. This can be undone using the `drive_untrash()` function.

The trash can also be emptied using `drive_empty_trash()`.

```
drive_trash("tidyverse.txt")
drive_untrash("tidyverse.txt")
drive_empty_trash()
```

To permanently remove a file you can use the `drive_rm()` function. This does not keep the file in trash.

```
drive_rm("tidyverse.txt")
```

Files can be shared using the `drive_share()` function. The sharing status of a file can be checked using `drive_reveal()`.

```
drive_share(file = "tidyverse",
            role = "commenter",
            type = "user",
            emailAddress = "someone@example.com",
            emailMessage = "Would greatly appreciate your feedback.")

drove_share_anyone(file = "tidyverse", verbose = TRUE) # anyone with link can read

drive_reveal(file = "tidyverse", what = "permissions")
```

There are many other helpful functions for interacting with the files in your Google Drive within the `googledrive` package.

Case Studies

Now we will demonstrate how to import data using our case study examples.

Case Study #1: Health Expenditures

The data for this case study are available in [CSVs hosted on GitHub](#). CSVs from URLs can be read directly using `read_csv()` from `readr` (a core tidyverse package).

As a reminder, we're ultimately interested in answering the following questions with these data:

1. Is there a relationship between health care coverage and health care spending in the United States?
2. How does the spending distribution change across geographic regions in the United States?
3. Does the relationship between health care coverage and health care spending in the United States change from 2013 to 2014?

Health Care Coverage Data

We'll first read the data in. Note that we have to skip the first two lines, as there are two lines in the CSV that store information about the file before we get to the actual data.

To see what we mean, you can always use the `read_lines()` function from `readr` to see the first 7 lines with the `n_max` argument:

```
read_lines(file = 'https://raw.githubusercontent.com/opencasestudies/ocs-health\expenditure/master/data/KFF/healthcare-coverage.csv', n_max = 7)
[1] "\"Title: Health Insurance Coverage of the Total Population | The Henry J. \
Kaiser Family Foundation\""
[2] "\"Timeframe: 2013 - 2016\""
[3] "\"Location\", \"2013_Employer\", \"2013_Non-Group\", \"2013_Medicaid\", \"2\\
013_Medicare\", \"2013_Other Public\", \"2013_Uninsured\", \"2013_Total\", \"20\\
14_Employer\", \"2014_Non-Group\", \"2014_Medicaid\", \"2014_Medicare\", \"2014\\
_Other Public\", \"2014_Uninsured\", \"2014_Total\", \"2015_Employer\", \"2015\\
_Non-Group\", \"2015_Medicaid\", \"2015_Medicare\", \"2015_Other Public\", \"201\\
5_Uninsured\", \"2015_Total\", \"2016_Employer\", \"2016_Non-Group\", \"2016_M\\
edicaid\", \"2016_Medicare\", \"2016_Other Public\", \"2016_Uninsured\", \"2016\\
_Total\""
[4] "\"United States\", \"155696900\", \"13816000\", \"54919100\", \"40876300\", \"6\\
295400\", \"41795100\", \"313401200\", \"154347500\", \"19313000\", \"61650400\", \"4\\
1896500\", \"5985000\", \"32967500\", \"316159900\", \"155965800\", \"21816500\", \"6\\
2384500\", \"43308400\", \"6422300\", \"28965900\", \"318868500\", \"157381500\", \"2\\
1884400\", \"62303400\", \"44550200\", \"6192200\", \"28051900\", \"32037200\""
[5] "\"Alabama\", \"2126500\", \"174200\", \"869700\", \"783000\", \"85600\", \"72480\\
0\", \"4763900\", \"2202800\", \"288900\", \"891900\", \"718400\", \"143900\", \"52220\\
0\", \"4768000\", \"2218000\", \"291500\", \"911400\", \"719100\", \"174600\", \"51940\\
0\", \"4833900\", \"2263800\", \"262400\", \"997000\", \"761200\", \"128800\", \"42080\\
0\", \"4834100\""
```

```
\ 
\ 

[6] "\"Alaska\"", \"364900\", \"24000\", \"95000\", \"55200\", \"60600\", \"102200\", \\
\"702000\", \"345300\", \"26800\", \"130100\", \"55300\", \"37300\", \"100800\", \"6957\\
00\", \"355700\", \"22300\", \"128100\", \"60900\", \"47700\", \"90500\", \"705300\", \\
\"324400\", \"20300\", \"145400\", \"68200\", \"55600\", \"96900\", \"710800\""
\ 
\ 
\ 
\ 

[7] "\"Arizona\"", \"2883800\", \"170800\", \"1346100\", \"842000\", \"N/A\", \"122300\\
0\", \"6603100\", \"2835200\", \"333500\", \"1639400\", \"911100\", \"N/A\", \"827100\\
\", \"6657200\", \"2766500\", \"278400\", \"1711500\", \"949000\", \"189300\", \"844800\\
\", \"6739500\", \"3010700\", \"377000\", \"1468400\", \"1028000\", \"172500\", \"8337\\
00\", \"6890200\""
\ 
\ 
\ 
```

Looks like we don't need the first two lines, so we'll read in the data, starting with the third line of the file:

```
coverage <- read_csv('https://raw.githubusercontent.com/opencasestudies/ocs-hea\\
1thexpenditure/master/data/KFF/healthcare-coverage.csv',
skip = 2)

coverage
Warning: One or more parsing issues, see `problems()` for details
# A tibble: 78 × 29
  Location `2013_Employer` `2013_Non-Grou... `2013_Medicaid` `2013_Medicare`\\
<chr>      <dbl>          <dbl>          <dbl>          <dbl>
1 United S... 155696900    13816000    54919100    40876300
2 Alabama     2126500       174200       869700       783000
3 Alaska      364900        24000        95000        55200
4 Arizona     2883800       170800       1346100      842000
5 Arkansas    1128800       155600       600800       515200
6 Californ... 17747300      1986400      8344800      3828500
7 Colorado    2852500       426300       697300       549700
```

```

8 Connecticut      2030500        126800        532000        475300
9 Delaware        473700         25100         192700        141300
10 District of Co 324300         30400         174900        59900
# ... with 68 more rows, and 24 more variables: 2013_Other Public <chr>,
#   2013_Uninsured <dbl>, 2013_Total <dbl>, 2014_Employer <dbl>,
#   2014_Non-Group <dbl>, 2014_Medicaid <dbl>, 2014_Medicare <dbl>,
#   2014_Other Public <chr>, 2014_Uninsured <dbl>, 2014_Total <dbl>,
#   2015_Employer <dbl>, 2015_Non-Group <dbl>, 2015_Medicaid <dbl>,
#   2015_Medicare <dbl>, 2015_Other Public <chr>, 2015_Uninsured <dbl>,
#   2015_Total <dbl>, 2016_Employer <dbl>, 2016_Non-Group <dbl>, ...

```

So, the first few lines of the dataset appear to store information for each state (observation) in the rows and different variables in the columns. What about the final few lines of the file?

```

tail(coverage, n = 30)
# A tibble: 30 × 29
  Location `2013_Employer` `2013_Non-Grou... `2013_Medicaid` `2013_Medicare` 
  <chr>       <dbl>          <dbl>          <dbl>          <dbl>
1 "Washing...  3541600        309000        1026800        879000
2 "West Vi...  841300         42600         382500        329400
3 "Wiscons...  3154500        225300        907600        812900
4 "Wyoming"   305900         19500         74200         65400 
5 "Notes"     NA             NA             NA             NA
6 "The maj...  NA             NA             NA             NA
7 <NA>         NA             NA             NA             NA
8 "In this...  NA             NA             NA             NA
9 <NA>         NA             NA             NA             NA
10 "Data ex...  NA            NA             NA             NA
# ... with 20 more rows, and 24 more variables: 2013_Other Public <chr>,
#   2013_Uninsured <dbl>, 2013_Total <dbl>, 2014_Employer <dbl>,
#   2014_Non-Group <dbl>, 2014_Medicaid <dbl>, 2014_Medicare <dbl>,
#   2014_Other Public <chr>, 2014_Uninsured <dbl>, 2014_Total <dbl>,
#   2015_Employer <dbl>, 2015_Non-Group <dbl>, 2015_Medicaid <dbl>,
#   2015_Medicare <dbl>, 2015_Other Public <chr>, 2015_Uninsured <dbl>,
#   2015_Total <dbl>, 2016_Employer <dbl>, 2016_Non-Group <dbl>, ...

```

Looks like there's a lot of missing information there at the end of the file due the "Notes" observation. Seems as though Notes were added to the file that are not the actual data. We'll

want to only include rows before the value of “Notes” for the `Location` variable at the end of the file. Using `n_max` and the `==` operator, we can specify that we want all the lines up to and including where the `Location` variable “is equal to” “Notes”. Using `-1` we can also remove the last line, which will be the line that contains “Notes”.

```
## read coverage data into R
coverage <- read_csv('https://raw.githubusercontent.com/opencasestudies/ocs-hea\
lthexpenditure/master/data/KFF/healthcare-coverage.csv',
                      skip = 2,
                      n_max  = which(coverage$Location == "Notes")-1)

tail(coverage)
# A tibble: 6 × 29
  Location     `2013_Employer` `2013_Non-Grou... `2013_Medicaid` `2013_Medic\
are` 
  <chr>          <dbl>        <dbl>        <dbl>        <\n
dbl>
1 Vermont       317700       26200       123400      9\
6600
2 Virginia      4661600      364800      773200      96\
8000
3 Washington    3541600      309000      1026800     87\
9000
4 West Virginia 841300       42600       382500      32\
9400
5 Wisconsin     3154500      225300      907600      81\
2900
6 Wyoming        305900       19500       74200       6\
5400
# ... with 24 more variables: 2013_Other Public <chr>, 2013_Uninsured <dbl>,
#   2013_Total <dbl>, 2014_Employer <dbl>, 2014_Non-Group <dbl>,
#   2014_Medicaid <dbl>, 2014_Medicare <dbl>, 2014_Other Public <chr>,
#   2014_Uninsured <dbl>, 2014_Total <dbl>, 2015_Employer <dbl>,
#   2015_Non-Group <dbl>, 2015_Medicaid <dbl>, 2015_Medicare <dbl>,
#   2015_Other Public <chr>, 2015_Uninsured <dbl>, 2015_Total <dbl>,
#   2016_Employer <dbl>, 2016_Non-Group <dbl>, 2016_Medicaid <dbl>, ...
```

Looks much better now! We can then use the `glimpse()` function of the `dplyr` package to get a sense of what types of information are stored in our dataset.

```

glimpse(coverage)
Rows: 52
Columns: 29
$ Location          <chr> "United States", "Alabama", "Alaska", "Arizona", ...
$ `2013_Employer`   <dbl> 155696900, 2126500, 364900, 2883800, 1128800, 177...
$ `2013_Non-Group`  <dbl> 13816000, 174200, 24000, 170800, 155600, 1986400, ...
$ `2013_Medicaid`   <dbl> 54919100, 869700, 95000, 1346100, 600800, 8344800...
$ `2013_Medicare`   <dbl> 40876300, 783000, 55200, 842000, 515200, 3828500, ...
$ `2013_Other Public` <chr> "6295400", "85600", "60600", "N/A", "67600", "675...
$ `2013_Uninsured`  <dbl> 41795100, 724800, 102200, 1223000, 436800, 559410...
$ `2013_Total`      <dbl> 313401200, 4763900, 702000, 6603100, 2904800, 381...
$ `2014_Employer`   <dbl> 154347500, 2202800, 345300, 2835200, 1176500, 177...
$ `2014_Non-Group`  <dbl> 19313000, 288900, 26800, 333500, 231700, 2778800, ...
$ `2014_Medicaid`   <dbl> 61650400, 891900, 130100, 1639400, 639200, 961880...
$ `2014_Medicare`   <dbl> 41896500, 718400, 55300, 911100, 479400, 4049000, ...
$ `2014_Other Public` <chr> "5985000", "143900", "37300", "N/A", "82000", "63...
$ `2014_Uninsured`  <dbl> 32967500, 522200, 100800, 827100, 287200, 3916700...
$ `2014_Total`      <dbl> 316159900, 4768000, 695700, 6657200, 2896000, 387...
$ `2015_Employer`   <dbl> 155965800, 2218000, 355700, 2766500, 1293700, 177...
$ `2015_Non-Group`  <dbl> 21816500, 291500, 22300, 278400, 200200, 3444200, ...
$ `2015_Medicaid`   <dbl> 62384500, 911400, 128100, 1711500, 641400, 101381...
$ `2015_Medicare`   <dbl> 43308400, 719100, 60900, 949000, 484500, 4080100, ...
$ `2015_Other Public` <chr> "6422300", "174600", "47700", "189300", "63700", ...
$ `2015_Uninsured`  <dbl> 28965900, 519400, 90500, 844800, 268400, 2980600, ...
$ `2015_Total`      <dbl> 318868500, 4833900, 705300, 6739500, 2953000, 391...
$ `2016_Employer`   <dbl> 157381500, 2263800, 324400, 3010700, 1290900, 181...
$ `2016_Non-Group`  <dbl> 21884400, 262400, 20300, 377000, 252900, 3195400, ...
$ `2016_Medicaid`   <dbl> 62303400, 997000, 145400, 1468400, 618600, 985380...
$ `2016_Medicare`   <dbl> 44550200, 761200, 68200, 1028000, 490000, 4436000...
$ `2016_Other Public` <chr> "6192200", "128800", "55600", "172500", "67500", ...
$ `2016_Uninsured`  <dbl> 28051900, 420800, 96900, 833700, 225500, 3030800, ...
$ `2016_Total`      <dbl> 320372000, 4834100, 710800, 6890200, 2945300, 391...

```

This gives an us output with all the variables listed on the far left. Thus essentially the data is rotated from the way it would be shown if we used `head()` instead of `glimpse()`. The first few observations for each variable are shown for each variable with a comma separating each observation.

Looks like we have a whole bunch of numeric variables (indicated by `<dbl>`), but a few that

appear like they *should* be numeric, but are actually strings (indicated by <chr>). We'll keep this in mind for when we wrangle the data!

health care Spending Data

Now, we're ready to read in our health care spending data, using a similar approach as we did for the coverage data.

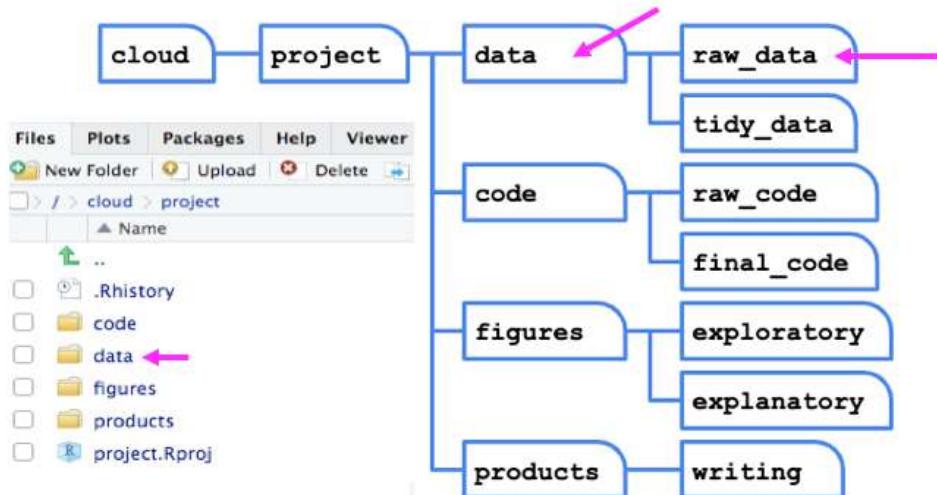
```
## read spending data into R
spending <- read_csv('https://raw.githubusercontent.com/opencasestudies/ocs-hea\
lthexpenditure/master/data/KFF/healthcare-spending.csv',
                      skip = 2)
#got some parsing errors...
spending <- read_csv('https://raw.githubusercontent.com/opencasestudies/ocs-hea\
lthexpenditure/master/data/KFF/healthcare-spending.csv',
                      skip = 2,
                      n_max = which(spending$Location == "Notes")-1)
Warning: One or more parsing issues, see `problems()` for details
tail(spending)
# A tibble: 6 × 25
  Location     `1991__Total He... `1992__Total He... `1993__Total He... `1994__Total\
He...
  <chr>          <dbl>          <dbl>          <dbl>          <dbl>
1 Vermont       1330           1421           1522           1625
2 Virginia      14829          15599          16634          17637
3 Washington    12674          13859          14523          15303
4 West Virginia 4672            5159           5550           5891
5 Wisconsin     12694          13669          14636          15532
6 Wyoming        1023           1067           1171           1265
# ... with 20 more variables: 1995__Total Health Spending <dbl>,
#   1996__Total Health Spending <dbl>, 1997__Total Health Spending <dbl>,
#   1998__Total Health Spending <dbl>, 1999__Total Health Spending <dbl>,
```

```
# 2000_Total Health Spending <dbl>, 2001_Total Health Spending <dbl>,
# 2002_Total Health Spending <dbl>, 2003_Total Health Spending <dbl>,
# 2004_Total Health Spending <dbl>, 2005_Total Health Spending <dbl>,
# 2006_Total Health Spending <dbl>, 2007_Total Health Spending <dbl>, ...
```

Recall from the introduction, that in data science workflows, we perform multiple steps in evaluating data. To keep this process tidy and reproducible, it is often helpful to save our data in a raw state and in processed states to allow for easy comparison. So let's save our case study 1 data to use in later sections of the course.

We can use the `here` package described in the introduction to help us make this process easier. Recall that the `here` package allows us to quickly reference the directory in which the `.Rproj` file is located.

Assuming we created a project called “project”, let's save our raw coverage data in a directory called `raw_data` within a directory called `data` inside of our RStudio project similarly to the workflows that we have seen in the introduction.



File Structure

After creating a directory called `raw_data` within a directory that we called `data`, we can now save our raw data for case study #1 using the `here` package by simply typing:

```

library(here)
here() starts at /Users/rdpeng/books/tidyversecourse
save(coverage, spending, file = here::here("data", "raw_data", "case_study_1.rda"))
#the coverage object and the spending object will get saved as case_study_1.rda\
within the raw_data directory which is a subdirectory of data
#the here package identifies where the project directory is located based on th\
e .Rproj, and thus the path to this directory is not needed

```

Case Study #2: Firearms

We've got a whole bunch of datasets that we'll need to read in for this case study. They are from a number of different sources and are stored in different file formats. This means we'll need to use various functions to read the data into R.

As a reminder, we're interested in the following question: At the state-level, what is the relationship between firearm legislation strength and annual rate of fatal police shootings?

Census Data

Population characteristics at the state level for 2017 are available [here](#). Let's read it into R.

```

# read in the census data
census <- read_csv('https://raw.githubusercontent.com/opencasestudies/ocs-polici\
e-shootings-firearm-legislation/master/data/sc-est2017-alldata6.csv',
                   n_max = 236900)

census
# A tibble: 236,844 × 19
  SUMLEV REGION DIVISION STATE NAME      SEX ORIGIN   RACE   AGE CENSUS2010POP
    <chr>    <dbl>    <dbl> <chr> <chr>    <dbl> <dbl> <dbl> <dbl>    <dbl>
1 040        3        6 01 Alabama     0       0     1     0     37991
2 040        3        6 01 Alabama     0       0     1     1     38150
3 040        3        6 01 Alabama     0       0     1     2     39738
4 040        3        6 01 Alabama     0       0     1     3     39827
5 040        3        6 01 Alabama     0       0     1     4     39353
6 040        3        6 01 Alabama     0       0     1     5     39520
7 040        3        6 01 Alabama     0       0     1     6     39813
8 040        3        6 01 Alabama     0       0     1     7     39695

```

```

 9 040      3      6 01    Alabama      0      0      1      8      40012
10 040      3      6 01    Alabama      0      0      1      9      42073
# ... with 236,834 more rows, and 9 more variables: ESTIMATESBASE2010 <dbl>,
#   POPESTIMATE2010 <dbl>, POPESTIMATE2011 <dbl>, POPESTIMATE2012 <dbl>,
#   POPESTIMATE2013 <dbl>, POPESTIMATE2014 <dbl>, POPESTIMATE2015 <dbl>,
#   POPESTIMATE2016 <dbl>, POPESTIMATE2017 <dbl>

```

Counted Data

The [Counted](#) project started to count persons killed by police in the US due to the fact that as stated by Jon Swaine “the US government has no comprehensive record of the number of people killed by law enforcement”. These data can be read in from the CSV stored on [GitHub](#), for 2015:

```

# read in the counted data
counted15 <- read_csv("https://raw.githubusercontent.com/opencasestudies/ocs-po\
lice-shootings-firearm-legislation/master/data/the-counted-2015.csv")

```

Suicide Data

Information about suicide and suicide as a result of firearms can also be directly read into R from the CSVs stored on GitHub:

```

# read in suicide data
suicide_all <- read_csv("https://raw.githubusercontent.com/opencasestudies/ocs-\
police-shootings-firearm-legislation/master/data/suicide_all.csv")
suicide_all
# A tibble: 51 × 12
  Sex       Race     State      Ethnicity `Age Group` `First Year` `Last Ye\
ar` 
  <chr>     <chr>     <chr>      <chr>      <chr>      <dbl>      <d\
b1> 
  1 Both Sexes All Races Alabama      Both       All Ages        2015      2\
016
  2 Both Sexes All Races Alaska      Both       All Ages        2015      2\
016
  3 Both Sexes All Races Arizona    Both       All Ages        2015      2\

```

```
016
  4 Both Sexes All Races Arkansas    Both      All Ages        2015      2 \
016
  5 Both Sexes All Races California Both      All Ages        2015      2 \
016
  6 Both Sexes All Races Colorado   Both      All Ages        2015      2 \
016
  7 Both Sexes All Races Connecticut Both      All Ages        2015      2 \
016
  8 Both Sexes All Races Delaware  Both      All Ages        2015      2 \
016
  9 Both Sexes All Races Florida   Both      All Ages        2015      2 \
016
10 Both Sexes All Races Georgia   Both      All Ages        2015      2 \
016
# ... with 41 more rows, and 5 more variables: Cause of Death <chr>,
# Deaths <dbl>, Population <dbl>, Crude Rate <dbl>, Age-Adjusted Rate <chr>

# read in firearm suicide data
suicide_firearm <- read_csv("https://raw.githubusercontent.com/opencasestudies/\
ocs-police-shootings-firearm-legislation/master/data/suicide_firearm.csv")
suicide_firearm
# A tibble: 51 × 12
  Sex       Race     State      Ethnicity `Age Group` `First Year` `Last Ye\
ar` 
  <chr>     <chr>    <chr>      <chr>      <chr>      <dbl>      <d\
bl>
  1 Both Sexes All Races Alabama   Both      All Ages        2015      2 \
016
  2 Both Sexes All Races Alaska   Both      All Ages        2015      2 \
016
  3 Both Sexes All Races Arizona  Both      All Ages        2015      2 \
016
  4 Both Sexes All Races Arkansas Both      All Ages        2015      2 \
016
  5 Both Sexes All Races California Both      All Ages        2015      2 \
016
  6 Both Sexes All Races Colorado  Both      All Ages        2015      2 \
016
```

```
7 Both Sexes All Races Connecticut Both      All Ages      2015      2 \
016
8 Both Sexes All Races Delaware    Both      All Ages      2015      2 \
016
9 Both Sexes All Races Florida     Both      All Ages      2015      2 \
016
10 Both Sexes All Races Georgia   Both      All Ages      2015      2 \
016
# ... with 41 more rows, and 5 more variables: Cause of Death <chr>,
#   Deaths <dbl>, Population <dbl>, Crude Rate <dbl>, Age-Adjusted Rate <chr>
```

Brady Data

For the Brady Scores data, quantifying numerical scores for firearm legislation in each state, we'll need the `httr` package, as these data are stored in an Excel spreadsheet. Note, we could download these files to our local computer, store them, and read this file in using `readxl`'s `read_excel()` file, or we can use the `httr` package to download and store the file in a temporary directory, followed by `read_excel` to read them into R. We'll go with this second option here to demonstrate how it works.

```
library(readxl)
library(httr)

# specify URL to file
url = "https://github.com/opencasestudies/ocs-police-shootings-firearm-legislat\
ion/blob/master/data/Brady-State-Scorecard-2015.xlsx?raw=true"

# Use httr's GET() and read_excel() to read in file
GET(url, write_disk(tf <- tempfile(fileext = ".xlsx")))
Response [https://raw.githubusercontent.com/opencasestudies/ocs-police-shootings\
-firearm-legislation/master/data/Brady-State-Scorecard-2015.xlsx]
Date: 2021-09-03 18:53
Status: 200
Content-Type: application/octet-stream
Size: 66.2 kB
<ON DISK> /var/folders/xn/fncwm3zs5t36q6chqx1nxktr0000gn/T//RtmpdYj2qv/file13d\
ed59c09d07.xlsx
brady <- read_excel(tf, sheet = 1)
```

```
brady
# A tibble: 116 × 54
`States can recei... `Category Point... `Sub Category P... Points AL     AK     AR
<chr>                 <dbl>           <dbl>   <dbl> <chr> <chr> <chr>
1 TOTAL STATE POINTS      NA            NA      NA -18    -30    -24
2 CATEGORY 1: KEEP...       50            NA      NA <NA> <NA> <NA>
3 BACKGROUND CHECKS...      NA            25      NA AL     AK     AR
4 Background Checks...      NA            NA      25 <NA> <NA> <NA>
5 Background Checks...      NA            NA      20 <NA> <NA> <NA>
6 Background Checks...      NA            NA      5  <NA> <NA> <NA>
7 Verify Legal Pur...       NA            NA      20 <NA> <NA> <NA>
8 TOTAL                   NA            NA      NA  0     0     0
9 <NA>                     NA            NA      NA <NA> <NA> <NA>
10 OTHER LAWS TO STO...      NA            12      NA AL     AK     AR
# ... with 106 more rows, and 47 more variables: AZ <chr>, CA <chr>, CO <chr>,
# CT <chr>, DE <chr>, FL <chr>, GA <chr>, HI <chr>, ID <chr>, IL <chr>,
# IN <chr>, IA <chr>, KS <chr>, KY <chr>, LA <chr>, MA <chr>, MD <chr>,
# ME <chr>, MI <chr>, MN <chr>, MO <chr>, MT <chr>, MS <chr>, NC <chr>,
# ND <chr>, NE <chr>, NH <chr>, NJ <chr>, NM <chr>, NV <chr>, NY <chr>,
# OK <chr>, OH <chr>, OR <chr>, PA <chr>, RI <chr>, SC <chr>, SD <chr>,
# TN <chr>, TX <chr>, UT <chr>, VA <chr>, VT <chr>, WA <chr>, WI <chr>, ...

```

Crime Data

Crime data, from the FBI's Uniform Crime Report, are stored as an Excel file, so we'll use a similar approach as above for these data:

```
# specify URL to file
url = "https://github.com/opencasestudies/ocs-police-shootings-firearm-legislat\ion/blob/master/data/table_5_crime_in_the_united_states_by_state_2015.xls?raw=t\ru"
# Use httr's GET() and read_excel() to read in file
GET(url, write_disk(tf <- tempfile(fileext = ".xls")))
Response [https://raw.githubusercontent.com/opencasestudies/ocs-police-shootings\firearm-legislation/master/data/table_5_crime_in_the_united_states_by_state_2\015.xls]
```

```

Date: 2021-09-03 18:53
Status: 200
Content-Type: application/octet-stream
Size: 98.3 kB
<ON DISK> /var/folders/xn/fncwm3zs5t36q6chqx1nxktr0000gn/T//RtmpdYj2qv/file13d\
ed618fb492.xls
crime <- read_excel(tf, sheet = 1, skip = 3)

# see data
crime
# A tibble: 510 × 14
  State   Area     ...3 Population `Violent\ncrime... `Murder and \nnonne...
  <chr>  <chr>    <chr>      <chr>           <dbl>          <dbl>
1 ALABAMA Metropoli... <NA>      3708033        NA             NA
2 <NA>    <NA>     Area ac... 0.97099999... 18122          283
3 <NA>    <NA>     Estimat... 1            18500          287
4 <NA>    Cities ou... <NA>      522241         NA             NA
5 <NA>    <NA>     Area ac... 0.97399999... 3178           32
6 <NA>    <NA>     Estimat... 1            3240           33
7 <NA>    Nonmetrop... <NA>      628705         NA             NA
8 <NA>    <NA>     Area ac... 0.99399999... 1205           28
9 <NA>    <NA>     Estimat... 1            1212           28
10 <NA>   State Tot... <NA>      4858979       22952          348
# ... with 500 more rows, and 8 more variables: Rape
(revised
definition)2 <dbl>,
#   Rape
(legacy
definition)3 <dbl>, Robbery <dbl>, Aggravated
assault <dbl>,
#   Property
crime <dbl>, Burglary <dbl>, Larceny-
theft <dbl>,
#   Motor
vehicle
theft <dbl>

```

Note, however, there are slight differences in the code used here, relative to the Brady data. We have to use `skip = 3` to skip the first three lines of this file. Also, this file has the extension

.xls rather than .xlsx, which we specify within the fileext argument.

Land Area Data

US Census 2010 land area data are also stored in an excel spreadsheet. So again we will use a similar method.

```
# specify URL to file
url = "https://github.com/opencasestudies/ocs-police-shootings-firearm-legislat\ion/blob/master/data/LND01.xls?raw=true"

# Use httr's GET() and read_excel() to read in file
GET(url, write_disk(tf <- tempfile(fileext = ".xls")))
Response [https://raw.githubusercontent.com/opencasestudies/ocs-police-shootings\firearm-legislation/master/data/LND01.xls]
Date: 2021-09-03 18:53
Status: 200
Content-Type: application/octet-stream
Size: 1.57 MB
<ON DISK> /var/folders/xn/fncwm3zs5t36q6chqx1nxktr0000gn/T//RtmpdYj2qv/file13d\ed6135133.xls
land <- read_excel(tf, sheet = 1)

# see data
land
# A tibble: 3,198 × 34
  Areaname    STCOU LND010190F LND010190D LND010190N1 LND010190N2 LND010200F
  <chr>      <chr>     <dbl>     <dbl>   <chr>      <chr>     <dbl>
1 UNITED STATES 00000        0  3787425. 0000    0000        0
2 ALABAMA       01000        0   52423. 0000    0000        0
3 Autauga, AL   01001        0    604. 0000    0000        0
4 Baldwin, AL   01003        0   2027. 0000    0000        0
5 Barbour, AL   01005        0    905. 0000    0000        0
6 Bibb, AL      01007        0    626. 0000    0000        0
7 Blount, AL    01009        0    651. 0000    0000        0
8 Bullock, AL   01011        0    626. 0000    0000        0
9 Butler, AL    01013        0    778. 0000    0000        0
10 Calhoun, AL  01015        0    612. 0000    0000        0
# ... with 3,188 more rows, and 27 more variables: LND010200D <dbl>,
```

```
#  LND010200N1 <chr>, LND010200N2 <chr>, LND110180F <dbl>, LND110180D <dbl>,
#  LND110180N1 <chr>, LND110180N2 <chr>, LND110190F <dbl>, LND110190D <dbl>,
#  LND110190N1 <chr>, LND110190N2 <chr>, LND110200F <dbl>, LND110200D <dbl>,
#  LND110200N1 <chr>, LND110200N2 <chr>, LND110210F <dbl>, LND110210D <dbl>,
#  LND110210N1 <chr>, LND110210N2 <chr>, LND210190F <dbl>, LND210190D <dbl>,
#  LND210190N1 <chr>, LND210190N2 <chr>, LND210200F <dbl>, LND210200D <dbl>, ...
```

Unemployment Data

This data is available online from the [Bureau of Labor Statistics \(BLS\)](#), but there is no easy download of the table. It is also difficult to simply copy and paste; it doesn't hold its table format. Thus we will want to use web scraping to most easily and accurately obtain this information using the `rvest` package.

As a reminder, to view the HTML of a webpage, right-click and select "View page source."

```
library(rvest)

# specify URL to where we'll be web scraping
url <- read_html("https://web.archive.org/web/20210205040250/https://www.bls.gov/lau/lastrk15.htm")

# scrape specific table desired
out <- html_nodes(url, "table") %>%
  .[2] %>%
  html_table(fill = TRUE)

# store as a tibble
unemployment <- as_tibble(out[[1]])

unemployment
# A tibble: 54 × 3
  State      `2015rate` Rank
  <chr>       <chr>     <chr>
  1 "United States" "5.3"    ""
  2 ""          ""        ""
  3 "North Dakota" "2.8"    "1"
  4 "Nebraska"   "3.0"    "2"
```

```
5 "South Dakota"    "3.1"      "3"
6 "New Hampshire"   "3.4"      "4"
7 "Hawaii"          "3.6"      "5"
8 "Utah"            "3.6"      "5"
9 "Vermont"         "3.6"      "5"
10 "Minnesota"       "3.7"     "8"
# ... with 44 more rows
```

Then we get the values from each column of the data table. The `html_nodes()` function acts as a CSS selector. The “table” class returns two tables from the webpage and we specify that we want the second table. From the object `out` we select the first in the list and store this as a tibble.

Now that we have gathered all the raw data we will need for our second case study, let’s save it using the `here` package:

```
library(here)
save(census, counted15, suicide_all, suicide_firearm, brady, crime, land, unemp\loyment , file = here::here("data", "raw_data", "case_study_2.rda"))
#all of these objects (census, counted15 etc) will get saved as case_study_2.rda
#within the raw_data directory which is a subdirectory of data
#the here package identifies where the project directory is located based on th\re .Rproj, and thus the path to this directory is not needed
```

3. Wrangling Data in the Tidyverse

In the last course we spent a ton of time talking about all the most common ways data are stored and reviewed how to get them into a tibble (or data.frame) in R.

So far we've discussed what tidy and untidy data are. We've (hopefully) convinced you that tidy data are the right type of data to work with. What we may not have made perfectly clear yet is that data are *not* always the tidiest when they come to you at the start of a project. An incredibly important skill of a data scientist is to be able to take data from an untidy format and get it into a tidy format. This process is often referred to as **data wrangling**. Generally, data wranglings skills are those that allow you to wrangle data from the format they're currently in into the tidy format you actually want them in.

Beyond data wrangling, it's also important to make sure the data you have are accurate and what you need to answer your question of interest. After wrangling the data into a tidy format, there is often further work that has to be done to **clean** the data.

About This Course

Data never arrive in the condition that you need them in order to do effective data analysis. Data need to be re-shaped, re-arranged, and re-formatted, so that they can be visualized or be inputted into a machine learning algorithm. This course addresses the problem of wrangling your data so that you can bring them under control and analyze them effectively. The key goal in data wrangling is transforming non-tidy data into tidy data.

This course covers many of the critical details about handling tidy and non-tidy data in R such as converting from wide to long formats, manipulating tables with the `dplyr` package, understanding different R data types, processing text data with regular expressions, and conducting basic exploratory data analyses. Investing the time to learn these data wrangling techniques will make your analyses more efficient, more reproducible, and more understandable to your data science team.

In this book we assume familiarity with the R programming language. If you are not yet familiar with R, we suggest you first complete [R Programming](#) before returning to complete this course.

Tidy Data Review

Before we move any further, let's review the requirements for a tidy dataset:

1. Each variable is stored in a column
2. Each observation is stored in a row
3. Each cell stores a single value

We had four tidy data principles in an earlier lesson, where the fourth was that each table should store a single *type* of information. That's less critical here, as we'll be working at first with single datasets, so let's just keep those three tidy data principles at the front of our minds.

Reshaping Data

Tidy data generally exist in two forms: wide data and long data. Both types of data are used and needed in data analysis, and fortunately, there are tools that can take you from wide-to-long format and from long-to-wide format. This makes it easy to work with any tidy dataset. We'll discuss the basics of what wide and long data are and how to go back and forth between the two in R. Getting data into the right format will be crucial later when summarizing data and visualizing it.

Wide Data

Wide data has a column for each variable and a row for each observation. Data are often entered and stored in this manner. This is because wide data are often easy to understand at a glance. For example, this is a wide dataset:

wide data

	A	B	C	D	E	F	G
1	ID	LastName	FirstName	Height_inches	Weight_lbs	Insulin	Glucose
2	1004	Smith	Jane	65	180	0.60	163
3	4587	Nayef	Mohammed	75	215	1.46	150
4	1727	Doe	Janice	62	124	0.72	177
5	6879	Jordan	Alex	77	160	1.23	205



Wide dataset

Up until this point, we would have described this dataset as a rectangular, tidy dataset. With the additional information just introduced, we can also state that it is a *wide* dataset. Here, you can clearly see what measurements were taken for each individual and can get a sense of how many individuals are contained in the dataset.

Specifically, each individual is in a different row with each variable in a different column. At a glance we can quickly see that we have information about four different people and that each person was measured in four different ways.

Long Data

Long data, on the other hand, has one column indicating the type of variable contained in that row and then a separate row for the value for that variable. Each row contains a single observation for a single variable. It's *still* a tidy datasets, but the information is stored in a long format:

	A	B	C
1	ID	Variable	Value
2	1004	LastName	Smith
3	4587	LastName	Nayef
4	1727	LastName	Doe
5	6879	LastName	Jordan
6	1004	FirstName	Jane
7	4587	FirstName	Mohammed
8	1727	FirstName	Janice
9	6879	FirstName	Alex
10	1004	Height_inches	65
11	4587	Height_inches	75
12	1727	Height_inches	62
13	6879	Height_inches	77
14	1004	Weight_lbs	180
15	4587	Weight_lbs	215
16	1727	Weight_lbs	124
17	6879	Weight_lbs	160
18	1004	insulin	0.80
19	4587	insulin	1.46
20	1727	insulin	0.72
21	6879	insulin	1.23
22	1004	Glucose	163
23	4587	Glucose	150
24	1727	Glucose	177
25	6879	Glucose	205

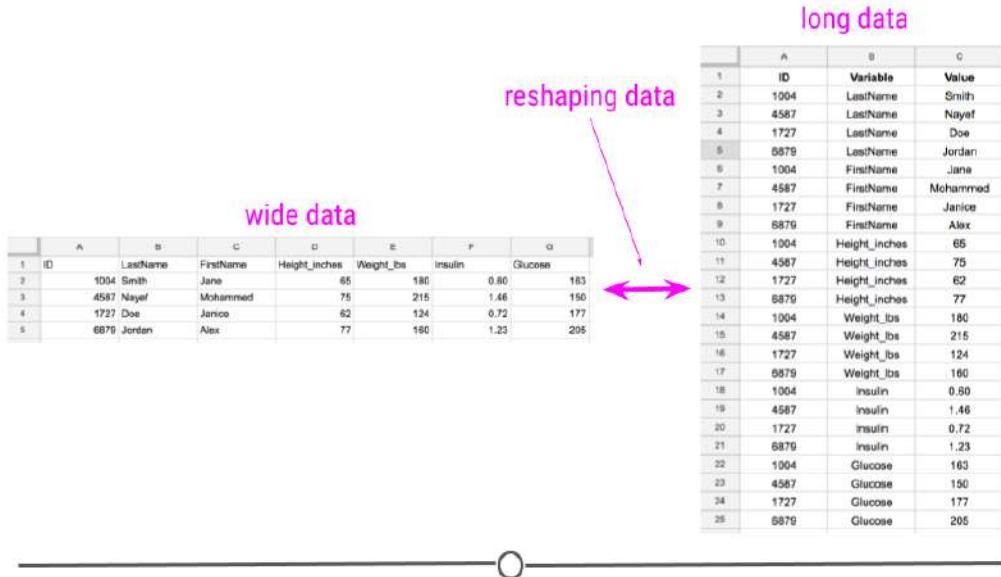
Long dataset

This long dataset includes the exact same information as the previous wide dataset; it is just stored differently. It's harder to see visually how many different measurements were taken and on how many different people, but the same information is there.

While long data formats are less readable than wide data at a glance, they are often a lot easier to work with during analysis. Most of the tools we'll be working with use long data. Thus, to go from how data are often stored (wide) to working with the data during analysis (long), we'll need to understand what tools are needed to do this and how to work with them.

Reshaping the Data

Converting your data from wide-to-long or from long-to-wide data formats is referred to as **reshaping** your data.



Reshaping data

Within the tidyverse, `tidyverse` is the go-to package for accomplishing this task. Within the `tidyverse` package, you'll have to become familiar with a number of functions. The two most pertinent to reshaping data are: `pivot_wider()` and `pivot_longer()`.

For these examples, we'll work with the `airquality` dataset available in R. The data in this dataset includes “Daily air quality measurements in New York, May to September 1973.” This is a wide dataset because each day is in a separate row and there are multiple columns with each including information about a different variable (ozone, solar.r, wind, temp, month, and day).

We'll load in the `tidyverse`, so that we can convert this `data.frame` to a `tibble` and see the first few lines of this dataset using the following code:

```
library(tidyverse)

airquality <- as_tibble(airquality)
airquality
# A tibble: 153 × 6
  Ozone Solar.R Wind Temp Month Day
  <int>    <int> <dbl> <int> <int> <int>
1     41      190   7.4   67     5     1
2     36      118    8     72     5     2
3     12      149  12.6   74     5     3
4     18      313  11.5   62     5     4
5     NA      NA  14.3   56     5     5
6     28      NA  14.9   66     5     6
7     23      299   8.6   65     5     7
8     19      99   13.8   59     5     8
9      8      19  20.1   61     5     9
10    NA      194   8.6   69     5    10
# ... with 143 more rows
```

Again, wide data are easy to decipher at a glance. We can see that we have six different variables for each day, with each one of these variables (measurements) being stored in a separate column.

tidyr

The `tidyr` package is part of the tidyverse, so its functionality is available to you since you've loaded in the tidyverse. The two main functions we mentioned above will help you reshape your data in the following ways:

- `pivot_longer()`: go from wide data to long data
- `pivot_wider()`: go from long data to wide data

To get started, you'll need to be sure that the `tidyr` package is installed and loaded into your RStudio session.

`pivot_longer()`

As data are often stored in wide formats, you'll likely use `pivot_longer()` a lot more frequently than you'll use `pivot_wider()`. This will allow you to get the data into a long format that will be easy to use for analysis.

In `tidyR`, `pivot_longer()` will take the `airquality` dataset from wide to long, putting each column name into the first column and each corresponding value into the second column. Here, the first column will be called `name`. The second column will still be `value`.

```
## use pivot_longer() to reshape from wide to long
gathered <- airquality %>%
  pivot_longer(everything())

## take a look at first few rows of long data
gathered
# A tibble: 918 × 2
  name    value
  <chr>   <dbl>
1 Ozone     41
2 Solar.R 190
3 Wind      7.4
4 Temp      67
5 Month     5
6 Day       1
7 Ozone     36
8 Solar.R 118
9 Wind      8
10 Temp     72
# ... with 908 more rows
```

```
> head(airquality)
  ozone solar.r wind temp month day
1   41     190  7.4   67     5   1
2   36     118  8.0   72     5   2
3   12     149 12.6   74     5   3
4   18     313 11.5   62     5   4
5   NA      NA 14.3   56     5   5
6   28     NA 14.9   66     5   6
```

airquality %>%
pivot_longer(
 everything()
)



```
> head(gathered)
  key value
1 ozone 41
2 ozone 36
3 ozone 12
4 ozone 18
5 ozone NA
6 ozone 28
```

Longer dataset

However, it's very easy to change the names of these columns within `pivot_longer()`. To do so you specify what the `names_to` and `values_to` columns names should be within `pivot_longer()`:

```
## to rename the column names that gather provides,
## change key and value to what you want those column names to be
gathered <- airquality %>%
  pivot_longer(everything(), names_to = "variable", values_to = "value")

## take a look at first few rows of long data
gathered
# A tibble: 918 × 2
  variable value
  <chr>    <dbl>
1 Ozone      41
2 Solar.R   190
3 Wind       7.4
4 Temp       67
5 Month      5
```

```

6 Day      1
7 Ozone    36
8 Solar.R 118
9 Wind     8
10 Temp    72
# ... with 908 more rows

```

```

> head(airquality)
#> #> ozone solar.r wind temp month day
#> #> 1 41    190  7.4  67   5   1
#> #> 2 36    118  8.0  72   5   2
#> #> 3 12    149 12.6  74   5   3
#> #> 4 18    313 11.5  62   5   4
#> #> 5 NA    NA 14.3  56   5   5
#> #> 6 28    NA 14.9  66   5   6

```

airquality %>%
 pivot_longer(
 names_to="variable",
 values_to="value"
)

→

	variable	value
1	ozone	41
2	ozone	36
3	ozone	12
4	ozone	18
5	ozone	NA
6	ozone	28

gather column names changed

However, you're likely not interested in your day and month variable being separated out into their own variables within the variable column. In fact, knowing the day and month associated with a particular data point helps identify that particular data point. To account for this, you can exclude day and month from the variables being included in the variable column by specifying all the variables that you *do* want included in the variable column. Here, that means specifying Ozone, Solar.R, Wind, and Temp. This will keep Day and Month in their own columns, allowing each row to be identified by the specific day and month being discussed.

```
## in pivot_longer(), you can specify which variables
## you want included in the long format
## it will leave the other variables as is
gathered <- airquality %>%
  pivot_longer(c(Ozone, Solar.R, Wind, Temp),
               names_to = "variable",
               values_to = "value")

## take a look at first few rows of long data
gathered
# A tibble: 612 × 4
  Month   Day variable value
  <int> <int> <chr>    <dbl>
1     5     1 Ozone      41
2     5     1 Solar.R   190
3     5     1 Wind       7.4
4     5     1 Temp       67
5     5     2 Ozone      36
6     5     2 Solar.R   118
7     5     2 Wind       8
8     5     2 Temp       72
9     5     3 Ozone      12
10    5     3 Solar.R   149
# ... with 602 more rows
```

```
> head(airquality)
#> #> ozone solar.r wind temp month day
#> #> 1 41 190 7.4 67 5 1
#> #> 2 36 118 8.0 72 5 2
#> #> 3 12 149 12.6 74 5 3
#> #> 4 18 313 11.5 62 5 4
#> #> 5 NA NA 14.3 56 5 5
#> #> 6 28 NA 14.9 66 5 6
```

airquality %>%
pivot_longer(
 names_to = "variable",
 values_to = "value",
 ozone, solar.r, wind, temp)

→

	month	day	variable	value
1	5	1	ozone	41
2	5	2	ozone	36
3	5	3	ozone	12
4	5	4	ozone	18
5	5	5	ozone	NA
6	5	6	ozone	28

ozone, solar.r, wind, temp:
The variables to move into the variable column.
The other variables keep their original column.

gather specifying which variables to include in long format

Now, when you look at the top of this object, you'll see that Month and Day remain in the data frame and that variable combines information from the other columns in airquality (Ozone, Solar.R, Wind, Temp). This is still a long format dataset; however, it has used Month and Day as IDs when reshaping the data frame.

`pivot_wider()`

To return your long data back to its original form, you can use `pivot_wider()`. Here you specify two columns: the column that contains the names of what your wide data columns should be (`names_from`) and the column that contains the values that should go in these columns (`values_from`). The data frame resulting from `pivot_wider()` will have the original information back in the wide format (again, the columns will be in a different order). But, we'll discuss how to rearrange data in the next lesson!

```
## use pivot_wider() to reshape from long to wide
spread_data <- gathered %>%
  pivot_wider(names_from = "variable",
             values_from = "value")

## take a look at the wide data
spread_data
# A tibble: 153 × 6
  Month   Day Ozone Solar.R   Wind   Temp
  <int> <int> <dbl>    <dbl> <dbl>   <dbl>
1     5     1    41      190    7.4    67
2     5     2    36      118     8    72
3     5     3    12      149   12.6    74
4     5     4    18      313   11.5    62
5     5     5    NA      NA    14.3    56
6     5     6    28      NA    14.9    66
7     5     7    23      299    8.6    65
8     5     8    19      99    13.8    59
9     5     9     8      19    20.1    61
10    5    10    NA      194    8.6    69
# ... with 143 more rows

## compare that back to the original
airquality
# A tibble: 153 × 6
  Ozone Solar.R   Wind   Temp Month   Day
  <int>    <int> <dbl>    <int> <int> <int>
1     41      190    7.4     67     5     1
2     36      118     8     72     5     2
3     12      149   12.6     74     5     3
4     18      313   11.5     62     5     4
5     NA      NA    14.3     56     5     5
6     28      NA    14.9     66     5     6
7     23      299    8.6     65     5     7
8     19      99    13.8     59     5     8
9      8      19    20.1     61     5     9
10    NA      194    8.6     69     5    10
# ... with 143 more rows
```

```
> head(spread_data)
   month day ozone solar.r temp wind
1      5   1     41     190    67  7.4
2      5   2     36     118    72  8.0
3      5   3     12     149    74 12.6
4      5   4     18     313    62 11.5
5      5   5      NA      NA    56 14.3
6      5   6     28      NA    66 14.9

> head(airquality)
  ozone solar.r wind temp month day
1     41     190  7.4   67     5   1
2     36     118  8.0   72     5   2
3     12     149 12.6   74     5   3
4     18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
```



spread data

While reshaping data may not *read* like the most exciting topic, having this skill will be indispensable as you start working with data. It's best to get these skills down pat early!

Data Wrangling

Once you've read your data into R and have it in the appropriately wide- or long-format, it's time to wrangle the data, so that it is in the appropriate format and includes the information you need.

R Packages

While there are *tons* of R packages out there to help you work with data, we're going to cover the packages and functions within those packages that you'll absolutely want and need to work with when working with data.

dplyr

There is a package specifically designed for helping you wrangle your data. This package is called `dplyr` and will allow you to easily accomplish many of the data wrangling tasks

necessary. Like `tidyverse`, this package is a core package within the `tidyverse`, and thus it was loaded in for you when you ran `library(tidyverse)` earlier. We will cover a number of functions that will help you wrangle data using `dplyr`:

- `%>%` - pipe operator for chaining a sequence of operations
- `glimpse()` - get an overview of what's included in dataset
- `filter()` - filter rows
- `select()` - select, rename, and reorder columns
- `rename()` - rename columns
- `arrange()` - reorder rows
- `mutate()` - create a new column
- `group_by()` - group variables
- `summarize()` - summarize information within a dataset
- `left_join()` - combine data across data frame
- `tally()` - get overall sum of values of specified column(s) or the number of rows of tibble
- `count()` - get counts of unique values of specified column(s) (shortcut of `group_by()` and `tally()`)
- `add_count()` - add values of `count()` as a new column
- `add_tally()` - add value(s) of `tally()` as a new column

tidyr

We will also return to the `tidyr` package. The same package that we used to reshape our data will be helpful when wrangling data. The main functions we'll cover from `tidyr` are:

- `unite()` - combine contents of two or more columns into a single column
- `separate()` - separate contents of a column into two or more columns

janitor

The third package we'll include here is the `janitor` package. While not a core `tidyverse` package, this `tidyverse`-adjacent package provides tools for cleaning messy data. The main functions we'll cover from `janitor` are:

- `clean_names()` - clean names of a data frame
- `tabyl()` - get a helpful summary of a variable
- `get_dupes()` - identify duplicate observations

If you have not already, you'll want to be sure this package is installed and loaded:

```
#install.packages('janitor')
library(janitor)
```

skimr

The final package we'll discuss here is the `skimr` package. This package provides a quick way to summarize a `data.frame` or `tibble` within the tidy data framework. We'll discuss its most useful function here:

- `skim()` - summarize a data frame

If you have not already, you'll want to be sure this package is installed and loaded:

```
#install.packages('skimr')
library(skimr)
```

The Pipe Operator

Before we get into the important functions within `dplyr`, it will be very useful to discuss what is known as the **pipe operator**. The pipe operator looks like this in R: `%>%`. Whenever you see the pipe `%>%`, think of the word “then”, so if you saw the sentence “I went to the store and `%>%` I went back to my house,” you would read this as I went to the store and *then* I went back to my house. The pipe tells you to do one thing and *then* do another.

Generally, the pipe operator allows you to string a number of different functions together in a particular order. If you wanted to take data frame A and carry out function B on it in R, you could depict this with an arrow pointing from A to B:

A → B

Here you are saying, “Take A and *then* feed it into function B.”

In base R syntax, what is depicted by the arrow above would be carried out by calling the function B on the data frame object A:

`B(A)`

Alternatively, you could use the pipe operator (`%>%`):

```
A %>% B
```

However, often you are not performing just one action on a data frame, but rather you are looking to carry out multiple functions. We can again depict this with an arrow diagram.

A → B → C → D

Here you are saying that you want to take data frame A and carry out function B, *then* you want to take the output from that and *then* carry out function C. Subsequently you want to take the output of that and *then* carry out function D. In R syntax, we would first apply function B to data frame A, then apply function C to this output, then apply function D to this output. This results in the following syntax that is hard to read because multiple calls to functions are nested within each other:

```
D(C(B(A)))
```

Alternatively, you could use the pipe operator. Each time you want take the output of one function and carry out something new on that output, you will use the pipe operator:

```
A %>% B %>% C %>% D
```

And, even more readable is when each of these steps is separated out onto its own individual line of code:

```
A %>%  
  B %>%  
  C %>%  
  D
```

While both of the previous two code examples would provide the same output, the one below is more readable, which is a large part of why pipes are used. It makes your code more understandable to you and others.

Below we'll use this pipe operator a lot. Remember, it takes output from the left hand side and feeds it into the function that comes after the pipe. You'll get a better understanding of how it works as you run the code below. But, when in doubt remember that the pipe operator should be read as *then*.

Filtering Data

When working with a large dataset, you’re often interested in only working with a portion of the data at any one time. For example, if you had data on people from ages 0 to 100 years old, but you wanted to ask a question that only pertained to children, you would likely want to only work with data from those individuals who were less than 18 years old. To do this, you would want to **filter** your dataset to only include data from these select individuals. Filtering can be done by row or by column. We’ll discuss the syntax in R for doing both. Please note that the examples in this lesson and the organization for this lesson were adapted from [Suzan Baert’s](#) wonderful `dplyr` tutorials. Links to the all four tutorials can be found in the “Additional Resources” section at the bottom of this lesson.

For the examples below, we’ll be using a dataset from the `ggplot2` package called `msleep`. (You’ll learn more about this package in a later course on data visualization. For now, it’s a core tidyverse package so it’s loaded in along with the other tidyverse packages using `library(tidyverse)`.) This dataset includes sleep times and weights from a number of different mammals. It has 83 rows, with each row including information about a different type of animal, and 11 variables. As each row is a different animal and each column includes information about that animal, this is a **wide** dataset.

To get an idea of what variables are included in this data frame, you can use `glimpse()`. This function summarizes how many rows there are (`Observations`) and how many columns there are (`Variables`). Additionally, it gives you a glimpse into the type of data contained in each column. Specifically, in this dataset, we know that the first column is `name` and that it contains a character vector (`chr`) and that the first three entries are “Cheetah”, “Owl monkey”, and “Mountain beaver.” It works similarly to the base R `summary()` function.

```
## take a look at the data
library(ggplot2)
glimpse(msleep)
```

```
> glimpse(msleep)
Observations: 83
Variables: 11
$ name      <chr> "Cheetah", "Owl monkey", "Mountain beaver", ...
$ genus     <chr> "Actomyx", "Aotus", "Aplopontia", "Batarina", ...
$ vore      <chr> "carni", "omni", "herbi", "herbi", "carni", NA, ...
$ order     <chr> "Carnivora", "Primates", "Rodentia", "Soricomorpha", ...
$ conservation <chr> "lc", NA, "nt", "lc", "domesticated", NA, "vu", NA, ...
$ sleep_total <dbl> 12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 10.1, ...
$ sleep_rem   <dbl> NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA, 0.6, ...
$ sleep_cycle <dbl> 0.8, 0.7, 1.5, 2.2, 2.0, 1.4, 3.1, ...
$ awake      <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, ...
$ brainwt    <dbl> 0.01550, NA, 0.00029, 0.42300, NA, NA, 0.07000, 0.09820, ...
$ bodywt     <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.490, 0.045, ...

There are:
• 83 rows
• 11 columns
```

The first 5 columns are character variables

The first three names of the animals in the dataset

The names of the columns

Glimpse of msleep dataset

Filtering Rows

If you were only interested in learning more about the sleep times of “Primates,” we could filter this dataset to include only data about those mammals that are also Primates. As we can see from `glimpse()`, this information is contained within the `order` variable. So to do this within R, we use the following syntax:

```
# filter to only include primates
msleep %>%
  filter(order == "Primates")
# A tibble: 12 × 11
  name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr> <chr> <chr> <chr> <chr>       <dbl>      <dbl>      <dbl>      <dbl>
1 Owl   m... Aotus omni  Prim... <NA>        17         1.8      NA       7
2 Grivet Cerc... omni  Prim... lc          10         0.7      NA      14
3 Patas... Eryt... omni  Prim... lc          10.9       1.1      NA     13.1
4 Galago Gala... omni  Prim... <NA>        9.8        1.1      0.55    14.2
5 Human  Homo  omni  Prim... <NA>         8         1.9      1.5      16
```

```

6 Mongo... Lemur herbi Prim... vu           9.5    0.9    NA    14.5
7 Macaq... Maca... omni  Prim... <NA>      10.1   1.2    0.75   13.9
8 Slow ... Nyct... carni Prim... <NA>      11     NA    NA    13
9 Chimp... Pan   omni  Prim... <NA>      9.7    1.4    1.42   14.3
10 Baboon Papio omni  Prim... <NA>      9.4    1     0.667  14.6
11 Potto  Pero... omni  Prim... lc        11     NA    NA    13
12 Squir... Saim... omni  Prim... <NA>      9.6    1.4    NA    14.4
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>

```

Note that we are using the equality == comparison operator that you learned about in the previous course. Also note that we have used the pipe operator to feed the `msleep` data frame into the `filter()` function.

The above is shorthand for:

```

filter(msleep, order == "Primates")
# A tibble: 12 × 11
  name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr> <chr> <chr> <chr> <chr>       <dbl>      <dbl>      <dbl> <dbl>
1 Owl   m... Aotus omni  Prim... <NA>      17       1.8    NA    7
2 Grivet Cerc... omni  Prim... lc        10       0.7    NA    14
3 Patas... Eryt... omni  Prim... lc        10.9     1.1    NA   13.1
4 Galago Gala... omni  Prim... <NA>      9.8    1.1    0.55   14.2
5 Human  Homo  omni  Prim... <NA>      8       1.9    1.5    16
6 Mongo... Lemur herbi Prim... vu        9.5    0.9    NA   14.5
7 Macaq... Maca... omni  Prim... <NA>      10.1   1.2    0.75   13.9
8 Slow ... Nyct... carni Prim... <NA>      11     NA    NA   13
9 Chimp... Pan   omni  Prim... <NA>      9.7    1.4    1.42   14.3
10 Baboon Papio omni  Prim... <NA>      9.4    1     0.667  14.6
11 Potto  Pero... omni  Prim... lc        11     NA    NA   13
12 Squir... Saim... omni  Prim... <NA>      9.6    1.4    NA   14.4
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>

```

The output is the same as above here, but the code is slightly less readable. This is why we use the pipe (%>%)!

Equivalent to:

				order	conservation	sleep_total	sleep_rem	sleep_cycle	awake	brainwt	bodywt
					<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Owl monkey	Aotus	omni	Prima_<NA>	17.0	1.80	NA	7.00	0.0155	0.480	
2	Grevet	Cercopithe_	omni	Prima_ lc	10.0	0.700	NA	14.0	NA	4.75	
3	Patas monkey	Erythroceb_	omni	Prima_ lc	10.9	1.10	NA	13.1	0.115	10.0	
4	Galago	Galago	omni	Prima_ <NA>	9.80	1.10	0.550	14.2	0.00500	0.200	
5	Human	Homo	omni	Prima_ <NA>	8.00	1.90	1.50	16.0	1.32	62.0	
6	Mongoose lemur	Lemur	herbi	Prima_ vu	9.50	0.900	NA	14.5	NA	1.67	
7	Macaque	Macaca	omni	Prima_ <NA>	10.1	1.20	0.750	13.9	0.179	6.80	
8	Slow loris	Nycticebus	carni	Prima_ <NA>	11.0	NA	NA	13.0	0.0125	1.40	
9	Chimpanzee	Pan	omni	Prima_ <NA>	9.70	1.40	1.42	14.3	0.440	52.2	
10	Baboon	Papio	omni	Prima_ <NA>	9.40	1.00	0.667	14.6	0.180	25.2	
11	Potto	Perodictic_	omni	Prima_ lc	11.0	NA	NA	13.0	NA	1.10	
12	Squirrel monkey	Saimiri	omni	Prima_ <NA>	9.60	1.40	NA	14.4	0.0200	0.743	

Filtered to only include Primates

Now, we have a smaller dataset of only 12 mammals (as opposed to the original 83) and we can see that the `order` variable column only includes “Primates.”

But, what if we were only interested in Primates who sleep more than 10 hours total per night? This information is in the `sleep_total` column. Fortunately, `filter()` also works on numeric variables. To accomplish this, you would use the following syntax, separating the multiple filters you want to apply with a comma:

```
msleep %>%
  filter(order == "Primates", sleep_total > 10)
# A tibble: 5 × 11
  name   genus  vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>  <chr> <chr> <chr>    <chr>      <dbl>     <dbl>     <dbl> <dbl>
1 Owl m... Aotus  omni  Prima_ <NA>       17        1.8      NA     7
2 Patas... Eryth... omni  Prima_ lc        10.9      1.1      NA    13.1
3 Macaq... Macaca omni  Prima_ <NA>       10.1      1.2      0.75  13.9
4 Slow ... Nycti... carni Prima_ <NA>       11        NA      NA     13
5 Potto   Perod... omni  Prima_ lc        11        NA      NA     13
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Note that we have used the “greater than” comparison operator with `sleep_total`.

Now, we have a dataset focused in on only 5 mammals, all of which are primates who sleep for more than 10 hours a night total.

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10)
# A tibble: 5 x 11
  name    genus     vore  order  conservation sleep_total sleep_rem sleep_cycle awake brainwt bodywt
  <chr>   <chr>    <chr> <chr>   <chr>        <dbl>      <dbl>      <dbl>    <dbl>    <dbl>
1 Owl monkey Aotus     omni  Primates <NA>         17.0       1.80      NA       7.00    0.0155  0.480
2 Patas monkey Erythrocebus omni  Primates lc          10.9       1.10      NA       13.1    0.0115   10.0
3 Macaque     Macaca    omni  Primates <NA>         10.1       1.20      0.750   13.9    0.0179   6.80
4 Slow loris Nycticebus carni Primates <NA>         11.0       NA        NA       13.0    0.0125   1.40
5 Potto       Perodicticus omni  Primates lc          11.0       NA        NA       13.0    NA        1.10
```

Gives the same results: `msleep %>%
filter(order == "Primates" & sleep_total > 10)`

Numerically filtered dataset

We can obtain the same result with the AND & logical operator instead of separating filtering conditions with a comma:

```
msleep %>%
  filter(order == "Primates" & sleep_total > 10)
# A tibble: 5 x 11
  name    genus     vore  order  conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr>    <chr> <chr>   <chr>        <dbl>      <dbl>      <dbl>    <dbl>
1 Owl m... Aotus     omni  Prim... <NA>         17         1.8      NA       7
2 Patas... Eryth... omni  Prim... lc          10.9       1.1      NA      13.1
3 Macaq... Macaca    omni  Prim... <NA>         10.1       1.2      0.75   13.9
4 Slow ... Nycti... carni Prim... <NA>         11         NA      NA       13
5 Potto   Perod... omni  Prim... lc          11         NA      NA       13
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Note that the number of columns hasn't changed. All 11 variables are still shown in columns

because the function `filter()` filters on rows, not columns.

Selecting Columns

While `filter()` operates on rows, it *is* possible to filter your dataset to only include the columns you're interested in. To select columns so that your dataset only includes variables you're interested in, you will use `select()`.

Let's start with the code we just wrote to only include primates who sleep a lot. What if we only want to include the first column (the name of the mammal) and the sleep information (included in the columns `sleep_total`, `sleep_rem`, and `sleep_cycle`)? We would do this by starting with the code we just used, adding another pipe, and using the function `select()`. Within `select`, we specify which columns we want in our output.

```
msleep %>%  
  filter(order == "Primates", sleep_total > 10) %>%  
  select(name, sleep_total, sleep_rem, sleep_cycle)  
# A tibble: 5 × 4  
  name      sleep_total sleep_rem sleep_cycle  
  <chr>        <dbl>     <dbl>       <dbl>  
1 Owl monkey      17       1.8        NA  
2 Patas monkey    10.9      1.1        NA  
3 Macaque         10.1      1.2       0.75  
4 Slow loris      11        NA        NA  
5 Potto           11        NA        NA
```

```
> msleep %>%  
+   filter(order == "Primates", sleep_total > 10) %>%  
+     select(name, sleep_total, sleep_rem, sleep_cycle)  
# A tibble: 5 x 4  
  name       sleep_total sleep_rem sleep_cycle  
  <chr>        <dbl>      <dbl>       <dbl>  
1 Owl monkey     17.0      1.80        NA  
2 Patas monkey    10.9      1.10        NA  
3 Macaque         10.1      1.20       0.750  
4 Slow loris      11.0      NA          NA  
5 Potto           11.0      NA          NA
```

Without the pipe operator: `select(filter(msleep, order == "Primates", sleep_total > 10), name, sleep_total, sleep_rem, sleep_cycle)`

Data with selected columns

Now, using `select()` we see that we still have the five rows we filtered to before, but we only have the four columns specified using `select()`. Here you can hopefully see the power of the pipe operator to chain together several commands in a row. Without the pipe operator, the full command would look like this:

```
select(filter(msleep, order == "Primates", sleep_total > 10), name, sleep_total\  
, sleep_rem, sleep_cycle)
```

Yuck. Definitely harder to read. We'll stick with the above approach!

Renaming Columns

`select()` can also be used to rename columns. To do so, you use the syntax: `new_column_name = old_column_name` within `select`. For example, to select the same columns and rename them `total`, `rem` and `cycle`, you would use the following syntax:

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, total = sleep_total, rem = sleep_rem, cycle = sleep_cycle)
# A tibble: 5 × 4
  name      total    rem cycle
  <chr>     <dbl>   <dbl> <dbl>
1 Owl monkey    17     1.8 NA
2 Patas monkey  10.9   1.1 NA
3 Macaque       10.1   1.2  0.75
4 Slow loris    11     NA    NA
5 Potto         11     NA    NA
```

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   select(name, total=sleep_total, rem=sleep_rem, cycle=sleep_cycle)
# A tibble: 5 × 4
  name      total    rem cycle
  <chr>     <dbl>   <dbl> <dbl>
1 Owl monkey    17.0   1.80 NA
2 Patas monkey  10.9   1.10 NA
3 Macaque       10.1   1.20  0.750
4 Slow loris    11.0   NA    NA
5 Potto         11.0   NA    NA
```

Data with renamed columns names with `select()`

It's important to keep in mind that when using `select()` to rename columns, only the specified columns will be included and renamed in the output. If you, instead, want to change the names of a few columns but return *all* columns in your output, you'll want to use `rename()`. For example, the following, returns a data frame with all 11 columns, where the column names for three columns specified within `rename()` function have been renamed.

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  rename(total = sleep_total, rem = sleep_rem, cycle = sleep_cycle)
# A tibble: 5 × 11
  name    genus   vore   order conservation total    rem   cycle awake brainwt bodywt
  <chr>   <chr>   <chr>   <chr>   <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 Owl     Aotus    omni    Prim... <NA>      17     1.8 NA      7     0.0155  0.48
2 Patas   Eryth... omni    Prim... lc       10.9    1.1 NA     13.1    0.115   10
3 Macaque Macaca   omni    Prim... <NA>      10.1    1.2  0.75  13.9    0.179   6.8
4 Slow l... Nycti... carni   Prim... <NA>      11     NA    NA     13     0.0125  1.4
5 Potto   Perod... omni    Prim... lc       11     NA    NA     13     NA      1.1
```

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   rename(total=sleep_total, rem=sleep_rem, cycle=sleep_cycle)
# A tibble: 5 × 11
  name    genus   vore   order   conservation total    rem   cycle awake brainwt bodywt
  <chr>   <chr>   <chr>   <chr>   <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 Owl monkey Aotus    omni    Primates <NA>      17     1.8 NA      7     0.0155  0.48
2 Patas monkey Erythrocebus omni    Primates lc       10.9    1.1 NA     13.1    0.115   10
3 Macaque Macaca   omni    Primates <NA>      10.1    1.2  0.75  13.9    0.179   6.8
4 Slow loris Nyctibeus carni   Primates <NA>      11     NA    NA     13     0.0125  1.4
5 Potto   Perodicticus omni    Primates lc       11     NA    NA     13     NA      1.1
```

○

Data with renamed columns names using `rename()`

Reordering

In addition to filtering rows and columns, often, you'll want the data arranged in a particular order. It may order the columns in a logical way, or it could be to sort the data so that the data are sorted by value, with those having the smallest value in the first row and the largest value in the last row. All of this can be achieved with a few simple functions.

Reordering Columns

The `select()` function is powerful. Not only will it filter and rename columns, but it can also be used to reorder your columns. Using our example from above, if you wanted `sleep_rem` to be the first sleep column and `sleep_total` to be the last column, all you have to do is reorder them within `select()`. The output from `select()` would then be reordered to match the order specified within `select()`.

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, sleep_rem, sleep_cycle, sleep_total)
# A tibble: 5 × 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>      <dbl>
1 Owl monkey    1.8       NA        17
2 Patas monkey   1.1       NA       10.9
3 Macaque        1.2      0.75      10.1
4 Slow loris     NA        NA        11
5 Potto          NA        NA        11
```

Here we see that `sleep_rem` `name` is displayed first followed by `sleep_rem`, `sleep_cycle`, and `sleep_total`, just as it was specified within `select()`.

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   select(name, sleep_rem, sleep_cycle, sleep_total)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>       <dbl>
1 Owl monkey    1.80        NA        17.0
2 Patas monkey   1.10        NA        10.9
3 Macaque       1.20      0.750       10.1
4 Slow loris      NA        NA        11.0
5 Potto          NA        NA        11.0
```

○

Data with reordered columns names

Reordering Rows

Rows can also be reordered. To reorder a variable in ascending order (from smallest to largest), you'll want to use `arrange()`. Continuing on from our example above, to now sort our rows by the amount of total sleep each mammal gets, we would use the following syntax:

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, sleep_rem, sleep_cycle, sleep_total) %>%
  arrange(sleep_total)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>       <dbl>
1 Macaque     1.2        0.75       10.1
2 Patas monkey  1.1        NA        10.9
3 Slow loris    NA        NA        11.0
4 Potto        NA        NA        11.0
5 Owl monkey    1.8        NA        17.0
```

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   select(name, sleep_rem, sleep_cycle, sleep_total) %>%
+   arrange(sleep_total)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>      <dbl>
1 Macaque    1.20      0.750     10.1
2 Patas monkey 1.10      NA        10.9
3 Slow loris  NA        NA        11.0
4 Potto      NA        NA        11.0
5 Owl monkey  1.80      NA        17.0
```

smallest ↑
largest ↓

Data arranged by total sleep in ascending order

While `arrange` sorts variables in ascending order, it's also possible to sort in descending (largest to smallest) order. To do this you just use `desc()` with the following syntax:

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, sleep_rem, sleep_cycle, sleep_total) %>%
  arrange(desc(sleep_total))
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>      <dbl>
1 Owl monkey  1.8      NA        17
2 Slow loris  NA       NA        11
3 Potto      NA       NA        11
4 Patas monkey 1.1      NA        10.9
5 Macaque    1.2      0.75     10.1
```

By putting `sleep_total` within `desc()`, `arrange()` will now sort your data from the primates with the longest total sleep to the shortest.

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   select(name, sleep_rem, sleep_cycle, sleep_total) %>%
+   arrange(sleep_total)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>      <dbl>
1 Macaque    1.20      0.750     10.1
2 Patas monkey 1.10      NA        10.9
3 Slow loris  NA        NA        11.0
4 Potto      NA        NA        11.0
5 Owl monkey  1.80      NA        17.0
```




○

Data arranged by total sleep in descending order

`arrange()` can also be used to order non-numeric variables. For example, `arrange()` will sort character vectors alphabetically.

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, sleep_rem, sleep_cycle, sleep_total) %>%
  arrange(name)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>      <dbl>
1 Macaque    1.2      0.75      10.1
2 Owl monkey 1.8      NA        17.0
3 Patas monkey 1.1      NA        10.9
4 Potto      NA        NA        11.0
5 Slow loris  NA        NA        11.0
```

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   select(name, sleep_rem, sleep_cycle, sleep_total) %>%
+   arrange(name)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>       <dbl>
1 Macaque    1.20      0.750      10.1
2 Owl monkey 1.80      NA          17.0
3 Patas monkey 1.10      NA          10.9
4 Potto       NA          NA          11.0
5 Slow loris  NA          NA          11.0
```

Sorted alphabetically by name

Sort alphabetically by name, then total sleep:
arrange(name, sleep_total)

Data arranged alphabetically by name

If you would like to reorder rows based on information in multiple columns, you can specify them separated by commas. This is useful if you have repeated labels in one column and want to sort within a category based on information in another column. In the example here, if there were repeated primates, this would sort the repeats based on their total sleep.

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, sleep_rem, sleep_cycle, sleep_total) %>%
  arrange(name, sleep_total)
# A tibble: 5 x 4
  name      sleep_rem sleep_cycle sleep_total
  <chr>     <dbl>      <dbl>       <dbl>
1 Macaque    1.2        0.75      10.1
2 Owl monkey 1.8        NA          17
3 Patas monkey 1.1        NA          10.9
4 Potto       NA         NA          11
5 Slow loris  NA         NA          11
```

Creating New Columns

You will often find when working with data that you need an additional column. For example, if you had two datasets you wanted to combine, you may want to make a new column in each dataset called `dataset`. In one dataset you may put `datasetA` in each row. In the second dataset, you could put `datasetB`. This way, once you combined the data, you would be able to keep track of which dataset each row came from originally. More often, however, you'll likely want to create a new column that calculates a new variable based on information in a column you already have. For example, in our mammal sleep dataset, `sleep_total` is in hours. What if you wanted to have that information in minutes? You could create a new column with this very information! The function `mutate()` was *made* for all of these new-column-creating situations. This function has a lot of capabilities. We'll cover the basics here.

Returning to our `msleep` dataset, after filtering and re-ordering, we can create a new column with `mutate()`. Within `mutate()`, we will calculate the number of minutes each mammal sleeps by multiplying the number of hours each animal sleeps by 60 minutes.

```
msleep %>%
  filter(order == "Primates", sleep_total > 10) %>%
  select(name, sleep_rem, sleep_cycle, sleep_total) %>%
  arrange(name) %>%
  mutate(sleep_total_min = sleep_total * 60)
# A tibble: 5 × 5
  name      sleep_rem sleep_cycle sleep_total sleep_total_min
  <chr>     <dbl>     <dbl>       <dbl>        <dbl>
1 Macaque    1.2       0.75       10.1         606
2 Owl monkey 1.8       NA          17           1020
3 Patas monkey 1.1       NA          10.9         654
4 Potto       NA         NA          11           660
5 Slow loris  NA         NA          11           660
```

```
> msleep %>%
+   filter(order == "Primates", sleep_total > 10) %>%
+   select(name, sleep_rem, sleep_cycle, sleep_total) %>%
+   arrange(name) %>%
+   mutate(sleep_total_min = sleep_total * 60)
# A tibble: 5 x 5
  name      sleep_rem sleep_cycle sleep_total sleep_total_min
  <chr>     <dbl>     <dbl>       <dbl>        <dbl>
1 Macaque    1.20     0.750      10.1         606
2 Owl monkey 1.80     NA          17.0        1020
3 Patas monkey 1.10     NA          10.9         654
4 Potto       NA        NA          11.0         660
5 Slow loris  NA        NA          11.0         660
```

A whole new column!

Mutate to add new column to data

Separating Columns

Sometimes multiple pieces of information are merged within a single column even though it would be more useful during analysis to have those pieces of information in separate columns. To demonstrate, we'll now move from the `msleep` dataset to talking about another dataset that includes information about conservation abbreviations in a single column.

To read this file into R, we'll use the `readr` package.

```
## download file
conservation <- read_csv("https://raw.githubusercontent.com/suzanbaert/Dplyr_Tutorials/master/conservation_explanation.csv")

## take a look at this file
conservation
# A tibble: 11 x 1
`conservation abbreviation`
<chr>
```

```
1 EX = Extinct
2 EW = Extinct in the wild
3 CR = Critically Endangered
4 EN = Endangered
5 VU = Vulnerable
6 NT = Near Threatened
7 LC = Least Concern
8 DD = Data deficient
9 NE = Not evaluated
10 PE = Probably extinct (informal)
11 PEW = Probably extinct in the wild (informal)
```

```
> conservation
```

```
# A tibble: 11 x 1
```

```
`conservation abbreviation`  
<chr>
```

```
1 EX = Extinct
2 EW = Extinct in the wild
3 CR = Critically Endangered
4 EN = Endangered
5 VU = Vulnerable
6 NT = Near Threatened
7 LC = Least Concern
8 DD = Data deficient
9 NE = Not evaluated
10 PE = Probably extinct (informal)
11 PEW = Probably extinct in the wild (informal)
```

Tidy data violation!

Space in column name should be an underscore.

Tidy data violation!

There are two pieces of information in a column: (1) abbreviation and (2) description.

Conservation dataset

In this dataset, we see that there is a single column that includes *both* the abbreviation for the conservation term as well as what that abbreviation means. Recall that this violates one of the tidy data principles covered in the first lesson: Put just one thing in a cell. To work with these data, you could imagine that you may want these two pieces of information (the abbreviation and the description) in two different columns. To accomplish this in R, you'll want to use `separate()` from `tidyverse`.

The `separate()` function requires the name of the existing column that you want to separate

(conservation abbreviation), the desired column names of the resulting separated columns (into = c("abbreviation", "description")), and the characters that currently separate the pieces of information (sep = " = "). We have to put conservation abbreviation in back ticks in the code below because the column name contains a space. Without the back ticks, R would think that conservation and abbreviation were two separate things. This is another violation of tidy data! Variable names should have underscores, not spaces!

```
conservation %>%
  separate(`conservation abbreviation`,
           into = c("abbreviation", "description"), sep = " = ")
# A tibble: 11 × 2
  abbreviation description
  <chr>       <chr>
1 EX          Extinct
2 EW          Extinct in the wild
3 CR          Critically Endangered
4 EN          Endangered
5 VU          Vulnerable
6 NT          Near Threatened
7 LC          Least Concern
8 DD          Data deficient
9 NE          Not evaluated
10 PE         Probably extinct (informal)
11 PEW        Probably extinct in the wild (informal)
```

The output of this code shows that we now have two separate columns with the information in the original column separated out into abbreviation and description.

```
> conservation %>%
+   separate(`conservation abbreviation`,
+             into = c("abbreviation", "description"), sep = " = ")
# A tibble: 11 x 2
  abbreviation description
  <chr>        <chr>
1 EX           Extinct
2 EW           Extinct in the wild
3 CR           Critically Endangered
4 EN           Endangered
5 VU           Vulnerable
6 NT           Near Threatened
7 LC           Least Concern
8 DD           Data deficient
9 NE           Not evaluated
10 PE          Probably extinct (informal)
11 PEW         Probably extinct in the wild (informal)
```

Output of separate()

Merging Columns

The opposite of `separate()` is `unite()`. So, if you have information in two or more different columns but wish it were in one single column, you'll want to use `unite()`. Using the code forming the two separate columns above, we can then add on an extra line of `unite()` code to re-join these separate columns, returning what we started with.

```
conservation %>%
  separate(`conservation abbreviation`,
           into = c("abbreviation", "description"), sep = " = ") %>%
  unite(united_col, abbreviation, description, sep = " = ")
# A tibble: 11 x 1
  united_col
  <chr>
1 EX = Extinct
2 EW = Extinct in the wild
3 CR = Critically Endangered
4 EN = Endangered
```

```
5 VU = Vulnerable
6 NT = Near Threatened
7 LC = Least Concern
8 DD = Data deficient
9 NE = Not evaluated
10 PE = Probably extinct (informal)
11 PEW = Probably extinct in the wild (informal)

> conservation %>%
+   separate(`conservation abbreviation`,
+             into = c("abbreviation", "description"), sep = " = ")
+   unite(united_col, abbreviation, description, sep=" = ")
# A tibble: 11 x 1
  united_col
  <chr>
1 EX = Extinct
2 EW = Extinct in the wild
3 CR = Critically Endangered
4 EN = Endangered
5 VU = Vulnerable
6 NT = Near Threatened
7 LC = Least Concern
8 DD = Data deficient
9 NE = Not evaluated
10 PE = Probably extinct (informal)
11 PEW = Probably extinct in the wild (informal)
```

○

Output of `unite()`

Cleaning Column Names

While maybe not quite as important as some of the other functions mentioned in this lesson, a function that will likely prove very helpful as you start analyzing lots of different datasets is `clean_names()` from the `janitor` package. This function takes the existing column names of your dataset, converts them all to lowercase letters and numbers, and separates all words using the underscore character. For example, there is a space in the column name for `conservation`. The `clean_names()` function will convert `conservation abbreviation` to `conservation_abbreviation`. These cleaned up column names are a lot easier to work with when you have large datasets.

So remember this is what the data first looked like:

```
> conservation
# A tibble: 11 × 1
  `conservation abbreviation` <chr>
1 EX = Extinct
2 EW = Extinct in the wild
3 CR = Critically Endangered
4 EN = Endangered
5 VU = Vulnerable
6 NT = Near Threatened
7 LC = Least Concern
8 DD = Data deficient
9 NE = Not evaluated
10 PE = Probably extinct (informal)
11 PEW = Probably extinct in the wild (informal)
```

Tidy data violation!

Space in column name should be an underscore.

Tidy data violation!

There are two pieces of information in a column: (1) abbreviation and (2) description.

Conservation dataset

And now with “clean names” it looks like this:

```
conservation %>%
  clean_names()
# A tibble: 11 × 1
  conservation_abbreviation <chr>
1 EX = Extinct
2 EW = Extinct in the wild
3 CR = Critically Endangered
4 EN = Endangered
5 VU = Vulnerable
6 NT = Near Threatened
7 LC = Least Concern
8 DD = Data deficient
9 NE = Not evaluated
10 PE = Probably extinct (informal)
11 PEW = Probably extinct in the wild (informal)
```

```
> conservation %>%  
+   clean_names()  
# A tibble: 11 x 1  
  conservation_abbreviation  
  <chr>  
1 EX = Extinct          Adds underscore to column name  
2 EW = Extinct in the wild  
3 CR = Critically Endangered  
4 EN = Endangered  
5 VU = Vulnerable  
6 NT = Near Threatened  
7 LC = Least Concern  
8 DD = Data deficient  
9 NE = Not evaluated  
10 PE = Probably extinct (informal)  
11 PEW = Probably extinct in the wild (informal)
```

clean_names() output

Combining Data Across Data Frames

There is often information stored in two separate data frames that you'll want in a single data frame. There are *many* different ways to join separate data frames. They are discussed in more detail in [this tutorial](#) from Jenny Bryan. Here, we'll demonstrate how the `left_join()` function works, as this is used frequently.

Let's try to combine the information from the two different datasets we've used in this lesson. We have `msleep` and `conservation`. The `msleep` dataset contains a column called `conservation`. This column includes lowercase abbreviations that overlap with the uppercase abbreviations in the `abbreviation` column in the `conservation` dataset.

To handle the fact that in one dataset the abbreviations are lowercase and the other they are uppercase, we'll use `mutate()` to take all the lowercase abbreviations to uppercase abbreviations using the function `toupper()`.

We'll then use `left_join()` which takes all of the rows in the first dataset mentioned (`msleep`, below) and incorporates information from the second dataset mentioned (`conservation`, below), when information in the second dataset is available. The `by =` argument states what columns to join by in the first ("conservation") and second ("abbreviation") datasets.

This join adds the `description` column from the `conserve` dataset onto the original dataset (`msleep`). Note that if there is no information in the second dataset that matches with the information in the first dataset, `left_join()` will add NA. Specifically, for rows where conservation is “DOMESTICATED” below, the `description` column will have NA because “DOMESTICATED” is not an abbreviation in the `conserve` dataset.

```
## take conservation dataset and separate information
## into two columns
## call that new object `conserve`
conserve <- conservation %>%
  separate(`conservation abbreviation`,
           into = c("abbreviation", "description"), sep = " = ")

## now lets join the two datasets together
msleep %>%
  mutate(conservation = toupper(conservation)) %>%
  left_join(conserve, by = c("conservation" = "abbreviation"))
```

```
> msleep %>%
+   mutate(conservation = toupper(conservation)) %>%
+   left_join(conserve, by = c("conservation" = "abbreviation"))
# A tibble: 83 x 12
   name   genus  vore  order conservation sleep_total sleep_rem sleep_cycle awake  brainwt  bodywt description
   <chr>  <chr> <chr> <chr> <dbl>      <dbl>      <dbl> <dbl>    <dbl>    <dbl> <chr>
1 Cheetah Acinod. carni Carniv. LC     12.1      NA        NA    11.9    NA    5.00e+1 Least Conc...
2 Owl mon. Aotus omni Prim. <NA>     17.0      1.80      NA    7.00    1.55e-2 4.80e-1 <NA>
3 Mountain. Aplop. herbi Rodeo. NT     14.4      2.40      NA    9.60    NA    1.35e+0 Near Threa...
4 Greater. Blari. omni Sori. LC     14.9      2.30      0.133   9.10    2.90e-4 1.90e-2 Least Conc...
5 Cow      Bos    herbi Arti. DOMESTICATED 4.00      0.700     0.667   20.0    4.23e-1 6.00e+2 <NA>
6 Three-t. Brady. herbi Pilo. <NA>     14.4      2.20      0.767   9.60    NA    3.85e+0 <NA>
7 Northern. Callo. carni Carn. VU     8.70      1.40      0.383   15.3    NA    2.05e+1 Vulnerable
8 Vesper. Calom. <NA> Rode. <NA>     7.00      NA        NA    17.0    NA    4.50e-2 <NA>
9 Dog      Canis carni Carn. DOMESTICATED 10.1      2.90      0.333   13.9    7.00e-2 1.40e+1 <NA>
10 Roe deer Capre. herbi Arti. LC     3.00      NA        NA    21.0    9.82e-2 1.48e+1 Least Conc...
```



Data resulting from `left_join`

It's important to note that there are many other ways to join data, which we covered earlier in a previous course and are covered in more detail on this [dplyr join cheatsheet](#) from Jenny Bryan. For now, it's important to know that joining datasets is done easily in R using tools in `dplyr`. As you join data frames in your own work, it's a good idea to refer back to this cheatsheet for assistance.

Grouping Data

Often, data scientists will want to summarize information in their dataset. You may want to know how many people are in a dataset. However, more often, you'll want to know how many people there are within a group in your dataset. For example, you may want to know how many males and how many females there are. To do this, grouping your data is necessary. Rather than looking at the total number of individuals, to accomplish this, you first have to **group the data** by the gender of the individuals. Then, you count within those groups. Grouping by variables within `dplyr` is straightforward.

`group_by()`

There is an incredibly helpful function within `dplyr` called `group_by()`. The `group_by()` function groups a dataset by one or more variables. On its own, it does not appear to change the dataset very much. The difference between the two outputs below is subtle:

```
msleep  
msleep %>%  
  group_by(order)
```

```
> msleep
# A tibble: 83 x 11
   name    genus  vore order  conservation sleep_total sleep_rem sleep_cycle awake brainwt bodywt
   <chr>   <chr> <chr> <chr> <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
1 Cheetah Acinonyx carni Carniva... lc     12.1       NA        NA    11.9  NA    5.00e+1
2 Owl monkey Aotus omni Primates <NA>    17.0       1.80      NA    7.00  1.55e-2 4.80e-1
3 Mountain beaver Aplodon herbi Rodentia nt   14.4       2.40      NA    9.60  NA    1.35e+0
4 Greater short-t... Blarina omni Soricom... lc    14.9       2.30      0.133  9.10  2.90e-4 1.90e-2
5 Cow      Bos    herbi Artioda... domesticated 4.00       0.700     0.667 20.0   4.23e-1 6.00e-2
6 Three-toed slo... Bradypus herbi Pilosa <NA>   14.4       2.20      0.767  9.60  NA    3.85e+0
7 Northern fur s... Callorh... carni Carniva... vu   8.70       1.40      0.383 15.3   NA    2.05e-1
8 Vesper mouse Calomys <NA>  Rodentia <NA>    7.00       NA        NA    17.0  NA    4.50e-2
9 Dog      Canis carni Carniva... domesticated 10.1       2.90      0.333 13.9   7.00e-2 1.40e-1
10 Roe deer Capreol herbi Artioda... lc    3.00       NA        NA    21.0  NA    9.82e-2 1.48e-1
# ... with 73 more rows

> msleep %>%
+   group_by(order)
# A tibble: 83 x 11
# Groups:   order [19]
   name    genus  vore order  conservation sleep_total sleep_rem sleep_cycle awake brainwt bodywt
   <chr>   <chr> <chr> <chr> <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
1 Cheetah Acinonyx carni Carniva... lc     12.1       NA        NA    11.9  NA    5.00e+1
2 Owl monkey Aotus omni Primates <NA>    17.0       1.80      NA    7.00  1.55e-2 4.80e-1
3 Mountain beaver Aplodon herbi Rodentia nt   14.4       2.40      NA    9.60  NA    1.35e+0
4 Greater short-t... Blarina omni Soricom... lc    14.9       2.30      0.133  9.10  2.90e-4 1.90e-2
5 Cow      Bos    herbi Artioda... domesticated 4.00       0.700     0.667 20.0   4.23e-1 6.00e-2
6 Three-toed slo... Bradypus herbi Pilosa <NA>   14.4       2.20      0.767  9.60  NA    3.85e+0
7 Northern fur s... Callorh... carni Carniva... vu   8.70       1.40      0.383 15.3   NA    2.05e-1
8 Vesper mouse Calomys <NA>  Rodentia <NA>    7.00       NA        NA    17.0  NA    4.50e-2
9 Dog      Canis carni Carniva... domesticated 10.1       2.90      0.333 13.9   7.00e-2 1.40e-1
10 Roe deer Capreol herbi Artioda... lc    3.00       NA        NA    21.0  NA    9.82e-2 1.48e-1
# ... with 73 more rows
```

group_by() output

In fact, the only aspect of the output that is different is that the number of different orders is now printed on your screen. However, in the next section, you'll see that the output from any further functions you carry out at this point will differ between the two datasets.

Summarizing Data

Throughout data cleaning and analysis it will be important to summarize information in your dataset. This may be for a formal report or for checking the results of a data tidying operation.

`summarize()`

Continuing on from the previous examples, if you wanted to figure out how many samples are present in your dataset, you could use the `summarize()` function.

```
msleep %>%
# here we select the column called genus, any column would work
  select(genus) %>%
  summarize(N=n())
# A tibble: 1 × 1
      N
  <int>
1     83

msleep %>%
# here we select the column called vore, any column would work
  select(vore) %>%
  summarize(N=n())
# A tibble: 1 × 1
      N
  <int>
1     83
```

This provides a summary of the data with the new column name we specified above (`N`) and the number of samples in the dataset. Note that we could also obtain the same information by directly obtaining the number of rows in the data frame with `nrow(msleep)`.

```
> msleep %>%  
+   select(order) %>%  
+   summarize(N=n())  
# A tibble: 1 x 1  
      N  
  <int>  
1     83  
Same as nrow(msleep)
```



Summarize with n()

However, if you wanted to count how many of each different `order` of mammal you had. You would first `group_by(order)` and then use `summarize()`. This will summarize within group.

```
msleep %>%  
  group_by(order) %>%  
  select(order) %>%  
  summarize(N=n())  
# A tibble: 19 x 2  
  order          N  
  <chr>     <int>  
1 Afrosoricida    1  
2 Artiodactyla     6  
3 Carnivora       12  
4 Cetacea          3  
5 Chiroptera       2  
6 Cingulata         2  
7 Didelphimorphia   2  
8 Diprotodontia     2  
9 Erinaceomorpha     2
```

10	Hyracoidea	3
11	Lagomorpha	1
12	Monotremata	1
13	Perissodactyla	3
14	Pilosa	1
15	Primates	12
16	Proboscidea	2
17	Rodentia	22
18	Scandentia	1
19	Soricomorpha	5

The output from this, like above, includes the column name we specified in summarize (N). However, it includes the number of samples in the group_by variable we specified (order).

```
> msleep %>%
+   group_by(order) %>%
+   select(-order) %>%
+   summarize(N=n())
# A tibble: 19 x 2
  order       N
  <chr>     <int>
1 Afrosoricida    1
2 Artiodactyla     6
3 Carnivora      12
4 Cetacea          3
5 Chiroptera       2
6 Cingulata         2
7 Didelphimorphia   2
8 Diprotodontia     2
9 Erinaceomorpha    2
10 Hyracoidea       3
11 Lagomorpha       1
12 Monotremata      1
13 Perissodactyla    3
14 Pilosa           1
15 Primates        12
16 Proboscidea       2
17 Rodentia         22
18 Scandentia        1
19 Soricomorpha      5
```

group_by() and summarize with n()

There are other ways in which the data can be summarized using summarize(). In addition to using n() to count the number of samples within a group, you can also summarize using other helpful functions within R, such as mean(), median(), min(), and max().

For example, if we wanted to calculate the average (mean) total sleep each order of mammal got, we could use the following syntax:

```
msleep %>%
  group_by(order) %>%
  select(order, sleep_total) %>%
  summarize(N=n(), mean_sleep=mean(sleep_total))

# A tibble: 19 × 3
# Groups:   order [19]
  order        N  mean_sleep
  <chr>     <int>     <dbl>
1 Afrosoricida     1      15.6
2 Artiodactyla      6      4.52
3 Carnivora        12     10.1
4 Cetacea           3      4.5
5 Chiroptera        2     19.8
6 Cingulata          2     17.8
7 Didelphimorphia    2     18.7
8 Diprotodontia      2     12.4
9 Erinaceomorpha     2     10.2
10 Hyracoidea        3      5.67
11 Lagomorpha         1      8.4
12 Monotremata       1      8.6
13 Perissodactyla     3      3.47
14 Pilosa             1     14.4
15 Primates           12     10.5
16 Proboscidea        2      3.6
17 Rodentia            22     12.5
18 Scandentia          1      8.9
19 Soricomorpha        5     11.1
```

```
> msleep %>%  
+   group_by(order) %>%  
+   select(order, sleep_total) %>%  
+   summarize(N=n(), mean_sleep=mean(sleep_total))  
# A tibble: 19 x 3  
  order      N  mean_sleep  
  <chr>    <int>     <dbl>  
1 Afrosoricida     1     15.6  
2 Artiodactyla      6      4.52  
3 Carnivora        12     10.1  
4 Cetacea           3      4.50  
5 Chiroptera        2     19.8  
6 Cingulata          2     17.8  
7 Didelphimorphia    2     18.7  
8 Diprotodontia      2     12.4  
9 Erinaceomorpha     2     10.2  
10 Hyracoidea        3      5.67  
11 Lagomorpha        1      8.40  
12 Monotremata       1      8.60  
13 Perissodactyla     3      3.47  
14 Pilosa             1     14.4  
15 Primates           12     10.5  
16 Proboscidea         2      3.60  
17 Rodentia           22     12.5  
18 Scandentia          1      8.90  
19 Soricomorpha        5     11.1
```

summarize using mean()

`tabyl()`

In addition to using `summarize()` from `dplyr`, the `tabyl()` function from the `janitor` package can be incredibly helpful for summarizing categorical variables quickly and discerning the output at a glance. It is similar to the `table()` function from base R, but is explicit about missing data, rather than ignoring missing values by default.

Again returning to our `msleep` dataset, if we wanted to get a summary of how many samples are in each order category and what percent of the data fall into each category we could call `tabyl` on that variable. For example, if we use the following syntax, we easily get a quick snapshot of this variable.

```
msleep %>%  
  tabyl(order)  
    order  n      percent  
  Afrosoricida  1  0.01204819  
  Artiodactyla  6  0.07228916  
  Carnivora   12 0.14457831  
  Cetacea     3  0.03614458  
  Chiroptera   2  0.02409639  
  Cingulata    2  0.02409639  
  Didelphimorphia  2  0.02409639  
  Diprotodontia  2  0.02409639  
  Erinaceomorpha  2  0.02409639  
  Hyracoidea   3  0.03614458  
  Lagomorpha   1  0.01204819  
  Monotremata   1  0.01204819  
  Perissodactyla 3  0.03614458  
  Pilosa       1  0.01204819  
  Primates    12 0.14457831  
  Proboscidea   2  0.02409639  
  Rodentia     22 0.26506024  
  Scandentia    1  0.01204819  
  Soricomorpha  5  0.06024096
```

```
> msleep %>%
+   tabyl(order)
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
#> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #> #>
```

summarize using tabyl() from janitor

Note, that `tabyl` assumes categorical variables. If you want to summarize numeric variables `summary()` works well. For example, this code will summarize the values in `msleep$awake` for you.

```
summary(msleep$awake)
Min. 1st Qu. Median Mean 3rd Qu. Max.
4.10    10.25   13.90  13.57   16.15  22.10
```

```
> summary(msleep$awake)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
4.1	10.2	13.9	13.6	16.1	22.1

○

summarize numeric variables

```
tally()
```

We can use the tally function to get the total number of samples in a tibble or the total number of rows very simply.

```
msleep %>%  
  tally()  
# A tibble: 1 × 1  
      n  
  <int>  
1     83
```

We can see that this is very similar to our previous use of summarize.

```
msleep %>%  
  # here we select the column called genus, any column would work  
  select(genus) %>%  
  summarise(N=n())  
# A tibble: 1 × 1  
      N  
  <int>  
1     83
```

We can also use this function to get a sum of the values of a column (if the values are numeric).

```
msleep %>%  
  tally(sleep_total)  
# A tibble: 1 × 1  
      n  
  <dbl>  
1     866
```

Thus overall, all the animals in the dataset sleep 866 hours in total.

This is the equivalent to using the `sum()` function with the `summarize()` function.

```
msleep %>%  
  summarize(sum_sleep_total = sum(sleep_total))  
# A tibble: 1 × 1  
  sum_sleep_total  
  <dbl>  
1     866
```

We could also use the `pull()` function of the `dplyr` package, to get the sum of just the `sleep_total` column, as the `pull()` function extracts or “pulls” the values of a column.

```
msleep %>%  
  pull(sleep_total)%>%  
  sum()  
[1] 866
```

add_tally()

We can quickly add our tally values to our tibble using `add_tally()`.

```
msleep %>%
  add_tally() %>%
  glimpse()
Rows: 83
Columns: 12
$ name      <chr> "Cheetah", "Owl monkey", "Mountain beaver", "Greater shor...
$ genus     <chr> "Acinonyx", "Aotus", "Aplodontia", "Blarina", "Bos", "Bra...
$ vore       <chr> "carni", "omni", "herbi", "omni", "herbi", "herbi", "carni...
$ order      <chr> "Carnivora", "Primates", "Rodentia", "Soricomorpha", "Art...
$ conservation <chr> "lc", NA, "nt", "lc", "domesticated", NA, "vu", NA, "dome...
$ sleep_total <dbl> 12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 10.1, 3.0, 5...
$ sleep_rem   <dbl> NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA, 0.6, 0.8, ...
$ sleep_cycle <dbl> NA, NA, NA, 0.1333333, 0.6666667, 0.7666667, 0.3833333, N...
$ awake       <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, 21.0, 1...
$ brainwt     <dbl> NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA, 0.07000, 0...
$ bodywt      <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.490, 0.04...
$ n           <int> 83, 83, 83, 83, 83, 83, 83, 83, 83, 83, 83, 83, 83, 83, 8...
```

Notice the new column called “n” that repeats the total number of samples for each row.

Or we can add a column that repeats the total hours of sleep of all the animals.

```
msleep %>%
  add_tally(sleep_total) %>%
  glimpse()
Rows: 83
Columns: 12
$ name      <chr> "Cheetah", "Owl monkey", "Mountain beaver", "Greater shor...
$ genus     <chr> "Acinonyx", "Aotus", "Aplodontia", "Blarina", "Bos", "Bra...
$ vore       <chr> "carni", "omni", "herbi", "omni", "herbi", "herbi", "carni...
$ order      <chr> "Carnivora", "Primates", "Rodentia", "Soricomorpha", "Art...
$ conservation <chr> "lc", NA, "nt", "lc", "domesticated", NA, "vu", NA, "dome...
$ sleep_total <dbl> 12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 10.1, 3.0, 5...
$ sleep_rem   <dbl> NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA, 0.6, 0.8, ...
$ sleep_cycle <dbl> NA, NA, NA, 0.1333333, 0.6666667, 0.7666667, 0.3833333, N...
```

```
$ awake      <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, 21.0, 1...
$ brainwt    <dbl> NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA, 0.07000, 0...
$ bodywt     <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.490, 0.04...
$ n          <dbl> 866, 866, 866, 866, 866, 866, 866, 866, 866, 86...
```

`count()`

The `count()` function takes the `tally()` function a step further to determine the count of unique values for specified variable(s)/column(s).

```
msleep %>%
  count(vore)
# A tibble: 5 × 2
  vore     n
  <chr>   <int>
1 carni     19
2 herbi     32
3 insecti    5
4 omni      20
5 <NA>       7
```

This is the same as using `group_by()` with `tally()`

```
msleep %>%
  group_by(vore) %>%
  tally()
# A tibble: 5 × 2
  vore     n
  <chr>   <int>
1 carni     19
2 herbi     32
3 insecti    5
4 omni      20
5 <NA>       7
```

Multiple variables can be specified with `count()`.

This can be really useful when getting to know your data.

```
msleep %>%
  count(vore, order)
# A tibble: 32 × 3
  vore   order       n
  <chr> <chr>     <int>
1 carni Carnivora    12
2 carni Cetacea      3
3 carni Cingulata    1
4 carni Didelphimorphia 1
5 carni Primates     1
6 carni Rodentia     1
7 herbi Artiodactyla  5
8 herbi Diprotodontia 1
9 herbi Hyracoidea    2
10 herbi Lagomorpha    1
# ... with 22 more rows
```

```
add_count()
```

The `add_count()` function is similar to the `add_tally()` function:

```
msleep %>%
  add_count(vore, order) %>%
  glimpse()
Rows: 83
Columns: 12
$ name      <chr> "Cheetah", "Owl monkey", "Mountain beaver", "Greater shor...
$ genus     <chr> "Acinonyx", "Aotus", "Aplodontia", "Blarina", "Bos", "Bra...
$ vore       <chr> "carni", "omni", "herbi", "omni", "herbi", "herbi", "carni...
$ order      <chr> "Carnivora", "Primates", "Rodentia", "Soricomorpha", "Art...
$ conservation <chr> "lc", NA, "nt", "lc", "domesticated", NA, "vu", NA, "dome...
$ sleep_total <dbl> 12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 10.1, 3.0, 5...
$ sleep_rem   <dbl> NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA, 0.6, 0.8, ...
$ sleep_cycle <dbl> NA, NA, NA, 0.1333333, 0.6666667, 0.7666667, 0.3833333, N...
$ awake       <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, 21.0, 1...
$ brainwt     <dbl> NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA, 0.07000, 0...
$ bodywt      <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.490, 0.04...
$ n           <int> 12, 10, 16, 3, 5, 1, 12, 3, 12, 5, 5, 16, 10, 16, 3, 2, 3...
```

get_dupes()

Another common issue in data wrangling is the presence of duplicate entries. Sometimes you *expect* multiple observations from the same individual in your dataset. Other times, the information has accidentally been added more than once. The `get_dupes()` function becomes very helpful in this situation. If you want to identify duplicate entries during data wrangling, you'll use this function and specify which columns you're looking for duplicates in.

For example, in the `msleep` dataset, if you expected to only have one mammal representing each genus and vore you could double check this using `get_dupes()`.

```
# identify observations that match in both genus and vore
msleep %>%
  get_dupes(genus, vore)
# A tibble: 10 × 12
  genus      vore  dupe_count name   order conservation sleep_total sleep_rem
  <chr>     <chr>    <int> <chr>  <chr>    <chr>        <dbl>      <dbl>
1 Equus     herbi       2 Horse  Peri... domesticated      2.9       0.6
2 Equus     herbi       2 Donkey Peri... domesticated      3.1       0.4
3 Panthera carni       3 Tiger   Carn... en            15.8      NA
4 Panthera carni       3 Jaguar  Carn... nt            10.4      NA
5 Panthera carni       3 Lion    Carn... vu            13.5      NA
6 Spermophilus herbi   3 Arcti... Rode... lc            16.6      NA
7 Spermophilus herbi   3 Thirt... Rode... lc            13.8      3.4
8 Spermophilus herbi   3 Golde... Rode... lc            15.9      3
9 Vulpes     carni       2 Arcti... Carn... <NA>          12.5      NA
10 Vulpes    carni      2 Red f... Carn... <NA>           9.8       2.4
# ... with 4 more variables: sleep_cycle <dbl>, awake <dbl>, brainwt <dbl>,
#   bodywt <dbl>
```

The output demonstrates there are 10 mammals that overlap in their genus and vore. Note that the third column of the output counts *how many* duplicate observations there are. This can be very helpful when you're checking your data!

skim()

When you would rather get a snapshot of the entire dataset, rather than just one variable, the `skim()` function from the `skimr` package can be very helpful. The output from `skim()` breaks the data up by variable type. For example, the `msleep` dataset is broken up into character

and numeric variable types. The data are then summarized in a meaningful way for each. This function provides a lot of information about the entire dataset. So, when you want a summarize a dataset and quickly get a sense of your data, `skim()` is a great option!

```
# summarize dataset
skim(msleep)
```

```
> skim(msleep)
Skim summary statistics
n obs: 83
n variables: 11

Variable type: character
  variable missing complete n min max empty n_unique
conservation     29      54 83   2 12    0       6
  genus          0      83 83   3 13    0       77
  name           0      83 83   3 30    0       83
  order          0      83 83   6 15    0       19
  vore           7      76 83   4  7    0       4

Variable type: numeric
  variable missing complete n   mean     sd    p0    p25    p50    p75    p100  hist
  awake          0      83 83 13.57  4.45 4.1  10.25  13.9  16.15  22.1
  bodywt         0      83 83 166.14 786.84 0.005  0.17  1.67  41.75 6654
  brainwt        27      56 83  0.28  0.98 0.00014  0.0029  0.012  0.13  5.71
  sleep_cycle    51      32 83  0.44  0.36 0.12   0.18  0.33  0.58  1.5
  sleep_rem      22      61 83  1.88  1.3  0.1    0.9   1.5   2.4   6.6
  sleep_total     0      83 83 10.43  4.45 1.9   7.85  10.1  13.75 19.9
```

summarize entire dataset using `skim()` from `skimr`

Note that this function allows for you to specify which columns you'd like to summarize, if you're not interested in seeing a summary of the entire dataset:

```
# see summary for specified columns
skim(msleep, genus, vore, sleep_total)
```

```
— Data Summary —————
Name                         Values
msleep
Number of rows                83
Number of columns              11

————— Column type frequency: —————
character                      2
numeric                        1

————— Group variables —————
None

————— Variable type: character —————
skim_variable n_missing complete_rate   min    max empty n_unique whitespace
1 genus                     0        1      3     13      0      77      0
2 vore                      7      0.916     4      7      0       4      0

————— Variable type: numeric —————
skim_variable n_missing complete_rate   mean      sd    p0    p25    p50    p75    p100 hist
1 sleep_total                 0          1    10.4    4.45   1.9    7.85   10.1   13.8   19.9 ████

summarize more specifically with skim() from skimr
```

It is also possible to group data (using dplyr's `group_by()`) before summarizing. Notice in the summary output that each variable specified (`genus` and `sleep_total`) are now broken down within each of the `vore` categories.

```
msleep %>%
  group_by(vore) %>%
  skim(genus, sleep_total)
```

```
— Data Summary —
  Values
Name          Piped data
Number of rows 83
Number of columns 11

Column type frequency:
  character     1
  numeric       1

Group variables   vore

— Variable type: character —
skim_variable vore  n_missing complete_rate   min   max empty n_unique whitespace
1 genus        carni    0             1   5   13   0      16      0
2 genus        herbi    0             1   3   12   0      29      0
3 genus        insecti   0             1   6   12   0      5      0
4 genus        omni     0             1   3   13   0      20      0
5 genus        <NA>     0             1   6   11   0      7      0

— Variable type: numeric —
skim_variable vore  n_missing complete_rate   mean    sd    p0    p25   p50    p75   p100 hist
1 sleep_total  carni    0             1 10.4  4.67  2.7  6.25 10.4  13  19.4 
2 sleep_total  herbi    0             1 9.51  4.88  1.9  4.3   10.3 14.2 16.6 
3 sleep_total  insecti   0             1 14.9  5.92  8.4  8.6   18.1 19.7 19.9 
4 sleep_total  omni     0             1 10.9  2.95  8   9.1   9.9 10.9 18 
5 sleep_total  <NA>     0             1 10.2  3.00  5.4  8.65 10.6 12.2 13.7 
```

summarize groups with skim

```
summary()
```

While base R has a `summary()` function, this can be combined with the `skimr` package to provide you with a quick summary of the dataset at large.

```
skim(msleep) %>%
  summary()
```

<hr/> — Data Summary —	
	Values
Name	msleep
Number of rows	83
Number of columns	11
<hr/>	
Column type frequency:	
character	5
numeric	6
<hr/>	
Group variables	None

summarize entire dataset using `skim()` from `skimr`

Operations Across Columns

Sometimes it is valuable to apply a certain operation across the columns of a data frame. For example, it may be necessary to compute the mean or some other summary statistics for each column in the data frame. In some cases, these operations can be done by a combination of `pivot_longer()` along with `group_by()` and `summarize()`. However, in other cases it is more straightforward to simply compute the statistic on each column.

The `across()` function is needed to operate across the columns of a data frame. For example, in our `airquality` dataset, if we wanted to compute the mean of `Ozone`, `Solar.R`, `Wind`, and `Temp`, we could do:

```
airquality %>%
  summarize(across(Ozone:Temp, mean, na.rm = TRUE))
# A tibble: 1 × 4
  Ozone Solar.R Wind Temp
  <dbl>   <dbl> <dbl> <dbl>
1  42.1    186.  9.96  77.9
```

The `across()` function can be used in conjunction with the `mutate()` and `filter()` functions to construct joint operations across different columns of a data frame. For example, suppose we wanted to filter the rows of the `airquality` data frame so that we only retain rows that do not have missing values for `Ozone` and `Solar.R`. Generally, we might use the `filter()` function for this, as follows:

```
airquality %>%  
  filter(!is.na(Ozone),  
        !is.na(Solar.R))
```

Because we are only filtering on two columns here, it's not too difficult to write out the expression. However, if we were filtering on many columns, it would become a challenge to write out every column. This is where the `across()` function comes in handy. With the `across()` function, we can specify columns in the same way that we use the `select()` function. This allows us to use short-hand notation to select a large set of columns.

We can use the `across()` function in conjunction with `filter()` to achieve the same result as above.

```
airquality %>%  
  filter(across(Ozone:Solar.R, ~ !is.na(.)))  
# A tibble: 111 × 6  
  Ozone Solar.R  Wind   Temp Month Day  
  <int>    <int> <dbl>  <int> <int> <int>  
1     41      190    7.4    67     5     1  
2     36      118     8     72     5     2  
3     12      149   12.6    74     5     3  
4     18      313   11.5    62     5     4  
5     23      299    8.6    65     5     7  
6     19       99   13.8    59     5     8  
7      8       19   20.1    61     5     9  
8     16      256    9.7    69     5    12  
9     11      290    9.2    66     5    13  
10    14      274   10.9    68     5    14  
# ... with 101 more rows
```

Here, the `~` in the call to `across()` indicates that we are passing an anonymous function (see the section on Functional Programming for more details) and the `.` is a stand-in for the name of the column.

If we wanted to filter the data frame to remove rows with missing values in `Ozone`, `Solar.R`, `Wind`, and `Temp`, we only need to make a small change.

```
airquality %>%  
  filter(across(Ozone:Temp, ~ !is.na(.)))  
# A tibble: 111 × 6  
  Ozone Solar.R Wind Temp Month Day  
  <int>   <int> <dbl> <int> <int> <int>  
1     41     190    7.4    67     5     1  
2     36     118     8     72     5     2  
3     12     149   12.6    74     5     3  
4     18     313   11.5    62     5     4  
5     23     299    8.6    65     5     7  
6     19      99   13.8    59     5     8  
7      8      19   20.1    61     5     9  
8     16     256    9.7    69     5    12  
9     11     290    9.2    66     5    13  
10    14     274   10.9    68     5    14  
# ... with 101 more rows
```

The `across()` function can also be used with `mutate()` if we want to apply the same transformation to multiple columns. For example, suppose we want to cycle through each column and replace all missing values (NAs) with zeros. We could use `across()` to accomplish this.

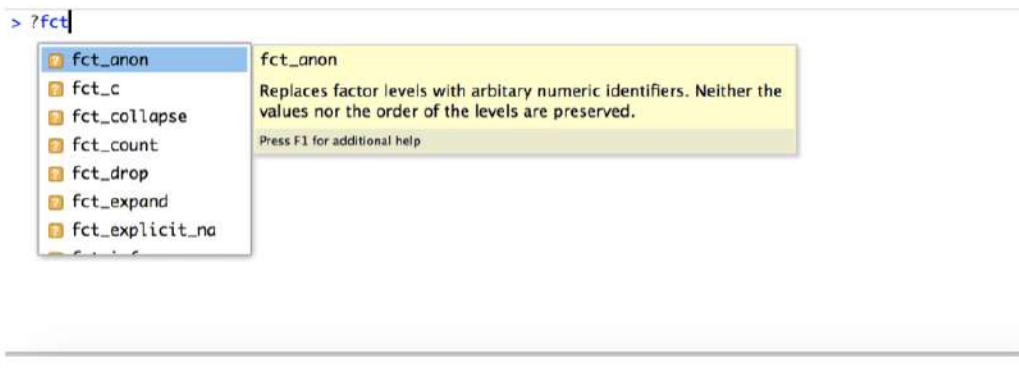
```
airquality %>%  
  mutate(across(Ozone:Temp, ~ replace_na(., 0)))  
# A tibble: 153 × 6  
  Ozone Solar.R Wind Temp Month Day  
  <dbl>   <dbl> <dbl> <dbl> <int> <int>  
1     41     190    7.4    67     5     1  
2     36     118     8     72     5     2  
3     12     149   12.6    74     5     3  
4     18     313   11.5    62     5     4  
5      0      0   14.3    56     5     5  
6     28      0   14.9    66     5     6  
7     23     299    8.6    65     5     7  
8     19      99   13.8    59     5     8  
9      8      19   20.1    61     5     9  
10    14     194    8.6    69     5    10  
# ... with 143 more rows
```

Again, the `.` is used as a stand-in for the name of the column. This expression essentially applies the `replace_na()` function to each of the columns between Ozone and Temp in the data frame.

Working With Factors

In R, categorical data are handled as factors. By definition, categorical data are limited in that they have a set number of possible values they can take. For example, there are 12 months in a calendar year. In a month variable, each observation is limited to taking one of these twelve values. Thus, with a limited number of possible values, month is a categorical variable. Categorical data, which will be referred to as factors for the rest of this lesson, are regularly found in data. Learning how to work with this type of variable effectively will be incredibly helpful.

To make working with factors simpler, we'll utilize the `forcats` package, a core tidyverse package. All functions within `forcats` begin with `fct_`, making them easier to look up and remember. As before, to see available functions you can type `?fct_` in your RStudio console. A drop-down menu will appear with all the possible `forcats` functions.



fct_ output from RStudio

Factor Review

In R, factors are comprised of two components: the actual **values** of the data and the possible **levels** within the factor. Thus, to create a factor, you need to supply both these pieces of information.

For example, if we were to create a character vector of the twelve months, we could certainly do that:

```
## all 12 months
all_months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", \
"Oct", "Nov", "Dec")

## our data
some_months <- c("Mar", "Dec", "Jan", "Apr", "Jul")
```

However, if we were to sort this vector, R would sort this vector alphabetically.

```
# alphabetical sort
sort(some_months)
[1] "Apr" "Dec" "Jan" "Jul" "Mar"
```

```
## all 12 months
all_months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
               "Oct", "Nov", "Dec")

## our data
some_months <- c("Mar", "Dec", "Jan", "Apr", "Jul")
```

```
> sort(some_months)
```

```
[1] "Apr" "Dec" "Jan" "Jul" "Mar"
```

Sorts alphabetically

sort sorts variable alphabetically

While you and I know that this is not how months should be ordered, we haven't yet told R that. To do so, we need to let R know that it's a factor variable and what the levels of that factor variable should be.

```
# create factor
mon <- factor(some_months, levels = all_months)

# look at factor
mon
[1] Mar Dec Jan Apr Jul
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

# look at sorted factor
sort(mon)
[1] Jan Mar Apr Jul Dec
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

```
## all 12 months
all_months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
               "Oct", "Nov", "Dec")

## our data
some_months <- c("Mar", "Dec", "Jan", "Apr", "Jul")

> mon <- factor(some_months, levels = all_months)
>
> mon
[1] Mar Dec Jan Apr Jul
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
>
> sort(mon)
[1] Jan Mar Apr Jul Dec
```

Sorts in order of specified levels
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

○

defining the factor levels sorts this variable sensibly

Here, we specify all the possible values that the factor could take in the `levels = all_months` argument. So, even though not all twelve months are included in the `some_months` object, we've stated that all of the months are possible values. Further, when you sort this variable, it now sorts in the sensible way!

Manually Changing the Labels of Factor Levels: `fct_relevel()`

What if you wanted your months to start with July first? That can be accomplished using `fct_relevel()`. To use this function, you simply need to state what you'd like to relevel (`mon`) followed by the levels you want to relevel. If you want these to be placed in the beginning, the `after` argument should be `after = 0`. You can play around with this setting to see how changing `after` affects the levels in your output.

```
mon_relevel <- fct_relevel(mon, "Jul", "Aug", "Sep", "Oct", "Nov", "Dec", after\\
= 0)

# relevelled
mon_relevel
[1] Mar Dec Jan Apr Jul
Levels: Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr May Jun

# relevelled and sorted
sort(mon_relevel)
[1] Jul Dec Jan Mar Apr
Levels: Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr May Jun
```

```
> mon_relevel <- fct_relevel(mon, "Jul", "Aug", "Sep", "Oct", "Nov", "Dec", after = 0)
>
> mon_relevel
[1] Mar Dec Jan Apr Jul
Levels: Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr May Jun
>
> sort(mon_relevel)
[1] Jul Dec Jan Mar Apr
Levels: Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr May Jun
```

Sorts in order of re-ordered levels



fct_relevel enables you to change the order of your factor levels

After re-leveling, when we sort this factor, we see that Jul is placed first, as specified by the level re-ordering.

Keeping the Order of the Factor Levels: fct_inorder()

Now, if you're not interested in the months being in calendar year order, you can always state that you want the levels to stay in the same order as the data you started with, you

simply specify with `fct_inorder()`.

```
# keep order of appearance
mon_inorder <- fct_inorder(some_months)

# output
mon_inorder
[1] Mar Dec Jan Apr Jul
Levels: Mar Dec Jan Apr Jul

# sorted
sort(mon_inorder)
[1] Mar Dec Jan Apr Jul
Levels: Mar Dec Jan Apr Jul

some_months <- c("Mar", "Dec", "Jan", "Apr", "Jul")

> mon_inorder <- fct_inorder(some_months)
>
> mon_inorder
[1] Mar Dec Jan Apr Jul
Levels: Mar Dec Jan Apr Jul
>
> sort(mon_inorder)
[1] Mar Dec Jan Apr Jul
Levels: Mar Dec Jan Apr Jul

Levels match order of appearance in data
```

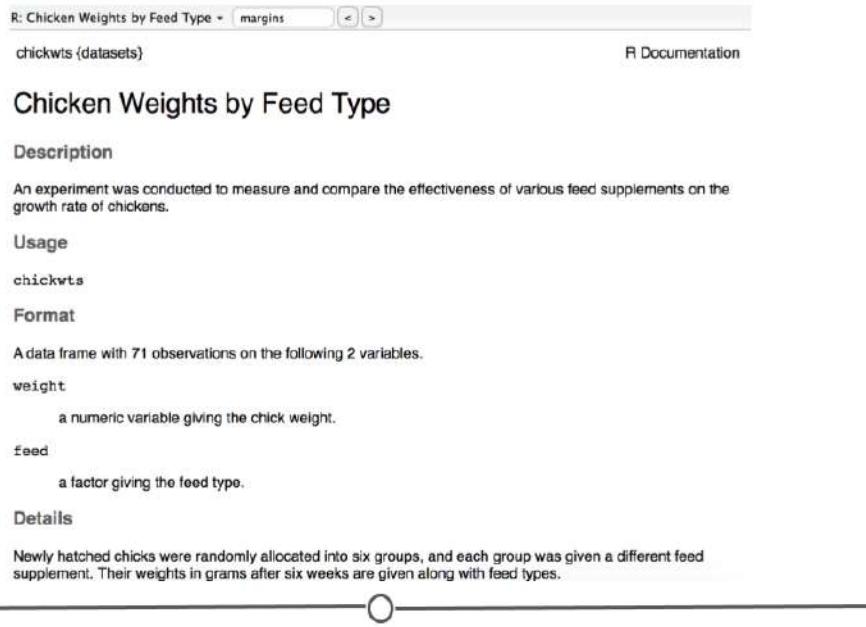
○

`fct_inorder()` assigns levels in the same order the level is seen in the data

We see now with `fct_inorder()` that even when we sort the output, it does not sort the factor alphabetically, nor does it put it in calendar order. In fact, it stays in the same order as the input, just as we specified.

Advanced Factoring

For the remainder of this lesson, we’re going to return to using a dataset that’s in R by default. We’ll use the `chickwts` dataset for exploring the remaining advanced functions. This dataset includes data from an experiment that was looking to compare the “effectiveness of various feed supplements on the growth rate of chickens.”



The screenshot shows the R documentation for the `chickwts` dataset. At the top, there's a navigation bar with tabs for "R: Chicken Weights by Feed Type" and "margins". Below the navigation bar, the title "chickwts (datasets)" is displayed, along with a link to "R Documentation". The main content area is titled "Chicken Weights by Feed Type". It contains sections for "Description", "Usage", "Format", and "Details". The "Description" section states: "An experiment was conducted to measure and compare the effectiveness of various feed supplements on the growth rate of chickens." The "Usage" section shows the command `chickwts`. The "Format" section indicates it's a data frame with 71 observations and 2 variables. The "Details" section provides more context: "Newly hatched chicks were randomly allocated into six groups, and each group was given a different feed supplement. Their weights in grams after six weeks are given along with feed types." A horizontal line with a circle at the center separates the details from the dataset name.

chickwts dataset

Re-ordering Factor Levels by Frequency: `fct_infreq()`

To re-order factor levels by frequency of the value in the dataset, you’ll want to use `fct_infreq()`. Below, we see from the output from `tabyl()` that ‘soybean’ is the most frequent feed in the dataset while ‘horsebean’ is the least frequent. Thus, when we order by frequency, we can expect these two feeds to be at opposite ends for our levels.

```

## take a look at frequency of each level
## using tabyl() from `janitor` package
tabyl(chickwts$feed)
chickwts$feed n percent
  casein 12 0.1690141
  horsebean 10 0.1408451
  linseed 12 0.1690141
  meatmeal 11 0.1549296
  soybean 14 0.1971831
  sunflower 12 0.1690141

## order levels by frequency
fct_infreq(chickwts$feed) %>% head()
[1] horsebean horsebean horsebean horsebean horsebean horsebean
Levels: soybean casein linseed sunflower meatmeal horsebean

> ## take a look at frequency of each level
> ## using tabyl() from `janitor` package
> library(janitor)
> tabyl(chickwts$feed)
chickwts$feed n percent
1    casein 12  0.169
2    horsebean 10  0.141
3    linseed 12  0.169
4    meatmeal 11  0.155
5    soybean 14  0.197
6    sunflower 12  0.169
>
> ## order levels by frequency
> fct_infreq(chickwts$feed) %>% head()
[1] horsebean horsebean horsebean horsebean horsebean horsebean
Levels: soybean casein linseed sunflower meatmeal horsebean
  Most frequent ←
  Least frequent →

```

fct_infreq orders levels based on frequency in dataset

As expected, soybean, the most frequent level, appears as the first level and horsebean, the least frequent level, appears last. The rest of the levels are sorted by frequency.

Reversing Order Levels: `fct_rev()`

If we wanted to sort the levels from least frequent to most frequent, we could just put `fct_rev()` around the code we just used to reverse the factor level order.

```
## reverse factor level order
fct_rev(fct_infreq(chickwts$feed)) %>% head()
[1] horsebean horsebean horsebean horsebean horsebean
Levels: horsebean meatmeal sunflower linseed casein soybean
```

> ## order levels by frequency
> fct_infreq(chickwts\$feed) %>% head()
[1] horsebean horsebean horsebean horsebean horsebean horsebean
Levels: soybean casein linseed sunflower meatmeal horsebean
Most frequent ← Least frequent



```
> ## reverse factor level order
> fct_rev(fct_infreq(chickwts$feed)) %>% head()
[1] horsebean horsebean horsebean horsebean horsebean horsebean
Levels: horsebean meatmeal sunflower linseed casein soybean  

Least frequent → Most frequent
```

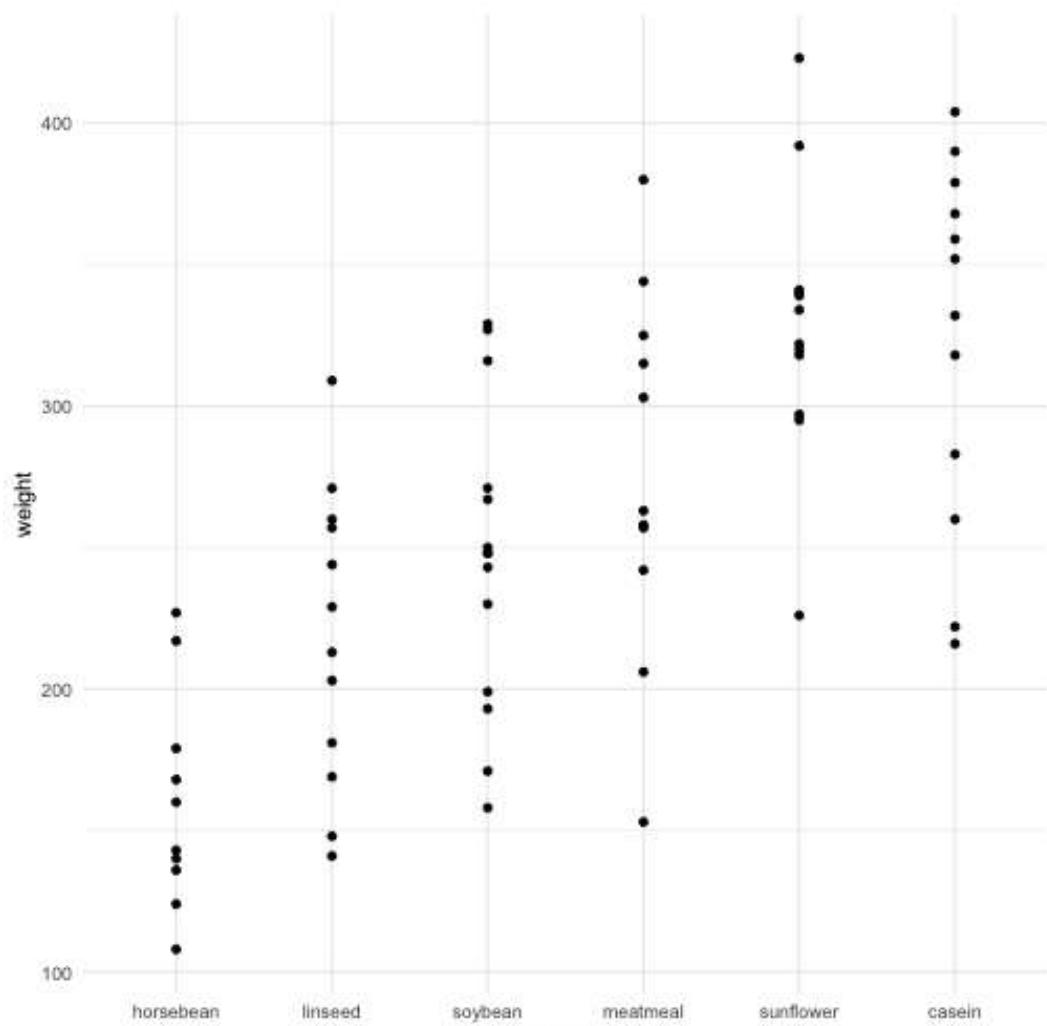
`fct_rev()` reverses the factor level order

Re-ordering Factor Levels by Another Variable: `fct_reorder()`

At times you may want to reorder levels of a factor by another variable in your dataset. This is often helpful when generating plots (which we'll get to in a future lesson!). To do this you specify the variable you want to reorder, followed by the numeric variable by which you'd like the factor to be re-leveled. Here, we see that we're re-leveling feed by the weight of the chickens. While we haven't discussed plotting yet, the best way to demonstrate how this works is by plotting the feed against the weights. We can see that the order of the factor is

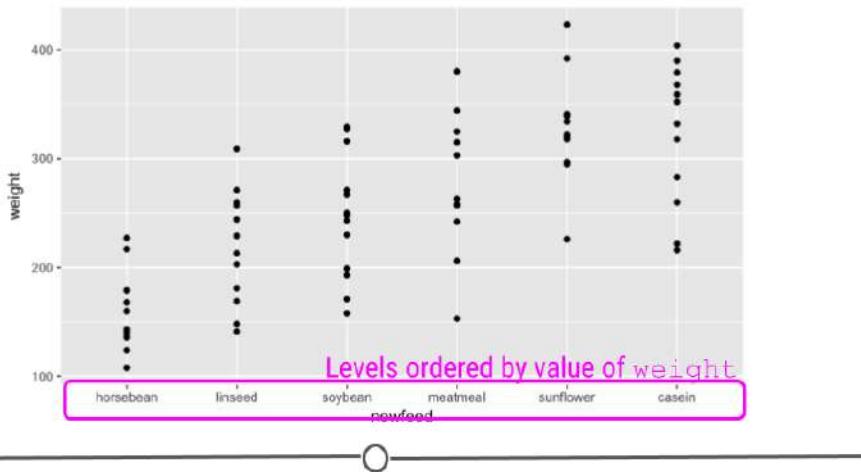
such that those chickens with the lowest median weight (horsebean) are to the left, while those with the highest median weight (casein) are to the right.

```
## order levels by a second numeric variable
chickwts %>%
  mutate(newfeed = fct_reorder(feed, weight)) %>%
  ggplot(., aes(newfeed, weight)) +
  geom_point()
```



plot of chunk unnamed-chunk-64

```
## order levels by a second numeric variable
chickwts %>%
  mutate(newfeed = fct_reorder(feed, weight)) %>%
  ggplot(., aes(newfeed, weight)) +
  geom_point()
```



`fct_reorder` allows you to re-level a factor based on a secondary numeric variable

Combining Several Levels into One: `fct_recode()`

To demonstrate how to combine several factor levels into a single level, we'll continue to use our 'chickwts' dataset. Now, I don't know much about chicken feed, and there's a good chance you know a lot more. However, let's *assume* (even if it doesn't make good sense with regards to chicken feed) you wanted to combine all the feeds with the name "bean" in it to a single category and you wanted to combine "linseed" and "sunflower" into the category "seed". This can be simply accomplished with `fct_recode`. In fact, below, you see we can rename all the levels to a simpler term (the values on the left side of the equals sign) by re-naming the original level names (the right side of the equals sign). This code will create a new column, called `feed_recode` (accomplished with `mutate()`). This new column will combine "horsebean" and "soybean feeds", grouping them both into the larger level "bean". It will similarly group "sunflower" and "linseed" into the larger level "seed." All other feed types will also be renamed. When we look at the summary of this new column by using `tabyl()`, we see that all of the feeds have been recoded, just as we specified! We now have four different feed types, rather than the original six.

```
## we can use mutate to create a new column
## and fct_recode() to:
## 1. group horsebean and soybean into a single level
## 2. rename all the other levels.
chickwts %>%
  mutate(feed_recode = fct_recode(feed,
    "seed"      = "linseed",
    "bean"       = "horsebean",
    "bean"       = "soybean",
    "meal"       = "meatmeal",
    "seed"       = "sunflower",
    "casein"     = "casein"
  )) %>%
  tabyl(feed_recode)
feed_recode n percent
  casein 12 0.1690141
  bean   24 0.3380282
  seed   24 0.3380282
  meal   11 0.1549296
```

```

> ## we can use mutate to create a new column
> ## and fct_recode() to:
> ## 1. group horsebean and soybean into a single level
> ## 2. rename all the other levels.
> chickwts %>%
+   mutate(feed_recode = fct_recode(feed,
+     "seed" = "linseed",
+     "bean" = "horsebean", Group horsebean
+     "bean" = "soybean", and soybean into
+     "meal" = "meatmeal", a single level
+     "seed" = "sunflower", called "bean"
+     "casein" = "casein"
+   )) %>%
+   tabyl(feed_recode)
#> #> feed_recode n percent
#> #> 1 casein 12  0.169
#> #> 2 bean 24  0.338
#> #> 3 seed 24  0.338
#> #> 4 meal 11  0.155

```

fct_recode() can be used to group multiple levels into a single level and/or to rename levels

Converting Numeric Levels to Factors: ifelse() + factor()

Finally, when working with factors, there are times when you want to convert a numeric variable into a factor. For example, if you were talking about a dataset with BMI for a number of individuals, you may want to categorize people based on whether or not they are underweight ($BMI < 18.5$), of a healthy weight (BMI between 18.5 and 29.9), or obese ($BMI \geq 30$). When you want to take a numeric variable and turn it into a categorical factor variable, you can accomplish this easily by using `ifelse()` statements. Within a single statement we provide R with a condition: `weight <= 200`. With this, we are stating that the condition is if a chicken's weight is less than or equal to 200 grams. Then, if that condition is true, meaning if a chicken's weight is less than or equal to 200 grams, let's assign that chicken to the category `low`. Otherwise, and this is the `else{}` part of the `ifelse()` function, assign that chicken to the category `high`. Finally, we have to let R know that `weight_recode` is a factor variable, so we call `factor()` on this new column. This way we take a numeric variable (`weight`), and turn it into a factor variable (`weight_recode`).

```
## convert numeric variable to factor
chickwts %>%
  mutate(weight_recode = ifelse(weight <= 200, "low", "high"),
        weight_recode = factor(weight_recode)) %>%
  tabyl(weight_recode)
weight_recode n percent
  high 54 0.7605634
  low 17 0.2394366

> ## convert numeric variable to factor
> chickwts %>%
+   mutate(weight_recode = ifelse(weight <= 200, "low", "high"),
+         weight_recode = factor(weight_recode)) %>%
+   tabyl(weight_recode)
weight_recode n percent
1      high 54  0.761
2      low 17  0.239
```

converting a numeric type variable to a factor

Working With Dates and Times

In earlier lessons, you were introduced to different types of objects in R, such as characters and numeric. Then we covered how to work with factors in detail. A remaining type of variable we haven't yet covered is how to work with dates and time in R.

As with strings and factors, there is a tidyverse package to help you work with dates more easily. The `lubridate` package is not part of the core tidyverse packages, so it will have to be loaded individually. This package will make working with dates and times easier. Before

working through this lesson, you'll want to be sure that `lubridate` has been installed and loaded in:

```
#install.packages('lubridate')
library(lubridate)
```

Dates and Times Basics

When working with dates and times in R, you can consider either **dates**, **times**, or **date-times**. Date-times refer to dates plus times, specifying an exact moment in time. It's always best to work with the simplest possible object for your needs. So, if you don't need to refer to date-times specifically, it's best to work with dates.

Creating Dates and Date-Time Objects

To get objects into dates and date-times that can be more easily worked with in R, you'll want to get comfortable with a number of functions from the `lubridate` package. Below we'll discuss how to create date and date-time objects from (1) strings and (2) individual parts.

From strings

Date information is often provided as a string. The functions within the `lubridate` package can effectively handle this information. To use them to generate date objects, you can call a function using `y`, `m`, and `d` in the order in which the year (`y`), month (`m`), and date (`d`) appear in your data. The code below produces identical output for the date September 29th, 1988, despite the three distinct input formats. This uniform output makes working with dates much easier in R.

```
# year-month-date  
ymd("1988-09-29")  
[1] "1988-09-29"  
  
#month-day-year  
mdy("September 29th, 1988")  
[1] "1988-09-29"  
  
#day-month-year  
dmy("29-Sep-1988")  
[1] "1988-09-29"
```

date
objects

```
> ymd("1988-09-29")  
[1] "1988-09-29"  
>  
> mdy("September 29th, 1988")  
[1] "1988-09-29"  
>  
> dmy("29-Sep-1988")  
[1] "1988-09-29"
```

date-time
object

```
> ymd_hms("1988-09-29 20:11:59")  
[1] "1988-09-29 20:11:59 UTC"
```

creating date and date-time objects

However, this has only covered working with date objects. To work with date-time objects, you have to further include hour (h), minute(m), and second (s) into the function. For example, in the code below, you can see that the output contains time information in addition to the date information generated in the functions above:

```
ymd_hms("1988-09-29 20:11:59")
[1] "1988-09-29 20:11:59 UTC"
```

From individual parts

If you have a dataset where month, date, year, and/or time information are included in separate columns, the functions within `lubridate` can take this separate information and create a date or date-time object. To work through examples using the functions `make_date()` and `make_timedate()`, we'll use a dataset called `nycflights13`. As this dataset is *not* included with the R by default, you'll have to install and load it in directly:

```
#install.packages('nycflights13')
library(nycflights13)
```

Loading this package makes a data frame called `flights`, which includes “on-time data for all flights that departed NYC in 2013,” available. We will work with this dataset to demonstrate how to create a date and date-time object from a dataset where the information is spread across multiple columns.

First, to create a new column, as we've done throughout the lessons in this course, we will use `mutate()`. To create a date object, we'll use the function `make_date()`. We just then need to supply the names of the columns containing the year, month, and day information to this function.

```
## make_date() creates a date object
## from information in separate columns
flights %>%
  select(year, month, day) %>%
  mutate(departure = make_date(year, month, day))
# A tibble: 336,776 × 4
  year month   day departure
  <int> <int> <int> <date>
1 2013     1     1 2013-01-01
2 2013     1     1 2013-01-01
3 2013     1     1 2013-01-01
4 2013     1     1 2013-01-01
5 2013     1     1 2013-01-01
6 2013     1     1 2013-01-01
```

```

7 2013     1      1 2013-01-01
8 2013     1      1 2013-01-01
9 2013     1      1 2013-01-01
10 2013    1      1 2013-01-01
# ... with 336,766 more rows

> ## make_date() creates a date object
> ## from information in separate columns
> flights %>%
+   select(year, month, day) %>%
+   mutate(departure = make_date(year, month, day))
# A tibble: 336,776 x 4
  year month day   departure
  <int> <int> <int> <date>
1 2013     1     1 2013-01-01
2 2013     1     1 2013-01-01
3 2013     1     1 2013-01-01
4 2013     1     1 2013-01-01
5 2013     1     1 2013-01-01
6 2013     1     1 2013-01-01
7 2013     1     1 2013-01-01
8 2013     1     1 2013-01-01
9 2013     1     1 2013-01-01
10 2013    1     1 2013-01-01
# ... with 336,766 more rows

```

○

`mutate()` and `make_date()` create a new column – `departure` – with a date object

A similar procedure is used to create a date-time object; however, this requires the function `make_datetime()` and requires columns with information about time be specified. Below, `hour` and `minute` are included to the function's input.

```

## make_datetime() creates a date-time object
## from information in separate columns
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
# A tibble: 336,776 x 6
  year month day hour minute   departure
  <int> <int> <int> <dbl> <dbl> <dttm>
1 2013     1     1     5      15 2013-01-01 05:15:00

```

```

2 2013 1 1 5 29 2013-01-01 05:29:00
3 2013 1 1 5 40 2013-01-01 05:40:00
4 2013 1 1 5 45 2013-01-01 05:45:00
5 2013 1 1 6 0 2013-01-01 06:00:00
6 2013 1 1 5 58 2013-01-01 05:58:00
7 2013 1 1 6 0 2013-01-01 06:00:00
8 2013 1 1 6 0 2013-01-01 06:00:00
9 2013 1 1 6 0 2013-01-01 06:00:00
10 2013 1 1 6 0 2013-01-01 06:00:00
# ... with 336,766 more rows

```

```

> ## make_datetime() creates a date-time object
> ## from information in separate columns
> flights %>%
+   select(year, month, day, hour, minute) %>%
+   mutate(departure = make_datetime(year, month, day, hour, minute))
# A tibble: 336,776 x 6
  year month day hour minute departure
  <int> <int> <int> <dbl> <dbl> <dttm>
1 2013 1 1 5.00 15.0 2013-01-01 05:15:00
2 2013 1 1 5.00 29.0 2013-01-01 05:29:00
3 2013 1 1 5.00 40.0 2013-01-01 05:40:00
4 2013 1 1 5.00 45.0 2013-01-01 05:45:00
5 2013 1 1 6.00 0 2013-01-01 06:00:00
6 2013 1 1 5.00 58.0 2013-01-01 05:58:00
7 2013 1 1 6.00 0 2013-01-01 06:00:00
8 2013 1 1 6.00 0 2013-01-01 06:00:00
9 2013 1 1 6.00 0 2013-01-01 06:00:00
10 2013 1 1 6.00 0 2013-01-01 06:00:00
# ... with 336,766 more rows

```

	year	month	day	hour	minute	departure
	<int>	<int>	<int>	<dbl>	<dbl>	<dttm>
1	2013	1	1	5.00	15.0	2013-01-01 05:15:00
2	2013	1	1	5.00	29.0	2013-01-01 05:29:00
3	2013	1	1	5.00	40.0	2013-01-01 05:40:00
4	2013	1	1	5.00	45.0	2013-01-01 05:45:00
5	2013	1	1	6.00	0	2013-01-01 06:00:00
6	2013	1	1	5.00	58.0	2013-01-01 05:58:00
7	2013	1	1	6.00	0	2013-01-01 06:00:00
8	2013	1	1	6.00	0	2013-01-01 06:00:00
9	2013	1	1	6.00	0	2013-01-01 06:00:00
10	2013	1	1	6.00	0	2013-01-01 06:00:00

date-time
object

mutate() and make_datetime() create a new column – departure – with a date-time object

Working with Dates

The reason we've dedicated an entire lesson to working with dates and have shown you how to create date and date-time objects in this lesson is because you often want to plot data over time or calculate how long something has taken. Being able to accomplish these tasks is an important job for a data scientist. So, now that you know how to create date and date-time objects, we'll work through a few examples of how to work with these objects.

Getting components of dates

Often you're most interested in grouping your data by year, or just looking at monthly or weekly trends. To accomplish this, you have to be able to extract just a component of your date object. You can do this with the functions: `year()`, `month()`, `mday()`, `wday()`, `hour()`, `minute()` and `second()`. Each will extract the specified piece of information from the date or date-time object.

```
mydate <- ymd("1988-09-29")

## extract year information
year(mydate)

## extract day of the month
mday(mydate)

## extract weekday information
wday(mydate)

## label with actual day of the week
wday(mydate, label = TRUE)
```

```
> mydate <- ymd("1988-09-29")
>
> ## extract year information
> year(mydate)
[1] 1988
>
> ## extract day of the month
> mday(mydate)
[1] 29
>
> ## extract weekday information
> wday(mydate)
[1] 5
>
> ## label with actual day of the week
> wday(mydate, label = TRUE)
[1] Thu
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

lubridate has specific functions to extract components from date and date-time objects

Time Spans

In addition to being able to look at trends by month or year, which requires being able to extract that component from a date or date-time object, it's also important to be able to operate over dates. If I give you a date of birth and ask you how old that person is today, you'll want to be able to calculate that. This is possible when working with date objects. By subtracting this birth date from today's date, you'll learn how many days old this person is. By specifying this object using `as.duration()`, you'll be able to extract how old this person is in years.

```
## how old is someone born on Sept 29, 1988
mydate <- ymd("1988-09-29")

## subtract birthday from todays date
age <- today() - mydate
age
Time difference of 12027 days

## a duration object can get this information in years
as.duration(age)
[1] "1039132800s (~32.93 years)"
```

```
> ## how old is someone born on Sept 29, 1988
> mydate <- ymd("1988-09-29")
>
> ## subtract birthday from todays date
> age <- today() - mydate
> age
Time difference of 10808 days
>
> ## a duration object can get this information in years
> as.duration(age)
[1] "933811200s (~29.59 years)"
```

○

dates and date-times can be operated upon

Using addition, subtraction, multiplication, and division is possible with date objects, and accurately takes into account things like leap years and different number of days each month. This capability and the additional functions that exist within lubridate can be enormously helpful when working with dates and date-time objects.

Working With Strings

You're likely familiar with strings generally; however, to review briefly here:

A string is a sequence of characters, letters, numbers or symbols.

So within R, you could create a string using this syntax. Note that the string begins and ends with quotation marks:

```
stringA <- "This sentence is a string."
```

Multiple strings can be stored within vectors. So, if you have multiple vectors that you want to store in a single object, you could do so by using `c()` around the strings you want to store and commas to separate each individual string:

```
objectA <- c( "This sentence is a string.", "Short String", "Third string" )
```

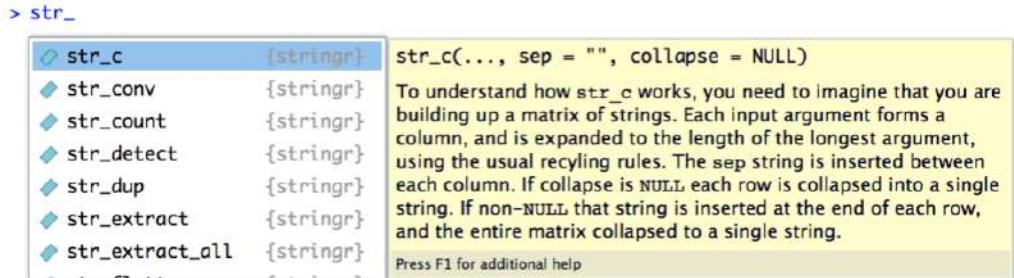
stringr

`stringr` is a core tidyverse package specifically designed to help make your life easier when working with strings. Similar to what we saw with `forcats` functions starting with `fct_`, all functions within this package start with `str_`, as you'll see below. There are *many* helpful functions within the `stringr` package. We'll only review the basics here, but if you're looking to accomplish something with a string and aren't sure how to approach it, the `stringr` package is a good first place to look.

The best way to work through this lesson is to copy and paste every line of code into your RStudio window and see if the output makes sense to you. Working with strings and regular expressions is best learned by practice.

Available functions

As we'll only cover a few of the functions within `stringr` in this lesson, it's important to remember that if you start typing "`str_`" within RStudio, a list of the many options will show up.



str_ image

String Basics

When working with strings, some of the most frequent tasks you'll need to complete are:

- determine the length of a string
- combine strings together
- subset strings

String length

Returning to our object with three strings from earlier in the lesson, we can determine the length of each string in the vector.

```
objectA <- c( "This sentence is a string.", "Short String", "Third string" )
```

```
str_length(objectA)
[1] 26 12 12
```

```
> objectA <- c( "This sentence is a string.", "Short String", "Third string" )
>
> str_length(objectA)
[1] 26 12 12
```

str_length() output

Here we see that the first string has a length of 26. If you were to go back and count the characters in the first string, you would see that this 26 includes each letter, space, and period in that string. The length of a string does not just could the letters in its length. The length includes every character. The second and third strings each have length 12.

Combining strings: str_c()

If you were interested in combining strings, you'd want to use str_c.

```
str_c( "Good", "Morning")
[1] "GoodMorning"
```

```
> str_c("Good", "Morning")
[1] "GoodMorning"
>
> str_c("Good", "Morning", sep=" ")
[1] "Good Morning"
```



str_c()

However, the output from this doesn't look quite right. You may want a space between these two words when you combine the two strings. That can be controlled with the `sep` argument.

```
str_c("Good", "Morning", sep=" ")
[1] "Good Morning"
```

Subsetting strings: str_sub()

Often, it's important to get part of a string out. To do this, you'll want to subset the string using the `str_sub()` function. For example, if you wanted only the first three characters in the string below, you would specify that within `str_sub()`.

```
object <- c("Good", "Morning")
str_sub(object, 1, 3)
[1] "Goo" "Mor"
```

```
> object <- c("Good", "Morning")
>
> str_sub(object, 1, 3)
[1] "Goo" "Mor"
```



str_sub() output

You can also use negative numbers to count from the end of the string. For example, below we see code that returns the last three positions in the string.

```
object <- c("Good", "Morning")
str_sub(object, -3, -1)
[1] "ood" "ing"
```

```
> object <- c("Good", "Morning")
>
> str_sub(object, -3, -1)
[1] "ood" "ing"
```

○

str_sub() output counting from end of string

String sorting: str_sort()

Finally, if you wanted to sort a string alphabetically, str_sort() can help you accomplish that.

```
names <- c("Keisha", "Mohammed", "Jane")

str_sort(names)
[1] "Jane"      "Keisha"     "Mohammed"
```

```
> names <- c("Keisha", "Mohammed", "Jane")
>
> str_sort(names)
[1] "Jane"      "Keisha"     "Mohammed"
```

○

str_sort() output sorts strings

Regular Expressions

Above we discussed the basics of working with strings within `stringr`. However, working with strings becomes infinitely easier with an understanding of regular expressions. Regular expressions (regexps) are used to **describe patterns within strings**. They can take a little while to get the hang of but become very helpful once you do. With regexps, instead of specifying that you want to extract the first three letters of a string (as we did above), you could more generally specify that you wanted to extract all strings that start with a specific letter or that contain a specific word somewhere in the string using regexps. We'll explore the basics of regexps here.

The use them in `stringr`, the general format is `function(string , pattern = regexp)`, which you'll see used in practice below.

We'll cover a number of helpful `stringr` functions:

- `str_view()` - View the first occurrence in a string that matches the regex
- `str_view_all()` - View all occurrences in a string that match the regex
- `str_count()` - count the number of times a regex matches within a string

- `str_detect()` - determine if regex is found within string
- `str_subset()` - return subset of strings that match the regex
- `str_extract()` - return portion of each string that matches the regex
- `str_replace()` - replace portion of string that matches the regex with something else

Anchors

If interested in finding a pattern at the beginning (^) or end (\$) of a string, you can specify that using a regexp. For example, if you wanted to only look at names that started with the letter "M", you would specify that using a regexp. The pattern you would include would be "`^M`" to identify all strings that start with the letter M. To specify those strings that end with a capital M, you would specify the pattern "`$M`".

Show matches: `str_view()`

To get comfortable with using regexps with strings, `str_view()` can be very helpful. The output from `str_view()` highlights what portion of your string match the pattern specified in your regexp with a gray box. For example, to we'll start using anchors and `str_view()` below:

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
  
## identify strings that start with "M"  
str_view(names, "^M")
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
## identify strings that start with "M"  
str_view(names, "^M")
```



`str_view()` identifies names that start with M

In this first example we see in the Viewer Panel that `str_view()` has identified the names that start with the letter “M”.

However, if you try to match strings that end with the letter “M”, no match is found.

```
## identify strings that end with "M"  
str_view(names, "M$")
```

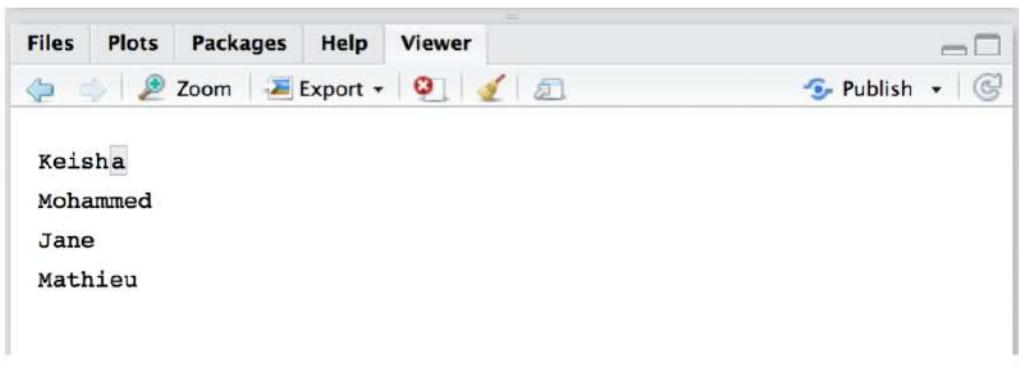
```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
## identify strings that end with "M"  
str_view(names, "M$")
```



`str_view()` does not identify any names that end with M
To identify names by that end with the letter “a”, you would use the following.

```
## identify strings that end with "a"  
str_view(names, "a$")
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
## identify strings that end with "a"  
str_view(names, "a$")
```



`str_view()` identifies names that end with a

Note, however, that regexps are case sensitive. To match patterns, you have to consider that “A” and “a” are different characters.

```
## identify strings that end with "A"  
str_view(names, "A$")
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
## identify strings that end with "A"  
str_view(names, "A$")
```



`str_view()` does not identify any names that end with A

Count matches: `str_count()`

To count the number of matches within your strings, you would use `str_count()`. Below, using the names vector we've been using, we see that `str_count()` produces a 1 for those names that start with "M" and a 0 otherwise.

```
## identify strings that start with "M"  
## return count of the number of times string matches pattern  
str_count(names, "^M")
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
  
## identify strings that start with "M"  
## return count of the number of times  
string matches pattern  
  
> str_count(names, "^M")  
[1] 0 1 0 1
```



`str_count()` strings that start with “M”

However, if we instead wanted a count of the numbers of lowercase “m”s, we could still use `str_count()` to accomplish that. Notice below we’ve removed the specification to just look at the beginning of the string. Here, we’re looking for lowercase m’s anywhere in the string and counting them:

```
## identify strings that have a lowercase "m"  
## return count of the number of times string matches pattern  
str_count(names, "m")  
[1] 0 2 0
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
  
## identify strings that have a lowercase "m"  
## return count of the number of times string  
matches pattern  
  
> str_count(names, "m")  
[1] 0 2 0 0
```



str_count() strings that have an m in them

Detect matches: str_detect()

Instead of returning a count, at times you're just interested in knowing which strings match the pattern you're searching for. In these cases you'll want to use str_detect(). This function simply returns a TRUE if the string matches the pattern specified and FALSE otherwise.

```
## identify strings that start with "M"  
## return TRUE if they do; FALSE otherwise  
str_detect(names, "^M")  
[1] FALSE TRUE FALSE
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
  
## identify strings that start with "M"  
## return TRUE if they do; FALSE  
otherwise  
  
> str_detect(names, "^M")  
[1] FALSE  TRUE FALSE  TRUE
```



`str_detect()` returns TRUE for strings that match the specified pattern; FALSE otherwise

Subset matches: `str_subset()`

To return the actual string that matches the specified pattern, rather than a TRUE/FALSE, you'll look to `str_subset()`. This function pulls out those strings that match the specified pattern. For example, to obtain the subset of names whose values start with the capital letter "M", you would use the following:

```
## identify strings that start with "M"  
## return whole string  
str_subset(names, "M")  
[1] "Mohammed"
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
  
## identify strings that start with "M"  
## return whole string  
  
> str_subset(names, "^M")  
[1] "Mohammed" "Mathieu"
```

○

`str_subset()` returns the strings that match the pattern specified

Extract matches: `str_extract()`

To extract only the portions of the string that match the specified pattern, you would use `str_extract()`. This function returns the pattern specified for strings where it is found and NA otherwise. For example, by searching for names that start with M, below, we see that the second and fourth strings in our vector return the pattern specified ("M") and that the first and third strings in the vector return NA, as they do not start with a capital "M".

```
## return "M" from strings that start with "M"  
## otherwise, return NA  
str_extract(names, "^M")  
[1] NA "M" NA
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")
```

```
## return "M" from strings with "M" in it  
## otherwise, return NA
```

```
> str_extract(names, "^M")
```

```
[1] NA    "M"   NA    "M"
```



`str_extract()` returns the portions of the strings that match the pattern specified

Replace matches: `str_replace()`

The final basic function from `stringr` that we'll discuss is `str_replace()`. This function identifies a regex and replaces each occurrence with whatever replacement the user specifies. For example, below we search for strings that start with the capital letter "M" and replace each of them with a question mark. All strings that do *not* match the regex are returned unchanged.

```
## replace capital M with a question mark  
str_replace(names, "M", "?")  
[1] "Keisha"   "?ohammed" "Jane"
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")
```

```
## replace capital M with a question mark
```

```
> str_replace(names, "^M", "?")
[1] "Keisha"    "?ohammed"  "Jane"       "?athieu"
```

○

`str_replace()` replaces regex with specified characters

Common regular expressions

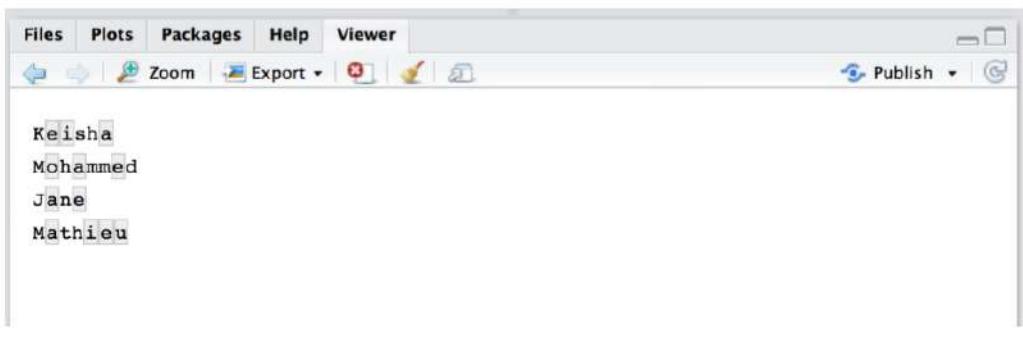
Above we discuss two common patterns searched for using regular expressions: starts with (^) and ends with (\$). However, there are a number of additional common ways to match patterns. They are listed here, and we'll discuss each one in slightly more detail below.

Searching for characters

To search for a set of characters, you place these characters within brackets. Below, this will identify anywhere in the strings where you have a lowercase vowel. Note, that we're now using `str_view_all()` to identify all occurrences of these characters, rather than `str_view()`, which only identifies the first occurrence in each string.

```
## identify all lowercase vowels
str_view_all(names, "[aeiou]")
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
## identify all lowercase vowels  
str_view_all(names, "[aeiou]")
```



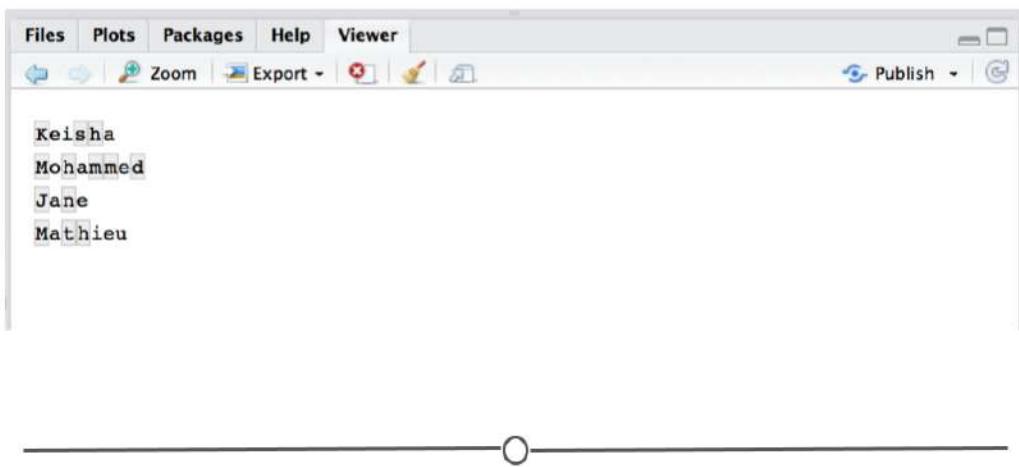
brackets specify which characters to search for

Searching for anything *other* than a set of characters

By adding a caret (^) before the vowels within the brackets, this regular expression specifies that you are searching for any character that is not a lowercase vowel within your strings.

```
## identify anything that's NOT a lowercase vowel  
str_view_all(names, "[^aeiou]")
```

```
names <- c("Keisha", "Mohammed", "Jane", "Mathieu")  
## identify anything that's NOT a lowercase vowel  
str_view_all(names, "[^aeiou]")
```



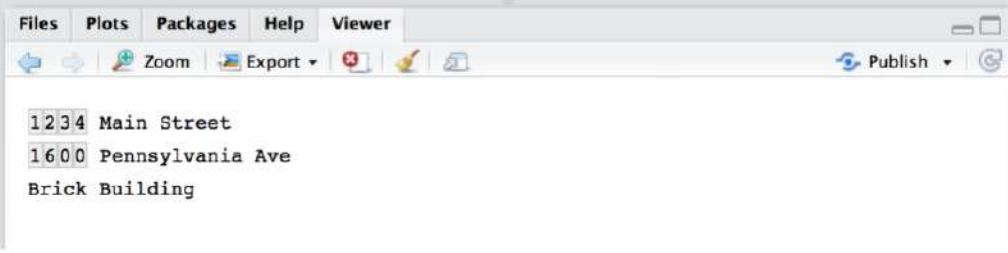
brackets with a caret first specify which characters NOT to search for

Search for digits

To search for digits (numeric variable between 0 and 9) in a string you use “\d”; however, backslashes are protected characters in R. This means that you have to escape this character first with an additional backslash (\), to let R know that you want to search for the regular expression “\\d”.

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify anything that's a digit  
str_view_all(addresses, "\\d")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify anything that's a digit  
str_view_all(addresses, "\\\d")
```



The screenshot shows the RStudio interface with the 'Viewer' tab selected in the top menu bar. Below the menu is a toolbar with icons for file operations like Open, Save, and Print, as well as Publish and Help. The main viewer area displays three lines of address data, each with digits highlighted in red:

```
1234 Main Street  
1600 Pennsylvania Ave  
Brick Building
```

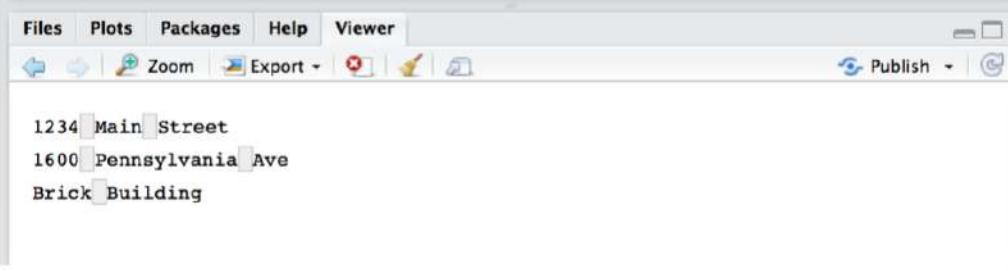
\d searches for digits

Search for whitespace

Identifying whitespace in R identifies any spaces, tabs or newlines. Note that again we have to escape the “\s” with a backslash for R to recognize the regular expression.

```
## identify any whitespace  
str_view_all(addresses, "\\\s")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify any whitespace  
str_view_all(addresses, "\\\s")
```



The screenshot shows the RStudio interface with the 'Viewer' tab selected. The code `str_view_all(addresses, "\\s")` is run, and its output is displayed. The output consists of three lines of text, each with a matching bracket on the left and a cursor on the right, indicating the position of the whitespace character identified by the regular expression. The lines are:

```
1234 Main Street  
1600 Pennsylvania Ave  
Brick Building
```

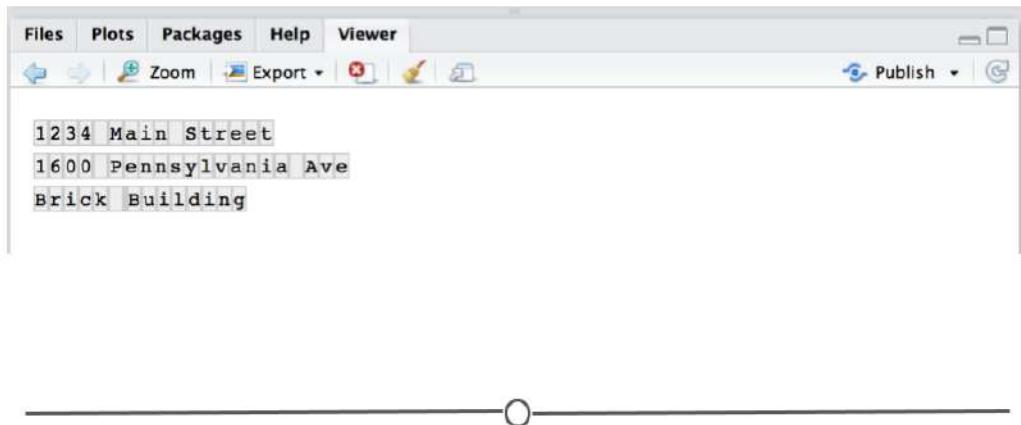
\s searches for whitespace

Identify any character (except newline)

To identify any character except for a newline you'll use ". ". Notice in our addresses example that there are no newlines, so this pattern will match with the entire string.

```
## identify any character  
str_view_all(addresses, ".")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify any character  
str_view_all(addresses, ".")
```



. searches for any character

Repetition within regular expressions

Searches for regular expressions allow you to specify how many times a pattern should be found within the string. To do so, you use the following:

- ?: 0 or 1
- +: 1 or more
- *: 0 or more
- {n}: exactly n times
- {n,}: n or more times
- {n,m}: between n and m times

Examples of repetition within regular expressions

Using the definitions above, we can see that the following code will identify patterns within the addresses vector where n shows up one or more times in a string.

```
## identify any time n shows up one or more times
str_view_all(addresses, "n+")

addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")

## identify any time n shows up one or more times
str_view_all(addresses, "n+")
```



+ specifies to match the pattern one or more times

While the difference is slight in the output here, we're identifying portions of the string where n shows up exactly once. So, instead of the 'nn' in Pennsylvania matching together, the code here splits these up, due to the fact that we're specifying the pattern match 'n' exactly one time:

```
## identify any time n shows up
str_view_all(addresses, "n{1}")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify any time n shows up  
str_view_all(addresses, "n{1}")
```

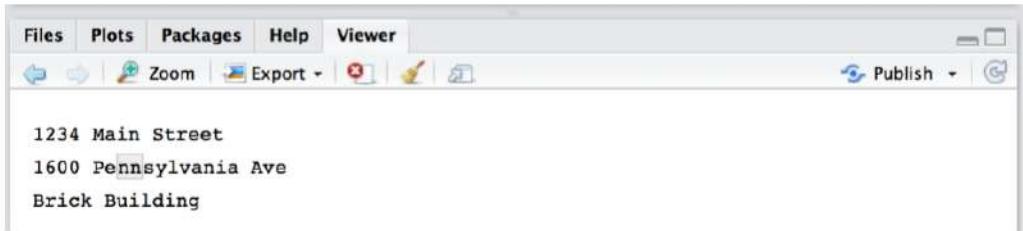


{#} looks to match the pattern exactly the number of times within the curly braces

If you only wanted to match strings where n showed up twice in a row, you could specify that in this way:

```
## identify any time n shows up exactly two times in a row  
str_view_all(addresses, "n{2}")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify any time n shows up exactly two times in a row  
str_view_all(addresses, "n{2}")
```



{2} specifies that the pattern must be found exactly twice

This could similarly be achieved by specifying to search for the pattern 'nn' one or more times (+):

```
## identify any time 'nn' shows up one or more times  
str_view_all(addresses, "nn+")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")  
## identify any time 'nn' shows up one or more times  
str_view_all(addresses, "nn+")
```



`nn+` searches for double n one or more times in a string

You can also specify a range of the number of times to search for a pattern within your string. Below, we see that if we specify n be searched for at least two and at most 3 times, the pattern matches within our string. However, if we increase that to between three and four times, no pattern matching occurs, as there are never three or four n's in a row in our strings.

```
## identify any time n shows up two or three times  
str_view_all(addresses, "n{2,3}")  
  
## identify any time n shows up three or four times  
str_view_all(addresses, "n{3,4}")
```

```
addresses <- c("1234 Main Street", "1600 Pennsylvania Ave", "Brick Building")
## identify any time n shows up two or three times
str_view_all(addresses, 'n{2,3}')
```

The screenshot shows the RStudio interface with the 'Viewer' tab selected. The code `str_view_all(addresses, 'n{2,3}')` is run, and the output is displayed as follows:

```
1234 Main Street
1600 Pennsylvania Ave
Brick Building
```

```
## identify any time n shows up three or four times
str_view_all(addresses, 'n{3,4}')
```

The screenshot shows the RStudio interface with the 'Viewer' tab selected. The code `str_view_all(addresses, 'n{3,4}')` is run, and the output is displayed as follows:

```
1234 Main Street
1600 Pennsylvania Ave
Brick Building
```

{n,m} looks to pattern match between n and m times

glue

Beyond using `stringr` to work with strings, there's an additional helpful package called `glue`. According to the `glue` website:

Glue offers interpreted string literals that are small, fast, and dependency-free. Glue does this by embedding R expressions in curly braces which are then evaluated and inserted into the argument string.

To get started with this package, it will have to be installed and loaded in, as it is not a core tidyverse package.

```
# install.packages("glue")
library(glue)
```

So, if you want to pass an R variable directly into a string, that becomes simpler with `glue`.

For example:

```
# use glue to interpret string literal
topic <- 'tidyverse'
glue('My favorite thing to learn about is the {topic}!')
My favorite thing to learn about is the tidyverse!
```

Note that the code above interprets the variable `topic` within the string specified in the `glue()` function. The variable is specified within curly braces: {}.

This becomes particularly helpful when combining information within a data frame.

For example, if we return to the `msleep` dataset with information about mammalian sleep, we could use `mutate()` to add a column summarizing the name of the animal, how many minutes the animal spends asleep and how many awake. Note that these columns are currently in hours, so we're going to convert that to minutes within the `glue` statement here:

```
# add a description column using glue
msleep %>%
  mutate(description = glue("The {name} typically sleeps for {sleep_total * 60}\nminutes and is awake for {awake * 60} minutes each day.")) %>%
  select(name, sleep_total, awake, description)
# A tibble: 83 × 4
  name           sleep_total   awake description
  <chr>          <dbl>     <dbl> <glue>
1 Cheetah        12.1      11.9 The Cheetah typically sleeps fo...
2 Owl monkey     17.0       7.0  The Owl monkey typically sleeps...
3 Mountain beaver 14.4      9.6  The Mountain beaver typically s...
4 Greater short-tailed shrew 14.9      9.1  The Greater short-tailed shrew ...
5 Cow             4.0       20.0  The Cow typically sleeps for 24...
6 Three-toed sloth 14.4      9.6  The Three-toed sloth typically ...
7 Northern fur seal 8.7      15.3  The Northern fur seal typically...
8 Vesper mouse    7.0       17.0  The Vesper mouse typically slee...
9 Dog             10.1     13.9  The Dog typically sleeps for 60...
10 Roe deer       3.0       21.0  The Roe deer typically sleeps f...
# ... with 73 more rows
```

Note that we've selected the four columns used in this example with `select()` to focus on the output created using `glue()` in the `description` column. For each observation the information in `description` utilizes the appropriate information for that observation and the columns specified within the curly braces within the `glue()` function.

Working With Text

Beyond working with single strings and string literals, sometimes the information you're analyzing is a whole body of text. This could be a speech, a novel, an article, or any other written document. In text analysis, the document(s) you've set out to analyze are referred to as a **corpus**. Linguists frequently analyze such types of data and doing so within R in a tidy data format has become simpler thanks to the `tidytext` package and the package-accompanying book *Text Mining with R*.

To get started, the package must be installed and loaded in:

```
# install.packages("tidytext")
library(tidytext)
```

Tidy Text Format

If we're thinking about all the text in a novel, it's pretty clear that it is not in a format that is easy to analyze computationally. To analyze the text in the novel computationally and say, determine what words are used most frequently, or what topics are discussed, we need to convert the text in the novel into a format that a computer can interpret. And, as with all types of data discussed in these courses, we want this to be a tidy format where (1) each observation is a row (2) each variable is a column, and (3) each observational unit is a table. So, how do we take text from a novel and store the information in a tidy format?

The tidy text format requires that the data frame will store one **token** per row. This requires knowing that a **token** is a meaningful unit of text. How you define that unit is up to you, the analyst and is driven by the question you're asking. If you're looking to identify the words used most frequently in this analysis, the unit of your token would be individual words. You would then utilize your computer to generate a data frame with each row containing data about a single word. However, your token could be two words (a **bigram**), a sentence, or a paragraph. Whatever you decide is meaningful for your analysis will be the unit for your token. Each row will contain a separate token.

Tokenization

After determining what level of information you're most interested in, you need a way to go from a wall of text (say, all the text in a novel) to a data frame of tokens (say, individual words). To do this, the `unnest_tokens()` function is incredibly useful.

We'll use a bear bones example to demonstrate how it works. Below is text from the Shel Silverstein poem "Carrots" stored as a character vector:

```
carrots <- c("They say that carrots are good for your eyes",
           "They swear that they improve your sight",
           "But I'm seein' worse than I did last night -",
           "You think maybe I ain't usin' em right?")  
  
carrots  
[1] "They say that carrots are good for your eyes"  
[2] "They swear that they improve your sight"  
[3] "But I'm seein' worse than I did last night -"  
[4] "You think maybe I ain't usin' em right?"
```

For analysis, we'd need to get this into a tidy data format. So, first things first, let's get it into a data frame:

```
library(tibble)  
text_df <- tibble(line = 1:4, text = carrots)  
  
text_df  
# A tibble: 4 × 2  
  line   text  
  <int> <chr>  
1     1 They say that carrots are good for your eyes  
2     2 They swear that they improve your sight  
3     3 But I'm seein' worse than I did last night -  
4     4 You think maybe I ain't usin' em right?
```

At this point we have a tibble with each line of the poem in a separate row. Now, we want to convert this using `unnest_tokens()` so that each row contains a single token, where, for this example, our token will be an individual word. This process is known as *tokenization*.

```
text_df %>%  
  unnest_tokens(word, text)  
# A tibble: 33 × 2  
  line word  
  <int> <chr>  
1     1 they  
2     1 say  
3     1 that  
4     1 carrots  
5     1 are  
6     1 good  
7     1 for  
8     1 your  
9     1 eyes  
10    2 they  
# ... with 23 more rows
```

Notice that the two arguments to the `unnest_tokens()` function. The first (`word` in our example) is the name of the token column in the output. The second (`text` in our example) is the name of the column in the input data frame (`text_df`) that should be used for tokenization.

In the output we see that there is a single word (token) in each row, so our data are now in a tidy format, which makes further analysis simpler.

Finally, note that, by default `unnest_tokens()` strips punctuation and converts the tokens to lowercase.

Sentiment Analysis

Often, once you've tokenized your dataset, there is an analysis you want to do - a question you want to answer. Sometimes, this involves wanting to measure the sentiment of a piece by looking at the emotional content of the words in that piece.

To do this, the analyst must have access to or create a *lexicon*, a dictionary with the sentiment of common words. There are three single word-based lexicons available within the `tidytext` package: `afinn`, `bing`, `loughran` and `nrc`. Each differs in how they categorize sentiment, and to get a sense of how words are categorized in any of these lexicon, you can use the `get_sentiments()` function.

However, this requires an additional package: `textdata`. Be sure this has been installed before using the `get_sentiments()` function.

```
library(textdata)
# be sure textdata is installed
#install.packages("textdata", repos = 'http://cran.us.r-project.org')

# see information stored in NRC lexicon
get_sentiments('nrc')
# A tibble: 13,901 × 2
  word      sentiment
  <chr>    <chr>
1 abacus   trust
2 abandon   fear
3 abandon   negative
4 abandon   sadness
5 abandoned anger
6 abandoned fear
7 abandoned negative
8 abandoned sadness
9 abandonment anger
10 abandonment fear
# ... with 13,891 more rows
```

Note: The first time you use this function R will prompt you to verify that you want to download the lexicon.

In the output you'll see words in the first column and the sentiment attached to each word in the `sentiment` column. Notice that the same word can have multiple sentiments attached to it. All told, there are more than 13,000 word-sentiment pairs in this lexicon.

Let's quantify the sentiment in the "Carrots" poem from above:

```
text_df %>%  
  unnest_tokens(word, text) %>%  
  inner_join(get_sentiments('nrc'))  
Joining, by = "word"  
# A tibble: 14 × 3  
  line word    sentiment  
  <int> <chr>   <chr>  
1     1 good    anticipation  
2     1 good    joy  
3     1 good    positive  
4     1 good    surprise  
5     1 good    trust  
6     2 swear   positive  
7     2 swear   trust  
8     2 improve anticipation  
9     2 improve joy  
10    2 improve positive  
11    2 improve trust  
12    3 worse   fear  
13    3 worse   negative  
14    3 worse   sadness
```

Notice that the sentiments applied to each word are dependent upon the sentiments defined within the lexicon. Words that are missing or that are used differently than anticipated by those who generated the lexicon *could* be misclassified. Additionally, since we're using single word tokens, qualifiers are removed from context. So in the carrots poem, the word good in “are good for your eyes” would be given the same sentiment as good if the phrase were “*not* good for your eyes.” Thus, a lot context and nuance is lost in this approach. It’s always important to consider the limitations of your analytical approach!

Above we found the sentiments for each token, but let’s summarize that by counting the number of times each sentiment appears.

```
text_df %>%
  unnest_tokens(word, text) %>%
  inner_join(get_sentiments('nrc')) %>%
  count(sentiment, sort = TRUE)
Joining, by = "word"
# A tibble: 8 × 2
  sentiment     n
  <chr>      <int>
1 positive      3
2 trust         3
3 anticipation  2
4 joy           2
5 fear          1
6 negative      1
7 sadness        1
8 surprise       1
```

As we're analyzing a short poem, we see that only a few sentiments show up multiple times; however, using sentiment analysis on this poem suggests that the poem is generally positive, including words that convey trust, anticipation, and joy.

Analyzing a four line poem, however, is not typically what one would do. They would instead analyze the text across chapters in a book or across multiple books. Here, we've just demonstrated the concepts behind how you would go about carrying out sentiment analysis.

Word and Document Frequency

Beyond sentiment analysis, analysts of text are often interested in quantifying what a document is *about*. One could start by quantifying term frequency and looking at which terms occur most often; however, common words, such as the and and, are likely to appear most often. Those aren't unique to the work and hardly explain the text's topic. Often, these words, referred to as **stop words** are removed from analysis; however, these words are more important to some works relative to others. So, analysts tend to take a different approach: **inverse document frequency** (idf).

A document's **inverse document frequency** (idf) weights each term by its frequency in a collection of documents. Those words that are quite common in a set of documents are down-weighted. The weights for words that are less common are increased. By combining idf with term frequency (tf) (through multiplication), words that are common *and* unique to that document (relative to the collection of documents) stand out.

To see an example of this, we'll need a few more poems from Shel Silverstein for analysis. Here is *Invitation*:

```
library(tibble)
invitation <- c("If you are a dreamer, come in",
  "If you are a dreamer, a wisher, a liar",
  "A hope-er, a pray-er, a magic bean buyer...",
  "If you're a pretender, come sit by my fire",
  "For we have some flax-golden tales to spin.",
  "Come in!",
  "Come in!")

invitation <- tibble(line = 1:7, text = invitation, title = "Invitation")

invitation
# A tibble: 7 × 3
  line text          title
  <int> <chr>        <chr>
1     1 If you are a dreamer, come in,    Invitation
2     2 If you are a dreamer, a wisher, a liar   Invitation
3     3 A hope-er, a pray-er, a magic bean buyer... Invitation
4     4 If you're a pretender, come sit by my fire  Invitation
5     5 For we have some flax-golden tales to spin. Invitation
6     6 Come in!                               Invitation
7     7 Come in!                               Invitation
```

And, here is masks:

```
masks <- c("She had blue skin.",
  "And so did he.",
  "He kept it hid",
  "And so did she.",
  "They searched for blue",
  "Their whole life through",
  "Then passed right by",
  "And never knew")

masks <- tibble(line = 1:8, text = masks, title = "Masks")
```

```
masks
# A tibble: 8 × 3
  line text          title
  <int> <chr>        <chr>
1     1 She had blue skin. Masks
2     2 And so did he.  Masks
3     3 He kept it hid Masks
4     4 And so did she. Masks
5     5 They searched for blue Masks
6     6 Their whole life through Masks
7     7 Then passed right by— Masks
8     8 And never knew  Masks
```

We'll combine all three poems into a single data frame for TF-IDF analysis. To do so, we'll first add a column to our carrots example from above so that it has a column for title:

```
# add title to carrots poem
carrots <- text_df %>% mutate(title = "Carrots")

# combine all three poems into a tidy data frame
poems <- bind_rows(carrots, invitation, masks)
```

Now that we have our three documents (poems) in a single data frame, we can tokenize the text by word and calculate each tokens frequency within the document (poem).

```
# count number of times word appwars within each text
poem_words <- poems %>%
  unnest_tokens(word, text) %>%
  count(title, word, sort = TRUE)

# count total number of words in each poem
total_words <- poem_words %>%
  group_by(title) %>%
  summarize(total = sum(n))

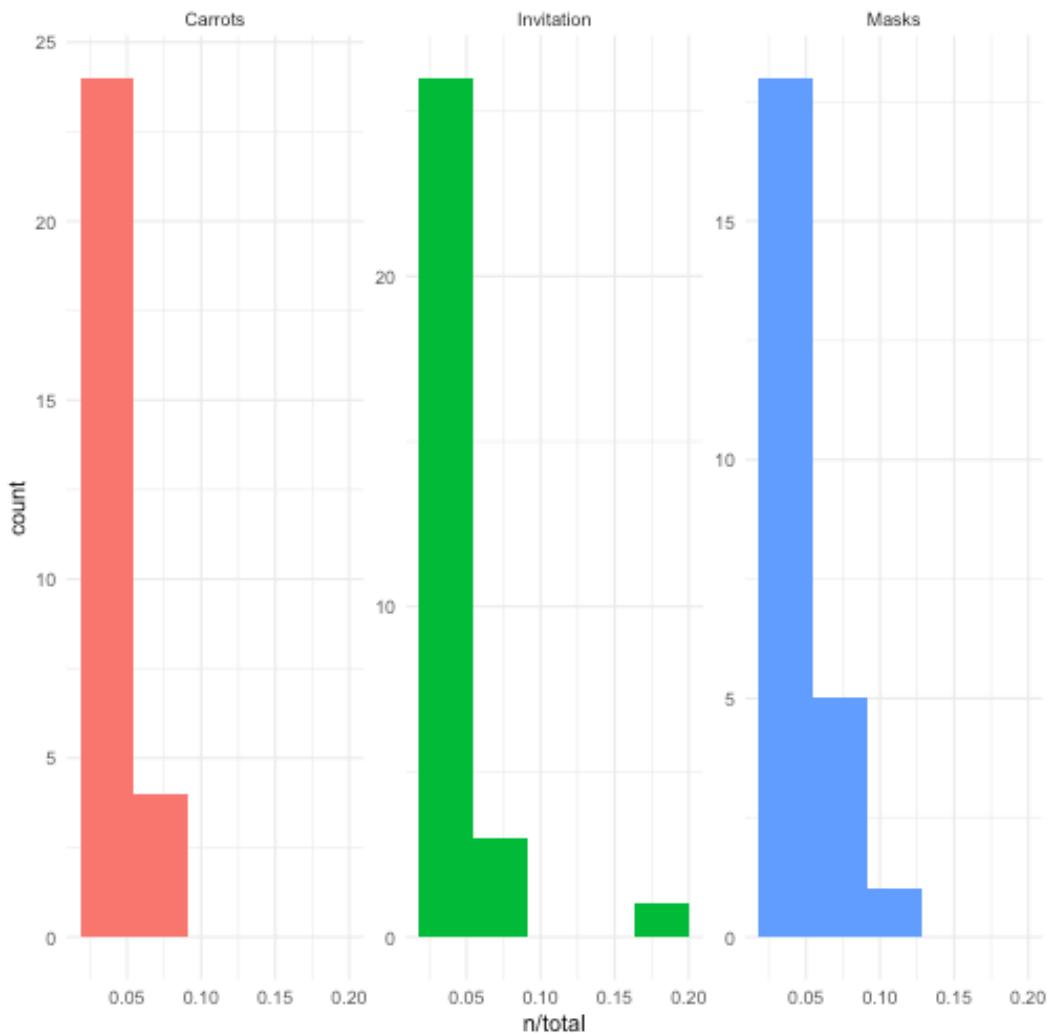
# combine data frames
poem_words <- left_join(poem_words, total_words)
```

```
Joining, by = "title"
poem_words
# A tibble: 82 × 4
  title     word     n total
  <chr>     <chr> <int> <int>
1 Invitation a      8    48
2 Invitation come   4    48
3 Carrots    they   3    33
4 Invitation if    3    48
5 Invitation in   3    48
6 Masks      and    3    31
7 Carrots    i      2    33
8 Carrots    that   2    33
9 Carrots    your   2    33
10 Invitation are   2    48
# ... with 72 more rows
```

Note that there are a different number of total words in each document, which is important to consider when you're comparing relative frequency between documents.

We could visualize the number of times a word appears relative to document length as follows:

```
library(ggplot2)
# visualize frequency / total words in poem
ggplot(poem_words, aes(n/total, fill = title)) +
  geom_histogram(show.legend = FALSE, bins = 5) +
  facet_wrap(~title, ncol = 3, scales = "free_y")
```



plot of chunk unnamed-chunk-117

With most documents there are only a few words that show up infrequently in the tail off to the right (rare words), while most words show up a whole bunch of times.

What we've just visualized is term frequency. We can add this quantity to our data frame:

```
freq_by_rank <- poem_words %>%
  group_by(title) %>%
  mutate(rank = row_number(),
    `term frequency` = n/total)
```

Notice that words that appear most frequently will have the largest term frequency. However, we're not just interested in word frequency, as stop words (such as "a") have the highest term frequency. Rather, we're interested in tf-idf - those words in a document that are unique relative to the other documents being analyzed.

To calculate tf-idf, we can use `bind_tf_idf()`, specifying three arguments: the column including the token (word), the column specifying the document from which the token originated (title), and the column including the number of times the word appears (n):

```
poem_words <- poem_words %>%
  bind_tf_idf(word, title, n)

# sort ascending
poem_words %>%
  arrange(tf_idf)
# A tibble: 82 × 7
  title     word      n total      tf     idf   tf_idf
  <chr>    <chr>  <int> <int>  <dbl>  <dbl>   <dbl>
1 Carrots   for      1    33  0.0303  0       0
2 Invitation for     1    48  0.0208  0       0
3 Masks     for      1    31  0.0323  0       0
4 Invitation by     1    48  0.0208  0.405  0.00845
5 Carrots   are      1    33  0.0303  0.405  0.0123
6 Carrots   did      1    33  0.0303  0.405  0.0123
7 Carrots   right    1    33  0.0303  0.405  0.0123
8 Carrots   you      1    33  0.0303  0.405  0.0123
9 Masks     by       1    31  0.0323  0.405  0.0131
10 Masks    right    1    31  0.0323  0.405  0.0131
# ... with 72 more rows
```

If we sort this output in ascending order by `tf_idf`, you'll notice that the word "for" has a `tf_idf` of 0. The data indicates that this word shows up with equal frequency across all three poems. It is *not* a word unique to any one poem.

```
# sort descending
poem_words %>%
  arrange(desc(tf_idf))
# A tibble: 82 × 7
  title     word      n total      tf     idf tf_idf
  <chr>    <chr> <int> <int> <dbl> <dbl>  <dbl>
1 Invitation a        8    48  0.167   1.10  0.183
2 Masks      and      3    31  0.0968  1.10  0.106
3 Invitation come     4    48  0.0833  1.10  0.0916
4 Masks      blue     2    31  0.0645  1.10  0.0709
5 Masks      he       2    31  0.0645  1.10  0.0709
6 Masks      she      2    31  0.0645  1.10  0.0709
7 Masks      so       2    31  0.0645  1.10  0.0709
8 Invitation if      3    48  0.0625  1.10  0.0687
9 Invitation in     3    48  0.0625  1.10  0.0687
10 Carrots     i       2    33  0.0606  1.10  0.0666
# ... with 72 more rows
```

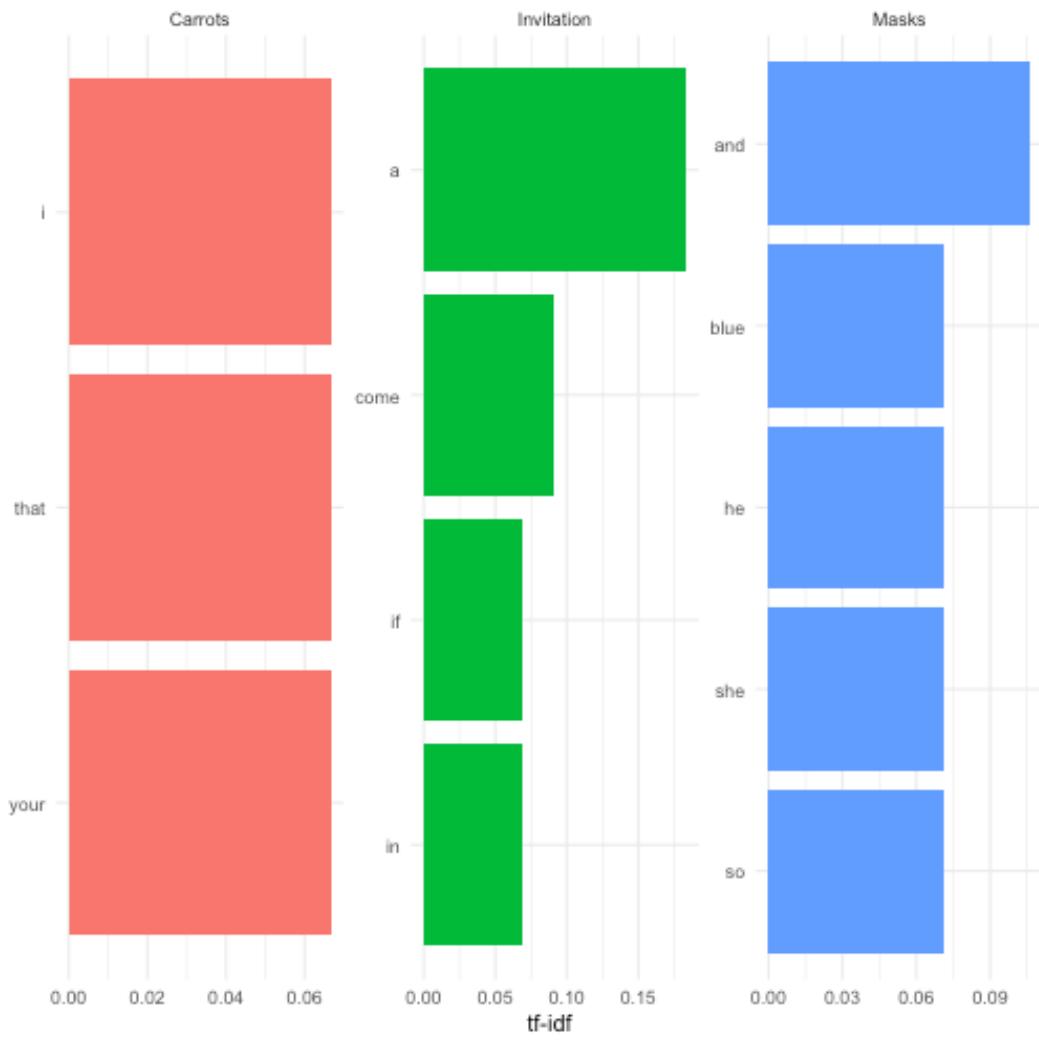
Alternatively, here we see the words most unique to the individual poems. “a” and “come” are most unique to *Invitation*, while “and” and “blue” are most unique to *Masks*. If we had removed stop words, we would have lost the fact that some common words are really unique in one of these poems relative to the others.

Again, we’re looking at a limited amount of text here, but this analysis can be applied to novels, works by different authors, or articles written in a newspaper.

We can summarize these tf-idf results by visualizing the words with the highest tf-idf in each of these poems:

```
poem_words %>%
  arrange(desc(tf_idf)) %>%
  mutate(word = factor(word, levels = rev(unique(word)))) %>%
  group_by(title) %>%
  top_n(3) %>%
  ungroup() %>%
  ggplot(aes(word, tf_idf, fill = title)) +
  geom_col(show.legend = FALSE) +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~title, ncol = 3, scales = "free") +
```

```
coord_flip()  
Selecting by tf_idf
```



plot of chunk unnamed-chunk-121

Functional Programming

Functional programming is an approach to programming in which the code evaluated is treated as a mathematical function. It is *declarative*, so expressions (or declarations) are used

instead of statements. Functional programming is often touted and used due to the fact that cleaner, shorter code can be written. In this shorter code, functional programming allows for code that is elegant but also understandable. Ultimately, the goal is to have simpler code that minimizes time required for debugging, testing, and maintaining.

R at its core is a functional programming language. If you're familiar with the `apply()` family of functions in base R, you've carried out some functional programming! Here, we'll discuss functional programming and utilize the `purrr` package, designed to enhance functional programming in R.

By utilizing functional programming, you'll be able to minimize redundancy within your code. The way this happens in reality is by determining what small building blocks your code needs. These will each be a function. These small building block functions are then combined into more complex structures to be your final program.

For Loops vs. Functionals

In base R, you likely found yourself writing for loops for iteration. For example, if you wanted to carry out an operation on every row of a data frame, you've likely written a for loop to do so that loops through each row of the data frame and carries out what you want to do. However, you also may have heard people bemoan this approach, arguing that it's slow and unnecessary. This is because R is a functional programming language. You can wrap for loops into a function and call the function instead of using the for loop.

Let's use an example to demonstrate what we mean by this. What if you had a data frame and wanted the median value for each column in the data frame? To see how you could approach this, we'll use the `trees` dataset available by default from R:

```
# see dataset
trees <- as_tibble(trees)
trees
# A tibble: 31 × 3
  Girth Height Volume
  <dbl>   <dbl>   <dbl>
1 8.3     70     10.3
2 8.6     65     10.3
3 8.8     63     10.2
4 10.5    72     16.4
5 10.7    81     18.8
6 10.8    83     19.7
```

```
7 11      66  15.6
8 11      75  18.2
9 11.1     80  22.6
10 11.2    75  19.9
# ... with 21 more rows
```

The dataset contains the diameter, height, and volume of 31 Black Cherry trees.

Copy + Paste Approach

To calculate the median for each column, you could do the following:

```
# calculate median of each column
median(trees$Girth)
[1] 12.9

median(trees$Height)
[1] 76

median(trees$Volume)
[1] 24.2
```

This would get you your answer; however, this breaks the programming rule that you shouldn't copy and paste more than once. And, you could imagine that if you had more than three columns, this would be a *huge* pain and involve a whole lot of copy and pasting and editing.

For Loop Approach

A second approach would be to use a for loop. You would loop through all the columns in the data frame, calculate the median, record that value and store that information in a variable.

```
# create output vector
output <- vector("double", ncol(trees))

# loop through columns
for (i in seq_along(trees)) {
  output[[i]] <- median(trees[[i]])
}
output
[1] 12.9 76.0 24.2
```

This allows us to obtain the same information as the copy + paste method; however, it scales better if there are more than three columns in your data frame, making it a better option than the copy + paste method.

But, what if you frequently want to take the median of the columns in your data frame? What if you want to do this more than once? You would have to go in, copy + paste this code and change the name of the data frame each time. This would break the don't copy + paste more than once rule.

Function Approach

This brings us to the function approach. Here, we wrap the for loop into a function so that we can execute a function on our data frame whenever we want to accomplish the task of calculating the median for each column:

```
# create function
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}

# execute function
col_median(trees)
[1] 12.9 76.0 24.2
```

Again, the output information from trees is the same for this specific example, but now we see that we could use the `col_median()` function any time we want to calculate the medians across columns within a data frame!

This is a much better approach as it allows you to generalize your code, *but* the above solution still requires you to loop through each column, making the code harder to read and understand at a glance. It fails to take advantage of R's functional programming capabilities.

purrr Approach

To really optimize this solution, we'll turn to `purrr`. Using `purrr` requires you to determine how to carry out your operation of interest for a single occurrence (i.e. calculate the median for a single column in your data frame). Then `purrr` takes care of carrying out that operation across your data frame. Further, once you break your problem down into smaller building blocks, `purrr` also helps you combine those smaller pieces into a functional program.

Let's use `purrr` (a core tidyverse package) to solve our calculate median for each column task. But, before we do that specifically, let's first introduce the general `map()` function.

map

We'll see usage of `map()` functions in just a second to accomplish our median for each column task, but before doing so, let's take a second to look at the generic usage for the family of `map` functions:

```
map(.x, .f, ...)  
map(INPUT, FUNCTION_TO_APPLY, OPTIONAL_OTHER_STUFF)
```

Note that the input to a `map` function requires you to first specify a vector input followed by the function you'd like to apply. Any other arguments to the function you want to pass follow at the end of the call.

When it comes to our specific task, this is implemented as follows using `map_db1()`:

```
library(purrr)  
# use purrr to calculate median  
map_db1(trees, median)  
Girth Height Volume  
12.9    76.0   24.2
```

Here, we use the `map_db1()` function from `purrr` to iterate over the columns of `trees` and calculate the median. And, it even displays the variable name in the output for us - all in a single function call.

Note the flexibility! We've just passed the `median()` function *into* another function: `map_db1`. This means that if we changed our minds and wanted mean instead, we could accomplish that with ease:

```
# use purrr to calculate mean
map_db1(trees, mean)
  Girth   Height   Volume
13.24839 76.00000 30.17097
```

This function exists because looping to do something to each element and saving the results is such a common task, that there is family of functions (one of which is `map_db1`) to do it for you to accomplish such tasks in `purrr`.

We'll note here that `purrr`'s functions are all implemented in the C programming language, making them faster than the function we generated previously.

In the example above `mean` could have been any function, denoted in the `purrr` documentation as `.f`. This specifies the function you'd like to apply to the vector you've specified.

After `.f` in `purrr` functions, you can pass additional arguments. These go *after* the specified function. For example, below, we specify that we'd like to remove NAs, by specifying an argument to be passed to the `mean()` function after the function call (`mean`):

```
# use purrr to calculate mean
map_db1(trees, mean, na.rm = TRUE)
  Girth   Height   Volume
13.24839 76.00000 30.17097
```

map Functions

The `map` family of functions from the `purrr` package are analogous to the `apply()` family of functions from base R. If you're familiar with `lapply()`, `vapply()`, `tapply`, and `sapply()`, the thinking will similar; however, `purrr` provides a much more consistent syntax and are much easier to learn and implement consistently.

As you saw in the median example above, `map` functions carry out an operation repeatedly and store the output of that operation for you. There are a number of different `map` functions. To determine which to use, consider the output you want to obtain from your operation. Above, we wanted a double vector, so we used `map_db1`. However, you can return a number of different outputs from the `map` functions. A few are listed here and we'll introduce even more shortly:

- `map()` - returns a list
- `map_lgl()` - returns a logical vector
- `map_int()` - returns an integer vector
- `map_db1()` - returns a double vector
- `map_chr()` - returns a character vector

These all take a vector and a function as an input. The function is applied to the vector and a new vector (of the same length & with the same names) is returned of the type specified in the `map` function call.

There are also the variations `map_df`, `map_dfr` and `map_dfc`, which will create a dataframe (the tidyverse version called a tibble) from the output by either combining the data by rows with `map_df()` and `map_dfr()` or by column with `map_dfc()`.

```
# use map_dfr to calculate mean and create a dataframe
map_dfr(trees, mean, na.rm = TRUE)
# A tibble: 1 × 3
  Girth Height Volume
  <dbl>   <dbl>   <dbl>
1 13.2     76    30.2
```

Multiple Vectors

So far, we've only looked at iterating over a single vector at a time; however in analysis, you'll often find that you need to iterate over more than one vector at a time. The `purrr` package has two functions that simplify this process for you: `map2` and `pmap`.

map2

`map2()` allows you to iterate over two vectors at the same time. The two vectors you want to iterate over are first specified within the `map2()` function call, followed by the function to execute. Any arguments after the function you'd like `map2()` to apply are specified at the end of the `map2()` call.

The generic usage for `map2()` is:

```
map2(.x, .y, .f, ...)  
map(INPUT_ONE, INPUT_TWO, FUNCTION_TO_APPLY, OPTIONAL_OTHER_STUFF)
```

What if we wanted to calculate the volume of each tree? There is a column for volume, but let's see if we can't use a little geometry to calculate it on our own.

If we assume that each tree is a cylinder, the volume of a cylinder is $V = \pi r^2 h$, where r is half the diameter. In the trees dataset, the diameter is stored in the `Girth` column, in *inches*. h is the height of the cylinder, which is stored in the `Height` column, in *feet*.

Thus, we have two vectors we want to operate over, `Girth` and `Height`, so we'll use `map2()`.

Let's first generate a function that will calculate volume for us from the information in our `trees` dataset:

```
# generate volume function  
volume <- function(diameter, height){  
  # convert diameter in inches to radius in feet  
  radius_ft <- (diameter/2)/12  
  # calculate volume  
  output <- pi * radius_ft^2 * height  
  return(output)  
}
```

Now, we can utilize `map2` then to calculate the volume from these two input vectors:

```
# calculate volume
map2_dbl(trees$Girth, trees$Height, volume)
[1] 26.30157 26.22030 26.60929 43.29507 50.58013 52.80232 43.55687
[8] 49.49645 53.76050 51.31268 55.01883 53.87046 53.87046 51.51672
[15] 58.90486 67.16431 77.14819 82.97153 72.68200 66.47610 83.38311
[22] 87.98205 84.85845 100.53096 111.58179 132.22227 136.96744 139.80524
[29] 141.37167 141.37167 201.36365
```

Here the output is on the same order as the `Volume` column from the dataset, but the numbers are off, suggesting that the dataset calculated volume of the tree differently than we did in our approach.

```
trees$Volume
[1] 10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 24.2 21.0 21.4 21.3 19.1
[16] 22.2 33.8 27.4 25.7 24.9 34.5 31.7 36.3 38.3 42.6 55.4 55.7 58.3 51.5 51.0
[31] 77.0
```

Note that there are all the same variations that exist for `map_` exist for `map2()`, so you're able to use `map2_chr()` and `map2_db1()`, etc.

Additionally, the `map` functions work well within our `dplyr` approach to working with data. Here, we add the output for our volume calculation to the `trees` dataset as well as a column (`volume_diff`) that displays the difference between our volume calculation and that reported in the dataset:

```
# calculate volume
trees %>%
  mutate(volume_cylinder = map2_dbl(trees$Girth, trees$Height, volume),
        volume_diff = Volume - volume_cylinder)
# A tibble: 31 × 5
  Girth Height Volume volume_cylinder volume_diff
  <dbl>   <dbl>   <dbl>          <dbl>        <dbl>
1    8.3     70    10.3         26.3       -16.0
2    8.6     65    10.3         26.2       -15.9
3    8.8     63    10.2         26.6       -16.4
4   10.5     72    16.4         43.3       -26.9
5   10.7     81    18.8         50.6       -31.8
6   10.8     83    19.7         52.8       -33.1
7   11.0     66    15.6         43.6       -28.0
```

```

8 11      75 18.2      49.5      -31.3
9 11.1     80 22.6      53.8      -31.2
10 11.2    75 19.9      51.3      -31.4
# ... with 21 more rows

```

pmap

While `map()` allows for iteration over a single vector, and `map2()` allows for iteration over *two* vectors, there is no `map3()`, `map4()`, or `map5()` because that would get too unwieldy. Instead, there is a single and more general `pmap()` - which stands for parallel map - function. The `pmap()` function takes a list of arguments over which you'd like to iterate:

The generic usage for this function is:

```

pmap(.l, .f, ...)
pmap(LIST_OF_INPUT_LISTS, FUNCTION_TO_APPLY, OPTIONAL_OTHER_STUFF)

```

Note that `.l` is a list of all the input vectors, so you are no longer specifying `.x` or `.y` individually. The rest of the syntax remains the same.

Anonymous Functions

In our `map2()` example we created a separate function to calculate volume; however, as this is a specific scenario for volume calculation, we likely won't need that function again later. In such scenarios, it can be helpful to utilize an **anonymous function**. This is a function that is not given a name but that is utilized within our `map` call. We are *not* able to refer back to this function later, but we *are* able to use it within our `map` call:

```

map2_db1(trees$Girth, trees$Height, function(x,y){ pi * ((x/2)^2 * y)})
[1] 26.30157 26.22030 26.60929 43.29507 50.58013 52.80232 43.55687
[8] 49.49645 53.76050 51.31268 55.01883 53.87046 53.87046 51.51672
[15] 58.90486 67.16431 77.14819 82.97153 72.68200 66.47610 83.38311
[22] 87.98205 84.85845 100.53096 111.58179 132.22227 136.96744 139.80524
[29] 141.37167 141.37167 201.36365

```

In this example, we create the anonymous function within the `map2_db1()` call. This allows volume to be calculated as before, but does so without having to define a function.

This becomes particularly helpful within `purrr` if you want to refer to the individual elements of your `map` call directly. This is done by specifying `.x` and `.y` to refer to the first and second input vectors, respectively:

```
map2_dbl(trees$Girth, trees$Height, ~ pi * ((.x/2)/12)^2 * .y)
[1] 26.30157 26.22030 26.60929 43.29507 50.58013 52.80232 43.55687
[8] 49.49645 53.76050 51.31268 55.01883 53.87046 53.87046 51.51672
[15] 58.90486 67.16431 77.14819 82.97153 72.68200 66.47610 83.38311
[22] 87.98205 84.85845 100.53096 111.58179 132.22227 136.96744 139.80524
[29] 141.37167 141.37167 201.36365
```

Here, we see the same output; however, the syntax defines an anonymous function using the formula syntax.

Exploratory Data Analysis

The goal of an exploratory analysis is to examine, or **explore** the data and find **relationships** that weren't previously known. Exploratory analyses explore how different measures might be related to each other but do not confirm that relationship as causal, i.e., one variable causing another. You've probably heard the phrase "Correlation does not imply causation," and exploratory analyses lie at the root of this saying. Just because you observe a relationship between two variables during exploratory analysis, it does not mean that one necessarily causes the other.

Because of this, exploratory analyses, while useful for discovering new connections, should not be the final say in answering a question! It can allow you to formulate hypotheses and drive the design of future studies and data collection, but exploratory analysis alone should never be used as the final say on why or how data might be related to each other. In short, exploratory analysis helps us ask better questions, but it does not answer questions. More specifically, we explore data in order to:

- Understand data properties such as nonlinear relationships, the existence of missing values, the existence of outliers, etc.
- Find patterns in data such as associations, group differences, confounders, etc.
- Suggest modeling strategies such as linear vs. nonlinear models, transformation
- "Debug" analyses
- Communicate results

General Principles of EDA

We can summarize the general principles of exploratory analysis as follows:

- Look for missing values
- Look for outlier values
- Use plots to explore relationships
- Use tables to explore relationships
- If necessary, transform variables

These principles may be more clear in an example. We will use a dataset from [Kaggle.com](#) that contains 120 years of Olympics history on athletes and results. If you don't have an account on Kaggle, create one and go to the link <https://www.kaggle.com/heesoo37/120-years-of-olympic-history-athletes-and-results> and under "Data Sources" download the `athlete_events.csv` to your computer.

Download this

<https://www.kaggle.com/heesoo37/120-years-of-olympic-history-athletes-and-results>

Dataset on 120 years of Olympics history on athletes and results

Upload the data in R and import the CSV file using the commands you have learned. Unfortunately, you cannot download the CSV file directly from the web address since downloading datasets on Kaggle requires logging in.

```
library(readr)
# Loading data as a data frame
df <- read_csv("athlete_events.csv")

## Parsed with column specifications:
## # colSpecs
##   ID = col_integer(),
##   Name = col_character(),
##   Sex = col_character(),
##   Age = col_integer(),
##   Height = col_integer(),
##   Weight = col_double(),
##   Team = col_character(),
##   NOC = col_character(),
##   Game = col_character(),
##   Year = col_integer(),
##   Season = col_character(),
##   City = col_character(),
##   Sport = col_character(),
##   Event = col_character(),
##   Medal = col_character()
## }
```

Importing data using `read_csv()`

As we learned before, we can use the package `skimr` to take a look at the data.

```

library(skimr)
#> #> summary of the dataset
skim(df)

## └─ Variable type:character
##   variable missing complete n    min    max empty n_unique
##   City      0    271116 4    22      0     42
##   Event     0    271116 271116 15   85      0     765
##   Games     0    271116 271116 11   11      0     31
##   Medal    231333 39783 271116 4    6      0     3
##   Name      0    271116 271116 2    108      0    134731
##   NOC       0    271116 271116 3    3      0     230
##   Season    0    271116 271116 6    6      0     2
##   Sex       0    271116 271116 1    1      0     2
##   Sport     0    271116 271116 4    25      0     46
##   Team      0    271116 271116 2    47      0    1184
## 
## └─ Variable type:integer
##   variable missing complete n    mean    sd   p0   p25   p50   p75
##   Age       9474  261842 271116 25.56  8.39  10   21   24   28
##   Height    60171 210945 271116 175.34 10.52 127  188  175  183
##   ID        0    271116 271116 68248.95 39022.29 1 36643 68205 1e+05
##   Year      0    271116 271116 1978.38 29.88 1836 1960 1988 2002
##   p100      hist
##   97
##   226
##   135571
##   2016
## 
## └─ Variable type:numeric
##   variable missing complete n    mean    sd   p0   p25   p50   p75   p100
##   Weight   62875  209241 271116 70.7 14.35 25  60   70   79   214
##   hist

```

Using the skimr package to have a summary of the data

We see that the dataset contains 15 variables and 271,116 observations. Some of the variables are of factor type and others are of integer or numeric type. The dataset includes variables on athletes such as name, sex, the sport played, whether they received a medal, age, and height. We first need to understand the data properties. So let's start with missing values.

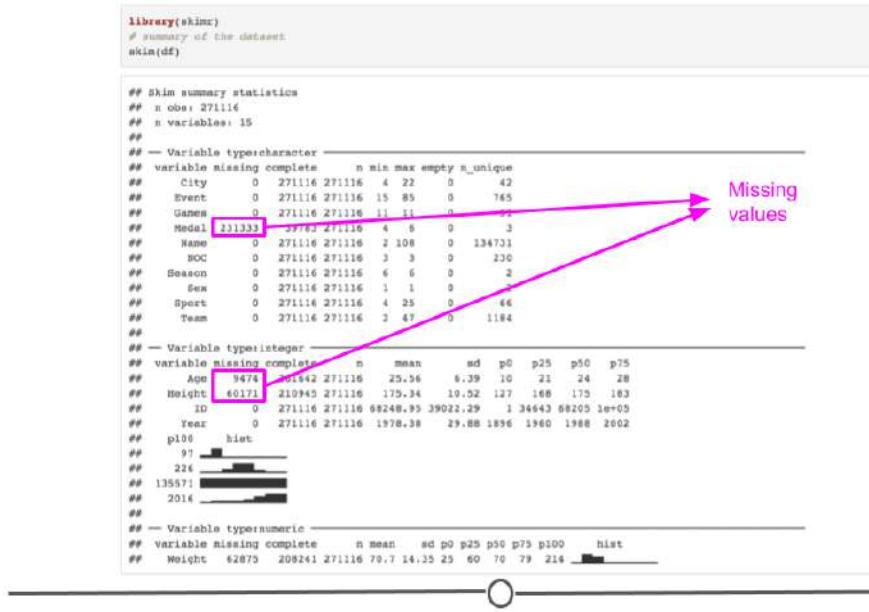
```

library(shiny)
# summary of the dataset
skim(df)

## # A tibble: 15 × 10
##   variable type    missing complete n    min   max empty n_unique
##   <chr>     <chr>    <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 City      character 0       271116 4     22    0     42
## 2 Event     character 0       271116 15    85    0     765
## 3 Games     character 0       271116 11    11    0     31
## 4 Medal     integer    231333 39783 271116 4     6     0     3
## 5 Name      character 0       271116 2     108   0     134731
## 6 NOC       character 0       271116 3     3     0     230
## 7 Season    character 0       271116 6     6     0     2
## 8 Sex       character 0       271116 1     1     0     2
## 9 Sport     character 0       271116 4     25    0     46
## 10 Team     character 0       271116 3     47    0     1184
## # ... with 5 more variables:
## #   Age <dbl> 261942 271116 25.56 8.39 10 21 24 28
## #   Height <dbl> 210945 271116 175.34 10.52 127 188 175 183
## #   ID <dbl> 0 271116 271116 68248.95 39022.29 1 36443 68205 1e+05
## #   Year <dbl> 0 271116 271116 1978.38 29.88 1836 1960 1988 2002
## #   p100 <dbl> hist
## #   97 [redacted]
## #   226 [redacted]
## #   135571 [redacted]
## #   2016 [redacted]
## # ... with 1 more variable:
## #   Weight <dbl> 209241 271116 70.7 14.35 25 60 70 79 214
## #   histogram [redacted]
## #   p100 [redacted]
```

We have different types of variables in our data

First, the results of the `skim()` function indicate that some of our variables have lots of missing values. For instance, the variable `Medal` has 231,333 missing values. Generally, this is a place for concern since most statistical analyses ignore observations with missing values. However, it is obvious that the missing values for the variable `Medal` are mainly because the athlete didn't receive any medals. So this kind of missing value should not be a problem. However, we have missing values in the variables `Height` and `Age`. Since we are going to use these variables in our analysis in this lesson, observations with missing values for these two variables will be dropped from our analysis. Remember that `NA` is the most common character for missing values, but sometimes they are coded as spaces, 999, -1 or "missing". Check for missing values in a variety of ways.



There are some missing values in the data

Second, we can see that there are some outliers in some of the numerical variables. For example, look at the summary of the variable `Age`. Although the average age among all the athletes is around 25, there is an individual who is 97 years old (fun fact: use the command `subset(df, df$Age == 97)` to check out the information about this athlete. You will see that the name of the athlete is John Quincy Adams Ward and he competed in the sport(!) Art Competitions Mixed Sculpturing in 1928. This artist is known for his George Washington statue in front of Federal Hall in Wall Street in New York City.) It is always good to know about the existence of outliers in your sample. Outliers can significantly skew the results of your analysis. You can find outliers by looking at the distribution of your variable too.

```

library(shinr)
# summary of the dataset
skim(df)

## ━━━━ Variable type:character ━━━━
## variable missing complete n min max empty n_unique
##   City      0 271116 271116 4 22      0     42
##   Event     0 271116 271116 15 85      0     765
##   Games     0 271116 271116 11 11      0     31
##   Medal    211333 39783 271116 4 6      0     3
##   Name      0 271116 271116 2 108      0 134731
##   NOC       0 271116 271116 3 3      0     230
##   Season    0 271116 271116 6 6      0     2
##   Sex       0 271116 271116 1 1      0     2
##   Sport     0 271116 271116 4 25      0     46
##   Team      0 271116 271116 2 47      0     1184
##   p100      111
##   97       224
##   135571
##   2016

## ━━━━ Variable type:integer ━━━━
## variable missing complete n mean sd p0 p25 p50 p75 p100
##   Age      9474 261942 271116 25.56 1.79 10 21 24 28
##   Height   60171 210945 271116 178.5 10.52 127 188 175 183
##   ID       0 271116 271116 89248.95 39022.29 1 36443 68205 1e+05
##   Year     0 271116 271116 1978.38 29.88 1836 1960 1988 2002
##   hist

## ━━━━ Variable type:numeric ━━━━
## variable missing complete n mean sd p0 p25 p50 p75 p100 hist
##   Weight   62875 209241 271116 70.7 14.35 25 60 70 79 214

```



Outlier

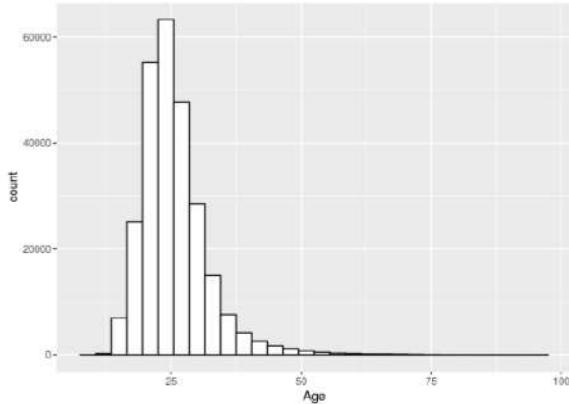
There is an outlier in the Age variable

Histograms, in general, are one of the best ways to look at a variable and find abnormalities. You can see that the age of most individuals in the sample are between 18-35.

```
library(ggplot2)
ggplot(df, aes(x=Age)) +
  geom_histogram(color="black", fill="white")

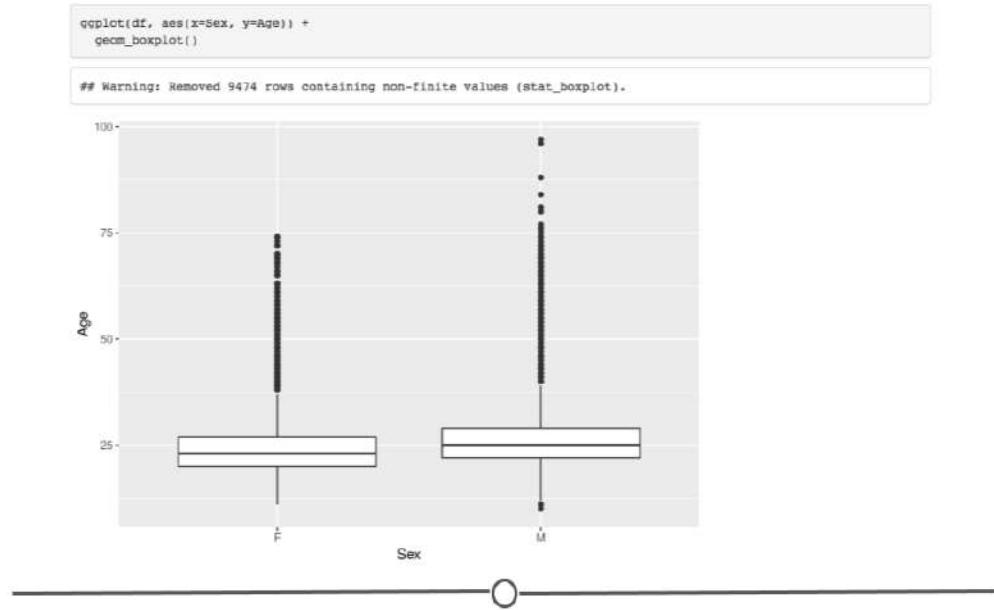
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 9474 rows containing non-finite values (stat_bin).
```



Histogram of the variable Age

Now, rather than just summarizing the data points within a single variable, we can look at how two or more variables might be related to each other. For instance, we like to know if there is an association between age of athletes and their gender. One of the ways to do this is to look at a boxplot of age grouped by gender, i.e., the distribution of age separated for male and female athletes. Boxplot shows the distribution of the variable age for the gender groups. You can see that the average age is slightly higher for men than for women.



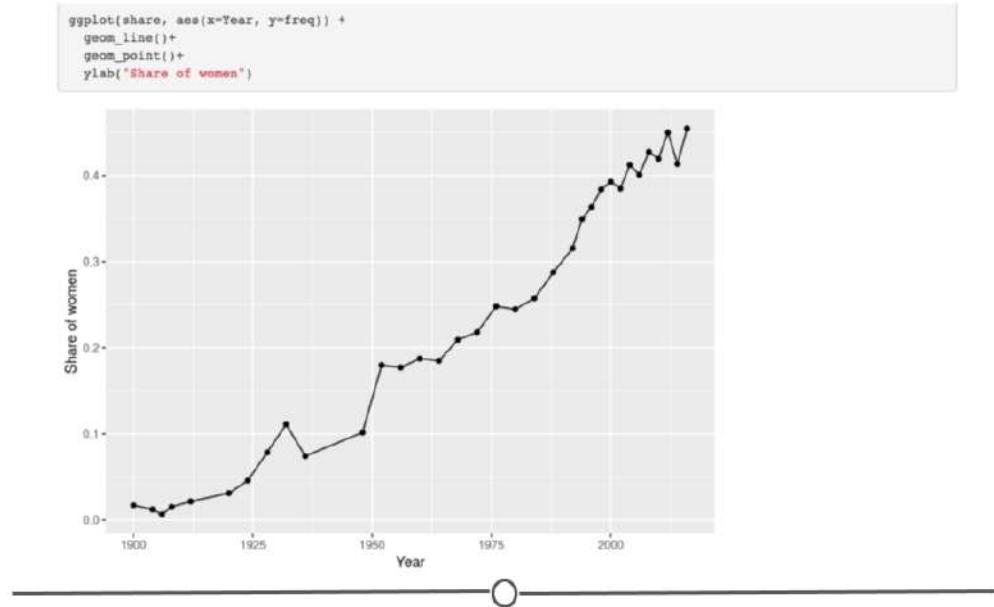
Boxplot of the variable Age for male and female individuals

If we are interested in looking at the distribution of male and female athletes over time, we can use frequency tables. Let us first create a frequency table of the share of women in each Olympic event. Tables are good for looking at factor or character variables.

```
share <- df %>%  
  group_by(Year, Sex) %>%  
  summarise(n = n()) %>%  
  mutate(freq = n / sum(n)) %>%  
  filter(Sex == "F")
```

```
share
# A tibble: 34 × 4
# Groups:   Year [34]
  Year Sex     n    freq
  <dbl> <chr> <int>   <dbl>
1 1900 F      33 0.0170
2 1904 F      16 0.0123
3 1906 F      11 0.00635
4 1908 F      47 0.0152
5 1912 F      87 0.0215
6 1920 F     134 0.0312
7 1924 F     261 0.0458
8 1928 F     437 0.0784
9 1932 F     369 0.111
10 1936 F     549 0.0742
# ... with 24 more rows
```

Now, if we want to plot this trend, we can use `geom_line()` from `ggplot`. It's interesting that the share of women among all athletes that was once at a very low level in the early 1900s has gone up to almost 50% in modern times.



Plot of the share of female athletes over time

In general, the most important plots in exploratory data analysis are:

- Scatterplots (`geom_point()`)
- Histograms (`geom_histogram()`)
- Density plots (`geom_density()`)
- Boxplots (`geom_boxplot()`)
- Barplots (`geom_bar()`)

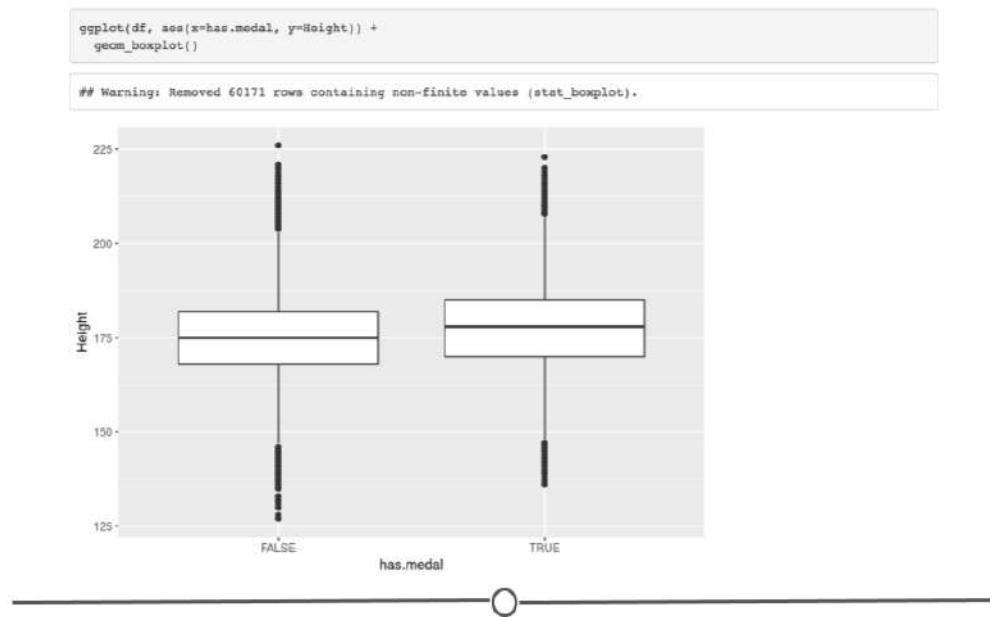
To end our lesson on exploratory analysis, let's consider a question: are taller athletes more likely to win a medal? To answer this question we can use different methods. We can look at the distribution of height for those who received a medal and those who didn't. We can use boxplots or barplots. The choice is yours but because boxplots are more informative, we will use them. We can first create a variable that indicates whether the athlete has any medal (the variable `Medal` indicates the type of medals). Note that the variable `has.medal` is a transformation of the variable `Medal`.

```
df <- df %>%
  mutate(has.medal=Medal %in% c("Gold", "Silver", "Bronze"))

table(df$has.medal)

##      FALSE    TRUE
## 231333 39783
```

Creating a variable that shows whether the athlete has a medal or not
And now, we use the following code to create the boxplot.



Boxplot for the relationship between height and having won a medal

What is obvious is that those who have a medal are taller. Can we say that being tall increases the probability of winning a medal in the Olympics? The answer to this question is that we don't know. There are some possible scenarios. For instance, it could be true that being tall increase the chances of winning medals. But it could also be that there are more medals awarded in sports such as volleyball or basketball that require taller athletes. In these sports, every member of the winning team gets a medal (even if country counts only one medal is counted for the country). As a result, we may end up having so many tall athletes with a medal in each Olympics. It could also be that there are other confounding factors involved that explain why an athlete wins a medal. We will learn about confounding variables in future lessons. For now, it's important to know, as we said in the beginning of this lesson, that association or correlation does not mean causation.

In the next module we will cover more methods for visualizing data.

Analyzing JSON in R

Above we discussed how to analyze pure text (meaning, text written by humans in their native written and spoken language). Here, we'll discuss how to briefly how others have wrangled text-based data from the Internet in the JSON format within R. This is possible

because of the R package `jsonlite`, which was used in the following example:

Kan Nishida, a data scientist, was interested in [understanding what restaurant types found most frequently in each state or province](#). To do this, he used JSON data originally released from Yelp. He wrangled the data from JSON format into a tabular format using `jsonlite` and other data wrangling packages, such as `dplyr`, to ultimately determine the types of restaurants found most frequently in a number of different states and provinces.

Analyzing XML in R

To see an example of not only using `xm12` to parse XML data, but also another example of using `rvest` to obtain the XML data, check out this post from [José Roberto Ayala Solares](#) where he took the text from a New York Times article called “[Trump’s Lies](#)”, scraped the data from the web (obtaining it in XML), and then [wrangled it into a tidy format](#) using `xm12`.

In this lesson, our goal is to make you aware that data from the Internet (and APIs in particular) will often come in either JSON or XML format. Thus, the JSON and XML examples provided here only give you a bit of an idea of what JSON and XML data are and how to work with them. Nevertheless, the more frequently you retrieve data from APIs and the Internet, the more comfortable you’ll have to become with both JSON and XML. And, `jsonlite` and `xm12` will help you as you work with these data in R!

Case Studies

So far, we’ve introduced the case studies and read the raw data into R.

Let’s load the raw data that we previously saved using the `here` package.

```
library(here)
load(here::here("data", "raw_data", "case_study_1.rda"))
load(here::here("data", "raw_data", "case_study_2.rda"))
#This loads all the data objects that we previously saved in our raw_data directory. Recall that this directory is located within a directory called data that\ is located within the directory where our project is located.
```

Now, we will work to get the data into two tidy formatted datasets that will include the information needed to answer our questions of interest.

Case Study #1: Health Expenditures

We've already read in the datasets we'll use for this health expenditures case study, but they're not yet cleaned and wrangled. So, we'll do that here!

As a reminder, we're ultimately interested in answering the following questions with these data:

1. Is there a relationship between health care coverage and health care spending in the United States?
2. How does the spending distribution change across geographic regions in the United States?
3. Does the relationship between health care coverage and health care spending in the United States change from 2013 to 2014?

This means that we'll need all the data from the variables necessary to answer this question in our tidy dataset.

health care Coverage Data

Let's remind ourselves before we get to wrangling what data we have when it comes to health care coverage.

```
coverage
# A tibble: 52 × 29
  Location `2013_Employer` `2013_Non-Grou... `2013_Medicaid` `2013_Medicare` 
  <chr>      <dbl>          <dbl>          <dbl>          <dbl>        
1 United S... 155696900    13816000     54919100    40876300  
2 Alabama     2126500      174200       869700      783000    
3 Alaska       364900       24000        95000       55200      
4 Arizona     2883800      170800       1346100     842000    
5 Arkansas    1128800      155600       600800      515200    
6 Californ... 17747300     1986400      8344800     3828500   
7 Colorado    2852500      426300       697300      549700    
8 Connecti... 2030500      126800       532000      475300    
9 Delaware    473700       25100        192700      141300    
10 District... 324300       30400        174900      59900      
# ... with 42 more rows, and 24 more variables: 2013_Other Public <chr>,
```

```
# 2013_Uninsured <dbl>, 2013_Total <dbl>, 2014_Employer <dbl>,
# 2014_Non-Group <dbl>, 2014_Medicaid <dbl>, 2014_Medicare <dbl>,
# 2014_Other Public <chr>, 2014_Uninsured <dbl>, 2014_Total <dbl>,
# 2015_Employer <dbl>, 2015_Non-Group <dbl>, 2015_Medicaid <dbl>,
# 2015_Medicare <dbl>, 2015_Other Public <chr>, 2015_Uninsured <dbl>,
# 2015_Total <dbl>, 2016_Employer <dbl>, 2016_Non-Group <dbl>, ...
```

At a glance, we see that state-level information is stored in rows (with the exception of the first row, which stores country-level information) with columns corresponding to the amount of money spent on each type of health care, by year.

States Data

To work with these data, we'll also want to be able to switch between full state names and two letter abbreviations. There's data in R available to you for just this purpose!

```
library(datasets)
data(state)
state.name
[1] "Alabama"      "Alaska"        "Arizona"       "Arkansas"
[5] "California"   "Colorado"      "Connecticut"   "Delaware"
[9] "Florida"       "Georgia"       "Hawaii"        "Idaho"
[13] "Illinois"     "Indiana"       "Iowa"          "Kansas"
[17] "Kentucky"      "Louisiana"    "Maine"         "Maryland"
[21] "Massachusetts" "Michigan"      "Minnesota"    "Mississippi"
[25] "Missouri"      "Montana"       "Nebraska"     "Nevada"
[29] "New Hampshire" "New Jersey"   "New Mexico"   "New York"
[33] "North Carolina" "North Dakota" "Ohio"          "Oklahoma"
[37] "Oregon"        "Pennsylvania" "Rhode Island" "South Carolina"
[41] "South Dakota"   "Tennessee"   "Texas"         "Utah"
[45] "Vermont"        "Virginia"     "Washington"  "West Virginia"
[49] "Wisconsin"      "Wyoming"
```

Before going any further, let's add some information about Washington, D.C, the nation's capital, which is not a state, but a territory.

```

state.abb <- c(state.abb, "DC")
state.region <- as.factor(c(as.character(state.region), "South"))
state.name <- c(state.name, "District of Columbia")
state_data <- tibble(Location = state.name,
                      abb = state.abb,
                      region = state.region)

state_data
# A tibble: 51 × 3
  Location    abb   region
  <chr>      <chr> <fct>
1 Alabama     AL    South
2 Alaska      AK    West
3 Arizona     AZ    West
4 Arkansas    AR    South
5 California  CA    West
6 Colorado    CO    West
7 Connecticut CT    Northeast
8 Delaware    DE    South
9 Florida     FL    South
10 Georgia    GA   South
# ... with 41 more rows

```

If we focus in on the columns within this dataframe, we see that we have a number of different types of health care (i.e. employer, medicare, medicaid, etc.) for each year between 2013 and 2016:

```

names(coverage)
[1] "Location"          "2013_Employer"       "2013_Non-Group"
[4] "2013_Medicaid"      "2013_Medicare"        "2013_Other Public"
[7] "2013_Uninsured"     "2013_Total"          "2014_Employer"
[10] "2014_Non-Group"     "2014_Medicaid"        "2014_Medicare"
[13] "2014_Other Public"  "2014_Uninsured"       "2014_Total"
[16] "2015_Employer"      "2015_Non-Group"       "2015_Medicaid"
[19] "2015_Medicare"      "2015_Other Public"    "2015_Uninsured"
[22] "2015_Total"         "2016_Employer"        "2016_Non-Group"
[25] "2016_Medicaid"      "2016_Medicare"        "2016_Other Public"
[28] "2016_Uninsured"     "2016_Total"          ""

```

While a lot of information in here will be helpful, it's not in a tidy format. This is because,

each variable is not in a separate column. For example, each column includes year, the type of coverage *and* the amount spent by state. We'll want to use each piece of information separately downstream as we start to visualize and analyze these data. So, let's work to get these pieces of information separated out now.

To accomplish this, the first thing we'll have to do is reshape the data, using the `pivot_longer()` function from the `tidyr` package. As a reminder, this function gathers multiple columns and collapses them into new name-value pairs. This transforms data from wide format into a long format, where:

- The first argument defines the columns to gather
- The `names_to` argument is the name of the new column that you are creating which contains the values of the column headings that you are gathering
- The `values_to` argument is the name of the new column that will contain the values themselves; you can indicate the name of this column with the `values_to` argument.

Here, we create a column titled `year_type` and `tot_coverage`, storing this newly formatted data frame back into the variable name `coverage`. We also want to keep the `Location` column as it is because it also contains observational level data.

```
coverage <- coverage %>%
  mutate(across(starts_with("20"),
               as.integer)) %>% ## Convert all year-based columns to integer
  pivot_longer(-Location, ## Use all columns BUT 'Location'
               names_to = "year_type",
               values_to = "tot_coverage")
Warning in mask$eval_all_mutate(quo): NAs introduced by coercion

coverage
# A tibble: 1,456 × 3
  Location    year_type      tot_coverage
  <chr>        <chr>           <int>
1 United States 2013_Employer     155696900
2 United States 2013_Non-Group    13816000
```

```

3 United States 2013_Medicaid      54919100
4 United States 2013_Medicare       40876300
5 United States 2013_Other Public   6295400
6 United States 2013_Uninsured     41795100
7 United States 2013_Total         313401200
8 United States 2014_Employer       154347500
9 United States 2014_Non-Group     19313000
10 United States 2014_Medicaid      61650400
# ... with 1,446 more rows

```

Great! We still have `Location` stored in a single column, but we've separated out `year_type` and `tot_coverage` into their own columns, storing all of the information in a `long` data format.

Unfortunately, the `year_type` column still contains two pieces of information. We'll want to separate these out to ensure that the data are in a properly tidy format. To do this, we'll use the `separate()` function, which allows us to separate out the information stored in a single column into two columns. We'll also use the `convert=TRUE` argument to convert the character to an integer.

```

coverage <- coverage %>%
  separate(year_type, sep = "__",
           into = c("year", "type"),
           convert = TRUE)

coverage
# A tibble: 1,456 × 4
  Location     year type     tot_coverage
  <chr>       <int> <chr>        <int>
  1 United States 2013 Employer    155696900
  2 United States 2013 Non-Group   13816000
  3 United States 2013 Medicaid    54919100
  4 United States 2013 Medicare    40876300
  5 United States 2013 Other Public 6295400
  6 United States 2013 Uninsured   41795100
  7 United States 2013 Total       313401200
  8 United States 2014 Employer    154347500
  9 United States 2014 Non-Group   19313000
 10 United States 2014 Medicaid    61650400
# ... with 1,446 more rows

```

Perfect! We now have the four columns we wanted, each storing a separate piece of information, *and* the year column is an integer, as you would want it to be!

Let's go one step further and add in the state-level abbreviations and region for each row. We'll utilize our state datasets that we read in previously to accomplish this! Because we formatted the state data as a tibble, we can simply join it with our coverage dataset to get the state and region information.

```
coverage <- coverage %>%
  left_join(state_data, by = "Location")

coverage
# A tibble: 1,456 × 6
  Location     year   type      tot_coverage   abb   region
  <chr>        <int>  <chr>        <int> <chr> <fct>
  1 United States 2013 Employer    155696900 <NA> <NA>
  2 United States 2013 Non-Group  13816000  <NA> <NA>
  3 United States 2013 Medicaid   54919100  <NA> <NA>
  4 United States 2013 Medicare   40876300  <NA> <NA>
  5 United States 2013 Other Public 6295400  <NA> <NA>
  6 United States 2013 Uninsured  41795100  <NA> <NA>
  7 United States 2013 Total     313401200 <NA> <NA>
  8 United States 2014 Employer   154347500 <NA> <NA>
  9 United States 2014 Non-Group  19313000  <NA> <NA>
 10 United States 2014 Medicaid   61650400  <NA> <NA>
# ... with 1,446 more rows
```

Perfect! At this point, each row is an observation and each column stores a single piece of information. This dataset is now in good shape!

health care Spending Data

We'll have to take a similar approach when it comes to tidying the spending data as it has a similar structure to how the coverage data were stored.

```

spending
# A tibble: 52 × 25
  Location `1991__Total He... `1992__Total He... `1993__Total He... `1994__Total He...
  <chr>     <dbl>        <dbl>        <dbl>        <dbl>
1 United S...    675896      731455      778684      820172
2 Alabama       10393       11284       12028       12742 
3 Alaska         1458        1558        1661        1728  
4 Arizona        9269       9815       10655       11364 
5 Arkansas       5632       6022       6397        6810  
6 Californ...    81438      87949      91963      94245 
7 Colorado       8460       9215       9803       10382 
8 Connecti...    10950      11635      12081      12772 
9 Delaware       1938       2111       2285       2489  
10 District...   2800        3098       3240       3255 

# ... with 42 more rows, and 20 more variables:
#   1995__Total Health Spending <dbl>, 1996__Total Health Spending <dbl>,
#   1997__Total Health Spending <dbl>, 1998__Total Health Spending <dbl>,
#   1999__Total Health Spending <dbl>, 2000__Total Health Spending <dbl>,
#   2001__Total Health Spending <dbl>, 2002__Total Health Spending <dbl>,
#   2003__Total Health Spending <dbl>, 2004__Total Health Spending <dbl>,
#   2005__Total Health Spending <dbl>, 2006__Total Health Spending <dbl>, ...

```

Here, we reshape the data using `year` and `tot_spending` for the key and value. We also want to keep `Location` like before. Then, in the `separate()` function, we create two new columns called `year` and `name`. Then, we ask to return all the columns, *except* `name`. To select all the columns except a specific column, use the `-` (subtraction) operator. (This process is also referred to as negative indexing.)

```

# take spending data from wide to long
spending <- spending %>%
  pivot_longer(-Location,
               names_to = "year",
               values_to = "tot_spending")

# separate year and name columns
spending <- spending %>%
  separate(year, sep="__",
           into = c("year", "name"),
           convert = TRUE) %>%

```

```
select(-name)

# look at the data
spending
# A tibble: 1,248 × 3
  Location     year tot_spending
  <chr>       <int>      <dbl>
1 United States 1991      675896
2 United States 1992      731455
3 United States 1993      778684
4 United States 1994      820172
5 United States 1995      869578
6 United States 1996      917540
7 United States 1997      969531
8 United States 1998     1026103
9 United States 1999     1086280
10 United States 2000     1162035
# ... with 1,238 more rows
```

Perfect, we have a tidy dataset and the type of information stored in each column is appropriate for the information being stored in the column!

Join the Data

At this point, we have a `coverage` dataset and a `spending` dataset, but ultimately, we want all of this information in a single tidy data frame. To do this, we'll have to join the datasets together.

We have to decide what type of join we want to do. For our questions, we only want information from years that are found in both the `coverage` and the `spending` datasets. This means that we'll want to do an `inner_join()`. This will keep the data from the intersection of years from `coverage` and `spending` (meaning only 2013 and 2014). We'll store this in a new variable: `hc`.

```
# inner join to combine data frames
hc <- inner_join(coverage, spending,
                  by = c("Location", "year"))

hc
# A tibble: 728 × 7
  Location     year type      tot_coverage abb   region tot_spending
  <chr>       <int> <chr>        <int> <chr> <fct>    <dbl>
1 United States 2013 Employer 155696900 <NA>  <NA>    2435624
2 United States 2013 Non-Group 13816000 <NA>  <NA>    2435624
3 United States 2013 Medicaid  54919100 <NA>  <NA>    2435624
4 United States 2013 Medicare  40876300 <NA>  <NA>    2435624
5 United States 2013 Other Public 6295400 <NA>  <NA>    2435624
6 United States 2013 Uninsured 41795100 <NA>  <NA>    2435624
7 United States 2013 Total    313401200 <NA>  <NA>    2435624
8 United States 2014 Employer  154347500 <NA>  <NA>    2562824
9 United States 2014 Non-Group 19313000 <NA>  <NA>    2562824
10 United States 2014 Medicaid  61650400 <NA>  <NA>    2562824
# ... with 718 more rows
```

Great, we've combined the information in our datasets. But, we've got a bit of extraneous information remaining. For example, we want to look only at the state-level. So, let's filter out the country-level summary row:

```
# filter to only include state level
hc <- hc %>%
  filter(Location != "United States")
```

Another problem is that inside our `hc` dataset, there are multiple types of health care coverage.

```
table(hc$type)
```

	Employer	Medicaid	Medicare	Non-Group	Other	Public	Total
	102	102	102	102	102	102	102
Uninsured							
	102						

The “Total” type is not really a formal type of health care coverage. It really represents just the total number of people in the state. This is useful information and we can include it as a column called `tot_pop`. To accomplish this, we’ll first store this information in a data frame called `pop`.

```
pop <- hc %>%
  filter(type == "Total") %>%
  select(Location, year, tot_coverage)

pop
# A tibble: 102 × 3
  Location   year tot_coverage
  <chr>     <int>      <int>
1 Alabama    2013     4763900
2 Alabama    2014     4768000
3 Alaska     2013     702000
4 Alaska     2014     695700
5 Arizona    2013     6603100
6 Arizona    2014     6657200
7 Arkansas   2013     2904800
8 Arkansas   2014     2896000
9 California 2013     38176400
10 California 2014    38701300
# ... with 92 more rows
```

We can then, using a `left_join` to ensure we keep all of the rows in the `hc` data frame in tact, add this population level information while simultaneously removing the rows where type is “Total” from the dataset. Finally, we’ll rename the columns to be informative of the information stored within:

```
# add population level information
hc <- hc %>%
  filter(type != "Total") %>%
  left_join(pop, by = c("Location", "year")) %>%
  rename(tot_coverage = tot_coverage.x,
         tot_pop = tot_coverage.y)

hc
# A tibble: 612 × 8
  Location year type      tot_coverage abb   region tot_spending tot_pop
  <chr>     <int> <chr>        <int> <chr> <fct>    <dbl>    <int>
1 Alabama    2013 Employer      2126500 AL    South     33788 4763900
2 Alabama    2013 Non-Group    174200  AL    South     33788 4763900
3 Alabama    2013 Medicaid     869700  AL    South     33788 4763900
4 Alabama    2013 Medicare     783000  AL    South     33788 4763900
5 Alabama    2013 Other Public 85600   AL    South     33788 4763900
6 Alabama    2013 Uninsured    724800  AL    South     33788 4763900
7 Alabama    2014 Employer     2202800 AL    South     35263 4768000
8 Alabama    2014 Non-Group    288900  AL    South     35263 4768000
9 Alabama    2014 Medicaid     891900  AL    South     35263 4768000
10 Alabama   2014 Medicare    718400  AL    South     35263 4768000
# ... with 602 more rows
```

From here, instead of only storing the absolute number of people who are covered (tot_coverage), we will calculate the proportion of people who are coverage in each state, year and type, storing this information in prop_coverage.

```
# add proportion covered
hc <- hc %>%
  mutate(prop_coverage = tot_coverage/tot_pop)

hc
# A tibble: 612 × 9
  Location year type      tot_coverage abb   region tot_spending tot_pop
  <chr>     <int> <chr>        <int> <chr> <fct>    <dbl>    <int>
1 Alabama    2013 Employer      2126500 AL    South     33788 4763900
2 Alabama    2013 Non-Group    174200  AL    South     33788 4763900
3 Alabama    2013 Medicaid     869700  AL    South     33788 4763900
4 Alabama    2013 Medicare     783000  AL    South     33788 4763900
```

```

5 Alabama 2013 Other Public      85600 AL    South     33788 4763900
6 Alabama 2013 Uninsured       724800 AL    South     33788 4763900
7 Alabama 2014 Employer        2202800 AL   South    35263 4768000
8 Alabama 2014 Non-Group       288900 AL    South    35263 4768000
9 Alabama 2014 Medicaid        891900 AL    South    35263 4768000
10 Alabama 2014 Medicare       718400 AL   South    35263 4768000
# ... with 602 more rows, and 1 more variable: prop_coverage <dbl>

```

The tot_spending column is reported in millions (1e6). Therefore, to calculate spending_capita we will need to adjust for this scaling factor to report it on the original scale (just dollars) and then divide by tot_pop. We can again use mutate() to accomplish this:

```

# get spending capita in dollars
hc <- hc %>%
  mutate(spending_capita = (tot_spending*1e6) / tot_pop)

hc
# A tibble: 612 × 10
  Location year type      tot_coverage abb  region tot_spending tot_pop
  <chr>    <int> <chr>      <int> <chr> <fct>    <dbl>    <int>
  1 Alabama  2013 Employer    2126500 AL    South     33788 4763900
  2 Alabama  2013 Non-Group   174200 AL    South     33788 4763900
  3 Alabama  2013 Medicaid    869700 AL    South     33788 4763900
  4 Alabama  2013 Medicare    783000 AL    South     33788 4763900
  5 Alabama  2013 Other Public 85600 AL    South     33788 4763900
  6 Alabama  2013 Uninsured   724800 AL    South     33788 4763900
  7 Alabama  2014 Employer    2202800 AL   South    35263 4768000
  8 Alabama  2014 Non-Group   288900 AL    South    35263 4768000
  9 Alabama  2014 Medicaid    891900 AL    South    35263 4768000
 10 Alabama 2014 Medicare     718400 AL   South    35263 4768000
# ... with 602 more rows, and 2 more variables: prop_coverage <dbl>,
#   spending_capita <dbl>

```

Yes! At this point we have a single tidy data frame storing all the information we'll need to answer our questions!

Let's save our new tidy data for case study #1.

```
save(hc, file = here::here("data", "tidy_data", "case_study_1_tidy.rda"))
```

Case Study #2: Firearms

For our second case study, we're interested in the following question: At the state-level, what is the relationship between firearm legislation strength and annual rate of fatal police shootings? Time to wrangle *all* those many datasets we read in previously!

Census Data

Let's take a look at the raw data to remind ourselves of what information we have:

```
census
# A tibble: 236,844 × 19
  SUMLEV REGION DIVISION STATE NAME    SEX ORIGIN RACE   AGE CENSUS2010POP
  <chr>    <dbl> <dbl> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
1 040        3      6 01 Alabama  0     0     1     0     37991
2 040        3      6 01 Alabama  0     0     1     1     38150
3 040        3      6 01 Alabama  0     0     1     2     39738
4 040        3      6 01 Alabama  0     0     1     3     39827
5 040        3      6 01 Alabama  0     0     1     4     39353
6 040        3      6 01 Alabama  0     0     1     5     39520
7 040        3      6 01 Alabama  0     0     1     6     39813
8 040        3      6 01 Alabama  0     0     1     7     39695
9 040        3      6 01 Alabama  0     0     1     8     40012
10 040       3      6 01 Alabama  0     0     1     9     42073
# ... with 236,834 more rows, and 9 more variables: ESTIMATESBASE2010 <dbl>,
#   POPESTIMATE2010 <dbl>, POPESTIMATE2011 <dbl>, POPESTIMATE2012 <dbl>,
#   POPESTIMATE2013 <dbl>, POPESTIMATE2014 <dbl>, POPESTIMATE2015 <dbl>,
#   POPESTIMATE2016 <dbl>, POPESTIMATE2017 <dbl>
```

These data look reasonably tidy to start; however, the information stored in each column is not particularly clear at a glance. For example, what is a RACE of 1? What does that mean?

Well, if we look at the data dictionary in the document [sc-est2017-alldata6.pdf](#), we learn that:

The key for SEX is as follows: - 0 = Total - 1 = Male - 2 = Female

The key for ORIGIN is as follows: - 0 = Total - 1 = Not Hispanic - 2 = Hispanic

The key for RACE is as follows: - 1 = White Alone - 2 = Black or African American Alone - 3 = American Indian and Alaska Native Alone - 4 = Asian Alone - 5 = Native Hawaiian and Other Pacific Islander Alone - 6 = Two or more races

With that information in mind, we can then use the `dplyr` package to `filter`, `group_by`, and `summarize` the data in order to calculate the necessary statistics we'll need to answer our question.

For each state, we add rows in the column `POPESTIMATE2015` since we are looking at the year 2015. Setting the `ORIGIN` or `SEX` equal to 0 ensures we don't add duplicate data, since 0 is the key for both Hispanic and non Hispanic residents and total male and female residents. We group by each state since all data in this study should be at the state level.

We store each of these pieces of information in its own column within the new dataframe we've created `census_stats`

```
# summarize by ethnicity
census_stats <- census %>%
  filter(ORIGIN == 0, SEX == 0) %>%
  group_by(NAME) %>%
  summarize(white = sum(POPESTIMATE2015[RACE == 1])/sum(POPESTIMATE2015)*100,
            black = sum(POPESTIMATE2015[RACE == 2])/sum(POPESTIMATE2015)*100)

# add hispanic information
census_stats$hispanic <- census %>%
  filter(SEX == 0) %>%
  group_by(NAME) %>%
  summarize(x = sum(POPESTIMATE2015[ORIGIN == 2])/sum(POPESTIMATE2015[ORIGIN == 0])*100) %>%
  pull(x)

# add male information
census_stats$male <- census %>%
  filter(ORIGIN == 0) %>%
  group_by(NAME) %>%
  summarize(x = sum(POPESTIMATE2015[SEX == 1])/sum(POPESTIMATE2015[SEX == 0])*100) %>%
  pull(x)

# add total population information
```

```

census_stats$total_pop <- census %>%
  filter(ORIGIN == 0, SEX == 0) %>%
  group_by(NAME) %>%
  summarize(total = sum(POPESTIMATE2015)) %>%
  pull(total)

# lowercase state name for consistency
census_stats$NAME <- tolower(census_stats$NAME)

census_stats
# A tibble: 51 × 6
  NAME          white black hispanic male total_pop
  <chr>        <dbl> <dbl>    <dbl> <dbl>    <dbl>
1 alabama      69.5  26.7     4.13  48.5    4850858
2 alaska        66.5   3.67    6.82  52.4    737979
3 arizona       83.5   4.80    30.9   49.7   6802262
4 arkansas      79.6  15.7     7.18  49.1   2975626
5 california    73.0   6.49    38.7   49.7   39032444
6 colorado      87.6   4.47    21.3   50.3   5440445
7 connecticut   80.9  11.6     15.3   48.8   3593862
8 delaware       70.3  22.5     8.96  48.4   944107
9 district of columbia 44.1  48.5     10.7  47.4   672736
10 florida       77.7  16.9     24.7  48.9   20268567
# ... with 41 more rows

```

We can approach the age data similarly, where we get the number of people within each state at each age:

```

# get state-level age information
age_stats <- census %>%
  filter(ORIGIN == 0, SEX == 0) %>%
  group_by(NAME, AGE) %>%
  summarize(sum_ages = sum(POPESTIMATE2015))
`summarise()` has grouped output by 'NAME'. You can override using the `.`groups` argument.

age_stats
# A tibble: 4,386 × 3
# Groups:   NAME [51]
  NAME      AGE sum_ages
  <chr>    <dbl>    <dbl>
1 alabama  0-4        1000
2 alaska    0-4        1000
3 arizona   0-4        1000
4 arkansas  0-4        1000
5 california 0-4        1000
6 colorado  0-4        1000
7 connecticut 0-4        1000
8 delaware  0-4        1000
9 district of columbia 0-4        1000
10 florida  0-4        1000
# ... with 4,382 more rows

```

```

NAME      AGE sum_ages
<chr>    <dbl>   <dbl>
1 Alabama  0     59080
2 Alabama  1     58738
3 Alabama  2     57957
4 Alabama  3     58800
5 Alabama  4     59329
6 Alabama  5     59610
7 Alabama  6     59977
8 Alabama  7     62282
9 Alabama  8     62175
10 Alabama 9     61249
# ... with 4,376 more rows

```

This information is in a long format, but it likely makes more sense to store this information in a wide format, where each column is a different state and each row is an age. To do this:

```

age_stats <- age_stats %>%
  pivot_wider(names_from = "NAME",
              values_from = "sum_ages")

age_stats
# A tibble: 86 × 52
  AGE Alabama Alaska Arizona Arkansas California Colorado Connecticut
  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1     0     59080   11253   86653   38453   500834   66222   36414
2     1     58738   11109   86758   38005   499070   66528   36559
3     2     57957   11009   86713   37711   499614   66144   36887
4     3     58800   10756   86914   38381   498536   67065   37745
5     4     59329   10895   87624   38443   510026   68443   38962
6     5     59610   10537   87234   38582   498754   69823   39182
7     6     59977   10352   89215   38630   497444   69691   39871
8     7     62282   10431   93236   40141   516916   71415   41438
9     8     62175   10302   93866   40677   518117   72384   42359
10    9     61249   10055   92531   39836   511610   72086   43032
# ... with 76 more rows, and 44 more variables: Delaware <dbl>,
#   District of Columbia <dbl>, Florida <dbl>, Georgia <dbl>, Hawaii <dbl>,
#   Idaho <dbl>, Illinois <dbl>, Indiana <dbl>, Iowa <dbl>, Kansas <dbl>,
#   Kentucky <dbl>, Louisiana <dbl>, Maine <dbl>, Maryland <dbl>,
#   Massachusetts <dbl>, Michigan <dbl>, Minnesota <dbl>, Missouri <dbl>,
#   Montana <dbl>, Nebraska <dbl>, Nevada <dbl>, New Hampshire <dbl>,
#   New Jersey <dbl>, New Mexico <dbl>, New York <dbl>, North Carolina <dbl>,
#   North Dakota <dbl>, Ohio <dbl>, Oklahoma <dbl>, Oregon <dbl>, Pennsylvania <dbl>,
#   Rhode Island <dbl>, South Carolina <dbl>, South Dakota <dbl>, Texas <dbl>,
#   Utah <dbl>, Vermont <dbl>, Washington <dbl>, West Virginia <dbl>,
#   Wisconsin <dbl>, Wyoming <dbl>

```

```
#  Massachusetts <dbl>, Michigan <dbl>, Minnesota <dbl>, Mississippi <dbl>,
#  Missouri <dbl>, Montana <dbl>, Nebraska <dbl>, Nevada <dbl>,
#  New Hampshire <dbl>, New Jersey <dbl>, New Mexico <dbl>, New York <dbl>, ...
```

Now that we've made the data easier to work with, we need to find a way to get the median. One method is to take the cumulative sum of each column and then divide all the rows by the last row in each respective column, calculating a percentile/quantile for each age. To do this, we first remove the AGE column, as we don't want to calculate the median for this column. We then apply the `cumsum()` function and an anonymous function using `purrr`'s `map_dfc` function. This is a special variation of the `map()` function that returns a data frame instead of a list by combining the data by column. But, of course, we do still want the AGE information in there, so we add that column back in using `mutate()` and then reorder the columns so that AGE is at the front again using `select()`.

First let's see what would happen if we used `map()` instead of `map_dfc()`:

```
age_stats %>%
  select(-AGE) %>%
  map(cumsum) %>%
  map(function(x) x/x[nrow(age_stats)]) %>%
  glimpse
List of 51
 $ Alabama      : num [1:86] 0.0122 0.0243 0.0362 0.0484 0.0606 ...
 $ Alaska       : num [1:86] 0.0152 0.0303 0.0452 0.0598 0.0746 ...
 $ Arizona      : num [1:86] 0.0127 0.0255 0.0382 0.051 0.0639 ...
 $ Arkansas     : num [1:86] 0.0129 0.0257 0.0384 0.0513 0.0642 ...
 $ California   : num [1:86] 0.0128 0.0256 0.0384 0.0512 0.0643 ...
 $ Colorado     : num [1:86] 0.0122 0.0244 0.0366 0.0489 0.0615 ...
 $ Connecticut  : num [1:86] 0.0101 0.0203 0.0306 0.0411 0.0519 ...
 $ Delaware     : num [1:86] 0.0116 0.0233 0.0348 0.0466 0.0587 ...
 $ District of Columbia: num [1:86] 0.0146 0.0276 0.0407 0.0533 0.0657 ...
 $ Florida      : num [1:86] 0.011 0.0219 0.0328 0.0437 0.0547 ...
 $ Georgia      : num [1:86] 0.0128 0.0256 0.0384 0.0514 0.0647 ...
 $ Hawaii       : num [1:86] 0.0128 0.0258 0.039 0.0519 0.065 ...
 $ Idaho        : num [1:86] 0.0139 0.0274 0.0413 0.0549 0.069 ...
 $ Illinois     : num [1:86] 0.0123 0.0245 0.0366 0.0488 0.0611 ...
 $ Indiana      : num [1:86] 0.0126 0.0253 0.038 0.0507 0.0635 ...
 $ Iowa         : num [1:86] 0.0127 0.0254 0.038 0.0506 0.063 ...
 $ Kansas        : num [1:86] 0.0133 0.0268 0.0404 0.054 0.0677 ...
```

```
$ Kentucky          : num [1:86] 0.0126 0.0251 0.0376 0.05 0.0624 ...
$ Louisiana         : num [1:86] 0.0137 0.0272 0.0405 0.0536 0.0667 ...
$ Maine             : num [1:86] 0.00949 0.01906 0.02881 0.0386 0.04839 ...
$ Maryland           : num [1:86] 0.0123 0.0245 0.0367 0.049 0.0614 ...
$ Massachusetts      : num [1:86] 0.0106 0.0213 0.0319 0.0427 0.0536 ...
$ Michigan            : num [1:86] 0.0114 0.023 0.0345 0.046 0.0577 ...
$ Minnesota          : num [1:86] 0.0127 0.0256 0.0383 0.0511 0.0639 ...
$ Mississippi         : num [1:86] 0.0128 0.0255 0.0382 0.0512 0.0641 ...
$ Missouri            : num [1:86] 0.0123 0.0248 0.0371 0.0494 0.0618 ...
$ Montana             : num [1:86] 0.0123 0.0244 0.0364 0.0484 0.0604 ...
$ Nebraska            : num [1:86] 0.0141 0.0281 0.042 0.0557 0.0696 ...
$ Nevada              : num [1:86] 0.0125 0.0248 0.0374 0.0498 0.0626 ...
$ New Hampshire       : num [1:86] 0.00932 0.01866 0.02852 0.03814 0.04831 ...
$ New Jersey          : num [1:86] 0.0115 0.0232 0.0349 0.0468 0.0589 ...
$ New Mexico           : num [1:86] 0.0124 0.025 0.0376 0.0504 0.0634 ...
$ New York             : num [1:86] 0.0122 0.0241 0.036 0.0478 0.0598 ...
$ North Carolina       : num [1:86] 0.012 0.024 0.0359 0.0479 0.06 ...
$ North Dakota         : num [1:86] 0.015 0.0296 0.0438 0.0576 0.0709 ...
$ Ohio                 : num [1:86] 0.012 0.024 0.036 0.048 0.0601 ...
$ Oklahoma             : num [1:86] 0.0136 0.0272 0.0409 0.0546 0.0683 ...
$ Oregon               : num [1:86] 0.0114 0.0229 0.0344 0.046 0.0577 ...
$ Pennsylvania         : num [1:86] 0.0111 0.0222 0.0333 0.0445 0.0558 ...
$ Rhode Island          : num [1:86] 0.0103 0.0205 0.0309 0.0413 0.0518 ...
$ South Carolina        : num [1:86] 0.0118 0.0236 0.0354 0.0474 0.0594 ...
$ South Dakota          : num [1:86] 0.0144 0.029 0.0433 0.0575 0.0714 ...
$ Tennessee            : num [1:86] 0.0123 0.0245 0.0367 0.049 0.0612 ...
$ Texas                : num [1:86] 0.0146 0.0292 0.0435 0.0578 0.0724 ...
$ Utah                 : num [1:86] 0.0171 0.034 0.051 0.0676 0.0846 ...
$ Vermont              : num [1:86] 0.0096 0.0195 0.0292 0.039 0.0489 ...
$ Virginia             : num [1:86] 0.0123 0.0245 0.0367 0.0489 0.0612 ...
$ Washington            : num [1:86] 0.0124 0.0248 0.0373 0.0497 0.0623 ...
$ West Virginia         : num [1:86] 0.0108 0.0219 0.0331 0.0443 0.0554 ...
$ Wisconsin            : num [1:86] 0.0116 0.0232 0.0349 0.0467 0.0586 ...
$ Wyoming              : num [1:86] 0.0132 0.0262 0.0392 0.0523 0.0655 ...
```

We can see that we create a list of vectors for each state.

Now let's use `map_dfc()`:

```
# calculate median age for each state
age_stats <- age_stats %>%
  select(-AGE) %>%
  map_dfc(cumsum) %>%
  map_dfc(function(x) x/x[nrow(age_stats)]) %>%
  mutate(AGE = age_stats$AGE) %>%
  select(AGE, everything())

glimpse(age_stats)
Rows: 86
Columns: 52
$ AGE          <dbl> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1...
$ Alabama      <dbl> 0.01217929, 0.02428807, 0.03623586, 0.04835742, ...
$ Alaska        <dbl> 0.01524840, 0.03030168, 0.04521944, 0.05979438, ...
$ Arizona       <dbl> 0.01273885, 0.02549314, 0.03824081, 0.05101803, ...
$ Arkansas      <dbl> 0.01292266, 0.02569476, 0.03836806, 0.05126652, ...
$ California    <dbl> 0.01283122, 0.02561725, 0.03841722, 0.05118957, ...
$ Colorado      <dbl> 0.01217217, 0.02440058, 0.03655841, 0.04888552, ...
$ Connecticut   <dbl> 0.01013228, 0.02030490, 0.03056879, 0.04107142, ...
$ Delaware      <dbl> 0.01163322, 0.02326537, 0.03477678, 0.04661654, ...
$ `District of Columbia` <dbl> 0.01457035, 0.02759032, 0.04071434, 0.05331661, ...
$ Florida        <dbl> 0.01095628, 0.02192577, 0.03277035, 0.04371311, ...
$ Georgia        <dbl> 0.01284902, 0.02562823, 0.03837038, 0.05144559, ...
$ Hawaii         <dbl> 0.01275029, 0.02580557, 0.03896040, 0.05191402, ...
$ Idaho          <dbl> 0.01390752, 0.02744397, 0.04125812, 0.05494312, ...
$ Illinois       <dbl> 0.01233645, 0.02447549, 0.03656625, 0.04878623, ...
$ Indiana        <dbl> 0.01264485, 0.02525899, 0.03798841, 0.05071888, ...
$ Iowa           <dbl> 0.01267030, 0.02538518, 0.03802630, 0.05063696, ...
$ Kansas          <dbl> 0.01334268, 0.02681406, 0.04039729, 0.05398568, ...
$ Kentucky       <dbl> 0.01257763, 0.02509149, 0.03759246, 0.05002287, ...
$ Louisiana      <dbl> 0.01366990, 0.02722998, 0.04047237, 0.05358525, ...
$ Maine          <dbl> 0.00949098, 0.01906255, 0.02881486, 0.03860408, ...
$ Maryland        <dbl> 0.01233418, 0.02449538, 0.03670807, 0.04897542, ...
$ Massachusetts  <dbl> 0.01064012, 0.02127391, 0.03194332, 0.04267411, ...
$ Michigan        <dbl> 0.01142560, 0.02298902, 0.03448267, 0.04603339, ...
$ Minnesota      <dbl> 0.01271147, 0.02555680, 0.03831076, 0.05107128, ...
$ Mississippi    <dbl> 0.01277126, 0.02554419, 0.03822836, 0.05115873, ...
$ Missouri        <dbl> 0.01232034, 0.02475678, 0.03708173, 0.04938873, ...
$ Montana         <dbl> 0.01225303, 0.02440298, 0.03639442, 0.04838391, ...
```

```
$ Nebraska <dbl> 0.01411096, 0.02814111, 0.04197798, 0.05572191, ...
$ Nevada <dbl> 0.01250131, 0.02483232, 0.03739746, 0.04979506, ...
$ `New Hampshire` <dbl> 0.009324624, 0.018656767, 0.028516676, 0.038139...
$ `New Jersey` <dbl> 0.01153672, 0.02318527, 0.03489888, 0.04677209, ...
$ `New Mexico` <dbl> 0.01241485, 0.02496946, 0.03762395, 0.05035577, ...
$ `New York` <dbl> 0.01217189, 0.02407385, 0.03599014, 0.04784986, ...
$ `North Carolina` <dbl> 0.01200077, 0.02400872, 0.03589836, 0.04789096, ...
$ `North Dakota` <dbl> 0.01498293, 0.02957241, 0.04375387, 0.05755379, ...
$ Ohio <dbl> 0.01195482, 0.02398530, 0.03604946, 0.04804797, ...
$ Oklahoma <dbl> 0.01360302, 0.02720220, 0.04094481, 0.05455859, ...
$ Oregon <dbl> 0.01137846, 0.02290605, 0.03443489, 0.04598165, ...
$ Pennsylvania <dbl> 0.01105806, 0.02218984, 0.03332365, 0.04452838, ...
$ `Rhode Island` <dbl> 0.01029817, 0.02052152, 0.03085757, 0.04130916, ...
$ `South Carolina` <dbl> 0.01182747, 0.02361386, 0.03537920, 0.04736508, ...
$ `South Dakota` <dbl> 0.01441743, 0.02898356, 0.04325579, 0.05748470, ...
$ Tennessee <dbl> 0.01234189, 0.02454161, 0.03672630, 0.04900204, ...
$ Texas <dbl> 0.01460720, 0.02916560, 0.04354217, 0.05781300, ...
$ Utah <dbl> 0.01705642, 0.03397314, 0.05104564, 0.06758513, ...
$ Vermont <dbl> 0.009603574, 0.019524225, 0.029198261, 0.039019...
$ Virginia <dbl> 0.01229627, 0.02446226, 0.03665131, 0.04888352, ...
$ Washington <dbl> 0.01241119, 0.02476856, 0.03725623, 0.04972054, ...
$ `West Virginia` <dbl> 0.01083507, 0.02189516, 0.03312702, 0.04427952, ...
$ Wisconsin <dbl> 0.01158854, 0.02323645, 0.03492551, 0.04674149, ...
$ Wyoming <dbl> 0.01320760, 0.02620022, 0.03920990, 0.05229295, ...
```

Great, we have a tidy dataframe with a column for each state storing important census information for both ethnicity and age. Now onto the other datasets!

Violent Crime

For crime, we have the following data:

```

crime
# A tibble: 510 × 14
  State    Area      ...3    Population `Violent\ncrime... `Murder and \nnonne...
  <chr>   <chr>    <chr>    <chr>           <dbl>                <dbl>
1 ALABAMA Metropoli... <NA>    3708033          NA                  NA
2 <NA>     <NA>     Area ac... 0.97099999...    18122               283
3 <NA>     <NA>     Estimat... 1            18500               287
4 <NA>     Cities ou... <NA>    522241           NA                  NA
5 <NA>     <NA>     Area ac... 0.97399999...    3178                32
6 <NA>     <NA>     Estimat... 1            3240                33
7 <NA>     Nonmetrop... <NA>    628705           NA                  NA
8 <NA>     <NA>     Area ac... 0.99399999...    1205                28
9 <NA>     <NA>     Estimat... 1            1212                28
10 <NA>    State Tot... <NA>    4858979       22952               348
# ... with 500 more rows, and 8 more variables: Rape
(revised
definition)2 <dbl>,
#   Rape
(legacy
definition)3 <dbl>, Robbery <dbl>, Aggravated
assault <dbl>,
#   Property
crime <dbl>, Burglary <dbl>, Larceny-
theft <dbl>,
#   Motor
vehicle
theft <dbl>

```

If we take a look at what information is stored in each column...

```
colnames(crime)
[1] "State"
[2] "Area"
[3] "...3"
[4] "Population"
[5] "Violent\ncrime1"
[6] "Murder and \nnonnegligent \nmanslaughter"
[7] "Rape\n(revised\ndefinition)2"
[8] "Rape\n(legacy\ndefinition)3"
[9] "Robbery"
[10] "Aggravated \nassault"
[11] "Property \ncrime"
[12] "Burglary"
[13] "Larceny-\ntheft"
[14] "Motor \nvehicle \ntheft"
```

you see that it's kind of a mess and there's a whole bunch of information in there that we're not necessarily interested in for this analysis.

Because of the messy names here (we'll clean them up in a bit), we'll see the column index to select columns instead of the complicated names. Also, we print a specified row of violent crime to observe the `x__1` group we are looking for – Rate per 100,000 inhabitants (per the study.)

```
violentcrime <- crime %>%
  select(c(1,3,5))

violentcrime
# A tibble: 510 × 3
  State    ...3          `Violent\ncrime1` 
  <chr>   <chr>         <dbl>    
1 ALABAMA <NA>           NA      
2 <NA>     Area actually reporting 18122  
3 <NA>     Estimated total      18500   
4 <NA>     <NA>             NA      
5 <NA>     Area actually reporting 3178    
6 <NA>     Estimated total      3240    
7 <NA>     <NA>             NA      
8 <NA>     Area actually reporting 1205
```

```

9 <NA>     Estimated total          1212
10 <NA>      <NA>                  22952
# ... with 500 more rows

```

Great, so we're starting to home in on the data we're interested in but we're ultimately interested in Rate per 100,000 inhabitants, so we need get all rows where the second column is equal to Rate per 100,000 inhabitants.

However, as we can see above, the value for State in these rows is NA, so we need to `fill()` that value with the state name that is listed in a previous row. Then we can select the rows where the second column is Rate per 100,000 inhabitants. After that, we no longer need the second column, so we'll remove it.

```

violentcrime <- violentcrime %>%
  fill(State) %>%
  filter(.[[2]] == "Rate per 100,000 inhabitants") %>%
  rename( violent_crime = `Violent\ncrime1` ) %>%
  select(-`...3`)

```

	State	violent_crime
1	ALABAMA	472.
2	ALASKA	730.
3	ARIZONA	410.
4	ARKANSAS	521.
5	CALIFORNIA	426.
6	COLORADO	321
7	CONNECTICUT	218.
8	DELAWARE	499
9	DISTRICT OF COLUMBIA	1269.
10	FLORIDA	462.

```
# ... with 42 more rows
```

If we look closely at our data, we'll notice that some of our state names have 6s at the end of them. This will cause problems later on.

```
violentcrime$State[20]  
[1] "MAINE6"
```

So, let's clean that up now by removing those trailing numeric values *and* converting the names to lower case:

```
# lower case and remove numbers from State column  
violentcrime <- violentcrime %>%  
  mutate(State = tolower(gsub('[0-9]+', '', State)))  
  
violentcrime  
# A tibble: 52 × 2  
  State          violent_crime  
  <chr>           <dbl>  
1 alabama        472.  
2 alaska         730.  
3 arizona        410.  
4 arkansas       521.  
5 california     426.  
6 colorado       321  
7 connecticut    218.  
8 delaware        499  
9 district of columbia 1269.  
10 florida        462.  
# ... with 42 more rows
```

We've now got ourselves a tidy dataset with violent crime information that's ready to be joined with our census_stats data!

```
# join with census data
firearms <- left_join(census_stats, violentcrime,
                      by = c("NAME" = "State"))

firearms
# A tibble: 51 × 7
  NAME      white black hispanic male total_pop violent_crime
  <chr>    <dbl> <dbl>   <dbl> <dbl>    <dbl>        <dbl>
1 alabama     69.5  26.7    4.13  48.5  4850858       472.
2 alaska       66.5   3.67    6.82  52.4  737979        730.
3 arizona      83.5   4.80   30.9   49.7  6802262       410.
4 arkansas     79.6  15.7    7.18  49.1  2975626       521.
5 california   73.0   6.49   38.7   49.7  39032444      426.
6 colorado     87.6   4.47   21.3   50.3  5440445        321
7 connecticut   80.9  11.6    15.3   48.8  3593862       218.
8 delaware      70.3  22.5    8.96  48.4  944107        499
9 district of columbia 44.1  48.5   10.7   47.4  672736       1269.
10 florida      77.7  16.9   24.7   48.9  20268567      462.
# ... with 41 more rows
```

Brady Scores

The study by AJPH groups the scores using 7 different categories. The study removed all weightings of the different laws in favor of a “1 law 1 point” system, since the weightings were “somewhat arbitrary.”

For the purpose of practice and simplification we will just keep the first line of “total state points” from the Brady Scorecard as they are given. This will be where our analysis differs from the study. We need to transform the data frame so that we have a column of state names and a column of the corresponding total scores.

```
brady
# A tibble: 116 × 54
`States can recei...` `Category Point...` `Sub Category P...` Points AL     AK     AR
<chr>                <dbl>           <dbl>   <dbl> <chr> <chr> <chr>
1 TOTAL STATE POINTS    NA             NA      NA -18   -30   -24
2 CATEGORY 1: KEEP...     50            NA      NA <NA> <NA> <NA>
3 BACKGROUND CHECKS...    NA             25     NA AL     AK     AR
4 Background Checks...    NA             NA      NA 25 <NA> <NA> <NA>
5 Background Checks...    NA             NA      NA 20 <NA> <NA> <NA>
6 Background Checks...    NA             NA      NA  5 <NA> <NA> <NA>
7 Verifiy Legal Pur...    NA             NA      NA 20 <NA> <NA> <NA>
8 TOTAL                  NA             NA      NA  0    0    0
9 <NA>
10 OTHER LAWS TO STO...   NA             12     NA AL     AK     AR
# ... with 106 more rows, and 47 more variables: AZ <chr>, CA <chr>, CO <chr>,
# CT <chr>, DE <chr>, FL <chr>, GA <chr>, HI <chr>, ID <chr>, IL <chr>,
# IN <chr>, IA <chr>, KS <chr>, KY <chr>, LA <chr>, MA <chr>, MD <chr>,
# ME <chr>, MI <chr>, MN <chr>, MO <chr>, MT <chr>, MS <chr>, NC <chr>,
# ND <chr>, NE <chr>, NH <chr>, NJ <chr>, NM <chr>, NV <chr>, NY <chr>,
# OK <chr>, OH <chr>, OR <chr>, PA <chr>, RI <chr>, SC <chr>, SD <chr>,
# TN <chr>, TX <chr>, UT <chr>, VA <chr>, VT <chr>, WA <chr>, WI <chr>, ...
```

This dataset includes a lot of information, but we’re interested in the brady scores for each state. These are stored in the row where the first column is equal to “TOTAL STATE POINTS,” so we `filter()` to only include that row. We then want to only receive the scores for each state, and not the information in the first few columns, so we specify that using `select()`. With the information we’re interested in, we then take the data from wide to long using `pivot_longer()`, renaming the columns as we go. Finally, we specify that the information in the `brady_scores` column is numeric, not a character.

```
brady <- brady %>%
  rename(Law = `States can receive a maximum of 100 points`) %>%
  filter(Law == "TOTAL STATE POINTS") %>%
  select((ncol(brady) - 49):ncol(brady)) %>%
  pivot_longer(everything(),
               names_to = "state",
               values_to = "brady_scores") %>%
  mutate_at("brady_scores", as.numeric)

brady
# A tibble: 50 × 2
  state    brady_scores
  <chr>      <dbl>
1 AL          -18
2 AK          -30
3 AR          -24
4 AZ          -39
5 CA           76
6 CO           22
7 CT           73
8 DE           41
9 FL          -20.5
10 GA          -18
# ... with 40 more rows
```

Only problem now is that we have the two letter state code, rather than the full state name we've been joining on so far here. We can, however, use the state datasets we used in the first case study here!

```
brady <- brady %>%
  left_join(rename(state_data, state = abb),
            by = "state") %>%
  select(Location, brady_scores) %>%
  rename(state = Location) %>%
  mutate(state = tolower(state))
```

```
brady
# A tibble: 50 × 2
  state    brady_scores
```

```

<chr>      <dbl>
1 alabama    -18
2 alaska     -30
3 arkansas   -24
4 arizona    -39
5 california 76
6 colorado   22
7 connecticut 73
8 delaware   41
9 florida    -20.5
10 georgia   -18
# ... with 40 more rows

```

Now, it's time to join this information into our growing dataframe `firearms`:

```

firearms <- left_join(firearms, brady, by = c("NAME" = "state"))

firearms
# A tibble: 51 × 8
  NAME          white black hispanic male total_pop violent_crime brad\
y_scores
  <chr>      <dbl> <dbl>    <dbl> <dbl>    <dbl>        <dbl>    \
  <dbl>
1 alabama    69.5 26.7     4.13  48.5  4850858     472.    \
-18
2 alaska     66.5 3.67     6.82  52.4  737979      730.    \
-30
3 arizona    83.5 4.80     30.9   49.7  6802262     410.    \
-39
4 arkansas   79.6 15.7     7.18   49.1  2975626     521.    \
-24
5 california 73.0 6.49     38.7   49.7  39032444    426.    \
76
6 colorado   87.6 4.47     21.3   50.3  5440445      321    \
22
7 connecticut 80.9 11.6    15.3   48.8  3593862     218.    \
73
8 delaware   70.3 22.5     8.96   48.4  944107      499    \
41

```

```

9 district of columbia 44.1 48.5      10.7   47.4    672736      1269.     \
NA
10 florida            77.7 16.9      24.7   48.9  20268567      462.     \
-20.5
# ... with 41 more rows

```

The Counted Fatal Shootings

We're making progress, but we have a ways to go still! Let's get working on incorporating data from [The Counted](#).

As a reminder, we have a datasets here with data from 2015:

```

counted15
# A tibble: 1,146 × 6
  gender raceethnicity state classification lawenforcementage... armed
  <chr>  <chr>        <chr>  <chr>                <chr>                    <chr>
1 Male   Black         GA     Death in custo... Chatham County Sh... No
2 Male   White         OR     Gunshot          Washington County... Firea...
3 Male   White         HI     Struck by vehi... Kauai Police Depa... No
4 Male   Hispanic/Latino KS     Gunshot          Wichita Police De... No
5 Male   Asian/Pacific Islander WA     Gunshot          Mason County Sher... Firea...
6 Male   White         CA     Gunshot          San Francisco Pol... Non-l...
7 Male   Hispanic/Latino AZ     Gunshot          Chandler Police D... Firea...
8 Male   Hispanic/Latino CO     Gunshot          Evans Police Depa... Other
9 Male   White         CA     Gunshot          Stockton Police D... Knife
10 Male  Black          CA     Taser            Los Angeles Count... No
# ... with 1,136 more rows

```

The data from each year are in a similar format with each row representing a different individual and the columns being consistent between the two datasets.

Because of this consistent format, we can combine these two datasets using `bind_rows()`. By specifying `id = "dataset"`, a column called `dataset` will store which dataset each row came from originally. We can then use `mutate()` and `ifelse()` to conditionally specify the year – 2015 or 2016 – from which the data originated. We'll also be sure to change the two letter state abbreviation to the lower case state name, to allow for each merging.

```
counted15 <- counted15 %>%
  mutate(state = tolower(state.name[match(state, state.abb)]))
```

At this point, we have a lot of information at the individual level, but we'd like to summarize this at the state level by ethnicity, gender, and armed status. The researchers "calculated descriptive statistics for the proportion of victims that were male, armed, and non-White," so we'll do the same. We can accomplish this using dplyr. The tally() function will be particularly helpful here to count the number of observations in each group. We're calculating this for each state as well as calculating the annualized rate per 1,000,000 residents. This utilizes the total_pop column from the census_stats data frame we used earlier.

```
# get overall stats
counted_stats <- counted15 %>%
  group_by(state) %>%
  filter(classification == "Gunshot") %>%
  tally() %>%
  rename("gunshot_tally" = "n")

# get summary for subset of population
gunshot_filtered <- counted15 %>%
  group_by(state) %>%
  filter(classification == "Gunshot", raceethnicity != "white", armed != "No", gender == "Male") %>%
  tally() %>%
  rename("gunshot_filtered" = "n")

# join data together
counted_stats <- left_join(counted_stats, gunshot_filtered, by = "state") %>%
  mutate(total_pop = census_stats$total_pop[match(state, census_stats$NAME)],
        gunshot_rate = (gunshot_tally/total_pop)*1000000/2) %>%
  select(-total_pop)

counted_stats
# A tibble: 50 × 4
  state      gunshot_tally gunshot_filtered gunshot_rate
  <chr>          <int>           <int>        <dbl>
1 alabama         18              15       1.86
2 alaska            4              4       2.71
3 arizona          43             37       3.16
```

```

4 arkansas          5          4    0.840
5 california       196        150   2.51
6 colorado         29         27    2.67
7 connecticut      2          2    0.278
8 delaware         3          2    1.59
9 district of columbia 5          4    3.72
10 florida        64         54   1.58
# ... with 40 more rows

```

Time to merge this into the data frame we've been compiling:

```
firearms <- left_join(firearms, counted_stats, by = c("NAME" = "state"))
```

Unemployment Data

Let's recall the table we scraped from the web, which is currently storing our unemployment data:

```

unemployment
# A tibble: 54 × 3
  State      `2015rate` Rank
  <chr>      <chr>     <chr>
1 "United States" "5.3"    ""
2 ""          ""        ""
3 "North Dakota"  "2.8"    "1"
4 "Nebraska"     "3.0"    "2"
5 "South Dakota" "3.1"    "3"
6 "New Hampshire" "3.4"    "4"
7 "Hawaii"       "3.6"    "5"
8 "Utah"         "3.6"    "5"
9 "Vermont"      "3.6"    "5"
10 "Minnesota"    "3.7"   "8"
# ... with 44 more rows

```

Let's first rename the columns to clean things up. You'll note that there are more rows in this data frame (due to an empty row, the United States, and a note being in this dataset); however, when we `left_merge()` in just a second these will disappear, so we can ignore them for now.

```

unemployment <- unemployment %>%
  rename("state" = "State",
         "unemployment_rate" = "2015rate",
         "unemployment_rank" = "Rank") %>%
  mutate(state = tolower(state)) %>%
  arrange(state)

unemployment
# A tibble: 54 × 3
  state      unemployment_rate  unemployment_rank
  <chr>        <chr>            <chr>
1 ""          ""                ""
2 "alabama"   "6.1"             "42"
3 "alaska"    "6.5"             "47"
4 "arizona"   "6.1"             "42"
5 "arkansas"  "5.0"             "24"
6 "california" "6.2"            "44"
7 "colorado"  "3.9"             "10"
8 "connecticut" "5.7"            "35"
9 "delaware"   "4.9"             "22"
10 "district of columbia" "6.9"            "51"
# ... with 44 more rows

```

Let's do that join now. Let's add unemployment information to our growing data frame!

```
firearms <- left_join(firearms, unemployment, by = c("NAME" = "state"))
```

If we take a look at the data we now have in our growing data frame, using `glimpse()`, we see that type is correct for most of our variables *except* `unemployment_rate` and `unemployment_rank`. This is due to that “Note” and empty (“”) row in the `unemployment` dataset. So, let’s be sure to get that variable to a numeric now as it should be:

```
glimpse(firearms)
Rows: 51
Columns: 13
$ NAME          <chr> "alabama", "alaska", "arizona", "arkansas", "califor...
$ white         <dbl> 69.50197, 66.51368, 83.52295, 79.57623, 72.97555, 87...
$ black         <dbl> 26.7459489, 3.6679906, 4.7978011, 15.6634268, 6.4910...
$ hispanic      <dbl> 4.129434, 6.821197, 30.873010, 7.180439, 38.727129, ...
$ male          <dbl> 48.46650, 52.36978, 49.71595, 49.12855, 49.67815, 50...
$ total_pop     <dbl> 4850858, 737979, 6802262, 2975626, 39032444, 5440445...
$ violent_crime <dbl> 472.4, 730.2, 410.2, 521.3, 426.3, 321.0, 218.5, 499...
$ brady_scores   <dbl> -18.0, -30.0, -39.0, -24.0, 76.0, 22.0, 73.0, 41.0, ...
$ gunshot_tally   <int> 18, 4, 43, 5, 196, 29, 2, 3, 5, 64, 29, 2, 7, 22, 19...
$ gunshot_filtered <int> 15, 4, 37, 4, 150, 27, 2, 2, 4, 54, 25, 2, 7, 21, 15...
$ gunshot_rate    <dbl> 1.8553419, 2.7101042, 3.1607133, 0.8401593, 2.510731...
$ unemployment_rate <chr> "6.1", "6.5", "6.1", "5.0", "6.2", "3.9", "5.7", "4...
$ unemployment_rank <chr> "42", "47", "42", "24", "44", "10", "35", "22", "51"...
```



```
# convert type for unemployment columns
firearms <- firearms %>%
  mutate_at("unemployment_rate", as.numeric) %>%
  mutate_at("unemployment_rank", as.integer)
```

Population Density: 2015

Population density for 2015 can be calculated from the Census data in combination with the land area data we've read in. This is calculated (rather than simply imported) because accurate data for state population in 2015 was not available in a downloadable format nor was it easy to scrape.

From the census data, we can obtain total population counts:

```

totalPop <- census %>%
  filter(ORIGIN == 0, SEX == 0) %>%
  group_by(NAME) %>%
  summarize(total = sum(POPESTIMATE2015)) %>%
  mutate(NAME = tolower(NAME))

totalPop
# A tibble: 51 × 2
  NAME              total
  <chr>            <dbl>
1 alabama        4850858
2 alaska          737979
3 arizona         6802262
4 arkansas        2975626
5 california     39032444
6 colorado        5440445
7 connecticut    3593862
8 delaware        944107
9 district of columbia 672736
10 florida       20268567
# ... with 41 more rows

```

Then, we select `LND110210D` by looking at the `land` table and comparing values on other sites (such as the census or Wikipedia) to find the correct column. This column corresponds to land area in square miles. We'll convert all state names to lower case for easy merging with our growing data frame in a few steps.

```

landSqMi <- land %>%
  select(Areaname, land_area = LND110210D) %>%
  mutate(Areaname = tolower(Areaname))

landSqMi
# A tibble: 3,198 × 2
  Areaname      land_area
  <chr>           <dbl>
1 united states  3531905.
2 alabama        50645.
3 autauga, al     594.
4 baldwin, al     1590.

```

```

5 barbour, al      885.
6 babb, al        623.
7 blount, al      645.
8 bullock, al     623.
9 butler, al      777.
10 calhoun, al    606.
# ... with 3,188 more rows

```

Since `landSqMi` gives us area for each town in addition to the states, we will want merge on the state names to obtain only the area for each state, removing the city- and nation-level data. Also, because “district of columbia” appears twice, we’ll use the `distinct()` function to only include one entry for “district of columbia”

We can then calculate density and remove the `total` and `land_area` columns to only keep state name and density for each state:

```

popdensity <- left_join(totalPop, landSqMi, by=c("NAME" = "Areaname")) %>%
  distinct() %>%
  mutate(density = total/land_area) %>%
  select(-c(total, land_area))

```

```

popdensity
# A tibble: 51 × 2
  NAME            density
  <chr>          <dbl>
1 alabama        95.8 
2 alaska         1.29  
3 arizona        59.9 
4 arkansas       57.2 
5 california     251.  
6 colorado       52.5 
7 connecticut    742.  
8 delaware       485. 
9 district of columbia 11019.
10 florida        378. 
# ... with 41 more rows

```

This can now be joined with our growing data frame:

```
firearms <- left_join(firearms, popdensity, by="NAME")
```

Firearm Ownership

Last but not least, we calculate firearm ownership as a percent of firearm suicides to all suicides.

```
ownership_df <- as_tibble(list("NAME" = tolower(suicide_all$State),
                               "ownership" = suicide_firearm$Deaths/suicide_all$Deat\hs*100))
ownership_df
# A tibble: 51 × 2
  NAME      ownership
  <chr>     <dbl>
1 alabama    70.1
2 alaska      59.9
3 arizona     57.4
4 arkansas    59.5
5 california   37.3
6 colorado     51.0
7 connecticut  27.4
8 delaware     49.8
9 florida      52.0
10 georgia     62.1
# ... with 41 more rows
```

This can now be joined onto our tidy data frame:

```
firearms <- left_join(firearms, ownership_df, by="NAME")
```

And, with that, we've wrangled and tidied all these datasets into a single data frame. This can now be used for visualization and analysis!

Let's save our new tidy data for case study #2.

```
save(firearms, file = here::here("data", "tidy_data", "case_study_2_tidy.rda"))
```

4. Visualizing Data in the Tidyverse

About This Course

Data visualization is a critical part of any data science project. Once data have been imported and wrangled into place, visualizing your data can help you get a handle on what's going on in the dataset. Similarly, once you've completed your analysis and are ready to present your findings, data visualizations are a highly effective way to communicate your results to others. In this course we will cover what data visualization is and define some of the basic types of data visualizations.

In this course you will learn about the `ggplot2` R package, a powerful set of tools for making stunning data graphics that has become the industry standard. You will learn about different types of plots, how to construct effect plots, and what makes for a successful or unsuccessful visualization.

In this book we assume familiarity with the R programming language. If you are not yet familiar with R, we suggest you first complete [R Programming](#) before returning to complete this course.

Data Visualization Background

At its core, the term 'data visualization' refers to any visual display of data that helps us understand the underlying data better. This can be a plot or figure of some sort or a table that summarizes the data. Generally, there are a few characteristics of all good plots.

General Features of Plots

Good plots have a number of features. While not exhaustive, good plots have:

1. Clearly-labeled axes.
2. Text that are large enough to see.
3. Axes that are not misleading.

4. Data that are displayed appropriately considering the type of data you have.

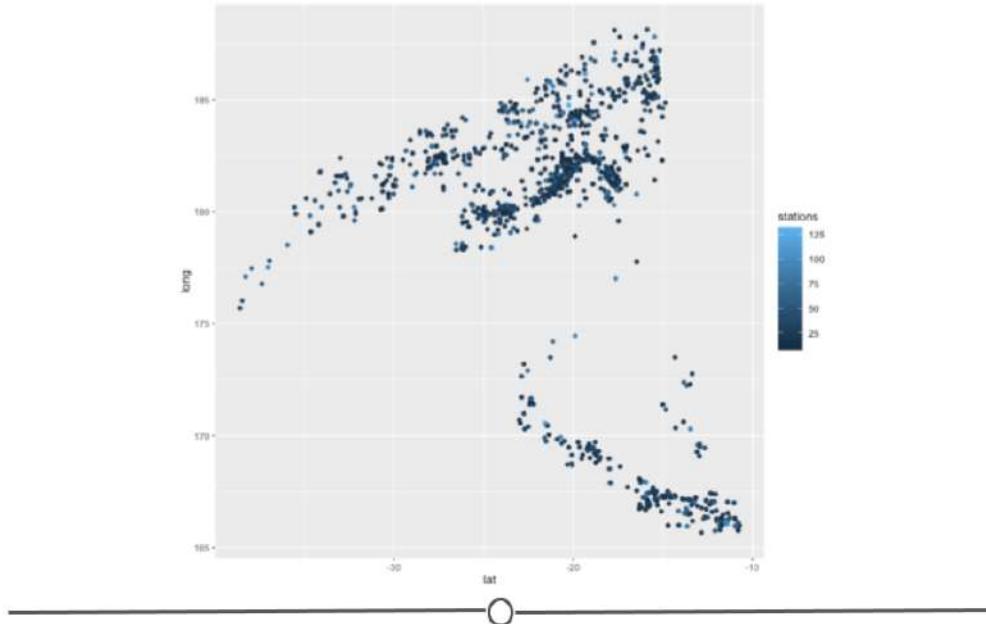
More specifically, however, there are two general approaches to data visualization: **exploratory plots** and **explanatory plots**.

Exploratory Plots

These are **data displays to help you better understand and discover hidden patterns in the data** you're working with. These won't be the prettiest plots, but they will be incredibly helpful. Exploratory visualizations have a number of general characteristics:

- They are made quickly.
- You'll make a large number of them.
- The axes and legends are cleaned up.

Below we have a graph where the axes are labeled and general pattern can be determined. This is a great example of an exploratory plot. It lets you the analyst know what's going on in your data, but it isn't yet ready for a big presentation.



Exploratory Plot

As you're trying to understand the data you have on hand, you'll likely make a lot of plots and tables just to figure out to explore and understand the data. Because there are a lot of them and they're for your use (rather than for communicating with others), you don't have to spend all your time making them perfect. But, you do have to spend enough time to make sure that you're drawing the right conclusions from this. Thus, you don't have to spend a long time considering what colors are perfect on these, but you do want to make sure your axes are not cut off.

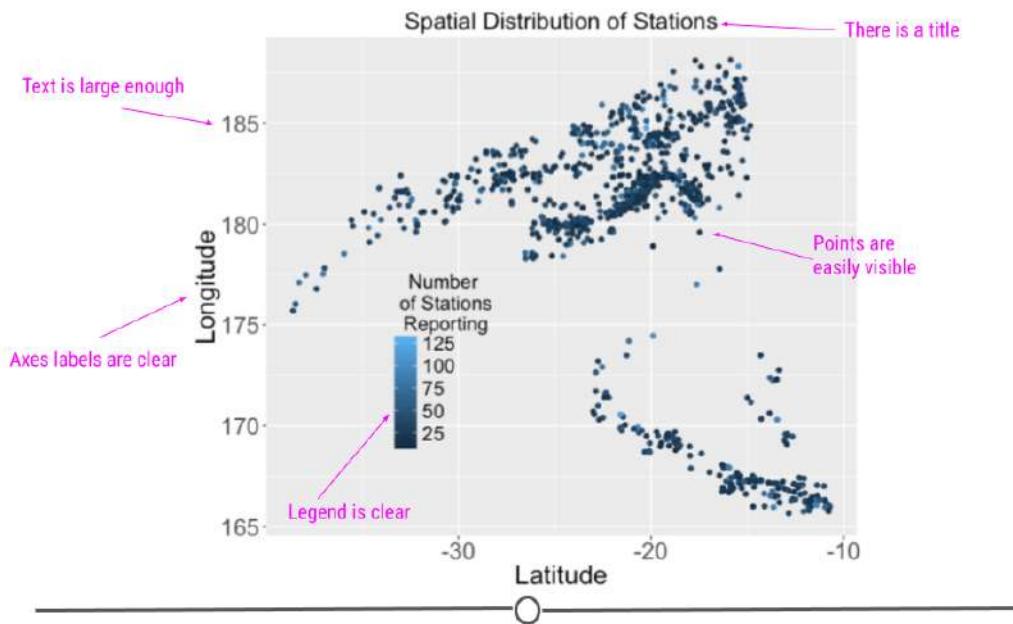
Other Exploratory Plotting Examples: [Air Quality Data](#)

Explanatory Plots

These are data displays that aim to **communicate insights to others**. These are plots that you spend a lot of time making sure they're easily interpretable by an audience. General characteristics of explanatory plots:

- They take a while to make.
- There are only a few of these for each project.
- You've spent a lot of time making sure the colors, labels, and sizes are all perfect for your needs.

Here we see an improvement upon the exploratory plot we looked at previously. Here, the axis labels are more descriptive. All of the text is larger. The legend has been moved onto the plot. The points on the plot are larger. And, there is a title. All of these changes help to improve the plot, making it an explanatory plot that would be presentation-ready.



Explanatory Plots

Explanatory plots are made after you've done an analysis and once you really understand the data you have. The goal of these plots is to communicate your findings clearly to others. To do so, you want to make sure these plots are made carefully - the axis labels should all be clear, the labels should all be large enough to read, the colors should all be carefully chosen, etc.. As this takes times and because you do not want to overwhelm your audience, you only want to have a few of these for each project. We often refer to these as "publication ready" plots. These are the plots that would make it into an article at the New York Times or in your presentation to your bosses.

Other Explanatory Plotting Examples:

- How the Recession Shaped the Economy (NYT)
- 2018 Flu Season (FiveThirtyEight)

Plot Types

Above we saw data displayed as both an exploratory plot and an explanatory plot. That plot was an example of a scatterplot. However, there are many types of plots that are helpful.

We'll discuss a few basic ones below and will include links to a few galleries where you can get a sense of the many different types of plots out there.

To do this, we'll use the “Davis” dataset of the `carData` package which includes, height and weight information for 200 people.

To use this data first make sure the `carData` package is installed and load it.

```
#install.packages(carData)
library(carData)
Davis <- carData::Davis
```

A portion of
the Davis
dataset we'll
be working
with

	sex	weight	height	repwt	repht
1	M	77	182	77	180
2	F	58	161	51	159
3	F	53	161	54	158
4	M	68	177	70	175
5	F	59	157	59	155
6	M	76	170	76	165
7	M	76	167	77	165
8	M	69	186	73	180
9	M	71	178	71	175
10	M	65	171	64	170
11	M	70	175	75	174
12	F	51	161	52	158
13	F	64	166	64	165
14	F	52	163	57	160
15	F	65	166	66	165
16	M	92	187	101	185
17	F	62	168	62	165
18	M	76	197	75	200
19	F	61	175	61	171
20	M	119	180	124	178
21	F	61	170	61	170
22	M	65	175	66	173
23	M	66	173	70	170
24	F	54	171	59	168

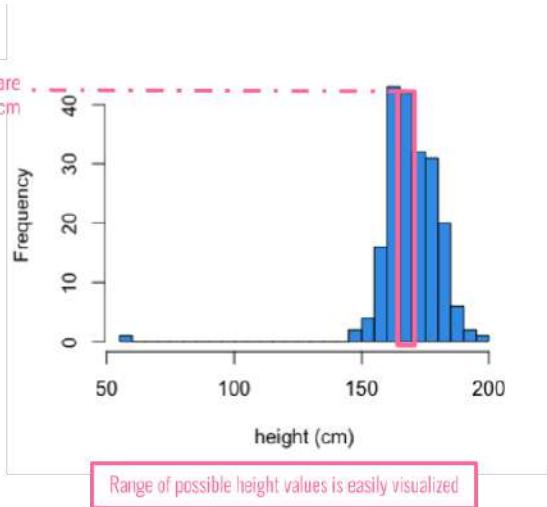
Dataset

Histogram

Histograms are helpful when you want to **better understand what values you have in your dataset for a single set of numbers**. For example, if you had a dataset with information about many people, you may want to know how tall the people in your dataset are. To quickly visualize this, you could use a histogram. Histograms let you know what range of values you have in your dataset. For example, below you can see that in this dataset, the

height values range from around 50 to around 200 cm. The shape of the histogram also gives you information about the individuals in your dataset. The number of people at each height are also counted. So, the tallest bars show that there are about 40 people in the dataset whose height is between 165 and 170 cm. Finally, you can quickly tell, at a glance that most people in this dataset are at least 150 cm tall, but that there is at least one individual whose reported height is much lower.

Histograms
Information about
a single
quantitative
variable



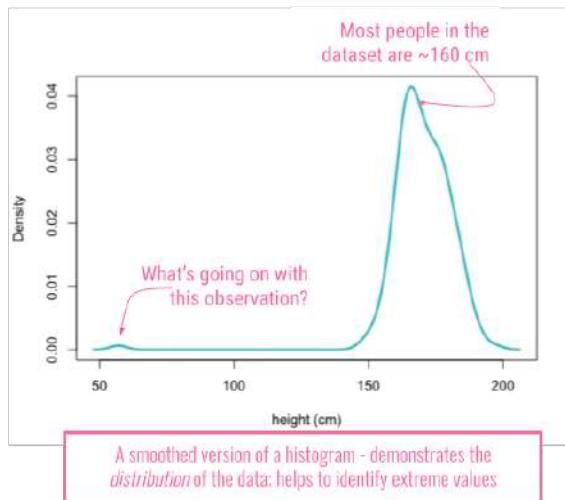
Histogram

Densityplot

Densityplots are smoothed versions of histograms, visualizing the distribution of a continuous variable. These plots effectively visualize the distribution shape and are, unlike histograms, are not sensitive to the number of bins chosen for visualization.

Densityplot

Information about
a single
quantitative
variable



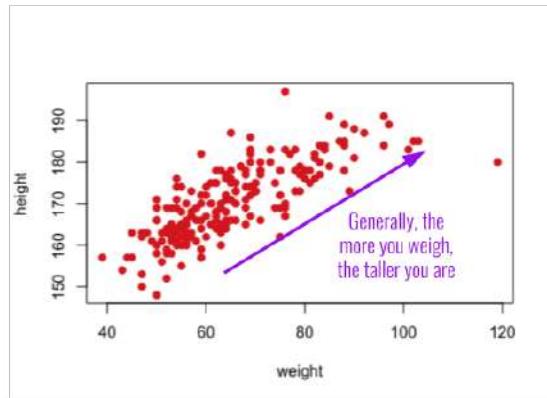
Densityplot

Scatterplot

Scatterplots are helpful when you have **numerical values for two different pieces of information** and you want to understand the relationship between those pieces of information. Here, each dot represents a different person in the dataset. The dot's position on the graph represents that individual's height and weight. Overall, in this dataset, we can see that, in general, the more someone weighs, the taller they are. Scatterplots, therefore help us at a glance better understand the relationship between two sets of numbers.

Scatterplot

Relationship between
two quantitative
variables



Scatter Plot

Barplot

When you only have a **single categorical variable that you want broken down and quantified by category**, a barplot will be ideal. For example if you wanted to look at how many females and how many males you have in your dataset, you could use a barplot. The comparison in heights between bars clearly demonstrates that there are more females in this dataset than males.

Barplot
Count of values
within a **single**
categorical variable



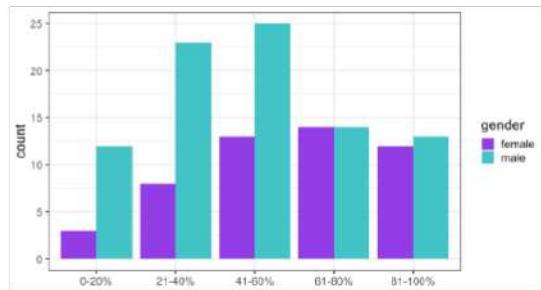
Barplot

Grouped Barplot

Grouped barplots, like simple barplots, demonstrate the counts for a group; however, they break this down by an additional categorical variable. For example, here we see the number of individuals within each % category along the x-axis. But, these data are further broken down by gender (an additional categorical variable). Comparisons between bars that are side-by-side are made most easily by our visual system. So, it's important to ensure that the bars you want viewers to be able to compare most easily are next to one another in this plot type.

Grouped Barplot

Count of values broken down across categorical variables

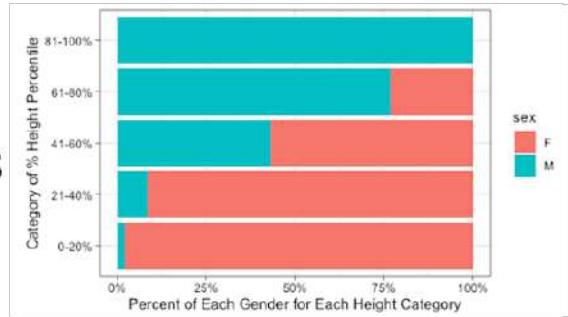


Grouped Barplot

Stacked Barplot

Another common variation on barplots are stacked barplots. Stacked barplots take the information from a grouped barplot but stacks them on top of one another. This is most helpful when the bars add up to 100%, such as in a survey response where you're measuring percent of respondents within each category. Otherwise, it can be hard to compare between the groups within each bar.

Stacked Barplot
Count/proportion of values
broken down across
categorical variables



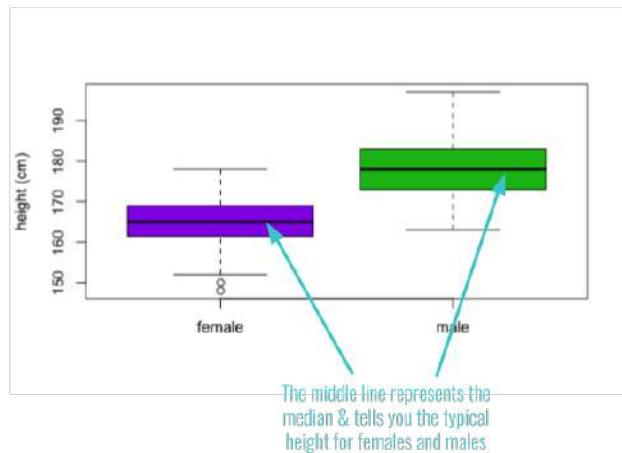
Stacked Barplot

Boxplot

Boxplots also summarize **numerical values across a category**; however, instead of just comparing the heights of the bar, they give us an idea of the range of values that each category can take. For example, if we wanted to compare the heights of men to the heights of women, we could do that with a boxplot.

Boxplot

Summary of a quantitative variable broken down by a categorical variable

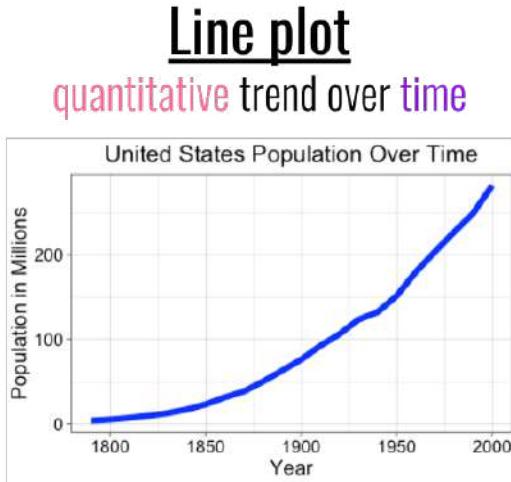


Boxplot

To interpret a boxplot, there are a few places where we'll want to focus our attention. For each category, the horizontal line through the middle of the box corresponds to the median value for that group. So, here, we can say that the median, or most typical height for females is about 165 cm. For males, this value is higher, just under 180 cm. Outside of the colored boxes, there are dashed lines. The ends of these lines correspond to the typical range of values. Here, we can see that females tend to have heights between 150 and 180cm. Lastly, when individuals have values outside the typical range, a boxplot will show these individuals as circles. These circles are referred to as outliers.

Line Plots

The final type of basic plot we'll discuss here are line plots. Line plots are most effective at showing a quantitative trend over time.



Line Plot

Resources to look at these and other types of plots:

- [R Graph Gallery](#)
- [Ferdio Data Visualization Catalog](#)

Making Good Plots

The goal of data visualization in data analysis is to improve understanding of the data. As mentioned in the last lesson, this could mean improving our own understanding of the data or using visualization to improve someone else's understanding of the data.

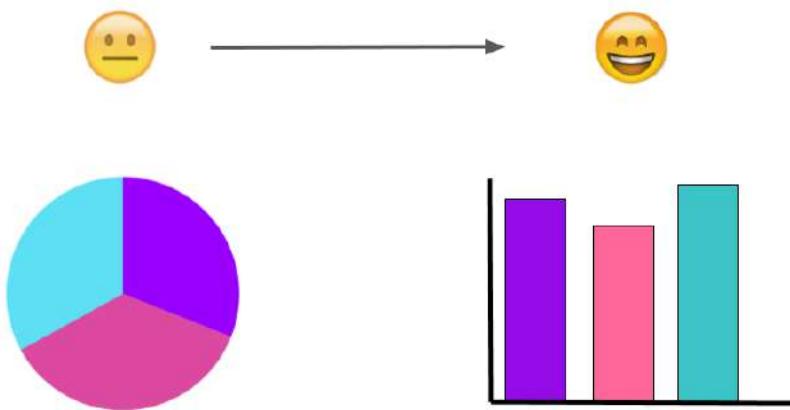
We discussed some general characteristics and basic types of plots in the last lesson, but here we will step through a number of general tips for making good plots.

When generating exploratory or explanatory plots, you'll want to ensure information being displayed is being done so accurately and in a way that best reflects the reality within the dataset. Here, we provide a number of tips to keep in mind when generating plots.

Choose the Right Type of Plot

If your goal is to allow the viewer to compare values across groups, pie charts should largely be avoided. This is because it's easier for the human eye to differentiate between bar heights than it is between similarly-sized slices of a pie. Thinking about the best way to visualize your data before making the plot is an important step in the process of data visualization.

Choose the right type of plot



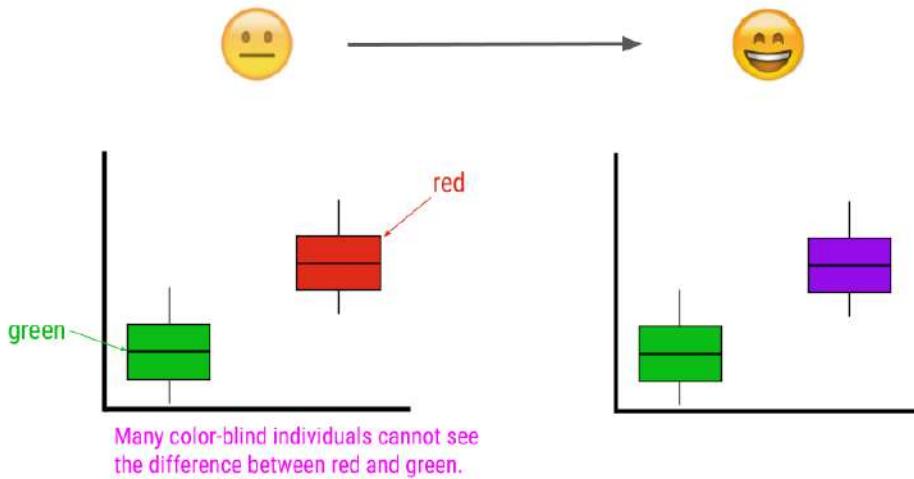
When looking at values, bar charts make it much easier to see the difference between groups!

Choose an appropriate plot for the data you're visualizing.

Be Mindful When Choosing Colors

Choosing colors that work for the story you're trying to convey with your visualization is important. Avoiding colors that are hard to see on a screen or when projected, such as pastels, is a good idea. Additionally, red-green color blindness is common and leads to difficulty in distinguishing reds from greens. Simply avoiding making comparisons between these two colors is a good first step when visualizing data.

Be mindful when choosing colors



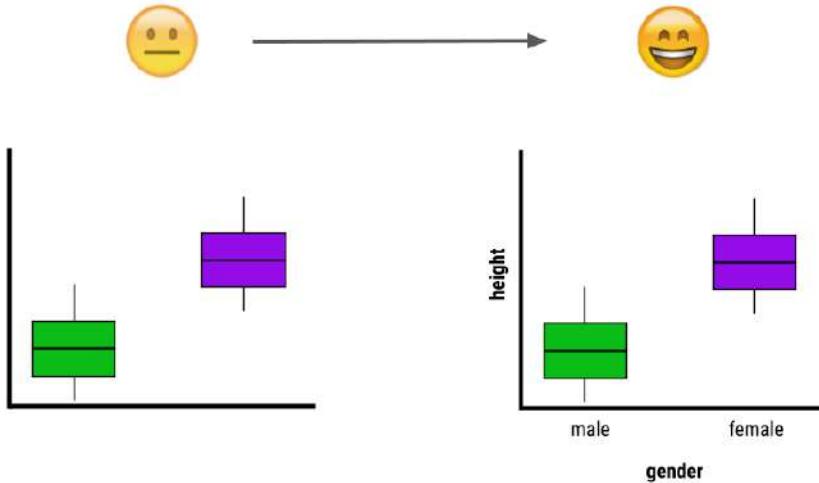
Choosing appropriate colors for visualizations is important

Beyond red-green color blindness, there is an entire group of experts out there in color theory. To learn more about available [color palettes in R](#) or to read more from a pro named Lisa Charlotte Rost [talking about color choices in data visualization](#), feel free to read more.

Label the Axes

Whether you're making an exploratory or explanatory visualization, labeled axes are a must. They help tell the story of the figure. Making sure the axes are clearly labeled is also important. Rather than labeling the graph below with "h" and "g," we chose the labels "height" and "gender," making it clear to the viewer exactly what is being plotted.

Label your axes!

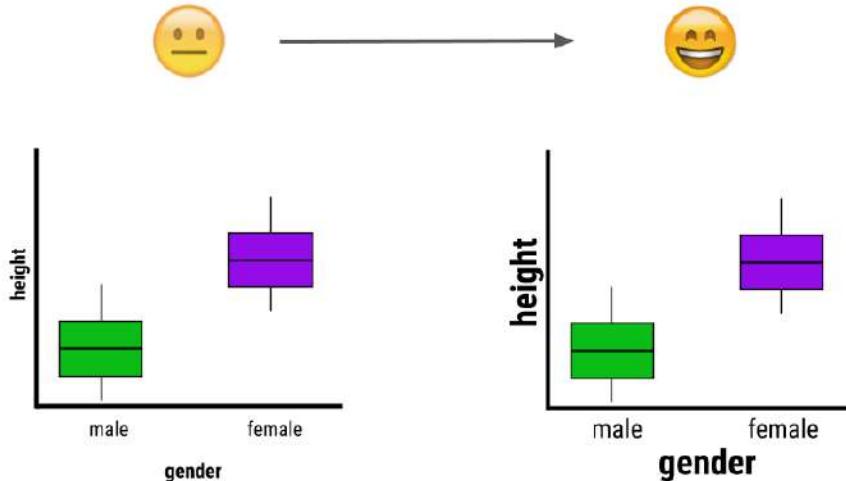


Having descriptive labels on your axes is critical

Make Sure the Text is Readable

Often text on plots is too small for viewers to read. By being mindful of the size of the text on your axes, in your legend, and used for your labels, your visualizations will be greatly improved.

Make sure the text size is big enough!

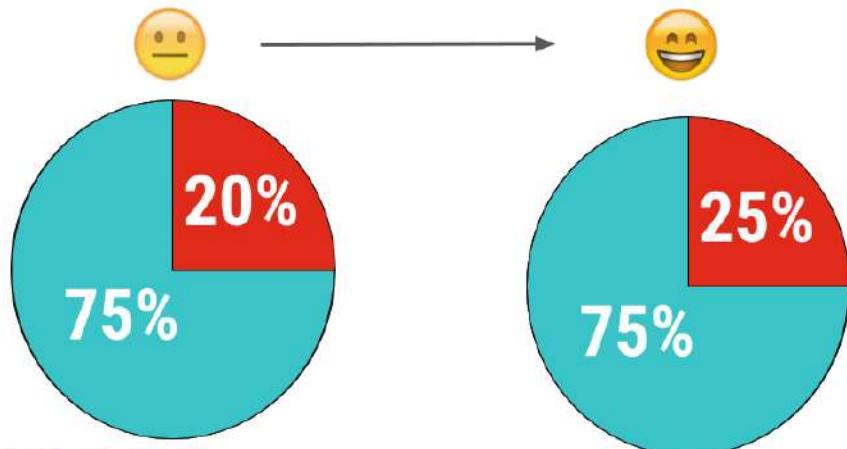


On the right, we see that the text is easily readable

Make Sure the Numbers Add Up

When you're making a plot that should sum to 100, make sure that it in fact does. Taking a look at visualizations after you make them to ensure that they make sense is an important part of the data visualization process.

Make sure your numbers add up!



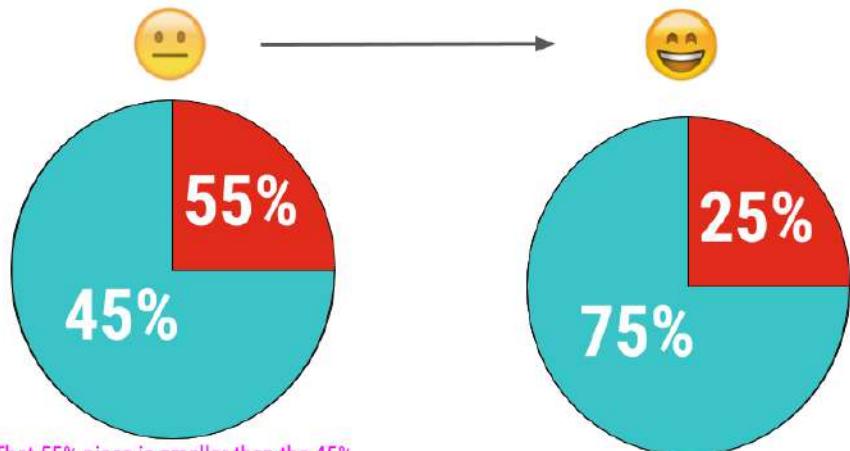
$75 + 20 = 95 \dots \text{whoops!}$
Pie charts should always add up to 100%!

At left, the pieces of the pie only add up to 95%. On the right, this error has been fixed and the pieces add up to 100%

Make Sure the Numbers and Plots Make Sense Together

Another common error is having labels that don't reflect the underlying graphic. For example, here, we can see on the left that the turquoise piece is more than half the graph, and thus the label 45% must be incorrect. At right, we see that the labels match what we see in the figure.

Make sure the numbers and graphic represent the data



That 55% piece is smaller than the 45% piece. That doesn't make sense!

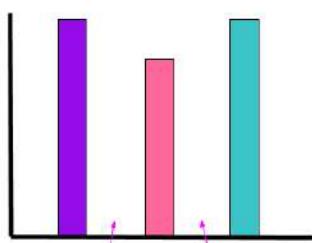
Checking to make sure the numbers and plot make sense together is important

Make Comparisons Easy on Viewers

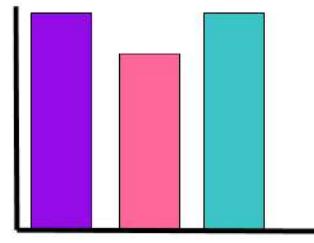
There are many ways in which you can make comparisons easier on the viewer. For example, avoiding unnecessary whitespace between the bars on your graph can help viewers make comparisons between the bars on the barplot.

Make comparisons easy on your readers

Avoid unnecessary whitespace



When the bars are spread out, there is unnecessary whitespace!

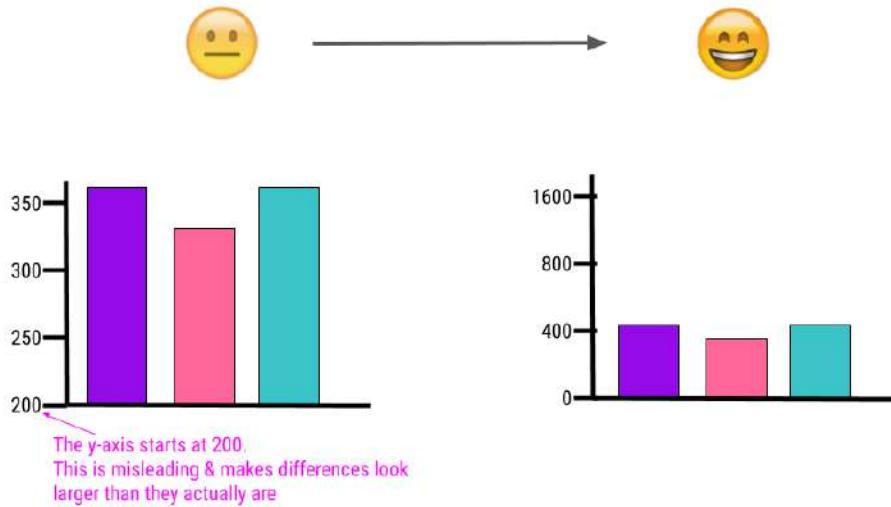


At left, there is extra white space between the bars of the plot that should be removed. On the right, we see an improved plot

Use y-axes That Start at Zero

Often, in an attempt to make differences between groups look larger than they are, y-axis will be started at a value other than zero. This is misleading. Y-axis for numerical information should start at zero.

Use y-axes that start at 0 for barplots

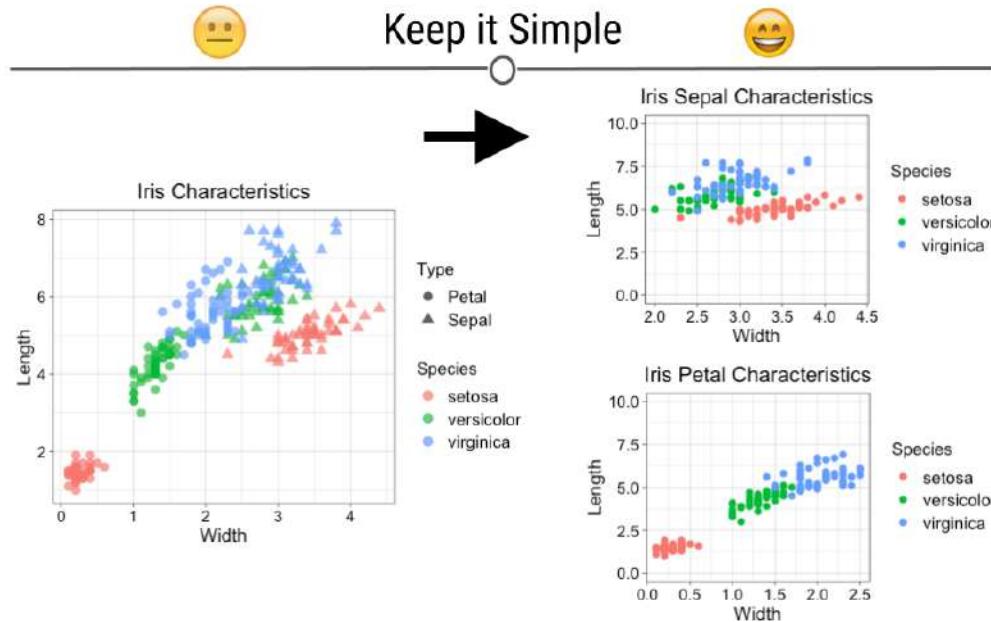


At left, the differences between the vars appears larger than on the right; however, this is just because the y-axis starts at 200. The proper way to start this graph is to start the y-axis at 0.

Keep It Simple

The goal of data visualization is to improve understanding of data. Sometimes complicated visualizations cannot be avoided; however, when possible, keep it simple.

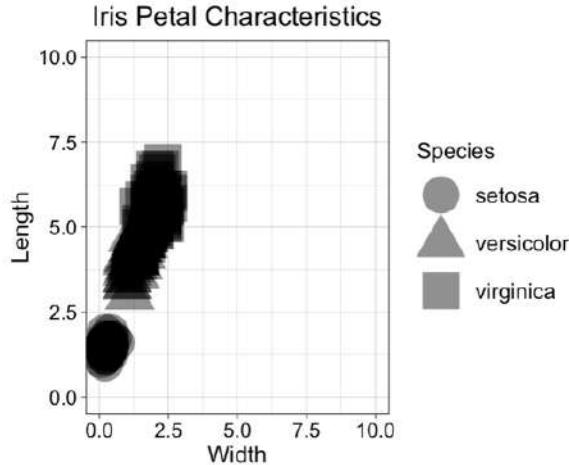
Here, the graphic on the left does not immediately convey a main point. It's hard to interpret what each point means or what the story of this graphic is supposed to be. In contrast, the graphics on the right are simpler and each show a more obvious story. Make sure that your main point comes through:



Main point unclear

Similarly, the intention of your graphic should never be to mislead or confuse. Be sure that your data visualizations improve viewers' understanding. Using unusual axes limits or point sizes, or using vague labels can make plots misleading. This plot creates an effective exclamation mark shape which is fun, but it is no longer clear what points correspond to what species. Furthermore, this plot makes it look like petal width is not very distinguishable across the different species (particularly for versicolor and virginica), which is the opposite of what the previous petal plot conveyed.

Don't Mislead!



Confusion is conveyed here

Plot Generation Process

Having discussed some general guidelines, there are a number of questions you should ask yourself before making a plot. These have been nicely laid out in a [blog post](#) from the wonderful team at [Chartable](#), [Datawrapper](#)'s blog and we will summarize them here. The post argues that there are three main questions you should ask any time you create a visual display of your data. We will discuss these three questions below

What's your point?

Whenever you have data you're trying to plot, think about what you're actually trying to show. Once you've taken a look at your data, a good title for the plot can be helpful. Your title should tell viewers what they'll see when they look at the plot.

How can you emphasize your point in your chart?

We talked about it in the last lesson, but an incredibly important decision is choosing an appropriate chart for the type of data you have. In the next section of this lesson, we'll discuss

what type of data are appropriate for each type of plot in R; however, for now, we'll just focus on an iPhone data example. With this example, we'll discuss that you can emphasize your point by:

- Adding data
- Highlighting data with color
- Annotating your plot

Adding data

In any plot that makes a specific claim, it usually important to show additional data as a reference for comparison. For example, if you were making a plot of that suggests that the iPhone has been Apple's most successful product, it would be helpful for the plot to compare iPhone sales with other Apple products, say, the iPad or the iPod. By adding data about other Apple products over time, we can visualize just how successful the iPhone has been compared to other products.

Highlighting data with color

Colors help direct viewers' eyes to the most important parts of the figure. Colors tell your readers where to focus their attention. Grays help to tell viewers where to focus less of their attention, while other colors help to highlight the point your trying to make.

Annotate your plot

By highlighting parts of your plot with arrows or text on your plot, you can further draw viewers' attention to certain part of the plot. These are often details that are unnecessary in exploratory plots, where the goal is just to better understand the data, but are very helpful in explanatory plots, when you're trying to draw conclusions from the plot.

What Does Your Final Chart Show?

A plot title should first tell viewers what they would see in the plot. The second step is to show them with the plot. The third step is to make it extra clear to viewers what they should be seeing with descriptions, annotations, and legends. You explain to viewers what they should be seeing in the plot and the source of your data. Again, these are important pieces of creating a complete explanatory plot, but are not all necessary when making exploratory plots.

Write precise descriptions

Whether it's a figure legend at the bottom of your plot, a subtitle explaining what data are plotted, or clear axes labels, text describing clearly what's going on in your plot is important. Be sure that viewers are able to easily determine what each line or point on a plot represents.

Add a source

When finalizing an explanatory plot, be sure to source your data. It's always best for readers to know where you obtained your data and what data are being used to create your plot. Transparency is important.

ggplot2: Basics

R was initially developed for statisticians, who often are interested in generating plots or figures to visualize their data. As such, a few basic plotting features were built in when R was first developed. These are all still available; however, over time, a new approach to graphing in R was developed. This new approach implemented what is known as the [grammar of graphics](#), which allows you to develop elegant graphs flexibly in R. Making plots with this set of rules requires the R package `ggplot2`. This package is a core package in the tidyverse. So as long as the tidyverse has been loaded, you're ready to get started.

```
# load the tidyverse
library(tidyverse)
```

ggplot2 Background

The grammar of graphics implemented in `ggplot2` is based on the idea that you can build *any* plot as long as you have a few pieces of information. To start building plots in `ggplot2`, we'll need some data and we'll need to know the type of plot we want to make. The type of plot you want to make in `ggplot2` is referred to as a geom. This will get us started, but the idea behind `ggplot2` is that every new concept we introduce will be layered on top of the information you've already learned. In this way, `ggplot2` is *layered* - layers of information add on top of each other as you build your graph. In code written to generate a `ggplot2` figure, you will see each line is separated by a plus sign (+). Think of each line as a different layer of the graph. We're simply adding one layer on top of the previous layers to generate the graph. You'll see exactly what we mean by this throughout each section in this lesson.

To get started, we'll start with the two basics (data and a geom) and build additional layers from there.

As we get started plotting in `ggplot2`, plots will take the following general form:

```
ggplot(data = DATASET) +  
  geom_PLOT_TYPE(mapping = aes(VARIABLE($)))
```

When using `ggplot2` to generate figures, you will always begin by calling the `ggplot()` function. You'll then specify your dataset within the `ggplot()` function. Then, before making your plot you will also have to specify what `geom` type you're interested in plotting. We'll focus on a few basic geoms in the next section and give examples of each plot type (`geom`), but for now we'll just work with a single `geom`: `geom_point`.

`geom_point` is most helpful for creating scatterplots. As a reminder from an earlier lesson, scatterplots are useful when you're looking at the relationship between two numeric variables. Within `geom` you will specify the arguments needed to tell `ggplot2` how you want your plot to look.

You will map your variables using the aesthetic argument `aes`. We'll walk through examples below to make all of this clear. However, get comfortable with the overall look of the code now.

Example Dataset: diamonds

To build your first plot in `ggplot2` we'll make use of the fact that there are some datasets already available in R. One frequently-used dataset is known as `diamonds`. This dataset contains prices and other attributes of 53,940 diamonds, with each row containing information about a different diamond. If you look at the first few rows of data, you can get an idea of what data are included in this dataset.

```

diamonds <- as_tibble(diamonds)
diamonds
# A tibble: 53,940 × 10
  carat cut      color clarity depth table price     x     y     z
  <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1 0.23 Ideal     E      SI2     61.5   55    326   3.95  3.98  2.43
2 0.21 Premium   E      SI1     59.8   61    326   3.89  3.84  2.31
3 0.23 Good      E      VS1     56.9   65    327   4.05  4.07  2.31
4 0.29 Premium   I      VS2     62.4   58    334   4.2   4.23  2.63
5 0.31 Good      J      SI2     63.3   58    335   4.34  4.35  2.75
6 0.24 Very Good J      VVS2    62.8   57    336   3.94  3.96  2.48
7 0.24 Very Good I      VVS1    62.3   57    336   3.95  3.98  2.47
8 0.26 Very Good H      SI1     61.9   55    337   4.07  4.11  2.53
9 0.22 Fair      E      VS2     65.1   61    337   3.87  3.78  2.49
10 0.23 Very Good H     VS1     59.4   61    338   4     4.05  2.39
# ... with 53,930 more rows

```

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
6	0.24	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
7	0.24	Very Good	I	VVS1	62.3	57.0	336	3.95	3.98	2.47
8	0.26	Very Good	H	SI1	61.9	55.0	337	4.07	4.11	2.53
9	0.22	Fair	E	VS2	65.1	61.0	337	3.87	3.78	2.49
10	0.23	Very Good	H	VS1	59.4	61.0	338	4.00	4.05	2.39
11	0.30	Good	J	SI1	64.0	55.0	339	4.25	4.28	2.73
12	0.23	Ideal	J	VS1	62.8	56.0	340	3.93	3.90	2.46

First 12 rows of diamonds dataset

Here you see a lot of numbers and can get an idea of what data are available in this dataset. For example, in looking at the column names across the top, you can see that we have

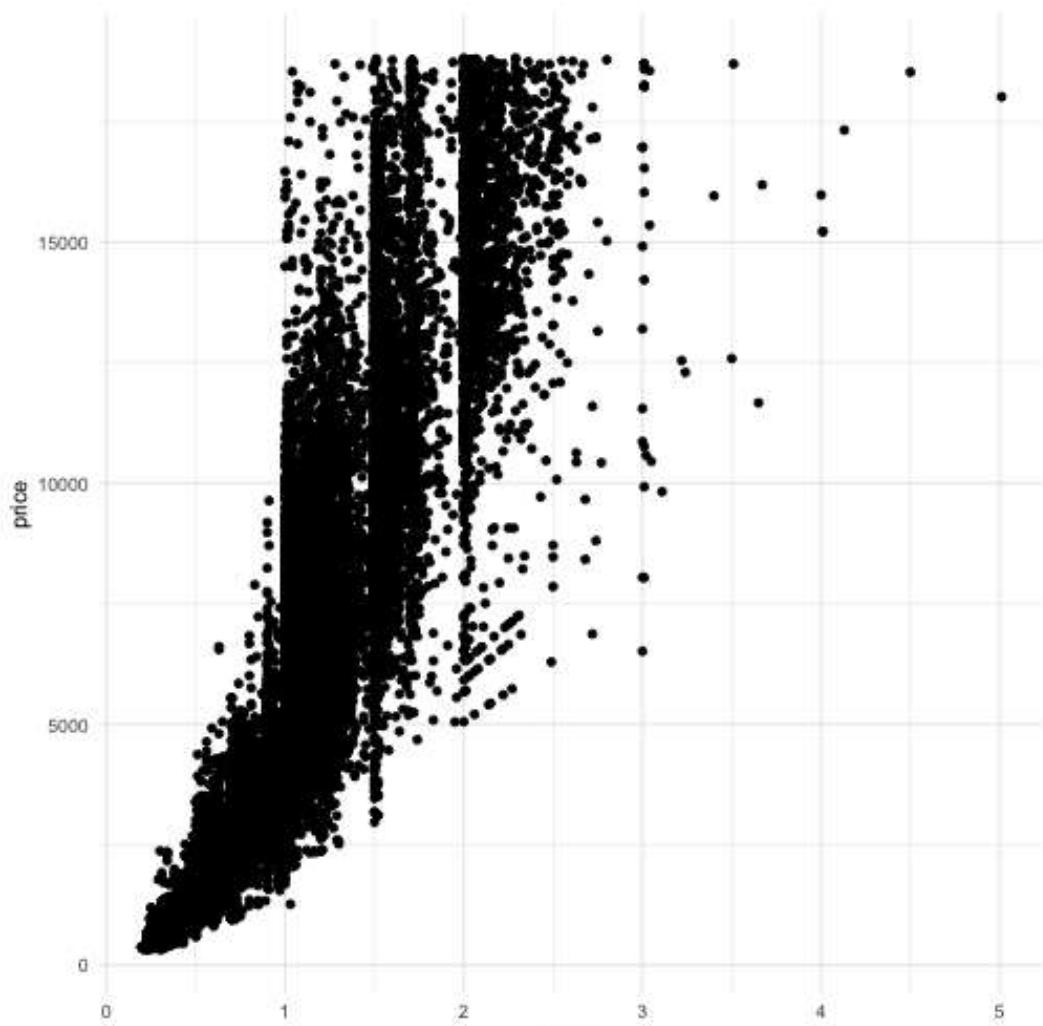
information about how many carats each diamond is (`carat`), some information on the quality of the diamond cut (`cut`), the color of the diamond from J (worst) to D (best) (`color`), along with a number of other pieces of information about each diamond.

We will use this dataset to better understand how to generate plots in R, using `ggplot2`.

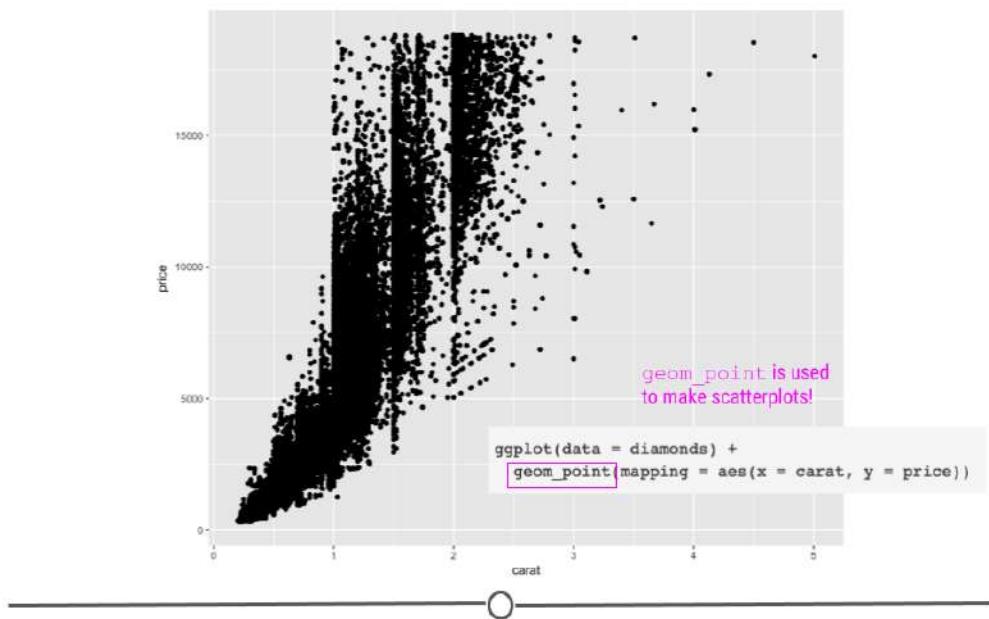
Scatterplots: `geom_point()`

In `ggplot2` we specify these by defining `x` and `y` *within* the `aes()` argument. The `x` argument defines which variable will be along the bottom of the plot. The `y` refers to which variable will be along the left side of the plot. If we wanted to understand the relationship between the number of carats in a diamond and that diamond's price, we may do the following:

```
# generate scatterplot with geom_point()
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))
```



plot of chunk unnamed-chunk-10



diamonds scatterplot

In this plot, we see that, in general, the larger the diamond is (or the more carats it has), the more expensive the diamond is (price), which is probably what we would have expected. However, now, we have a plot that definitively supports this conclusion!

Aesthetics

What if we wanted to alter the size, color or shape of the points? Probably unsurprisingly, these can all be changed within the aesthetics argument. After all, something's aesthetic refers to how something looks. Thus, if you want to change the look of your graph, you'll want to play around with the plot's aesthetics.

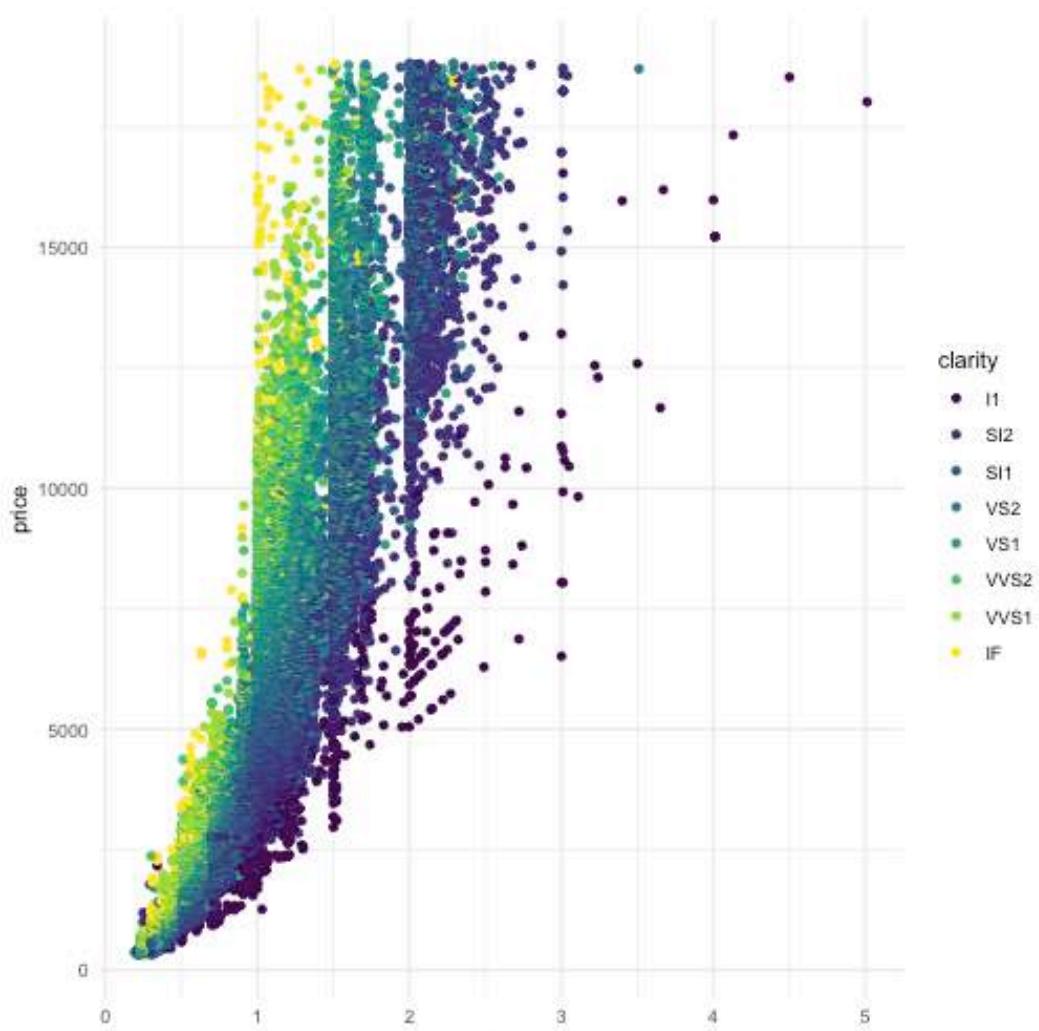
In fact, in the plots above you'll notice that we specified what should be on the x and y axis within the `aes()` call. These are aesthetic mappings too! We were telling `ggplot2` what to put on each axis, which will clearly affect how the plot looks, so it makes sense that these calls have to occur within `aes()`. Additionally now, we'll focus on arguments within `aes()` that change how the points on the plot look.

Point color

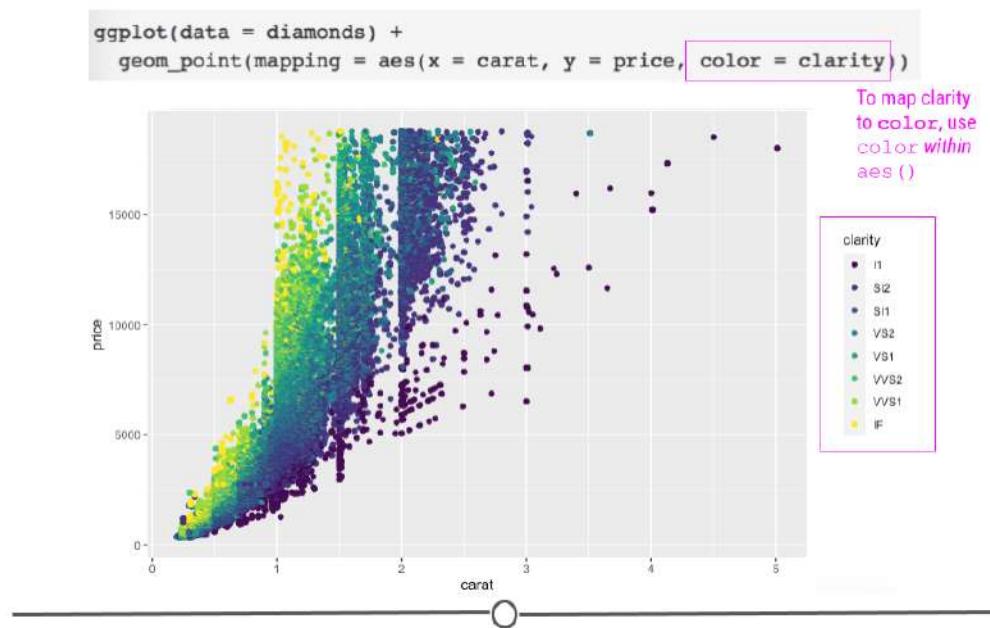
In the scatterplot we just generated, we saw that there was a relationship between carat and price, such that the more carats a diamond has, generally, the higher the price. But, it's not a perfectly linear trend. What we mean by that is that not all diamonds that were 2 carats were exactly the same price. And, not all 3 carat diamonds were exactly the same price. What if we were interested in finding out a little bit more about why this is the case?

Well, we could look at the clarity of the diamonds to see whether or not that affects the price of the diamonds? To add clarity to our plot, we could change the color of our points to differ based on clarity:

```
# adjusting color within aes
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price, color = clarity))
```



plot of chunk unnamed-chunk-11



changing point colors helps us better understand the data

Here, we see that not only are the points now colored by clarity, ggplot2 has also automatically added a legend for us with the various classes and their corresponding point color.

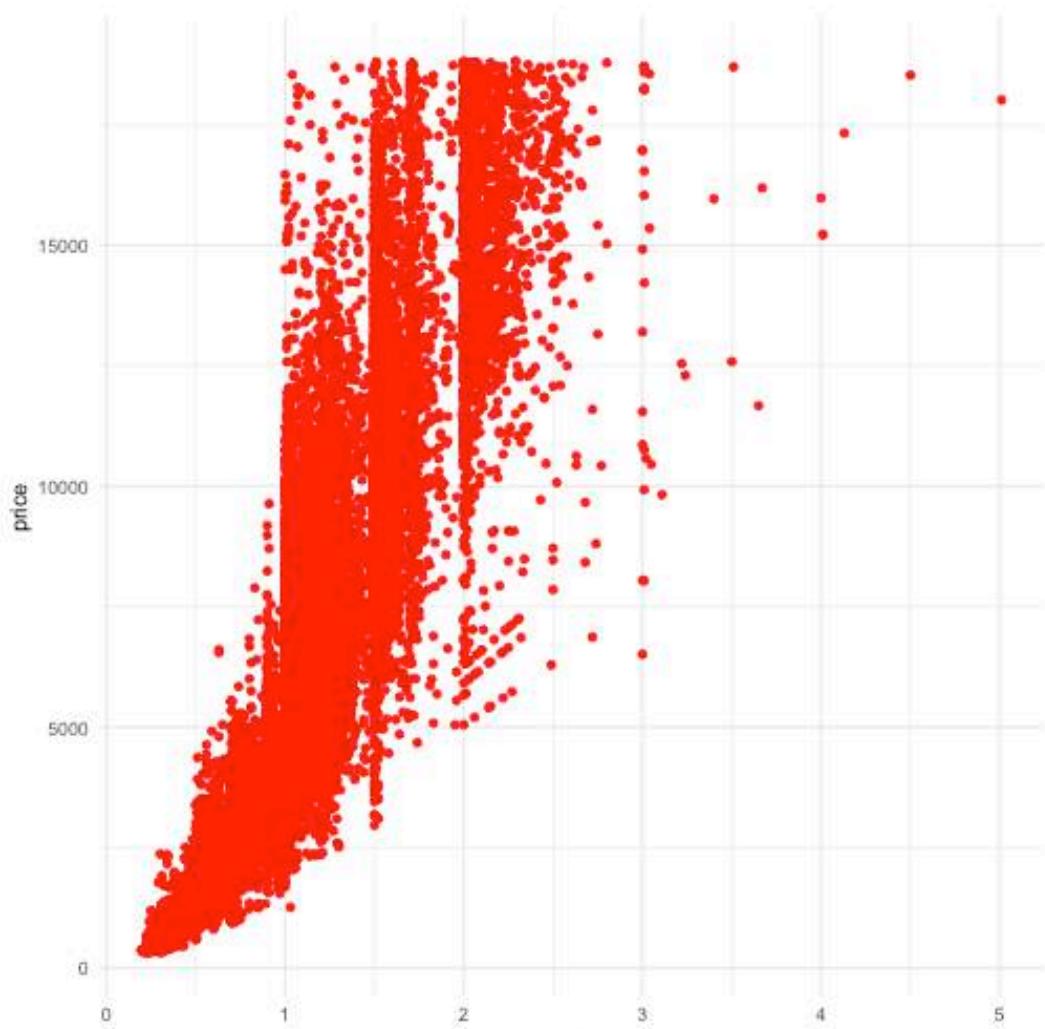
The Help pages of the diamonds dataset (accessed using `?diamonds`) state that clarity is “a measurement of how clear the diamond is.” The documentation also tells us that I1 is the worst clarity and IF is the best (Full scale: I1, SI1, SI2, VS1, VS2, VVS1, VVS2, IF). This makes sense with what we see in the plot. Small (<1 carat) diamonds that have the best clarity level (IF) are some of the most expensive diamonds. While, relatively large diamonds (diamonds between 2 and 3 carats) of the lowest clarity (I1) tend to cost less.

By coloring our points by a different variable in the dataset, we now understand our dataset better. This is one of the goals of data visualization! And, specifically, what we’re doing here in ggplot2 is known as **mapping a variable to an aesthetic**. We took another variable in the dataset, mapped it to a color, and then put those colors on the points in the plot. Well, we only told ggplot2 what variable to map. It took care of the rest!

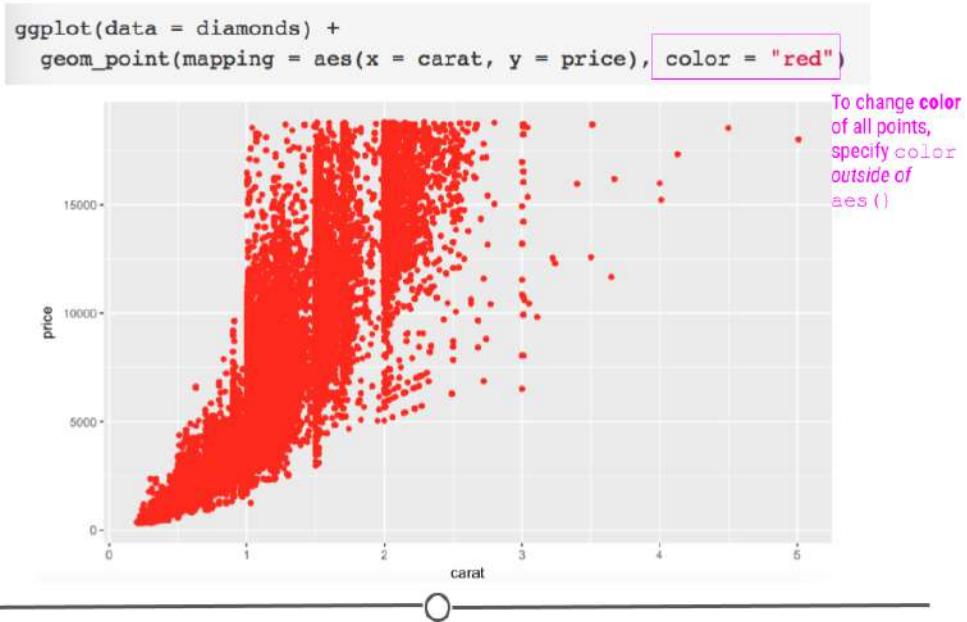
Of course, we can also *manually* specify the colors of the points on our graph; however, manually specifying the colors of points happens *outside* of the `aes()` call. This is because ggplot2 does not have to go through the process of mapping the variable to an aesthetic (color in this case). In the code here, ggplot2 doesn’t have to go through the trouble of figuring out

which level of the variable is going to be which color on the plot (the mapping to the aesthetic part of the process). Instead, it just colors every point red. Thus, **manually specifying the color of your points happens *outside* of `aes()`**:

```
# manually control color point outside aes
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price), color = "red")
```



plot of chunk unnamed-chunk-12

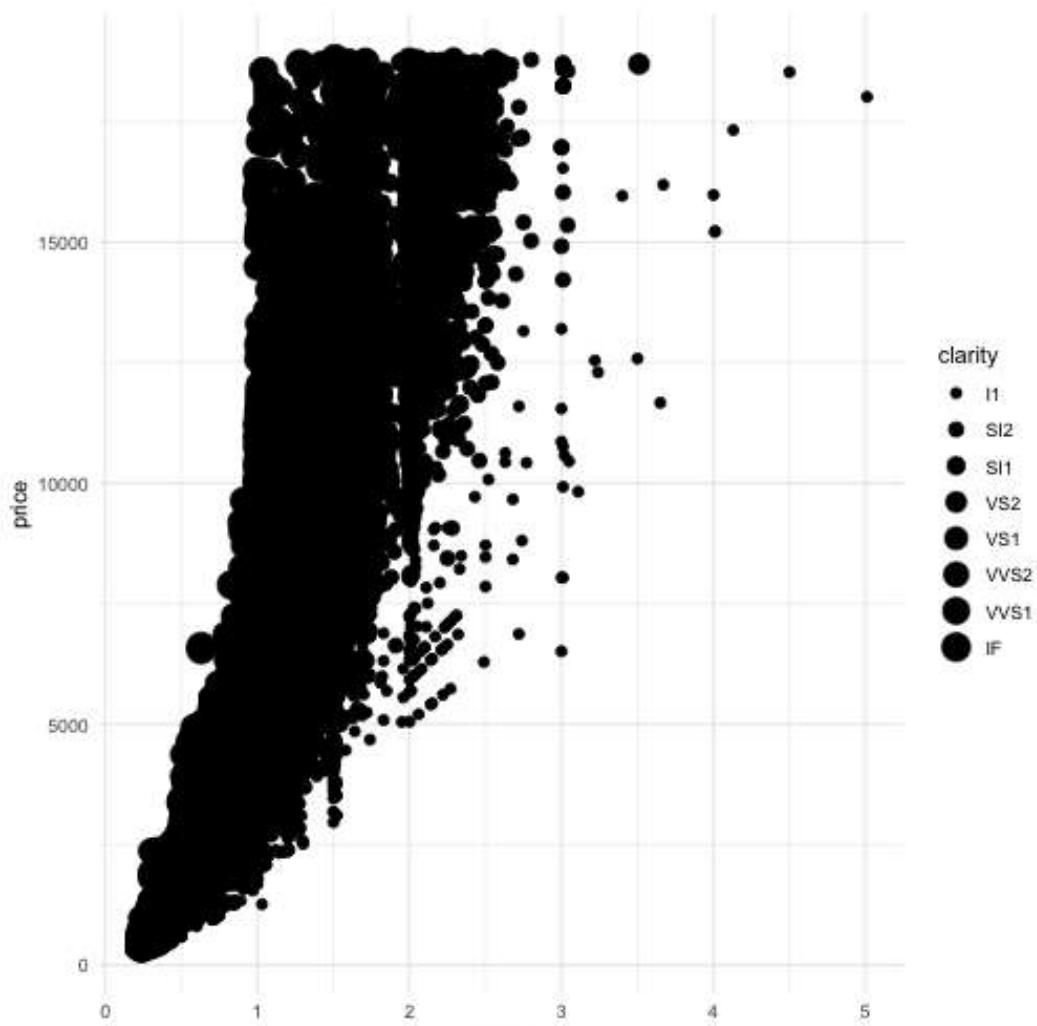


manually specifying point color occurs outside of `aes()`

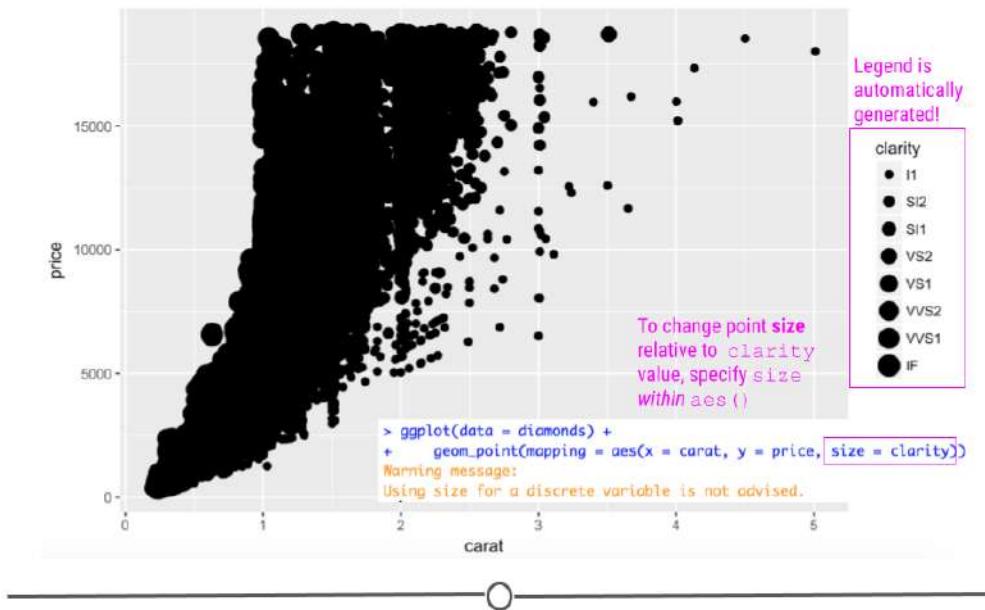
Point size

As above, we can change the point size by mapping another variable to the `size` argument within `aes`:

```
# adjust point size within aes
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price, size = clarity))
```



plot of chunk unnamed-chunk-13



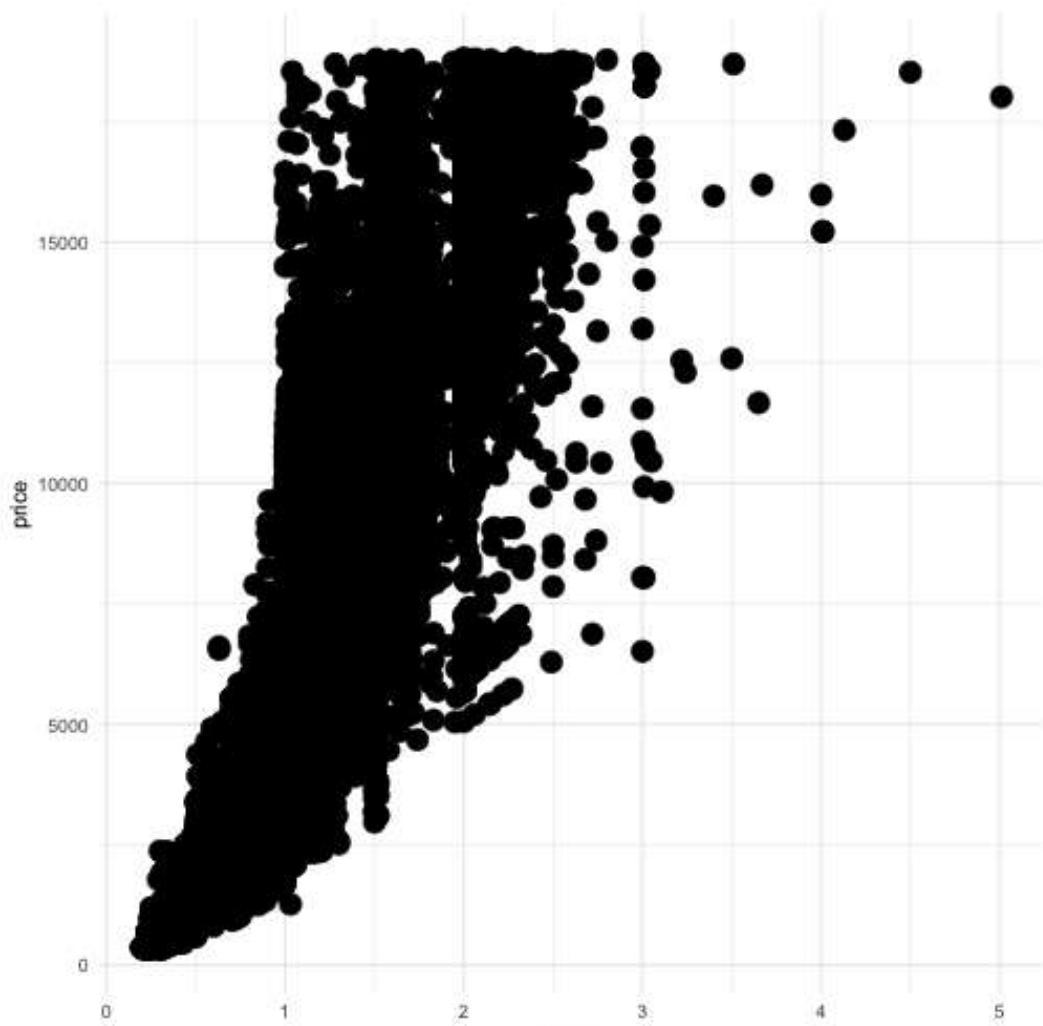
mapping to size changes point size on plot

As above, ggplot2 handles the mapping process. All you have to do is specify what variable you want mapped (clarity) and how you want ggplot2 to handle the mapping (change the point size). With this code, you do get a warning when you run it in R that using a “discrete variable is not advised.” This is because mapping to size is usually done for numeric variables, rather than categorical variables like clarity.

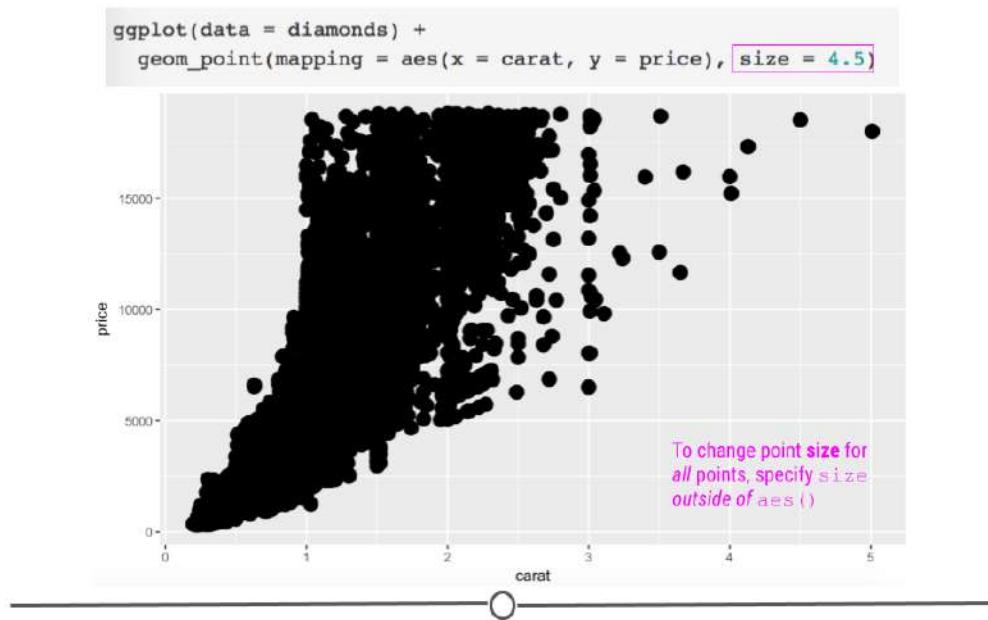
This makes sense here too. The relationship between clarity, carat, and price was easier to visualize when clarity was mapped to color than here where it is mapped to size.

Like the above example with color, the size of *every* point can be changed by calling `size` outside of `aes`:

```
# global control of point size
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price), size = 4.5)
```



plot of chunk unnamed-chunk-14

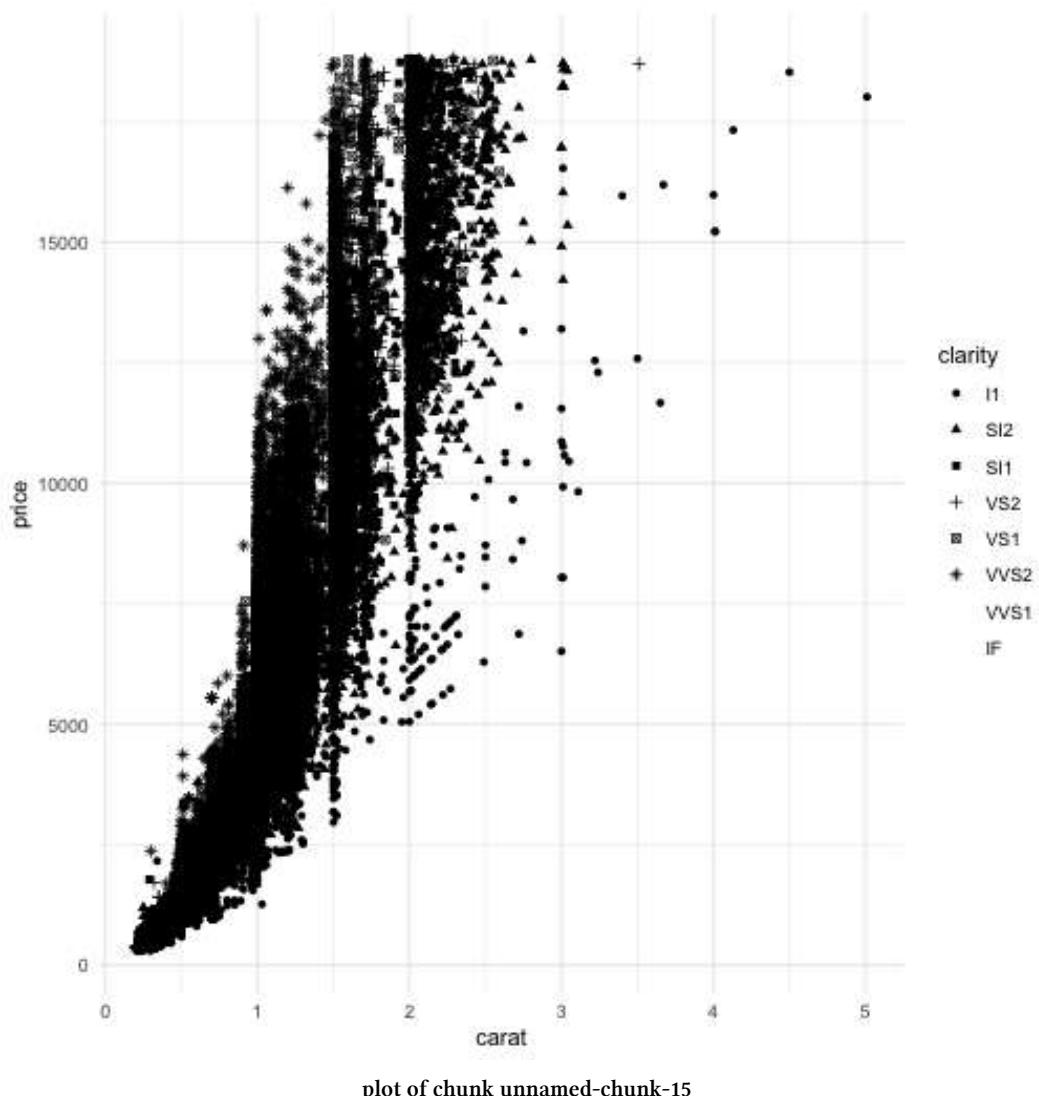


manually specifying point size of all points occurs outside of `aes()`
 Here, we have manually increased the size of *all* the points on the plot.

Point Shape

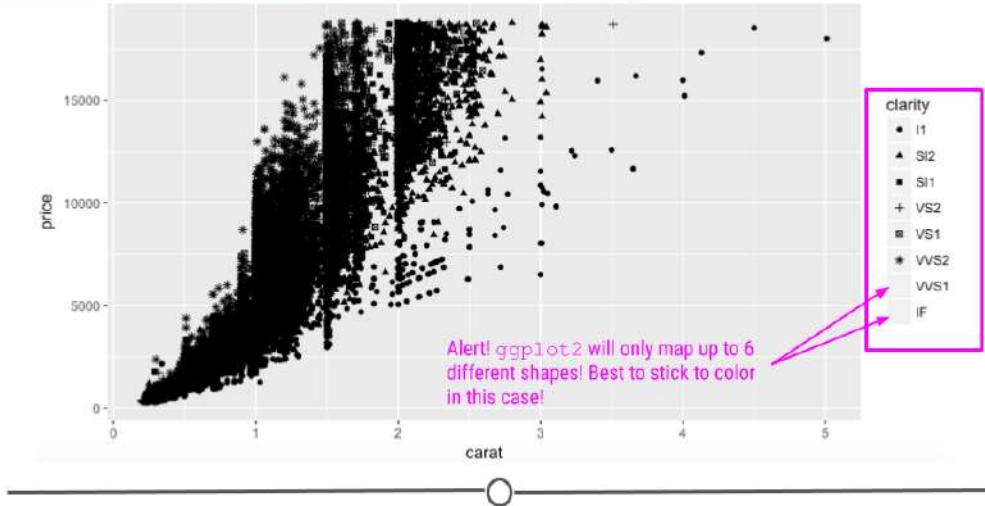
You can also change the shape of the points (`shape`). We've used solid, filled circles thus far (the default in `geom_point`), but we could specify a different shape for each clarity.

```
# map clarity to point shape within aes
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price, shape = clarity))
Warning: Using shapes for an ordinal variable is not advised
Warning: The shape palette can deal with a maximum of 6 discrete values because
more than 6 becomes difficult to discriminate; you have 8. Consider
specifying shapes manually if you must have them.
Warning: Removed 5445 rows containing missing values (geom_point).
```



```
> ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price, shape = clarity))
Warning message:
1: The shape palette can deal with a maximum of 6 discrete values because more than 6 becomes difficult to discriminate; you have 8. Consider specifying shapes manually if you must have them.
2: Removed 5445 rows containing missing values (geom_point).
```

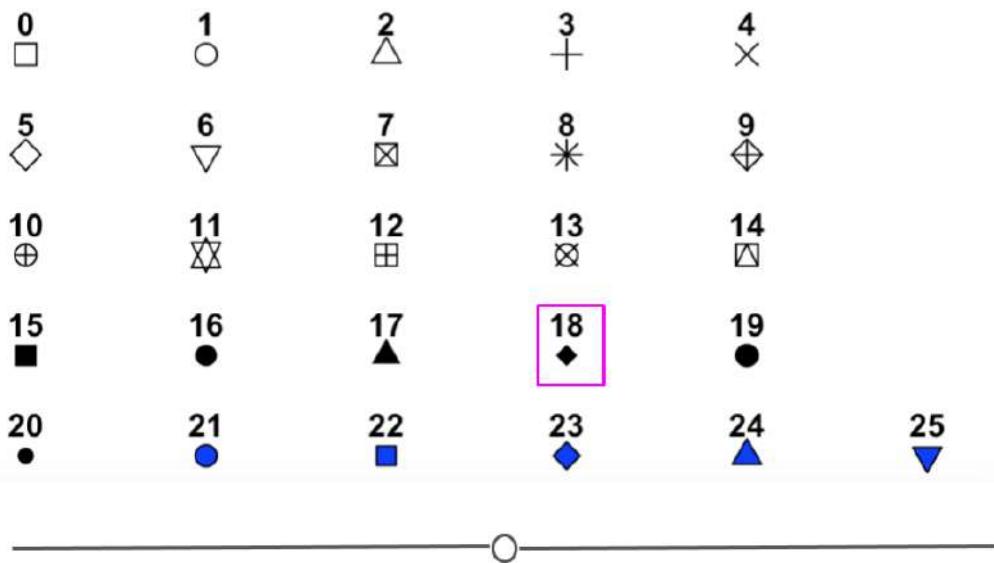
To change point **shape** relative to each diamond's clarity value, specify **shape** **within** **aes()**



mapping clarity to shape

Here, while the mapping occurs correctly within ggplot2, we do get a warning message that discriminating more than six different shapes is difficult for the human eye. Thus, ggplot2 won't allow more than six different shapes on a plot. This suggests that while you *can* do something, it's not always the *best* to do that thing. Here, with more than six levels of clarity, it's best to stick to mapping this variable to color as we did initially.

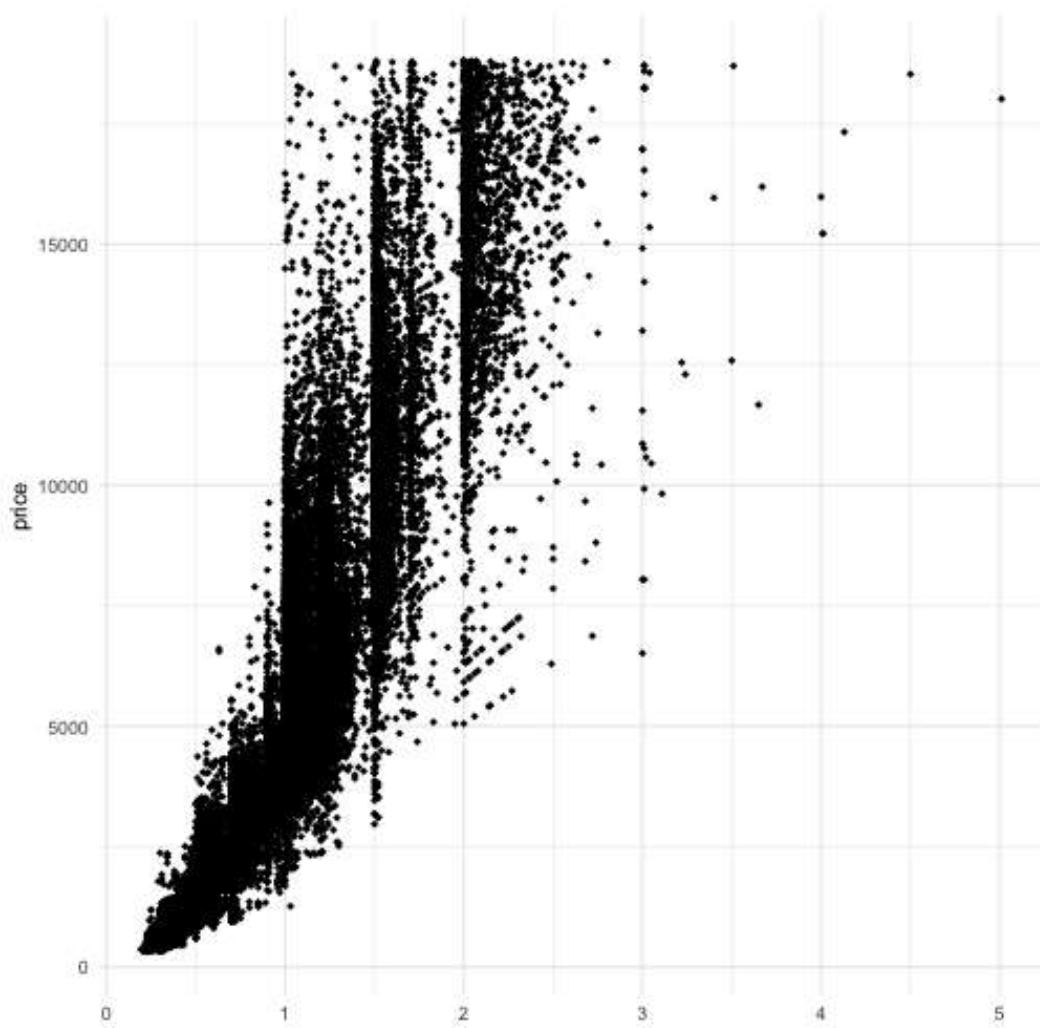
To manually specify a shape for all the points on your plot, you would specify it outside of **aes** using one of the twenty-five different shape options available:



options for points in ggplot2's shape

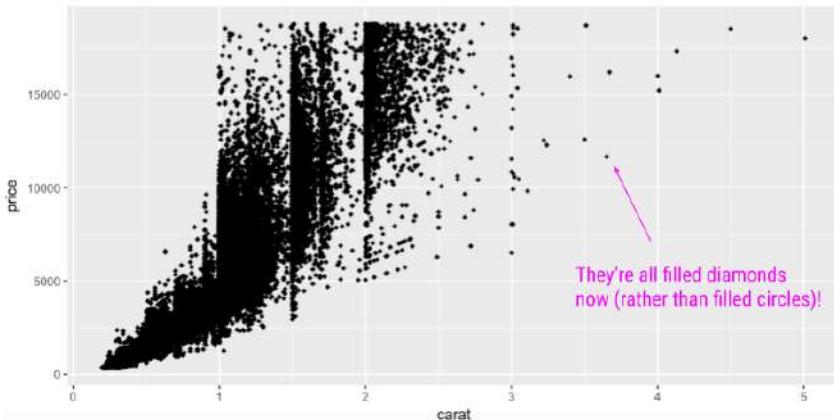
For example, to plot all of the points on the plot as filled diamonds (it is a dataset about diamonds after all...), you would specify shape '18':

```
# global control of point shape outside aes
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price), shape = 18)
```



plot of chunk unnamed-chunk-16

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price), shape = 18)
```



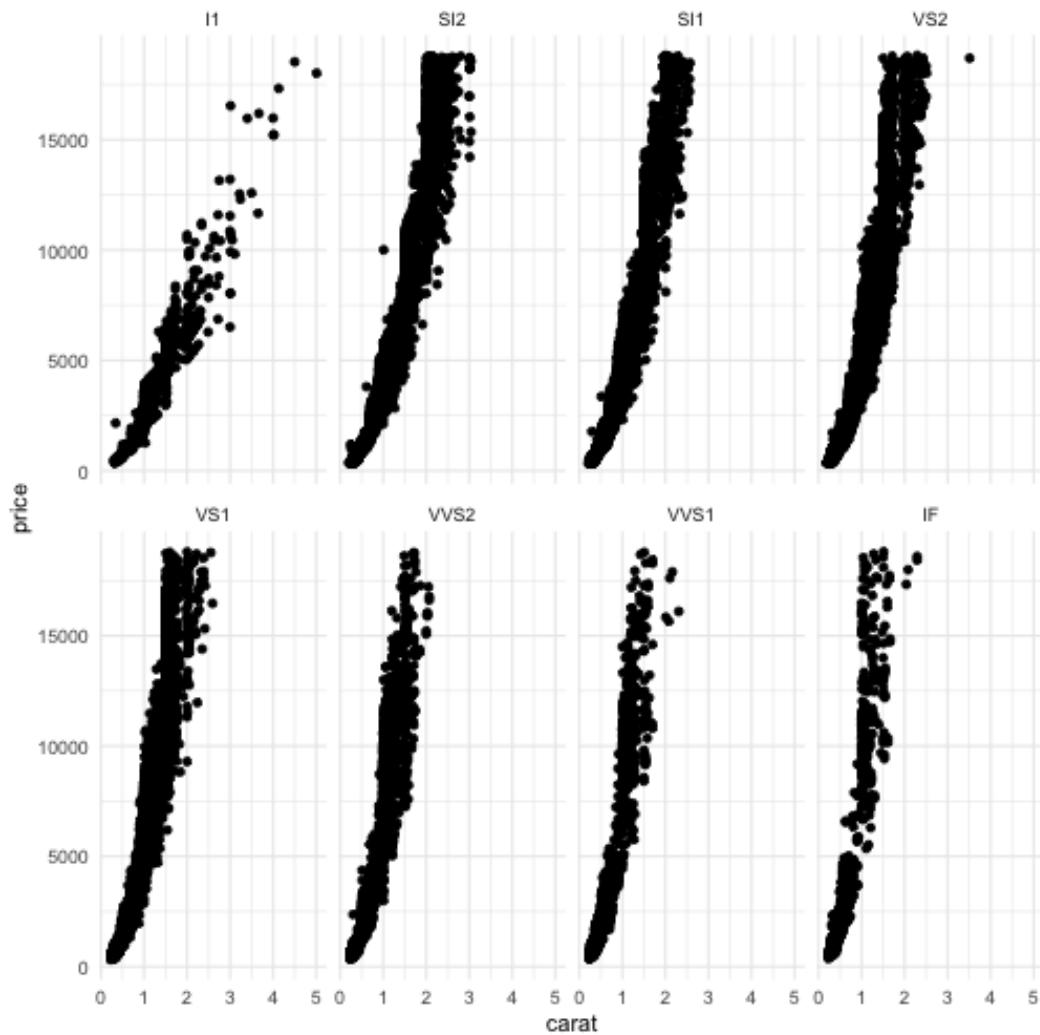
specifying filled diamonds as shape for all points manually

Facets

In addition to mapping variables to different aesthetics, you can also opt to use facets to help make sense of your data visually. Rather than plotting all the data on a single plot and visually altering the point size or color of a third variable in a scatterplot, you could break each level of that third variable out into a separate subplot. To do this, you would use facetting. Facetting is particularly helpful for looking at categorical variables.

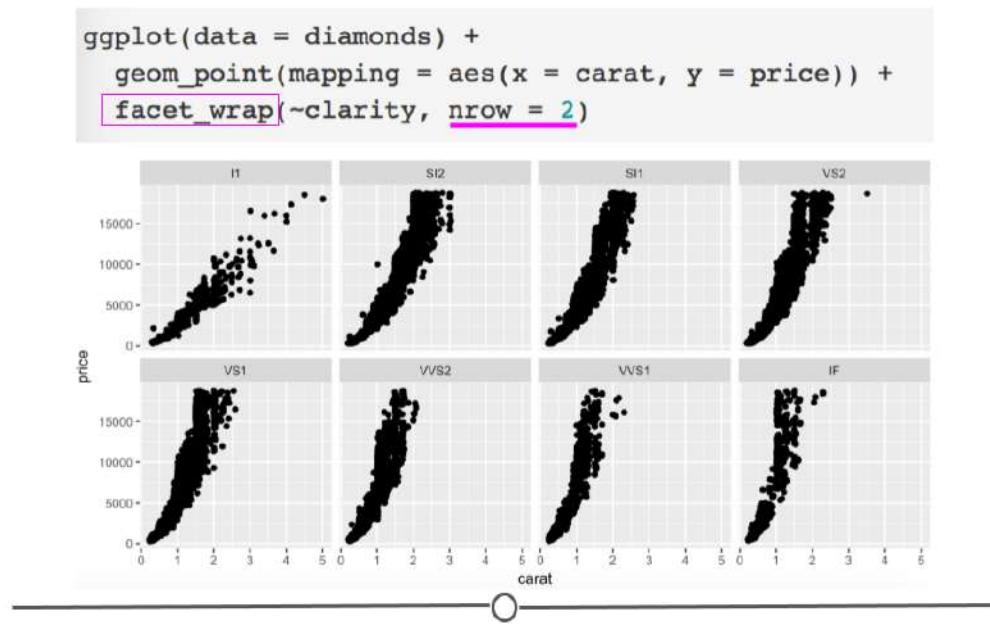
To use facetting, you would add an additional layer (+) to your code and use the `facet_wrap()` function. Within facet wrap, you specify the variable by which you want your subplots to be made:

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price)) +
  # facet by clarity
  facet_wrap(~clarity, nrow = 2)
```



plot of chunk unnamed-chunk-17

Here, read the tilde as the word “by”. Specifically here, we want a scatterplot of the relationship between carat and price and we want it faceted (broken down) by (\sim) clarity.



`facet_wrap` breaks plots down into subplots

Now, we have eight different plots, one for each level of clarity, where we can see the relationship between diamond carats and price.

You'll note here we've opted to specify that we want 2 rows of subplots (`nrow = 2`). You can play around with the number of rows you want in your output to customize how your output plot appears.

Geoms

Thus far in this lesson we've only looked at scatterplots, which means we've only called `geom_point`. However, there are *many* additional geoms that we could call to generate different plots. Simply, a *geom* is just a shape we use to represent the data. In the case of scatterplots, they don't *really* use a geom since each actual point is plotted individually. Other plots, such as the boxplots, barplots, and histograms we described in previous lessons help to summarize or represent the data in a meaningful way, without plotting each individual point. The shapes used in these different types of plots to represent what's going on in the data is that plot's geom.

To see exactly what we mean by geoms being "shapes that represent the data", let's keep using the `diamonds` dataset, but instead of looking at the relationship between two numeric

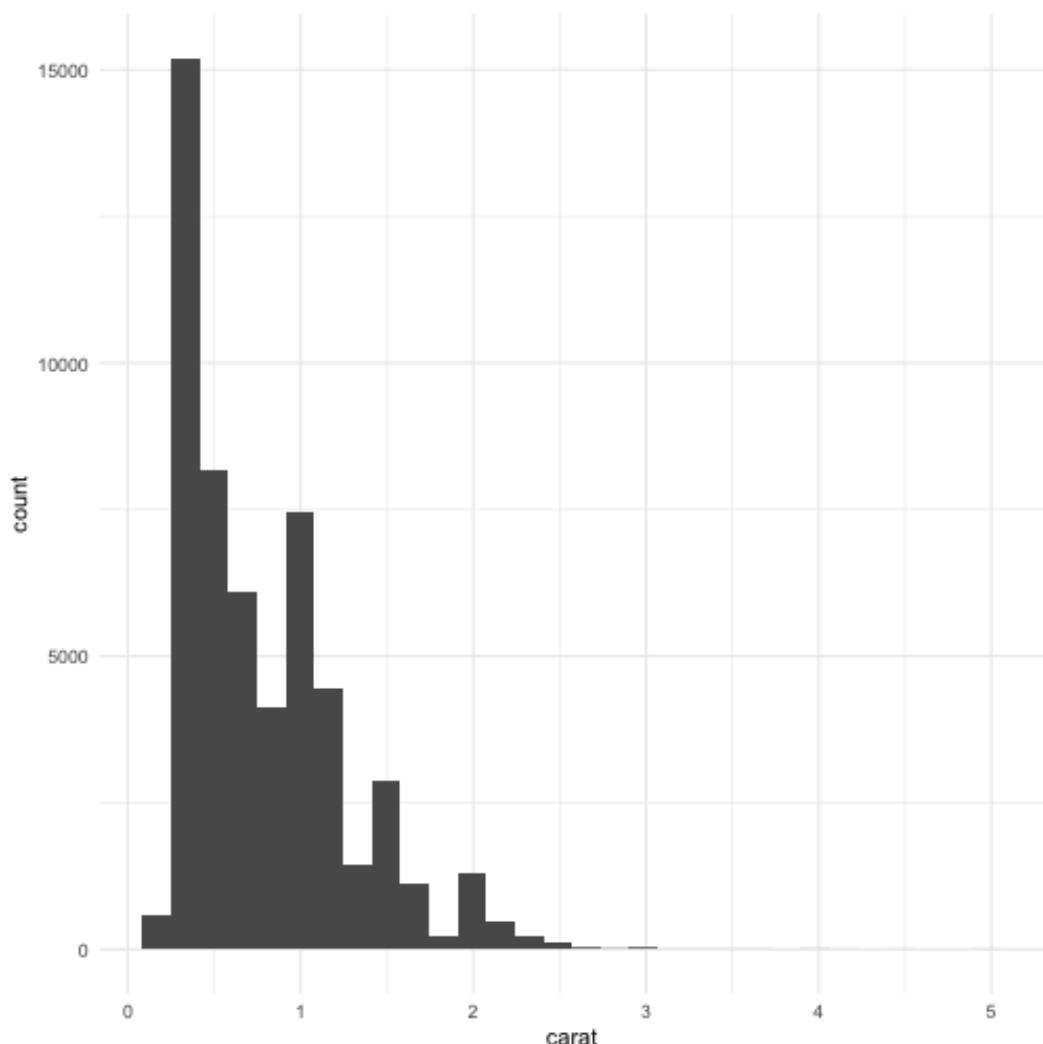
variables in a scatterplot, let's take a step back and take a look at a single numeric variable using a histogram.

Histograms: `geom_histogram`

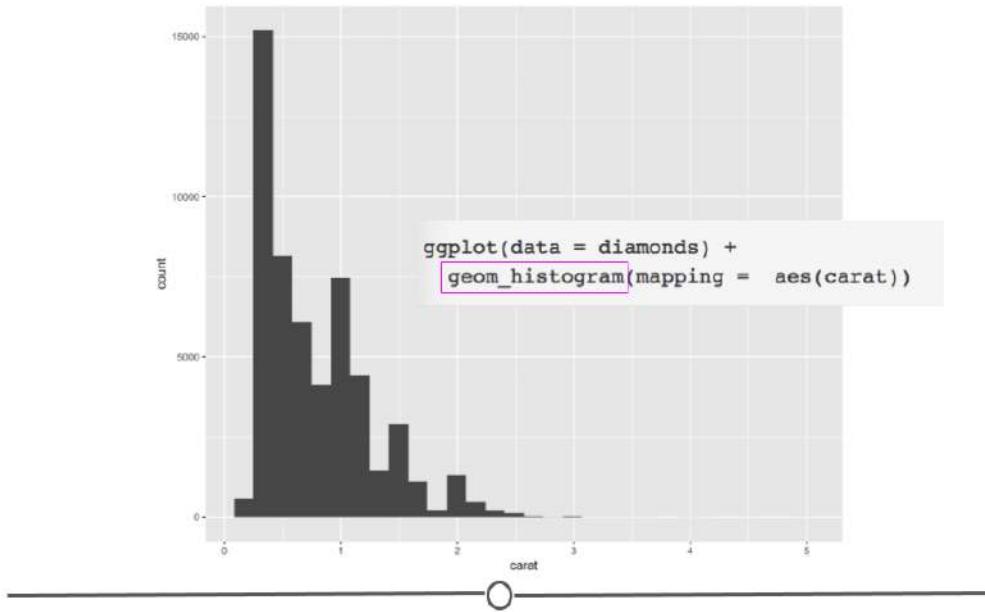
To review, histograms allow you to quickly visualize the range of values your variable takes and the shape of your data. (Are all the numbers clustered around center? Or, are they all at the extremes of the range? Somewhere in between? The answers to these questions describe the “shape” of the values of your variable.)

For example, if we wanted to see what the distribution of carats was for these data, we could do the following.

```
# change geom_ to generate histogram
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(carat))
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



plot of chunk unnamed-chunk-18



histogram of carat shows range and shape

The code follows what we've seen so far in this lesson; however, we've now called `geom_histogram` to specify that we want to plot a histogram rather than a scatterplot.

Here, the rectangular boxes on the plot are geoms (shapes) that represent the number of diamonds that fall into each bin on the plot. Rather than plotting each individual point, histograms use rectangular boxes to summarize the data. This summarization helps us quickly understand what's going on in our dataset.

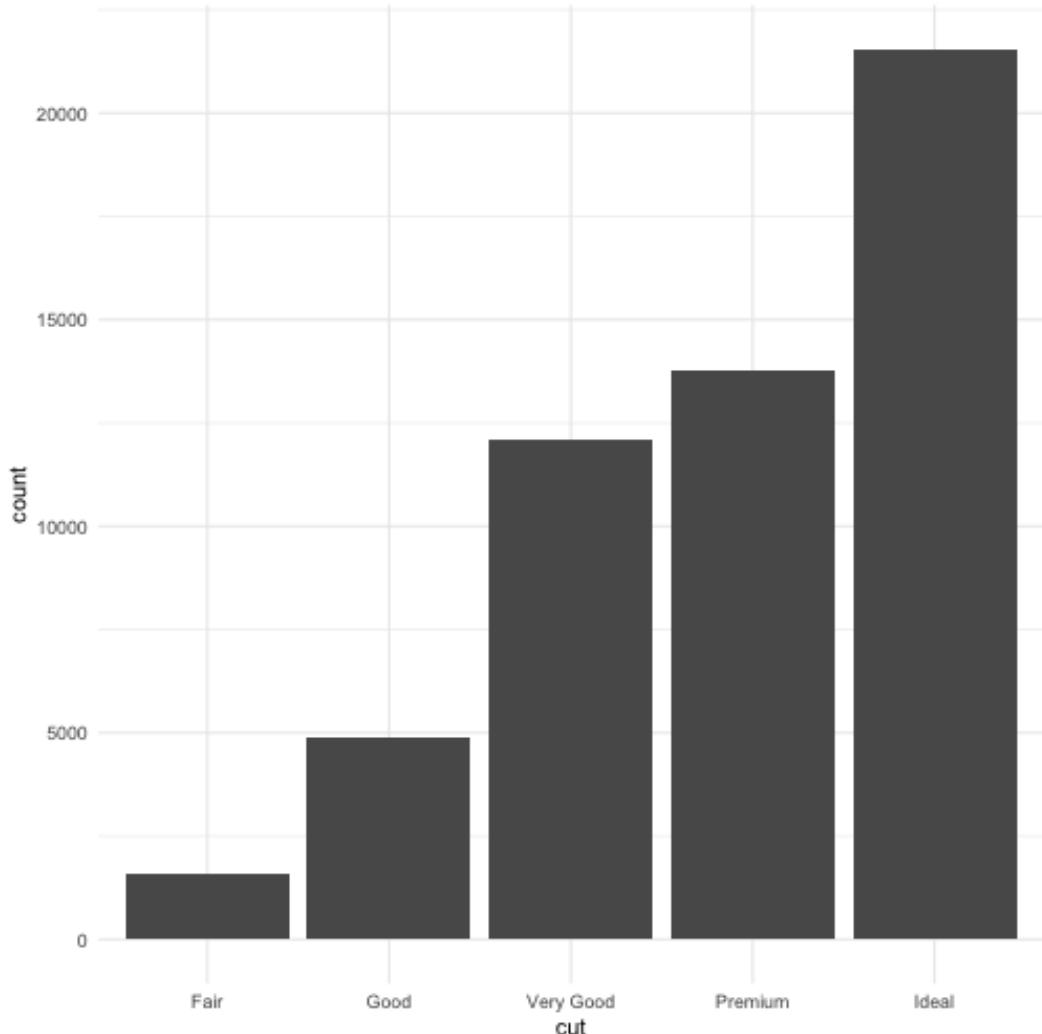
Specifically here, we can quickly see that most of the diamonds in the dataset are less than 1 carat. This is not necessarily something we could be sure of from the scatterplots generated previously in this lesson (since some points could have been plotted directly on top of one another). Thus, it's often helpful to visualize your data in a number of ways when you first get a dataset to ensure that you understand the variables and relationships between variables in your dataset!

Barplots: `geom_bar`

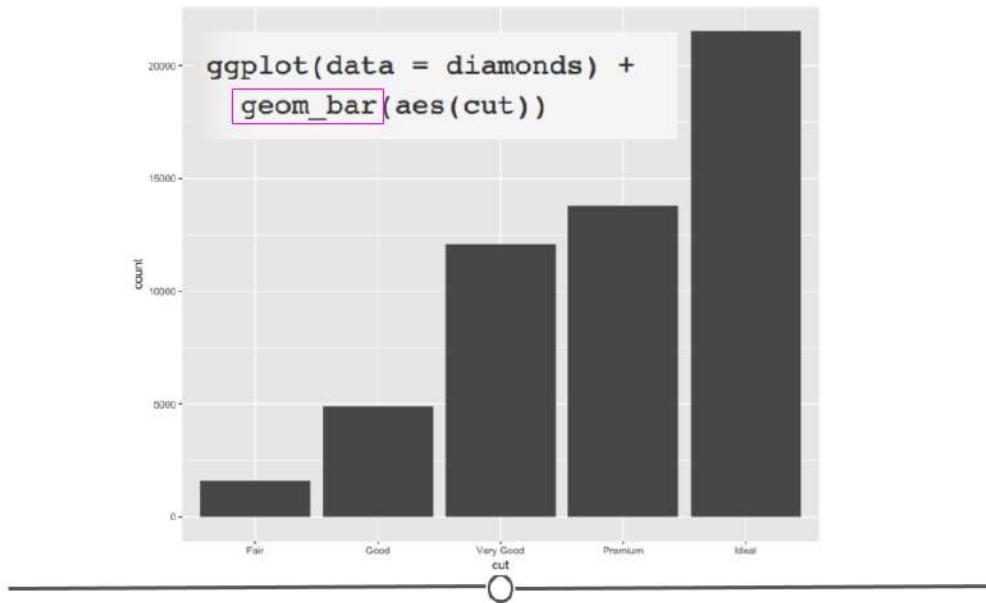
Barplots show the relationship between a set of numbers and a **categorical** variable. In the diamonds dataset, we may be interested in knowing how many diamonds there are of each

cut of diamonds. There are five categories for cut of diamond. If we make a barplot for this variable, we can see the number of diamonds in each category.

```
# geom_bar for bar plots
ggplot(data = diamonds) +
  geom_bar(mapping = aes(cut))
```



Again, the changes to the code are minimal. We are now interested in plotting the categorical variable `cut` and state that we want a bar plot, by including `geom_bar()`.



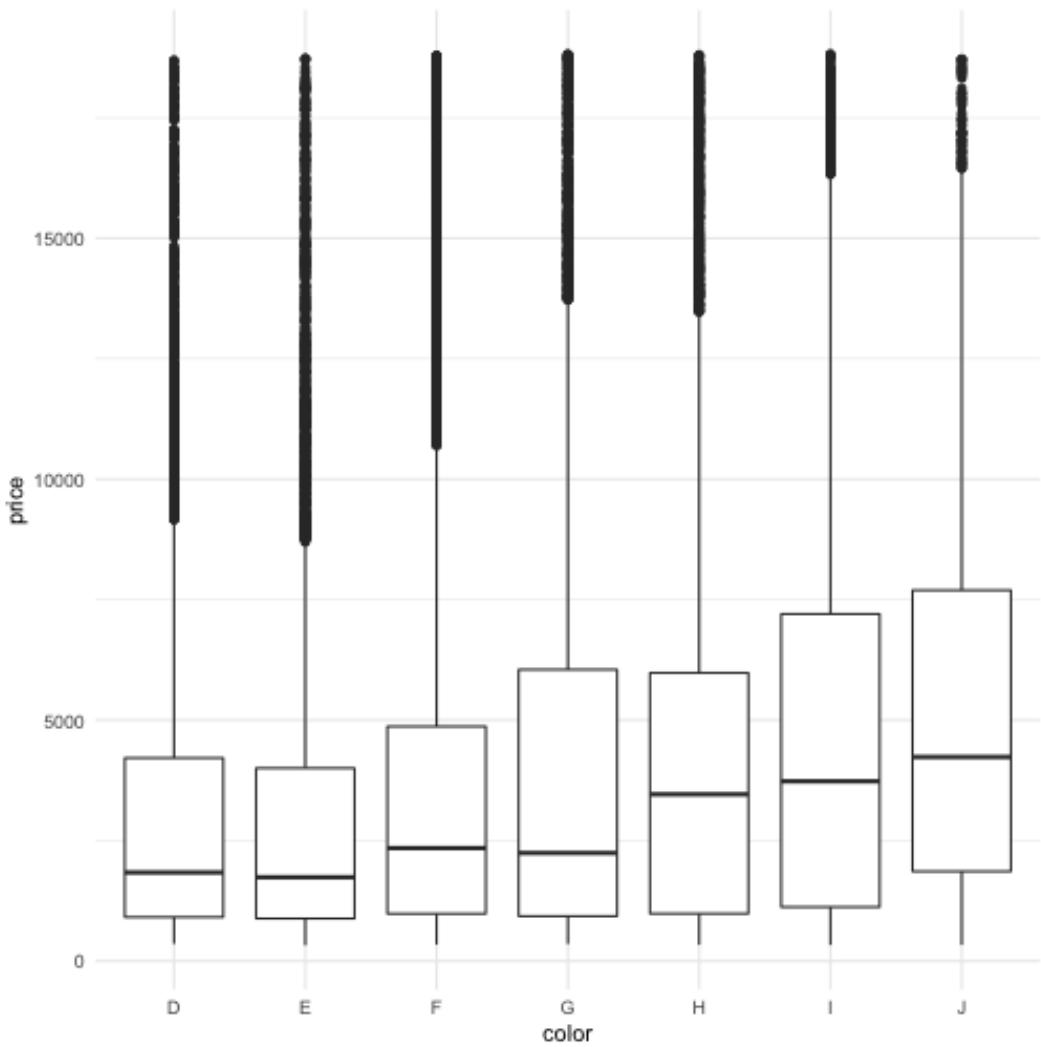
diamonds barplot

Here, we again use rectangular shapes to represent the data, but we're not showing the distribution of a single variable (as we were with `geom_histogram`). Rather, we're using rectangles to show the count (number) of diamonds within each category within `cut`. Thus, we need a different geom: `geom_bar`!

Boxplots: `geom_boxplot`

Boxplots provide a summary of a numerical variable across categories. For example, if you were interested to see how the price of a diamond (a numerical variable) changed across different diamond color categories (categorical variable), you may want to use a boxplot. To do so, you would specify that using `geom_boxplot`:

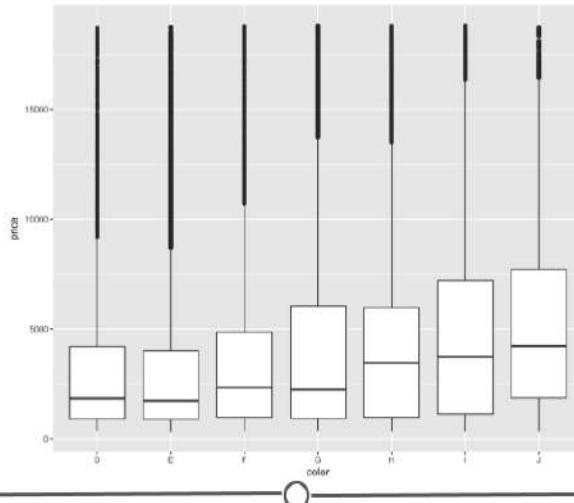
```
# geom_boxplot for boxplots
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(x = color, y = price))
```



plot of chunk unnamed-chunk-20

In the code, we see that again, we only have to change what variables we want to be included in the plot and the type of plot (or geom). We want to use `geom_boxplot()` to get a basic boxplot.

```
ggplot(data = diamonds) +  
  geom_boxplot(aes(x = color, y = price))
```

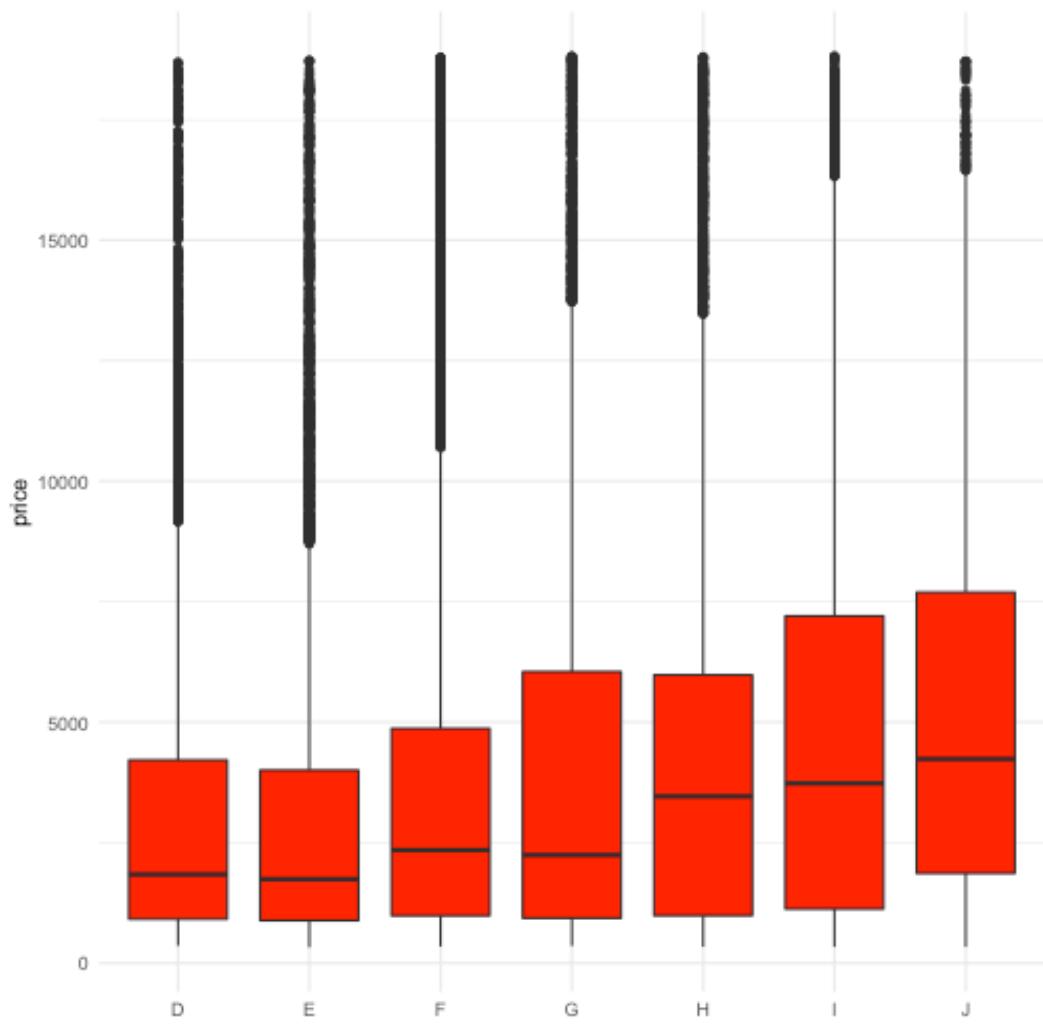


diamonds boxplot

In the figure itself we see that the median price (the black horizontal bar in the middle of each box represents the median for each category) increases as the diamond color increases from the worst category (J) to the best (D).

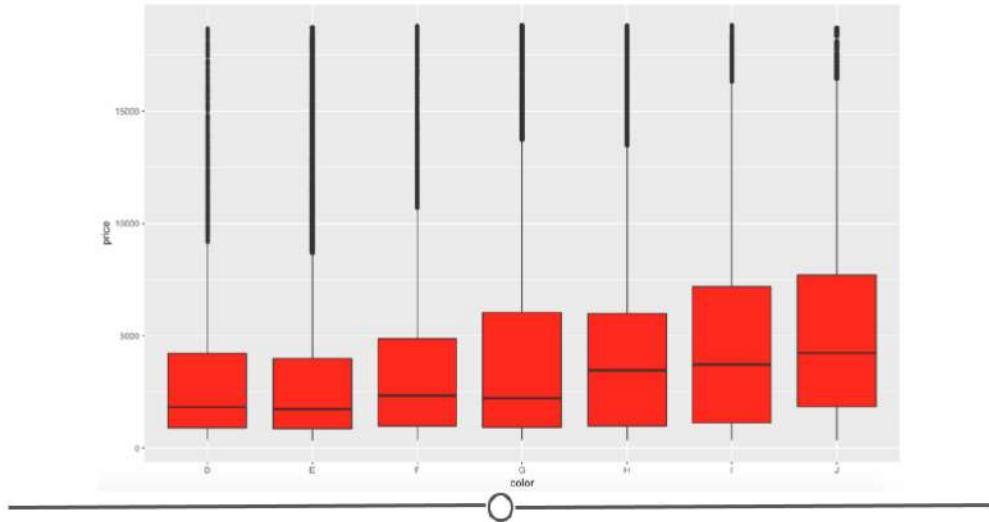
Now, if you wanted to change the color of this boxplot, it would just take a small addition to the code for the plot you just generated.

```
# fill globally changes bar color outside aes  
ggplot(data = diamonds) +  
  geom_boxplot(mapping = aes(x = color, y = price),  
               fill = "red")
```



plot of chunk unnamed-chunk-21

```
ggplot(data = diamonds) +  
  geom_boxplot(aes(x = color, y = price), fill = "red")
```

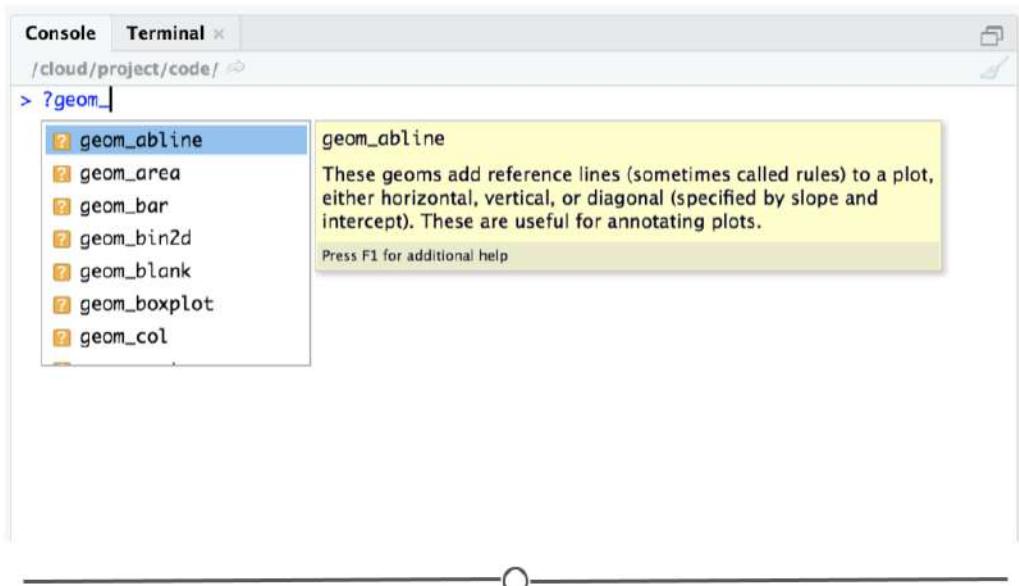


diamonds boxplot with red fill

Here, by specifying the color “red” in the `fill` argument, you’re able to change the plot’s appearance. In the next lesson, we’ll go deeper into the many ways in which a plot can be customized within `ggplot2`!

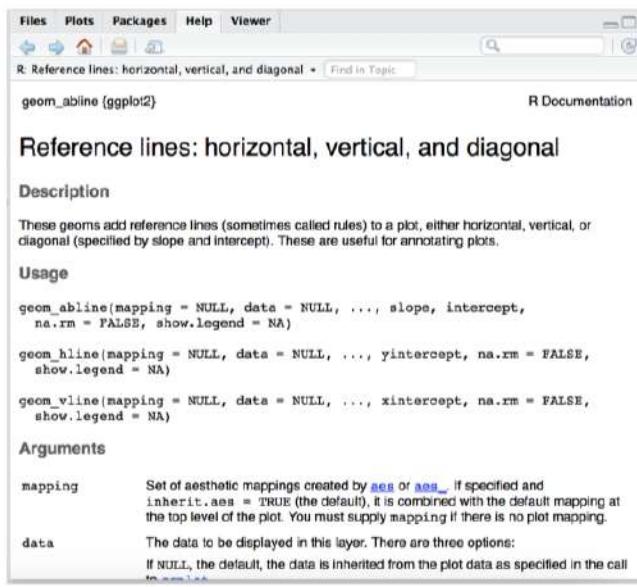
Other plots

While we’ve reviewed basic code to make a few common types of plots, there are a number of other plot types that can be made in `ggplot2`. These are listed in the [online reference material for ggplot2](#) or can be accessed through RStudio directly. To do so, you would type `?geom_` into the Console in RStudio. A list of geoms will appear. You can hover your cursor over any one of these to get a short description.



?geom in Console

Or, you can select a geom from this list and click enter. After selecting a geom, such as `geom_abline` and hitting 'Enter,' the help page for that geom will pop up in the 'Help' tab at bottom right. Here, you can find more detailed information about the selected geom.



[geom_abline help page](#)

EDA Plots

As mentioned previously, an important step after you've read your data into R and wrangled it into a tidy format is to carry out **Exploratory Data Analysis (EDA)**. EDA is the process of understanding the data in your dataset fully. To understand your dataset fully, you need a full understanding of the variables stored in your dataset, what information you have and what information you don't have (missingness!). To gain this understanding, we've discussed using packages like `skimr` to get a quick idea of what information is stored in your dataset. However, generating plots is another critical step in this process. We encourage you to use `ggplot2` to understand the distribution of each single variable as well as the relationship between each variable in your dataset.

In this process, using `ggplot2` defaults is totally fine. These plots do not have to be the most effective visualizations for communication, so you don't want to spend a ton of time making them visually perfect. Only spend as much time on these as you need to understand your data!

ggplot2: Customization

So far, we have walked through the steps of generating a number of different graphs (using different geoms) in ggplot2. We discussed the basics of mapping variables to your graph to customize its appearance or aesthetic (using size, shape, and color within `aes()`). Here, we'll build on what we've previously learned to really get down to how to customize your plots so that they're as clear as possible for communicating your results to others.

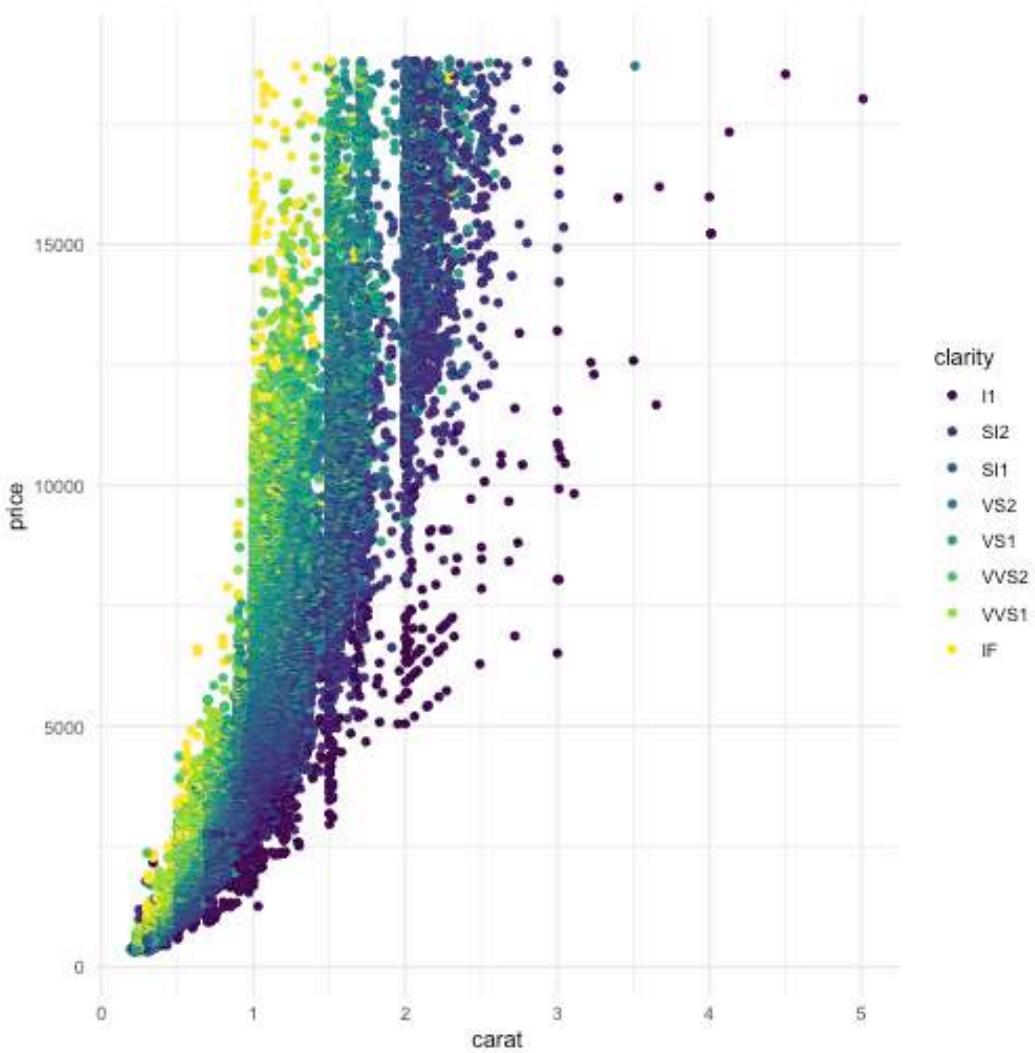
The skills learned in this lesson will help take you from generating exploratory plots that help *you* better understand your data to explanatory plots – plots that help you communicate your results *to others*. We'll cover how to customize the colors, labels, legends, and text used on your graph.

Since we're already familiar with it, we'll continue to use the `diamonds` dataset that we've been using to learn about ggplot2.

Colors

To get started, we'll learn how to control color across plots in ggplot2. Previously, we discussed using color within `aes()` on a scatterplot to automatically color points by the clarity of the diamond when looking at the relationship between price and carat.

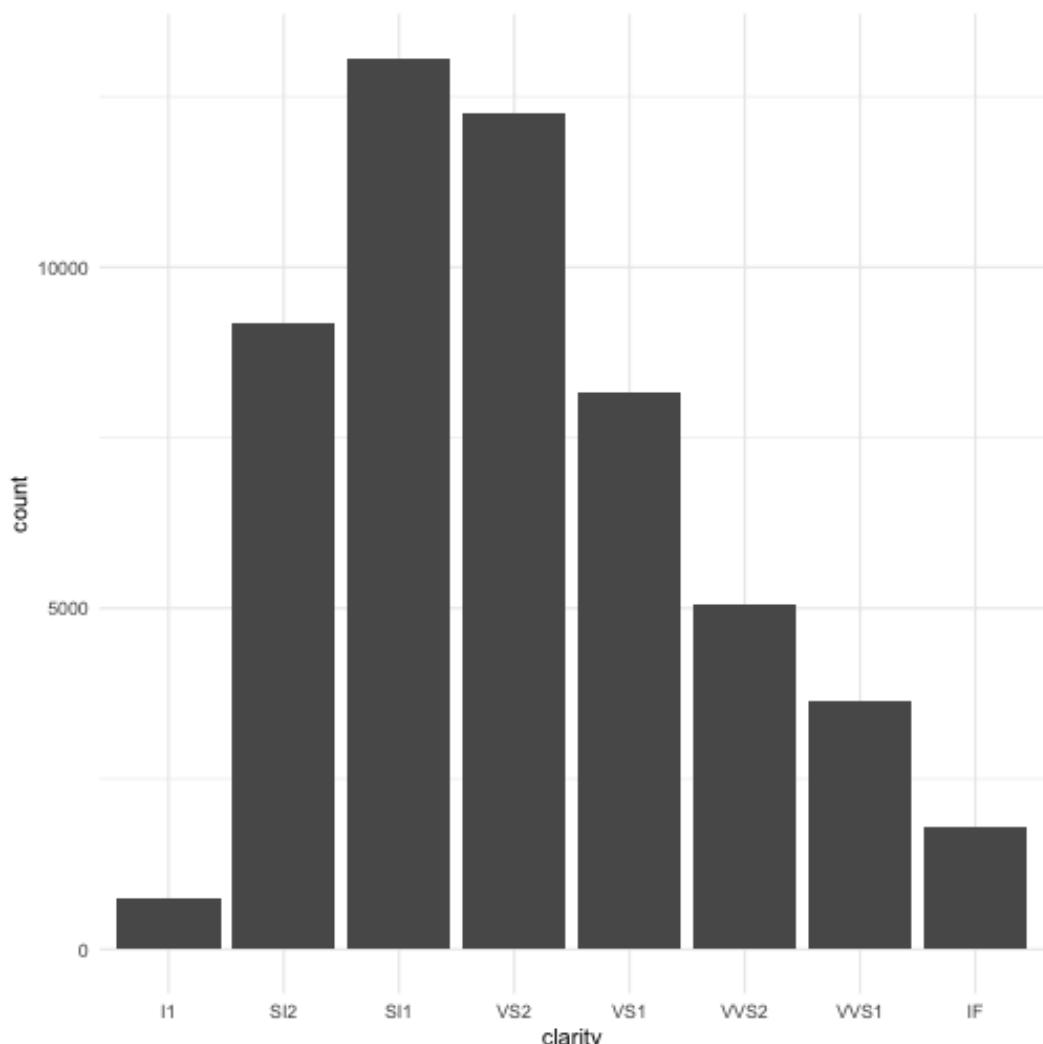
```
ggplot(diamonds) +  
  geom_point(mapping = aes(x = carat, y = price, color = clarity))
```

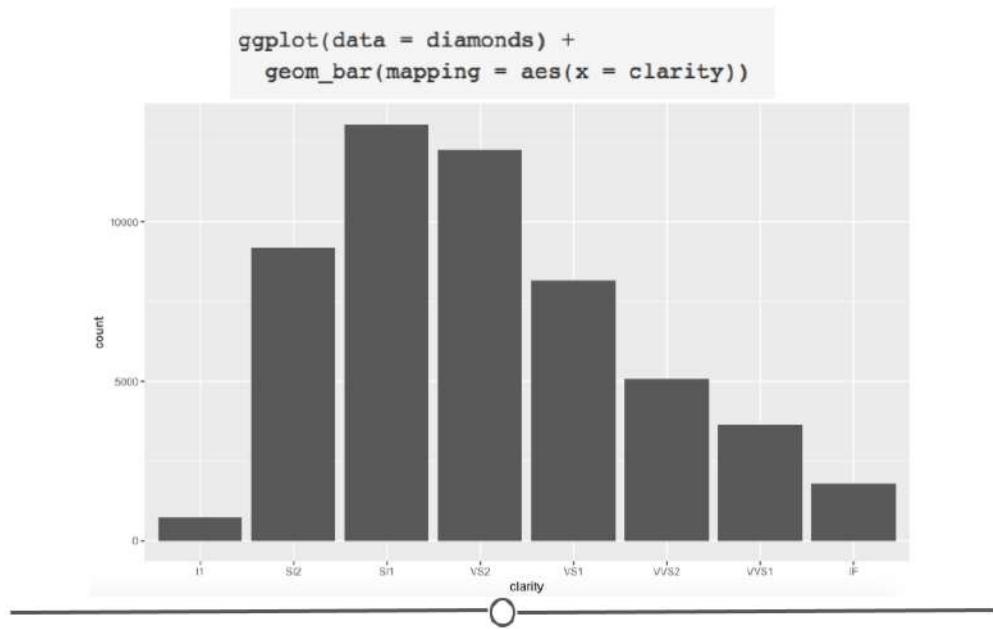


plot of chunk unnamed-chunk-22

However, what if we wanted to carry this concept over to a bar plot and look at how many diamonds we have of each clarity group?

```
# generate bar plot
ggplot(diamonds) +
  geom_bar(aes(x = clarity))
```



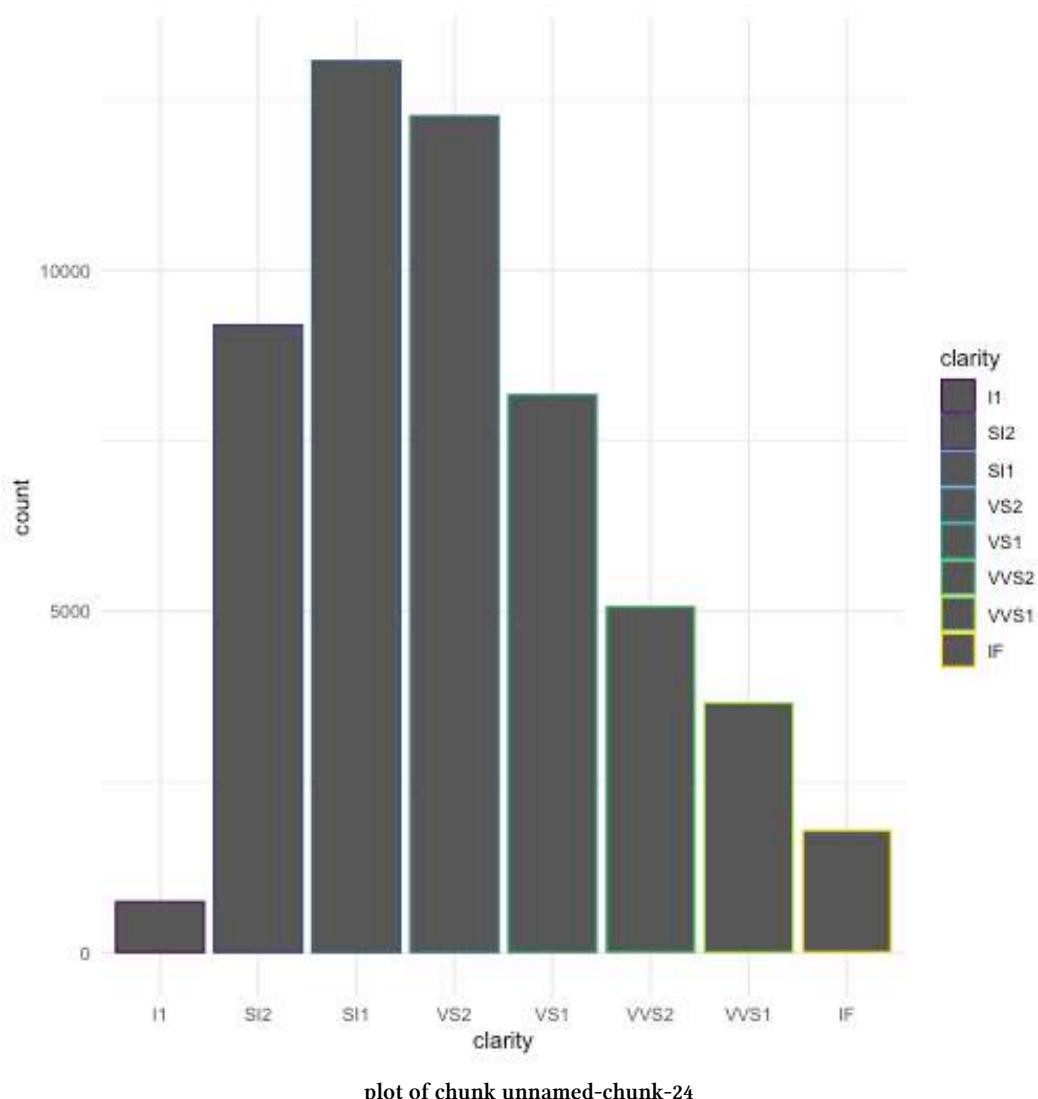


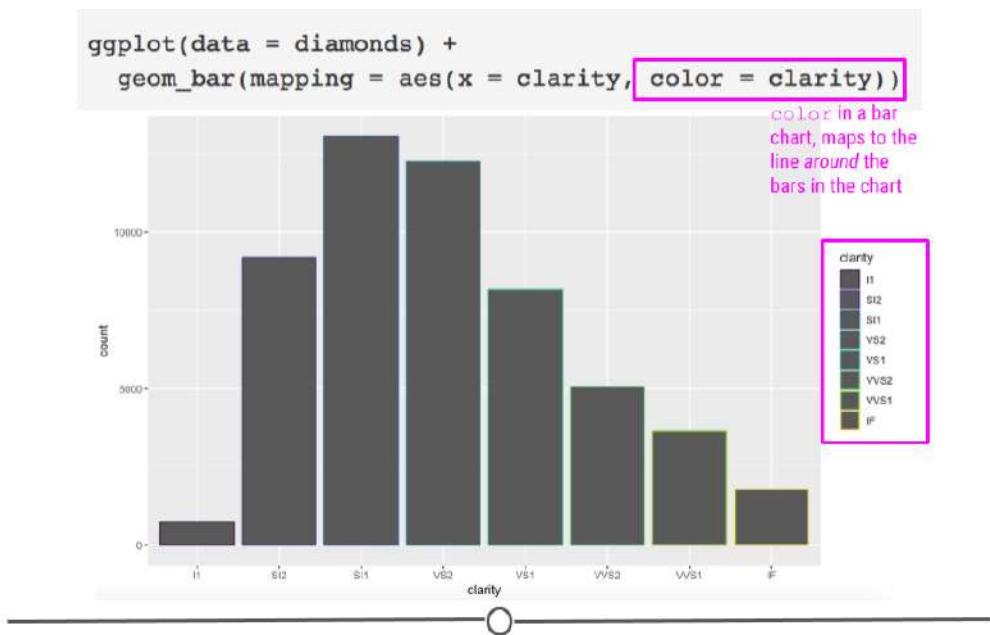
diamonds broken down by clarity

As a general note, we've stopped including `data =` and `mapping =` here within our code. We included it so far to be explicit; however, in code you see in the world, the names of the arguments will typically be excluded and we want you to be familiar with code that appears as you see above.

OK, well that's a start since we see the breakdown, but all the bars are the same color. What if we adjusted color within `aes()`?

```
# color changes outline of bar  
ggplot(diamonds) +  
  geom_bar(aes(x = clarity, color = clarity))
```

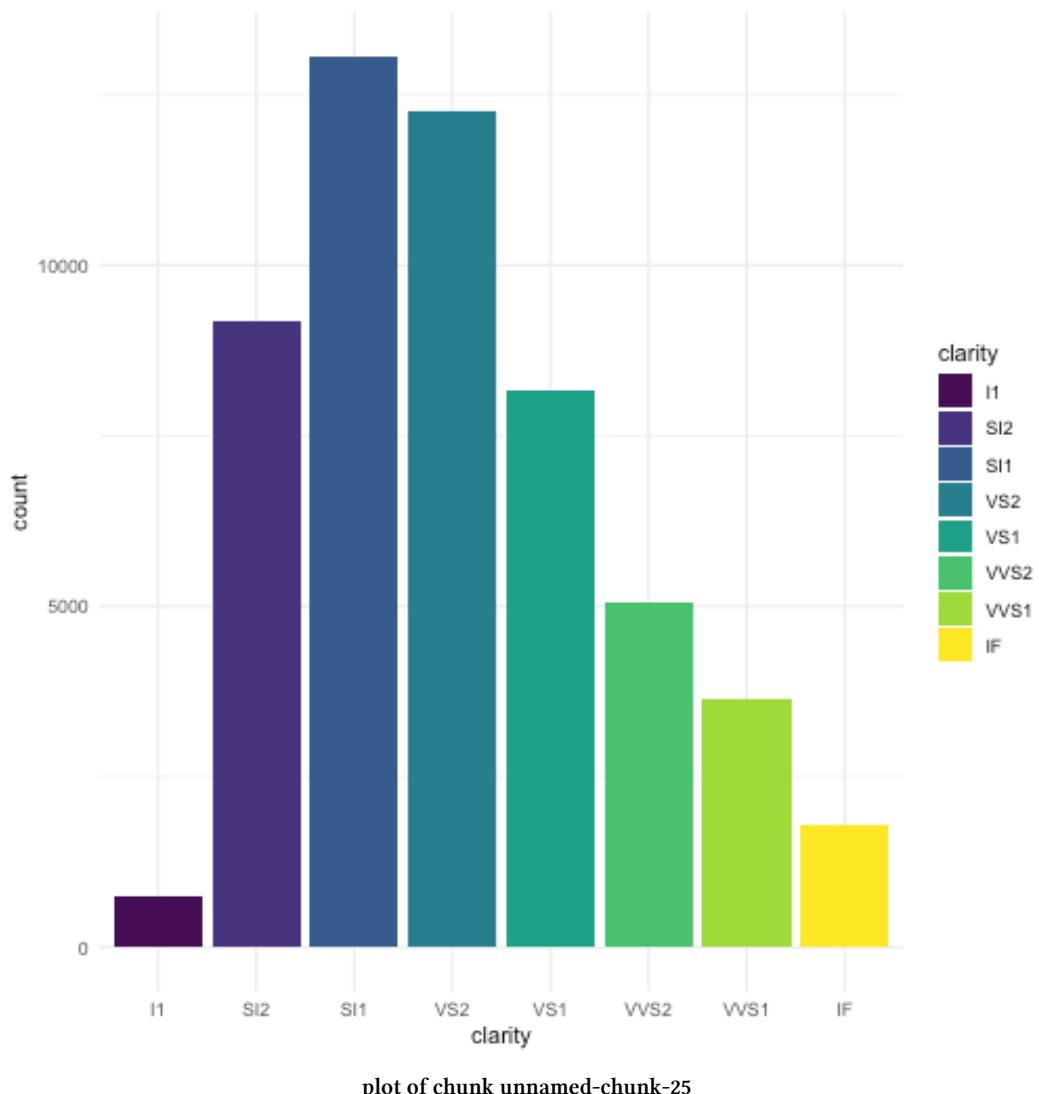


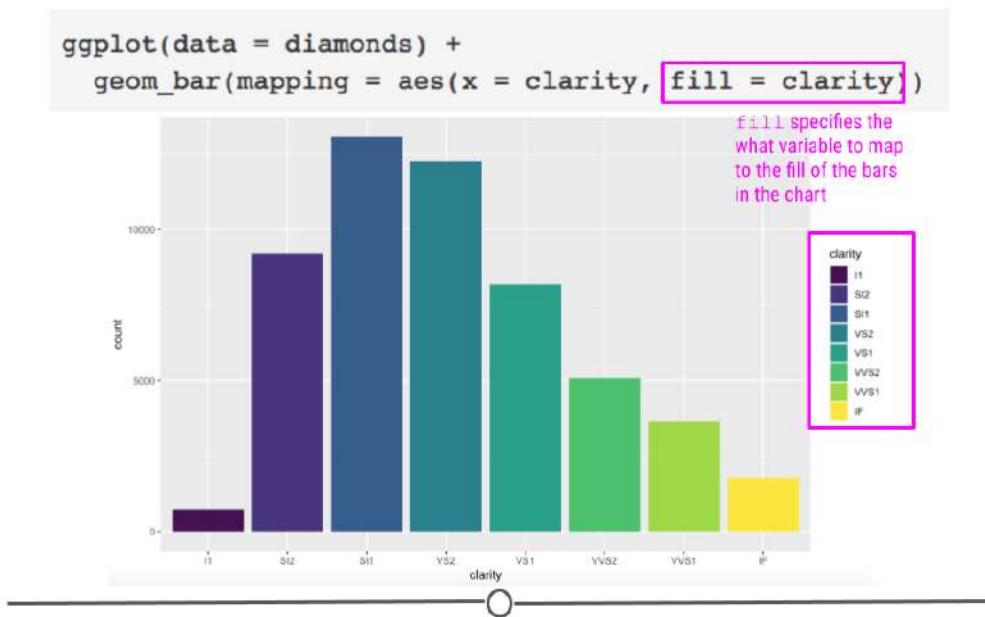


color does add color but around the bars

As expected, color added a legend for each level of clarity; however, it colored the lines around the bars on the plot, rather than the bars themselves. In order to color the bars themselves, you want to specify the more helpful argument `fill`:

```
# use fill to color bars
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = clarity))
```



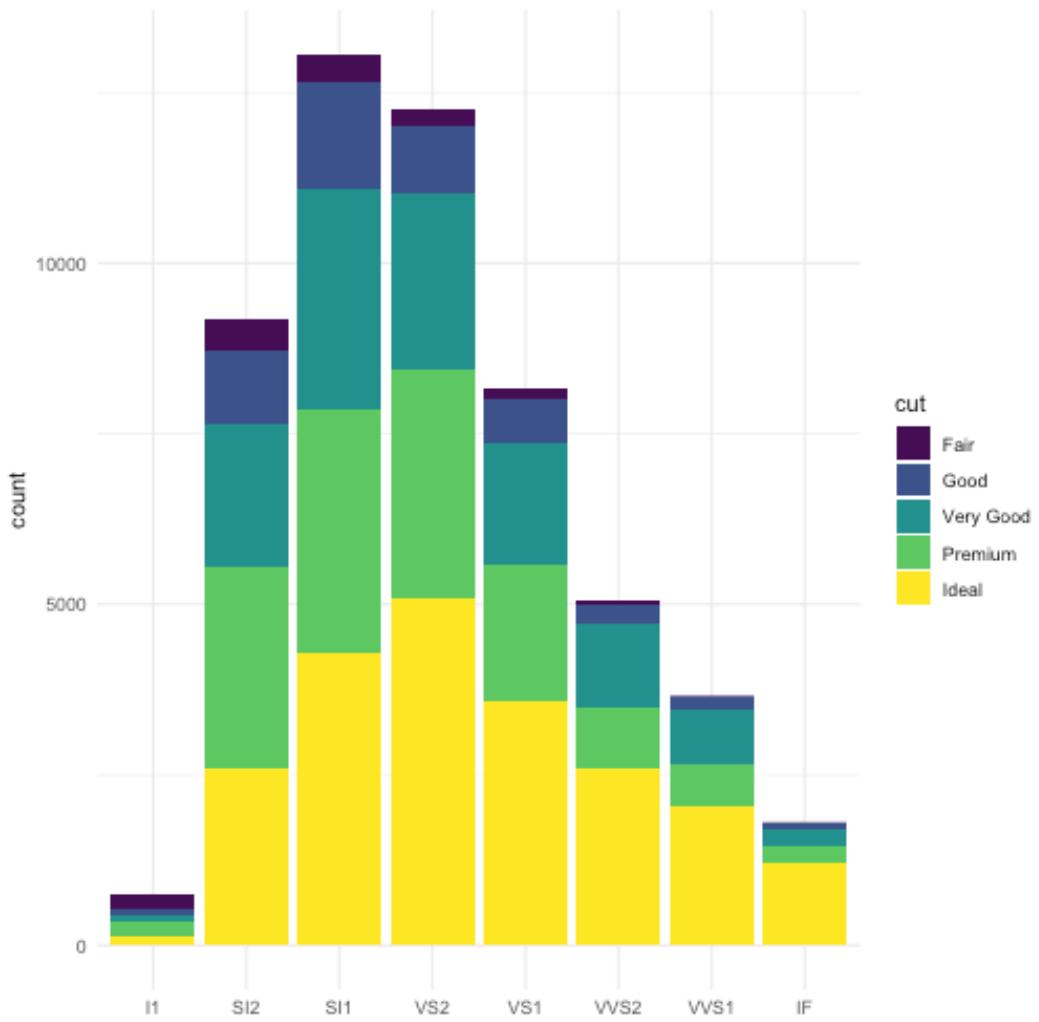


fill automatically colors the bars

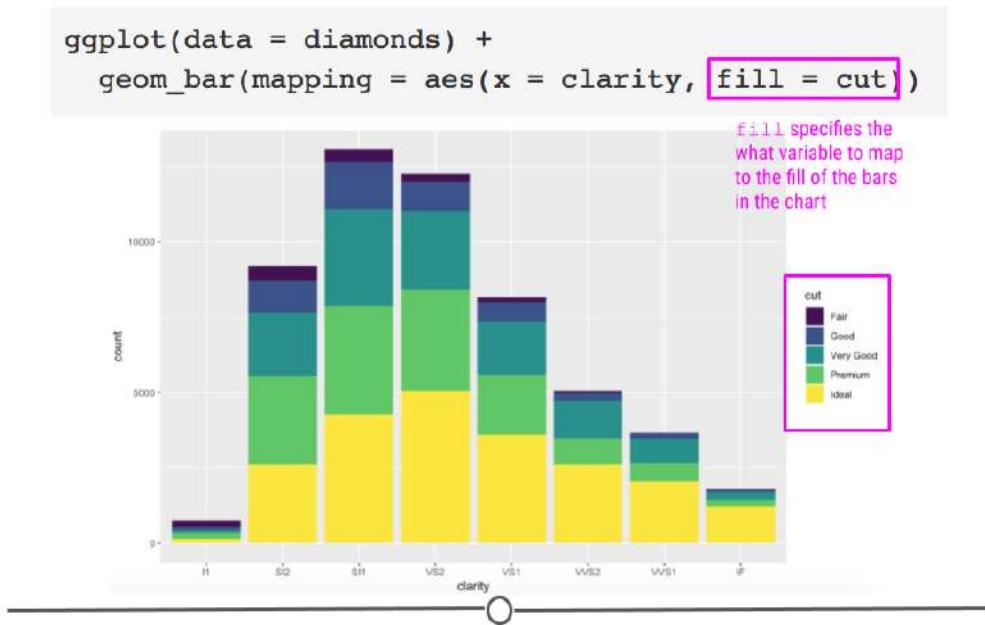
Great! We now have a plot with bars of different colors, which was our first goal! However, adding colors here, while maybe making the plot prettier doesn't actually give us any more information. We can see the same pattern of which clarity is most frequent among the diamonds in our dataset like we could see in the first plot we made.

Color is particularly helpful here, however, if we wanted to map a *different* variable onto each bar. For example, what if we wanted to see the breakdown of diamond "cut" within each "clarity" bar?

```
# fill by separate variable (cut) = stacked bar chart
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut))
```



plot of chunk unnamed-chunk-26

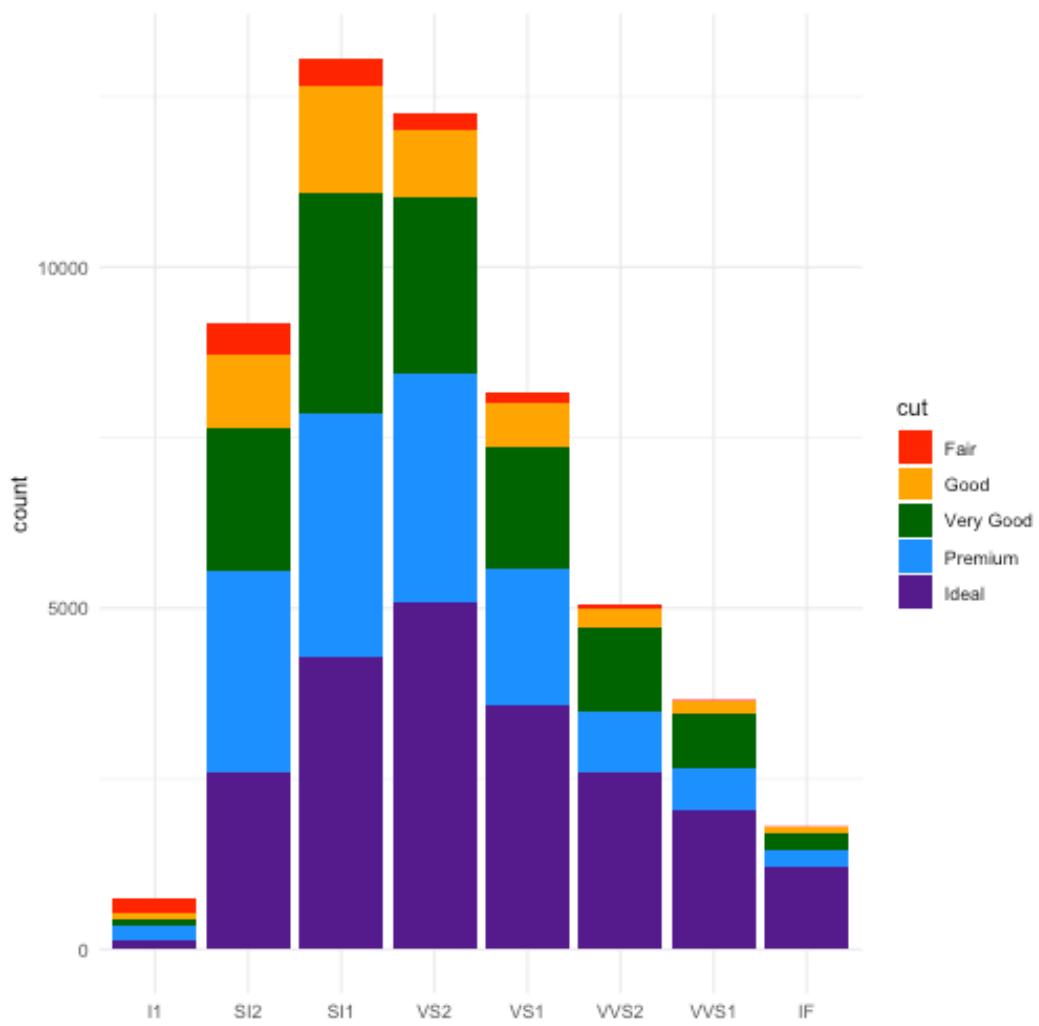


mapping a different variable to fill provides new information

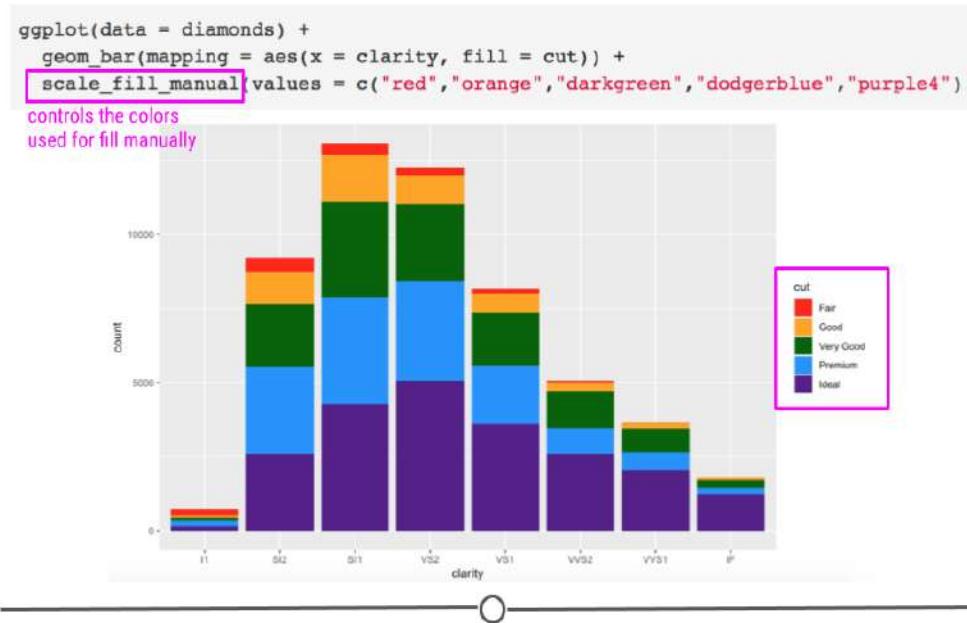
Now we're getting some new information! We can see that each level in clarity appears to have diamonds of all levels of cut. Color here has really helped us understand more about the data.

But what if we were going to present these data? While there is no comparison between red and green (which is good!), there is a fair amount of yellow in this plot. Some projectors don't handle projecting yellow well, and it will show up too light on the screen. To avoid this, let's manually change the colors in this bar chart! To do so we'll add an additional layer to the plot using `scale_fill_manual`.

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut)) +
  # manually control colors used
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple"))
```



plot of chunk unnamed-chunk-27



manually setting colors using `scale_fill_manual`

Here, we've specified five different colors within the `values` argument of `scale_fill_manual()`, one for each cut of diamond. The names of these colors can be specified using the names explained on the third page of the cheatsheet [here](#). (Note: There are other ways to specify colors within R. Explore the details in that cheatsheet to better understand the various ways!)

Additionally, it's important to note that here we've used `scale_fill_manual()` to adjust the color of what was mapped using `fill = cut`. If we had colored our chart using `color` within `aes()`, there is a different function called `scale_color_manual`. This makes good sense! You use `scale_fill_manual()` with `fill` and `scale_color_manual()` with `color`. Keep that in mind as you adjust colors in the future!

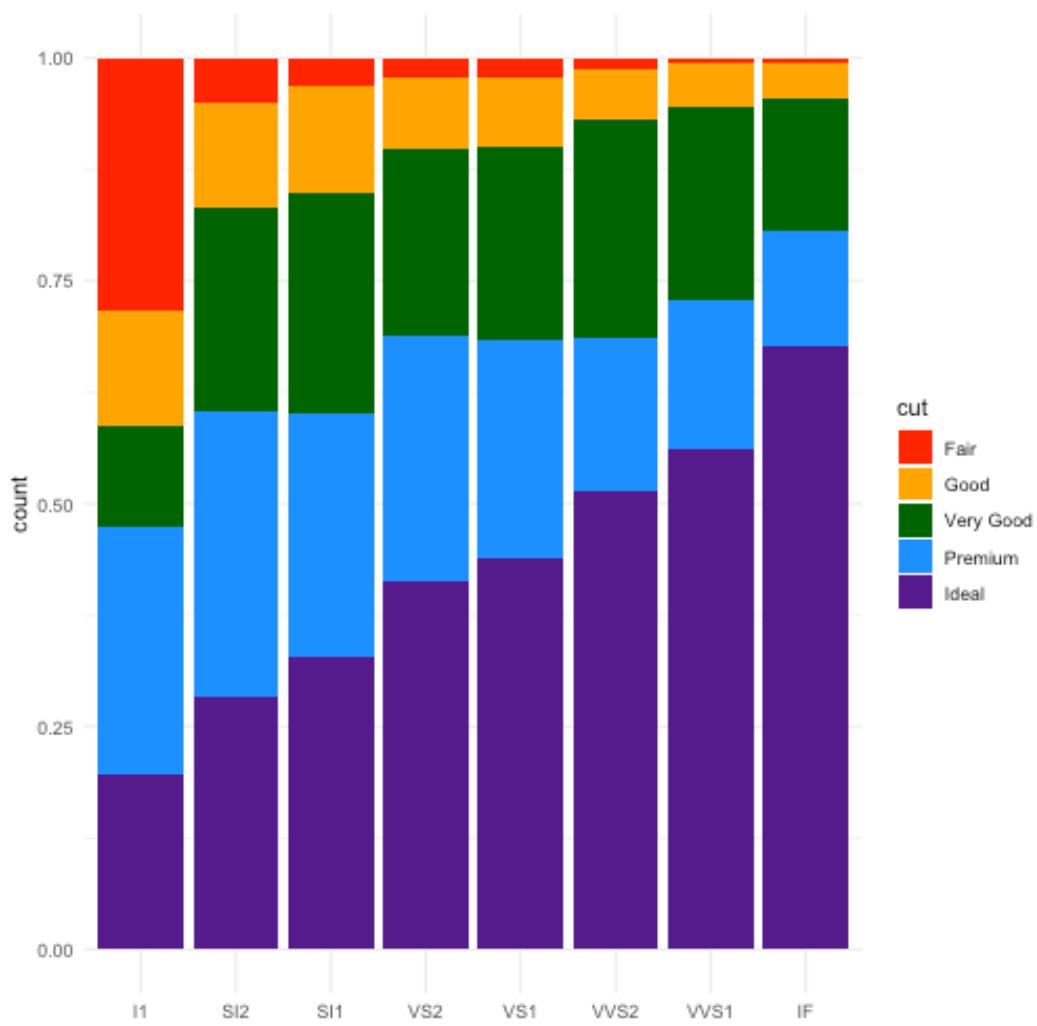
Now that we have some sense of which clarity is most common in our diamonds dataset and now that we are able to successfully specify the colors we want manually in order to make this plot useful for presentation, what if we wanted to compare the proportion of each cut across the different clarities? Currently, that's difficult because there is a different number within each clarity. In order to compare the proportion of each cut we have to use **position adjustment**.

What we've just generated is a **stacked bar chart**. It's a pretty good name for this type of

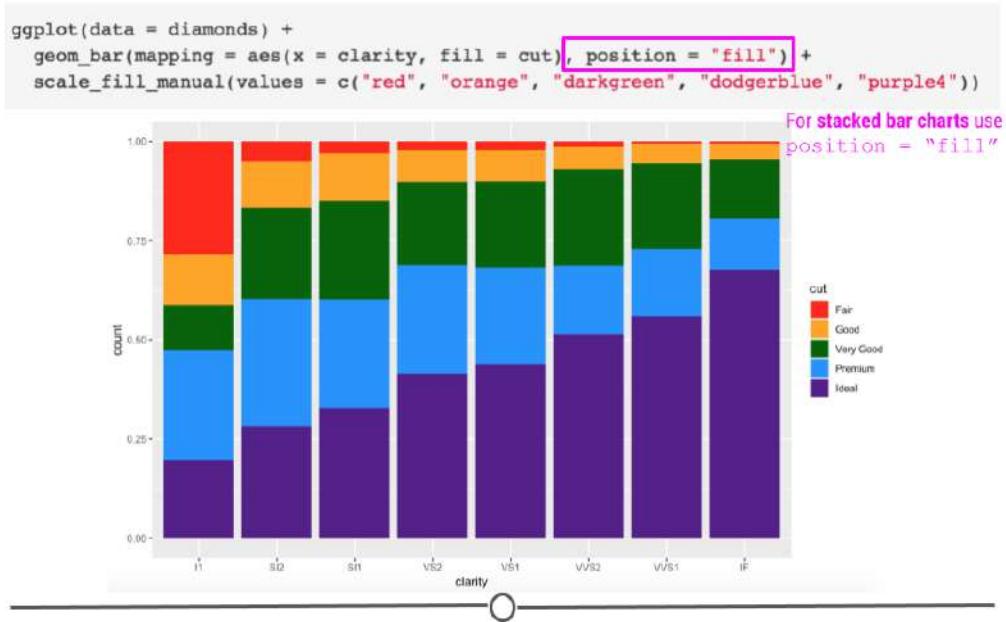
chart as the bars for cut are all stacked on top of one another. If you don't want a stacked bar chart you could use one of the other position options: `identity`, `fill`, or `dodge`.

Returning to our question about proportion of each cut within each clarity group, we'll want to use `position = "fill"` within `geom_bar()`. Building off of what we've already done:

```
ggplot(diamonds) +  
  # fill scales to 100%  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple"))
```



plot of chunk unnamed-chunk-28



`position = "fill"` allows for comparison of proportion across groups

Here, we've specified how we want to adjust the position of the bars in the plot. Each bar is now of equal height and we can compare each colored bar across the different clarities. As expected, we see that among the best clarity group (IF), we see more diamonds of the best cut ("Ideal")!

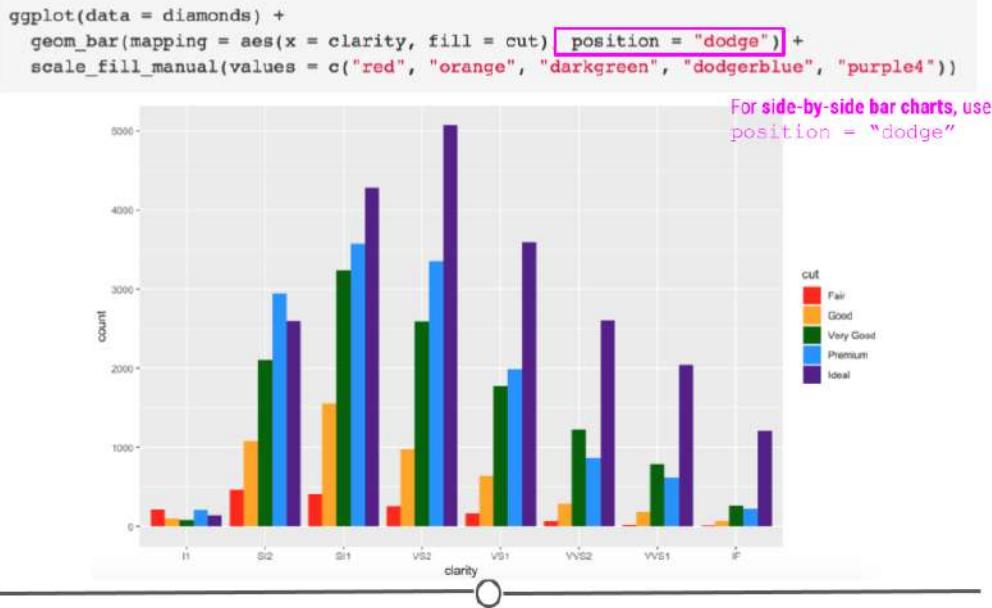
Briefly, we'll take a quick detour to look at `position = "dodge"`. This position adjustment places each object *next to one another*. This will not allow for easy comparison across groups, as we just saw with the last group but will allow values within each clarity group to be visualized.

```
ggplot(diamonds) +  

  # dodge rather than stack produces grouped bar plot  

  geom_bar(aes(x = clarity, fill = cut), position = "dodge") +  

  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4"))
```



`position = "dodge"` helps compare values within each group

Unlike in the first plot where we specified `fill = cut`, we can actually see the relationship between each cut within the lowest clarity group (I1). Before, when the values were stacked on top of one another, we were not able to visually see that there were more “Fair” and “Premium” cut diamonds in this group than the other cuts. Now, with `position = "dodge"`, this information is visually apparent.

Note: `position = "identity"` is not very useful for bars, as it *places each object exactly where it falls within the graph*. For bar charts, this will lead to *overlapping bars*, which is not visually helpful. However, for scatterplots (and other 2-Dimensional charts), this is the default and is exactly what you want.

Labels

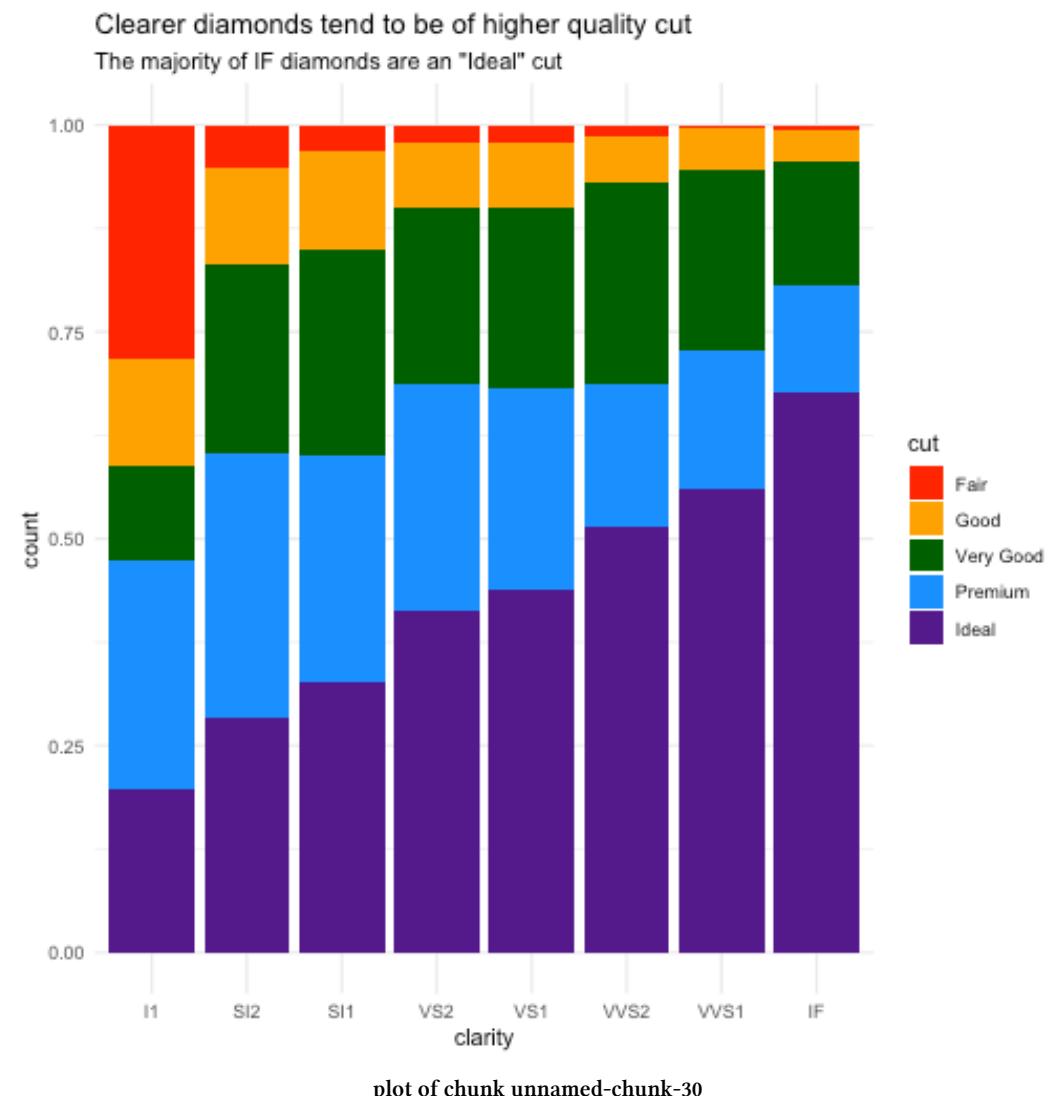
Text on plots is incredibly helpful. A good title tells viewers what they should be getting out of the plot. Axis labels are incredibly important to inform viewers of what’s being plotted. Annotations on plots help guide viewers to important points in the plot. We’ll discuss how to control all of these now!

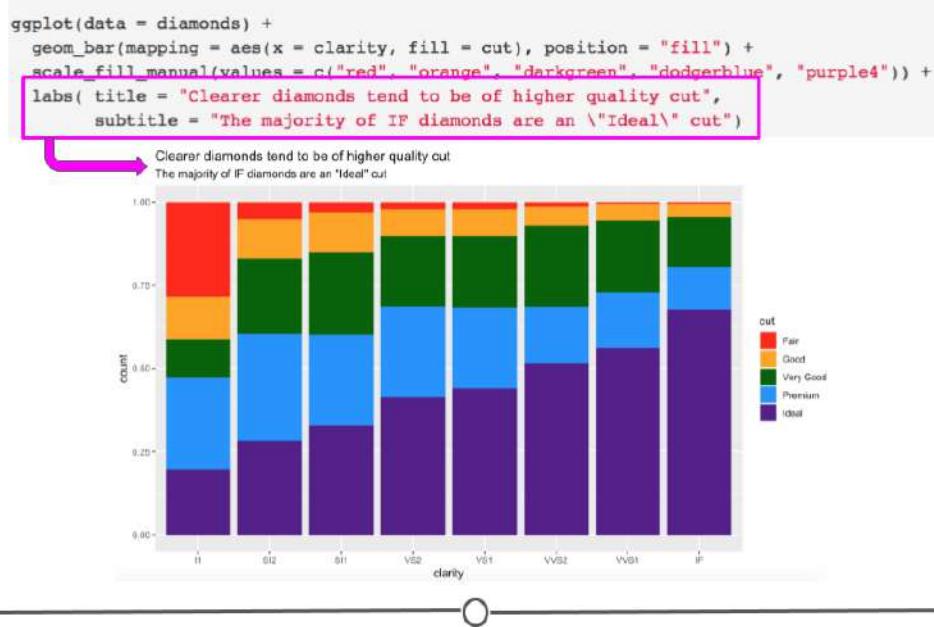
Titles

Now that we have an understanding of how to manually adjust color, let's improve the clarity of our plots by including helpful labels by adding an additional `labs()` layer. We'll return to the plot where we were comparing proportions of diamond cut across diamond clarity groups.

You can include a `title`, `subtitle`, and/or `caption` within the `labs()` function. Each argument, as per usual, will be specified by a comma.

```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +  
  # add titles  
  labs(title = "Clearer diamonds tend to be of higher quality cut",  
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut")
```



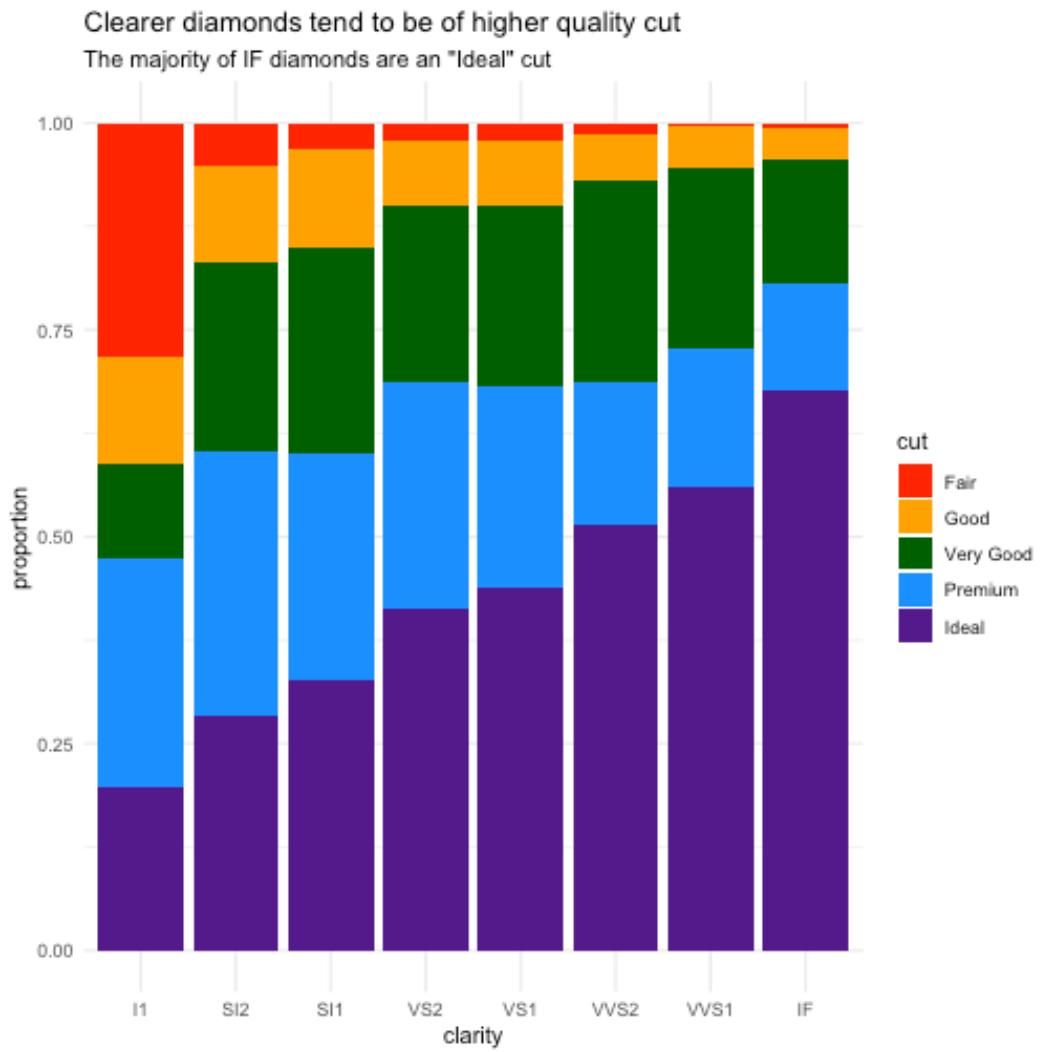


`labs()` adds helpful titles, subtitles, and captions

Axis labels

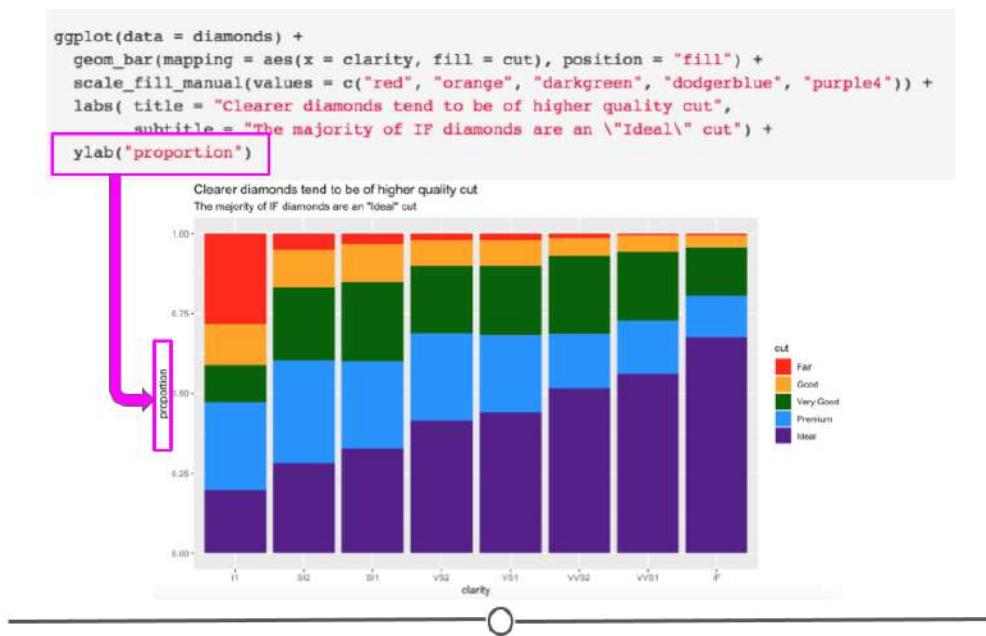
You may have noticed that our y-axis label says “count”, but it’s not actually a count anymore. In reality, it’s a proportion. Having appropriately labeled axes is *so important*. Otherwise, viewers won’t know what’s being plotted. So, we should really fix that now using the `ylab()` function. Note: we won’t be changing the x-axis label, but if you were interested in doing so, you would use `xlab("label")`.

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +
  labs(title = "Clearer diamonds tend to be of higher quality cut",
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +
  # add y axis label explicitly
  ylab("proportion")
```



plot of chunk unnamed-chunk-31

Note that the x- and y- axis labels can *also* be changed within `labs()`, using the argument (`x =` and `y =` respectively).

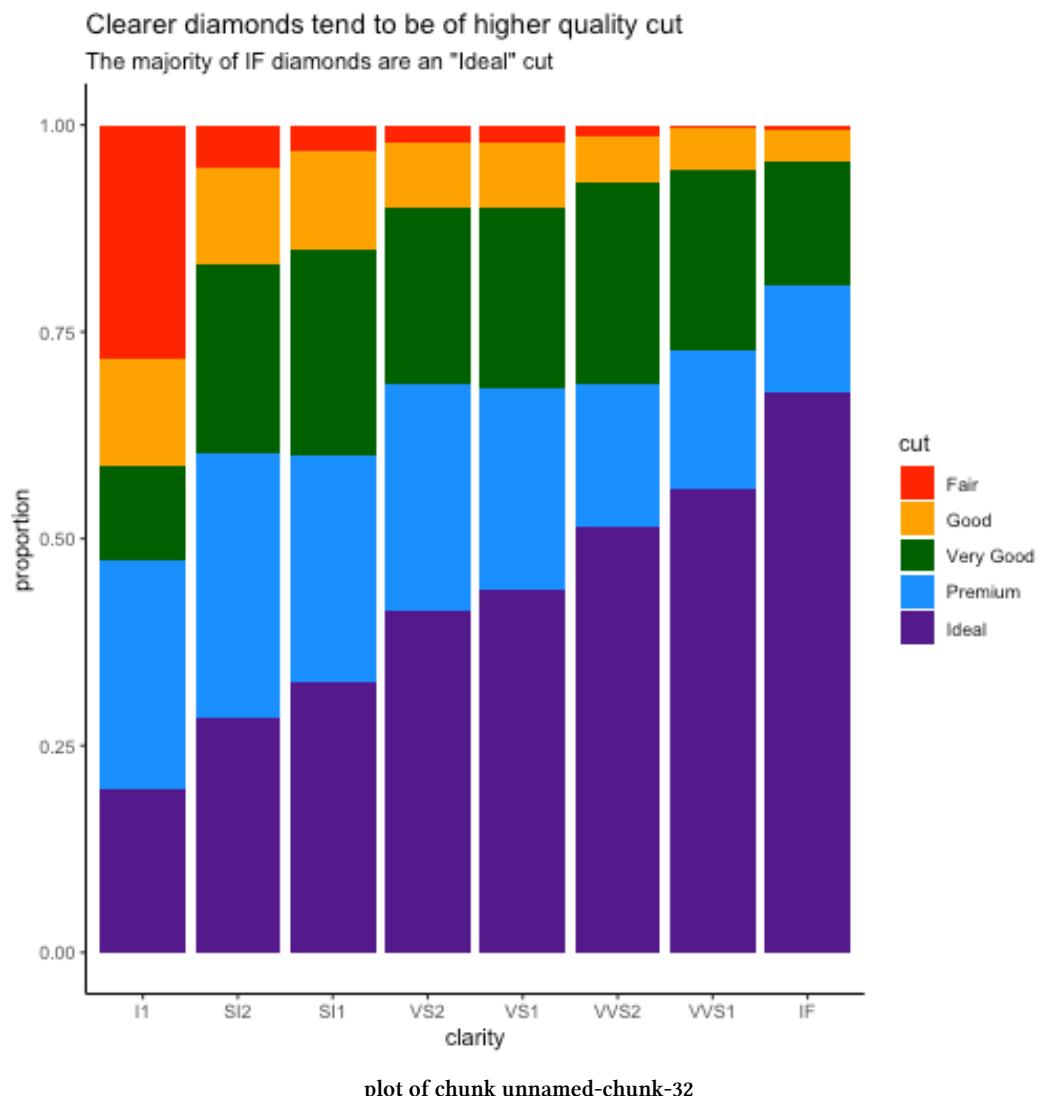


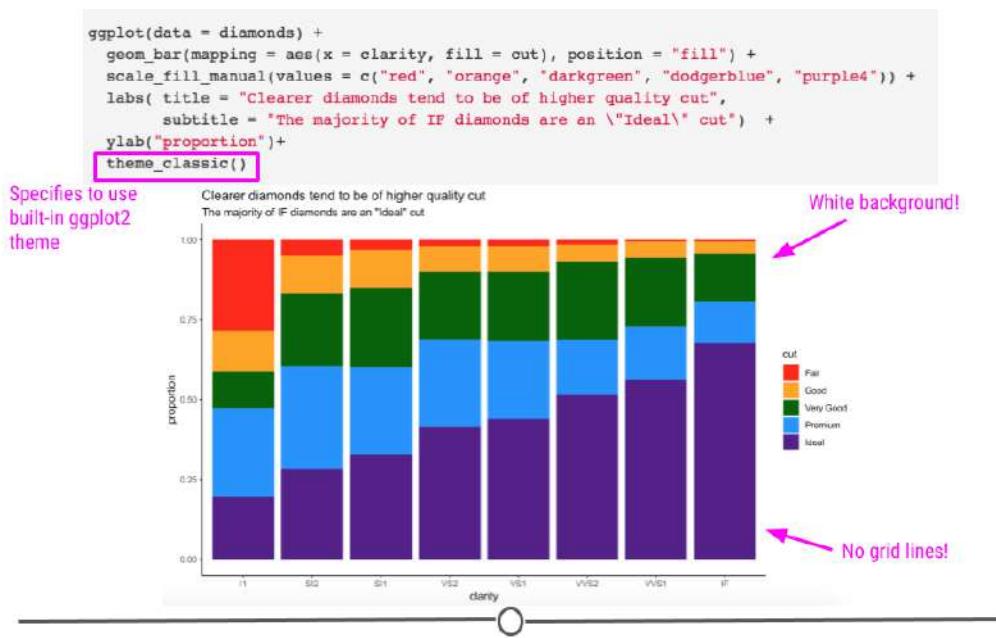
Accurate axis labels are incredibly important

Themes

To change the overall aesthetic of your graph, there are 8 themes built into ggplot2 that can be added as an additional layer in your graph. For example, if we wanted remove the gridlines and grey background from the chart, we would use `theme_classic()`. Building on what we've already generated:

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +
  labs(title = "Clearer diamonds tend to be of higher quality cut",
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +
  ylab("proportion") +
  # change plot theme
  theme_classic()
```





`theme_classic` changes aesthetic of our plot

We now have a pretty good looking plot! However, a few additional changes would make this plot *even better* for communication.

Note: Additional themes are available from the [ggthemes package](#). Users can also generate their own themes.

Custom Theme

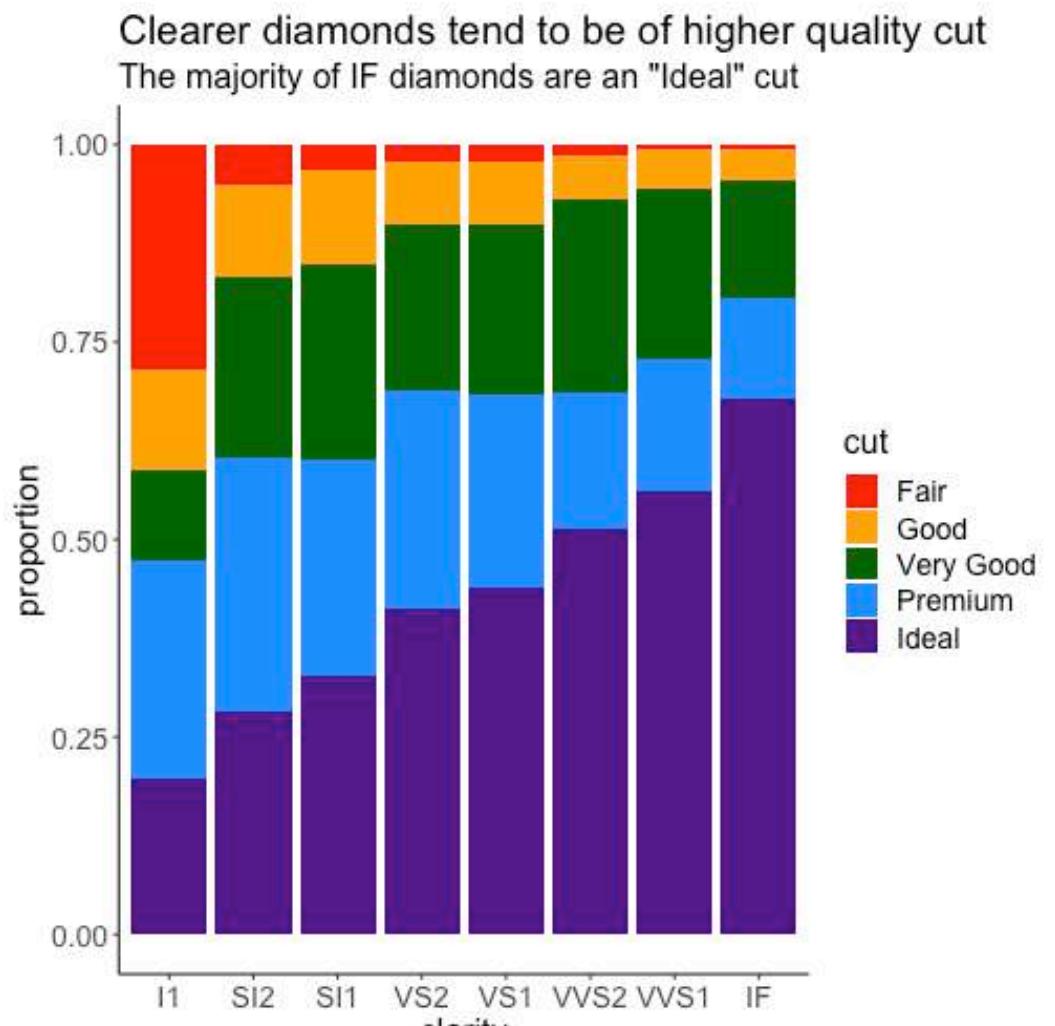
In addition to using available themes, we can also adjust parts of the theme of our graph using an additional `theme()` layer. There are a lot of options within `theme`. To see them all, look at the help documentation within RStudio Cloud using: `?theme`. We'll simply go over the syntax for using a few of them here to get you comfortable with adjusting your theme. Later on, you can play around with all the options on your own to become an expert!

Altering text size

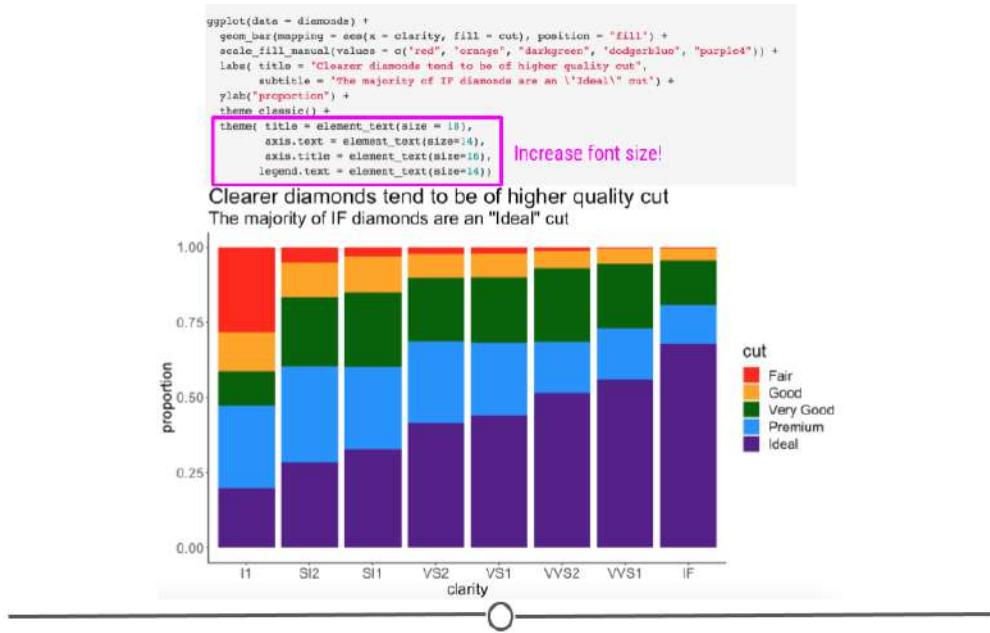
For example, if we want to increase text size to make our plots more easily to view when presenting, we could do that within `theme`. Notice here that we're increasing the text size of the `title`, `axis.text`, `axis.title`, and `legend.text` all within `theme()`! The syntax here is

important. Within each of the elements of the theme you want to alter, you have to specify what it is you want to change. Here, for all three, we want to alter the text, so we specify `element_text()`. Within that, we specify that it's `size` that we want to adjust.

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +
  labs(title = "Clearer diamonds tend to be of higher quality cut",
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +
  ylab("proportion") +
  theme_classic() +
  # control theme
  theme(title = element_text(size = 16),
        axis.text = element_text(size = 14),
        axis.title = element_text(size = 16),
        legend.text = element_text(size = 14))
```



plot of chunk unnamed-chunk-33



`theme()` allows us to adjust font size

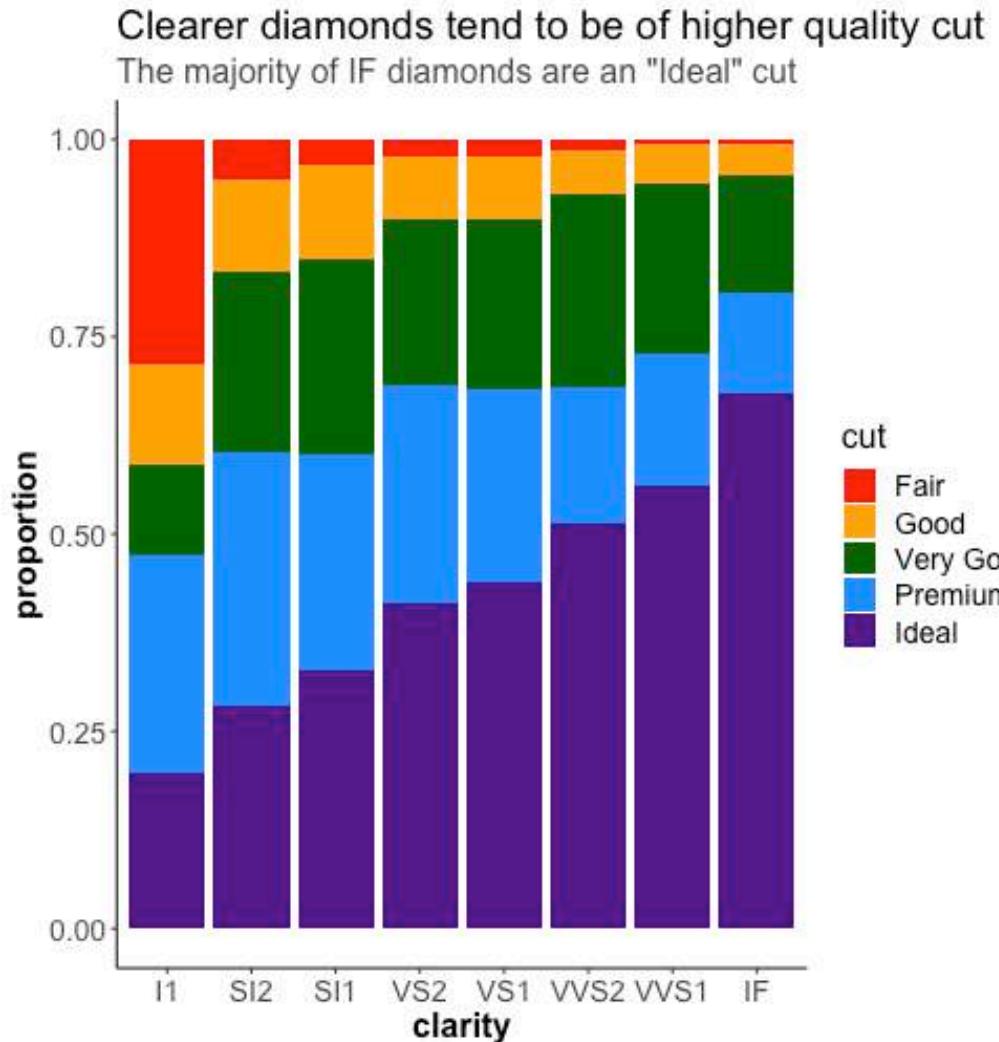
Additional text alterations

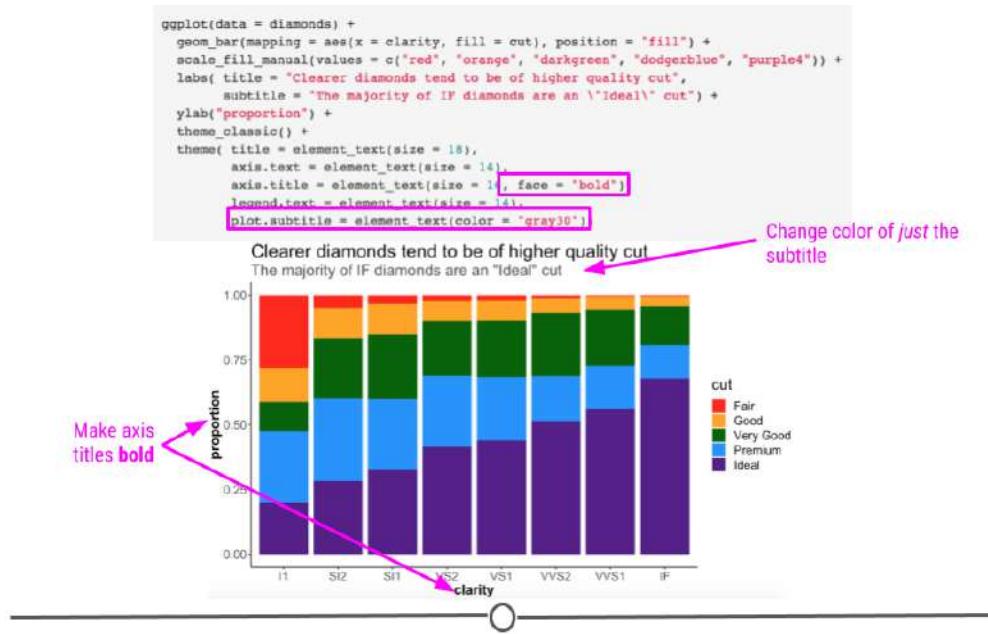
Changing the size of text on your plot is not the only thing you can control within `theme()`. You can make text **bold** and change its color within `theme()`. Note here that multiple changes can be made to a single element. We can change size and make the text **bold**. All we do is separate each argument with a comma, per usual.

```

ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple")) +
  labs(title = "Clearer diamonds tend to be of higher quality cut",
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +
  ylab("proportion") +
  theme_classic() +
  theme(title = element_text(size = 16),
        axis.text = element_text(size = 14),
        axis.title = element_text(size = 16, face = "bold")),
  
```

```
legend.text = element_text(size = 14),  
# additional control  
plot.subtitle = element_text(color = "gray30"))
```





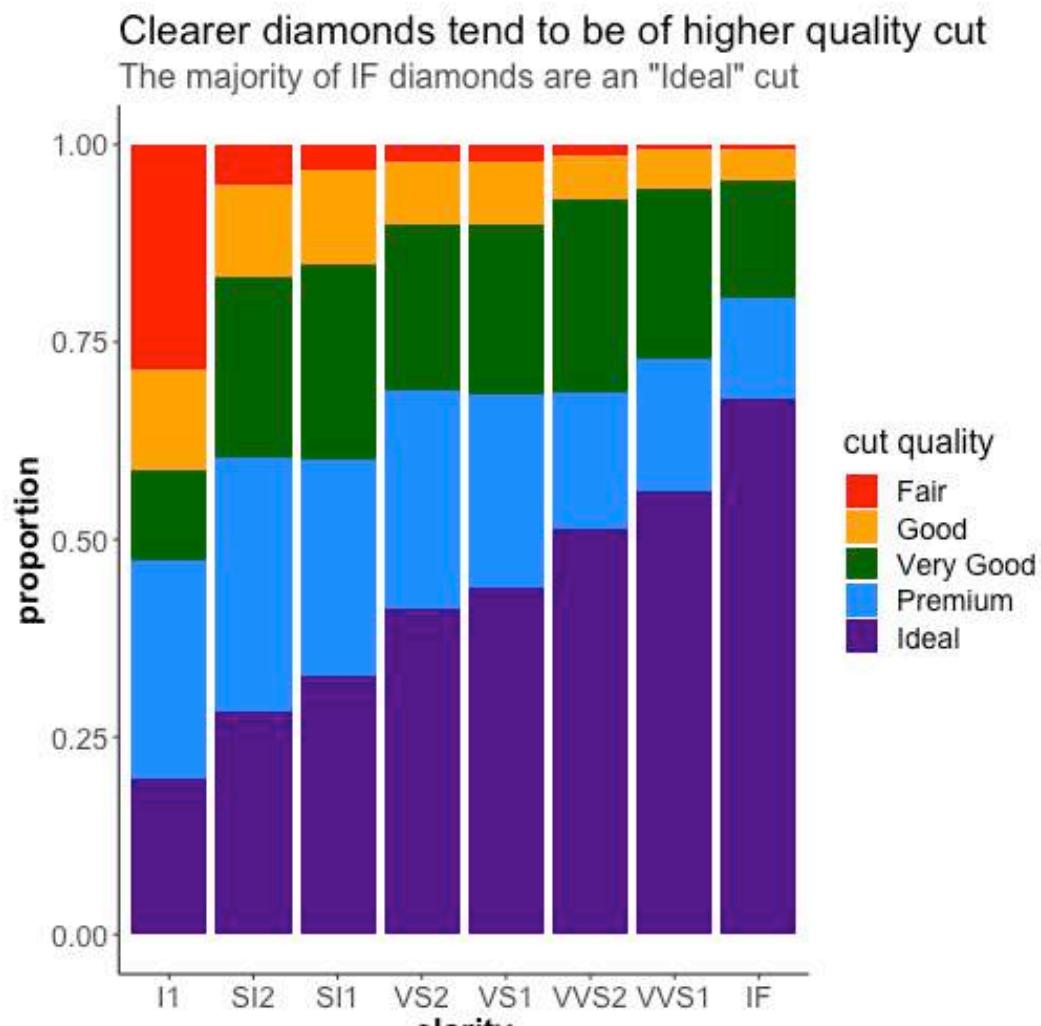
`theme()` allows us to tweak many parts of our plot

Any alterations to plot spacing/background, title, axis, and legend will all be made within `theme()`.

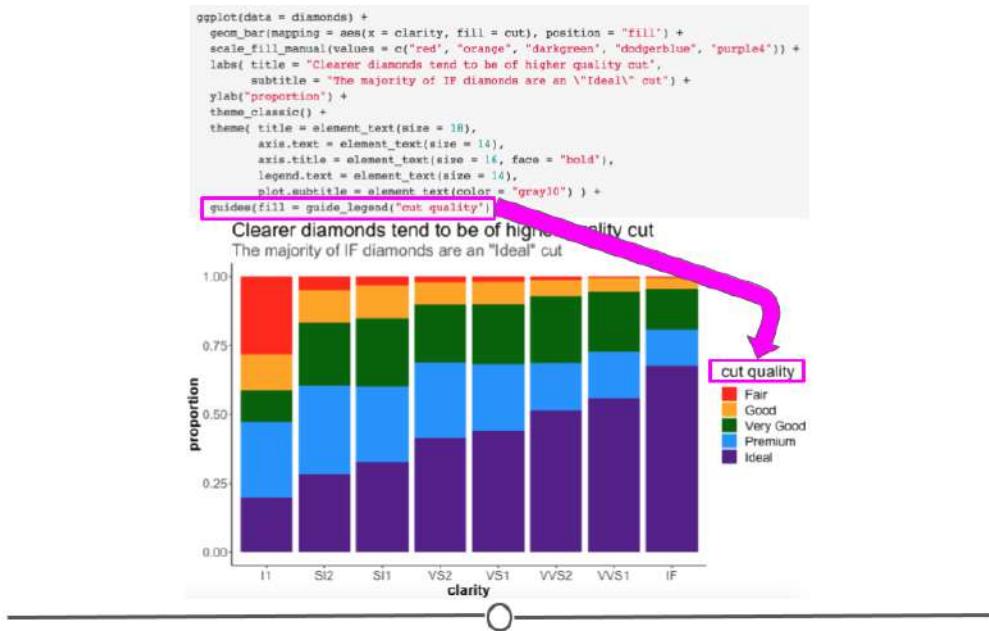
Legends

At this point, all the text on the plot is pretty visible! However, there's one thing that's still not quite clear to viewers. In daily life, people refer to the "cut" of a diamond by terms like "round cut" or "princess cut" to describe the *shape* of the diamond. That's not what we're talking about here when we're discussing "cut". In these data, "cut" refers to the quality of the diamond, not the shape. Let's be sure that's clear as well! We can change the name of the legend by using an additional layer and the `guides()` and `guide_legend()` functions of the `ggplot2` package!

```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +  
  labs(title = "Clearer diamonds tend to be of higher quality cut",  
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +  
  ylab("proportion") +  
  theme_classic() +  
  theme(title = element_text(size = 16),  
        axis.text = element_text(size = 14),  
        axis.title = element_text(size = 16, face = "bold"),  
        legend.text = element_text(size = 14),  
        plot.subtitle = element_text(color = "gray30")) +  
  # control legend  
  guides(fill = guide_legend("cut quality"))
```



plot of chunk unnamed-chunk-35



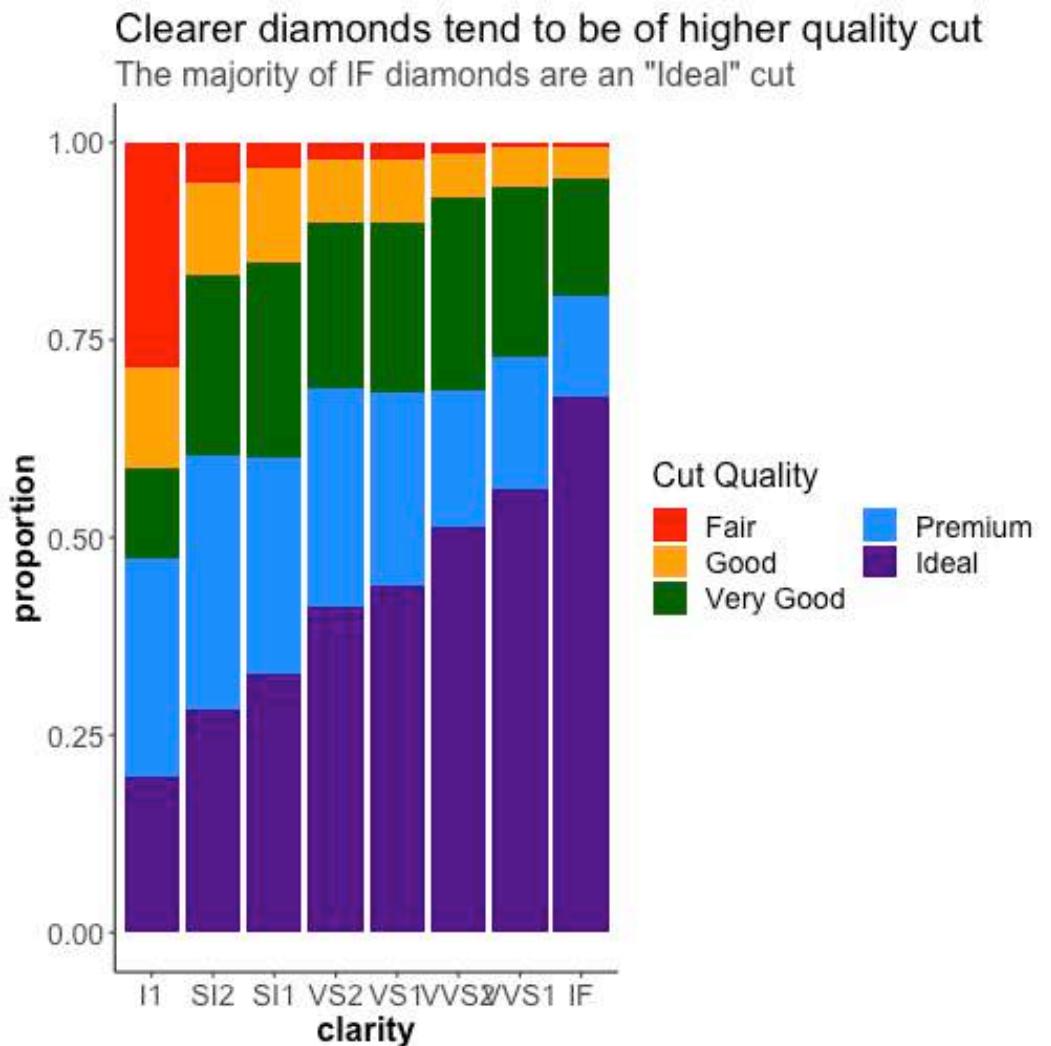
`guides()` allows us to change the legend title

This `guides()` function, as well as the `guides_*` functions allow us to modify legends even further.

This is especially useful if you have many colors in your legend and you want to control how the legend is displayed in terms of the number of columns and rows using `ncol` and `nrow` respectively.

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +
  labs(title = "Clearer diamonds tend to be of higher quality cut",
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +
  ylab("proportion") +
  theme_classic() +
  theme(title = element_text(size = 16),
        axis.text = element_text(size = 14),
        axis.title = element_text(size = 16, face = "bold"),
        legend.text = element_text(size = 14),
        plot.subtitle = element_text(color = "gray30")) +
  guides(fill = guide_legend("cut quality"))
```

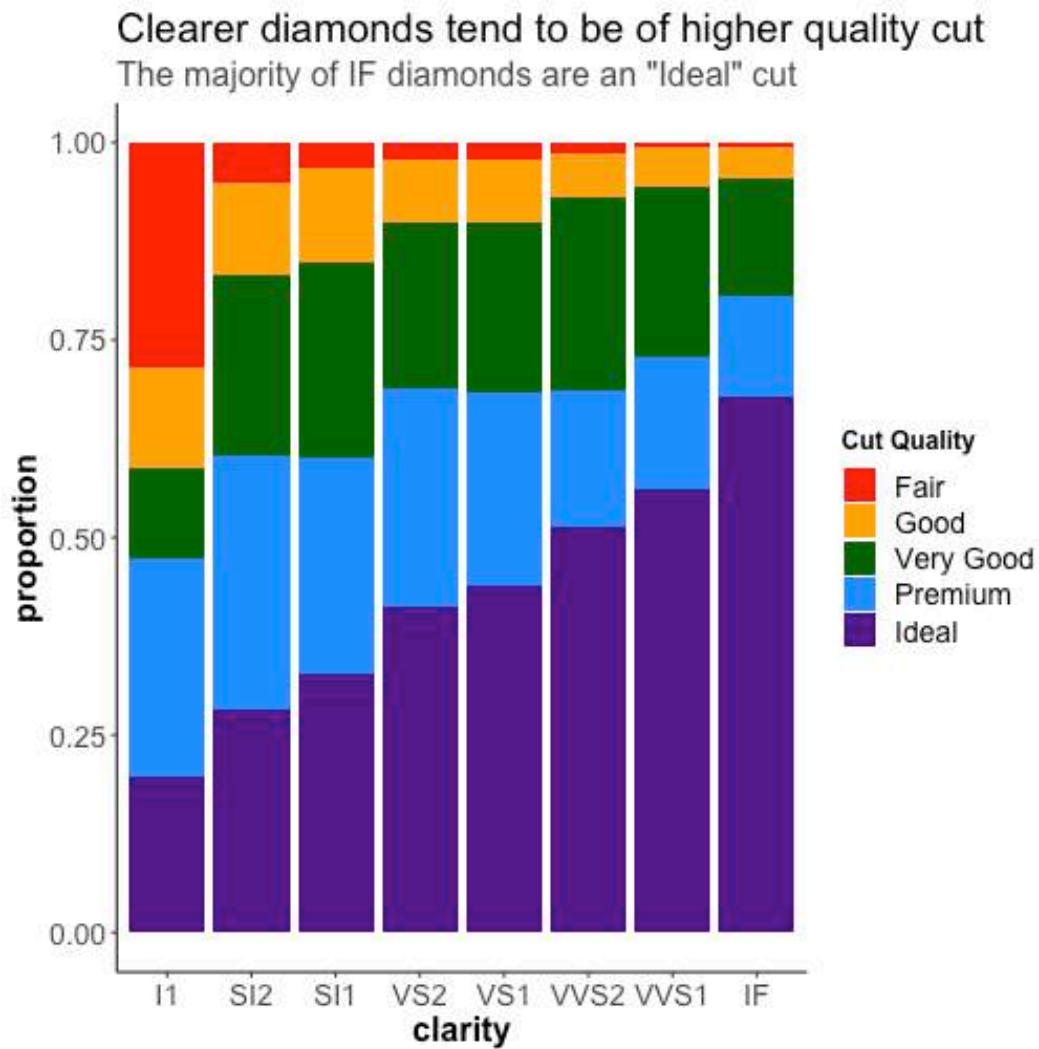
```
# control legend  
guides(fill = guide_legend("Cut Quality",  
                           ncol = 2))
```



plot of chunk unnamed-chunk-36

Or, we can modify the font of the legend title using `title.theme()`.

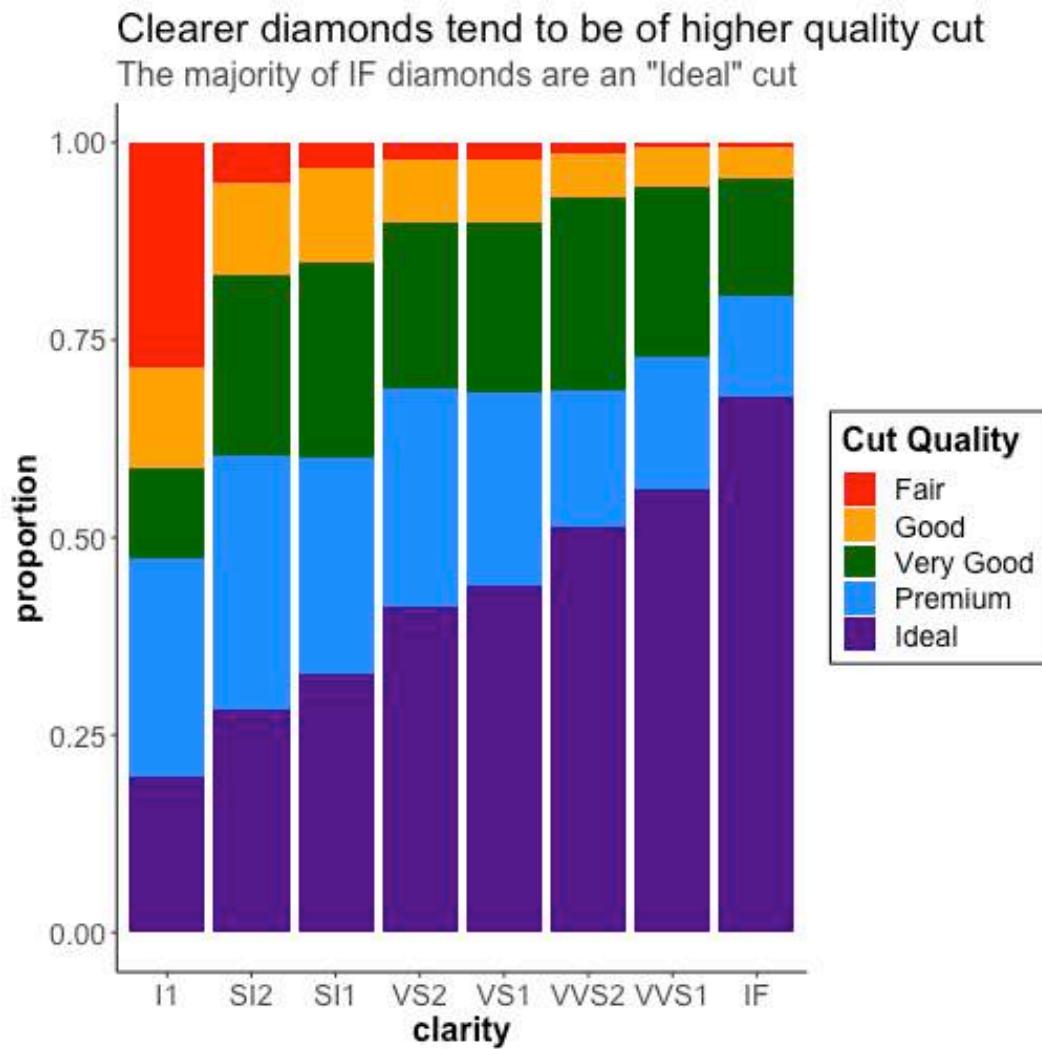
```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +  
  labs(title = "Clearer diamonds tend to be of higher quality cut",  
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +  
  ylab("proportion") +  
  theme_classic() +  
  theme(title = element_text(size = 16),  
        axis.text = element_text(size = 14),  
        axis.title = element_text(size = 16, face = "bold"),  
        legend.text = element_text(size = 14),  
        plot.subtitle = element_text(color = "gray30")) +  
  # control legend  
  guides(fill = guide_legend("Cut Quality",  
                            title.theme = element_text(face = "bold"))))
```



plot of chunk unnamed-chunk-37

Alternatively, we can do this modification, as well as other legend modifications, like adding a rectangle around the legend, using the `theme()` function.

```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +  
  labs(title = "Clearer diamonds tend to be of higher quality cut",  
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +  
  ylab("proportion") +  
# changing the legend title:  
  guides(fill = guide_legend("Cut Quality")) +  
  theme_classic() +  
  theme(title = element_text(size = 16),  
        axis.text = element_text(size = 14),  
        axis.title = element_text(size = 16, face = "bold"),  
        legend.text = element_text(size = 14),  
        plot.subtitle = element_text(color = "gray30"),  
# changing the legend style:  
  legend.title = element_text(face = "bold"),  
  legend.background = element_rect(color = "black"))
```



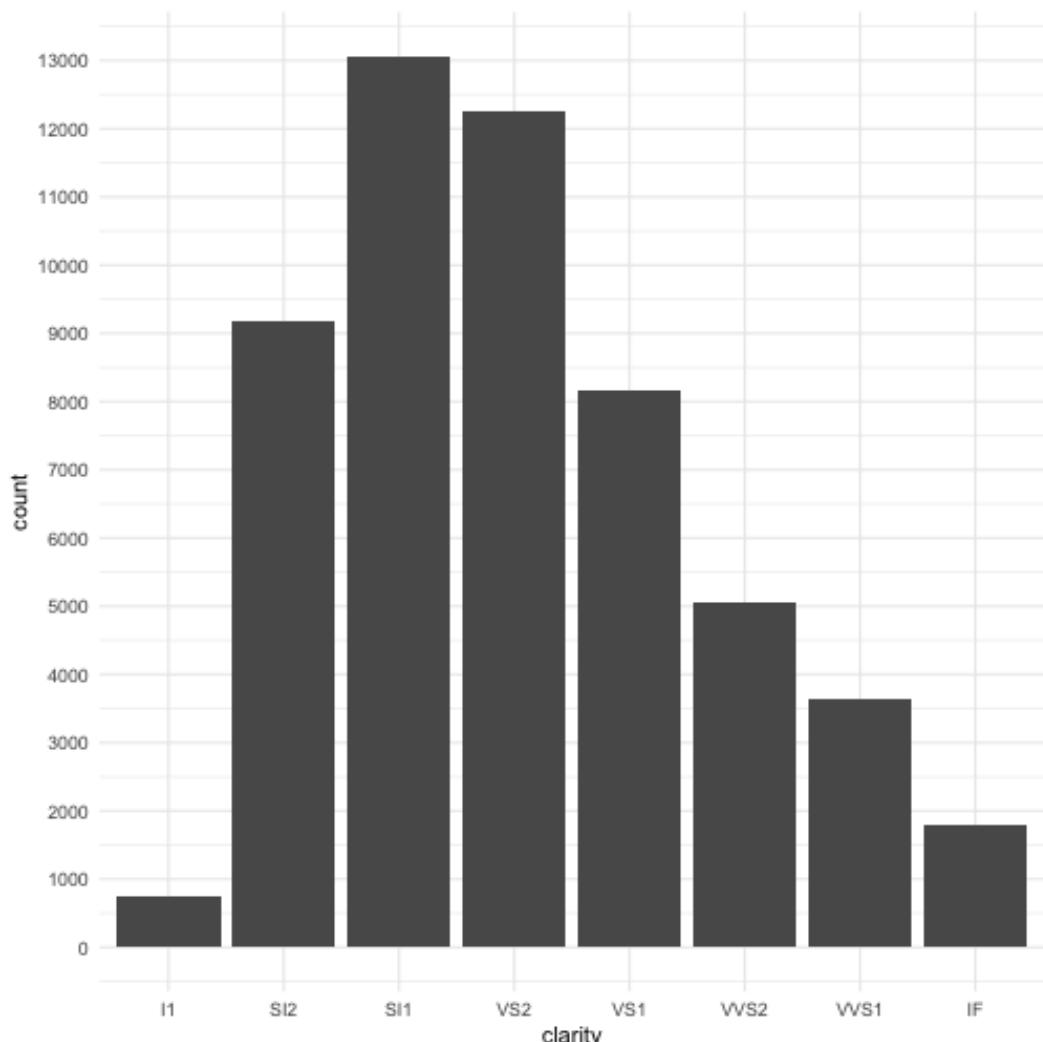
At this point, we have an informative title, clear colors, a well-labeled legend, and text that is large enough throughout the graph. This is certainly a graph that could be used in a presentation. We've taken it from a graph that is useful to just ourselves (exploratory) and made it into a plot that can communicate our findings well to others (explanatory)!

We have touched on a number of alterations you can make by adding additional layers to a ggplot. In the rest of this lesson we'll touch on a few more changes you can make within ggplot2.

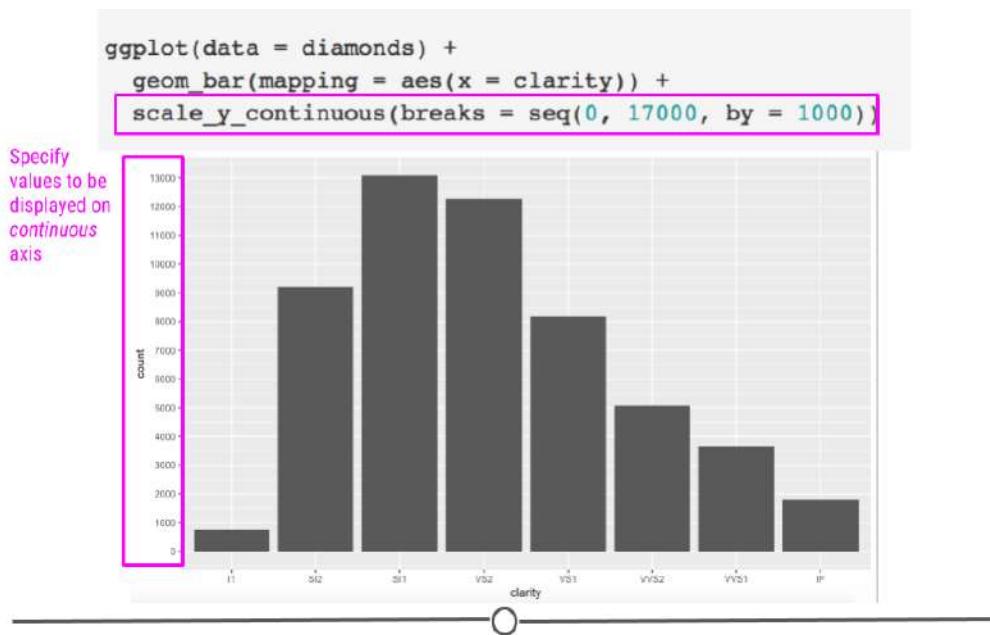
Scales

There may be times when you want a different number of values to be displayed on an axis. The scale of your plot for **continuous variables** (i.e. numeric variables) can be controlled using `scale_x_continuous` or `scale_y_continuous`. Here, we want to increase the number of labels displayed on the y-axis, so we'll use `scale_y_continuous`:

```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity)) +  
  # control scale for continuous variable  
  scale_y_continuous(breaks = seq(0, 17000, by = 1000))
```



plot of chunk unnamed-chunk-39



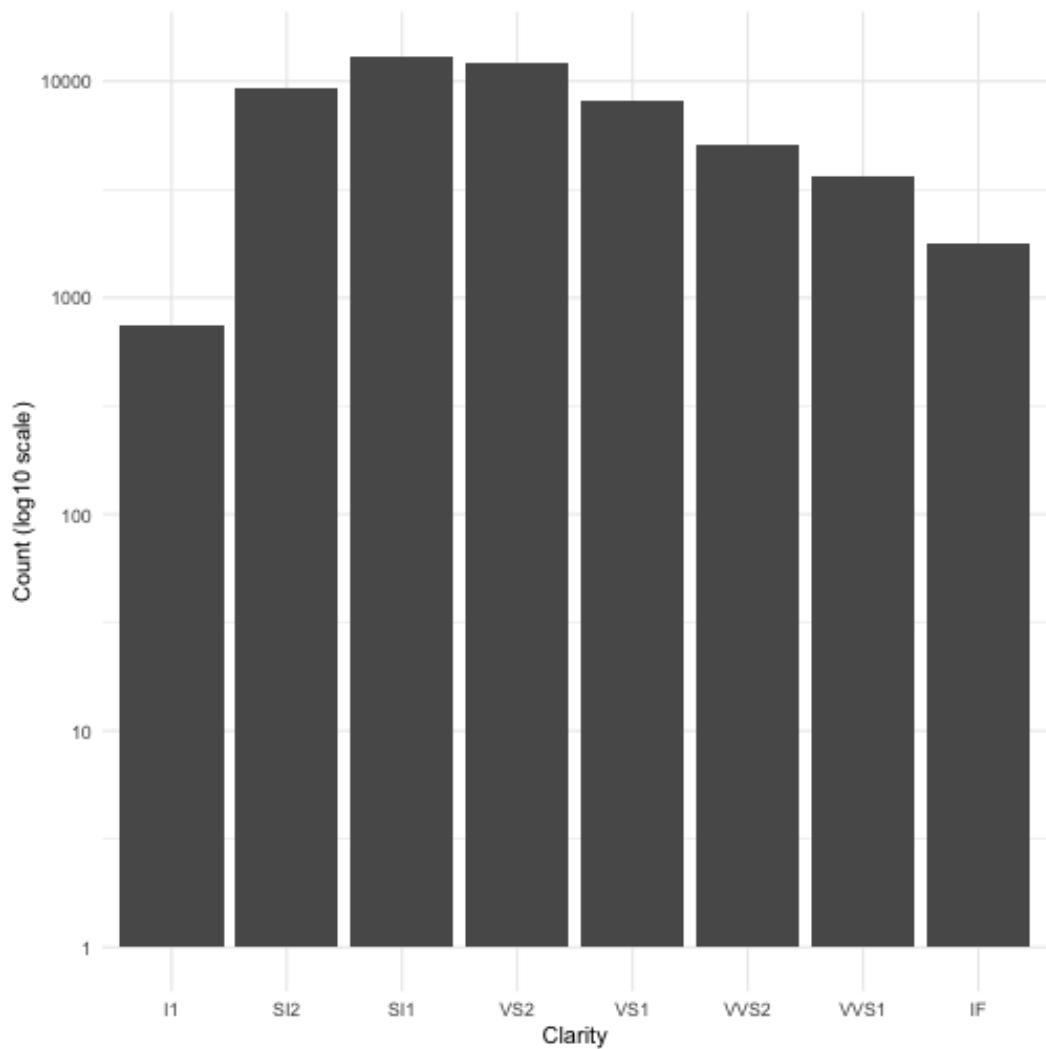
Continuous scales can be altered

There is very handy argument called `trans` for the `scale_y_continuous` or the `scale_x_continuous` functions to change the scale of the axes. For example, it can be very useful to show the logarithmic version of the scale if you have very high values with large differences.

According to the documentation for the `trans` argument:

Built-in transformations include “asn”, “atanh”, “boxcox”, “date”, “exp”, “hms”, “identity”, “log”, “log10”, “log1p”, “log2”, “logit”, “modulus”, “probability”, “probit”, “pseudo_log”, “reciprocal”, “reverse”, “sqrt” and “time”.

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity)) +
  # control scale for continuous variable
  scale_y_continuous(trans = "log10") +
  labs(y = "Count (log10 scale)",
       x = "Clarity")
```

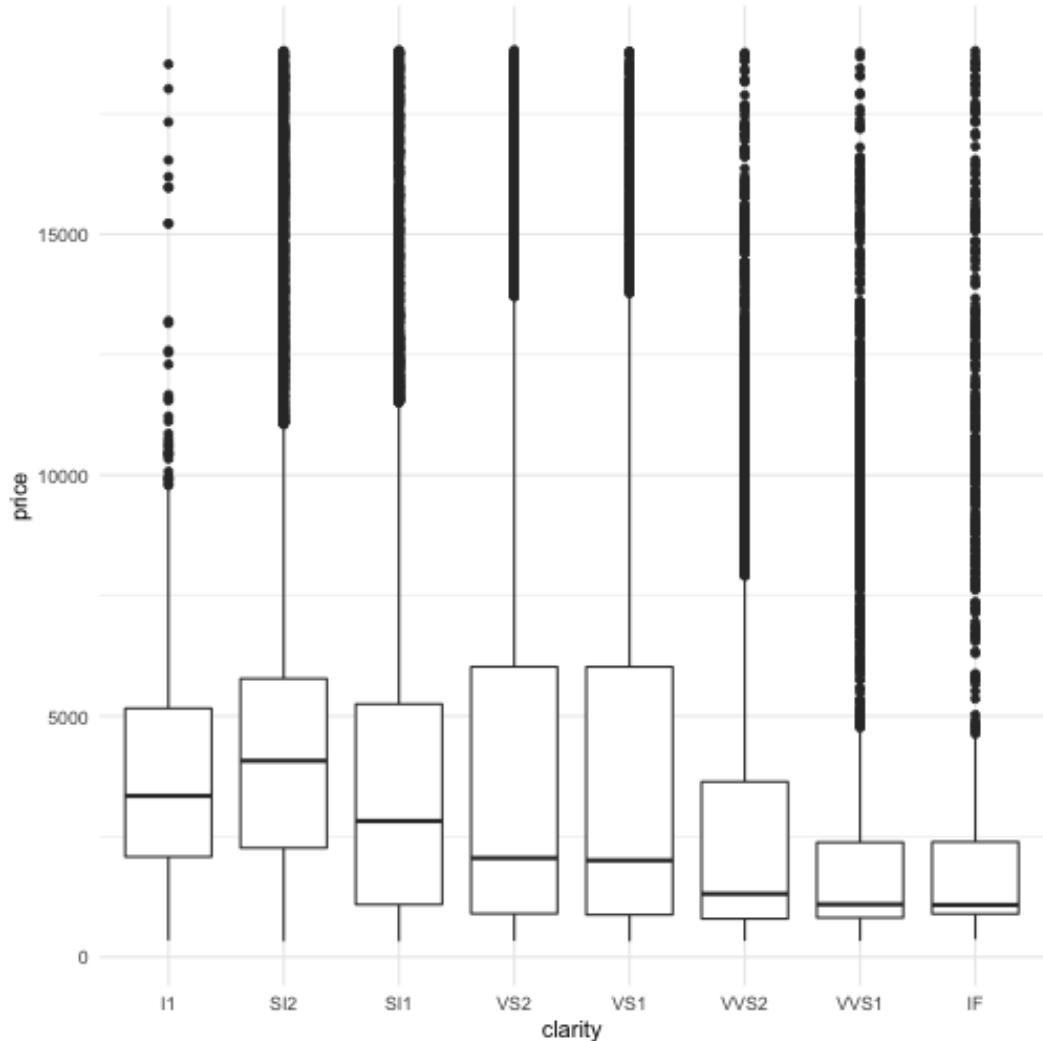


plot of chunk unnamed-chunk-40

Notice that the values are not changed, just the way they are plotted. Now the y-axis increases by a factor of 10 for each break.

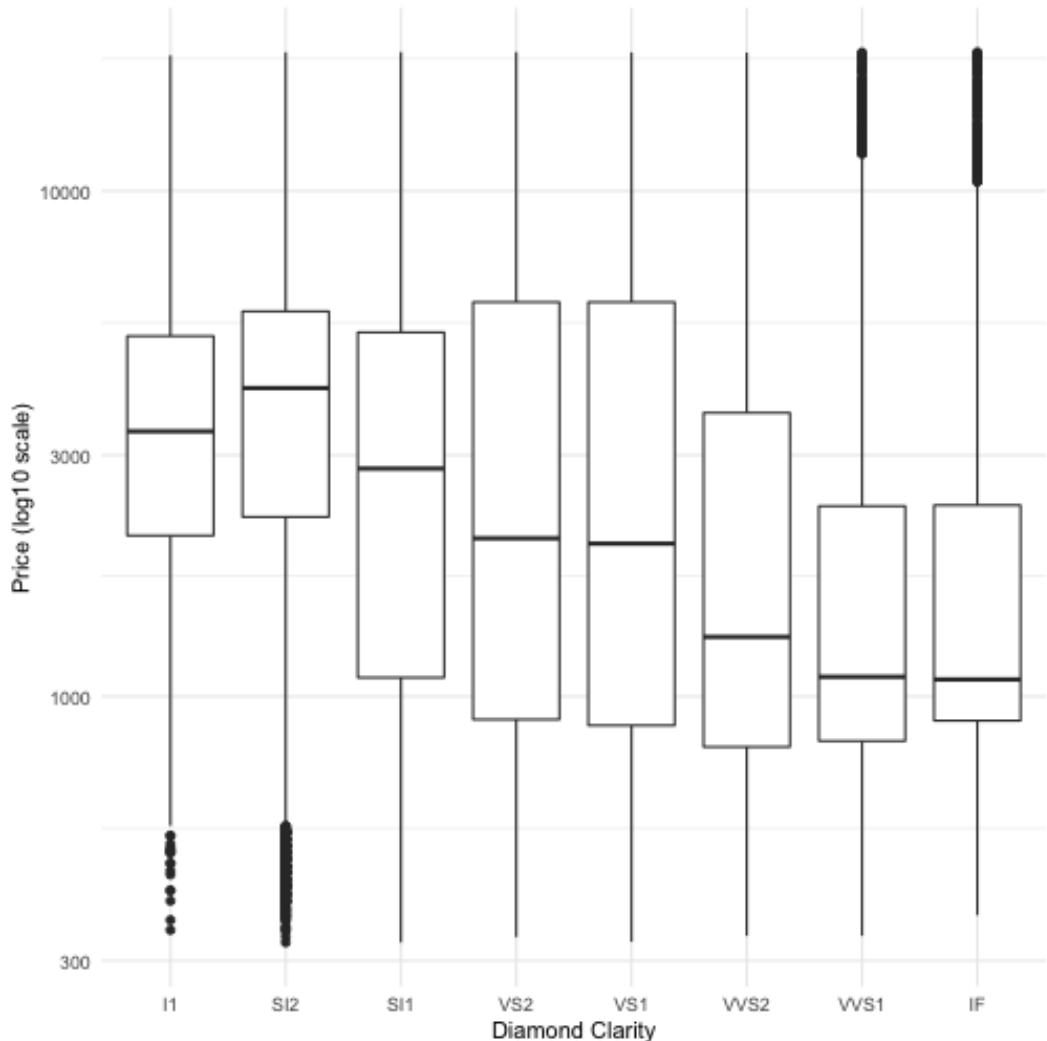
We will create a plot of the price of the diamonds to demonstrate the utility of creating a plot with a log10 scaled y-axis.

```
ggplot(diamonds) +  
  geom_boxplot(aes(y = price, x = clarity))
```



plot of chunk unnamed-chunk-41

```
ggplot(diamonds) +  
  geom_boxplot(aes(y = price, x = clarity)) +  
  scale_y_continuous(trans = "log10") +  
  labs(y = "Price (log10 scale)",  
       x = "Diamond Clarity")
```



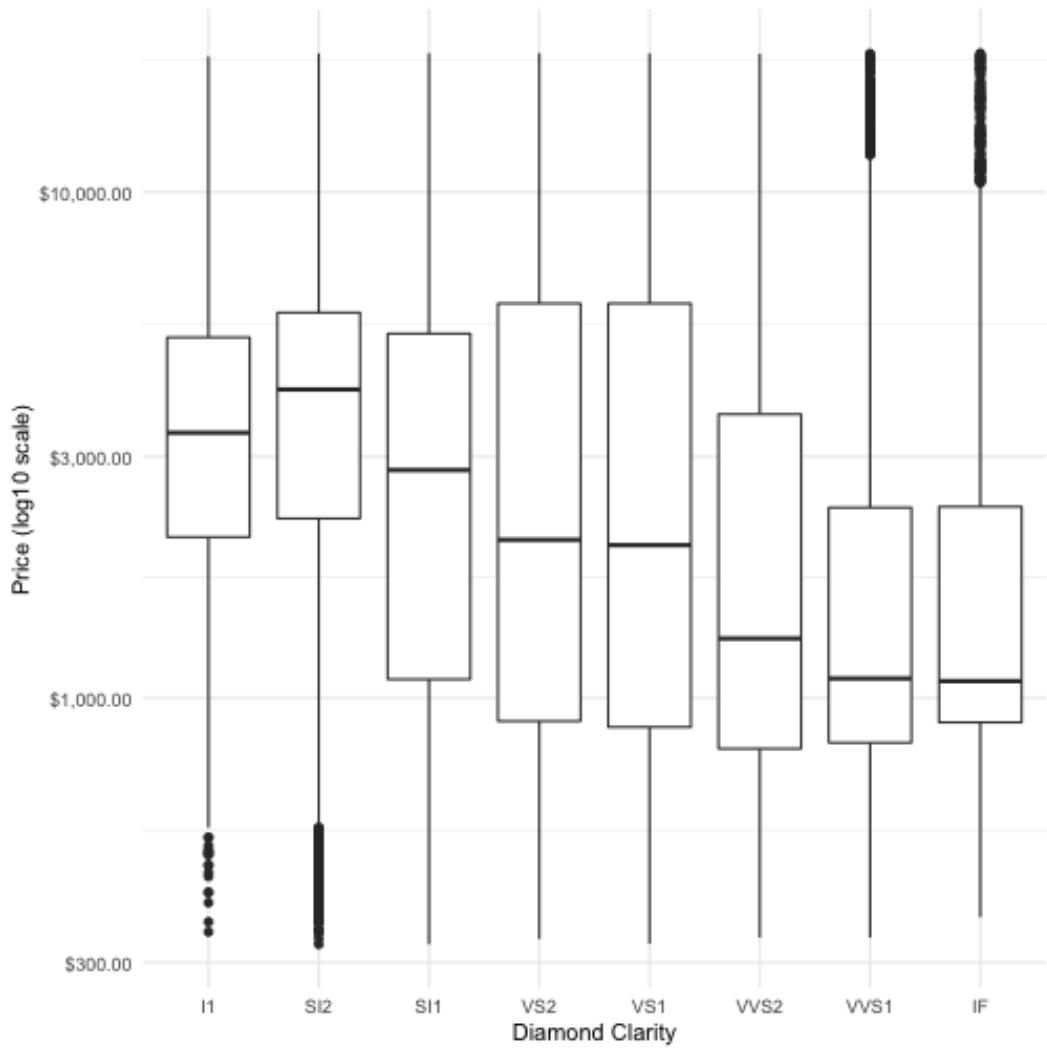
plot of chunk unnamed-chunk-41

In the first plot, it is difficult to tell what values the boxplots correspond to and it is difficult to compare the boxplots (particularly for the last three clarity categories), however this is

greatly improved in the second plot.

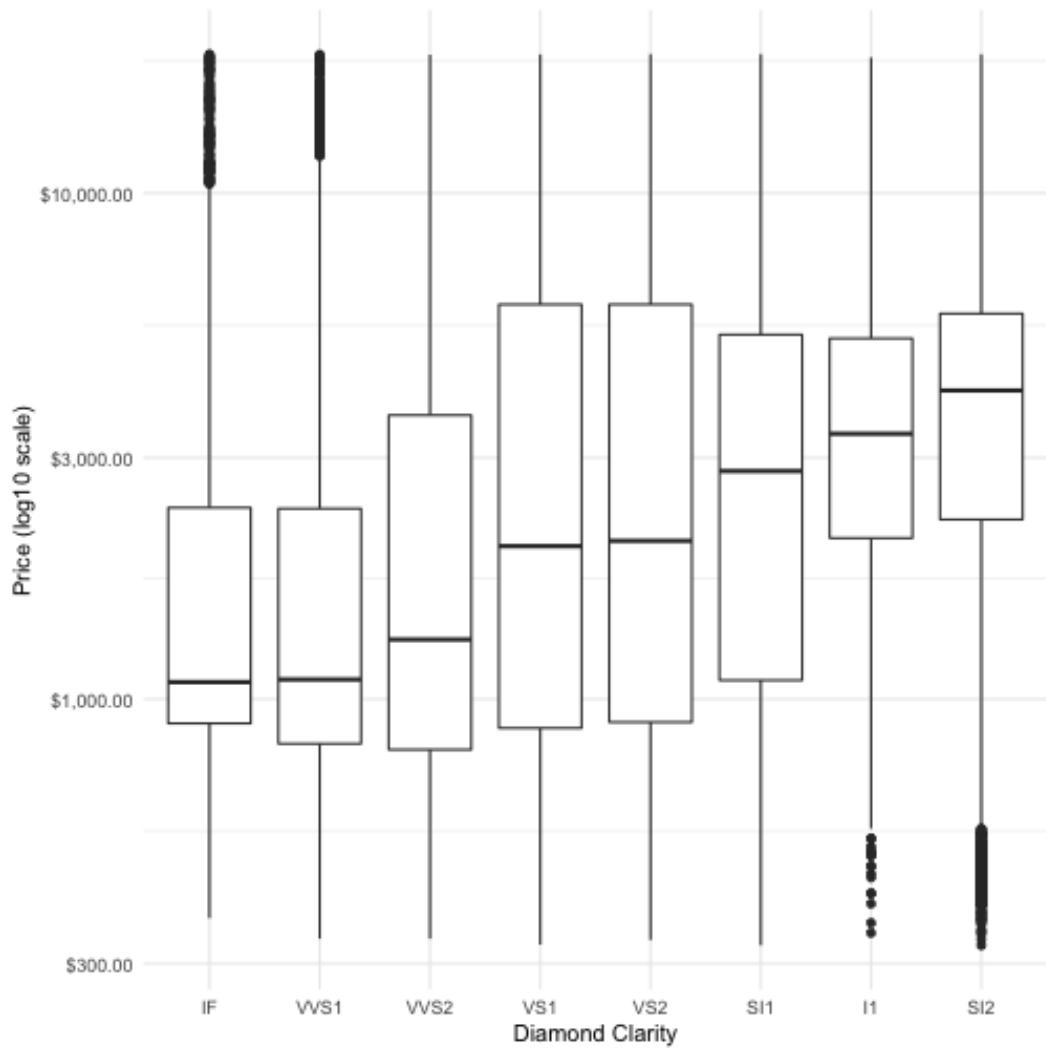
We can also use another argument of the `scale_y_continuous()` function to add specific labels to our plot. For example, it would be nice to add dollar signs to the y-axis. We can do so using the `labels` argument. A variety of `label_*` functions within the `scales` package can be used to modify axis labels. See [here](#) to take a look at the many options.

```
ggplot(diamonds) +  
  geom_boxplot(aes(y = price, x = clarity)) +  
  scale_y_continuous(trans = "log10",  
                     labels = scales::label_dollar()) +  
  labs(y = "Price (log10 scale)",  
       x = "Diamond Clarity")
```



In the above plot, we might also want to order the boxplots by the median price, we can do so using the `fct_reorder` function of `forcats` package to change the order for the clarity levels to be based on the median of the price values.

```
ggplot(diamonds) +  
  geom_boxplot(aes(y = price, x = forcats::fct_reorder(clarity, price, .fun = m\  
edian))) +  
  scale_y_continuous(trans = "log10",  
                     labels = scales::label_dollar()) +  
  labs(y = "Price (log10 scale)",  
       x = "Diamond Clarity")
```



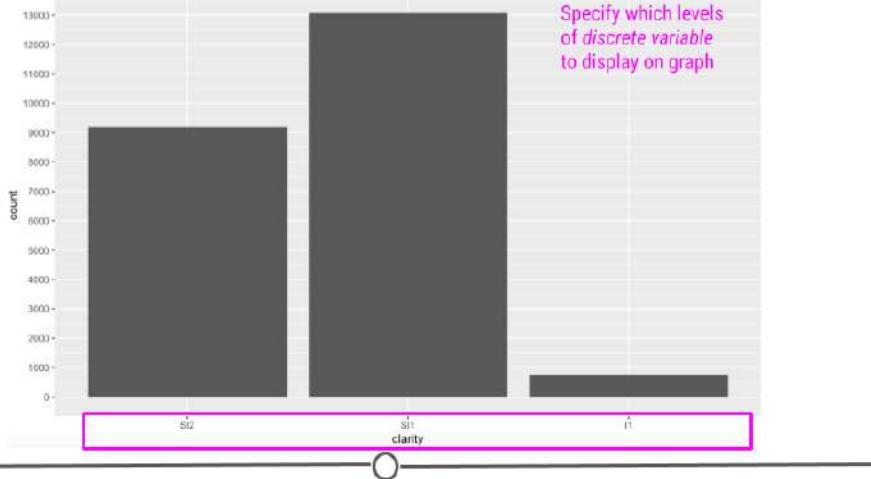
plot of chunk unnamed-chunk-43

Now we can more easily determine that the SI2 diamonds are the most expensive.

Another way to modify **discrete variables** (aka factors or categorical variables where there is a limited number of levels), is to use `scale_x_discrete` or `scale_y_discrete`. In this case we will just pick a few of the clarity categories to plot and we will specify the order.

```
ggplot(diamonds) +
  geom_bar(aes(x = clarity)) +
  # control scale for discrete variable
  scale_x_discrete(limit = c("SI2", "SI1", "I1")) +
  scale_y_continuous(breaks = seq(0, 17000, by = 1000))
```

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = clarity)) +
  scale_y_continuous(breaks = seq(0, 17000, by = 1000)) +
  scale_x_discrete(limit = c("SI2", "SI1", "I1"))
```



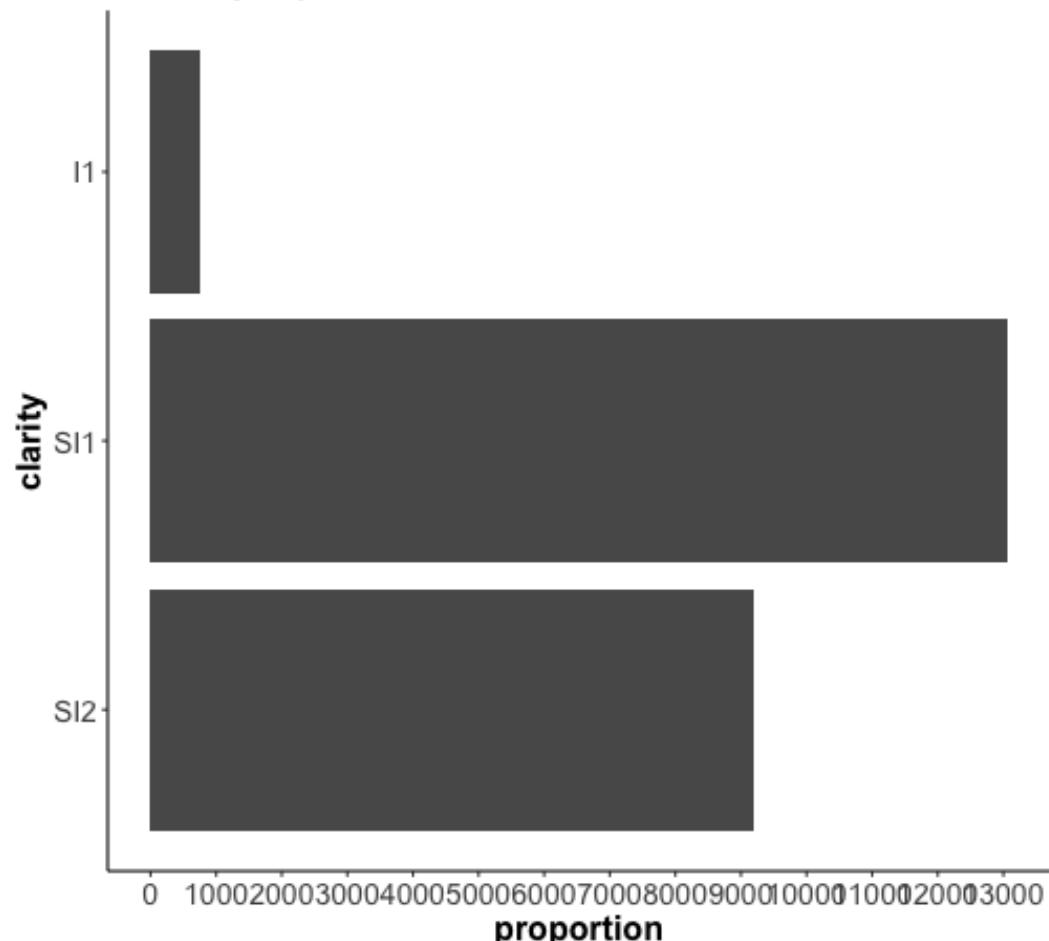
Discrete scales can be altered

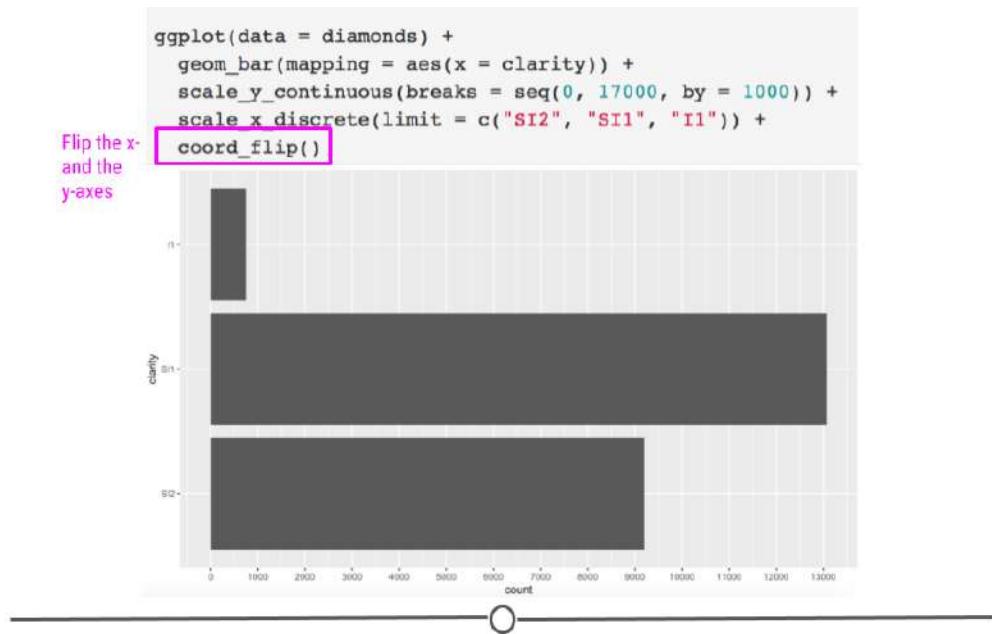
Coordinate Adjustment

There are times when you'll want to flip your axis. This can be accomplished using `coord_flip()`. Adding an additional layer to the plot we just generated switches our x- and y-axes, allowing for horizontal bar charts, rather than the default vertical bar charts:

```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity)) +  
  scale_y_continuous(breaks = seq(0, 17000, by = 1000)) +  
  scale_x_discrete(limit = c("SI2", "SI1", "I1")) +  
  # flip coordinates  
  coord_flip() +  
  labs(title = "Clearer diamonds tend to be of higher quality cut",  
       subtitle = "The majority of IF diamonds are an \"Ideal\" cut") +  
  ylab("proportion") +  
  theme_classic() +  
  theme(title = element_text(size = 18),  
        axis.text = element_text(size = 14),  
        axis.title = element_text(size = 16, face = "bold"),  
        legend.text = element_text(size = 14),  
        plot.subtitle = element_text(color = "gray30")) +  
  guides(fill = guide_legend("cut quality"))  
Warning: Removed 30940 rows containing non-finite values (stat_count).
```

Clearer diamonds tend to be of higher quality cut
The majority of IF diamonds are an "Ideal" cut



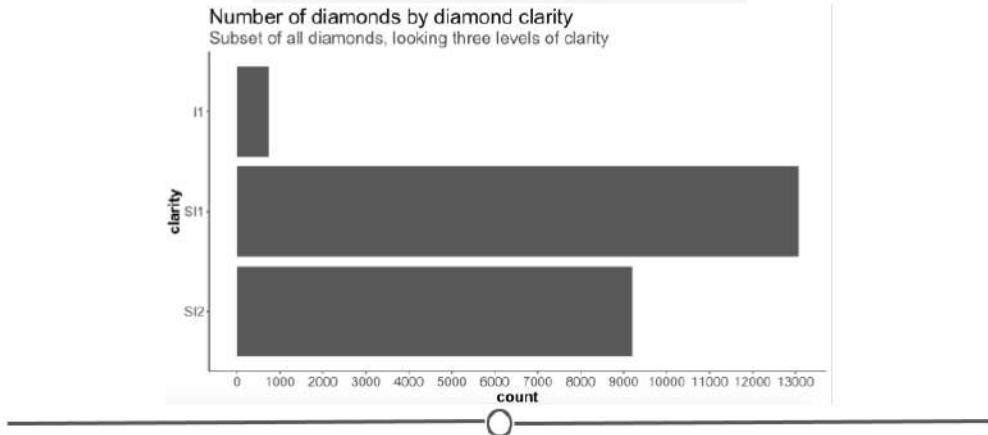


Axes can be flipped using `coord_flip`

It's important to remember that all the additional alterations we already discussed can still be applied to this graph, due to the fact that ggplot2 uses layering!

```
p <- ggplot(diamonds) +
  geom_bar(mapping = aes(x = clarity)) +
  scale_y_continuous(breaks = seq(0, 17000, by = 1000)) +
  scale_x_discrete(limit = c("SI2", "SI1", "I1")) +
  coord_flip() +
  labs(title = "Number of diamonds by diamond clarity",
       subtitle = "Subset of all diamonds, looking three levels of clarity") +
  theme_classic() +
  theme(title = element_text(size = 18),
        axis.text = element_text(size = 14),
        axis.title = element_text(size = 16, face = "bold"),
        legend.text = element_text(size = 14),
        plot.subtitle = element_text(color = "gray30"))
```

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = clarity)) +
  scale_y_continuous(breaks = seq(0, 15000, by = 1000)) +
  scale_x_discrete(limits = c("SI2", "SI1", "I1")) +
  coord_flip() +
  labs( title = "Number of diamonds by diamond clarity",
        subtitle = "Subset of all diamonds, looking three levels of clarity") +
  theme_minimal() +
  theme( title = element_text(size = 18),
        axis.title = element_text(size = 14, face = "bold"),
        legend.text = element_text(size = 14),
        plot.subtitle = element_text(color = "gray30") )
```



Additional layers will help customize this plot

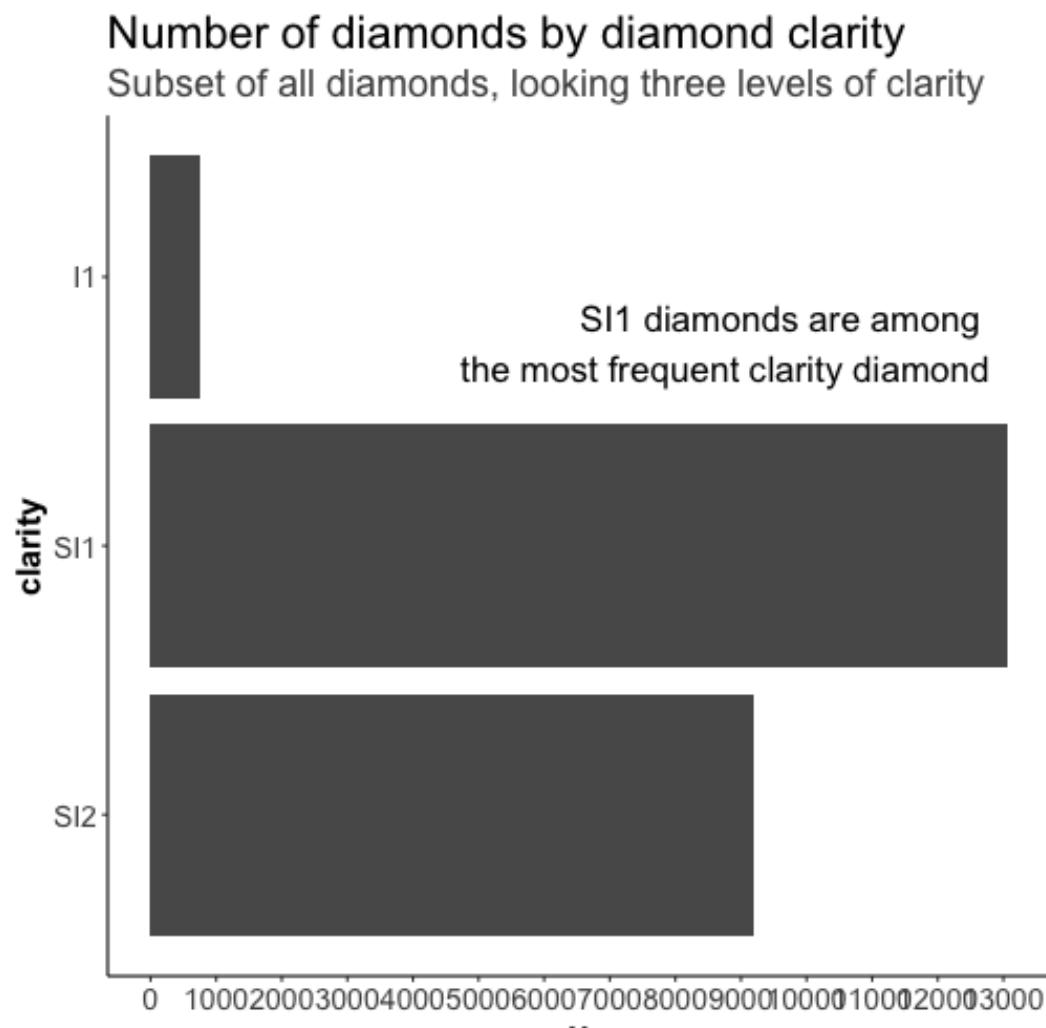
Annotation

Finally, there will be times when you'll want to add text to a plot or to annotate points on your plot. We'll discuss briefly how to accomplish that here!

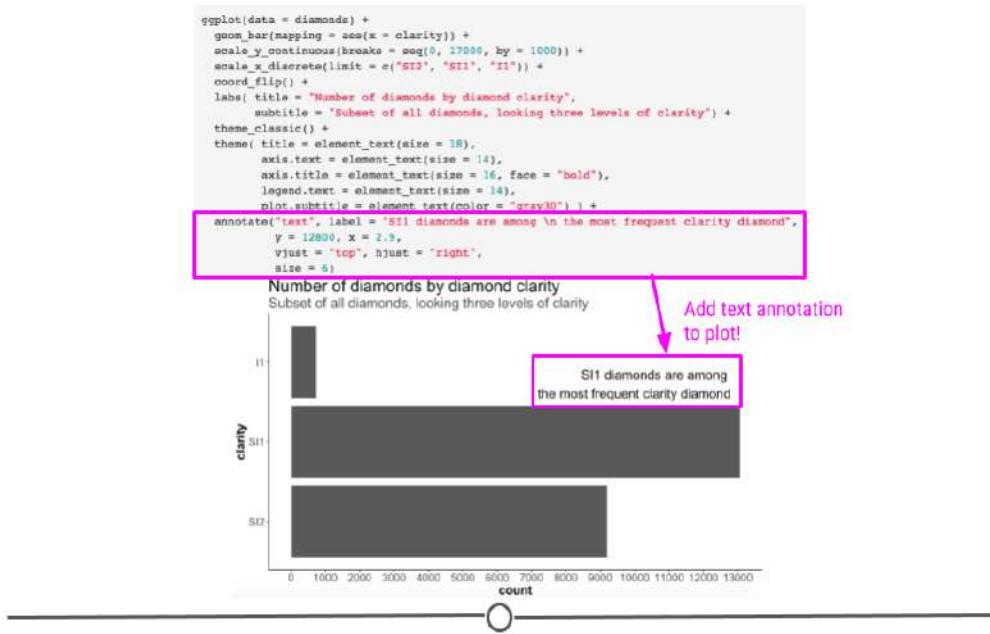
To add text to your plot, we can use the function `annotate`. This requires us to specify that we want to annotate here with “text” (rather than say a shape, like a rectangle - “rect” - which you can also do!). Additionally, we have to specify what we’d like that text to say (using the `label` argument), where on the plot we’d like that text to show up (using `x` and `y` for coordinates), how we’d like the text aligned (using `hjust` for horizontal alignment where the options are “left”, “center”, or “right” and `vjust` for vertical alignment where the arguments are “top”, “center”, or “bottom”), and how big we’d like that text to be (using `size`):

```
ggplot(diamonds) +  
  geom_bar(aes(x = clarity)) +  
  scale_y_continuous(breaks = seq(0, 17000, by = 1000)) +  
  scale_x_discrete(limit = c("SI2", "SI1", "I1")) +  
  coord_flip() +  
  labs(title = "Number of diamonds by diamond clarity",  
       subtitle = "Subset of all diamonds, looking three levels of clarity") +  
  theme_classic() +  
  theme(title = element_text(size = 18),  
        axis.text = element_text(size = 14),  
        axis.title = element_text(size = 16, face = "bold"),  
        legend.text = element_text(size = 14),  
        plot.subtitle = element_text(color = "gray30")) +  
  # add annotation  
  annotate("text", label = "SI1 diamonds are among \n the most frequent clarity\\  
diamond",  
          y = 12800, x = 2.9,  
          vjust = "top", hjust = "right",  
          size = 6)
```

Warning: Removed 30940 rows containing non-finite values (stat_count).



plot of chunk unnamed-chunk-47



`annotate` helps add text to our plot

Note: we could have accomplished this by adding an additional `geom: geom_text`. However, this requires creating a new dataframe, as explained [here](#). This can also be used to **label the points on your plot**. Keep this reference in mind in case you have to do that in the future.

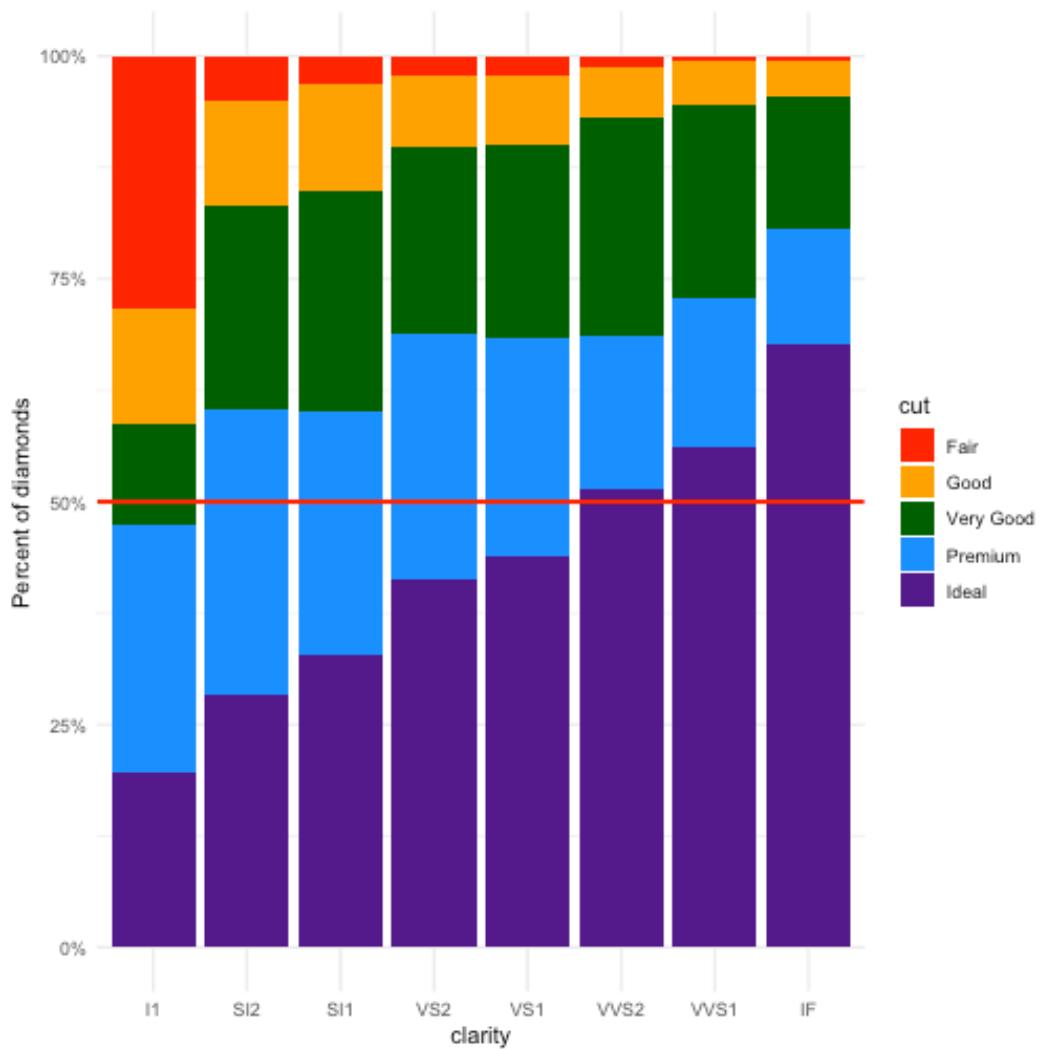
Vertical and Horizontal Lines

Sometimes it is very useful to add a line to our plot to indicate an important threshold. We can do so by using the `geom_hline()` function for a horizontal line and `geom_vline()` for a vertical line.

In each case, the functions require that a y-axis intercept or x-axis intercept be specified respectively.

For example, it might be useful to add a horizontal line to indicate 50% of the total counts for each of the clarity categories. We will also use the `scale_y_continuous()` function to change the y-axis to show percentages.

```
ggplot(diamonds) +  
  # fill scales to 100%  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +  
  scale_y_continuous(labels = scales::percent) +  
  labs(y = "Percent of diamonds") +  
  geom_hline(yintercept = 0.5, color = "red", size = 1)
```

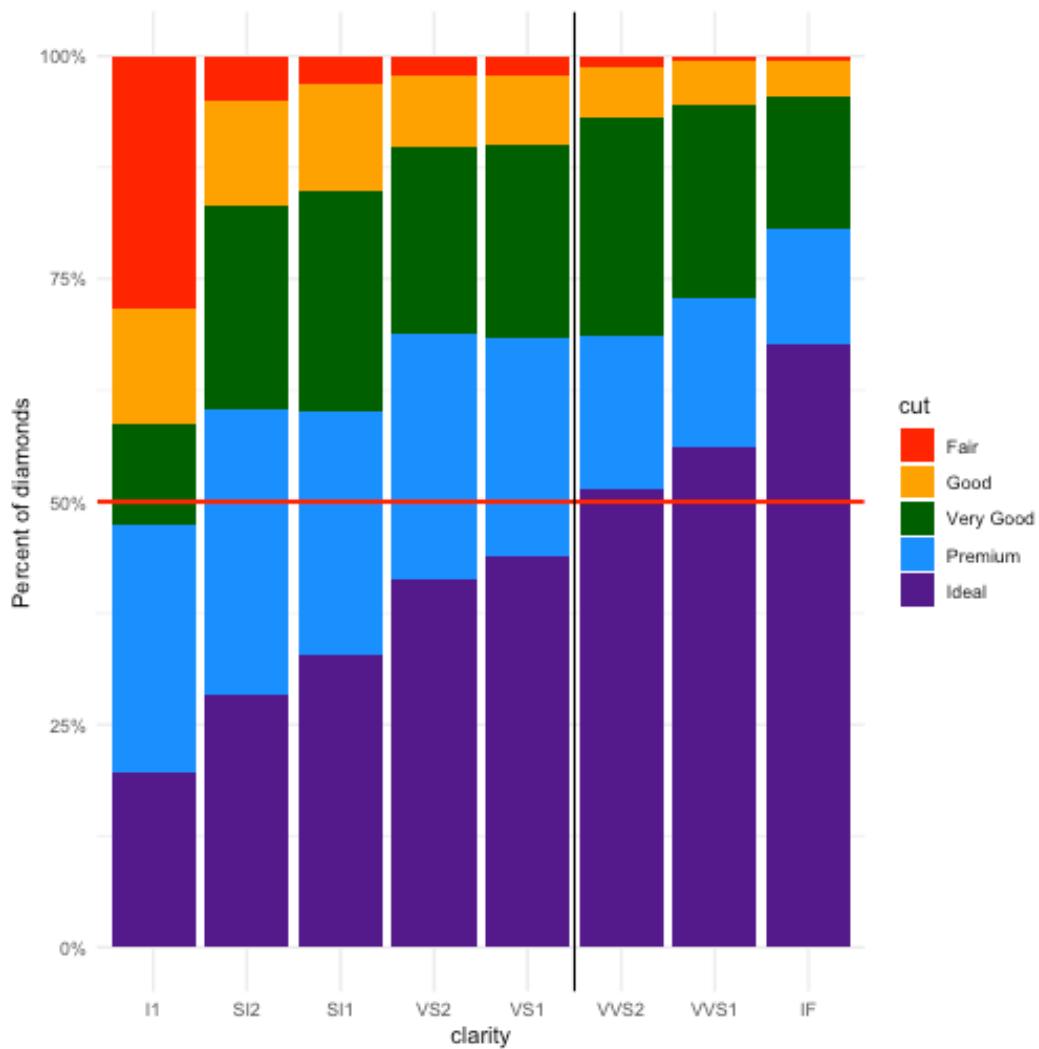


plot of chunk unnamed-chunk-48

Now, it is easier to tell that slightly over half of the VVS2 diamonds have an `Ideal` cut. This would be much more difficult to see without the horizontal line.

To add a vertical line we would instead use the `geom_vline()` function and we would specify an x-axis intercept. Since this plot has a discrete x-axis, numeric values specify a categorical value based on the order, thus a value of 4 would create a line down the center of the VS2 bar. However, if we used 5.5 we could add a line offset from the center of a bar, as you can see in the following example:

```
ggplot(diamonds) +  
  # fill scales to 100%  
  geom_bar(aes(x = clarity, fill = cut), position = "fill") +  
  scale_fill_manual(values = c("red", "orange", "darkgreen", "dodgerblue", "purple4")) +  
  scale_y_continuous(labels = scales::percent) +  
  labs(y = "Percent of diamonds") +  
  geom_hline(yintercept = 0.5, color = "red", size = 1 ) +  
  geom_vline(xintercept = 5.5, color = "black", size = .5)
```



plot of chunk unnamed-chunk-49

This would be helpful if we wanted to especially point out differences between the last three clarity categories of diamonds compared to the other categories.

Tables

While we have focused on figures here so far, tables can be incredibly informative at a glance too. If you are looking to display summary numbers, a **table can also visually display**

information.

Using this same dataset, we can use a table to get a quick breakdown of how many males and females there are in the dataset and what percentage of each gender there is.

A few things to keep in mind when making tables is that it's best to:

- Limit the number of digits in the table
- Include a caption
- When possible, keep it simple.

Tables

Effective ways to display data summaries

	Number	Percentage (%)
male	111	55.8
female	88	44.2

Table 1: Gender Breakdown Across Davis Dataset



Table

Tables in R

Now that we have a good understanding of what to consider when making tables, we can practice making good tables in R. To do this, we'll continue to use the diamonds dataset (which is part of the `ggplot2` package). As a reminder, this dataset contains prices and other information about ~54,000 different diamonds. If we want to provide viewers with a summary of these data, we may want to provide information about diamonds broken down by the quality of the diamond's cut. To get our data in the form we want we will use the `dplyr` package, which we discussed in a lesson earlier.

Getting the Data in Order

To start figuring out how the quality of the cut of the diamond affects the price of that diamond, we first have to get the data in order. To do that we'll use the `dplyr` package. This allows us to group the data by the quality of the cut (`cut`) before summarizing the data to determine the number of diamonds in each category (`N`), the minimum price of the diamonds in this category (`min`), the average price (`avg`), and the highest price in the category (`max`).

To get these data in order, you could use the following code. This code groups the data by cut (quality of the diamond) and then calculates the number of diamonds in each group (`N`), the minimum price across each group (`min`), the average price of diamonds across each group (`avg`), and the maximum price within each group (`max`):

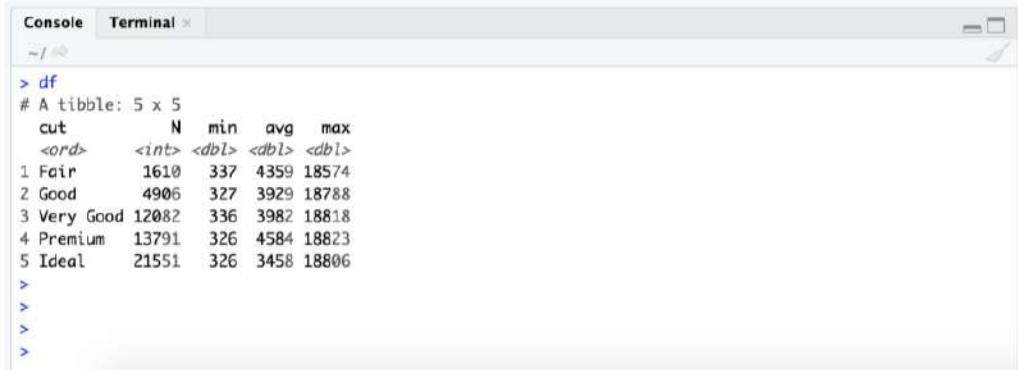
```
# get summary data for table in order
df <- diamonds %>%
  group_by(cut) %>%
  dplyr::summarize(
    N = n(),
    min = min(price),
    avg = mean(price),
    max = max(price),
    .groups = "drop"
  )
```

An Exploratory Table

```
# look at data
df
# A tibble: 5 × 5
  cut          N   min   avg   max
  <ord>     <int> <int> <dbl> <int>
1 Fair       1610   337  4359. 18574
2 Good      4906   327  3929. 18788
3 Very Good 12082   336  3982. 18818
4 Premium    13791   326  4584. 18823
5 Ideal      21551   326  3458. 18806
```

By getting the data summarized into a single object in R (`df`), we're on our way to making an informative table. However, this is clearly just an exploratory table. The output in R from

this code follows some of the good table rules above, but not all of them. At a glance, it will help you to understand the data, but it's not the finished table you would want to send to your boss.



```

Console Terminal ×
~/
> df
# A tibble: 5 × 5
  cut     N   min   avg   max
  <ord> <int> <dbl> <dbl> <dbl>
1 Fair    1610   337  4359 18574
2 Good    4906   327  3929 18788
3 Very Good 12082   336  3982 18818
4 Premium  13791   326  4584 18823
5 Ideal    21551   326  3458 18806
>
>
>
>

```

Exploratory diamonds table

From this output, you, the creator of the table, would be able to see that there are a number of **good qualities**:

- there is a **reasonable number of rows and columns** - There are 5 rows and 5 columns. A viewer can quickly look at this table and determine what's going on.
- the first column **cut** is **organized logically** - The lowest quality diamond category is first and then they are ordered vertically until the highest quality cut (**ideal**)
- comparisons are made **top to bottom** - To compare between the groups, your eye only has to travel up and down, rather than from left to right.

There are also things that **need to be improved** on this table:

- **column headers** could be even more clear
- there's **no caption/title**
- it could be more **aesthetically pleasing**

Improving the Table Output

By-default, R outputs tables in the Console using a monospaced font. However, this limits our ability to format the appearance of the table. To fix the remaining few problems with the table's format, we'll use the `kable()` function from the R package `knitr` and the additional formatting capabilities of the R packages `kableExtra`.

The first step to a prettier table just involves using the `kable()` function from the `knitr` package, which improves the readability of this table. The `knitr` package is *not* a core tidyverse package, so you'll have to be sure it's installed and loaded.

```
# install.packages("knitr")
library(knitr)
```

```
kable(df)
```

cut	N	min	avg	max
Fair	1610	337	4358.758	18574
Good	4906	327	3928.864	18788
Very Good	12082	336	3981.760	18818
Premium	13791	326	4584.258	18823
Ideal	21551	326	3457.542	18806

```
kable(df)
```

cut	N	min	avg	max
Fair	1610	337	4358.758	18574
Good	4906	327	3928.864	18788
Very Good	12082	336	3981.760	18818
Premium	13791	326	4584.258	18823
Ideal	21551	326	3457.542	18806



kable basic output

However, there are still a few issues we want to improve upon:

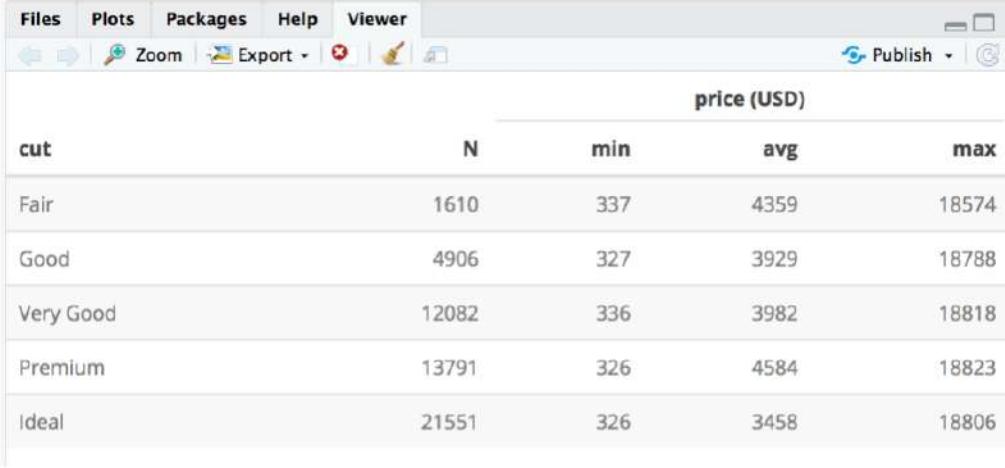
- column names could be more informative
- too many digits in the avg column
- caption/title is missing
- source of data not included

To begin addressing these issues, we can use the `add_header_above` function from `kableExtra()` to specify that the min, avg, and max columns refer to `price` in US dollars (USD). Additionally, `kable()` takes a `digits` argument to specify how many significant digits to display. This takes care of the display of too many digits in the avg column. Finally, we can also style the table so that every other row is shaded, helping our eye to keep each row's information separate from the other rows using `kable_styling()` from `kableExtra`. These few changes really improve the readability of the table.

If you copy this code into your R console, the formatted table will show up in the Viewer tab at the bottom right-hand side of your RStudio console and the HTML code used to generate that table will appear in your console.

```
# install.packages("kableExtra")
library(kableExtra)

# use kable_styling to control table appearance
kable(df, digits=0, "html") %>%
  kable_styling("striped", "bordered") %>%
  add_header_above(c(" " = 2, "price (USD)" = 3))
```



The screenshot shows the RStudio interface with the 'Viewer' tab selected. The menu bar at the top includes 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu is a toolbar with icons for file operations like Open, Save, and Print, as well as 'Zoom' and 'Export' options. The main area displays a table titled 'price (USD)' with the following data:

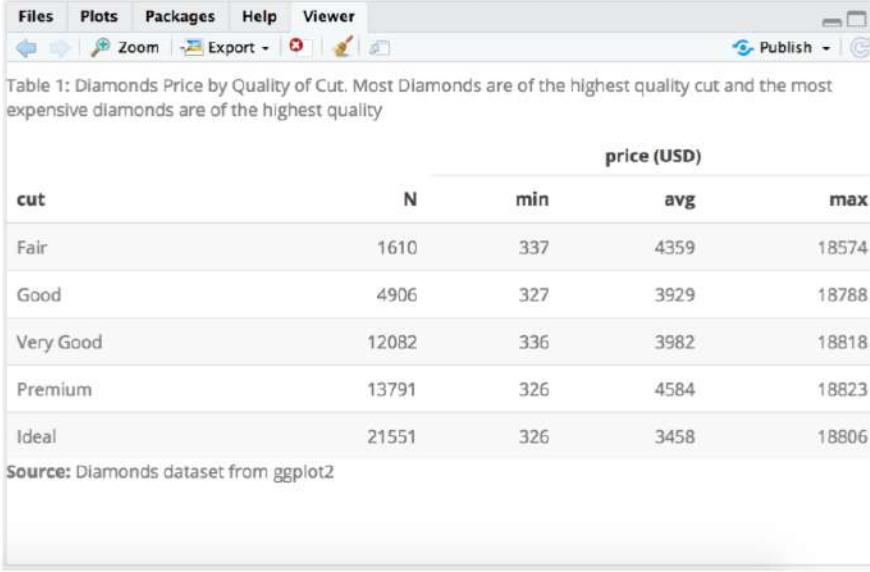
cut	N	min	avg	max
Fair	1610	337	4359	18574
Good	4906	327	3929	18788
Very Good	12082	336	3982	18818
Premium	13791	326	4584	18823
Ideal	21551	326	3458	18806

Viewer tab with formatted table

Annotating Your Table

We mentioned earlier that captions and sourcing your data are incredibly important. The `kable` package allows for a `caption` argument. Below, an informative caption has been included. Additionally, `kableExtra` has a `footnote()` function. This allows you to include the source of your data at the bottom of the table. With these final additions, you have a table that clearly displays the data and could be confidently shared with your boss.

```
kable(df, digits=0, "html", caption="Table 1: Diamonds Price by Quality of Cut.\nMost Diamonds are of the highest quality cut and the most expensive diamonds are of the highest quality") %>%\n  kable_styling("striped", "bordered") %>%\n  # add headers and footnote\n  add_header_above(c(" " = 2, "price (USD)" = 3)) %>%\n  footnote(general = "Diamonds dataset from ggplot2", general_title = "Source:",\n, footnote_as_chunk = TRUE)
```



The screenshot shows the RStudio interface with the 'Viewer' tab selected. The title bar includes 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the title bar are various icons for file operations like saving, opening, and printing. The main content area displays a table titled 'Table 1: Diamonds Price by Quality of Cut.' with the following data:

cut	N	price (USD)		
		min	avg	max
Fair	1610	337	4359	18574
Good	4906	327	3929	18788
Very Good	12082	336	3982	18818
Premium	13791	326	4584	18823
Ideal	21551	326	3458	18806

Below the table, a note reads: 'Source: Diamonds dataset from ggplot2'.

Viewer tab with annotated and formatted table

ggplot2: Extensions

Beyond the *many* capabilities of `ggplot2`, there are a few additional packages that build *on top of* `ggplot2`'s capabilities. We'll introduce a few packages here so that you can:

- directly annotate points on plots (`ggrepel` and `directlabels`)
- combine multiple plots (`cowplot` + `patchwork`)
- generate animated plots (`ggridge`)

These are referred to as `ggplot2` extensions.

ggrepel

`ggrepel` “provides geoms for `ggplot2` to repel overlapping text labels.”

To demonstrate the functionality within the `ggrepel` package and demonstrate cases where it would be needed, let’s use a dataset available from R - the `mtcars` dataset:

The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

This dataset includes information about 32 different cars. Let’s first convert this from a `data.frame` to a `tibble`. Note that we will keep the rownames and make it a new variable called `model`.

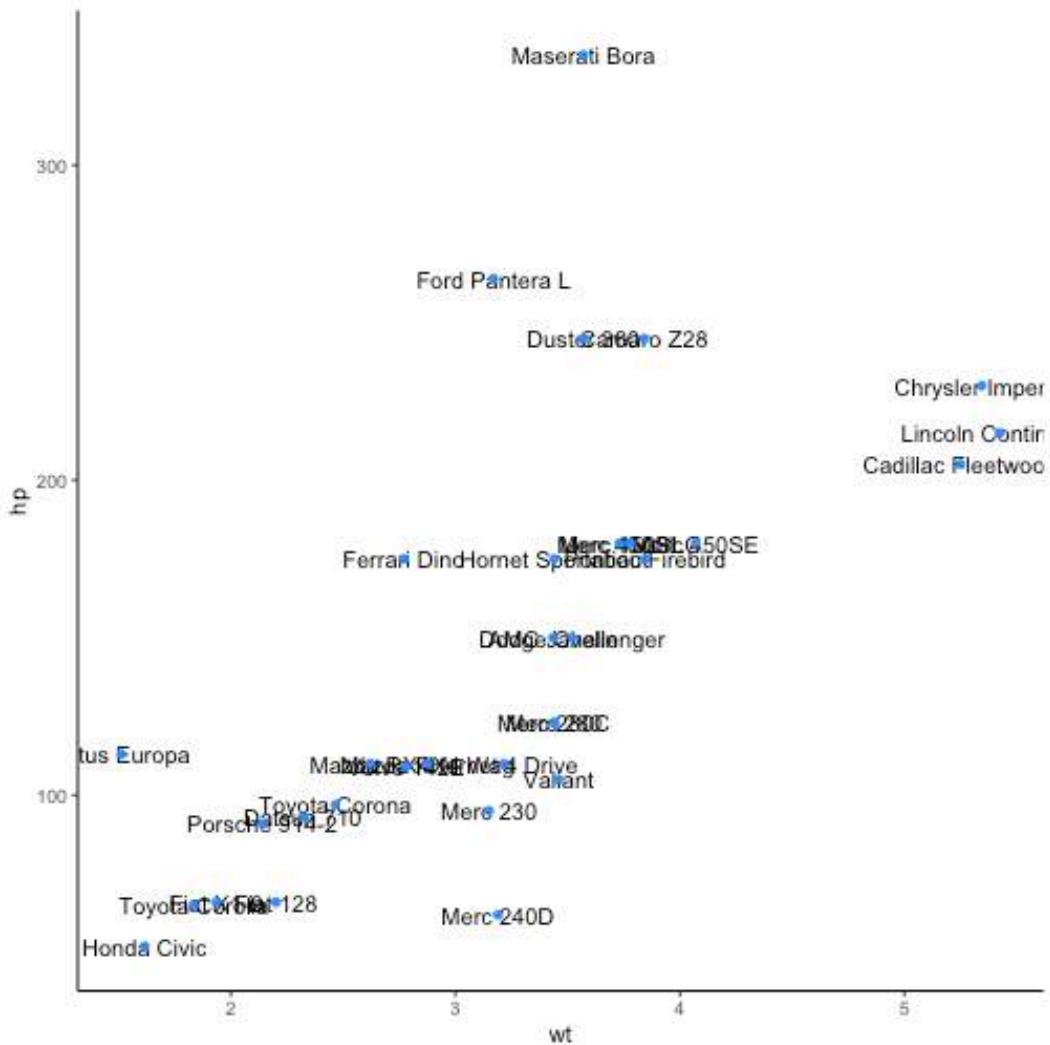
```
# see first 6 rows of mtcars
mtcars <- mtcars %>%
  as_tibble(rownames = "model")
head(mtcars)

# A tibble: 6 × 12
  model      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <chr>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Mazda RX4    21     6   160   110   3.9   2.62  16.5     0     1     4     4
2 Mazda RX4 W...  21     6   160   110   3.9   2.88  17.0     0     1     4     4
3 Datsun 710   22.8    4   108    93   3.85  2.32  18.6     1     1     4     1
4 Hornet 4 Dr... 21.4    6   258   110   3.08  3.22  19.4     1     0     3     1
5 Hornet Spor... 18.7    8   360   175   3.15  3.44  17.0     0     0     3     2
6 Valiant     18.1    6   225   105   2.76  3.46  20.2     1     0     3     1
```

What if we were to plot a scatterplot between horsepower (`hp`) and weight (`wt`) of each car and wanted to label each point in that plot with the care model.

If we were to do that with `ggplot2`, we’d see the following:

```
# label points without ggrepel
ggplot(mtcars, aes(wt, hp, label = model)) +
  geom_text() +
  geom_point(color = 'dodgerblue') +
  theme_classic()
```



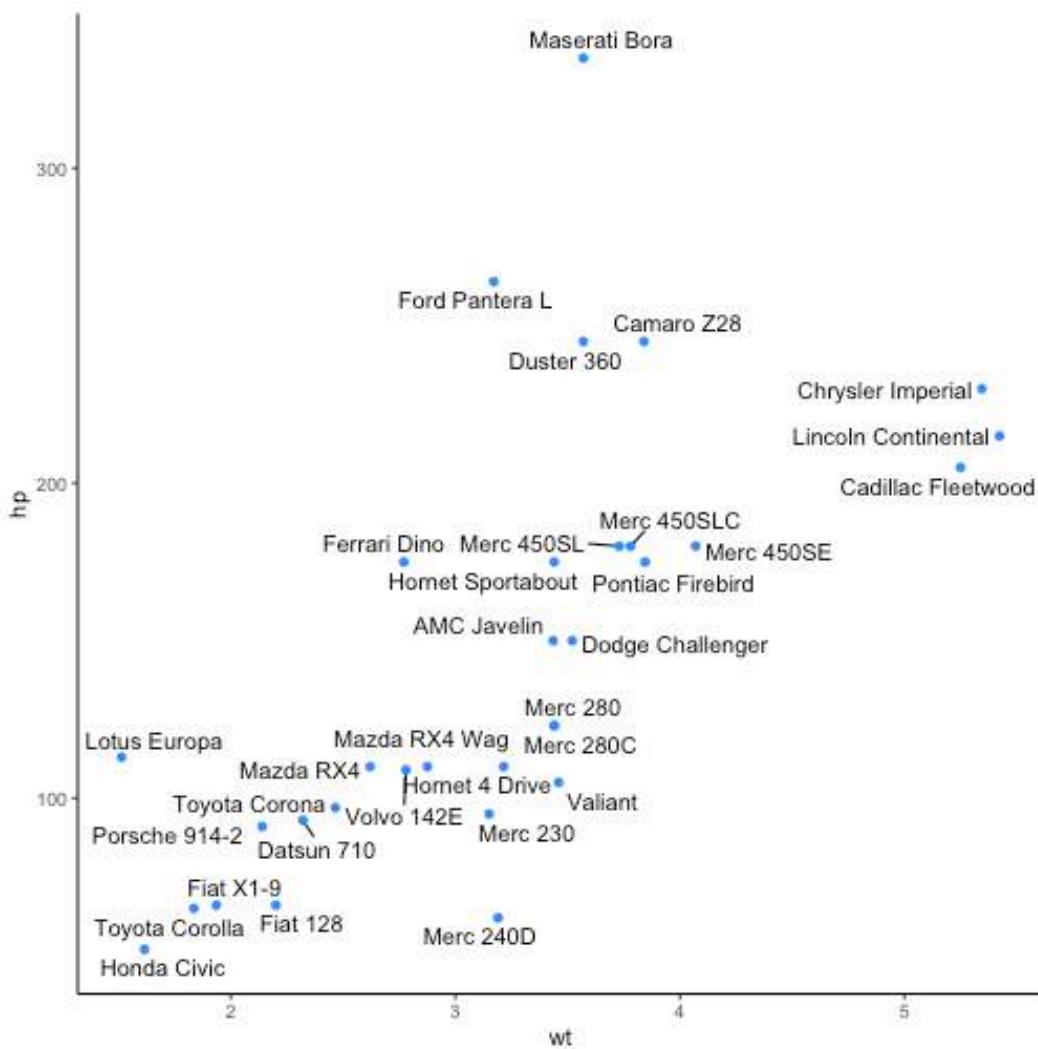
plot of chunk unnamed-chunk-56

The overall trend is clear here - the more a car weights, the more horsepower it tends to have. However, many of the labels are overlapping and impossible to read - this is where `ggrepel`

plays a role:

```
# install and load package
# install.packages("ggrepel")
library(ggrepel)

# label points with ggrepel
ggplot(mtcars, aes(wt, hp, label = model)) +
  geom_text_repel() +
  geom_point(color = 'dodgerblue') +
  theme_classic()
```



The only bit of code here that changed was that we changed `geom_text()` to `geom_text_repel()`. This, like `geom_text()` adds text directly to the plot. However, it also helpfully repels overlapping labels away from one another and away from the data points on the plot.

Custom Formatting

Within `geom_text_repel()`, there are a number of additional formatting options available. We'll cover a number of the most important here, but the [ggrepel vignettes](#) explore these

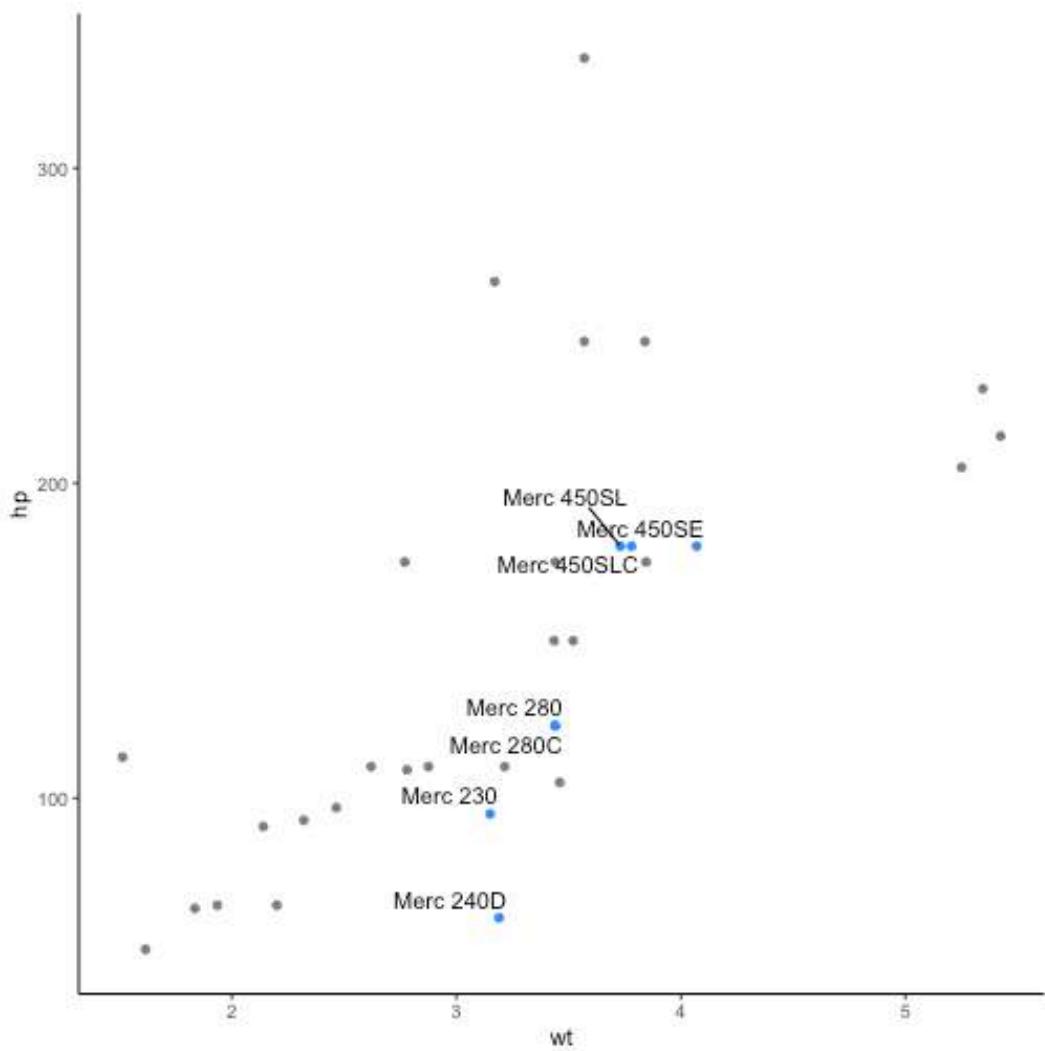
further.

Highlighting Specific Points

Often, you do not want to highlight *all* the points on a plot, but want to draw your viewer's attention to a few *specific* points. To do this and say highlight only cars that are the make Mercedes, you could use the following approach:

```
# create a new column "merc" with true or false for Mercedes
# value is true for rows with "Merc" in model column
mtcars <- mtcars%>%
  mutate(merc = str_detect(string = model, pattern = "Merc"))

# Let's just label these items and manually color the points
ggplot(mtcars, aes(wt, hp, label = model)) +
  geom_point(aes(color = merc)) +
  scale_color_manual(values = c("grey50", "dodgerblue")) +
  geom_text_repel(data = filter(mtcars, merc == TRUE),
    nudge_y = 1,
    hjust = 1,
    direction = "y") +
  theme_classic() +
  theme(legend.position = "none")
```



Here, notice that we first create a new column in our dataframe called `merc`. Here, we include the model of the car, only if “Merc” is in the model of the car’s name.

We then, specify that only these Mercedes cars should be labeled *and* that we only want them colored blue if they are Mercedes. This allows us to focus in on those few points we’re interested in. And, we can see that among the cars in this dataset, Mercedes tend to be of average weight but have varying horsepower, depending on the model of the car.

Text Alignment

Other times, you want to ensure that your labels are aligned on the top or bottom edge, relative to one another. This can be controlled using the `hjust` and `vjust` arguments. The values for particular alignment are:

- `hjust = 0` | to left-align
- `hjust = 0.5` | to center
- `hjust = 1` | to right-align

Additionally, you can adjust the starting position of text vertically with `nudge_y` (or horizontally with `nudge_x`). To use this, you'll specify the distance from the point to the label you want to use.

You can also allow the labels to move horizontally with `direction = "x"` (or vertically with `direction = "y"`). This will put the labels at a right angle from the point. The default here is “both”, which allows labels to move in both directions.

For example, what if we wanted to plot the relationship between quarter mile time (`qsec`) and miles per gallon (`mpg`) to identify cars that get the best gas mileage.

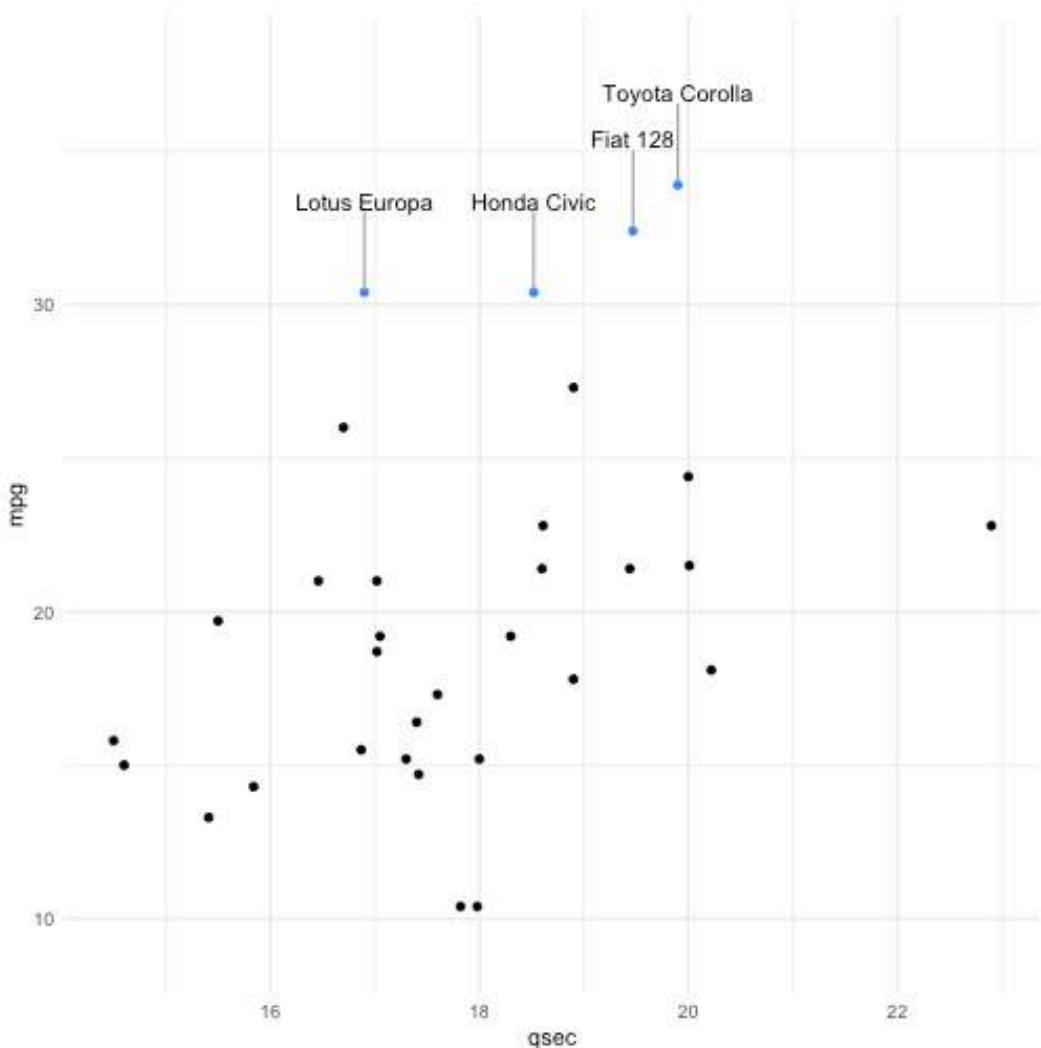
To do this, we're specifying within `geom_text_repel()` the following arguments:

- `data` | only label those cars with gas mileage > 30 mpg
- `nudge_y` | position all labels so that they're vertically aligned
- `hjust` | center-align the labels
- `direction` | allow labels to move horizontally

For further customization, we're also changing the segment color from the default black to a light gray (“gray60”).

```
# customize within geom_text_repel
# first create a new column for mpg > 30 within mtcars and pipe this into ggplot
mtcars %>%
  mutate(mpg_highlight = case_when(mpg > 30 ~ "high", mpg < 30 ~ "low")) %>%
  ggplot(aes(qsec, mpg, label = model)) +
  geom_point(aes(color = mpg_highlight)) +
  scale_color_manual(values = c("dodgerblue", "black")) +
  theme_minimal() +
  theme(legend.position = "none") +
  geom_text_repel(data = mtcars %>% filter(mpg > 30),
```

```
nudge_y = 3,  
hjust = 0.5,  
direction = "x",  
segment.color = "gray60") +  
scale_x_continuous(expand = c(0.05, 0.05)) +  
scale_y_continuous(limits = c(9, 38))
```



Notice that we've also had to provide the plot with more room by customizing the x- and y-

axes, using the final two lines of code you see above.

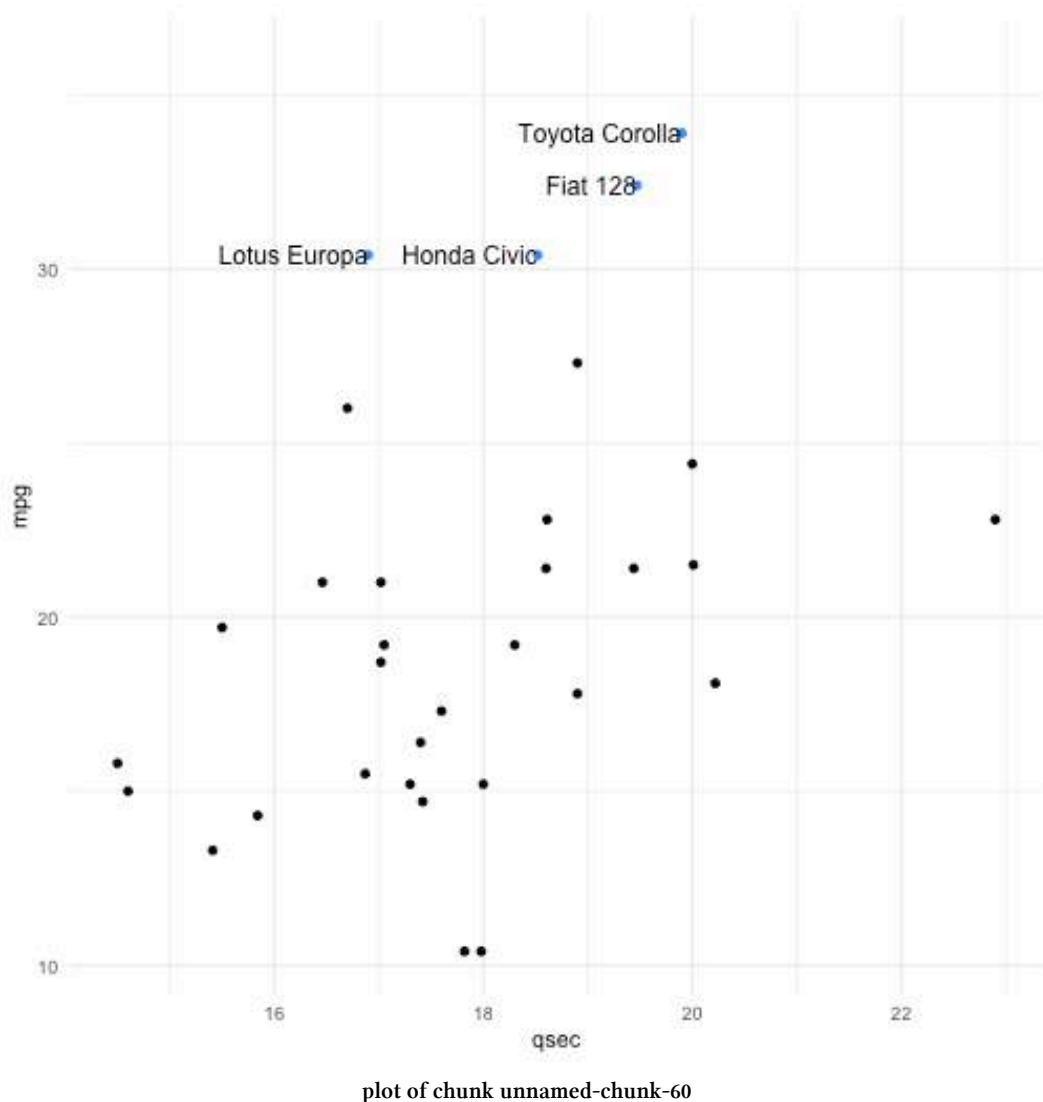
With this plot, it is clear that there are four cars with $\text{mpg} > 30$. And, among these, we now know that the Toyota Corolla has the fastest quarter mile time, all thanks to direct labeling of the points!

directlabels

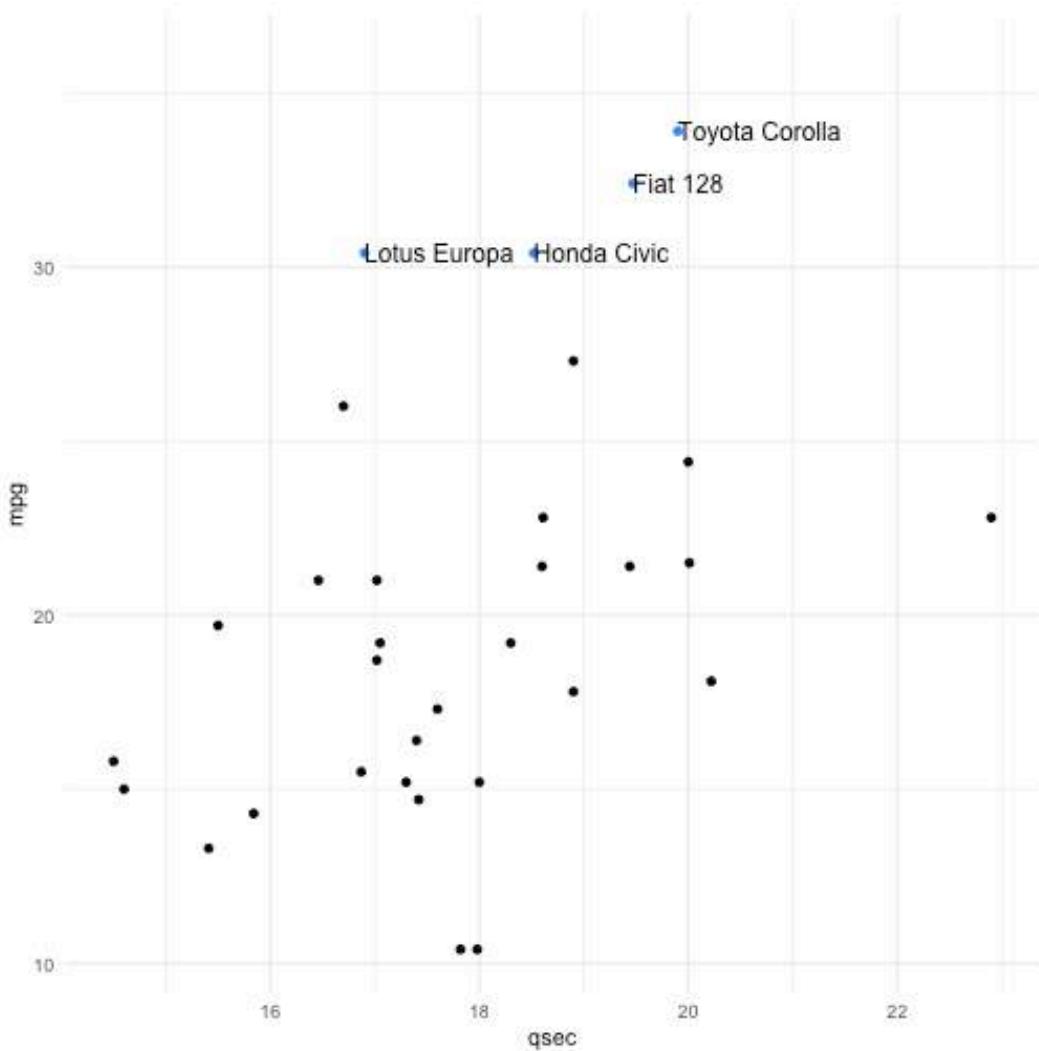
The `directlabels` package also helps you to add labels directly to plots. There are functions that allow you to also add labels that generally don't overlap using less code than `ggrepel`, however there are less specification options.

There are several method options for adding direct labels to scatter plots, such as: `first.points` (which will place the label on the left in a scatterplot), and `last.points` (which will place the label on the right in a scatterplot).

```
#install.packages("directlabels")
library(directlabels)
mtcars %>%
  mutate(mpg_highlight = case_when(mpg > 30 ~ "high", mpg < 30 ~ "low")) %>%
  ggplot(aes(qsec, mpg, label = model)) +
  geom_point(aes(color = mpg_highlight)) +
  scale_color_manual(values = c("dodgerblue", "black")) +
  scale_x_continuous(expand = c(0.05, 0.05)) +
  scale_y_continuous(limits = c(NA, 36)) +
  geom_dl(data = filter(mtcars, mpg > 30), aes(label = model),
          method = list(c("first.points")),
          cex = 1) +
  theme_minimal() +
  theme(legend.position = "none")
```

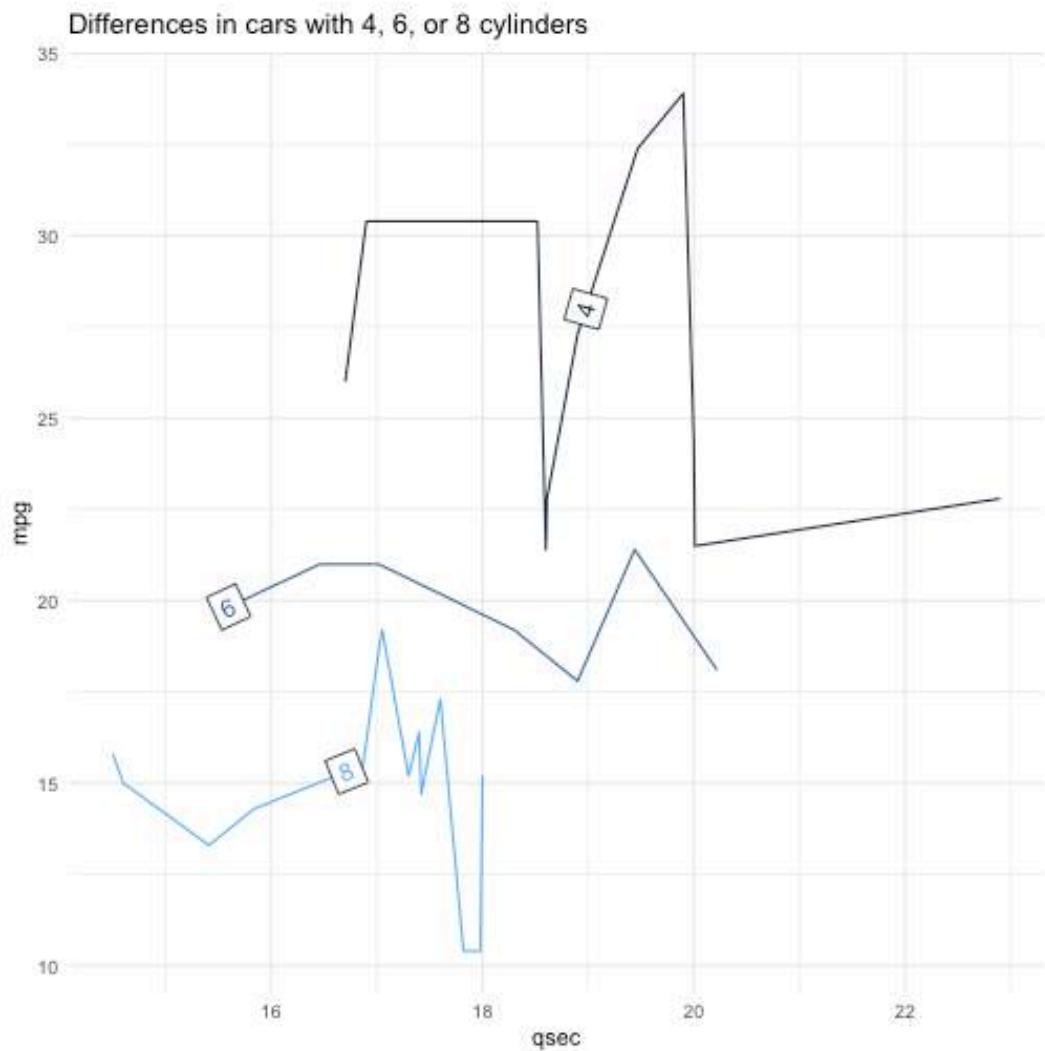


```
mtcars %>%
  mutate(mpg_highlight = case_when(mpg > 30 ~ "high", mpg < 30 ~ "low")) %>%
  ggplot(aes(qsec, mpg, label = model)) +
  geom_point(aes(color = mpg_highlight)) +
  scale_color_manual(values = c("dodgerblue", "black")) +
  scale_x_continuous(expand = c(0.05, 0.05)) +
  scale_y_continuous(limits = c(NA, 36)) +
  geom_dl(data = filter(mtcars, mpg > 30), aes(label = model),
         method = list(c("last.points"),
                     cex = 1)) +
  theme_minimal() +
  theme(legend.position = "none")
```



This package is especially useful for labeling lines in a lineplot. There are several methods, one of which is the `angled.boxes` method. This often negates the need for a legend.

```
ggplot(mtcars, aes(qsec, mpg, color = cyl, group = cyl)) +
  geom_line() +
  geom_dl(aes(label = cyl),
    method = list(c("angled.boxes")),
    cex = 1) +
  theme_minimal() +
  theme(legend.position = "none") +
  labs(title = "Differences in cars with 4, 6, or 8 cylinders")
```



See [here](#) for more information about this package.

cowplot

Beyond customization within `ggplot2` and labeling points, there are times you'll want to arrange multiple plots together in a single grid, applying a standard theme across all plots. This often occurs when you're ready to present or publish your visualizations. When you're ready to share and present your work with others, you want to be sure your visualizations are

conveying the point you want to convey to your viewer in as simple a plot as possible. And, if you’re presenting multiple plots, you typically want to ensure that they have a consistent theme from one plot to the next. This allows the viewer to focus on the data you’re plotting, rather than changes in theme. The `cowplot` package assists in this process!

Theme

The standard theme within `cowplot`, which works for many types of plots is `theme_cowplot()`. This theme is very similar to `theme_classic()`, which removes the background color, removes grid lines, and plots only the x- and y- axis lines (rather than a box around all the data). We’ll use this theme for the examples from this package. However, note that there are a number of additional themes available from the `cowplot` package. We will use the number 12 within this function to indicate that we want to use a font size of 12.

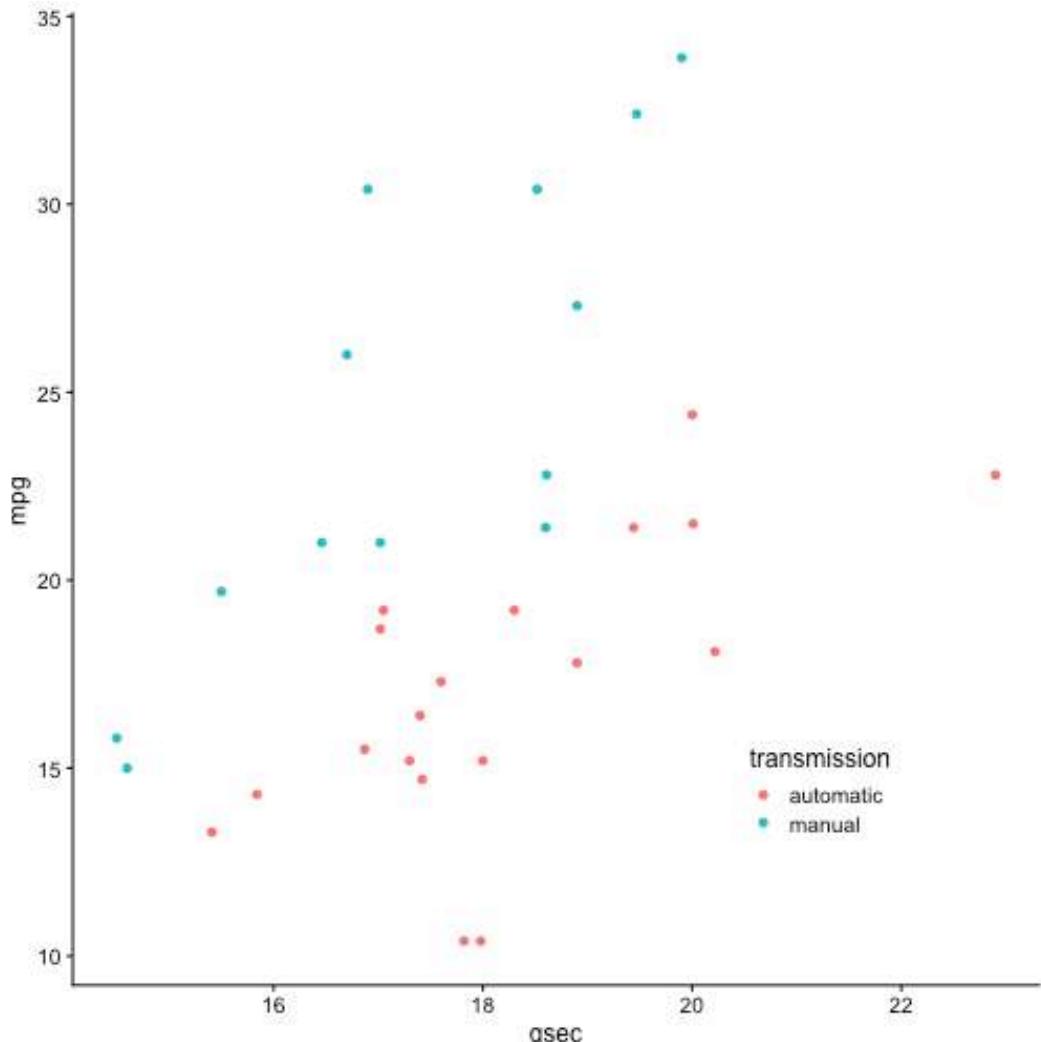
```
# install and load package
# install.packages("cowplot")
library(cowplot)
```

We’ll continue to use the `mtcars` dataset for these examples. Here, using the `forcats` package (which is part of the core tidyverse), we’ll add two new columns: `transmission`, where we recode the `am` column to be “automatic” if `am == 0` and “manual” if `am == 1`, and `engine`, where we recode the `vs` column to be “v-shaped” if `vs == 0` and “straight” if `vs == 1`.

```
mtcars <- mtcars %>%
  mutate(transmission = fct_recode(as.factor(am), "automatic" = "0", "manual" =\n"1"),
        engine = fct_recode(as.factor(vs), "v-shaped" = "0", "straight" = "1"))
```

We’ll use these data to generate a scatterplot plotting the relationship between 1/4 mile speed (`qsec`) and gas mileage (`mpg`). We’ll color the points by this new column `transmission` and apply `theme_cowplot`. Finally, we’ll assign this to `p1`, as we’ll ultimately generate a few different plots.

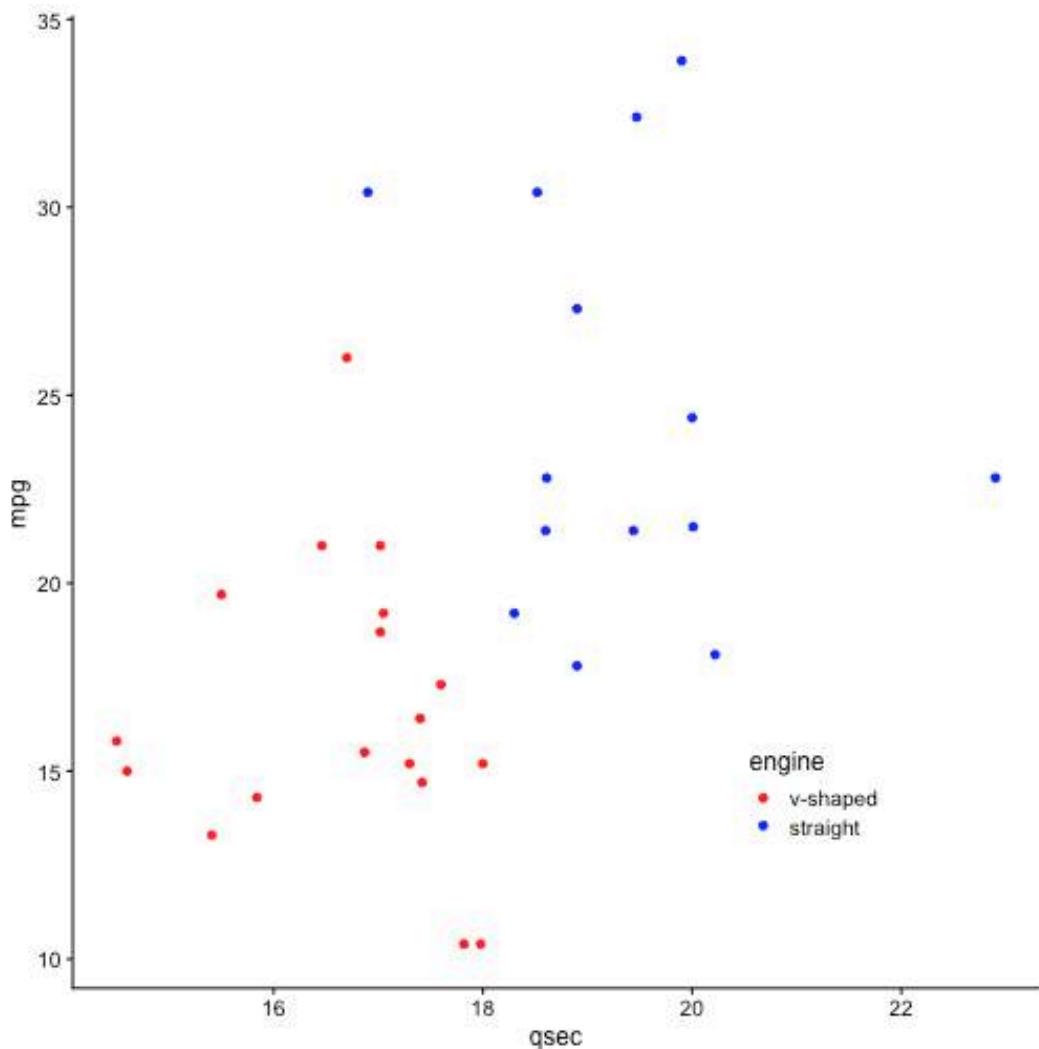
```
p1 <- ggplot(mtcars, aes(qsec, mpg, color = transmission)) +  
  geom_point() +  
  theme_cowplot(12) +  
  theme(legend.position = c(0.7, 0.2))  
p1
```



Let's make a similar plot, but color by engine type. We'll want to manually change the colors here so that we aren't using the same colors for transmission and engine. We'll store this in

p2.

```
p2 <- ggplot(mtcars, aes(qsec, mpg, color = engine)) +  
  geom_point() +  
  scale_color_manual(values = c("red", "blue")) +  
  theme_cowplot(12) +  
  theme(legend.position = c(0.7, 0.2))  
p2
```



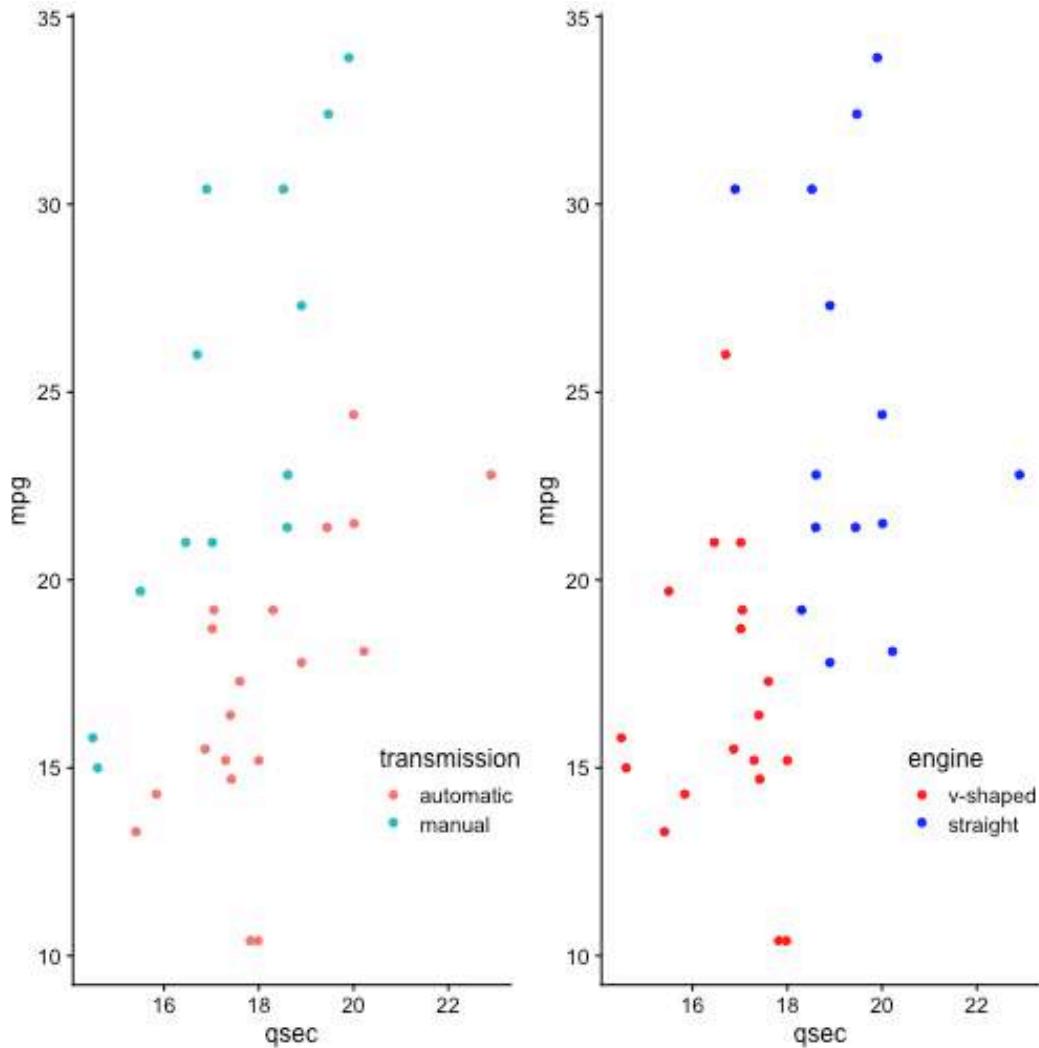
plot of chunk unnamed-chunk-65

Great - we've now got two plots with the same theme and similar appearance. What if we wanted to combine these into a single grid for presentation purposes?

Multiple Plots

Combining plots is made simple within the `cowplot` package using the `plot_grid()` function:

```
# plot side by side  
plot_grid(p1, p2, ncol = 2)
```

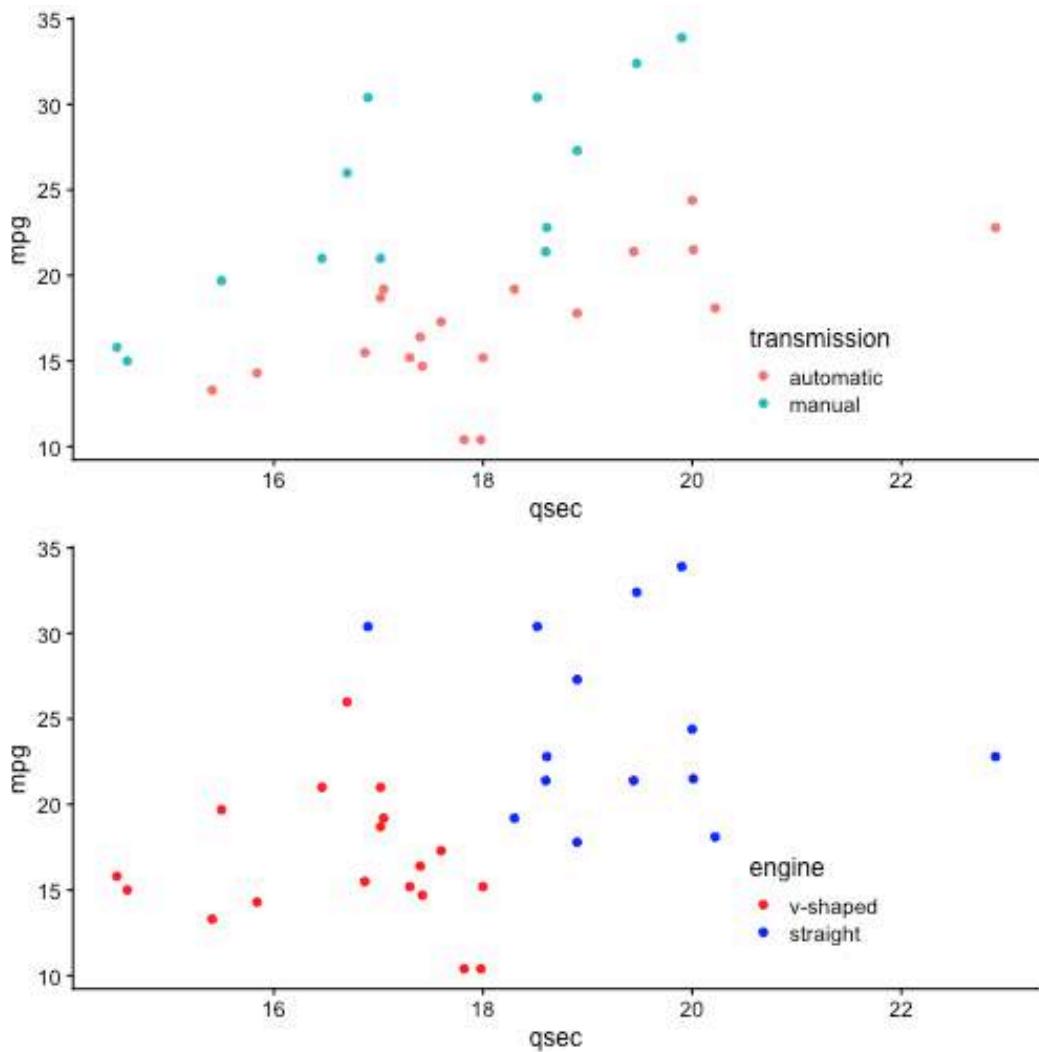


plot of chunk unnamed-chunk-66

Here, we specify the two plots we'd like to plot on a single grid and we also optionally include how many columns we'd like using the `ncol` parameter.

To plot these one on top of the other, you could specify for `plot_grid()` to use a single column.

```
# plot on top of one another
plot_grid(p1, p2, ncol = 1)
```

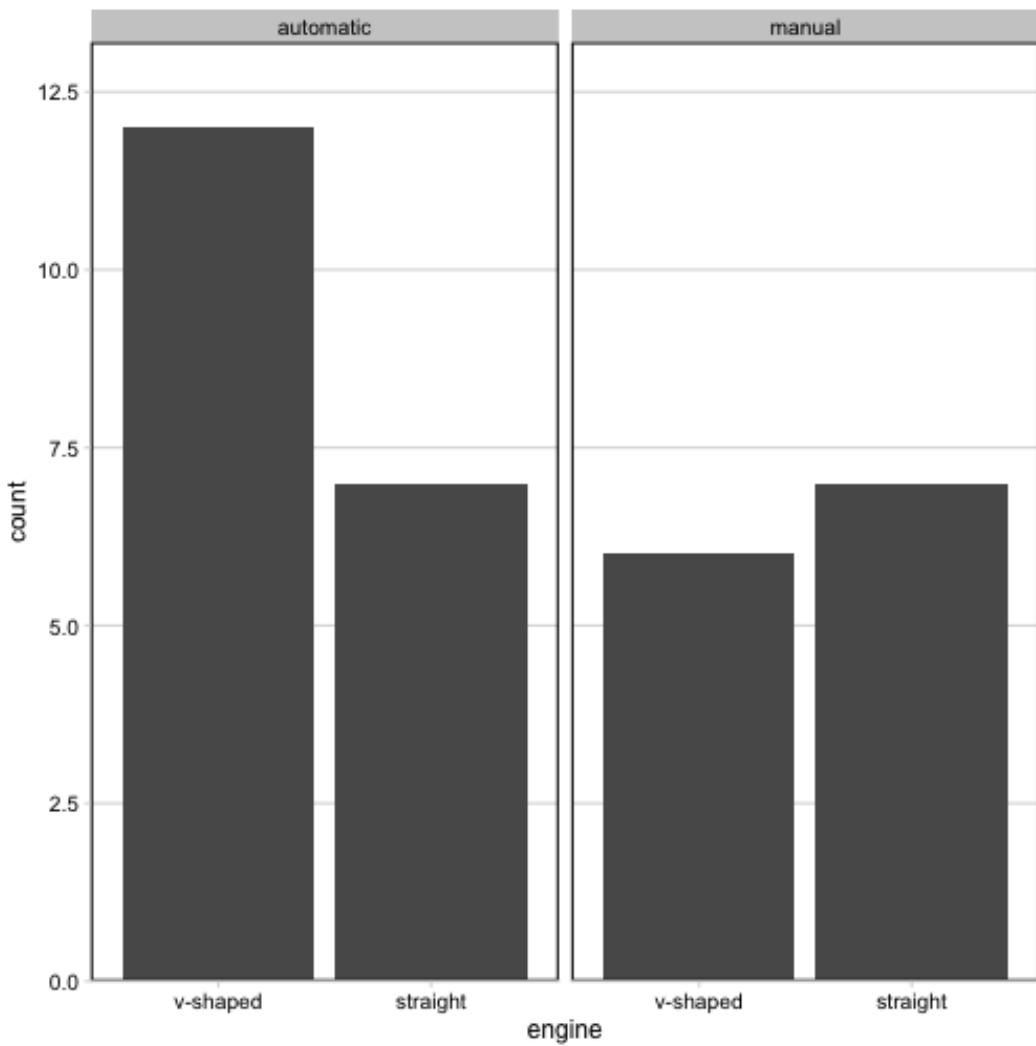


plot of chunk unnamed-chunk-67

Note that by default, the plots will share the space equally, but it's also possible to make one larger than the other within the grid using `rel_widths` and `rel_heights`.

For example, if you had a faceted plot summarizing the number of cars by transmission and engine (which we'll call `p3`):

```
# generate faceted plot
p3 <- ggplot(mtcars, aes(engine)) +
  geom_bar() +
  facet_wrap(~transmission) +
  scale_y_continuous(expand = expand_scale(mult = c(0, 0.1))) +
  theme_minimal_hgrid(12) +
  panel_border(color = "black") +
  theme(strip.background = element_rect(fill = "gray80"))
Warning: `expand_scale()` is deprecated; use `expansion()` instead.
p3
```

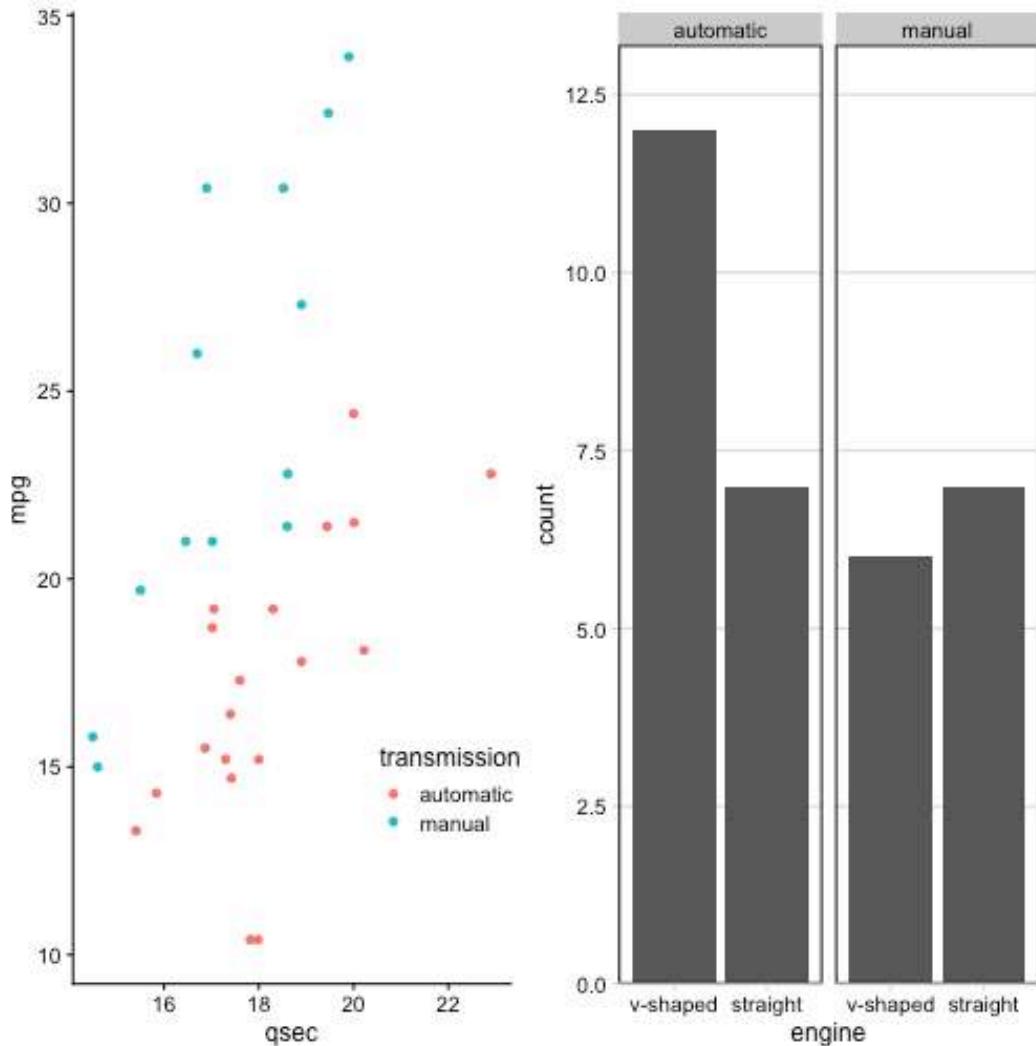


plot of chunk unnamed-chunk-68

Note that for this plot we've chosen a different theme, allowing for horizontal grid lines. This can be helpful when visualizing bar plots.

If we were to plot these next to one another using the defaults, the faceted plot would be squished:

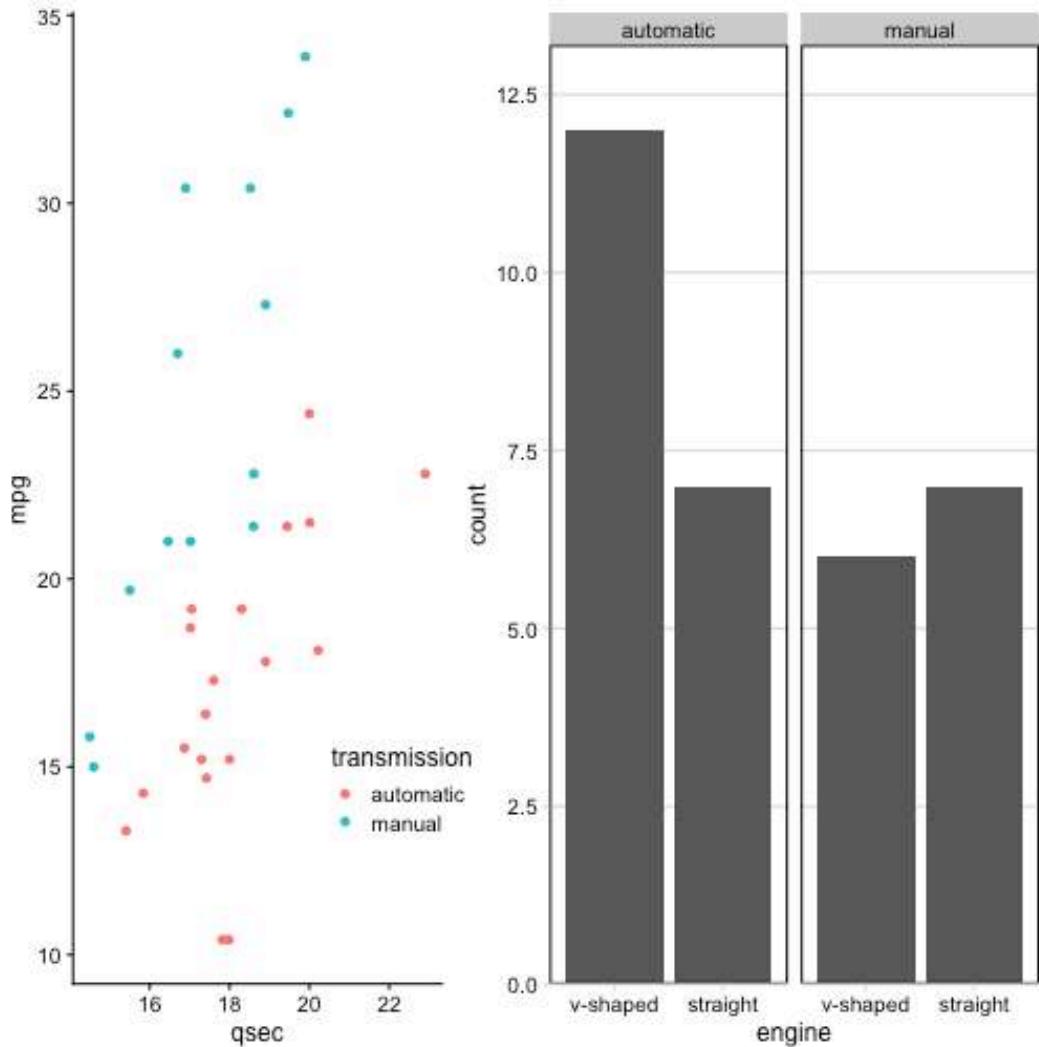
```
# plot next to one another  
plot_grid(p1, p3)
```



plot of chunk unnamed-chunk-69

We can use `rel_widths` to specify the relative width for the plot on the left relative to the plot on the right:

```
# control relative spacing within grid  
plot_grid(p1, p3, rel_widths = c(1, 1.3))
```

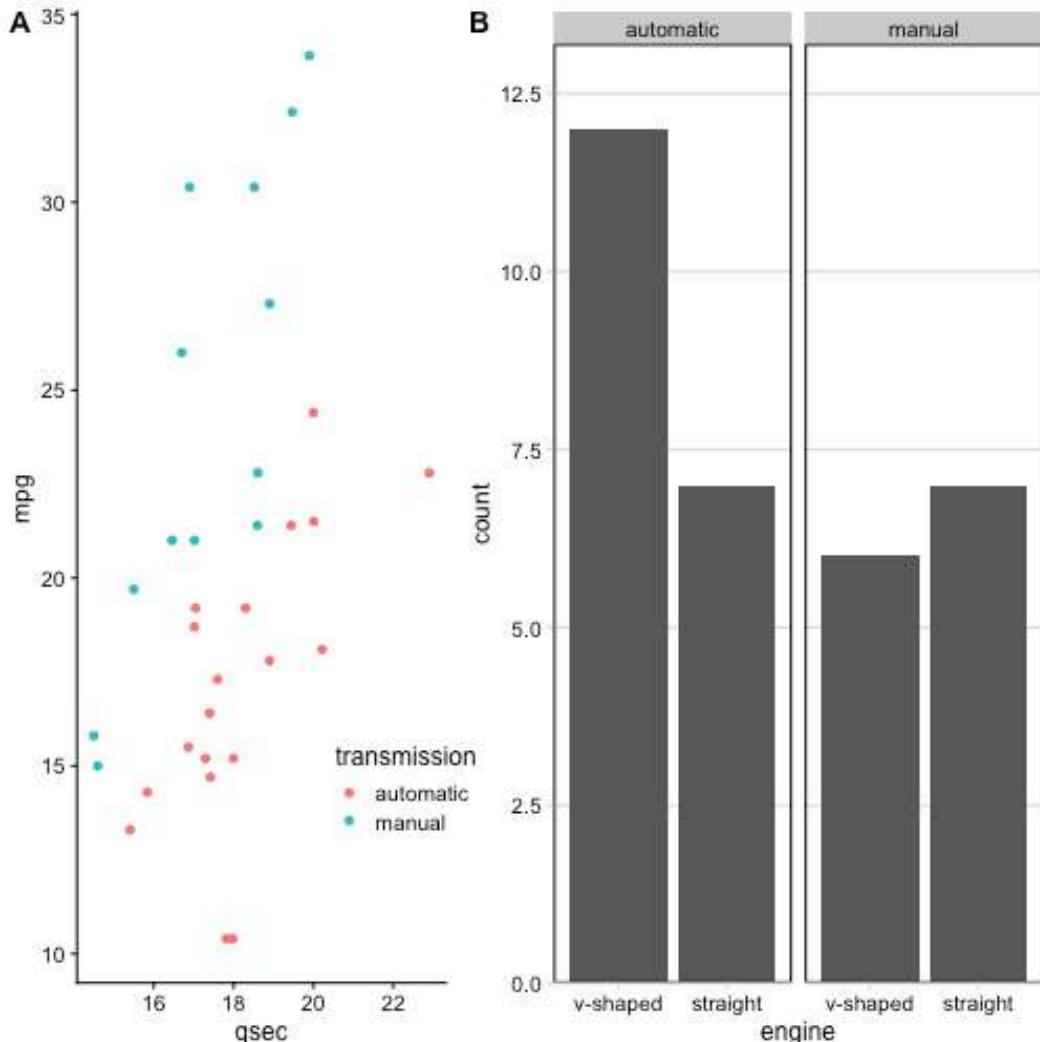


plot of chunk unnamed-chunk-70

Adding Labels

Within these grids, you'll often want to label these plots so that you can refer to them in your report or presentation. This can be done using the `labels` parameter within `plot_grid()`:

```
# add A and B labels to plots
plot_grid(p1, p3, labels = "AUTO", rel_widths = c(1, 1.3))
```



plot of chunk unnamed-chunk-71

Adding Joint Titles

Finally, when generating grids with multiple plots, at times you'll often want a single title to explain what's going on across the plots. Here, the process *looks* slightly confusing, but this is only because we're putting all of these `cowplot` pieces together.

Generally, there are three steps:

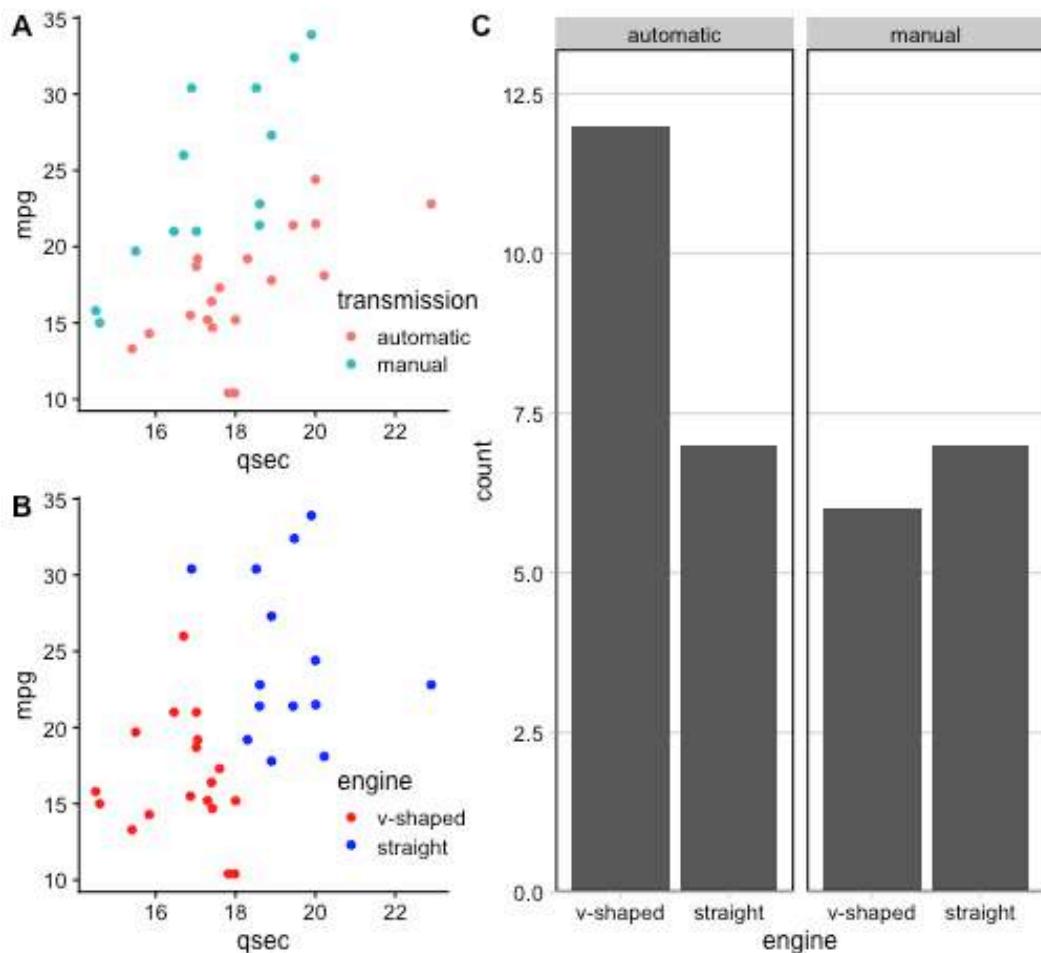
1. Create grid with plots
2. Create title object
3. Piece title and grid of plots together

```
# use plot_grid to generate plot with 3 plots together
first_col <- plot_grid(p1, p2, nrow = 2, labels = c('A', 'B'))
three_plots <- plot_grid(first_col, p3, ncol = 2, labels = c('', 'C'), rel_widths =
hs = c(1, 1.3))

# specify title
title <- ggdraw() +
  # specify title and alignment
  draw_label("Transmission and Engine Type Affect Mileage and 1/4 mile time",
            fontface = 'bold', x = 0, hjust = 0) +
  # add margin on the left of the drawing canvas,
  # so title is aligned with left edge of first plot
  theme(plot.margin = margin(0, 0, 0, 7))

# put title and plots together
plot_grid(title, three_plots, ncol = 1, rel_heights = c(0.1, 1))
```

Transmission and Engine Type Affect Mileage and 1/4 mile time



plot of chunk unnamed-chunk-72

And, just like that we've got three plots, labeled, spaced out nicely in a grid, with a shared title, all thanks to the functionality within the `cowplot` package.

patchwork

The `patchwork` package is similar to the `cowplot` package in that they both are helpful for combining plots together. They each allow for different plot modifications, so it is useful to know about both packages.

With the `patchwork` package, plots can be combined using a few operators, such as `"+"`, `"/"`, and `"|"`.

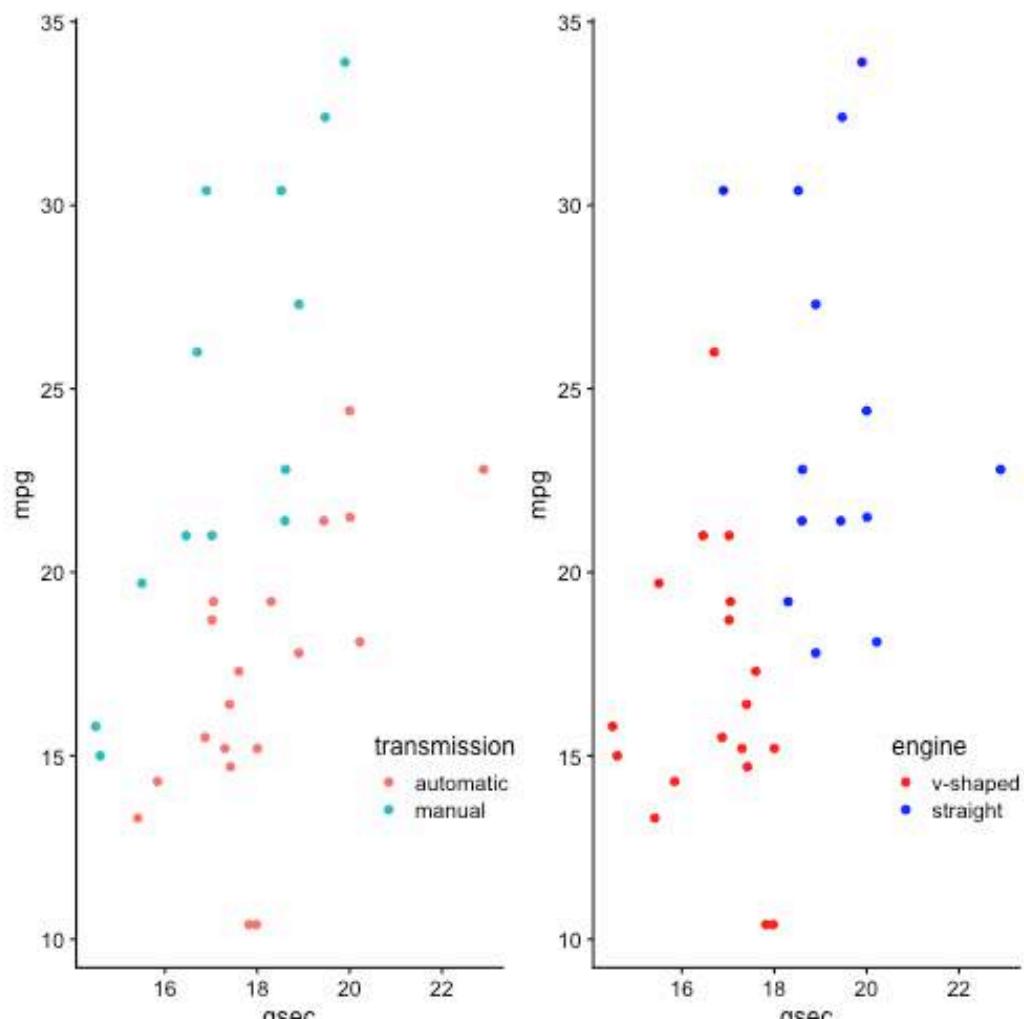
To combine two plots together we can simply add them together with the `+` sign or place them next to one another using the `|`:

```
#install.packages(patchwork)
library(patchwork)
```

```
Attaching package: 'patchwork'
The following object is masked from 'package:cowplot':

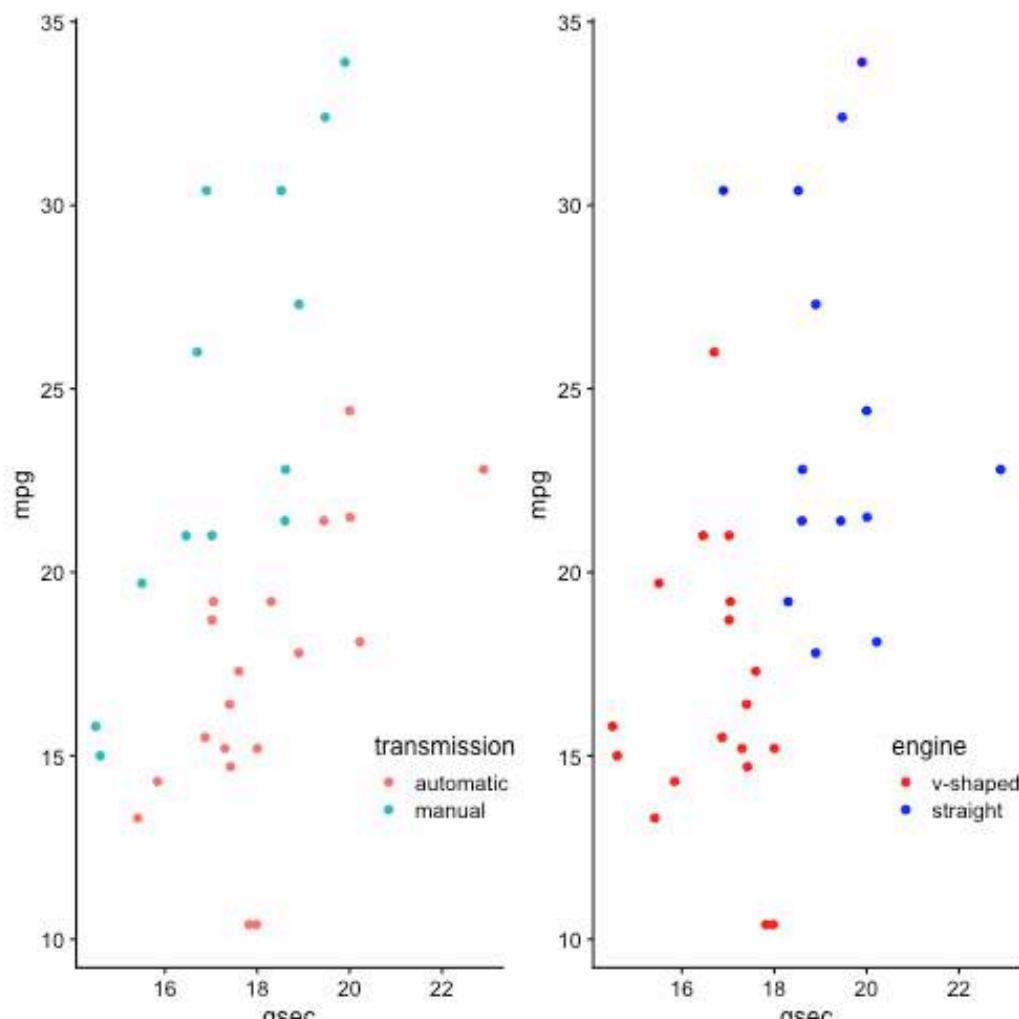
```

```
align_plots
p1 + p2
```



plot of chunk unnamed-chunk-73

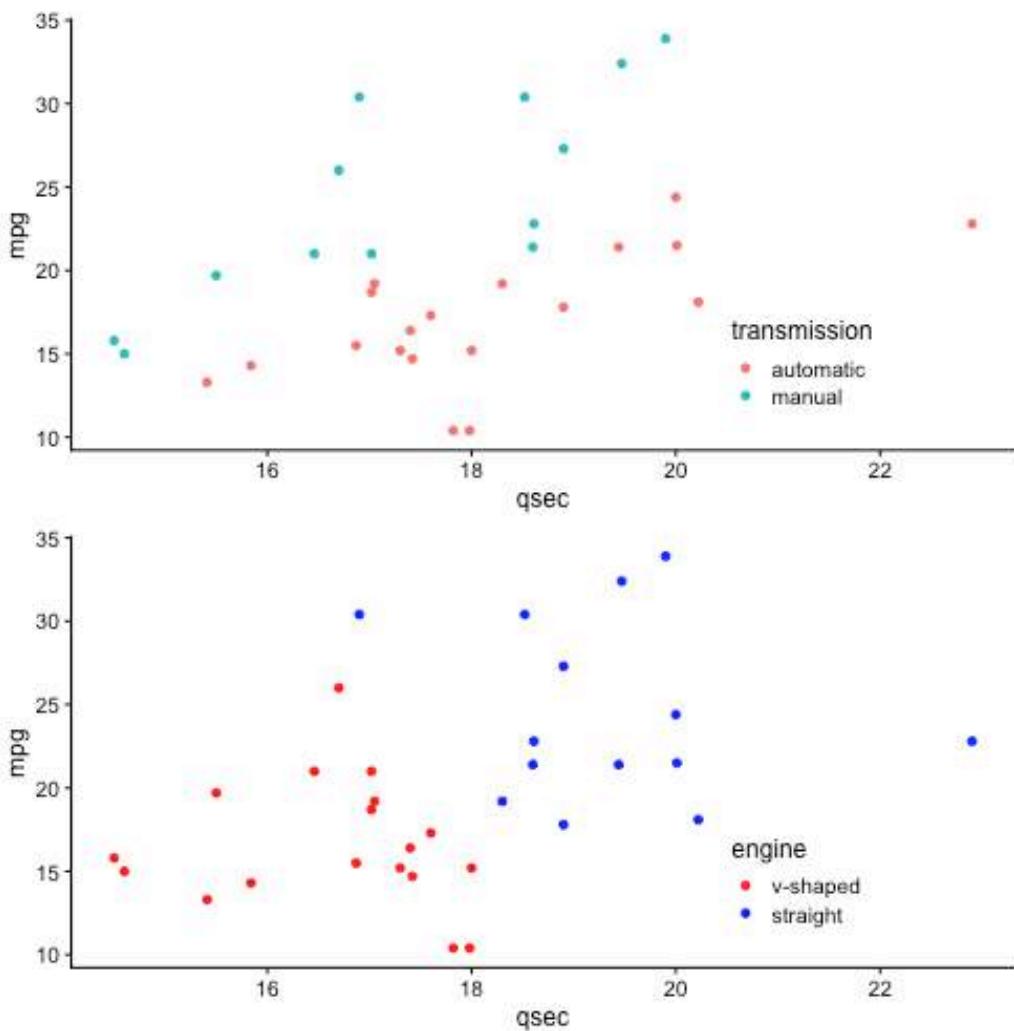
p1 | p2



plot of chunk unnamed-chunk-73

If we want a plot above another plot we can use the "/" symbol:

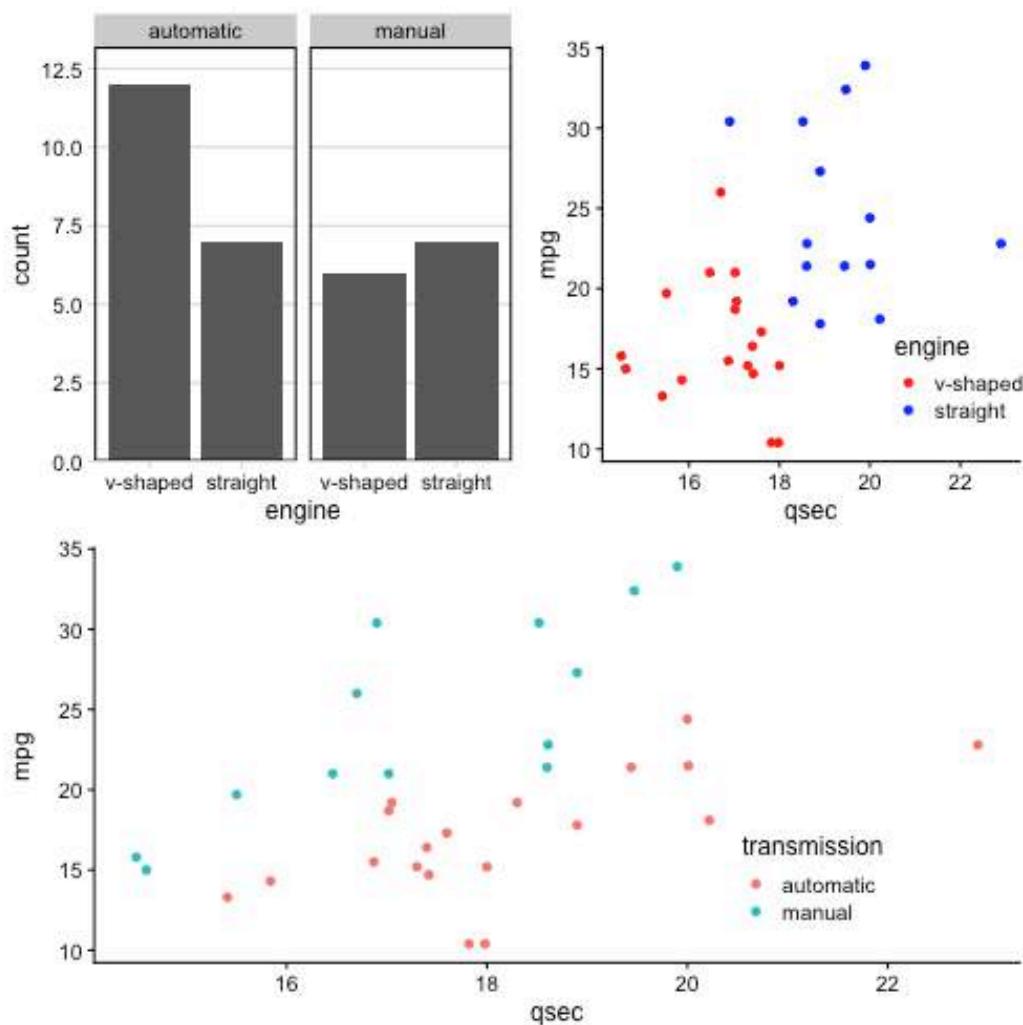
p1 / p2

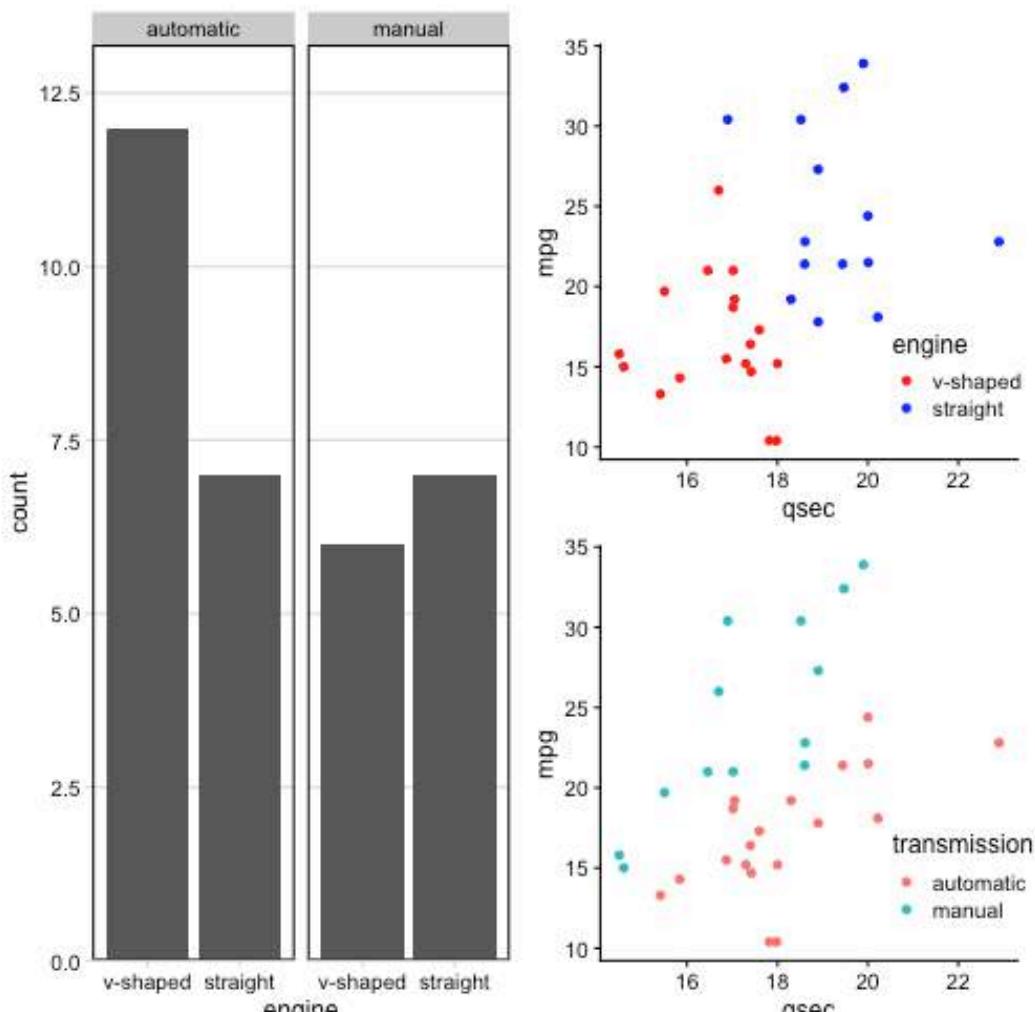


plot of chunk unnamed-chunk-74

Thus, to combine multiple plots in a more complicated layout, one can combine two plots on one row and have a third plot on another row like this:

```
(p3 + p2) / p1
```

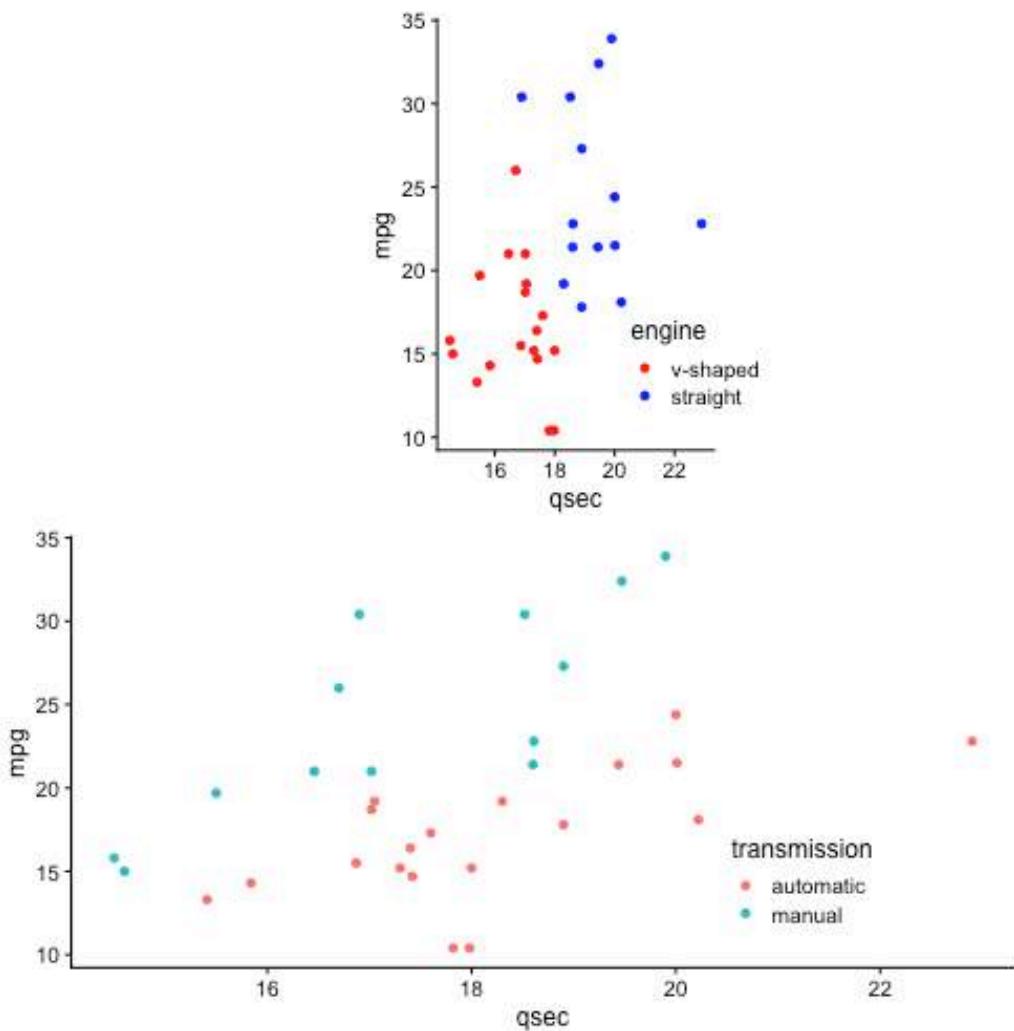




plot of chunk unnamed-chunk-76

You can also empty plot spacers using the `plot_spacer()` function like so:

```
(plot_spacer() + p2 + plot_spacer()) / p1
```

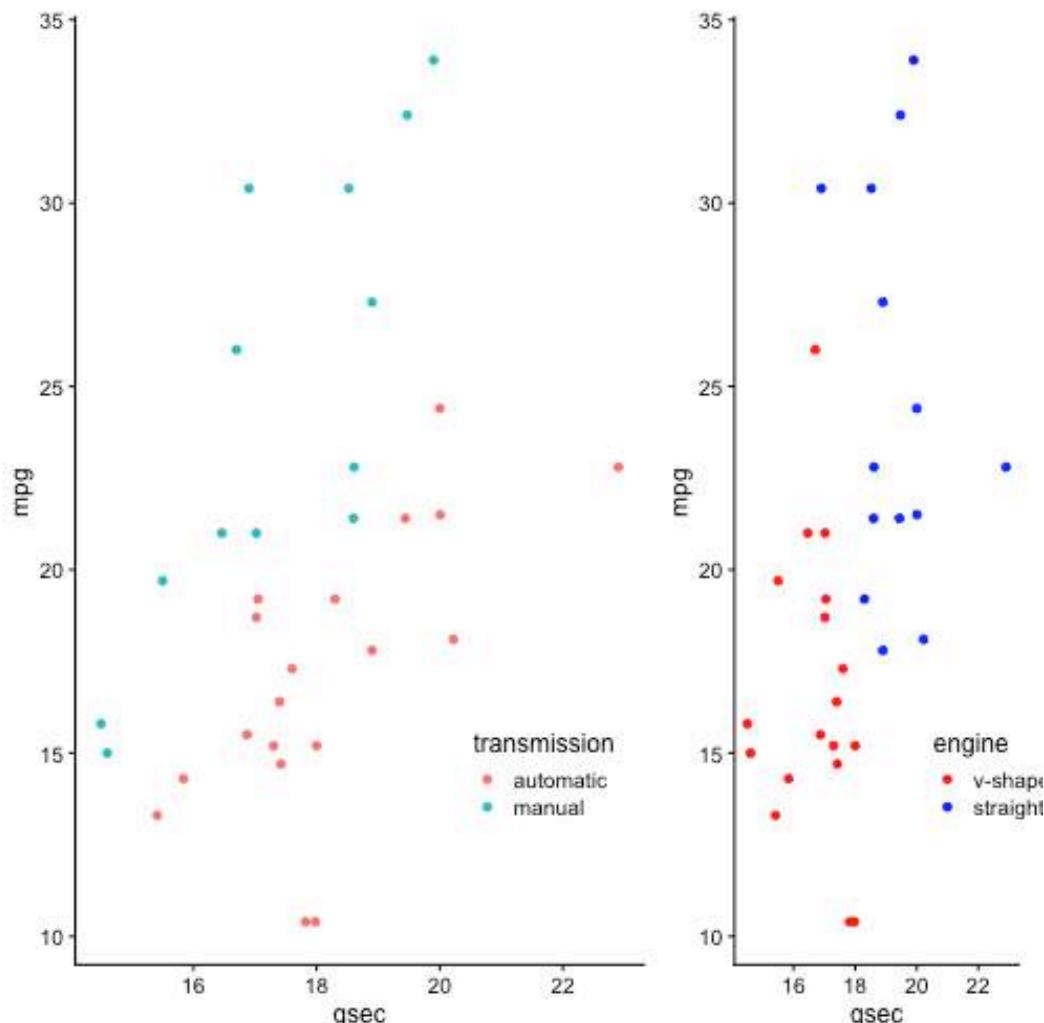


plot of chunk unnamed-chunk-77

You can modify the widths of the plots using the `widths` argument of the `plot_layout()` function. In the following example we will make the width of the plot on the left 2 times that of the plot on the right. Any numeric values will do, it is the ratio of the numbers that make the difference.

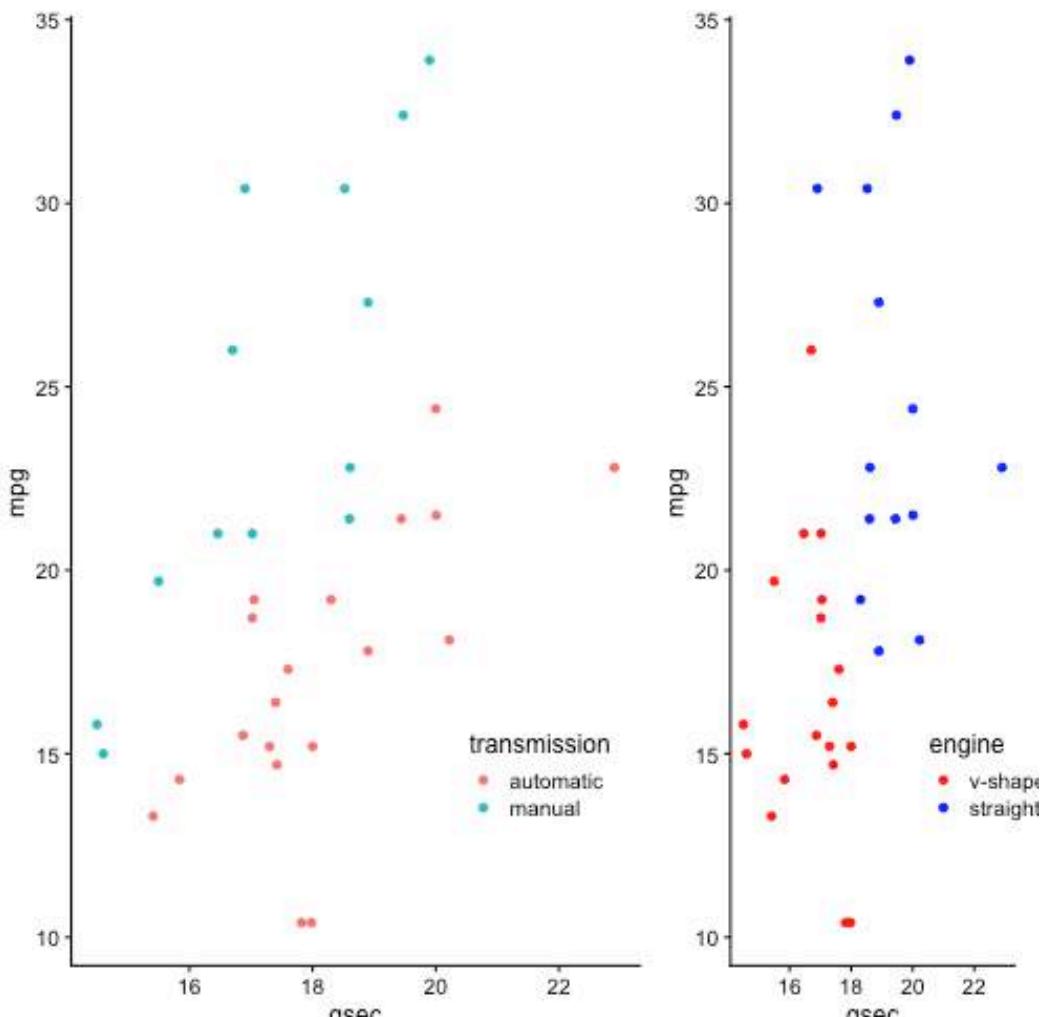
Thus, both `p1 + p2 + plot_layout(widths = c(2, 1))` and `p1 + p2 + plot_layout(widths = c(60, 30))` will result in the same relative size difference between `p1` and `p2`.

```
p1 + p2 + plot_layout(widths = c(2, 1))
```



plot of chunk unnamed-chunk-78

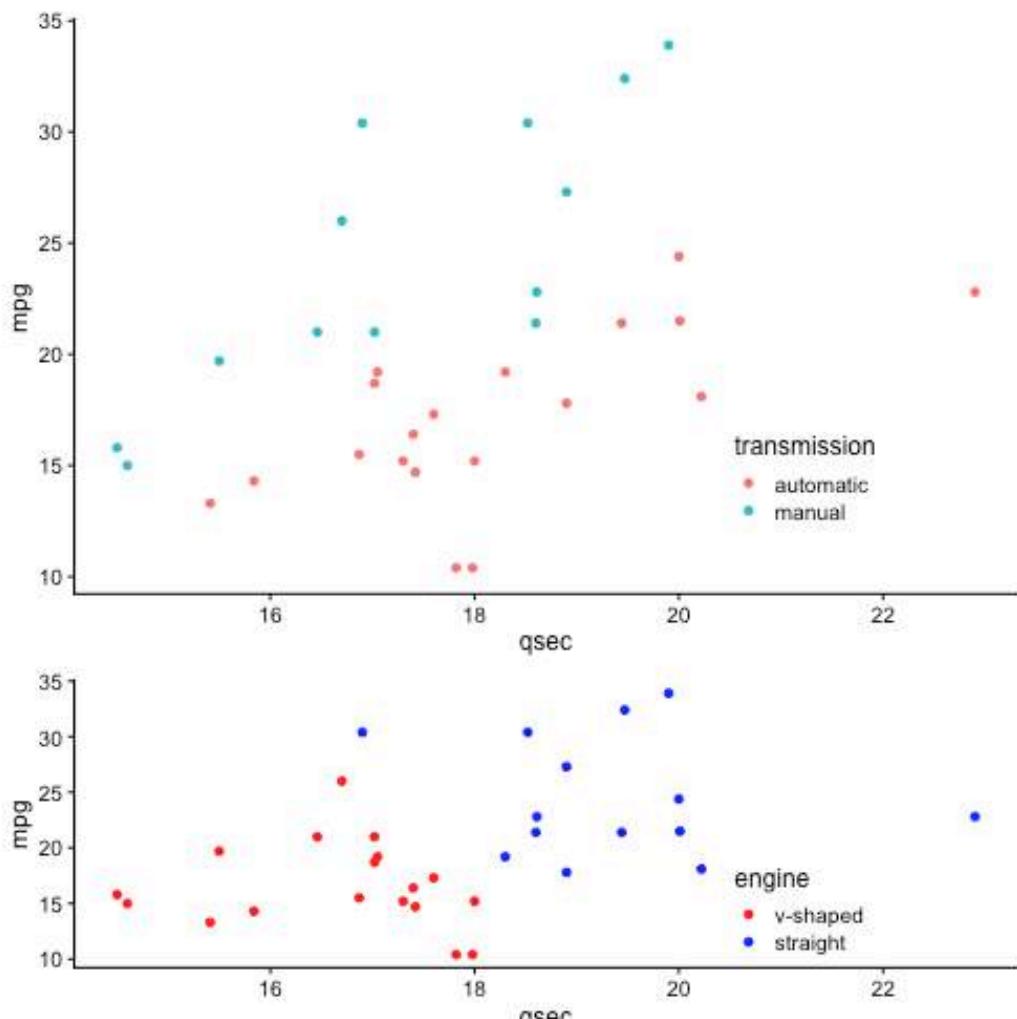
```
p1 + p2 + plot_layout(widths = c(60, 30))
```



plot of chunk unnamed-chunk-78

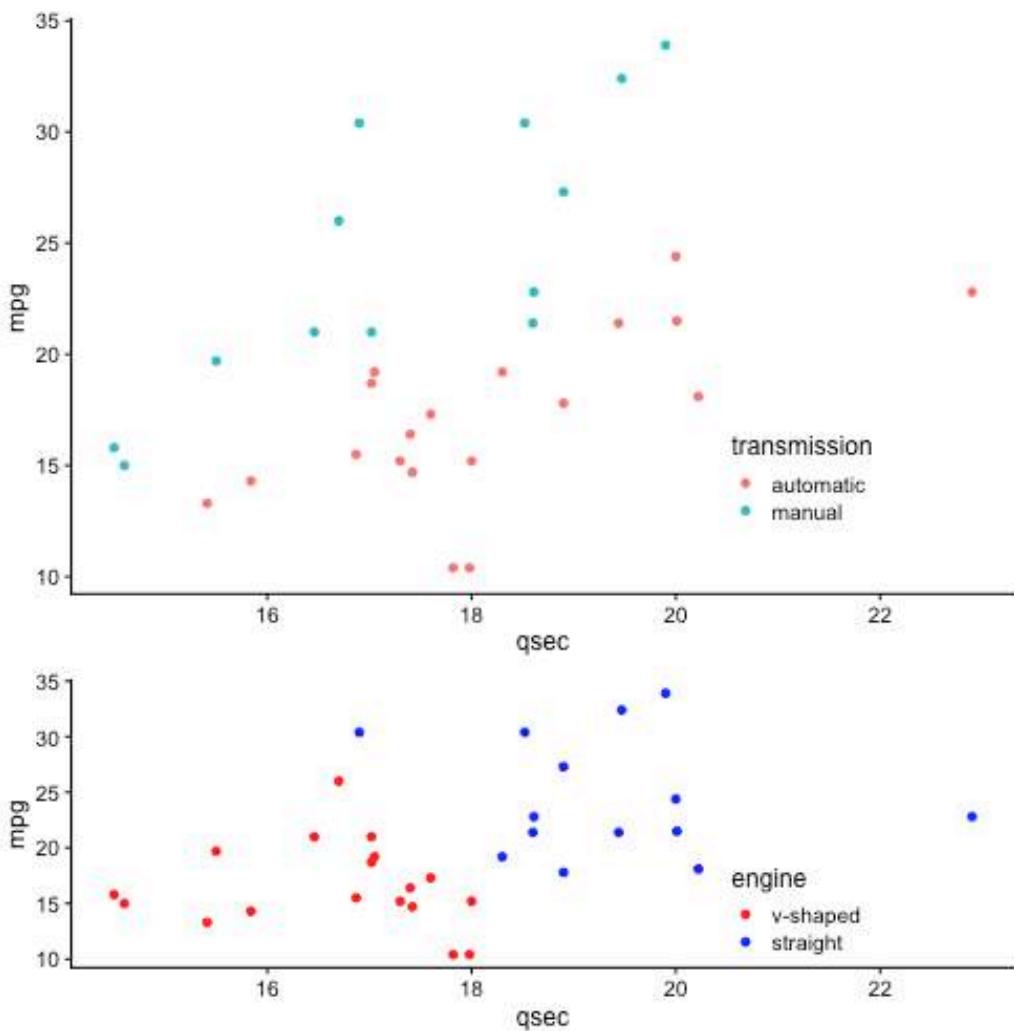
The relative heights of the plots can also be modified using a `heights` argument with the same function.

```
p1 + p2 + plot_layout(heights = c(2, 1))
```



plot of chunk unnamed-chunk-79

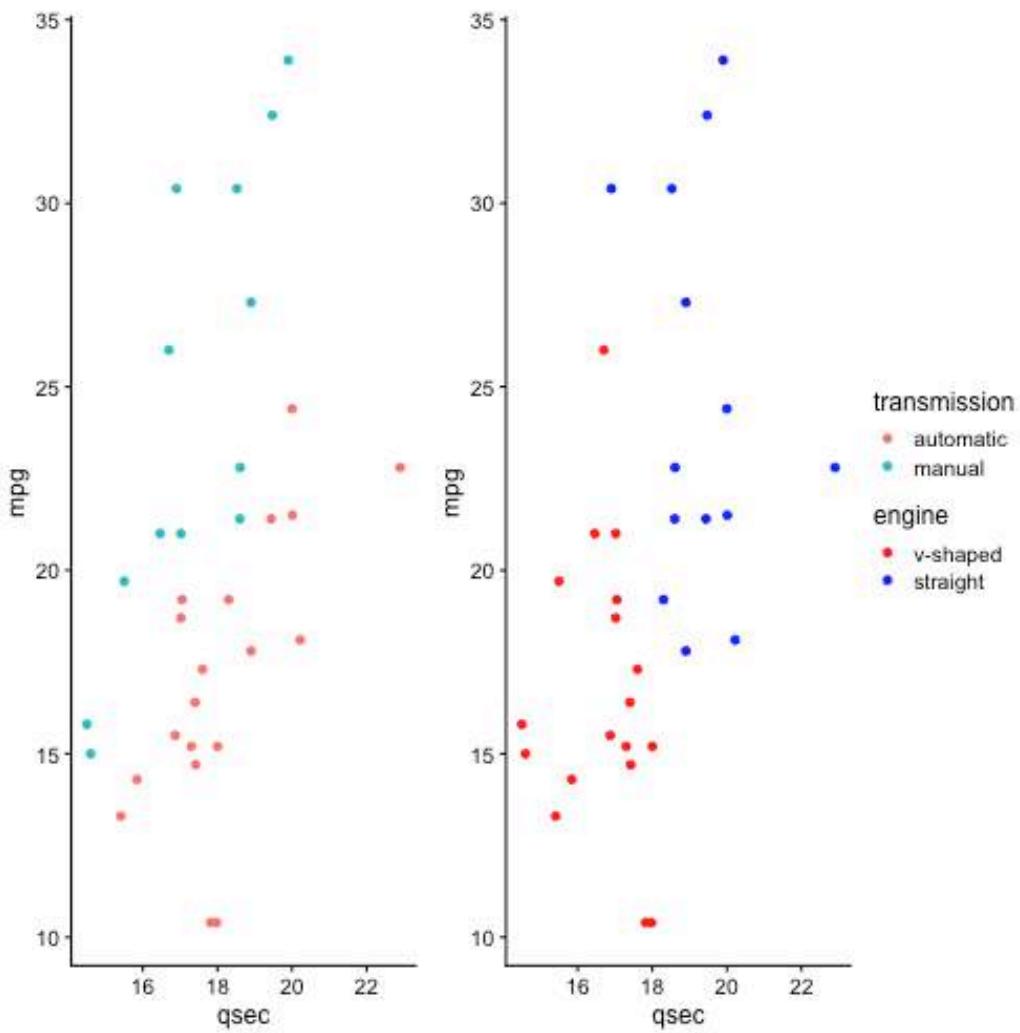
```
p1 + p2 + plot_layout(heights = c(60, 30))
```



plot of chunk unnamed-chunk-79

This package also allows for modification of legends. For example, legends can be gathered together to one side of the combined plots using the `guides = 'collect'` argument of the `plot_layout()` function.

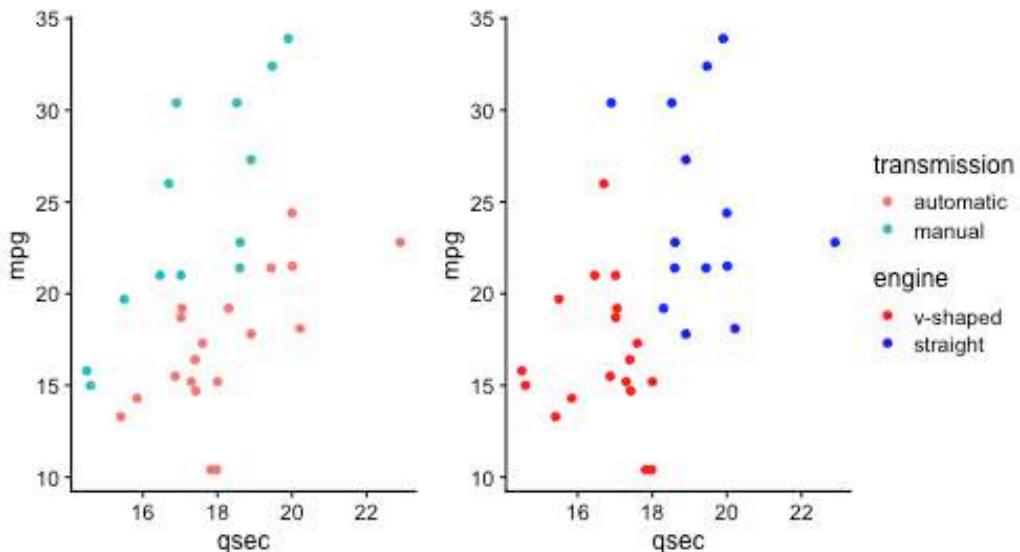
```
p1 + p2 + plot_layout(guides = "collect")
```



plot of chunk unnamed-chunk-80

You can also specify the number of columns or rows using this same function with the `ncol` or `nrow` as you would with `facet_wrap()` of the `ggplot2` package, where plots are added to complete a row before they will be added to a new row. For example, the following will result in an empty 2nd row below the plots.

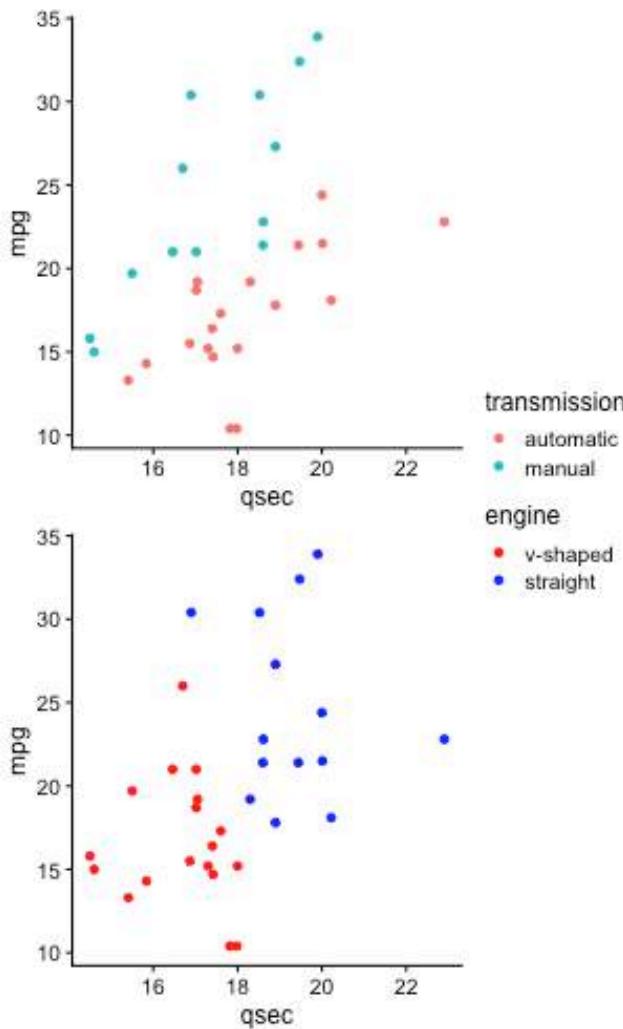
```
p1 + p2 + plot_layout(nrow = 2, ncol = 2, guides = "collect")
```



plot of chunk unnamed-chunk-81

However, the `byrow = FALSE` argument can disrupt this behavior and result in an empty 2nd column:

```
p1 + p2 + plot_layout(nrow = 2, ncol = 2, byrow = FALSE, guides = "collect")
```

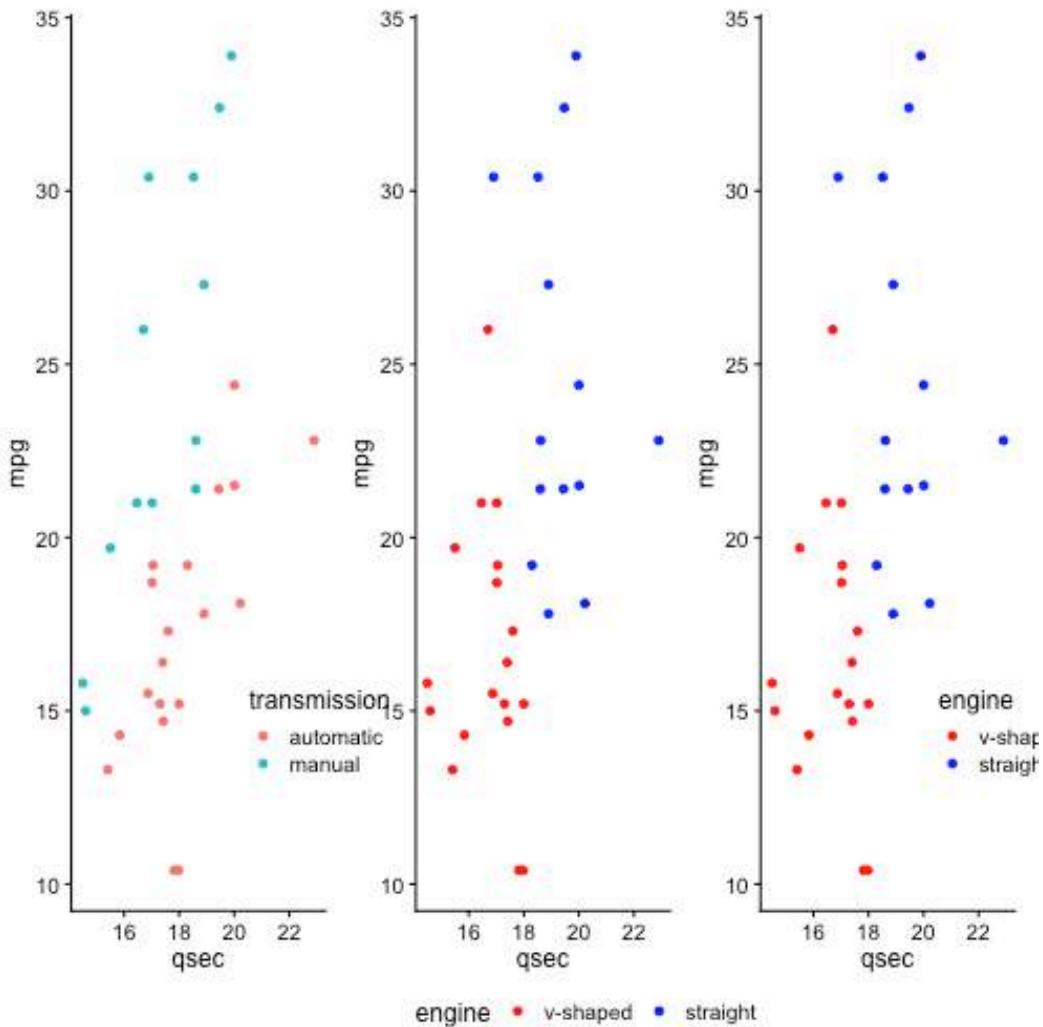


plot of chunk unnamed-chunk-82

In this case the **columns will be preferentially completed** before placing a plot in a new column.

We can also use the package to change the theme specifications of specific plots. By default the plot just prior to the `theme()` function will be the only plot changed.

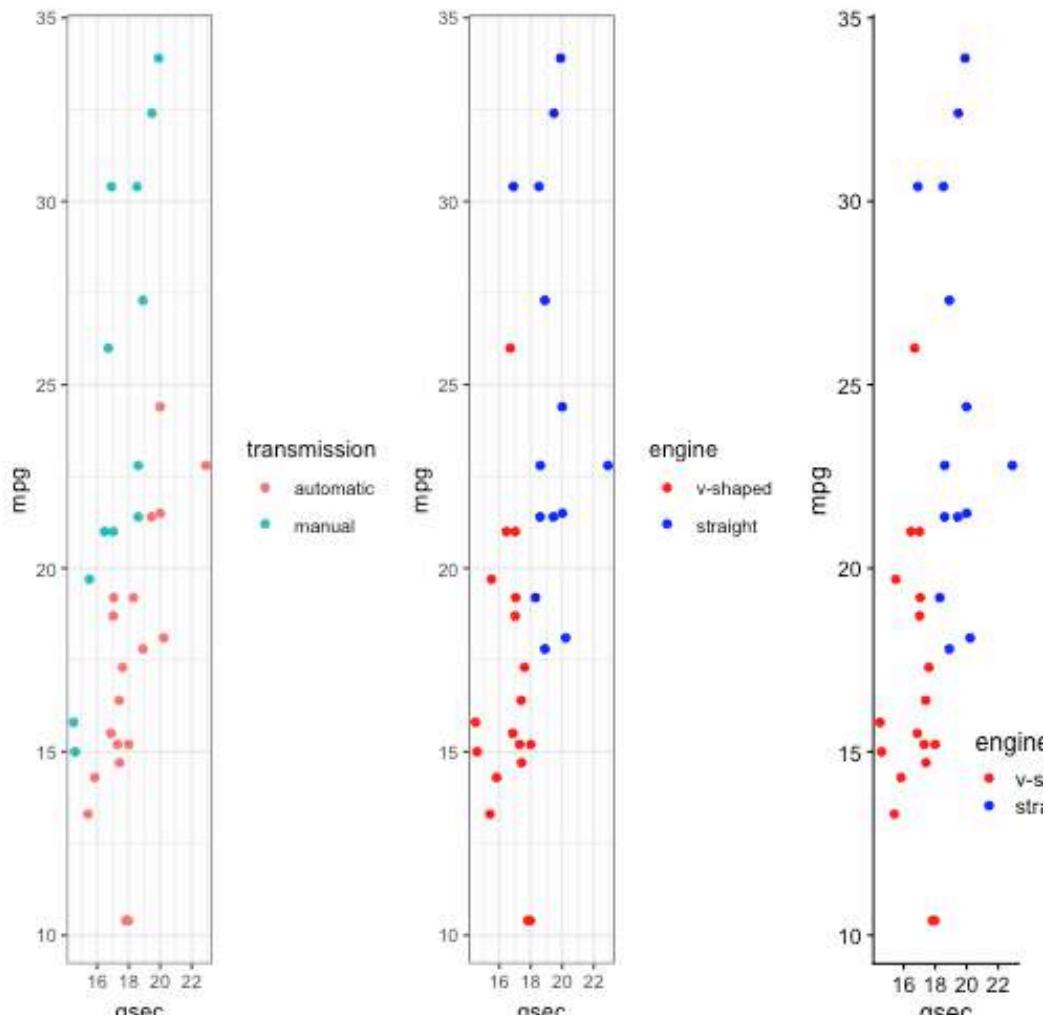
```
p1 + p2 + theme(legend.position = "bottom") + p2
```



plot of chunk unnamed-chunk-83

Using the `*`, themes can be added to all plots that are nested together.

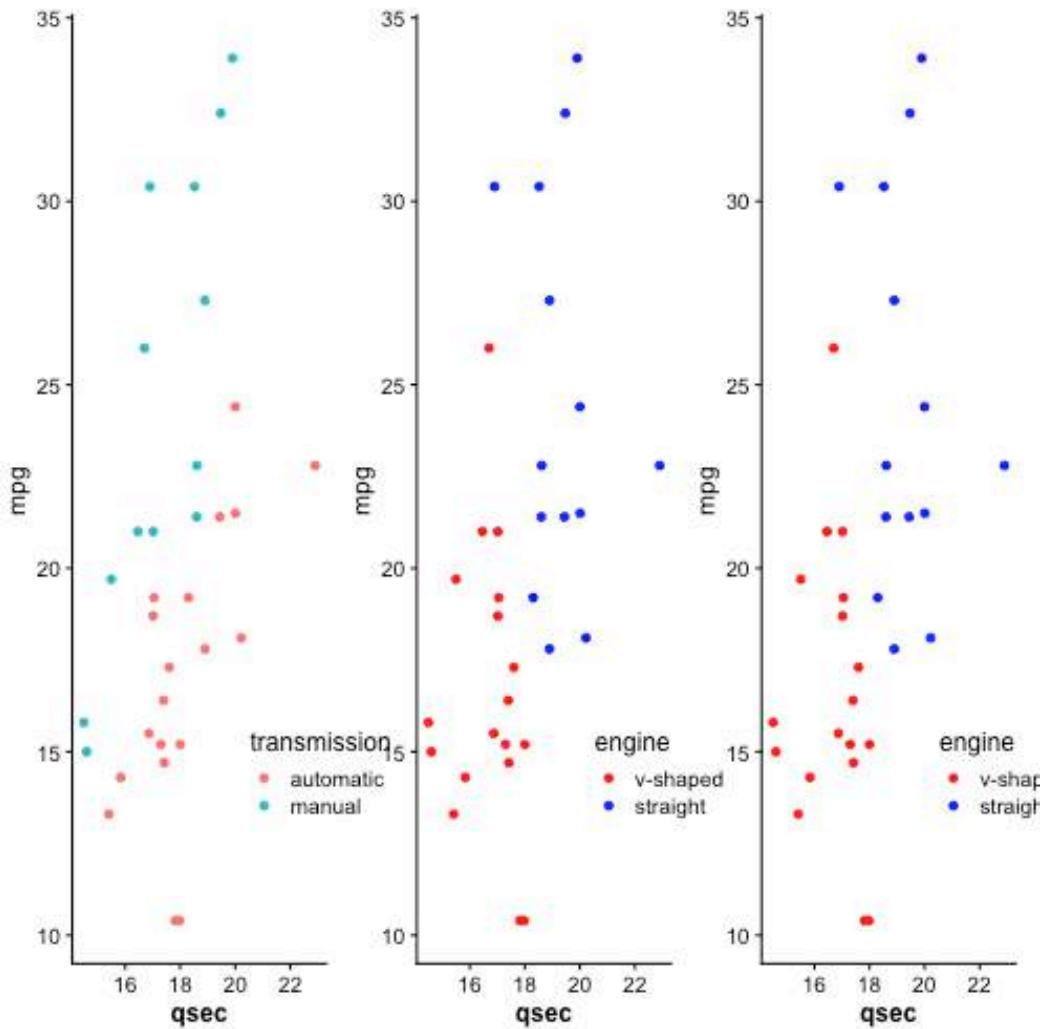
```
(p1 + p2) *theme_bw() + p2
```



plot of chunk unnamed-chunk-84

The & adds themes to all plots.

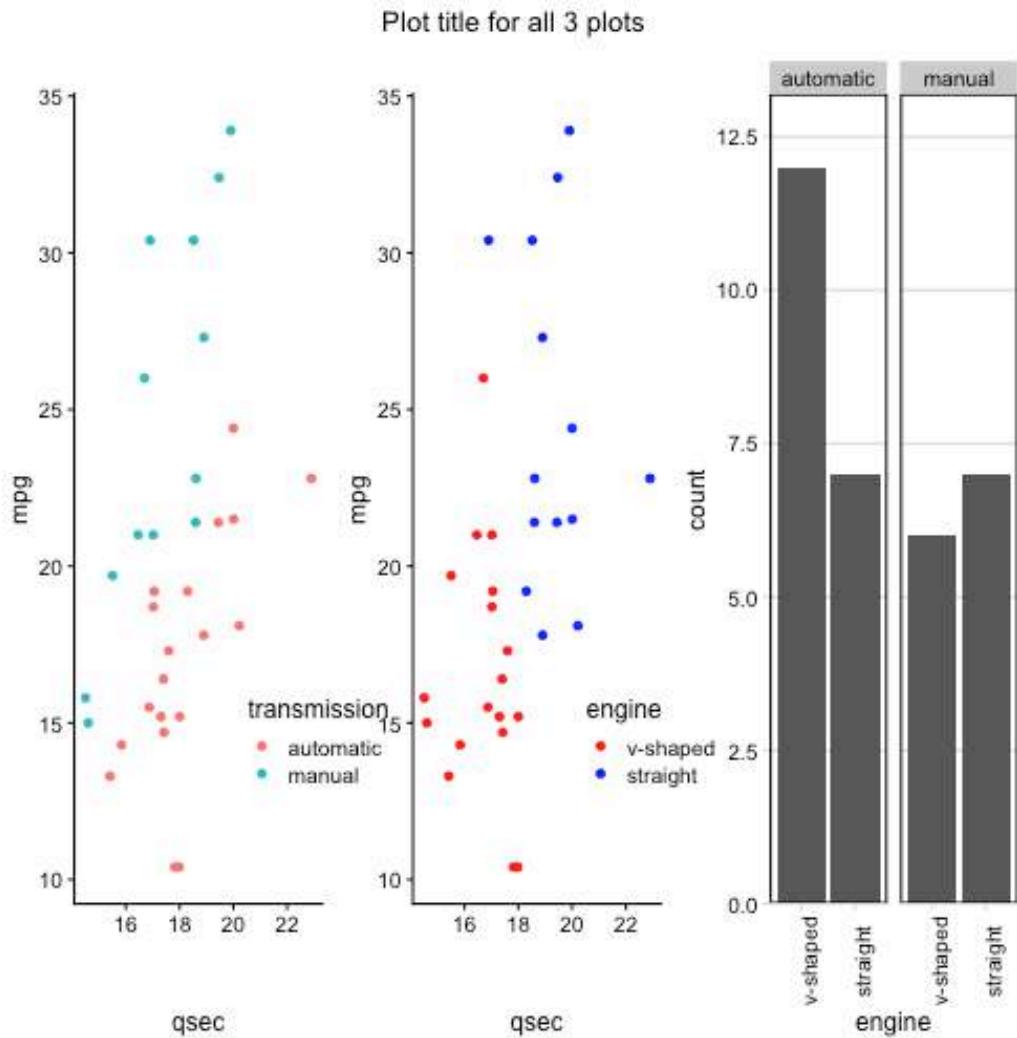
```
(p1 + p2) + p2 & theme(axis.title.x = element_text(face = "bold"))
```



plot of chunk unnamed-chunk-85

Annotations for all the plots combined can also be added using the `plot_annotation()` function, which can also take `theme()` function specifications with the `theme` argument.

```
(p1 + p2) + p3 + theme(axis.text.x = element_text(angle = 90)) +
  plot_annotation(title = "Plot title for all 3 plots",
    theme = theme(plot.title = element_text(hjust = 0.5)))
```



plot of chunk unnamed-chunk-86

See [here](#) for more information about the `patchwork` package.

gganimate

The final `ggplot2` extension we'll discuss here is `gganimate`. This extends the grammar of graphics implemented in the `ggplot2` package to work with animated plots. To get started, you'll need to install and load the package:

```
library(gganimate)
```

The `gganimate` package adds functionality by providing a number of these grammar classes. Across the animation being generated, the following classes are made available, with each classes' corresponding functionality:

- `transition_*`() | specifies how data should transition
- `enter_*/exit_*`() | specifies how data should appear and disappear
- `ease_aes`() | specifies how different aesthetics should be eased during transitions
- `view_*`() | specifies how positional scales should change
- `shadow_*`() | specifies how points from a previous frame should be displayed in a current frame

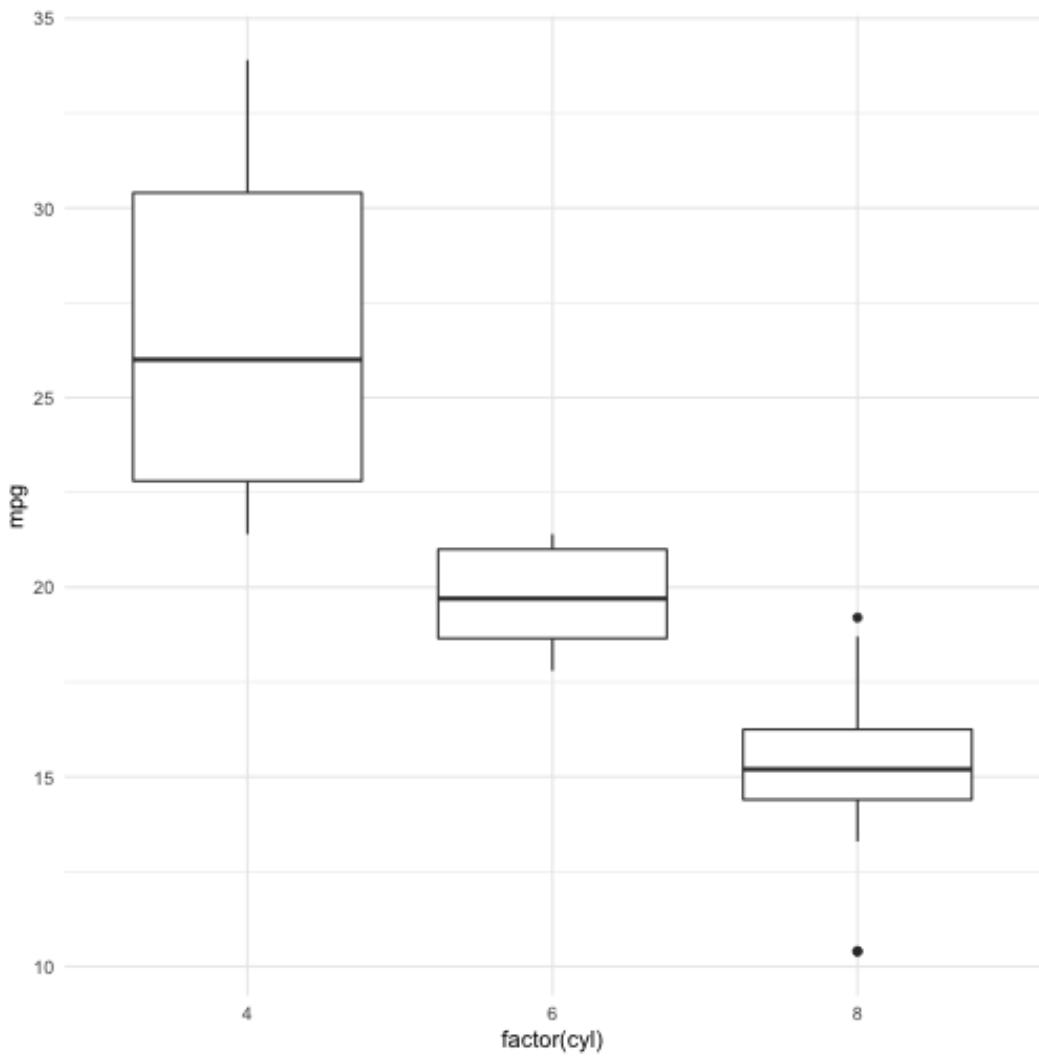
We'll walk through these grammar classes using the `mtcars` dataset.

Example: `mtcars`

As noted, `gganimate` builds on top of the functionality within `ggplot2`, so the code should look pretty familiar by now.

First, using the `mtcars` dataset we generate a static boxplot looking at the relationship between the number of cylinders a car has (`cyl`) and its gas mileage (`mpg`).

```
# generate static boxplot
ggplot(mtcars) +
  geom_boxplot(aes(factor(cyl), mpg))
```

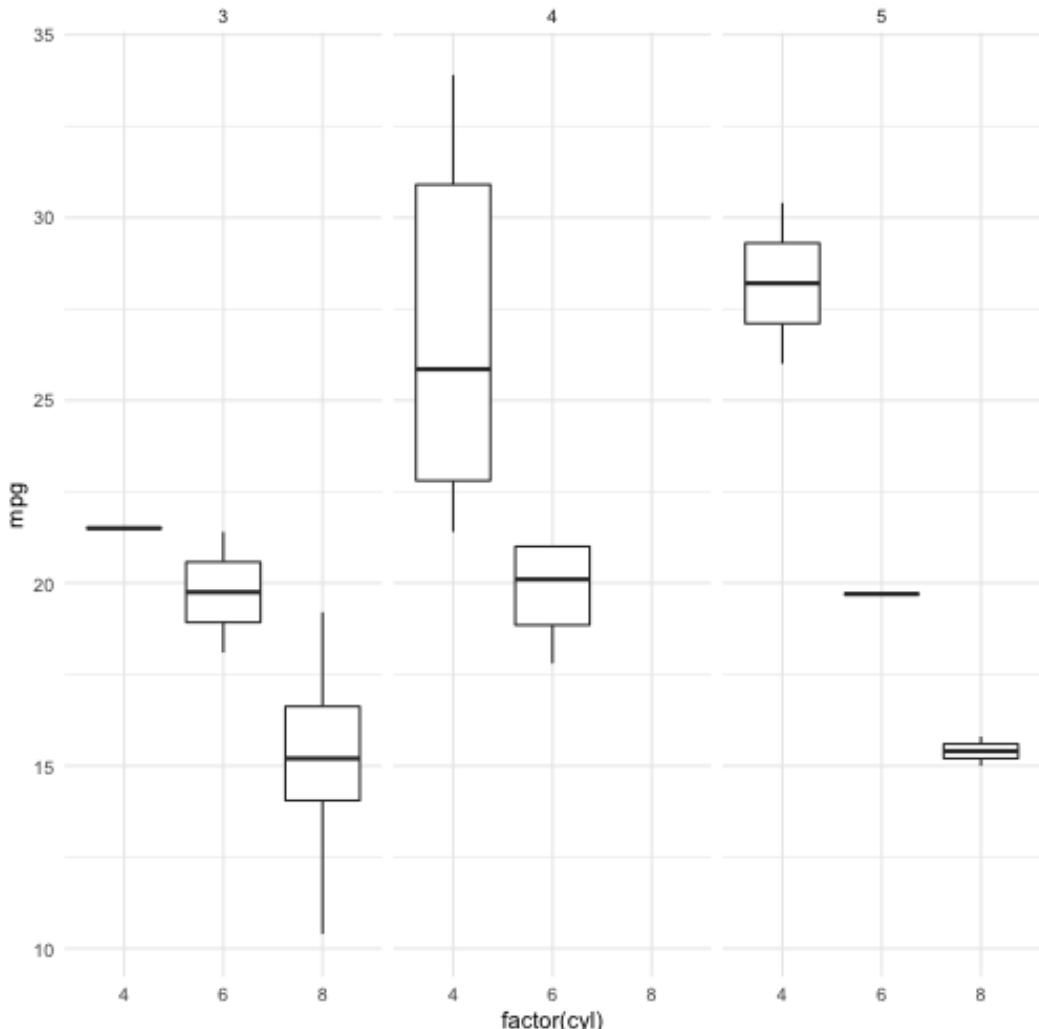


plot of chunk unnamed-chunk-88

But what if we wanted to understand the relationship between those two variables and the number of gears a car has (gear)?

One option would be to use facetting:

```
# facet by gear
ggplot(mtcars) +
  geom_boxplot(aes(factor(cyl), mpg)) +
  facet_wrap(~gear)
```



plot of chunk unnamed-chunk-89

Alternatively, we could animate the plot, using `gganimate` so that on a single plot we rotate between each of these three plots.

NOTE: For this book, we only show the code for the animations, not the output. In order to

see the animations live, you can see the [web version of this book](#).

Transitions

We'll transition in this example between frames by gear:

```
# animate it!
mtcars %>%
  mutate(gear = factor(gear),
         cyl = factor(cyl)) %>%
  ggplot() +
  geom_boxplot(aes(cyl, mpg)) +
  transition_manual(gear)
```

Note that here, `transition_manual()` is a new grammar class that we add right on top of our `ggplot2` plot code! Within this grammar class, we specify the variable upon which we'd like the transition in the animation to occur.

This means that the number of frames in the animation will be the number of levels in the variable by which you're transitioning. Here, there will be 3 frames, as there are three different levels of `gear` in the `mtcars` dataset.

`transition_manual()` is not the only option for transition. In fact, there are a number of different `transition_*` options, depending upon your animation needs. One of the most commonly used is `transition_states()`. The animation will plot the same information as the example above; however, it allows for finer tune control of the animation

```
# animate it!
anim <- ggplot(mtcars) +
  geom_boxplot(aes(factor(cyl), mpg)) +
  transition_states(gear,
                    transition_length = 2,
                    state_length = 1)

anim
```

Note here that we've stored the output from `transition_states` in the object `anim`. We'll build on this object below.

Labeling

Currently, it's hard to know which frame corresponds to which gear. To make this easier on the viewer, let's label our animation:

```
# animate it!
anim <- anim +
  labs(title = 'Now showing gear: {closest_state}')
anim
```

Note that we're referring to the appropriate gear within the animation frame by specifying `{closest_state}`.

Easing

Easing determines how the change from one value to another should occur. The default is linear, but there are other options that will control the appearance of progression in your animation. for example, 'cubic-in-out' allows for the start and end of the animation to have a smoother look.

```
anim <- anim +
  # Slow start and end for a smoother look
  ease_aes('cubic-in-out')
anim
```

There are a number of easing functions, all of which are listed within the documentation, which can be viewed, as always, using the `?help` function: `?ease_aes`.

Enter & Exit

Building on top of easing, we can also control how the data appears (enters) and disappears (exits), using `enter_*` and `exit_*`:

```
anim <- anim +  
  # fade to enter  
  enter_fade() +  
  # shrink on exit  
  exit_shrink()  
  
anim
```

The changes are subtle but you'll notice that on transition the data fades in to appear and shrinks upon exit.

Position Scales

To demonstrate how changing positional scales can be adjusted, let's take a look at a scatterplot. Here, we're plotting the relationship between 1/4 mile speed and miles per gallon and we'll be transitioning between gear.

The static plot would be as follows:

```
ggplot(mtcars) +  
  geom_point(aes(qsec, mpg))
```

plot of chunk unnamed-chunk-95

However, upon adding animation, we see how the data changes by gear.

```
# animate it!
scat <- ggplot(mtcars) +
  geom_point(aes(qsec, mpg)) +
  transition_states(gear,
    transition_length = 2,
    state_length = 1) +
  labs(title = 'Now showing gear: {closest_state}')
```

scat

However, the x- and y-axes remain constant throughout the animation.

If you wanted to allow these positional scales to adjust between each gear level, you could use `view_step`:

```
# allow positional scaling for each level in gear
scat +
  view_step(pause_length = 2,
            step_length = 1,
            nsteps = 3,
            include = FALSE)
```

Example: gapminder

One of the most famous examples of an animated plot uses the gapminder dataset. This dataset includes economic data for countries all over the world. The animated plot here demonstrates the relationship between GDP (per capita) and life expectancy over time, by continent. Note that the year whose data is being displayed is shown at the top left portion of the plot.

```
# install.packages("gapminder")
library(gapminder)

gap <- ggplot(gapminder, aes(gdpPercap, lifeExp, size = pop, colour = country))\
+
  geom_point(alpha = 0.7, show.legend = FALSE) +
  scale_colour_manual(values = country_colors) +
  scale_size(range = c(2, 12)) +
  scale_x_log10() +
  facet_wrap(~continent, ncol = 5) +
  theme_half_open(12) +
  panel_border() +
  # animate it!
  labs(title = 'Year: {frame_time}', x = 'GDP per capita', y = 'life expectancy') +
  transition_time(year)

gap
```

Note that in this example, we're now using `transition_time()` rather than `transition_states()`. This is a variant of `transition_states()` that is particularly useful when the states represent points in time, such as the years we're animating through in the plot above. The transition length is set to correspond to the time difference between the points in the data.

Shadow

However, what if we didn't want the data to completely disappear from one frame to the next and instead wanted to see the pattern emerge over time? We didn't demonstrate this using the `mtcars` dataset because each observation in that dataset is a different car. However, here, with the `gapminder` dataset, where we're looking at a trend over time, it makes more sense to include a trail.

To do this, we would use `shadow_*`. Here, we'll use `shadow_trail()` to see a trail of the data from one frame to the next:

```
gap +
  shadow_trail(distance = 0.05,
                alpha = 0.3)
```

Here, distance specifies the temporal distance between the frames to show and alpha specifies the transparency of the trail, so that the current frame's data is always discernible.

Case Studies

At this point, we've done a lot of work with our case studies. We've introduced the case studies, read them into R, and have wrangled the data into a usable format. Now, we get to peak at the data using visualizations to better understand each dataset's observations and variables!

Let's start by loading our wrangled tidy data that we previously saved:

```
load(here::here("data", "tidy_data", "case_study_1_tidy.rda"))

load(here::here("data", "tidy_data", "case_study_2_tidy.rda"))
```

Case Study #1: Health Expenditures

We've now got the data in order so that we can start to explore the relationships between the variables contained in the health care dataset (`hc`) to answer our questions of interest:

1. Is there a relationship between health care coverage and health care spending in the United States?
2. How does the spending distribution change across geographic regions in the United States?
3. Does the relationship between health care coverage and health care spending in the United States change from 2013 to 2014?

```
# see health care data
hc
# A tibble: 612 × 10
  Location year type      tot_coverage abb   region tot_spending tot_pop
  <chr>     <int> <chr>        <int> <chr> <fct>    <dbl>    <int>
1 Alabama   2013 Employer     2126500 AL    South    33788 4763900
2 Alabama   2013 Non-Group    174200  AL    South    33788 4763900
3 Alabama   2013 Medicaid     869700  AL    South    33788 4763900
4 Alabama   2013 Medicare     783000  AL    South    33788 4763900
5 Alabama   2013 Other Public 85600   AL    South    33788 4763900
6 Alabama   2013 Uninsured   724800  AL    South    33788 4763900
7 Alabama   2014 Employer     2202800 AL    South    35263 4768000
8 Alabama   2014 Non-Group    288900  AL    South    35263 4768000
9 Alabama   2014 Medicaid     891900  AL    South    35263 4768000
10 Alabama  2014 Medicare    718400  AL    South   35263 4768000
# ... with 602 more rows, and 2 more variables: prop_coverage <dbl>,
#   spending_capita <dbl>
```

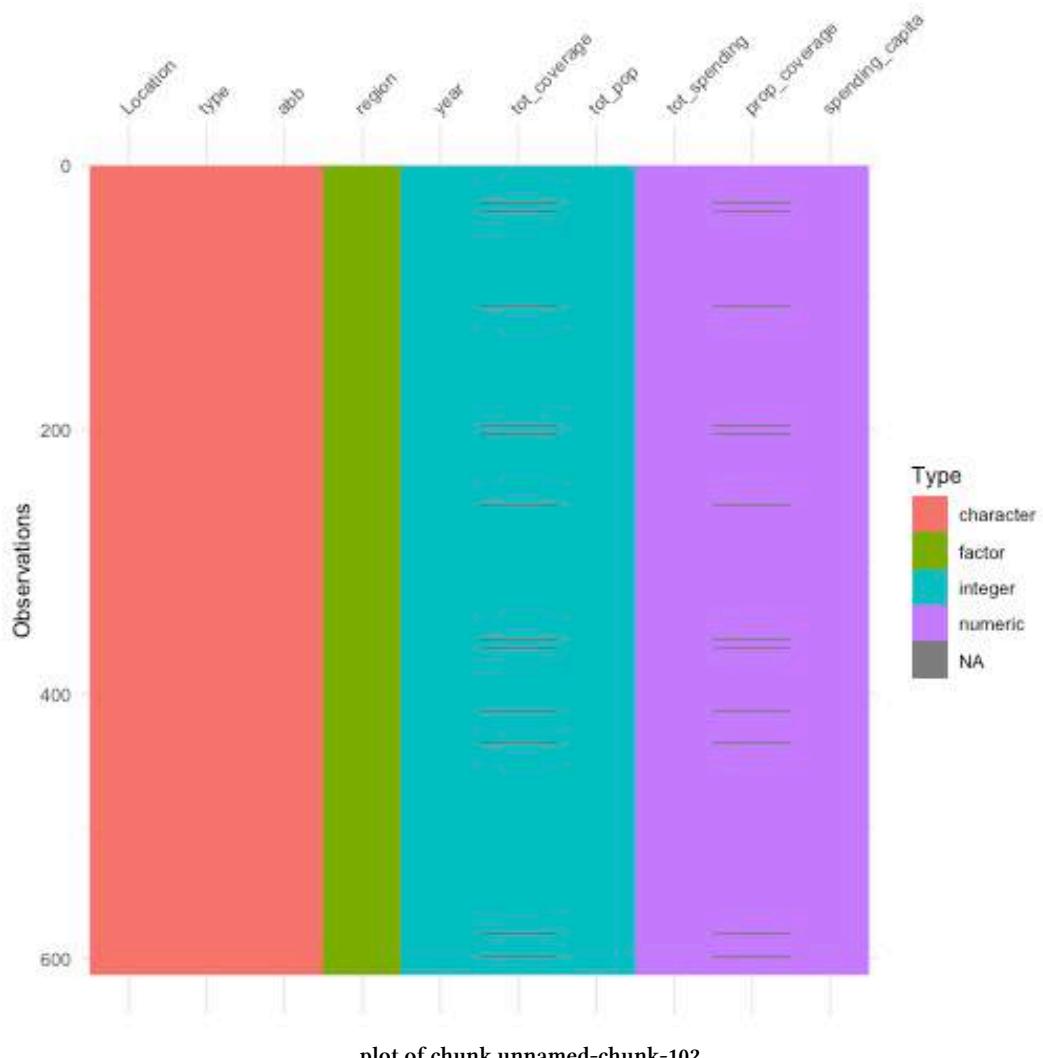
As a reminder, we have state level data, broken down by year and type of insurance. For each, we have the total number of individuals who have health care coverage (tot_coverage), the amount spent on coverage (tot_spending), the proportion of individuals covered (prop_coverage), and the amount spent per capita (spending_capita). Additionally, we have the state name (Location), the two letter state abbreviation (abb), and the region of the United States where the state is located (region). Let's get visualizing!

Exploratory Data Analysis (EDA)

To first get a sense of what information we do and do not have, the visdat package can be very helpful. This package uses ggplot2 to visualize missingness in a dataframe. For example, vis_dat() takes the dataframe as an input and visualizes the observations on the left and the variables across the top. Solid colors indicate that a value is present. Each type of variable is represented by a different color. Grey lines indicate missing values.

```
# install.packages("visdat")
library(visdat)

vis_dat(hc)
```

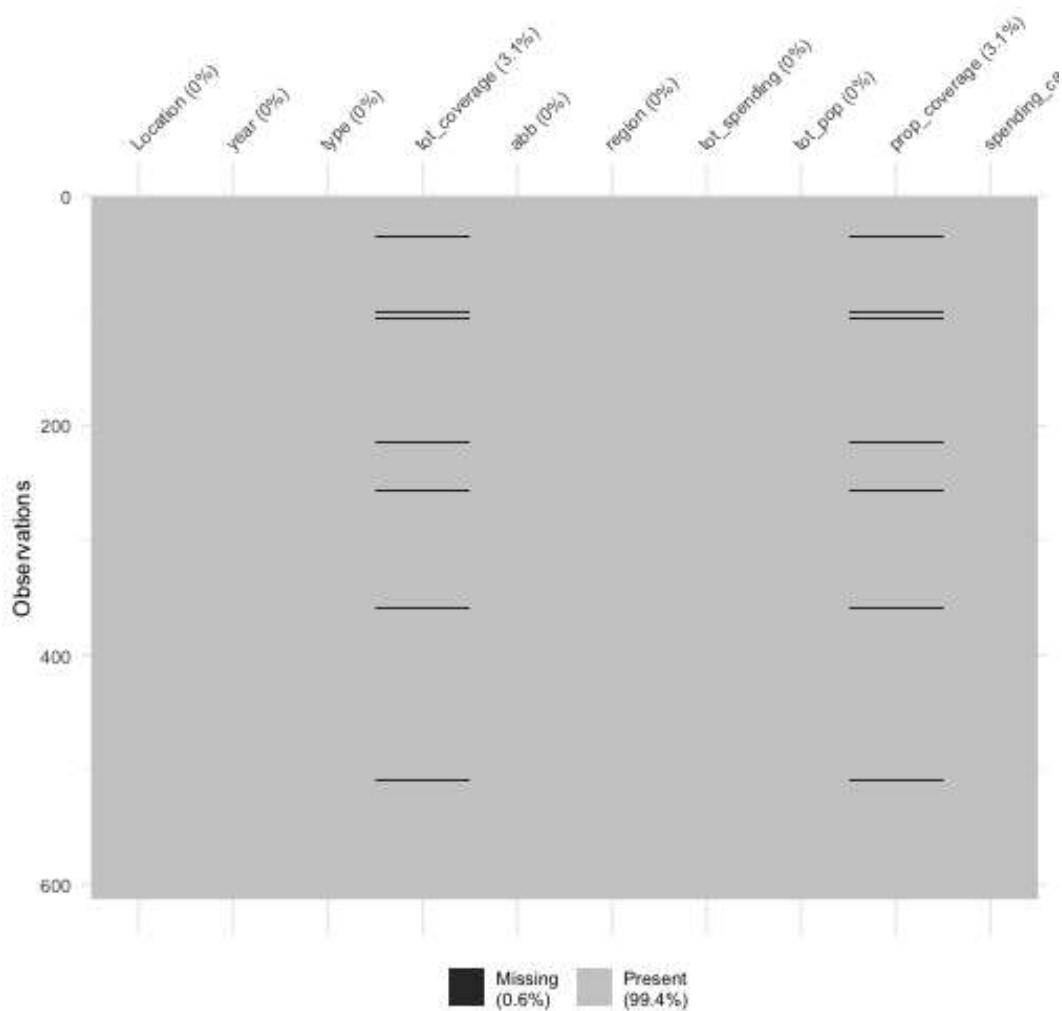


plot of chunk unnamed-chunk-102

We now have a sense that a few states, for some years are missing coverage data, which also affects the ability to calculate proportion covered.

To see these values highlighted more specifically, we can use the related `vis_miss()` function.

```
vis_miss(hc)
```



plot of chunk unnamed-chunk-103

Here we see that missing values only occur 0.6% of the time, with 3.1% of the observations missing entries for tot_coverage and prop_coverage. So, all in all, there is not a lot of missing data, but we still want to be sure we understand where missingness occurs before answering our questions.

Let's use `dplyr` to see *which* observations have missing data:

```

hc %>%
  filter(is.na(tot_coverage))
# A tibble: 19 × 10
  Location      year type tot_coverage abb   region tot_spending tot_\
pop
  <chr>        <int> <chr>     <int> <chr> <fct>    <dbl>    <i\
nt>
  1 Arizona      2013 Othe...     NA AZ    West     41481 6603\
100
  2 Arizona      2014 Othe...     NA AZ    West     43356 6657\
200
  3 District of Columbia 2013 Othe...     NA DC    South    7443  652\
100
  4 District of Columbia 2014 Othe...     NA DC    South    7871  656\
900
  5 Indiana       2014 Othe...     NA IN    North...  54741 6477\
500
  6 Kansas        2013 Othe...     NA KS    North...  21490 2817\
600
  7 Kansas        2014 Othe...     NA KS    North...  22183 2853\
000
  8 Kentucky      2014 Othe...     NA KY    South    35323 4315\
700
  9 Massachusetts 2013 Othe...     NA MA    North...  68899 6647\
700
 10 Massachusetts 2014 Othe...     NA MA    North...  71274 6658\
100
 11 New Hampshire 2014 Othe...     NA NH    North...  12742 1319\
700
 12 New Jersey    2013 Othe...     NA NJ    North...  75148 8807\
400
 13 North Dakota 2013 Othe...     NA ND    North...  6795  714\
500
 14 Oklahoma       2013 Othe...     NA OK    South    28097 3709\
400
 15 Rhode Island   2014 Othe...     NA RI    North...  10071 1048\
200
 16 Tennessee      2013 Othe...     NA TN    South    46149 6400\
200

```

```
17 Tennessee          2014 Other...      NA TN   South     48249 6502\  
000  
18 West Virginia    2013 Other...      NA WV   South     16622 1822\  
000  
19 Wisconsin         2014 Other...      NA WI   North...  50109 5747\  
200  
# ... with 2 more variables: prop_coverage <dbl>, spending_capita <dbl>
```

Ah, so we see that the “Other” type of coverage is missing in both 2013 and 2014 for a subset of states. We’ll be focusing on the non-“Other” types of health care coverage, so this shouldn’t pose a problem, but it is good to know!

Taking this one step further, let’s skim the entire dataset to get a sense of the information stored in each variable:

```
library(skimr)  
  
# get summary of the data  
skim(hc)
```

```
— Data Summary —————
  Values
Name          hc
Number of rows 612
Number of columns 10

Column type frequency:
 character      3
 factor         1
 numeric        6

Group variables   None

— Variable type: character —————
skim_variable n_missing complete_rate   min   max empty n_unique whitespace
1 Location       0           1     4    20    0     51      0
2 type           0           1     8    12    0     6      0
3 abb            0           1     2     2    0     51      0

— Variable type: factor —————
skim_variable n_missing complete_rate ordered n_unique top_counts
1 region         0           1 FALSE        4 Sou: 204, Wes: 156, Nor: 144, Nor: 108

— Variable type: numeric —————
skim_variable n_missing complete_rate      mean       sd      p0      p25      p50      p75
1 year           0           1    2014.     0.500    2013    2013    2014.    2014
2 tot_coverage   19          0.969 1059392. 1864111. 9900 135700 436800 1036800
3 tot_spending   0           1    49004. 54031. 4639 12742 33939 60364
4 tot_pop        0           1    6172245. 6983404. 572000 1610200 4357900 6866000
5 prop_coverage  19          0.969  0.171   0.164  0.00536  0.0577  0.126  0.180
6 spending_capita 0           1    8246.   1264.   5677.  7396.  8049.  8994.

p100 hist
1 2014
2 17747300
3 291989
4 38791300
5 0.609
6 11982.
```

skim output

At a glance, by looking at the `hist` column of the output for our numeric/integer variables, we see that there is a long right tail in `tot_coverage` and `tot_pop`.

We see that proportion coverage varies from 0.0054 (`p0`) to 0.61 (`p100`). So, we'll want to know which states are at each end of the spectrum here.

We also see that for `tot_spending`, the `mean` is 49004 and the median (`p50`) is slightly lower and that this variable also has a long right tail (`hist`).

We also know that these data come from two different years, so we can group by `year` and again summarize the data:

```
# group by year
hc %>%
  group_by(year) %>%
  skim()
```

```

— Data Summary —————
      Values
Name          Piped data
Number of rows 612
Number of columns 10

Column type frequency:
  character     3
  factor         1
  numeric        5

Group variables year

— Variable type: character —————
  skim_variable  year n_missing complete_rate   min   max empty n_unique whitespace
1 Location       2013     0             1     4    20    0     51      0
2 Location       2014     0             1     4    20    0     51      0
3 type           2013     0             1     8    12    0      6      0
4 type           2014     0             1     8    12    0      6      0
5 abb            2013     0             1     2     2    0     51      0
6 abb            2014     0             1     2     2    0     51      0

— Variable type: factor —————
  skim_variable  year n_missing complete_rate ordered n_unique top_counts
1 region         2013     0             1 FALSE      4 Sou: 102, Wes: 78, Nor: 72, Nor: 54
2 region         2014     0             1 FALSE      4 Sou: 102, Wes: 78, Nor: 72, Nor: 54

— Variable type: numeric —————
  skim_variable  year n_missing complete_rate      mean        sd      p0      p25      p50
1 tot_coverage   2013      9      0.971 1052928. 1863159. 9900 132300 426300
2 tot_coverage   2014     10      0.967 1065878. 1868198. 9900 137700 450150
3 tot_spending   2013      0      1     47757. 52637. 4639 12392 33468
4 tot_spending   2014      0      1     50251. 55449. 4856 12742 35299
5 tot_pop        2013      0      1   6145122. 6945678. 582200 1600600 4400100
6 tot_pop        2014      0      1   6199369. 7032201. 572000 1610200 4315700
7 prop_coverage  2013      9      0.971     0.171    0.165  0.00690  0.0499  0.128
8 prop_coverage  2014     10      0.967     0.172    0.163  0.00536  0.0619  0.122
9 spending_capita 2013      0      1     8065.    1228.  5677.  7211.  7843.
10 spending_capita 2014      0      1     8428.    1275.  6007.  7545.  8243.

  p75      p100 hist
1 1034500  17747300 █■
2 1056725  17703700 █■
3 60364   278168 █■
4 62847   291989 █■
5 6866000 38176400 █■
6 7085000 38701300 █■
7 0.178   0.601  █■
8 0.182   0.609  █■
9 8883.   11414. █■
10 9378.  11982. █■

```

skim output

At a glance, there doesn't appear to be a huge difference in the variables from one year to the next, but we'll explore this again in a bit.

With at least some understanding of the data in our dataset, let's start generating exploratory plots that will help us answer each of our questions of interest.

Q1: Relationship between coverage and spending?

To answer the question:

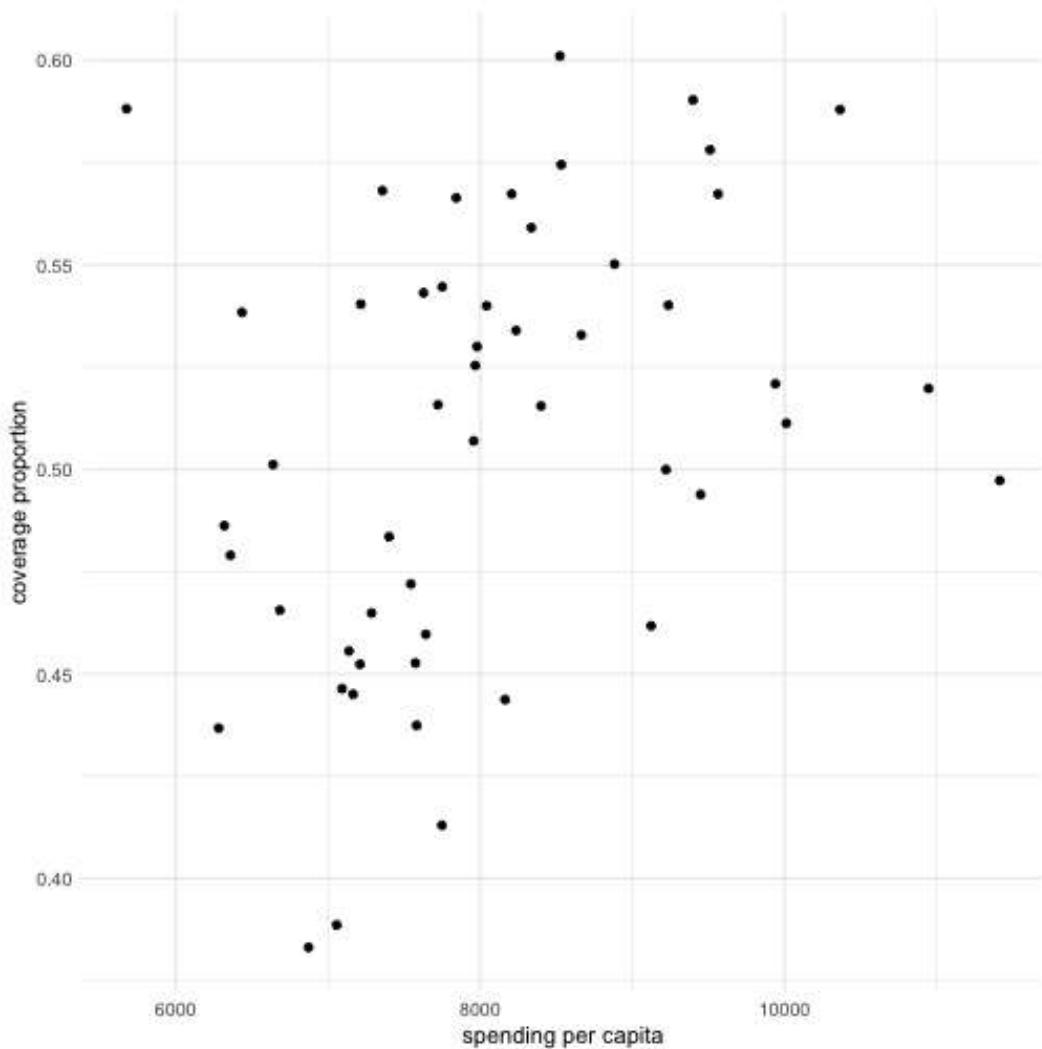
Is there a relationship between health care coverage and health care spending in the United States?

We'll have to visualize coverage and spending data across the United States.

And, to understand the relationship between two continuous variables - coverage and spending - a scatterplot will be the most effective visualization.

We'll first look at a scatterplot:

```
hc %>%
  filter(type == "Employer",
         year == "2013") %>%
  ggplot(aes(x = spending_capita,
             y = prop_coverage)) +
  geom_point() +
  labs(x = "spending per capita",
       y = "coverage proportion")
```

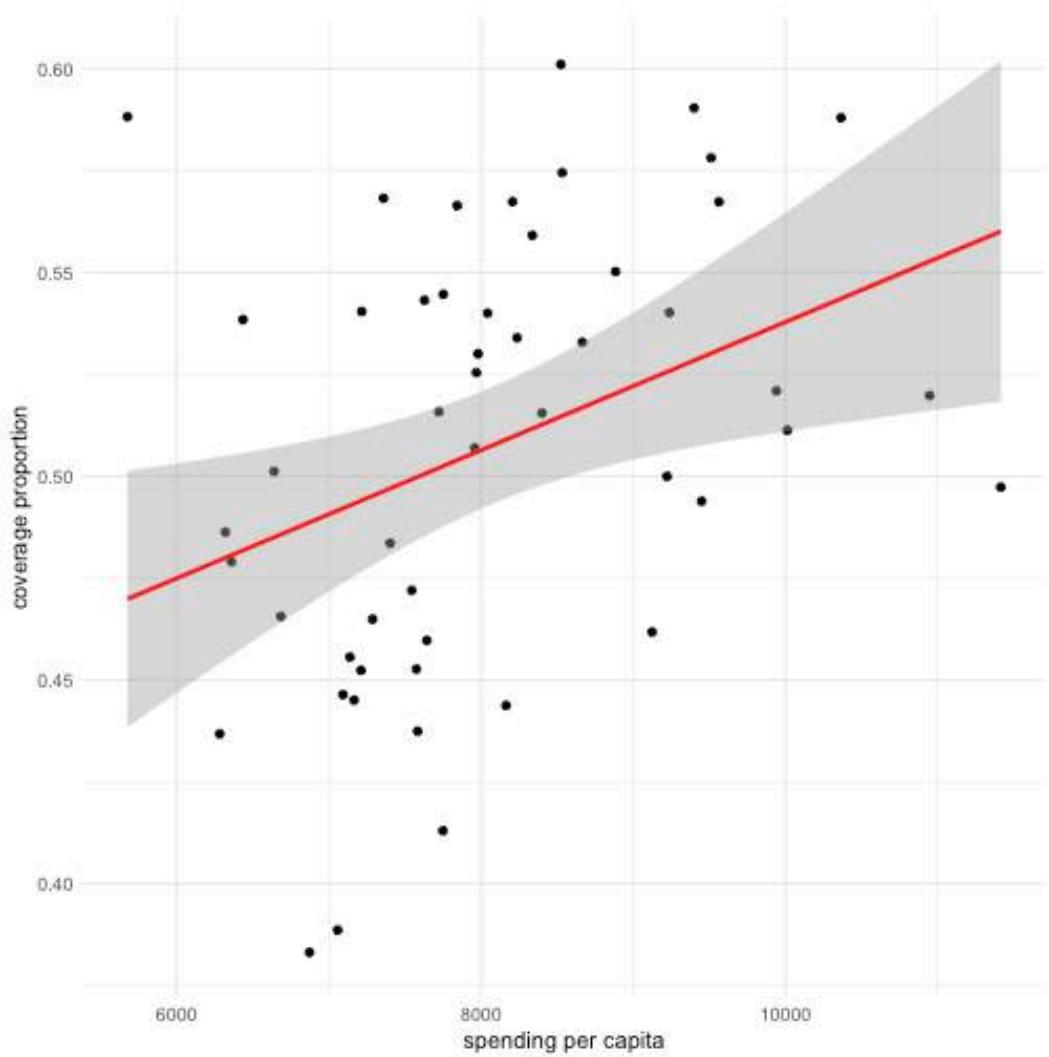


plot of chunk unnamed-chunk-107

We see that there appears to be some relationship, with those states that spend more per capita also having higher proportions of their population having health care coverage.

We can continue to improve this plot to better understand the underlying data. For example, we can add a best-fit line using `geom_smooth()` to visualize the magnitude of the linear relationship:

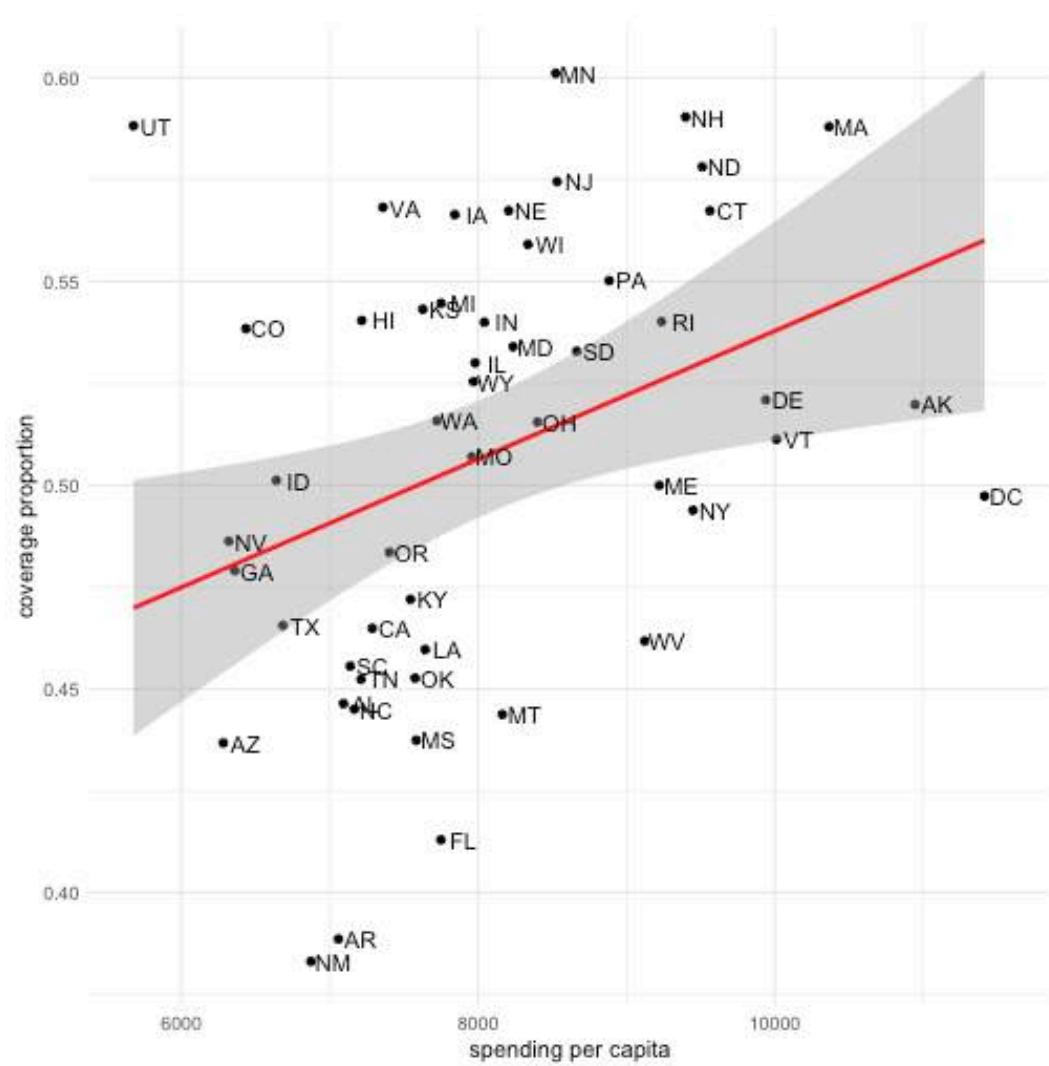
```
# generate scatterplot with fit line
hc %>%
  filter(type == "Employer",
         year == "2013") %>%
  ggplot(aes(x = spending_capita,
             y = prop_coverage)) +
  geom_point() +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red")
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-108

Beyond that, we likely want to know which point represents which state, so we can add state labels:

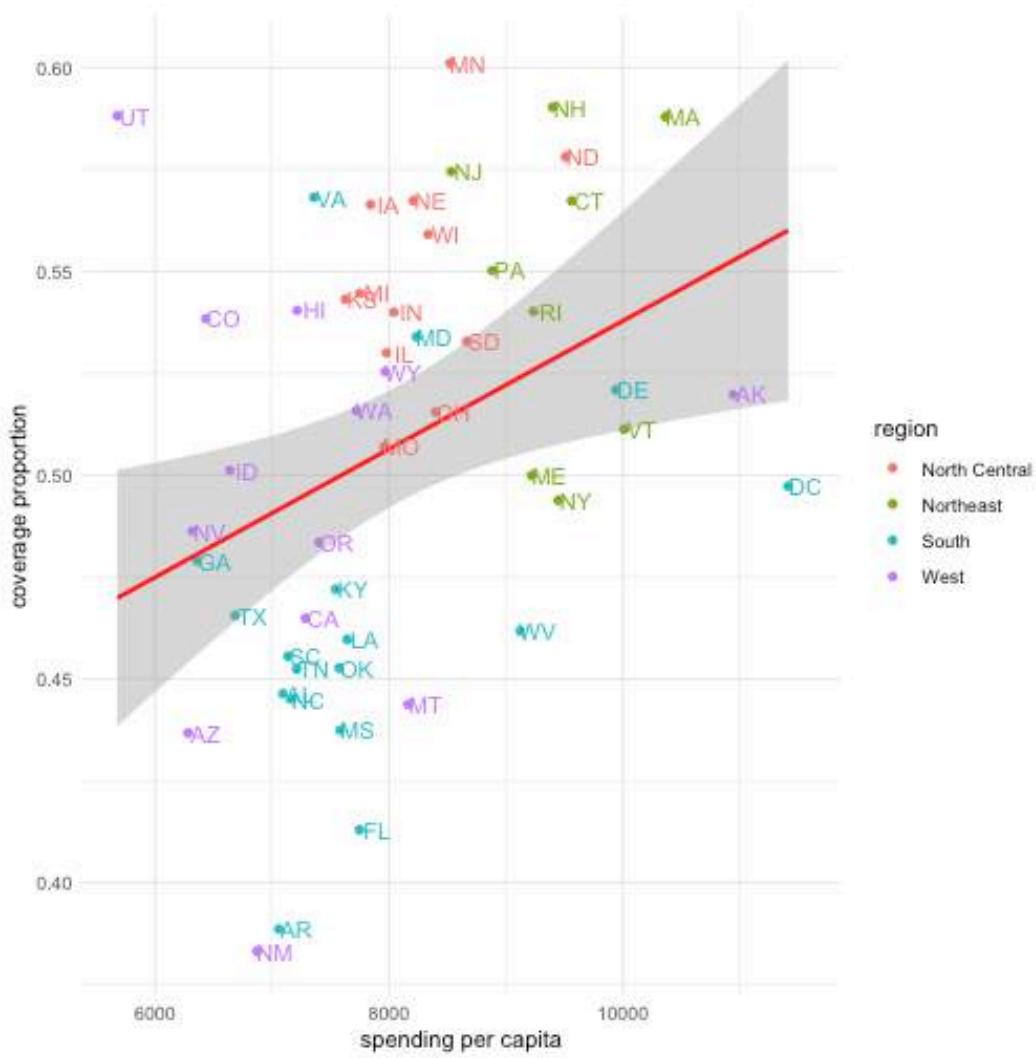
```
# add state abbreviation labels
hc %>%
  filter(type == "Employer",
         year == "2013") %>%
  ggplot(aes(x = spending_capita,
              y = prop_coverage)) +
  geom_point() +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150)
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-109

From there, it'd likely be helpful to have information from each region:

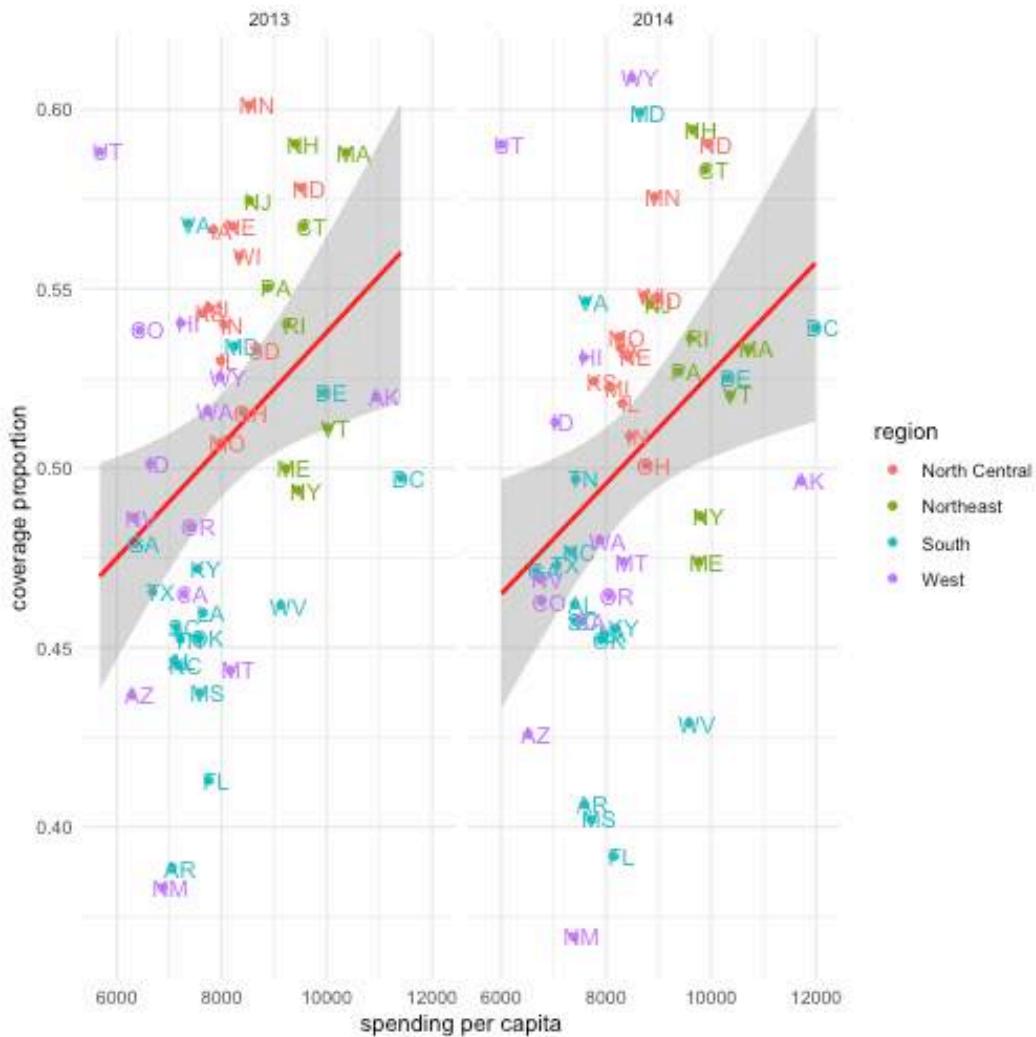
```
# color by region
hc %>%
  filter(type == "Employer",
         year == "2013") %>%
  ggplot(aes(x = spending_capita,
              y = prop_coverage,
              color = region)) +
  geom_point() +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-110

So far, we've only been focusing on data from 2013. What about looking at data from both 2013 and 2014? We can do that using `facet_wrap()`:

```
# color by region
hc %>%
  filter(type == "Employer") %>%
  ggplot(aes(x = spending_capita,
             y = prop_coverage,
             color = region)) +
  geom_point() +
  facet_wrap(~year) +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
```

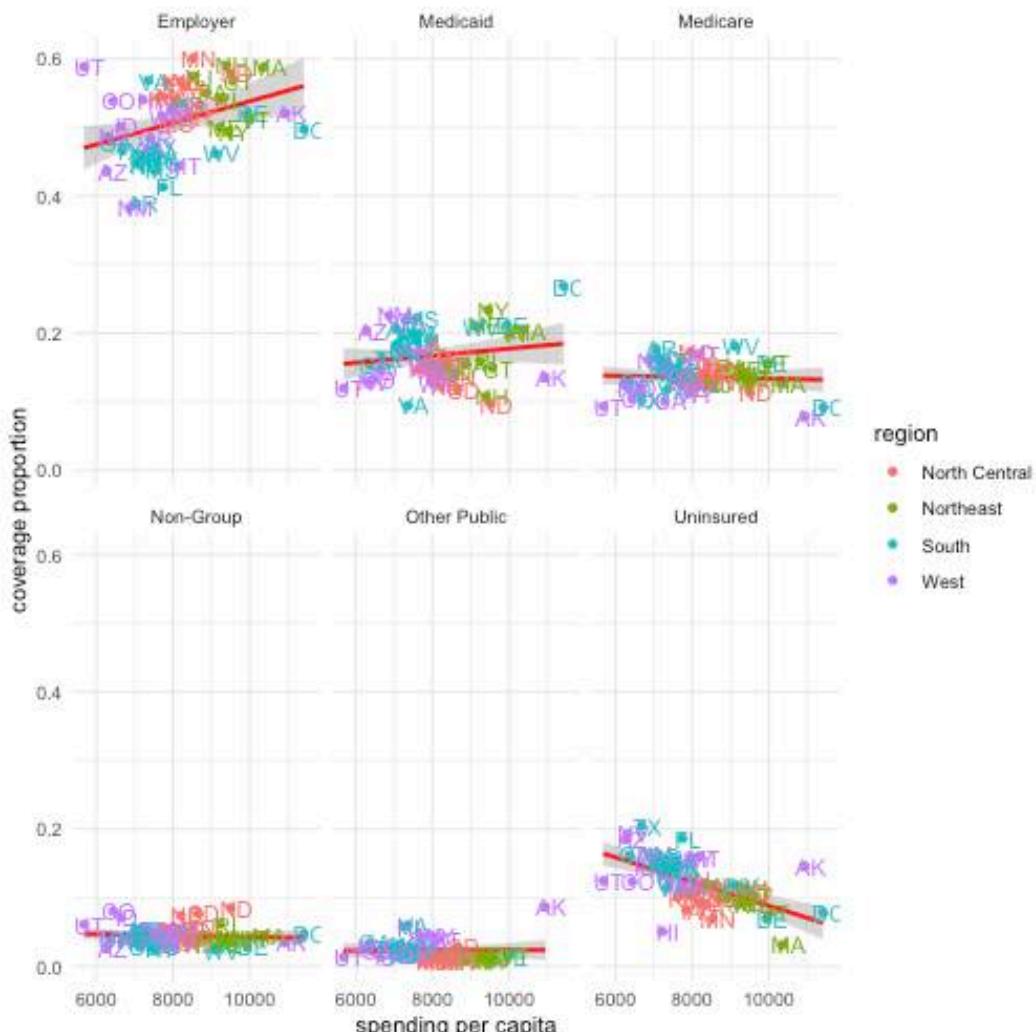


plot of chunk unnamed-chunk-111

We see that the overall trend holds, but there has been some movement. For example, we see at a glance that DC has a higher proportion of its population covered in 2014 relative to 2013, while MA saw a drop in coverage. UT appears to be an outlier in both years having low spending but a high proportion of individuals covered.

Beyond “Employer”-held health care coverage, let’s look at the other types of coverage data we have. Here, we’ll facet by type, rather than year, again focusing on just data from 2013.

```
# visualize 2013 data by type
hc %>%
  filter(year == "2013") %>%
  ggplot(aes(x = spending_capita,
             y = prop_coverage,
             color = region)) +
  geom_point() +
  facet_wrap(~type) +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
```

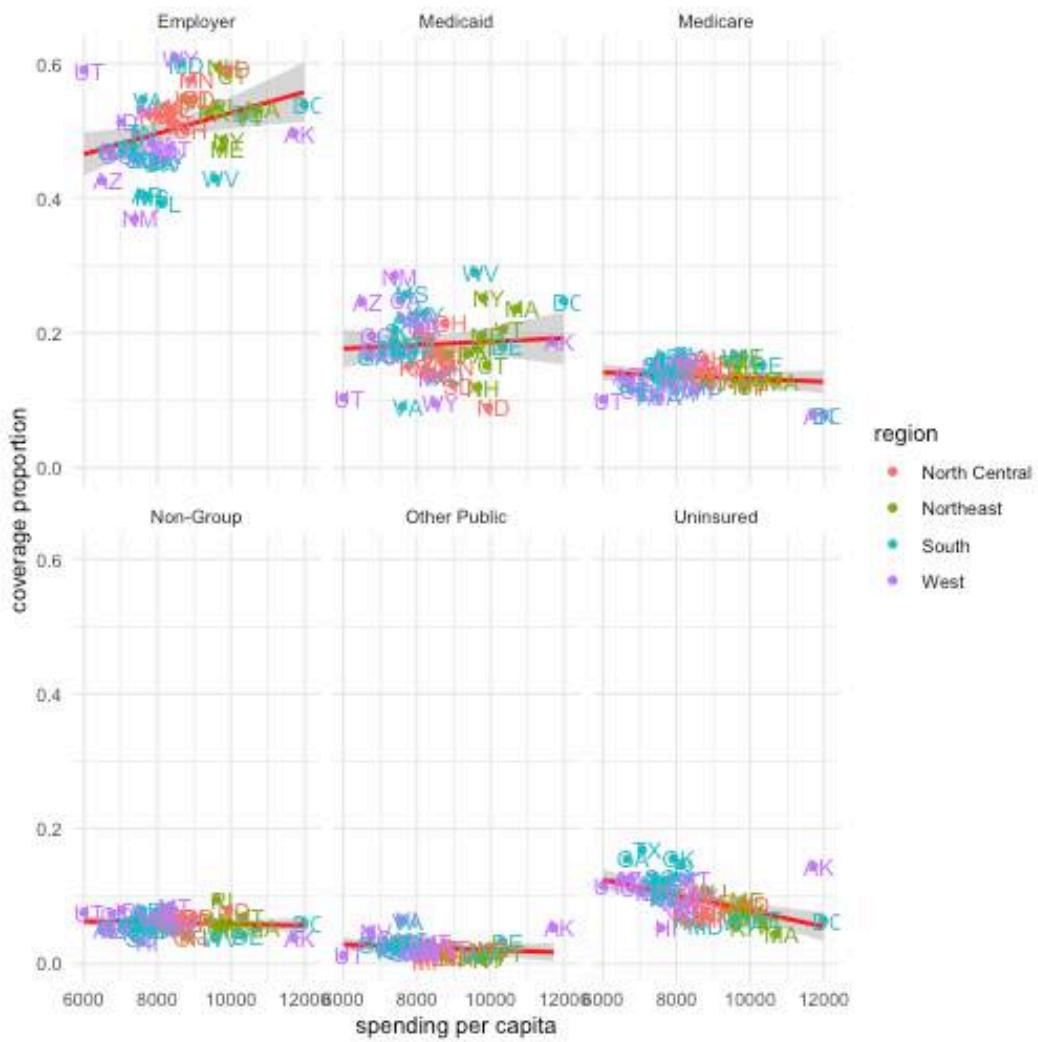


plot of chunk unnamed-chunk-112

From these data, we see that Employer health care coverage is the most popular way in which individuals receive their health insurance across all states. We also see a flat or positive relationship for all other types of insurance, except for “Uninsured”. We see that the more money spent per capita the fewer individuals the state has without insurance, as one might expect.

We can quickly peek at the data from 2014 to be sure the same general patterns hold:

```
# visualize 2014 data by type
hc %>%
  filter(year == "2014") %>%
  ggplot(aes(x = spending_capita,
             y = prop_coverage,
             color = region)) +
  geom_point() +
  facet_wrap(~type) +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-113

The same general patterns hold in 2014 as we saw in 2013; however, the patterns are not exactly the same.

With these plots we have a pretty good understanding of the relationship between spending and coverage across this country when it comes to health care.

Let's save some of these plots for later:

```
pdf(here::here("figures", "exploratory", "2013and2014_spending_and_coverage.pdf"))
))

hc %>%
  filter(type == "Employer") %>%
  ggplot(aes(x = spending_capita,
              y = prop_coverage,
              color = region)) +
  geom_point() +
  facet_wrap(~year) +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
dev.off()
quartz_off_screen
  2

pdf(here::here("figures", "exploratory", "2013_coverage_type.pdf"))
hc %>%
  filter(year == "2013") %>%
  ggplot(aes(x = spending_capita,
              y = prop_coverage,
              color = region)) +
  geom_point() +
  facet_wrap(~type) +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
Warning: Removed 9 rows containing non-finite values (stat_smooth).
Warning: Removed 9 rows containing missing values (geom_point).
Warning: Removed 9 rows containing missing values (geom_text).
```

```
dev.off()
quartz_off_screen
2

pdf(here::here("figures", "exploratory", "2014_coverage_type.pdf"))
hc %>%
  filter(year == "2014") %>%
  ggplot(aes(x = spending_capita,
              y = prop_coverage,
              color = region)) +
  geom_point() +
  facet_wrap(~type) +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  geom_text(aes(label=abb),
            nudge_x = 150,
            show.legend = FALSE)
`geom_smooth()` using formula 'y ~ x'
Warning: Removed 10 rows containing non-finite values (stat_smooth).
Warning: Removed 10 rows containing missing values (geom_point).
Warning: Removed 10 rows containing missing values (geom_text).
dev.off()
quartz_off_screen
2
```

Q2: Spending Across Geographic Regions?

To answer the question:

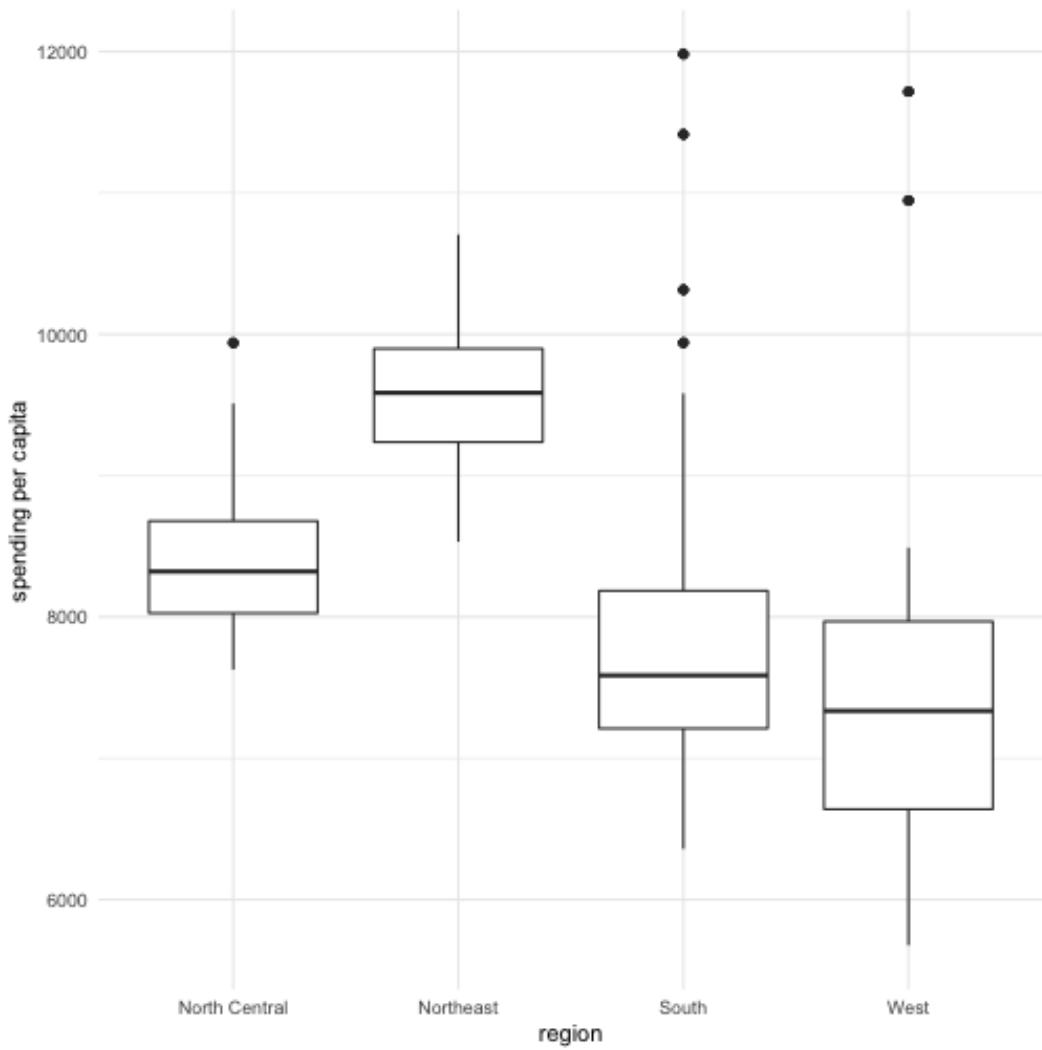
Which US states spend the most and which spend the least on health care? How does the spending distribution change across geographic regions in the United States?

We'll want to visualize health care spending across regions of the US.

We saw in the previous plots that there are some regional effects when it comes to spending, as the states from the different regions tended to cluster in the previous plot. But, let's look at this explicitly now.

We're looking to visualize a continuous variable (`spending_capita`) by a categorical variable (`region`), so a boxplot is always a good place to start:

```
# generate boxplot
hc %>%
  ggplot(aes(x = region,
              y = spending_capita)) +
  geom_boxplot() +
  labs(y = "spending per capita")
```

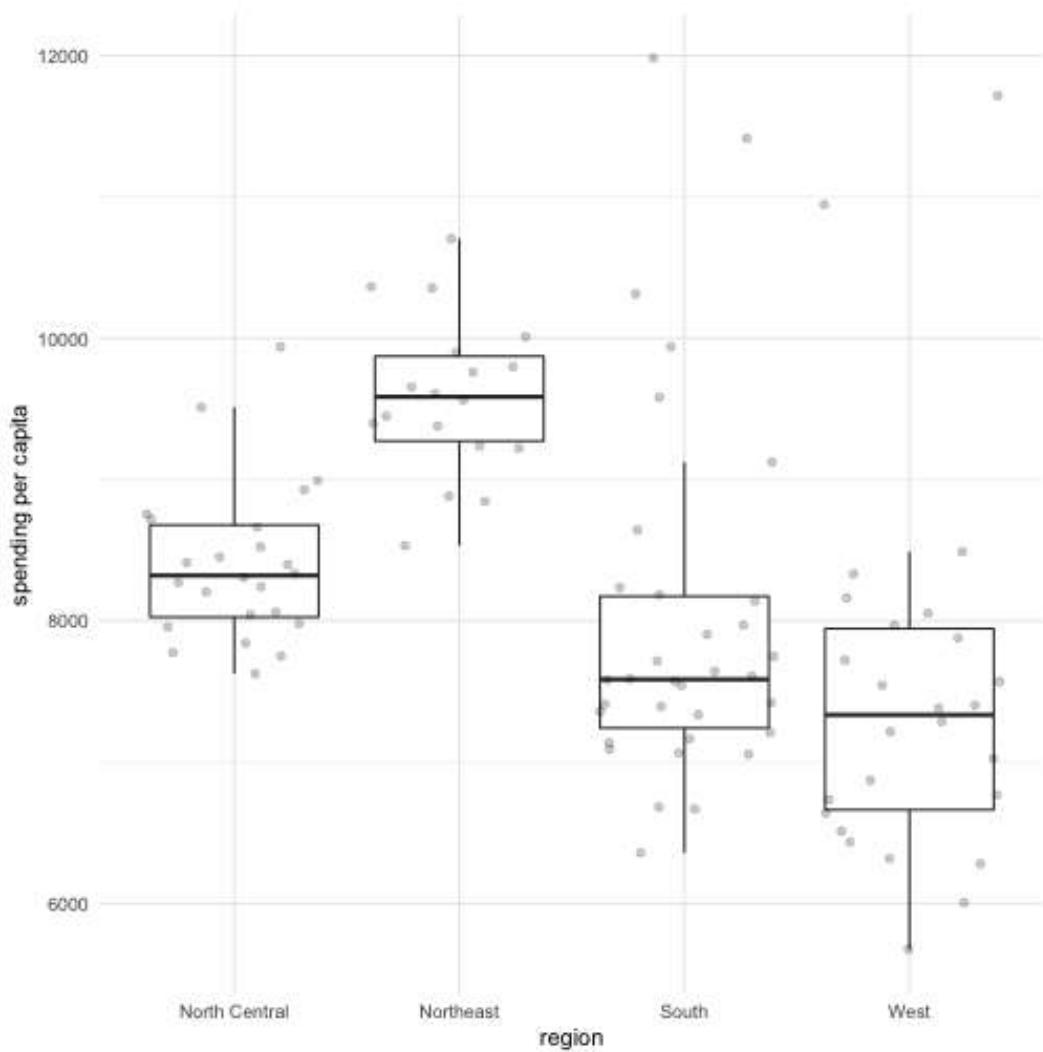


plot of chunk unnamed-chunk-115

Here, we get a sense of the overall trend, seeing that states in the Northeast tend to spend the most on health care, while states in the West spend the least.

But, sometimes, it can be helpful to see the distribution of points along with the boxplot, so we can add those onto the plot. We'll use `geom_jitter()` rather than `geom_point()` here so that we can see all the points without overlap. Note that we also use `alpha = 0.2` to increase the transparency of the points and `outlier.shape = NA` to hide the outliers on the boxplot. This way, each observation is only plotted a single time:

```
# add data points to boxplot
hc %>%
  filter(type == "Employer") %>%
  ggplot(aes(x = region,
             y = spending_capita)) +
  geom_boxplot(outlier.shape = NA) +
  geom_jitter(alpha = 0.2) +
  labs(y = "spending per capita")
```



plot of chunk unnamed-chunk-116

This gives us a sense of the variation in spending for states in each region. Of note, there are more outliers in the South and West, with a few states spending more on health care than even states in the Northeast, where spending tends to be higher.

With this we have a good sense of the regional effects of spending across the United states.

Q3: Coverage and Spending Change Over Time?

To answer the question:

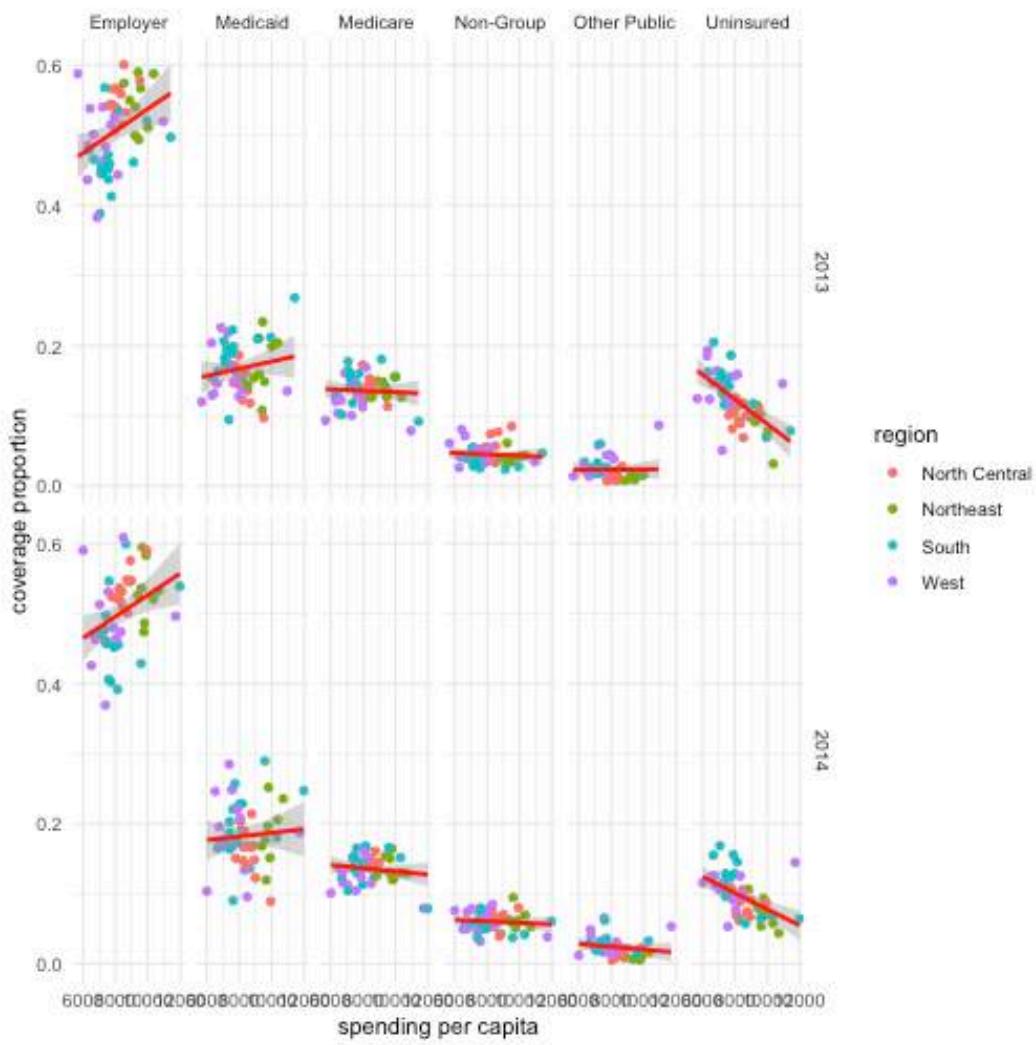
Does the relationship between health care coverage and health care spending in the United States change from 2013 to 2014?

we'll need to visualize a whole bunch of variables: coverage, spending, year and type of insurance. We can return to a scatterplot again, but now we'll put all these pieces we looked at separately in Q1 together at once to answer this temporal question visually.

We won't filter the dataset but will instead use `facet_grid()`, which "forms a matrix of panels defined by row and column facetting variables":

`facet_wrap()`, which we used previously, *typically* utilizes screen space better than `facet_grid()`; however, in this case, where we want 2013 and 2014 to each be in a separate row, `facet_grid()` is the better visual choice.

```
# color by region
hc %>%
  ggplot(aes(x = spending_capita,
             y = prop_coverage,
             color = region)) +
  geom_point() +
  labs(x = "spending per capita",
       y = "coverage proportion") +
  geom_smooth(method = "lm", col = "red") +
  facet_grid(year~type)
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-117

With this output, the top row shows the data from 2013 and the bottom shows the data from 2014. We can then visually compare the top plot to the bottom plot for each time of insurance.

Visually, we can start to get a sense that a few things changed from 2013 to 2014. For example, as we saw previously, individual states changed from one year to the next, but overall patterns seem to hold pretty steady between these two years.

We were able to start answering all our questions of interest with a number of quick plots using `ggplot2`.

Now, of course there are *many* ways in which we could customize each of these plots to be “publication-ready,” but for now, we’ll stop with this case study having gained a lot of new insight into these data and answers to our questions we now have, simply from generating a few exploratory plots.

Case Study #2: Firearms

For our second case study, we’re interested in the following question:

At the state-level, what is the relationship between firearm legislation strength and annual rate of fatal police shootings?

In the previous course, we wrangled the data into a single, helpful dataframe: `firearms`.

```
# see firearms data
firearms
# A tibble: 51 × 15
  NAME      white black hispanic male total_pop violent_crime brad\
y_scores
  <chr>     <dbl> <dbl>    <dbl> <dbl>    <dbl>          <dbl>   \
  <dbl>
  1 alabama  69.5  26.7     4.13  48.5    4850858        472.   \
  -18
  2 alaska   66.5   3.67    6.82  52.4    737979         730.   \
  -30
  3 arizona  83.5   4.80    30.9   49.7    6802262        410.   \
  -39
  4 arkansas 79.6   15.7    7.18   49.1    2975626        521.   \
  -24
  5 california 73.0   6.49    38.7   49.7    39032444       426.   \
  -76
  6 colorado  87.6   4.47    21.3   50.3    5440445         321   \
  -22
  7 connecticut 80.9  11.6    15.3   48.8    3593862        218.   \
  -73
  8 delaware   70.3  22.5    8.96   48.4    944107         499   \
  -41
  9 district of columbia 44.1  48.5    10.7   47.4    672736        1269.  \
```

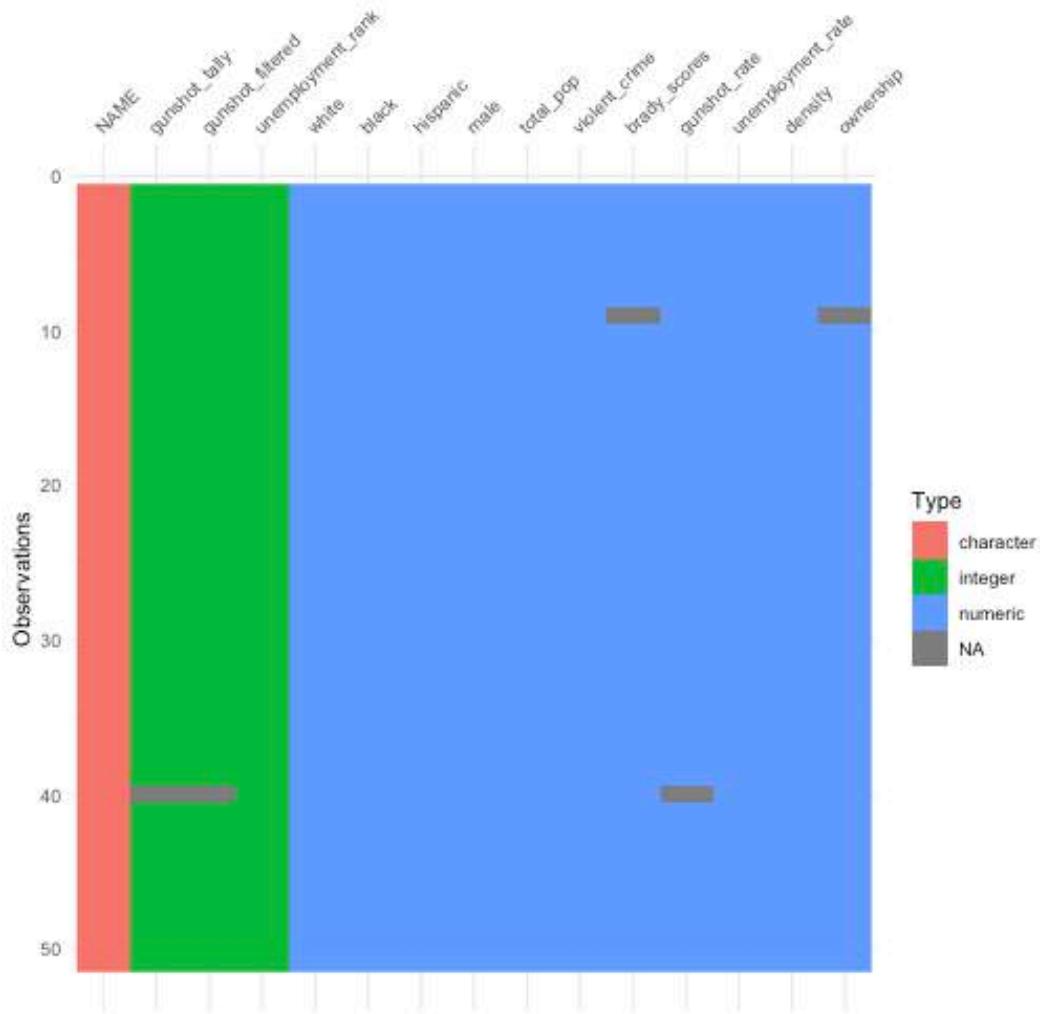
```
NA
10 florida          77.7 16.9      24.7    48.9  20268567        462.      \
-20.5
# ... with 41 more rows, and 7 more variables: gunshot_tally <int>,
#   gunshot_filtered <int>, gunshot_rate <dbl>, unemployment_rate <dbl>,
#   unemployment_rank <int>, density <dbl>, ownership <dbl>
```

This dataset contains state level information about firearm ownership (broken down by ethnicity and gender), the population of each state (total_pop), the number of violent crimes (violent_crime), the “total state points” from the Brady Scorecard (brady_scores), the number of gunshots (gunshot_tally), the number of gunshots from armed, non-white, male individuals (gunshot_filtered), the annualized rate per 1,000,000 residents (gunshot_rate), the unemployment_rate and unemployment_rank, population density (density), and firearm ownership as a percent of firearm suicides to all suicides (ownership).

Exploratory Data Analysis (EDA)

Similar to how we approached the health care case study, let’s get an overall understanding of the data in our dataset, starting with understanding where data are missing in our dataset:

```
# visualize missingness
vis_dat(firearms)
```



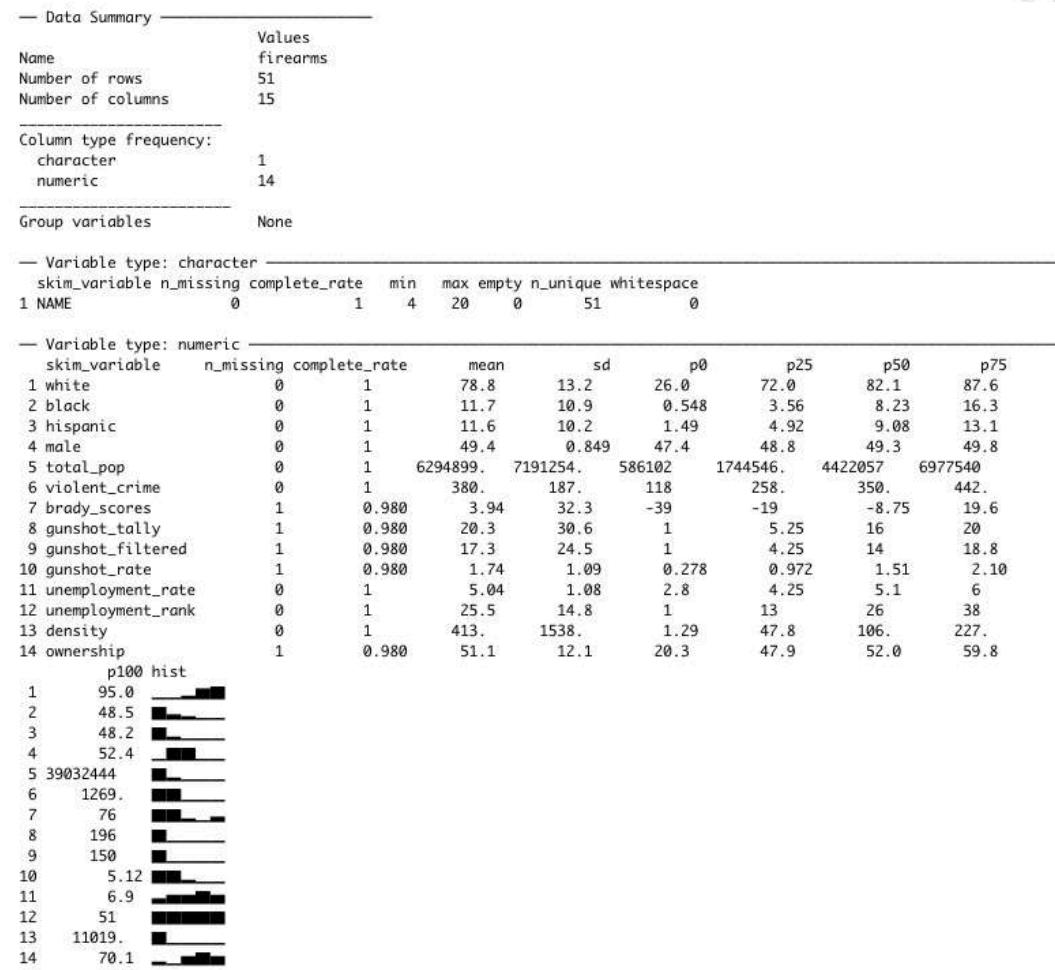
plot of chunk unnamed-chunk-119

We see that we have data for all 50 states (and Washington, D.C.) for most variables in our dataset; however, we're missing information for one state when it comes to gunshot information and another state when it comes to Brady scores and ownership by suicide rates.

Additionally, we see that most of our variables are numeric (either integers or numeric) while the state name (NAME) is a character, confirming what we expect for these data.

We will also again use `skim()` to get a better sense of the values in our dataset:

```
# summarize data
skim(firearms)
```

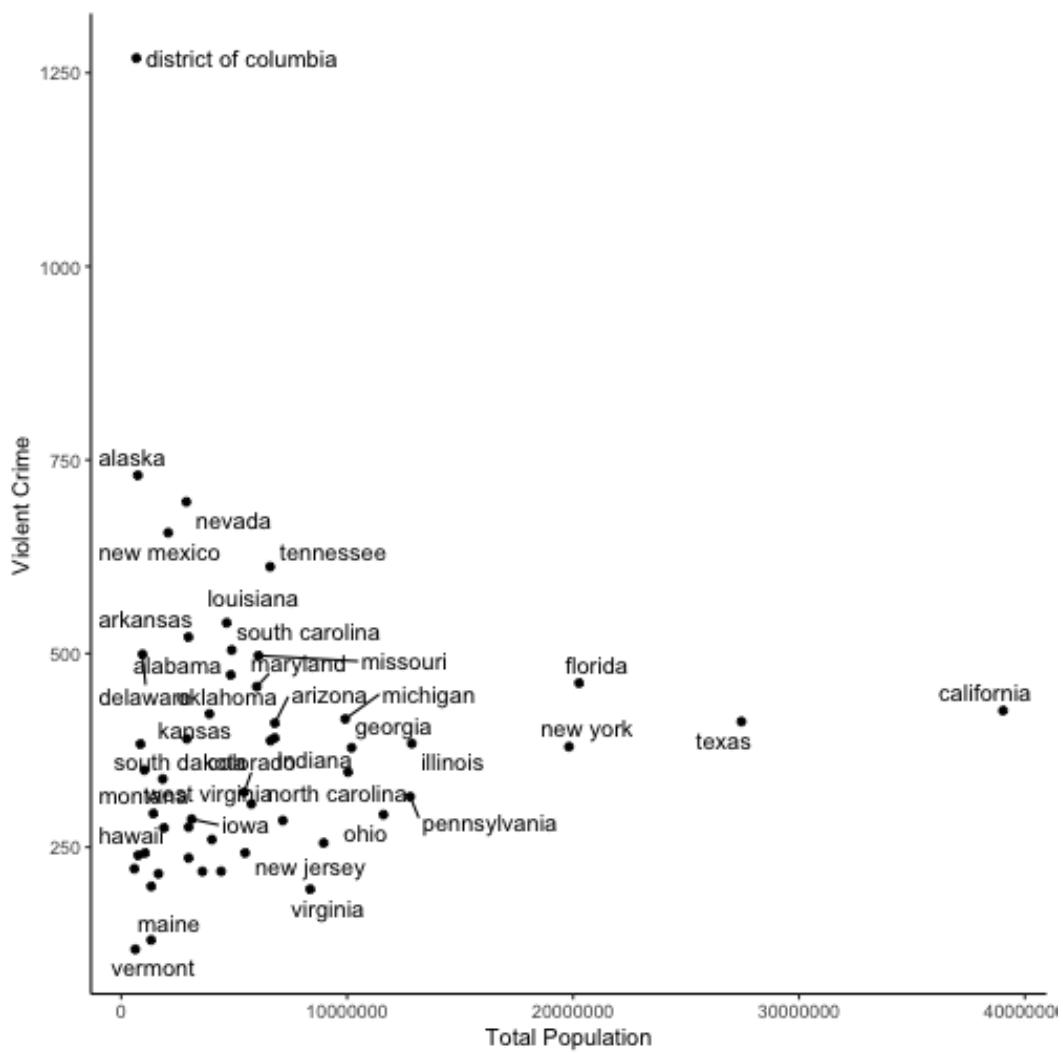


skim output

Ultimately, we're interested in firearm legislation and fatal police shootings, which we'll get to, but let's explore the relationship between other variables in our dataset first to gain a more complete understanding.

For example, determining if there's a relationship between the number of violent crimes and the total population in a state is good to know. Outliers in this relationship would be important to note as they would be states with either lower or higher crime by population. Let's use a scatterplot to better understand this relationship:

```
# violent crimes
ggplot(firearms,
       aes(x = total_pop,
           y = violent_crime)) +
  geom_point() +
  labs(x = "Total Population",
       y = "Violent Crime") +
  theme_classic() +
  geom_text_repel(aes(label = NAME))
Warning: ggrepel: 15 unlabeled data points (too many overlaps). Consider
increasing max.overlaps
```

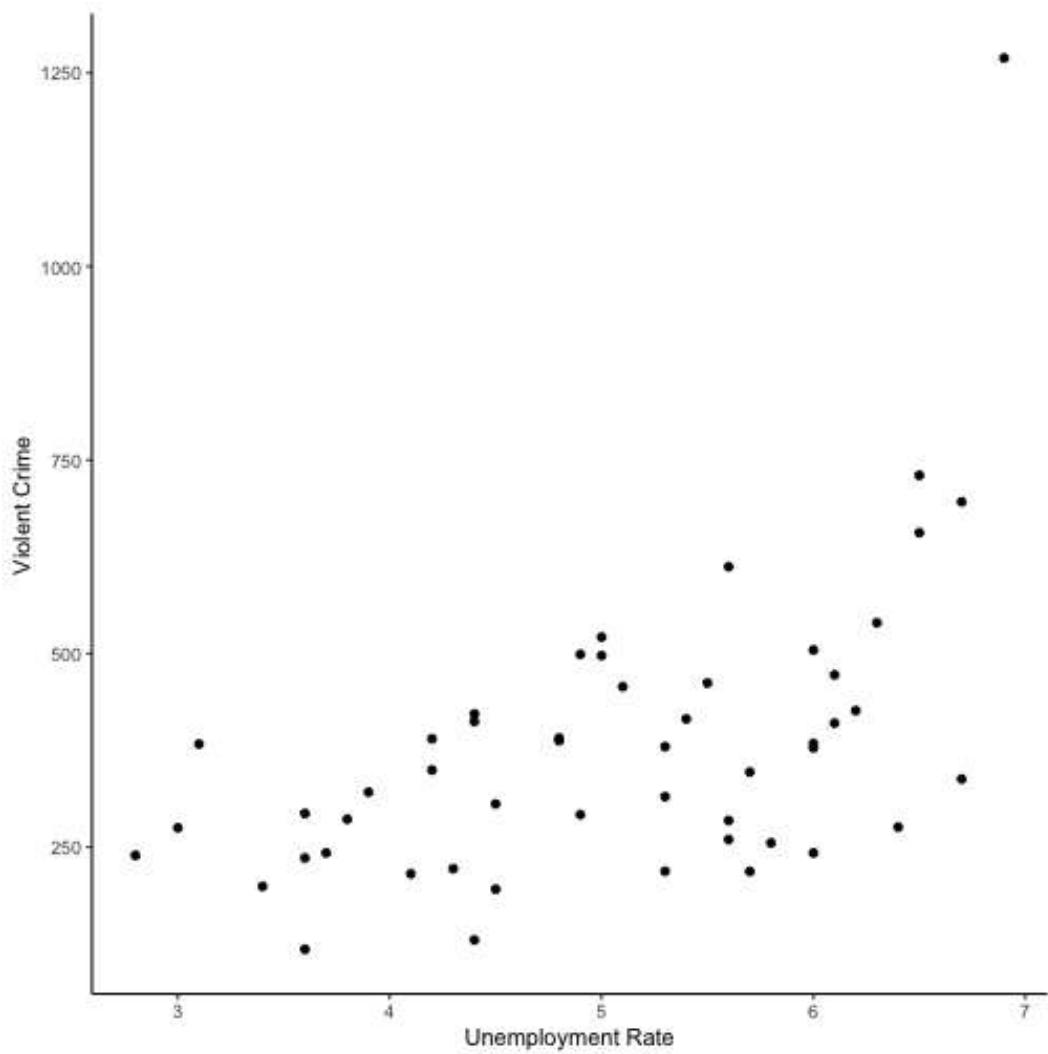


plot of chunk unnamed-chunk-121

Here we see that the visualization of this relationship is largely dependent upon the state's population, with large states like Texas and California sticking out. However, we do see that, for its size, Washington D.C. had many more violent cries than other states, despite its small population.

It would also be helpful to understand the relationship between unemployment and violent crime:

```
# violent crimes
ggplot(firearms,
       aes(x = unemployment_rate,
            y = violent_crime)) +
  geom_point() +
  labs(x = "Unemployment Rate",
       y = "Violent Crime") +
  theme_classic()
```

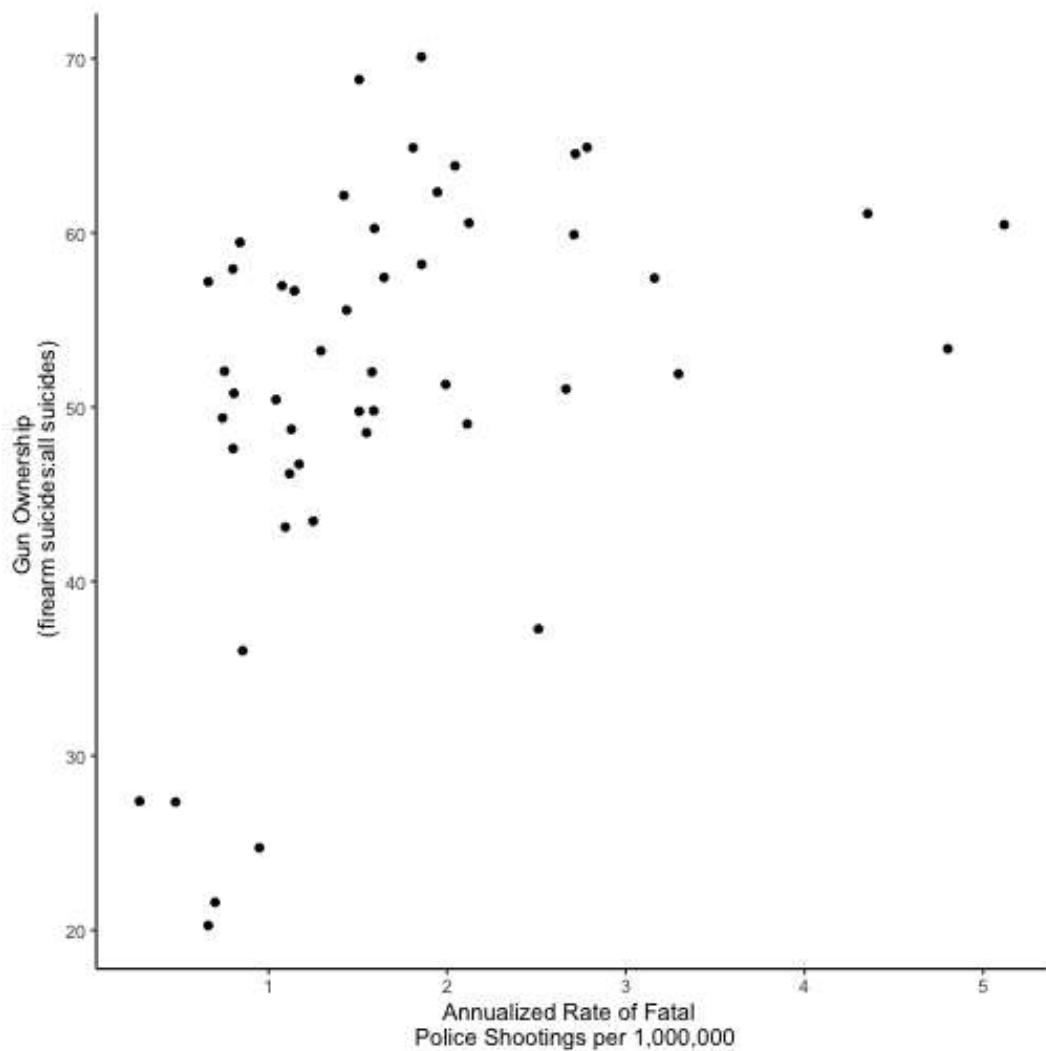


plot of chunk unnamed-chunk-122

Here, there appears to be some relationship with states that have higher rates of unemployment having slightly more violent crimes, but violent crimes is not adjusted by population, so this is likely not *that* helpful.

What about the relationship between fatal police shootings and gun ownership as a percent of firearm suicides to all suicides.

```
# violent crimes
ggplot(firearms,
       aes(x = gunshot_rate,
           y = ownership)) +
  geom_point() +
  labs(x = "Annualized Rate of Fatal \n Police Shootings per 1,000,000",
       y = "Gun Ownership \n (firearm suicides:all suicides)") +
  theme_classic()
Warning: Removed 2 rows containing missing values (geom_point).
```



plot of chunk unnamed-chunk-123

This suggests that states with more fatal police shootings *tend* to have more firearm suicides relative to non-firearm suicides; however, this relationship is non linear.

With these plots, we're starting to get an understanding of the data and see that there are patterns and don't appear to be wild outliers in any one variable (although, we should keep an eye on Washington, D.C. as it appeared as an outlier in a few plots). With that we're confident we can move on to start looking into our question of interest.

Q: Relationship between Fatal Police Shootings and Legislation?

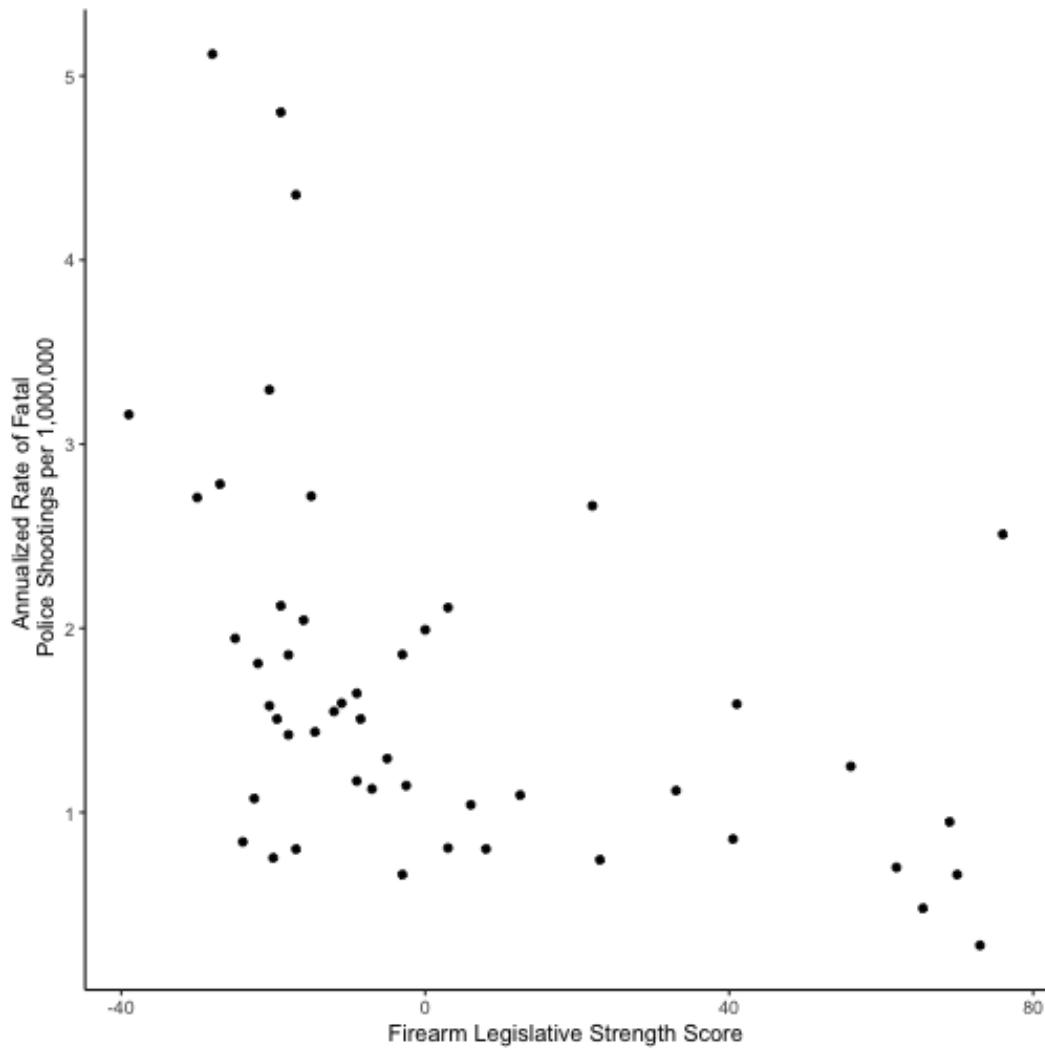
Ultimately, we're interested in firearm legislation and fatal police shootings, so let's focus in on Brady scores here, which measure legislation and gunshot_tally, a measure of the rate of fatal police shootings.

We see that the average brady_score (mean) is 3.94, with state values ranging from -39 to 76.

When it comes to the rate of police shootings (gunshot_rate), we see the average number (mean) for a state is 1.74, with state values ranging from 0.28 to 5.12, depending on the state.

To start to understand the relationship between the two, we'll want to visualize this relationship using a scatterplot:

```
# visualize legislation and shootings
ggplot(firearms, aes(x = brady_scores,
                      y = gunshot_rate)) +
  geom_point() +
  labs(x = "Firearm Legislative Strength Score",
       y = "Annualized Rate of Fatal \n Police Shootings per 1,000,000") +
  theme_classic()
Warning: Removed 2 rows containing missing values (geom_point).
```

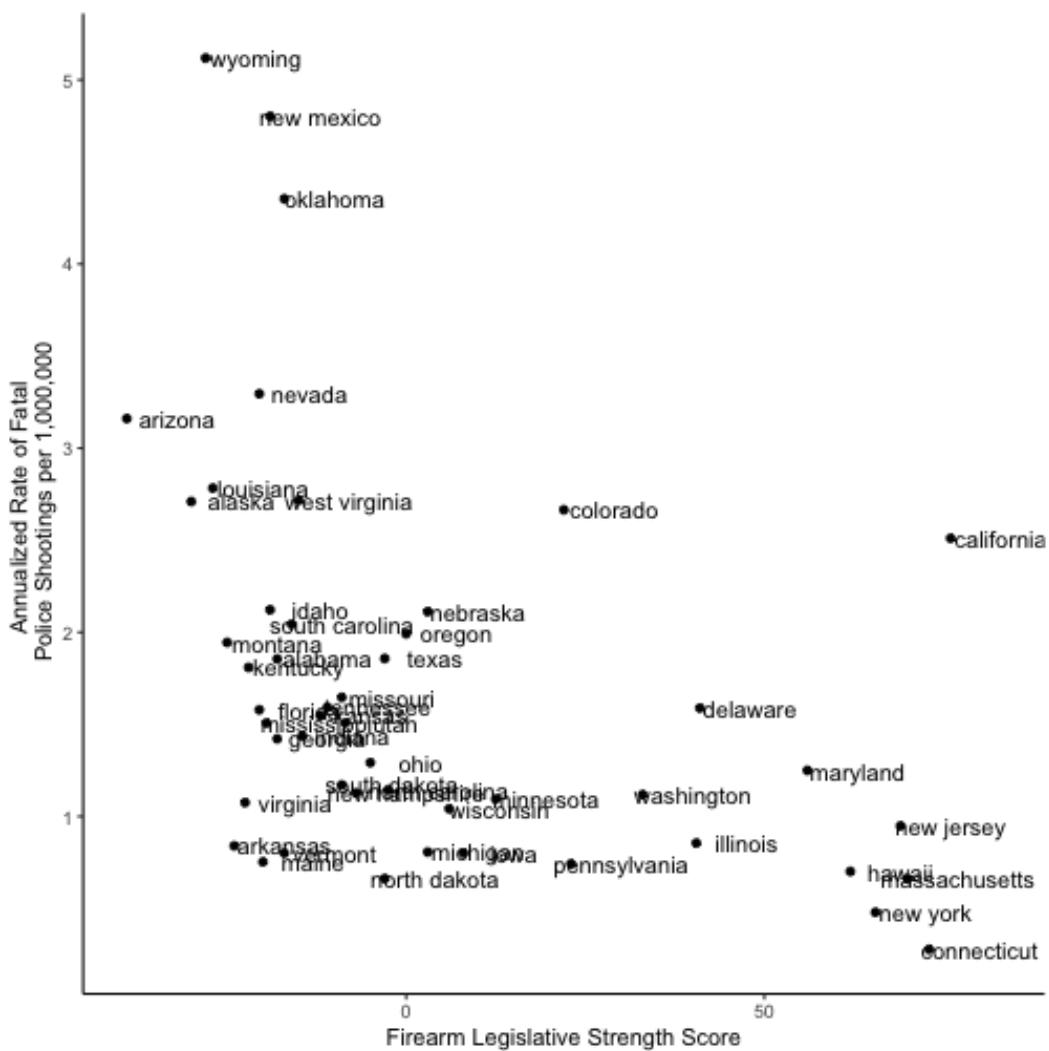


plot of chunk unnamed-chunk-124

In this plot, we see that there is a relationship, but it is non-linear. Overall, the higher the legislative strength score (`brady_scores`), the lower the rate of police shootings; however, this decrease is nonlinear, as all states with a positive Brady Score have a similar police shooting rate.

For now, we won't explore a model, but we will label the points to better contextualize the information displayed. To see which states are at either end of the distribution, using `geom_text()` is a good option:

```
# label points with state name
ggplot(firearms, aes(x = brady_scores,
                      y = gunshot_rate)) +
  geom_point() +
  labs(x = "Firearm Legislative Strength Score",
       y = "Annualized Rate of Fatal \n Police Shootings per 1,000,000") +
  theme_classic() +
  geom_text(aes(label = NAME),
            nudge_x = 7)
Warning: Removed 2 rows containing missing values (geom_point).
Warning: Removed 2 rows containing missing values (geom_text).
```

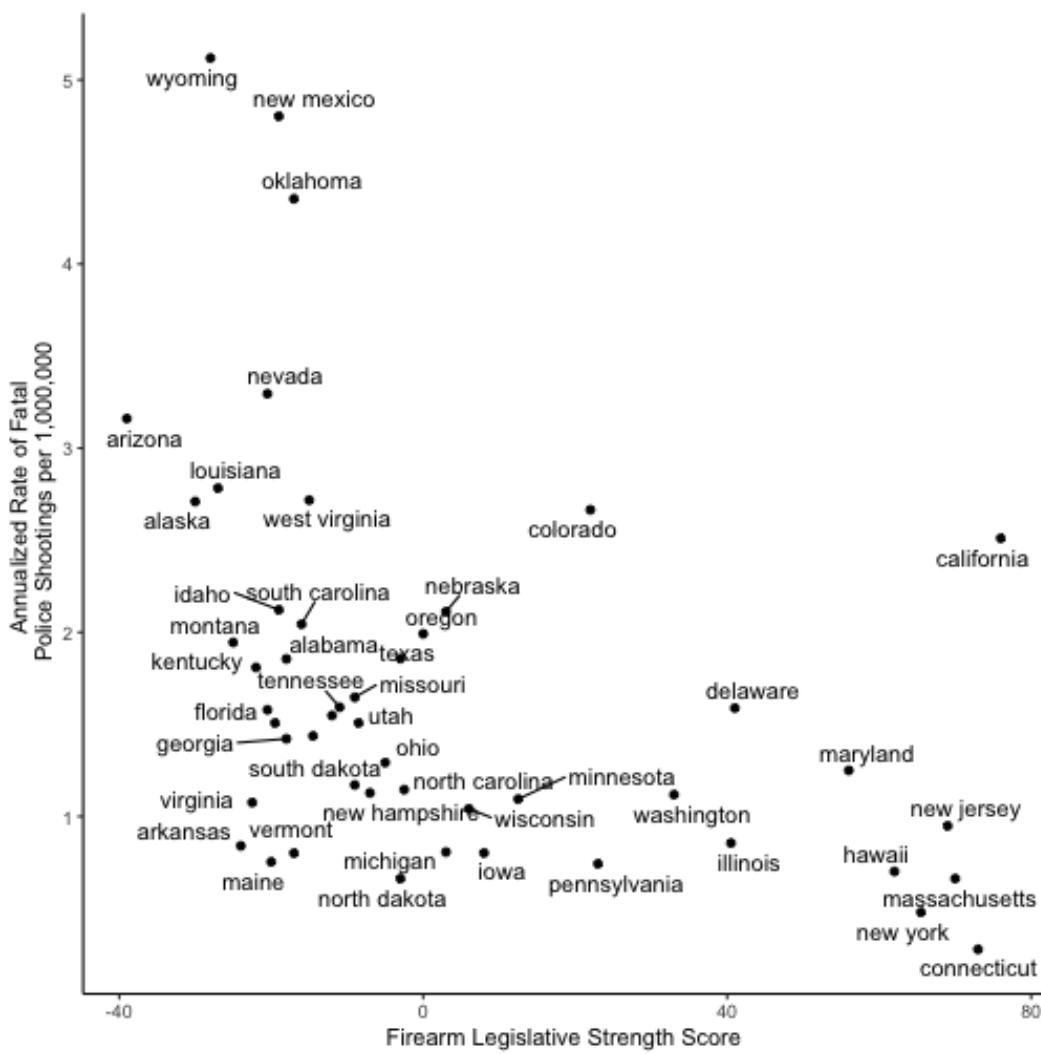


This makes it clear that Wyoming, New Mexico, Oklahoma, Arizona, and Nevada have some of the highest rates of fatal police shootings, while Connecticut, New York, Pennsylvania, and North Dakota are among the lowest.

However, a number of labels overlap there, so it can be helpful to use `ggrepel` if you want to be able to match each point to its label:

```
library(ggrepel)

# repel points with state name
ggplot(firearms, aes(x = brady_scores,
                      y = gunshot_rate)) +
  geom_point() +
  labs(x = "Firearm Legislative Strength Score",
       y = "Annualized Rate of Fatal \n Police Shootings per 1,000,000") +
  theme_classic() +
  geom_text_repel(aes(label = NAME))
Warning: Removed 2 rows containing missing values (geom_point).
Warning: Removed 2 rows containing missing values (geom_text_repel).
Warning: ggrepel: 3 unlabeled data points (too many overlaps). Consider
increasing max.overlaps
```



plot of chunk unnamed-chunk-126

If we wanted to save this particular plot as a pdf, we could do so like this to save the plot in a directory called exploratory within a directory called figures:

```
pdf(here::here("figures", "exploratory", "Fatal_police_shootings_and_firearm_le\
gislative_strength.pdf"))

ggplot(firearms, aes(x = brady_scores,
                      y = gunshot_rate)) +
  geom_point() +
  labs(x = "Firearm Legislative Strength Score",
       y = "Annualized Rate of Fatal \n Police Shootings per 1,000,000") +
  theme_classic() +
  geom_text_repel(aes(label = NAME))
Warning: Removed 2 rows containing missing values (geom_point).
Warning: Removed 2 rows containing missing values (geom_text_repel).
Warning: ggrepel: 3 unlabeled data points (too many overlaps). Consider
increasing max.overlaps

dev.off()
quartz_off_screen
2
```

5. Modeling Data in the Tidyverse

About This Course

Developing insights about your organization, business, or research project depends on effective modeling and analysis of the data you collect. Building effective models requires understanding the different types of questions you can ask and how to map those questions to your data. Different modeling approaches can be chosen to detect interesting patterns in the data and identify hidden relationships.

This course covers the types of questions you can ask of data and the various modeling approaches that you can apply. Topics covered include hypothesis testing, linear regression, nonlinear modeling, and machine learning. With this collection of tools at your disposal, as well as the techniques learned in the other courses in this specialization, you will be able to make key discoveries from your data for improving decision-making throughout your organization.

In this specialization we assume familiarity with the R programming language. If you are not yet familiar with R, we suggest you first complete [R Programming](#) before returning to complete this course.

The Purpose of Data Science

Data science has multiple definitions. For this module we will use the definition:

Data science is the process of formulating a quantitative question that can be answered with data, collecting and cleaning the data, analyzing the data, and communicating the answer to the question to a relevant audience.

In general the data science process is iterative and the different components blend together a little bit. But for simplicity lets discretize the tasks into the following 7 steps:

1. Define the question you want to ask the data
2. Get the data
3. Clean the data

4. Explore the data
5. Fit statistical models
6. Communicate the results
7. Make your analysis reproducible

This module is focused on three of these steps: (1) defining the question you want to ask, (4) exploring the data and (5) fitting statistical models to the data.

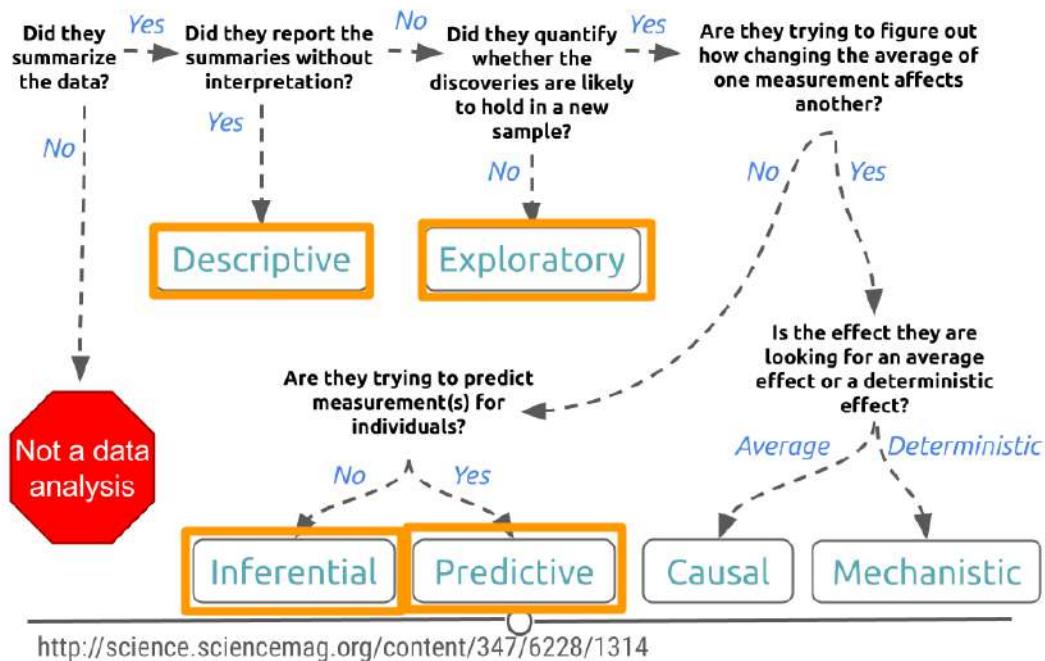
We have seen previously how to extract data from the web and from databases and we have seen how to clean it up and tidy the data. You also know how to use plots and graphs to visualize your data. You can think of this module as using those tools to start to answer questions using the tools you have already learned about.

Types of Data Science Questions

We will look at a few different types of questions that you might want to answer from data. This flowchart gives some questions you can ask to figure out what type of question your analysis focuses on. Each type of question has different goals.

There are four classes of questions that we will focus on:

1. **Descriptive:** The goal of descriptive data science questions is to understand the components of a dataset, describe what they are, and explain that description to others who might want to understand the data. This is the simplest type of data analysis.
2. **Exploratory:** The goal of exploratory data science questions is to find unknown relationships between the different variables you have measured in your dataset. Exploratory analysis is open ended and designed to find expected or unexpected relationships between different measurements. We have already seen how plotting the data can be very helpful to get a general understanding about how variables relate to one another.
3. **Inferential:** The goal of inferential data science questions is to use a small sample of data to say something about what would happen if we collected more data. Inferential questions come up because we want to understand the relationships between different variables but it is too expensive or difficult to collect data on every person or object.
4. **Predictive:** The goal of predictive data science question is to use data from a large collection to predict values for new individuals. This might be predicting what will happen in the future or predicting characteristics that are difficult to measure. Predictive data science is sometimes called machine learning.



One primary thing we need to be aware of is that just because two variables are correlated with each other, doesn't mean that changing one causes a change in the other.

One way that people illustrate this idea is to look at data where two variables show a relationship, but are clearly not related to each other. For example, in a specific time range, the number of people who drown while falling into a pool is related to the number of films that Nicholas Cage appears in. These two variables are clearly unrelated to each other, but the data seems to show a relationship. We'll discuss more later.

Data Needs

Let's assume you have the dataset that contains the variables you are looking for to evaluate the question(s) you are interested in, and it is tidy and ready to go for your analysis. It's always nice to step back to make sure the data is the right data before you spend hours and hours on your analysis. So, let's discuss some of the potential and common issues people run into with their data.

Number of observations is too small

It happens quite often that collecting data is expensive or not easy. For instance, in a medical study on the effect of a drug on patients with Alzheimer disease, researchers will be happy if they can get a sample of 100 people. These studies are expensive, and it's hard to find volunteers who enroll in the study. It is also the case with most social experiments. While data are everywhere, the data you need may not be. Therefore, most data scientists at some point in their career face the curse of small sample size. Small sample size makes it hard to be confident about the results of your analysis. So when you can, and it's feasible, a large sample is preferable to a small sample. But when your only available dataset to work with is small you will have to note that in your analysis. Although we won't learn them in this course, there are particular methods for inferential analysis when sample size is small.

Dataset does not contain the exact variables you are looking for

In data analysis, it is common that you don't always have what you need. You may need to know individuals' IQ, but all you have is their GPA. You may need to understand food expenditure, but you have total expenditure. You may need to know parental education, but all you have is the number of books the family owns. It is often that the variable that we need in the analysis does not exist in the dataset and we can't measure it. In these cases, our best bet is to find the closest variables to that variable. Variables that may be different in nature but are highly correlated with (similar to) the variable of interest are what are often used in such cases. These variables are called proxy variables.

For instance, if we don't have parental education in our dataset, we can use the number of books the family has in their home as a proxy. Although the two variables are different, they are highly correlated (very similar), since more educated parents tend to have more books at home. So in most cases where you can't have the variable you need in your analysis, you can replace it with a proxy. Again, it must always be noted clearly in your analysis why you used a proxy variable and what variable was used as your proxy.

Variables in the dataset are not collected in the same year

Imagine we want to find the relationship between the effect of cab prices and the number of rides in New York City. We want to see how people react to price changes. We get a hold of data on cab prices in 2018, but we only have data on the number of rides from 2015. Can these two variables be used together in our analysis? Simply, no. If we want to answer this

question, we can't match these two sets of data. If we're using the prices from 2018, we should find the number of rides from 2018 as well. Unfortunately, a lot of the time, this is an issue you'll run into. You'll either have to find a way to get the data from the same year or go back to the drawing board and ask a different question. This issue can be ignored only in cases where we're confident the variables does not change much from year to year.

Dataset is not representative of the population that you are interested in

You will hear the term **representative sample**, but what is it? Before defining a representative sample, let's see what a population is in statistical terms. We have used the word population without really getting into its definition.

A sample is part of a **population**. A population, in general, is every member of the whole group of people we are interested in. Sometimes it is possible to collect data for the entire population, like in the U.S. Census, but in most cases, we can't. So we collect data on only a subset of the population. For example, if we are studying the effect of sugar consumption on diabetes, we can't collect data on the entire population of the United States. Instead, we collect data on a sample of the population. Now, that we know what sample and population are, let's go back to the definition of a representative sample.

A representative sample is a sample that accurately reflects the larger population. For instance, if the population is every adult in the United States, the sample includes an appropriate share of men and women, racial groups, educational groups, age groups, geographical groups, and income groups. If the population is supposed to be every adult in the U.S., then you can't collect data on just people in California, or just young people, or only men. This is the idea of a representative sample. It has to model the broader population in all major respects.

We give you one example in politics. Most recent telephone poles in the United States have been bad at predicting election outcomes. Why? This is because by calling people's landlines you can't guarantee you will have a representative sample of the voting age population since younger people are not likely to have landlines. Therefore, most telephone polls are skewed toward older adults.

Random sampling is a necessary approach to having a representative sample. Random sampling in data collection means that you randomly choose your subjects and don't choose who gets to be in the sample and who doesn't. In random sampling, you select your subjects from the population at random like based on a coin toss. The following are examples of lousy sampling:

- A research project on attitudes toward owning guns through a survey sent to subscribers of a gun-related magazine (gun magazine subscribers are not representative of the general population, and the sample is very biased).
- A research project on television program choices by looking at Facebook TV interests (not everybody has a Facebook account. Most online surveys or surveys on social media have to be taken with a grain of salt because not all members of all social groups have an online presentation or use social media.)
- A research study on school meals and educational outcomes done in a neighborhood with residents mainly from one racial group (school meals can have a different effect on different income and ethnic groups).

The moral of the story is to always think about what your population is. Your population will change from one project to the next. If you are researching the effect of smoking on pregnant women, then your population is, well, pregnant women (and not men). After you know your population, then you will always want collect data from a sample that is representative of your population. Random sampling helps.

And lastly, if you have no choice but to work with a dataset that is not collected randomly and is biased, be careful not to generalize your results to the entire population. If you collect data on pregnant women of age 18-24, you can't generalize your results to older women. If you collect data from the political attitudes of residents of Washington, DC, you can't say anything about the whole nation.

Some variables in the dataset are measured with error

Another curse of a dataset is measurement error. In simple, measurement error refers to incorrect measurement of variables in your sample. Just like measuring things in the physical world comes with error (like measuring distance, exact temperature, BMI, etc.), measuring variables in the social context can come with an error. When you ask people how many books they have read in the past year, not everyone remembers it correctly. Similarly, you may have measurement error when you ask people about their income. A good researcher recognizes measurement error in the data before any analysis and takes it into account during their analysis.

Variables are confounded

What if you were interested in determining what variables lead to increases in crime? To do so, you obtain data from a US city with lots of different variables and crime rates for

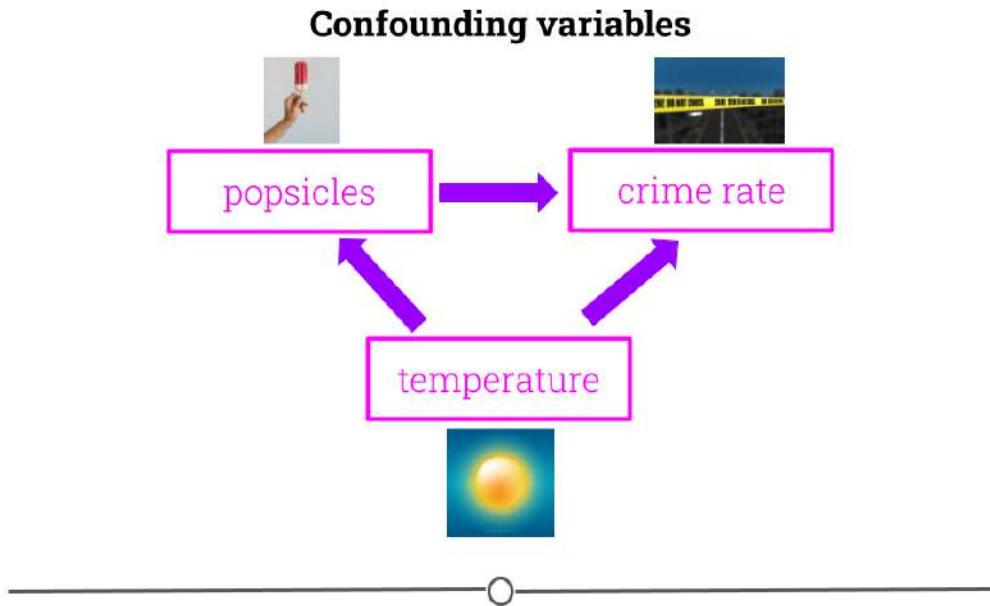
a particular time period. You would then wrangle the data and at first you look at the relationship between popsicle sales and crime rates. You see that the more popsicles that are sold, the higher the crime rate.



Source: freepik.com

Your first thought may be that popsicles lead to crimes being committed. However, there is a confounder that's not being considered!

In short, confounders are other variables that may affect our outcome but are also correlated with (have a relationship with) our main variable of interest. In the popsicle example, temperature is an important confounder. More crimes happen when it's warm out and more popsicles are sold. It's not the popsicles at all driving the relationship. Instead temperature is likely the culprit.



This is why getting an understanding of what data you have and how the variables relate to one another is so vital before moving forward with inference or prediction. We have already described exploratory analysis to some extent using visualization methods. Now we will recap a bit and discuss descriptive analysis.

Descriptive and Exploratory Analysis

Descriptive and Exploratory analysis will first and foremost generate simple summaries about the samples and their measurements to describe the data you're working with and how the variables might relate to one another. There are a number of common descriptive statistics that we'll discuss in this lesson: measures of central tendency (eg: mean, median, mode) or measures of variability (eg: range, standard deviations, or variance).

Descriptive Analysis



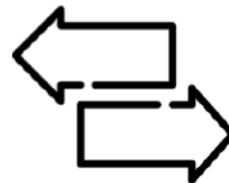
Size



Missingness



Shape

Central
Tendency

Variability

This type of analysis is aimed at summarizing your dataset. Unlike analysis approaches we'll discuss in later, descriptive and exploratory analysis is not for generalizing the results of the analysis to a larger population nor trying to draw any conclusions. Description of data is separated from interpreting the data. Here, we're just summarizing what we're working with.

Some examples of purely descriptive analysis can be seen in censuses. In a census, the government collects a series of measurements on all of the country's citizens. After collecting these data, they are summarized. From this descriptive analysis, we learn a lot about a country. For example, you can learn the age distribution of the population by looking at U.S. census data.

Subject	Total
	Estimate
Total population	309,349,689
AGE	
Under 5 years	6.5%
5 to 9 years	6.6%
10 to 14 years	6.7%
15 to 19 years	7.1%
20 to 24 years	7.0%
25 to 29 years	6.8%
30 to 34 years	6.5%
35 to 39 years	6.5%
40 to 44 years	6.8%
45 to 49 years	7.3%
50 to 54 years	7.2%
55 to 59 years	6.4%
60 to 64 years	5.5%
65 to 69 years	4.0%
70 to 74 years	3.0%
75 to 79 years	2.3%
80 to 84 years	1.9%
85 years and over	1.8%

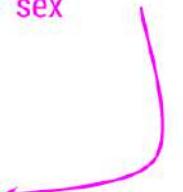
<https://factfinder.census.gov/>

2010 US Census Data
Summary Table
(broken down by age)

This can be further broken down (or stratified) by sex to describe the age distribution by sex. The goal of these analyses is to describe the population. No inferences are made about what this means nor are predictions made about how the data might trend in the future. The point of this (and every!) descriptive analysis is only to summarize the data collected.

Subject	United States		
	Total	Male	Female
	Estimate	Estimate	Estimate
Total population	309,349,689	152,089,450	157,260,239
AGE			
Under 5 years	6.5%	6.8%	6.3%
5 to 9 years	6.6%	6.8%	6.4%
10 to 14 years	6.7%	7.0%	6.4%
15 to 19 years	7.1%	7.5%	6.8%
20 to 24 years	7.0%	7.3%	6.7%
25 to 29 years	6.8%	6.9%	6.6%
30 to 34 years	6.5%	6.6%	6.4%
35 to 39 years	6.5%	6.6%	6.5%
40 to 44 years	6.8%	6.9%	6.7%
45 to 49 years	7.3%	7.3%	7.3%
50 to 54 years	7.2%	7.2%	7.2%
55 to 59 years	6.4%	6.3%	6.5%
60 to 64 years	5.5%	5.4%	5.6%
65 to 69 years	4.0%	3.9%	4.2%
70 to 74 years	3.0%	2.8%	3.2%
75 to 79 years	2.3%	2.1%	2.6%
80 to 84 years	1.9%	1.5%	2.2%
85 years and over	1.8%	1.2%	2.4%

... and
stratified by
sex



<https://factfinder.census.gov/>

Recall that the `glimpse()` function of the `dplyr` package can help you to see what data you are working with.

```
## load packages
library(tidyverse)
df <- msleep # this data comes from ggplot2
## get a glimpse of your data
glimpse(df)
```

```
> glimpse(df)
```

Observations: 83
Variables: 11 size of dataframe

variable names	class of each variable	First few values of each variable
\$ name	<chr>	"Cheetah", "Owl monkey", "Mountain beaver", "G...
\$ genus	<chr>	"Acinonyx", "Aotus", "Aplopontia", "Blarina", ...
\$ vore	<chr>	"carni", "omni", "herbi", "omni", "herbi", "he...
\$ order	<chr>	"Carnivora", "Primates", "Rodentia", "Soricomo...
\$ conservation	<chr>	"lc", NA, "nt", "lc", "domesticated", NA, "vu"...
\$ sleep_total	<dbl>	12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 1...
\$ sleep_rem	<dbl>	NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA,...
\$ sleep_cycle	<dbl>	NA, NA, NA, 0.1333333, 0.6666667, 0.7666667, 0...
\$ awake	<dbl>	11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13...
\$ brainwt	<dbl>	NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA,...
\$ bodywt	<dbl>	50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 2...

Also because the data is in tibble format, we can gain a lot of information by just viewing the data itself.

```
df
```

Here we also get information about the dimensions of our data object and the name and class of our variables.

```
> df
# A tibble: 83 x 11
  name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr> <chr> <chr> <chr> <chr>      <dbl>     <dbl>     <dbl>    <dbl>
1 Chee... Acin... carni Carn... lc       12.1      NA       NA     11.9
2 Owl ... Aotus omni Prim... NA      variable class 17        1.8      NA      7
3 Moun... Aplo... herbi Rode... nt      14.4      2.4      NA     9.6
4 Grea... Blar... omni Sori... lc       14.9      2.3      0.133   9.1
5 Cow    Bos    herbi Arti... domesticated 4        0.7      0.667   20
6 Thre... Brad... herbi Pilo... NA      14.4      2.2      0.767   9.6
7 Nort... Call... carni Carn... vu      8.7       1.4      0.383   15.3
8 Ves... Calo... Rode... NA      7        NA       NA     17
9 Dog    Canis carni Carn... domesticated 10.1      2.9      0.333   13.9
10 Roe ... Capr... herbi Arti... lc      3        NA       NA     21
# ... with 73 more rows, and 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Missing Values

In any analysis, missing data can cause a problem. Thus, it's best to get an understanding of missingness in your data right from the start. Missingness refers to observations that are not included for a variable. In R, NA is the preferred way to specify missing data, so if you're ever generating data, its best to include NA wherever you have a missing value.

However, individuals who are less familiar with R code missingness in a number of different ways in their data: -999, N/A, ., or a blank space. As such, it's best to check to see how missingness is coded in your dataset. A reminder: sometimes different variables within a single dataset will code missingness differently. This shouldn't happen, but it does, so always use caution when looking for missingness.

In this dataset, all missing values are coded as NA, and from the output of `str(df)` (or `glimpse(df)`), we see that at least a few variables have NA values. We'll want to quantify this missingness though to see which variables have missing data and how many observations within each variable have missing data.

To do this, we can write a function that will calculate missingness within each of our variables. To do this we'll combine a few functions. In the code here, `is.na()` returns a logical (TRUE/FALSE) depending upon whether or not the value is missing (TRUE if it is missing). The `sum()` function then calculates the number of TRUE values there are within an observation. We then use `map()` to calculate the number of missing values in each variable. The second bit of code does the exact same thing but divides those numbers by the total number of observations (using `nrow(df)`). For each variable, this returns the proportion of missingness:

```
library(purrr)
## calculate how many NAs there are in each variable
df %>%
  map(is.na) %>%
  map(sum)

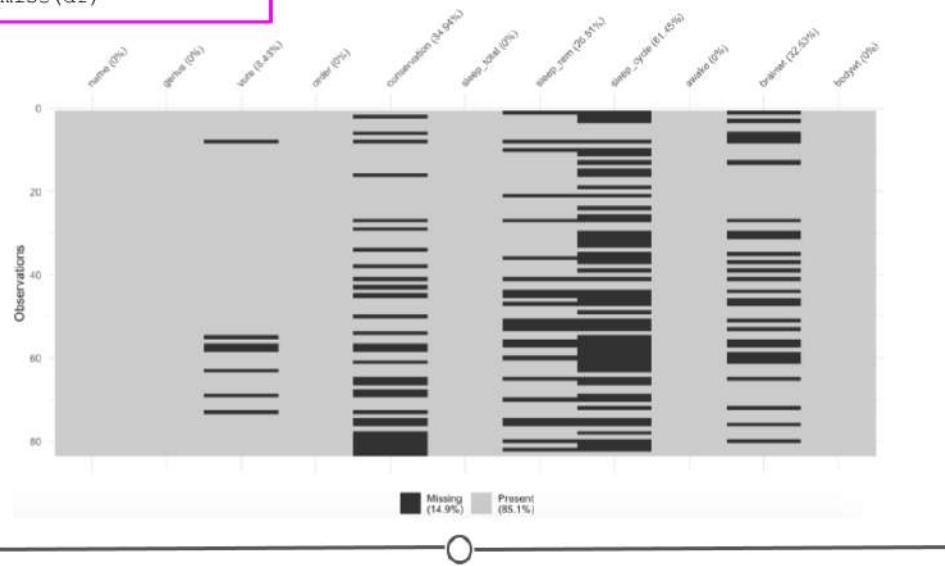
## calculate the proportion of missingness
## for each variable
df %>%
  map(is.na) %>%
  map(sum)%>%
  map(~ . / nrow(df))%>%
  bind_cols()
```

There are also some useful visualization methods for evaluating missingness. You could manually do this with `ggplot2`, but there are two packages called `naniar` and `visdat` written by [Nicholas Tierney](#) that are very helpful. The `visdat` package was used previously in one of our case studies.

```
#install.packages("naniar")
library(naniar)

## visualize missingness
vis_miss(df)
```

```
## visualize missingness
vis_miss(df)
```



Here, we see the variables listed along the top with percentages summarizing how many observations are missing data for that particular variable. Each row in the visualization is a different observation. Missing data are black. Non-missing values are in grey. Focusing again on `brainwt`, we can see the 27 missing values visually. We can also see that `sleep_cycle` has the most missingness, while many variables have no missing data.

Shape

Determining the shape of your variable is essential before any further analysis is done. Statistical methods used for inference often require your data to be distributed in a certain manner before they can be applied to the data. Thus, being able to describe the shape of your variables is necessary during your descriptive analysis.

When talking about the shape of one's data, we're discussing how the values (observations) within the variable are distributed. Often, we first determine how spread out the numbers are from one another (do all the observations fall between 1 and 10? 1 and 1000? -1000 and 10?). This is known as the range of the values. The range is described by the minimum and maximum values taken by observations in the variable.

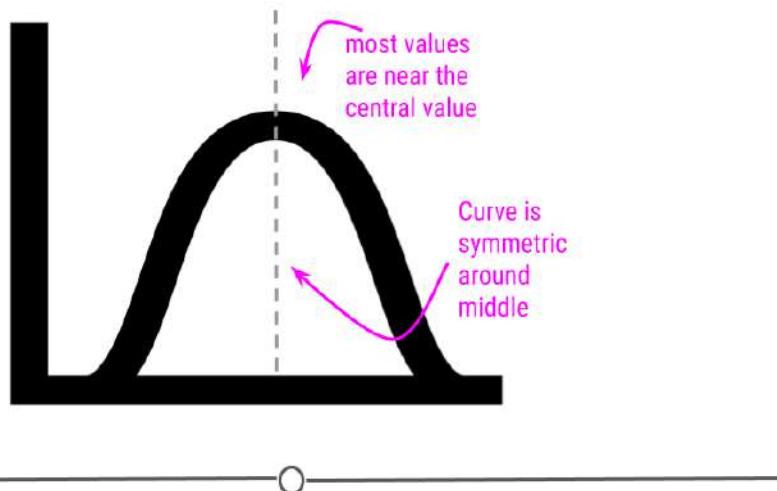
After establishing the range, we determine the shape or distribution of the data. More explicitly, the distribution of the data explains how the data are spread out over this range.

Are most of the values all in the center of this range? Or, are they spread out evenly across the range? There are a number of distributions used commonly in data analysis to describe the values within a variable. We'll cover just a few, but keep in mind this is certainly not an exhaustive list.

Normal Distribution

The Normal distribution (also referred to as the Gaussian distribution) is a very common distribution and is often described as a bell-shaped curve. In this distribution, the values are symmetric around the central value with a high density of the values falling right around the central value. The left hand of the curve mirrors the right hand of the curve.

A Normal Distribution



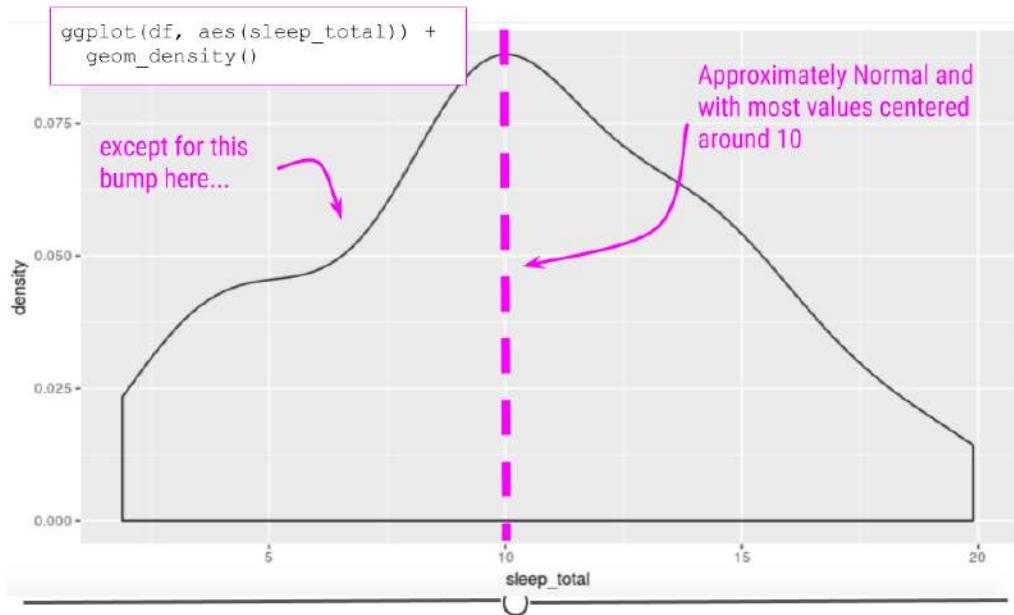
Normal Distribution

A variable can be described as normally distributed if:

- There is a strong tendency for data to take a central value - many of the observations are centered around the middle of the range.
- Deviations away from the central value are equally likely in both directions - the frequency of these deviations away from the central value occurs at the same rate on either side of the central value.

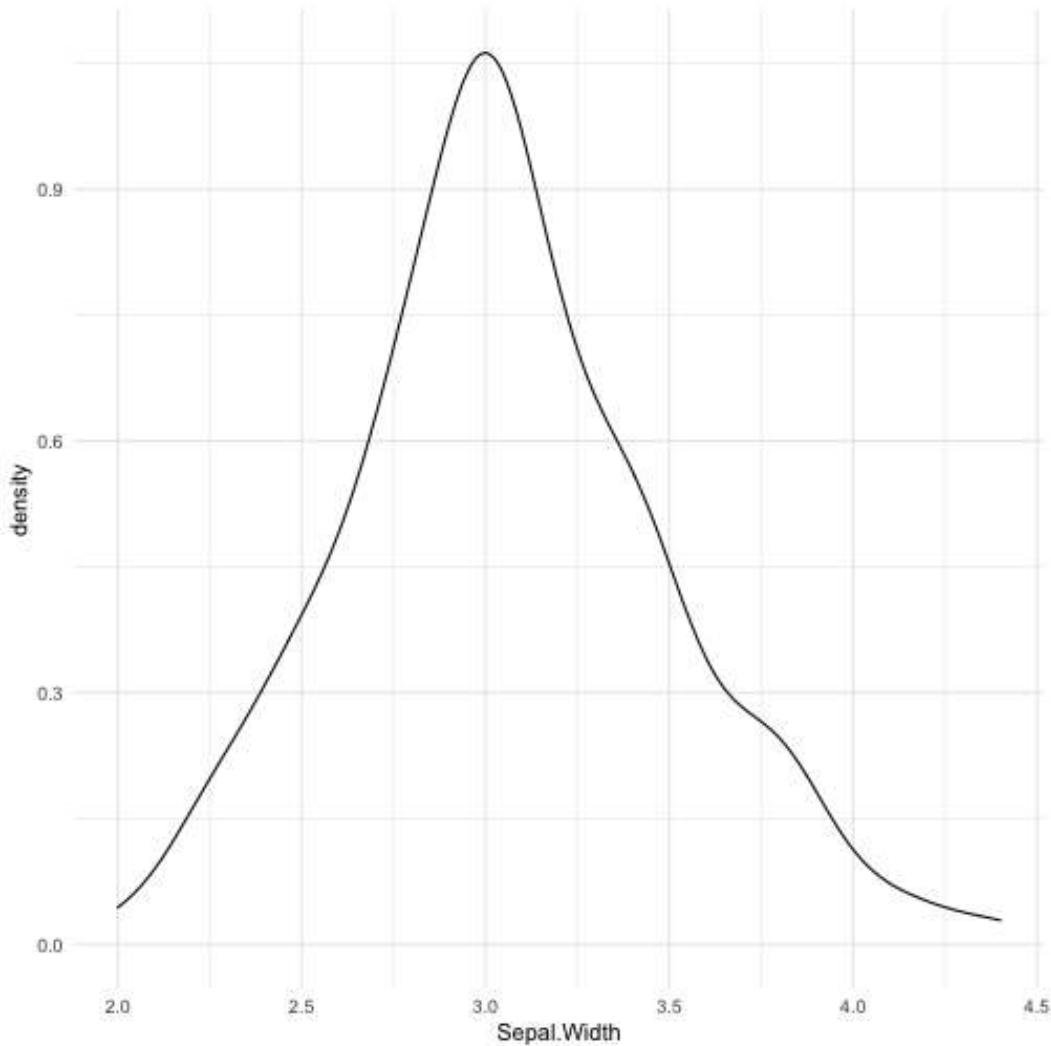
Taking a look at the sleep_total variable within our example dataset, we see that the data are somewhat normal; however, they aren't entirely symmetric.

```
ggplot(df, aes(sleep_total)) +  
  geom_density()
```



A variable that is distributed more normally can be seen in the iris dataset, when looking at the Sepal.Width variable.

```
iris %>%  
  ggplot() +  
  geom_density(aes(x=Sepal.Width))
```

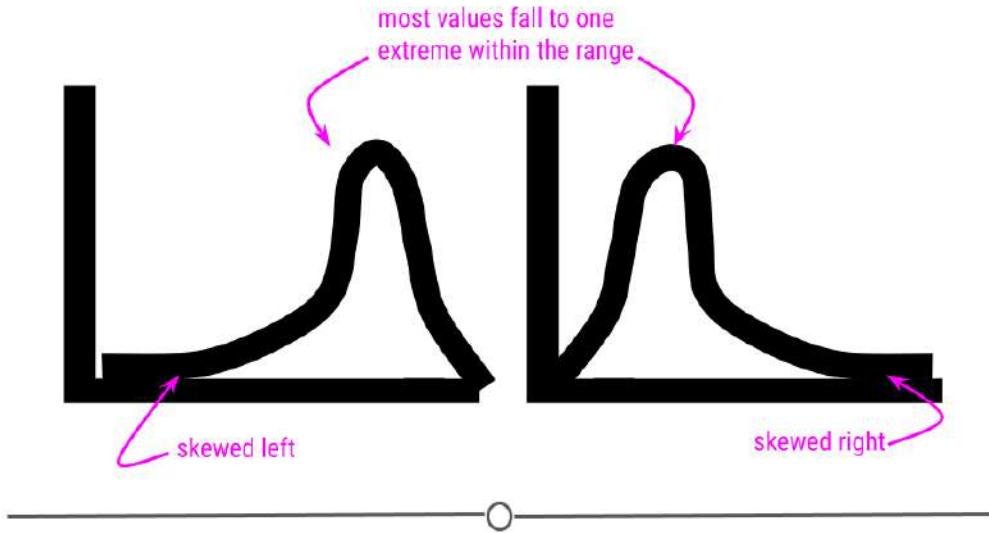


plot of chunk unnamed-chunk-8

Skewed Distribution

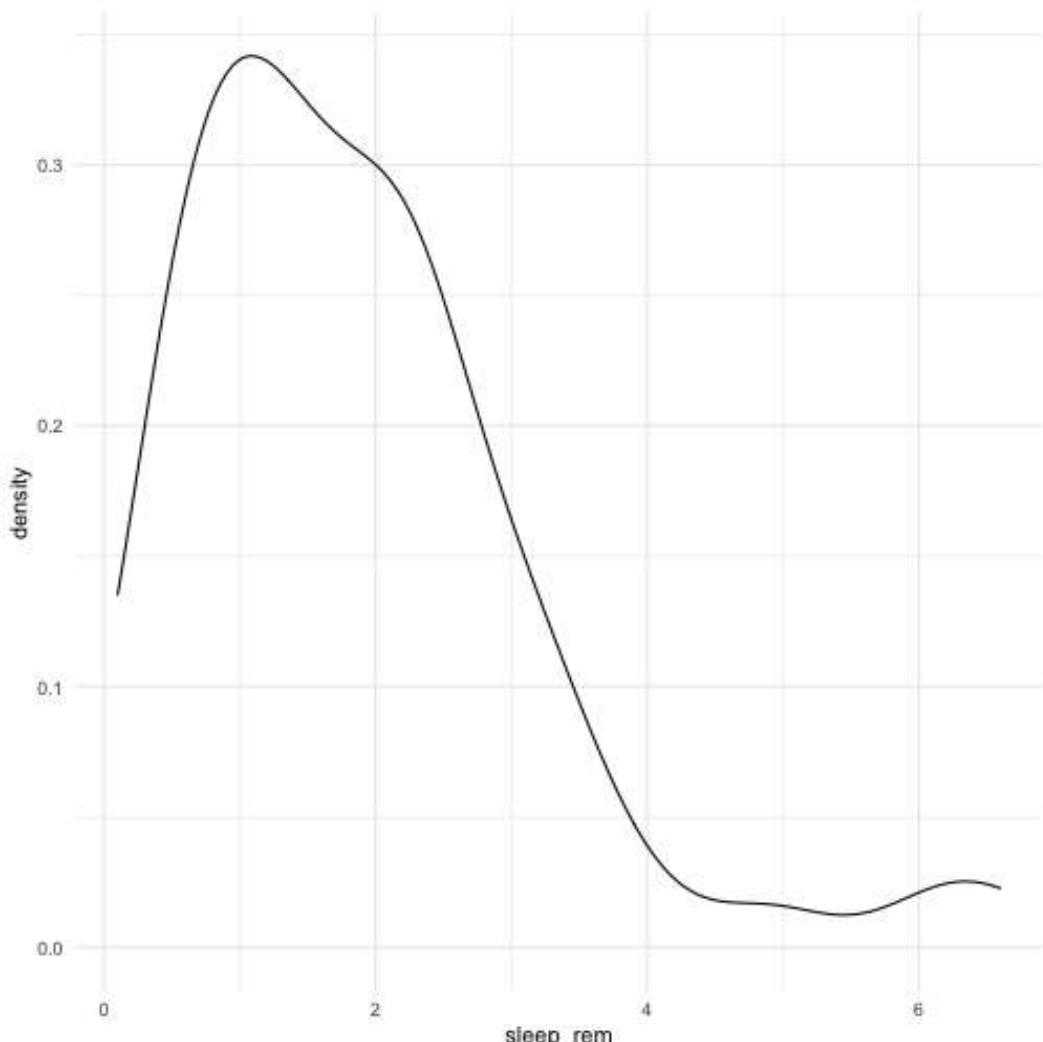
Alternatively, sometimes data follow a skewed distribution. In a skewed distribution, most of the values fall to one end of the range, leaving a tail off to the other side. When the tail is off to the left, the distribution is said to be skewed left. When off to the right, the distribution is said to be skewed right.

A Skewed Distribution



To see an example from the `msleep` dataset, we'll look at the variable `sleep_rem`. Here we see that the data are skewed right, given the shift in values away from the right, leading to a long right tail. Here, most of the values are at the lower end of the range.

```
ggplot(df, aes(sleep_rem)) +  
  geom_density()  
Warning: Removed 22 rows containing non-finite values (stat_density).
```



plot of chunk unnamed-chunk-9

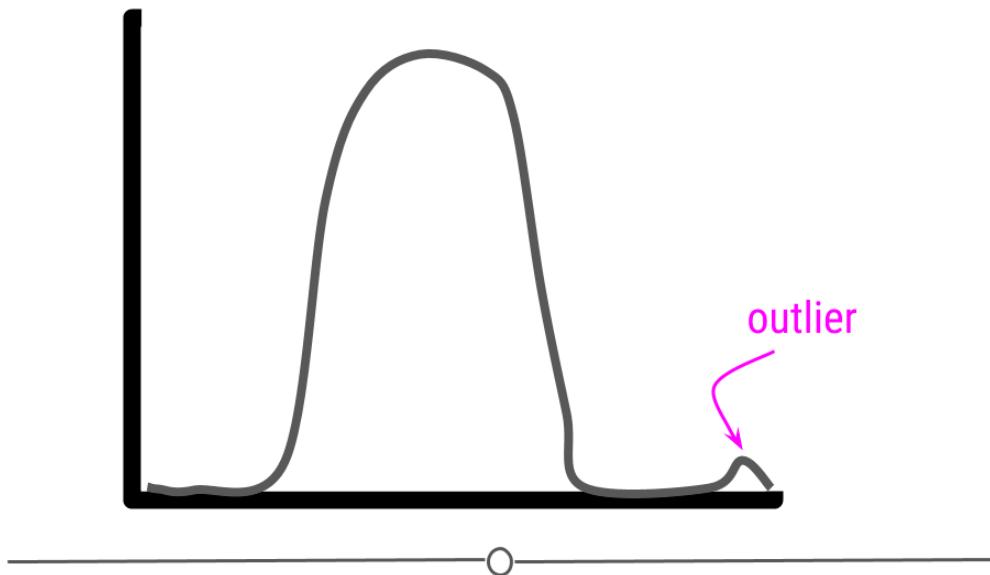
Uniform Distribution

Finally, in distributions we'll discuss today, sometimes values for a variable are equally likely to be found along any portion of the distribution. The curve for this distribution looks more like a rectangle, since the likelihood of an observation taking a value is constant across the range of possible values.

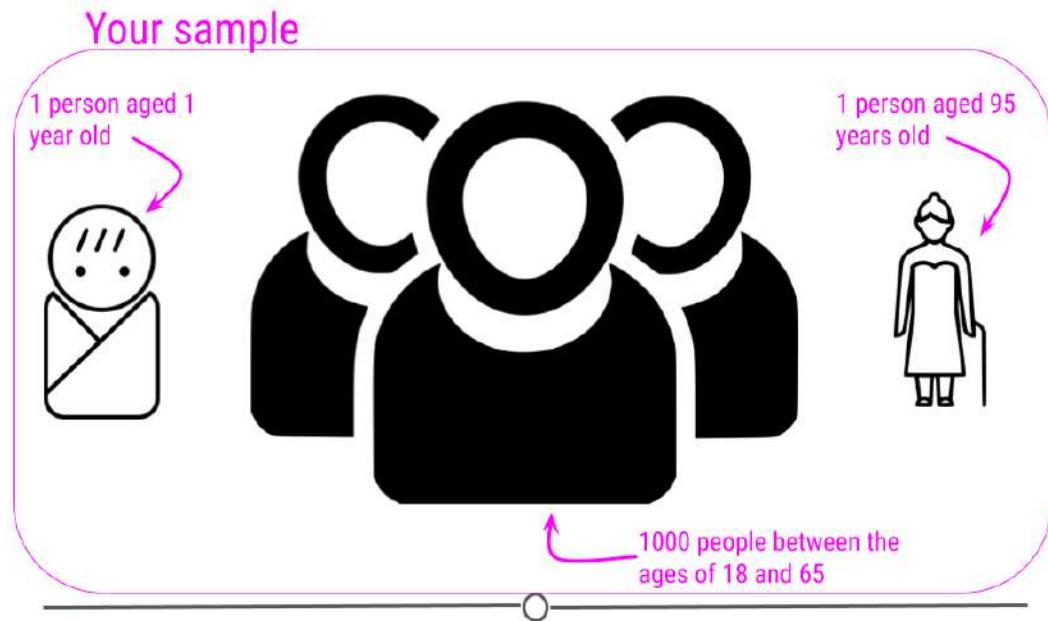
Outliers

Now that we've discussed distributions, it's important to discuss outliers in more depth.

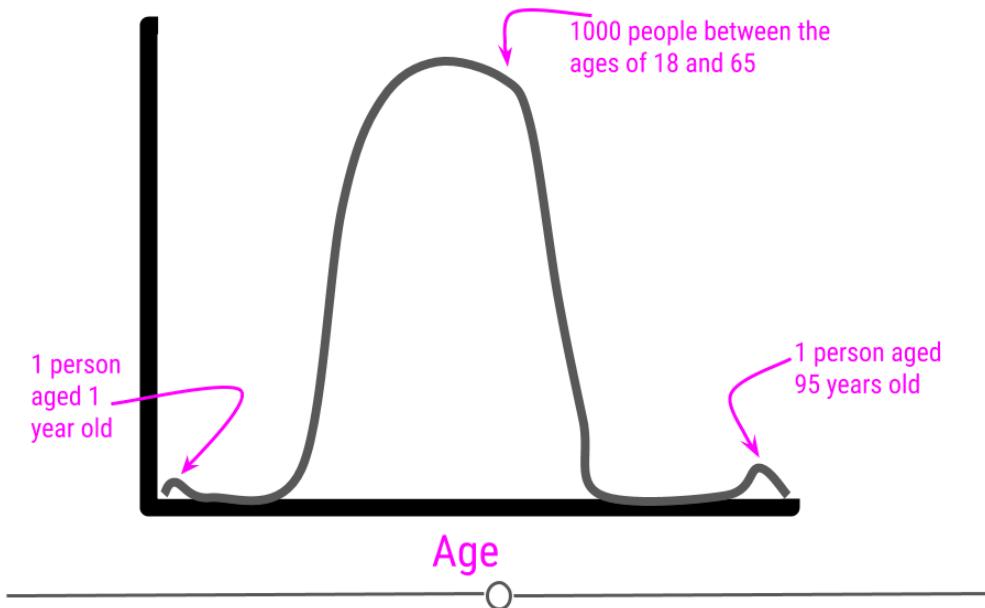
An outlier is an observation that falls far away from the rest of the observations in the distribution. If you were to look at a density curve, you could visually identify outliers as observations that fall far from the rest of the observations.



For example, imagine you had a sample where all of the individuals in your sample are between the ages of 18 and 65, but then you have one sample that is 1 year old and another that is 95 years old.



If we were to plot the age data on a density plot, it would look something like this:



It can sometimes be difficult to decide whether or not a sample should be removed from the dataset. In the simplest terms, no observation should be removed from your dataset unless there is a valid reason to do so. For a more extreme example, what if that dataset we just discussed (with all the samples having ages between 18 and 65) had one sample with the age 600? Well, if these are human data, we clearly know that is a data entry error. Maybe it was supposed to be 60 years old, but we may not know for sure. If we can follow up with that individual and double-check, it's best to do that, correct the error, make a note of it, and continue with the analysis. However, that's often not possible. In the cases of obvious data entry errors, it's likely that you'll have to remove that observation from the dataset. It's valid to do so in this case since you know that an error occurred and that the observation was not accurate.

Outliers do not only occur due to data entry errors. Maybe you were taking weights of your observations over the course of a few weeks. On one of these days, your scale was improperly calibrated, leading to incorrect measurements. In such a case, you would have to remove these incorrect observations before analysis.

Outliers can occur for a variety of reasons. Outliers can occur due to human error during data entry, technical issues with tools used for measurement, as a result of weather changes that affect measurement accuracy, or due to poor sampling procedures. It's **always** important to look at the distribution of your observations for a variable to see if any observations are

falling far away from the rest of the observations. It's then important to think about why this occurred and determine whether or not you have a valid reason to remove the observations from the data.

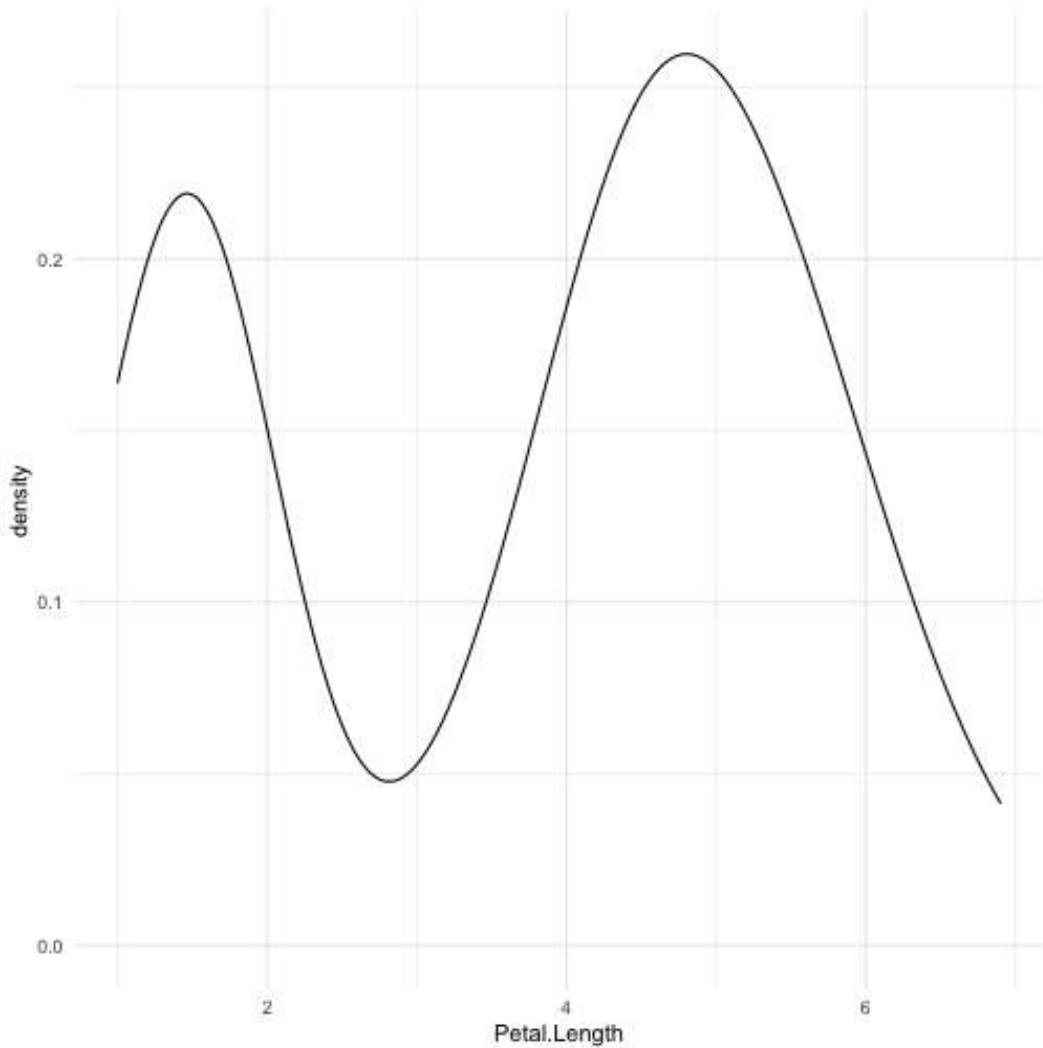
An important note is that observations should never be removed just to make your results look better! Wanting better results is not a valid reason for removing observations from your dataset.

Identifying Outliers

To identify outliers visually, density plots and boxplots can be very helpful.

For example, if we returned to the iris dataset and looked at the distribution of Petal.Length, we would see a bimodal distribution (yet another distribution!). Bimodal distributions can be identified by density plots that have two distinct humps. In these distributions, there are two different modes – this is where the term “bimodal” comes from. In this plot, the curve suggests there are a number of flowers with petal length less than 2 and many with petal length around 5.

```
## density plot
library(ggplot2)
iris %>%
  ggplot(aes(Petal.Length)) +
  geom_density()
```

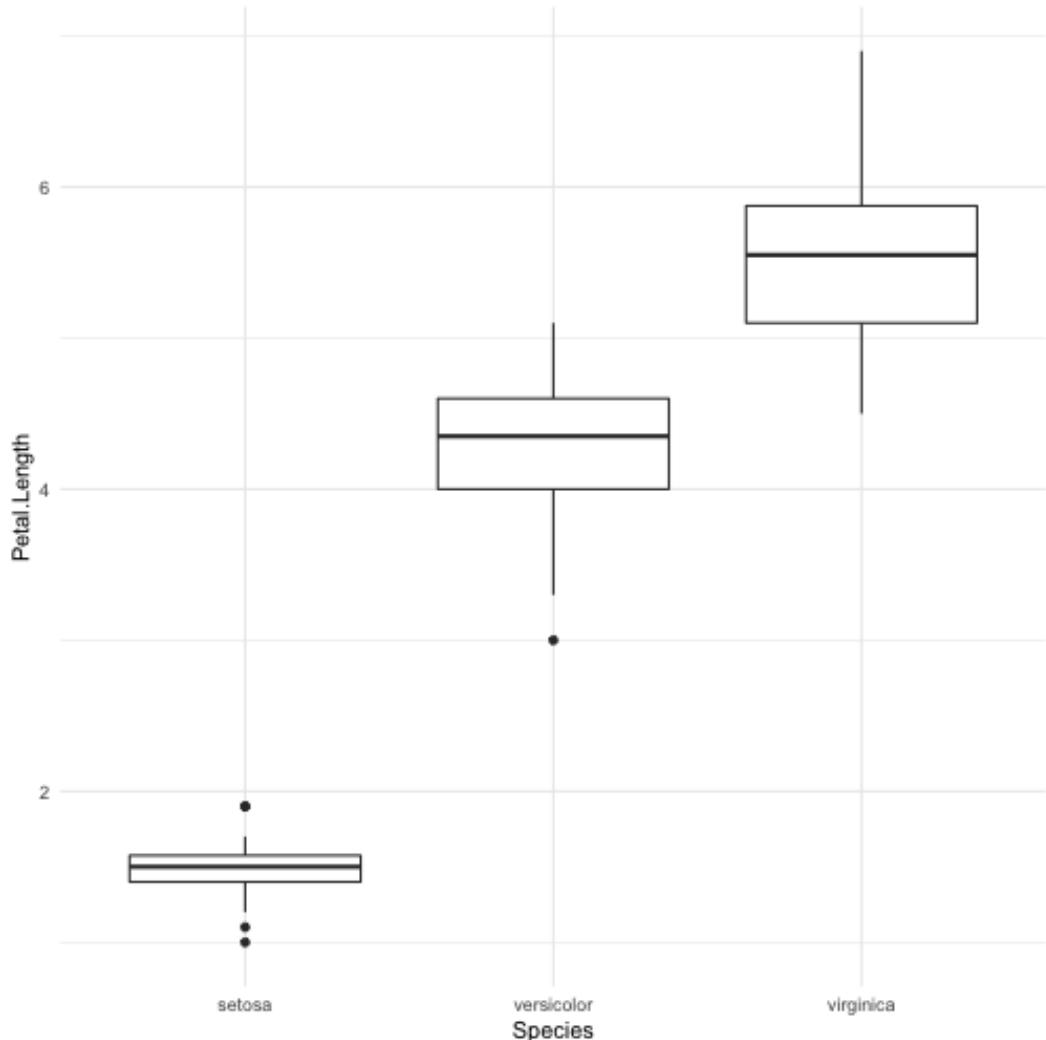


plot of chunk unnamed-chunk-10

Since the two humps in the plot are about the same height, this shows that it's not just one or two flowers with much smaller petal lengths, but rather that there are many. Thus, these observations aren't likely outliers.

To investigate this further, we'll look at petal length broken down by flower species:

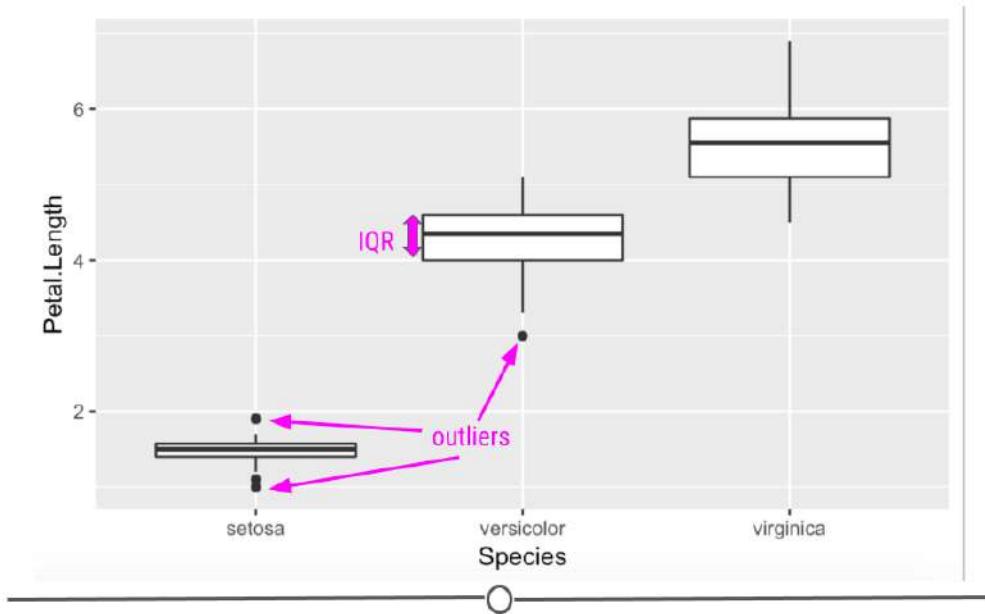
```
## box plot
iris %>%
  ggplot(aes(Species, Petal.Length)) +
  geom_boxplot()
```



plot of chunk unnamed-chunk-11

In this boxplot, we see in fact that setosa have a shorter petal length while virginica have the longest. Had we simply removed all the shorter petal length flowers from our dataset, we would have lost information about an entire species!

Boxplots are also helpful because they plot “outlier” samples as points outside the box. By default, boxplots define “outliers” as observations as those that are $1.5 \times \text{IQR}$ (interquartile range). The IQR is the distance between the first and third quartiles. This is a mathematical way to determine if a sample may be an outlier. It is visually helpful, but then it’s up to the analyst to determine if an observation should be removed. While the boxplot identifies outliers in the setosa and versicolor species, these values are all within a reasonable distance of the rest of the values, and unless I could determine why this occurred, I would not remove these observations from the dataset.



Evaluating Variables

Central Tendency

Once you know how large your dataset is, what variables you have information on, how much missing data you’ve got for each variable, and the shape of your data, you’re ready to start understanding the information within the values of each variable.

Some of the simplest and most informative measures you can calculate on a numeric variable are those of central tendency. The two most commonly used measures of central tendency are: mean and median. These measures provide information about the typical or central value in the variable.

mean

The mean (often referred to as the average) is equal to the sum of all the observations in the variable divided by the total number of observations in the variable. The mean takes all the values in your variable and calculates the most common value.

median

The median is the middle observation for a variable after the observations in that variable have been arranged in order of magnitude (from smallest to largest). The median is the middle value.

Variability

In addition to measures of central tendency, measures of variability are key in describing the values within a variable. Two common and helpful measures of variability are: standard deviation and variance. Both of these are measures of how spread out the values in a variable are.

Variance

The variance tells you how spread out the values are. If all the values within your variable are exactly the same, that variable's variance will be zero. The larger your variance, the more spread out your values are. Take the following vector and calculate its variance in R using the `var()` function:

```
## variance of a vector where all values are the same
a <- c(29, 29, 29, 29)
var(a)
[1] 0

## variance of a vector with one very different value
b <- c(29, 29, 29, 29, 723678)
var(b)
[1] 104733575040
```

The only difference between the two vectors is that the second one has one value that is much larger than “29”. The variance for this vector is thus much higher.

Standard Deviation

By definition, the standard deviation is the square root of the variance, thus if we were to calculate the standard deviation in R using the `sd()` function, we'd see that the `sd()` function is equal to the square root of the variance:

```
## calculate standard deviation
sd(b)
[1] 323625.7

## this is the same as the square root of the variance
sqrt(var(b))
[1] 323625.7
```

For both measures of variance, the minimum value is 0. The larger the number, the more spread out the values in the variable are.

Summarizing Your Data

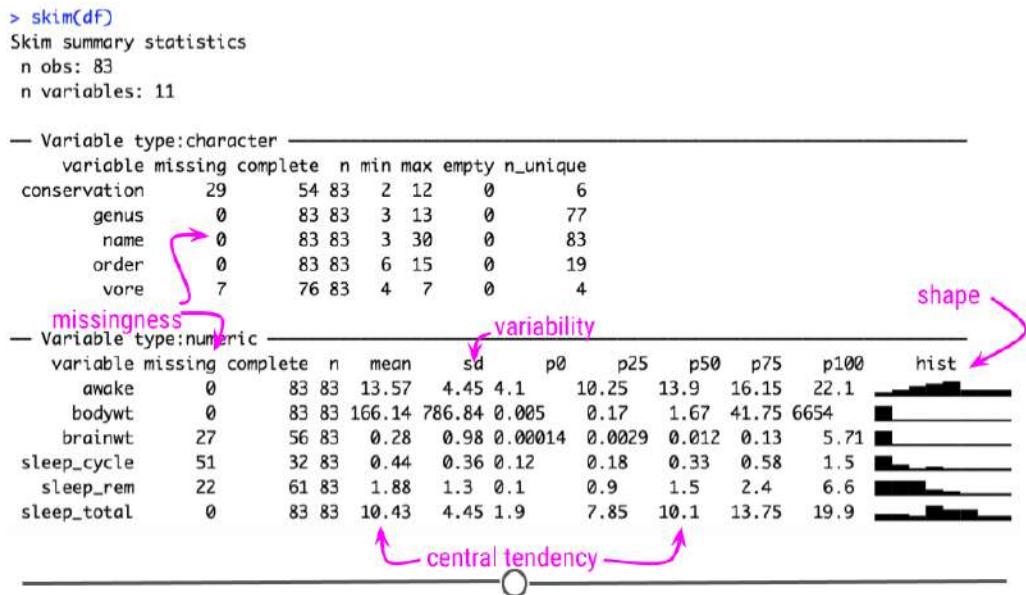
Often, you'll want to include tables in your reports summarizing your dataset. These will include the number of observations in your dataset and maybe the mean/median and standard deviation of a few variables. These could be organized into a table using what you learned in the data visualization course about generating tables.

skimr

Alternatively, there is a helpful package that will summarize all the variables within your dataset. The `skimr` package provides a tidy output with information about your dataset.

To use `skimr`, you'll have to install and load the package before using the helpful function `skim()` to get a snapshot of your dataset.

```
#install.packages("skimr")
library(skimr)
skim(df)
```



Using this function we can quickly get an idea about missingness, variability, central tendency, and shape, and outliers all at once.

The output from `skim()` separately summarizes categorical and continuous variables. For continuous variables you get information about the mean and median (`p50`) column. You know what the range of the variable is (`p0` is the minimum value, `p100` is the maximum value for continuous variables). You also get a measure of variability with the standard deviation (`sd`). It even quantifies the number of missing values (`missing`) and shows you the distribution or shape of each variable (`hist`)! Potential outliers can also be identified from the `hist` column and the `p100` and `p0` columns. This function can be incredibly useful to get a quick snapshot of what's going on with your dataset.

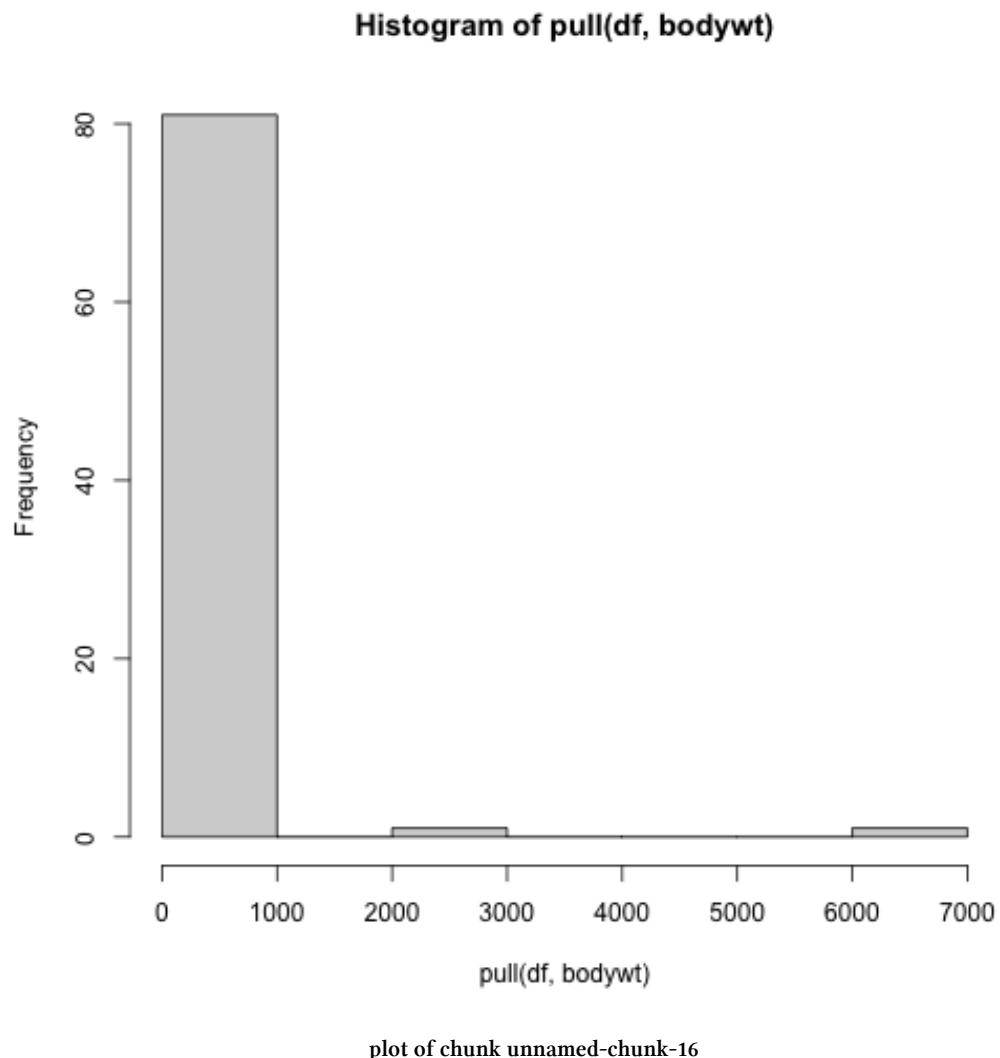
If we take a look closer at the `bodywt` and `brainwt` variables, we can see that there may be outliers. The maximum value of the `bodywt` variable looks very different from the `mean` value.

```
dplyr::filter(df, bodywt == 6654)
# A tibble: 1 × 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>          <dbl>      <dbl>      <dbl> <dbl>
1 Africa... Loxo... herbi Prob... vu           3.3        NA        NA     20.7
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Ah! looks like it is an elephant, that makes sense.

Taking a deeper look at the histogram we can see that there are two values that are especially different.

```
hist(pull(df, bodywt))
```



```
dplyr::filter(df, bodywt > 2000)
# A tibble: 2 × 11
  name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>  <chr> <chr> <chr> <chr>          <dbl>      <dbl>      <dbl>    <dbl>
1 Asian ... Elep... herbi Prob... en        3.9       NA        NA     20.1
2 Africa... Loxo... herbi Prob... vu        3.3       NA        NA     20.7
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Looks like both data points are for elephants.

Therefore, we might consider performing an analysis both with and without the elephant data, to see if it influences the overall result.

Evaluating Relationships

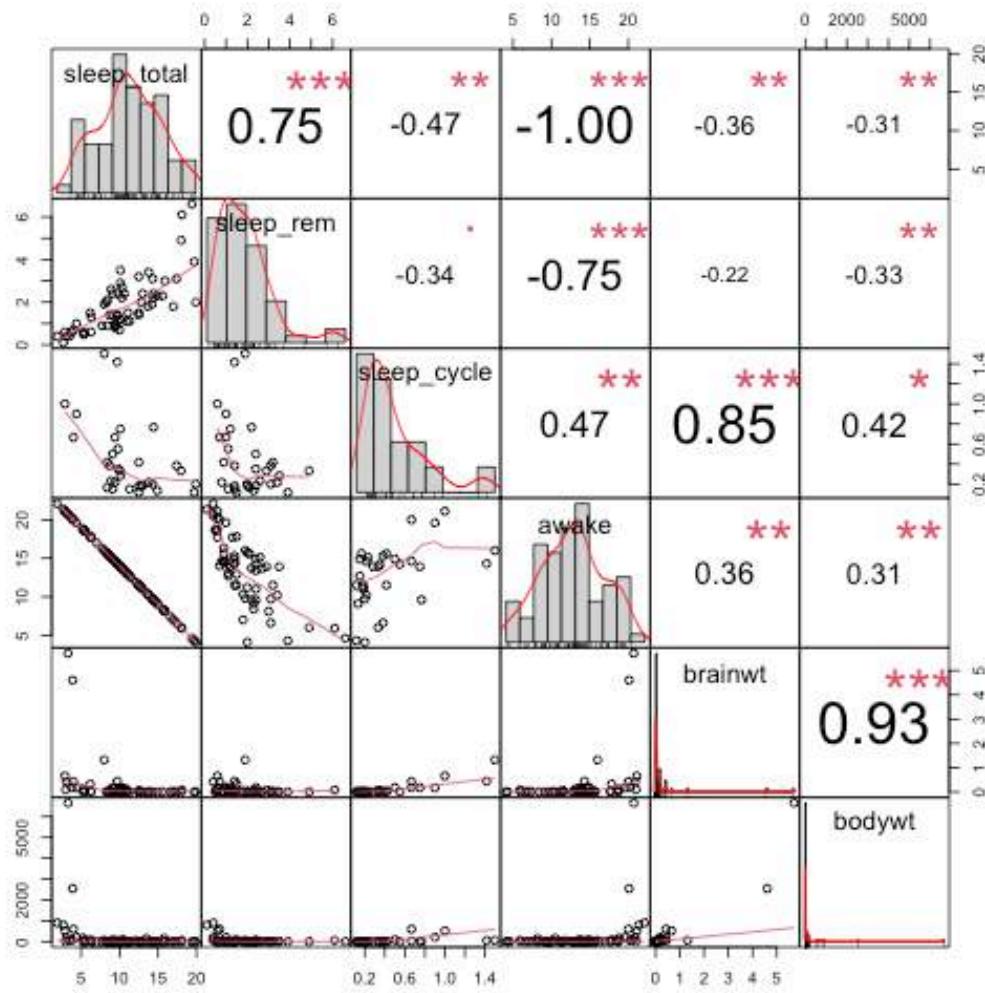
Another important aspect of exploratory analysis is looking at relationships between variables.

Again visualizations can be very helpful.

We might want to look at the relationships between all of our continuous variables. A good way to do this is to use a visualization of **correlation**. As a reminder, correlation is a measure of the relationship or interdependence of two variables. In other words, how much do the values of one variable change with the values of another. Correlation can be either positive or negative and it ranges from -1 to 1, with 1 and -1 indicating perfect correlation (1 being positive and -1 being negative) and 0 indicating no correlation. We will describe this in greater detail when we look at associations.

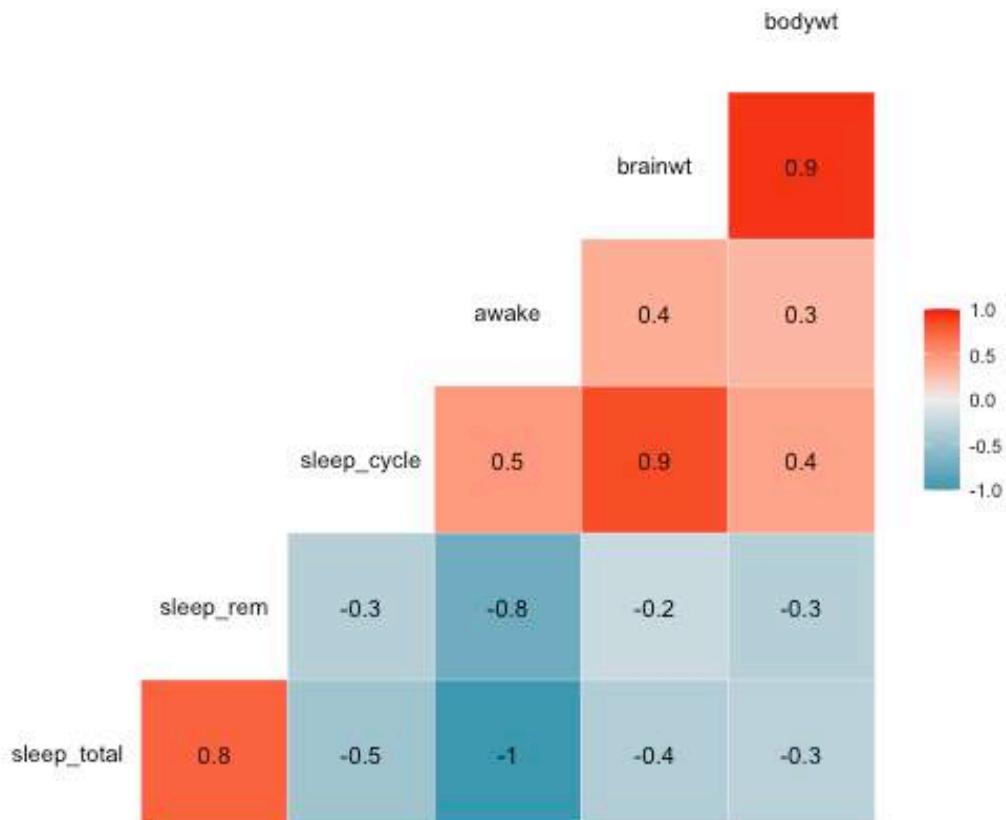
Here are some very useful plots that can be generated using the `GGally` package and the `PerformanceAnalytics` package to examine if variables are correlated.

```
library(PerformanceAnalytics)
df %>%
dplyr::select_if(is.numeric) %>%
  chart.Correlation()
```



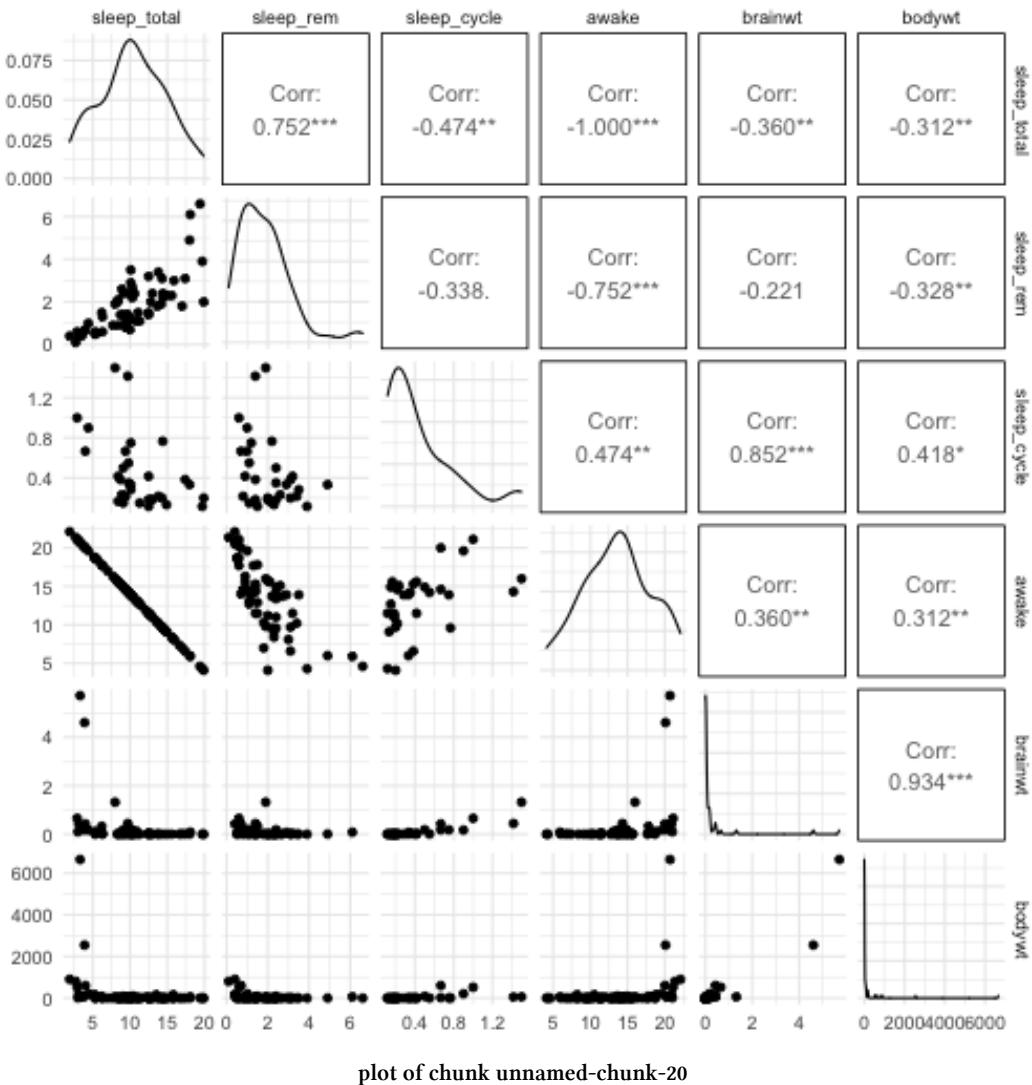
plot of chunk unnamed-chunk-18

```
library(GGally)
Registered S3 method overwritten by 'GGally':
  method  from
  +.gg    ggplot2
df %>%
dplyr::select_if(is.numeric) %>%
  ggcorm(label = TRUE)
```



plot of chunk unnamed-chunk-19

```
library(GGally)
df %>%
dplyr::select_if(is.numeric) %>%
  ggpairs()
```

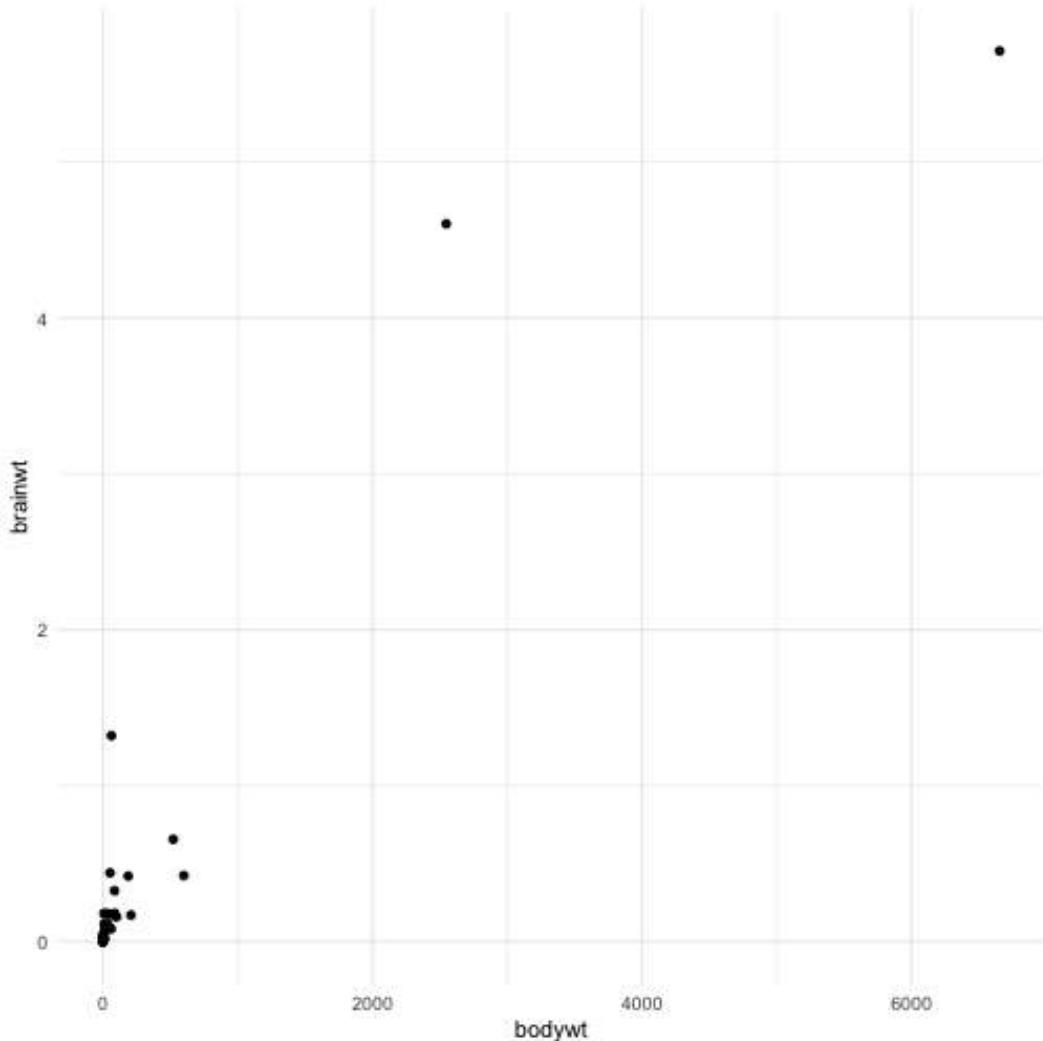


We can see from these plots that the `awake` variable and the `sleep_total` variable are perfectly correlated. This becomes important for choosing what to include in models when we try to perform prediction or inference analyses.

We may be especially interested in how brain weight (`brain_wt`) relates to body weight (`body_wt`). We might assume that these two variables might be related to one another.

Here is a plot of these two variables including the elephant data:

```
library(ggplot2)
ggplot(df, aes(x = bodywt, y = brainwt)) +
  geom_point()
```

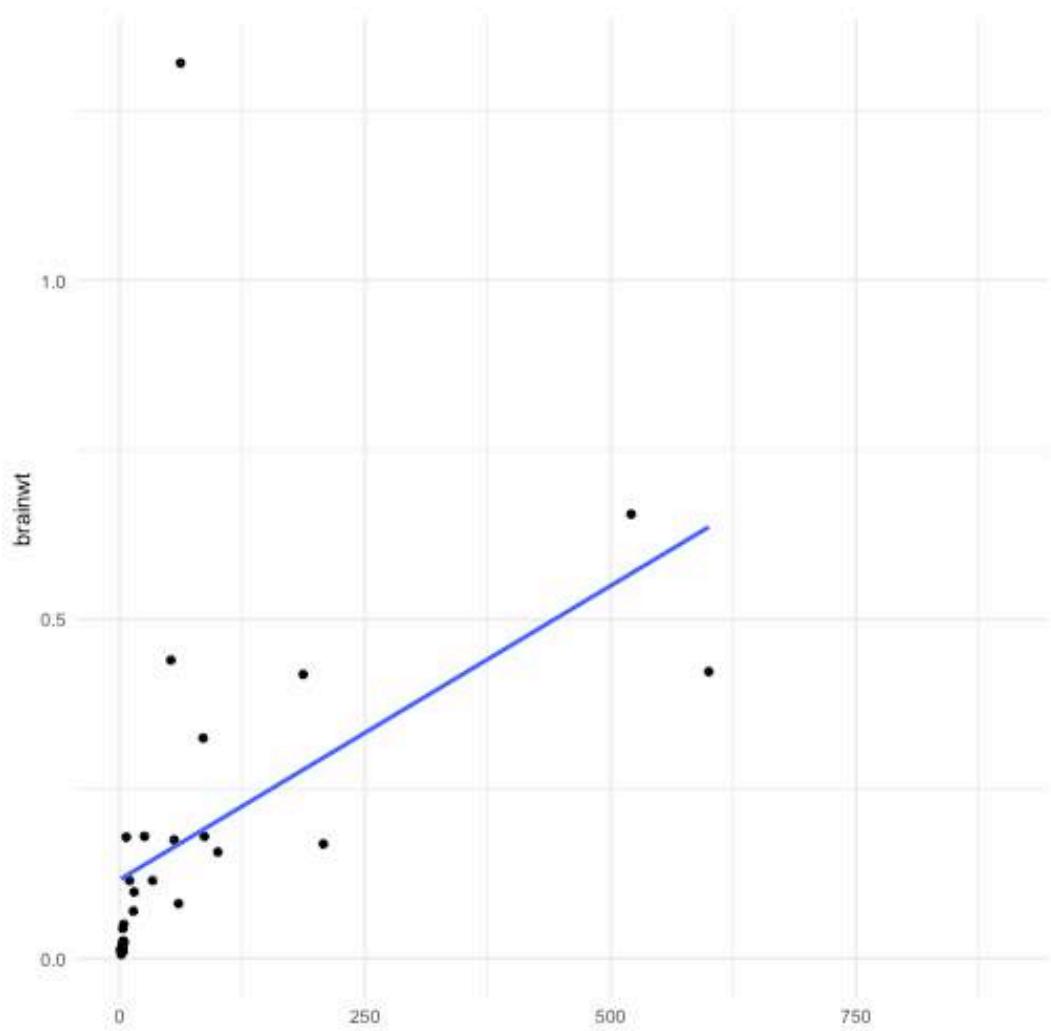


plot of chunk unnamed-chunk-21

Clearly, including the elephant data points makes it hard to look at the other data points, it is possible that these points are driving the positive correlation that we get when we use all the data. Here is a plot of the relationship between these two variables excluding the elephant data and the very low body weight organisms:

```
library(ggplot2)

df %>%
  filter(bodywt<2000 & bodywt >1) %>%
  ggplot(aes(x = bodywt, y = brainwt)) +
  geom_point()+
  geom_smooth(method = "lm", se = FALSE)
`geom_smooth()` using formula 'y ~ x'
```



```
cor.test(pull(df %>% filter(bodywt<2000 & bodywt >1),bodywt),
         pull(df %>%filter(bodywt<2000 & bodywt >1),brainwt))

Pearson's product-moment correlation

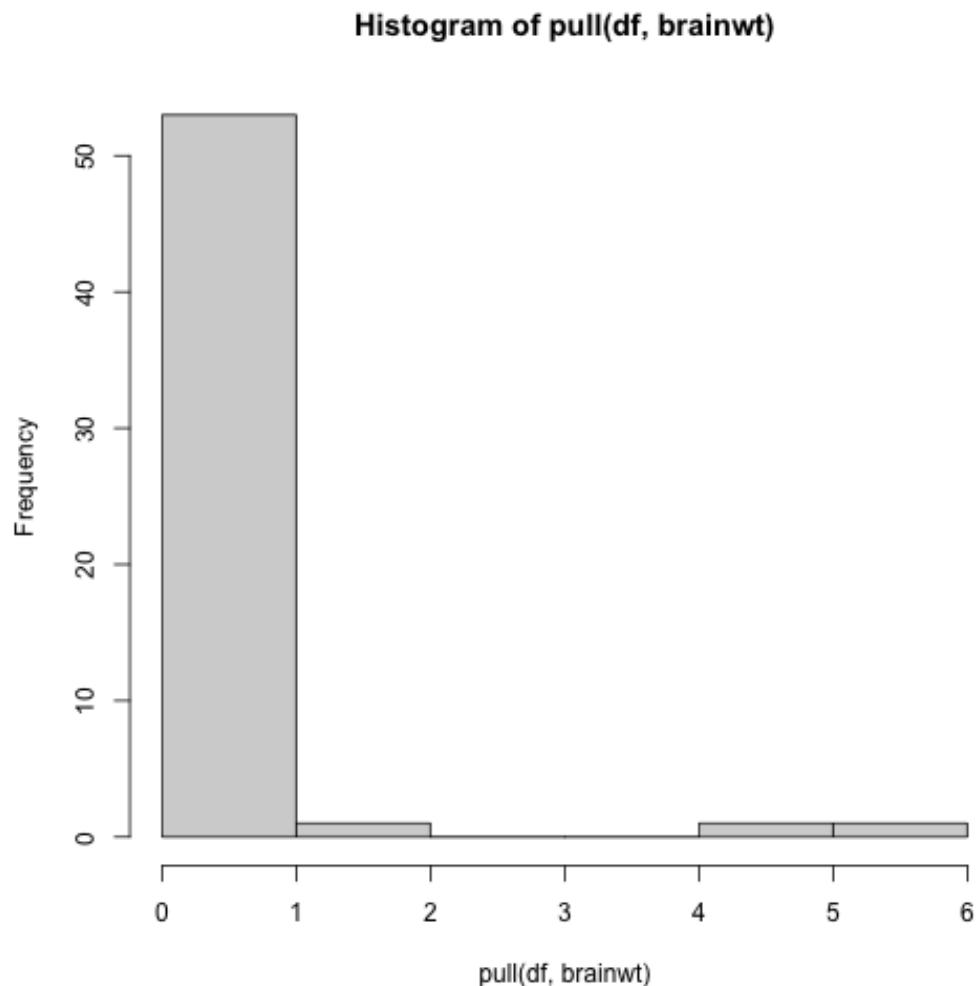
data: pull(df %>% filter(bodywt < 2000 & bodywt > 1), bodywt) and pull(df %>% \
filter(bodywt < 2000 & bodywt > 1), brainwt)
t = 2.7461, df = 28, p-value = 0.01042
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1203262 0.7040582
sample estimates:
cor
0.4606273
```

We can see from this plot that in general `brainwt` is correlated with `bodywt`. Or in other words, `brainwt` tends to increase with `bodywt`.

But it also looks like we have an outlier for our `brainwt` variable! There is a very high `brainwt` value that is greater than 1.

We can also see it in our histogram of this variable:

```
hist(pull(df, brainwt))
```



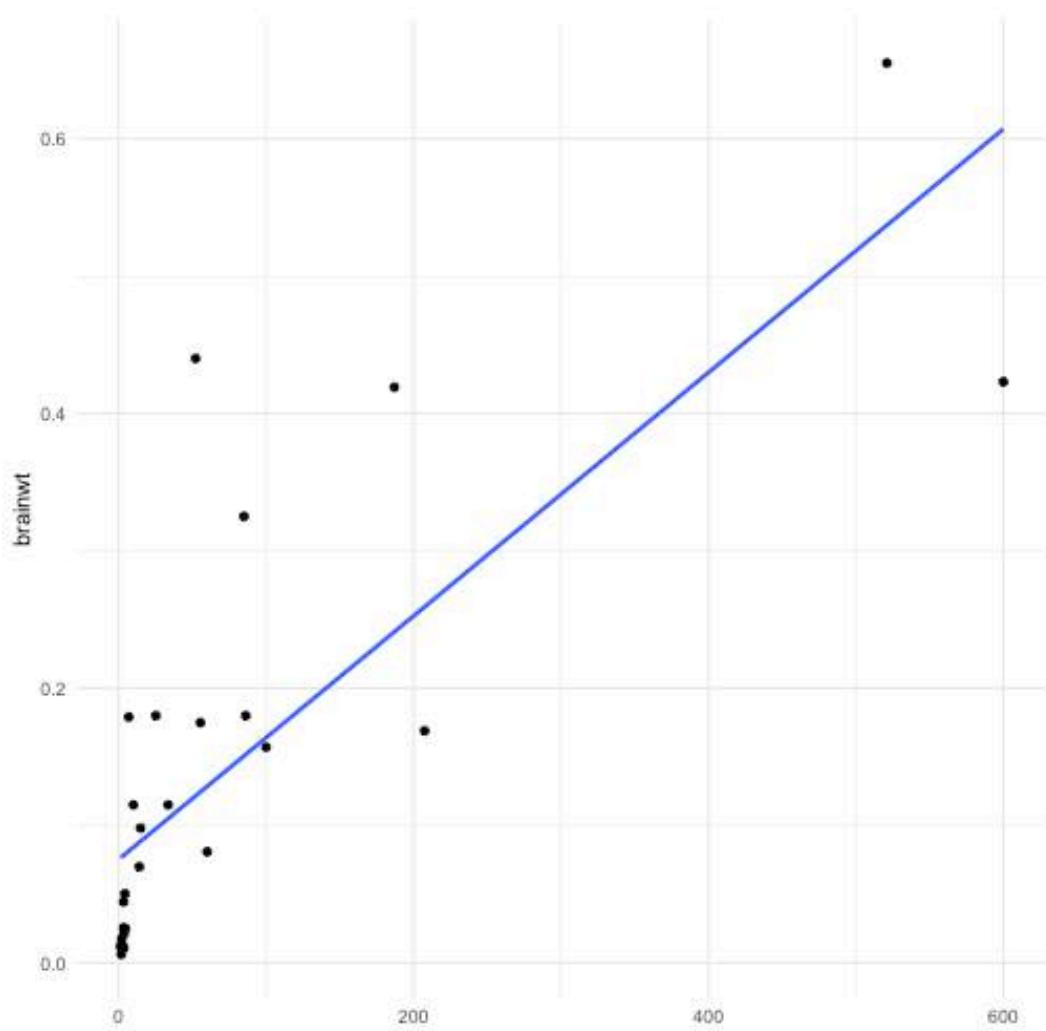
plot of chunk unnamed-chunk-23

Let's see which organism this is:

```
df %>%
  filter(brainwt >=1)
# A tibble: 3 × 11
  name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>  <chr> <chr> <chr> <chr>      <dbl>     <dbl>     <dbl> <dbl>
1 Asian ... Elep... herbi  Prob... en        3.9       NA       NA    20.1
2 Human   Homo  omni   Prim... <NA>        8         1.9      1.5    16
3 Africa... Loxo... herbi  Prob... vu        3.3       NA       NA    20.7
# ... with 2 more variables: brainwt <dbl>, bodywt <dbl>
```

It is humans! Let's see what the plot looks like without humans:

```
library(ggplot2)
df %>%
  filter(bodywt<2000 & bodywt >1 & brainwt<1) %>%
  ggplot(aes(x = bodywt, y = brainwt)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-25

```
cor.test(pull(df %>% filter(bodywt<2000 & bodywt >1 & brainwt<1),bodywt),
         pull(df %>%filter(bodywt<2000 & bodywt >1 & brainwt<1),brainwt))

Pearson's product-moment correlation

data: pull(df %>% filter(bodywt < 2000 & bodywt > 1 & brainwt < 1), bodywt) an\
d pull(df %>% filter(bodywt < 2000 & bodywt > 1 & brainwt < 1), brainwt)
t = 6.6127, df = 27, p-value = 0.0000004283
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.5897381 0.8949042
sample estimates:
cor
0.7862926
```

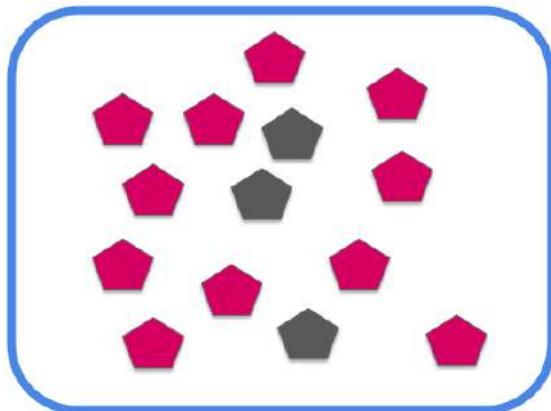
We can see from these plots that the `brainwt` variable seems to have a relationship (correlation value = 0.79) with `bodywt` and it increases with the `bodywt` variable, however this relationship is less strong when humans are included (correlation value = 0.46). This information would be important to keep in mind when trying to model this data to make inference or predictions about the animals included.

Inference

Inferential Analysis is what analysts carry out after they've described and explored their dataset. After understanding your dataset better, analysts often try to infer something from the data. This is done using **statistical tests**.

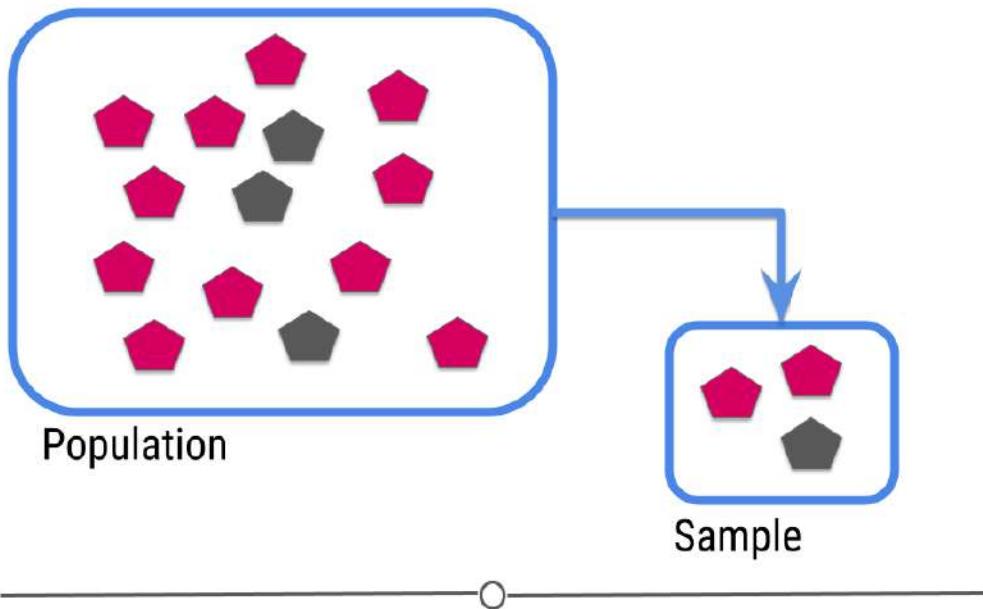
We discussed a bit about how we can use models to perform inference and prediction analyses. What does this mean?

The goal of inferential analyses is to use a relatively **small** sample of data to **infer** or say something about the **population at large**. This is required because often we want to answer questions about a population. Let's take a dummy example here where we have a population of 14 shapes. Here, in this graphic, the shapes represent individuals in the population and the colors of the shapes can be either pink or grey:

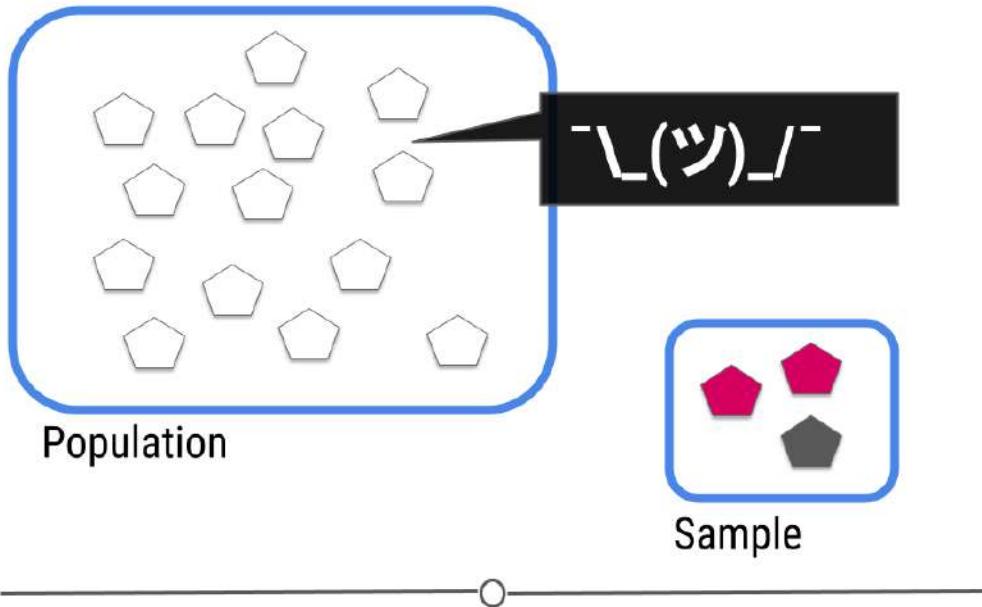


Population

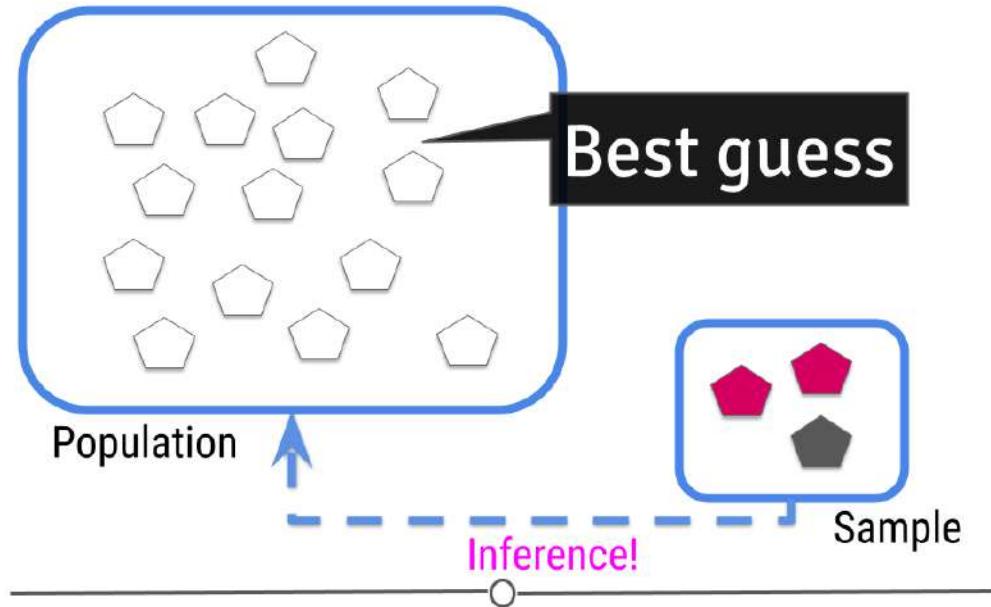
In this example we only have fourteen shapes in the population; however, in inferential data analysis, it's not usually possible to sample *everyone* in the population. Consider if this population were everyone in the United States or every college student in the world. As getting information from every individual would be infeasible. Data are instead collected on a subset, or a **sample** of the individuals in the larger population.



In our example, we've been showing you how many pink and how many gray shapes are in the larger population. However, in real life, we don't know what the answer is in the larger population. That's why we collected the sample!

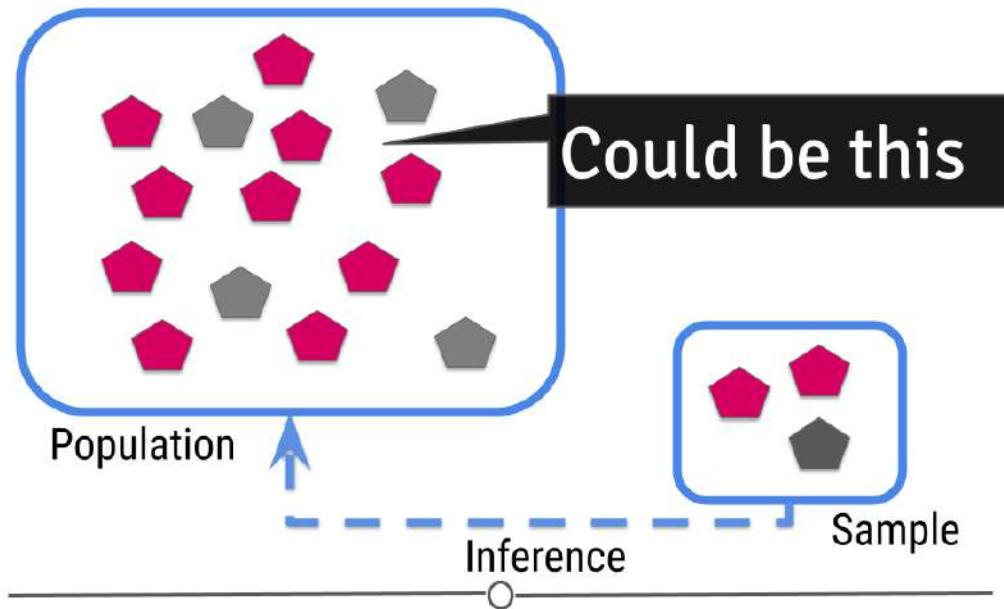


This is where **inference** comes into play. We analyze the data collected in our sample and then do our best to infer what the answer is in the larger population. In other words, inferential data analysis uses data from a sample to make its best guess as to what the answer would be in the population if we were able to measure every individual.

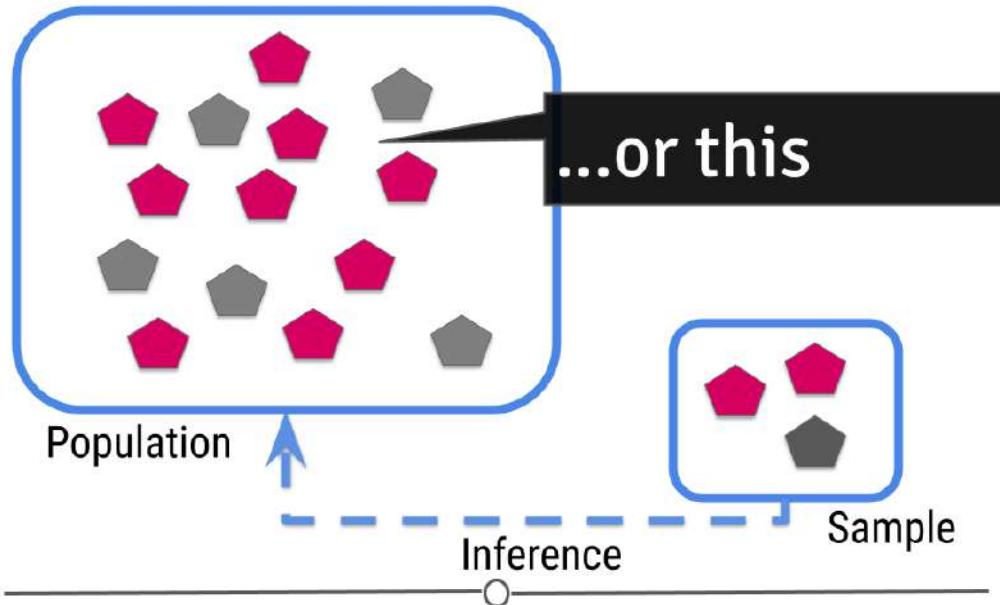


Uncertainty

Because we haven't directly measured the population but have only been able to take measurements on a sample of the data, when making our inference we can't be exactly sure that our inference about the population is exact. For example, in our sample one-third of the shapes are grey. We'd expect about one-third of the shapes in our population to be grey then too! Well, one-third of 14 (the number of shapes in our population) is 4.667. Does this mean four shapes are truly gray?

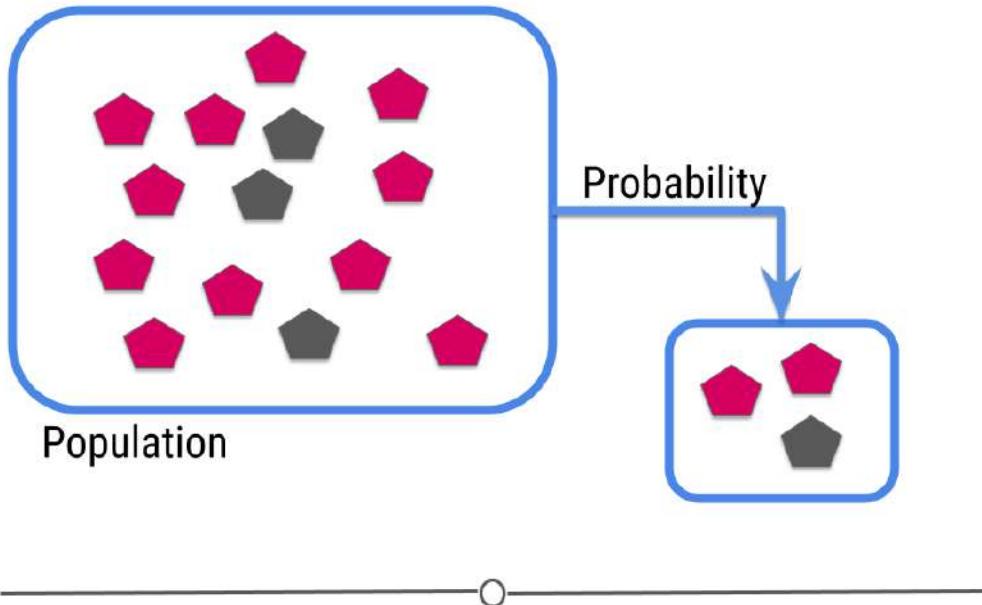


Or maybe five shapes in the population are grey?



Given the sample we've taken, we can guess that 4-5 shapes in our population will be grey, but we aren't certain exactly what that number is. In statistics, this "best guess" is known as an **estimate**. This means that we estimate that 4.667 shapes will be gray. But, there is uncertainty in that number. Because we're taking our best guess at figuring out what that estimate should be, there's also a measure of uncertainty in that estimate. Inferential data analysis includes **generating the estimate and the measure of uncertainty around that estimate**.

Let's return back to the example where we *know* the truth in the population. Hey look! There were actually only three grey shapes after all. It is totally possible that if you put all those shapes into a bag and pulled three out that two would be pink and one would be grey. As statisticians, we'd say that getting this sample was **probable** (it's within the realm of possibility). This really drives home why it's important to add uncertainty to your estimate whenever you're doing inferential analysis!

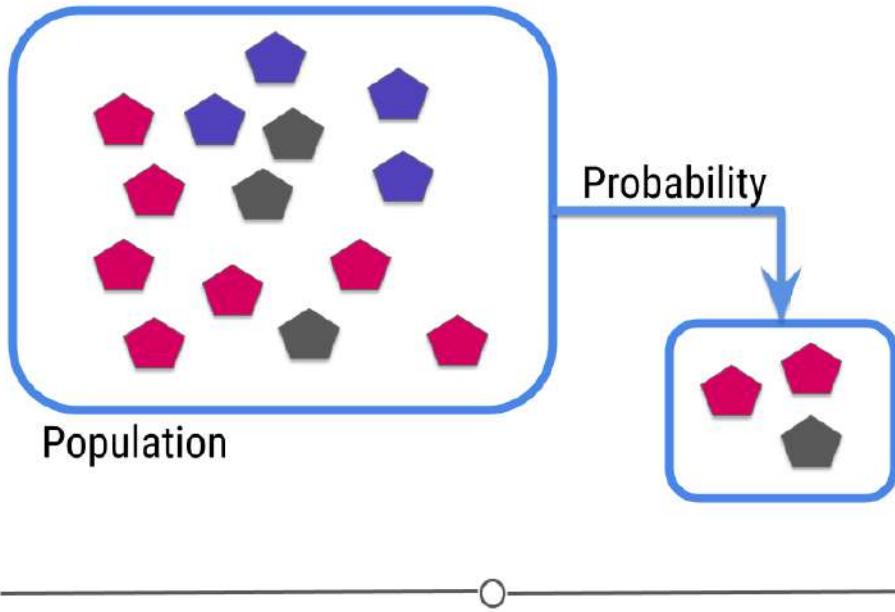


Random Sampling

Since you are moving from a *small* amount of data and trying to generalize to a *larger* population, your ability to accurately infer information about the larger population depends heavily on how the data were sampled.

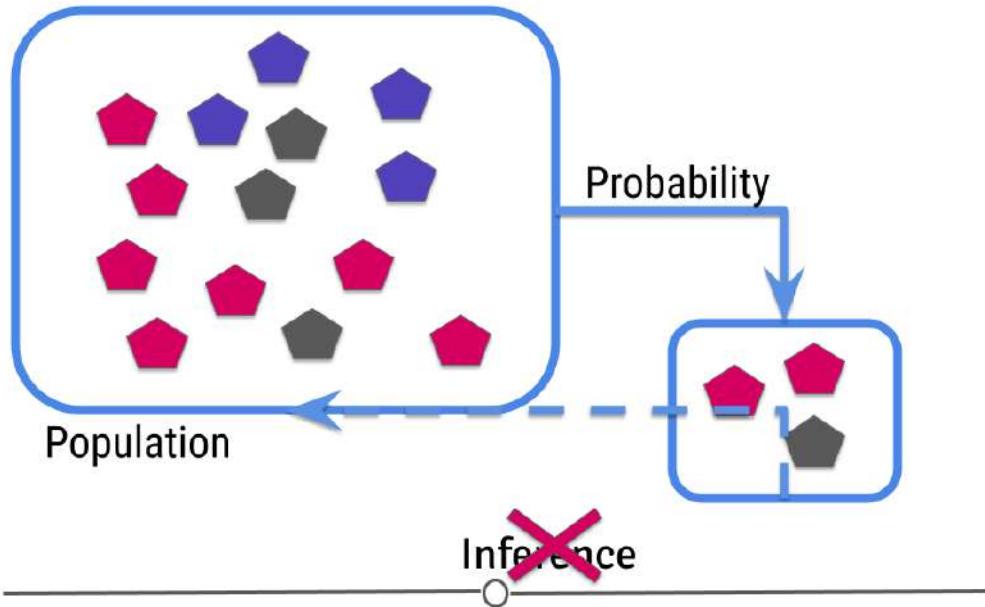
The data in your sample *must* be representative of your larger population to be used for inferential data analysis. Let's discuss what this means.

Using the same example, what if, in your larger population, you didn't just have grey and pink shapes, but you also had blue shapes?



Well, if your sample only has pink and grey shapes, when you go to make an inference, there's no way you'd infer that there should be blue shapes in your population since you didn't capture any in your sample.

In this case, your sample is *not* representative of your larger population. In cases where you do not have a representative sample, you can not carry out inference, since you will not be able to correctly infer information about the larger population.



This means that you have to design your analysis so that you're collecting representative data and that you have to check your data after data collection to make sure that you were successful.

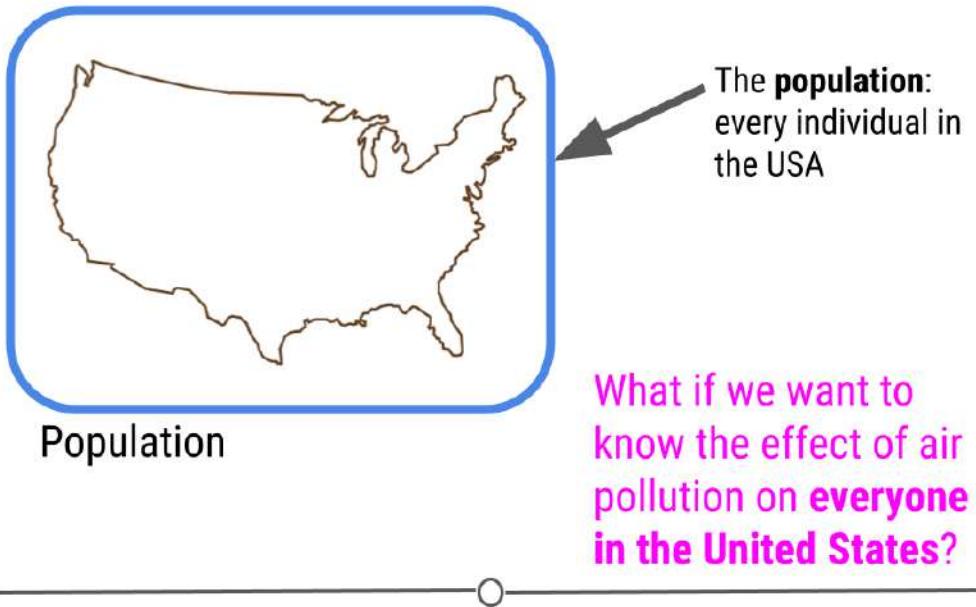
You may at this point be thinking to yourself. "Wait a second. I thought I didn't know what the truth was in the population. How can I make sure it's representative?" Good point! With regards to the measurement you're making (color distribution of the shapes, in this example), you don't know the truth. But, you should know other information about the population. What is the age distribution of your population? Your sample should have a similar age distribution. What proportion of your population is female? If it's half, then your sample should be comprised of half females. Your data collection procedure should be set up to ensure that the sample you collect is representative (very similar to) your larger population. Then, once the data are collected, your descriptive analysis should check to ensure that the data you've collected are in fact representative of your larger population. Randomly sampling your larger population helps ensure that the inference you make about the measurement of interest (color distribution of the shapes) will be the most accurate.

To reiterate: If the data you collect is not from a representative sample of the population, the generalizations you infer won't be accurate for the population.

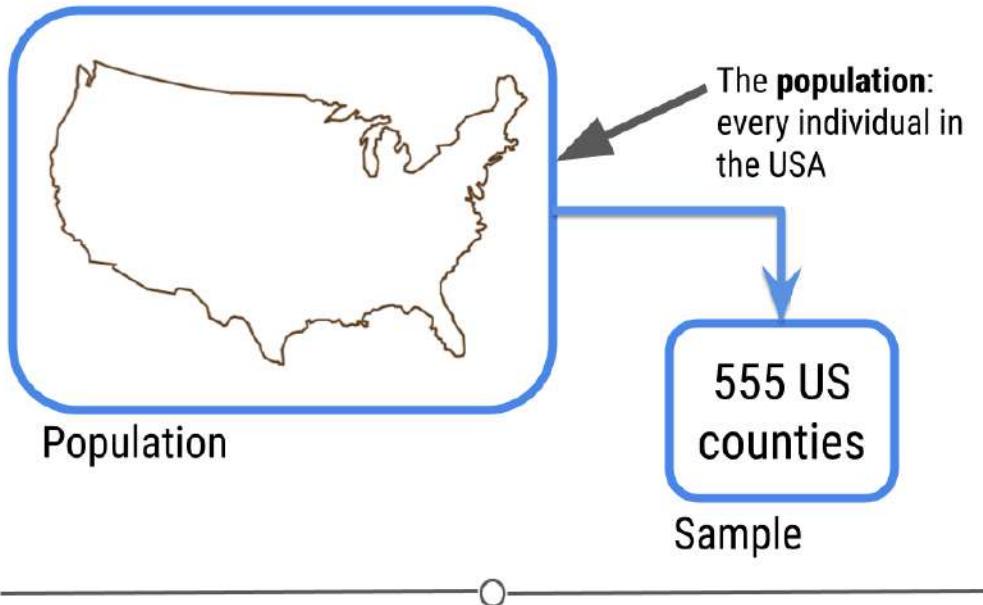
An example of inferential data analysis

Unlike in our previous examples, Census data wouldn't be used for inferential analysis. By definition, a census already collects information on (functionally) the entire population. Thus, there is no population on which to infer. Census data are the rare exception where a whole population is included in the dataset. Further, using data from the US census to infer information about another country would not be a good idea because the US isn't necessarily representative of the other country.

Instead, a better example of a dataset on which to carry out inferential analysis would be the data used in the study: [The Effect of Air Pollution Control on Life Expectancy in the United States: An Analysis of 545 US counties for the period 2000 to 2007](#). In this study, researchers set out to understand the effect of air pollution on everyone in the United States



To answer this question, a subset of the US population was studied, and the researchers looked at the level of air pollution experienced and life expectancy. It would have been nearly impossible to study every individual in the United States year after year. Instead, this study used the data they collected from a sample of the US population to infer how air pollution might be impacting life expectancy in the entire US!

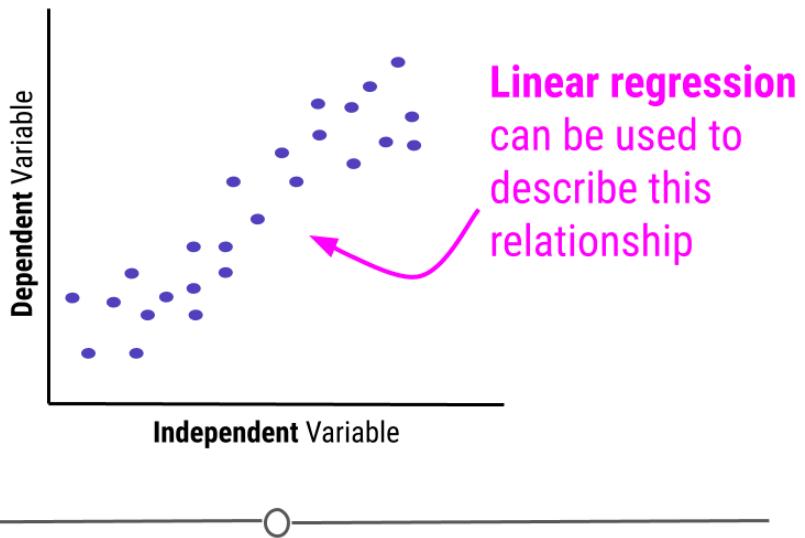


Linear Modeling

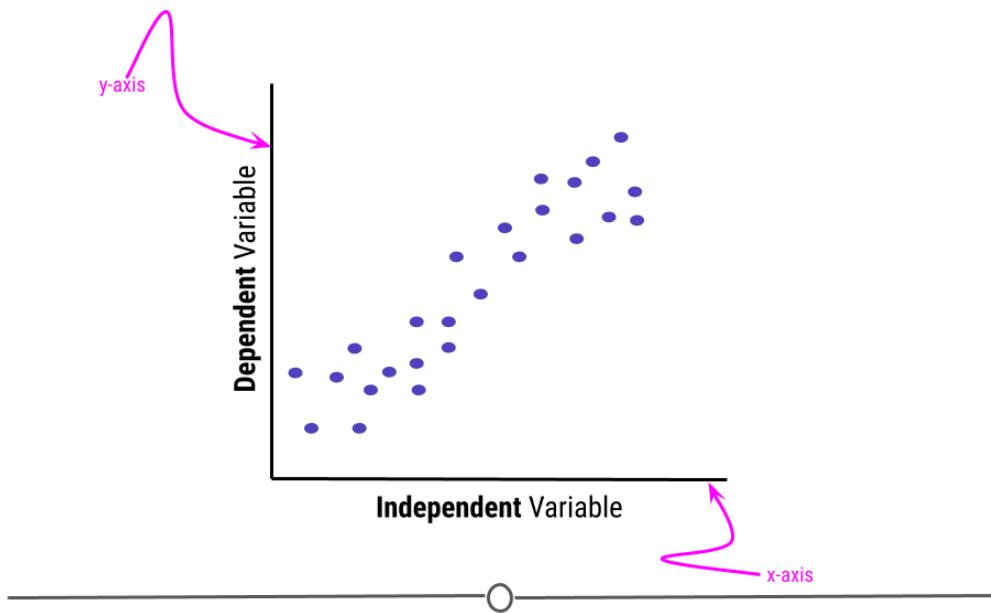
Linear Regression

Inferential analysis is commonly the goal of statistical modeling, where you have a small amount of information to extrapolate and generalize that information to a larger group. One of the most common approaches used in statistical modeling is known as linear regression. Here, we'll discuss how to recognize when using **linear regression** is appropriate, how to carry out the analysis in R, and how to interpret the results from this statistical approach.

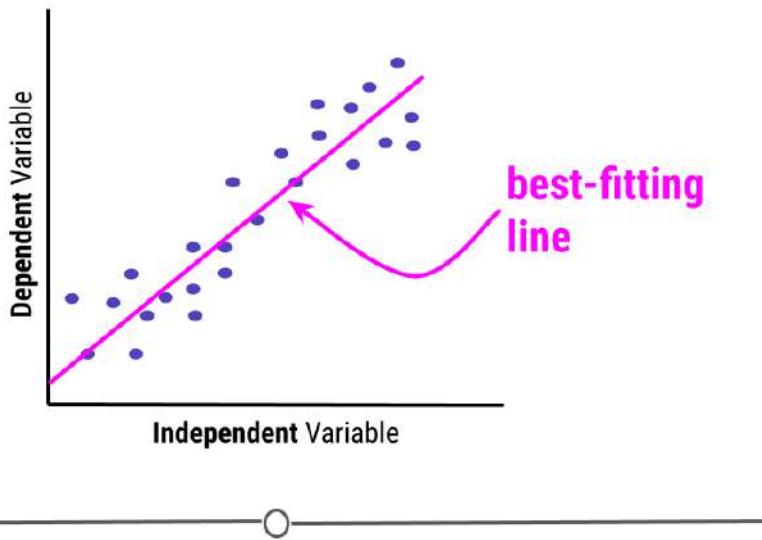
When discussing linear regression, we're trying to describe (model) the relationship between a **dependent variable** and an **independent variable**.



When visualizing a linear relationship, the independent variable is plotted along the bottom of the graph, on the **x-axis** and the dependent variable is plotted along the side of the plot, on the **y-axis**.

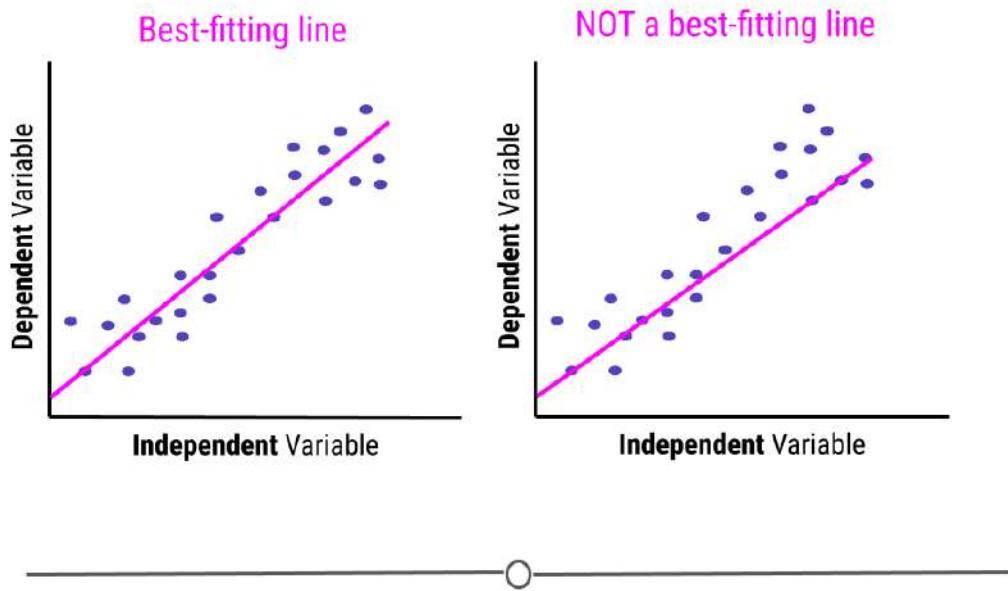


When carrying out linear regression, a **best-fitting line** is drawn through the data points to describe the relationship between the variables.

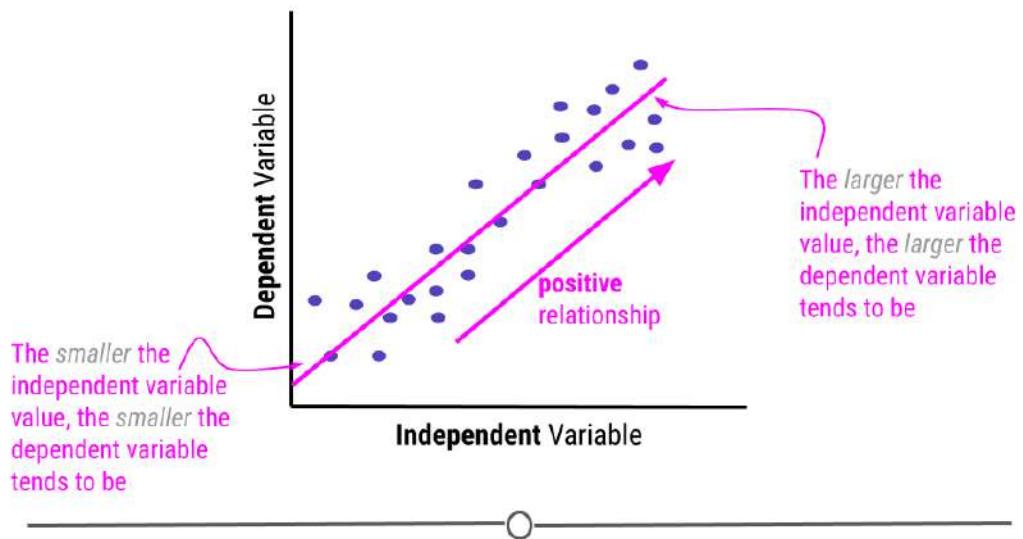


A best-fitting line, technically-speaking, minimizes the sum of the squared errors. In simpler terms, this means that the line that minimizes the distance of all the points from the line is the best-fitting line. Or, most simply, there are about the same number of points above the line as there are below the line. In total, the distance from the line for the points above the line will be the same as the distance from the points to the line below the line.

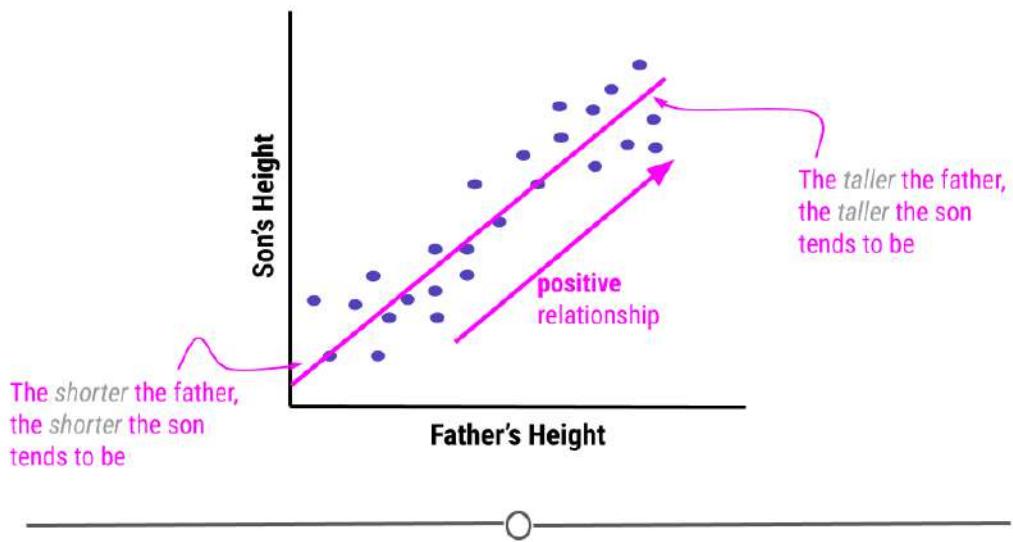
Note that the best fitting line does *not* have to go through any points to be the best-fitting line. Here, on the right, we see a line that goes through seven points on the plot (rather than the four the best-fitting line goes through, on the left). However, this is not a best-fitting line, as there are way more points above the line than there are below the line.



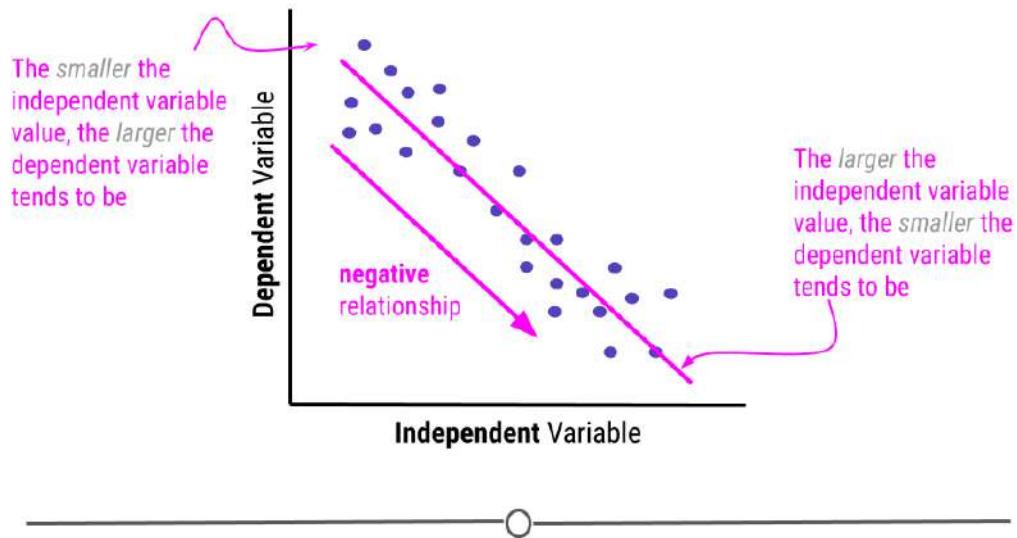
This line describes the relationship between the two variables. If you look at the direction of the line, it will tell you whether there is a positive or a negative relationship between the variables. In this case, the larger the value of the independent variable, the larger the value of the dependent variable. Similarly, the smaller the value of the independent variable, the smaller the value of the dependent variable. When this is the case, there is a **positive relationship** between the two variables.



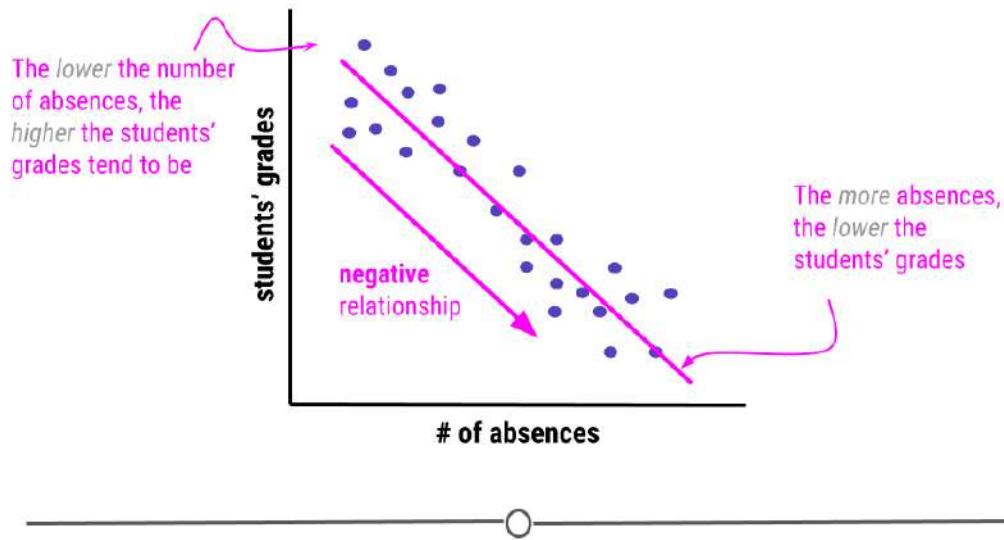
An example of variables that have a positive relationship would be the height of fathers and their sons. In general, the taller a father is, the taller his son will be. And, the shorter a father is the more likely his son is to be short.



Alternatively, when the higher the value of the independent variable, the lower the value of the dependent variable, this is a negative relationship.

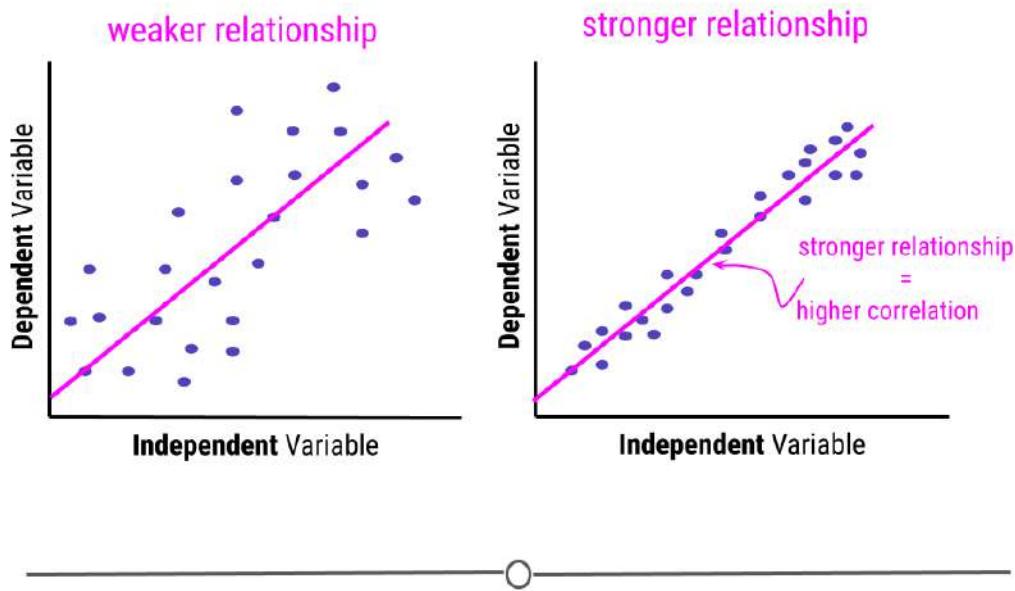


An example of variables that have a negative relationship would be the relationship between a students' absences and their grades. The more absences a student has, the lower their grades tend to be.



Linear regression, in addition to describing the direction of the relationship, it can also be used to determine the **strength** of that relationship.

This is because the assumption with linear regression is that the true relationship is being described by the best-fitting line. Any points that fall away from the line do so due to error. This means that if all the points fall directly on top of the line, there is no error. The further the points fall from the line, the greater the error. When points are further from the best-fitting line, the relationship between the two variables is weaker than when the points fall closer to the line.



In this example, the pink line is exactly the same best-fitting line in each graph. However, on the left, where the points fall further from the line, the strength of the relationship between these two variables is weaker than on the right, where the points fall closer to the line, where the relationship is stronger. The strength of this relationship is measured using **correlation**. The closer the points are to the line the more **correlated** the two variables are, meaning the relationship between the two variables is stronger.

Assumptions

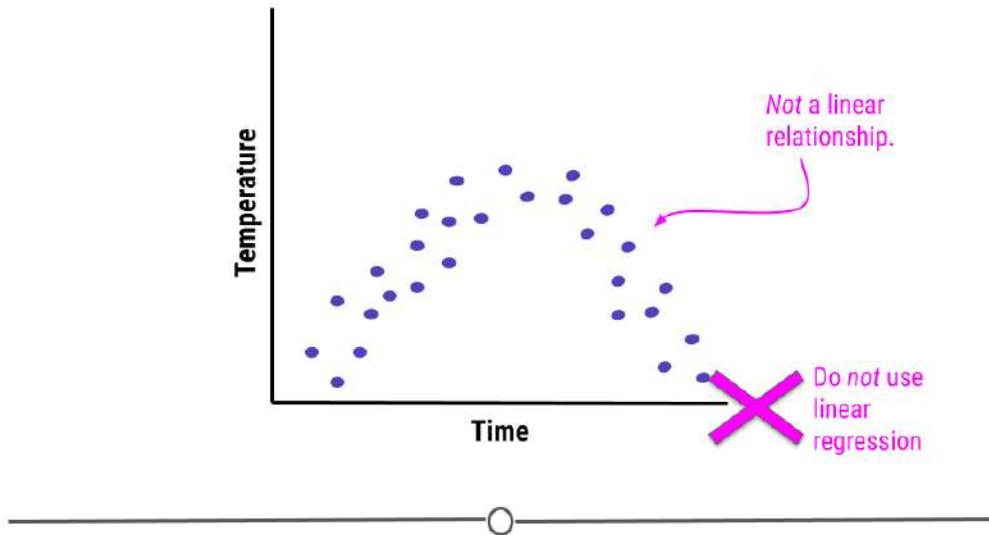
Thus far we have focused on drawing linear regression lines. Linear regression lines *can* be drawn on any plot, but just because you can do something doesn't mean you actually *should*. When it comes to linear regression, in order to carry out any inference on the relationship between two variables, there are a few assumptions that must hold before inference from linear regression can be done.

The two assumptions of simple linear regression are **linearity** and **homoscedasticity**.

Linearity

The relationship between the two variables must be linear.

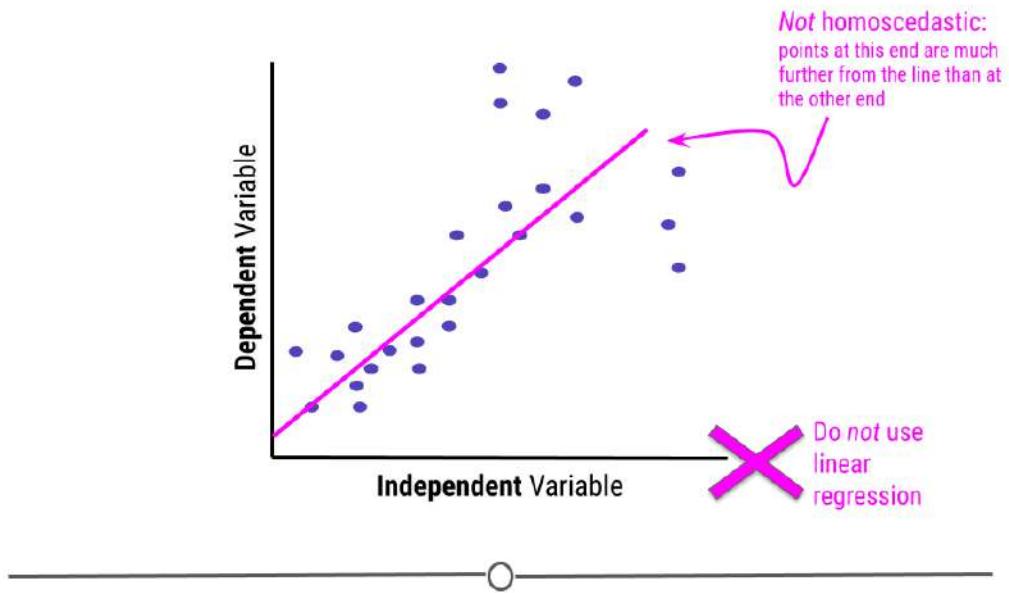
For example, what if we were plotting data from a single day and we were looking at the relationship between temperature and time. Well, we know that generally temperature increases throughout the day and then decreases in the evening. Here, we see some example data reflective of this relationship. The upside-down u-shape of the data suggests that the relationship is not in fact linear. While we *could* draw a straight line through these data, it would be inappropriate. In cases where the relationship between the variables cannot be well-modeled with a straight line, linear regression should not be used.



Homoscedasticity

In addition to displaying a linear relationship, the random variables must demonstrate **homoscedasticity**. In other words, the variance (distance from the line) must be constant throughout the variable.

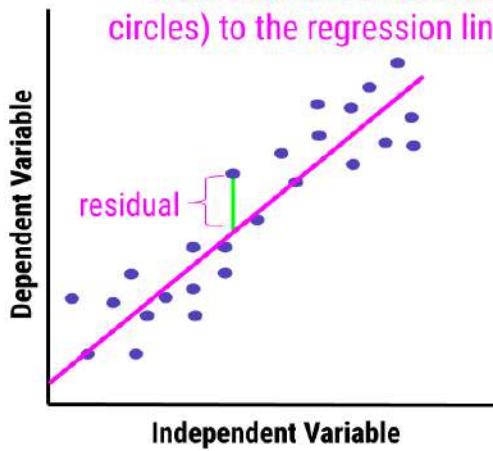
If points at one end are much closer to the best-fitting line than points are at the other end, homoscedasticity has been violated and linear regression is not appropriate for the data.



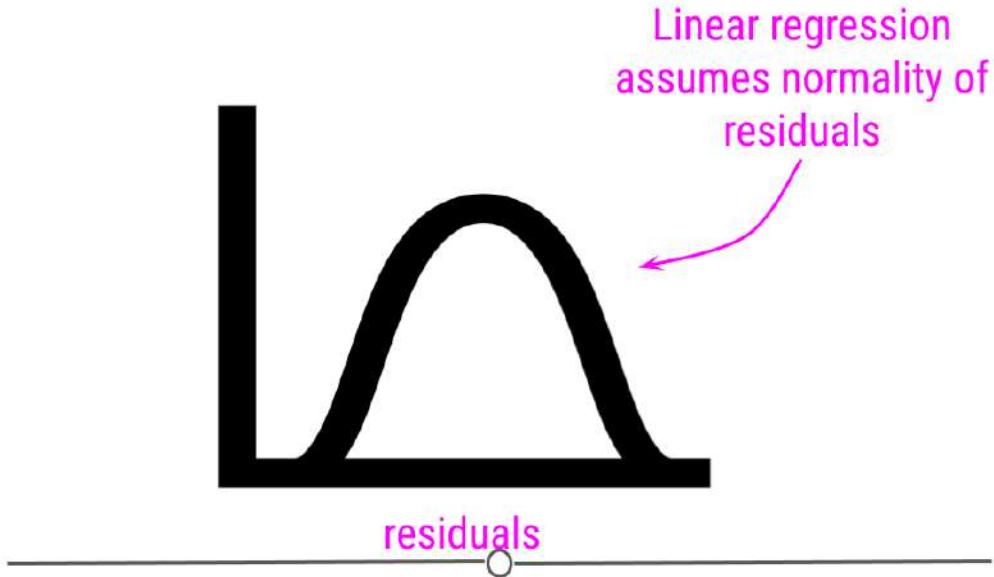
Normality of residuals

When we fit a linear regression, typically the data do not fall *perfectly* along the regression line. Rather, there is some distance from each point to the line. Some points are quite close to the line, while others are further away. Each point's distance to the regression line can be calculated. This distance is the **residual** measurement.

Residuals (green vertical line) are the distances from each observed data point (purple closed circles) to the regression line (the pink solid line)



In linear regression, one assumption is that these residuals follow a Normal distribution. This means that if you were to calculate each residual (each point's distance to the regression line) and then plot a histogram of all of those values - that plot should look like a Normal Distribution.



If you do not see normality of residuals, this can suggest that outlier values - observations more extreme than the rest of the data - may exist in your data. This can severely affect your regression results and lead you to conclude something that is untrue about your data.

Thus, it is your job, when running linear regression to check for:

- Non-linearity
- Heteroscedasticity
- Outlier values
- Normality of residuals

We'll discuss how to use diagnostic plots below to check that these assumptions have been met and that outlier values are not severely affecting your results.

What can linear regression infer?

Now that we understand what linear regression is and what assumptions must hold for its use, when would you actually use it? Linear regression can be used to answer many different questions about your data. Here we'll discuss specifically how to make inferences about the relationship between two numeric variables.

Association

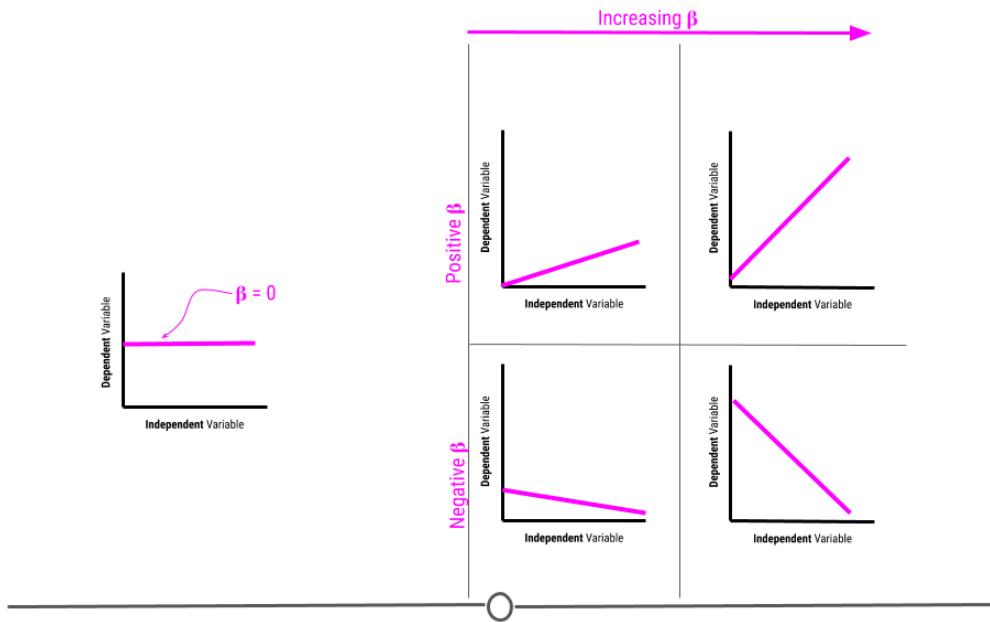
Often when people are carrying out linear regression, they are looking to better understand the relationship between two variables. When looking at this relationship, analysts are specifically asking “What is the **association** between these two variables?” Association between variables describes the trend in the relationship (positive, neutral, or negative) *and* the strength of that relationship (how correlated the two variables are).

After determining that the assumptions of linear regression are met, in order to determine the association between two variables, one would carry out a linear regression. From the linear regression, one would then interpret the **Beta estimate** and the **standard error** from the model.

Beta estimate

The beta estimate determines the direction and strength of the relationship between the two variables.

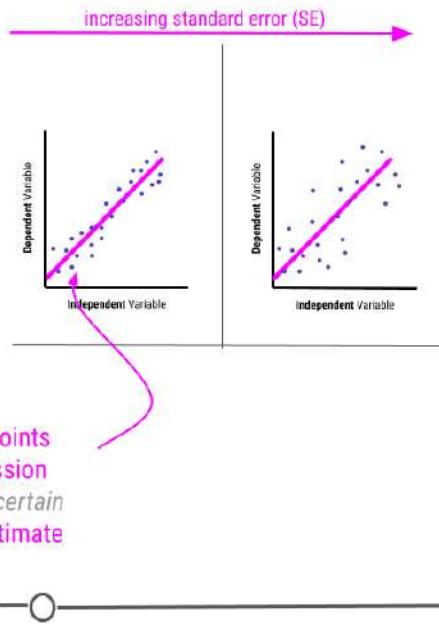
A beta of zero suggests there is no association between the two variables. However, if the beta value is positive, the relationship is positive. If the value is negative, the relationship is negative. Further, the larger the number, the bigger the effect is. We'll discuss effect size and how to interpret the value in more detail later in this lesson.



Standard error

The standard error determines how uncertain the beta estimate is. The larger the standard error, the *more* uncertain we are in the estimate. The smaller the standard error, the less uncertain we are in the estimate.

Standard errors are calculated based on how well the best-fitting line models the data. The closer the points are to the line, the lower the standard error will be, reflecting our decreased uncertainty. However, as the points are further from the regression line, our uncertainty in the estimate will increase, and the standard error will be larger.



Remember when carrying out inferential data analysis, you will always want to report an estimate and a measure of uncertainty. For linear regression, this will be the **beta estimate** and the **standard error**.

You may have heard talk of **p-values** at some point. People tend to use p-values to describe the strength of their association due to its simplicity. The p-value is a single number that takes into account both the estimate (beta estimate) and the uncertainty in that estimate (SE). The lower a p-value the more significant the association is between two variables. However, while it is a simple value, it doesn't tell you nearly as much information as reporting the estimates and standard errors directly. Thus, if you're reporting p-values, it's best to also include the estimate and standard errors as well.

That said, the general interpretation of a p-value is “the probability of getting the observed results (or results more extreme) by chance alone.” Since it’s a probability, the value will always be between 0 and 1. Then, for example, a p-value of 0.05, means that 5 percent of the time (or 1 in 20), you’d observe results this extreme simply by chance.

Association Testing in R

Now that we've discussed what you can learn from an association test, let's look at an example in R. For this example we'll use the `trees` dataset available in R, which includes

girth, height, and volume measurements for 31 black cherry trees.

With this dataset, we'll answer the question:

Can we infer the height of a tree given its girth?

Presumably, it's easier to measure a tree's girth (width around) than it is to measure its height. Thus, here we want to know whether or not height and girth are associated.

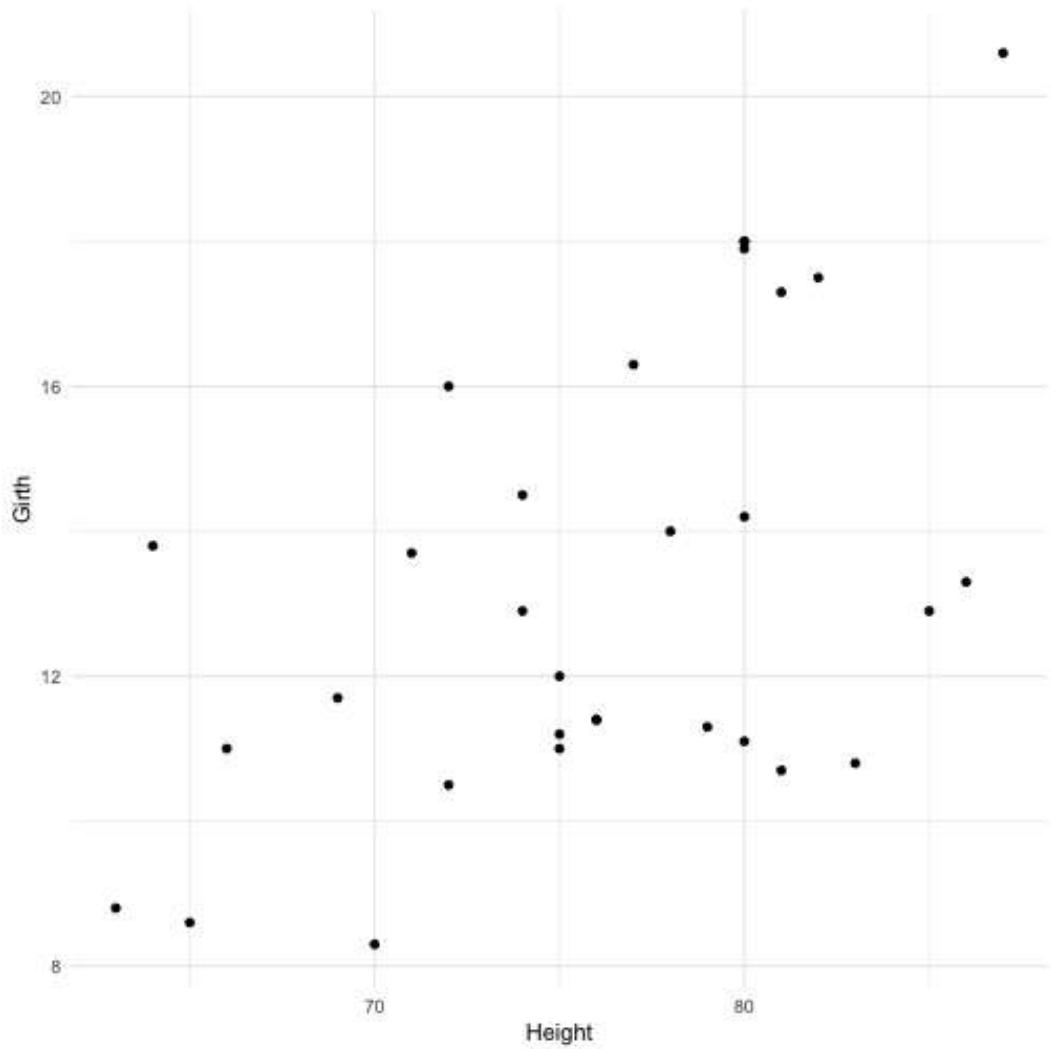
In this case, since we're asking if we can infer height from girth, girth is the independent variable and height is the dependent variable. In other words, we're asking does height depend on girth?

First, before carrying out the linear regression to test for association and answer this question, we have to be sure linear regression is appropriate. We'll test for linearity and homoscedasticity.

To do so, we'll first use ggplot2 to generate a scatterplot of the variables of interest.

```
library(ggplot2)

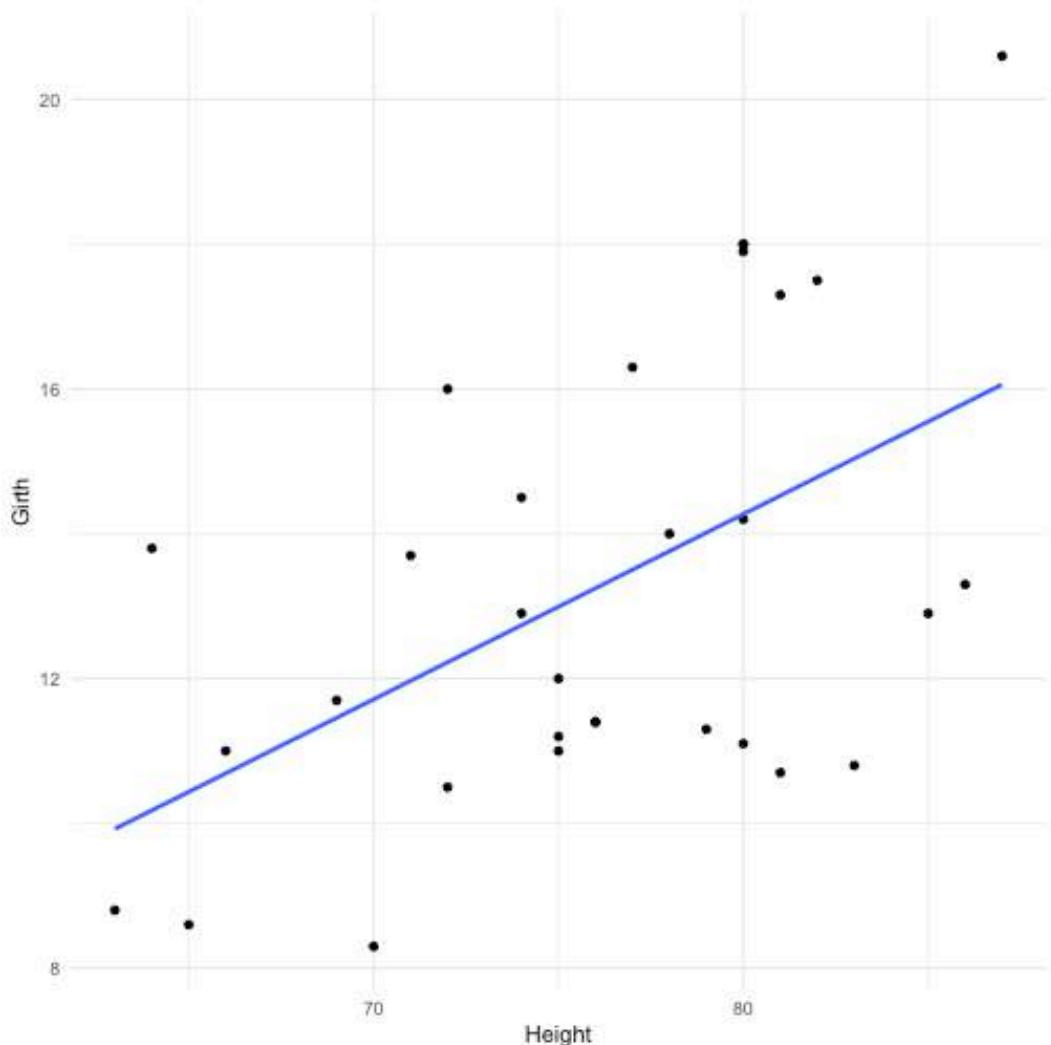
trees %>%
  ggplot() +
  geom_point(aes(Height, Girth))
```



plot of chunk unnamed-chunk-26

From the looks of this plot, the relationship looks approximately linear, but to visually make this a little easier, we'll add a line of best fit to the plot.

```
trees %>%  
  ggplot(aes(Height, Girth)) +  
  geom_point() +  
  geom_smooth(method = "lm", se = FALSE)  
 `geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-27

On this graph, the relationship looks approximately linear and the variance (distance from points to the line) is constant across the data. Given this, it's appropriate to use linear

regression for these data.

Fitting the Model

Now that that's established, we can run the linear regression. To do so, we'll use the `lm()` function to **fit the model**. The syntax for this function is `lm(dependent_variable ~ independent_variable, data = dataset)`.

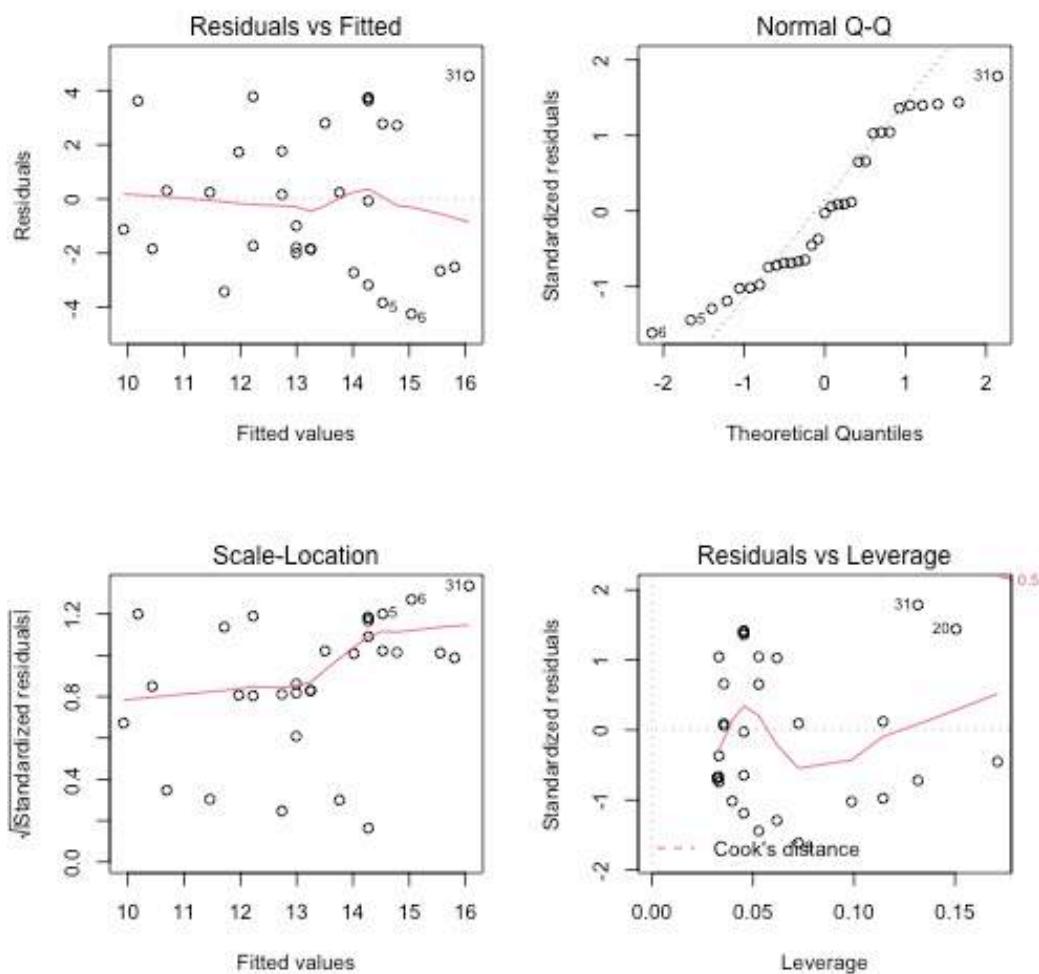
```
## run the regression
fit <- lm(Girth ~ Height , data = trees)
```

Model Diagnostics

Above, we discussed a number of assumptions of linear regression. After fitting a model, it's necessary to check the model to see if the model satisfies the assumptions of linear regression. If the model does not fit the data well (for example, the relationship is nonlinear), then you cannot use and interpret the model.

In order to assess your model, a number of diagnostic plots can be very helpful. Diagnostic plots can be generated using the `plot()` function with the fitted model as an argument.

```
par(mfrow = c(2, 2))
plot(fit)
```



plot of chunk unnamed-chunk-29

This generates four plots: 1) **Residuals vs Fitted** - checks linear relationship assumption of linear regression. A linear relationship will demonstrate a horizontal red line here. Deviations from a horizontal line suggest nonlinearity and that a different approach may be necessary.

2) **Normal Q-Q** - checks whether or not the residuals (the difference between the observed and predicted values) from the model are normally distributed. The best fit models points fall along the dashed line on the plot. Deviation from this line suggests that a different analytical

approach may be required.

- 3) **Scale-Location** - checks the homoscedasticity of the model. A horizontal red line with points equally spread out indicates a well-fit model. A non-horizontal line or points that cluster together suggests that your data are not homoscedastic.
- 4) **Residuals vs Leverage** - helps to identify outlier or extreme values that may disproportionately affect the model's results. Their inclusion or exclusion from the analysis may affect the results of the analysis. Note that the top three most extreme values are identified with numbers next to the points in all four plots.

Tree Girth and Height Example

In our example looking at the relationship between tree girth and height, we can first check **linearity** of the data by looking at the **Residuals vs Fitted** plot. Here, we do see a red line that is approximately horizontal, which is what we're looking for. Additionally, we're looking to be sure there is no clear pattern in the points on the plot - we want them to be random on this plot. Clustering of a bunch of points together or trends in this plot would indicate that the data do not have a linear relationship.

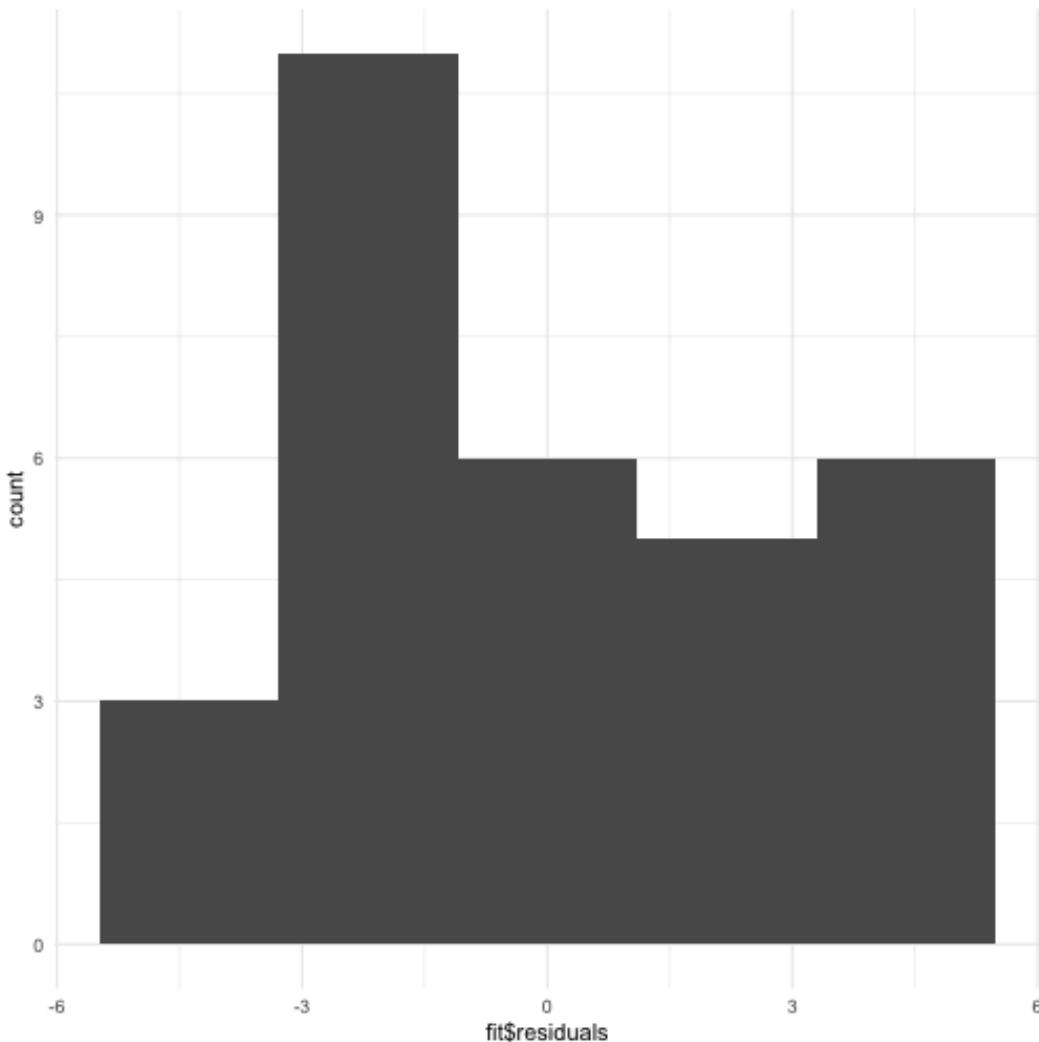
To check for **homogeneity of the variance**, we can turn to the **Scale-Location** plot. Here, we're again looking for a horizontal red line. In this dataset, there's a suggestion that there is some heteroscedasticity, with points not being equally far from the regression line across the observations.

While not discussed explicitly here in this lesson, we will note that when the data are nonlinear or the variances are not homogeneous (are not homoscedastic), **transformations** of the data can often be applied and then linear regression can be used.

QQ Plots are very helpful in assessing the **normality of residuals**. Normally distributed residuals will fall along the grey dotted line. Deviation from the line suggests the residuals are not normally distributed. Here, in this example, we do not see the points fall perfectly along the dotted line, suggesting that our residuals are not normally distributed.

A **histogram** (or densityplot) of the residuals can also be used for this portion of regression diagnostics. Here, we're looking for a **Normal distribution** of the residuals.

```
library(ggplot2)
ggplot(fit, aes(fit$residuals)) +
  geom_histogram(bins = 5)
```



plot of chunk unnamed-chunk-30

The QQ Plot and the histogram of the residuals will always give the same answer. Here, we see that with our limited sample size, we do not have perfectly Normally distributed residuals; however, the points do not fall wildly far from the dotted line.

Finally, whether or not **outliers** (extreme observations) are driving our results can be assessed by looking at the **Residuals vs Leverage** plot.

Generally speaking, standardized residuals greater than 3 or less than -3 are to be considered as outliers. Here, we do not see any values in that range (by looking at the y-axis), suggesting

that there are no extreme outliers driving the results of our analysis.

Interpreting the Model

While the relationship in our example appears to be linear, does not indicate being driven by outliers, is approximately homoscedastic and has residuals that are not perfectly Normally distributed, but fall close to the line in the QQ plot, we can discuss how to interpret the results of the model.

```
## take a look at the output
summary(fit)
```

The `summary()` function summarizes the model as well as the output of the model. We can see the values we're interested in in this summary, including the beta estimate, the standard error (SE), and the p-value.

Specifically, from the beta estimate, which is positive, we confirm that the relationship is positive (which we could also tell from the scatterplot). We can also interpret this beta estimate explicitly.

```
> fit <- lm(Girth ~ Height , data = trees)
>
> summary(fit)

Call:
lm(formula = Girth ~ Height, data = trees)

Residuals:
    Min      1Q  Median      3Q     Max 
-4.2386 -1.9205 -0.0714  2.7450  4.5384 

For every one
inch increase
in height, the
girth will
increase by
0.255 inches
          
Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -6.18839  5.96020 -1.038  0.30772  
Height       0.25575  0.07816  3.272  0.00276 ** 
---
           $\beta$  estimate   SE      p-value
Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.728 on 29 degrees of freedom
Multiple R-squared:  0.2697,    Adjusted R-squared:  0.2445 
F-statistic: 10.71 on 1 and 29 DF,  p-value: 0.002758
```

The **beta estimate** (also known as the beta coefficient or coefficient in the Estimate column) is the amount the dependent variable will change given a one unit increase in the independent variable. In the case of the trees, a beta estimate of 0.256, says that for every inch a tree's girth increases, its height will increase by 0.256 inches. Thus, we not only know that there's a positive relationship between the two variables, but we know by precisely how much one variable will change given a single unit increase in the other variable. Note that we're looking at the second row in the output here, where the row label is "Height". This row quantifies the relationship between our two variables. The first row quantifies the intercept, or where the line crosses the y-axis.

The standard error and p-value are also included in this output. Error is typically something we want to minimize (in life and statistical analyses), so the *smaller* the error, the *more confident* we are in the association between these two variables.

The beta estimate and the standard error are then both considered in the calculation of the p-value (found in the column $\text{Pr} [> |t|]$). The smaller this value is, the more confident we are that this relationship is not due to random chance alone.

Variance Explained

Additionally, the strength of this relationship is summarized using the adjusted R-squared metric. This metric explains how much of the variance this regression line explains. The more variance explained, the closer this value is to 1. And, the closer this value is to 1, the closer the points in your dataset fall to the line of best fit. The further they are from the line, the closer this value will be to zero.

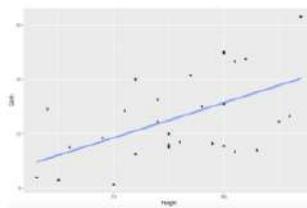
```
> fit <- lm(Girth ~ Height , data = trees)
>
> summary(fit)

Call:
lm(formula = Girth ~ Height, data = trees)

Residuals:
    Min      1Q  Median      3Q     Max 
-4.2386 -1.9205 -0.0714  2.7450  4.5384 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -6.18839   5.96020  -1.038  0.30772    
Height       0.25575   0.07816   3.272  0.00276 **  
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.728 on 29 degrees of freedom
Multiple R-squared:  0.2697,  Adjusted R-squared:  0.2445 
F-statistic: 10.71 on 1 and 29 DF,  p-value: 0.002758
```



Describes the strength of
the correlation

As we saw in the scatterplot, the data are not right up against the regression line, so a value of 0.2445 seems reasonable, suggesting that this model (this regression line) explains 24.45% of the variance in the data.

Using broom

Finally, while the `summary()` output are visually helpful, if you want to get any of the numbers out from that model, it's not always straightforward. Thankfully, there is a package to help you with that! The `tidy()` function from the `broom` package helps take the summary output from a statistical model and organize it into a tabular output.

```
#install.packages("broom")
library(broom)
tidy(fit)
# A tibble: 2 × 5
  term      estimate std.error statistic p.value
  <chr>     <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) -6.19      5.96     -1.04   0.308
2 Height       0.256     0.0782     3.27  0.00276
```

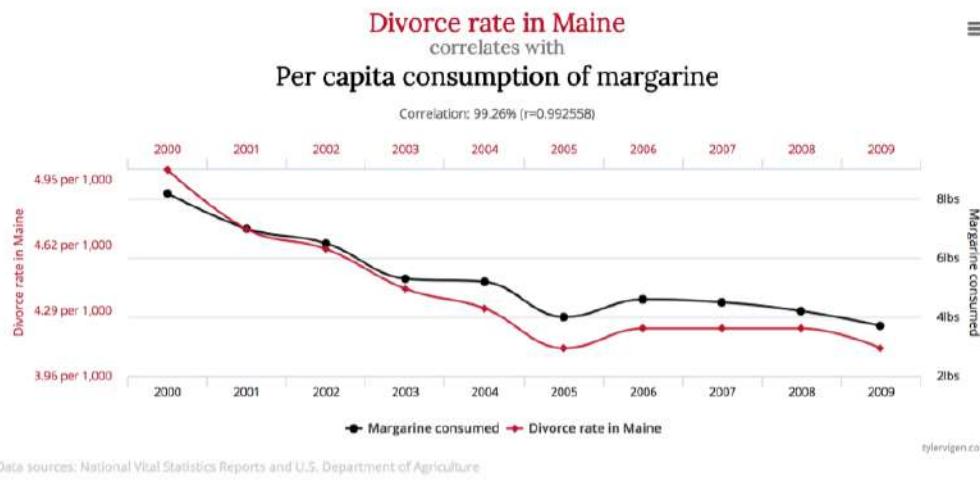
Note that the values *haven't* changed. They're just organized into an easy-to-use table. It's helpful to keep in mind that this function and package exist as you work with statistical models.

Finally, it's important to always keep in mind that the **interpretation of your inferential data analysis** is incredibly important. When you use linear regression to test for association, you're looking at the relationship between the two variables. While girth can be used to infer a tree's height, this is just a correlation. It **does not mean** that an increase in girth **causes** the tree to grow more. Associations are *correlations*. They are **not** causal.

For now, however, in response to our question, can we infer a black cherry tree's height from its girth, the answer is yes. We would expect, on average, a tree's height to increase 0.255 inches for every one inch increase in girth.

Correlation Is Not Causation

You've likely heard someone say before that "correlation is not causation," and it's true! In fact, there are [entire websites](#) dedicated to this concept. Let's make sure we know exactly what that means before moving on. In the plot you see here, as the divorce rate in Maine decreases, so does per capita consumption of margarine. These two lines are clearly correlated; however, there isn't really a strong (or any) argument to say that one caused the other. Thus, just because you see two things with the same trend does not mean that one caused the other. These are simply **spurious correlations** – things that trend together by chance. Always keep this in mind when you're doing inferential analysis, and be sure that you **never draw causal claims when all you have are associations**.



<http://www.tylervigen.com/spurious-correlations>

In fact, one could argue that the only time you can make causal claims are when you have carried out a randomized experiment. **Randomized experiments** are studies that are designed and carried out by randomly assigning certain subjects to one treatment and the rest of the individuals to another treatment. The treatment is then applied and the results are then analyzed. In the case of a randomized experiment, causal claims can start to be made. Short of this, however, be careful with the language you choose and do not overstate your findings.

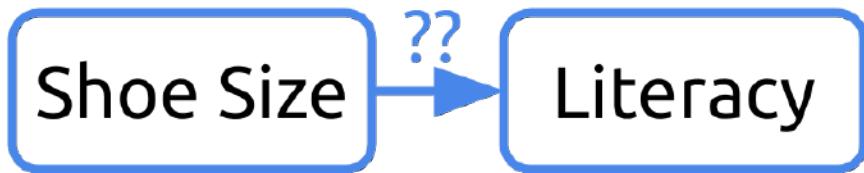
Confounding

Confounding is something to watch out for in any analysis you're doing that looks at the relationship between two more variables. So...what is confounding?

Let's consider an example. What if we were interested in understanding the relationship between shoe size and literacy. To do so, we took a look at this small sample of two humans, one who wears small shoes and is not literate and one adult who wears big shoes and is literate.



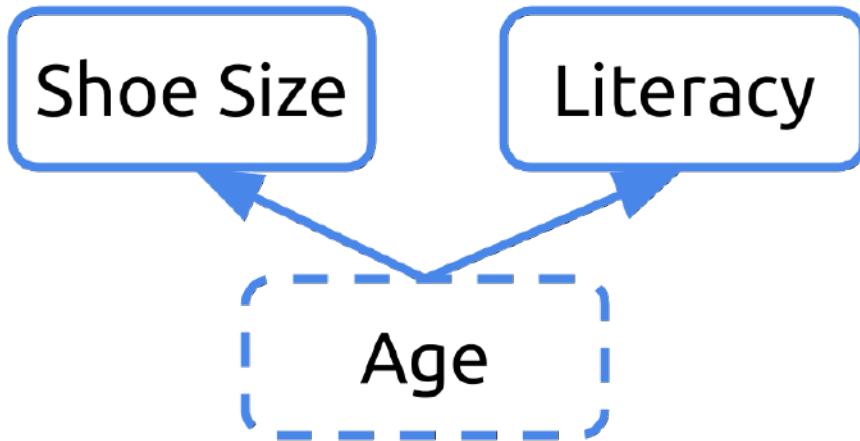
If we were to diagram this question, we may ask “Can we infer literacy rates from shoe size?”



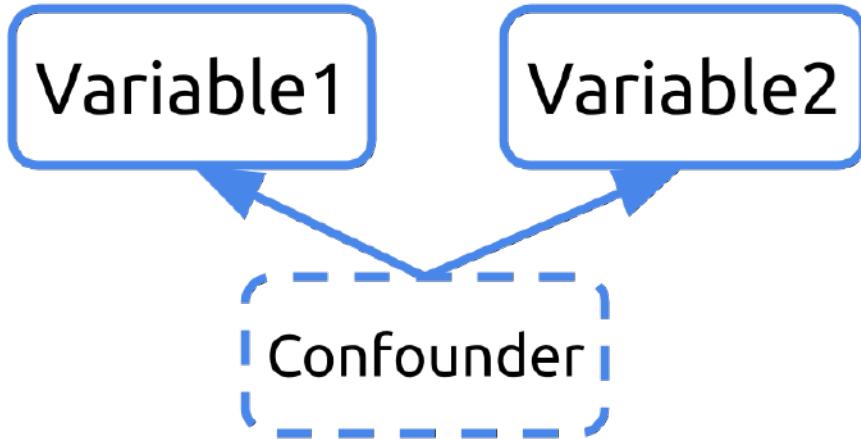
If we return to our sample, it'd be important to note that one of the humans is a young child and the other is an adult.



Our initial diagram failed to take into consideration the fact that these humans differed in their age. Age affects their shoe size and their literacy rates. In this example, age is a confounder.



Any time you have a variable that affects both your dependent and independent variables, it's a confounder. Ignoring confounders is not appropriate when analyzing data. In fact, in this example, you would have concluded that people who wear small shoes have lower literacy rates than those who wear large shoes. That would have been incorrect. In fact, that analysis was *confounded* by age. Failing to correct for confounding has led to misreporting in the media and retraction of scientific studies. You don't want to be in that situation. So, **always consider and check for confounding** among the variables in your dataset.



Multiple Linear Regression

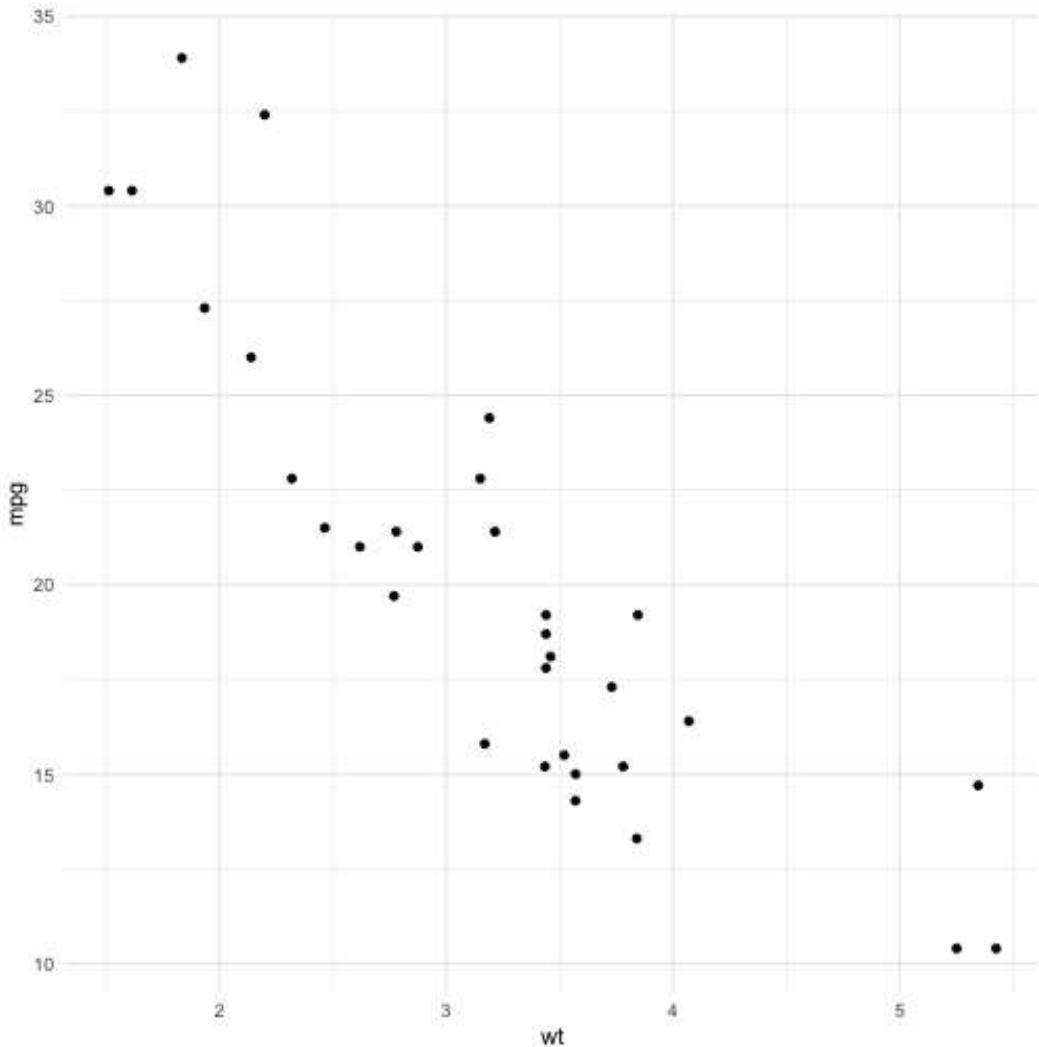
There are ways to effectively handle confounders within an analysis. Confounders can be included in your linear regression model. When included, the analysis takes into account the fact that these variables are confounders and carries out the regression, removing the effect of the confounding variable from the estimates calculated for the variable of interest.

This type of analysis is known as **multiple linear regression**, and the general format is: `lm(dependent_variable ~ independent_variable + confounder, data = dataset)`.

As a simple example, let's return to the `mtcars` dataset, which we've worked with before. In this dataset, we have data from 32 automobiles, including their weight (`wt`), miles per gallon (`mpg`), and Engine (`vs`, where 0 is “V-shaped” and 1 is “straight”).

Suppose we were interested in inferring the mpg a car would get based on its weight. We'd first look at the relationship graphically:

```
## take a look at scatterplot
ggplot(mtcars, aes(wt, mpg)) +
  geom_point()
```



plot of chunk unnamed-chunk-33

From the scatterplot, the relationship looks approximately linear and the variance looks constant. Thus, we could model this using linear regression:

```
## model the data without confounder
fit <- lm(mpg ~ wt, data = mtcars)
tidy(fit)

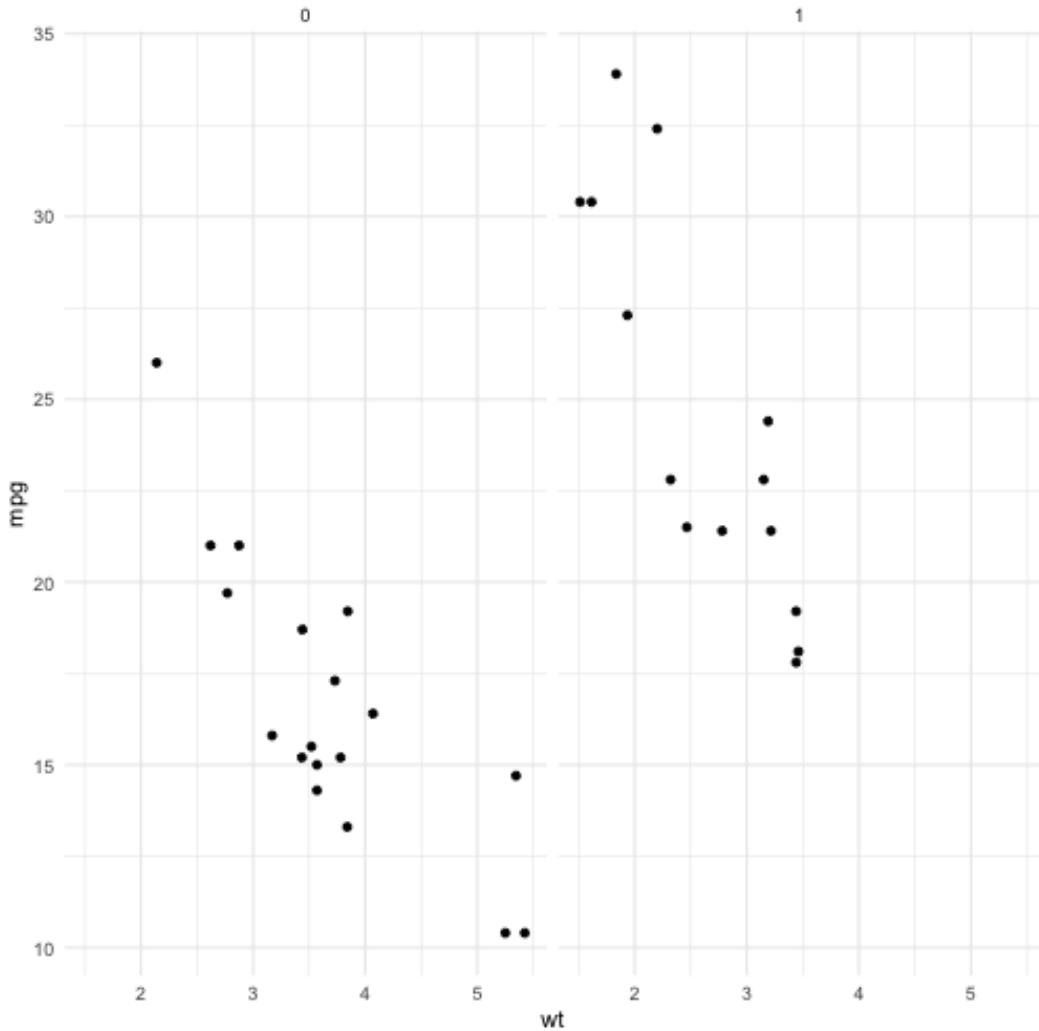
> ## model the data without confounder
> fit <- lm(mpg ~ wt, data = mtcars)
> tidy(fit)
  term estimate std.error statistic      p.value
1 (Intercept) 37.285126  1.877627 19.857575 8.241799e-19
2       wt -5.344472  0.559101 -9.559044 1.293959e-10
.
.
.

For every 1000 lb increase
in weight, there is a 5.34
mpg decrease in gas
mileage
```

From this analysis, we would infer that for every increase 1000 lbs more a car weighs, it gets 5.34 miles less per gallon.

However, we know that the weight of a car doesn't necessarily tell the whole story. The type of engine in the car likely affects both the weight of the car and the miles per gallon the car gets. Graphically, we could see if this were the case by looking at these scatterplots:

```
## look at the difference in relationship
## between Engine types
ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  facet_wrap(~vs)
```



plot of chunk unnamed-chunk-35

From this plot, we can see that V-shaped engines ($vs = 0$), tend to be heavier and get fewer miles per gallon while straight engines ($vs = 1$) tend to weigh less and get more miles per gallon. Importantly, however, we see that a car that weighs 3000 points ($wt = 3$) and has a V-Shaped engine ($vs = 0$) gets fewer miles per gallon than a car of the same weight with a straight engine ($vs = 1$), suggesting that simply modeling a linear relationship between weight and mpg is not appropriate.

Let's then model the data, taking this confounding into account:

```
## include engine (vs) as a confounder
fit <- lm(mpg ~ wt + vs, data = mtcars)
tidy(fit)

> ## include engine (vs) as a confounder
> fit <- lm(mpg ~ wt + vs, data = mtcars)
> tidy(fit)
  term estimate std.error statistic    p.value
1 (Intercept) 33.004233 2.3553946 14.012188 1.920621e-14
2       wt -4.442814 0.6133645 -7.243350 5.632548e-08
3        vs  3.154367 1.1907378  2.649086 1.292580e-02
```

For a V-Shaped engine, for
every 1000 lb increase in
weight, there is a 4.44
mpg decrease in gas
mileage

Here, we get a more accurate picture of what's going on. Interpreting multiple regression models is slightly more complicated since there are more variables; however, we'll practice how to do so now.

The best way to interpret the coefficients in a multiple linear regression model is to focus on a single variable of interest and hold all other variables constant. For instance, we'll focus on weight (`wt`) while holding (`vs`) constant to interpret. This means that for a V-shaped engine, we expect to see a 4.44 miles per gallon decrease for every 1000 lb increase in weight.

We can similarly interpret the coefficients by focusing on the engines (`vs`). For example, for two cars that weigh the same, we'd expect a straight engine (`vs = 1`) to get 3.15 more miles per gallon than a V-Shaped engine (`vs = 0`).

```
> ## include engine (vs) as a confounder
> fit <- lm(mpg ~ wt + vs, data = mtcars)
> tidy(fit)
  term estimate std.error statistic    p.value
1 (Intercept) 33.004233 2.3553946 14.012188 1.920621e-14
2          wt -4.442814 0.6133645 -7.243350 5.632548e-08
3          vs  3.154367 1.1907378  2.649086 1.292580e-02
```

For two vehicles of the same weight, a straight engine will get 3.15 more mpg (on average) than a V-Shaped engine

Finally, we'll point out that the p-value for `wt` decreased in this model relative to the model where we didn't account for confounding. This is because the model was not initially taking into account the engine difference. Frequently when confounders are accounted for, the p-value will increase, and that's OK. What's important is that the data are most appropriately modeled.

Beyond Linear Regression

While we've focused on linear regression in this lesson on inference, linear regression isn't the only analytical approach out there. However, it is arguably the most commonly used. And, beyond that, there are *many* statistical tests and approaches that are slight variations on linear regression, so having a solid foundation and understanding of linear regression makes understanding these other tests and approaches much simpler.

For example, what if you didn't want to measure the linear relationship between two variables, but instead wanted to know whether or not the average observed is different from expectation?

Mean Different From Expectation?

To answer a question like this, let's consider the case where you're interested in analyzing data about a single numeric variable. If you were doing descriptive statistics on this dataset, you'd likely calculate the mean for that variable. But, what if, in addition to knowing the mean, you wanted to know if the values in that variable were all within the bounds of normal variation. You could calculate that using inferential data analysis. You could use the data you have to *infer* whether or not the data are within the expected bounds.

For example, let's say you had a dataset that included the number of ounces *actually* included in 100 cans of a soft drink. You'd expect that each can have exactly 12 oz of liquid; however, there is some variation in the process. So, let's test whether or not you're consistently getting shorted on the amount of liquid in your can.

In fact, let's go ahead and generate the dataset ourselves!

```
## generate the dataset
set.seed(34)
soda_ounces <- rnorm(100, mean = 12, sd = 0.04)
head(soda_ounces)
[1] 11.99444 12.04799 11.97009 11.97699 11.98946 11.98178
```

In this code, we're specifying that we want to take a random draw of 100 different values (representing our 100 cans of soft drink), where the mean is 12 (representing the 12 ounces of soda expected to be within each can), and allowing for some variation (we've set the standard deviation to be 0.04).

We can see that the values are approximately, but not always exactly equal to the expected 12 ounces.

Testing Mean Difference From Expectation in R

To make an inference as to whether or not we're consistently getting shorted, we're going to use this sample of 100 cans. Note that we're using this sample of cans to infer something about all cans of this soft drink, since we aren't able to measure the number of ounces in all cans of the soft drink generated.

To carry out this statistical test, we'll use a t-test.

Wait, we haven't talked about that statistical test yet. So, let's take a quick detour to discuss t-tests and how they relate to linear regression.

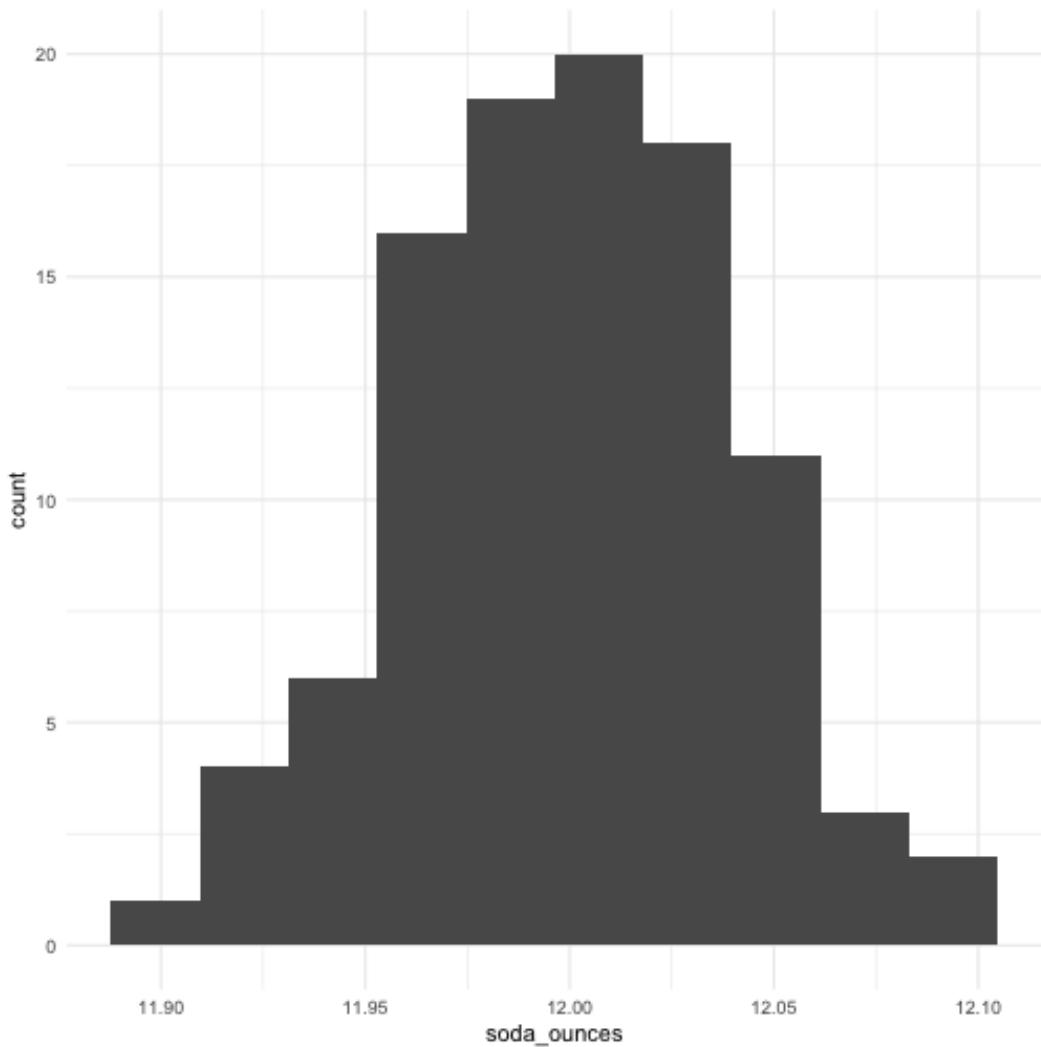
R has a built in t-test function: `t.test()`.

However, I mentioned earlier that many statistical tests are simply extension of linear regression. In fact, a t-test is simply a linear model where we specify to only fit an intercept (where the data crosses the y-axis). In other words, this specifies to calculate the mean...which is exactly what we're looking to do here with our t-test! We'll compare these two approaches below.

However, before we can do so, we have to ensure that the data follow a normal distribution, since this is the primary assumption of the t-test.

```
library(ggplot2)

## check for normality
ggplot(as.data.frame(soda_ounces))+
  geom_histogram(aes(soda_ounces), bins = 10)
```



plot of chunk unnamed-chunk-38

Here, we see that the data are approximately normally distributed.

A t-test will check whether the observed ounces differs from the expected mean (12 oz). As mentioned above, to run a t-test in R, most people use the built-in function: `t.test()`.

```
## carry out t-test
t.test(soda_ounces, mu = 12)

One Sample t-test

data: soda_ounces
t = -0.074999, df = 99, p-value = 0.9404
alternative hypothesis: true mean is not equal to 12
95 percent confidence interval:
11.99187 12.00754
sample estimates:
mean of x
11.9997
```

In the output from this function, we'll focus on the 95 percent confidence interval. Confidence Intervals provide the range of values likely to contain the unknown population parameter. Here, the population parameter we're interested in is the mean. Thus, the 95% Confidence Intervals provides us the range where, upon repeated sampling, the calculated mean would fall 95 percent of the time. More specifically, if the 95 percent confidence interval contains the expected mean (12 oz), then we can be confident that the company is not shorting us on the amount of liquid they're putting into each can.

Here, since 12 is between 11.99187 and 12.00754, we can see that the amounts in the 100 sampled cans are within the expected variation. We could infer from this sample that the population of all cans of this soft drink are likely to have an appropriate amount of liquid in the cans.

However, as mentioned previously, t-tests are an extension of linear regression. We could also look to see whether or not the cans had the expected average of 12 oz in the data collected using `lm()`.

```
# from linear regression
regression_output <- lm(soda_ounces ~ 1)

# calculate confidence interval
confint(regression_output)
```

```

> ## carry out t-test
> t.test(soda_ounces, mu = 12)

One Sample t-test

data: soda_ounces
t = -0.074999, df = 99, p-value = 0.9404
alternative hypothesis: true mean is not equal to 12
95 percent confidence interval:
11.99187 12.00754
sample estimates:
mean of x
11.9997

> # from linear regression
> regression_output <- lm(soda_ounces ~ 1)
>
> # calculate confidence interval
> confint(regression_output)
2.5 % 97.5 %
(Intercept) 11.99187 12.00754

```

Specify to fit linear regression using intercept only

95% Confidence Interval is the exact same as using t.test()

Note that the confidence interval is the exact same here using `lm()` as above when we used `t.test()`! We bring this up not to confuse you, but to guide you away from trying to memorize each individual statistical test and instead understand how they relate to one another.

More Statistical Tests

Now that you've seen how to measure the linear relationship between variables (linear regression) and how to determine if the mean of a dataset differs from expectation (t-test), it's important to know that you can ask lots of different questions using extensions of linear regression. These have been nicely summarized by Jonas Kristoffer Lindelov in his blog post [Common statistical tests are linear models \(or: how to teach stats\)](#).

Hypothesis Testing

You may have noticed in the previous sections that we were asking a question about the data. We did so by testing if a particular answer to the question was true.

For example:

1) In the cherry tree analysis we asked “Can we infer the height of a tree given its girth?” We expected that we could. Thus we had a statement that “tree height can be inferred by its girth or can be predicted by girth”

2) In the car mileage analysis we asked “Can we infer the miles the car can go per gallon of gasoline based on the car weight?”

We expected that we could. Thus we had a statement that “car mileage can be inferred by car weight”

We took this further and asked “Can we infer the miles the car can go per gallon of gasoline based on the car weight and care engine type?”

We again expected that it did. Thus we had a statement that “ car mileage can be inferred by weight and engine type”

3) In the soda can analysis we asked “Do soda cans really have 12 ounces of fluid”. We expected that often do. Thus we had a statement that “soda cans typically have 12 ounces, the mean amount is 12”.

A common problem in many data science problem involves developing evidence for or against certain testable statements like these statements above. These testable statements are called *hypotheses*. Typically, the way these problems are structured is that a statement is made about the world (the hypothesis) and then the data are used (usually in the form of a summary statistic) to support or reject that statement.

Recall that we defined a p-value as “the probability of getting the observed results (or results more extreme) by chance alone.” Often p-values are used to determine if one should accept or reject that statement.

Typically a p-value of 0.05 is used as the threshold, however remember that it is best to report more than just the p-value, but also estimates and standard errors among other statistics. Different statistical tests allow for testing different hypotheses.

The `infer` Package

The `infer` package simplifies inference analyses. Users can quickly calculate a variety of statistics and perform statistical tests including those that require resampling, permutation, or simulations using data that is in `tidy` format.

In fact users can even perform analyses based on specified hypotheses with the `hypothesize()` function.

We will perform the same analysis about soda cans that we just did with this package to illustrate how to use it.

Recall that we wanted to know if the observed ounces of soda can differs from the expected mean of 12 ounces. Also recall that we had measurements for 100 soda cans (we made up this data). We had a testable statement or hypothesis that “soda cans typically have 12 ounces, the mean amount is 12” and we wanted to know if this was true.

This type of hypothesis is called a null hypothesis because it is a statement that expects no difference or change. The alternative hypothesis is the complement statement. It would be that the mean is not 12.

OK, so now we will use the `infer` package to test if our null hypothesis is true.

First, we need to get our data into a tidy format. Thus we will use the `as_tibble()` function of the `tidyverse` package.

```
soda_ounces <- as_tibble(soda_ounces)
soda_ounces
# A tibble: 100 × 1
  value
  <dbl>
1 12.0
2 12.0
3 12.0
4 12.0
5 12.0
6 12.0
7 12.0
8 12.0
9 12.0
10 12.0
# ... with 90 more rows
```

Now we will use the `specify()` function of the `infer` package to indicate that the `value` variable is our response variable that will be used in our hypothesis. This is as you might expect more important when we have multiple variables in our data. Then we can specify our null hypothesis with the `hypothesize()` function.

There are two options for the `null` argument of this function:

- 1) point - this option should be used when there is one variable in the hypothesis, such as “the mean of this data x”.
- 2) independence - this option should be used when there are two populations, such as “ the means of these two groups identical” or “this variable influences this other variable”.

Then if the point option is used, there are several additional arguments regarding what is being tested about that one variable. One can test a particular `mu` for mean, `med` for median, `sigma` for standard deviation, or `p` for the proportion of successes (for a categorical variable).

Our hypothesis was “the mean amount of soda ounces is 12” thus we will use the `point` option for our `null` argument and we will specify a mean with the `mu` argument as 12.

The major benefit of this package, besides allowing the user to think about the statistical analysis more than the programming required, is that the user can easily implement iterative methods like resampling.

What do we mean by this?

Resampling is a method where a random samples are drawn from the original data to create a dataset of the same size as the original data (but with some samples repeated) and this is done repetitively over and over. This process is called Bootstrapping. This provides more information about the confidence in our estimations from our sample about the true population that we are trying to investigate, as it gives us more of a sense of the range of values for statistics like mean and median might vary using other samples.

To perform resampling, users can use the `generate()` function with the `type` argument set to "bootstrap" and use the `rep` argument to specify how many bootstrap resamples to generate.

The `calculate()` function then allows for many different statistics to be calculated including:

- 1) `mean`
- 2) `median`
- 3) `sum`
- 4) `sd` for standard deviation
- 5) `prop` for proportion for categorical variables
- 6) `count`
- 7) `diff in means`
- 8) `diff in medians`
- 9) `diff in props`
- 10) `Chisq`
- 11) `F`
- 12) `slope`
- 13) `correlation`
- 14) `t`
- 15) `z`
- 16) `ratio of props`
- 17) `odds ratio`

Finally, the `get_confidence_interval()` as you might guess calculates a confidence interval. Now we will use these functions on our data.

```
library(infer)
set.seed(342)

CI <- soda_ounces %>%
  specify(response = value) %>%
  hypothesize(null = "point", mu = 12) %>%
  generate(rep = 1000, type = "bootstrap") %>%
  calculate(stat = "mean") %>%
  get_confidence_interval()

Using `level = 0.95` to compute confidence interval.

CI
# A tibble: 1 × 2
  lower_ci upper_ci
    <dbl>     <dbl>
1      12.0     12.0
```

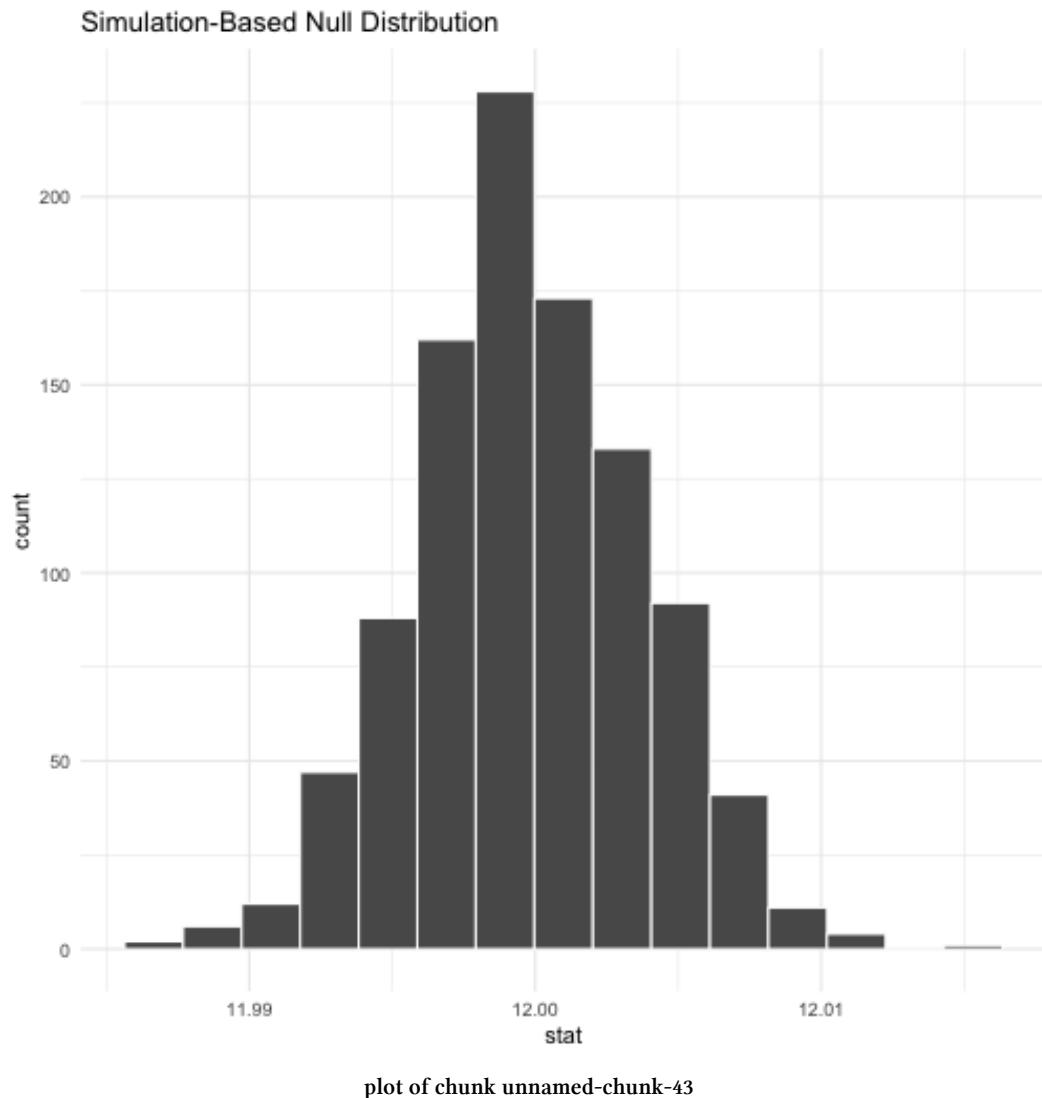
We can see that our confidence interval is very similar but slightly different from the results we obtained using the `t.test()` function and the `lm()` function. This is because we used a different method to calculate the confidence interval based on the bootstrap samples. Furthermore, the results will vary every time the code is run because the bootstrap samples are randomly created each time (unless you set a seed with `set.seed`).

We can also make a visualization of the null distribution of the bootstrap samples using the `visualize()` function.

```
set.seed(342)

bootstrap_means <- soda_ounces %>%
  specify(response = value) %>%
  hypothesize(null = "point", mu = 12) %>%
  generate(rep = 1000, type = "bootstrap") %>%
  calculate(stat = "mean")

bootstrap_means %>%
  visualize()
```



Prediction Modeling

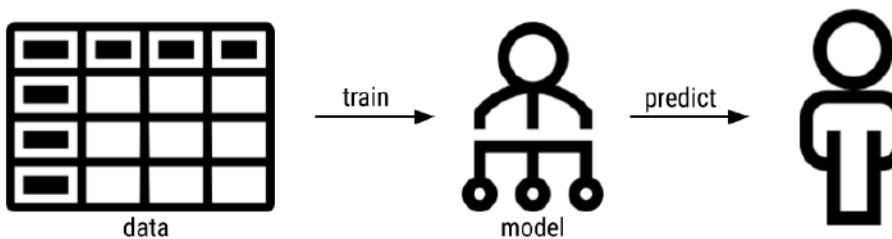
While the goal in inference is to learn something about the population, when we're talking about **prediction**, the focus is on the individual. The goal of predictive analysis and machine learning approaches is to **train a model using data** to make predictions about an individual.

In other words, the **goal of predictive analysis** is to use data you have now to make

predictions about future data.

We spend a lot of time trying to predict things in daily life- the upcoming weather, the outcomes of sports events, and the outcomes of elections. Using historical polling data and trends and current polling, Nate Silver's [FiveThirtyEight](#) builds models to predict the outcomes and the next US Presidential vote - and has been fairly accurate at doing so! FiveThirtyEight's models accurately predicted the 2008 and 2012 elections and was widely considered an outlier in the 2016 US elections, as it was one of the few models to suggest Donald Trump had a chance of winning.

Predicting the outcome of elections is a key example of predictive analysis, where historical data (data they have now) are used to predict something about the future.



What is Machine Learning?

So far we've been discussing predictive analysis. But, you may have heard people on the news or in daily life talking about "machine learning." The goal of machine learning in prediction is to build models (often referred to as algorithms) from the patterns in data that can be used for predictions in the future. Here, machine learning refers to using the relationships within a dataset to build a model that can be used for prediction.

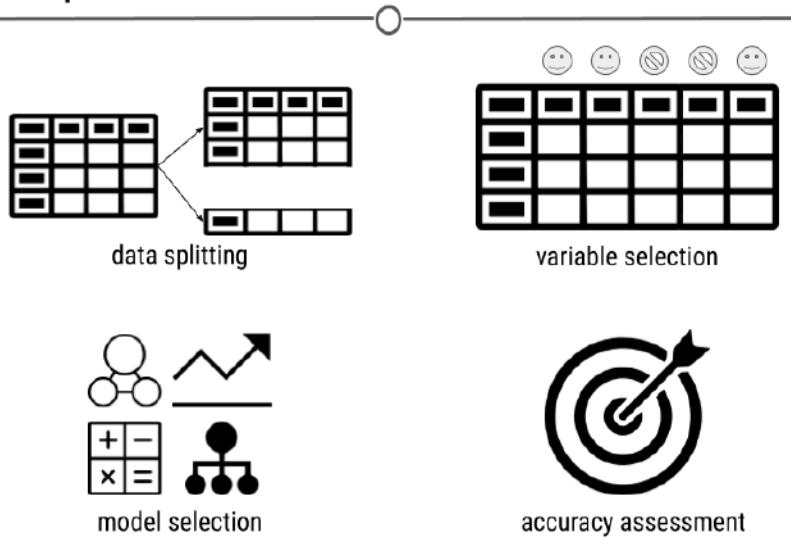
That said, there is without a doubt an entire field of individuals dedicating themselves to machine learning. This lesson will just touch on the very basics within the field.

Machine Learning Steps

In order to make predictions for the future using data you have now, there are four general steps:

- 1) Data Splitting - what data are you going to use to train your model? To tune your model? To test your model?
- 2) Variable Selection - what variable(s) from the data you have now are you going to use to predict future outcomes?
- 3) Model Selection - How are you going to model the data?
- 4) Accuracy Assessment - How are you going to assess accuracy of your predictions?

Basic Steps to Prediction



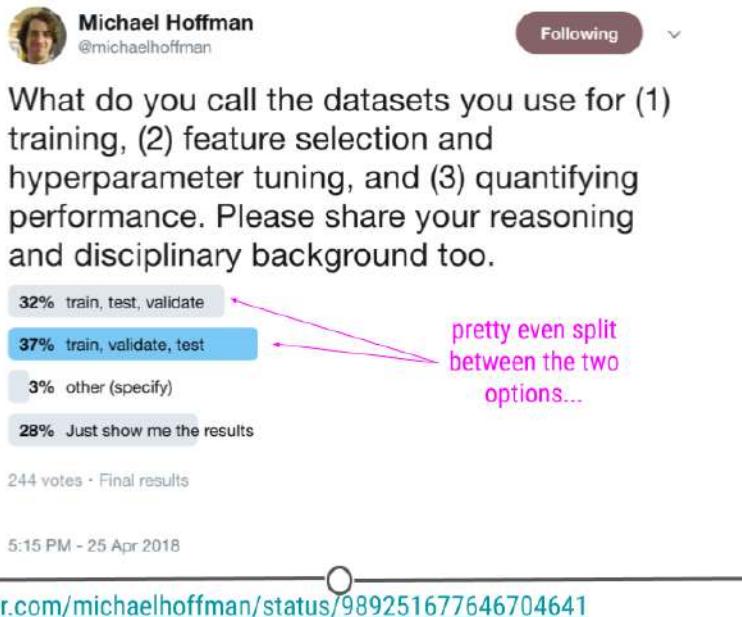
Data Splitting

For predictive analysis (or machine learning), you need data on which to train your model. These are the set of observations and corresponding variables that you're going to use to build your predictive model. But, a predictive model is only worth something if it can predict

accurately in a future dataset. Thus, often, in machine learning there are three datasets used to build a predictive model. The names for these datasets vary to some degree.

Train, Test, Validate

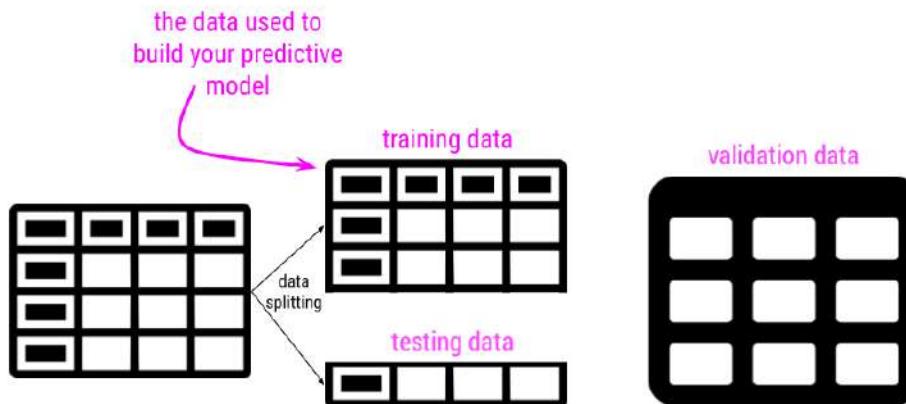
Many people use train, validate, and test. However, almost as many people use train, test, and validate, as evidenced by this Twitter poll:



In this lesson, we've decided to go with the terminology that fits the `tidymodels` terminology: train, test, validate.

Train

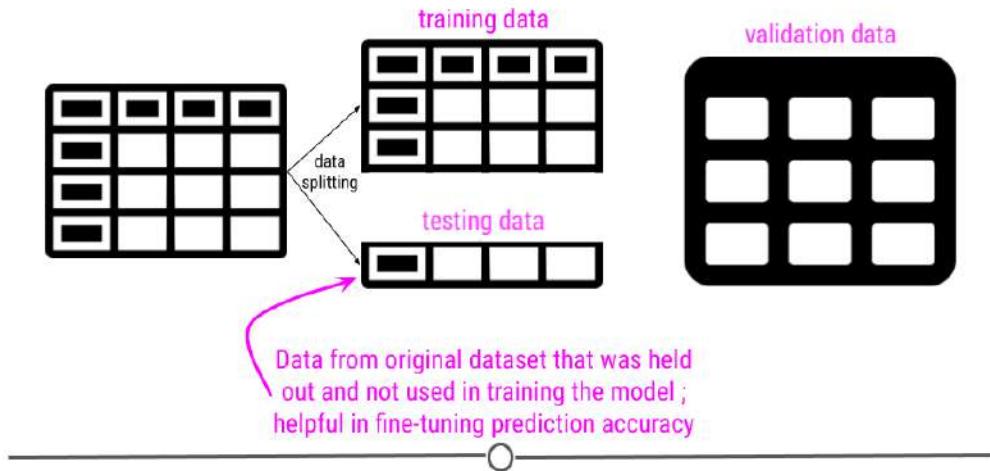
Training data are the data we described above. It is the data used to *build* your predictive model. These data are referred to as your training set.



Test

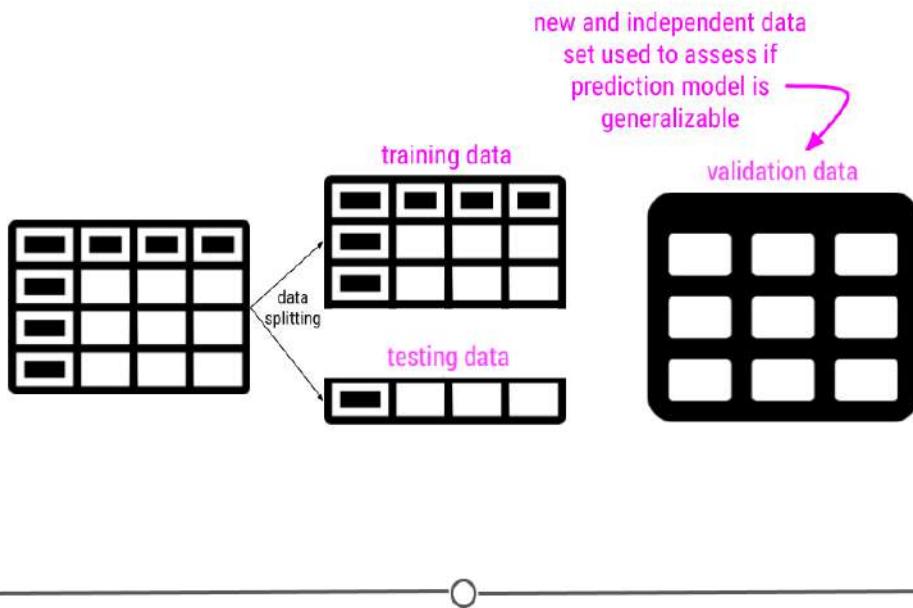
Before getting started, your original dataset is often split. Some (often 70%-75%) of the observations in your dataset are used to train the model, while 25%-30% are held out.

These hold-out samples are used to see whether or not your predictive model accurately makes predictions in the set of samples not used to train the model.



Validate

Finally, an independent dataset – one that is not from the same experiment or source as the data used to train and test your model are used to see whether or not your predictive model makes accurate predictions in a completely new dataset. Predictive models that can be generalized to and make accurate predictions in new datasets are the best predictive models.



Ultimately, we want to create a model that will perform well with any new data that we try to use. In other words we want the model to be generalizable so we can use it again to make predictions with new data. We don't want the model to only work well with the data that we used to train the model (this is called [overfitting](#)). Using a validation set helps us to assess how well our model might work with new data in the future. This is part of what we call [out-of-sample testing](#), as we evaluate the performance of the model on independent data that was not a part of the sample used to train the model.

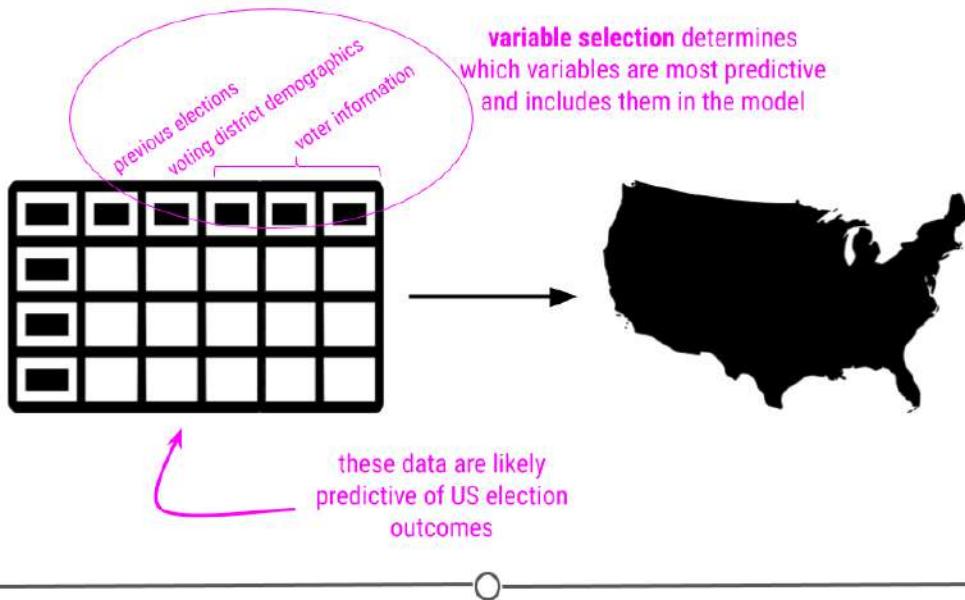
Variable Selection

For predictive analysis to be worth anything, you have to be able to predict an outcome accurately with the data you have on hand.

If all the data you have on hand are the heights of elephants in Asia, you're likely not going to be able to predict the outcome of the next US election. Thus, the variables in the data you have on hand have to be related to the outcome you're interested in predicting in some way (which is not the case for the heights of elephants and US elections).

Instead, to predict US elections, you'd likely want some data on outcomes of previous elections, maybe some demographic information about the voting districts, and maybe some information about the ages or professions of the people voting. All of these variables are likely

to be helpful in predicting the outcome in a future election, but which ones are actually predictive? All of them? Some of them? The process of deciding which variables to use for prediction is called **variable selection**.



You ideally want to include the *fewest variables* in your model as possible. Only having a few variables in your model avoids you having to collect a ton of data or build a really complicated model. But, you want the model to be as accurate as possible in making predictions. Thus, there's always a *balance* between minimizing the variables included (to only include the most predictive variables!) and maximizing your model's predictive accuracy. In other words, like in inferential analysis, your ability to make accurate predictions is dependent on whether or not you have measurements on the right variables. If you aren't measuring the right variables to predict an outcome, your predictions aren't going to be accurate. Thus, **variable selection**, is incredibly important.

All that said, there *are* machine learning approaches that carry out variable selection for you, using all the data to determine which variables in the dataset are most helpful for prediction. Nevertheless, whether you are deciding on the variables to include or the computer is deciding for you, variable selection is important to accurate prediction.

Lack of Causality Reminder

As a reminder, as was discussed in the inferential analysis, just because one variable may predict another, it *does not mean that one causes the other*. In predictive analysis, you are taking advantage of the relationship between two variables, using one variable (or *one* set of variables) to predict a second variable. Just because one variable accurately predicts another variable does *not* mean that they are causally related.

Model Selection

Additionally, there are many ways to generate prediction models. Each model was developed for a different and specific purpose. However, regardless of which model you choose to use for prediction, it's best to keep in mind that, in general, the **more data** you have and the **simpler your model is**, the best chance you have at accurately predicting future outcomes:

- More data - The more observations you have and the more variables you have to choose from to include in your model, the more likely you are to generate an accurate predictive model. Note, however, large datasets with lots of missing data or data that have been incorrectly entered are *not* better than small, complete, and accurate datasets. Having a trustworthy dataset to build your model is critical.
- Simple Models - If you can accurately predict an individual's height by only considering that person's parents height, then go for it. There's no need to include other variables if a single variable generates accurate predictions. A simple model that predicts accurately (regardless of the dataset in which you're predicting) is better than a complicated model.

Regression vs. Classification

Before we jump into discussing the various models you can use for predictive analysis, it's important to first note the difference between regression and classification. **Regression** is used when you're trying to predict a continuous variable. For example if you're trying to predict an individual's age, you would use regression. On the other hand, **classification** is used for categorical variables, as it predicts which *group* an individual belongs to. An example of a classification would be predicting someone's education level, as there are only a limited number of groups into which one would be.



Regression:
predicting continuous
variables
(i.e. Age)



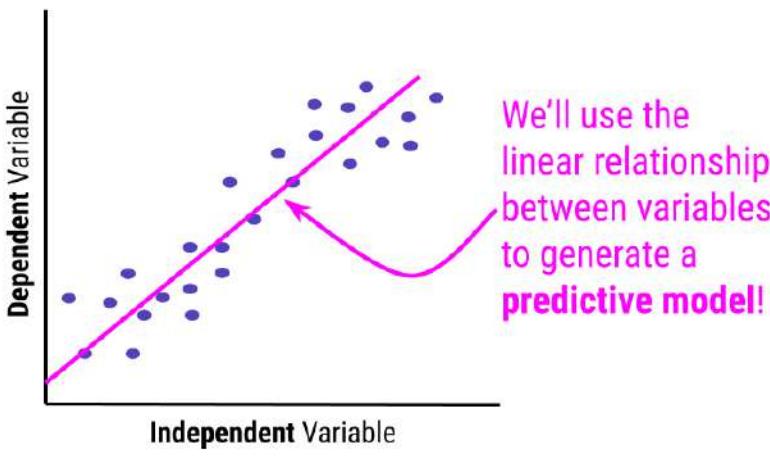
Classification:
predicting categorical
variables
(i.e. education level)

With regards to machine learning, certain methods can be used for both regression and classification, while others are designed exclusively for one or the other.

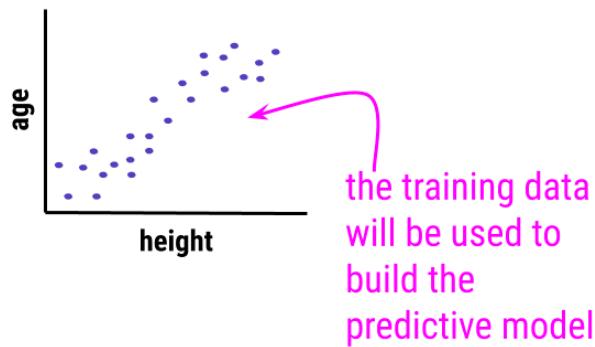
In this lesson we'll discuss one regression model and one classification model. However, there are literally [hundreds of models](#) available for predictive modeling. Thus, it's important to keep in mind that we're really just scratching the surface here.

Linear Regression

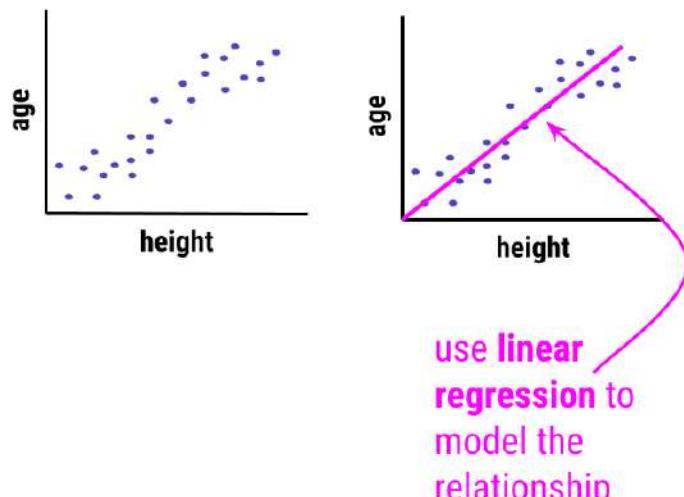
Just like in the previous lesson in inferential analysis, linear regression is an incredibly powerful method in machine learning! The concept here is the same as it was in the last lesson: we're going to capitalize on the linear relationship between variables. However, instead of using linear regression to estimate something about a larger population, we're going to use linear regression for prediction of a **continuous variable**.



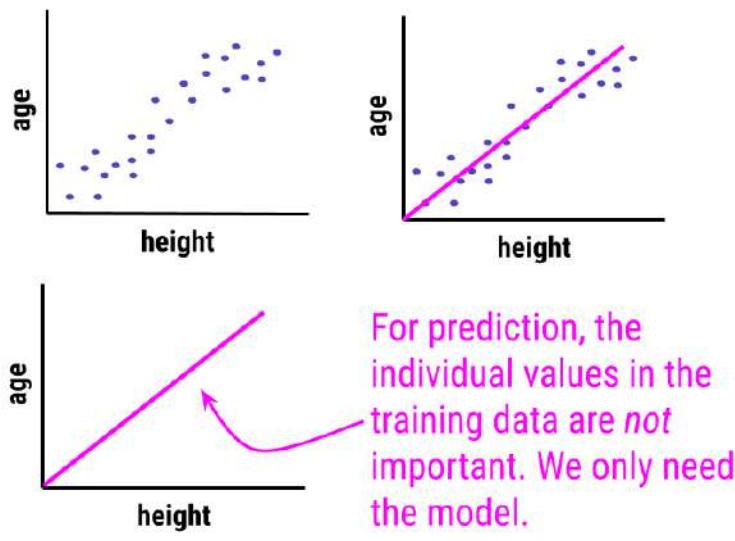
To better understand this, let's use a conceptual example. Consider trying to predict a child's age from their height. You'd likely expect that a taller child was older. So, let's imagine that we're looking here at the training data. We see the expected relationship between height and age in this scatterplot.



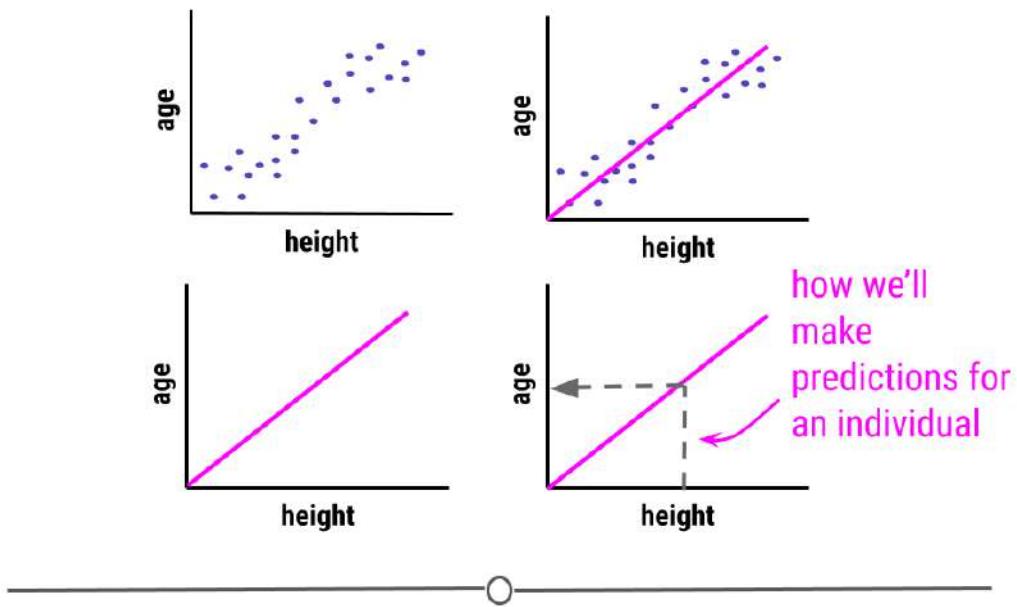
Using the training data, linear regression is then carried out to model the relationship.



Now that we have our model, we no longer care about the individual data points in the training data. We'll simply use the linear regression model to make our predictions.



Then, in the future when we know a child's height, we can return to our linear regression, supply it with the new child's height and it will return the child's age using the model we've built.



Conceptually, this is what will happen whenever we use linear regression for machine learning. However, it will be carried out mathematically, rather than graphically. This means you won't have to look on the graph to see your predictions. You'll just have to run a few lines of code that will carry out the necessary calculations to generate predictions.

Additionally, here we're using a single variable (height) to model age. Clearly, there are other variables (such as a child's sex) that could affect this prediction. Often, regression models will include multiple predictor variables that will improve prediction accuracy of the outcome variable.

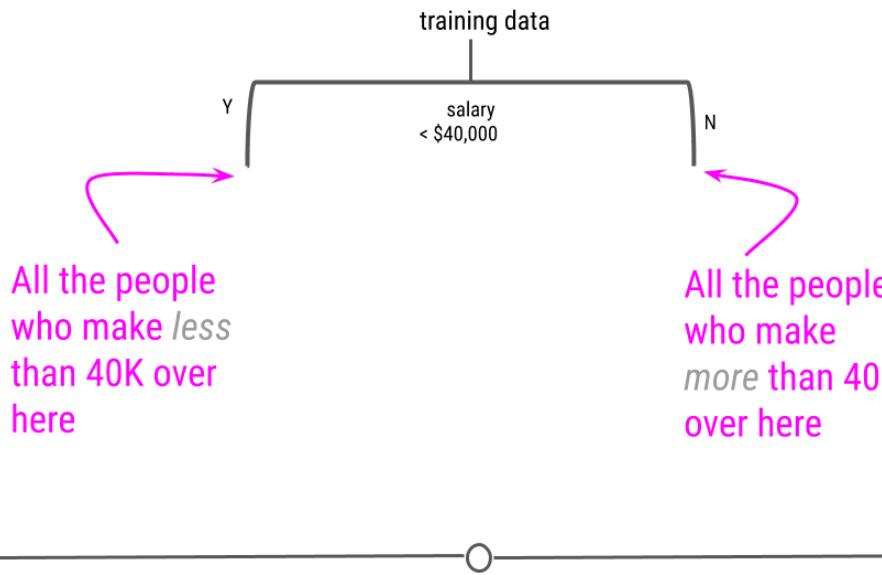
Classification and Regression Trees (CART)

Alternatively, when trying to predict a **categorical variable**, you'll want to look at classification methods, rather than regression (which is for continuous variables). In these cases you may consider using a **classification and regression tree (CART)** for prediction. While not the only classification method for machine learning, CARTs are a basic and commonly-used approach to prediction for categorical variables.

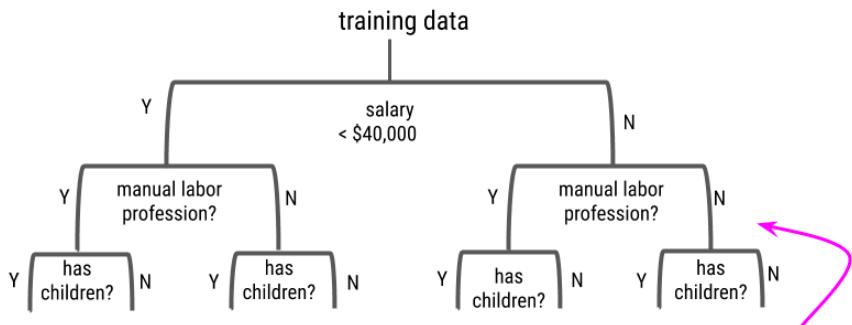
Conceptually, when using a CART for prediction, a **decision tree** is generated from the training data. A decision tree branches the data based on variables within the data. For example, if we were trying to predict an individual's education level, we would likely use

a dataset with information about many different people's income level, job title, and the number of children they have. These variables would then be used to generate the tree.

For example, maybe the first branch would separate individuals who make less than 40,000 dollars a year. All of those in the training data who made less than 40K would go down the left-hand branch, while everyone else would go down the right-hand branch.

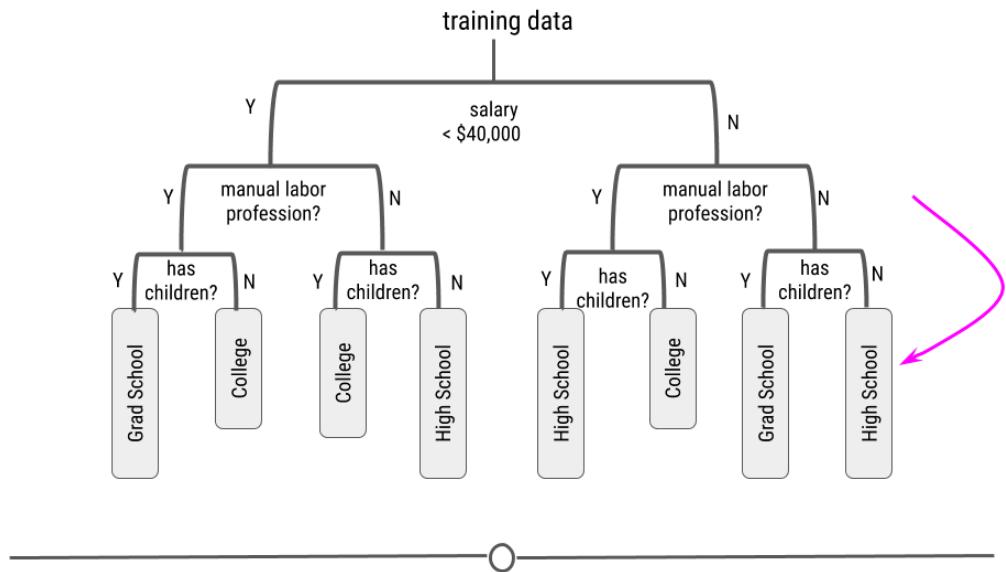


At each level, the data will continue to be split, using the information in the training data.

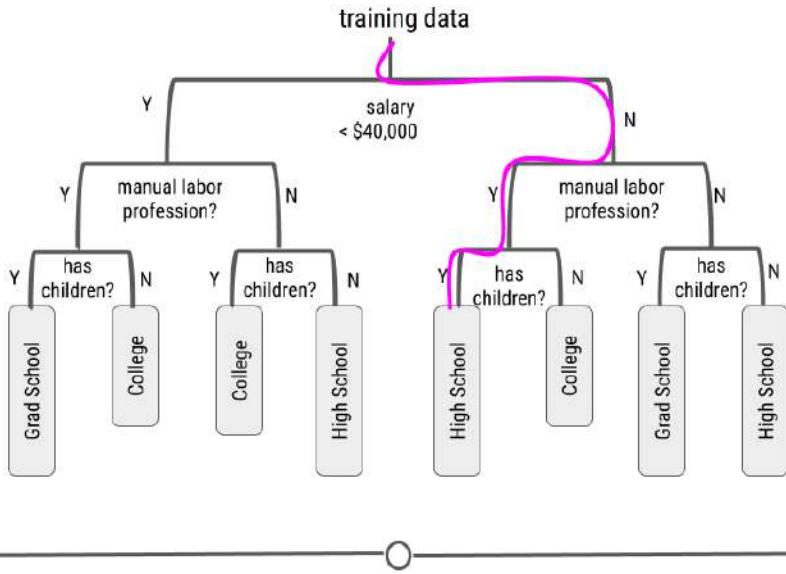


Continue building the decision tree where the variables and information in the training data decide who goes down which branch

Finally, a full decision tree will be constructed, such that there will be a label for the variable we're trying to predict at the end of each branch.



This CART will then be used for prediction in future samples. Thus, if you follow the path along the decision tree, for this example CART, an individual who made more than \$40,000 a year, was in a manual labor profession, and had children, this CART would predict that that individual's education level was "High School."



Again, this is conceptually and graphically how a CART works; however, when generating a CART yourself, it again only takes a few lines of code to generate the model and carry out the necessary math.

Model Accuracy

A common saying is that prediction is hard, especially about the future. This is true in predictive analysis. Thus, it's important to always carefully evaluate the accuracy of your model and to never overstate how well you are able to make predictions.

Generally, if your predictions are correct, you're doing well! If your predictions are wrong, you're not doing as well. But, how do we define "well"?

Error Rates

To assess whether or not our predictive models are doing well, we calculate error rates. There are metrics used to measure model performance, however, the two most common ways to assess how well our predictive models are doing are:

- 1) RMSE (Root-mean-square Error)
- 2) Accuracy

We'll note here that in order to assess error, you have to know the truth (the actual value) in addition to the predicted value. Thus, RMSE and accuracy are assessed in the training and tuning data, where you *know* the actual value as well as the predicted value.

RMSE

The root-mean-square error (RMSE) is a measure used to assess prediction error for continuous variables. Generally, we want to minimize error in prediction. Thus, a small RMSE is better than a large RMSE.

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (Predicted_i - Actual_i)^2}{N}}$$

Mathematically speaking, the RMSE is the square root of the variance. From earlier, we know that **variance** has something to do with how confident we are in our estimate. Since we're trying to determine how close our predictions are to the actual value, this seems like a good place to start.

When we look at the equation, we can see that the difference between the predicted and actual values is calculated (*Predicted* - *Actual*) and that this value is then squared (*Predicted* - *Actual*)². These differences squared are then added for every individual in your dataset (that's what the sigma, or big E says). This value (the sum of all the errors squared) is then divided by the number of individuals in your dataset (N). This square root of this value is then taken. This is how RMSE is calculated.

We went through that description because we want to point out that when differences are squared ($\text{Predicted} - \text{Actual}$)², outliers, or samples whose prediction was far off from their actual value are going to increase the RMSE a lot. Thus, *a few outliers can lead to really high RMSE values*, even if all the other predictions were pretty good. This means it's important to check to see if a few outliers (meaning a few bad predictions) are leading to a high RMSE value.

Accuracy

Alternatively, to assess error in the prediction of categorical variables, **accuracy** is frequently used. Accuracy looks to determine the number of predictions that match their actual values.

$$\text{Accuracy} = \frac{\text{\# of samples predicted correctly}}{\text{\# of samples predicted}} * 100$$

The closer this value is to 100%, the better your predictive model was. The closer to 0%, the worse your model's predictions are.

Accuracy is a helpful way to assess error in categorical variables, but it can be used for numeric values too. However, it will only account a prediction "correct" if it matches exactly. In the case of age, if a sample's age is 10 and the model predicts it to be 10, the model will say it's been predicted correctly. However, if a sample's age is 10 and it is predicted to be 9, it will be counted as incorrect, even though it was close. A prediction off by a year will be marked just as incorrect as a sample predicted off by 50 years. Due to this, RMSE is often opted for instead of accuracy for continuous variables.

It is also important to realize that having low RMSE or high accuracy with your training does not necessarily indicate that your model is generalizable. It indicates that your model works well with your training data. However if your model fits your training data too well it may not be very good at predicting with new data. This is why we have a subset of the original data to use to test our model.

The `tidymodels` Ecosystem

There are *incredibly* helpful packages available in R thanks to the work of [Max Kuhn](#) at RStudio. As mentioned above, there are hundreds of different machine learning algorithms. Max's R packages have put many of them into a single framework, allowing you to use *many* different machine learning models easily. Additionally, he has written a very [helpful book](#) about predictive modeling. There are also many [helpful links](#) about each of the packages. Max previously developed the `caret` package (short for Classification And Regression Training) which has been widely used. [Here](#) you can see some of the discussion about the difference between `caret` and `tidymodels`.

In this [rstudio community thread](#) you can see that Max stated that “The tidyverse is more about modular packages that are designed to play well with one another. The main issue with caret is that, being all in one package, it is very difficult to extend it into areas that people are interested in...The bottom line is that the `tidymodels` set should do what `caret` does and more.” We will describe some of the advantages of the `tidymodels` packages.

Benefits of `tidymodels`

The two major benefits of `tidymodels` are:

1. Standardized workflow/format/notation across different types of machine learning algorithms

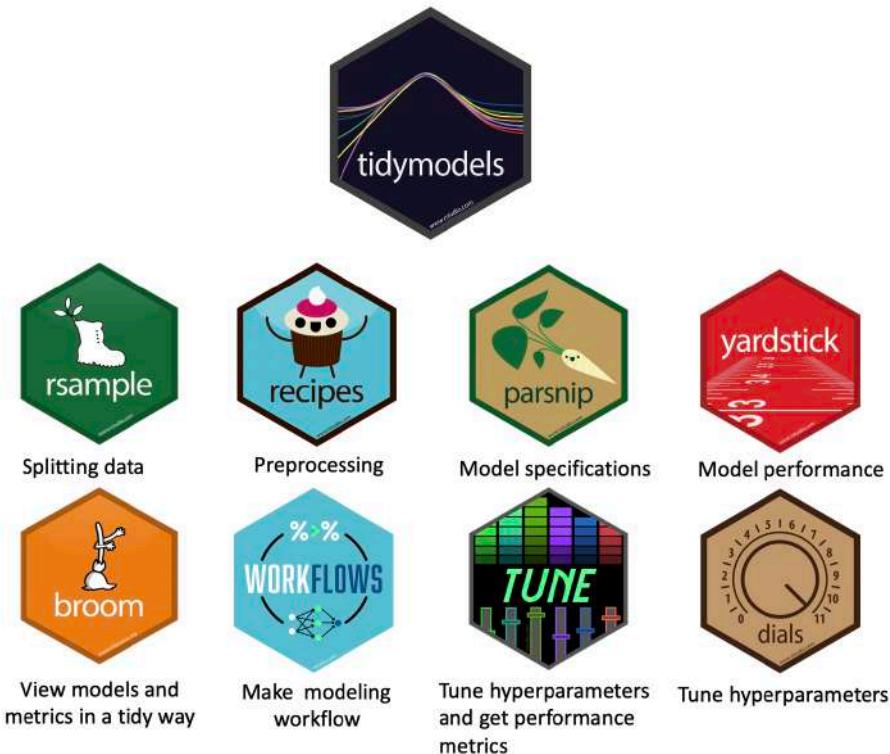
Different notations are required for different algorithms as the algorithms have been developed by different people. This would require the painstaking process of reformatting the data to be compatible with each algorithm if multiple algorithms were tested.

1. Can easily modify preprocessing, algorithm choice, and hyperparameter tuning making optimization easy

Modifying a piece of the overall process is now easier than before because many of the steps are specified using the `tidymodels` packages in a convenient manner. Thus the entire process can be rerun after a simple change to preprocessing without much difficulty.

Packages of `tidymodels`

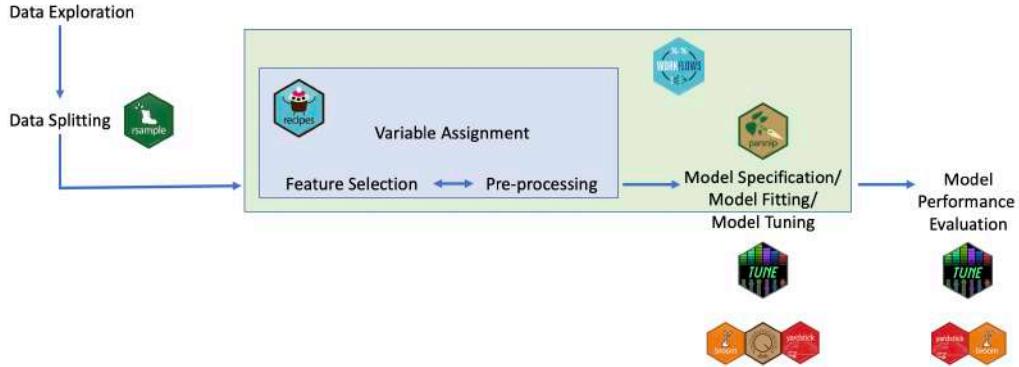
We will focus on the following packages although there are many more in the `tidymodels` ecosystem:



- 1) `rsamples` - to split the data into training and testing sets (as well as cross validation sets - more on that later!)
- 2) `recipes` - to prepare the data with preprocessing (assign variables and preprocessing steps)
- 3) `parsnip` - to specify and fit the data to a model
- 4) `yardstick` and `tune` - to evaluate model performance
- 5) `workflows` - combining `recipe` and `parsnip` objects into a workflow (this makes it easier to keep track of what you have done and it makes it easier to modify specific steps)
- 6) `tune` and `dials` - model optimization (more on what hyperparameters are later too!)

7) `broom` - to make the output from fitting a model easier to read

Here you can see a visual of how these packages work together in the process of performing a machine learning analysis:



To illustrate how to use each of these packages, we will work through some examples.

These are the major steps that we will cover in addition to some more advanced methods:

Overview of <i>tidymodels</i> Basics		
Package	Step	Functions
	1. Split into testing and training sets	<code>initial_split()</code> <code>training()</code> <code>testing()</code>
	2. Create recipe + assign variable roles	<code>recipe()</code> <code>update_role()</code>
	3. Specify model, engine, and mode	<code>parsnip</code> function for specifying model (ex. <code>decision_tree()</code>) (https://www.tidymodels.org/find/parsnip/) <code>set_engine()</code> <code>set_mode()</code>
	4. Create workflow, add recipe, add model	<code>workflow()</code> <code>add_recipe()</code> <code>add_model()</code>
	5. Fit workflow	<code>fit()</code>
	6. Get predictions	<code>predict()</code>
	7. Use predictions to get performance metrics	<code>rmse()</code> (continuous outcome) <code>accuracy()</code> (categorical outcome) <code>metrics()</code> (either type of outcome)

Other *tidymodels* packages include:



[[source](#)]

- 1) `applicable` compares new data points with the training data to see how much the new data points appear to be an extrapolation of the training data
- 2) `baguette` is for speeding up bagging pipelines
- 3) `butcher` is for dealing with pipelines that create model objects that take up too much memory
- 4) `discrim` has more model options for classification
- 5) `embed` has extra preprocessing options for categorical predictors
- 6) `hardhat` helps you to make new modeling packages
- 7) `corrr` has more options for looking at correlation matrices
- 8) `rules` has more model options for prediction rule ensembles

- 9) `text_recipes` has extra preprocessing options for using text data
- 10) `tidypredict` is for running predictions inside SQL databases
- 11) `modeldb` is also for working within SQL databases and it allows for `dplyr` and `tidyeval` use within a database
- 12) `tidybayesian` compares models using resampling statistics

Most of these packages offer advanced modeling options and we will not be covering how to use them.

Example of Continuous Variable Prediction

For this example, we'll keep it simple and use a dataset you've seen before: the `iris` dataset. This way you can focus on the syntax used in the `tidymodels` packages and the steps of predictive analysis. In this example, we'll attempt to use the data in the `iris` dataset to predict `Sepal.Length`.

Step 1: Example of Data Splitting with `rsample`

As mentioned above, one of the first steps is to take your dataset and split it into a training set and a testing set. To do this, we'll load the `rsample` package and use the `initial_split()` function to split the dataset.

We can specify what proportion of the data we would like to use for training using the `prop` argument.

Since the split is performed randomly, it is a good idea to use the `set.seed()` function in base R to ensure that if you rerun your code that your split will be the same next time.

```
library(rsample)
set.seed(1234)
split_iris <- initial_split(iris, prop = 2/3)
split_iris
<Analysis/Assess/Total>
<100/50/150>
# the default proportion is 1/4 testing and 3/4 training
```

This results in printing the number of training data rows, the number testing data rows, and the total rows - each is printed with the “/” as a division between the values. Here the training set is called the analysis set, while the testing set is called the assess set.

We can see that about 70% of our observations are in the training dataset and the other 30% are in the tuning dataset, as we specified.

We can then extract the training and testing datasets by using the `training()` and `testing()` functions, also of the `rsample` package.

```
training_iris <- training(split_iris)
head(training_iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
28          5.2       3.5        1.5      0.2   setosa
80          5.7       2.6        3.5      1.0 versicolor
101         6.3       3.3        6.0      2.5  virginica
111         6.5       3.2        5.1      2.0  virginica
137         6.3       3.4        5.6      2.4  virginica
133         6.4       2.8        5.6      2.2  virginica

testing_iris <- testing(split_iris)
head(testing_iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1       3.5        1.4      0.2   setosa
7           4.6       3.4        1.4      0.3   setosa
11          5.4       3.7        1.5      0.2   setosa
12          4.8       3.4        1.6      0.2   setosa
15          5.8       4.0        1.2      0.2   setosa
16          5.7       4.4        1.5      0.4   setosa
```

Step 2: Example of preparing for preprocessing the data with `recipes`

After splitting the data, the next step is to process the training and testing data so that the data are compatible and optimized to be used with the model. This involves assigning variables to specific roles within the model and preprocessing like scaling variables and removing redundant variables. This process is also called feature engineering.

To do this in `tidymodels`, we will create what's called a "recipe" using the `recipes` package, which is a standardized format for a sequence of steps for preprocessing the data. This can be very useful because it makes testing out different preprocessing steps or different algorithms with the same preprocessing very easy and reproducible.

Creating a recipe specifies **how a dataframe of predictors should be created** - it specifies what variables to be used and the preprocessing steps, but it **does not execute these steps** or create the dataframe of predictors.

Step 1: Specify variables with the `recipe()` function

The first thing to do to create a recipe is to specify which variables we will be using as our outcome and predictors using the `recipe()` function. In terms of the metaphor of baking, we can think of this as listing our ingredients. Translating this to the `recipes` package, we use the `recipe()` function to assign roles to all the variables.

We can do so in two ways:

1. Using formula notation
2. Assigning roles to each variable

Let's look at the first way using formula notation, which looks like this:

`outcome(s) ~ predictor(s)`

If in the case of multiple predictors or a multivariate situation with two outcomes, use a plus sign:

`outcome1 + outcome2 ~ predictor1 + predictor2`

If we want to include all predictors we can use a period like so:

`outcome_variable_name ~ .`

Let's make our first recipe with the `iris` data! We will first try to predict `Sepal.Length` in our training data based on `Sepal.Width` and the `Species`. Thus, `Sepal.Length` is our outcome variable and `Sepal.Width` and `Species` are our predictor variables.

First we can specify our variables using formula notation:

```
library(recipes)
```

```
Attaching package: 'recipes'
```

```
The following object is masked from 'package:stringr':
```

```
fixed
```

```
The following object is masked from 'package:stats':
```

```
step
```

```
first_recipe <- training_iris %>%
```

```
  recipe(Sepal.Length ~ Sepal.Width + Species)
```

```
first_recipe
```

Data Recipe

Inputs:

```
role #variables
outcome      1
predictor    2
```

Alternatively, we could also specify the outcome and predictor(s) by assigning roles to the variables by using the `update_role()` function. Please see [here](#) for examples of the variety of roles variables can take.

We first need to use the `recipe()` function with this method to specify what data we are using.

```
first_recipe <- recipe(training_iris) %>%
  recipes::update_role(Sepal.Length, new_role = "outcome") %>%
  recipes::update_role(Sepal.Width, new_role = "predictor") %>%
  recipes::update_role(Species, new_role = "predictor")
first_recipe
Data Recipe
```

Inputs:

```
role #variables
outcome      1
predictor    2

2 variables with undeclared roles
```

We can view our recipe using the base `summary()` function.

```
summary(first_recipe)
# A tibble: 5 × 4
  variable     type    role    source
  <chr>       <chr>   <chr>   <chr>
1 Sepal.Length numeric outcome original
2 Sepal.Width  numeric predictor original
3 Petal.Length numeric <NA>      original
4 Petal.Width  numeric <NA>      original
5 Species      nominal predictor original
```

Step 2: Specify the preprocessing steps with `step*()` functions

Next, we use the `step*()` functions from the `recipe` package to specify preprocessing steps.

This [link](#) and this [link](#) show the many options for recipe step functions.

There are step functions for a variety of purposes:

1. **Imputation** – filling in missing values based on the existing data
2. **Transformation** – changing all values of a variable in the same way, typically to make it more normal or easier to interpret
3. **Discretization** – converting continuous values into discrete or nominal values - binning for example to reduce the number of possible levels (However this is generally not advisable!)
4. **Encoding / Creating Dummy Variables** – creating a numeric code for categorical variables ([More on Dummy Variables and one-hot encoding](#))
5. **Data type conversions** – which means changing from integer to factor or numeric to date etc.
6. **Interaction** term addition to the model – which means that we would be modeling for predictors that would influence the capacity of each other to predict the outcome
7. **Normalization** – centering and scaling the data to a similar range of values
8. **Dimensionality Reduction/ Signal Extraction** – reducing the space of features or predictors to a smaller set of variables that capture the variation or signal in the original variables (ex. Principal Component Analysis and Independent Component Analysis)
9. **Filtering** – filtering options for removing variables (ex. remove variables that are highly correlated to others or remove variables with very little variance and therefore likely little predictive capacity)
10. **Row operations** – performing functions on the values within the rows (ex. rearranging, filtering, imputing)

11. Checking functions – Sanity checks to look for missing values, to look at the variable classes etc.

All of the step functions look like `step_*`() with the * replaced with a name, except for the check functions which look like `check_*`().

There are several ways to select what variables to apply steps to:

1. Using `tidyselect` methods: `contains()`, `matches()`, `starts_with()`, `ends_with()`, `everything()`, `num_range()`
2. Using the type: `all_nominal()`, `all_numeric()`, `has_type()`
3. Using the role: `all_predictors()`, `all_outcomes()`, `has_role()`
4. Using the name - use the actual name of the variable/variables of interest

Let's try adding a preprocessing step to our recipe.

We might want to potentially one-hot encode some of our categorical variables so that they can be used with certain algorithms like a linear regression require numeric predictors.

We can do this with the `step_dummy()` function and the `one_hot = TRUE` argument. One-hot encoding means that we do not simply encode our categorical variables numerically, as our numeric assignments can be interpreted by algorithms as having a particular rank or order. Instead, binary variables made of 1s and 0s are used to arbitrarily assign a numeric value that has no apparent order.

```
first_recipe <- first_recipe %>%  
  step_dummy(Species, one_hot = TRUE)
```

```
first_recipe  
Data Recipe
```

Inputs:

```
role #variables  
outcome      1  
predictor    2  
  
2 variables with undeclared roles
```

Operations:

Dummy variables from Species

Step 3: Example of optionally performing the preprocessing to see how it influences the data

Optionally one can use the `prep()` function of the `recipes` package to update the recipe for manually performing the preprocessing to see how this influences the data. This step is however not required when using the `workflows` package. The preprocessed training data can than be viewed by using the `bake()` function with the `new_data = NULL` argument, while preprocessed testing data can be viewed using the `bake()` function and specifying that the testing data is the `new_data`.

The `prep()` function estimates parameters (estimating the required quantities and statistics required by the steps for the variables) for preprocessing and updates the variables roles, as sometimes predictors may be removed, this allows the recipe to be ready to use on other datasets.

It does not necessarily actually execute the preprocessing itself, however we will specify using the `retain` argument for it to do this so that we can take a look at the preprocessed data.

There are some important arguments to know about:

1. `training` - you must supply a training dataset to estimate parameters for preprocessing operations (recipe steps) - this may already be included in your recipe - as is the case for us
2. `fresh` - if `fresh=TRUE`, - will retrain and estimate parameters for any previous steps that were already prepped if you add more steps to the recipe
3. `verbose` - if `verbose=TRUE`, shows the progress as the steps are evaluated and the size of the preprocessed training set
4. `retain` - if `retain=TRUE`, then the preprocessed training set will be saved within the recipe (as template). This is good if you are likely to add more steps and do not want to rerun the `prep()` on the previous steps. However this can make the recipe size large. This is necessary if you want to actually look at the preprocessed data.

Let's try out the `prep()` function:

```
prepped_rec <- prep(first_recipe, verbose = TRUE, retain = TRUE )
oper 1 step dummy [training]
The retained training set is ~ 0.01 Mb in memory.
prepped_rec
Data Recipe
```

Inputs:

```
role #variables
outcome      1
predictor    2

2 variables with undeclared roles
```

Training data contained 100 data points and no missing data.

Operations:

```
Dummy variables from Species [trained]
names(prepped_rec)
[1] "var_info"        "term_info"       "steps"          "template"
[5] "retained"        "tr_info"         "orig_lvls"     "last_term_info"
```

There are also lots of useful things to checkout in the output of `prep()`. You can see:

1. the steps that were run
2. the original variable info (`var_info`)
3. the updated variable info after preprocessing (`term_info`)
4. the new levels of the variables
5. the original levels of the variables (`orig_lvls`)
6. info about the training dataset size and completeness (`tr_info`)

We can see these using the \$ notation:

```
prepped_rec$var_info
# A tibble: 5 × 4
  variable     type   role    source
  <chr>       <chr>  <chr>   <chr>
1 Sepal.Length numeric outcome original
2 Sepal.Width  numeric predictor original
3 Petal.Length numeric <NA>      original
4 Petal.Width  numeric <NA>      original
5 Species      nominal predictor original
```

Now we can use `bake` to see the preprocessed training data.

Note: this used to require the `juice()` function.

Since we are using our training data we need to specify that we don't have `new_data` with `new_data = NULL`.

```
preproc_train <- recipes::bake(prepped_rec, new_data = NULL)
glimpse(preproc_train)
Rows: 100
Columns: 7
$ Sepal.Length      <dbl> 5.2, 5.7, 6.3, 6.5, 6.3, 6.4, 6.8, 7.9, 6.2, 7.1, ...
$ Sepal.Width       <dbl> 3.5, 2.6, 3.3, 3.2, 3.4, 2.8, 3.2, 3.8, 2.9, 3.0, ...
$ Petal.Length      <dbl> 1.5, 3.5, 6.0, 5.1, 5.6, 5.6, 5.9, 6.4, 4.3, 5.9, ...
$ Petal.Width       <dbl> 0.2, 1.0, 2.5, 2.0, 2.4, 2.2, 2.3, 2.0, 1.3, 2.1, ...
$ Species_setosa    <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, ...
$ Species_versicolor <dbl> 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, ...
$ Species_virginica <dbl> 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, ...
```

We can see that the `Species` variable has been replaced by 3 variables representing the 3 different species numerically with zeros and ones.

Now we do the same for our testing data using `bake()`. You generally want to leave your testing data alone, but it is good to look for issues like the introduction of NA values if you have complicated preprocessing steps and you want to make sure this performs as you expect.

```
baked_test_pm <- recipes::bake(prepped_rec, new_data = testing_iris)
glimpse(baked_test_pm)
Rows: 50
Columns: 7
$ Sepal.Length      <dbl> 5.1, 4.6, 5.4, 4.8, 5.8, 5.7, 5.1, 4.6, 5.1, 5.2, ...
$ Sepal.Width       <dbl> 3.5, 3.4, 3.7, 3.4, 4.0, 4.4, 3.5, 3.6, 3.3, 3.4, ...
$ Petal.Length      <dbl> 1.4, 1.4, 1.5, 1.6, 1.2, 1.5, 1.4, 1.0, 1.7, 1.4, ...
$ Petal.Width       <dbl> 0.2, 0.3, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.5, 0.2, 0...
$ Species_setosa    <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ Species_versicolor <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ Species_virginica <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

Great! Now back to the typical steps.

Step 4: Example of specifying the model with `parsnip`

So far we have used the packages `rsample` to split the data and `recipes` to assign variable types, and to specify and prep our preprocessing (as well as to optionally extract the preprocessed data).

We will now use the `parsnip` package (which is similar to the previous `caret` package - and hence why it is named after the vegetable) to specify our model.

There are four things we need to define about our model:

1. The **type** of model (using specific functions in `parsnip` like `rand_forest()`, `logistic_reg()` etc.)
2. The package or **engine** that we will use to implement the type of model selected (using the `set_engine()` function)
3. The **mode** of learning - classification or regression (using the `set_mode()` function)
4. Any **arguments** necessary for the model/package selected (using the `set_args()` function - for example the `mtry` = argument for random forest which is the number of variables to be used as options for splitting at each tree node)

Let's walk through these steps one by one. For our case, we are going to start our analysis with a linear regression but we will demonstrate how we can try different models.

The first step is to define what type of model we would like to use. See [here](#) for modeling options in `parsnip`.

We want to do a linear regression so we will use the `linear_reg()` function of the `parsnip` package.

```
Lin_reg_model <- parsnip::linear_reg()  
Lin_reg_model  
Linear Regression Model Specification (regression)  
  
Computational engine: lm
```

OK. So far, all we have defined is that we want to use a linear regression. Now let's tell `parsnip` more about what we want.

We would like to use the [ordinary least squares](#) method to fit our linear regression. So we will tell `parsnip` that we want to use the `lm` package to implement our linear regression (there are many options actually such as `rstan`, `glmnet`, `keras`, and `sparklyr`). See [here](#) for a description of the differences and using these different engines with `parsnip`.

We will do so by using the `set_engine()` function of the `parsnip` package.

```
Lin_reg_model <-  
  Lin_reg_model %>%  
  parsnip::set_engine("lm")  
  
Lin_reg_model  
Linear Regression Model Specification (regression)  
  
Computational engine: lm
```

Some packages can do either classification or regression, so it is a good idea to specify which mode you intend to perform. Here, we aim to predict a continuous variable, thus we want to perform a regression analysis. You can do this with the `set_mode()` function of the `parsnip` package, by using either `set_mode("classification")` or `set_mode("regression")`.

```
Lin_reg_model <-  
  Lin_reg_model %>%  
  parsnip::set_engine("lm") %>%  
  parsnip::set_mode("regression")  
  
Lin_reg_model  
Linear Regression Model Specification (regression)  
  
Computational engine: lm
```

Step 5: Example of fitting the model

We can use the `parsnip` package with a newer package called `workflows` to fit our model.

The `workflows` package allows us to keep track of both our preprocessing steps and our model specification. It also allows us to implement fancier optimizations in an automated way and it can also handle post-processing operations.

We begin by creating a workflow using the `workflow()` function in the `workflows` package.

Next, we use `add_recipe()` (our preprocessing specifications) and we add our model with the `add_model()` function – both functions from the `workflows` package.

Note: We do not need to actually `prep()` our recipe before using `workflows` - this was just optional so we could take a look at the preprocessed data!

```
iris_reg_wflow <- workflows::workflow() %>%  
  workflows::add_recipe(first_recipe) %>%  
  workflows::add_model(Lin_reg_model)  
  
iris_reg_wflow  
= Workflow ━━━━━━\\  
  ━━━━  
  Preprocessor: Recipe  
  Model: linear_reg()  
  
— Preprocessor ━━━━━━\\  
  ━━━━  
  1 Recipe Step  
    • step_dummy()
```

```
— Model ━━━━━━＼  
—  
Linear Regression Model Specification (regression)  
Computational engine: lm
```

Ah, nice. Notice how it tells us about both our preprocessing steps and our model specifications.

Next, we “prepare the recipe” (or estimate the parameters) and fit the model to our training data all at once. Printing the output, we can see the coefficients of the model.

```
iris_reg_wf_fit <- parsnip::fit(iris_reg_wf, data = training_iris)  
iris_reg_wf_fit  
= Workflow [trained] ━━━━━━＼  
—  
Preprocessor: Recipe  
Model: linear_reg()  
  
— Preprocessor ━━━━━━＼  
—  
1 Recipe Step  


- step_dummy()

  
— Model ━━━━━━＼  
—  
Call:  
stats::lm(formula = ..y ~ ., data = data)  
  
Coefficients:  
              (Intercept)          Sepal.Width      Species_setosa  Species_versicolor  
                4.4406            0.7228           -1.9013            -0.5232  
Species_virginica  
                    NA
```

Step 6: Example of assessing the model performance

Recall that often for regression analysis we use the RMSE to assess model performance.

To get this we first need to get the predicted (also called “fitted”) values.

We can get these values using the `pull_workflow_fit()` function of the `workflows` package. These values are in the `fitfitted.values` slot of the output. Alternatively, we can use the `predict()` function with the workflow and the training data specified as the `new_data`.

```
library(workflows)
wf_fit <- iris_reg_wflow_fit %>%
  pull_workflow_fit()
FALSE Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.
FALSE Please use `extract_fit_parsnip()` instead.

head(wf_fit$fit$fitted.values)
FALSE      1      2      3      4      5      6
FALSE 5.069194 5.796771 6.825955 6.753671 6.898239 6.464535

predict(iris_reg_wflow_fit, new_data = training_iris)
FALSE Warning in predict.lm(object = object$fit, newdata = new_data, type =
FALSE "response"): prediction from a rank-deficient fit may be misleading
FALSE # A tibble: 100 × 1
FALSE   .pred
FALSE   <dbl>
FALSE  1  5.07
FALSE  2  5.80
FALSE  3  6.83
FALSE  4  6.75
FALSE  5  6.90
FALSE  6  6.46
FALSE  7  6.75
FALSE  8  7.19
FALSE  9  6.01
FALSE 10  6.61
FALSE # ... with 90 more rows
```

To get more information about the prediction for each sample, we can use the `augment()` function of the `broom` package. This requires using the preprocessed training data from `bake()`

(or with previous versions `juice()`), as well as the predicted values from either of the two previous methods.

```
wf_fitted_values <-
  broom::augment(wf_fit$fit, data = preproc_train) %>%
  select(Sepal.Length, .fitted:.std.resid)

head(wf_fitted_values)
#> # A tibble: 6 × 6
#>   FALSE Sepal.Length .fitted    .hat .sigma  .cooks.d .std.resid
#>   FALSE      <dbl>     <dbl>    <dbl>  <dbl>      <dbl>
#> 1 FALSE       5.2      5.07  0.0336  0.459  0.000738   0.292
#> 2 FALSE       5.7      5.80  0.0340  0.459  0.000409  -0.216
#> 3 FALSE       6.3      6.83  0.0362  0.456  0.0129  -1.17
#> 4 FALSE       6.5      6.75  0.0315  0.458  0.00259  -0.565
#> 5 FALSE       6.3      6.90  0.0429  0.455  0.0201  -1.34
#> 6 FALSE       6.4      6.46  0.0317  0.459  0.000169  -0.144

# other option:
# wf_fitted_values <-
#   broom::augment(predict(iris_reg_wflow_fit, new_data = training_iris),
#   data = preproc_train) %>%
#   select(Sepal.Length, .fitted:.std.resid)
#
# head(wf_fitted_values)
```

Nice, now we can see what the original value for `Sepal.Length` right next to the predicted `.fitted` value, as well as standard errors and other metrics for each value.

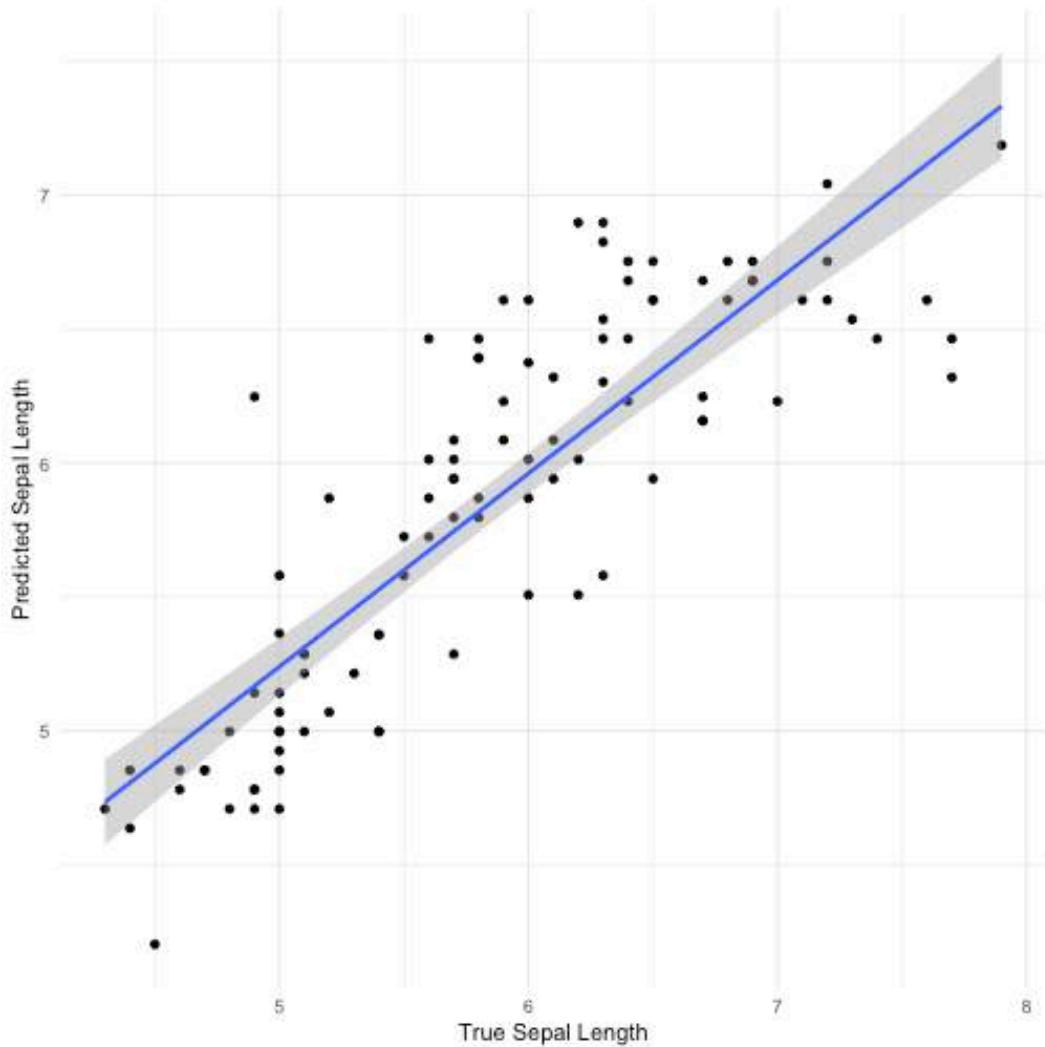
Now we can use the `rmse()` function of the `yardstick` package to compare the truth, which is the `Sepal.Length` variable, to the predicted or estimate variable which in the previous output is called `.fitted`.

```
yardstick::rmse(wf_fitted_values,
  truth = Sepal.Length,
  estimate = .fitted)
# A tibble: 1 × 3
  .metric .estimator .estimate
  <chr>   <chr>        <dbl>
1 rmse    standard     0.447
```

We can see that our RMSE was 0.4472046. This is fairly low, so our model did pretty well.

We can also make a plot to visualize how well we predicted Sepal.Length.

```
wf_fitted_values %>%
  ggplot(aes(x = Sepal.Length, y = .fitted)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs( x = "True Sepal Length", y = "Predicted Sepal Length")
`geom_smooth()` using formula 'y ~ x'
```



We can see that overall our model predicted the sepal length fairly well, as the predicted values are fairly close to the true values. We can also see that the predictions were similar to the truth for the full range of true sepal length values.

Typically we might modify our preprocessing steps or try a different model until we were satisfied with the performance on our training data. Assuming we are satisfied, we could then perform a final assessment of our model using the testing data.

With the `workflows` package, we can use the splitting information for our original data

`split_iris` to fit the final model on the full training set and also on the testing data using the `last_fit()` function of the `tune` package. No preprocessing steps are required.

We can do this by using the `last_fit()` function of the `tune` package.

```
overallfit <- iris_reg_wf %>%
  tune::last_fit(split_iris)
#> #> FALSE Registered S3 method overwritten by 'tune':
#> #> FALSE   method                  from
#> #> FALSE   required_pkgs.model_spec parsnip
#> #> FALSE ! train/test split: preprocessor 1/1, model 1/1 (predictions): prediction\
#> #>   from a rank-defici...
overallfit
#> #> FALSE Warning: This tuning result has notes. Example notes on model fitting inc\
lude:
#> #> FALSE preprocessor 1/1, model 1/1 (predictions): prediction from a rank-deficie\
nt fit may be misleading
#> #> FALSE # Resampling results
#> #> FALSE # Manual resampling
#> #> FALSE # A tibble: 1 × 6
#> #> FALSE   splits         id      .metrics    .notes    .predictions    .w\
orkflow
#> #> FALSE   <list>        <chr>     <list>      <list>      <list>      <l\
ist>
#> #> FALSE 1 <split [100/50]> train/test split <tibble [...] <tibble ... <tibble [50 ... <w\
orkflo
```

We can then use the `collect_metrics()` function of the `tune` package to get the RMSE:

```
collect_metrics(overallfit)
# A tibble: 2 × 4
  .metric .estimator .estimate .config
  <chr>   <chr>        <dbl> <chr>
1 rmse    standard     0.403 Preprocessor1_Model1
2 rsq     standard     0.733 Preprocessor1_Model1
```

We can see that our RMSE is pretty similar for the testing data as well.

Example of Categorical Variable Prediction

Now we are going to show an example of using the `tidymodels` packages to perform prediction of a categorical variable.

Again, we will use the `iris` dataset. However, this time the will predict the identity of the flower species (which is categorical) based on the other variables.

We have already split our data into testing and training sets, so we don't necessarily need to do that again.

However, we can stratify our split by a particular feature of the data using the `strata` argument of the `initial_split()` function.

This is useful to make sure that there is good representation of each species in our testing and training data.

```
set.seed(1234)
initial_split(iris, strata = Species, prop = 2/3)
<Analysis/Assess/Total>
<99/51/150>

training_iris <- training(split_iris)
head(training_iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          5.1         3.5          1.4         0.2     setosa
2          4.9         3.0          1.4         0.2     setosa
3          4.7         3.2          1.3         0.2     setosa
4          4.6         3.1          1.5         0.2     setosa
5          5.0         3.6          1.4         0.2 versicolor
6          5.4         3.9          1.7         0.4 versicolor

count(training_iris, Species)
  Species n
1  setosa 32
2 versicolor 32
3 virginica 36

testing_iris <- testing(split_iris)
head(testing_iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2     setosa
```

```
7      4.6    3.4    1.4    0.3  setosa
11     5.4    3.7    1.5    0.2  setosa
12     4.8    3.4    1.6    0.2  setosa
15     5.8    4.0    1.2    0.2  setosa
16     5.7    4.4    1.5    0.4  setosa
count(testing_iris, Species)
  Species n
1   setosa 18
2 versicolor 18
3 virginica 14
```

Great, indeed we have good representation of all 3 species in both the training and testing sets.

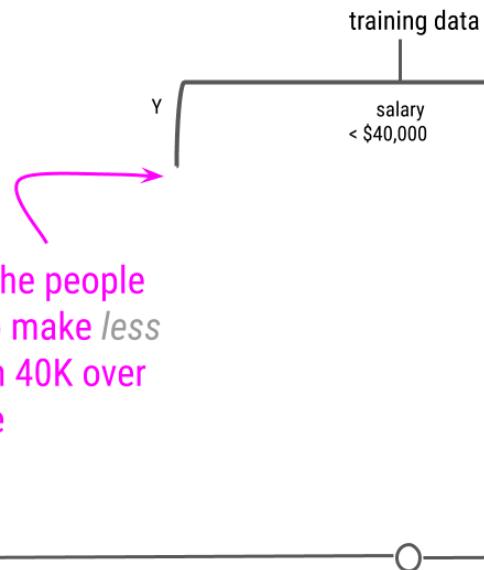
This time we will also show an example of how to perform what is called cross validation. This process allows us to get a better estimate about the performance of our model using just our training data by splitting it into multiple pieces to assess the model fit over and over. This is helpful for making sure that our model will be generalizable, meaning that it will work well with a variety of new datasets. Recall that using an independent validation set is part of what we call **out-of-sample testing** to get a sense of how our model might perform with new datasets. Cross validation helps us to get a sense of this using our training data, so that we can build a better more generalizable model.

By creating subsets of the data, we can test the model performance on each subset which is also a type of out-of-sample testing, as we are not using the entire training dataset, but subsets of the data which may have different properties than that of the full training dataset or each other. For example certain subsets may happen to have unusual values for a particular predictor that are muted by the larger training dataset. With each round of cross validation we perform training and testing on subsets of the training data. This gives us estimates of the out-of-sample performance, where the **out-of-sample error or generalization error** indicates how often predictions are incorrect in the smaller testing subsets of the training data.

Cross validation is also helpful for optimizing what we call hyperparameters.

Hyperparameters are aspects about the model that we need to specify. Often packages will choose a default value, however it is better to use the training data to see what value appears to yield the best model performance.

For example, the different options at each split in a decision tree is called a node. The minimum number of data points for a node to be split further when creating a decision tree model is a hyperparameter.



Recall from our example of a decision tree:

If there were only 3 people who made more than 40,000 and our hyperparameter for the minimum number of data points to continue creating new branches was 6, then this side of the tree would stop here.

We will show how to optimize this using cross validation, this process is also called “tuning”, as we are tuning or adjusting the hyperparameter until we see the best performance with our training data.

The first thing we need to do to perform this process is split our training data into cross validation samples.

Technically creating our testing and training set out of our original training data is sometimes considered a form of **cross validation**, called the holdout method.

The reason we do this is so we can get a better sense of the accuracy of our model using data that we did not train on.

However, we can do a better job of optimizing our model for accuracy if we also perform another type of **cross validation** on just the newly defined training set that we just created.

There are many **cross validation** methods and most can be easily implemented using the `rsample` package. See [here](#) for options.

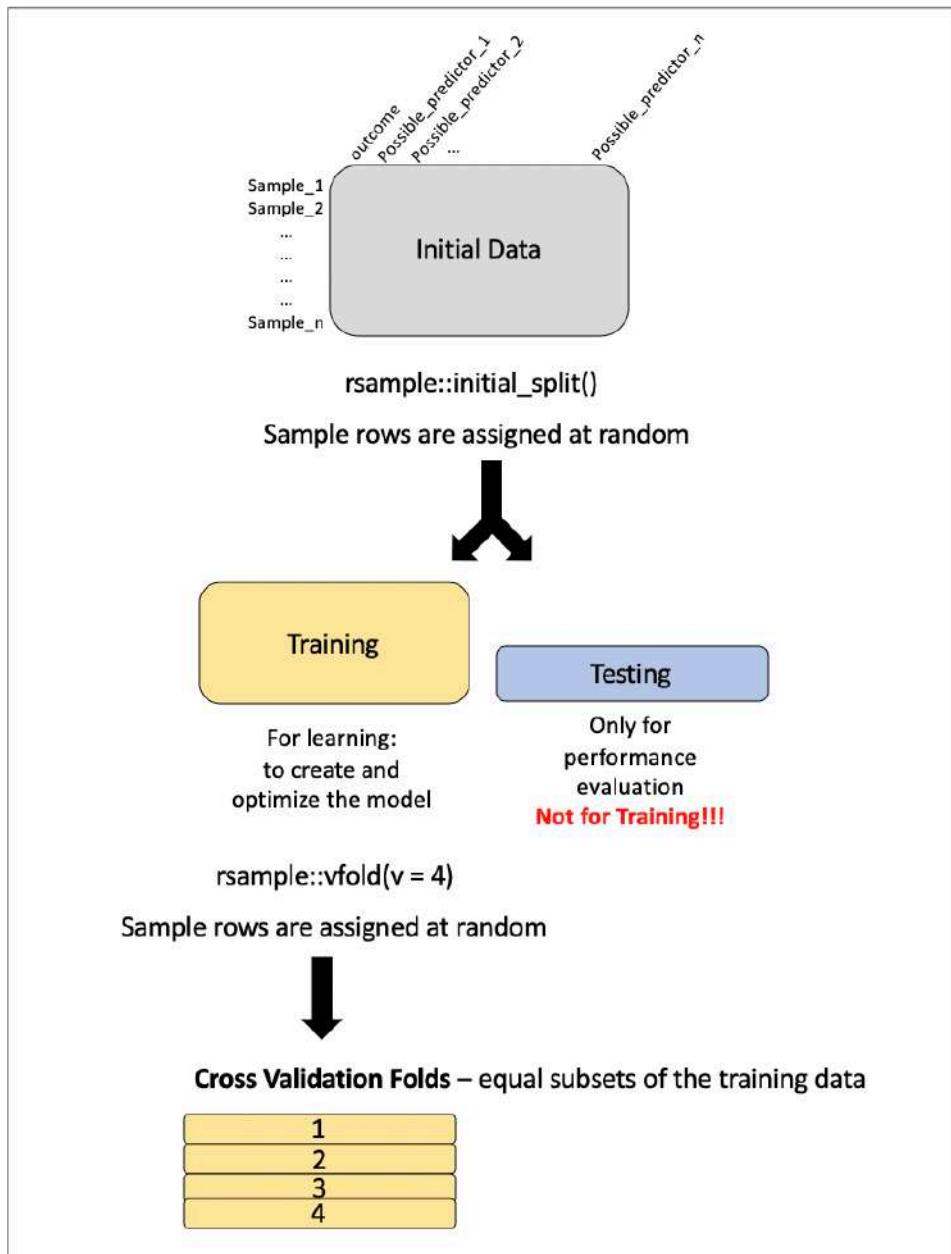
Here, we will use a very popular method called either **v-fold** or **k-fold cross validation**.

This method involves essentially performing the holdout method iteratively with the training data.

First, the training set is divided into v (or often called called k) equally sized smaller pieces. The number of v subsets to use is also a bit arbitrary, although generally speaking using 10 folds is good practice, but this depends on the variability and size of your dataset.

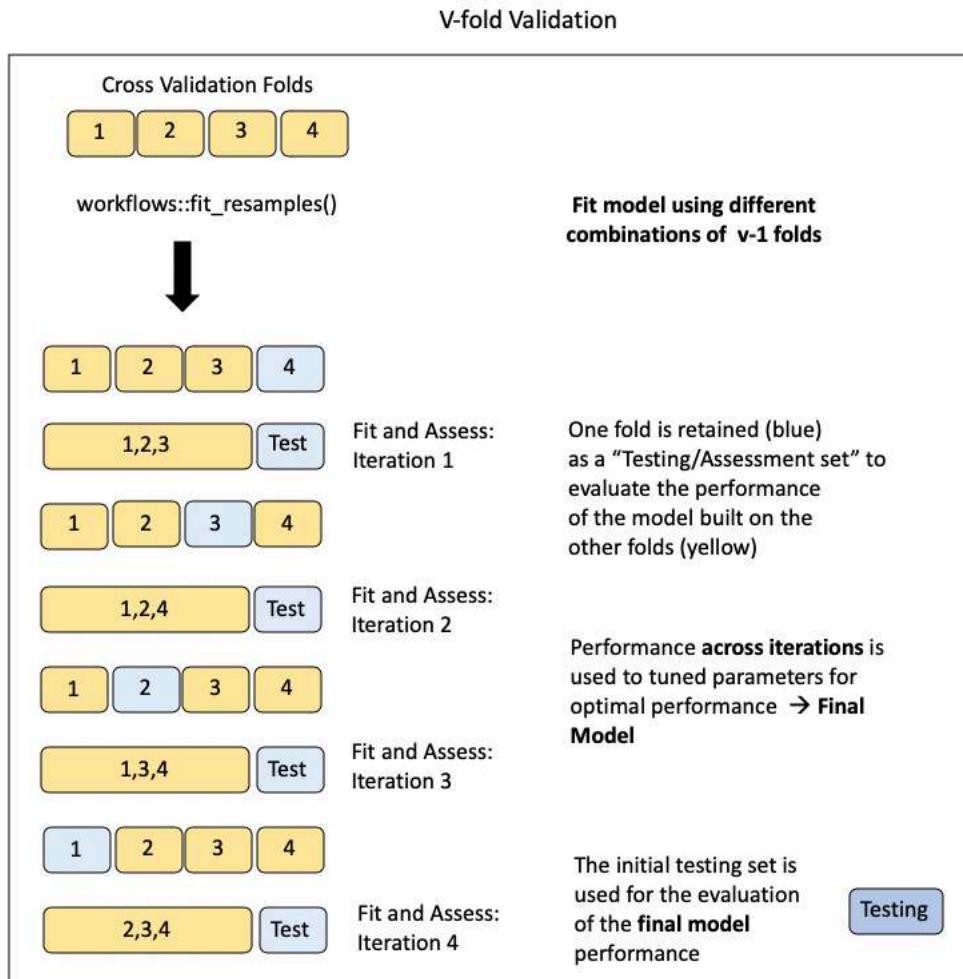
We are going to use 4 folds for the sake of expediency and simplicity.

Splitting the data for vfold validation



The model will be trained on $v-1$ subsets of the data iteratively (removing a different v until all possible $v-1$ sets have been evaluated), while one fold will be saved to act as a test set. This will give us a sense of the out-of-sample (meaning not the entire training sample) performance of the model.

In the case of tuning, multiple values for the hyperparameter are tested to determine what yields the best model performance.



Example of creating cross validation samples with `rsample`

The `vfold_cv()` function of the `rsample` package can be used to parse the training data into folds for v -fold cross validation.

- The `v` argument specifies the number of folds to create.
- The `repeats` argument specifies if any samples should be repeated across folds - default is `FALSE`.
- The `strata` argument specifies a variable to stratify samples across folds - just like in `initial_split()`.

Again, because these are created at random, we need to use the base `set.seed()` function in order to obtain the same results each time.

Remember that only the training data is used to create the cross validation samples.

```
set.seed(1234)
vfold_iris <- rsample::vfold_cv(data = training_iris, v = 4)
vfold_iris
# 4-fold cross-validation
# A tibble: 4 × 2
  splits          id
  <list>         <chr>
1 <split [75/25]> Fold1
2 <split [75/25]> Fold2
3 <split [75/25]> Fold3
4 <split [75/25]> Fold4
pull(vfold_iris, splits)
[[1]]
<Analysis/Assess/Total>
<75/25/100>

[[2]]
<Analysis/Assess/Total>
<75/25/100>

[[3]]
<Analysis/Assess/Total>
<75/25/100>
```

```
[ [4] ]
<Analysis/Assess/Total>
<75/25/100>
```

Now we can see that we have created 4 folds of the data and we can see how many values were set aside for testing (called assessing for cross validation sets) and training (called analysis for cross validation sets) within each fold.

First we will just use cross validation to get a better sense of the **out-of-sample** performance of our model using just the training data. Then we will show how to modify this to perform tuning.

If we want to take a look at the cross validation splits we can do so like this:

```
first_fold <- vfold_iris$splits[[1]]
head(as.data.frame(first_fold, data = "analysis")) # training set of this fold
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
28          5.2       3.5        1.5      0.2   setosa
80          5.7       2.6        3.5      1.0 versicolor
101         6.3       3.3        6.0      2.5  virginica
133         6.4       2.8        5.6      2.2  virginica
144         6.8       3.2        5.9      2.3  virginica
132         7.9       3.8        6.4      2.0  virginica
head(as.data.frame(first_fold, data = "assessment")) # test set of this fold
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
111         6.5       3.2        5.1      2.0  virginica
137         6.3       3.4        5.6      2.4  virginica
62          5.9       3.0        4.2      1.5 versicolor
143         5.8       2.7        5.1      1.9  virginica
47          5.1       3.8        1.6      0.2   setosa
84          6.0       2.7        5.1      1.6 versicolor
```

Example of creating another recipe, model and workflow

We also need to create a new recipe with different variables assigned to different roles. This time we want to use `Species` as the outcome. We can use the `.` notation to indicate that we want to use the rest of the variables as predictors. Thus we will create a new `cat_recipe` where we are using a `categorical` variable as the outcome.

```
cat_recipe <- training_iris %>%
  recipe(Species ~ .)
```

This time we will also not have any preprocessing steps for simplicity sake, thus our recipe is actually already finished.

Now our next step is to specify our model. Again the modeling options for parsnip are [here](#). We will be using a Classification And Regression Tree (CART), which we discussed previously. This method can be used for either classification or regression (categorical or continuous outcome variables). Thus it is important that we set the mode for classification. We will use the `rpart` package as our engine. To tune using this model we would need to specify it here as well. We will show that in just a bit.

```
library(rpart)

Attaching package: 'rpart'
The following object is masked from 'package:dials':
  prune

prune
cat_model <- parsnip::decision_tree() %>%
  parsnip::set_mode("classification") %>%
  parsnip::set_engine("rpart")

cat_model
Decision Tree Model Specification (classification)

Computational engine: rpart
```

Great! Now we will make a workflow for this.

```
iris_cat_wf <- workflows::workflow() %>%
  workflows::add_recipe(cat_recipe) %>%
  workflows::add_model(cat_model)

iris_cat_wf
= Workflow =====\

Preprocessor: Recipe
Model: decision_tree()

— Preprocessor =====\
```

```
0 Recipe Steps
```

```
— Model ————— \
```

```
Decision Tree Model Specification (classification)
```

```
Computational engine: rpart
```

So our next step is to fit and tune the model with our training data cross validation subsets.

Example of assessing model performance with cross validation using `tune`

First we will demonstrate how we could fit the model using our entire training dataset like we did previously and use yardstick to check the accuracy this time instead of RMSE.

```
iris_cat_wf_fit <- parsnip::fit(iris_cat_wf, data = training_iris)
iris_cat_wf_fit
== Workflow [trained] == \
```

```
Preprocessor: Recipe
Model: decision_tree()
```

```
— Preprocessor ————— \
```

```
0 Recipe Steps
```

```
— Model ————— \
```

```
n= 100
```



```
node), split, n, loss, yval, (yprob)
  * denotes terminal node
```



```
1) root 100 64 virginica (0.32000000 0.32000000 0.36000000)
  2) Petal.Length< 2.6 32  0 setosa (1.00000000 0.00000000 0.00000000) *
  3) Petal.Length>=2.6 68  32 virginica (0.00000000 0.47058824 0.52941176)
     6) Petal.Length< 4.85 33  2 versicolor (0.00000000 0.93939394 0.06060606) *
```

```
7) Petal.Length>=4.85 35 1 virginica (0.00000000 0.02857143 0.97142857) *  
  
wf_fit_cat <- iris_cat_wf_fit %>%  
  pull_workflow_fit()  
Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.  
Please use `extract_fit_parsnip()` instead.
```

The output is a bit different for categorical variables. We can also see variable importance from the model fit, which shows which variables were most important for classifying the data values. This lists a score for each variable which shows the decrease in error when splitting by this variable relative to others.

```
wf_fit_cat$fit$variable.importance  
Petal.Length  Petal.Width Sepal.Length  Sepal.Width  
 60.85957      55.73558      42.94372      19.08610
```

We can see that `Petal.Width` was the most important for predicting `Species`.

Recall that since we are using a categorical outcome variable, we want to use accuracy to assess model performance. Thus, we can use the `accuracy()` function of the `yardstick` package instead of the `rmse()` function to assess the model. We first need to get the predicted values using the `predict()` function, as these are not in the fit output.

```
pred_species<-predict(iris_cat_wf_fit, new_data = training_iris)  
  
yardstick::accuracy(training_iris,  
  truth = Species, estimate = pred_species$.pred_class)  
# A tibble: 1 × 3  
  .metric   .estimator .estimate  
  <chr>     <chr>        <dbl>  
1 accuracy  multiclass    0.97
```

It looks like 97% of the time our model correctly predicted the right species.

We can also see which species were correctly predicted using `count` function.

```
count(training_iris, Species)
#> #>   Species n
#> #>   1 setosa 32
#> #>   2 versicolor 32
#> #>   3 virginica 36
count(pred_species, .pred_class)
#> #> # A tibble: 3 × 2
#> #>   .pred_class     n
#> #>   <fct>       <int>
#> #>   1 setosa        32
#> #>   2 versicolor    33
#> #>   3 virginica    35
```

We can see that one extra versicolor iris was predicted, and one fewer virginica iris.

To see exactly which rows resulted in incorrect predictions, we can bind the predicted species to the training data like so. This can be helpful to see if there is something particular about the incorrectly predicted values that might explain why they are incorrectly predicted.

```
predicted_and_truth <- bind_cols(training_iris,
                                   predicted_species = pull(pred_species, .pred_class))

head(predicted_and_truth)
#> #> #> Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
#> #> #> 1         5.2       3.5       1.5       0.2   setosa
#> #> #> 2         5.7       2.6       3.5      1.0 versicolor
#> #> #> 3         6.3       3.3       6.0      2.5  virginica
#> #> #> 4         6.5       3.2       5.1      2.0  virginica
#> #> #> 5         6.3       3.4       5.6      2.4  virginica
#> #> #> 6         6.4       2.8       5.6      2.2  virginica
#> #> #> predicted_species
#> #> #>   1       setosa
#> #> #>   2 versicolor
#> #> #>   3  virginica
#> #> #>   4  virginica
#> #> #>   5  virginica
#> #> #>   6  virginica
```

However, to fit the model to our cross validation folds we can use the `fit_resamples()`

function of the `tune` package, by specifying our `workflow` object and the cross validation fold object we just created. See [here](#) for more information.

```
library(tune)
set.seed(122)
resample_fit <- tune::fit_resamples(iris_cat_wf, vfold_iris)
```

We can now take a look at various performance metrics based on the fit of our cross validation “resamples”.

To do this we will use the `collect_metrics` function of the `tune` package. This will show us the mean of the accuracy estimate of the different cross validation folds.

```
resample_fit
# Resampling results
# 4-fold cross-validation
# A tibble: 4 × 4
  splits          id    .metrics      .notes
  <list>         <chr> <list>        <list>
1 <split [75/25]> Fold1 <tibble [2 × 4]> <tibble [0 × 1]>
2 <split [75/25]> Fold2 <tibble [2 × 4]> <tibble [0 × 1]>
3 <split [75/25]> Fold3 <tibble [2 × 4]> <tibble [0 × 1]>
4 <split [75/25]> Fold4 <tibble [2 × 4]> <tibble [0 × 1]>
collect_metrics(resample_fit)
# A tibble: 2 × 6
  .metric .estimator   mean     n std_err .config
  <chr>   <chr>     <dbl> <int>   <dbl> <chr>
1 accuracy multiclass 0.95     4  0.0191 Preprocessor1_Model1
2 roc_auc   hand_till  0.964    4  0.0137 Preprocessor1_Model1
```

The accuracy appears to be 94 percent. Often the performance will be reduced using cross validation.

Example of tuning

Nice, let’s see how this changes when we now tune a hyperparameter. We want to tune the `min_n` argument to tune for the minimum number of data points for each node. The arguments may vary for the engine that you are using. We need to specify this when we fit the model using the `tune()` function like so:

```
set.seed(122)
library(tune)
cat_model_tune <- parsnip::decision_tree(min_n = tune()) %>%
  parsnip::set_mode("classification") %>%
  parsnip::set_engine("rpart")
cat_model_tune
Decision Tree Model Specification (classification)

Main Arguments:
  min_n = tune()

Computational engine: rpart
```

Now we can create a new workflow using the categorical recipe and the tuning model:

```
iris_cat_wf_tune <- workflows::workflow() %>%
  workflows::add_recipe(cat_recipe) %>%
  workflows::add_model(cat_model_tune)
```

We can use the `tune_grid()` function of the `tune()` package to use the workflow and fit the `vfold_iris` cross validation samples of our training data to test out a number of different values for the `min_n` argument for our model. The `grid()` argument specifies how many different values to try out.

```
reasmples_fit <- tune::tune_grid(iris_cat_wf_tune, resamples = vfold_iris, grid = 4)
```

Again we can use the `collect_metrics()` function to get the accuracy. Or, we can use the `show_best()` function of the `tune` package to see the `min_n` values for the top performing models (those with the highest accuracy).

```
tune::collect_metrics(resample_fit)
# A tibble: 2 × 6
  .metric  .estimator  mean    n std_err .config
  <chr>    <chr>      <dbl> <int>   <dbl> <chr>
1 accuracy multiclass 0.95     4  0.0191 Preprocessor1_Model1
2 roc_auc  hand_till   0.964    4  0.0137 Preprocessor1_Model1
tune::show_best(resample_fit, metric = "accuracy")
# A tibble: 1 × 6
  .metric  .estimator  mean    n std_err .config
  <chr>    <chr>      <dbl> <int>   <dbl> <chr>
1 accuracy multiclass  0.95     4  0.0191 Preprocessor1_Model1
```

Case Studies

Now we will demonstrate a more involved example with a case study.

Case Study #1: Predicting Annual Air Pollution

A variety of different sources contribute different types of pollutants to what we call air pollution.

- 1) **Gaseous** - Carbon Monoxide (CO), Ozone (O₃), Nitrogen Oxides(NO, NO₂), Sulfur Dioxide (SO₂)
- 2) **Particulate** - small liquids and solids suspended in the air (includes lead- can include certain types of dust)
- 3) **Dust** - small solids (larger than particulates) that can be suspended in the air for some time but eventually settle
- 4) **Biological** - pollen, bacteria, viruses, mold spores

Air pollution particulates are generally described by their **size**.

There are 3 major categories:

- 1) **Large Coarse Particulate Matter** - has diameter of >10 micrometers (10 μm)
 - 2) **Coarse Particulate Matter** (called PM_{10-2.5}) - has diameter of between 2.5 μm and 10 μm
 - 3) **Fine Particulate Matter** (called PM_{2.5}) - has diameter of < 2.5 μm
- PM₁₀ includes any particulate matter <10 μm (both coarse and fine particulate matter)

Of these different sizes, fine particulate matter air pollution is the most associated with health risk.

In this case study, we will use fine particulate matter air pollution monitor data from [gravimetric monitors](#)] operated by the US [Environmental Protection Agency \(EPA\)](#).

Roughly 90% of these monitors are located within cities. Hence, there is limited data about air pollution levels of more rural areas.

To get a better sense of the pollution exposures for the individuals living in these areas, methods like machine learning can be useful to estimate air pollution levels in **areas with little to no monitoring**.

We will use data like population density, road density, among other features, to build a model to predict the known monitored fine particulate air pollution levels. Such a model can then be used to estimate levels of pollution in places with poor monitoring.

The Data

There are 48 predictors with values for 876 monitors (observations).

The data comes from the US [Environmental Protection Agency \(EPA\)](#), the [National Aeronautics and Space Administration \(NASA\)](#), the US [Census](#), and the [National Center for Health Statistics \(NCHS\)](#).

See an [online version of the book](#) to see a table about the set of features.

Many of these features have to do with the circular area around the monitor called the “buffer”, but you don’t need to worry much about this to understand this as an example.

Data Import

We have one CSV file that contains both our single **outcome variable** and all of our **features** (or predictor variables).

Next, we import our data into R now so that we can explore the data further. We will call our data object `pm` for particulate matter. We import the data using the `read_csv()` function from the `readr` package.

```
library(here)
pm <- readr::read_csv(here("data", "tidy_data", "pm25_data.csv"))
Rows: 876 Columns: 50
— Column specification ——————\

Delimiter: ","
chr (3): state, county, city
dbl (47): id, value, fips, lat, lon, CMAQ, zcta, zcta_area, zcta_pop, imp_a5...

```

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

Data Exploration and Wrangling

First, let's just get a general sense of our data.

```
library(dplyr)
pm %>%
  glimpse()
Rows: 876
Columns: 50
$ id                               <dbl> 1003.001, 1027.000, 1033.100, 1049.100, 10...
$ value                            <dbl> 9.597647, 10.800000, 11.212174, 11.659091, ...
$ fips                             <dbl> 1003, 1027, 1033, 1049, 1055, 1069, 1073, ...
$ lat                              <dbl> 30.49800, 33.28126, 34.75878, 34.28763, 33...
$ lon                              <dbl> -87.88141, -85.80218, -87.65056, -85.96830...
$ state                            <chr> "Alabama", "Alabama", "Alabama", "Alabama"...
$ county                           <chr> "Baldwin", "Clay", "Colbert", "DeKalb", "E...
$ city                             <chr> "Fairhope", "Ashland", "Muscle Shoals", "C...
$ CMAQ                             <dbl> 8.098836, 9.766208, 9.402679, 8.534772, 9...
$ zcta                            <dbl> 36532, 36251, 35660, 35962, 35901, 36303, ...
$ zcta_area                         <dbl> 190980522, 374132430, 16716984, 203836235, ...
$ zcta_pop                          <dbl> 27829, 5103, 9042, 8300, 20045, 30217, 901...
$ imp_a500                          <dbl> 0.01730104, 1.96972318, 19.17301038, 5.782...
$ imp_a1000                         <dbl> 1.4096021, 0.8531574, 11.1448962, 3.867647...
$ imp_a5000                         <dbl> 3.3360118, 0.9851479, 15.1786154, 1.231141...
$ imp_a10000                        <dbl> 1.9879187, 0.5208189, 9.7253870, 1.0316469...
```

```
$ imp_a15000 <dbl> 1.4386207, 0.3359198, 5.2472094, 0.9730444...
$ county_area <dbl> 4117521611, 1564252280, 1534877333, 201266...
$ county_pop <dbl> 182265, 13932, 54428, 71109, 104430, 10154...
$ log_dist_to_prisec <dbl> 4.648181, 7.219907, 5.760131, 3.721489, 5...
$ log_pri_length_5000 <dbl> 8.517193, 8.517193, 8.517193, 8.517193, 9...
$ log_pri_length_10000 <dbl> 9.210340, 9.210340, 9.274303, 10.409411, 1...
$ log_pri_length_15000 <dbl> 9.630228, 9.615805, 9.658899, 11.173626, 1...
$ log_pri_length_25000 <dbl> 11.32735, 10.12663, 10.15769, 11.90959, 12...
$ log_prisec_length_500 <dbl> 7.295356, 6.214608, 8.611945, 7.310155, 8...
$ log_prisec_length_1000 <dbl> 8.195119, 7.600902, 9.735569, 8.585843, 9...
$ log_prisec_length_5000 <dbl> 10.815042, 10.170878, 11.770407, 10.214200...
$ log_prisec_length_10000 <dbl> 11.88680, 11.40554, 12.84066, 11.50894, 12...
$ log_prisec_length_15000 <dbl> 12.205723, 12.042963, 13.282656, 12.353663...
$ log_prisec_length_25000 <dbl> 13.41395, 12.79980, 13.79973, 13.55979, 13...
$ log_nei_2008_pm25_sum_10000 <dbl> 0.318035438, 3.218632928, 6.573127301, 0.0...
$ log_nei_2008_pm25_sum_15000 <dbl> 1.967358961, 3.218632928, 6.581917457, 3.2...
$ log_nei_2008_pm25_sum_25000 <dbl> 5.067308, 3.218633, 6.875900, 4.887665, 4...
$ log_nei_2008_pm10_sum_10000 <dbl> 1.35588511, 3.31111648, 6.69187313, 0.0000...
$ log_nei_2008_pm10_sum_15000 <dbl> 2.26783411, 3.31111648, 6.70127741, 3.3500...
$ log_nei_2008_pm10_sum_25000 <dbl> 5.628728, 3.311116, 7.148858, 5.171920, 4...
$ popdens_county <dbl> 44.265706, 8.906492, 35.460814, 35.330814...
$ popdens_zcta <dbl> 145.716431, 13.639555, 540.887040, 40.7189...
$ nohs <dbl> 3.3, 11.6, 7.3, 14.3, 4.3, 5.8, 7.1, 2.7, ...
$ somehs <dbl> 4.9, 19.1, 15.8, 16.7, 13.3, 11.6, 17.1, 6...
$ hs <dbl> 25.1, 33.9, 30.6, 35.0, 27.8, 29.8, 37.2, ...
$ somecollege <dbl> 19.7, 18.8, 20.9, 14.9, 29.2, 21.4, 23.5, ...
$ associate <dbl> 8.2, 8.0, 7.6, 5.5, 10.1, 7.9, 7.3, 8.0, 4...
$ bachelor <dbl> 25.3, 5.5, 12.7, 7.9, 10.0, 13.7, 5.9, 17...
$ grad <dbl> 13.5, 3.1, 5.1, 5.8, 5.4, 9.8, 2.0, 8.7, 2...
$ pov <dbl> 6.1, 19.5, 19.0, 13.8, 8.8, 15.6, 25.5, 7...
$ hs_orless <dbl> 33.3, 64.6, 53.7, 66.0, 45.4, 47.2, 61.4, ...
$ urc2013 <dbl> 4, 6, 4, 6, 4, 4, 1, 1, 1, 1, 1, 1, 1, 2, ...
$ urc2006 <dbl> 5, 6, 4, 5, 4, 4, 1, 1, 1, 1, 1, 1, 1, 2, ...
$ aod <dbl> 37.36364, 34.81818, 36.00000, 33.08333, 43...
```

We can see that there are 876 monitors (rows) and that we have 50 total variables (columns) - one of which is the outcome variable. In this case, the outcome variable is called `value`.

Notice that some of the variables that we would think of as factors (or categorical data)

are currently of class character as indicated by the `<chr>` just to the right of the column names/variable names in the `glimpse()` output. This means that the variable values are character strings, such as words or phrases.

The other variables are of class `<dbl>`, which stands for double precision which indicates that they are numeric and that they have decimal values. In contrast, one could have integer values which would not allow for decimal numbers. Here is a [link](#) for more information on double precision numeric values.

Another common data class is factor which is abbreviated like this: `<fct>`. A factor is something that has unique levels but there is no appreciable order to the levels. For example we can have a numeric value that is just an id that we want to be interpreted as just a unique level and not as the number that it would typically indicate. This would be useful for several of our variables:

1. the monitor ID (`id`)
2. the Federal Information Processing Standard number for the county where the monitor was located (`fips`)
3. the zip code tabulation area (`zcta`)

None of the values actually have any real numeric meaning, so we want to make sure that R does not interpret them as if they do.

So let's convert these variables into factors. We can do this using the `across()` function of the `dplyr` package and the `as.factor()` base function. The `across()` function has two main arguments: (i) the columns you want to operate on and (ii) the function or list of functions to apply to each column.

```
library(magrittr)

Attaching package: 'magrittr'
The following object is masked from 'package:rlang':
  set_names

The following object is masked from 'package:purrr':
  set_names

The following object is masked from 'package:tidyverse':
  extract
```

```
pm <- pm %>%
  mutate(across(c(id, fips, zcta), as.factor))

glimpse(pm)
Rows: 876
Columns: 50
$ id                      <fct> 1003.001, 1027.0001, 1033.1002, 1049.1003, ...
$ value                   <dbl> 9.597647, 10.800000, 11.212174, 11.659091, ...
$ fips                     <fct> 1003, 1027, 1033, 1049, 1055, 1069, 1073, ...
$ lat                      <dbl> 30.49800, 33.28126, 34.75878, 34.28763, 33...
$ lon                      <dbl> -87.88141, -85.80218, -87.65056, -85.96830...
$ state                    <chr> "Alabama", "Alabama", "Alabama", "Alabama"...
$ county                   <chr> "Baldwin", "Clay", "Colbert", "DeKalb", "E...
$ city                      <chr> "Fairhope", "Ashland", "Muscle Shoals", "C...
$ CMAQ                     <dbl> 8.098836, 9.766208, 9.402679, 8.534772, 9...
$ zcta                     <fct> 36532, 36251, 35660, 35962, 35901, 36303, ...
$ zcta_area                 <dbl> 190980522, 374132430, 16716984, 203836235, ...
$ zcta_pop                  <dbl> 27829, 5103, 9042, 8300, 20045, 30217, 901...
$ imp_a500                  <dbl> 0.01730104, 1.96972318, 19.17301038, 5.782...
$ imp_a1000                 <dbl> 1.4096021, 0.8531574, 11.1448962, 3.867647...
$ imp_a5000                 <dbl> 3.3360118, 0.9851479, 15.1786154, 1.231141...
$ imp_a10000                <dbl> 1.9879187, 0.5208189, 9.7253870, 1.0316469...
$ imp_a15000                <dbl> 1.4386207, 0.3359198, 5.2472094, 0.9730444...
$ county_area                <dbl> 4117521611, 1564252280, 1534877333, 201266...
$ county_pop                 <dbl> 182265, 13932, 54428, 71109, 104430, 10154...
$ log_dist_to_prisec        <dbl> 4.648181, 7.219907, 5.760131, 3.721489, 5...
$ log_pri_length_5000       <dbl> 8.517193, 8.517193, 8.517193, 8.517193, 9...
$ log_pri_length_10000      <dbl> 9.210340, 9.210340, 9.274303, 10.409411, 1...
$ log_pri_length_15000      <dbl> 9.630228, 9.615805, 9.658899, 11.173626, 1...
$ log_pri_length_25000      <dbl> 11.32735, 10.12663, 10.15769, 11.90959, 12...
$ log_prisec_length_500     <dbl> 7.295356, 6.214608, 8.611945, 7.310155, 8...
$ log_prisec_length_1000    <dbl> 8.195119, 7.600902, 9.735569, 8.585843, 9...
$ log_prisec_length_5000    <dbl> 10.815042, 10.170878, 11.770407, 10.214200...
$ log_prisec_length_10000   <dbl> 11.88680, 11.40554, 12.84066, 11.50894, 12...
$ log_prisec_length_15000   <dbl> 12.205723, 12.042963, 13.282656, 12.353663...
$ log_prisec_length_25000   <dbl> 13.41395, 12.79980, 13.79973, 13.55979, 13...
$ log_nei_2008_pm25_sum_10000 <dbl> 0.318035438, 3.218632928, 6.573127301, 0.0...
$ log_nei_2008_pm25_sum_15000 <dbl> 1.967358961, 3.218632928, 6.581917457, 3.2...
$ log_nei_2008_pm25_sum_25000 <dbl> 5.067308, 3.218633, 6.875900, 4.887665, 4...
```

```
$ log_nei_2008_pm10_sum_10000 <dbl> 1.35588511, 3.31111648, 6.69187313, 0.0000...
$ log_nei_2008_pm10_sum_15000 <dbl> 2.26783411, 3.31111648, 6.70127741, 3.3500...
$ log_nei_2008_pm10_sum_25000 <dbl> 5.628728, 3.311116, 7.148858, 5.171920, 4...
$ popdens_county <dbl> 44.265706, 8.906492, 35.460814, 35.330814, ...
$ popdens_zcta <dbl> 145.716431, 13.639555, 540.887040, 40.7189...
$ nohs <dbl> 3.3, 11.6, 7.3, 14.3, 4.3, 5.8, 7.1, 2.7, ...
$ somehs <dbl> 4.9, 19.1, 15.8, 16.7, 13.3, 11.6, 17.1, 6...
$ hs <dbl> 25.1, 33.9, 30.6, 35.0, 27.8, 29.8, 37.2, ...
$ somecollege <dbl> 19.7, 18.8, 20.9, 14.9, 29.2, 21.4, 23.5, ...
$ associate <dbl> 8.2, 8.0, 7.6, 5.5, 10.1, 7.9, 7.3, 8.0, 4...
$ bachelor <dbl> 25.3, 5.5, 12.7, 7.9, 10.0, 13.7, 5.9, 17...
$ grad <dbl> 13.5, 3.1, 5.1, 5.8, 5.4, 9.8, 2.0, 8.7, 2...
$ pov <dbl> 6.1, 19.5, 19.0, 13.8, 8.8, 15.6, 25.5, 7...
$ hs_orless <dbl> 33.3, 64.6, 53.7, 66.0, 45.4, 47.2, 61.4, ...
$ urc2013 <dbl> 4, 6, 4, 6, 4, 4, 1, 1, 1, 1, 1, 1, 1, 2, ...
$ urc2006 <dbl> 5, 6, 4, 5, 4, 4, 1, 1, 1, 1, 1, 1, 1, 2, ...
$ aod <dbl> 37.36364, 34.81818, 36.00000, 33.08333, 43...
```

Great! Now we can see that these variables are now factors as indicated by `<fct>` after the variable name.

The `skim()` function of the `skimr` package is also really helpful for getting a general sense of your data. By design, it provides summary statistics about variables in the dataset.

```
library(skimr)
skim(pm)
```

<hr/> — Data Summary —————												
	Values											
Name	pm											
Number of rows	876											
Number of columns	50											
Column type frequency:												
character	3											
factor	3											
numeric	44											
Group variables												
None												
— Variable type: character —————												
skim_variable	n_missing	complete_rate	min	max	empty	n_unique whitespace						
1 state	0	1	4	20	0	49 0						
2 county	0	1	3	20	0	471 0						
3 city	0	1	4	48	0	607 0						
— Variable type: factor —————												
skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts							
1 id	0	1 FALSE		876	100: 1, 102: 1, 103: 1, 104: 1							
2 fips	0	1 FALSE		569	170: 12, 603: 10, 261: 9, 107: 8							
3 zcta	0	1 FALSE		842	475: 3, 110: 2, 160: 2, 290: 2							
— Variable type: numeric —————												
skim_variable	n_missing	complete_rate	mean	sd	p0	p25						
1 value	0	1	10.8	2.58e+0	3.02	9.27						
2 lat	0	1	38.5	4.62e+0	25.5	35.0						
3 lon	0	1	-91.7	1.50e+1	-124.	-99.2						
4 CMAQ	0	1	8.41	2.97e+0	1.63	6.53						
5 zcta_area	0	1	183173482.	5.43e+8	15459	14204602.						
6 zcta_pop	0	1	24228.	1.78e+4	0	9797						
7 imp_a500	0	1	24.7	1.93e+1	0	3.70						
8 imp_a1000	0	1	24.3	1.80e+1	0	5.32						
9 imp_a5000	0	1	19.9	1.47e+1	0.0534	6.79						
10 imp_a10000	0	1	15.8	1.38e+1	0.0942	4.54						
11 imp_a15000	0	1	13.4	1.31e+1	0.108	3.24						
12 county_area	0	1	3768701992.	6.21e+9	33703512	1116536298.						
13 county_pop	0	1	687298.	1.29e+6	783	100948						
14 log_dist_to_prisec	0	1	6.19	1.41e+0	-1.46	5.43						
15 log_pri_length_5000	0	1	9.82	1.08e+0	8.52	8.52						
16 log_pri_length_10000	0	1	10.9	1.13e+0	9.21	9.80						
17 log_pri_length_15000	0	1	11.5	1.15e+0	9.62	10.9						
18 log_pri_length_25000	0	1	12.2	1.10e+0	10.1	11.7						
19 log_prisec_length_500	0	1	6.99	9.53e-1	6.21	6.21						
20 log_prisec_length_1000	0	1	8.56	7.90e-1	7.60	7.60						
21 log_prisec_length_5000	0	1	11.3	7.81e-1	8.52	10.9						
22 log_prisec_length_10000	0	1	12.4	7.33e-1	9.21	12.0						
23 log_prisec_length_15000	0	1	13.0	7.23e-1	9.62	12.6						
24 log_prisec_length_25000	0	1	13.8	6.99e-1	10.1	13.4						
25 log_nei_2008_pm25_sum_10000	0	1	3.97	2.35e+0	0	2.15						
26 log_nei_2008_pm25_sum_15000	0	1	4.72	2.25e+0	0	3.47						
27 log_nei_2008_pm25_sum_25000	0	1	5.67	2.11e+0	0	4.66						
28 log_nei_2008_pm10_sum_10000	0	1	4.35	2.32e+0	0	2.69						
29 log_nei_2008_pm10_sum_15000	0	1	5.10	2.18e+0	0	3.87						
30 log_nei_2008_pm10_sum_25000	0	1	6.07	2.01e+0	0	5.10						
31 popdens_county	0	1	552.	1.71e+3	0.263	40.8						
32 popdens_zcta	0	1	1280.	2.76e+3	0	101.						
33 nohs	0	1	6.99	7.21e+0	0	2.7						
34 somehs	0	1	10.2	6.20e+0	0	5.9						
35 hs	0	1	30.3	1.14e+1	0	23.8						
36 somecollege	0	1	21.6	8.60e+0	0	17.5						
37 associate	0	1	7.13	4.01e+0	0	4.9						
38 bachelor	0	1	14.9	9.71e+0	0	8.8						
39 grad	0	1	8.91	8.65e+0	0	3.9						
40 pov	0	1	15.0	1.13e+1	0	6.5						
41 hs_orless	0	1	47.5	1.68e+1	0	37.9						
42 urc2013	0	1	2.92	1.52e+0	1	2						
43 urc2006	0	1	2.97	1.52e+0	1	2						

	p50	p75	p100	hist
1	11.2	12.4	2.32e 1	
2	39.3	41.7	4.84e 1	
3	-87.5	-80.7	-6.80e 1	
4	8.62	10.2	2.31e 1	
5	37653560.	160041508.	8.16e 9	
6	22014	35005.	9.54e 4	
7	25.1	40.2	6.96e 1	
8	24.5	38.6	6.75e 1	
9	19.1	30.1	7.46e 1	
10	12.4	24.2	7.21e 1	
11	9.67	20.6	7.11e 1	
12	1690826566.	2878192209	5.19e10	
13	280730.	743159	9.82e 6	
14	6.36	7.15	1.05e 1	
15	10.1	10.7	1.20e 1	
16	11.2	11.8	1.30e 1	
17	11.7	12.4	1.36e 1	
18	12.5	13.1	1.44e 1	
19	6.21	7.82	9.40e 0	
20	8.66	9.20	1.05e 1	
21	11.4	11.8	1.28e 1	
22	12.5	12.9	1.38e 1	
23	13.1	13.6	1.44e 1	
24	13.9	14.4	1.52e 1	
25	4.29	5.69	9.12e 0	
26	5.80	6.35	9.42e 0	
27	5.91	7.28	9.65e 0	
28	4.62	6.07	9.34e 0	
29	5.39	6.72	9.71e 0	
30	6.37	7.52	9.88e 0	
31	157.	511.	2.68e 4	
32	610.	1383.	3.04e 4	
33	5.1	8.8	1.00e 2	
34	9.4	13.9	7.22e 1	
35	30.8	36.1	1.00e 2	
36	21.3	24.7	1.00e 2	
37	7.1	8.8	7.14e 1	
38	13.0	19.2	1.00e 2	
39	6.7	11	1.00e 2	
40	12.1	21.2	6.59e 1	
41	48.7	59.1	1.00e 2	
42	3	4	6.00e 0	
43	3	4	6.00e 0	
44	40.2	49.7	1.43e 2	

skim output

Notice how there is a column called `n_missing` about the number of values that are missing.

This is also indicated by the `complete_rate` variable (or missing/number of observations).

In our dataset, it looks like our data do not contain any missing data.

Also notice how the function provides separate tables of summary statistics for each data type: character, factor and numeric.

Next, the `n_unique` column shows us the number of unique values for each of our columns. We can see that there are 49 states represented in the data.

We can see that for many variables there are many low values as the distribution shows two peaks, one near zero and another with a higher value.

This is true for the `imp` variables (measures of development), the `nei` variables (measures of emission sources) and the road density variables.

We can also see that the range of some of the variables is very large, in particular the area and population related variables.

Let's take a look to see which states are included using the `distinct()` function of the `dplyr` package:

```
pm %>%
  distinct(state)
# A tibble: 49 × 1
  state
  <chr>
 1 Alabama
 2 Arizona
 3 Arkansas
 4 California
 5 Colorado
 6 Connecticut
 7 Delaware
 8 District Of Columbia
 9 Florida
10 Georgia
# ... with 39 more rows
```

Evaluate Correlation

In prediction analyses, it is also useful to evaluate if any of the variables are correlated. Why should we care about this?

If we are using a linear regression to model our data then we might run into a problem called multicollinearity which can lead us to misinterpret what is really predictive of our outcome variable. This phenomenon occurs when the predictor variables actually predict one another.

Another reason we should look out for correlation is that we don't want to include redundant variables. This can add unnecessary noise to our algorithm causing a reduction in prediction

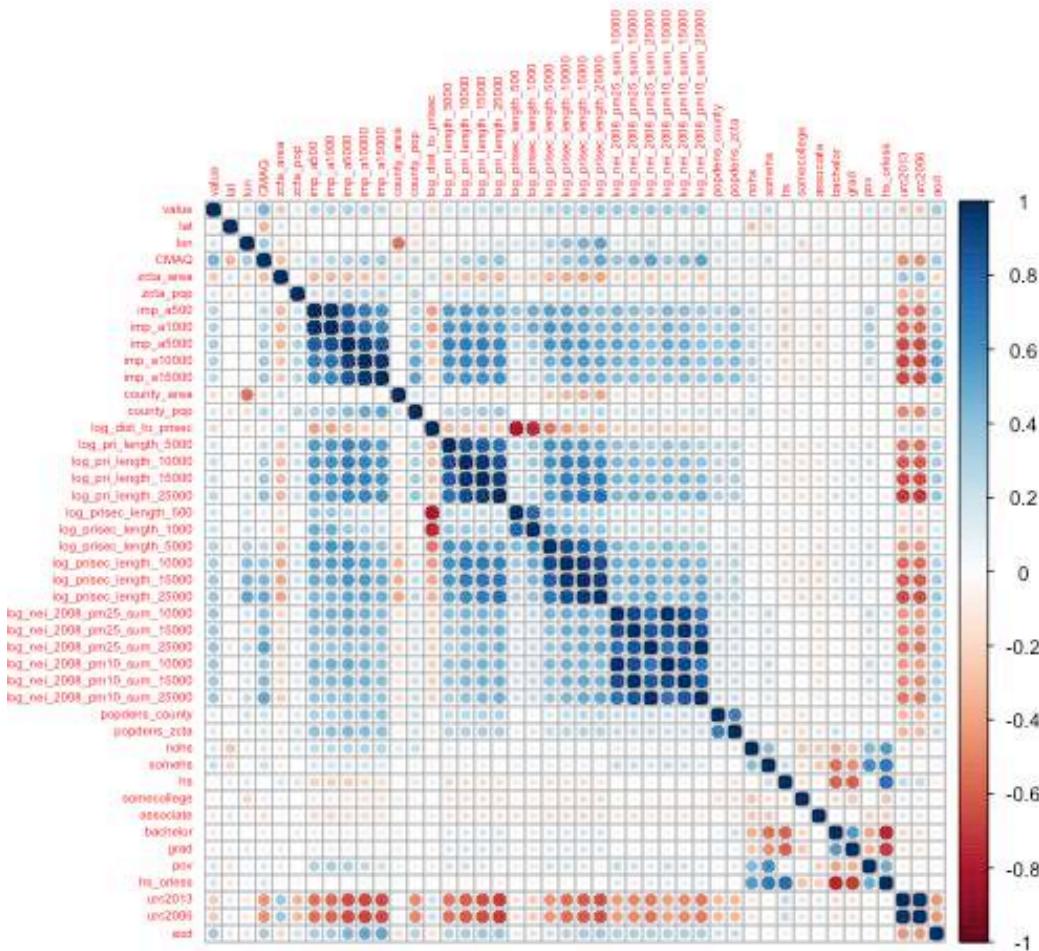
accuracy and it can cause our algorithm to be unnecessarily slower. Finally, it can also make it difficult to interpret what variables are actually predictive.

Intuitively we can expect some of our variables to be correlated.

The `corrplot` package is another option to look at correlation among possible predictors, and particularly useful if we have many predictors.

First, we calculate the Pearson correlation coefficients between all features pairwise using the `cor()` function of the `stats` package (which is loaded automatically). Then we use the `corrplot::corrplot()` function. First we need to select only the numeric variables using `dplyr`.

```
# install.packages("corrplot")
library(corrplot)
corrplot 0.90 loaded
PM_cor <- cor(pm %>% dplyr::select_if(is.numeric))
corrplot::corrplot(PM_cor, tl.cex = 0.5)
```



plot of chunk unnamed-chunk-87

We can see that the development variables (`imp`) variables are correlated with each other as we might expect. We also see that the road density variables seem to be correlated with each other, and the emission variables seem to be correlated with each other.

Also notice that none of the predictors are highly correlated with our outcome variable (`value`).

Now that we have a sense of what our data are, we can get started with building a machine learning model to predict air pollution.

Splitting the Data

```
library(tidymodels)
— Attaching packages tidymodel\ 
  s 0.1.3 —
  modeldata      0.1.1      workflowsets 0.1.0
— Conflicts tidymodels_conf\
  licts() —
  rlang::%@%()      masks purrr::%@%
  rlang::as_function() masks purrr::as_function()
  glue::collapse()    masks dplyr::collapse()
  vctrs::data_frame() masks dplyr::data_frame(), tibble::data_frame()
  scales::discard()   masks purrr::discard()
  magrittr::extract() masks tidyverse::extract()
  dplyr::filter()     masks stats::filter()
  xts::first()        masks dplyr::first()
  recipes::fixed()    masks stringr::fixed()
  rlang::flatten()    masks jsonlite::flatten(), purrr::flatten()
  rlang::flatten_chr() masks purrr::flatten_chr()
  rlang::flatten_dbl() masks purrr::flatten_dbl()
  rlang::flatten_int() masks purrr::flatten_int()
  rlang::flatten_lgl() masks purrr::flatten_lgl()
  rlang::flatten_raw() masks purrr::flatten_raw()
  dbplyr::ident()     masks dplyr::ident()
  rlang::invoke()     masks purrr::invoke()
  dplyr::lag()         masks stats::lag()
  xts::last()          masks dplyr::last()
  rlang::list_along()  masks purrr::list_along()
  rlang::modify()      masks purrr::modify()
  rlang::prepend()     masks purrr::prepend()
  rpart::prune()       masks dials::prune()
  magrittr::set_names() masks rlang::set_names(), purrr::set_names()
  yardstick::spec()    masks readr::spec()
  rlang::splice()       masks purrr::splice()
  dbplyr::sql()         masks dplyr::sql()
  recipes::step()       masks stats::step()
  rlang::unbox()        masks jsonlite::unbox()
• Use tidymodels_prefer() to resolve common conflicts.
set.seed(1234)
```

```
pm_split <- rsample::initial_split(data = pm, prop = 2/3)
pm_split
<Analysis/Assess/Total>
<584/292/876>
```

We can see the number of monitors in our training, testing, and original data by typing in the name of our split object. The result will look like this: <training data sample number, testing data sample number, original sample number>

Importantly the `initial_split()` function only determines what rows of our `pm` dataframe should be assigned for training or testing, it does not actually split the data.

To extract the testing and training data we can use the `training()` and `testing()` functions also of the `rsample` package.

```
train_pm <- rsample::training(pm_split)
test_pm <- rsample::testing(pm_split)
```

Great!

Now let's make a recipe!

Making a Recipe

Now with our data, we will start by making a recipe for our training data. If you recall, the continuous outcome variable is `value` (the average annual gravimetric monitor PM \sim 2.5 \sim concentration in ug/m 3). Our features (or predictor variables) are all the other variables except the monitor ID, which is an `id` variable.

The reason not to include the `id` variable is because this variable includes the county number and a number designating which particular monitor the values came from (of the monitors there are in that county). Since this number is arbitrary and the county information is also given in the data, and the fact that each monitor only has one value in the `value` variable, nothing is gained by including this variable and it may instead introduce noise. However, it is useful to keep this data to take a look at what is happening later. We will show you what to do in this case in just a bit.

In the simplest case, we might use all predictors like this:

```
#install.packages(tidymodels)
library(tidymodels)
simple_rec <- train_pm %>%
  recipes::recipe(value ~ .)

simple_rec
Data Recipe
```

Inputs:

```
role #variables
outcome      1
predictor    49
```

Now, let's get back to the `id` variable. Instead of including it as a predictor variable, we could also use the `update_role()` function of the `recipes` package.

```
simple_rec <- train_pm %>%
  recipes::recipe(value ~ .) %>%
  recipes::update_role(id, new_role = "id variable")

simple_rec
Data Recipe
```

Inputs:

```
role #variables
id variable      1
outcome          1
predictor        48
```

This [link](#) and this [link](#) show the many options for recipe step functions.

All of the step functions look like `step_*`() with the * replaced with a name, except for the check functions which look like `check_*`().

There are several ways to select what variables to apply steps to:

1. Using `tidyselect` methods: `contains()`, `matches()`, `starts_with()`, `ends_with()`, `everything()`, `num_range()`

2. Using the type: `all_nominal()`, `all_numeric()`, `has_type()`
3. Using the role: `all_predictors()`, `all_outcomes()`, `has_role()`
4. Using the name - use the actual name of the variable/variables of interest

Let's try adding some steps to our recipe.

We want to dummy encode our categorical variables so that they are numeric as we plan to use a linear regression for our model.

We will use the one-hot encoding means that we do not simply encode our categorical variables numerically, as our numeric assignments can be interpreted by algorithms as having a particular rank or order. Instead, binary variables made of 1s and 0s are used to arbitrarily assign a numeric value that has no apparent order.

```
simple_rec %>%  
  step_dummy(state, county, city, zcta, one_hot = TRUE)  
Data Recipe
```

Inputs:

```
  role #variables  
id variable      1  
  outcome        1  
predictor       48
```

Operations:

```
Dummy variables from state, county, city, zcta
```

Our `fips` variable includes a numeric code for state and county - and therefore is essentially a proxy for county. Since we already have county, we will just use it and keep the `fips` ID as another ID variable.

We can remove the `fips` variable from the predictors using `update_role()` to make sure that the role is no longer "predictor".

We can make the role anything we want actually, so we will keep it something identifiable.

```
simple_rec %>%
  update_role("fips", new_role = "county id")
Data Recipe
```

Inputs:

role	#variables
county id	1
id variable	1
outcome	1
predictor	47

We also want to remove variables that appear to be redundant and are highly correlated with others, as we know from our exploratory data analysis that many of our variables are correlated with one another. We can do this using the `step_corr()` function.

We don't want to remove some of our variables, like the `CMAQ` and `aod` variables, we can specify this using the `-` sign before the names of these variables like so:

```
simple_rec %>%
  step_corr(all_predictors(), - CMAQ, - aod)
Data Recipe
```

Inputs:

role	#variables
id variable	1
outcome	1
predictor	48

Operations:

Correlation filter on `all_predictors()`, `-CMAQ`, `-aod`

It is also a good idea to remove variables with near-zero variance, which can be done with the `step_nzv()` function.

Variables have low variance if all the values are very similar, the values are very sparse, or if they are highly imbalanced. Again we don't want to remove our `CMAQ` and `aod` variables.

```
simple_rec %>%
  step_nzv(all_predictors(), - CMAQ, - aod)
```

Data Recipe

Inputs:

	role #variables
id variable	1
outcome	1
predictor	48

Operations:

Sparse, unbalanced variable filter on `all_predictors()`, -CMAQ, -aod

Let's put all this together now.

Remember: it is important to add the steps to the recipe in an order that makes sense just like with a cooking recipe.

First, we are going to create numeric values for our categorical variables, then we will look at correlation and near-zero variance. Again, we do not want to remove the `CMAQ` and `aod` variables, so we can make sure they are kept in the model by excluding them from those steps. If we specifically wanted to remove a predictor we could use `step_rm()`.

```
simple_rec <- train_pm %>%
  recipes::recipe(value ~ .) %>%
  recipes::update_role(id, new_role = "id variable") %>%
  update_role("fips", new_role = "county id") %>%
  step_dummy(state, county, city, zcta, one_hot = TRUE) %>%
  step_corr(all_predictors(), - CMAQ, - aod) %>%
  step_nzv(all_predictors(), - CMAQ, - aod)
```

simple_rec
Data Recipe

Inputs:

	role #variables
county id	1

```
id variable      1  
  outcome       1  
 predictor     47
```

Operations:

```
Dummy variables from state, county, city, zcta  
Correlation filter on all_predictors(), -CMAQ, -aod  
Sparse, unbalanced variable filter on all_predictors(), -CMAQ, -aod
```

Nice! Now let's check our preprocessing.

Running Preprocessing

First we need to use the `prep()` function of the `recipes` package to prepare for preprocessing. However, we will specify that we also want to run and retain the preprocessing for the training data using the `retain = TRUE` argument.

```
prepped_rec <- prep(simple_rec, verbose = TRUE, retain = TRUE)  
oper 1 step dummy [training]  
oper 2 step corr [training]  
Warning in cor(x, use = use, method = method): the standard deviation is zero  
Warning: The correlation matrix has missing values. 273 columns were excluded  
from the filter.  
oper 3 step nzv [training]  
The retained training set is ~ 0.26 Mb in memory.  
names(prepped_rec)  
[1] "var_info"        "term_info"       "steps"          "template"  
[5] "levels"          "retained"        "tr_info"        "orig_lvls"  
[9] "last_term_info"
```

Since we retained our preprocessed training data (i.e. `prep(retain=TRUE)`), we can take a look at it by using the `bake()` function of the `recipes` package like this (this previously used the `juice()` function):

```
preproc_train <- bake(prepped_rec, new_data = NULL)
glimpse(preproc_train)
Rows: 584
Columns: 37
$ id                               <fct> 18003.0004, 55041.0007, 6065.1003, 39009.0...
$ fips                             <fct> 18003, 55041, 6065, 39009, 39061, 24510, 6...
$ lat                                <dbl> 41.09497, 45.56300, 33.94603, 39.44217, 39...
$ lon                                <dbl> -85.10182, -88.80880, -117.40063, -81.9088...
$ CMAQ                             <dbl> 10.383231, 3.411247, 11.404085, 7.971165, ...
$ zcta_area                         <dbl> 16696709, 370280916, 41957182, 132383592, ...
$ zcta_pop                          <dbl> 21306, 4141, 44001, 1115, 6566, 934, 41192...
$ imp_a500                           <dbl> 28.9783737, 0.0000000, 30.3901384, 0.00000...
$ imp_a15000                         <dbl> 13.0547959, 0.3676404, 23.7457506, 0.33079...
$ county_area                       <dbl> 1702419942, 2626421270, 18664696661, 13043...
$ county_pop                        <dbl> 355329, 9304, 2189641, 64757, 802374, 6209...
$ log_dist_to_prisec                <dbl> 6.621891, 8.415468, 7.419762, 6.344681, 5...
$ log_pri_length_5000               <dbl> 8.517193, 8.517193, 10.150514, 8.517193, 9...
$ log_pri_length_25000              <dbl> 12.77378, 10.16440, 13.14450, 10.12663, 13...
$ log_prisec_length_500             <dbl> 6.214608, 6.214608, 6.214608, 6.214608, 7...
$ log_prisec_length_1000            <dbl> 9.240294, 7.600902, 7.600902, 8.793450, 8...
$ log_prisec_length_5000            <dbl> 11.485093, 9.425537, 10.155961, 10.562382...
$ log_prisec_length_10000            <dbl> 12.75582, 11.44833, 11.59563, 11.69093, 12...
$ log_nei_2008_pm10_sum_10000        <dbl> 4.91110140, 3.86982666, 4.03184660, 0.0000...
$ log_nei_2008_pm10_sum_15000        <dbl> 5.399131, 3.883689, 5.459257, 0.000000, 6...
$ log_nei_2008_pm10_sum_25000        <dbl> 5.816047, 3.887264, 6.884537, 3.765635, 6...
$ popdens_county                    <dbl> 208.719947, 3.542463, 117.314577, 49.64834...
$ popdens_zcta                      <dbl> 1276.059851, 11.183401, 1048.711994, 8.422...
$ nohs                             <dbl> 4.3, 5.1, 3.7, 4.8, 2.1, 0.0, 2.5, 7.7, 0...
$ somehs                           <dbl> 6.7, 10.4, 5.9, 11.5, 10.5, 0.0, 4.3, 7.5, ...
$ hs                                <dbl> 31.7, 40.3, 17.9, 47.3, 30.0, 0.0, 17.8, 2...
$ somecollege                      <dbl> 27.2, 24.1, 26.3, 20.0, 27.1, 0.0, 26.1, 2...
$ associate                         <dbl> 8.2, 7.4, 8.3, 3.1, 8.5, 71.4, 13.2, 7.6, ...
$ bachelor                           <dbl> 15.0, 8.6, 20.2, 9.8, 14.2, 0.0, 23.4, 17...
$ grad                               <dbl> 6.8, 4.2, 17.7, 3.5, 7.6, 28.6, 12.6, 12.3...
$ pov                                <dbl> 13.500, 18.900, 6.700, 14.400, 12.500, 3.5...
$ hs_orless                          <dbl> 42.7, 55.8, 27.5, 63.6, 42.6, 0.0, 24.6, 3...
$ urc2006                            <dbl> 3, 6, 1, 5, 1, 1, 2, 1, 2, 6, 4, 4, 4, 4, ...
$ aod                                <dbl> 54.11111, 31.16667, 83.12500, 33.36364, 50...
$ value                             <dbl> 11.699065, 6.956780, 13.289744, 10.742000, ...
```

```
$ state_California          <dbl> 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, ...
$ city_Not.in.a.city       <dbl> 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, ...
```

Notice that this requires the `new_data = NULL` argument when we are using the training data.

For easy comparison sake - here is our original data:

```
glimpse(pm)
Rows: 876
Columns: 50
$ id                      <fct> 1003.001, 1027.0001, 1033.1002, 1049.1003, ...
$ value                   <dbl> 9.597647, 10.800000, 11.212174, 11.659091, ...
$ fips                     <fct> 1003, 1027, 1033, 1049, 1055, 1069, 1073, ...
$ lat                      <dbl> 30.49800, 33.28126, 34.75878, 34.28763, 33...
$ lon                      <dbl> -87.88141, -85.80218, -87.65056, -85.96830...
$ state                    <chr> "Alabama", "Alabama", "Alabama", "Alabama", ...
$ county                   <chr> "Baldwin", "Clay", "Colbert", "DeKalb", "E...
$ city                     <chr> "Fairhope", "Ashland", "Muscle Shoals", "C...
$ CMAQ                     <dbl> 8.098836, 9.766208, 9.402679, 8.534772, 9...
$ zcta                     <fct> 36532, 36251, 35660, 35962, 35901, 36303, ...
$ zcta_area                <dbl> 190980522, 374132430, 16716984, 203836235, ...
$ zcta_pop                 <dbl> 27829, 5103, 9042, 8300, 20045, 30217, 901...
$ imp_a500                 <dbl> 0.01730104, 1.96972318, 19.17301038, 5.782...
$ imp_a1000                <dbl> 1.4096021, 0.8531574, 11.1448962, 3.867647...
$ imp_a5000                <dbl> 3.3360118, 0.9851479, 15.1786154, 1.231141...
$ imp_a10000               <dbl> 1.9879187, 0.5208189, 9.7253870, 1.0316469...
$ imp_a15000               <dbl> 1.4386207, 0.3359198, 5.2472094, 0.9730444...
$ county_area              <dbl> 4117521611, 1564252280, 1534877333, 201266...
$ county_pop               <dbl> 182265, 13932, 54428, 71109, 104430, 10154...
$ log_dist_to_prisec       <dbl> 4.648181, 7.219907, 5.760131, 3.721489, 5...
$ log_pri_length_5000      <dbl> 8.517193, 8.517193, 8.517193, 8.517193, 9...
$ log_pri_length_10000     <dbl> 9.210340, 9.210340, 9.274303, 10.409411, 1...
$ log_pri_length_15000     <dbl> 9.630228, 9.615805, 9.658899, 11.173626, 1...
$ log_pri_length_25000     <dbl> 11.32735, 10.12663, 10.15769, 11.90959, 12...
$ log_prisec_length_500    <dbl> 7.295356, 6.214608, 8.611945, 7.310155, 8...
$ log_prisec_length_1000   <dbl> 8.195119, 7.600902, 9.735569, 8.585843, 9...
$ log_prisec_length_5000   <dbl> 10.815042, 10.170878, 11.770407, 10.214200...
$ log_prisec_length_10000  <dbl> 11.88680, 11.40554, 12.84066, 11.50894, 12...
$ log_prisec_length_15000  <dbl> 12.205723, 12.042963, 13.282656, 12.353663...
```

```
$ log_prisec_length_25000      <dbl> 13.41395, 12.79980, 13.79973, 13.55979, 13...
$ log_nei_2008_pm25_sum_10000 <dbl> 0.318035438, 3.218632928, 6.573127301, 0.0...
$ log_nei_2008_pm25_sum_15000 <dbl> 1.967358961, 3.218632928, 6.581917457, 3.2...
$ log_nei_2008_pm25_sum_25000 <dbl> 5.067308, 3.218633, 6.875900, 4.887665, 4...
$ log_nei_2008_pm10_sum_10000 <dbl> 1.35588511, 3.31111648, 6.69187313, 0.0000...
$ log_nei_2008_pm10_sum_15000 <dbl> 2.26783411, 3.31111648, 6.70127741, 3.3500...
$ log_nei_2008_pm10_sum_25000 <dbl> 5.628728, 3.311116, 7.148858, 5.171920, 4...
$ popdens_county              <dbl> 44.265706, 8.906492, 35.460814, 35.330814, ...
$ popdens_zcta                <dbl> 145.716431, 13.639555, 540.887040, 40.7189...
$ nohs                         <dbl> 3.3, 11.6, 7.3, 14.3, 4.3, 5.8, 7.1, 2.7, ...
$ somehs                        <dbl> 4.9, 19.1, 15.8, 16.7, 13.3, 11.6, 17.1, 6...
$ hs                            <dbl> 25.1, 33.9, 30.6, 35.0, 27.8, 29.8, 37.2, ...
$ somecollege                  <dbl> 19.7, 18.8, 20.9, 14.9, 29.2, 21.4, 23.5, ...
$ associate                     <dbl> 8.2, 8.0, 7.6, 5.5, 10.1, 7.9, 7.3, 8.0, 4...
$ bachelor                      <dbl> 25.3, 5.5, 12.7, 7.9, 10.0, 13.7, 5.9, 17...
$ grad                          <dbl> 13.5, 3.1, 5.1, 5.8, 5.4, 9.8, 2.0, 8.7, 2...
$ pov                           <dbl> 6.1, 19.5, 19.0, 13.8, 8.8, 15.6, 25.5, 7...
$ hs_orless                     <dbl> 33.3, 64.6, 53.7, 66.0, 45.4, 47.2, 61.4, ...
$ urc2013                       <dbl> 4, 6, 4, 6, 4, 4, 1, 1, 1, 1, 1, 1, 1, 2, ...
$ urc2006                       <dbl> 5, 6, 4, 5, 4, 4, 1, 1, 1, 1, 1, 1, 1, 2, ...
$ aod                           <dbl> 37.36364, 34.81818, 36.00000, 33.08333, 43...
```

Notice how we only have 36 variables now instead of 50! Two of these are our ID variables (`fips` and the actual monitor ID (`id`)) and one is our outcome (`value`). Thus we only have 33 predictors now. We can also see that we no longer have any categorical variables. Variables like `state` are gone and only `state_California` remains as it was the only state identity to have nonzero variance. We can also see that there were more monitors listed as "Not in a city" than any city.

Extract preprocessed testing data using `bake()`

According to the `tidymodels` documentation:

`bake()` takes a trained recipe and applies the operations to a data set to create a design matrix. For example: it applies the centering to new data sets using these means used to create the recipe.

Therefore, if you wanted to look at the preprocessed testing data you would use the `bake()` function of the `recipes` package.

```
baked_test_pm <- recipes::bake(prepped_rec, new_data = test_pm)
Warning: There are new levels in a factor: Colbert, Etowah, Russell, Walker,
Cochise, Coconino, Mohave, Arkansas, Crittenden, Garland, Phillips, Pope,
Calaveras, Humboldt, Inyo, Merced, Monterey, San Benito, Sonoma, Tulare,
Arapahoe, Elbert, Larimer, Mesa, Brevard, Leon, Sarasota, Paudling, Ada,
Benewah, Lemhi, Shoshone, DuPage, McHenry, Winnebago, Elkhart, St. Joseph,
Tippecanoe, Vanderburgh, Black Hawk, Pottawattamie, Van Buren, Wright, Boyd,
Christian, Daviess, Hardin, Kenton, Pike, Calcasieu, St. Bernard, Tangipahoa,
Baltimore, Cecil, Bristol, Berrien, Missaukee, Muskegon, Oakland, Dakota,
Olmsted, Grenada, Lauderdale, Cascade, Sanders, Silver Bow, Scotts Bluff,
Merrimack, Rockingham, Atlantic, Gloucester, Lea, San Juan, Santa Fe, Onondaga,
Steuben, Westchester, Martin, Pitt, Swain, Watauga, Billings, Lorain, Muskogee,
Sequoyah, Deschutes, Multnomah, Umatilla, Berks, Bucks, Cambria, Centre,
Lackawanna, Northampton, York, Edgefield, Florence, Pennington, Bowie, Ector,
Cache, Frederick, Bristol City, Norfolk City, Berkeley, Wood, La Crosse,
Ozaukee, Waukesha, Laramie, Sublette
Warning: There are new levels in a factor: Muscle Shoals, Gadsden, Dothan,
Chickasaw, Montgomery, Phenix City, Douglas, Flagstaff, Peach Springs,
Apache Junction, Stuttgart, Hot Springs (Hot Springs National Park), Newport,
Russellville, Springdale, San Andreas, Eureka, El Centro, Keeler, Lakeport,
Azusa, Burbank, Merced, Salinas, Grass Valley, Anaheim, Quincy, Hollister,
Ontario, Victorville, San Bernardino, Chula Vista, San Diego, San Jose, Live
Oak, Santa Rosa, Visalia, Piru, Simi Valley, Littleton, Fort Collins, Grand
Junction, Bridgeport, Dover, Wilmington, Melbourne, Tampa, Tallahassee, Palm
Springs North, Orlando, Belle Glade, Ridge Wood Heights, Savannah, Kennesaw,
Doraville, Boise (corporate name Boise City), Pinehurst (Pine Creek), Champaign,
Summit, Naperville, Elgin, Zion, Cary, Alton, Braidwood, Rockford, Elkhart,
Anderson, South Bend, Evansville, Waterloo, Keokuk, Council Bluffs, Clarion,
Frankfort, Elizabethtown, Covington, Richmond, Pikeville, Vinton, Lake Charles,
Chalmette, Cockeysville, Essex, Bladensburg, Fall River, Lynn, Bay City, Coloma,
Luna Pier, Muskegon, Oak Park, Livonia, Dearborn, Apple Valley, Richfield,
Bayport, Grenada, Gulfport, Meridian, Great Falls, Columbia Falls, Kalispell,
Seeley Lake, Thompson Falls, Butte-Silver Bow (Remainder), Scottsbluff, Blair,
Las Vegas, Jean, Suncook, Galloway (Township of), Atlantic City, Camden,
Greenwich (Township of), Phillipsburg, Silver City, Hobbs, Farmington, Santa Fe,
Buffalo, East Syracuse, Mamaroneck, Fayetteville, Jamesville, Bryson City (RR
name Bryson), Boone, Fairfield, Newburgh Heights, Sheffield, Dayton, Muskogee,
Marble City Community, Bend, Central Point, Portland, Pendleton, La Grande,
Hillsboro, Pittsburgh, Harrison Township, Bristol, Johnstown, State College,
```

Erie, Scranton, Norristown, Freemansburg, York, Pawtucket, East Providence,
Dentsville (Dents), Rapid City, Chattanooga, Texarkana, Odessa, Baytown,
Logan, Harrisville, Bensley, Annandale, Norfolk, Martinsburg, South Charleston,
Fairmont, Vienna, Green Bay, La Crosse, Waukesha, Cheyenne, Pinedale

```
glimpse(baked_test_pm)
```

Rows: 292

Columns: 37

```
$ id                      <fct> 1033.1002, 1055.001, 1069.0003, 1073.0023, ...
$ fips                    <fct> 1033, 1055, 1069, 1073, 1073, 1073, ...
$ lat                     <dbl> 34.75878, 33.99375, 31.22636, 33.55306, 33...
$ lon                     <dbl> -87.65056, -85.99107, -85.39077, -86.81500...
$ CMAQ                    <dbl> 9.402679, 9.241744, 9.121892, 10.235612, 1...
$ zcta_area                <dbl> 16716984, 154069359, 162685124, 26929603, ...
$ zcta_pop                 <dbl> 9042, 20045, 30217, 9010, 16140, 3699, 137...
$ imp_a500                  <dbl> 19.17301038, 16.49307958, 19.13927336, 41...
$ imp_a1500                 <dbl> 5.2472094, 5.1612102, 4.7401296, 17.452484...
$ county_area               <dbl> 1534877333, 1385618994, 1501737720, 287819...
$ county_pop                <dbl> 54428, 104430, 101547, 658466, 658466, 194...
$ log_dist_to_prisec        <dbl> 5.760131, 5.261457, 7.112373, 6.600958, 6...
$ log_pri_length_5000       <dbl> 8.517193, 9.066563, 8.517193, 11.156977, 1...
$ log_pri_length_25000      <dbl> 10.15769, 12.01356, 10.12663, 12.98762, 12...
$ log_prisec_length_500     <dbl> 8.611945, 8.740680, 6.214608, 6.214608, 6...
$ log_prisec_length_1000    <dbl> 9.735569, 9.627898, 7.600902, 9.075921, 8...
$ log_prisec_length_5000    <dbl> 11.770407, 11.728889, 12.298627, 12.281645...
$ log_prisec_length_10000   <dbl> 12.840663, 12.768279, 12.994141, 13.278416...
$ log_nei_2008_pm10_sum_10000 <dbl> 6.69187313, 4.43719884, 0.92888890, 8.2097...
$ log_nei_2008_pm10_sum_15000 <dbl> 6.70127741, 4.46267932, 3.67473904, 8.6488...
$ log_nei_2008_pm10_sum_25000 <dbl> 7.148858, 4.678311, 3.744629, 8.858019, 8...
$ popdens_county            <dbl> 35.460814, 75.367038, 67.619664, 228.77763...
$ popdens_zcta              <dbl> 540.8870404, 130.1037411, 185.7391706, 334...
$ nohs                      <dbl> 7.3, 4.3, 5.8, 7.1, 2.7, 11.1, 9.7, 3.0, 8...
$ somehs                     <dbl> 15.8, 13.3, 11.6, 17.1, 6.6, 11.6, 21.6, 1...
$ hs                         <dbl> 30.6, 27.8, 29.8, 37.2, 30.7, 46.0, 39.3, ...
$ somecollege                <dbl> 20.9, 29.2, 21.4, 23.5, 25.7, 17.2, 21.6, ...
$ associate                  <dbl> 7.6, 10.1, 7.9, 7.3, 8.0, 4.1, 5.2, 6.6, 4...
$ bachelor                   <dbl> 12.7, 10.0, 13.7, 5.9, 17.6, 7.1, 2.2, 7.8...
$ grad                        <dbl> 5.1, 5.4, 9.8, 2.0, 8.7, 2.9, 0.4, 4.2, 3...
$ pov                          <dbl> 19.0, 8.8, 15.6, 25.5, 7.3, 8.1, 13.3, 23...
$ hs_orless                  <dbl> 53.7, 45.4, 47.2, 61.4, 40.0, 68.7, 70.6, ...
```

```
$ urc2006          <dbl> 4, 4, 4, 1, 1, 1, 2, 3, 3, 3, 2, 5, 4, 1, ...
$ aod              <dbl> 36.000000, 43.416667, 33.000000, 39.583333...
$ value            <dbl> 11.212174, 12.375394, 10.508850, 15.591017...
$ state_California <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ city_Not.in.a.city <dbl> NA, NA, NA, 0, 1, 1, 1, NA, NA, NA, 0, NA, ...
```

Notice that our `city_Not.in.a.city` variable seems to be NA values. Why might that be?

Ah! Perhaps it is because some of our levels were not previously seen in the training set!

Let's take a look using the `set operations` of the `dplyr` package. We can take a look at cities that were different between the test and training set.

```
traincities <- train_pm %>% distinct(city)
testcities <- test_pm %>% distinct(city)

#get the number of cities that were different
dim(dplyr::setdiff(traincities, testcities))
[1] 381   1

#get the number of cities that overlapped
dim(dplyr::intersect(traincities, testcities))
[1] 55   1
```

Indeed, there are lots of different cities in our test data that are not in our training data!

So, let's go back to our `pm` dataset and modify the `city` variable to just be values of `in a city` or `not in a city` using the `case_when()` function of `dplyr`. This function allows you to vectorize multiple `if_else()` statements.

```
pm %>%
  mutate(city = case_when(city == "Not in a city" ~ "Not in a city",
                         city != "Not in a city" ~ "In a city"))

# A tibble: 876 x 50
  id      value fips     lat     lon state    county    city    CMAQ zcta  zcta_a\
rea
  <fct>    <dbl> <fct> <dbl> <dbl> <chr>    <chr>    <chr>  <dbl> <fct>    <d\
b1>
  1 1003.001  9.60 1003    30.5 -87.9 Alabama Baldwin    In a...  8.10 36532 190980\
522
```

```

2 1027.0001 10.8 1027 33.3 -85.8 Alabama Clay In a... 9.77 36251 374132\
430
3 1033.1002 11.2 1033 34.8 -87.7 Alabama Colbert In a... 9.40 35660 16716\
984
4 1049.1003 11.7 1049 34.3 -86.0 Alabama DeKalb In a... 8.53 35962 203836\
235
5 1055.001 12.4 1055 34.0 -86.0 Alabama Etowah In a... 9.24 35901 154069\
359
6 1069.0003 10.5 1069 31.2 -85.4 Alabama Houston In a... 9.12 36303 162685\
124
7 1073.0023 15.6 1073 33.6 -86.8 Alabama Jefferson In a... 10.2 35207 26929\
603
8 1073.1005 12.4 1073 33.3 -87.0 Alabama Jefferson Not ... 10.2 35111 166239\
542
9 1073.1009 11.1 1073 33.5 -87.3 Alabama Jefferson Not ... 8.16 35444 385566\
685
10 1073.101 13.1 1073 33.5 -86.5 Alabama Jefferson In a... 9.30 35094 148994\
881
# ... with 866 more rows, and 39 more variables: zcta_pop <dbl>, imp_a500 <dbl>,
# imp_a1000 <dbl>, imp_a5000 <dbl>, imp_a10000 <dbl>, imp_a15000 <dbl>,
# county_area <dbl>, county_pop <dbl>, log_dist_to_prisec <dbl>,
# log_pri_length_5000 <dbl>, log_pri_length_10000 <dbl>,
# log_pri_length_15000 <dbl>, log_pri_length_25000 <dbl>,
# log_prisec_length_500 <dbl>, log_prisec_length_1000 <dbl>,
# log_prisec_length_5000 <dbl>, log_prisec_length_10000 <dbl>, ...

```

Alternatively you could create a [custom step function](#) to do this and add this to your recipe, but that is beyond the scope of this case study.

We will need to repeat all the steps (splitting the data, preprocessing, etc) as the levels of our variables have now changed.

While we are doing this, we might also have this issue for `county`.

The `county` variables appears to get dropped due to either correlation or near zero variance. It is likely due to near zero variance because this is the more granular of these geographic categorical variables and likely sparse.

```

pm %<>%
  mutate(city = case_when(city == "Not in a city" ~ "Not in a city",
                          city != "Not in a city" ~ "In a city"))

set.seed(1234) # same seed as before
pm_split <- rsample::initial_split(data = pm, prop = 2/3)
pm_split
<Analysis/Assess/Total>
<584/292/876>
train_pm <- rsample::training(pm_split)
test_pm <- rsample::testing(pm_split)

```

Now we will create a new recipe:

```

novel_rec <- train_pm %>%
  recipe() %>%
  update_role(everything(), new_role = "predictor") %>%
  update_role(value, new_role = "outcome") %>%
  update_role(id, new_role = "id variable") %>%
  update_role("fips", new_role = "county id") %>%
  step_dummy(state, county, city, zcta, one_hot = TRUE) %>%
  step_corr(all_numeric()) %>%
  step_nzv(all_numeric())

```

Now we will check the preprocessed data again to see if we still have NA values.

```

prepped_rec <- prep(novel_rec, verbose = TRUE, retain = TRUE)
oper 1 step dummy [training]
oper 2 step corr [training]
Warning in cor(x, use = use, method = method): the standard deviation is zero
Warning: The correlation matrix has missing values. 273 columns were excluded
from the filter.
oper 3 step nzv [training]
The retained training set is ~ 0.27 Mb in memory.
preproc_train <- bake(prepped_rec, new_data = NULL)
glimpse(preproc_train)
Rows: 584
Columns: 38

```

```

$ id <fct> 18003.0004, 55041.0007, 6065.1003, 39009.0...
$ value <dbl> 11.699065, 6.956780, 13.289744, 10.742000, ...
$ fips <fct> 18003, 55041, 6065, 39009, 39061, 24510, 6...
$ lat <dbl> 41.09497, 45.56300, 33.94603, 39.44217, 39...
$ lon <dbl> -85.10182, -88.80880, -117.40063, -81.9088...
$ CMAQ <dbl> 10.383231, 3.411247, 11.404085, 7.971165, ...
$ zcta_area <dbl> 16696709, 370280916, 41957182, 132383592, ...
$ zcta_pop <dbl> 21306, 4141, 44001, 1115, 6566, 934, 41192...
$ imp_a500 <dbl> 28.9783737, 0.0000000, 30.3901384, 0.00000...
$ imp_a1500 <dbl> 13.0547959, 0.3676404, 23.7457506, 0.33079...
$ county_area <dbl> 1702419942, 2626421270, 18664696661, 13043...
$ county_pop <dbl> 355329, 9304, 2189641, 64757, 802374, 6209...
$ log_dist_to_prisec <dbl> 6.621891, 8.415468, 7.419762, 6.344681, 5...
$ log_pri_length_5000 <dbl> 8.517193, 8.517193, 10.150514, 8.517193, 9...
$ log_pri_length_25000 <dbl> 12.77378, 10.16440, 13.14450, 10.12663, 13...
$ log_prisec_length_500 <dbl> 6.214608, 6.214608, 6.214608, 6.214608, 7...
$ log_prisec_length_1000 <dbl> 9.240294, 7.600902, 7.600902, 8.793450, 8...
$ log_prisec_length_5000 <dbl> 11.485093, 9.425537, 10.155961, 10.562382, ...
$ log_prisec_length_10000 <dbl> 12.75582, 11.44833, 11.59563, 11.69093, 12...
$ log_prisec_length_25000 <dbl> 13.98749, 13.15082, 13.44293, 13.58697, 14...
$ log_nei_2008_pm10_sum_10000 <dbl> 4.91110140, 3.86982666, 4.03184660, 0.0000...
$ log_nei_2008_pm10_sum_15000 <dbl> 5.399131, 3.883689, 5.459257, 0.000000, 6...
$ log_nei_2008_pm10_sum_25000 <dbl> 5.816047, 3.887264, 6.884537, 3.765635, 6...
$ popdens_county <dbl> 208.719947, 3.542463, 117.314577, 49.64834...
$ popdens_zcta <dbl> 1276.059851, 11.183401, 1048.711994, 8.422...
$ nohs <dbl> 4.3, 5.1, 3.7, 4.8, 2.1, 0.0, 2.5, 7.7, 0...
$ somehs <dbl> 6.7, 10.4, 5.9, 11.5, 10.5, 0.0, 4.3, 7.5, ...
$ hs <dbl> 31.7, 40.3, 17.9, 47.3, 30.0, 0.0, 17.8, 2...
$ somecollege <dbl> 27.2, 24.1, 26.3, 20.0, 27.1, 0.0, 26.1, 2...
$ associate <dbl> 8.2, 7.4, 8.3, 3.1, 8.5, 71.4, 13.2, 7.6, ...
$ bachelor <dbl> 15.0, 8.6, 20.2, 9.8, 14.2, 0.0, 23.4, 17...
$ grad <dbl> 6.8, 4.2, 17.7, 3.5, 7.6, 28.6, 12.6, 12.3...
$ pov <dbl> 13.500, 18.900, 6.700, 14.400, 12.500, 3.5...
$ hs_orless <dbl> 42.7, 55.8, 27.5, 63.6, 42.6, 0.0, 24.6, 3...
$ urc2006 <dbl> 3, 6, 1, 5, 1, 1, 2, 1, 2, 6, 4, 4, 4, 4, ...
$ aod <dbl> 54.11111, 31.16667, 83.12500, 33.36364, 50...
$ state_California <dbl> 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, ...
$ city_Not.in.a.city <dbl> 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, ...
baked_test_pm <- recipes::bake(prepped_rec, new_data = test_pm)

```

Warning: There are new levels in a factor: Colbert, Etowah, Russell, Walker, Cochise, Coconino, Mohave, Arkansas, Crittenden, Garland, Phillips, Pope, Calaveras, Humboldt, Inyo, Merced, Monterey, San Benito, Sonoma, Tulare, Arapahoe, Elbert, Larimer, Mesa, Brevard, Leon, Sarasota, Paudling, Ada, Benewah, Lemhi, Shoshone, DuPage, McHenry, Winnebago, Elkhart, St. Joseph, Tippecanoe, Vanderburgh, Black Hawk, Pottawattamie, Van Buren, Wright, Boyd, Christian, Daviess, Hardin, Kenton, Pike, Calcasieu, St. Bernard, Tangipahoa, Baltimore, Cecil, Bristol, Berrien, Missaukee, Muskegon, Oakland, Dakota, Olmsted, Grenada, Lauderdale, Cascade, Sanders, Silver Bow, Scotts Bluff, Merrimack, Rockingham, Atlantic, Gloucester, Lea, San Juan, Santa Fe, Onondaga, Steuben, Westchester, Martin, Pitt, Swain, Watauga, Billings, Lorain, Muskogee, Sequoyah, Deschutes, Multnomah, Umatilla, Berks, Bucks, Cambria, Centre, Lackawanna, Northampton, York, Edgefield, Florence, Pennington, Bowie, Ector, Cache, Frederick, Bristol City, Norfolk City, Berkeley, Wood, La Crosse, Ozaukee, Waukesha, Laramie, Sublette

```
glimpse(baked_test_pm)
```

Rows: 292

Columns: 38

Column	Type	Value
\$ id	<fct>	1033.1002, 1055.001, 1069.0003, 1073.0023, ...
\$ value	<dbl>	11.212174, 12.375394, 10.508850, 15.591017...
\$ fips	<fct>	1033, 1055, 1069, 1073, 1073, 1073, 1073, ...
\$ lat	<dbl>	34.75878, 33.99375, 31.22636, 33.55306, 33...
\$ lon	<dbl>	-87.65056, -85.99107, -85.39077, -86.81500...
\$ CMAQ	<dbl>	9.402679, 9.241744, 9.121892, 10.235612, 1...
\$ zcta_area	<dbl>	16716984, 154069359, 162685124, 26929603, ...
\$ zcta_pop	<dbl>	9042, 20045, 30217, 9010, 16140, 3699, 137...
\$ imp_a500	<dbl>	19.17301038, 16.49307958, 19.13927336, 41...
\$ imp_a15000	<dbl>	5.2472094, 5.1612102, 4.7401296, 17.452484...
\$ county_area	<dbl>	1534877333, 1385618994, 1501737720, 287819...
\$ county_pop	<dbl>	54428, 104430, 101547, 658466, 658466, 194...
\$ log_dist_to_prisec	<dbl>	5.760131, 5.261457, 7.112373, 6.600958, 6...
\$ log_pri_length_5000	<dbl>	8.517193, 9.066563, 8.517193, 11.156977, 1...
\$ log_pri_length_25000	<dbl>	10.15769, 12.01356, 10.12663, 12.98762, 12...
\$ log_prisec_length_500	<dbl>	8.611945, 8.740680, 6.214608, 6.214608, 6...
\$ log_prisec_length_1000	<dbl>	9.735569, 9.627898, 7.600902, 9.075921, 8...
\$ log_prisec_length_5000	<dbl>	11.770407, 11.728889, 12.298627, 12.281645...
\$ log_prisec_length_10000	<dbl>	12.840663, 12.768279, 12.994141, 13.278416...
\$ log_prisec_length_25000	<dbl>	13.79973, 13.70026, 13.85550, 14.45221, 13...
\$ log_nei_2008_pm10_sum_10000	<dbl>	6.69187313, 4.43719884, 0.92888890, 8.2097...

```
$ log_nei_2008_pm10_sum_15000 <dbl> 6.70127741, 4.46267932, 3.67473904, 8.6488...
$ log_nei_2008_pm10_sum_25000 <dbl> 7.148858, 4.678311, 3.744629, 8.858019, 8...
$ popdens_county <dbl> 35.460814, 75.367038, 67.619664, 228.77763...
$ popdens_zcta <dbl> 540.8870404, 130.1037411, 185.7391706, 334...
$ nohs <dbl> 7.3, 4.3, 5.8, 7.1, 2.7, 11.1, 9.7, 3.0, 8...
$ somehs <dbl> 15.8, 13.3, 11.6, 17.1, 6.6, 11.6, 21.6, 1...
$ hs <dbl> 30.6, 27.8, 29.8, 37.2, 30.7, 46.0, 39.3, ...
$ somecollege <dbl> 20.9, 29.2, 21.4, 23.5, 25.7, 17.2, 21.6, ...
$ associate <dbl> 7.6, 10.1, 7.9, 7.3, 8.0, 4.1, 5.2, 6.6, 4...
$ bachelor <dbl> 12.7, 10.0, 13.7, 5.9, 17.6, 7.1, 2.2, 7.8...
$ grad <dbl> 5.1, 5.4, 9.8, 2.0, 8.7, 2.9, 0.4, 4.2, 3...
$ pov <dbl> 19.0, 8.8, 15.6, 25.5, 7.3, 8.1, 13.3, 23...
$ hs_orless <dbl> 53.7, 45.4, 47.2, 61.4, 40.0, 68.7, 70.6, ...
$ urc2006 <dbl> 4, 4, 4, 1, 1, 2, 3, 3, 3, 2, 5, 4, 1, ...
$ aod <dbl> 36.000000, 43.416667, 33.000000, 39.583333...
$ state_California <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ city_Not.in.a.city <dbl> 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, ...
```

Great, now we no longer have NA values!

Specifying the Model

The first step is to define what type of model we would like to use. For our case, we are going to start our analysis with a linear regression but we will demonstrate how we can try different models.

See [here](#) for modeling options in `parsnip`.

```
PM_model <- parsnip::linear_reg() # PM was used in the name for particulate matter
PM_model
Linear Regression Model Specification (regression)

Computational engine: lm
```

OK. So far, all we have defined is that we want to use a linear regression...
Let's tell `parsnip` more about what we want.

We would like to use the [ordinary least squares](#) method to fit our linear regression. So we will tell `parsnip` that we want to use the `lm` package to implement our linear regression (there are

many options actually such as `rstan`, `glmnet`, `keras`, and `sparklyr`). See [here](#) for a description of the differences and using these different engines with `parsnip`.

We will do so by using the `set_engine()` function of the `parsnip` package.

```
lm_PM_model <-  
  PM_model %>%  
  parsnip::set_engine("lm")  
  
lm_PM_model  
Linear Regression Model Specification (regression)  
  
Computational engine: lm
```

Here, we aim to predict the air pollution. You can do this with the `set_mode()` function of the `parsnip` package, by using either `set_mode("classification")` or `set_mode("regression")`.

```
lm_PM_model <-  
  PM_model %>%  
  parsnip::set_engine("lm") %>%  
  set_mode("regression")  
  
lm_PM_model  
Linear Regression Model Specification (regression)  
  
Computational engine: lm
```

Now we will use the `workflows` package to keep track of both our preprocessing steps and our model specification. It also allows us to implement fancier optimizations in an automated way.

If you recall `novel_rec` is the recipe we previously created with the `recipes` package and `lm_PM_model` was created when we specified our model with the `parsnip` package. Here, we combine everything together into a workflow.

```
PM_wf <- workflows::workflow() %>%
      workflows::add_recipe(novel_rec) %>%
      workflows::add_model(lm_PM_model)
PM_wf
== Workflow _____\`_____
=====
Preprocessor: Recipe
Model: linear_reg()

— Preprocessor _____\`_____
=====
3 Recipe Steps

• step_dummy()
• step_corr()
• step_nzv()

— Model _____\`_____
=====
Linear Regression Model Specification (regression)

Computational engine: lm
```

Ah, nice. Notice how it tells us about both our preprocessing steps and our model specifications.

Next, we “prepare the recipe” (or estimate the parameters) and fit the model to our training data all at once. Printing the output, we can see the coefficients of the model.

```
PM_wf_fit <- parsnip::fit(PM_wf, data = train_pm)
Warning in cor(x, use = use, method = method): the standard deviation is zero
Warning: The correlation matrix has missing values. 273 columns were excluded
from the filter.
PM_wf_fit
== Workflow [trained] _____\`_____
=====
Preprocessor: Recipe
Model: linear_reg()
```

— Preprocessor —————＼

3 Recipe Steps

- `step_dummy()`
- `step_corr()`
- `step_nzv()`

— Model —————＼

Call:

```
stats::lm(formula = ..y ~ ., data = data)
```

Coefficients:

	(Intercept)	lat
	2.936e+02	3.261e-02
lon		CMAQ
	1.586e-02	2.463e-01
zcta_area		zcta_pop
	-3.433e-10	1.013e-05
imp_a500		imp_a15000
	5.064e-03	-3.066e-03
county_area		county_pop
	-2.324e-11	-7.576e-08
log_dist_to_prisec		log_pri_length_5000
	6.214e-02	-2.006e-01
log_pri_length_25000		log_prisec_length_500
	-5.411e-02	2.204e-01
log_prisec_length_1000		log_prisec_length_5000
	1.154e-01	2.374e-01
log_prisec_length_10000		log_prisec_length_25000
	-3.436e-02	5.224e-01
log_nei_2008_pm10_sum_10000		log_nei_2008_pm10_sum_15000
	1.829e-01	-2.355e-02
log_nei_2008_pm10_sum_25000		popdens_county
	2.403e-02	2.203e-05
popdens_zcta		nohs
	-2.132e-06	-2.983e+00

somehs		hs	
-2.956e+00		-2.962e+00	
somecollege		associate	
-2.967e+00		-2.999e+00	
bachelor		grad	
-2.979e+00		-2.978e+00	
pov		hs_orless	
1.859e-03		NA	
urc2006		aod	
2.577e-01		1.535e-02	
state_California		city_Not.in.a.city	
3.114e+00		-4.250e-02	

Assessing the Model Fit

After we fit our model, we can use the `broom` package to look at the output from the fitted model in an easy/tidy way.

The `tidy()` function returns a tidy dataframe with coefficients from the model (one row per coefficient).

Many other `broom` functions currently only work with `parsnip` objects, not raw `workflows` objects.

However, we can use the `tidy` function if we first use the `pull_workflow_fit()` function.

```
wflowoutput <- PM_wflow_fit %>%
  pull_workflow_fit() %>%
  broom::tidy()

Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.
Please use `extract_fit_parsnip()` instead.

# A tibble: 36 × 5
  term      estimate std.error statistic   p.value
  <chr>     <dbl>    <dbl>     <dbl>    <dbl>
1 (Intercept) 2.94e+ 2  1.18e+ 2     2.49   0.0130
2 lat         3.26e- 2  2.28e- 2     1.43   0.153 
3 lon         1.59e- 2  1.01e- 2     1.58   0.115 
4 CMAQ        2.46e- 1  3.97e- 2     6.20   0.00000000108
5 zcta_area   -3.43e-10 1.60e-10    -2.15   0.0320
```

```
6 zcta_pop    1.01e- 5 5.33e- 6    1.90  0.0578
7 imp_a500    5.06e- 3 7.42e- 3    0.683  0.495
8 imp_a15000 -3.07e- 3 1.16e- 2   -0.263  0.792
9 county_area -2.32e-11 1.97e-11   -1.18  0.238
10 county_pop -7.58e- 8 9.29e- 8   -0.815  0.415
# ... with 26 more rows
```

We have fit our model on our training data, which means we have created a model to predict values of air pollution based on the predictors that we have included. Yay!

One last thing before we leave this section. We often are interested in getting a sense of which variables are the most important in our model. We can explore the variable importance using the `vip()` function of the `vip` package. This function creates a bar plot of variable importance scores for each predictor variable (or feature) in a model. The bar plot is ordered by importance (highest to smallest).

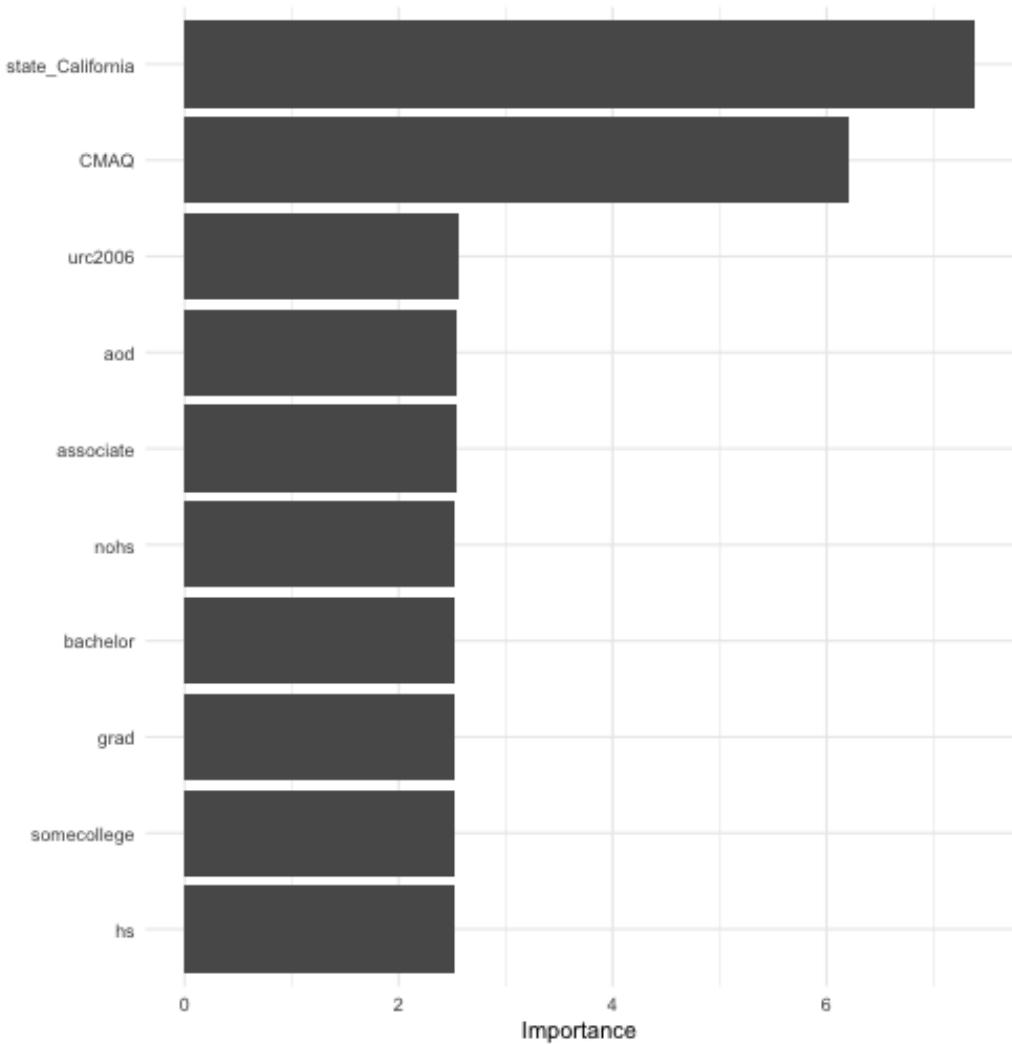
Notice again that we need to use the `pull_workflow_fit()` function.

Let's take a look at the top 10 contributing variables:

```
#install.packages(vip)
library(vip)

Attaching package: 'vip'
The following object is masked from 'package:utils':
  vi

PM_wf_low %>%
  pull_workflow_fit() %>%
  vip(num_features = 10)
Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.
Please use `extract_fit_parsnip()` instead.
```



plot of chunk unnamed-chunk-112

The state in which the monitor was located (in this case if it was in California or not), the CMAQ model, and the aod satellite information appear to be the most important for predicting the air pollution at a given monitor.

Model Performance

In this next section, our goal is to assess the overall model performance. The way we do this is to compare the similarity between the predicted estimates of the outcome variable

produced by the model and the true outcome variable values.

Machine learning (ML) is an optimization problem that tries to minimize the distance between our predicted outcome $\hat{Y} = f(X)$ and actual outcome Y using our features (or predictor variables) X as input to a function f that we want to estimate.

As our goal in this section is to assess overall model performance, we will now talk about different distance metrics that you can use.

First, let's pull out our predicted outcome values $\hat{Y} = f(X)$ from the models we fit (using different approaches).

```
wf_fit <- PM_wfflow_fit %>%
  pull_workflow_fit()
Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.
Please use `extract_fit_parsnip()` instead.

wf_fitted_values <- wf_fit$fit$fitted.values
head(wf_fitted_values)
  1      2      3      4      5      6
12.186782 9.139406 12.646119 10.377628 11.909934 9.520860
```

Alternatively, we can get the fitted values using the `augment()` function of the `broom` package using the output from `workflows`:

```
wf_fitted_values <-
  broom::augment(wf_fit$fit, data = preproc_train) %>%
  select(value, .fitted:.std.resid)

head(wf_fitted_values)
# A tibble: 6 × 6
  value .fitted   .hat .sigma   .cooksdi .std.resid
  <dbl>    <dbl> <dbl>   <dbl>      <dbl>       <dbl>
1 11.7     12.2  0.0370   2.05  0.0000648    -0.243
2 6.96     9.14  0.0496   2.05  0.00179     -1.09
3 13.3     12.6  0.0484   2.05  0.000151     0.322
4 10.7     10.4  0.0502   2.05  0.0000504     0.183
5 14.5     11.9  0.0243   2.05  0.00113      1.26
6 12.2     9.52  0.476   2.04  0.0850      1.81
```

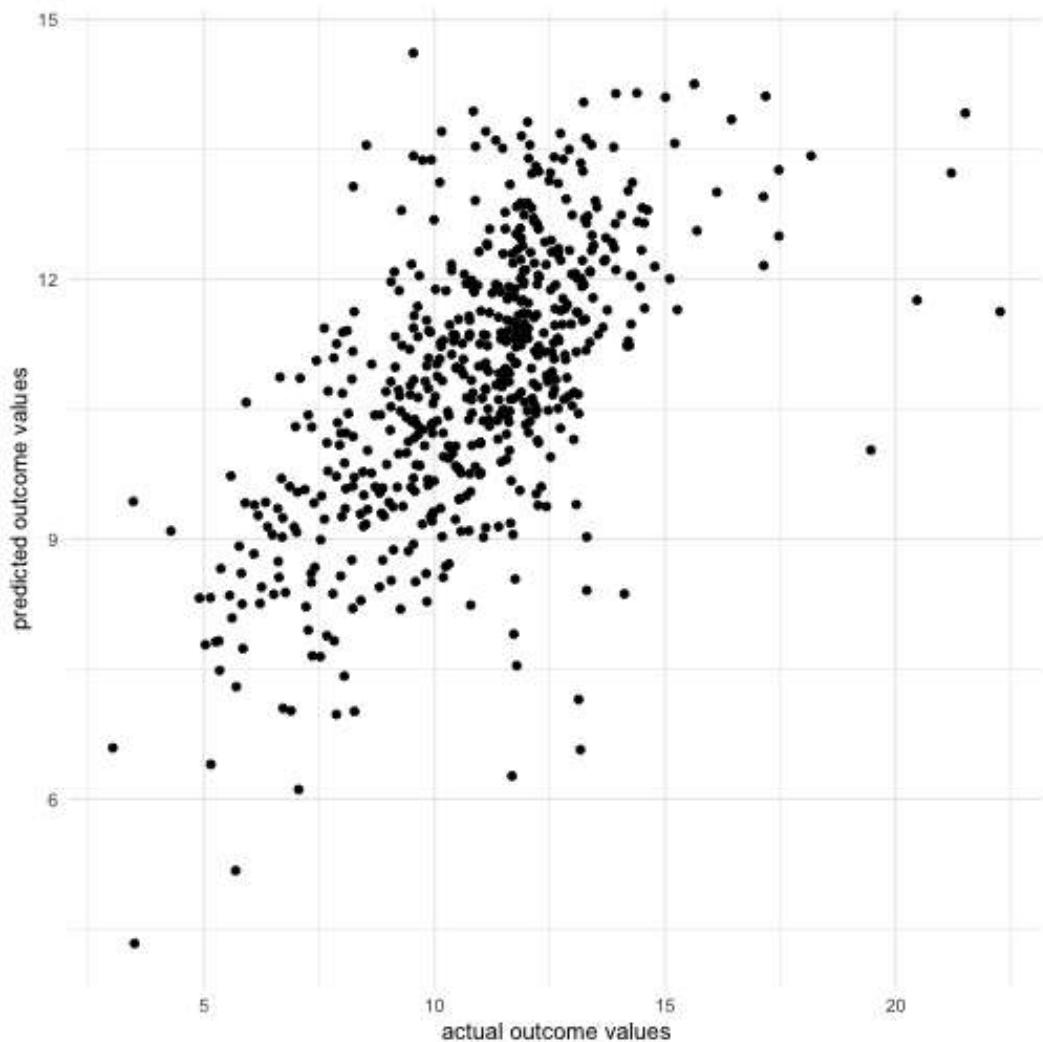
Finally, we can also use the `predict()` function. Note that because we use the actual workflow here, we can (and actually need to) use the raw data instead of the preprocessed data.

```
values_pred_train <-  
  predict(PM_wflow_fit, train_pm) %>%  
  bind_cols(train_pm %>% select(value, fips, county, id))  
Warning in predict.lm(object = object$fit, newdata = new_data, type =  
"response"): prediction from a rank-deficient fit may be misleading  
  
values_pred_train  
# A tibble: 584 × 5  
  .pred value fips   county      id  
  <dbl> <dbl> <fct> <chr>  
1 12.2  11.7  18003 Allen    18003.0004  
2 9.14   6.96  55041 Forest   55041.0007  
3 12.6   13.3  6065  Riverside 6065.1003  
4 10.4   10.7  39009 Athens   39009.0003  
5 11.9   14.5  39061 Hamilton 39061.8001  
6 9.52   12.2  24510 Baltimore (City) 24510.0006  
7 12.6   11.2  6061  Placer   6061.0006  
8 10.3   6.98  6065  Riverside 6065.5001  
9 8.74   6.61  44003 Kent    44003.0002  
10 10.0   11.6  37111 McDowell 37111.0004  
# ... with 574 more rows
```

Visualizing Model Performance

Now, we can compare the predicted outcome values (or fitted values) \hat{Y} to the actual outcome values Y that we observed:

```
library(ggplot2)
wf_fitted_values %>%
  ggplot(aes(x = value, y = .fitted)) +
  geom_point() +
  xlab("actual outcome values") +
  ylab("predicted outcome values")
```



plot of chunk unnamed-chunk-116

OK, so our range of the predicted outcome values appears to be smaller than the real values.

We could probably do a bit better.

Quantifying Model Performance

Next, let's use different distance functions $d(\cdot)$ to assess how far off our predicted outcome $\hat{Y} = f(X)$ and actual outcome Y values are from each other:

$$d(Y - \hat{Y})$$

There are entire scholarly fields of research dedicated to identifying different distance metrics $d(\cdot)$ for machine learning applications. However, we will focus on [root mean squared error \(rmse\)](#)

$$RMSE = \sqrt{\frac{\sum_{i=1}^n (\hat{y}_t - y_t)^2}{n}}$$

One way to calculate these metrics within the `tidymodels` framework is to use the `yardstick` package using the `metrics()` function.

```
yardstick::metrics(wf_fitted_values,
                   truth = value, estimate = .fitted)
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>   <chr>       <dbl>
1 rmse    standard     1.98
2 rsq     standard     0.392
3 mae     standard     1.47
```

Alternatively if you only wanted one metric you could use the `mae()`, `rsq()`, or `rmse()` functions, respectively.

```
yardstick::rmse(wf_fitted_values,
                 truth = value, estimate = .fitted)
# A tibble: 1 × 3
  .metric .estimator .estimate
  <chr>   <chr>        <dbl>
1 rmse    standard     1.98
```

Assessing Model Performance on v -folds Using `tune`

We also intend to perform cross validation, so we will now split the training data further using the `vfold_cv()` function of the `rsample` package.

Again, because these are created at random, we need to use the base `set.seed()` function in order to obtain the same results each time we knit this document. We will create 10 folds.

```
set.seed(1234)

vfold_pm <- rsample::vfold_cv(data = train_pm, v = 10)
vfold_pm
# 10-fold cross-validation
# A tibble: 10 × 2
  splits          id
  <list>         <chr>
1 <split [525/59]> Fold01
2 <split [525/59]> Fold02
3 <split [525/59]> Fold03
4 <split [525/59]> Fold04
5 <split [526/58]> Fold05
6 <split [526/58]> Fold06
7 <split [526/58]> Fold07
8 <split [526/58]> Fold08
9 <split [526/58]> Fold09
10 <split [526/58]> Fold10
pull(vfold_pm, splits)
[[1]]
<Analysis/Assess/Total>
<525/59/584>
```

```
[[2]]  
<Analysis/Assess/Total>  
<525/59/584>
```

```
[[3]]  
<Analysis/Assess/Total>  
<525/59/584>
```

```
[[4]]  
<Analysis/Assess/Total>  
<525/59/584>
```

```
[[5]]  
<Analysis/Assess/Total>  
<526/58/584>
```

```
[[6]]  
<Analysis/Assess/Total>  
<526/58/584>
```

```
[[7]]  
<Analysis/Assess/Total>  
<526/58/584>
```

```
[[8]]  
<Analysis/Assess/Total>  
<526/58/584>
```

```
[[9]]  
<Analysis/Assess/Total>  
<526/58/584>
```

```
[[10]]  
<Analysis/Assess/Total>  
<526/58/584>
```

We can fit the model to our cross validation folds using the `fit_resamples()` function of the `tune` package, by specifying our `workflow` object and the cross validation fold object we just created. See [here](#) for more information.

```
set.seed(122)
resample_fit <- tune::fit_resamples(PM_wflow, vfold_pm)
! Fold01: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold01: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold02: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold02: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold03: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold03: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold04: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold04: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold05: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold05: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold06: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold06: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold07: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold07: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold08: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold08: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold09: preprocessor 1/1: the standard deviation is zero, The correlation mat\
rix...
! Fold09: preprocessor 1/1, model 1/1 (predictions): There are new levels in a \
fac...
! Fold10: preprocessor 1/1: the standard deviation is zero, The correlation mat\
```

```
rix...
! Fold10: preprocess 1/1, model 1/1 (predictions): There are new levels in a \
fac...
```

We can now take a look at various performance metrics based on the fit of our cross validation “resamples”.

To do this we will use the `collect_metrics()` function of the `tune` package. This will show us the mean of the accuracy estimate of the different cross validation folds.

```
resample_fit
```

```
Warning: This tuning result has notes. Example notes on model fitting include:
preprocessor 1/1, model 1/1 (predictions): There are new levels in a factor: Fo\
rest, Niagara, Spencer, Trumbull, Preble, Washtenaw, LaPorte, Spartanburg, Port\
age, Talladega, Sussex, Haywood, Iberville, Plumas, Milwaukee, Sumner, Butler, \
Sebastian, Westmoreland, Rock Island, Custer, Kanawha, Fremont, Sarpy, Wilkinso\
n, Hampden, Yakima, Mobile, Glynn, prediction from a rank-deficient fit may be \
misleading
```

```
preprocessor 1/1: the standard deviation is zero, The correlation matrix has mi\
ssing values. 331 columns were excluded from the filter.
```

```
preprocessor 1/1, model 1/1 (predictions): There are new levels in a factor: St\
ark, St. Lucie, Ashland, Morgan, Box Elder, Yolo, Clarke, Chesterfield, Woodbur\
y, New London, Santa Cruz, Forrest, Howard, Buchanan, Ohio, Pueblo, Medina, McC\
racken, Potter, Missoula, Porter, Baldwin, Contra Costa, Apache, Outagamie, Lou\
doun, Dubois, Robeson, prediction from a rank-deficient fit may be misleading
```

```
# Resampling results
```

```
# 10-fold cross-validation
```

```
# A tibble: 10 × 4
```

splits	id	.metrics	.notes
<list>	<chr>	<list>	<list>
1 <split [525/59]>	Fold01	<tibble [2 × 4]>	<tibble [2 × 1]>
2 <split [525/59]>	Fold02	<tibble [2 × 4]>	<tibble [2 × 1]>
3 <split [525/59]>	Fold03	<tibble [2 × 4]>	<tibble [2 × 1]>
4 <split [525/59]>	Fold04	<tibble [2 × 4]>	<tibble [2 × 1]>
5 <split [526/58]>	Fold05	<tibble [2 × 4]>	<tibble [2 × 1]>
6 <split [526/58]>	Fold06	<tibble [2 × 4]>	<tibble [2 × 1]>
7 <split [526/58]>	Fold07	<tibble [2 × 4]>	<tibble [2 × 1]>
8 <split [526/58]>	Fold08	<tibble [2 × 4]>	<tibble [2 × 1]>
9 <split [526/58]>	Fold09	<tibble [2 × 4]>	<tibble [2 × 1]>

```
10 <split [526/58]> Fold10 <tibble [2 × 4]> <tibble [2 × 1]>
collect_metrics(resample_fit)
# A tibble: 2 × 6
  .metric .estimator   mean     n std_err .config
  <chr>   <chr>     <dbl>  <int>   <dbl> <chr>
1 rmse    standard    2.09    10   0.123 Preprocessor1_Model1
2 rsq     standard    0.321    10   0.0357 Preprocessor1_Model1
```

In the previous section, we demonstrated how to build a machine learning model (specifically a linear regression model) to predict air pollution with the `tidymodels` framework.

In the next few section, we will demonstrate another machine learning model.

Random Forest

Now, we are going to predict our outcome variable (air pollution) using a decision tree method called **random forest**.

In the case of random forest, multiple decision trees are created - hence the name forest, and each tree is built using a random subset of the training data (with replacement) - hence the full name random forest. This random aspect helps to keep the algorithm from overfitting the data.

The mean of the predictions from each of the trees is used in the final output.

In our case, we are going to use the random forest method of the the `randomForest` package.

This package is currently not compatible with categorical variables that have more than 53 levels. See [here](#) for the documentation about when this was updated from 25 levels. Thus we will remove the `zcta` and `county` variables.

Note that the `step_novel()` function is necessary here for the `state` variable to get all cross validation folds to work, because there will be different levels included in each fold test and training sets. Thus there are new levels for some of the test sets which would otherwise result in an error.

According to the [documentation](#) for the `recipes` package:

`step_novel` creates a specification of a recipe step that will assign a previously unseen factor level to a new value.

```
RF_rec <- recipe(train_pm) %>%
  update_role(everything(), new_role = "predictor")%>%
  update_role(value, new_role = "outcome")%>%
  update_role(id, new_role = "id variable") %>%
  update_role("fips", new_role = "county id") %>%
  step_novel("state") %>%
  step_string2factor("state", "county", "city") %>%
  step_rm("county") %>%
  step_rm("zcta") %>%
  step_corr(all_numeric())%>%
  step_nzv(all_numeric())
```

The `rand_forest()` function of the `parsnip` package has three important arguments that act as an interface for the different possible engines to perform a random forest analysis:

1. `mtry` - The number of predictor variables (or features) that will be randomly sampled at each split when creating the tree models. The default number for regression analyses is the number of predictors divided by 3.
2. `min_n` - The minimum number of data points in a node that are required for the node to be split further.
3. `trees` - the number of trees in the ensemble

We will start by trying an `mtry` value of 10 and a `min_n` value of 4.

Now that we have our recipe (`RF_rec`), let's specify the model with `rand_forest()` from `parsnip`.

```
PMtree_model <-
  parsnip::rand_forest(mtry = 10, min_n = 4)
PMtree_model
Random Forest Model Specification (unknown)
```

Main Arguments:

```
  mtry = 10
  min_n = 4
```

Computational engine: ranger

Next, we set the engine and mode:

Note that you could also use the `ranger` or `spark` packages instead of `randomForest`. If you were to use the `ranger` package to implement the random forest analysis you would need to specify an `importance` argument to be able to evaluate predictor importance. The options are `impurity` or `permutation`.

These other packages have different advantages and disadvantages- for example `ranger` and `spark` are not as limiting for the number of categories for categorical variables. For more information see their documentation: [here](#) for `ranger`, [here](#) for `spark`, and [here](#) for `randomForest`.

See [here](#) for more documentation about implementing these engine options with `tidymodels`. Note that there are also [other](#) R packages for implementing random forest algorithms, but these three packages (`ranger`, `spark`, and `randomForest`) are currently compatible with `tidymodels`.

```
library(randomForest)
randomForest 4.6-14
Type rfNews() to see new features/changes/bug fixes.
```

```
Attaching package: 'randomForest'
The following object is masked from 'package:dplyr':

```

```
combine
```

```
The following object is masked from 'package:ggplot2':

```

```
margin
RF_PM_model <-
  PMtree_model %>%
  set_engine("randomForest") %>%
  set_mode("regression")
```

```
RF_PM_model
Random Forest Model Specification (regression)
```

Main Arguments:

```
mtry = 10
min_n = 4
```

```
Computational engine: randomForest
```

Then, we put this all together into a workflow:

```
RF_wflow <- workflows::workflow() %>%
  workflows::add_recipe(RF_rec) %>%
  workflows::add_model(RF_PM_model)

RF_wflow
== Workflow =====\\
=====

Preprocessor: Recipe
Model: rand_forest()

— Preprocessor —————\\
———
6 Recipe Steps

• step_novel()
• step_string2factor()
• step_rm()
• step_rm()
• step_corr()
• step_nzv()

— Model —————\\
———
Random Forest Model Specification (regression)

Main Arguments:
  mtry = 10
  min_n = 4

Computational engine: randomForest
```

Finally, we fit the data to the model:

```
RF_wflow_fit <- parsnip::fit(RF_wflow, data = train_pm)
```

```
RF_wflow_fit  
= Workflow [trained] =  
=====  
Preprocessor: Recipe  
Model: rand_forest()
```

```
— Preprocessor —————\n=====
```

6 Recipe Steps

- step_novel()
- step_string2factor()
- step_rm()
- step_rm()
- step_corr()
- step_nzv()

```
— Model —————\n=====
```

Call:

```
randomForest(x = maybe_data_frame(x), y = y, mtry = min_cols(~10,           x), nod\\  
esize = min_rows(~4, x))
```

Type of random forest: regression

Number of trees: 500

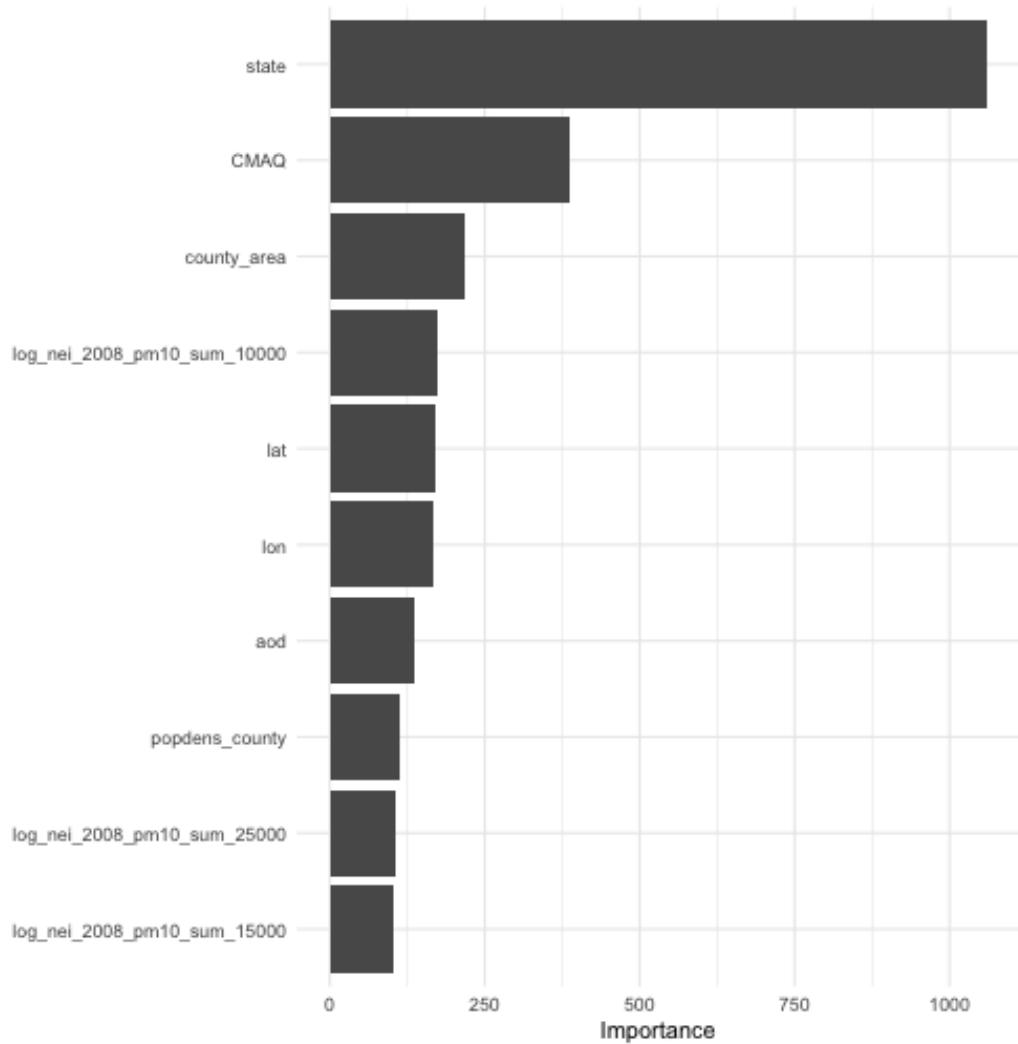
No. of variables tried at each split: 10

Mean of squared residuals: 2.698998

% Var explained: 58.28

Now, we will look at variable importance:

```
RF_wflow_fit %>%  
  pull_workflow_fit() %>%  
  vip(num_features = 10)  
Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.  
Please use `extract_fit_parsnip()` instead.
```



plot of chunk unnamed-chunk-128

Interesting! In the previous model the CMAQ values and the state where the monitor was located were also the top two most important, however predictors about education levels of the communities where the monitor was located was among the top most important. Now we see that population density and proximity to sources of emissions and roads are among the top ten.

Now let's take a look at model performance by fitting the data using cross validation:

```
set.seed(456)
resample_RF_fit <- tune::fit_resamples(RF_wflow, vfold_pm)
collect_metrics(resample_RF_fit)
```

OK, so our first model had a mean `rmse` value of 2.13. It looks like the random forest model had a much lower `rmse` value of 1.68.

If we tuned our random forest model based on the number of trees or the value for `mtry` (which is “The number of predictors that will be randomly sampled at each split when creating the tree models”), we might get a model with even better performance.

However, our cross validated mean `rmse` value of 1.68 is quite good because our range of true outcome values is much larger: (4.298, 23.161).

Model Tuning

Hyperparameters are often things that we need to specify about a model. For example, the number of predictor variables (or features) that will be randomly sampled at each split when creating the tree models called `mtry` is a hyperparameter. The default number for regression analyses is the number of predictors divided by 3. Instead of arbitrarily specifying this, we can try to determine the best option for model performance by a process called tuning.

Now let’s try some tuning.

Let’s take a closer look at the `mtry` and `min_n` hyperparameters in our Random Forest model.

We aren’t exactly sure what values of `mtry` and `min_n` achieve good accuracy yet keep our model generalizable for other data.

This is when our cross validation methods become really handy because now we can test out different values for each of these hyperparameters to assess what values seem to work best for model performance on these resamples of our training set data.

Previously we specified our model like so:

```
RF_PM_model <-  
  parsnip::rand_forest(mtry = 10, min_n = 4) %>%  
  set_engine("randomForest") %>%  
  set_mode("regression")
```

```
RF_PM_model  
Random Forest Model Specification (regression)
```

Main Arguments:

```
mtry = 10  
min_n = 4
```

```
Computational engine: randomForest
```

Now instead of specifying a value for the `mtry` and `min_n` arguments, we can use the `tune()` function of the `tune` package like so: `mtry = tune()`. This indicates that these hyperparameters are to be tuned.

```
tune_RF_model <- rand_forest(mtry = tune(), min_n = tune()) %>%  
  set_engine("randomForest") %>%  
  set_mode("regression")
```

```
tune_RF_model  
Random Forest Model Specification (regression)
```

Main Arguments:

```
mtry = tune()  
min_n = tune()
```

```
Computational engine: randomForest
```

Again we will add this to a workflow, the only difference here is that we are using a different model specification with `tune_RF_model` instead of `RF_model`:

```
RF_tune_wf <- workflows::workflow() %>%  
  workflows::add_recipe(RF_rec) %>%  
  workflows::add_model(tune_RF_model)  
  
RF_tune_wf  
== Workflow ================ \\  
=====  
Preprocessor: Recipe  
Model: rand_forest()  
  
— Preprocessor ————— \\  
=====  
6 Recipe Steps  
  
• step_novel()  
• step_string2factor()  
• step_rm()  
• step_rm()  
• step_corr()  
• step_nzv()  
  
— Model ————— \\  
=====  
Random Forest Model Specification (regression)  
  
Main Arguments:  
  mtry = tune()  
  min_n = tune()  
  
Computational engine: randomForest
```

Now we can use the `tune_grid()` function of the `tune` package to evaluate different combinations of values for `mtry` and `min_n` using our cross validation samples of our training set (`vfold_pm`) to see what combination of values performs best.

To use this function we will specify the workflow using the `object` argument and the samples to use using the `resamples` argument. The `grid` argument specifies how many possible options for each argument should be attempted.

By default 10 different values will be attempted for each hyperparameter that is being tuned. We can use the `doParallel` package to allow us to fit all these models to our cross validation

samples faster. So if you were performing this on a computer with multiple cores or processors, then different models with different hyperparameter values can be fit to the cross validation samples simultaneously across different cores or processors.

You can see how many cores you have access to on your system using the `detectCores()` function in the `parallel` package.

```
library(parallel)
parallel::detectCores()
[1] 16
```

The `registerDoParallel()` function will use the number of cores specified using the `cores=` argument, or it will assign it automatically to one-half of the number of cores detected by the `parallel` package.

We need to use `set.seed()` here because the values chosen for `mtry` and `min_n` may vary if we perform this evaluation again because they are chosen semi-randomly (meaning that they are within a range of reasonable values but still random).

Note: this step will take some time.

```
doParallel::registerDoParallel(cores=2)
set.seed(123)
tune_RF_results <- tune_grid(object = RF_tune_wflow, resamples = vfold_pm, grid\
= 20)
i Creating pre-processing data to finalize unknown parameter: mtry

tune_RF_results
Warning: This tuning result has notes. Example notes on model fitting include:
preprocessor 1/1, model 15/20: 36 columns were requested but there were 35 pred\
ictors in the data. 35 will be used.
preprocessor 1/1, model 15/20: 36 columns were requested but there were 35 pred\
ictors in the data. 35 will be used.
# Tuning results
# 10-fold cross-validation
# A tibble: 10 × 4
  splits          id     .metrics      .notes
  <list>        <chr>   <list>       <list>
1 <split [525/59]> Fold01 <tibble [40 × 6]> <tibble [0 × 1]>
2 <split [525/59]> Fold02 <tibble [40 × 6]> <tibble [1 × 1]>
```

```
3 <split [525/59]> Fold03 <tibble [40 × 6]> <tibble [0 × 1]>
4 <split [525/59]> Fold04 <tibble [40 × 6]> <tibble [0 × 1]>
5 <split [526/58]> Fold05 <tibble [40 × 6]> <tibble [0 × 1]>
6 <split [526/58]> Fold06 <tibble [40 × 6]> <tibble [0 × 1]>
7 <split [526/58]> Fold07 <tibble [40 × 6]> <tibble [0 × 1]>
8 <split [526/58]> Fold08 <tibble [40 × 6]> <tibble [0 × 1]>
9 <split [526/58]> Fold09 <tibble [40 × 6]> <tibble [1 × 1]>
10 <split [526/58]> Fold10 <tibble [40 × 6]> <tibble [0 × 1]>
```

See [the tune getting started guide](#) for more information about implementing this in `tidymodels`.

If you wanted more control over this process you could specify the different possible options for `mtry` and `min_n` in the `tune_grid()` function using the `grid_*`() functions of the `dials` package to create a more specific grid.

By default the values for the hyperparameters being tuned are chosen semi-randomly (meaning that they are within a range of reasonable values but still random).

Now we can use the `collect_metrics()` function again to take a look at what happened with our cross validation tests. We can see the different values chosen for `mtry` and `min_n` and the mean `rmse` and `rsq` values across the cross validation samples.

```
tune_RF_results %>%
  collect_metrics() %>%
  head()

# A tibble: 6 × 8
  mtry min_n .metric .estimator  mean     n std_err .config
  <int> <int> <chr>   <chr>    <dbl> <int>  <dbl> <chr>
1    12    33 rmse    standard  1.64     10  0.120 Preprocessor1_Model01
2    12    33 rsq     standard  0.600    10  0.0299 Preprocessor1_Model01
3    27    35 rmse    standard  1.63     10  0.115 Preprocessor1_Model02
4    27    35 rsq     standard  0.591    10  0.0280 Preprocessor1_Model02
5    22    40 rmse    standard  1.64     10  0.117 Preprocessor1_Model03
6    22    40 rsq     standard  0.588    10  0.0289 Preprocessor1_Model03
```

We can now use the `show_best()` function as it was truly intended, to see what values for `min_n` and `mtry` resulted in the best performance.

```
show_best(tune_RF_results, metric = "rmse", n =1)
# A tibble: 1 × 8
  mtry min_n .metric .estimator  mean     n std_err .config
  <int> <int> <chr>   <chr>    <dbl> <int>  <dbl> <chr>
1     18      4 rmse    standard   1.57     10   0.111 Preprocessor1_Model17
```

There we have it... looks like an `mtry` of 17 and `min_n` of 4 had the best `rmse` value. You can verify this in the above output, but it is easier to just pull this row out using this function. We can see that the mean `rmse` value across the cross validation sets was 1.68. Before tuning it was 1.68 with a similar `std_err` so the performance was in this particular case wasn't really improved, but that will not always be the case.

Final model performance evaluation

Now that we have decided that we have reasonable performance with our training data, we can stop building our model and evaluate performance with our testing data.

Here, we will use the random forest model that we built to predict values for the monitors in the testing data and we will use the values for `mtry` and `min_n` that we just determined based on our tuning analysis to achieve the best performance.

So, first we need to specify these values in a workflow. We can use the `select_best()` function of the `tune` package to grab the values that were determined to be best for `mtry` and `min_n`.

```
tuned_RF_values<- select_best(tune_RF_results, "rmse")
tuned_RF_values
# A tibble: 1 × 3
  mtry min_n .config
  <int> <int> <chr>
1     18      4 Preprocessor1_Model17
```

Now we can finalize the model/workflow that we used for tuning with these values.

```
RF_tuned_wflow <- RF_tune_wflow %>%
  tune::finalize_workflow(tuned_RF_values)
```

With the `workflows` package, we can use the splitting information for our original data `pm_split` to fit the final model on the full training set and also on the testing data using the `last_fit()` function of the `tune` package. No preprocessing steps are required.

The results will show the performance using the testing data.

```
overallfit <- tune::last_fit(RF_tuned_wflow, pm_split)
# or
overallfit <- RF_wflow %>%
  tune::last_fit(pm_split)
```

The `overallfit` output has a lot of really useful information about the model, the data test and training split, and the predictions for the testing data.

To see the performance on the test data we can use the `collect_metrics()` function like we did before.

```
collect_metrics(overallfit)
# A tibble: 2 × 4
  .metric .estimator .estimate .config
  <chr>   <chr>       <dbl> <chr>
1 rmse    standard     1.75 Preprocessor1_Model1
2 rsq     standard     0.592 Preprocessor1_Model1
```

Awesome! We can see that our `rmse` of 1.43 is quite similar with our testing data cross validation sets. We achieved quite good performance, which suggests that we would could predict other locations with more sparse monitoring based on our predictors with reasonable accuracy.

Now if you wanted to take a look at the predicted values for the test set (the 292 rows with predictions out of the 876 original monitor values) you can use the `collect_predictions()` function of the `tune` package:

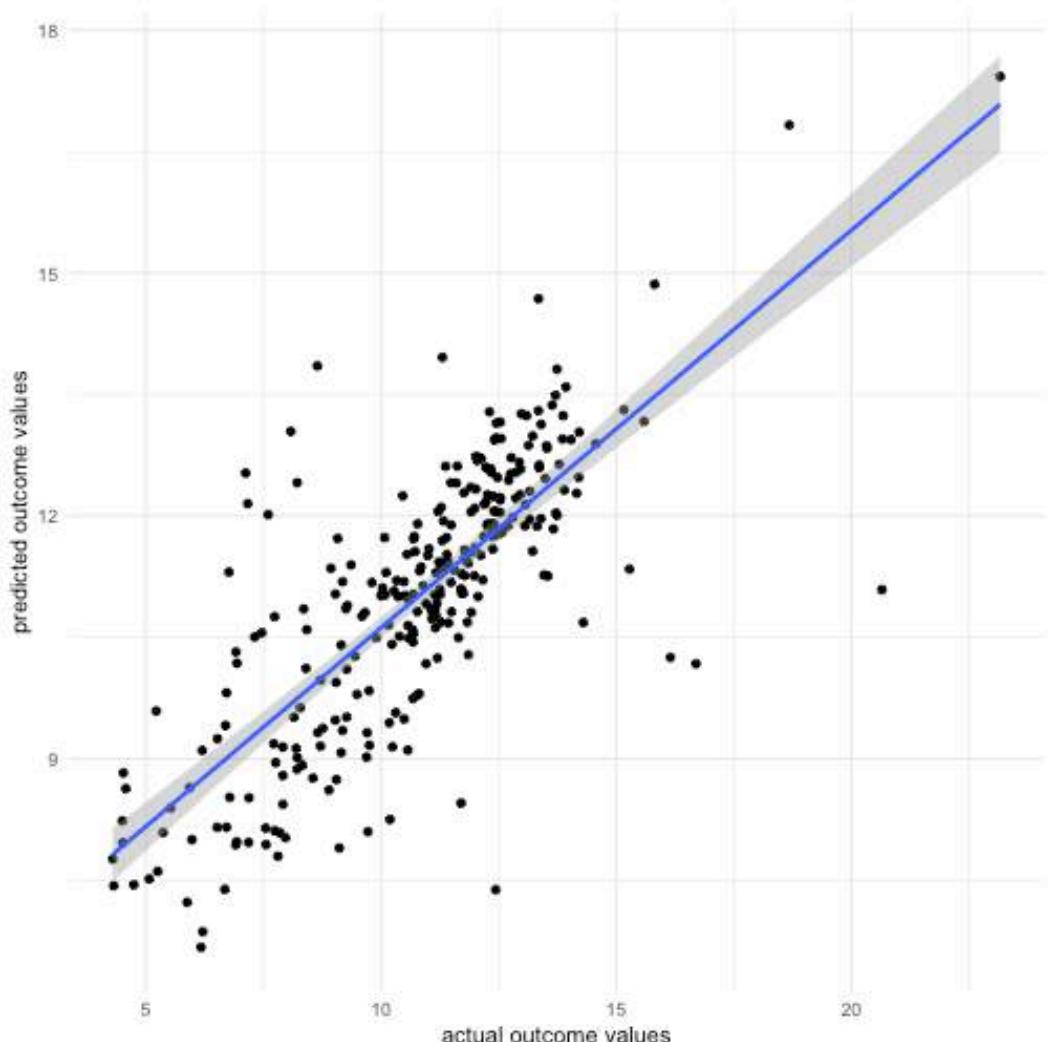
```
test_predictions <- collect_predictions(overallfit)
```

```
head(test_predictions)
# A tibble: 6 × 5
  id      .pred   .row value .config
  <chr>    <dbl> <int> <dbl> <chr>
1 train/test split 12.1     3  11.2 Preprocessor1_Model1
2 train/test split 11.6     5  12.4 Preprocessor1_Model1
3 train/test split 11.0     6  10.5 Preprocessor1_Model1
4 train/test split 13.2     7  15.6 Preprocessor1_Model1
5 train/test split 12.1     8  12.4 Preprocessor1_Model1
6 train/test split 10.8     9  11.1 Preprocessor1_Model1
```

Nice!

Now, we can compare the predicted outcome values (or fitted values) \hat{Y} to the actual outcome values Y that we observed:

```
test_predictions %>%
  ggplot(aes(x = value, y = .pred)) +
  geom_point() +
  xlab("actual outcome values") +
  ylab("predicted outcome values") +
  geom_smooth(method = "lm")
`geom_smooth()` using formula 'y ~ x'
```



plot of chunk unnamed-chunk-143

Great!

Summary of `tidymodels`

In summary, these are the minimal steps to perform a prediction analysis using `tidymodels`:

Overview of <i>tidymodels</i> Basics		
Package	Step	Functions
 rsample	1. Split into testing and training sets	<code>initial_split()</code> <code>training()</code> <code>testing()</code>
 recipes	2. Create recipe + assign variable roles	<code>recipe()</code> <code>update_role()</code>
 parsnip	3. Specify model, engine, and mode	<code>parsnip</code> function for specifying model (ex. <code>decision_tree()</code>) (https://www.tidymodels.org/find/parsnip/) <code>set_engine()</code> <code>set_mode()</code>
 workflow	4. Create workflow, add recipe, add model	<code>workflow()</code> <code>add_recipe()</code> <code>add_model()</code>
 fit	5. Fit workflow	<code>fit()</code>
 predict	6. Get predictions	<code>predict()</code>
 yardstick	7. Use predictions to get performance metrics	<code>rmse()</code> (continuous outcome) <code>accuracy()</code> (categorical outcome) <code>metrics()</code> (either type of outcome)

If you wish to perform preprocessing, cross validation, or tuning, these are the steps required:

Overview of <i>tidymodels</i>			
Use when?	Package	Step	Functions
Always		1. Split into testing and training sets	<code>initial_split()</code> <code>training()</code> <code>testing()</code>
Only if performing cross validation or tuning		Split training set into cross validation sets	Functions from <code>rsample</code> such as <code>vfold_cv()</code>
Always		2. Create recipe & assign variable roles	<code>recipe()</code> <code>update_role()</code>
Only if performing preprocessing		Specify preprocessing steps	<code>step_*</code> ()
Only if performing preprocessing and want to see the preprocessed data		Check the preprocessing	<code>prep(retain = TRUE)</code> <code>bake()</code>
Always		3. Specify model, engine, and mode	Parsnip function for specific modeling package (ex. <code>decision_tree()</code>) (https://www.tidymodels.org/find/parsnip/) <code>set_engine()</code> <code>set_mode()</code>
Only if tuning		Specify hyperparameters to tune	<code>tune()</code> within model function
Always		4. Create workflow, add recipe, add model	<code>workflow()</code> <code>add_recipe()</code> <code>add_model()</code>
Only if not using cross validation		5.1 Fit workflow	<code>fit()</code>
Only if performing cross validation		5.2 Fit workflow with cross validation	<code>fit_resamples()</code>
Only if tuning		5.3 Fit workflow with tuning	<code>tune_grid()</code>
Always		6. Get predictions	<code>predict()</code>
if not looking at resamples (cross validation, tuning)		7.1 Use predictions to get performance metrics	<code>rmse()</code> (continuous outcome) <code>accuracy()</code> (categorical outcome) <code>metrics()</code> (either type of outcome)
Only if performing cross validation or tuning		7.2 Use <code>fit_resamples()</code> or <code>tune_grid()</code> output to get performance metrics	<code>collect_metrics()</code> <code>show_best()</code>

About the Authors

Carrie Wright is a Research Associate in the Department of Biostatistics at the Johns Hopkins Bloomberg School of Public Health. She is faculty member of the Johns Hopkins Data Science Lab, where her work focuses on making data science and bioinformatics more approachable to a variety of audiences. She is also a faculty member of the [open case studies project](#) and a co-founder of the [LIBD rstats club](#), a community designed to encourage others to learn more about R programming and statistics. Carrie can be found on Twitter at [mirnas22](#) and on GitHub under the user name [carriewright11](#).

Shannon E. Ellis is an Assistant Teaching Professor of [Cognitive Science](#) at the [University of California San Deigo](#). She teaches data science to thousands of undergraduates each year and is a developer of the [Cloud-Based Data Science](#) Course Set on [Leanpub](#). Shannon can be found on Twitter at [Shannon_E_Ellis](#) and on GitHub under the user name [ShanEllis](#).

Stephanie C. Hicks is an Assistant Professor of [Biostatistics](#) at the Johns Hopkins Bloomberg School of Public Health. She is also a faculty member of the Johns Hopkins Data Science Lab, co-host of [The Corresponding Author](#) podcast and co-founder of [R-Ladies Baltimore](#). Stephanie can be found on Twitter and GitHub under the user name [stephaniehicks](#).

Roger D. Peng is a Professor of Biostatistics at the Johns Hopkins Bloomberg School of Public Health. He is also a Co-Founder of the [Johns Hopkins Data Science Specialization](#), which has enrolled over 5 million students, the [Johns Hopkins Executive Data Science Specialization](#), the [Simply Statistics](#) blog where he writes about statistics and data science for the general public, and the [Not So Standard Deviations](#) podcast. Roger can be found on Twitter and GitHub under the user name [rdpeng](#).