Introduction

Welcome to class!

1. Introductions
2. Class overview
3. Getting R up and running

[Photo by Belinda Fewings on Unsplash]

Before we start ..

Poll: How are you feeling right now?

About Us

Carrie Wright (she/her)

Senior Staff Scientist, Fred Hutchinson Cancer Center

Associate, Department of Biostatistics, JHSPH

PhD in Biomedical Sciences

Email: cwright2@fredhutch.org Web: `https://carriewright11.github.io`

About Us

Ava Hoffman (she/her)

Senior Staff Scientist, Fred Hutchinson Cancer Center

Associate, Department of Biostatistics, JHSPH

PhD in Ecology

Email: ahoffma2@fredhutch.org Web: `https://avahoffman.com`

About Us: TA

Elizabeth Humphries (she/her)

Staff Scientist, Fred Hutchinson Cancer Center

PhD in Molecular Epidemiology

Email: ehumphri@fredhutch.org

About you!

Please introduce yourself on Slack!

https://daseh.slack.com/

The Learning Curve

Learning a programming language can be very intense and sometimes overwhelming.

We recommend fully diving in and minimizing other commitments to get the most out of this course.

Like learning a spoken language, programming takes **practice**.

The Learning Curve

Learning R has been career changing for all of us, and we want to share that!

We want you to succeed – We will get through this together!

What is R?

- R is a language and environment for statistical computing and graphics developed in 1991
- R is both open source and open development

[source: http://www.r-project.org/]

Why R?

- Free (open source)
- High level language designed for statistical computing
- Powerful and flexible - especially for data wrangling and visualization
- Extensive add-on software (packages)
- Strong community

[source: https://rladies.org/]

Why not R?

- Little centralized support, relies on online community and package developers
- Annoying to update
- Slower, and more memory intensive, than the more traditional programming languages (C, Perl, Python)

[source -School vector created by nizovatina - www.freepik.com]

Introductions

What do you hope to get out of the class?

Why do you want to use R?

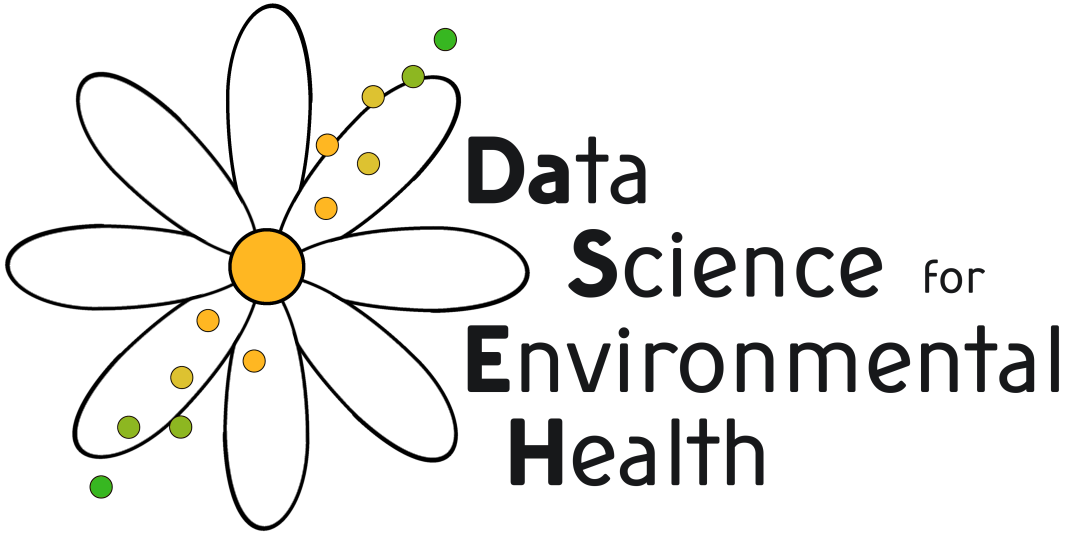
[Photo by Nick Fewings on Unsplash]

Logistics

Course Website

https://daseh.org/

Materials will be uploaded the night before class. We are constantly trying to improve content! Please refresh/download materials before class.



Learning Objectives

- Understanding basic programming syntax
- Reading data into R
- Recoding and manipulating data
- Using add-on packages (more on what this is soon!)
- Making exploratory plots
- Performing basic statistical tests
- Writing R functions
- **Building intuition**

Course Format

ONLINE VIRTUAL COURSE

- Lecture with slides, interactive
- Lab/Practical experience
- Two 10 min breaks each day - timing may vary
- June 8 - 18, 10:30am - 2:00pm PST on Zoom

Course Format

IN-PERSON CODE-A-THON

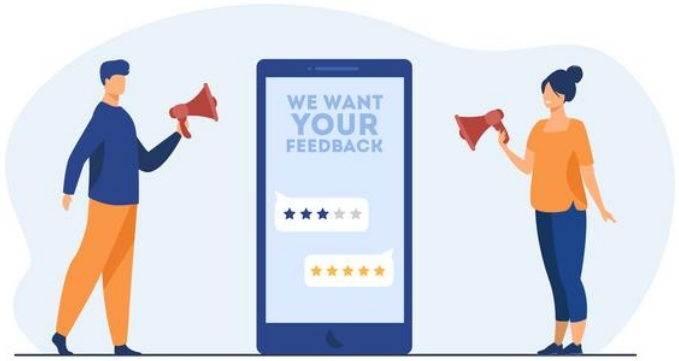
- Mostly independent group work
- Frequent check-ins with instructors and other groups
- Some lectures about the practical aspects of coding
- June 29 - July 1 (in person in Seattle)

Deadline for arranging travel assistance is June 18!

Pulse Check Survey

https://forms.gle/3AAf2WstqVsuiM8V7

Let us know anonymously how you're doing with the material.



[source - Banner vector created by pch.vector - www.freepik.com]

Homework

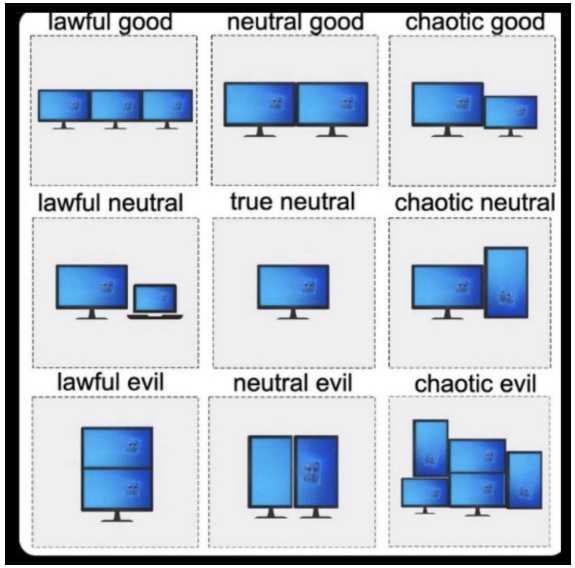
While we do have homework assignments on the course schedule, these are **strictly optional!!!**

We encourage you to try the assignments, as the best way to get comfortable with any programming language is through practice.

Your Setup

If you can, we suggest working virtually with a **large monitor or two screens**.

This setup allows you to follow along on Zoom while also doing the hands-on coding.



[source - reddit.com]

Research Survey

Research Survey

We are collecting data about user experience with our course to learn more about how to improve the data science education experience. This data may ultimately be used for a research publication and reporting to the NIH.

https://forms.gle/t8yEYJYh7BuXtnFB7

Where to find help

Useful (+ mostly Free) Resources

Found on our website under the `Resources` tab:

https://daseh.org/resources.html

- videos from previous offerings of the class
- cheatsheets for each class

Help!!!

Error messages can be scary!

- Check out the FAQ/Help page on the website: https://daseh.org/help.html
- Ask questions in Slack! Copy+pasting your error messages is really helpful!

We will also dedicate time today to debug any installation issues

Using AI

Large Language Models (ChatGPT, Claude, etc.) can be useful for programming. Specifically, they can help with:

- Writing or re-writing (refactoring) your code
- Annotation (code “notes”)
- Understanding unfamiliar code

Make sure to try coding on your own first (“programmer thinking”). Some research suggests using LLMs can cause deskilling.

We will talk more about AI + programming during the codeathon.

Installing R

- Install the latest R version (4.6.0 (called 'Because it was There') as of April 24, 2026)
- Install RStudio
- R Tools Issue

More detailed instructions on the website.

RStudio is an **integrated development environment** (IDE) that makes it easier to work with R.

More on that soon!

Summary

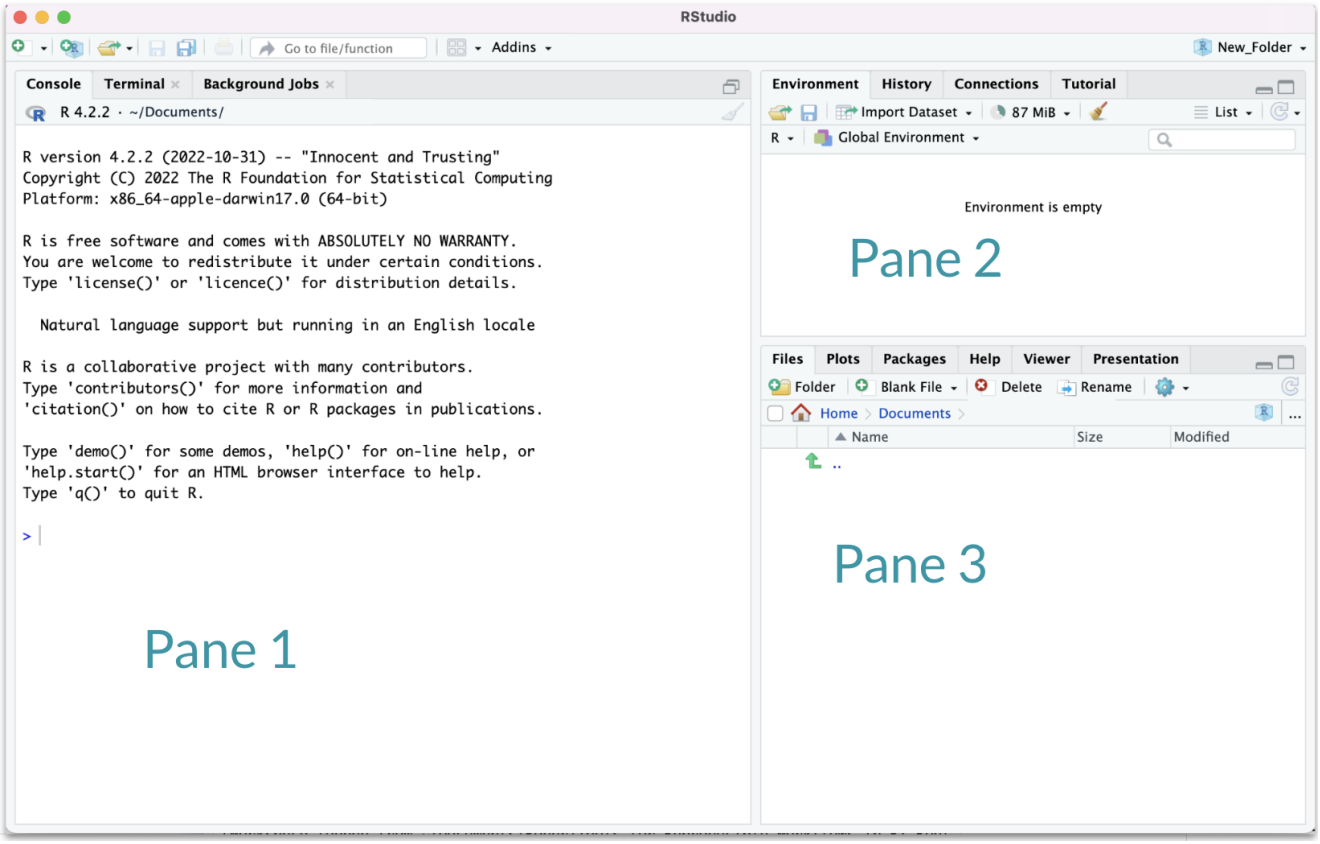
- Class Website - https://daseh.org/ - Logistics, resources, and help!
- Research Survey - https://forms.gle/t8yEYJYh7BuXtnFB7
- Pulse Check - https://forms.gle/3AAf2WstqVsuiM8V7

Image by Gerd Altmann from Pixabay

Basic R

Working in R

For now, we will be working in the **Console** (Pane 1)



FYI: Changing R versions

Check your R version.

You might change it later today if you aren't up to date with us.

Go to Tools > Global Options > General. Next to "R version", click Change, choose the newest version from the list, click Apply, and restart RStudio. (1, 2, 3, 4)

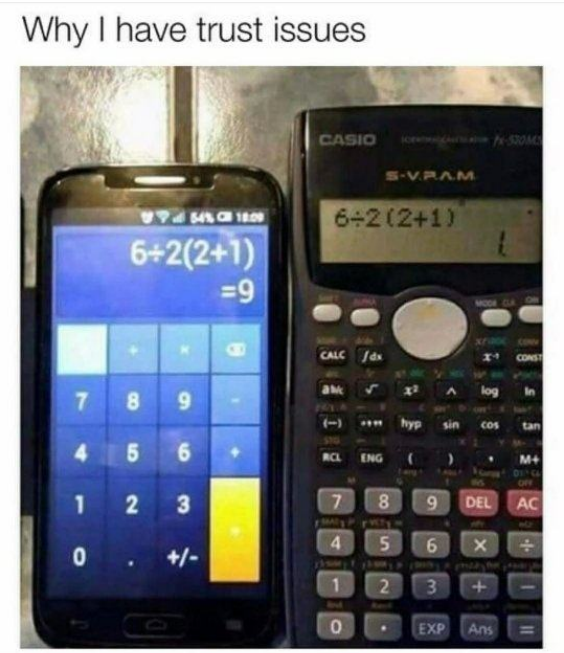
R as a calculator

- The R console is a full calculator
- Try to play around with it:
  - +, -, /, * are add, subtract, divide and multiply
  - ^ or ** is power
  - parentheses – (and) – work with order of operations
  - %% finds the remainder

R as a calculator

Try evaluating the following. Type these in the Console and press *return* to evaluate:

- `2 + 2`
- `2 * 3 / 4`
- `2^4 - 1`



Basic terms: "object"

Object - an object is something that can be worked with or on in R - can be lots of different things!

You can think of objects as **nouns** in R.

- a variable
- a dataset
- a plot

... many more

Assigning values to objects

- You can create objects within the R environment and from files on your computer
- R uses `<-` to create objects (you might also see `=` used, but this is not best practice)

```
x <- 2
x
```

```
[1] 2
```

```
x * 4
```

```
[1] 8
```

```
x + 2
```

```
[1] 4
```

GUT CHECK: What is an "object"?

A. Something I can touch

B. Something that can be worked with in R

C. A software version

Objects with text

Create objects with text using quotation marks:

```
y <- "hello world!"
y
```

```
[1] "hello world!"
```

numeric vs. character classes?

We will talk in-depth about classes. For now:

numeric

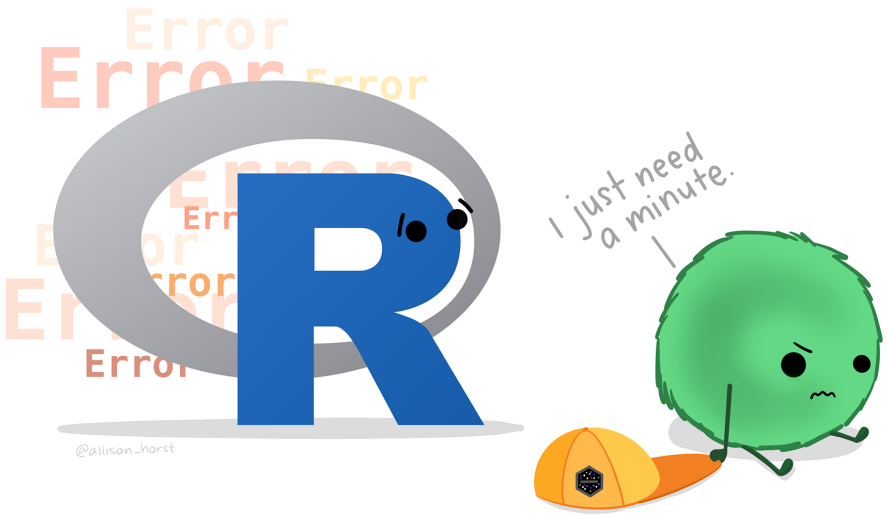
- Numbers
- No quotation marks

```
2
```

character

- Text with quotation marks
- Green lettering (default)

```
"hello!"
```



Common issues

TROUBLESHOOTING: R is case sensitive

Object names are case-sensitive, i.e., `X` and `x` are different

```
x
```

```
[1] 2
```

```
X
```

```
Error: object 'X' not found
```

TROUBLESHOOTING: No commas in big numbers

Commas separate objects in R, so they shouldn't be used when entering big numbers.

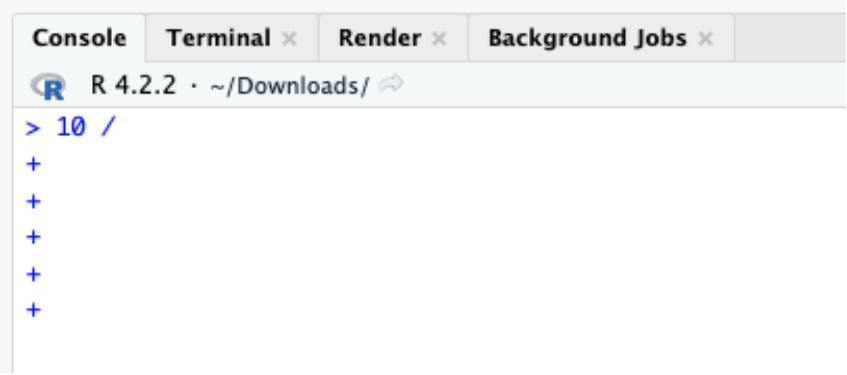
```
z <- 3,000
```

```
Error in parse(text = input): <text>:1:7: unexpected ','
1: z <- 3,
          ^
```

TROUBLESHOOTING: Complete the statement

```
10 /

Error in parse(text = input): <text>:2:0: unexpected end of input
1: 10 /
   ^
```

+ indicates an incomplete statement. Hit “esc” to clear and bring back the **>**.



TROUBLESHOOTING: Zoom Quotation Marks

```
y <- “hello world!”

Error in parse(text = input): <text>:1:6: unexpected input
1: y <- “
         ^
```

Retype the quotation marks!

```
y <- "hello world!"
```

Simple object practice

Try assigning your full name to an R object called `name`

Simple object practice

Try assigning your full name to an R object called `name`

```
name <- "Ava Hoffman"
name
```

```
[1] "Ava Hoffman"
```

Combining objects with `c()`

Use `c()` to collect/combine single R objects into a vector of R objects. It is mostly used for creating vectors of numbers and character strings.

```
x <- c(1, 4, 6, 8)
x
```

```
[1] 1 4 6 8
```

Combining objects with `c()`

Try assigning your first and last name as 2 separate character strings into a vector called `name2`

Combining objects with `c()`

Try assigning your first and last name as 2 separate character strings into a vector called `name2`

```
name2 <- c("Ava", "Hoffman")
name2
```

```
[1] "Ava"     "Hoffman"
```

Basic terms: "function"

Function - a function is a piece of code that allows you to do something in R. You can write your own, use functions that come directly from installing R, or use functions from additional packages.

You can think of a function as **verb** in R.

A function might help you add numbers together, create a plot, or organize your data.

Using functions on our vector

- `class()` tells us what kind of values the object contains (numeric, character, etc)
- `length()` tells us how many elements.

```
name
```

```
[1] "Ava Hoffman"
```

```
class(name)
```

```
[1] "character"
```

```
x
```

```
[1] 1 4 6 8
```

```
length(x)
```

```
[1] 4
```

GUT CHECK: What is a “function”?

A. a number or text

B. a button inside RStudio

C. code that does something

Combining vectors

It's fine to combine vectors, but all values will end up with the same class!

```
vect <- c(name, x)
vect
```

```
[1] "Ava Hoffman" "1"           "4"           "6"           "8"
```

```
class(vect)
```

```
[1] "character"
```

Practicing functions

What do you expect for the length of the `name2` object?

What is the class?

Practicing functions

What do you expect for the length of the `name2` object?

What is the class?

```
length(name2)
```

```
[1] 2
```

```
class(name2)
```

```
[1] "character"
```

Commenting in code

creates a comment in R code

```
# 1 + 2 <- this does not get run
```

```
1 + 2 # <- this does
```

```
[1] 3
```

Lab Part 1

- Assign values to objects with `<-` (new name on left side)
- Use the `c()` function to combine text/numbers/etc. into a vector
- `class()` tells you the class (kind) of object
- Use the `length()` function to determine number of elements
- # for comments or to deactivate a line of code

Just open up the file to see the questions for lab. More about the file type soon!

 Lab

Math + vector objects

You can perform math with vectors.

```
x + 2
```

```
[1]  3  6  8 10
```

```
x * 3
```

```
[1]  3 12 18 24
```

```
x + c(1, 2, 3, 4)
```

```
[1]  2  6  9 12
```

Math + vector objects

But math can only be performed on numbers.

```
name2 + 4
```

```
Error in name2 + 4: non-numeric argument to binary operator
```

Reassigning to a new object

Save these modified vectors as a new vector called `y`.

```
y <- x + c(1, 2, 3, 4)
y
```

```
[1]  2  6  9 12
```

Note that the R object `y` is no longer “hello world!” - It has been overwritten by assigning new data to the same name.

Reassigning to a new object

Reassigning allows you to make changes “in place”

```
# results not stored:
x + c(1, 2, 3, 4)

# x remains unchanged, results stored in `y`:
y <- x + c(1, 2, 3, 4)

# replace `x` in place
x <- x + c(1, 2, 3, 4)
```

R objects

You can get more attributes than just class. The function `str()` gives you the structure of the object.

```
str(x)
```

```
 num [1:4] 1 4 6 8
```

```
str(y)
```

```
 num [1:4] 2 6 9 12
```

This tells you that `x` is a numeric vector and tells you the length.

Basic terms: “argument”

Argument - what you pass to a function

- can be data like the number 1 or 20234
- can be options about how you want the function to work
- separated by commas

Like an **adverb**.

Create vectors with **seq()**

For numeric: `seq()`

- The `from` argument says what number to start on.
- The `to` argument says what number to not go above.
- The `by` argument says how much to increment by.
- The `length.out` argument says how long the vector should be overall.

```
seq(from = 0, to = 1, by = 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
seq(from = 0, to = 10, by = 1)
```

```
 [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
seq(from = -5, to = 5, length.out = 10)
```

```
 [1] -5.0000000 -3.8888889 -2.7777778 -1.6666667 -0.5555556  0.5555556
 [7]  1.6666667  2.7777778  3.8888889  5.0000000
```

Useful functions to create vectors `rep()`

For character: `rep()` can create very long vectors. Works for creating character and numeric vectors.

The `each` argument specifies how many of each item you want repeated. The `times` argument specifies how many times you want the vector repeated.

```
rep(WHAT_TO_REPEAT, arguments)
```

```
rep(c("black", "white"), each = 3)
```

```
[1] "black" "black" "black" "white" "white" "white"
```

```
rep(c("black", "white"), times = 3)
```

```
[1] "black" "white" "black" "white" "black" "white"
```

```
rep(c("black", "white"), each = 2, times = 2)
```

```
[1] "black" "black" "white" "white" "black" "black" "white" "white"
```

Creating numeric vectors `sample()`

You can use the `sample()` function to make a random sequence. The `x` argument specifies what you are sampling from. The `size` argument specifies how many values there should be. The `replace` argument specifies if values should be replaced or not.

```
seq_hun <- seq(from = 0, to = 100, by = 1)
seq_hun
```

```
  [1]   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
 [19]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
 [37]  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
 [55]  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
 [73]  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
 [91]  90  91  92  93  94  95  96  97  98  99 100
```

```
y <- sample(x = seq_hun, size = 5, replace = TRUE)
y
```

```
[1] 15  8 19 42 34
```

Installing packages to do more!

Some functions and data come with R right out of the box ("base R"). We will add more functionality with packages. Think of these like "expansion packs" for R.

Must be done **once** for each installation of R (e.g., version 4.2 >> 4.3).

An important package we will use is `tidyverse`. It is a mega-package great for data import, wrangling, and visualization.

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

Loading packages

After installing packages, you will need to "load" them into memory so that you can use them.

This must be done **every time** you start R.

We use a function called `library` to load packages.

```
library(tidyverse)
```

Installing + Loading packages

Images sourced from https://www.wikihow.com/Change-a-Light-Bulb

RStudio Shortcuts

	Windows	Mac
Insert assignment operator (<-)	Alt+-	Option+-

More soon! See shortcuts here.

Summary

- R functions as a calculator
- Use `<-` to save (assign) values to objects. Reassigning allows you to make changes "in place".
- Use `c()` to **combine** into vectors
- `length()`, `class()`, and `str()` tell you information about an object
- The sequence `seq()` function helps you create numeric vectors (`from`,`to`, `by`, and `length.out` arguments)
- The repeat `rep()` function helps you create vectors with the `each` and `times` arguments
- `sample()` makes random vectors
- `install.packages()` and `library()` install and load packages, respectively.

Summary

- Class Website
- Basic R Lab
- Day 1 Cheatsheet

Image by Gerd Altmann from Pixabay

RStudio

Working with R – RStudio

RStudio is an Integrated Development Environment (IDE) for R it helps you:

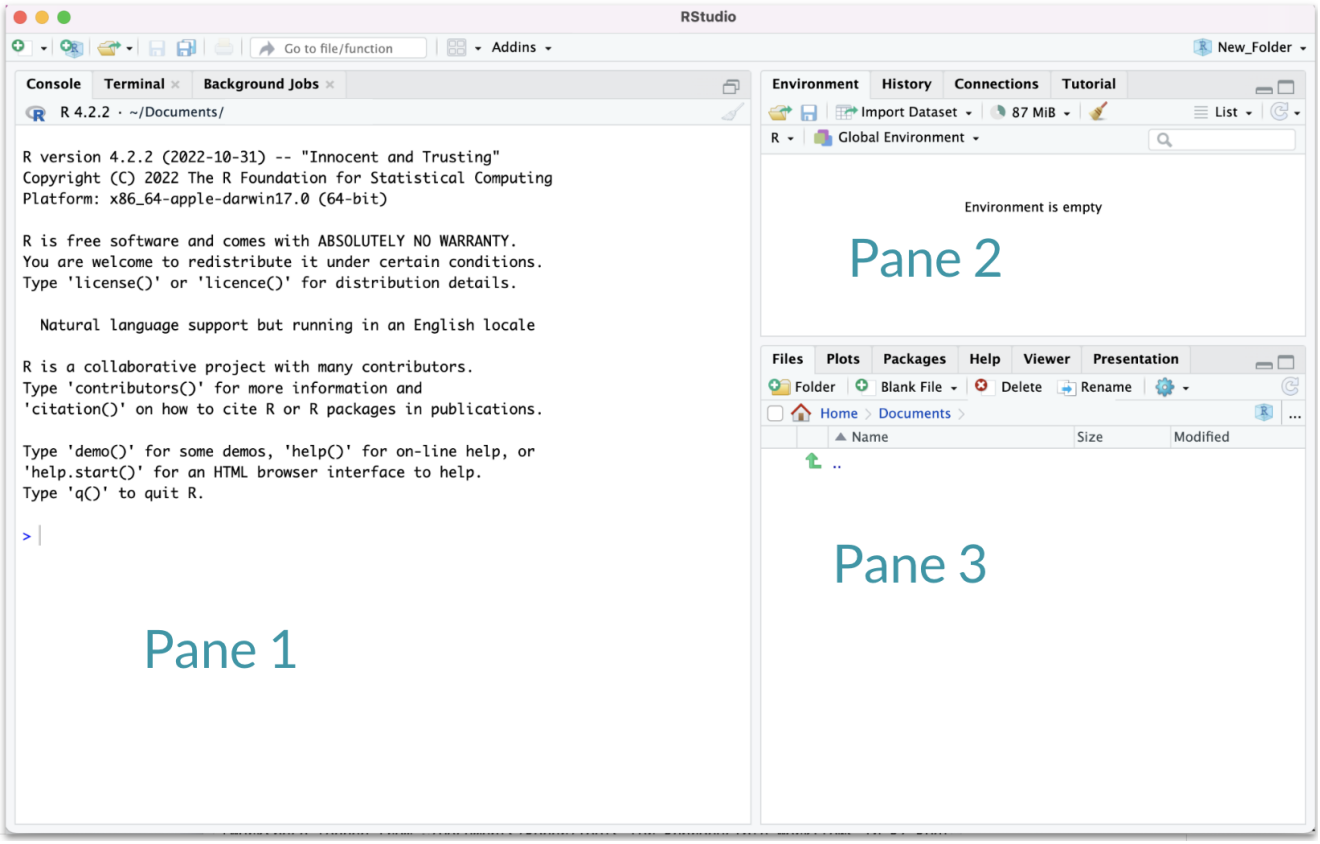
- Write code - makes suggestions
- View the output of your code, including plots
- Find errors
- Manage files
- View documentation

[source]

RStudio used to be the name of a company that is now called Posit.

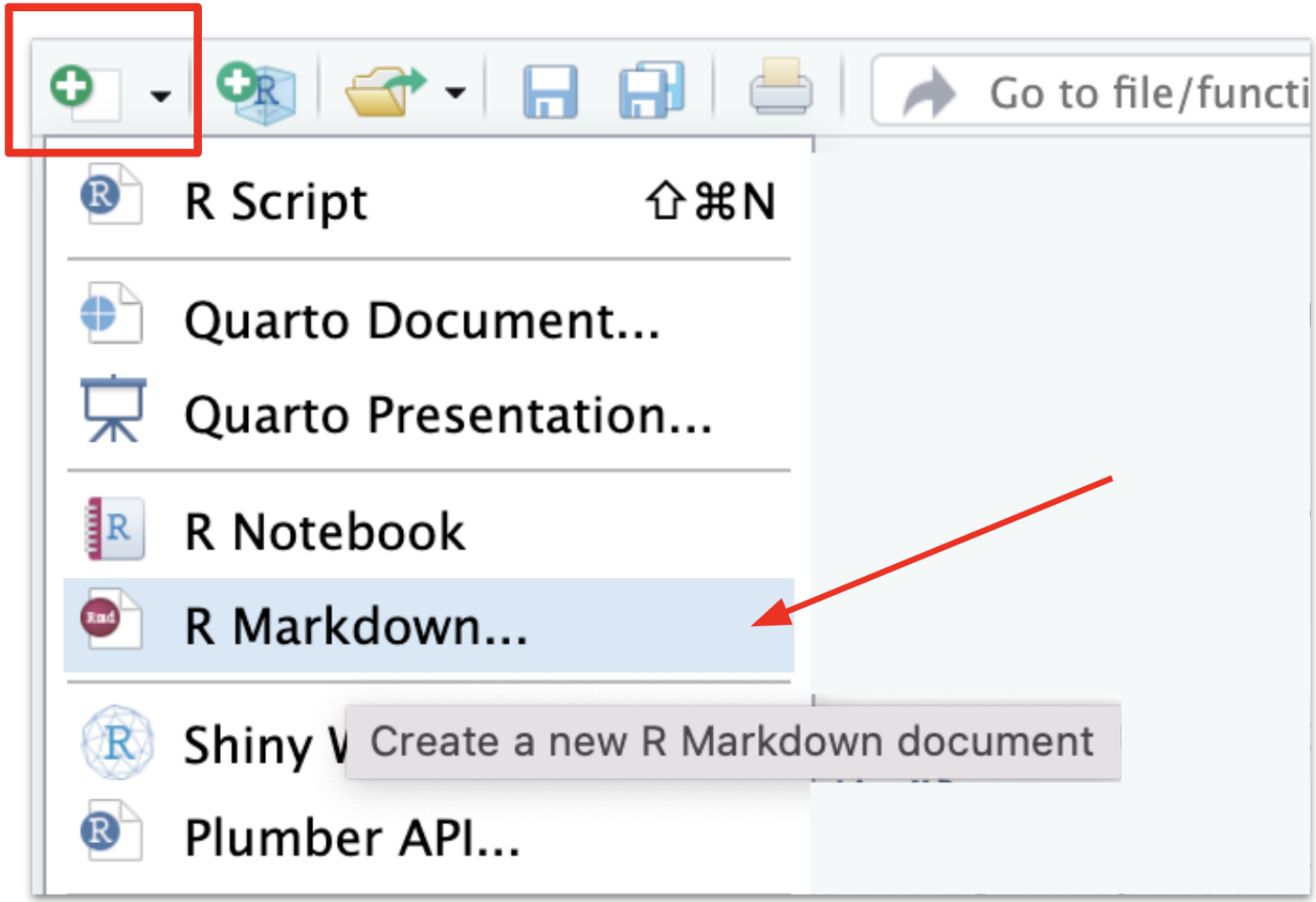
RStudio

First it is important to be familiar with the layout. When you first open RStudio, you will see 3 panes.



Hidden Pane

To save a copy of your code. You must open a file first - this will open a 4th pane. These files include Scripts or what are called R Markdown files.

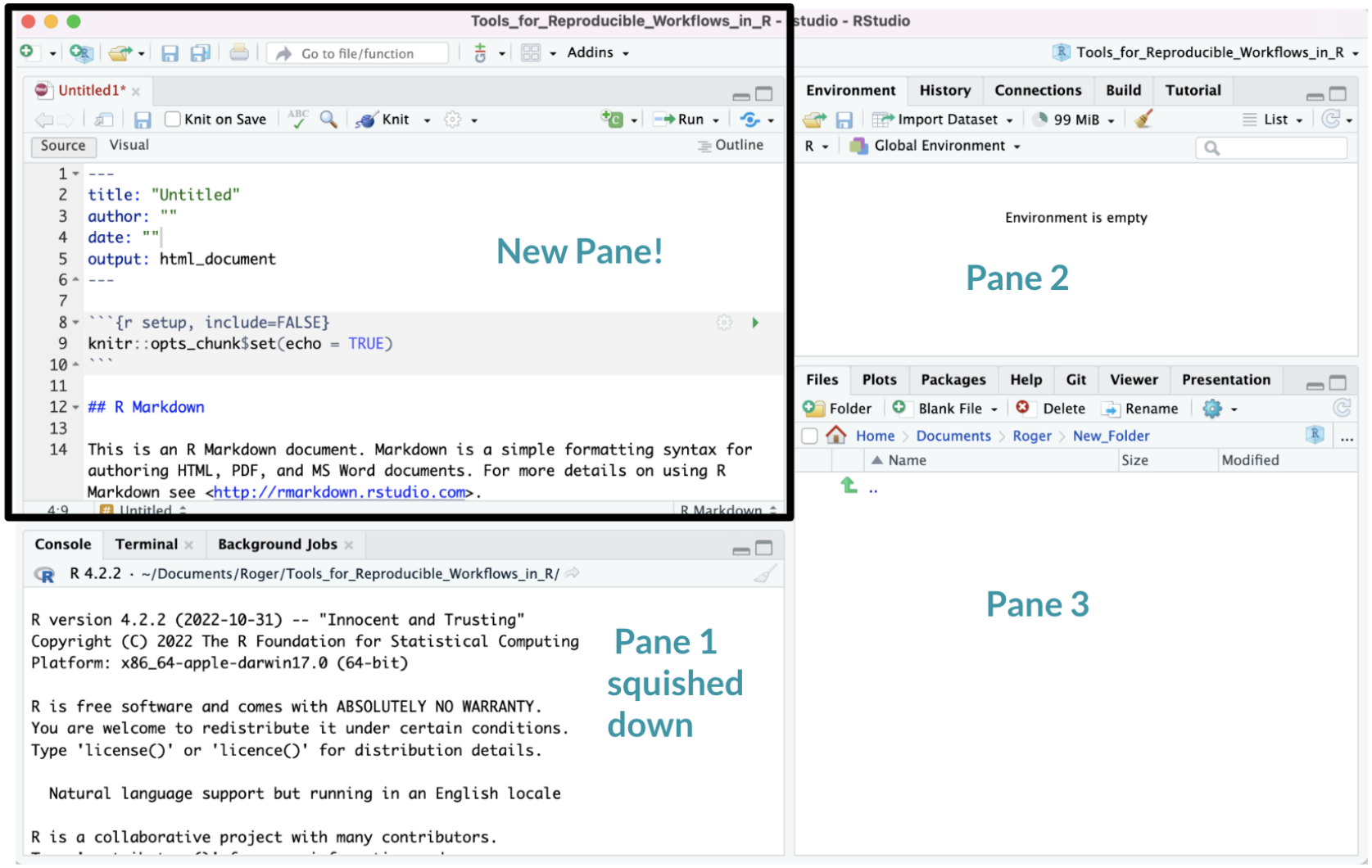


Hidden Pane

You will see a popup that you can just say “OK” to for now.

Hidden Pane

Nice! now we have a place to save code! This is where we will mostly be working.

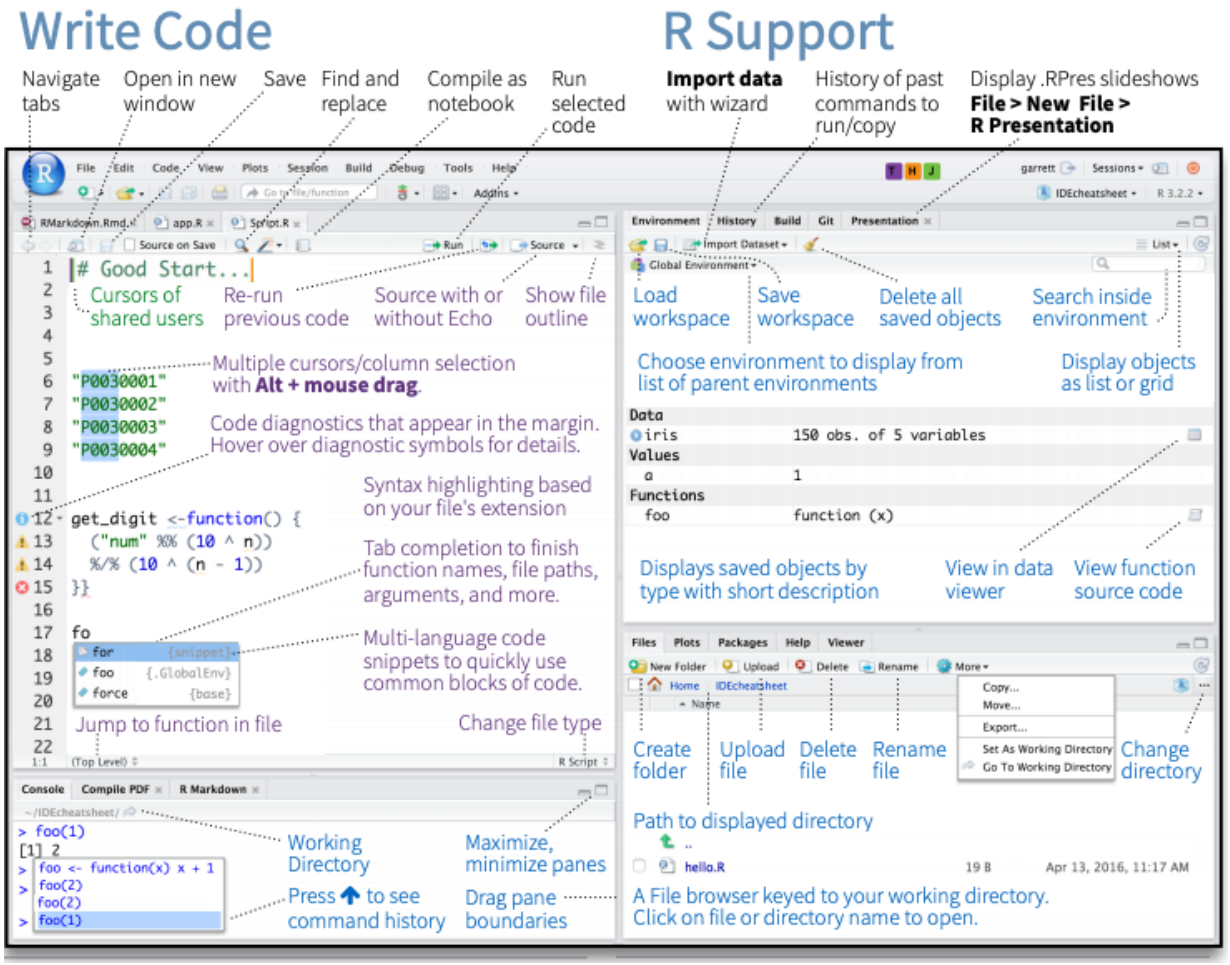


Working with R in R Studio - 2 major panes:

1. The **Source/Editor**:
   - Static copy of what you did (reproducibility)
   - Top by default
   - **Saves your code**
2. The **R Console**:
   - Try things out interactively, then add to your editor
   - Good for installing packages
   - Bottom by default
   - **Doesn't save your code**

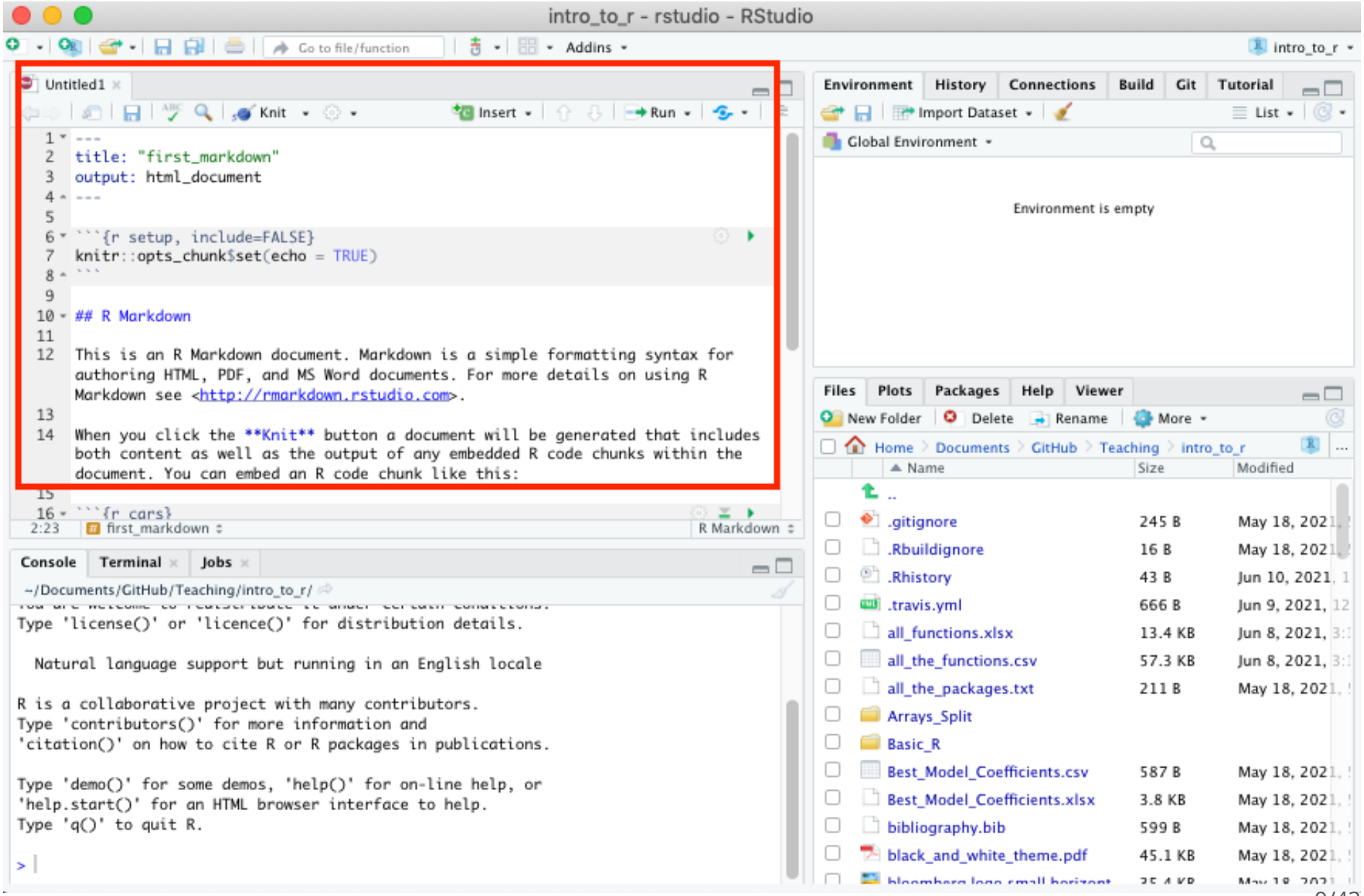
RStudio

Super useful “cheatsheet”: LINK

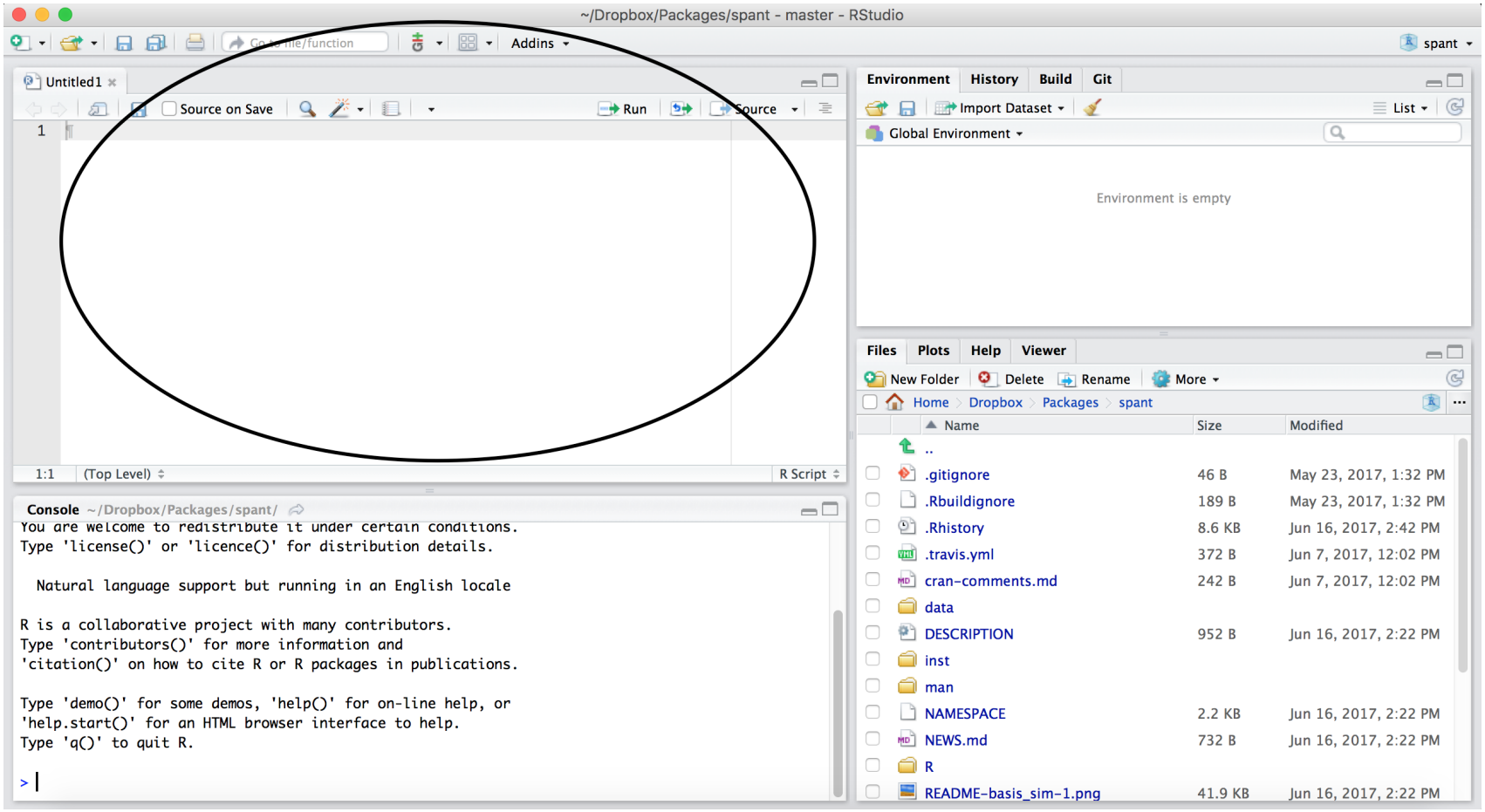


R Markdown files look different from scripts

It will look like this with text in it.



Scripts will just be empty

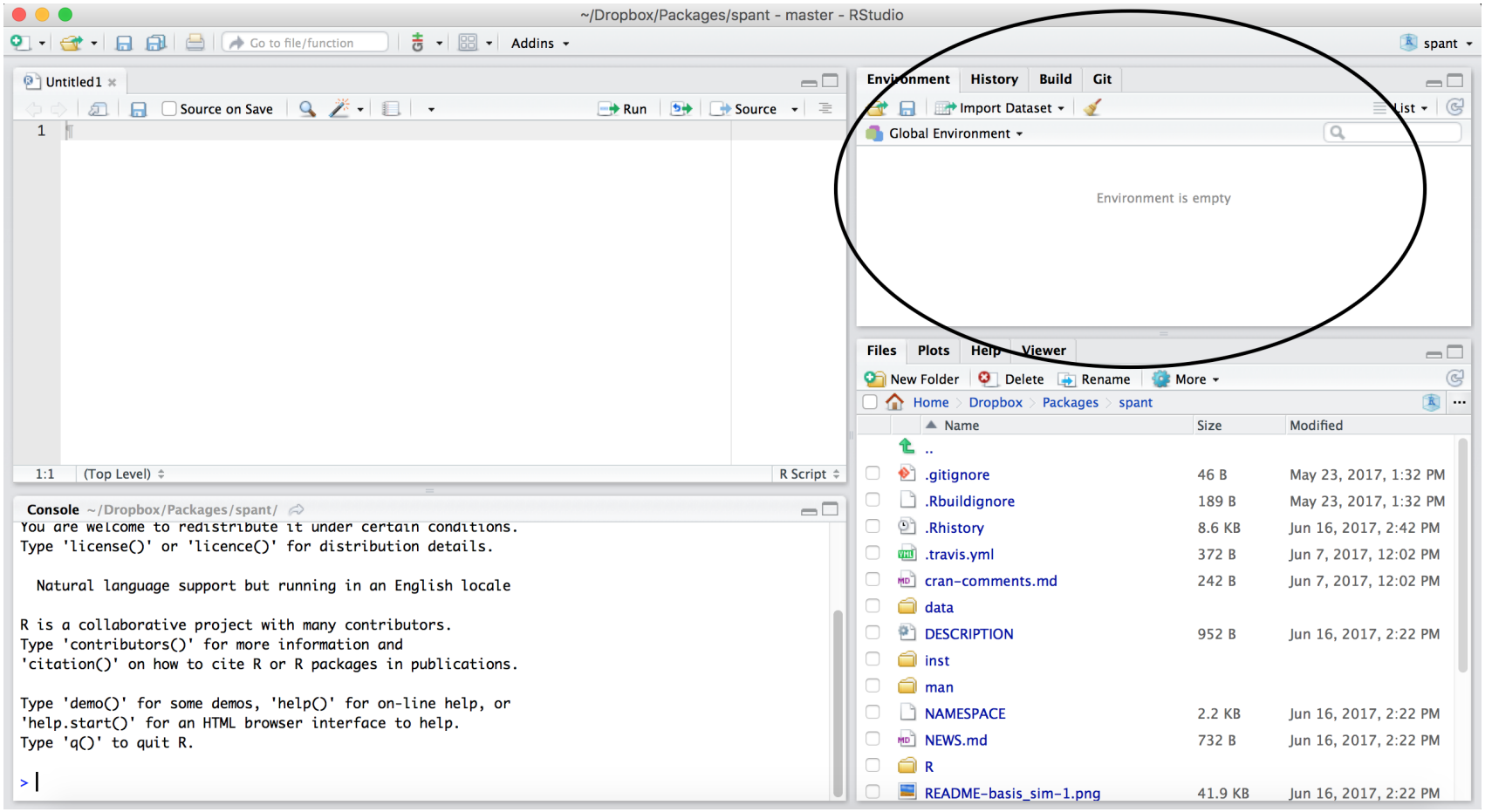


Scripts and R Markdown

Although people will use scripts often, and they are good for more programmatic purposes, we generally don't recommend them for analysis.

For data analyses, R Markdown files are generally superior because they allow you to check your code and write more info about your code.

Workspace/Environment



Workspace/Environment

- Tells you what **objects** are in R
- What exists in memory/what is loaded?/what did I read in?

History

- Shows previous commands. Good to look at for debugging, but **don't rely** on it.
  Instead use RMarkdown!
- Also type the "up" key in the Console to scroll through previous commands

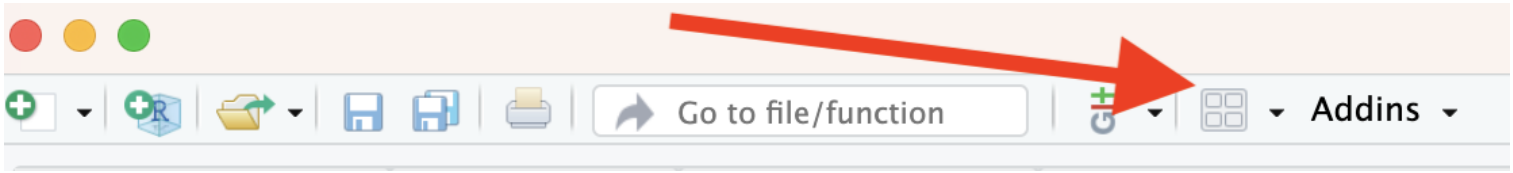
Lower right pane

- **Files** - shows the files on your computer of the directory you are working in
- **Help** - shows help of R commands
- **Plots** - pictures and figures

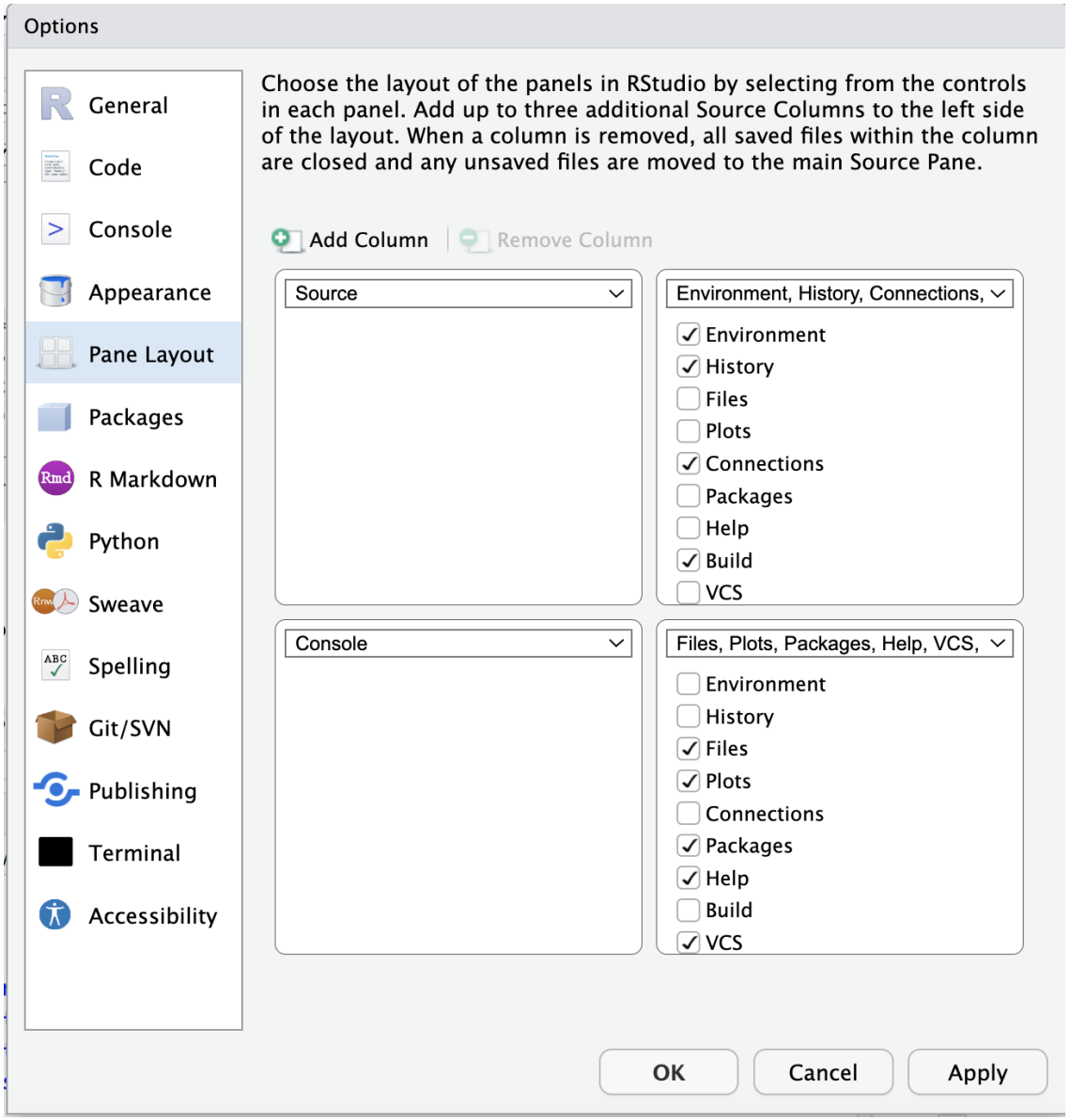
RStudio Layout

If RStudio doesn't look the way you want (or like our RStudio), then:

Click on the pane button, which looks like a waffle with 4 indentations. Scroll down to "Pane Layout".



Default Layout



Let’s take a look at R Studio ourselves!

R Markdown file

R Markdown files (.Rmd) help generate reports that include your code and output.

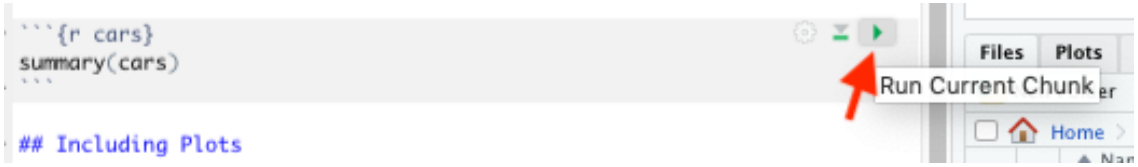
1. Helps you describe your code
2. Allows you to check the output
3. Can create many different file types

You may have also heard of quarto which does something similar.

Code chunks

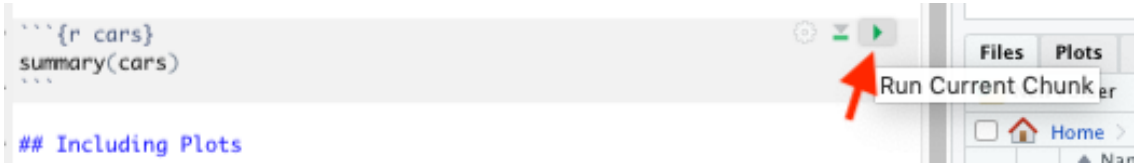
Within R Markdown files are code “chunks”.

This is where you can type R code and run it!



Run code in a chunk

Clicking the run (play) button runs the code in the chunk.



`Ctrl + Enter` on Windows or `Command + Enter` on Mac in your script evaluates that line of code

Running a chunk executes the code

- Generally see a preview of the output of the code just below the chunk
- See the code in the console

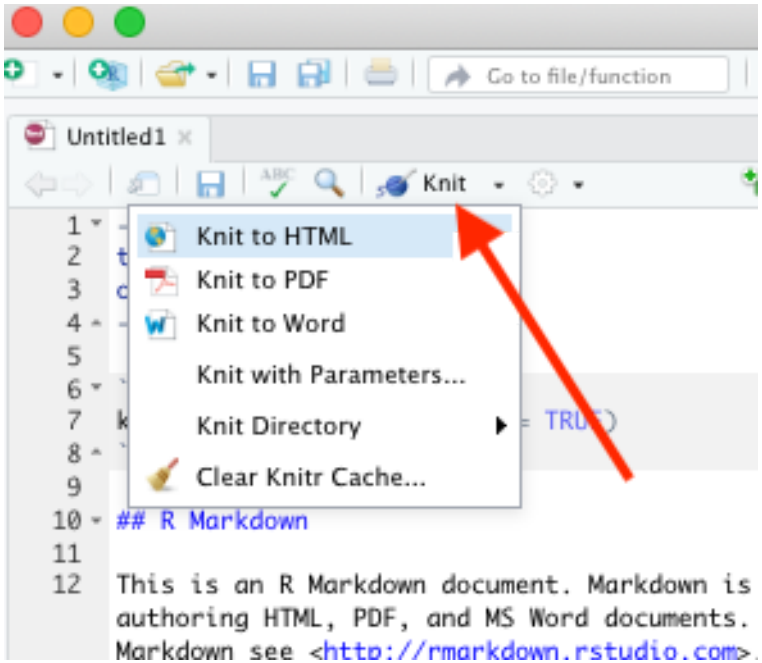
If you get annoyed by code previews in Markdown files…

See the Help page of the website. You can adjust this and change your RStudio settings:

Tools > Global Options > R Markdown

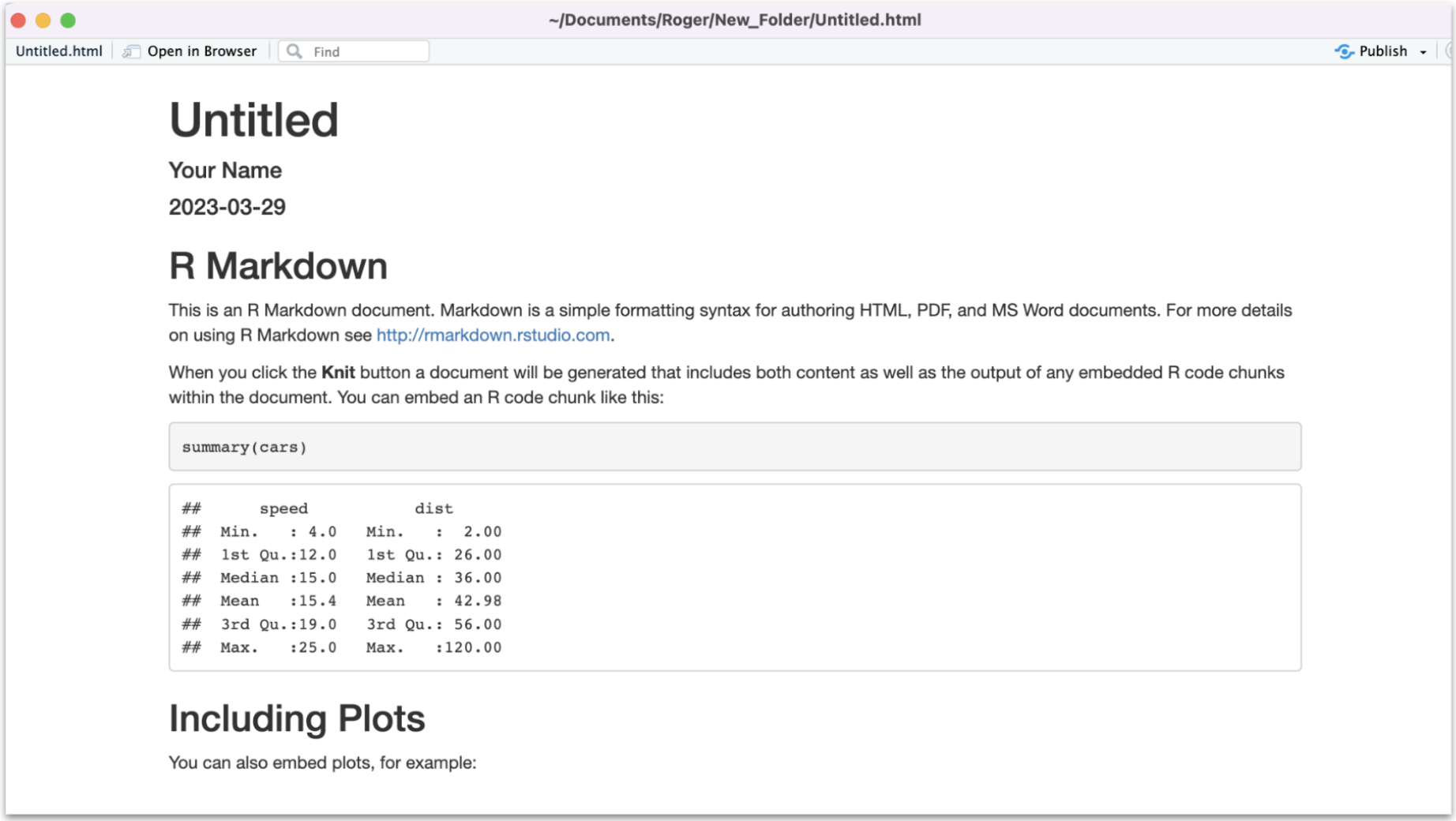
Knit file to html

Running all chunks - this will create a report from the R Markdown document! Note that it can't use anything not included in the file, it can't use objects in your environment that you were modifying interactively.



Nice report!

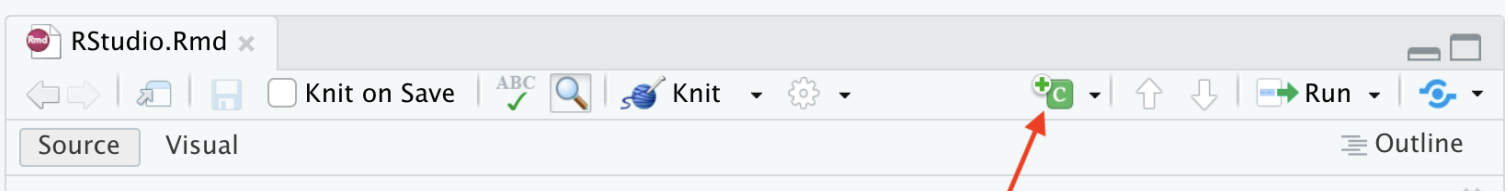
This generates a nice report that you can share with others who can open in any browser.



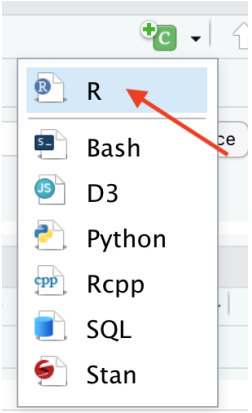
Create Chunks

To create a new R code chunk:

- Use the insert code chunk button at the top of RStudio.



- Select R (default) as the language:



Create Chunks

If you like keyboard shortcuts:

- Windows & Linux use Ctrl+Alt+I
- Mac use Command+Option+I

I is for insert.

Run previous chunks button

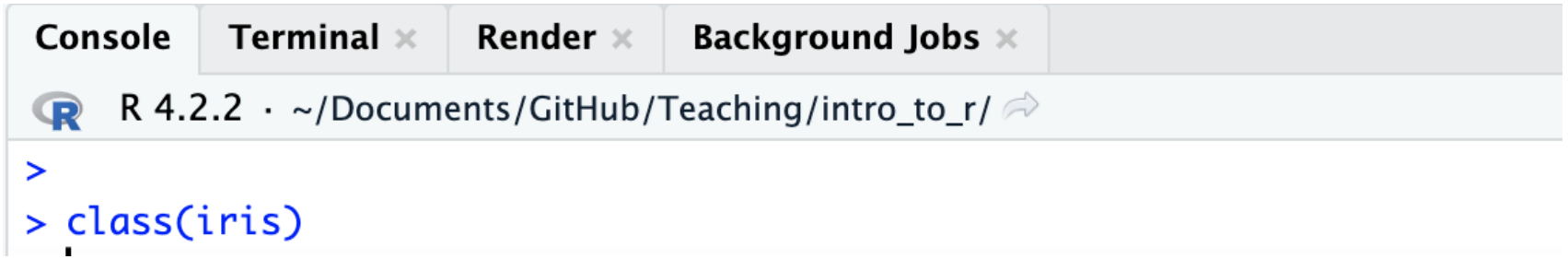
You can run all chunks above a specific chunk using this button:

Errors

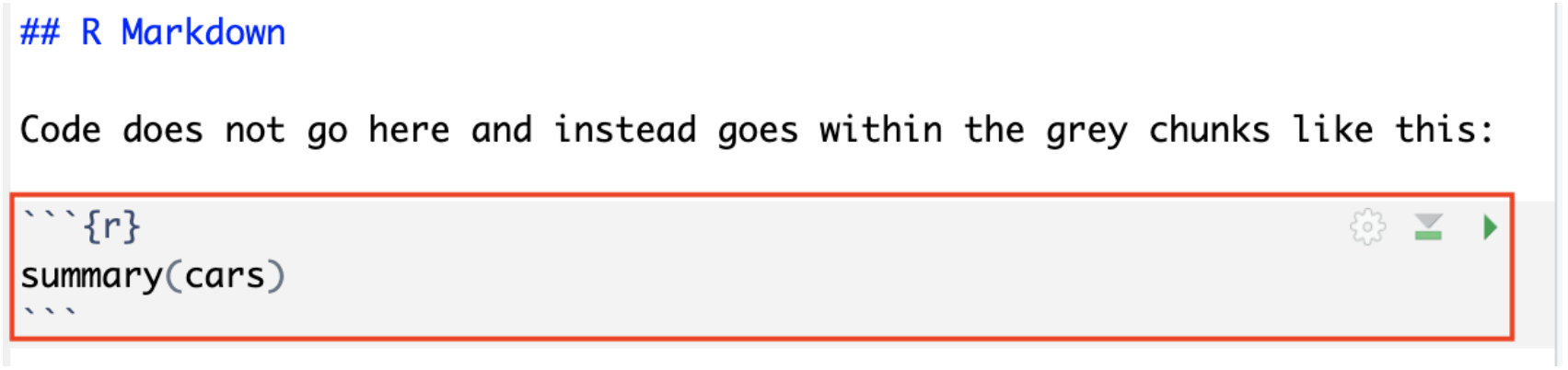
R studio can help you find issues in your code. Note that sometimes the error occurs earlier than RStudio thinks.

Recap of where code goes

- You can test code in the console

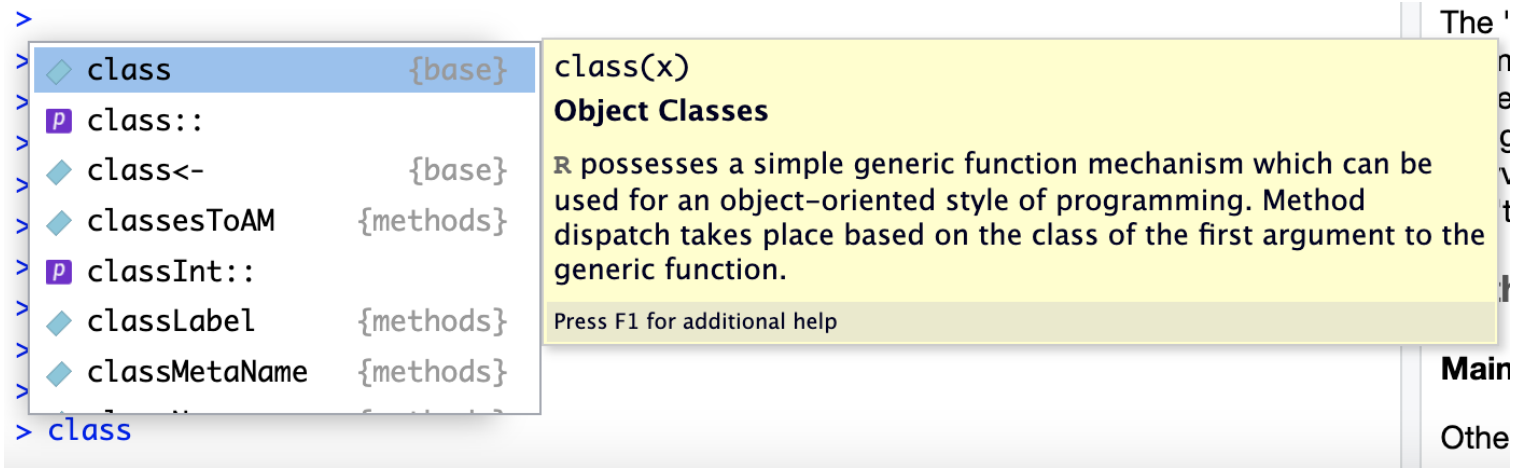


- You can save code in a chunk in the editor (Markdown file)



Getting help from the preview

When you type in a function name, a pop up will preview documentation to help you. It also helps you remember the name of the function if you don't remember all of it!



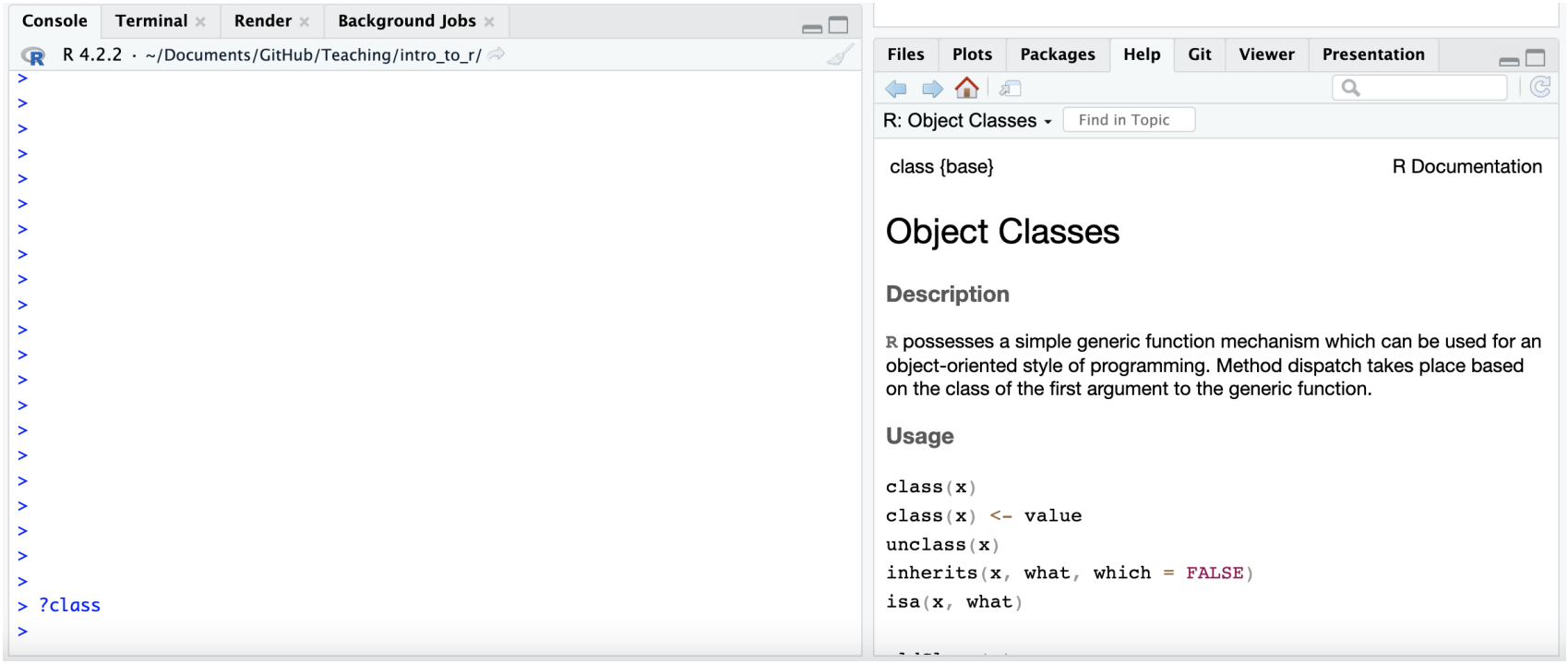
Get help with the help pane

Getting Help with ?

If you know the name of a package or function:

Type `?package_name` or `?function_name` in the console to get information about packages and functions.

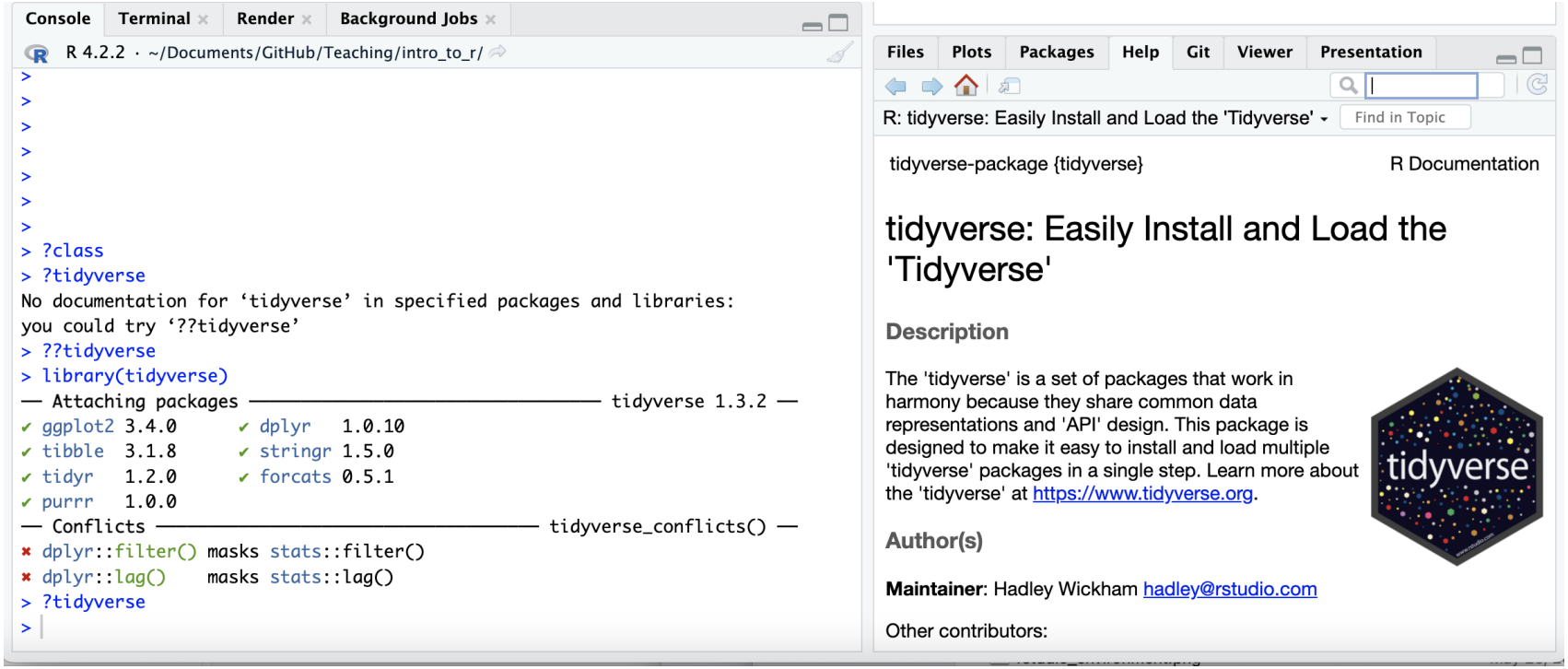
For example: `?readr` or `?read_csv`.



Double Question Mark

If you haven't loaded a package yet into R than you may get a response that there is no documentation.

Typing in `??package_name` can show you packages that you haven't loaded yet.



Gut Check

Why are R Markdown files so useful?

1. They let you test your code
2. They let you view the output of your code
3. They let you generate cool reports
4. All of the above

Gut Check

Where does code go typically in an Rmd file?

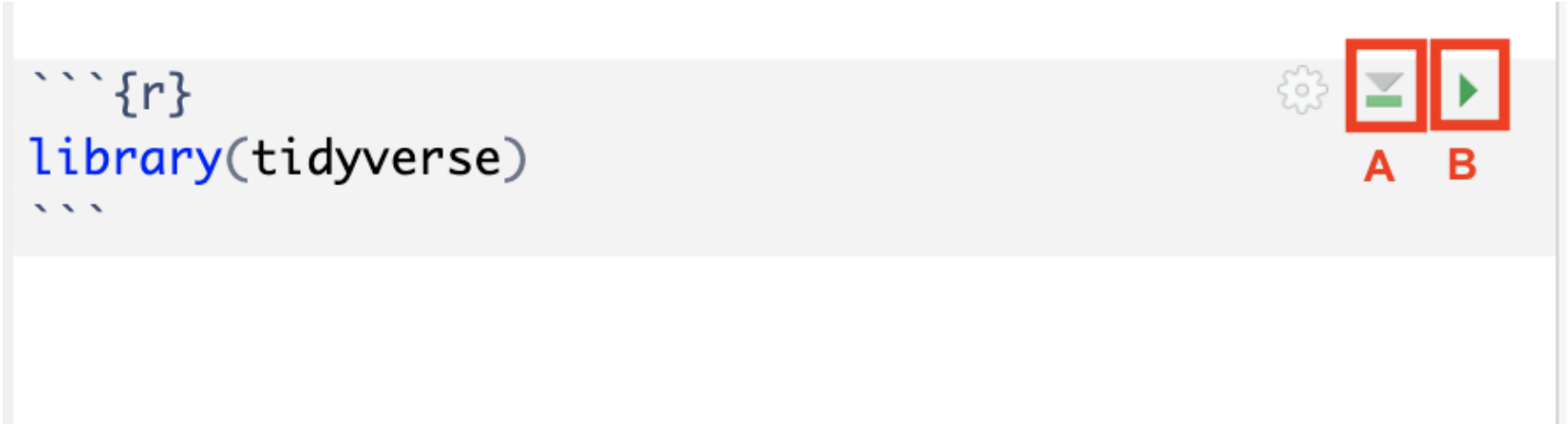
A

```{r}
B

```

C

Gut Check

Which button do you click to run the code in a current chunk?



Lab: Getting started

To do this lab we need to:

- Download the file at the on next slide by clicking on it or go to the website schedule page
- Find the downloaded file on your computer
- Open the file in RStudio (double clicking the file name typically works, otherwise right click)
- Might need to restart RStudio

These videos can help if you aren't sure where your downloads are:

If you have a PC: https://youtu.be/we6vwB7DsNU

If you have a Mac: https://www.youtube.com/watch?v=Ao9e0cDzMrE

You can find these on the resource page of the class website.

Summary

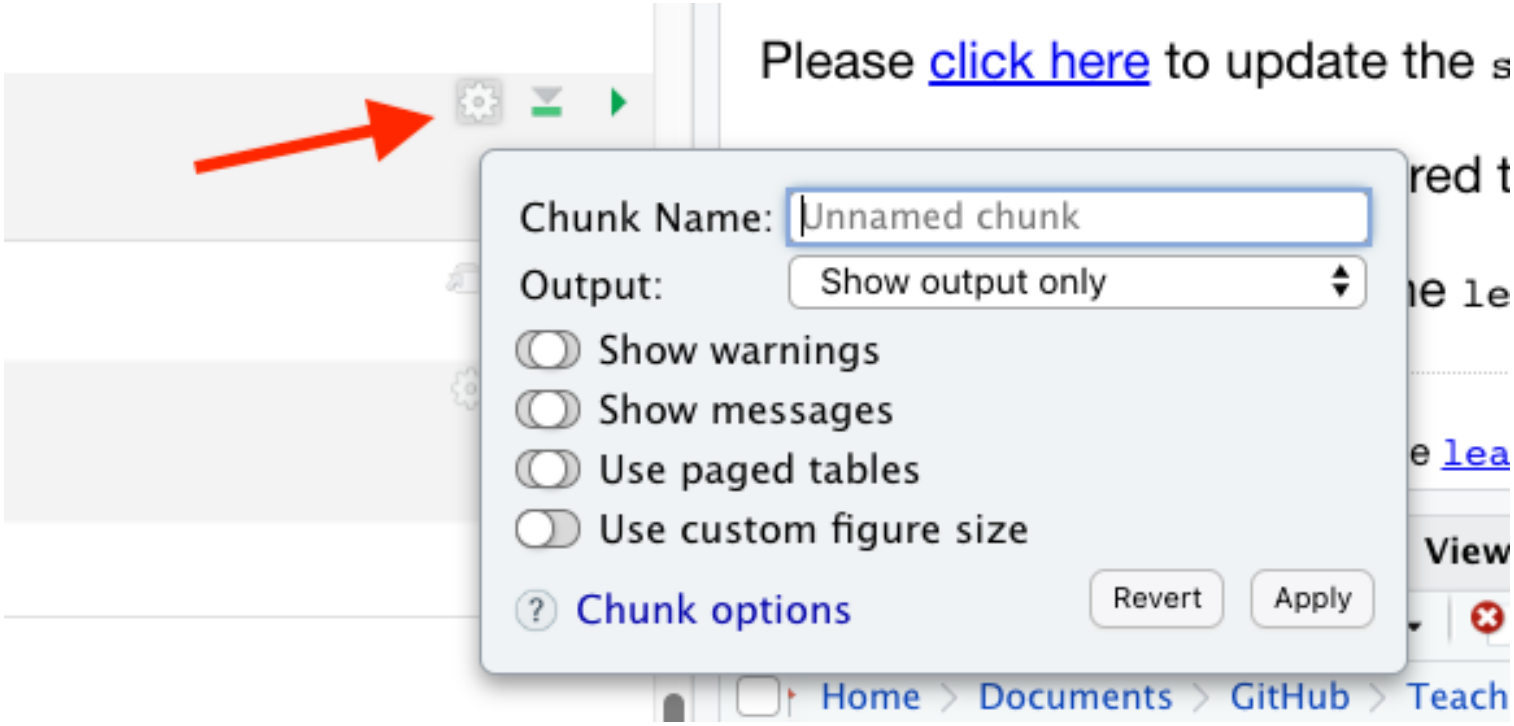
- RStudio makes working in R easier
- The Editor (top) is for static code like scripts or R Markdown documents
- The console is for testing code (bottom) - best to save your code though!
- R markdown documents are really helpful for lots of reasons!
- R code goes within what is called a chunk (the gray box with a green play button)
- Code chunks can be modified so that they show differently in reports

 Class Website. Lab. Posit Cheatsheet. Day 1 Cheatsheet.

Image by Gerd Altmann from Pixabay

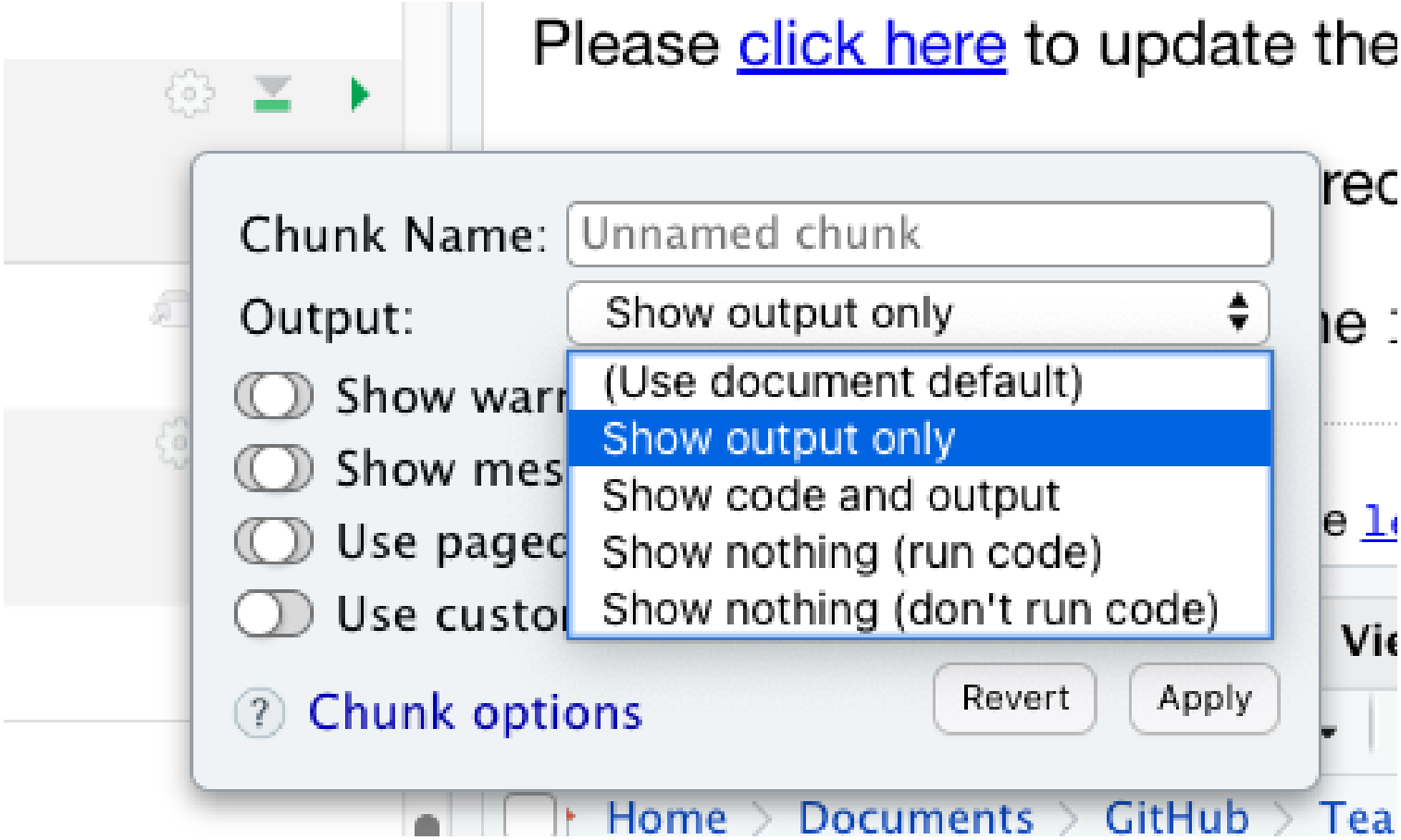
Extra Slides

Chunk settings



Chunk settings

You can specify if a chunk will be seen in the report or not.



Sometimes you want to hide your code

If you want to keep your code so people can see it if they want to there is a nice option called code folding - check it out here: https://stackoverflow.com/questions/69326576/show-output-but-hide-code-when-sending-rmd-to-other-people

Rainbow Parentheses

Tools –> Global Options –> Code –> Display –> Use rainbow parentheses

This can help you see your code more easily.

Press enter to save this setting and get out of this menu.

Reproducibility

What is Reproducibility

Photo by Simone Secci on Unsplash

Patil, Peng, Leek (2016) https://www.biorxiv.org/content/10.1101/066803v1

Content adapted from Candace Savonen.

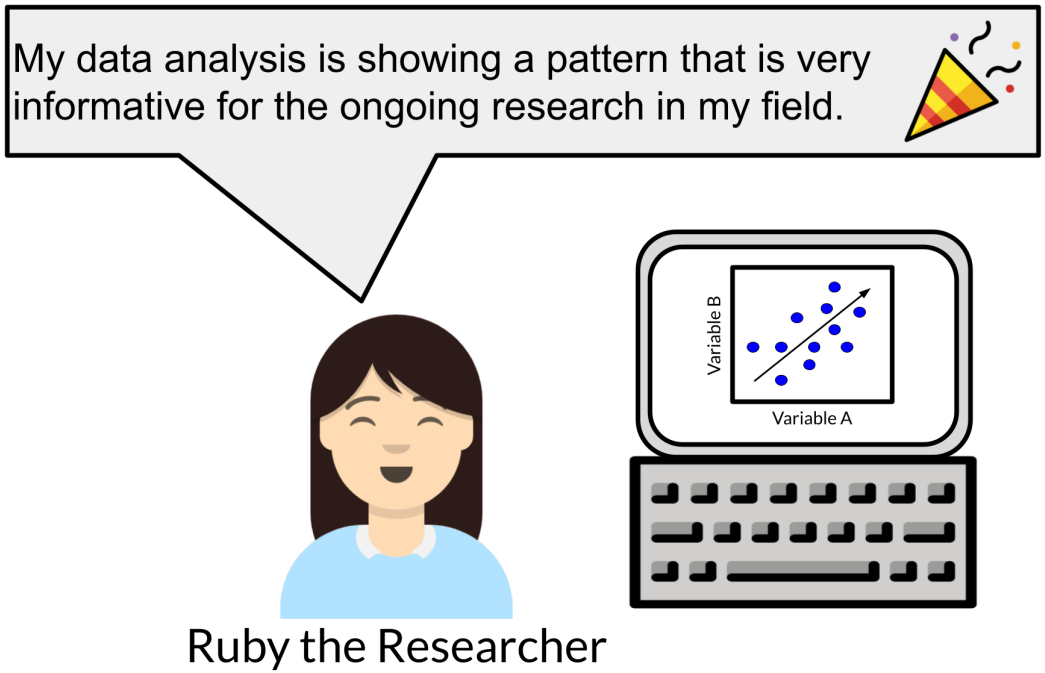
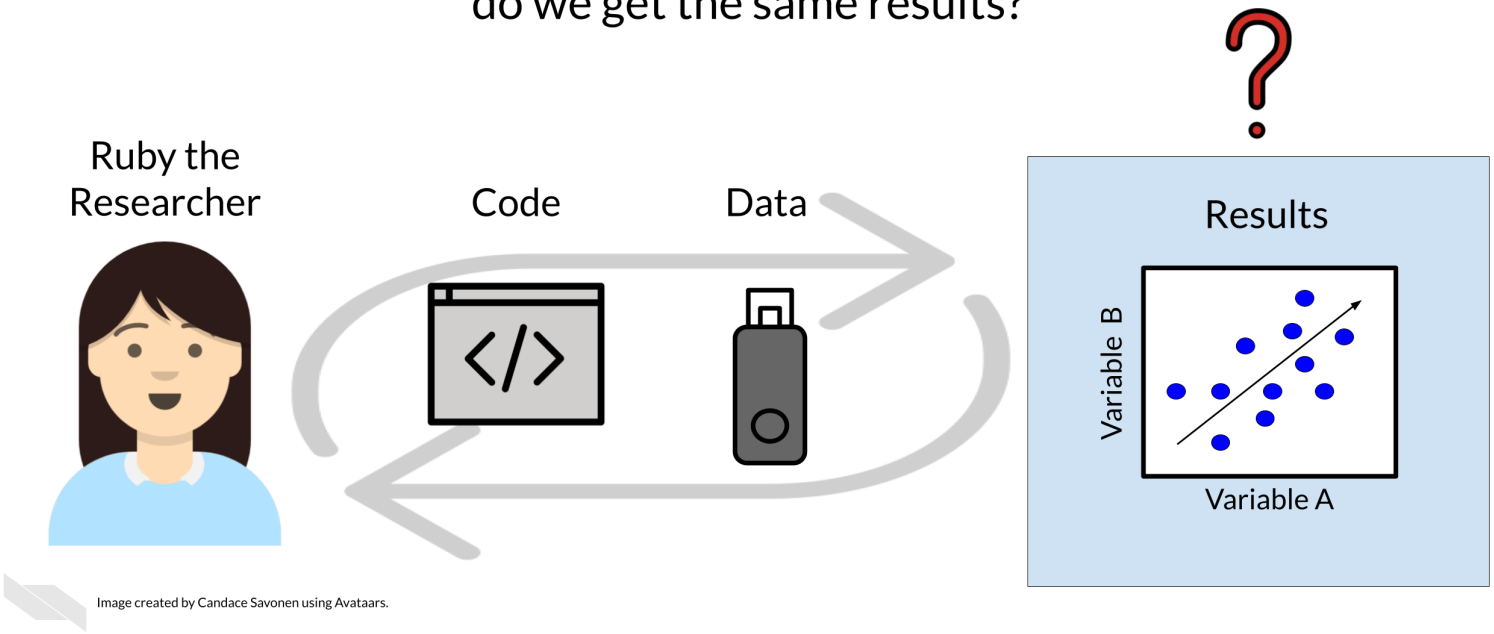


Image created by Candace Savonen using Avataars.

Repeatable: keeping everything the same but repeating the analysis - do we get the same results?



Reproducible: using the same data and analysis but in the hands of *another researcher* - do we get the same results?

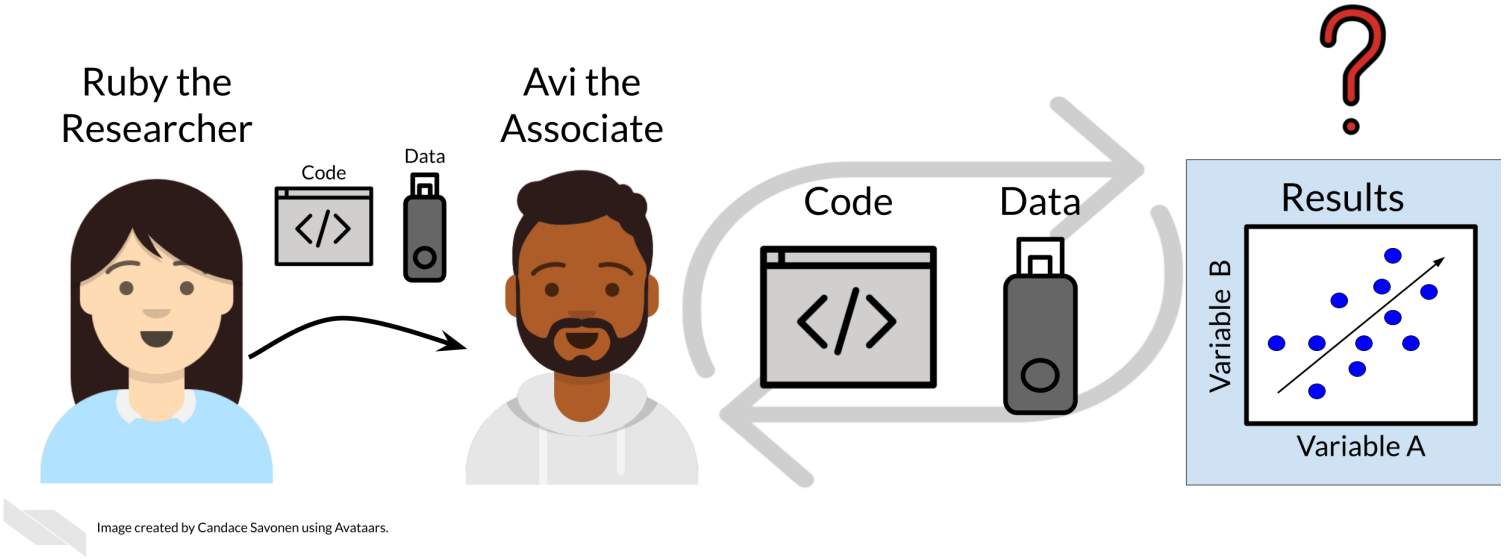
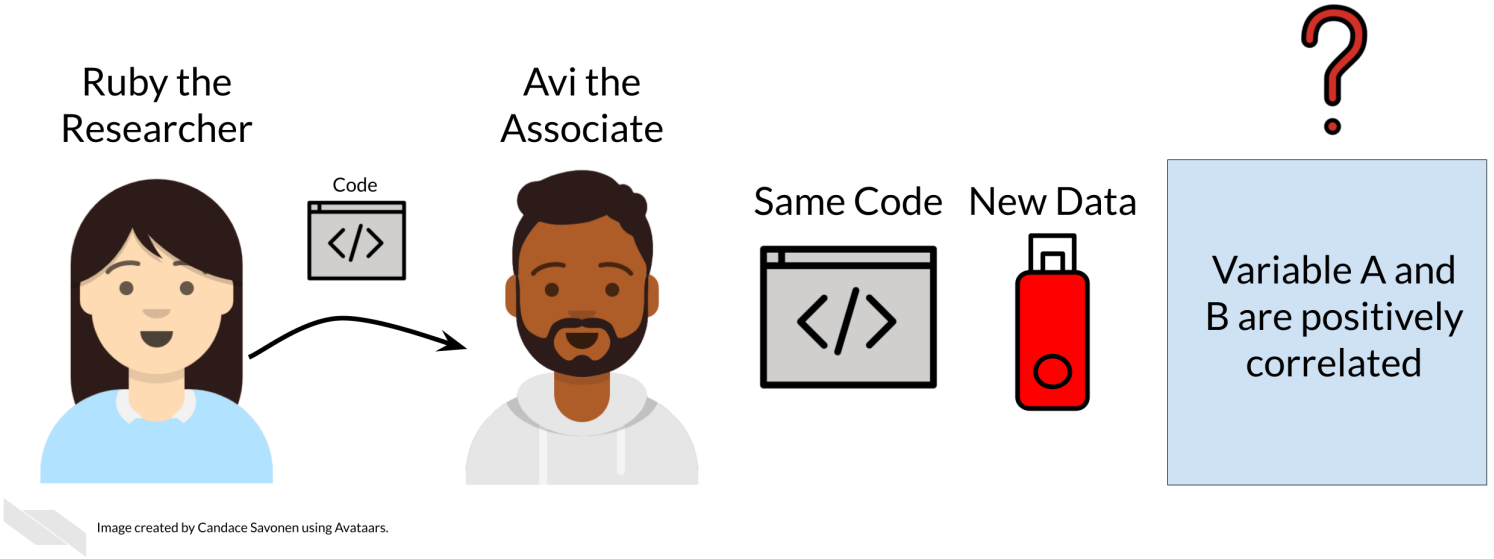
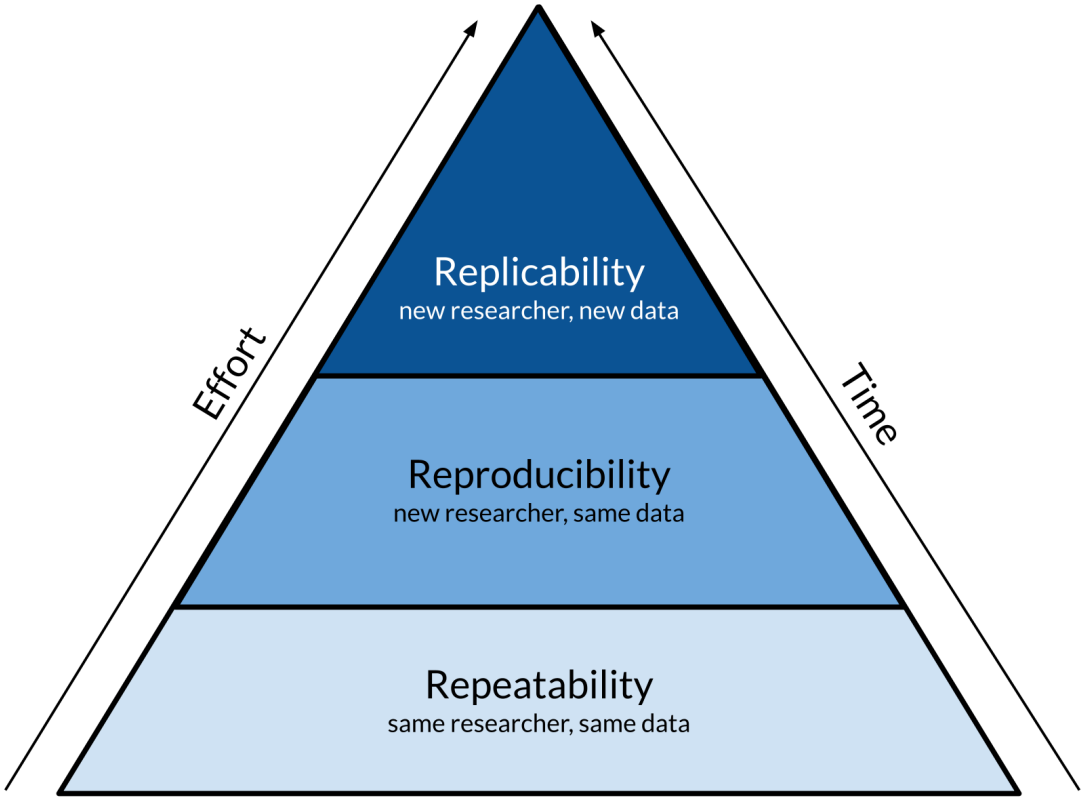


Image created by Candace Savonen using Avataars.

Replicable: with new data do we obtain the same inferences?



Reproducibility vs Repeatability vs Replicability



Based off of a figure from Essawy et al, 2020 https://doi.org/10.1016/j.envsoft.2020.104753

Why Reproducibility is important…

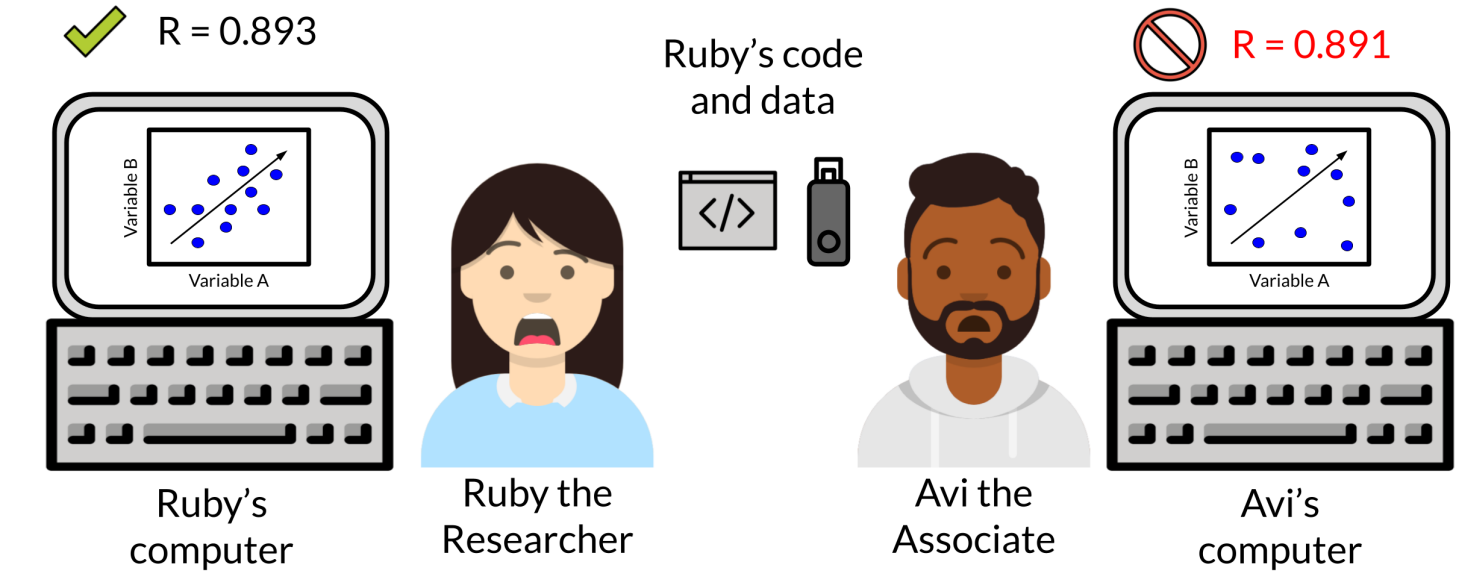


Image created by Candace Savonen using Avataars.

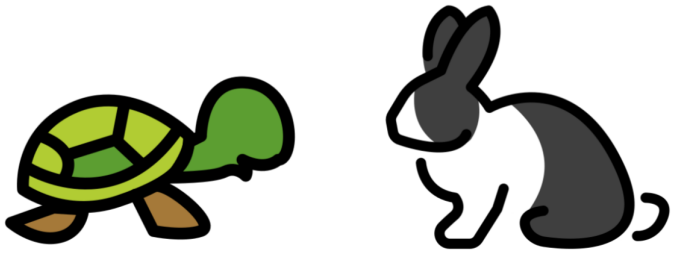
We can't get to replicability without reproducibility

Just because something is reproducible doesn't mean it is correct.

But it is a necessary step to help **check for correctness** and get to **replicability**.

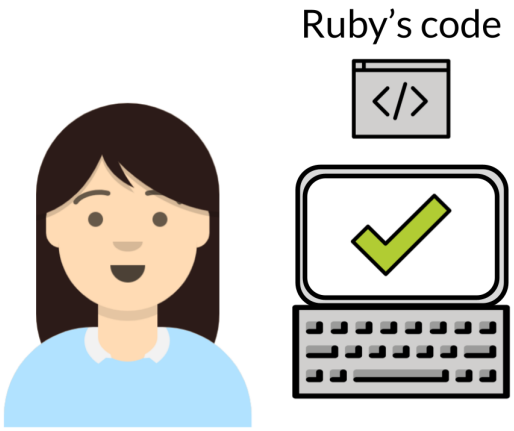
It's worth the wait

Reproducibility is a tortoise's game - it's an incremental and slow process *but* **it has high payoffs!**



Reproducibility can also be for your future self!

Now Ruby



Future Ruby

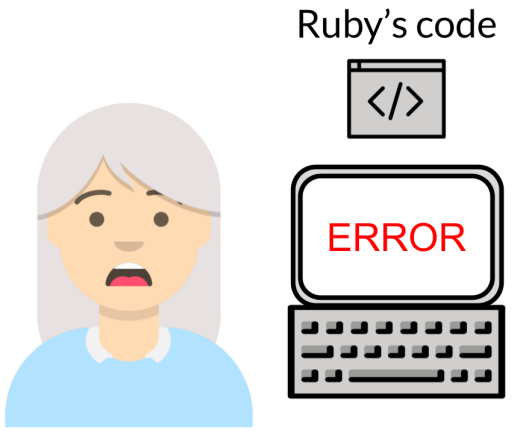
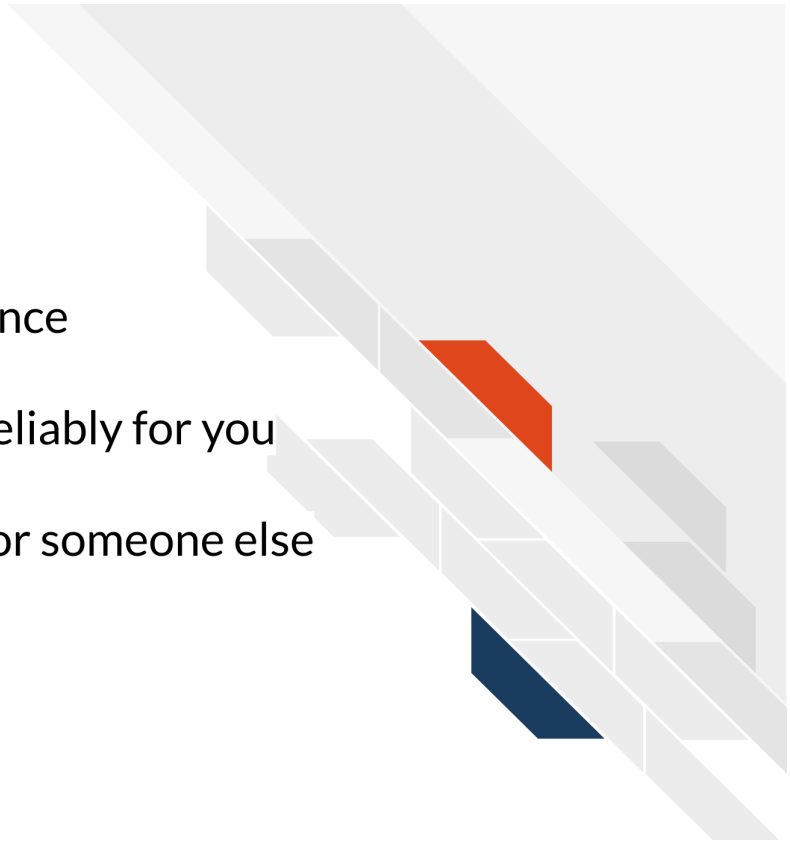


Image created by Candace Savonen using Avataars.

The process



Step 1) Get your code to work once

Step 2) Get your code to work reliably for you

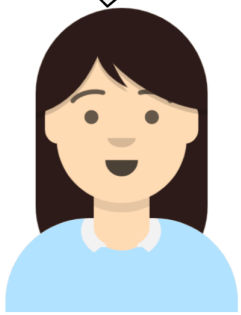
Step 3) Get your code to work for someone else

R Markdown

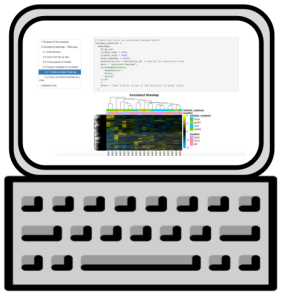
R Markdown notebooks are a handy tool for reproducibility!

R Markdown lets you test your work

Working from this notebook allows me to interactively develop on my data analysis and write down my thoughts about the process all in one place!



Ruby the Researcher



RMarkdown is conducive to interactive development!

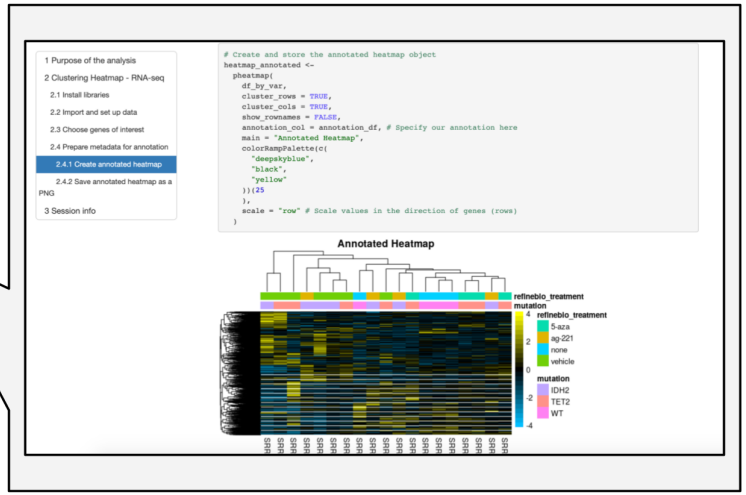
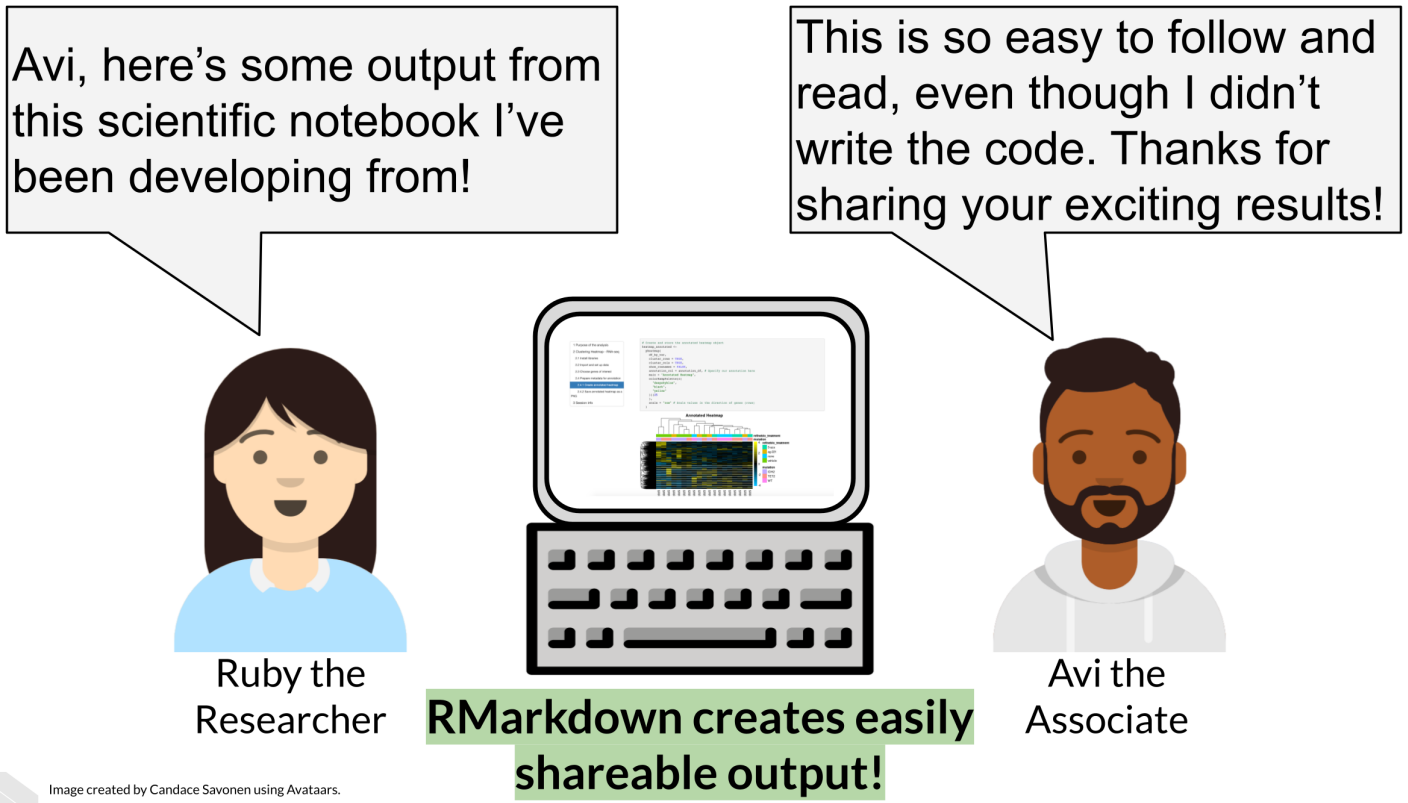


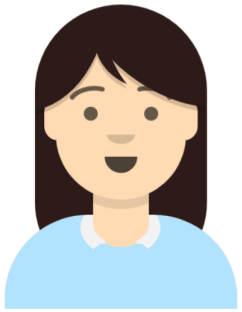
Image created by Candace Savonen using Avataars.

R Markdown allows you to more clearly show what you did

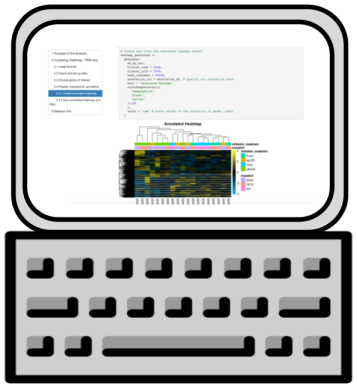


R Markdown makes it easier to update code and see results

Yay! I just got the data for 5 more samples. Because of my handy notebook set up, I can easily call one command and re-run the analysis so it is updated with the new samples included!



Ruby the Researcher



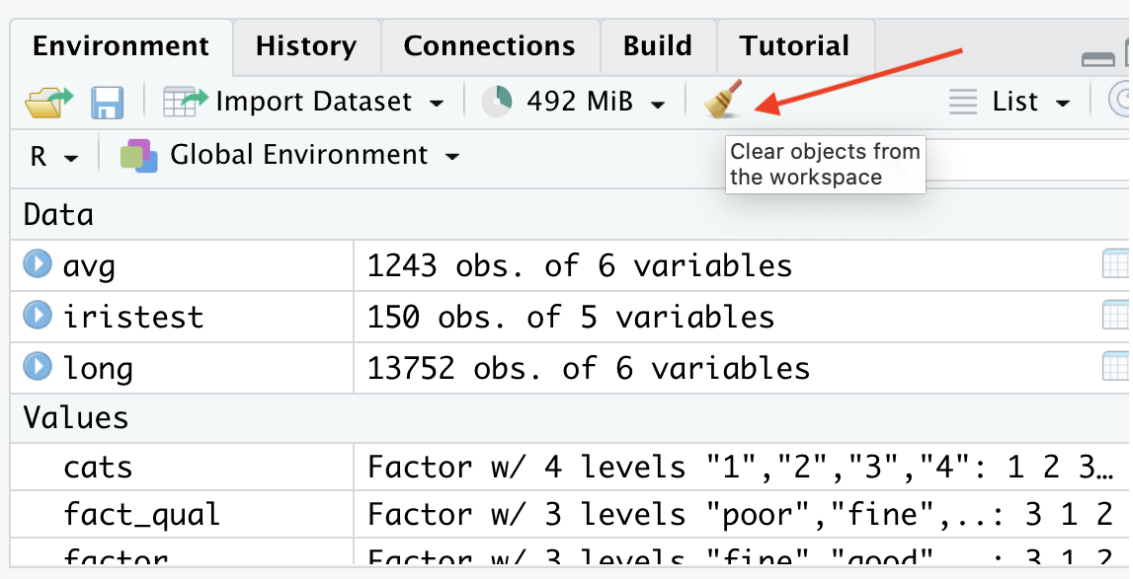
RMarkdown is handy for creating updateable reports!

Image created by Candace Savonen using Avataars.

Clean your environment

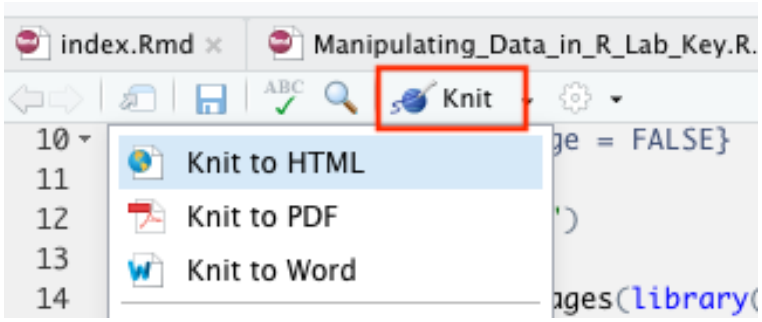
Regularly cleaning your environment and trying your code again, can help ensure that your code is running as expected.

Occasionally we might forget to save a step of our code in our R Markdown file that we ran only in the console. This will help us figure that out.



Check if your file knits regularly

Regularly checking if your file knits will help you spot a missing step or error earlier when you have less code to try to identify where your code might have gone wrong.



Tell your future self and others what you did!

Provide sufficient detail so that you can understand what you did.

Need random numbers to stay consistent?

Use `set.seed()` : sets the starting state for the random number generator (RNG) in R.

```
set.seed(123)
sample(10)
```

```
 [1]  3 10  2  8  6  9  1  7  5  4
```

```
set.seed(123)
sample(10)
```

```
 [1]  3 10  2  8  6  9  1  7  5  4
```

```
set.seed(456)
sample(10)
```

```
 [1]  5  3  6 10  4  9  1  2  8  7
```

Note that these are only psuedo random and the values are created doing calculations based on the given seed. Thus the same "random" values will be reproduced by everyone using the same seed with `set.seed`.

R Markdown syntax

Before:

```
# Header - biggest font created by hashtag and space
## SubHeader Second Biggest created by 2 hashtags and space

**bold** text
*italicized* text

`code` referenced outside of a chunk needs backticks
```

After knit:

Header - biggest font created by hashtag and space

SubHeader Second Biggest created by 2 hashtags and space

bold text *italicized* text

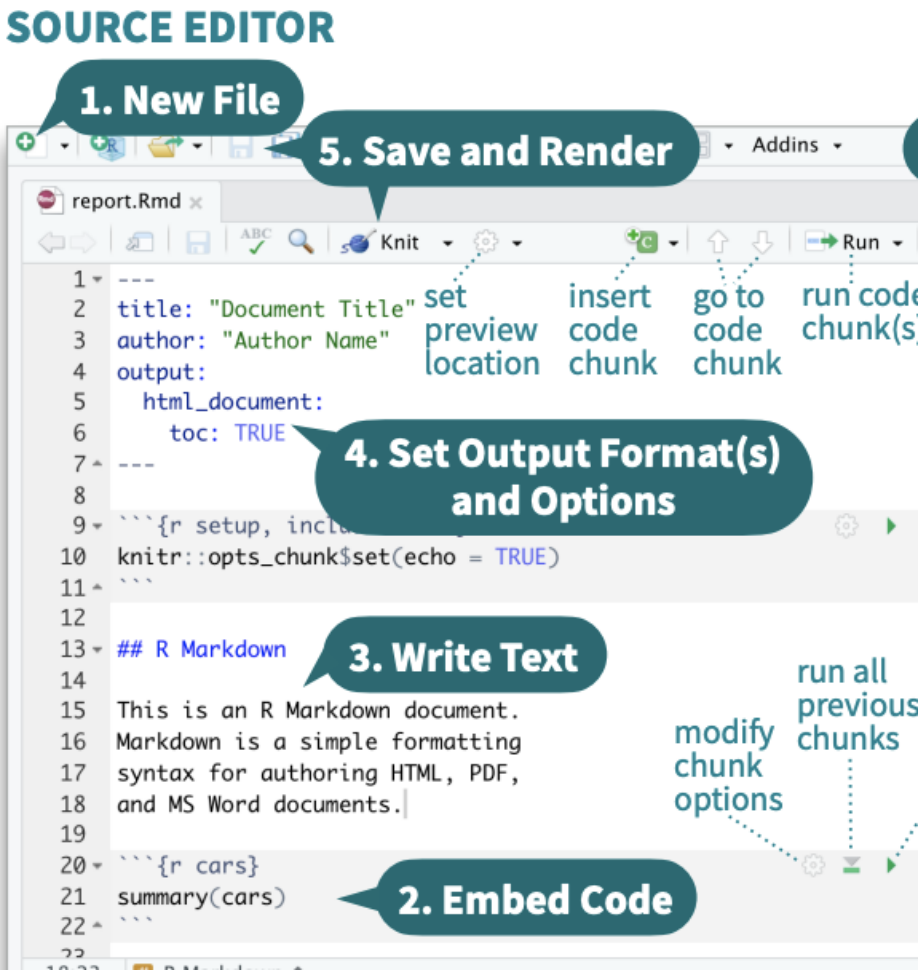
`code` referenced outside of a chunk needs backticks

R Markdown syntax

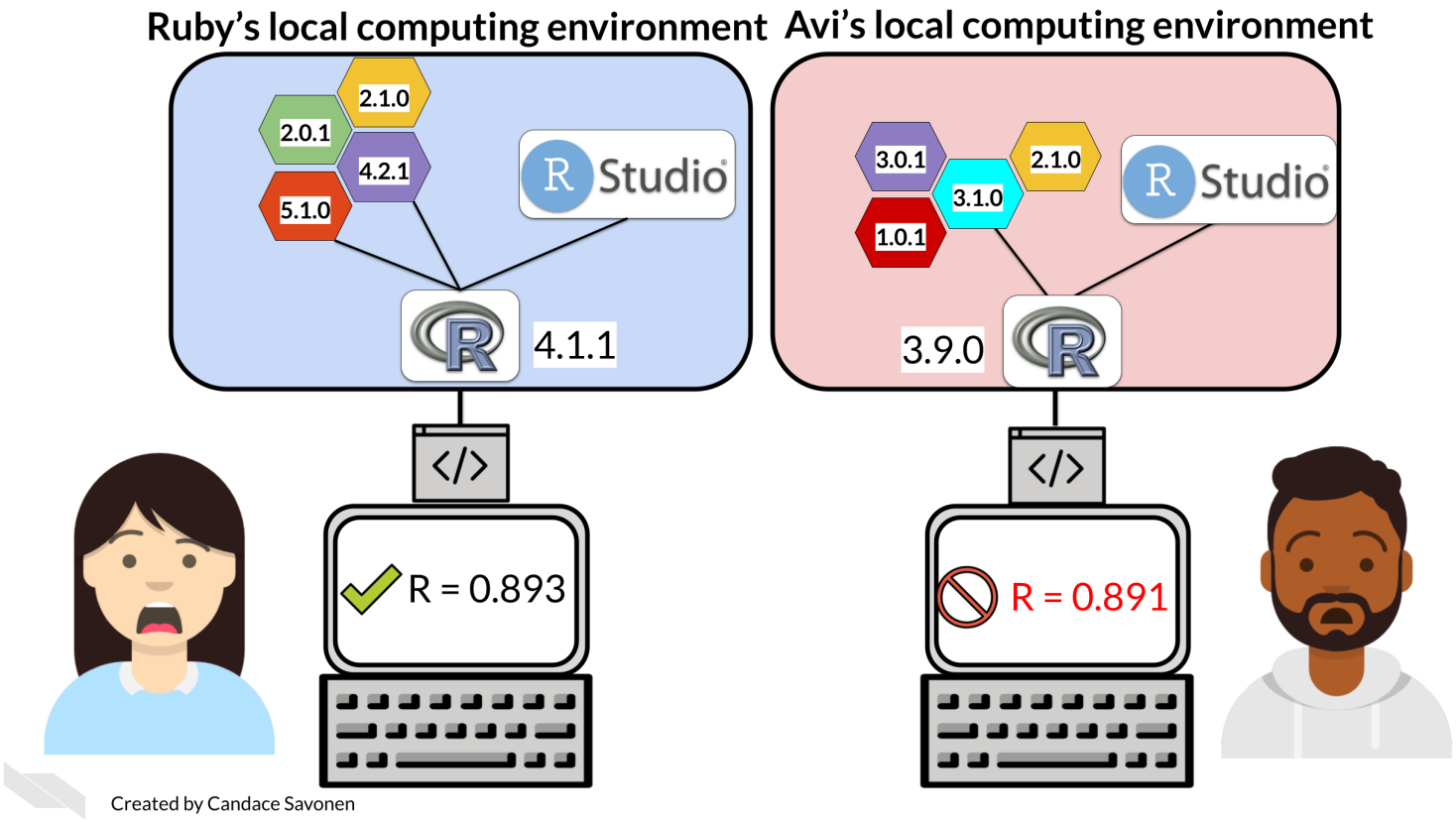
Go to Help > Cheat Sheets > R Markdown Cheat Sheet (which will download it)

Or checkout Help > Cheat Sheets > R Markdown Reference Guide

Or checkout the Class Website!



Versions matter



Session info can help

`sessionInfo()`

Ruby's session info print out

```
R version 4.0.2 (2020-06-22)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.2 LTS

Matrix products: default
BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.8.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=en_US.UTF-8     LC_MONE
 [7] LC_PAPER=en_US.UTF-8       LC_NAME
[10] LC_TELEPHONE=C             LC_MEAS

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] rmarkdown_2.4

loaded via a namespace (and not attached):
 [1] rstudioapi_0.11   knitr_1.30        magrittr_1.5      hms_0.5.3         tidyselect_1.
 [6] R6_2.4.1          rlang_0.4.7       fansi_0.4.1       dplyr_1.0.2       tools_4.0.2
[11] xfun_0.18         sessioninfo_1.1.1 tinytex_0.26      cli_2.0.2         withr_2.3.0
[16] htmltools_0.5.0   ellipsis_0.3.1    assertthat_0.2.1  yaml_2.2.1        digest_0.6.25
[21] tibble_3.0.3      lifecycle_0.2.0   crayon_1.3
[26] vctrs_0.3.4       glue_1.4.2        evaluate_0
[31] generics_0.0.2    jsonlite_1.7.1    pkgconfig_2.0.3
```

Avi's session info print out

```
R version 4.0.5 (2021-03-31)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Big Sur 10.16

Matrix products: default
LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dyli

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] rmarkdown_2.10

loaded via a namespace (and not attached):
 [1] lemmabuild_0.1.2     BiocManager_1.30.16 compiler_4.0.5      magrittr_2.0.1
 [5] fastmap_1.1.0        htmltools_0.5.2     tools_4.0.5         yaml_2.2.1
 [9] tinytex_0.33         knitr_1.33          digest_0.6.27       xfun_0.25
[13] rlang_0.4.11         evaluate_0.14
```

R version 4.0.2 vs 4.0.5

Different operating systems!

rmarkdown 2.4 vs 2.10

If Avi and Ruby have discrepancies in their results, the session info print out gives a record which may have clues to why that might be!

Image by Candace Savonen

GUT CHECK: Why is reproducibility so important?

A. It helps to ensure that your code is working consistently and it helps others understand what you did

B. It ensures that your code is correct

GUT CHECK: What is NOT a practice to improve the reproducibility of our work?

A. Using R Markdown files to describe what your code is doing

B. Using scripts instead of R Markdown files

C. Testing your code with R Markdown files or the run previous button

D. Regularly cleaning the environment

More resources

These are just some quick tips, for more information:

- Reproducibility in Cancer Informatics course
- Advanced Reproducibility in Cancer Informatics course
- The RMarkdown book
- Jenny Bryan's organizational strategies.
- Write efficient R code for science.

Summary

To help make your work more reproducible:

- Use R Markdown
- Clean your environment regularly
- Check the knit of your R Markdown regularly
- Tell your future self and others what you did!
- Print session info!

- Class Website
- Lab
- Day 2 Cheatsheet
- Posit's R Markdown Cheatsheet

Image by Gerd Altmann from Pixabay

Data Input

Quick reminder - Getting files from downloads

This course will involve moving files around on your computer and downloading files.

If you are new to this - check out the videos on the resource page of the website.

R Projects

R Projects

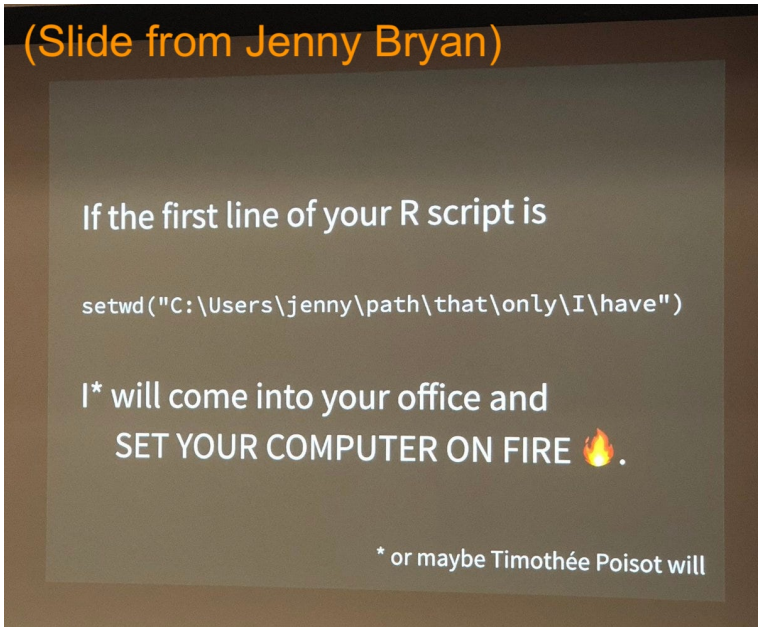
R Projects are a super helpful feature of RStudio. They help you:

- **Stay organized.** R Projects help in organizing your work into self-contained directories (folders), where all related scripts, data, and outputs are stored together. This organization simplifies file management and makes it easier to locate and manage files associated with your analysis or project.
- **Find the right files.** When you open an R Project, RStudio automatically sets the working directory to the project's directory. This is where RStudio "looks" for files. Because it's always the Project folder, it can help avoid common issues with file paths.
- **Be more reproducible.** You can share the entire project directory with others, and they can replicate your environment and analysis without much hassle.

Why projects?

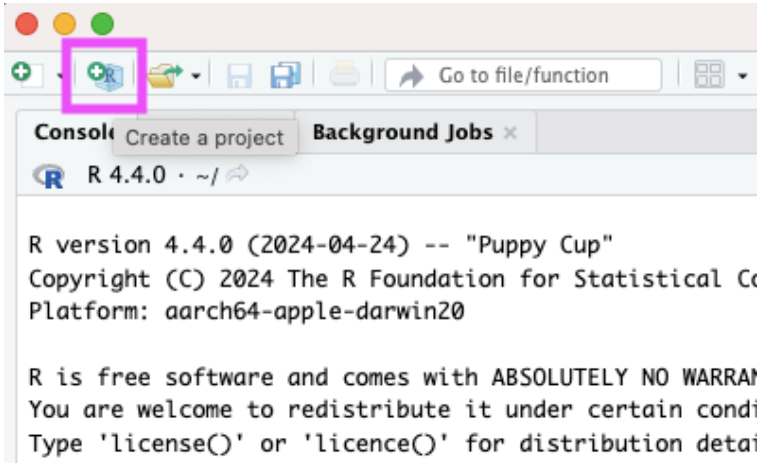
"The chance of the `setwd()` command having the desired effect – making the file paths work – for anyone besides its author is 0%. It's also unlikely to work for the author one or two years or computers from now. The project is not self-contained and portable." - Jenny Bryan

Let's go over how to create and use an R Project!



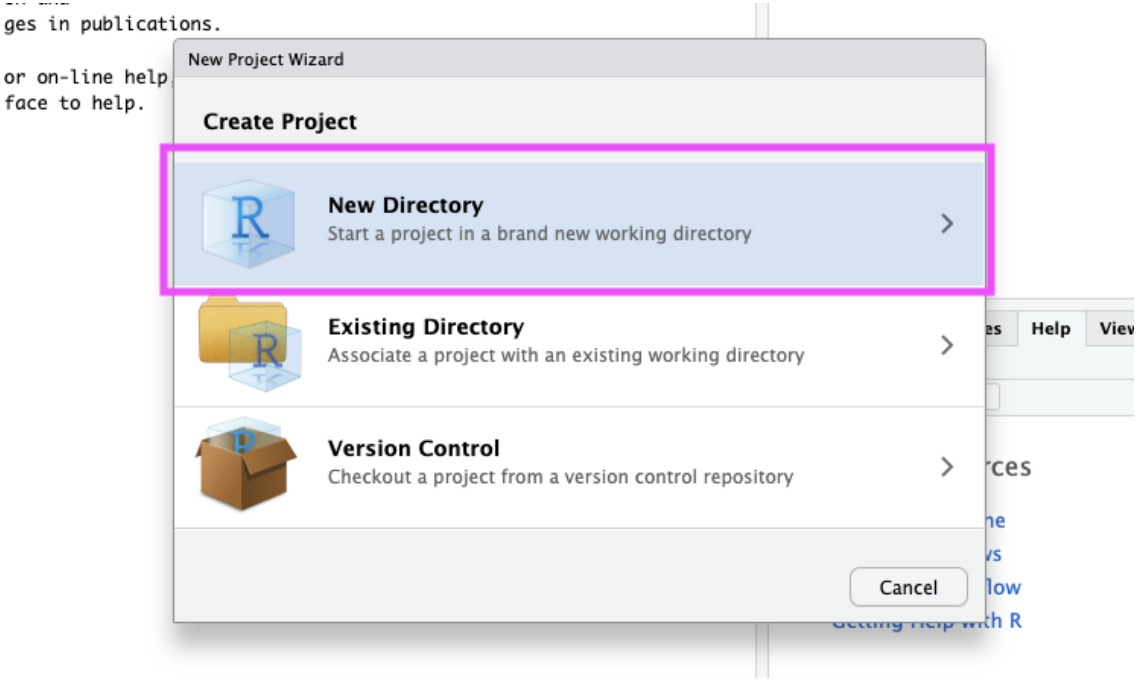
New R Project

Let's make an R Project so we can stay organized in the next steps. Click the new R Project button at the top left of RStudio:



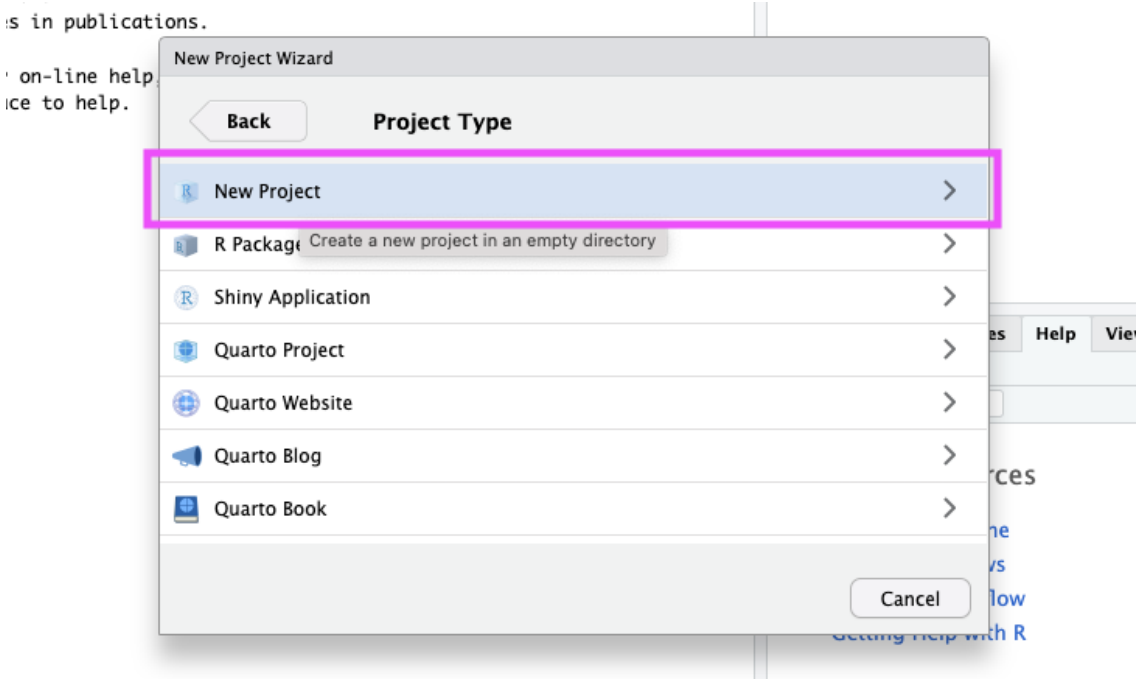
New R Project

In the New Project Wizard, click “New Directory”:



New R Project

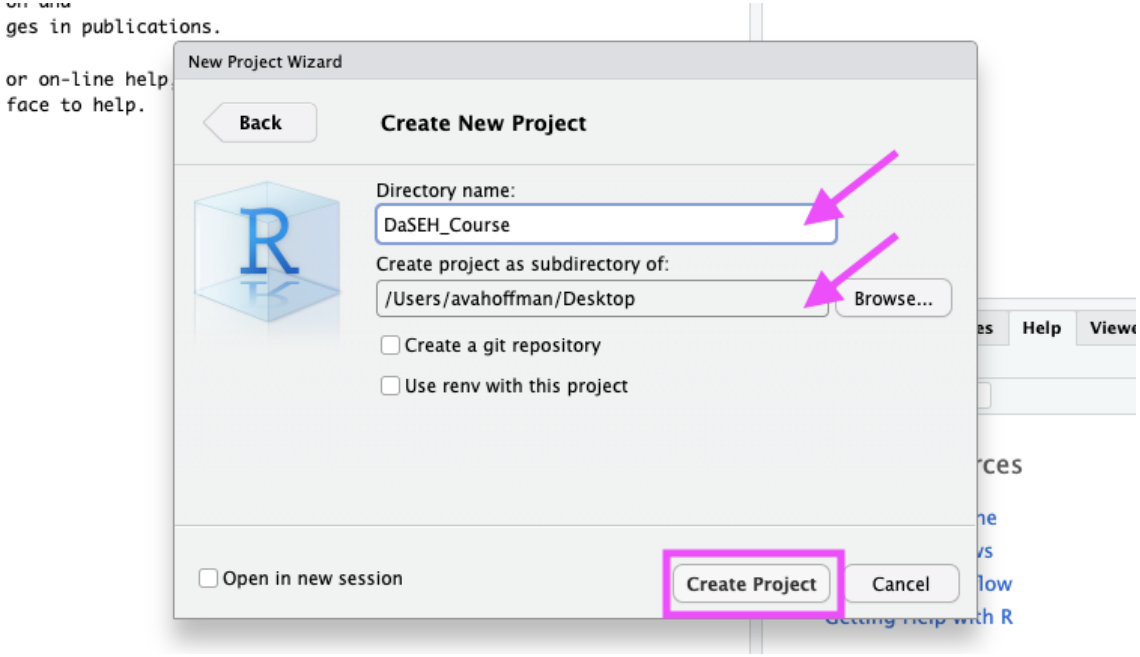
Click “New Project”:



New R Project

Type in a name for your new folder.

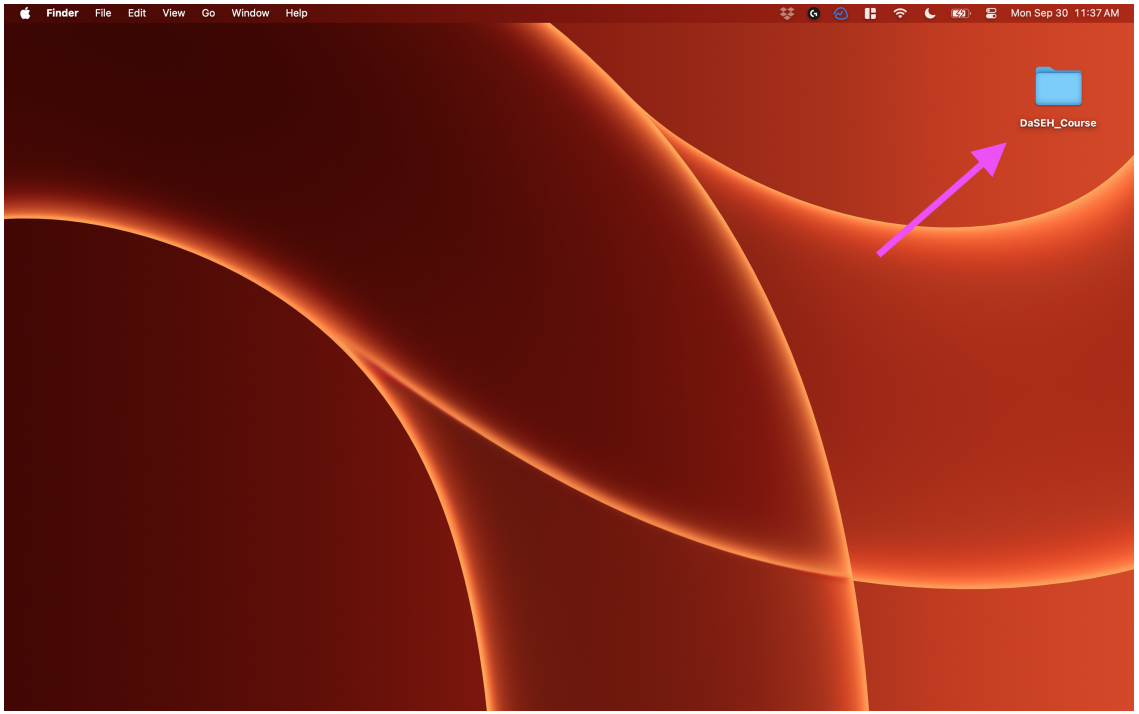
Store it somewhere easy to find, such as your Desktop:



New R Project

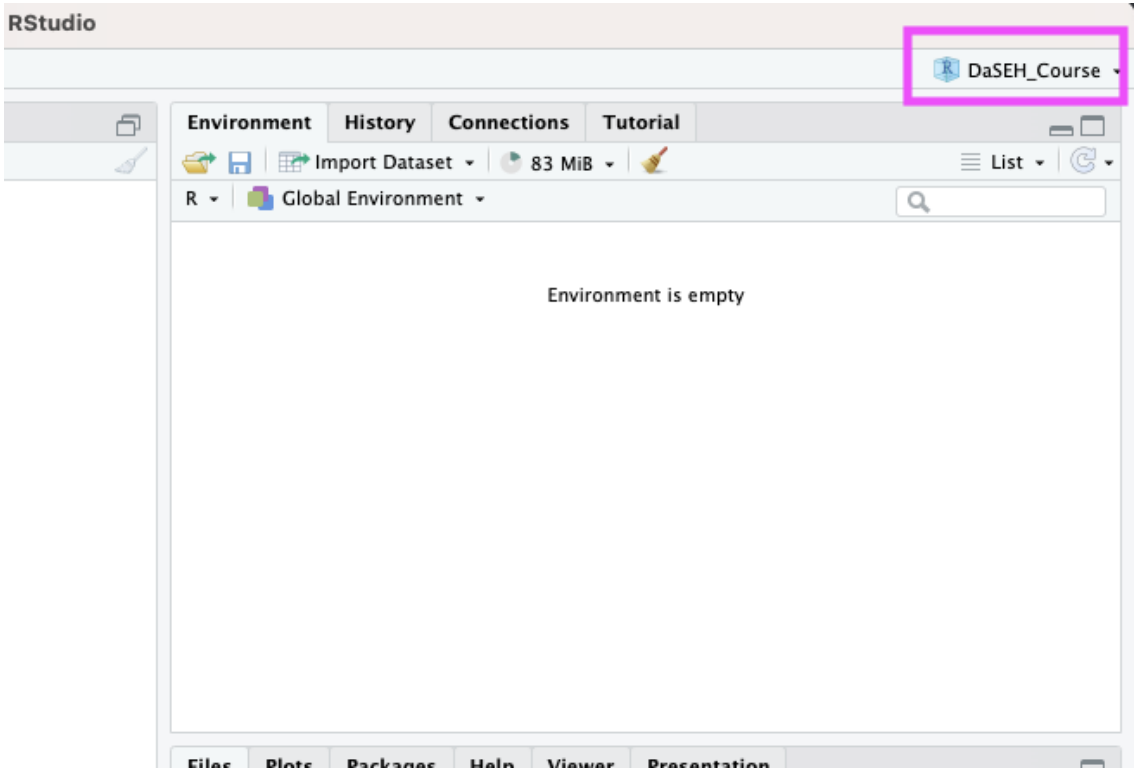
You now have a new R Project folder on your Desktop!

Make sure you add any scripts or data files to this folder as you go through your Intro to R lessons, or work on a new project. This will make sure R is able to “find” your files.



See and change projects

You can see what project you have open in the top right corner.



Getting data into R (manual/point and click)

Data Input

- 'Reading in' data is the first step of any real project/analysis
- R can read almost any file format, especially via add-on packages
- We are going to focus on simple delimited files first
    - comma separated (e.g. '.csv')
    - tab delimited (e.g. '.txt')

delimiters are symbols that separate cells in a simple-text file.

Data Input

CO Heat-related ER visits dataset:

We're going to load a dataset from the Colorado Environmental Public Health Tracking Program. This dataset has age-adjusted visit rates and total number of visits for all genders by Colorado county for 2011-2022.

- Check out the data at: https://coepht.colorado.gov/heat-related-illness

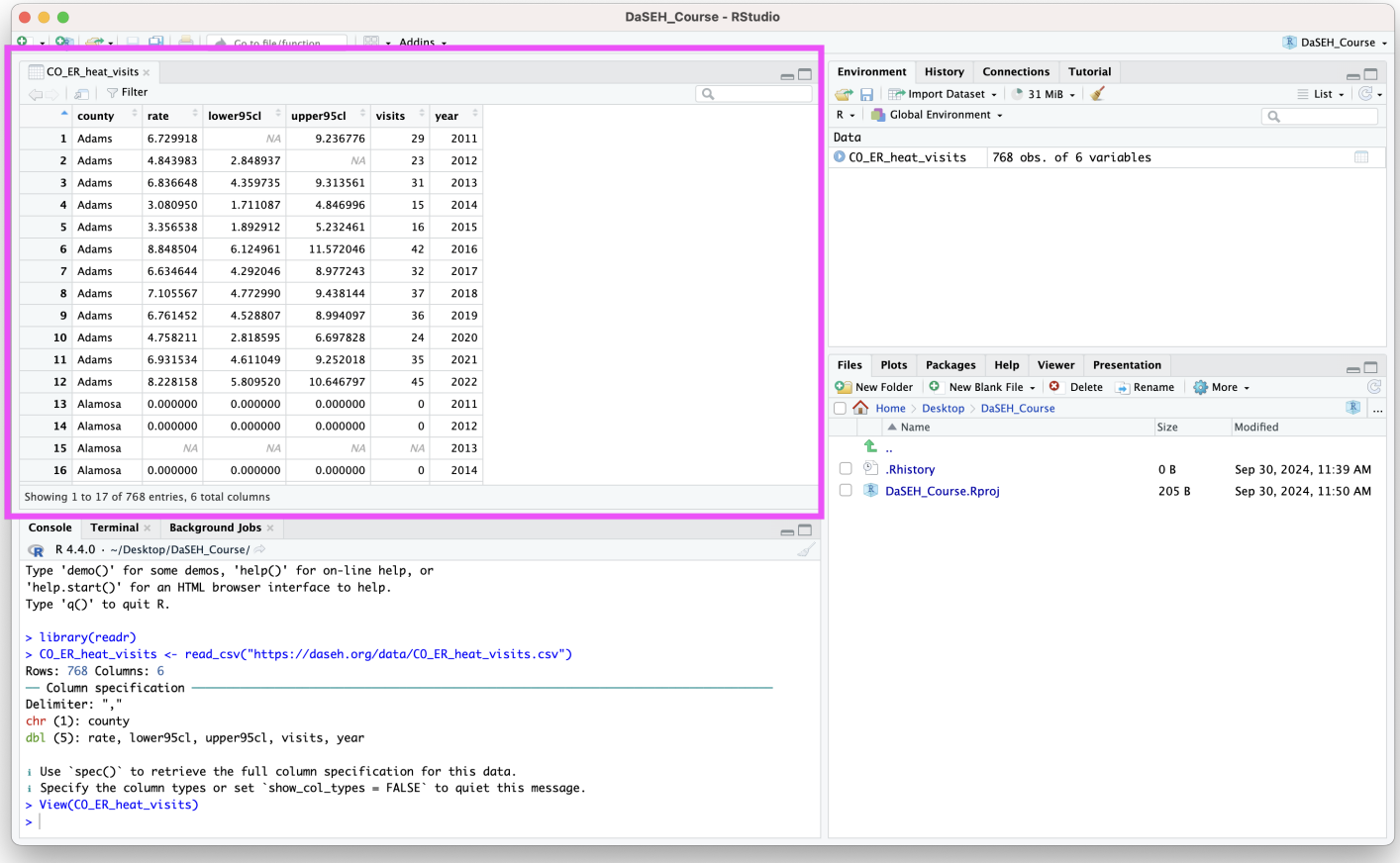
Import Dataset (URL)

- > File
- > Import Dataset
- > From Text (`readr`)
- > paste the url (https://daseh.org/data/CO_ER_heat_visits.csv)
- > click “Update” and “Import”

Saves data in memory, not to hard drive

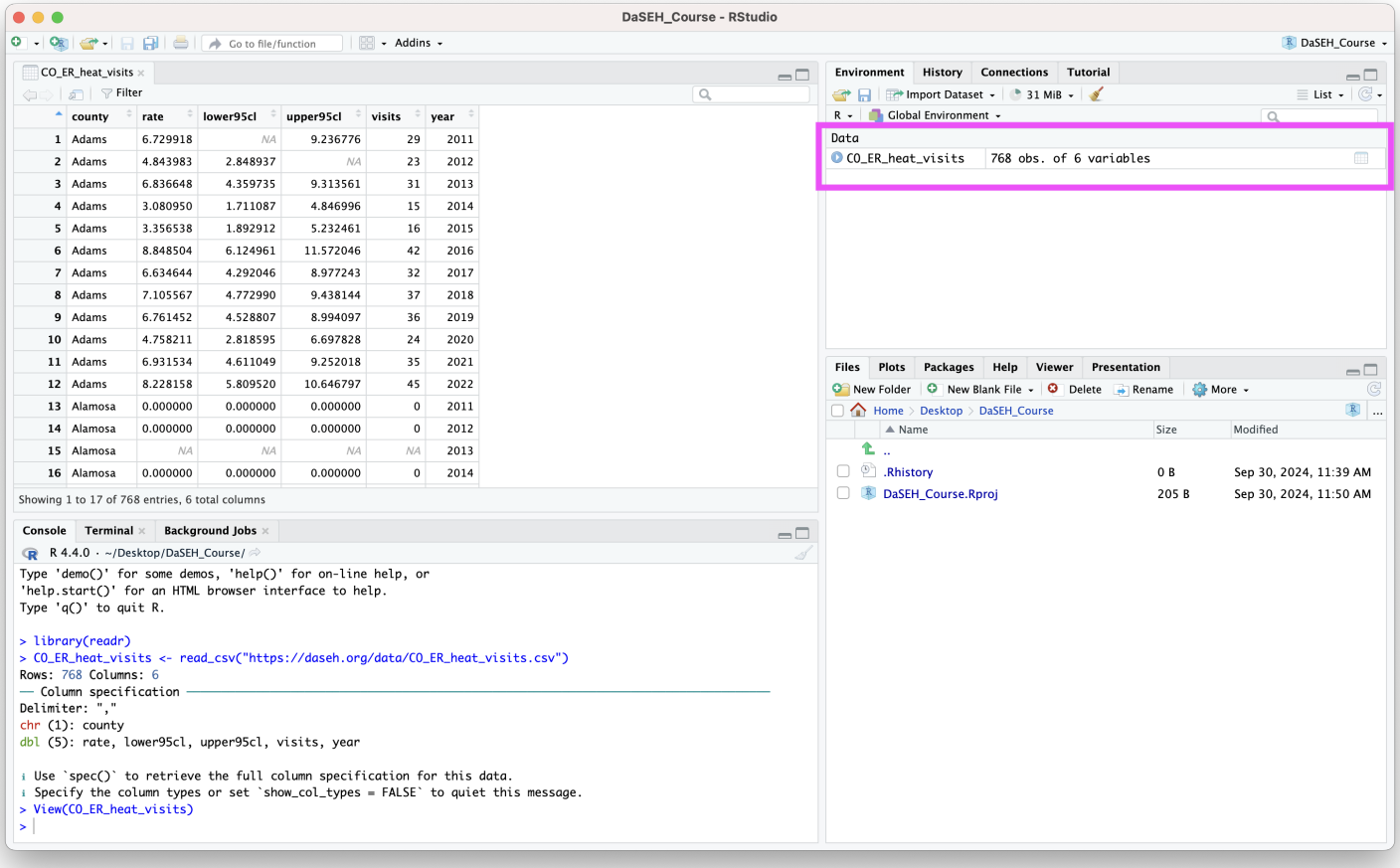
What Just Happened?

You see a preview of the data on the top left pane.



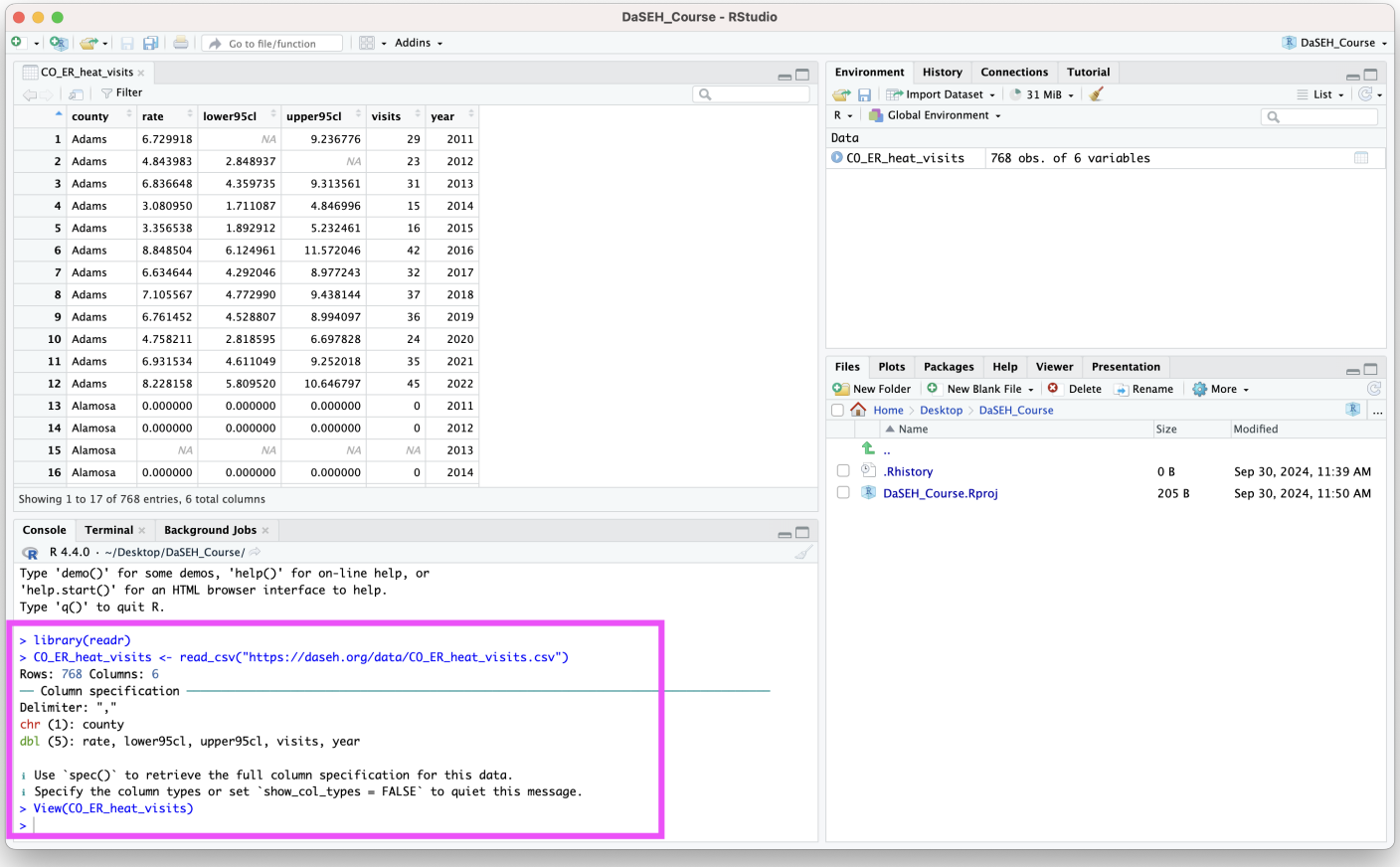
What Just Happened?

You see a new object called `CO_ER_heat_visits` in your environment pane (top right). The table button opens the data for you to view.

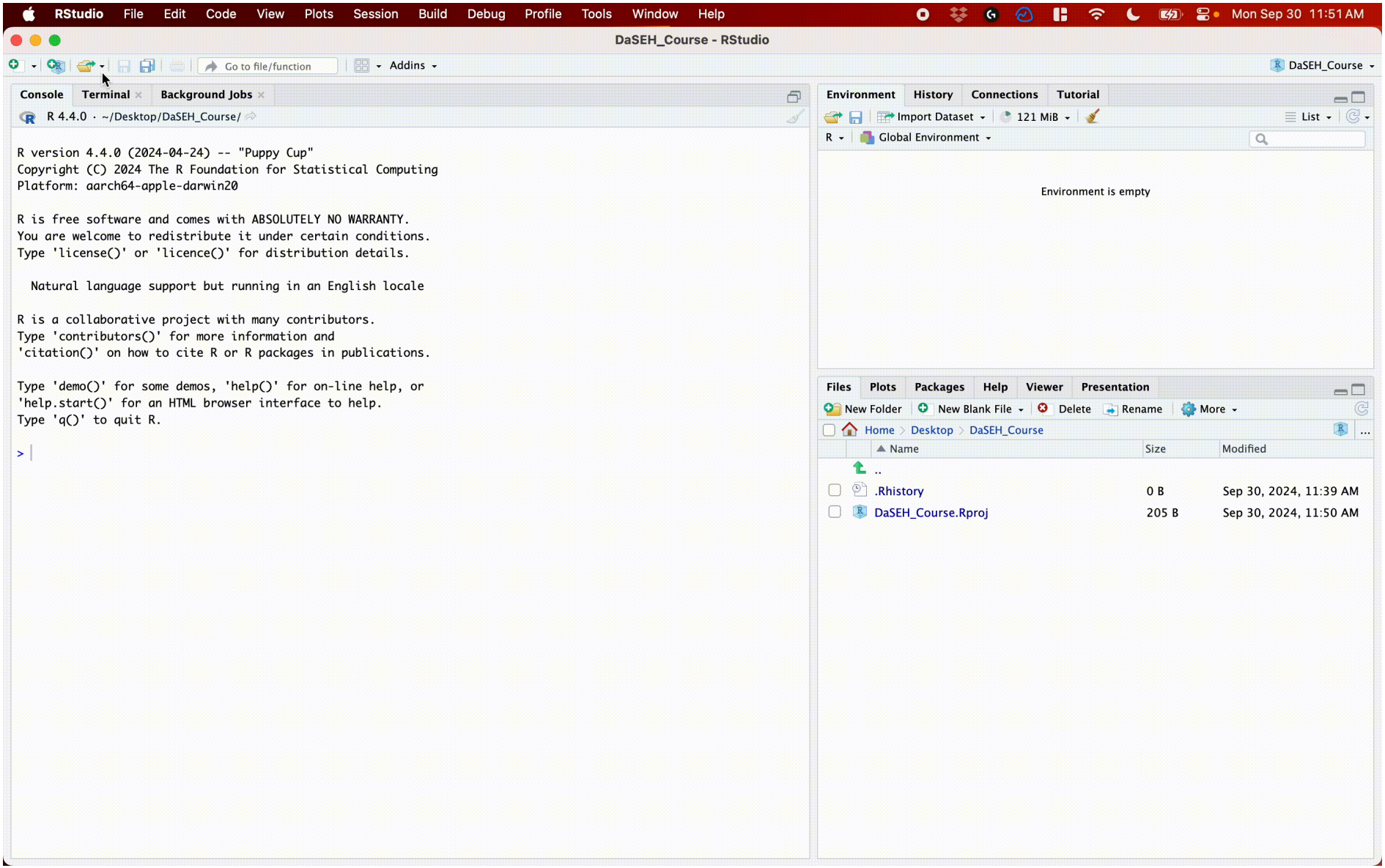


What Just Happened?

R ran some code in the console (bottom left).



Import Dataset



Import Dataset (file)

- > *Download the data*
- > *Put data in the project folder*
- > File, Import Dataset, From Text (`readr`)
- > browse for the file
- > click “Update” and “Import”

GUT CHECK!

How can we get data into R?

A. From a URL

B. From a file we downloaded

C. Both of these!

Lab - Part 1

- Class Website
- Data Input Lab

Manual Import: Pros and Cons

Pros: easy!!

Cons: obscures some of what's happening, others will have difficulty running your code

Getting data into R (directly)

Data Input: Read in Directly

The `tidyverse` contains a package `readr` that is handy for importing data.

```
library(tidyverse)
dat <- read_csv(
  file = "https://daseh.org/data/CO_ER_heat_visits.csv"
)

# `head` displays first few rows of a data frame. `tail()` works the same way.
head(dat, n = 5)

# A tibble: 5 × 6
  county  rate lower95cl upper95cl visits  year
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Adams   6.73     NA         9.24     29  2011
2 Adams   4.84      2.85     NA        23  2012
3 Adams   6.84      4.36      9.31     31  2013
4 Adams   3.08      1.71      4.85     15  2014
5 Adams   3.36      1.89      5.23     16  2015
```

Data Input: Declaring Arguments

```
dat <- read_csv(
  file = "https://daseh.org/data/CO_ER_heat_visits.csv"
)
# EQUIVALENT TO
dat <- read_csv(
  "https://daseh.org/data/CO_ER_heat_visits.csv"
)
```

Data Input: Read in Directly

`read_csv()` needs an argument `file =` **in quotation marks**.

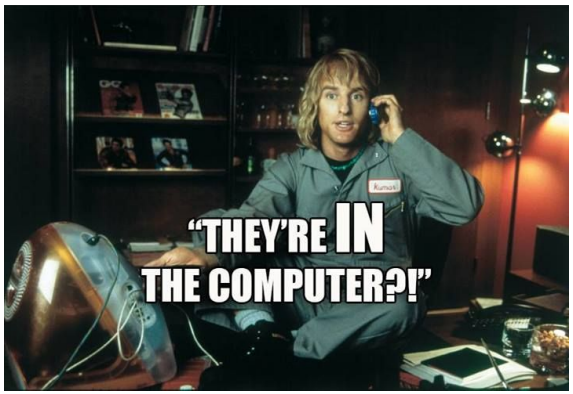
- can be path to a file on a website (URL)
- can be **path** in your local computer – absolute file path or relative file path

```
# URL
dat <- read_csv("https://daseh.org/data/CO_ER_heat_visits.csv")

# In project folder
dat <- read_csv("CO_ER_heat_visits.csv")
```

The working directory

When we work in an R Project, our project folder is our **working directory**.

Working directory is a folder (directory) that RStudio will use to find files.



Checking the working directory

Run the `getwd()` function to determine your working directory.

You can also click the ⚙ in the Files pane > Go To Working Directory.

```
# Get the working directory
getwd()
```

Setting the working directory

You can set the working directory manually with the `setwd()` function. But it's easier to set up a project :)

```
# set the working directory
setwd("/Users/avahoffman/Desktop")
```

Now what? Checking data & Other formats

Data Input: Checking the data

- the `View()` function shows your data in a new tab, in spreadsheet format
- be careful if your data is big!

```
View(dat)
```

Data Input: Other delimiters

`read_tsv()` can read tab delimited (separated) files.

`read_delim()` can be used to specify the delimiter.

- `file` is the path to your file, in quotes
- `delim` is what separates the fields within a record

```
## Examples
dat2 <- read_tsv(file = "table1.tsv", delim = "\t")

dat3 <- read_delim(file = "data.txt", delim = ":")
```

Data input: other file types

- `readxl` package can read excel files
- `haven` package has functions to read SAS, SPSS, Stata formats
- There are also resources for REDCap : `REDCapR`

WARNING! `read.csv` is * base R *

There are also data importing functions provided in base R (rather than the `readr` package), like `read.delim()` and `read.csv()`.

These functions have slightly different syntax for reading in data (e.g. `header` argument).

However, while many online resources use the base R tools, the latest version of RStudio switched to use these new `readr` data import tools, so we will use them in the class for slides. They are also up to two times faster for reading in large datasets, and have a progress bar which is nice.

Other Useful Functions

- The `str()` function can tell you about data/objects.
- We will also discuss the `glimpse()` function later, which does something very similar.
- `head()` shows first few rows
- `tail()` shows the last few rows

Summary

R Projects can make it easier to find files.

Importing data manually:

- File > Import Dataset > From Text (`readr`)
- Paste the url / browse
- Click “Update” and “Import”
- Review the process: `https://youtu.be/LEkNfJgpunQ`

Importing data programmatically:

- `read_csv()` function from tidyverse (`readr`) package
- Use `getwd()` to check your working directory, where R looks for your data files

Summary - Part 2

Look at your data!

- Check the environment for a data object
- `View()` gives you a preview of the data in a new tab

Other file types

- `readr` package: `read_delim()` for general delimited files
- other packages for more complicated files.

Don't forget to use `<-` to assign your data to an object!

Lab - Part 2

- Class Website
- Data Input Lab
- Posit's Data Import Cheatsheet
- Day 2 Cheatsheet

Image by Gerd Altmann from Pixabay

Subsetting Data in R

Reminder

Refresh the website and get the latest version of the labs and slides! We are constantly making improvements.

Recap

- Use `<-` to save (assign) values to objects
- Use `c()` to **combine** vectors
- `length()`, `class()`, and `str()` tell you information about an object
- The sequence `seq()` function helps you create numeric vectors (`from`,`to`, `by`, and `length.out` arguments)
- The repeat `rep()` function helps you create vectors with the `each` and `times` arguments
- Reproducible science makes everyone's life easier!
- The `readr` package has helpful functions like `read_csv()` that can help you import data into R

 Day 2 Cheatsheet

Overview

In this module, we will show you how to:

1. Look at your data in different ways
2. Create a data frame and a tibble
3. Create new variables/make rownames a column
4. Rename columns of a data frame
5. Subset rows of a data frame
6. Subset columns of a data frame
7. Add/remove new columns to a data frame
8. Order the columns of a data frame
9. Order the rows of a data frame

Setup

We will largely focus on the `dplyr` package which is part of the `tidyverse`.

Some resources on how to use `dplyr`:

- https://dplyr.tidyverse.org/
- https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html
- https://www.opencasestudies.org/

Why dplyr?

hadley commented on May 26, 2016 — Member

The d is for dataframes, the plyr is to evoke pliers. Pronounce however you like.

👍 1

The `dplyr` package is one of the most helpful packages for altering your data to get it into a form that is useful for creating visualizations, summarizing, or more deeply analyzing.

So you can imagine using pliers on your data.

Loading in dplyr and tidyverse

See this website for a list of the packages included in the `tidyverse`:
https://www.tidyverse.org/packages/

```
library(tidyverse) # loads dplyr and other packages!
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ forcats   1.0.0     ✔ readr     2.1.5
✔ ggplot2   3.5.2     ✔ stringr   1.5.1
✔ lubridate 1.9.4     ✔ tibble    3.2.1
✔ purrr     1.0.4     ✔ tidyr     1.3.1
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all confl
```

Getting data to work with

We will work with data called `er` about heat-related ER visits between 2011 and 2022, as reported by the state of Colorado, specifically made available by the Colorado Environmental Public Health Tracking program website. Full dataset available at https://coepht.colorado.gov/heat-related-illness.

```
er <-
  read_csv("https://daseh.org/data/CO_ER_heat_visits.csv")
```

```
Rows: 768 Columns: 6
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (1): county
dbl (5): rate, lower95cl, upper95cl, visits, year

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

Checking the data `dim()`

The `dim()`, `nrow()`, and `ncol()` functions are good options to check the dimensions of your data before moving forward.

```
dim(er) # rows, columns
```

```
[1] 768   6
```

```
nrow(er) # number of rows
```

```
[1] 768
```

```
ncol(er) # number of columns
```

```
[1] 6
```

Checking the data: `glimpse()`

In addition to `head()` and `tail()`, the `glimpse()`function of the `dplyr` package is another great function to look at your data.

```
glimpse(er)
```

```
Rows: 768
Columns: 6
$ county    <chr> "Adams", "Adams", "Adams", "Adams", "Adams", "Adams", "Adams…
$ rate      <dbl> 6.729918, 4.843983, 6.836648, 3.080950, 3.356538, 8.848504, …
$ lower95cl <dbl> NA, 2.848937, 4.359735, 1.711087, 1.892912, 6.124961, 4.2920…
$ upper95cl <dbl> 9.236776, NA, 9.313561, 4.846996, 5.232461, 11.572046, 8.977…
$ visits    <dbl> 29, 23, 31, 15, 16, 42, 32, 37, 36, 24, 35, 45, 0, 0, NA, 0,…
$ year      <dbl> 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, …
```

This dataset has information about both the *rate* of ER visits for heat-related illness and the *number of visits*, broken out by both year and CO county.

Checking your data: `slice_sample()`

What if you want to see the middle of your data? You can use the `slice_sample()` function of the `dplyr` package to see a **random** set of rows. You can specify the number of rows with the `n` argument.

```
slice_sample(er, n = 2)
```

```
# A tibble: 2 × 6
  county    rate lower95cl upper95cl visits  year
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Yuma        NA        NA        NA     NA  2012
2 Garfield    NA        NA        NA     NA  2015
```

```
slice_sample(er, n = 2)
```

```
# A tibble: 2 × 6
  county  rate lower95cl upper95cl visits  year
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Eagle      0         0         0      0  2015
2 Teller    NA        NA        NA     NA  2018
```

Data frames and tibbles

Data frames

An older version of data in tables is called a data frame. The mtcars dataset is an example of this.

```
class(mtcars)
```

```
[1] "data.frame"
```

```
head(mtcars)
```

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

tibble

Tibbles are a **fancier** version of data frames:

- We don't have to use head to see a preview of it
- We see the dimensions
- We see the data types for each column

```
er
```

```
# A tibble: 768 × 6
   county  rate lower95cl upper95cl visits  year
   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Adams   6.73     NA         9.24     29  2011
 2 Adams   4.84      2.85     NA        23  2012
 3 Adams   6.84      4.36      9.31     31  2013
 4 Adams   3.08      1.71      4.85     15  2014
 5 Adams   3.36      1.89      5.23     16  2015
 6 Adams   8.85      6.12     11.6      42  2016
 7 Adams   6.63      4.29      8.98     32  2017
 8 Adams   7.11      4.77      9.44     37  2018
 9 Adams   6.76      4.53      8.99     36  2019
10 Adams   4.76      2.82      6.70     24  2020
# ℹ 758 more rows
```

Creating a `tibble`

If we wanted to create a `tibble` ("fancy" data frame), we can using the `tibble()` function on a data frame.

```
tbl_mtcars <- tibble(mtcars)
tbl_mtcars
```

```
# A tibble: 32 × 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21        6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21        6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8      4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4      6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7      8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1      6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3      8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4      4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8      4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2      6  168.   123  3.92  3.44  18.3     1     0     4     4
# … 22 more rows
```

Note don't necessarily need to use `head()` with tibbles, as they conveniently print a portion of the data.

Summary of tibbles and data frames

We generally recommend using tibbles, but you are likely to run into lots of data frames with your work.

Most functions work for both so you don’t need to worry about it much!

It can be helpful to convert data frames to tibbles though just to see more about the data more easily. The `tibble()` function helps us do that.

Data frames vs tibbles - watch out for rownames

Note that this conversion can remove row names - which some data frames have. For example, `mtcars` (part of R) has row names. In this case we would want to make the rownames a new column first before making into a tibble.

```
head(mtcars, n = 2)
```

```
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4       21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag   21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```
head(tibble(mtcars), n = 2)
```

```
# A tibble: 2 × 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1    21     6   160   110   3.9  2.62  16.5     0     1     4     4
2    21     6   160   110   3.9  2.88  17.0     0     1     4     4
```

rownames_to_column function

There is a function that specifically helps you do that.

```
head(rownames_to_column(mtcars), n = 2)
```

```
        rowname mpg cyl disp  hp drat    wt  qsec vs am gear carb
1     Mazda RX4  21   6  160 110  3.9 2.620 16.46  0  1    4    4
2 Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```
head(tibble(rownames_to_column(mtcars)), n = 2)
```

```
# A tibble: 2 × 12
  rowname        mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <chr>        <dbl> <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
```

Data for now

Let's stick with the tibble ER data for our next lesson

```
head(er)
```

```
# A tibble: 6 × 6
  county  rate lower95cl upper95cl visits  year
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Adams   6.73     NA         9.24     29  2011
2 Adams   4.84      2.85     NA        23  2012
3 Adams   6.84      4.36      9.31     31  2013
4 Adams   3.08      1.71      4.85     15  2014
5 Adams   3.36      1.89      5.23     16  2015
6 Adams   8.85      6.12     11.6      42  2016
```

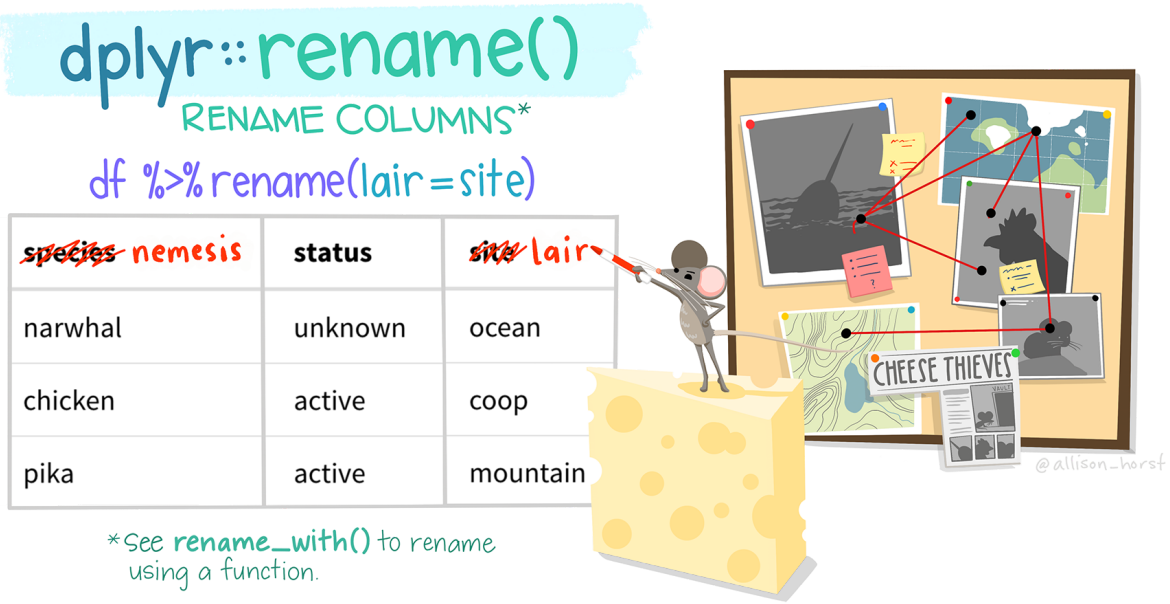
Renaming Columns

Why rename?

Renaming can:

- make it easier to work with your data
- make your column names more compatible with R
- make your column names more understandable by others

rename function



"Artwork by @allison_horst". https://allisonhorst.com/

checking names of columns, we can use the **`colnames()`** function

```
colnames(er)
```

```
[1] "county"    "rate"      "lower95cl" "upper95cl" "visits"    "year"
```

Renaming Columns of a data frame or tibble

To rename columns in `dplyr`, you can use the `rename` function.

Let's rename lower95CI to lower_95_CI_limit. Notice the new name is listed **first**, similar to how a new object is assigned on the left!

The lower95CI is the lower bound of the estimated rate of ER visits for a particular county and year.

```
# general format! not code!
{data you are creating or changing} <- rename({data you are using},
                                              {New Name} = {Old name})
```

```
renamed_er<- rename(er, lower_95_CI_limit = lower95cl)
head(renamed_er)
```

```
# A tibble: 6 × 6
  county  rate lower_95_CI_limit upper95cl visits  year
  <chr>  <dbl>             <dbl>     <dbl>  <dbl> <dbl>
1 Adams   6.73             NA         9.24     29  2011
2 Adams   4.84              2.85     NA        23  2012
3 Adams   6.84              4.36      9.31     31  2013
4 Adams   3.08              1.71      4.85     15  2014
5 Adams   3.36              1.89      5.23     16  2015
6 Adams   8.85              6.12     11.6      42  2016
```

Take Care with Column Names

When you can, avoid spaces, special punctuation, or numbers in column names, as these require special treatment to refer to them.

See https://daseh.org/resources/quotes_vs_backticks.html for more guidance.

```
 # this will cause an error
renamed_er <- rename(er, lower_95%_CI_limit = lower95cl)

# this will work
renamed_er <- rename(er, `lower_95%_CI_limit` = lower95cl)
head(renamed_er, 2)
```

```
# A tibble: 2 × 6
  county  rate `lower_95%_CI_limit` upper95cl visits  year
  <chr>  <dbl>                <dbl>     <dbl>  <dbl> <dbl>
1 Adams   6.73                   NA      9.24     29  2011
2 Adams   4.84                 2.85     NA        23  2012
```

Unusual Column Names

It’s best to avoid unusual column names where possible, as things get tricky later.

We just showed the use of ` backticks` . You may see people use quotes as well.

Other atypical column names are those with:

- spaces
- number without characters
- number starting the name
- other punctuation marks like % (besides “_” or “.” and not at the beginning)

A solution!

Rename tricky column names so that you don’t have to deal with them later!

Be careful about copy pasting code!

Curly quotes will not work!

```
# this will cause an error!
renamed_er <- rename(er, ‘lower_95%_CI_limit’ = lower95cl)

# this will work!
renamed_er <- rename(er, 'lower_95%_CI_limit' = lower95cl)
```

Also true for double quotes

```
 # this will cause an error!
renamed_er <- rename(er, “lower_95%_CI_limit” = lower95cl)

# this will work!
renamed_er <- rename(er, "lower_95%_CI_limit" = lower95cl)
```

Rename multiple columns

A comma can separate different column names to change.

```
renamed_er <- rename(er,
                     lower_95perc_CI_limit = lower95cl,
                     upper_95perc_CI_limit = upper95cl)
head(renamed_er, 3)
```

```
# A tibble: 3 × 6
  county  rate lower_95perc_CI_limit upper_95perc_CI_limit visits  year
  <chr>  <dbl>                 <dbl>                 <dbl>  <dbl> <dbl>
1 Adams   6.73                NA                     9.24     29  2011
2 Adams   4.84                 2.85                 NA        23  2012
3 Adams   6.84                 4.36                  9.31     31  2013
```

Renaming all columns of a data frame: dplyr

To rename all columns you use the `rename_with()`. In this case we will use `toupper()` to make all letters upper case. Could also use `tolower()` function.

```
er_upper <- rename_with(er, toupper)
head(er_upper, 3)
```

```
# A tibble: 3 × 6
  COUNTY  RATE LOWER95CL UPPER95CL VISITS  YEAR
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Adams   6.73     NA         9.24     29  2011
2 Adams   4.84      2.85     NA        23  2012
3 Adams   6.84      4.36      9.31     31  2013
```

```
rename_with(er_upper, tolower)
```

```
# A tibble: 768 × 6
   county  rate lower95cl upper95cl visits  year
   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Adams   6.73     NA         9.24     29  2011
 2 Adams   4.84      2.85     NA        23  2012
 3 Adams   6.84      4.36      9.31     31  2013
 4 Adams   3.08      1.71      4.85     15  2014
 5 Adams   3.36      1.89      5.23     16  2015
 6 Adams   8.85      6.12     11.6      42  2016
 7 Adams   6.63      4.29      8.98     32  2017
 8 Adams   7.11      4.77      9.44     37  2018
 9 Adams   6.76      4.53      8.99     36  2019
10 Adams   4.76      2.82      6.70     24  2020
# ℹ 758 more rows
```

janitor package

If you need to do lots of naming fixes - look into the janitor package!

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

janitor `clean_names`

The `clean_names` function can intuit what fixes you might need.

The yearly_co2_emissions dataset contains estimated CO2 emissions for 265 countries between the years 1751 and 2014.

```
yearly_co2 <-
  read_csv("https://daseh.org/data/Yearly_CO2_Emissions_1000_tonnes.csv")
```

```
Rows: 192 Columns: 265
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr   (1): country
dbl (264): 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761,...

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

yearly_co2 column names

```
head(yearly_co2, n = 2)
```

```
# A tibble: 2 × 265
  country  `1751` `1752` `1753` `1754` `1755` `1756` `1757` `1758` `1759` `1760`
  <chr>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Afghani…     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
2 Albania      NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
# ℹ 254 more variables: `1761` <dbl>, `1762` <dbl>, `1763` <dbl>, `1764` <dbl>,
#   `1765` <dbl>, `1766` <dbl>, `1767` <dbl>, `1768` <dbl>, `1769` <dbl>,
#   `1770` <dbl>, `1771` <dbl>, `1772` <dbl>, `1773` <dbl>, `1774` <dbl>,
#   `1775` <dbl>, `1776` <dbl>, `1777` <dbl>, `1778` <dbl>, `1779` <dbl>,
#   `1780` <dbl>, `1781` <dbl>, `1782` <dbl>, `1783` <dbl>, `1784` <dbl>,
#   `1785` <dbl>, `1786` <dbl>, `1787` <dbl>, `1788` <dbl>, `1789` <dbl>,
#   `1790` <dbl>, `1791` <dbl>, `1792` <dbl>, `1793` <dbl>, `1794` <dbl>, …
```

janitor `clean_names` can intuit fixes

The `clean_names` function can intuit what fixes you might need. Here it make sure year names aren't just a number, so that the colnames don't need ticks or quotes to be used.

```
clean_names(yearly_co2)
```

```
# A tibble: 192 × 265
   country     x1751 x1752 x1753 x1754 x1755 x1756 x1757 x1758 x1759 x1760 x1761
   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Afghanistan    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 2 Albania        NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 3 Algeria        NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 4 Andorra        NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 5 Angola         NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 6 Antigua an…    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 7 Argentina      NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 8 Armenia        NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
 9 Australia      NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
10 Austria        NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
# ℹ 182 more rows
# ℹ 253 more variables: x1762 <dbl>, x1763 <dbl>, x1764 <dbl>, x1765 <dbl>,
#   x1766 <dbl>, x1767 <dbl>, x1768 <dbl>, x1769 <dbl>, x1770 <dbl>,
#   x1771 <dbl>, x1772 <dbl>, x1773 <dbl>, x1774 <dbl>, x1775 <dbl>,
#   x1776 <dbl>, x1777 <dbl>, x1778 <dbl>, x1779 <dbl>, x1780 <dbl>,
#   x1781 <dbl>, x1782 <dbl>, x1783 <dbl>, x1784 <dbl>, x1785 <dbl>,
#   x1786 <dbl>, x1787 <dbl>, x1788 <dbl>, x1789 <dbl>, x1790 <dbl>, …
```

more of clean_names

`clean_names` can also get rid of spaces and replace them with _.

```
test <- tibble(`col 1` = c(1,2,3), `col 2` = c(2,3,4))
test
```

```
# A tibble: 3 × 2
  `col 1` `col 2`
    <dbl>   <dbl>
1       1       2
2       2       3
3       3       4
```

```
clean_names(test)
```

```
# A tibble: 3 × 2
  col_1 col_2
  <dbl> <dbl>
1     1     2
2     2     3
3     3     4
```

GUT CHECK: Which of the following would work well with a column called `counties_of_seattle_with_population_over_10,000`?

A. Renaming it using `rename` function to something simpler like `seattle_counties_over_10thous`.

B. Keeping it as is and use backticks around the column name when you use it.

C. Keeping it as is and use quotes around the column name when you use it.

Summary

- data frames are simpler version of a data table
- tibbles are fancier `tidyverse` version
- tibbles are made with `tibble()`
- if your original data has rownames, you need to use `rownames_to_column` before converting to tibble
- the `rename()` function of `dplyr` can help you rename columns
- avoid using punctuation (except underscores), spaces, and numbers (to start or alone) in column names
- if you must do a nonstandard column name - typically use backticks around it. See https://daseh.org/resources/quotes_vs_backticks.html.
- avoid copy and pasting code from other sources - quotation marks will change!
- check out `janitor` if you need to make lots of column name changes

Lab Part 1

- Class Website
- Lab
- Day 3 Cheatsheet
- Posit's `dplyr` Cheatsheet

Subsetting Columns

Why Subset?

Subsetting involves grabbing specific parts of your data to:

- Produce a smaller dataset
- Examine specific subsets of your data
- Use a particular part of the data for a specific analysis/visualization

Be cautious about removing columns/variables as you might find they are useful later.

You should be guided by your questions of interest.

Let's get our data again

We'll work with the CO heat-related ER visits dataset again.

This time lets also make it a smaller subset so it is easier for us to see the full dataset as we work through examples.

```
# er<-read_csv("https://daseh.org/data/CO_ER_heat_visits.csv")
set.seed(1234)
er_30 <-slice_sample(er, n = 30)
```

Subset columns of a data frame - `tidyverse` way:

To grab a vector version (or "pull" out) the `year` column the `tidyverse` way we can use the `pull` function:

```
pull(er_30, year)
```

```
 [1] 2018 2015 2021 2019 2014 2012 2017 2016 2012 2012 2017 2016 2020 2014 2020
[16] 2014 2011 2022 2018 2013 2017 2021 2015 2020 2012 2016 2013 2014 2017 2022
```

Subset columns of a data frame: dplyr

The `select` command from `dplyr` allows you to subset (still a `tibble`!)

```
select(er_30, year)
```

```
# A tibble: 30 × 1
    year
   <dbl>
 1  2018
 2  2015
 3  2021
 4  2019
 5  2014
 6  2012
 7  2017
 8  2016
 9  2012
10  2012
# … 20 more rows
```

GUT CHECK: What function would be useful for getting a vector version of a column?

A. `pull()`

B. `select()`

Select multiple columns

We can use `select` to select for multiple columns.

```
select(er_30, year, rate, county)
```

```
# A tibble: 30 × 3
    year  rate county
   <dbl> <dbl> <chr>
 1  2018  NA   Garfield
 2  2015  NA   Chaffee
 3  2021  15.9 Pueblo
 4  2019   0   Rio Grande
 5  2014   0   Lake
 6  2012  NA   Chaffee
 7  2017  NA   Chaffee
 8  2016   0   Teller
 9  2012  NA   Prowers
10  2012   0   Hinsdale
# ℹ 20 more rows
```

Select columns of a data frame: dplyr

The `select` command from `dplyr` allows you to subset columns matching patterns:

```
head(er_30, 2)
```

```
# A tibble: 2 × 6
  county    rate lower95cl upper95cl visits  year
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Garfield    NA        NA        NA     NA  2018
2 Chaffee     NA        NA        NA     NA  2015
```

```
select(er_30, ends_with("cl"))
```

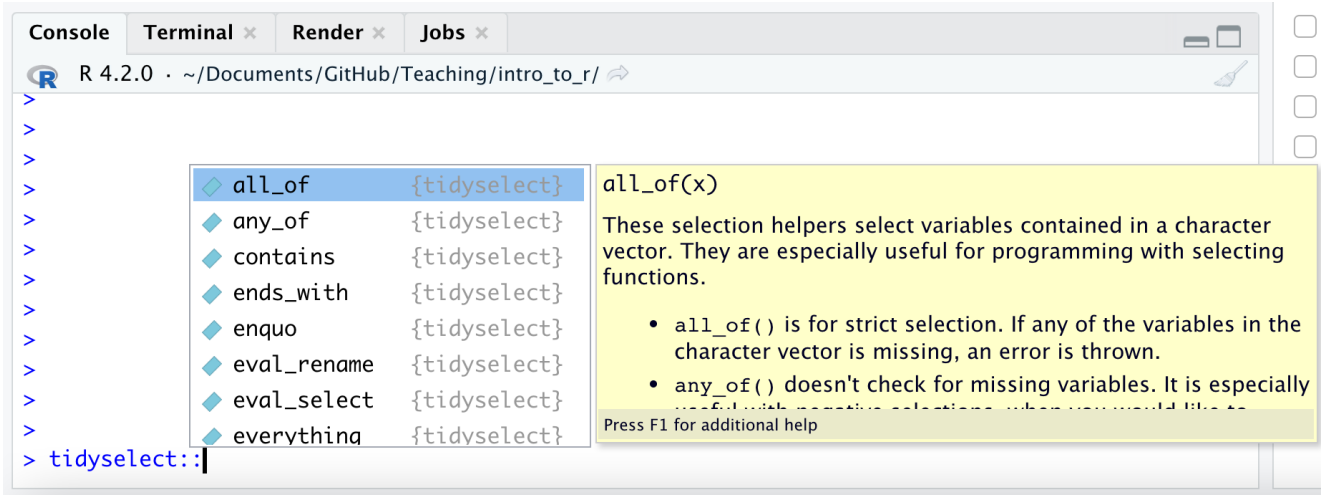
```
# A tibble: 30 × 2
   lower95cl upper95cl
       <dbl>     <dbl>
 1     NA         NA
 2     NA         NA
 3      9.54      22.2
 4      0          0
 5      0          0
 6     NA         NA
 7     NA         NA
 8      0          0
 9     NA         NA
10      0          0
# ℹ 20 more rows
```

See the Select "helpers"

Here are a few:

```
last_col()
starts_with()
ends_with()
contains()
```

Type `tidyselect::` in the **console** and see what RStudio suggests:



Combining tidyselect helpers with regular selection

```
head(er_30, 2)
```

```
# A tibble: 2 × 6
  county    rate lower95cl upper95cl visits  year
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Garfield    NA        NA        NA     NA  2018
2 Chaffee     NA        NA        NA     NA  2015
```

```
select(er_30, ends_with("cl"), year)
```

```
# A tibble: 30 × 3
   lower95cl upper95cl  year
       <dbl>     <dbl> <dbl>
 1     NA         NA    2018
 2     NA         NA    2015
 3      9.54      22.2  2021
 4      0          0    2019
 5      0          0    2014
 6     NA         NA    2012
 7     NA         NA    2017
 8      0          0    2016
 9     NA         NA    2012
10      0          0    2012
# ℹ 20 more rows
```

Multiple tidyselect functions

Follows OR logic.

```
select(er_30, ends_with("cl"), starts_with("r"))
```

```
# A tibble: 30 × 3
   lower95cl upper95cl  rate
       <dbl>     <dbl> <dbl>
 1     NA         NA    NA
 2     NA         NA    NA
 3      9.54      22.2  15.9
 4      0          0     0
 5      0          0     0
 6     NA         NA    NA
 7     NA         NA    NA
 8      0          0     0
 9     NA         NA    NA
10      0          0     0
# ℹ 20 more rows
```

The `where()` function can help select columns of a specific class

`is.character()` and `is.numeric()` are often the most helpful

```
head(er_30, 2)
```

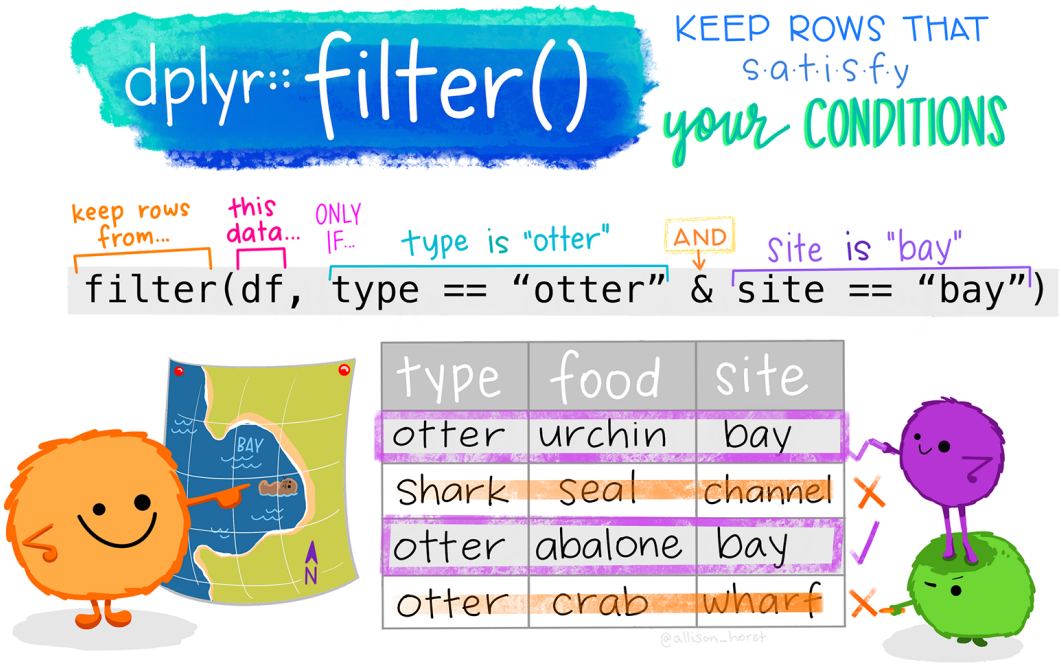
```
# A tibble: 2 × 6
  county    rate lower95cl upper95cl visits  year
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Garfield    NA        NA        NA     NA  2018
2 Chaffee     NA        NA        NA     NA  2015
```

```
select(er_30, where(is.numeric))
```

```
# A tibble: 30 × 5
    rate lower95cl upper95cl visits  year
   <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1  NA       NA         NA       NA  2018
 2  NA       NA         NA       NA  2015
 3  15.9      9.54      22.2     26  2021
 4   0        0          0        0  2019
 5   0        0          0        0  2014
 6  NA       NA         NA       NA  2012
 7  NA       NA         NA       NA  2017
 8   0        0          0        0  2016
 9  NA       NA         NA       NA  2012
10   0        0          0        0  2012
# … 20 more rows
```

Subsetting Rows

`filter` function



"Artwork by @allison_horst". https://allisonhorst.com/

Subset rows of a data frame: dplyr

The command in `dplyr` for subsetting rows is `filter`.

```
filter(er_30, year > 2020)
```

```
# A tibble: 4 × 6
  county   rate lower95cl upper95cl visits  year
  <chr>   <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Pueblo   15.9      9.54      22.2     26  2021
2 Otero    NA       NA         NA       NA  2022
3 Mesa     12.0      7.12      18.2     19  2021
4 Chaffee   0         0          0        0  2022
```

Subset rows of a data frame: dplyr

You can have multiple logical conditions using the following:

- `==` : equals to
- `!=`: not equal to (`!` : not/negation)
- `>` / `<`: greater than / less than
- `>=` or `<=`: greater than or equal to / less than or equal to
- `&` : AND
- `|` : OR

Common error for filter

If you try to filter for a column that does not exist it will not work:

- misspelled column name
- column that was already removed

Subset rows of a data frame: dplyr

You can filter by two conditions using & or commas (must meet both conditions):

```
filter(er_30, rate > 9, year == 2012)
```

```
filter(er_30, rate > 9 & year == 2012) # same result
```

```
# A tibble: 0 × 6
# ℹ 6 variables: county <chr>, rate <dbl>, lower95cl <dbl>, upper95cl <dbl>,
#   visits <dbl>, year <dbl>
```

Subset rows of a data frame: dplyr

If you want OR statements (meaning the data can meet either condition does not need to meet both), you need to use | between conditions:

```
filter(er_30, rate > 9 | year == 2012)
```

```
# A tibble: 6 × 6
  county    rate lower95cl upper95cl visits  year
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Pueblo    15.9      9.54      22.2     26  2021
2 Chaffee   NA       NA         NA       NA  2012
3 Prowers   NA       NA         NA       NA  2012
4 Hinsdale   0        0          0        0  2012
5 Mesa      12.0      7.12      18.2     19  2021
6 Phillips   0        0          0        0  2012
```

Subset rows of a data frame: dplyr

The `%in%` operator can be used find values from a pre-made list (using `c()`) for a **single column** at a time.

```
filter(er_30,  county %in% c("Denver","Larimer","Pueblo"))
```

```
# A tibble: 3 × 6
  county   rate lower95cl upper95cl visits  year
  <chr>   <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Pueblo  15.9       9.54     22.2      26  2021
2 Denver   2.95      1.75      4.46     19  2013
3 Larimer  5.49      3.16      8.45     17  2014
```

```
filter(er_30,  county == "Denver"| county == "Larimer"| county == "Pueblo") #equivalent
```

```
# A tibble: 3 × 6
  county   rate lower95cl upper95cl visits  year
  <chr>   <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Pueblo  15.9       9.54     22.2      26  2021
2 Denver   2.95      1.75      4.46     19  2013
3 Larimer  5.49      3.16      8.45     17  2014
```

Subset rows of a data frame: dplyr

The `%in%` operator can be used find values from a pre-made list (using `c()`) for a **single column** at a time with different columns.

```
filter(er_30, year %in% c(2013,2021), county %in% c("Denver","Pueblo"))
```

```
# A tibble: 2 × 6
  county  rate lower95cl upper95cl visits  year
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Pueblo 15.9       9.54     22.2      26  2021
2 Denver  2.95      1.75      4.46     19  2013
```

Be careful with column names and `filter`

This will not work the way you might expect! Best to stick with nothing but the column name if it is a typical name.

```
filter(er_30, "year" > 2014)
```

```
# A tibble: 30 × 6
   county      rate lower95cl upper95cl visits  year
   <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Garfield    NA       NA         NA       NA  2018
 2 Chaffee     NA       NA         NA       NA  2015
 3 Pueblo      15.9      9.54      22.2     26  2021
 4 Rio Grande   0        0          0        0  2019
 5 Lake         0        0          0        0  2014
 6 Chaffee     NA       NA         NA       NA  2012
 7 Chaffee     NA       NA         NA       NA  2017
 8 Teller       0        0          0        0  2016
 9 Prowers     NA       NA         NA       NA  2012
10 Hinsdale     0        0          0        0  2012
# ℹ 20 more rows
```

Don't use quotes for atypical names

Atypical names are those with punctuation, spaces, start with a number, or are just a number.

```
er_30_rename <- rename(er_30, `year!` = year)

filter(er_30_rename, "year!" > 2013) # will not work correctly
```

```
# A tibble: 30 × 6
   county      rate lower95cl upper95cl visits `year!`
   <chr>      <dbl>     <dbl>     <dbl>  <dbl>   <dbl>
 1 Garfield    NA       NA         NA       NA    2018
 2 Chaffee     NA       NA         NA       NA    2015
 3 Pueblo      15.9      9.54      22.2     26    2021
 4 Rio Grande   0        0          0        0    2019
 5 Lake         0        0          0        0    2014
 6 Chaffee     NA       NA         NA       NA    2012
 7 Chaffee     NA       NA         NA       NA    2017
 8 Teller       0        0          0        0    2016
 9 Prowers     NA       NA         NA       NA    2012
10 Hinsdale     0        0          0        0    2012
# ℹ 20 more rows
```

Be careful with column names and `filter`

Using backticks works!

```
filter(er_30_rename, `year!` > 2013)
```

```
# A tibble: 23 × 6
   county      rate lower95cl upper95cl visits `year!`
   <chr>      <dbl>     <dbl>     <dbl>  <dbl>   <dbl>
 1 Garfield   NA        NA        NA        NA    2018
 2 Chaffee    NA        NA        NA        NA    2015
 3 Pueblo     15.9       9.54     22.2      26    2021
 4 Rio Grande  0         0         0         0    2019
 5 Lake        0         0         0         0    2014
 6 Chaffee    NA        NA        NA        NA    2017
 7 Teller      0         0         0         0    2016
 8 Boulder     3.51      1.77      5.83     12    2017
 9 Fremont    NA        NA        NA        NA    2016
10 Kiowa      NA        NA        NA        NA    2020
# ℹ 13 more rows
```

Be careful with column names and `filter`

```
filter(er_30, "county" == "Denver") # this will not work
```

```
# A tibble: 0 × 6
# ℹ 6 variables: county <chr>, rate <dbl>, lower95cl <dbl>, upper95cl <dbl>,
#   visits <dbl>, year <dbl>
```

Be careful with column names and `filter`

```
filter(er_30, county == "Denver")# this works!
```

```
# A tibble: 1 × 6
  county  rate lower95cl upper95cl visits  year
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Denver  2.95      1.75      4.46     19  2013
```

`filter()` is tricky

Try not use anything special for the column names in `filter()`. This is why it is good to not use atypical column names. Then you can just use the column name!

Always good to check each step!

Did the filter work the way you expected? Did the dimensions change?

https://media.giphy.com/media/5b5OU7aUekfdSAER5I/giphy.gif

GUT CHECK: If we want to keep just rows that meet either of two conditions (not both), what code should we use?

A. `filter()` with `|`

B. `filter()` with `&`

Summary

- `pull()` to get values out of a data frame/tibble
- `select()` is the `tidyverse` way to get a tibble with only certain columns
- you can `select()` based on patterns in the column names
- you can also `select()` based on column class with the `where()` function
- you can combine multiple tidyselect functions together like `select(starts_with("C"), ends_with("state"))`
- you can combine multiple patterns with the `c()` function like `select(starts_with(c("A", "C")))` (see extra slides at the end for more info!)
- `filter()` can be used to filter out rows based on logical conditions
- avoid using quotes when referring to column names with `filter()`

Summary Continued

- `==` is the same as equivalent to
- `&` means both conditions must be met to remain after `filter()`
- `|` means either conditions needs to be met to remain after `filter()`

Lab Part 2

- Class Website
- Lab
- Day 3 Cheatsheet
- Posit's `dplyr` Cheatsheet

Get the data

```
#er <- read_csv("https://daseh.org/data/CO_ER_heat_visits.csv")
set.seed(1234)
er_30 <-slice_sample(er, n = 30)
```

Combining `filter` and `select`

You can combine `filter` and `select` to subset the rows and columns, respectively, of a data frame:

```
select(filter(er_30, year > 2012), county)
```

```
# A tibble: 25 × 1
   county
   <chr>
 1 Garfield
 2 Chaffee
 3 Pueblo
 4 Rio Grande
 5 Lake
 6 Chaffee
 7 Teller
 8 Boulder
 9 Fremont
10 Kiowa
# ℹ 15 more rows
```

Nesting

In R, the common way to perform multiple operations is to wrap functions around each other in a "nested" form.

```
head(select(er_30, year, county), 2)
```

```
# A tibble: 2 × 2
   year county
  <dbl> <chr>
1  2018 Garfield
2  2015 Chaffee
```

Nesting can get confusing looking

```
select(filter(er_30, year > 2000 & county == "Denver"), year, rate)
```

```
# A tibble: 1 × 2
   year  rate
  <dbl> <dbl>
1  2013  2.95
```

Assigning Temporary Objects

One can also create temporary objects and reassign them:

```
er_30_Den <- filter(er_30, year > 2000 & county == "Denver")
er_30_Den <- select(er_30_Den, year, rate)

head(er_30_Den)

# A tibble: 1 × 2
   year  rate
  <dbl> <dbl>
1  2013  2.95
```

Using the **`pipe`** (comes with **`dplyr`**):

The pipe `|>` makes this much more readable. It reads left side "pipes" into right side. Pipe `er_30` into `filter`, then pipe that into `select`:

```
er_30 |> filter(year > 2000 & county == "Denver") |> select(year, rate)
```

```
# A tibble: 1 × 2
   year  rate
  <dbl> <dbl>
1  2013  2.95
```

Alternative Pipes

There are multiple ways to write a pipe and you might also see this (they work the same!):

```
%>%  - called the tydiverse pipe
```

RStudio `CMD/Ctrl + Shift + M` shortcut.
We find that people often mix up `%>%` and `%in%`.

```
|> - called the base pipe
```

Follow this link to add a shortcut

Adding/Removing Columns

Adding columns to a data frame: dplyr (`tidyverse` way)

The `mutate` function in `dplyr` allows you to add or modify columns of a data frame.

```
# General format - Not the code!
{data object to update} <- mutate({data to use},
                          {new variable name} = {new variable source})

er_30 <- mutate(er_30, newcol = rate * 2)
head(er_30, 4)

# A tibble: 4 × 7
  county      rate lower95cl upper95cl visits  year newcol
  <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield    NA       NA         NA       NA  2018   NA
2 Chaffee     NA       NA         NA       NA  2015   NA
3 Pueblo      15.9      9.54      22.2     26  2021   31.8
4 Rio Grande   0        0          0        0  2019    0
```

Use mutate to modify existing columns

The `mutate` function in `dplyr` allows you to add or modify columns of a data frame.

```
# General format - Not the code!
{data object to update} <- mutate({data to use},
                          {variable name to change} = {variable modification})
```

```
er_30 <- mutate(er_30, newcol = newcol / 2)
head(er_30, 4)
```

```
# A tibble: 4 × 7
  county      rate lower95cl upper95cl visits  year newcol
  <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield    NA       NA         NA       NA  2018   NA
2 Chaffee     NA       NA         NA       NA  2015   NA
3 Pueblo      15.9      9.54      22.2     26  2021   15.9
4 Rio Grande   0        0          0        0  2019    0
```

You can pipe data into mutate

```
er_30 <- er_30 |> mutate(newcol = newcol / 2)
head(er_30,4)
```

```
# A tibble: 4 × 7
  county      rate lower95cl upper95cl visits  year newcol
  <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield    NA       NA         NA       NA  2018  NA
2 Chaffee     NA       NA         NA       NA  2015  NA
3 Pueblo      15.9      9.54      22.2     26  2021   7.95
4 Rio Grande   0        0          0        0  2019   0
```

`mutate` function

"Artwork by @allison_horst". https://allisonhorst.com/

Removing columns of a data frame: dplyr

The `NULL` method is still very common.

The `select` function can remove a column with exclamation mark (`!`) our using the minus sign (`-`):

```
select(er_30, !newcol)
```

```
# A tibble: 6 × 6
  county      rate lower95cl upper95cl visits  year
  <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Garfield    NA       NA         NA       NA  2018
2 Chaffee     NA       NA         NA       NA  2015
3 Pueblo      15.9      9.54      22.2     26  2021
4 Rio Grande   0        0          0        0  2019
5 Lake         0        0          0        0  2014
6 Chaffee     NA       NA         NA       NA  2012
```

Or, you can simply select the columns you want to keep, ignoring the ones you want to remove.

Removing columns in a data frame: dplyr

You can use `c()` to list the columns to remove.

Remove `newcol` and `year`:

```
select(er_30, !c(newcol, year))
```

```
# A tibble: 30 × 5
   county      rate lower95cl upper95cl visits
   <chr>      <dbl>     <dbl>     <dbl>  <dbl>
 1 Garfield    NA       NA         NA       NA
 2 Chaffee     NA       NA         NA       NA
 3 Pueblo      15.9      9.54      22.2     26
 4 Rio Grande   0        0          0        0
 5 Lake         0        0          0        0
 6 Chaffee     NA       NA         NA       NA
 7 Chaffee     NA       NA         NA       NA
 8 Teller       0        0          0        0
 9 Prowers     NA       NA         NA       NA
10 Hinsdale     0        0          0        0
# ℹ 20 more rows
```

Ordering columns

Ordering the columns of a data frame: dplyr

The `select` function can reorder columns.

```
head(er_30, 2)
```

```
# A tibble: 2 × 7
  county    rate lower95cl upper95cl visits  year newcol
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield    NA        NA        NA     NA  2018     NA
2 Chaffee     NA        NA        NA     NA  2015     NA
```

```
er_30 |> select(year, rate, county) |>
head(2)
```

```
# A tibble: 2 × 3
   year  rate county
  <dbl> <dbl> <chr>
1  2018    NA Garfield
2  2015    NA Chaffee
```

Ordering the columns of a data frame: dplyr

The `select` function can reorder columns. Put `newcol` first, then select the rest of columns:

```
select(er_30, newcol, everything())
```

```
# A tibble: 3 × 7
  newcol county    rate lower95cl upper95cl visits  year
   <dbl> <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1  NA    Garfield  NA       NA         NA       NA  2018
2  NA    Chaffee   NA       NA         NA       NA  2015
3   7.95 Pueblo    15.9      9.54      22.2     26  2021
```

Ordering the columns of a data frame: dplyr

In addition to `select` we can also use the `relocate()` function of dplyr to rearrange the columns for more complicated moves with the `.before` and `.after` arguments.

For example, let say we just wanted `year` to be before `rate``.

```
head(er_30, 1)
```

```
# A tibble: 1 × 7
  county    rate lower95cl upper95cl visits  year newcol
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield    NA        NA        NA     NA  2018     NA
```

```
er_year_fix <- relocate(er_30, year, .before = rate)
```

```
head(er_year_fix, 1)
```

```
# A tibble: 1 × 7
  county    year  rate lower95cl upper95cl visits newcol
  <chr>    <dbl> <dbl>     <dbl>     <dbl>  <dbl>  <dbl>
1 Garfield  2018    NA        NA        NA     NA     NA
```

Ordering rows

Ordering the rows of a data frame: dplyr

The `arrange` function can reorder rows By default, `arrange` orders in increasing order:

```
arrange(er_30, year)
```

```
# A tibble: 30 × 7
   county      rate lower95cl upper95cl visits  year newcol
   <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
 1 Saguache    0         0         0         0  2011   0
 2 Chaffee    NA        NA        NA        NA  2012  NA
 3 Prowers    NA        NA        NA        NA  2012  NA
 4 Hinsdale    0         0         0         0  2012   0
 5 Phillips    0         0         0         0  2012   0
 6 Denver      2.95      1.75      4.46     19  2013   1.47
 7 San Miguel NA        NA        NA        NA  2013  NA
 8 Lake        0         0         0         0  2014   0
 9 Delta       0         0         0         0  2014   0
10 Adams       3.08      1.71      4.85     15  2014   1.54
# ℹ 20 more rows
```

Ordering the rows of a data frame: dplyr

Use the `desc` to arrange the rows in descending order:

```
arrange(er_30, desc(year))
```

```
# A tibble: 30 × 7
   county      rate lower95cl upper95cl visits  year newcol
   <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
 1 Otero       NA       NA         NA       NA  2022  NA
 2 Chaffee      0        0          0        0  2022   0
 3 Pueblo      15.9      9.54      22.2     26  2021   7.95
 4 Mesa        12.0      7.12      18.2     19  2021   6.01
 5 Kiowa       NA       NA         NA       NA  2020  NA
 6 Park        NA       NA         NA       NA  2020  NA
 7 Rio Blanco  NA       NA         NA       NA  2020  NA
 8 Rio Grande   0        0          0        0  2019   0
 9 Garfield    NA       NA         NA       NA  2018  NA
10 Dolores      0        0          0        0  2018   0
# ℹ 20 more rows
```

Ordering the rows of a data frame: dplyr

You can combine increasing and decreasing orderings:

```
arrange(er_30, rate, desc(year)) |> head(n = 2)
```

```
# A tibble: 2 × 7
  county      rate lower95cl upper95cl visits  year newcol
  <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Chaffee        0         0         0      0  2022      0
2 Rio Grande     0         0         0      0  2019      0
```

```
arrange(er_30, desc(year), rate) |> head(n = 2)
```

```
# A tibble: 2 × 7
  county   rate lower95cl upper95cl visits  year newcol
  <chr>   <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Chaffee     0         0         0      0  2022      0
2 Otero      NA        NA        NA     NA  2022     NA
```

GUT CHECK: What function would be useful for changing a column to be a percentage instead of a ratio?

A. `filter()`

B. `select()`

C. `mutate()`

GUT CHECK: How would we interpret the following:

`er_30 |> filter(year > 2020) |> select(year, rate)`?

A. Get the `er_30` data, then filter it for rows with `year` values over 2020, then select only the `year` and `rate` columns.

B. Get the `er_30` data, then filter it rows with `year` values over 2020, then select for rows that have values for `year` and `rate`.

Summary

- `select()` and `filter()` can be combined together
- you can do sequential steps in a few ways:
  1. nesting them inside one another using parentheses `()`
  2. creating intermediate data objects in between
  3. using pipes `|>` (like "then" statements)
- `select()` and `relocate()` can be used to reorder columns
- `arrange()` can be used to reorder rows
- can remove rows with `filter()`
- can remove a column in a few ways:
  1. using `select()` with exclamation mark in front of column name(s)
  2. not selecting it (without exclamation mark)

Summary cont…

- `mutate()` can be used to create new variables or modify them

```
# General format - Not the code!
{data object to update} <- mutate({data to use},
                                  {new variable name} = {new variable source})

er_30 <- mutate(ER_30, newcol = count/2.2)
```

A note about base R:

The `$` operator is similar to `pull()`. This is the base R way to do this:

```
er_30$year
```

```
 [1] 2018 2015 2021 2019 2014 2012 2017 2016 2012 2012 2017 2016 2020 2014 2020
[16] 2014 2011 2022 2018 2013 2017 2021 2015 2020 2012 2016 2013 2014 2017 2022
```

Although it is easier (for this one task), mixing and matching the `$` operator with tidyverse functions usually doesn't work. Therefore, we want to let you know about it in case you see it, but we suggest that you try working with the tidyverse way.

Adding new columns to a data frame: base R

You can add a new column (or modify an existing one) using the `$` operator instead of `mutate`.

Just want you to be aware of this as it is very common.

```
er_30$newcol <- er_30$rate/2.2
head(er_30,3)
```

```
# A tibble: 3 × 7
  county    rate lower95cl upper95cl visits  year newcol
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield  NA       NA         NA       NA  2018  NA
2 Chaffee   NA       NA         NA       NA  2015  NA
3 Pueblo    15.9      9.54      22.2     26  2021   7.22
```

Even though `$` is easier for creating new columns, `mutate` is really powerful, so it's worth getting used to.

Lab Part 3

- Class Website
- Lab
- Day 3 Cheatsheet
- Posit's `dplyr` Cheatsheet

Image by Gerd Altmann from Pixabay

Extra Slides

Multiple patterns with tidyselect

Need to combine the patterns with the `c()` function.

```
select(er_30, ends_with("cl"), starts_with("r"))
```

```
# A tibble: 30 × 3
   lower95cl upper95cl  rate
       <dbl>     <dbl> <dbl>
 1     NA         NA    NA
 2     NA         NA    NA
 3      9.54      22.2  15.9
 4      0          0     0
 5      0          0     0
 6     NA         NA    NA
 7     NA         NA    NA
 8      0          0     0
 9     NA         NA    NA
10      0          0     0
# ℹ 20 more rows
```

```
select(er_30, starts_with(c("r", "l"))) # here we combine two patterns
```

```
# A tibble: 30 × 2
    rate lower95cl
   <dbl>     <dbl>
 1  NA       NA
 2  NA       NA
 3  15.9      9.54
 4   0        0
 5   0        0
```

Nuances about `filter()`

```
test <- tibble(A = c(1,2,3,4), B = c(1,2,3,4))
test
```

```
# A tibble: 4 × 2
      A     B
  <dbl> <dbl>
1     1     1
2     2     2
3     3     3
4     4     4
```

```
# These are technically the same but >= is easier to read
# Separating can cause issues
filter(test,  B > 2 | B==2)
```

```
# A tibble: 3 × 2
      A     B
  <dbl> <dbl>
1     2     2
2     3     3
3     4     4
```

```
filter(test, B >= 2)
```

```
# A tibble: 3 × 2
      A     B
  <dbl> <dbl>
1     2     2
```

Order of operations for `filter()`

Order can matter. Think of individual statements separately first.

```
filter(test,  A>3 | B==2 & B>2) # A is greater than 3 or B is equal to 2 AND (th

# A tibble: 1 × 2
      A     B
  <dbl> <dbl>
1     4     4

filter(test,  A>3 & B>2 | B==2) # A is greater than 3 AND B is greater than 2 le

# A tibble: 2 × 2
      A     B
  <dbl> <dbl>
1     2     2
2     4     4
```

Ordering the column names of a data frame: alphabetically

Using the base R `order()` function.

```
order(colnames(er_30))
```

```
[1] 1 3 7 2 4 5 6
```

```
er_30 |> select(order(colnames(er_30)))
```

```
# A tibble: 30 × 7
   county     lower95cl newcol  rate upper95cl visits  year
   <chr>          <dbl>  <dbl> <dbl>     <dbl>  <dbl> <dbl>
 1 Garfield       NA     NA     NA        NA       NA  2018
 2 Chaffee        NA     NA     NA        NA       NA  2015
 3 Pueblo          9.54   7.22  15.9      22.2     26  2021
 4 Rio Grande      0      0      0         0        0  2019
 5 Lake            0      0      0         0        0  2014
 6 Chaffee        NA     NA     NA        NA       NA  2012
 7 Chaffee        NA     NA     NA        NA       NA  2017
 8 Teller          0      0      0         0        0  2016
 9 Prowers        NA     NA     NA        NA       NA  2012
10 Hinsdale       0      0      0         0        0  2012
# ℹ 20 more rows
```

`which()` function

Instead of removing rows like filter, `which()` simply shows where the values occur if they pass a specific condition. We will see that this can be helpful later when we want to select and filter in more complicated ways.

```
which(select(er_30, year) == 2012)
```

```
[1]  6  9 10 25
```

```
select(er_30, year) == 2012 |> head(10)
```

```
       year
 [1,] FALSE
 [2,] FALSE
 [3,] FALSE
 [4,] FALSE
 [5,] FALSE
 [6,]  TRUE
 [7,] FALSE
 [8,] FALSE
 [9,]  TRUE
[10,]  TRUE
[11,] FALSE
[12,] FALSE
[13,] FALSE
[14,] FALSE
[15,] FALSE
[16,] FALSE
[17,] FALSE
```

Remove a column in base R

```
er_30$year <- NULL
```

Renaming Columns of a data frame: base R

We can use the `colnames` function to extract and/or directly reassign column names of `df`:

```
colnames(er_30) # just prints
```

```
[1] "county"    "rate"      "lower95cl" "upper95cl" "visits"    "year"
[7] "newcol"
```

```
colnames(er_30)[1:2] <- c("County", "Rate") # reassigns
head(er_30)
```

```
# A tibble: 6 × 7
  County      Rate lower95cl upper95cl visits  year newcol
  <chr>      <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield    NA       NA         NA       NA  2018  NA
2 Chaffee     NA       NA         NA       NA  2015  NA
3 Pueblo      15.9      9.54      22.2     26  2021   7.22
4 Rio Grande   0        0          0        0  2019   0
5 Lake         0        0          0        0  2014   0
6 Chaffee     NA       NA         NA       NA  2012  NA
```

Subset rows of a data frame with indices:

Let's select **rows** 1 and 3 from `df` using brackets:

```
er_30[ c(1, 3), ]
```

```
# A tibble: 2 × 7
  County    Rate lower95cl upper95cl visits  year newcol
  <chr>    <dbl>     <dbl>     <dbl>  <dbl> <dbl>  <dbl>
1 Garfield  NA       NA         NA       NA  2018  NA   
2 Pueblo    15.9      9.54      22.2     26  2021   7.22
```

Subset columns of a data frame:

We can also subset a data frame using the bracket `[, ]` subsetting.

For data frames and matrices (2-dimensional objects), the brackets are `[rows, columns]` subsetting. We can grab the `x` column using the index of the column or the column name ("`year`")

```
er_30[, 6]
```

```
# A tibble: 30 × 1
    year
   <dbl>
 1  2018
 2  2015
 3  2021
 4  2019
 5  2014
 6  2012
 7  2017
 8  2016
 9  2012
10  2012
# ℹ 20 more rows
```

```
er_30[, "year"]
```

```
# A tibble: 30 × 1
    year
   <dbl>
```

Subset columns of a data frame:

We can select multiple columns using multiple column names:

```
er_30[, c("County", "Rate")]
```

```
# A tibble: 30 × 2
   County      Rate
   <chr>      <dbl>
 1 Garfield    NA
 2 Chaffee     NA
 3 Pueblo      15.9
 4 Rio Grande   0
 5 Lake         0
 6 Chaffee     NA
 7 Chaffee     NA
 8 Teller       0
 9 Prowers     NA
10 Hinsdale     0
# ℹ 20 more rows
```

Data Classes

One dimensional vectors

Character and numeric

We have already covered `character` and `numeric` types.

```
class(c("tree", "cloud", "stars_&_sky"))
```

```
## [1] "character"
```

```
class(c(1, 4, 7))
```

```
## [1] "numeric"
```

Character and numeric

Character predominates if there are mixed classes.

```
class(c(1, 2, "tree"))
```

```
## [1] "character"
```

```
class(c("1", "4", "7"))
```

```
## [1] "character"
```

Logical

`logical` is a type that only has two possible elements: `TRUE` and `FALSE`

```
x <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
class(x)
```

```
## [1] "logical"
```

`logical` elements are NOT in quotes.

Why is Class important?

The class of the data tells R how to process the data.

For example, it determines whether you can make summary statistics (numbers) or if you can sort alphabetically (characters).

General Class Information

There is one useful functions associated with practically all R classes:

`as.CLASS_NAME(x)` **coerces between classes**. It turns `x` into a certain class.

Examples:

- `as.numeric()`
- `as.character()`
- `as.logical()`

Coercing: seamless transition

Sometimes coercing works great!

```
as.character(4)
```

```
## [1] "4"
```

```
as.numeric(c("1", "4", "7"))
```

```
## [1] 1 4 7
```

```
as.logical(c("TRUE", "FALSE", "FALSE"))
```

```
## [1]  TRUE FALSE FALSE
```

```
as.logical(0)
```

```
## [1] FALSE
```

Coercing: not-so-seamless

When interpretation is ambiguous, R will return `NA` (an R constant representing "**N**ot **A**vailable" i.e. missing value)

```
as.numeric(c("1", "4", "7a"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1  4 NA
```

```
as.logical(c("TRUE", "FALSE", "UNKNOWN"))
```

```
## [1]  TRUE FALSE    NA
```

GUT CHECK!

What is one reason we might want to convert data to numeric?

A. So we can take the mean

B. So the data looks better

C. So our data is correct

Number Subclasses

There are two major number subclasses or types

1. Double (1.003)
2. Integer (1)

Number Subclasses

`Double` is equivalent to `numeric`. It is a number that contains fractional values . Can be any amount of places after the decimal.

`Double` stands for double-precision

For most purposes, the difference between integers and doubles doesn't matter.

Significant figures and other formats

The `num` function of the `tibble` package can be used to change format. See here for more: https://tibble.tidyverse.org/articles/numbers.html

Factors

A `factor` is a special `character` vector where the elements have pre-defined groups or 'levels'. You can think of these as qualitative or categorical variables. Order is often important.

Examples:

- red, orange, yellow, green, blue, purple
- breakfast, lunch, dinner
- baby, toddler, child, teen, adult
- Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree
- beginner, novice, intermediate, expert

** We will learn more about factors in a later module. **

Classes Overview

Example	Class	Type	Notes
1.1	numeric	double	Default for numbers
1	integer	integer	Need to coerce to integer with `as.integer()` or use `sample()` or `seq()` to create whole numbers
“FALSE”, “Ball”	character	character	Need quotes
FALSE, TRUE	logical	logical	No quotes, 0 and 1, respectively
“Small”, “Large”	factor	factor	Need to coerce to factor with `factor()`

Special data classes

Dates

There are two most popular R classes used when working with dates and times:

- `Date` class - a calendar date
- `POSIXct` class - a calendar date with hours, minutes, seconds

Converting data from character to `Date`/`POSIXct` allows special functions date/date and time

`lubridate` (part of `tidyverse`) helps work with `Date` / `POSIXct` class objects

Creating `Date` class object

```
class("2021-06-15")
```

```
## [1] "character"
```

```
library(tidyverse) # Contains lubridate
```

```
x <- ymd("2021-06-15") # lubridate package Year Month Day
class(x)
```

```
## [1] "Date"
```

Note for function `ymd`: **y**ear **m**onth **d**ay

Dates are useful!

```
a <- ymd("2021-06-15")
b <- ymd("2021-06-18")
a - b
```

```
## Time difference of -3 days
```

The function matches the format

```
mdy("06/15/2021")
```

```
## [1] "2021-06-15"
```

```
dmy("15-June-2021")
```

```
## [1] "2021-06-15"
```

```
ymd("2021-06-15")
```

```
## [1] "2021-06-15"
```

Class conversion in a dataset

Here's a dataset on the SARS-CoV-2 viral load measured in wastewater between 2022 and 2024, collected by the collected by the National Wastewater Surveillance System.

Let's look at the *date_start* variable, the first date of the sampling window.

```
sars_ww <-
  read_csv("https://daseh.org/data/SARS-CoV-2_Wastewater_Data.csv")

# Selecting a few columns for easy viewing
sars_ww <- sars_ww |> select(town_name, date_start)
```

Class conversion in a dataset

Notice that `date_start` is `chr` class, not `date`.

```
sars_ww
```

```
## # A tibble: 2,813 × 2
##    town_name date_start
##    <chr>     <chr>
##  1 Barry     6/21/2020
##  2 Barry     6/22/2020
##  3 Barry     6/23/2020
##  4 Barry     6/24/2020
##  5 Barry     6/25/2020
##  6 Barry     6/26/2020
##  7 Barry     6/27/2020
##  8 Barry     6/28/2020
##  9 Barry     6/29/2020
## 10 Barry     6/30/2020
## # ℹ 2,803 more rows
```

Class conversion in with a dataset

We would need to use `mutate()` to help us modify that column.

```
sars_ww |>
  mutate(date_start_fixed = mdy(date_start))
```

```
## # A tibble: 2,813 × 3
##    town_name date_start date_start_fixed
##    <chr>     <chr>      <date>
##  1 Barry     6/21/2020  2020-06-21
##  2 Barry     6/22/2020  2020-06-22
##  3 Barry     6/23/2020  2020-06-23
##  4 Barry     6/24/2020  2020-06-24
##  5 Barry     6/25/2020  2020-06-25
##  6 Barry     6/26/2020  2020-06-26
##  7 Barry     6/27/2020  2020-06-27
##  8 Barry     6/28/2020  2020-06-28
##  9 Barry     6/29/2020  2020-06-29
## 10 Barry     6/30/2020  2020-06-30
## # ℹ 2,803 more rows
```

Other data classes

Two-dimensional data classes

Two-dimensional classes are those we would often use to store data read from a file

- a data frame (`data.frame` or `tibble` class)
- a matrix (`matrix` class)
  - also composed of rows and columns
  - unlike `data.frame` or `tibble`, the entire matrix is composed of one R class
  - for example: all entries are `numeric`, or all entries are `character`

Lists

- One other data type that is the most generic are `lists`.
- Can hold vectors, strings, matrices, models, list of other list!
- Lists are used when you need to do something repeatedly across lots of data - for example wrangling several similar files at once
- Lists are a bit more advanced but you may encounter them when you work with others or look up solutions

Making Lists

- Can be created using `list()`

```
mylist <- list(c("A", "b", "c"), c(1, 2, 3))
mylist
```

```
## [[1]]
## [1] "A" "b" "c"
##
## [[2]]
## [1] 1 2 3
```

```
class(mylist)
```

```
## [1] "list"
```

Summary

- coerce between classes using `as.numeric()` or `as.character()`
- data frames, tibbles, matrices, and lists are all classes of objects
- lists can contain multiples of any other class of data including lists!
- calendar dates can be represented with the `Date` class using `ymd()`, `mdy()` functions from `lubridate` package
- can then easily subtract `Date` or `POSIXct` class variables or pull out aspects like year

Lab

- Class Website
- Lab
- Day 4 Cheatsheet

For more advanced learning: see the extra slides in this file!

Image by Gerd Altmann from Pixabay

Extra Slides

Matrices

`as.matrix()` creates a matrix from a data frame or tibble (where all values are the same class).

`matrix()` creates a matrix from scratch.

```
matrix(1:6, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

More about Lists

List elements can be named

```
mylist_named <- list(
  letters = c("A", "b", "c"),
  numbers = c(1, 2, 3),
  one_matrix = matrix(1:4, ncol = 2)
)
mylist_named
```

```
## $letters
## [1] "A" "b" "c"
##
## $numbers
## [1] 1 2 3
##
## $one_matrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Some useful functions from `lubridate` to manipulate `Date` objects

```
x <- ymd(c("2021-06-15", "2021-07-15"))
x
```

```
## [1] "2021-06-15" "2021-07-15"
```

```
day(x) # see also: month(x) , year(x)
```

```
## [1] 15 15
```

```
x + days(10)
```

```
## [1] "2021-06-25" "2021-07-25"
```

```
x + months(1) + days(10)
```

```
## [1] "2021-07-25" "2021-08-25"
```

```
wday(x, label = TRUE)
```

```
## [1] Tue Thu
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Some useful functions from `lubridate` to manipulate `POSIXct` objects

```
x <- ymd_hms("2013-01-24 19:39:07")
x
```

```
## [1] "2013-01-24 19:39:07 UTC"
```

```
date(x)
```

```
## [1] "2013-01-24"
```

```
x + hours(3)
```

```
## [1] "2013-01-24 22:39:07 UTC"
```

```
floor_date(x, "1 hour") # see also: ceiling_date()
```

```
## [1] "2013-01-24 19:00:00 UTC"
```

Differences in dates

```
x1 <- ymd(c("2021-06-15"))
x2 <- ymd(c("2021-07-15"))

difftime(x2, x1, units = "weeks")

## Time difference of 4.285714 weeks

as.numeric(difftime(x2, x1, units = "weeks"))

## [1] 4.285714
```

Similar can be done with time (e.g. difference in hours).

Data Selection

Matrices

```
n <- 1:9
n
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

```
mat <- matrix(n, nrow = 3)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Vectors: data selection

To get element(s) of a vector (one-dimensional object):

- Type the name of the variable and open the rectangular brackets `[ ]`
- In the rectangular brackets, type index (/vector of indexes) of element (/elements) you want to pull. **In R, indexes start from 1** (not: 0)

```
x <- c("a", "b", "c", "d", "e", "f", "g", "h")
x
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h"
```

```
x[2]
```

```
## [1] "b"
```

```
x[c(1, 2, 100)]
```

```
## [1] "a" "b" NA
```

Matrices: data selection

Note you cannot use `dplyr` functions (like `select`) on matrices. To subset matrix rows and/or columns, use `matrix[row_index, column_index]`.

```
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
mat[1, 1] # individual entry: row 1, column 1
```

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

```
mat[1, 2] # individual entry: row 1, column 2
```

```
## [1] 4
```

```
mat[1, ] # first row
```

```
## [1] 1 4 7
```

```
mat[, 1] # first column
```

```
## [1] 1 2 3
```

```
mat[c(1, 2), c(2, 3)] # subset of original matrix: two rows and two columns
```

Lists: data selection

You can reference data from list using `$` (if elements are named) or using `[[ ]]`

```
mylist_named[[1]]
```

```
## [1] "A" "b" "c"
```

```
mylist_named[["letters"]] # works only for a list with elements' names
```

```
## [1] "A" "b" "c"
```

```
mylist_named$letters # works only for a list with elements' names
```

```
## [1] "A" "b" "c"
```

Data Cleaning

Recap on summarization

- `summary(x)`: quantile information
- `count(x)`: what unique values do you have?
  - `distinct()`: what are the distinct values?
  - `n_distinct()` with `pull()`: how many distinct values?
- `group_by()`: changes all subsequent functions
  - combine with `summarize()` to get statistics per group
  - combine with `mutate()` to add column
- `summarize()` with `n()` gives the count (NAs included)

 Day 4 Cheatsheet

Recap on data classes

- There are two types of number class objects: integer and double
- Logic class objects only have `TRUE` or `FALSE` (without quotes)
- `class()` can be used to test the class of an object x
- `as.CLASS_NAME(x)` can be used to change the class of an object x
- Factors are a special character class that has levels - more on that soon!
- tibbles show column classes!
- two dimensional object classes include: data frames, tibbles, matrices, and lists
- Dates can be handled with the `lubridate` package
- Make sure you choose the right function for the way the date is formatted!

 Day 4 Cheatsheet

Data Cleaning

In general, data cleaning is a process of investigating your data for inaccuracies, or recoding it in a way that makes it more manageable.

 MOST IMPORTANT RULE - LOOK AT YOUR DATA!

Dealing with Missing Data

Missing data types

One of the most important aspects of data cleaning is missing values.

Types of “missing” data:

- `NA` - general missing data
- `NaN` - stands for “**N**ot **a N**umber”, happens when you do 0/0.
- `Inf` and `-Inf` - Infinity, happens when you divide a positive number (or negative number) by 0.

Finding Missing data

- `is.na` - looks for `NAN` and `NA`
- `is.nan`- looks for `NAN`
- `is.infinite` - looks for Inf or -Inf

```
test <- c(0,NA, -1)
test/0
```

```
[1]  NaN   NA -Inf
```

```
test <- test/0
is.na(test)
```

```
[1]  TRUE  TRUE FALSE
```

```
is.nan(test)
```

```
[1]  TRUE FALSE FALSE
```

```
is.infinite(test)
```

```
[1] FALSE FALSE  TRUE
```

Useful checking functions

`any()` can help you check if there are any `NA` values in a vector

```
test
```

```
[1]  NaN   NA -Inf
```

```
any(is.na(test))
```

```
[1] TRUE
```

Finding NA values with count()

Check the values for your variables, are they what you expect?

`count()` is a great option because it helps you check if rare values make sense.

Let's look at the CO heat-related ER visits dataset again.

```
er <- read_csv(file =
    "https://daseh.org/data/CO_ER_heat_visits.csv")
er |> count(visits)
```

```
# A tibble: 37 × 2
   visits     n
    <dbl> <int>
 1      0   339
 2     11     2
 3     12     5
 4     13     8
 5     14     3
 6     15     2
 7     16     3
 8     17     3
 9     18     5
10     19     7
# ℹ 27 more rows
```

naniar

Sometimes you need to look at lots of data though… the `naniar` package is a good option.

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

"Artwork by @allison_horst". https://allisonhorst.com/

naniar: `pct_complete()`

This can tell you if there are missing values in the dataset.

```
pct_complete(er)
```

```
[1] 73.65451
```

Or for a particular variable:

```
er |> select(visits) |>
  pct_complete()
```

```
[1] 60.54688
```

```
er |> select(rate) |>
  pct_complete()
```

```
[1] 60.54688
```

naniar::`miss_var_summary()`

To get the percent missing (and counts) for each variable as a table, use this function:

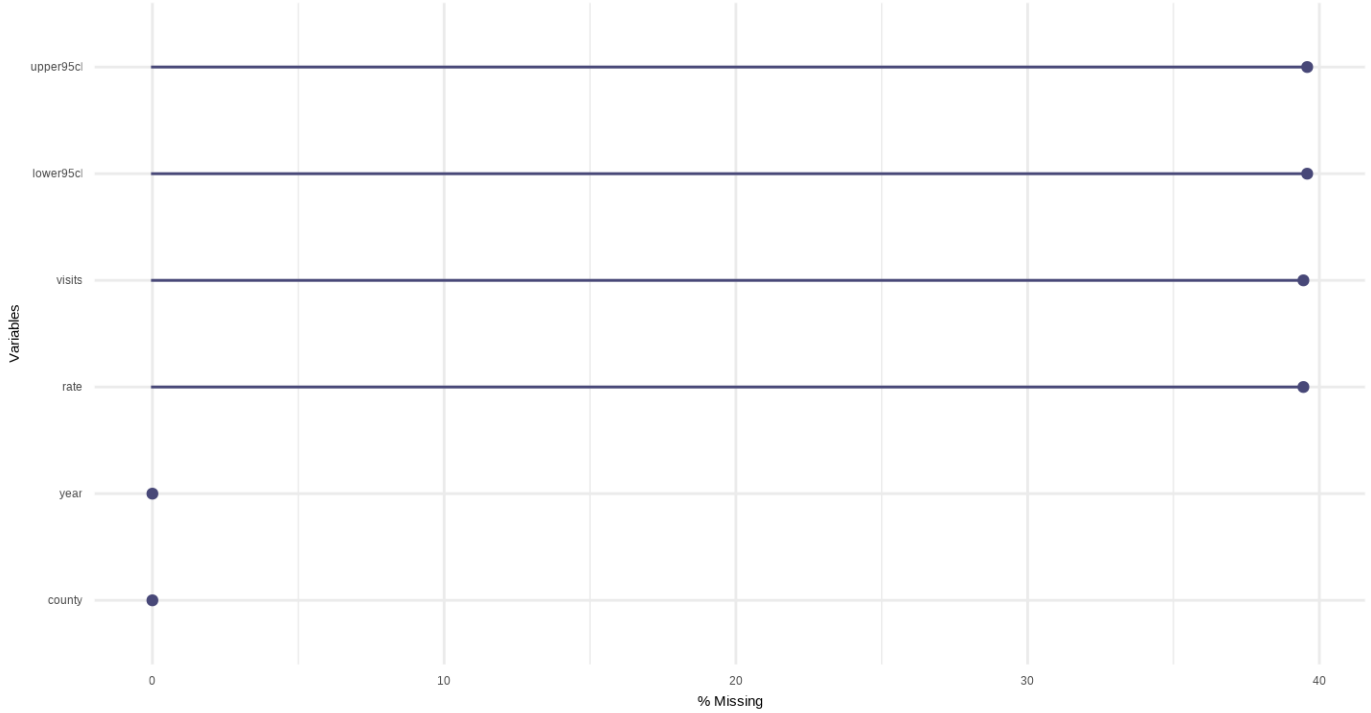
```
miss_var_summary(er)
```

```
# A tibble: 6 × 3
  variable  n_miss pct_miss
  <chr>      <int>    <num>
1 lower95cl    304     39.6
2 upper95cl    304     39.6
3 rate         303     39.5
4 visits       303     39.5
5 county         0      0
6 year           0      0
```

naniar plots

The `gg_miss_var()` function creates a nice plot about the number of missing values for each variable, (need a data frame).

```
gg_miss_var(er, show_pct = TRUE)
```



Missing Data Issues

Recall that mathematical operations with `NA` often result in `NA`s.

```
sum(c(1,2,3,NA))
```

```
[1] NA
```

```
mean(c(1,2,3,NA))
```

```
[1] NA
```

```
median(c(1,2,3,NA))
```

```
[1] NA
```

Missing Data Issues

This is also true for logical data. Recall that `TRUE` is evaluated as 1 and `FALSE` is evaluated as 0.

```
x <- c(TRUE, TRUE, TRUE, TRUE, FALSE, NA)
sum(x)
```

```
[1] NA
```

```
sum(x, na.rm = TRUE)
```

```
[1] 4
```

filter() and missing data

Be **careful** with missing data using subsetting!

`filter()` removes missing values by default. Because R can't tell for sure if an `NA` value meets the condition. To keep them need to add `is.na()` conditional.

Think about if this is OK or not - it depends on your data!

filter() and missing data

What if NA values represent values that are so low it is undetectable?

Filter will drop them from the data.

```
er |> filter(visits > 0)
```

```
# A tibble: 126 × 6
   county  rate lower95cl upper95cl visits  year
   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Adams   6.73     NA         9.24     29  2011
 2 Adams   4.84      2.85     NA        23  2012
 3 Adams   6.84      4.36      9.31     31  2013
 4 Adams   3.08      1.71      4.85     15  2014
 5 Adams   3.36      1.89      5.23     16  2015
 6 Adams   8.85      6.12     11.6      42  2016
 7 Adams   6.63      4.29      8.98     32  2017
 8 Adams   7.11      4.77      9.44     37  2018
 9 Adams   6.76      4.53      8.99     36  2019
10 Adams   4.76      2.82      6.70     24  2020
# ℹ 116 more rows
```

filter() and missing data

`is.na()` can help us keep them.

```
er |> filter(visits > 0 | is.na(visits))
```

```
# A tibble: 429 × 6
   county  rate lower95cl upper95cl visits  year
   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Adams   6.73     NA         9.24     29  2011
 2 Adams   4.84      2.85     NA        23  2012
 3 Adams   6.84      4.36      9.31     31  2013
 4 Adams   3.08      1.71      4.85     15  2014
 5 Adams   3.36      1.89      5.23     16  2015
 6 Adams   8.85      6.12     11.6      42  2016
 7 Adams   6.63      4.29      8.98     32  2017
 8 Adams   7.11      4.77      9.44     37  2018
 9 Adams   6.76      4.53      8.99     36  2019
10 Adams   4.76      2.82      6.70     24  2020
# ℹ 419 more rows
```

To remove **rows** with **NA** values for a **variable** use **drop_na( )**

A function from the `tidyr` package. (Need a data frame to start!)

Disclaimer: Don't do this unless you have thought about if dropping `NA` values makes sense based on knowing what these values mean in your data. **Also consider if you need those rows for values for other variables.**

```
dim(er)
```

```
[1] 768   6
```

```
er_drop <- er |> drop_na(lower95cl)
dim(er_drop)
```

```
[1] 464   6
```

Let's take a look

Can still have NAs for other columns

```
er_drop
```

```
# A tibble: 464 × 6
   county  rate lower95cl upper95cl visits  year
   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Adams   4.84      2.85     NA         23  2012
 2 Adams   6.84      4.36      9.31      31  2013
 3 Adams   3.08      1.71      4.85      15  2014
 4 Adams   3.36      1.89      5.23      16  2015
 5 Adams   8.85      6.12     11.6       42  2016
 6 Adams   6.63      4.29      8.98      32  2017
 7 Adams   7.11      4.77      9.44      37  2018
 8 Adams   6.76      4.53      8.99      36  2019
 9 Adams   4.76      2.82      6.70      24  2020
10 Adams   6.93      4.61      9.25      35  2021
# ℹ 454 more rows
```

To remove rows with `NA` values for a data frame use `drop_na()`

This function of the `tidyr` package drops rows with **any** missing data in **any** column when used on a df.

```
er_drop <- er |> drop_na()
er_drop
```

```
# A tibble: 463 × 6
   county  rate lower95cl upper95cl visits  year
   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
 1 Adams   6.84      4.36      9.31     31  2013
 2 Adams   3.08      1.71      4.85     15  2014
 3 Adams   3.36      1.89      5.23     16  2015
 4 Adams   8.85      6.12     11.6      42  2016
 5 Adams   6.63      4.29      8.98     32  2017
 6 Adams   7.11      4.77      9.44     37  2018
 7 Adams   6.76      4.53      8.99     36  2019
 8 Adams   4.76      2.82      6.70     24  2020
 9 Adams   6.93      4.61      9.25     35  2021
10 Adams   8.23      5.81     10.6      45  2022
# ℹ 453 more rows
```

Drop **columns** with any missing values

Use the `miss_var_which()` function from `naniar`

```
miss_var_which(er)# which columns have missing values
```

```
[1] "rate"      "lower95cl" "upper95cl" "visits"
```

Drop **columns** with any missing values

`miss_var_which` and function from `naniar` (need a data frame)

```
er_drop <- er |> select(!miss_var_which(er))
er_drop
```

```
# A tibble: 768 × 2
   county  year
   <chr>  <dbl>
 1 Adams   2011
 2 Adams   2012
 3 Adams   2013
 4 Adams   2014
 5 Adams   2015
 6 Adams   2016
 7 Adams   2017
 8 Adams   2018
 9 Adams   2019
10 Adams   2020
# ℹ 758 more rows
```

Change a value to be `NA`

Let's say we think that all 0 values should be `NA`.

Maybe we think the person who entered the CO heat-related ER visits data made a mistake in how they coded missing data.

```
er |> count(visits)
```

```
# A tibble: 37 × 2
   visits     n
    <dbl> <int>
 1      0   339
 2     11     2
 3     12     5
 4     13     8
 5     14     3
 6     15     2
 7     16     3
 8     17     3
 9     18     5
10     19     7
# ℹ 27 more rows
```

Change a value to be `NA`

The `na_if()` function of `dplyr` can be helpful for changing all 0 values to `NA`.

```
er_nozero <- er |>
  mutate(visits = na_if(visits, 0))

er_nozero |> count(visits)

# A tibble: 36 × 2
   visits     n
    <dbl> <int>
 1     11     2
 2     12     5
 3     13     8
 4     14     3
 5     15     2
 6     16     3
 7     17     3
 8     18     5
 9     19     7
10     20     2
# ℹ 26 more rows
```

Change `NA` to be a value

The `replace_na()` function (part of the `tidyr` package), can do the opposite of `na_if()`. (note that you must use numeric values as replacement - we will show how to replace with character strings soon)

```
er |>
  mutate(visits = replace_na(visits, 0)) |>
  count(visits)
```

```
# A tibble: 36 × 2
   visits     n
    <dbl> <int>
 1      0   642
 2     11     2
 3     12     5
 4     13     8
 5     14     3
 6     15     2
 7     16     3
 8     17     3
 9     18     5
10     19     7
# ℹ 26 more rows
```

Think about NA

THINK ABOUT YOUR DATA FIRST!

- Sometimes removing NA values leads to distorted math - be careful!
- Think about what your NA means for your data (are you sure ?).
  - Is an NA for values so low they could not be reported?
  - Or is it if it was too low and also if there was a different issue (like no one reported)?

Think about NA

If it is something more like a zero then you might want it included in your data like a zero instead of an NA.

Example: - survey reports NA if student has never tried cigarettes - survey reports 0 if student has tried cigarettes but did not smoke that week

 You might want to keep the NA values so that you know the original sample size.

Word of caution

- Calculating percentages will give you a different result depending on your choice to include NA values.!

This is because the denominator changes.

Word of caution - Percentages with `NA`

```
count(er, visits) |> mutate(percent = (n/(sum(n)) *100))
```

```
# A tibble: 37 × 3
   visits     n percent
    <dbl> <int>   <dbl>
 1      0   339  44.1
 2     11     2   0.260
 3     12     5   0.651
 4     13     8   1.04
 5     14     3   0.391
 6     15     2   0.260
 7     16     3   0.391
 8     17     3   0.391
 9     18     5   0.651
10     19     7   0.911
# ℹ 27 more rows
```

Word of caution - Percentages with `NA`

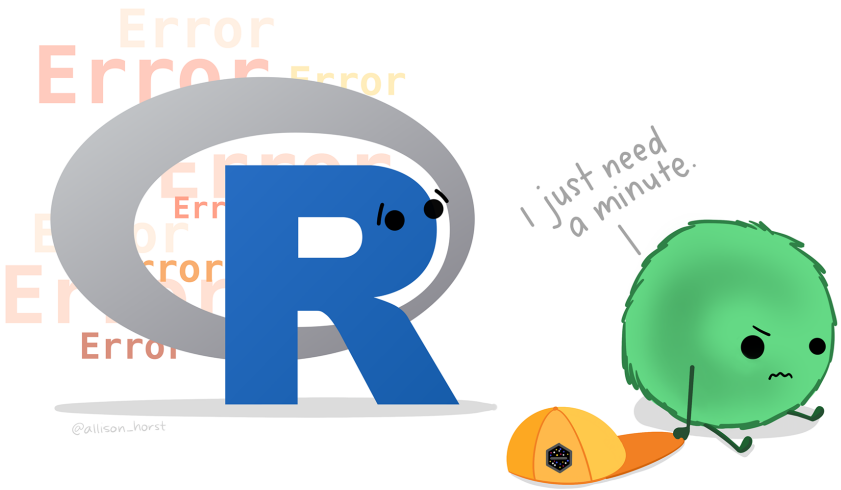
```
er |> drop_na(visits) |>
  count(visits) |> mutate(percent = (n/(sum(n)) *100))
```

```
# A tibble: 36 × 3
   visits     n percent
    <dbl> <int>   <dbl>
 1      0   339  72.9
 2     11     2   0.430
 3     12     5   1.08
 4     13     8   1.72
 5     14     3   0.645
 6     15     2   0.430
 7     16     3   0.645
 8     17     3   0.645
 9     18     5   1.08
10     19     7   1.51
# ℹ 26 more rows
```

Should you be dividing by the total count with `NA` values included?
It depends on your data and what `NA` might mean.
Pay attention to your data and your `NA` values!

Don’t forget about the common issues

- Extra or Missing commas
- Extra or Missing parentheses
- Case sensitivity
- Spelling



GUT CHECK: What function can be used to remove NA values from a full dataframe or for an individual column?

A. `drop_nulls()`

B. `drop_na()`

C. `rem_na()`

GUT CHECK: How can you keep NA values when using `filter`?

A. include `| is.na()`

B. include `& is.na()`

Summary

- `is.na()`,`any(is.na())`, `all(is.na())`,`count()`, and functions from `naniar` like `gg_miss_var()` and `miss_var_summary` can help determine if we have `NA` values
- `miss_var_which()` can help you drop columns that have any missing values.
- `filter()` automatically removes `NA` values - can't confirm or deny if condition is met (need `| is.na()` to keep them)
- `drop_na()` can help you remove `NA` values from a variable or an entire data frame
- `NA` values can change your calculation results
- think about what `NA` values represent - don't drop them if you shouldn't
- `na_if()` will make `NA` values for a particular value
- `replace_na()` will replace `NA values with a particular value

Lab Part 1

- Class Website
- Lab. Day 5 Cheatsheet

Recoding Variables

Example of Recoding

Let's upload some practice data about microplastics. Maybe we measured the level of microplastics in someone who eats a fish versus someone who is a vegetarian.

https://www.medicalnewstoday.com/articles/what-do-we-know-about-microplastics-in-food#Common-microplastics-in-food
https://www.sciencedirect.com/science/article/pii/S0045653523028072
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5708064/

```
plastics <- read_csv(file =
    "https://daseh.org/data/microplastics_in_blood.csv")
```

```
Rows: 12 Columns: 4
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (2): Foods, microplastic
dbl (2): blood_level_start_nM, blood_level_change_nM

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

microplastics data

```
plastics
```

```
# A tibble: 12 × 4
   Foods microplastic blood_level_start_nM blood_level_change_nM
   <chr> <chr>                        <dbl>                  <dbl>
 1 A     Dioxin                           1                    2.5
 2 B     Dioxin                           7                    2.5
 3 B     Other                            2                    0.5
 4 A     Bisphenol A                      3                   -0.5
 5 B     BPA                              5                    0.5
 6 B     dioxin                           8                    0.5
 7 A     bpa                              6                    2.5
 8 B     Other                            5                    3.5
 9 B     dioxin                           2                    0.5
10 A     bpa                              1                    1.5
11 B     BPA                              7                    1.5
12 B     Other                            3                    2.5
```

Oh dear…

This needs lots of recoding.

```
plastics |>
  count(microplastic)
```

```
# A tibble: 6 × 2
  microplastic     n
  <chr>        <int>
1 BPA              2
2 Bisphenol A      1
3 Dioxin           2
4 Other            3
5 bpa              2
6 dioxin           2
```

`dplyr` can help!

Using Excel to find all of the different ways `microplastic` has been coded, could be hectic! In `dplyr` you can use the `case_when` function.

(need `mutate` here too!)

Or you can use **case_when( )**

Need quotes for conditions and new values!

```
plastics |>
  mutate(microplastic_recoded = case_when(
                                microplastic == "Bisphenol A" ~ "BPA",
                                microplastic == "bpa" ~ "BPA",
                                microplastic == "dioxin" ~"Dioxin"))  |>
  count(microplastic, microplastic_recoded)

# A tibble: 6 × 3
  microplastic microplastic_recoded     n
  <chr>        <chr>                <int>
1 BPA          <NA>                     2
2 Bisphenol A  BPA                      1
3 Dioxin       <NA>                     2
4 Other        <NA>                     3
5 bpa          BPA                      2
6 dioxin       Dioxin                   2
```

What happened?

We seem to have `NA` values!

We didn't specify what happens to values that were already `Other` or `Dioxin` or `BPA`.

```
plastics |>
  mutate(microplastic_recoded = case_when(
                                microplastic == "Bisphenol A" ~ "BPA",
                                microplastic == "bpa" ~ "BPA",
                                microplastic == "dioxin" ~"Dioxin"))  |>
  count(microplastic, microplastic_recoded)
```

`case_when()` drops unspecified values

Note that automatically values not reassigned explicitly by `case_when()` will be `NA` unless otherwise specified.

```
# General Format - this is not code!
{data_input} |>
  mutate({variable_to_fix} = case_when({Variable_fixing}
             /some condition/ ~ {value_for_con},
                     .default = {value_for_not_meeting_condition}) # need this to avoid NAs
```

{value_for_not_meeting_condition} could be something new or it can be the original values of the column

case_when with .default = original variable name

```
plastics |>
  mutate(microplastic_recoded = case_when(
                                microplastic == "Bisphenol A" ~ "BPA",
                                microplastic == "bpa" ~ "BPA",
                                microplastic =="dioxin" ~"Dioxin",
                                .default = microplastic)) |>
  count(microplastic_recoded)

# A tibble: 3 × 2
  microplastic_recoded     n
  <chr>                <int>
1 BPA                      5
2 Dioxin                   4
3 Other                    3
```

Typically it is good practice to include the .default statement

You never know if you might be missing something - and if a value already was an NA it will stay that way.

```
plastics |>
  mutate(microplastic_recoded = case_when(
                                microplastic == "Bisphenol A" ~ "BPA",
                                microplastic == "bpa" ~ "BPA",
                                microplastic == "dioxin" ~"Dioxin",
                                .default = microplastic)) |>
  count(microplastic, microplastic_recoded)
```

case_when() can also overwrite/update a variable

You need to specify what we want in the first part of `mutate`.

```
plastics |>
  mutate(microplastic = case_when(
                              microplastic == "Bisphenol A" ~ "BPA",
                              microplastic == "bpa" ~ "BPA",
                              microplastic == "dioxin" ~"Dioxin",
                              .default = microplastic)) |>
  count(microplastic)

# A tibble: 3 × 2
  microplastic     n
  <chr>        <int>
1 BPA              5
2 Dioxin           4
3 Other            3
```

More complicated case_when()

`case_when` can do complicated statements and can match many patterns at a time.

```
plastics |>
  mutate(microplastic_recoded = case_when(
    microplastic %in% c("dioxin", "Dioxin") ~ "Dioxin",
    microplastic %in% c("BPA", "bpa", "Bisphenol A") ~ "BPA",
                     .default = microplastic)) |>

  count(microplastic, microplastic_recoded)
```

```
# A tibble: 6 × 3
  microplastic microplastic_recoded     n
  <chr>        <chr>                <int>
1 BPA          BPA                      2
2 Bisphenol A  BPA                      1
3 Dioxin       Dioxin                   2
4 Other        Other                    3
5 bpa          BPA                      2
6 dioxin       Dioxin                   2
```

Another reason for **case_when( )**

`case_when` can do very sophisticated comparisons!

Here we create a new variable called `Effect`.

```
plastics <- plastics |>
      mutate(Effect = case_when(
                    blood_level_change_nM > 0 ~ "Increase",
                    blood_level_change_nM == 0 ~ "Same",
                    blood_level_change_nM < 0 ~ "Decrease"))

head(plastics)
```

```
# A tibble: 6 × 5
  Foods microplastic blood_level_start_nM blood_level_change_nM Effect
  <chr> <chr>                       <dbl>                  <dbl> <chr>
1 A     Dioxin                          1                    2.5 Increase
2 B     Dioxin                          7                    2.5 Increase
3 B     Other                           2                    0.5 Increase
4 A     Bisphenol A                     3                   -0.5 Decrease
5 B     BPA                             5                    0.5 Increase
6 B     dioxin                          8                    0.5 Increase
```

Now it is easier to see what is happening

```
plastics |>
  count(Foods, Effect)

# A tibble: 3 × 3
  Foods Effect       n
  <chr> <chr>    <int>
1 A     Decrease     1
2 A     Increase     3
3 B     Increase     8
```

Note that if you change data classes this might impact .default

```
class(pull(plastics, blood_level_change_nM))

plastics <- plastics |>
      mutate(Effect = case_when(
                        blood_level_change_nM > 0 ~ "Increase",
                        blood_level_change_nM == 0 ~ "Same",
                        blood_level_change_nM < 0 ~ "Decrease",
                        .default = blood_level_change_nM))
#default is class double
# this will give an error!

plastics <- plastics |>
      mutate(Effect = case_when(
                        blood_level_change_nM > 0 ~ "Increase",
                        blood_level_change_nM == 0 ~ "Same",
                        blood_level_change_nM < 0 ~ "Decrease",
                        .default = as.character(blood_level_change_nM)))
# this works!
```

multiple conditions with **case_when** recoding

```
plastics |>
      mutate(Amt_change = case_when(
                    blood_level_change_nM > 0 & blood_level_change_nM < 2 ~ "Small",
                    blood_level_change_nM >= 2 ~ "Large",
                    blood_level_change_nM < 0 & blood_level_change_nM > -2 ~ "Small",
                    blood_level_change_nM <= -2 ~ "Large",
                    blood_level_change_nM == 0 ~ "none"))
```

```
# A tibble: 12 × 6
  Foods microplastic blood_level_start_nM blood_level_change_nM Effect
  <chr> <chr>                        <dbl>                 <dbl> <chr>
1 A     Dioxin                           1                   2.5 Increase
2 B     Dioxin                           7                   2.5 Increase
3 B     Other                            2                   0.5 Increase
4 A     Bisphenol A                      3                  -0.5 Decrease
5 B     BPA                              5                   0.5 Increase
6 B     dioxin                           8                   0.5 Increase
  Amt_change
  <chr>
1 Large
2 Large
3 Small
4 Small
5 Small
6 Small
# ℹ 6 more rows
```

GUT CHECK: we need to use what function with **`case_when()`** to modify or create a new variable?

A. `modify()`

B. `select()`

C. `mutate()`

GUT CHECK: If we want all unspecified values to remain the same with `case_when()`, how should we complete the `.default` statement?

A. = the name of the variable we are modifying or using as source

B. = “same”

Working with strings

Strings in R

- R can do much more than find exact matches for a whole string!

The `stringr` package

The `stringr` package:

- Modifying or finding **part** or all of a character string
- We will not cover `grep` or `gsub` - base R functions
  - are used on forums for answers
- Almost all functions start with `str_*`

`stringr`

`str_detect`, and `str_replace` search for matches to argument pattern within each element of a **character vector** (not data frame or tibble!).

- `str_detect` - returns `TRUE` if `pattern` is found
- `str_replace` - replaces `pattern` with `replacement`

`str_detect()`

The `string` argument specifies what to check
The `pattern` argument specifies what to check for (case sensitive)

```
Effect <- pull(plastics) |> head(n = 6)
Effect
```

```
[1] "Increase" "Increase" "Increase" "Decrease" "Increase" "Increase"
```

```
str_detect(string = Effect, pattern = "d")
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
str_detect(string = Effect, pattern = "D")
```

```
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

`str_replace()`

The `string` argument specifies what to check
The `pattern` argument specifies what to check for
The `replacement` argument specifies what to replace the pattern with

```
str_replace(string = Effect, pattern = "D", replacement = "d")
```

```
[1] "Increase" "Increase" "Increase" "decrease" "Increase" "Increase"
```

`str_replace()` only replaces the first instance of the pattern in each value

`str_replace_all()` can be used to replace all instances within each value

```
str_replace(string = Effect, pattern = "e", replacement = "E")
```

```
[1] "IncrEase" "IncrEase" "IncrEase" "DEcrease" "IncrEase" "IncrEase"
```

```
str_replace_all(string = Effect, pattern = "e", replacement = "E")
```

```
[1] "IncrEasE" "IncrEasE" "IncrEasE" "DEcrEasE" "IncrEasE" "IncrEasE"
```

Subsetting part of a string

`str_sub()` allows you to subset part of a string
The `string` argument specifies what strings to work with
The `start` argument specifies position of where to start
The `end` argument specifies position of where to end

```
str_sub(string = Effect, start = 1, end = 3)
```

```
[1] "Inc" "Inc" "Inc" "Dec" "Inc" "Inc"
```

`filter` and `stringr` functions

```
head(plastics, n = 4)
```

```
# A tibble: 4 × 5
  Foods microplastic blood_level_start_nM blood_level_change_nM Effect
  <chr> <chr>                        <dbl>                  <dbl> <chr>
1 A     Dioxin                           1                    2.5 Increase
2 B     Dioxin                           7                    2.5 Increase
3 B     Other                            2                    0.5 Increase
4 A     Bisphenol A                      3                   -0.5 Decrease
```

```
plastics |>
  filter(str_detect(string = microplastic,
                    pattern = "B"))
```

```
# A tibble: 3 × 5
  Foods microplastic blood_level_start_nM blood_level_change_nM Effect
  <chr> <chr>                        <dbl>                  <dbl> <chr>
1 A     Bisphenol A                      3                   -0.5 Decrease
2 B     BPA                              5                    0.5 Increase
3 B     BPA                              7                    1.5 Increase
```

OK back to our original problem

```
count(plastics, microplastic)

# A tibble: 6 × 2
  microplastic     n
  <chr>        <int>
1 BPA              2
2 Bisphenol A      1
3 Dioxin           2
4 Other            3
5 bpa              2
6 dioxin           2
```

`case_when()` made an improvement

But we still might miss a strange value - like a misspelling

```
plastics |>
  mutate(microplastic_recoded = case_when(
    microplastic %in% c("Dioxin", "dioxin") ~ "Dioxin",
    microplastic %in% c("BPA", "bpa", "Bisphenol A") ~ "BPA",
    .default = microplastic))
```

`case_when()` improved with `stringr`

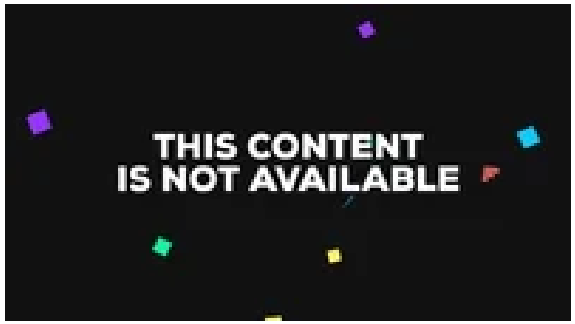
`^` indicates the beginning of a character string `$` indicates the end

```
plastics |>
  mutate(microplastic_recoded = case_when(
    str_detect(string = microplastic, pattern = "^b|^B") ~ "BPA",
    str_detect(string = microplastic, pattern = "^d|^D") ~ "Dioxin",
    .default =  microplastic)) |>
  count(microplastic, microplastic_recoded)
```

```
# A tibble: 6 × 3
  microplastic microplastic_recoded     n
  <chr>        <chr>                <int>
1 BPA          BPA                      2
2 Bisphenol A  BPA                      1
3 Dioxin       Dioxin                   2
4 Other        Other                    3
5 bpa          BPA                      2
6 dioxin       Dioxin                   2
```

This is a more robust solution! It will catch typos as long as the first letter is correct.

That's better!



GUT CHECK: What `stringr` function helps us find a string pattern?

A. `str_replace()`

B. `str_find()`

C. `str_detect()`

Separating and uniting data

Uniting columns

The `unite()` function can help combine columns
The `col` argument specifies new column name
The `sep` argument specifies what separator to use when combining -default is "_"
The `remove` argument specifies if you want to drop the old columns

```
plastics_comb <- plastics |>
  unite(Foods, Effect, col = "change", remove = TRUE)

plastics_comb

# A tibble: 12 × 4
   change     microplastic blood_level_start_nM blood_level_change_nM
   <chr>      <chr>                       <dbl>                  <dbl>
 1 A_Increase Dioxin                          1                    2.5
 2 B_Increase Dioxin                          7                    2.5
 3 B_Increase Other                           2                    0.5
 4 A_Decrease Bisphenol A                     3                   -0.5
 5 B_Increase BPA                             5                    0.5
 6 B_Increase dioxin                          8                    0.5
 7 A_Increase bpa                             6                    2.5
 8 B_Increase Other                           5                    3.5
 9 B_Increase dioxin                          2                    0.5
10 A_Increase bpa                             1                    1.5
11 B_Increase BPA                             7                    1.5
12 B_Increase Other                           3                    2.5
```

Separating columns based on a separator

The `separate()` function from `tidyr` can split a column into multiple columns.
The `col` argument specifies what column to work with
The `into` argument specifies names of new columns
The `sep` argument specifies what to separate by

```
plastics_comb <- plastics_comb |>
  separate(col = change, into = c("Foods", "Change"), sep = "_" )
plastics_comb
```

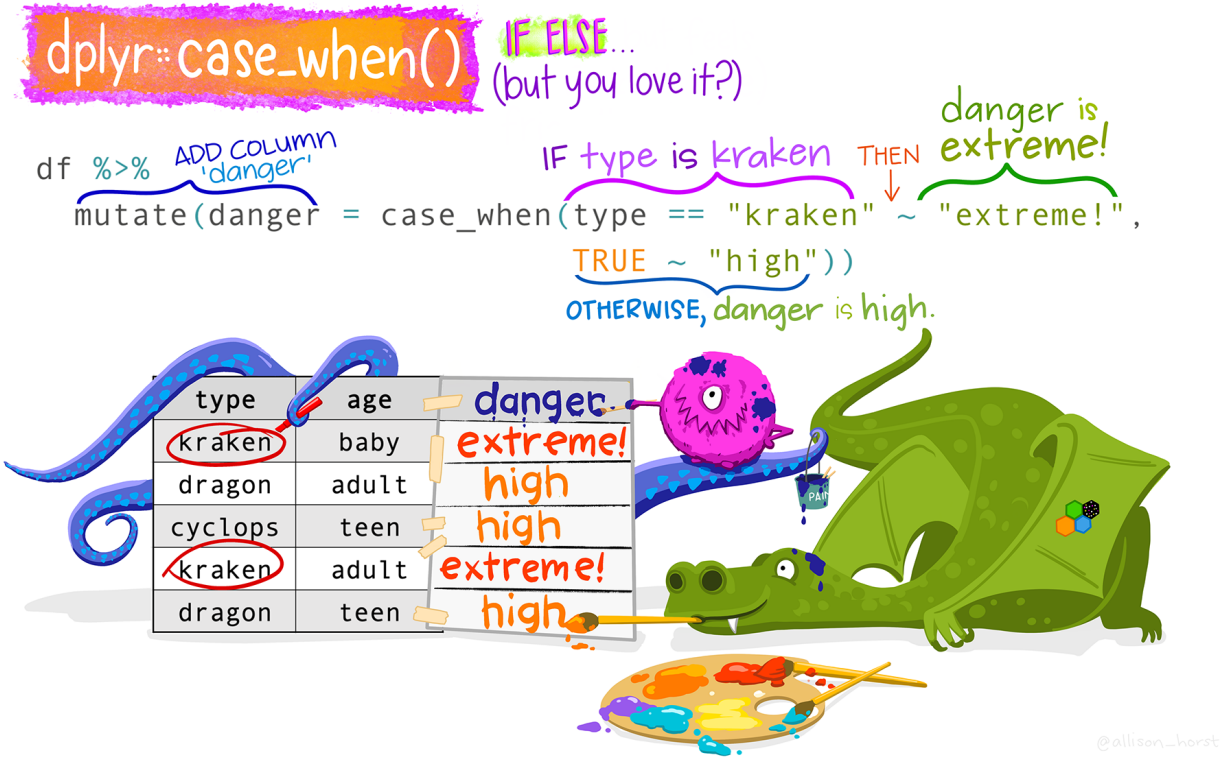
```
# A tibble: 12 × 5
   Foods Change   microplastic blood_level_start_nM blood_level_change_nM
   <chr> <chr>    <chr>                       <dbl>                  <dbl>
 1 A     Increase Dioxin                          1                    2.5
 2 B     Increase Dioxin                          7                    2.5
 3 B     Increase Other                           2                    0.5
 4 A     Decrease Bisphenol A                     3                   -0.5
 5 B     Increase BPA                             5                    0.5
 6 B     Increase dioxin                          8                    0.5
 7 A     Increase bpa                             6                    2.5
 8 B     Increase Other                           5                    3.5
 9 B     Increase dioxin                          2                    0.5
10 A     Increase bpa                             1                    1.5
11 B     Increase BPA                             7                    1.5
12 B     Increase Other                           3                    2.5
```

Summary

- `case_when()` requires `mutate()` when working with dataframes/tibbles
- `case_when()` can recode **entire values** based on **conditions** (need quotes for conditions and new values)
    - remember `case_when()` needs `.default = varaible` to keep values that aren't specified by conditions, otherwise will be `NA`

Note: you might see the `recode()` function, it only does some of what `case_when()` can do, so we skipped it, but it is in the extra slides at the end.

Summary continued



"Artwork by @allison_horst". https://allisonhorst.com/

Summary Continued

- `stringr` package has great functions for looking for specific **parts of values** especially `filter()` and `str_detect()` combined
- `stringr` also has other useful string functions like `str_detect()` (finding patterns in a column or vector), `str_subset()` (parsing text), `str_replace()` (replacing the first instance in values), `str_replace_all()` (replacing all instances in each value) and **more**!
- `separate()` can split columns into additional columns
- `unite()` can combine columns
- `:` can indicate when you want to start and end with columns next to one another

Lab Part 2

- Class Website
- Lab. - Day 5 Cheatsheet - Posit's `stringr` Cheatsheet

Image by Gerd Altmann from Pixabay

Extra Slides

`n_complete_row()` evaluating how many columns are complete for each row

```
head(plastics_comb)
```

```
# A tibble: 6 × 5
  Foods Change   microplastic blood_level_start_nM blood_level_change_nM
  <chr> <chr>    <chr>                        <dbl>                  <dbl>
1 A     Increase Dioxin                           1                    2.5
2 B     Increase Dioxin                           7                    2.5
3 B     Increase Other                            2                    0.5
4 A     Decrease Bisphenol A                      3                   -0.5
5 B     Increase BPA                              5                    0.5
6 B     Increase dioxin                           8                    0.5
```

```
head(plastics_comb) |> n_complete_row()
```

```
[1] 5 5 5 5 5 5
```

`recode()` function

This is similar to `case_when()` but it can't do as much.

(need `mutate` for data frames/tibbles!)

```
# General Format - this is not code!
{data_input} |>
  mutate({variable_to_fix_or_new} = recode({Variable_fixing}, {old_value} = {new_value},
                                           {another_old_value} = {new_value}))
```

recode() function

Need quotes for new values! Tolerates quotes for old values.

```
plastics |>
  mutate(microplastic_recoded = recode(microplastic,
                         "Bisphenol A" =  "BPA",
                                 "bpa" = "BPA",
                              "dioxin" = "Dioxin") |>
  count(microplastic, microplastic_recoded)
```

recode()

```
plastics |>
  mutate(microplastic_recoded = recode(microplastic,
                         "Bisphenol A" =  "BPA",
                                 "bpa" = "BPA",
                              "dioxin" = "Dioxin")) |>
  count(microplastic, microplastic_recoded)

# A tibble: 6 × 3
  microplastic microplastic_recoded     n
  <chr>        <chr>                <int>
1 BPA          BPA                      2
2 Bisphenol A  BPA                      1
3 Dioxin       Dioxin                   2
4 Other        Other                    3
5 bpa          BPA                      2
6 dioxin       Dioxin                   2
```

Can update or overwrite variables with recode too!

Just use the same variable name to change the variable within mutate.

```
plastics |>
  mutate(microplastic = recode(microplastic,
                       "Bisphenol A" =  "BPA",
                               "bpa" = "BPA",
                           "dioxin" = "Dioxin")) |>
  count(microplastic)

# A tibble: 3 × 2
  microplastic     n
  <chr>        <int>
1 BPA              5
2 Dioxin           4
3 Other            3
```

More complicated case_when

```
ces <- read_csv(file = "https://daseh.org/data/CalEnviroScreen_data.csv")

Rows: 8035 Columns: 67
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (3): CaliforniaCounty, ApproxLocation, CES4.0PercRange
dbl (64): CensusTract, ZIP, Longitude, Latitude, CES4.0Score, CES4.0Percenti...

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

set.seed(123)
ces |> mutate(new_col_case_when =
                case_when(Longitude < -121 & Latitude > 37.8 ~ "Distract A",
                          .default = "District B")) |>
                select(Longitude, Latitude, new_col_case_when) |>
                slice_sample(n = 6)

# A tibble: 6 × 3
  Longitude Latitude new_col_case_when
      <dbl>    <dbl> <chr>
1     -118.     34.1 District B
2     -118.     34.2 District B
3     -118.     34.1 District B
4     -122.     37.9 Distract A
5     -118.     33.7 District B
6     -118.     33.9 District B
```

Don't need **case_when( )** if just calculating new variables

```
ces |> mutate(num_col_mutate = Longitude * Latitude) |> pull(num_col_mutate)
```

```
  [1] -4628.628 -4626.923 -4626.181 -4627.226 -4627.539 -4626.747 -4626.999
  [8] -4627.869 -4627.019 -4625.697 -4625.301 -4625.090 -4623.895 -4624.469
 [15] -4625.011 -4624.364 -4625.312 -4624.039 -4623.577 -4623.748 -4623.065
 [22] -4622.705 -4623.340 -4622.929 -4622.560 -4621.886 -4622.292 -4620.986
 [29] -4622.073 -4623.599 -4623.264 -4622.834 -4622.748 -4622.293 -4622.547
 [36] -4623.105 -4624.034 -4624.719 -4624.272 -4624.876 -4626.124 -4625.382
 [43] -4623.124 -4623.478 -4621.235 -4621.058 -4620.332 -4620.480 -4621.368
 [50] -4621.776 -4621.443 -4621.355 -4621.027 -4620.606 -4620.087 -4620.586
 [57] -4620.667 -4620.065 -4619.671 -4619.019 -4619.601 -4619.386 -4617.044
 [64] -4618.494 -4617.961 -4618.796 -4619.591 -4618.345 -4618.680 -4619.270
 [71] -4619.649 -4618.712 -4617.975 -4617.781 -4617.642 -4617.198 -4617.174
 [78] -4614.951 -4615.876 -4615.344 -4616.535 -4616.034 -4616.425 -4617.171
 [85] -4619.375 -4615.766 -4615.195 -4614.288 -4613.738 -4613.512 -4614.361
 [92] -4614.937 -4614.010 -4613.137 -4610.191 -4610.268 -4609.656 -4610.560
 [99] -4611.650 -4612.492 -4612.416 -4612.547 -4612.646 -4612.075 -4609.889
[106] -4611.035 -4610.860 -4611.366 -4610.500 -4623.748 -4634.191 -4634.396
[113] -4634.714 -4634.751 -4633.355 -4633.094 -4633.820 -4633.931 -4633.377
[120] -4632.642 -4632.643 -4631.540 -4631.832 -4631.933 -4632.268 -4630.211
[127] -4631.774 -4631.392 -4631.163 -4630.990 -4630.892 -4630.246 -4629.335
[134] -4629.593 -4630.078 -4630.249 -4630.470 -4630.591 -4629.353 -4629.166
[141] -4628.991 -4628.580 -4629.126 -4628.569 -4627.522 -4628.107 -4628.018
[148] -4628.139 -4628.419 -4627.845 -4627.639 -4627.768 -4626.305 -4622.760
[155] -4623.706 -4616.068 -4619.294 -4619.348 -4619.609 -4618.471 -4618.285
[162] -4617.670 -4616.879 -4615.446 -4614.954 -4612.562 -4614.192 -4615.984
[169] -4616.531 -4617.827 -4620.699 -4603.301 -4611.086 -4604.367 -4605.354
[176] -4605.684 -4604.115 -4603.565 -4603.244 -4601.757 -4602.313 -4601.495
[183] -4600.486 -4600.793 -4609.306 -4609.328 -4609.417 -4607.750 -4607.706
[190] -4608.804 -4608.021 -4608.092 -4606.686 -4606.445 -4604.696 -4605.782
```

`case_when()` if you want NA values

```
ces |> mutate(num_new =  case_when(
  Longitude < -121 & Latitude > 37.8 ~  Longitude * Latitude),
  .default = NA) |>
  pull(num_new)
```

```
  [1] -4628.628 -4626.923 -4626.181 -4627.226 -4627.539 -4626.747 -4626.999
  [8] -4627.869 -4627.019 -4625.697 -4625.301 -4625.090 -4623.895 -4624.469
 [15] -4625.011 -4624.364 -4625.312 -4624.039 -4623.577 -4623.748 -4623.065
 [22] -4622.705 -4623.340 -4622.929 -4622.560 -4621.886 -4622.292        NA
 [29] -4622.073 -4623.599 -4623.264 -4622.834 -4622.748 -4622.293 -4622.547
 [36] -4623.105 -4624.034 -4624.719 -4624.272 -4624.876 -4626.124 -4625.382
 [43] -4623.124 -4623.478 -4621.235 -4621.058 -4620.332 -4620.480 -4621.368
 [50] -4621.776 -4621.443 -4621.355        NA        NA        NA        NA
 [57] -4620.667        NA        NA        NA        NA        NA        NA
 [64]        NA        NA        NA        NA        NA        NA        NA
 [71] -4619.649        NA        NA        NA        NA        NA        NA
 [78]        NA        NA        NA        NA        NA        NA        NA
 [85] -4619.375        NA        NA        NA        NA        NA        NA
 [92]        NA        NA        NA        NA        NA        NA        NA
 [99]        NA        NA        NA        NA        NA        NA        NA
[106]        NA        NA        NA        NA -4623.748 -4634.191 -4634.396
[113] -4634.714 -4634.751 -4633.355 -4633.094 -4633.820 -4633.931 -4633.377
[120] -4632.642 -4632.643 -4631.540 -4631.832 -4631.933 -4632.268 -4630.211
[127] -4631.774 -4631.392 -4631.163 -4630.990 -4630.892 -4630.246 -4629.335
[134] -4629.593 -4630.078 -4630.249 -4630.470 -4630.591 -4629.353 -4629.166
[141] -4628.991 -4628.580 -4629.126 -4628.569 -4627.522 -4628.107 -4628.018
[148] -4628.139 -4628.419 -4627.845 -4627.639 -4627.768 -4626.305 -4622.760
[155] -4623.706        NA        NA        NA        NA        NA        NA
[162]        NA        NA        NA        NA        NA        NA        NA
[169]        NA        NA        NA        NA        NA        NA        NA
```

String Splitting

- `str_split(string, pattern)` - splits strings up - returns list!

```
library(stringr)
x <- c("I really like writing R code")
df <- tibble(x = c("I really", "like writing", "R code programs"))
y <- unlist(str_split(x, " "))
y
```

```
[1] "I"       "really"  "like"    "writing" "R"       "code"
```

```
length(y)
```

```
[1] 6
```

A bit on Regular Expressions

- http://www.regular-expressions.info/reference.html
- They can use to match a large number of strings in one statement
- `.` matches any single character
- `*` means repeat as many (even if 0) more times the last character
- `?` makes the last thing optional
- `^` matches start of vector `^a` - starts with “a”
- `$` matches end of vector `b$` - ends with “b”

Let's look at modifiers for `stringr`

```
?modifiers
```

- `fixed` - match everything exactly
- `ignore_case` is an option to not have to use `tolower`

Using a fixed expression

One example case is when you want to split on a period "`.`". In regular expressions `.` means **ANY** character, so we need to specify that we want R to interpret "." as simply a period.

```
str_split("I.like.strings", ".")

[[1]]
 [1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""

str_split("I.like.strings", fixed("."))

[[1]]
[1] "I"       "like"    "strings"

str_split("I.like.strings", "\\.")

[[1]]
[1] "I"       "like"    "strings"
```

Pasting strings with **paste** and **paste0**

Paste can be very useful for joining vectors together:

```
paste("Visit", 1:5, sep = "_")
```

```
[1] "Visit_1" "Visit_2" "Visit_3" "Visit_4" "Visit_5"
```

```
paste("Visit", 1:5, sep = "_", collapse = "_")
```

```
[1] "Visit_1_Visit_2_Visit_3_Visit_4_Visit_5"
```

```
# and paste0 can be even simpler see ?paste0
paste0("Visit",1:5) # no space!
```

```
[1] "Visit1" "Visit2" "Visit3" "Visit4" "Visit5"
```

Comparison of `stringr` to base R - not covered

Splitting Strings

Substringing

`stringr`

- `str_split(string, pattern)` - splits strings up - returns list!

Splitting String:

In `stringr`, `str_split` splits a vector on a string into a `list`

```
library(stringr)
x <- c("I really", "like writing", "R code programs")
y <- str_split(x, pattern =  " ") # returns a list
y
```

```
[[1]]
[1] "I"      "really"

[[2]]
[1] "like"    "writing"

[[3]]
[1] "R"        "code"     "programs"
```

'Find' functions: stringr compared to base R

Base R does not use these functions. Here is a "translator" of the `stringr` function to base R functions

- `str_detect` - similar to `grepl` (return logical)
- `grep(value = FALSE)` is similar to `which(str_detect())`
- `str_subset` - similar to `grep(value = TRUE)` - return value of matched
- `str_replace` - similar to `sub` - replace one time
- `str_replace_all` - similar to `gsub` - replace many times

Important Comparisons

Base R:

- Argument order is `(pattern, x)`
- Uses option `(fixed = TRUE)`

`stringr`

- Argument order is `(string, pattern)` aka `(x, pattern)`
- Uses function `fixed(pattern)`

some data to work with

```
Sal <- read_csv(file =
    "https://daseh.org/data/Baltimore_City_Employee_Salaries_FY2015.csv")
```

Showing difference in `str_extract`

`str_extract` extracts just the matched string

```
ss <- str_extract(Sal$Name, "Rawling")
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
head(ss)
```

```
character(0)
```

```
ss[ !is.na(ss)]
```

```
character(0)
```

Showing difference in `str_extract` and `str_extract_all`

`str_extract_all` extracts all the matched strings

```
head(str_extract(Sal$AgencyID, "\\d"))
```

```
[1] "0" "2" "6" "9" "4" "9"
```

```
head(str_extract_all(Sal$AgencyID, "\\d"), 2)
```

```
[[1]]
[1] "0" "3" "0" "3" "1"

[[2]]
[1] "2" "9" "0" "4" "5"
```

Using Regular Expressions

- Look for any name that starts with:
  - Payne at the beginning,
  - Leonard and then an S
  - Spence then capital C

```
head(grep("^Payne.*", x = Sal$name, value = TRUE), 3)
```

```
[1] "Payne El,Boaz L"          "Payne El,Jackie"
[3] "Payne Johnson,Nickole A"
```

```
head(grep("Leonard.?S", x = Sal$name, value = TRUE))
```

```
[1] "Payne,Leonard S"      "Szumlanski,Leonard S"
```

```
head(grep("Spence.*C.*", x = Sal$name, value = TRUE))
```

```
[1] "Spencer,Charles A"  "Spencer,Clarence W" "Spencer,Michael C"
```

Using Regular Expressions: `stringr`

```
head(str_subset( Sal$name, "^Payne.*"), 3)
```

```
[1] "Payne El,Boaz L"          "Payne El,Jackie"
[3] "Payne Johnson,Nickole A"
```

```
head(str_subset( Sal$name, "Leonard.?S"))
```

```
[1] "Payne,Leonard S"      "Szumlanski,Leonard S"
```

```
head(str_subset( Sal$name, "Spence.*C.*"))
```

```
[1] "Spencer,Charles A"  "Spencer,Clarence W" "Spencer,Michael C"
```

Replace

Let's say we wanted to sort the data set by Annual Salary:

```
class(Sal$AnnualSalary)
```

```
[1] "character"
```

```
sort(c("1", "2", "10")) #  not sort correctly (order simply ranks the data)
```

```
[1] "1"  "10" "2"
```

```
order(c("1", "2", "10"))
```

```
[1] 1 3 2
```

Replace

So we must change the annual pay into a numeric:

```
head(Sal$AnnualSalary, 4)
```

```
[1] "$55314.00" "$74000.00" "$64500.00" "$46309.00"
```

```
head(as.numeric(Sal$AnnualSalary), 4)
```

```
Warning in head(as.numeric(Sal$AnnualSalary), 4): NAs introduced by coercion
```

```
[1] NA NA NA NA
```

R didn't like the `$` so it thought turned them all to `NA`.

`sub()` and `gsub()` can do the replacing part in base R.

Replacing and subbing

Now we can replace the `$` with nothing (used `fixed=TRUE` because `$` means ending):

```
Sal$AnnualSalary <- as.numeric(gsub(pattern = "$", replacement="",
                                    Sal$AnnualSalary, fixed=TRUE))
Sal <- Sal[order(Sal$AnnualSalary, decreasing=TRUE), ]
Sal[1:5, c("name", "AnnualSalary", "JobTitle")]
```

```
# A tibble: 5 × 3
  name            AnnualSalary JobTitle
  <chr>                  <dbl> <chr>
1 Mosby,Marilyn J       238772 STATE'S ATTORNEY
2 Batts,Anthony W       211785 Police Commissioner
3 Wen,Leana             200000 Executive Director III
4 Raymond,Henry J       192500 Executive Director III
5 Swift,Michael         187200 CONTRACT SERV SPEC II
```

Replacing and subbing: `stringr`

We can do the same thing (with 2 piping operations!) in dplyr

```
dplyr_sal <- Sal
dplyr_sal <- dplyr_sal |>
  mutate(AnnualSalary = str_replace(AnnualSalary, fixed("$"), "")) |>
  mutate(AnnualSalary = as.numeric(AnnualSalary)) |>
  arrange(desc(AnnualSalary))
check_Sal <- Sal
rownames(check_Sal) <- NULL
all.equal(check_Sal, dplyr_sal)
```

```
[1] TRUE
```

Creating Two-way Tables

A two-way table. If you pass in 2 vectors, `table` creates a 2-dimensional table.

```
tab <- table(c(0, 1, 2, 3, 2, 3, 3, 2,2, 3),
             c(0, 1, 2, 3, 2, 3, 3, 4, 4, 3),
              useNA = "always")
tab
```

```
      0 1 2 3 4 <NA>
  0   1 0 0 0 0    0
  1   0 1 0 0 0    0
  2   0 0 2 0 2    0
  3   0 0 0 4 0    0
  <NA> 0 0 0 0 0    0
```

Creating Two-way Tables

```
tab_df <- tibble(x = c(0, 1, 2, 3, 2, 3, 3, 2,2, 3),
                 y = c(0, 1, 2, 3, 2, 3, 3, 4, 4, 3))
tab_df |> count(x, y)
```

```
# A tibble: 5 × 3
      x     y     n
  <dbl> <dbl> <int>
1     0     0     1
2     1     1     1
3     2     2     2
4     2     4     2
5     3     3     4
```

Creating Two-way Tables

```
tab_df |>
  count(x, y) |>
  group_by(x) |> mutate(pct_x = n / sum(n))

# A tibble: 5 × 4
# Groups:   x [4]
      x     y     n pct_x
  <dbl> <dbl> <int> <dbl>
1     0     0     1   1
2     1     1     1   1
3     2     2     2   0.5
4     2     4     2   0.5
5     3     3     4   1
```

Creating Two-way Tables

```
library(scales)
tab_df |>
  count(x, y) |>
  group_by(x) |> mutate(pct_x = percent(n / sum(n)))

# A tibble: 5 × 4
# Groups:   x [4]
      x     y     n pct_x
  <dbl> <dbl> <int> <chr>
1     0     0     1 100%
2     1     1     1 100%
3     2     2     2 50%
4     2     4     2 50%
5     3     3     4 100%
```

Removing columns with threshold of percent missing values

```
is.na(df) |> head(n = 3)
```

```
         x
[1,] FALSE
[2,] FALSE
[3,] FALSE
```

```
colMeans(is.na(df))#TRUE and FALSE treated like 0 and 1
```

```
x
0
```

```
which(colMeans(is.na(df)) < 0.2) #the location of the columns <.2
```

```
x
1
```

```
df |> select(which(colMeans(is.na(df)) < 0.2))# remove if over 20% missing
```

```
# A tibble: 3 × 1
  x
  <chr>
1 I really
2 like writing
3 R code programs
```

Manipulating Data in R

Recap of Data Cleaning

- `is.na()`,`any(is.na())`, `all(is.na())`,`count()`, and functions from `naniar` like `gg_miss_var()` and `miss_var_summary` can help determine if we have `NA` values
- `miss_var_which()` can help you drop columns that have any missing values.
- `filter()` automatically removes `NA` values
- `drop_na()` can help you remove `NA` values
- `NA` values can change your calculation results
- think about what `NA` values represent - don't drop them if you shouldn't
- `replace_na()` will replace `NA values with a particular value

Recap of Data Cleaning

- `case_when()` can recode **entire values** based on conditions
  - remember `case_when()` needs `TRUE ~ variable` to keep values that aren't specified by conditions, otherwise will be `NA`
- `stringr` package has great functions for looking for specific **parts of values** especially `filter()` and `str_detect()` combined
  - also has other useful string manipulation functions like `str_replace()` and more!
  - `separate()` can split columns into additional columns
  - `unite()` can combine columns

 Cheatsheet

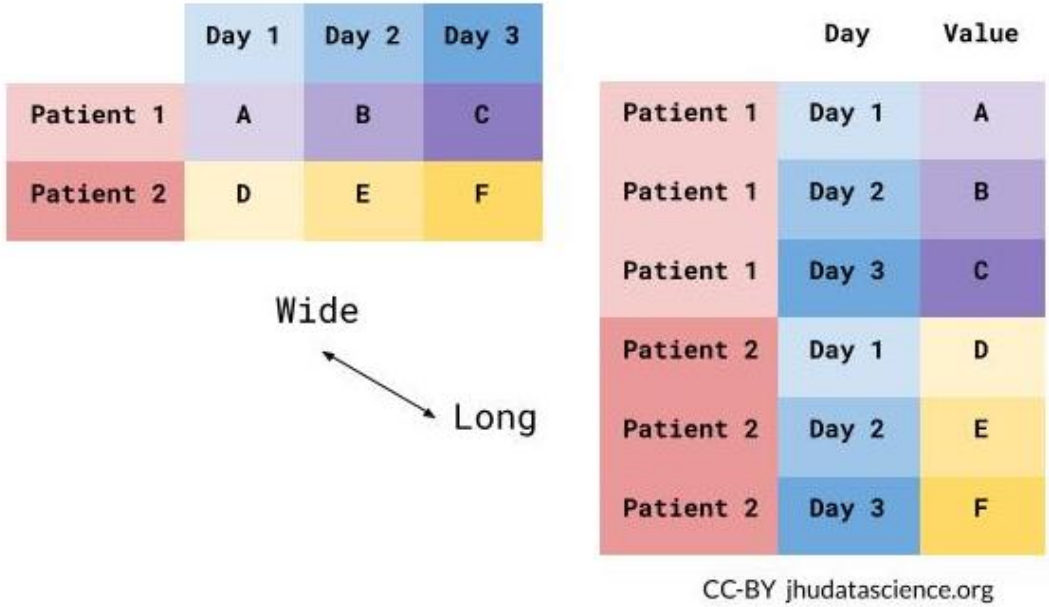
Manipulating Data

In this module, we will show you how to:

1. Reshape data from wide to long
2. Reshape data from long to wide
3. Merge Data/Joins

What is wide/long data?

Data is wide or long **with respect** to certain variables.



What is wide/long data?

Data is stored *differently* in the tibble.

Here's a small dataset looking at vaccination rates over three months in Alabama.

Wide: has many columns

```
# A tibble: 1 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514           0.511
```

Long: column names become data

```
# A tibble: 3 × 3
  State   name            value
  <chr>   <chr>           <dbl>
1 Alabama June_vacc_rate  0.516
2 Alabama May_vacc_rate   0.514
3 Alabama April_vacc_rate 0.511
```

What is wide/long data?

Wide: multiple columns per individual, values spread across multiple columns

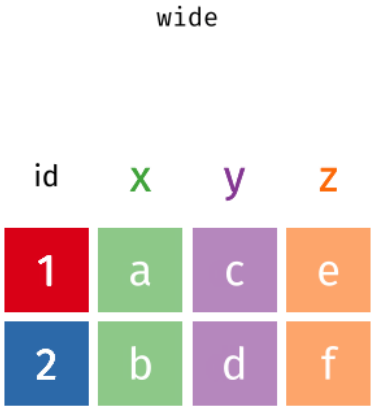
```
# A tibble: 2 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514           0.511
2 Alaska           0.627         0.626           0.623
```

Long: multiple rows per observation, a single column contains the values

```
# A tibble: 6 × 3
  State   name             value
  <chr>   <chr>            <dbl>
1 Alabama June_vacc_rate  0.516
2 Alabama May_vacc_rate   0.514
3 Alabama April_vacc_rate 0.511
4 Alaska  June_vacc_rate  0.627
5 Alaska  May_vacc_rate   0.626
6 Alaska  April_vacc_rate 0.623
```

What is wide/long data?

https://github.com/gadenbuie/tidyexplain/blob/main/images/tidyr-pivoting.gif



Why do we need to switch between wide/long data?

Wide: **Easier for humans to read**

```
# A tibble: 2 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514           0.511
2 Alaska           0.627         0.626           0.623
```

Long: **Easier for R to make plots & do analysis**

```
# A tibble: 6 × 3
  State   name             value
  <chr>   <chr>            <dbl>
1 Alabama June_vacc_rate  0.516
2 Alabama May_vacc_rate   0.514
3 Alabama April_vacc_rate 0.511
4 Alaska  June_vacc_rate  0.627
5 Alaska  May_vacc_rate   0.626
6 Alaska  April_vacc_rate 0.623
```

Pivoting using the `tidyr` package (part of `tidyverse`)

We will be talking about:

- `pivot_longer` - make multiple columns into variables, (wide to long)
- `pivot_wider` - make a variable into multiple columns, (long to wide)

You might see old functions `gather` and `spread` when googling. These are older iterations of `pivot_longer` and `pivot_wider`, respectively.

You might see other functions like `reshape()`, `melt()`, etc.

`pivot_longer`...

Reshaping data from **wide to long**

`pivot_longer()` - puts column data into rows (`tidyr` package)

- First describe which columns we want to “pivot_longer”

```
{long_data} <- {wide_data} |> pivot_longer(cols = {columns to pivot})
```

Reshaping data from wide to long

```
ex_wide
```

```
# A tibble: 2 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514           0.511
2 Alaska           0.627         0.626           0.623
```

```
ex_long <- ex_wide |> pivot_longer(cols = ends_with("rate"))
ex_long
```

```
# A tibble: 6 × 3
  State   name             value
  <chr>   <chr>            <dbl>
1 Alabama June_vacc_rate  0.516
2 Alabama May_vacc_rate   0.514
3 Alabama April_vacc_rate 0.511
4 Alaska  June_vacc_rate  0.627
5 Alaska  May_vacc_rate   0.626
6 Alaska  April_vacc_rate 0.623
```

GUT CHECK!

What does `pivot_longer()` do?

A. Summarize data

B. Import data

C. Reshape data

Reshaping wide to long: Better column names

`pivot_longer()` - puts column data into rows (`tidyr` package)

- First describe which columns we want to "pivot_longer"
- `names_to =` new name for old columns
- `values_to =` new name for old cell values

```
{long_data} <- {wide_data} |> pivot_longer(cols = {columns to pivot},
                                           names_to = {name for old columns},
                                           values_to = {name for cell values})
```

Reshaping wide to long: Better column names

Note Newly created column names (“Month” and “Rate”) in quotation marks.

```
ex_long <- ex_wide |> pivot_longer(cols = ends_with("rate"),
                                   names_to = "Month",
                                   values_to = "Rate")
ex_long
```

```
# A tibble: 6 × 3
  State   Month            Rate
  <chr>   <chr>           <dbl>
1 Alabama June_vacc_rate  0.516
2 Alabama May_vacc_rate   0.514
3 Alabama April_vacc_rate 0.511
4 Alaska  June_vacc_rate  0.627
5 Alaska  May_vacc_rate   0.626
6 Alaska  April_vacc_rate 0.623
```

Data used: Nitrate exposure

Let's look at some data on levels of nitrate in water from Washington. This dataset reports the amount of people in Washington exposed to excess levels of nitrate in their water between 1999 and 2020.

```
wide_nitrate <-
  read_csv(file = "https://daseh.org/data/Nitrate_Exposure_for_WA_Public_Water_Systems_byquarter_data.csv")
head(wide_nitrate)
```

```
# A tibble: 6 × 11
   year quarter pop_on_sampled_PWS `pop_0-3ug/L` `pop_>3-5ug/L` `pop_>5-10ug/L`
  <dbl> <chr>                 <dbl>         <dbl>          <dbl>           <dbl>
1  1999 Q1                  106720         67775              0              32
2  1999 Q2                   85541         55476              0             212
3  1999 Q3                  559137        319252         231186             212
4  1999 Q4                   26995         25969            420               0
5  2000 Q1                   34793          5904              0              92
6  2000 Q2                  184521        157396              0              32
# ℹ 5 more variables: `pop_>10-20ug/L` <dbl>, `pop_>20ug/L` <dbl>,
#   `pop_on_PWS_with_non-detect` <dbl>, pop_exposed_to_exceedances <dbl>,
#   perc_pop_exposed_to_exceedances <dbl>
```

Mission: Average population exposed by concentration

Let's imagine we want to see what proportion of the population was exposed to different nitrate concentrations. Results should look something like:

```
# A tibble: 3 × 2
  concentration_cat avg_prop_exposedpop
  <chr>                           <dbl>
1 0-3ug/L                       0.593
2 10-20ug/L                     0.000678
3 more_than_20ug/L              0.000129
```

Remove some columns we don't need

```
wide_nitrate <- wide_nitrate |>
  select(!ends_with("exceedances"))
wide_nitrate
```

```
# A tibble: 88 × 9
    year quarter pop_on_sampled_PWS `pop_0-3ug/L` `pop_>3-5ug/L` `pop_>5-10ug/L`
   <dbl> <chr>                 <dbl>          <dbl>           <dbl>            <dbl>
 1  1999 Q1                   106720          67775               0               32
 2  1999 Q2                    85541          55476               0              212
 3  1999 Q3                   559137         319252          231186              212
 4  1999 Q4                    26995          25969             420                0
 5  2000 Q1                    34793           5904               0               92
 6  2000 Q2                   184521         157396               0               32
 7  2000 Q3                    42081          20407             345                0
 8  2000 Q4                   407219         358828             995              412
 9  2001 Q1                    90054          49552             150                0
10  2001 Q2                    83521          43633            2536               90
# ℹ 78 more rows
# ℹ 3 more variables: `pop_>10-20ug/L` <dbl>, `pop_>20ug/L` <dbl>,
#   `pop_on_PWS_with_non-detect` <dbl>
```

Reshaping data from wide to long

```
long_nitrate <- wide_nitrate |>
  pivot_longer(!c(year, quarter, pop_on_sampled_PWS))
long_nitrate
```

```
# A tibble: 528 × 5
    year quarter pop_on_sampled_PWS name                          value
   <dbl> <chr>                 <dbl> <chr>                         <dbl>
 1  1999 Q1                   106720 pop_0-3ug/L                   67775
 2  1999 Q1                   106720 pop_>3-5ug/L                      0
 3  1999 Q1                   106720 pop_>5-10ug/L                    32
 4  1999 Q1                   106720 pop_>10-20ug/L                    0
 5  1999 Q1                   106720 pop_>20ug/L                       0
 6  1999 Q1                   106720 pop_on_PWS_with_non-detect    38913
 7  1999 Q2                    85541 pop_0-3ug/L                   55476
 8  1999 Q2                    85541 pop_>3-5ug/L                       0
 9  1999 Q2                    85541 pop_>5-10ug/L                    212
10  1999 Q2                    85541 pop_>10-20ug/L                    60
# ℹ 518 more rows
```

Reshaping data from **wide to long**

Un-pivoted columns (`year`, `quarter`, `pop_on_sampled_PWS`) are still columns.

```
long_nitrate
```

```
# A tibble: 528 × 5
    year quarter pop_on_sampled_PWS name                          value
   <dbl> <chr>                 <dbl> <chr>                         <dbl>
 1  1999 Q1                   106720 pop_0-3ug/L                   67775
 2  1999 Q1                   106720 pop_>3-5ug/L                      0
 3  1999 Q1                   106720 pop_>5-10ug/L                    32
 4  1999 Q1                   106720 pop_>10-20ug/L                    0
 5  1999 Q1                   106720 pop_>20ug/L                       0
 6  1999 Q1                   106720 pop_on_PWS_with_non-detect    38913
 7  1999 Q2                    85541 pop_0-3ug/L                   55476
 8  1999 Q2                    85541 pop_>3-5ug/L                       0
 9  1999 Q2                    85541 pop_>5-10ug/L                    212
10  1999 Q2                    85541 pop_>10-20ug/L                    60
# ℹ 518 more rows
```

Cleaning up long data

Let's make the `conc_count` into a proportion.

```
long_nitrate <- long_nitrate |>
  mutate(conc_prop = value / pop_on_sampled_PWS)
long_nitrate
```

```
# A tibble: 528 × 6
    year quarter pop_on_sampled_PWS name                       value conc_prop
   <dbl> <chr>                <dbl> <chr>                      <dbl>     <dbl>
 1  1999 Q1                  106720 pop_0-3ug/L                67775  0.635
 2  1999 Q1                  106720 pop_>3-5ug/L                   0  0
 3  1999 Q1                  106720 pop_>5-10ug/L                 32  0.000300
 4  1999 Q1                  106720 pop_>10-20ug/L                 0  0
 5  1999 Q1                  106720 pop_>20ug/L                    0  0
 6  1999 Q1                  106720 pop_on_PWS_with_non-detect 38913  0.365
 7  1999 Q2                   85541 pop_0-3ug/L                55476  0.649
 8  1999 Q2                   85541 pop_>3-5ug/L                    0  0
 9  1999 Q2                   85541 pop_>5-10ug/L                 212  0.00248
10  1999 Q2                   85541 pop_>10-20ug/L                 60  0.000701
# ℹ 518 more rows
```

Mission: Average population exposed by concentration

Now our data is more tidy, and we can take the averages easily!

```
long_nitrate |>
  group_by(name) |>
  summarize("avg_prop_exposedpop" = mean(conc_prop))
```

```
# A tibble: 6 × 2
  name                          avg_prop_exposedpop
  <chr>                                       <dbl>
1 pop_0-3ug/L                                0.593
2 pop_>10-20ug/L                             0.000678
3 pop_>20ug/L                                0.000129
4 pop_>3-5ug/L                               0.182
5 pop_>5-10ug/L                              0.0189
6 pop_on_PWS_with_non-detect                 0.206
```

Reshaping data from **wide to long**

There are many ways to **select** the columns we want.

Check out https://dplyr.tidyverse.org/reference/dplyr_tidy_select.html to look at more column selection options.

`pivot_wider`...

Reshaping data from **long to wide**

`pivot_wider()` - spreads row data into columns (`tidyr` package)

- `names_from =` the old column whose contents will be spread into multiple new column names.
- `values_from =` the old column whose contents will fill in the values of those new columns.

```
{wide_data} <- {long_data} |>
  pivot_wider(names_from = {Old column name: contains new column names},
              values_from = {Old column name: contains new cell values})
```

Reshaping data from long to wide

We can use `pivot_wider` to convert long data to wide format. Let's try it with the vaccine data from earlier.

```
ex_long
```

```
# A tibble: 6 × 3
  State   Month            Rate
  <chr>   <chr>           <dbl>
1 Alabama June_vacc_rate  0.516
2 Alabama May_vacc_rate   0.514
3 Alabama April_vacc_rate 0.511
4 Alaska  June_vacc_rate  0.627
5 Alaska  May_vacc_rate   0.626
6 Alaska  April_vacc_rate 0.623
```

Reshaping data from long to wide

We can use `pivot_wider` to convert long data to wide format. Let's try it with the vaccine data from earlier.

```
ex_wide2 <- ex_long |> pivot_wider(names_from = "Month",
                                   values_from = "Rate")
ex_wide2
```

```
# A tibble: 2 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514           0.511
2 Alaska           0.627         0.626           0.623
```

Reshaping nitrate exposure data

Let's go back to the nitrate exposure dataset. What if we wanted to make a wide version of the data that displayed the proportion of people at each level of nitrate exposure, with each quarter as a column?

```
long_nitrate
```

```
# A tibble: 528 × 6
    year quarter pop_on_sampled_PWS name                       value conc_prop
   <dbl> <chr>                <dbl> <chr>                      <dbl>     <dbl>
 1  1999 Q1                  106720 pop_0-3ug/L                67775  0.635
 2  1999 Q1                  106720 pop_>3-5ug/L                   0  0
 3  1999 Q1                  106720 pop_>5-10ug/L                 32  0.000300
 4  1999 Q1                  106720 pop_>10-20ug/L                 0  0
 5  1999 Q1                  106720 pop_>20ug/L                    0  0
 6  1999 Q1                  106720 pop_on_PWS_with_non-detect 38913  0.365
 7  1999 Q2                   85541 pop_0-3ug/L                55476  0.649
 8  1999 Q2                   85541 pop_>3-5ug/L                    0  0
 9  1999 Q2                   85541 pop_>5-10ug/L                 212  0.00248
10  1999 Q2                   85541 pop_>10-20ug/L                 60  0.000701
# ℹ 518 more rows
```

Reshaping nitrate exposure data

Drop some columns we don't need.

```
long_nitrate <- long_nitrate |>
  select(!c(pop_on_sampled_PWS, value))
long_nitrate
```

```
# A tibble: 528 × 4
    year quarter name                          conc_prop
   <dbl> <chr>   <chr>                             <dbl>
 1  1999 Q1      pop_0-3ug/L                       0.635
 2  1999 Q1      pop_>3-5ug/L                      0
 3  1999 Q1      pop_>5-10ug/L                     0.000300
 4  1999 Q1      pop_>10-20ug/L                    0
 5  1999 Q1      pop_>20ug/L                       0
 6  1999 Q1      pop_on_PWS_with_non-detect        0.365
 7  1999 Q2      pop_0-3ug/L                       0.649
 8  1999 Q2      pop_>3-5ug/L                      0
 9  1999 Q2      pop_>5-10ug/L                     0.00248
10  1999 Q2      pop_>10-20ug/L                    0.000701
# ℹ 518 more rows
```

Reshaping nitrate exposure data

Pivot the data!

```
wide_nitrate <- long_nitrate |>
  pivot_wider(names_from = "quarter", values_from = "conc_prop")
wide_nitrate
```

```
# A tibble: 132 × 6
    year name                              Q1       Q2       Q3      Q4
   <dbl> <chr>                          <dbl>    <dbl>    <dbl>   <dbl>
 1  1999 pop_0-3ug/L                  0.635    0.649    0.571    0.962
 2  1999 pop_>3-5ug/L                 0        0        0.413    0.0156
 3  1999 pop_>5-10ug/L                0.000300 0.00248  0.000379 0
 4  1999 pop_>10-20ug/L               0        0.000701 0        0
 5  1999 pop_>20ug/L                  0        0        0        0
 6  1999 pop_on_PWS_with_non-detect 0.365    0.348    0.0152   0.0224
 7  2000 pop_0-3ug/L                  0.170    0.853    0.485    0.881
 8  2000 pop_>3-5ug/L                 0        0        0.00820  0.00244
 9  2000 pop_>5-10ug/L                0.00264  0.000173 0        0.00101
10  2000 pop_>10-20ug/L               0        0        0        0
# ℹ 122 more rows
```

Summary

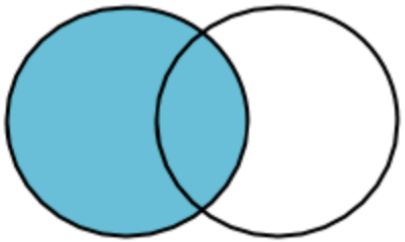
- `tidyr` package helps us convert between wide and long data
- `pivot_longer()` goes from wide -> long
  - Specify columns you want to pivot
  - Specify `names_to =` and `values_to =` for custom naming
- `pivot_wider()` goes from long -> wide
  - Specify `names_from =` and `values_from =`

Lab Part 1

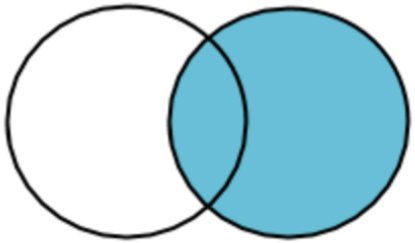
- Class Website
- Lab

Joining

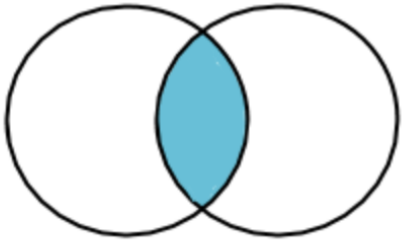
"Combining datasets"



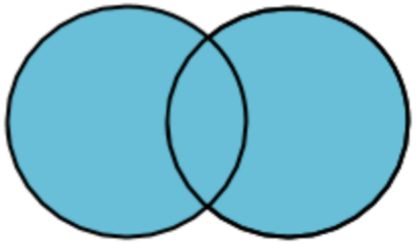
Left Join



Right Join



Inner Join



Full Outer Join

Joining in `dplyr`

- Merging/joining data sets together - usually on key variables, usually "id"
- `?join` - see different types of joining for `dplyr`
- `inner_join(x, y)` - only rows that match for `x` and `y` are kept
- `full_join(x, y)` - all rows of `x` and `y` are kept
- `left_join(x, y)` - all rows of `x` are kept even if not merged with `y`
- `right_join(x, y)` - all rows of `y` are kept even if not merged with `x`
- `anti_join(x, y)` - all rows from `x` not in `y` keeping just columns from `x`.

Merging: Simple Data

```
data_As <- read_csv(
  file = "https://daseh.org/data/data_As_1.csv")
data_cold <- read_csv(
  file = "https://daseh.org/data/data_cold_1.csv")
```

```
data_As
```

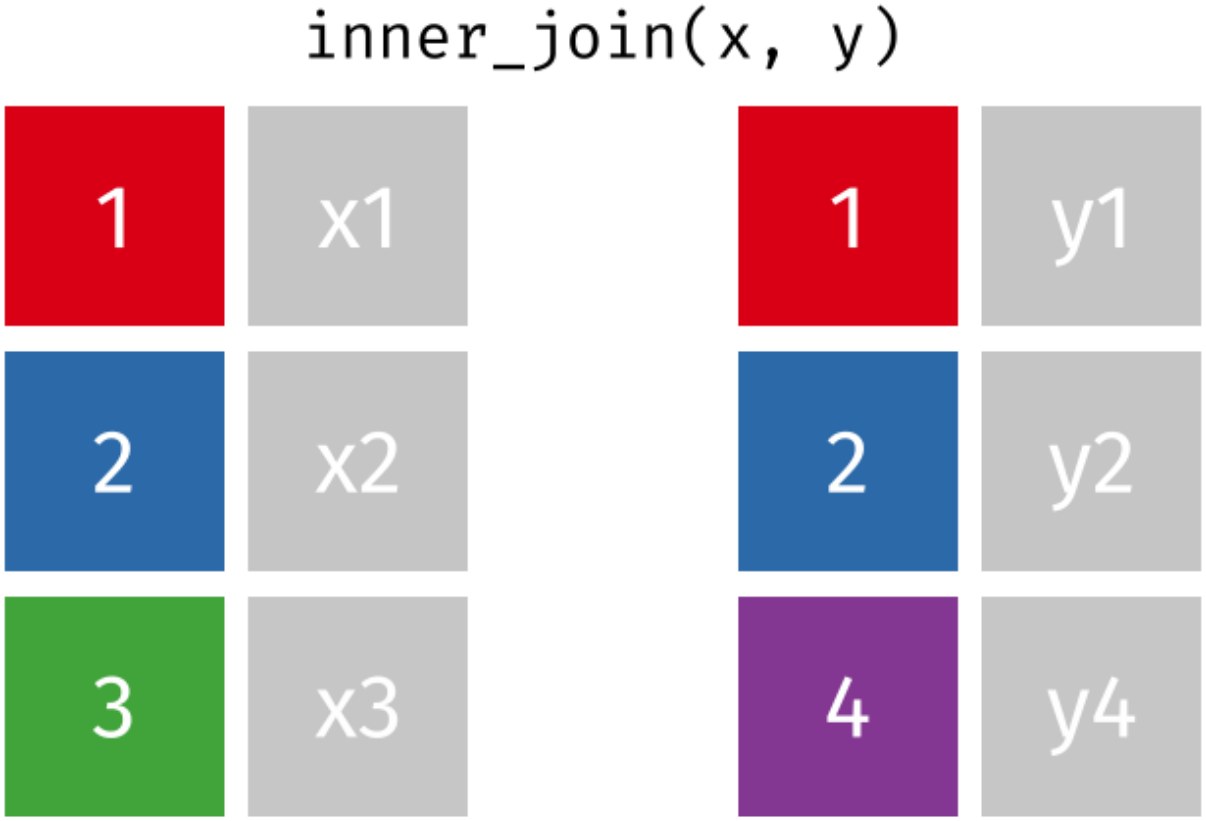
```
# A tibble: 2 × 3
  State   June_vacc_rate May_vacc_rate
  <chr>            <dbl>         <dbl>
1 Alabama          0.516         0.514
2 Alaska           0.627         0.626
```

```
data_cold
```

```
# A tibble: 2 × 2
  State  April_vacc_rate
  <chr>            <dbl>
1 Maine            0.795
2 Alaska           0.623
```

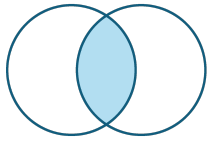
Inner Join

https://github.com/gadenbuie/tidyexplain/blob/main/images/inner-join.gif



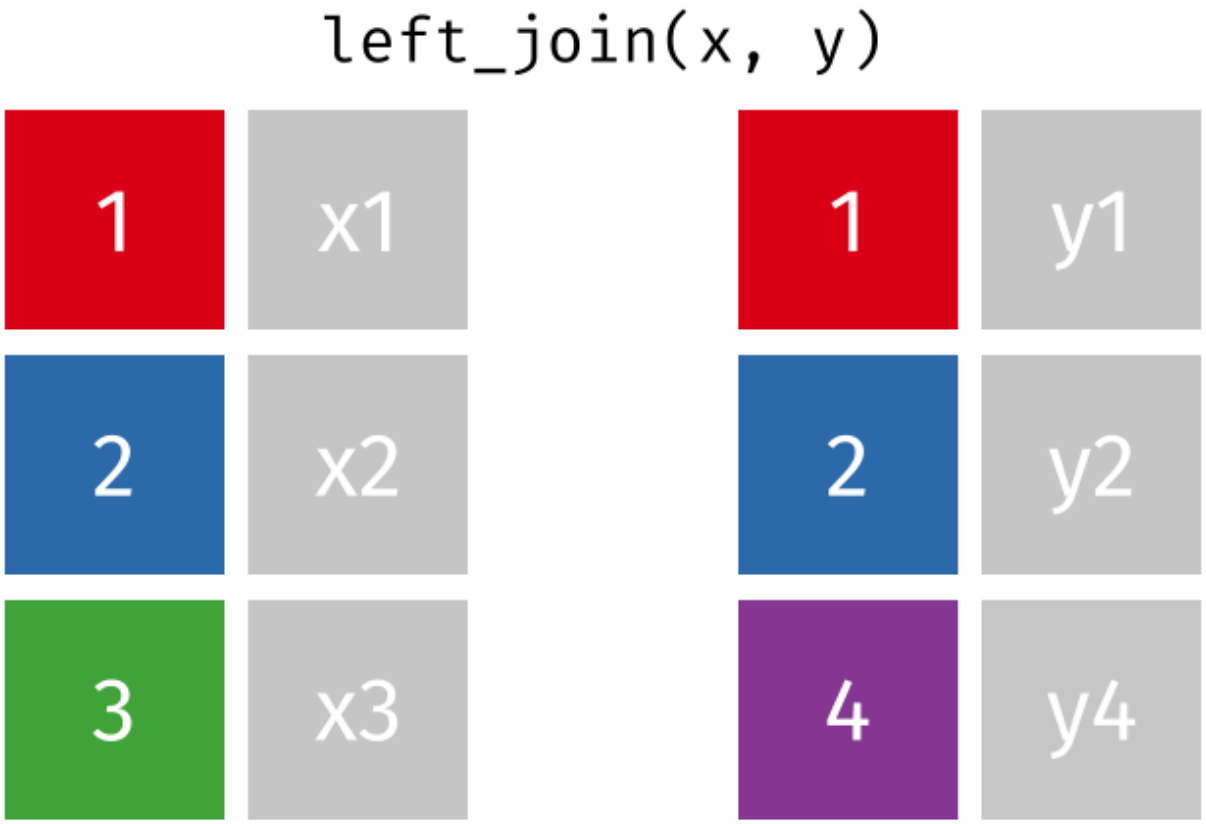
Inner Join

```
ij <- inner_join(data_As, data_cold)

Joining with `by = join_by(State)`

ij

# A tibble: 1 × 4
  State  June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>           <dbl>         <dbl>           <dbl>
1 Alaska          0.627         0.626           0.623
```



Left Join

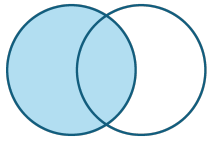
https://raw.githubusercontent.com/gadenbuie/tidyexplain/main/images/left-join.gif



Left Join

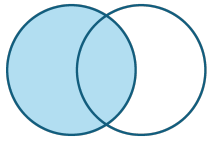
"Everything to the left of the comma"

```
lj <- left_join(data_As, data_cold)

Joining with `by = join_by(State)`

lj

# A tibble: 2 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514          NA
2 Alaska           0.627         0.626           0.623
```



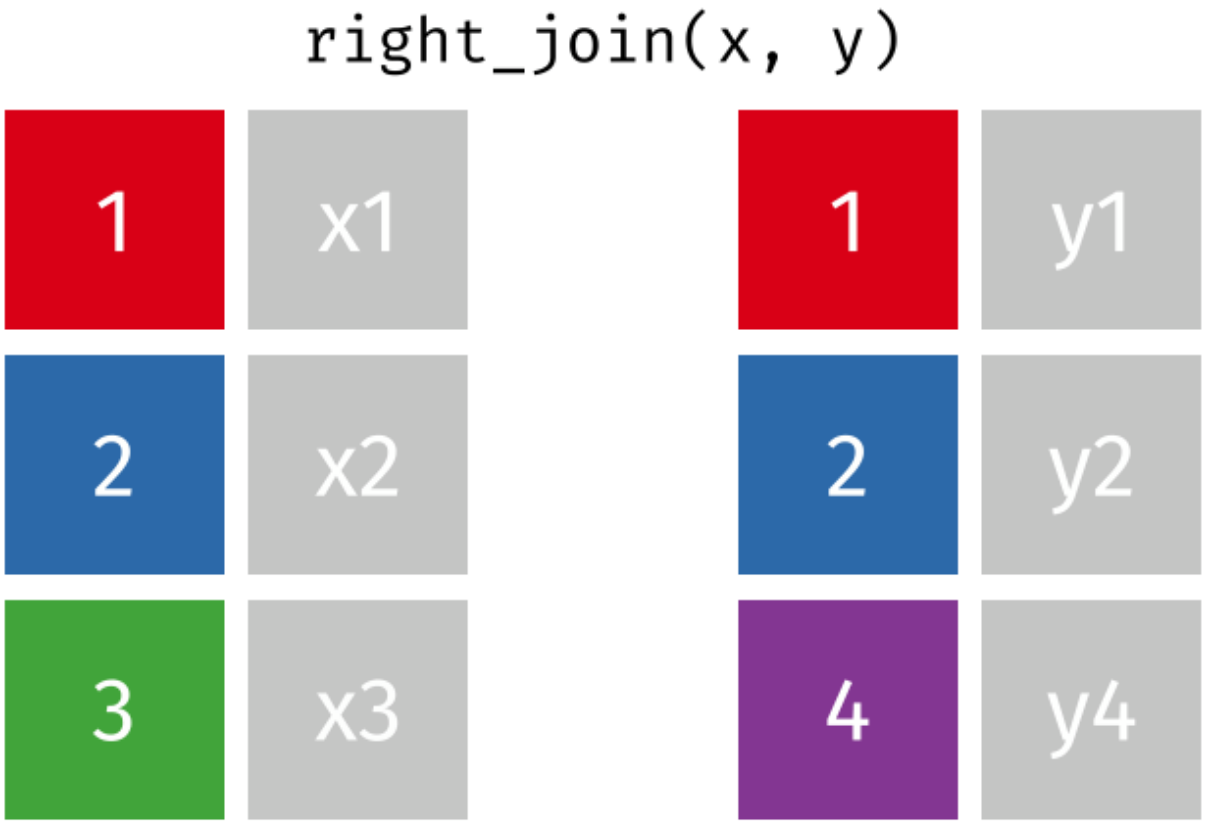
Install **tidylog** package to log outputs

```
# install.packages("tidylog")
library(tidylog)
left_join(data_As, data_cold)

Joining with `by = join_by(State)`
left_join: added one column (April_vacc_rate)
> rows only in data_As 1
> rows only in data_cold (1)
> matched rows 1
> ===
> rows total 2

# A tibble: 2 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514          NA
2 Alaska           0.627         0.626           0.623
```



Right Join

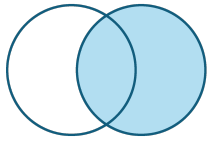
https://raw.githubusercontent.com/gadenbuie/tidyexplain/main/images/right-join.gif



Right Join

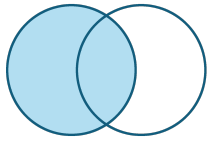
"Everything to the right of the comma"

```
rj <- right_join(data_As, data_cold)

Joining with `by = join_by(State)`
right_join: added one column (April_vacc_rate)
> rows only in data_As (1)
> rows only in data_cold 1
> matched rows 1
> ===
> rows total 2

rj

# A tibble: 2 × 4
  State  June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>           <dbl>         <dbl>           <dbl>
1 Alaska          0.627         0.626           0.623
2 Maine          NA            NA               0.795
```



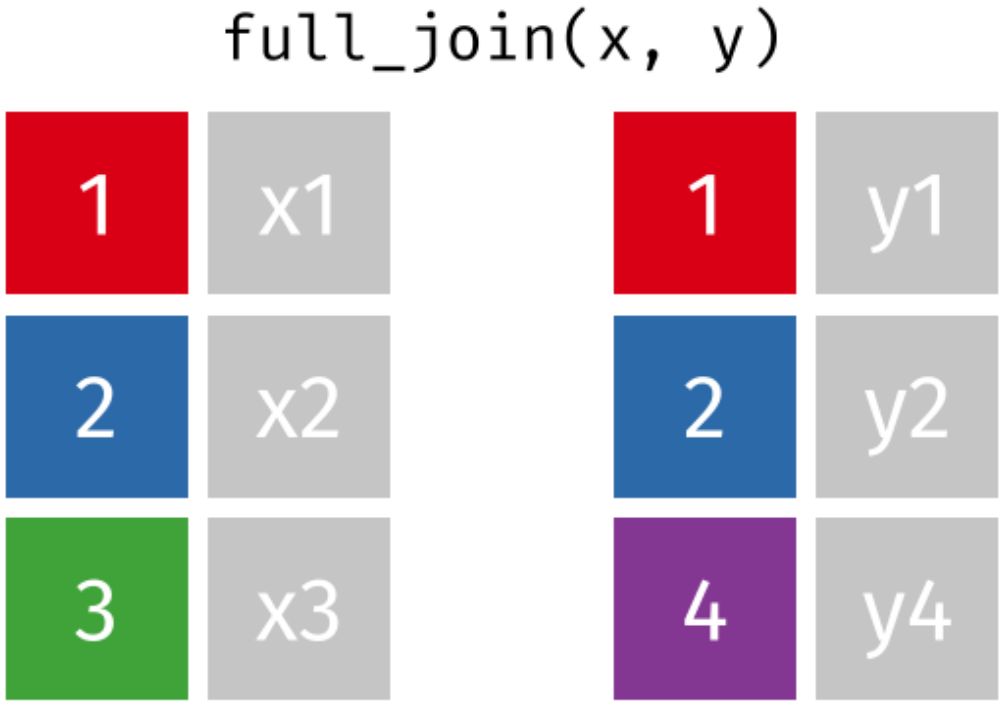
Left Join: Switching arguments

```
lj2 <- left_join(data_cold, data_As)

Joining with `by = join_by(State)`
left_join: added 2 columns (June_vacc_rate, May_vacc_rate)
> rows only in data_cold 1
> rows only in data_As (1)
> matched rows 1
> ===
> rows total 2

lj2

# A tibble: 2 × 4
  State  April_vacc_rate June_vacc_rate May_vacc_rate
  <chr>            <dbl>          <dbl>         <dbl>
1 Maine            0.795         NA            NA
2 Alaska           0.623          0.627         0.626
```



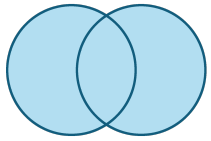
Full Join

https://raw.githubusercontent.com/gadenbuie/tidyexplain/main/images/full-join.gif



Full Join

```
fj <- full_join(data_As, data_cold)

Joining with `by = join_by(State)`
full_join: added one column (April_vacc_rate)
> rows only in data_As 1
> rows only in data_cold 1
> matched rows 1
> ===
> rows total 3

fj

# A tibble: 3 × 4
  State   June_vacc_rate May_vacc_rate April_vacc_rate
  <chr>            <dbl>         <dbl>           <dbl>
1 Alabama          0.516         0.514          NA
2 Alaska           0.627         0.626           0.623
3 Maine           NA            NA               0.795
```

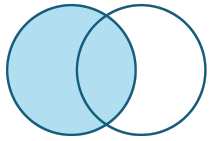


“includes duplicates”

```
data_As <- read_csv(
  file = "https://daseh.org/data/data_As_2.csv")
data_cold <- read_csv(
  file = "https://daseh.org/data/data_cold_2.csv")

data_As

# A tibble: 2 × 2
  State   state_bird
  <chr>   <chr>
1 Alabama wild turkey
2 Alaska  willow ptarmigan

data_cold

# A tibble: 3 × 3
  State  vacc_rate month
  <chr>      <dbl> <chr>
1 Maine      0.795 April
2 Alaska     0.623 April
3 Alaska     0.626 May
```

“includes duplicates”

```
lj <- left_join(data_As, data_cold)

Joining with `by = join_by(State)`
left_join: added 2 columns (vacc_rate, month)
> rows only in data_As 1
> rows only in data_cold (1)
> matched rows 2 (includes duplicates)
> ===
> rows total 3
```



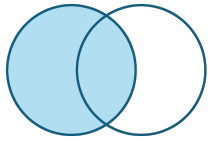
“includes duplicates”

Data including the joining column (“State”) has been duplicated.

```
lj
```

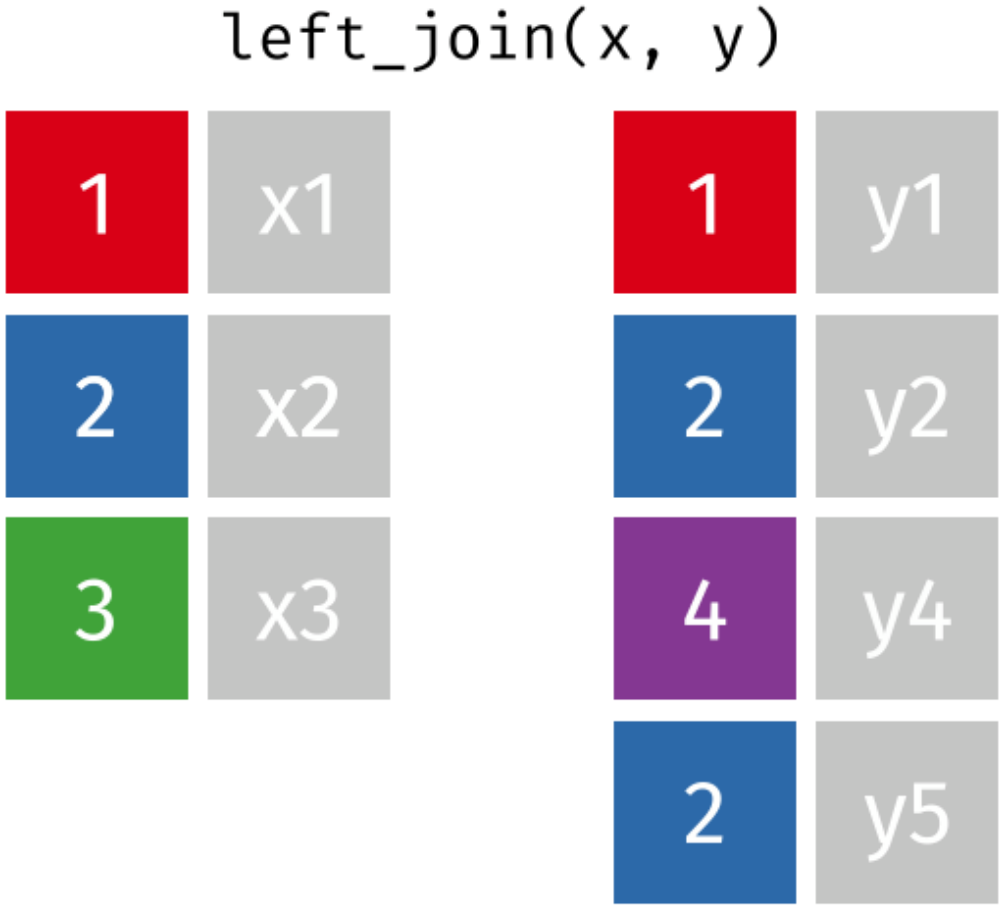
```
# A tibble: 3 × 4
  State   state_bird       vacc_rate month
  <chr>   <chr>                <dbl> <chr>
1 Alabama wild turkey         NA     <NA>
2 Alaska  willow ptarmigan     0.623 April
3 Alaska  willow ptarmigan     0.626 May
```

Note that “Alaska willow ptarmigan” appears twice.



"includes duplicates"

https://github.com/gadenbuie/tidyexplain/blob/main/images/left-join-extra.gif



Stop `tidylog`

`unloadNamespace()` does the opposite of `library()`.

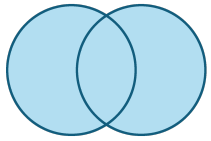
```
unloadNamespace("tidylog")
```

Using the **by** argument

By default joins use the intersection of column names. If `by` is specified, it uses that.

```
full_join(data_As, data_cold, by = "State")
```

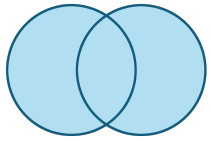
```
# A tibble: 4 × 4
  State   state_bird       vacc_rate month
  <chr>   <chr>                <dbl> <chr>
1 Alabama wild turkey             NA <NA>
2 Alaska  willow ptarmigan      0.623 April
3 Alaska  willow ptarmigan      0.626 May
4 Maine   <NA>                  0.795 April
```



Using the **by** argument

If the datasets have two different names for the same data, use `by =`. E.g.:

```
full_join(x, y, by = c("a" = "b"))
full_join(x, y, by = join_by(State == state_entity))
full_join(x, y, by = join_by(name == artist))
```



`anti_join` : what's missing

Entries in `data_As` but not in `data_cold`

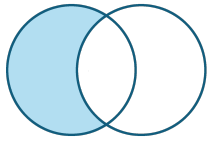
```
anti_join(data_As, data_cold, by = "State")
```

```
# A tibble: 1 × 2
  State   state_bird
  <chr>   <chr>
1 Alabama wild turkey
```

Entries in `data_cold` but not in `data_As`

```
anti_join(data_cold, data_As, by = "State") # order switched
```

```
# A tibble: 1 × 3
  State vacc_rate month
  <chr>     <dbl> <chr>
1 Maine     0.795 April
```



GUT CHECK!

Why use `join` functions?

A. Combine different data sources

B. Connect Rmd to other files

C. Using one data source is too easy and we want our analysis ~ fancy ~

Summary

- Merging/joining data sets together - assumes all column names that overlap
  - use the `by = c("a" = "b")` if they differ
- `inner_join(x, y)` - only rows that match for `x` and `y` are kept
- `full_join(x, y)` - all rows of `x` and `y` are kept
- `left_join(x, y)` - all rows of `x` are kept even if not merged with `y`
- `right_join(x, y)` - all rows of `y` are kept even if not merged with `x`
- Use the `tidylog` package for a detailed summary
- `anti_join(x, y)` shows what is only in `x` (missing from `y`)

Lab Part 2

- Class Website
- Lab
- Day 6 Cheatsheet
- Posit's `tidyr` Cheatsheet
- Posit's `dplyr` Cheatsheet

Image by Gerd Altmann from Pixabay

Additional Slides

Getting the set difference with `setdiff`

We might want to determine what indexes ARE in the first dataset that AREN'T in the second.

For this to work, the datasets need the same columns.

We'll just select the index using `select()`.

```
A_states <- data_As |> select(State)
cold_states <- data_cold |> select(State)
```

Getting the set difference with **`setdiff`**

States in `A_states` but not in `cold_states`

```
dplyr::setdiff(A_states, cold_states)
```

```
# A tibble: 1 × 1
  State
  <chr>
1 Alabama
```

States in `cold_states` but not in `A_states`

```
dplyr::setdiff(cold_states, A_states)
```

```
# A tibble: 1 × 1
  State
  <chr>
1 Maine
```

Getting the set difference with `setdiff`

Why did we use `dplyr::setdiff`?

There is a base R function, also called `setdiff` that requires vectors.

In other words, we use `dplyr::` to be specific about the package we want to use.

More set operations can be found here:
https://dplyr.tidyverse.org/reference/setops.html

Fast manipulation using **collapse** package

https://sebkrantz.github.io/collapse/

Might be helpful if your data is very large. However, `dplyr` and `tidyr` functions are great for most applications.

Data Visualization with Esquisse

Esquisse Package

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

Esquisse Package

The `esquisse` package is helpful for getting used to creating plots in R.

It is an interactive tool to help you in RStudio.

It's super **nifty**!

First, get some data..

We can use the CO heat-related ER visits dataset. This dataset contains information about the number and rate of visits for heat-related illness to ERs in Colorado from 2011-2022, adjusted for age.

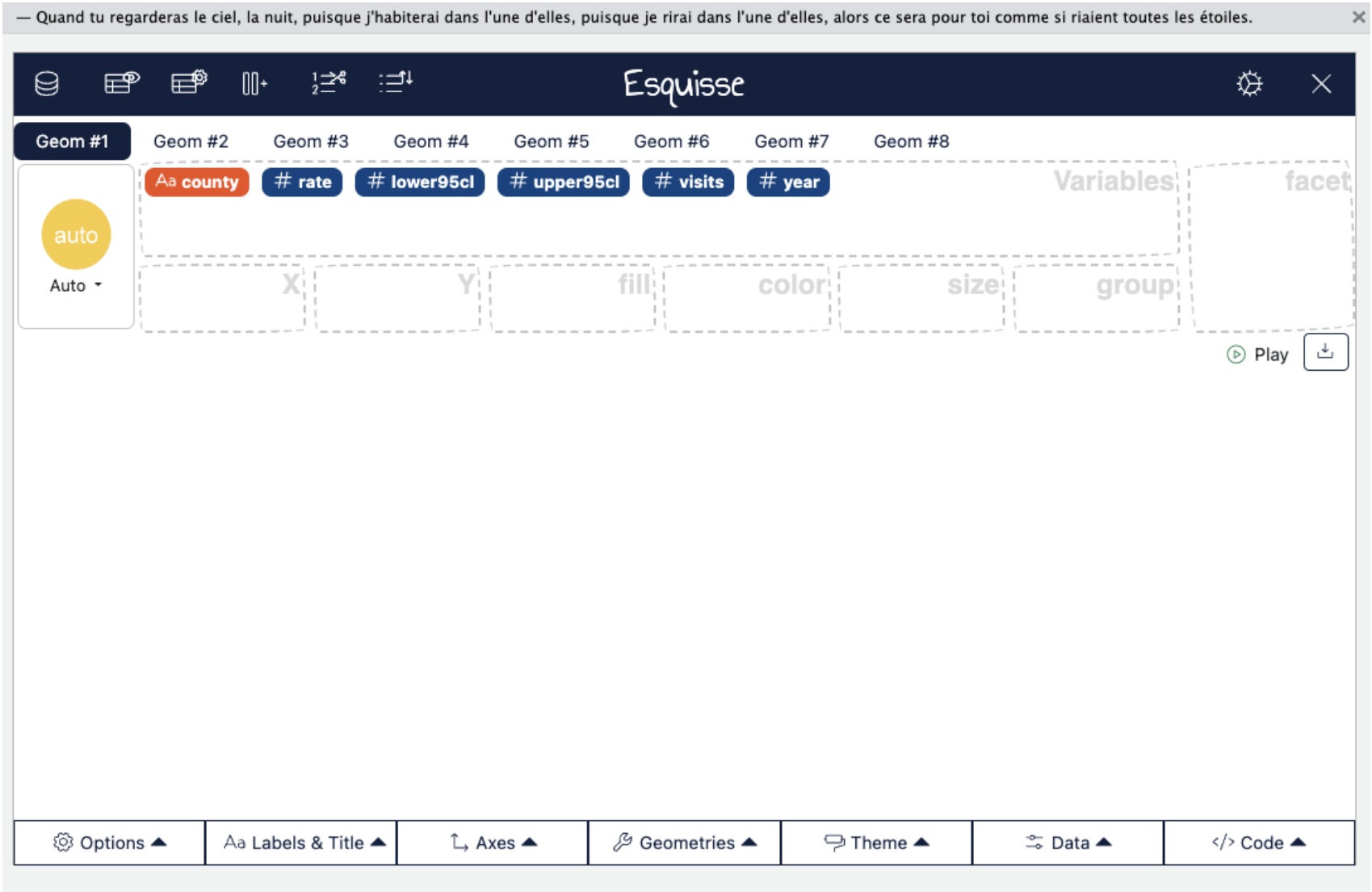
```
er <-
  read_csv("https://daseh.org/data/CO_ER_heat_visits.csv")

head(er)
```

```
## # A tibble: 6 × 6
##   county  rate lower95cl upper95cl visits  year
##   <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
## 1 Adams   6.73     NA         9.24     29  2011
## 2 Adams   4.84      2.85     NA        23  2012
## 3 Adams   6.84      4.36      9.31     31  2013
## 4 Adams   3.08      1.71      4.85     15  2014
## 5 Adams   3.36      1.89      5.23     16  2015
## 6 Adams   8.85      6.12     11.6      42  2016
```

Starting a plot

Using the `esquisser()` function you can start creating a plot for a `data.frame` or `tibble`. That's it!

```
esquisser(er)
```

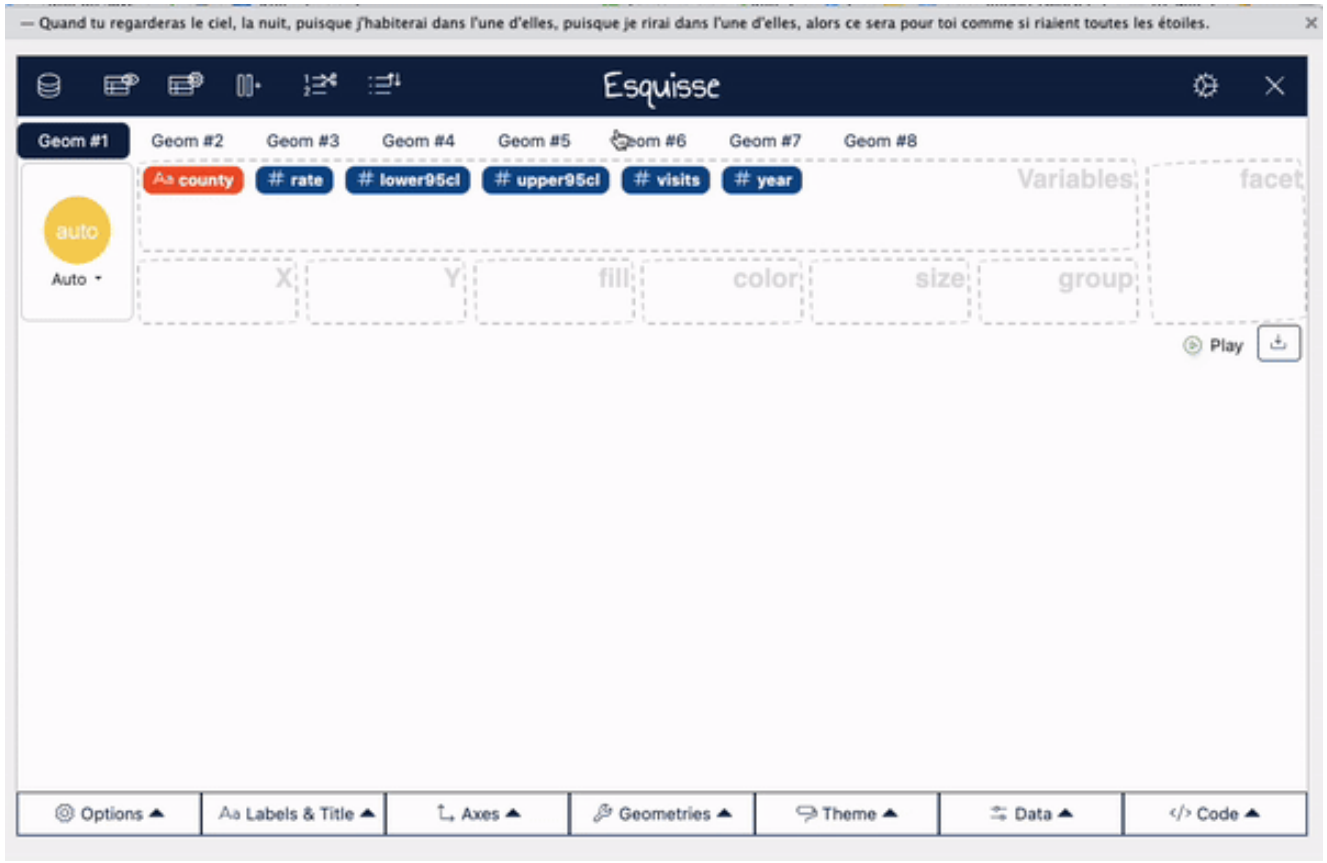


Show the plot in the browser

```
esquisse::esquisser(er, viewer = "browser")
```

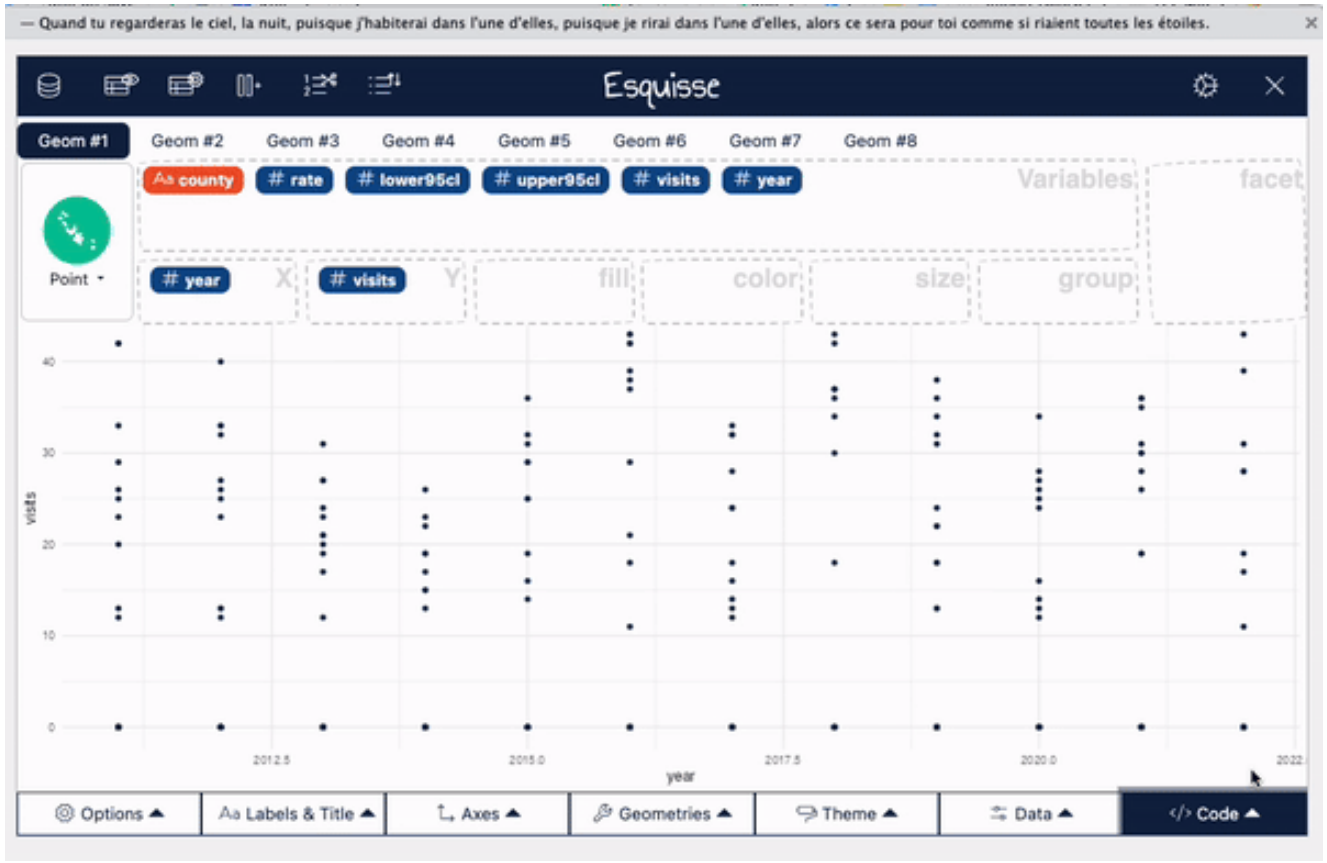
Select Variables

To select variables you can drag and drop variables to the respective axis that you would like the variable to be plotted on.



Find code

To select variables you can drag and drop variables to the respective axis that you would like the variable to be plotted on.

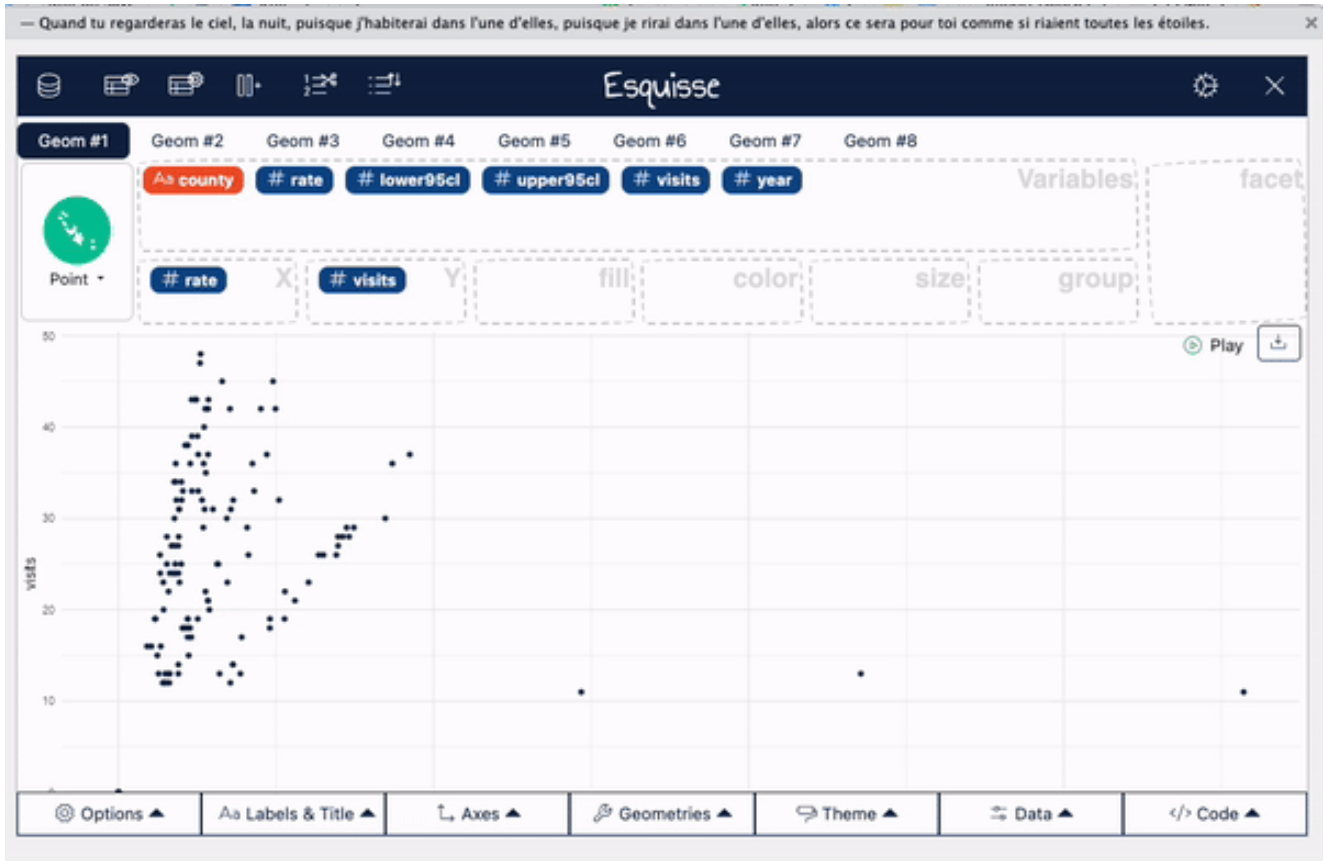


Change plot type

`esquisse` automatically assumes a plot type, but you might want to change this.

Add Facets

Facets create multiple plots based on the different values of a variable.



Add size

Sometimes it is useful to change the way points are plotted so that size represents a variable. This can especially be helpful if you need your plot to be black and white.

Add color

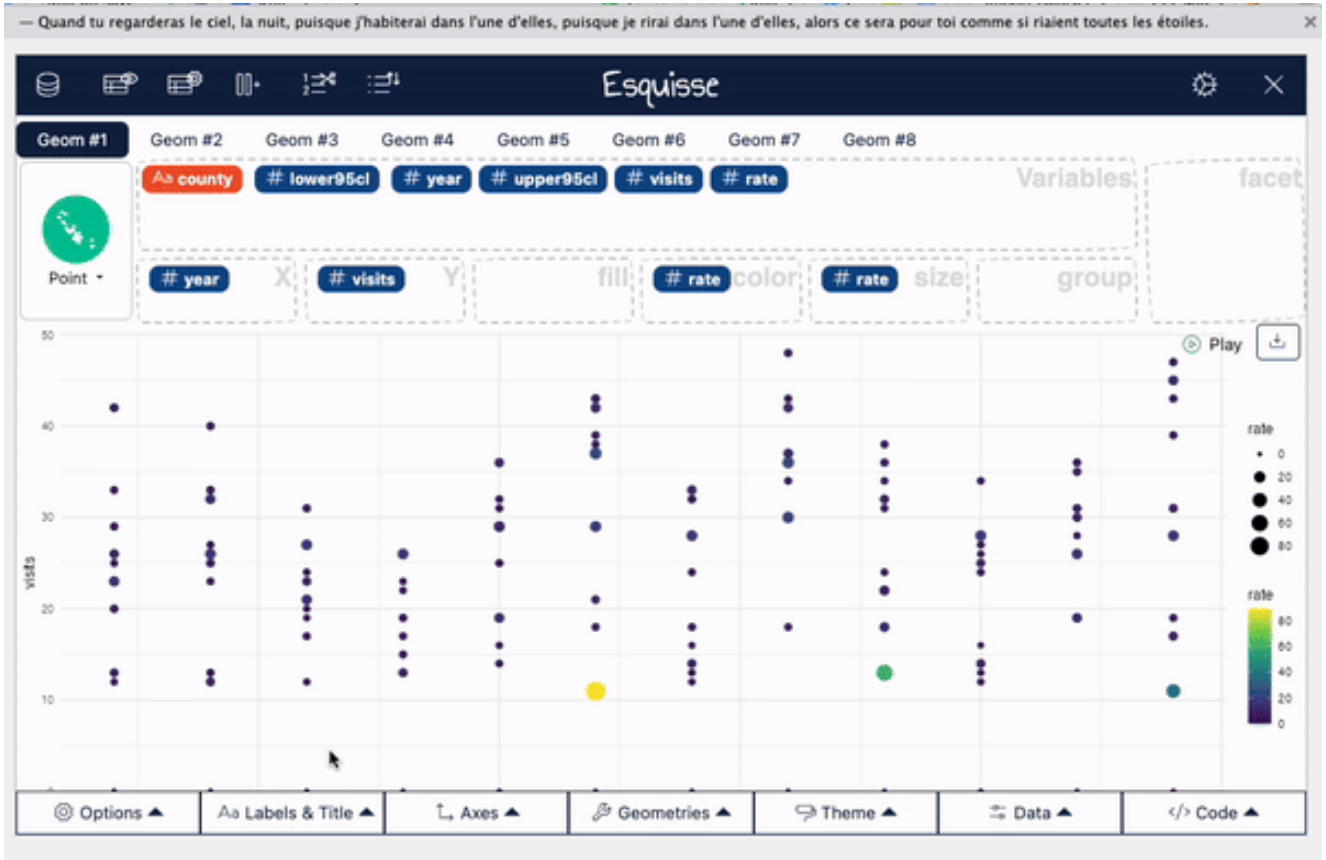
For plots with points use the color region to change coloring according to a variable. (use “fill” for bar plots)

Appearance

You can change the overall appearance with “Geometries” and “Theme”.

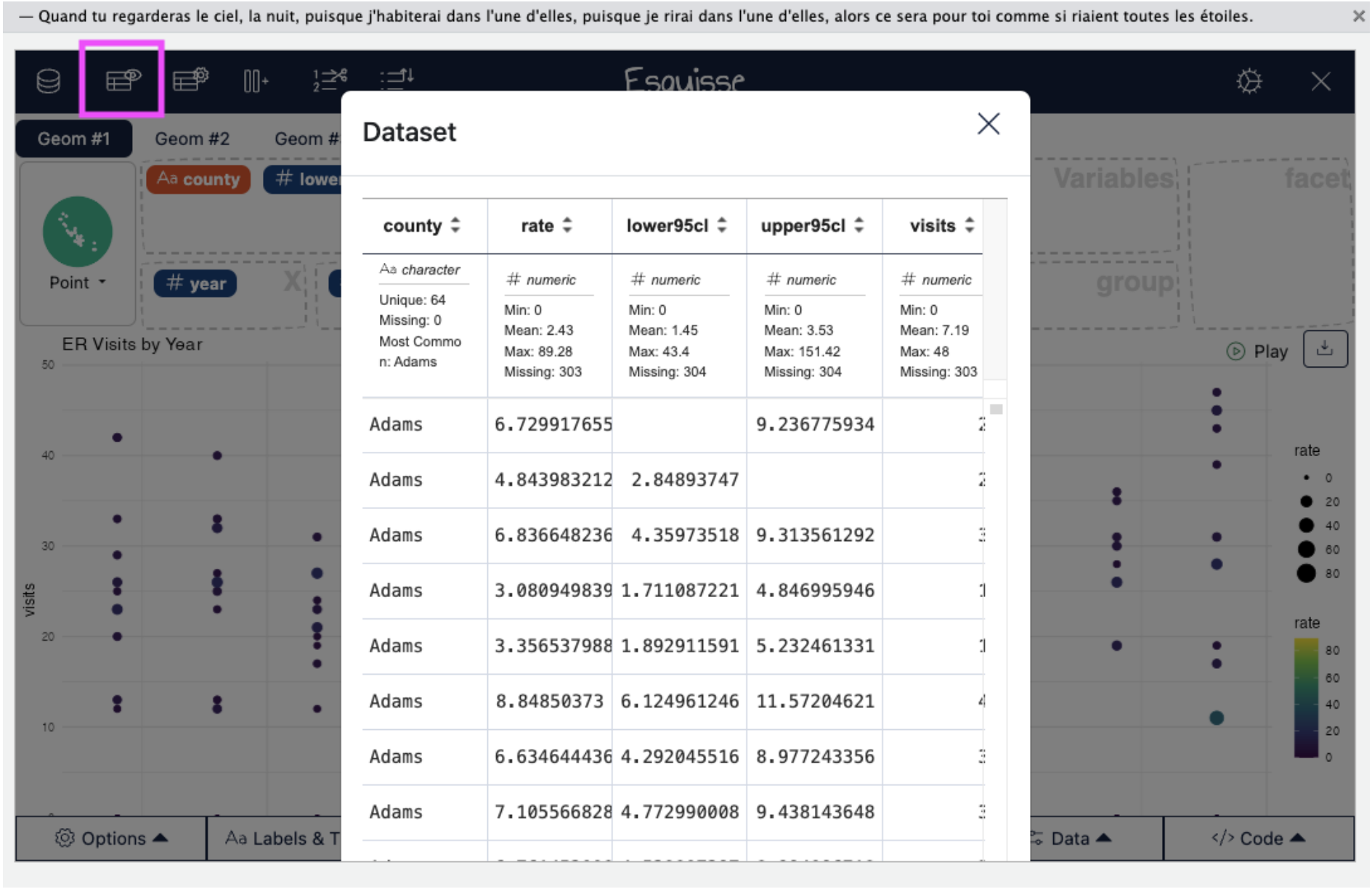
Change titles

To change titles on your plot, use the “Labels & Titles” tab.



View data

You can also easily view data

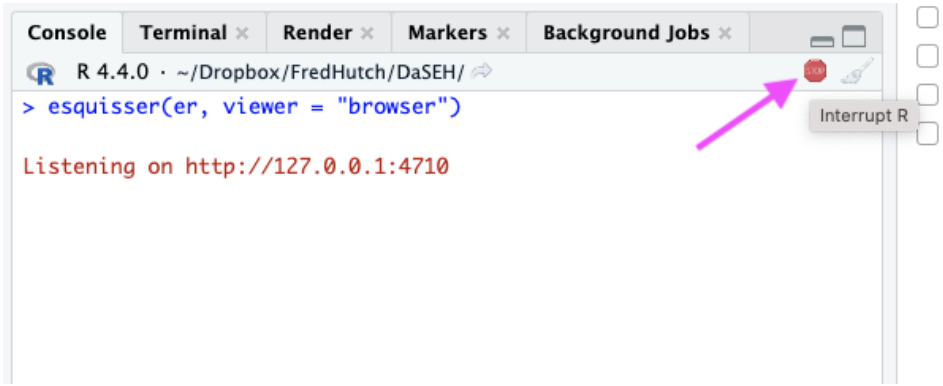


Interrupting Esquisse

You'll need to "interrupt" Esquisse to launch it with a new dataset.

1. Close the tab/window
2. Use the stop button to stop the Esquisse app

If you don't see the stop button, you need to resize your window.



Wide & Long Data ?

Let’s look at why we might want long data using Esquisse.

```
library(tidyverse)
long_er <- er |>
  select(c("county", "year", "visits"))
glimpse(long_er)
```

```
## Rows: 768
## Columns: 3
## $ county <chr> "Adams", "Adams", "Adams", "Adams", "Adams", "Adams", "Adams", …
## $ year   <dbl> 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 202…
## $ visits <dbl> 29, 23, 31, 15, 16, 42, 32, 37, 36, 24, 35, 45, 0, 0, NA, 0, NA…
```

Wide Data

As a comparison, let's also load a wide version of this dataset.

```
wide_er <- er |>
  select(county, visits, year) |>
  pivot_wider(names_from = county, values_from = visits)
```

Wide vs Long Data

```
head(long_er)
```

```
## # A tibble: 6 × 3
##   county  year visits
##   <chr>  <dbl>  <dbl>
## 1 Adams   2011     29
## 2 Adams   2012     23
## 3 Adams   2013     31
## 4 Adams   2014     15
## 5 Adams   2015     16
## 6 Adams   2016     42
```

```
head(wide_er)
```

```
## # A tibble: 6 × 65
##    year Adams Alamosa Arapahoe Archuleta  Baca  Bent Boulder Broomfield Chaff
##   <dbl> <dbl>   <dbl>    <dbl>     <dbl> <dbl> <dbl>   <dbl>      <dbl>   <db
## 1  2011    29       0       33         0     0     0      12         NA
## 2  2012    23       0       27         0     0    NA      13         NA
## 3  2013    31      NA       20         0     0     0      12         NA
## 4  2014    15       0       NA         0     0    NA      19         NA
## 5  2015    16      NA       31         0     0    NA      14         NA
## 6  2016    42      NA       39         0     0    NA      18         NA
## # … 55 more variables: Cheyenne <dbl>, `Clear Creek` <dbl>, Conejos <dbl>,
## #   Costilla <dbl>, Crowley <dbl>, Custer <dbl>, Delta <dbl>, Denver <dbl>,
## #   Dolores <dbl>, Douglas <dbl>, Eagle <dbl>, Elbert <dbl>, `El Paso` <dbl>,
## #   Fremont <dbl>, Garfield <dbl>, Gilpin <dbl>, Grand <dbl>, Gunnison <dbl>,
## #   Hinsdale <dbl>, Huerfano <dbl>, Jackson <dbl>, Jefferson <dbl>,
```

Make a plot of visits by year for different counties

```
esquisser(wide_er) # only one county at a time? Tricky!
esquisser(long_er) # county as color, visits as y, year as x!
```

GUT CHECK!

Why use Esquisse?

A. Explore your data

B. Get a “head start” on your code

C. Both of these!

Some Alternatives to **esquisse**

- `ggquickeda`: https://smouksassi.github.io/ggquickeda/
- `ggraptR`: https://github.com/cargomoose/ggraptR/
- `autoplot` can be helpful for some packages (see this blog post)

Summary

- Use the `esquisser()` function on a dataset
- Use the `viewer = "browser"` argument to launch in your browser.
- Code from Esquisse can copied into code chunks to be generated in the "Plots" pane
- It's easier if your code is in "long" form!

Lab

- Class Website
- Lab
- Day 6 Cheatsheet

Image by Gerd Altmann from Pixabay

Data Visualization

Recap

- `pivot_longer()` helps us take our data from wide to long format
  - `names_to =` gives a new name to the pivoted columns
  - `values_to =` gives a new name to the values that used to be in those columns
- `pivot_wider()` helps us take our data from long to wide format
  - `names_from` specifies the old column name that contains the new column names
  - `values_from` specifies the old column name that contains new cell values
- to merge/join data sets together need a variable in common - usually “id”

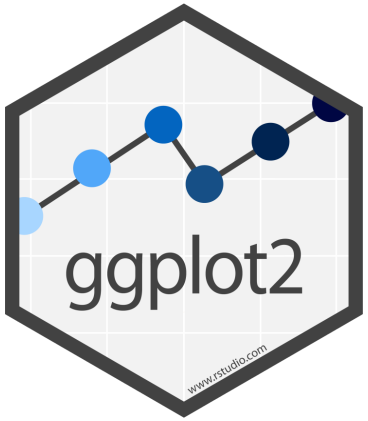
 Cheatsheet

Recap continued

- to merge/join data sets together need a variable in common - usually “id”
- `?join` - see different types of joining for `dplyr`
- `inner_join(x, y)` - only rows that match for `x` and `y` are kept
- `full_join(x, y)` - all rows of `x` and `y` are kept
- `left_join(x, y)` - all rows of `x` are kept even if not merged with `y`
- `right_join(x, y)` - all rows of `y` are kept even if not merged with `x`
- `anti_join(x, y)` - all rows from `x` not in `y` keeping just columns from `x`
- `esquisser()` function of the `esquisse` package can help make plot sketches

 Cheatsheet

esquisse and ggplot2



Why learn ggplot2?

More customization:

- branding
- making plots interactive
- combining plots

Easier plot automation (creating plots in scripts)

Faster (eventually)

ggplot2

- A package for producing graphics - gg = *Grammar of Graphics*
- Created by Hadley Wickham in 2005
- Belongs to “Tidyverse” family of packages
- *“Make a ggplot”* = Make a plot with the use of ggplot2 package

Resources:

- https://ggplot2-book.org/
- https://www.opencasestudies.org/

ggplot2

Based on the idea of:

layering

plot objects are placed on top of each other with +

 +

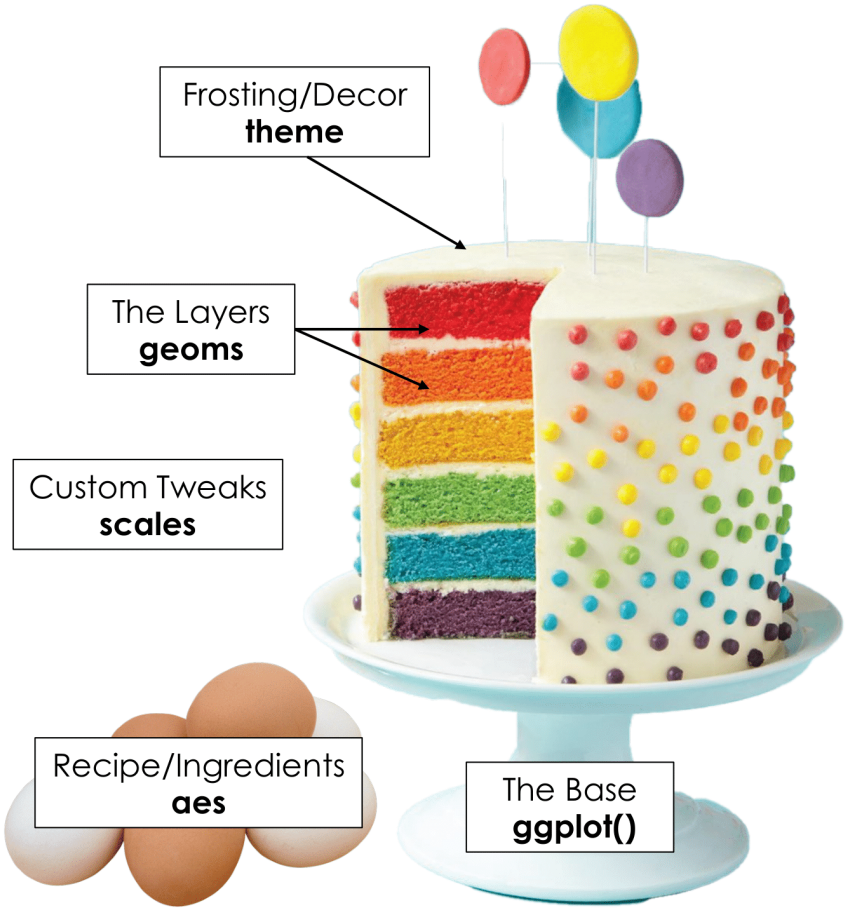
ggplot is a little bit like cake…

We *always* start by setting up the foundation with **ggplot()**

We specify our ingredients (data variables) with an **aes mapping**

We can create *layers* to our plot with **geoms**

We can style our ~~cake~~ ggplot with **themes.** We have out-of-the-box options, or we can go totally custom!



Slide Credit: Tanya Shapiro

ggplot2

- Pros: extremely powerful/flexible – allows combining multiple plot elements together, allows high customization of a look, many resources online
- Cons: ggplot2-specific “grammar of graphic” of constructing a plot
- ggplot2 gallery

Tidy data

To make graphics using `ggplot2`, our data needs to be in a **tidy** format

Tidy data:

1. Each variable forms a column.
2. Each observation forms a row.

Messy data:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.

Tidy data: example

Ideally we want each variable as a column and we want each observation in a row.

Column headers are values, not variable names:

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Table 4: The first ten rows of data on income and religion from the Pew Forum. Three columns, `$75-100k`, `$100-150k` and `>150k`, have been omitted

Now the the data is "tidy" and in long format

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Read more about tidy data and see other examples: Tidy Data tutorial

Data to plot

Let's plot the CO heat-related ER visits dataset we've been working with. First, we'll only consider data from Boulder county.

Is the data in tidy? Is it in long format?

```
er <-
  read_csv("https://daseh.org/data/CO_ER_heat_visits.csv")
er_Boulder <- er |> filter(county == "Boulder")

head(er_Boulder)
```

```
# A tibble: 6 × 6
  county   rate lower95cl upper95cl visits  year
  <chr>   <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Boulder  4.03      2.05      6.67     12  2011
2 Boulder  4.08      2.15      6.62     13  2012
3 Boulder  3.79      1.90      6.32     12  2013
4 Boulder  6.29      3.71      9.54     19  2014
5 Boulder  4.76      2.57      7.61     14  2015
6 Boulder  5.68      3.31      8.67     18  2016
```

First plot with `ggplot2` package

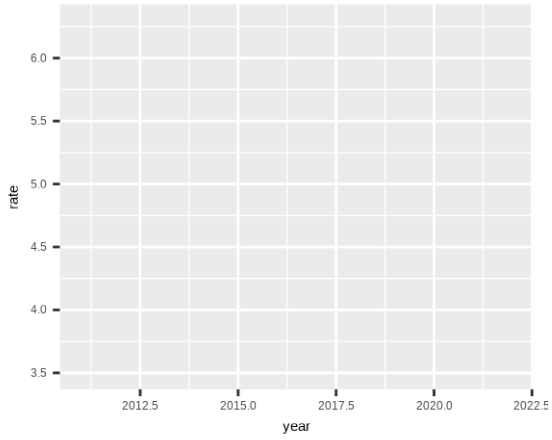
First layer of code with `ggplot2` package

Will set up the plot - it will be empty!

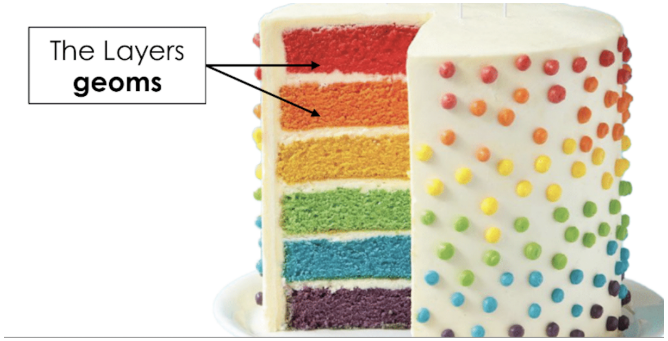
First layer of code with **ggplot2** package

- **Aesthetic mapping** `aes(x= , y =)` describes how variables in our data are mapped to elements of the plot - Note you don't need to use `mapping` but it is helpful to know what we are doing.

```
library(ggplot2) # don't forget to load ggplot2
# This is not code but shows the general format
ggplot({data_to plot}, aes(x = {var in data to plot},
                           y = {var in data to plot}))

ggplot(er_Boulder, aes(x = year, y = rate))
```



Next layer code with `ggplot2` package



There are many to choose from, to list just a few:

- `geom_point()` – points (we have seen)
- `geom_line()` – lines to connect observations
- `geom_boxplot()` – boxplots
- `geom_histogram()` – histogram
- `geom_bar()` – bar plot
- `geom_col()` – column plot
- `geom_tile()` – blocks filled with color

Next layer code with `ggplot2` package

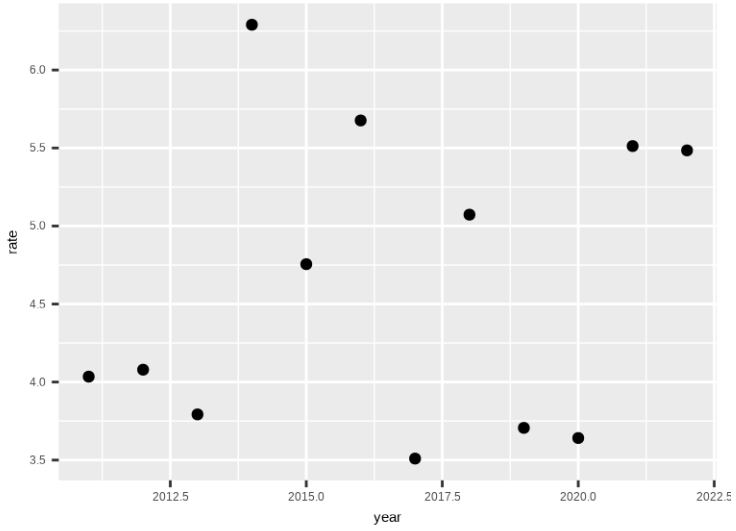
When to use what plot? A few examples:

- a scatterplot (`geom_point()`): to examine the relationship between two sets of continuous numeric data
- a barplot (`geom_bar()`): to compare the distribution of a quantitative variable (numeric) between groups or categories
- a histogram (`geom_hist()`): to observe the overall distribution of numeric data
- a boxplot (`geom_boxplot()`): to compare values between different factor levels or categories

Next layer code with `ggplot2` package

Need the + sign to add the next layer to specify the type of plot

```
ggplot({data_to plot}, aes(x = {var in data to plot},
                           y = {var in data to plot})) +
  geom_{type of plot}</div>

ggplot(er_Boulder, aes(x = year, y = rate)) +
  geom_point()
```



Tip - plus sign + must come at end of line

Having the + sign at the beginning of a line will not work!

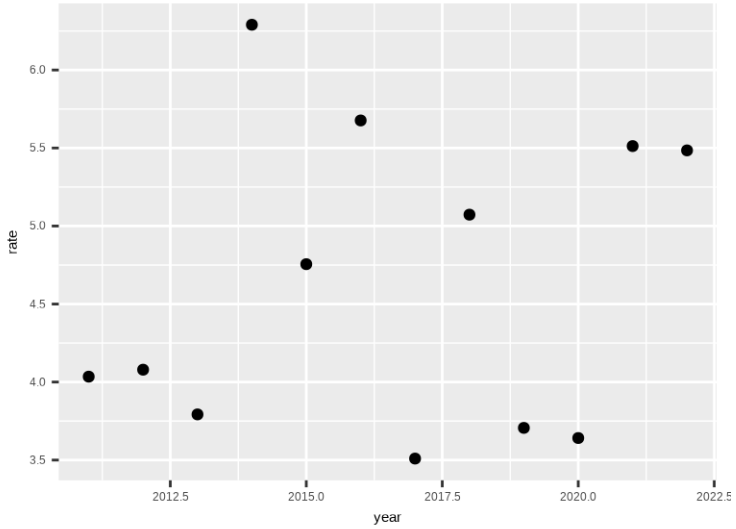
```
ggplot(er_Boulder, aes(x = year,
                       y = rate,
                       fill = item_categ))
 + geom_boxplot()
```

Pipes will also not work in place of +!

```
ggplot(er_Boulder, aes(x = year,
                       y = rate,
                       fill = item_categ))  |>
geom_boxplot()
```

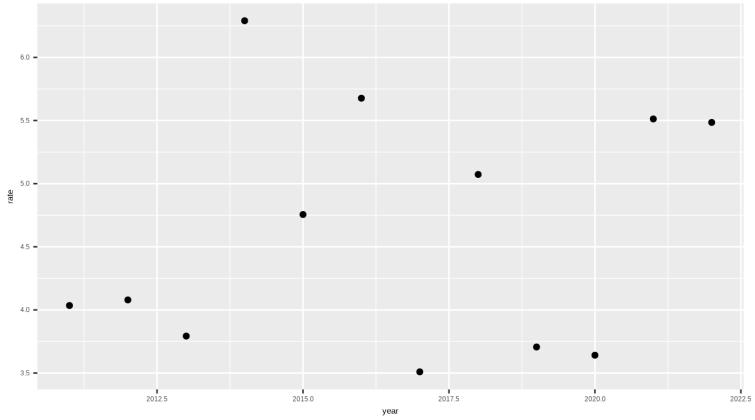
Plots can be assigned as an object

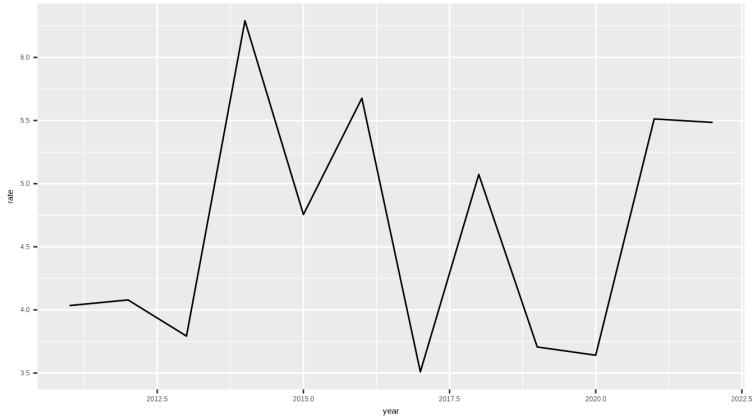
```
plt1 <- ggplot(er_Boulder, aes(x = year, y = rate)) +
          geom_point()

plt1
```



Examples of different geoms

```
plt1 <- ggplot(er_Boulder, aes(x = year, y = rate)) +
          geom_point()

plt2 <- ggplot(er_Boulder, aes(x = year, y = rate)) +
          geom_line()

plt1 # fig.show = "hold" makes plots appear
plt2 # next to one another in the chunk settings
```

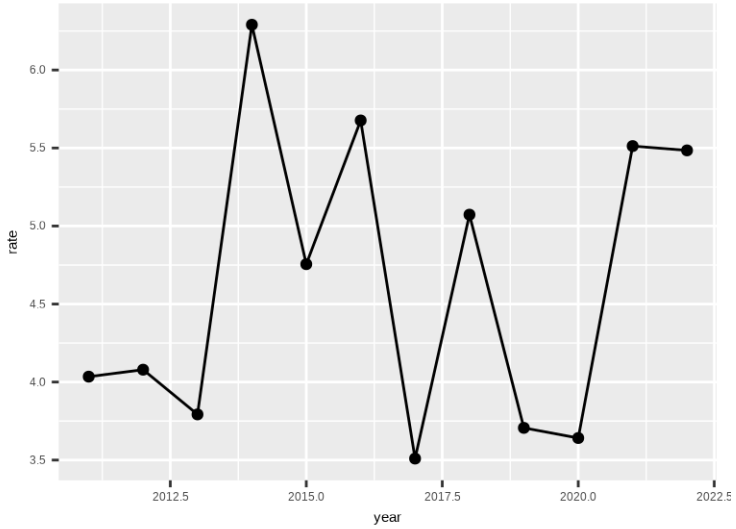




Specifying plot layers: combining multiple layers

Layer a plot on top of another plot with **+**

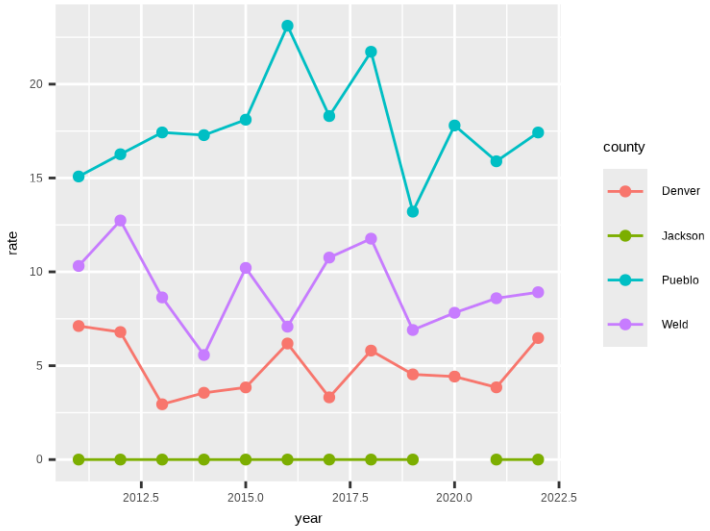
```
ggplot(er_Boulder, aes(x = year, y = rate)) +
  geom_point() +
  geom_line()
```

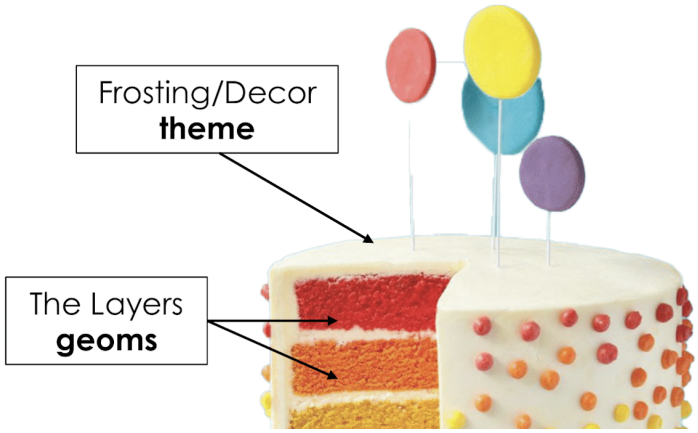


Adding color - can map color to a variable

Let’s map ER visit rates for four CO counties on the same plot

```
er_visits_4 <- er |>
  filter(county %in% c("Denver", "Weld", "Pueblo", "Jackson"))

ggplot(er_visits_4, aes(x = year, y = rate, color = county)) +
  geom_point() +
  geom_line()
```





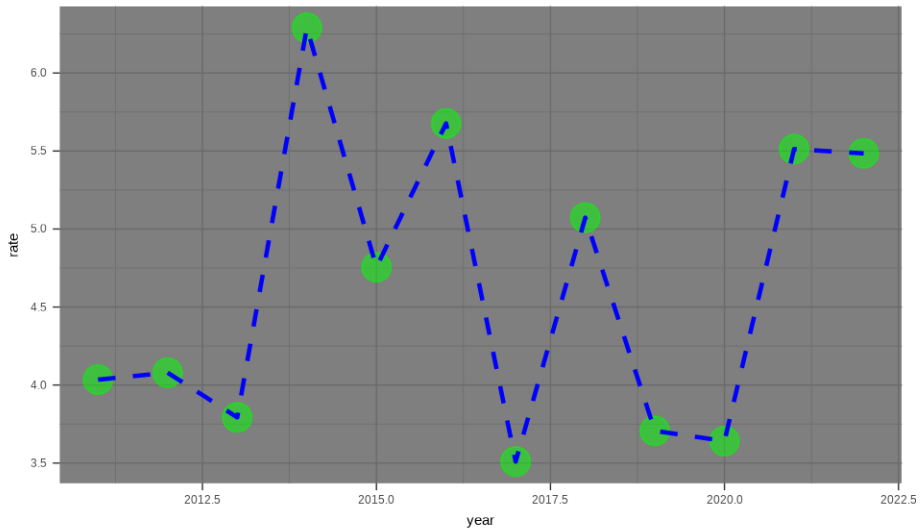
Customize the look of the plot

Customize the look of the plot

You can change the look of whole plot using `theme_*()` functions.

There are also `size`, `color`, `alpha`, and `linetype` arguments.

```
ggplot(er_Boulder, aes(x = year, y = rate)) +
  geom_point(size = 5, color = "green", alpha = 0.5) +
  geom_line(size = 0.8, color = "blue", linetype = 2) +
  theme_dark()
```



More themes!

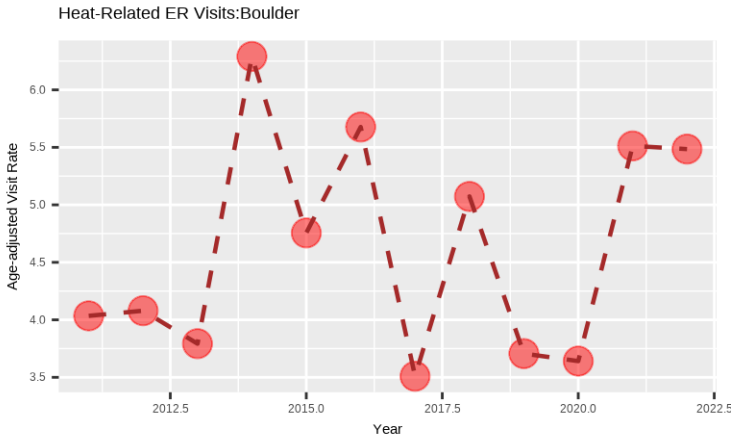
There's not only the built in ggplot2 themes but all kinds of themes from other packages!

- ggthemes
- ThemePark package
- hrbr themes

Adding labels

The `labs()` function can help you add or modify titles on your plot. The `title` argument specifies the title. The `x` argument specifies the x axis label. The `y` argument specifies the y axis label.

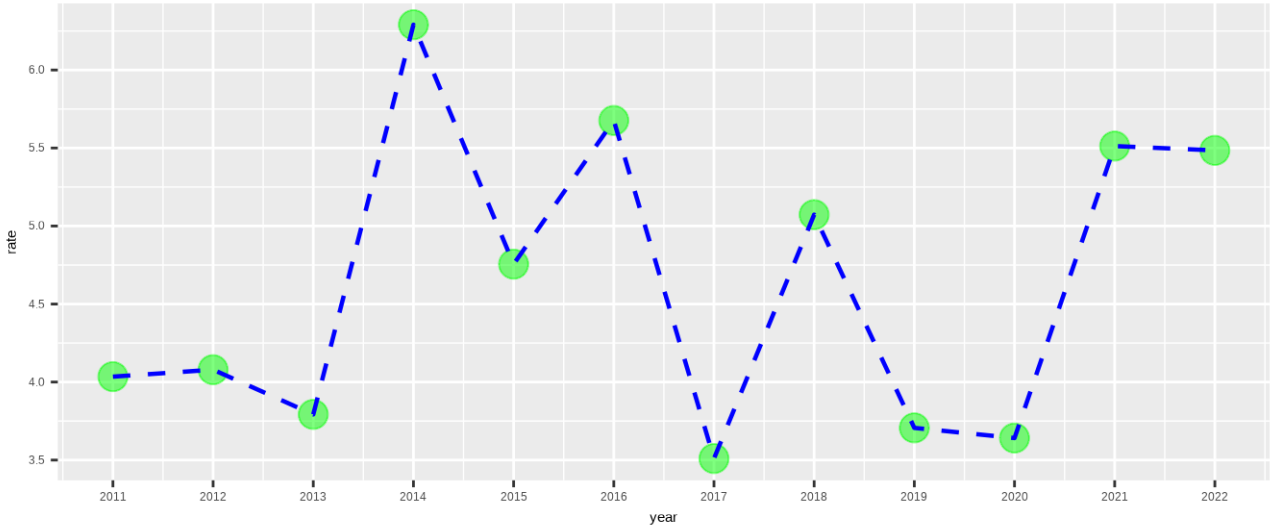
```
ggplot(er_Boulder, aes(x = year, y = rate)) +
            geom_point(size = 5, color = "red", alpha = 0.5) +
            geom_line(size = 0.8, color = "brown", linetype = 2) +
            labs(title = "Heat-Related ER Visits:Boulder",
              x = "Year",
              y = "Age-adjusted Visit Rate")
```



Changing axis: specifying axis scale

`scale_x_continuous()` and `scale_y_continuous()` can change how the axis is plotted. Can use the `breaks` argument to specify how you want the axis ticks.

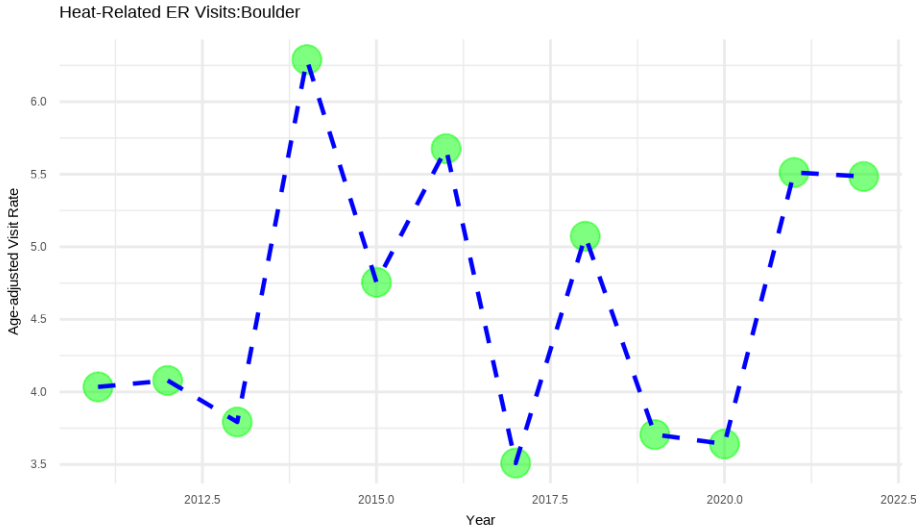
```
plot_scale <- ggplot(er_Boulder, aes(x = year, y = rate)) +
                geom_point(size = 5, color = "green", alpha = 0.5) +
                geom_line(size = 0.8, color = "blue", linetype = 2) +
                scale_x_continuous(breaks = seq(from = 2011, to = 2022, by = 1))
plot_scale
```



Modifying plot objects

You can add to a plot object to make changes! Note that we can save our plots as an object like `plt1` below. And now if we reference `plt1` again our plot will print out!

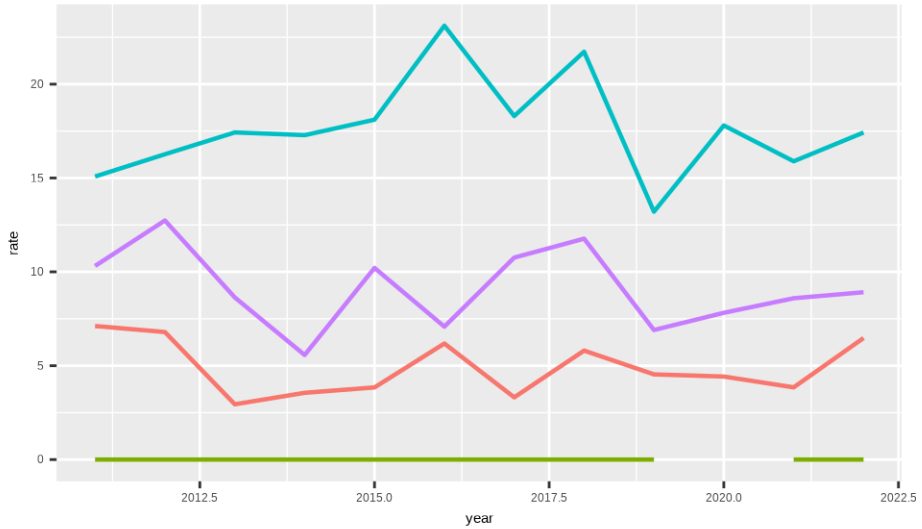
```
plt1 <- ggplot(er_Boulder, aes(x = year, y = rate,)) +
            geom_point(size = 5, color = "green", alpha = 0.5) +
  geom_line(size = 0.8, color = "blue", linetype = 2) +
  labs(title = "Heat-Related ER Visits:Boulder", x = "Year", y = "Age-adjusted Visit Rate")

plt1 + theme_minimal()
```



Removing the legend label

You can use `theme(legend.position = "none")` to remove the legend.

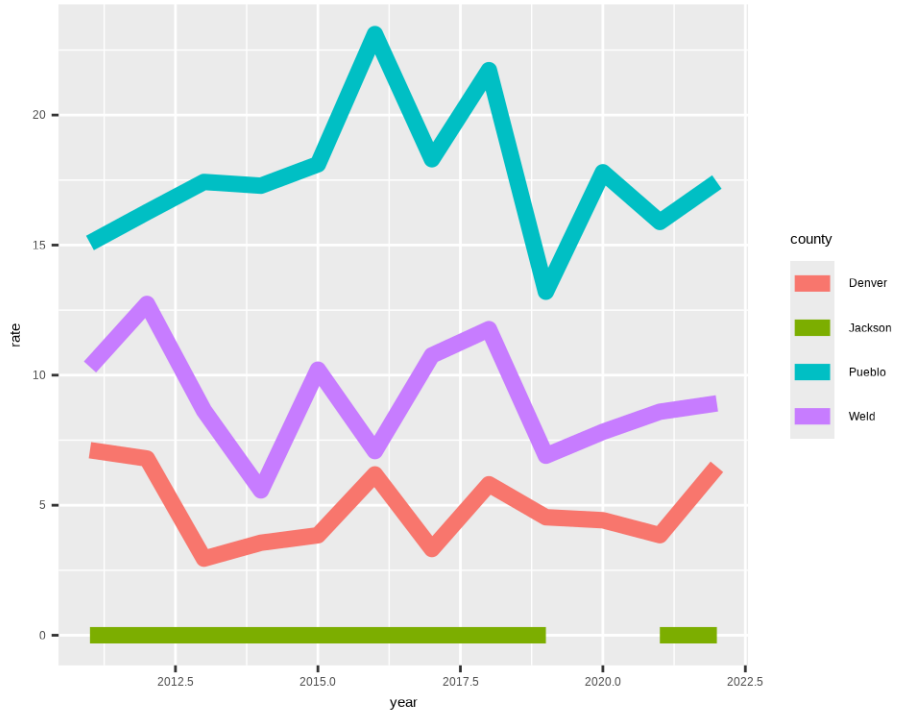
```
er_visits_4 |> ggplot(aes(x = year,
                         y = rate,
                         color = county)) +
  geom_line(size = 0.8) +
  theme(legend.position = "none")
```



Changes for a specific layer

It's possible to go in and change specifications with newer layers. Here is our original plot.

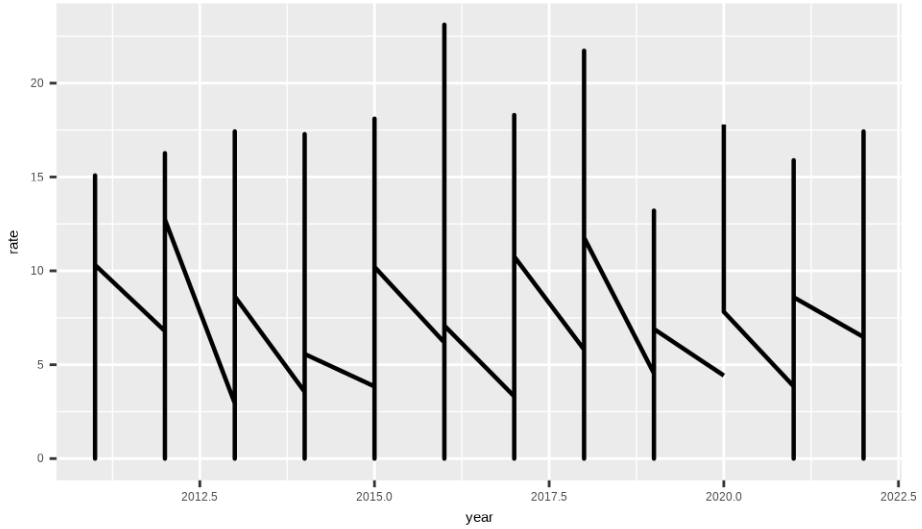
```
er_visits_4 |> ggplot(aes(x = year,
                         y = rate,
                         color = county)) +
  geom_line(size = 3)
```



Overwriting specifications

It's possible to go in and change specifications with newer layers.

```
er_visits_4 |> ggplot(aes(x = year,
                         y = rate,
                         color = county)) +
  geom_line(size = 0.8, color = "black")
```



This plot looks a bit odd, we will talk about that soon!

GUT CHECK: If we get an empty plot what might we need to do?

A. Add a `plot_` layer like `plot_point()`

B. Add a `geom_` layer like `geom_point()`

GUT CHECK: How do we add more layers in ggplot2 plots?

A. `|>`

B. `&`

C. `+`

Summary

- `ggplot()` specifies what data use and what variables will be mapped to where
- inside `ggplot()`, `aes(x = , y = , color =)` specify what variables correspond to what aspects of the plot in general
- layers of plots can be combined using the `+` at the **end** of lines
- special `theme_*()` functions can change the overall look
- individual layers can be customized using arguments like: `size`, `color` `alpha` (more transparent is closer to 0), and `linetype`
- labels can be added with the `labs()` function and `x`, `y`, `title` arguments
- `scale_x_continuous()` and `scale_y_continuous()` can modify the scale of the axes
- by default, `ggplot()` removes points with missing values from plots.

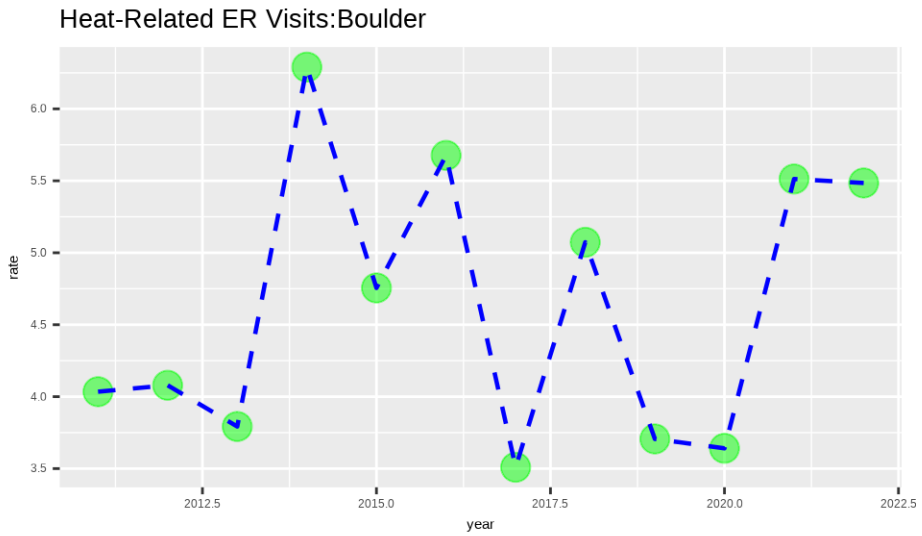
Lab 1

- Class Website
- Lab
- Day 6 Cheatsheet

theme() function:

The `theme()` function can help you modify various elements of your plot. Here we will adjust the font size of the plot title.

```
ggplot(er_Boulder, aes(x = year, y = rate)) +
            geom_point(size = 5, color = "green", alpha = 0.5) +
            geom_line(size = 0.8, color = "blue", linetype = 2) +
  labs(title = "Heat-Related ER Visits:Boulder") +
  theme(plot.title = element_text(size = 20))
```



theme() function

The `theme()` function always takes:

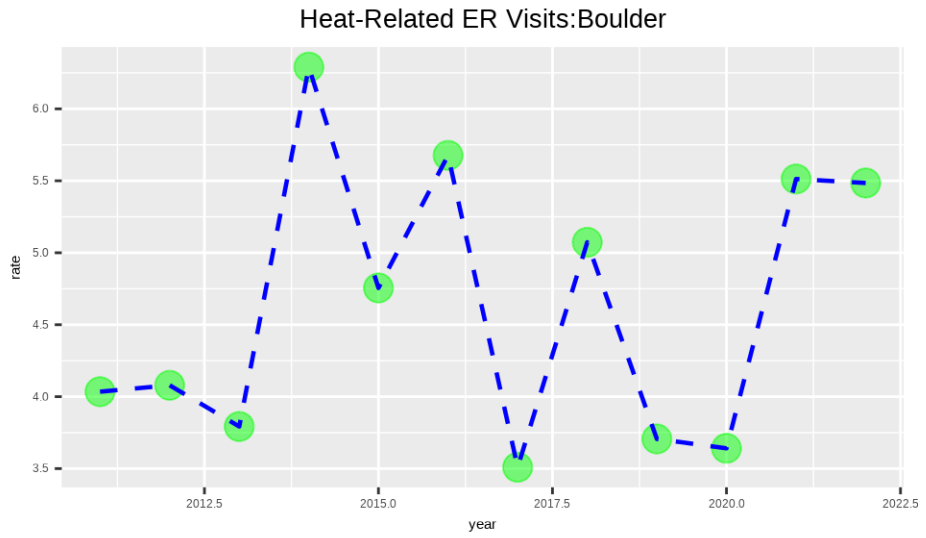
1. an object to change (use `?theme()` to see - `plot.title`, `axis.title`, `axis.ticks` etc.)
2. the aspect you are changing about this: `element_text()`, `element_line()`, `element_rect()`, `element_blank()`
3. what you are changing:
   - text: `size`, `color`, `fill`, `face`, `alpha`, `angle`, `hjust` (horizontal justification)
   - position: `"top"`, `"bottom"`, `"right"`, `"left"`, `"none"`
   - rectangle: `size`, `color`, `fill`, `linetype`
   - line: `size`, `color`, `linetype`

```
theme(plot.title = element_text(size = 20, hjust = 0.5))
```

theme() function: center title and change size

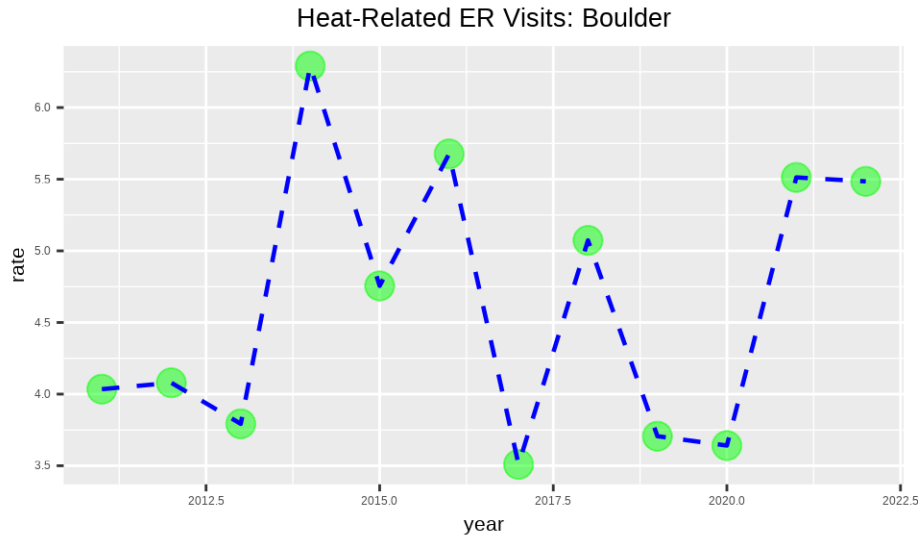
The `theme()` function can help you modify various elements of your plot. Here we will adjust the horizontal justification (`hjust`) of the plot title.

```
ggplot(er_Boulder, aes(x = year, y = rate)) +
            geom_point(size = 5, color = "green", alpha = 0.5) +
            geom_line(size = 0.8, color = "blue", linetype = 2) +
  labs(title = "Heat-Related ER Visits:Boulder") +

  theme(plot.title = element_text(size = 20, hjust = 0.5))
```



theme() function: change title and axis format

```
ggplot(er_Boulder, aes(x = year, y = rate)) +
            geom_point(size = 5, color = "green", alpha = 0.5) +
            geom_line(size = 0.8, color = "blue", linetype = 2) +
  labs(title = "Heat-Related ER Visits: Boulder") +
  theme(plot.title = element_text(size = 20, hjust = 0.5),
        axis.title = element_text(size = 16))
```

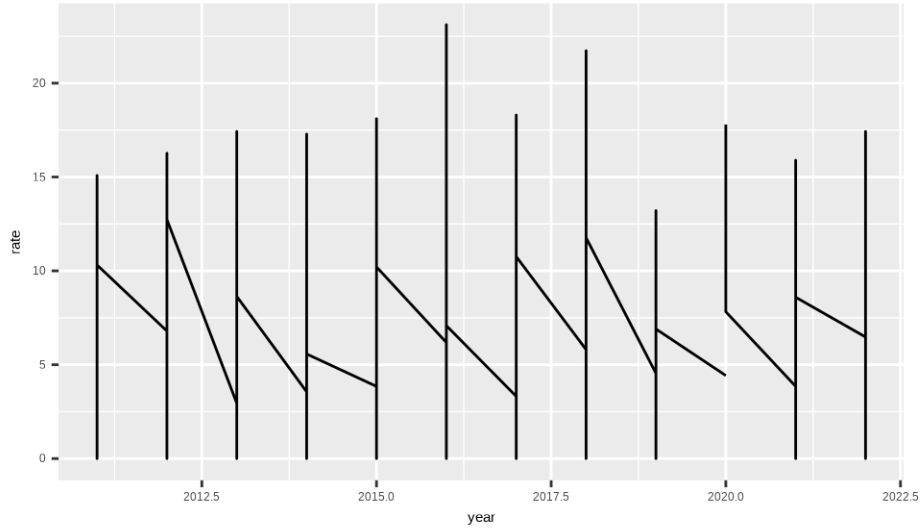


Cheatsheet about theme

Theme Cheatsheet

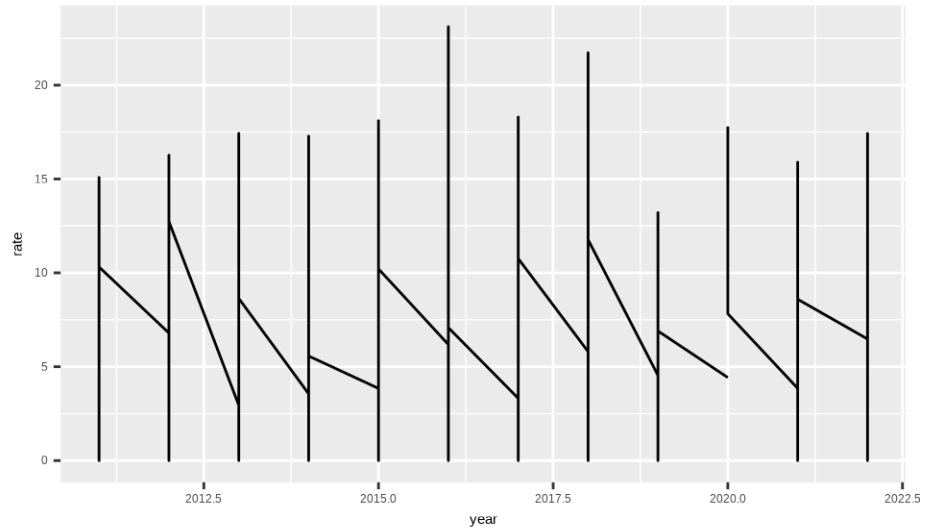
Let's take a look at that odd plot again

```
ggplot(er_visits_4, aes(x = year,
                        y = rate, color = county)) +
  geom_line(color = "black")
```



This is equivalent

```
ggplot(er_visits_4, aes(x = year,
                        y = rate)) +
  geom_line()
```

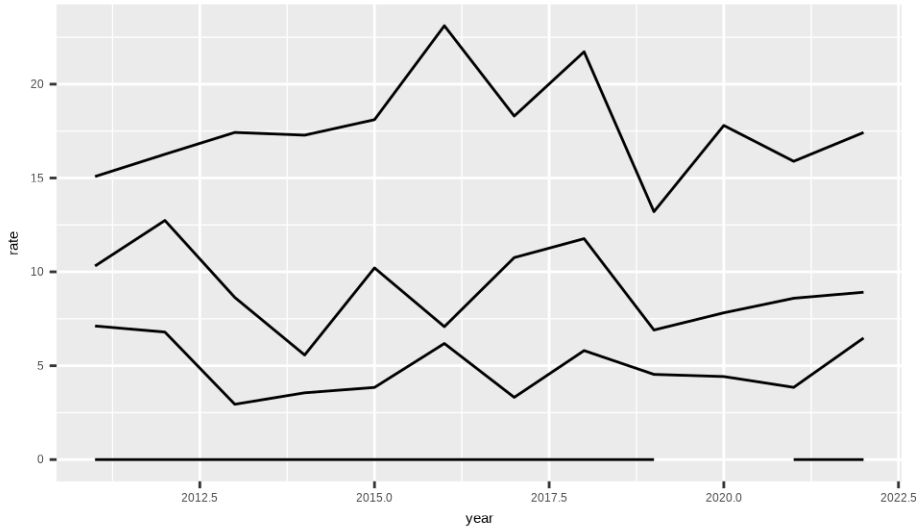


If it looks confusing to you, there might be something wrong

Using **group** in plots

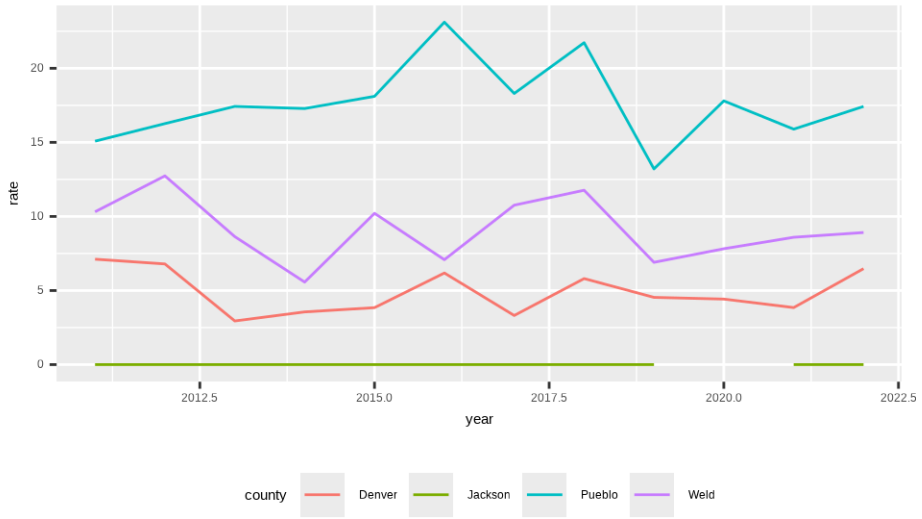
You can use `group` element in a mapping to indicate that each `county` will have a rate line.

```
ggplot(er_visits_4, aes(x = year,
                        y = rate,
                        group = county)) +
  geom_line()
```



Adding color will automatically group the data

```
ggplot(er_visits_4, aes(x = year,
                        y = rate,
                    color = county)) +
  geom_line()+
  theme(legend.position = "bottom")
```



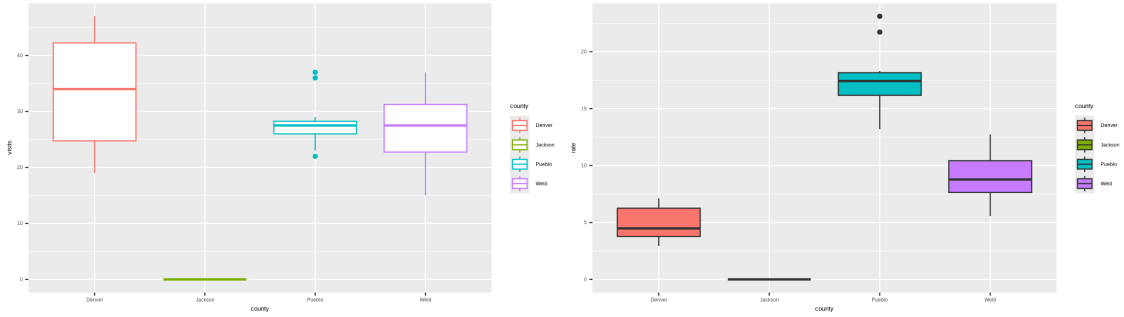
Tips!

Let’s talk additional tricks and tips for making ggplots!

Tips - Color vs Fill

- `color` is needed for points and lines
- `fill` is generally needed for boxes and bars

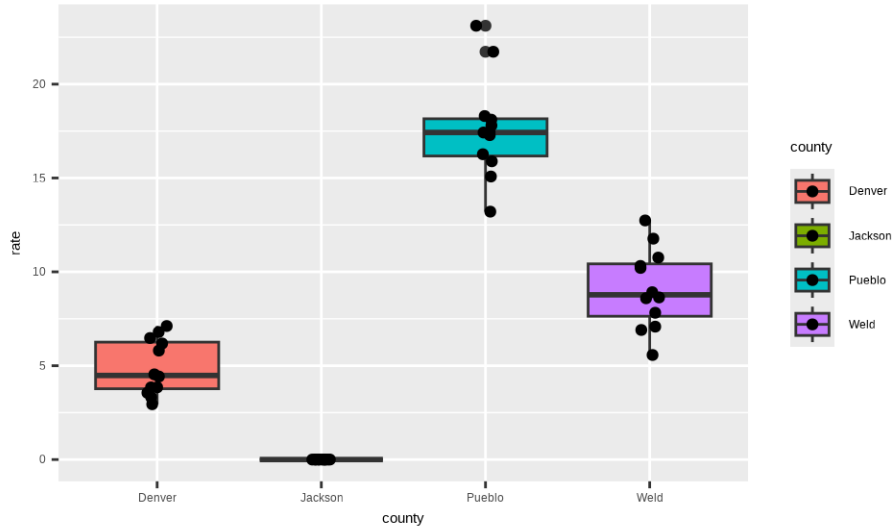
```
ggplot(er_visits_4, aes(x = county,
                        y = visits,
                        color = county)) + #color creates an outline
  geom_boxplot()

ggplot(er_visits_4, aes(x = county,
                        y = rate,
                        fill = county)) + # fills the boxplot
  geom_boxplot()
```



Tip - Good idea to add jitter layer to top of box plots

Can add `width` argument to make the jitter more narrow.

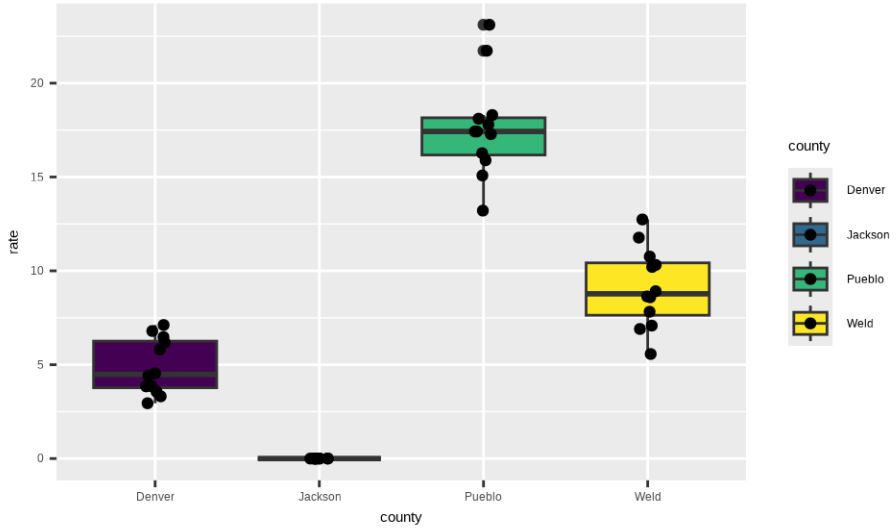
```
ggplot(er_visits_4, aes(x = county,
                        y = rate,
                        fill = county)) +
  geom_boxplot() +
  geom_jitter(width = .06)
```



Tip - be careful about colors for color vision deficiency

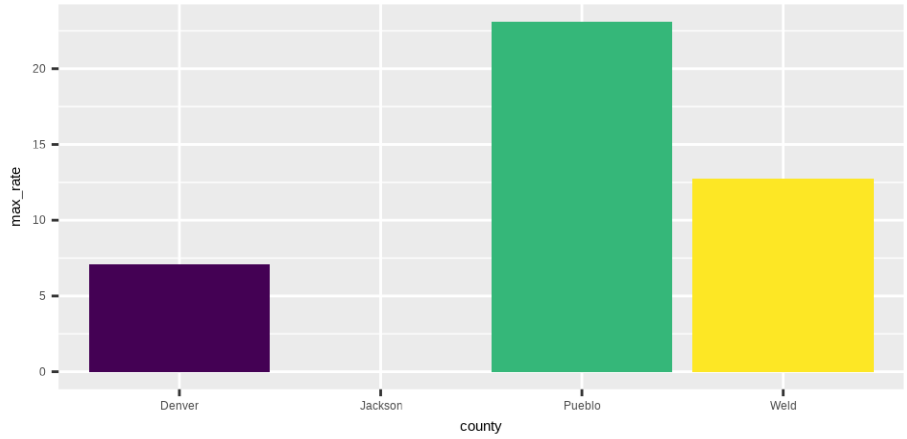
`scale_fill_viridis_d()` for discrete /categorical data
`scale_fill_viridis_c()` for continuous data

```
ggplot(er_visits_4, aes(x = county,
                        y = rate,
                        fill = county)) +
  geom_boxplot() +
  geom_jitter(width = .06) +
  scale_fill_viridis_d()
```



Tip - can pipe data after wrangling into ggplot()

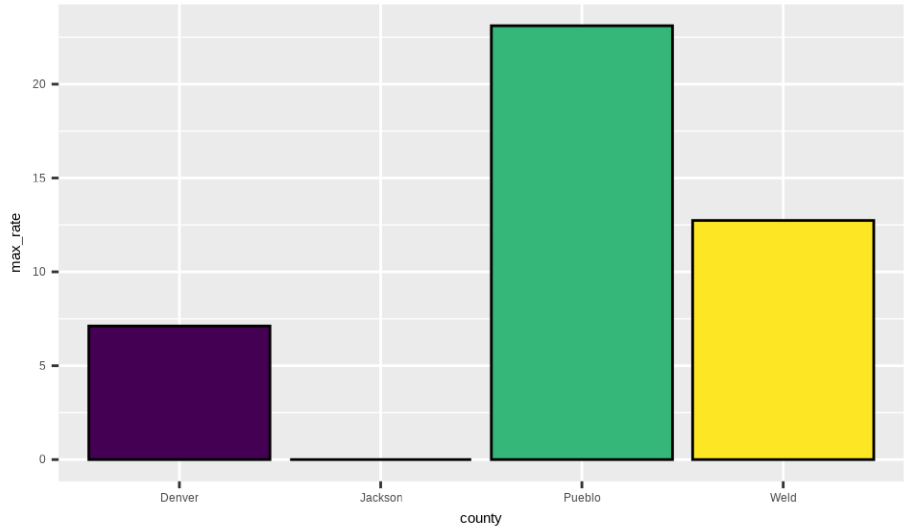
```
er_bar <- er_visits_4 |>
  group_by(county) |>
  summarize("max_rate" = max(rate, na.rm=T)) |>

ggplot(aes(x = county,
           y = max_rate,
           fill = county)) +
  scale_fill_viridis_d()+
  geom_col() +
  theme(legend.position = "none")

er_bar
```



Tip - color outside of **aes( )**

Can be used to add an outline around column/bar plots. Remember the columns were colored inside with `fill`.

```
er_bar +
   geom_col(color = "black")
```



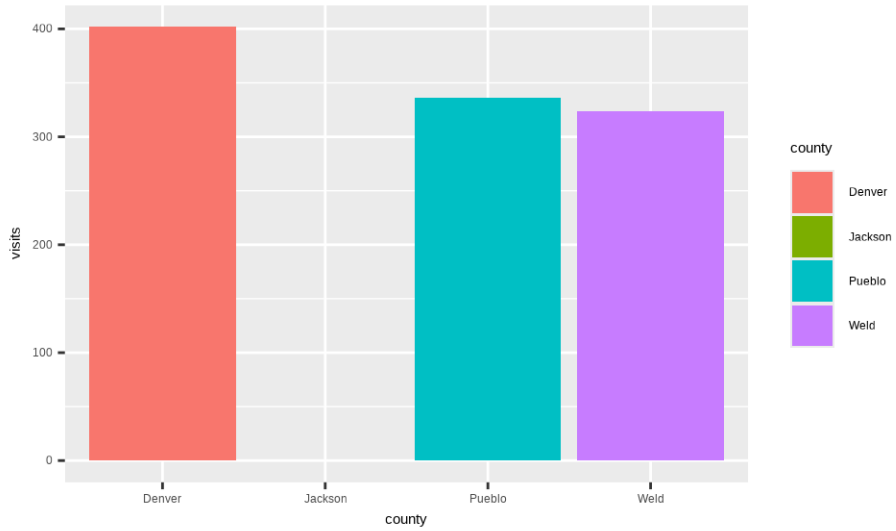
Tip - col vs bar

`geom_bar(x =)` can only use one `aes` mapping `geom_col(x = , y = )` can have two

Tip - Check what you plot

- May not be plotting what you think you are!

```
ggplot(er_visits_4, aes(x = county,
                        y = visits,
                        fill = county)) +
  geom_col()
```



What did we plot? Always good to check it is correct!

```
head(er_visits_4, n = 3)
```

```
# A tibble: 3 × 6
  county  rate lower95cl upper95cl visits  year
  <chr>  <dbl>     <dbl>     <dbl>  <dbl> <dbl>
1 Denver  7.11      4.89      9.34     42  2011
2 Denver  6.79      4.62      8.97     40  2012
3 Denver  2.95      1.75      4.46     19  2013
```

```
er_visits_4 |> group_by(county) |>
  summarize(sum = sum(visits, na.rm=T))
```

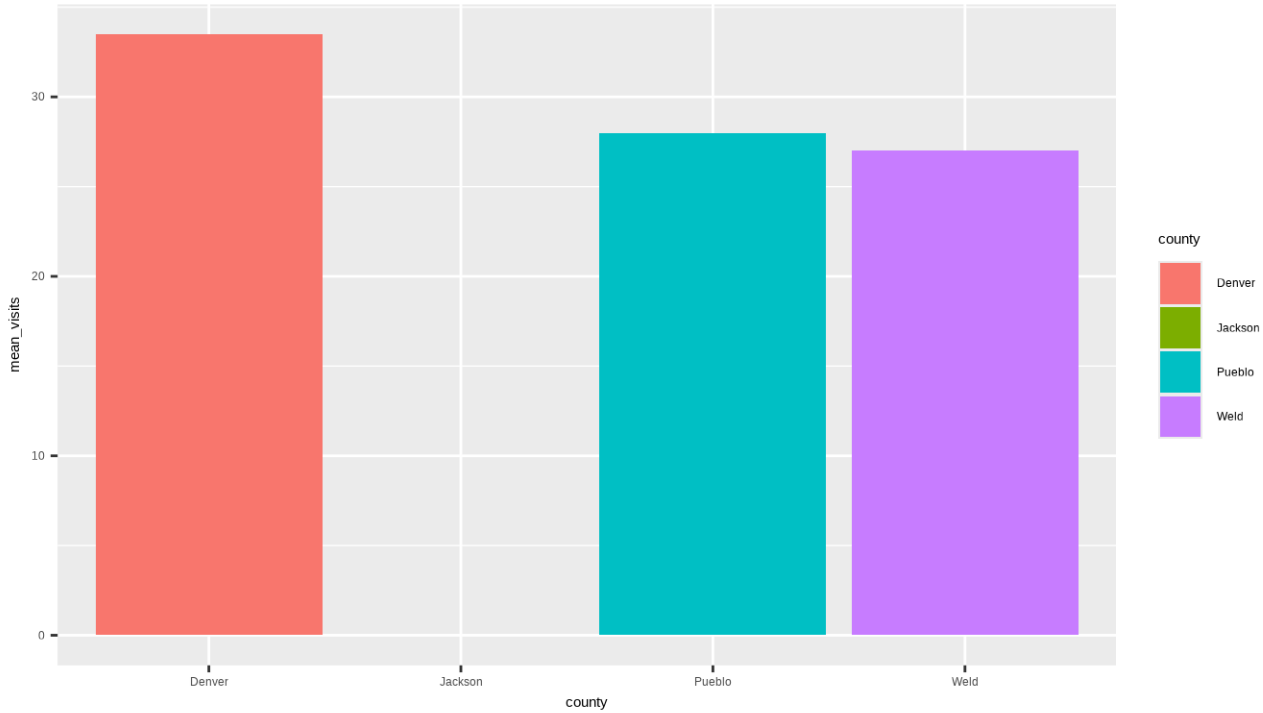
```
# A tibble: 4 × 2
  county    sum
  <chr>   <dbl>
1 Denver    402
2 Jackson     0
3 Pueblo    336
4 Weld      324
```

Try that again

```
er_visits_4 |> group_by(county) |>
  summarize(mean_visits = mean(visits, na.rm=T))

# A tibble: 4 × 2
  county  mean_visits
  <chr>         <dbl>
1 Denver         33.5
2 Jackson         0
3 Pueblo         28
4 Weld           27
```

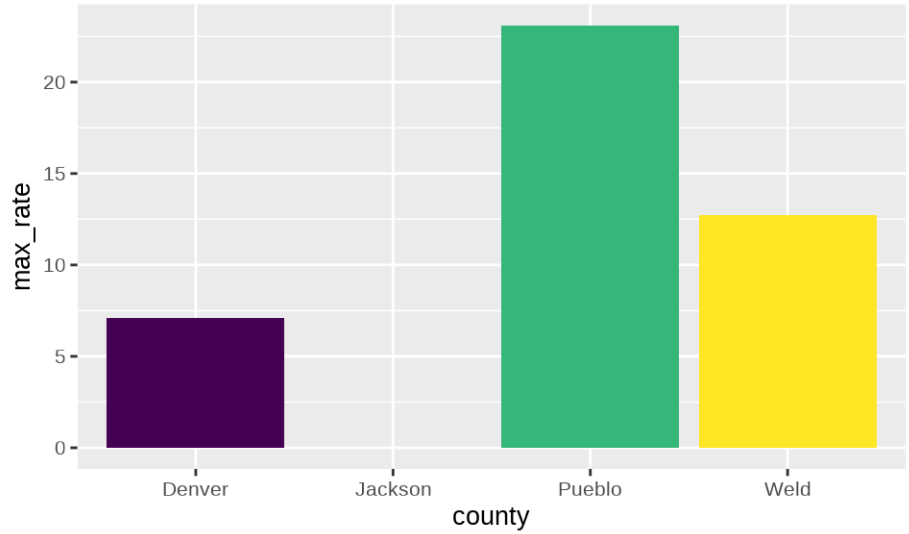
Try that again

```
er_visits_4 |> group_by(county) |>
  summarize(mean_visits = mean(visits, na.rm=T)) |>

ggplot(aes(x = county,
           y = mean_visits,
           fill = county)) +
  geom_col()
```



Tip - make sure labels aren't too small

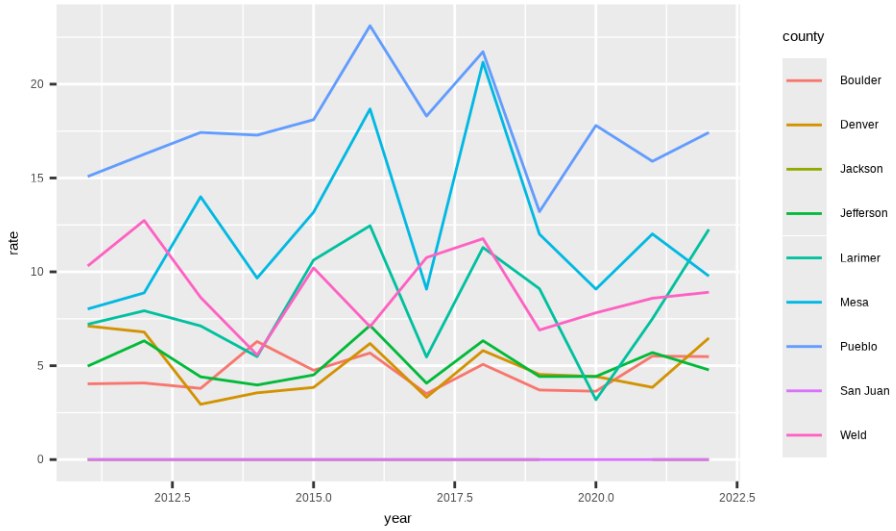
```
er_bar +
  theme(text = element_text(size = 20))
```



Sometimes we have many lines and it is hard to see what is happening

Let's look at visit rates for 9 CO counties.

```
er_visits_9 <- er |>
  filter(county %in% c("Denver", "Weld", "Pueblo", "Jackson",
                       "San Juan", "Mesa", "Jefferson", "Larimer", "Boulder"))

lots_of_lines <- ggplot(er_visits_9, aes(x = year,
                 y = rate,
                 color = county)) +
                 geom_line()
lots_of_lines
```



Adding a facet can help make it easier to see what is happening

Sometimes we have two many lines and can git difficult to see what is happening, facets can help!

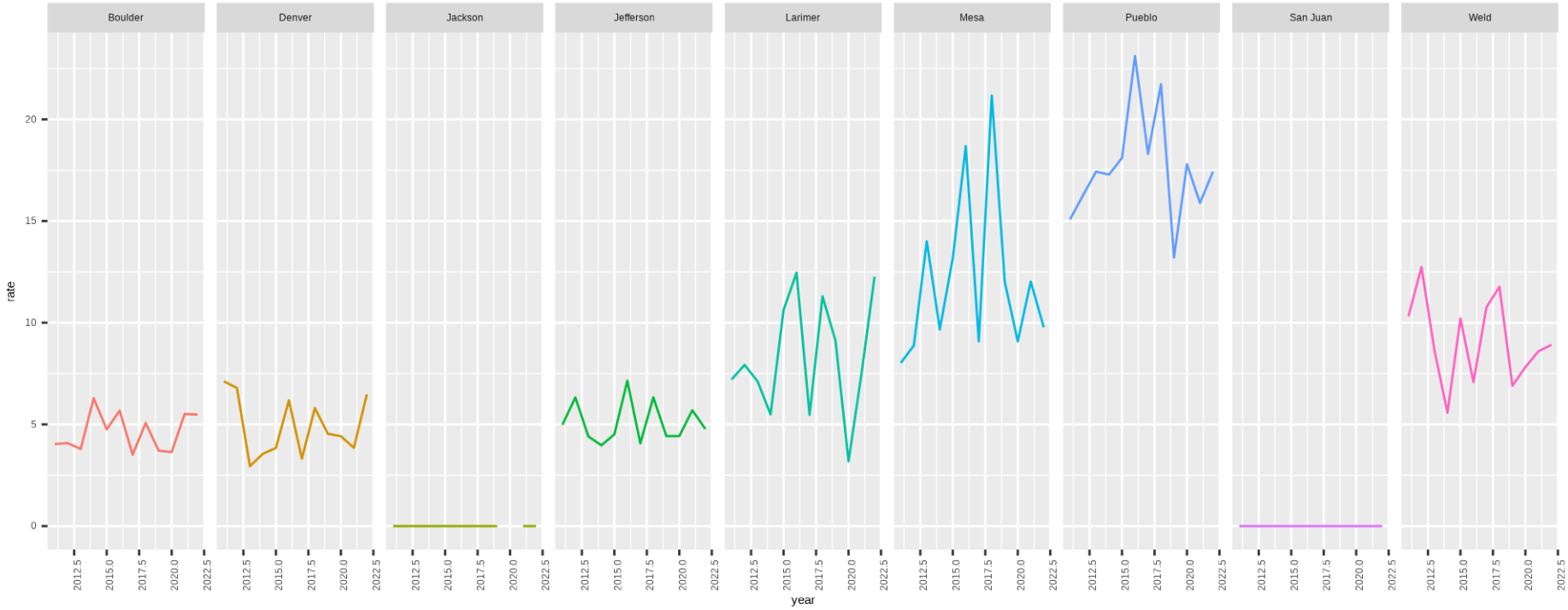
Two options: `facet_grid()`- creates a grid shape `facet_wrap()` -more flexible

Need to specify how you are faceting with the `~` sign.

```
lots_of_lines +
facet_grid( ~ county) +
theme(legend.position = "bottom")
```

Adding a facet can help make it easier to see what is happening

```
lots_of_lines +
facet_grid( ~ county) +
theme(legend.position = "none") +
theme(axis.text.x = element_text(angle = 90))
```

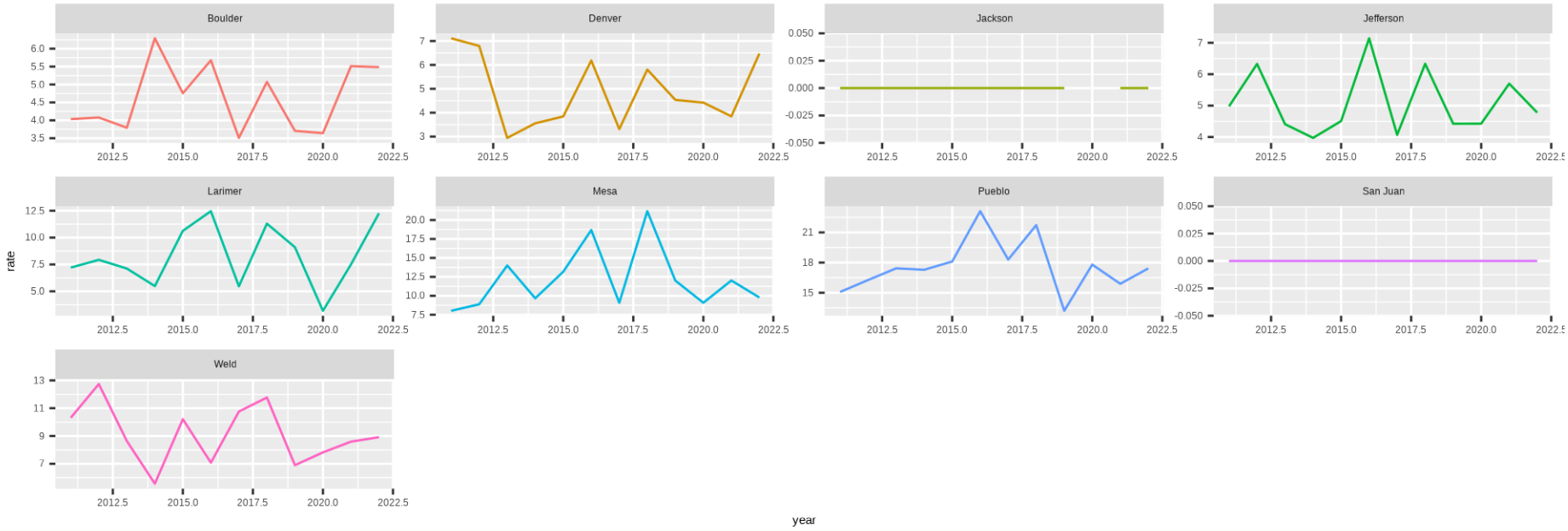


facet_wrap()

- more flexible - arguments `ncol` and `nrow` can specify layout
- can have different scales for axes using `scales = "free"`

```
rp_fac_plot <- lots_of_lines +
               facet_wrap( ~ county, ncol = 4, scales = "free") +
               theme(legend.position = "none")
```

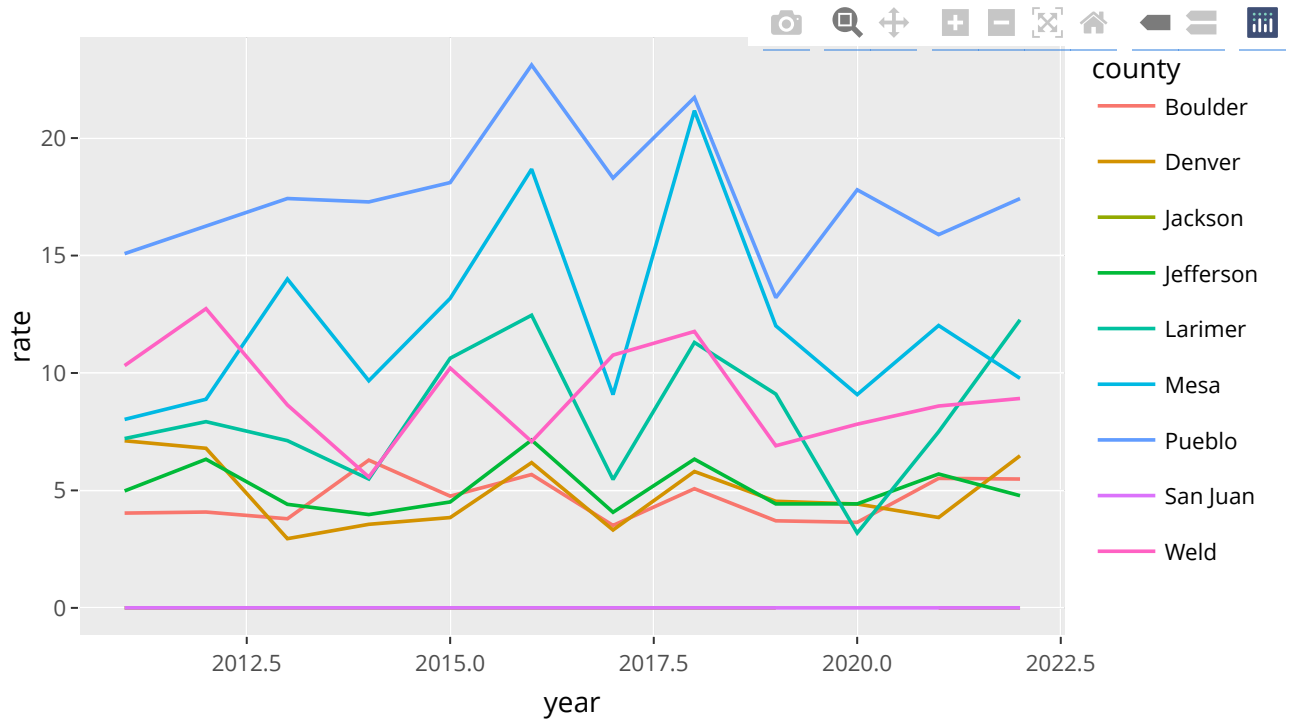
```
rp_fac_plot
```



Extensions

plotly

```
#install.packages("plotly")
library(plotly) # creates interactive plots!
ggplotly(lots_of_lines)
```



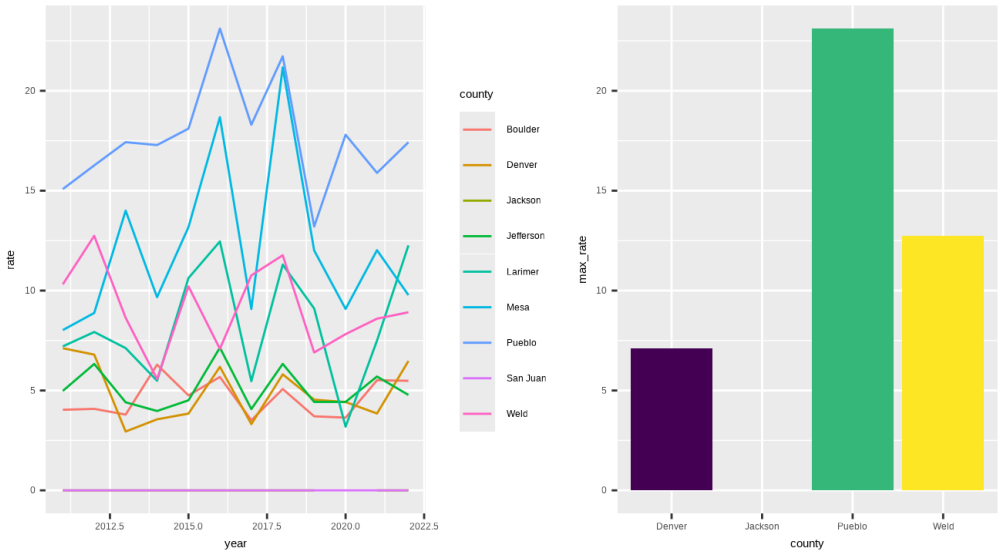
Also check out the `ggiraph` package

`patchwork` package

Great for combining plots together

Also check out the `patchwork` package

```
#install.packages("patchwork")
library(patchwork)
lots_of_lines + er_bar
```



Saving plots

Saving a ggplot to file

A few options:

- RStudio > Plots > Export > Save as image / Save as PDF
- RStudio > Plots > Zoom > [right mouse click on the plot] > Save image as
- In the code

```
ggsave(filename = "saved_plot.png",  # will save in working directory
       plot = rp_fac_plot,
       width = 6, height = 3.5)                # by default in inches
```

GUT CHECK: How to we make sure that the boxplots are filled with color instead of just the outside boarder?

A. Use the `fill` argument in the `aes` specification

B. Use `color` argument in `geom_boxplot()`

GUT CHECK: If our plot is too complicated to read, what might be a good option to fix this?

A. add more `theme()` layers

B. Use `facet_grid()` to split the plot up

Summary

- The `theme()` function helps you specify aspects about your plot
  - move or remove a legend with `theme(legend.position = "none")`
  - change font aspects of individual text elements `theme(plot.title = element_text(size = 20))`
  - center a title: `theme(plot.title = element_text(hjust = 0.5))`
- sometimes you need to add a `group` element to `aes()` if your plot looks strange
- make sure you are plotting what you think you are by checking the numbers!
- `facet_grid(~ variable)` and `facet_wrap(~variable)` can be helpful to quickly split up your plot
  - `facet_wrap()` allows for a `scales = "free"` argument so that you can have a different axis scale for different plots
- use `fill` to fill in boxplots

Good practices for plots

Check out this guide for more information!

Lab 2

- Class Website
- Lab
- Day 6 Cheatsheet
- Posit's theme cheatsheet

Additional resources: https://tidyplots.org/, https://www.rawgraphs.io/ (not R)

Image by Gerd Altmann from Pixabay

Extra Slides

Adding color - or change the color of each plot layer

You can change look of each layer separately. Note the arguments like `linetype` and `alpha` that allow us to change the opacity of the points and style of the line respectively.

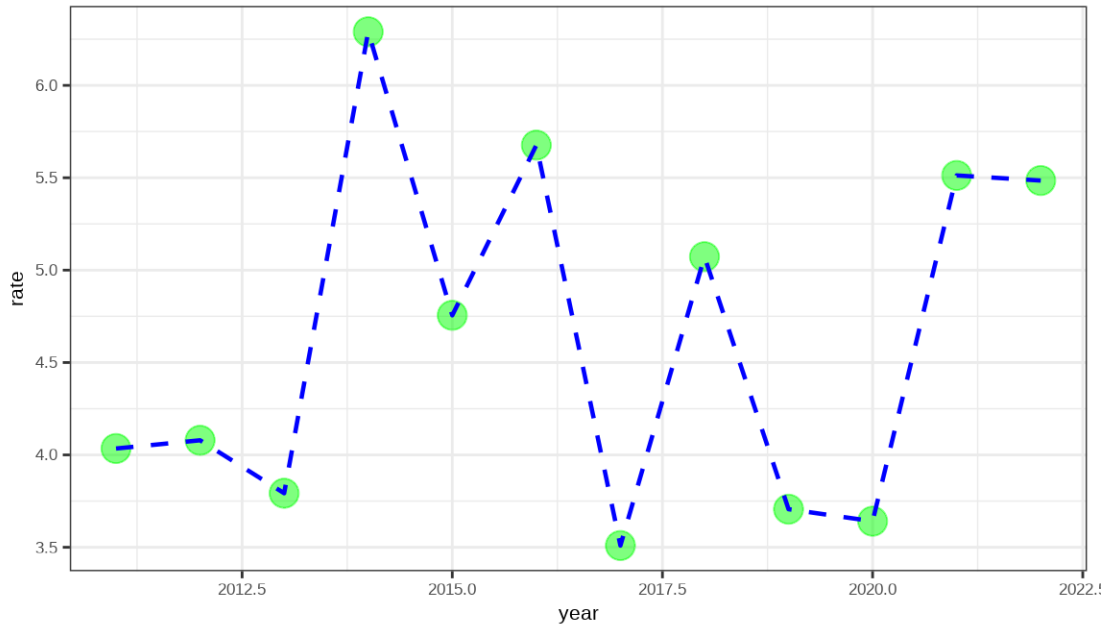
```
ggplot(er_Boulder, aes(x = year, y = rate)) +
  geom_point(size = 5, color = "red", alpha = 0.5) +
  geom_line(size = 0.8, color = "black", linetype = 2)
```

`linetype` can be given as a number. See the docs for what numbers correspond to what `linetype`!

Customize the look of the plot

You can change the look of whole plot - **specific elements, too** - like changing font and font size - or even more fonts

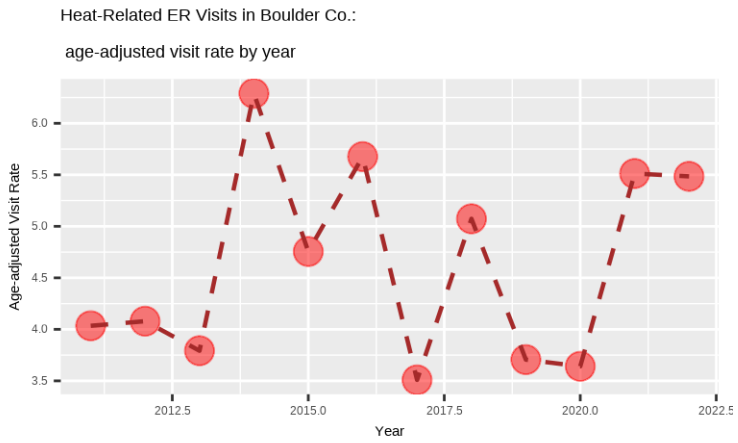
```
ggplot(er_Boulder, aes(x = year, y = rate)) +
  geom_point(size = 5, color = "green", alpha = 0.5) +
  geom_line(size = 0.8, color = "blue", linetype = 2) +
  theme_bw() +
  theme(text=element_text(size=16,  family="Comic Sans MS"))
```



Adding labels line break

Line breaks can be specified using `\n` within the `labs()` function to have a label with multiple lines.

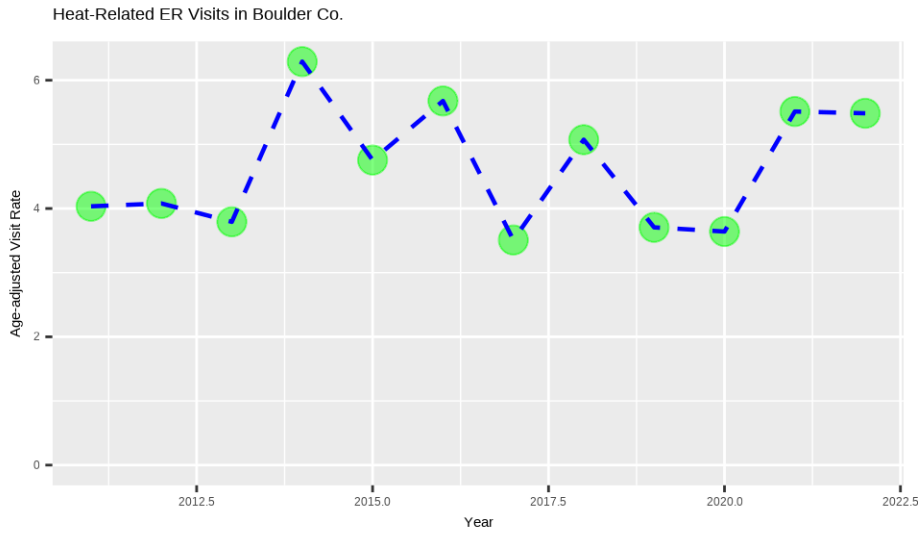
```
ggplot(er_Boulder, aes(x = year, y = rate)) +
           geom_point(size = 5, color = "red", alpha = 0.5) +
           geom_line(size = 0.8, color = "brown", linetype = 2) +
  labs(title = "Heat-Related ER Visits in Boulder Co.: \n age-adjusted visit rate by year",
            x = "Year",
            y = "Age-adjusted Visit Rate")
```



Changing axis: specifying axis limits

`xlim()` and `ylim()` can specify the limits for each axis

```
ggplot(er_Boulder, aes(x = year, y = rate)) +
            geom_point(size = 5, color = "green", alpha = 0.5) +
            geom_line(size = 0.8, color = "blue", linetype = 2) +
  labs(title = "Heat-Related ER Visits in Boulder Co.",
              x = "Year",
              y = "Age-adjusted Visit Rate") +
  ylim(0, max(pull(er_visits_4, rate)))
```

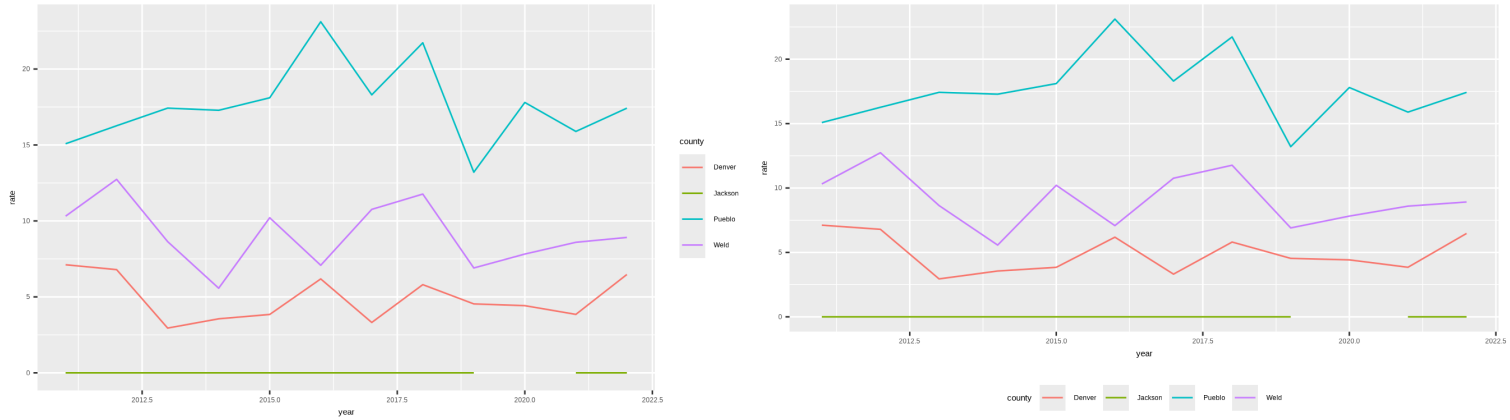


theme() function: moving (or removing) legend

If specifying position - use: “top”, “bottom”, “right”, “left”, “none”

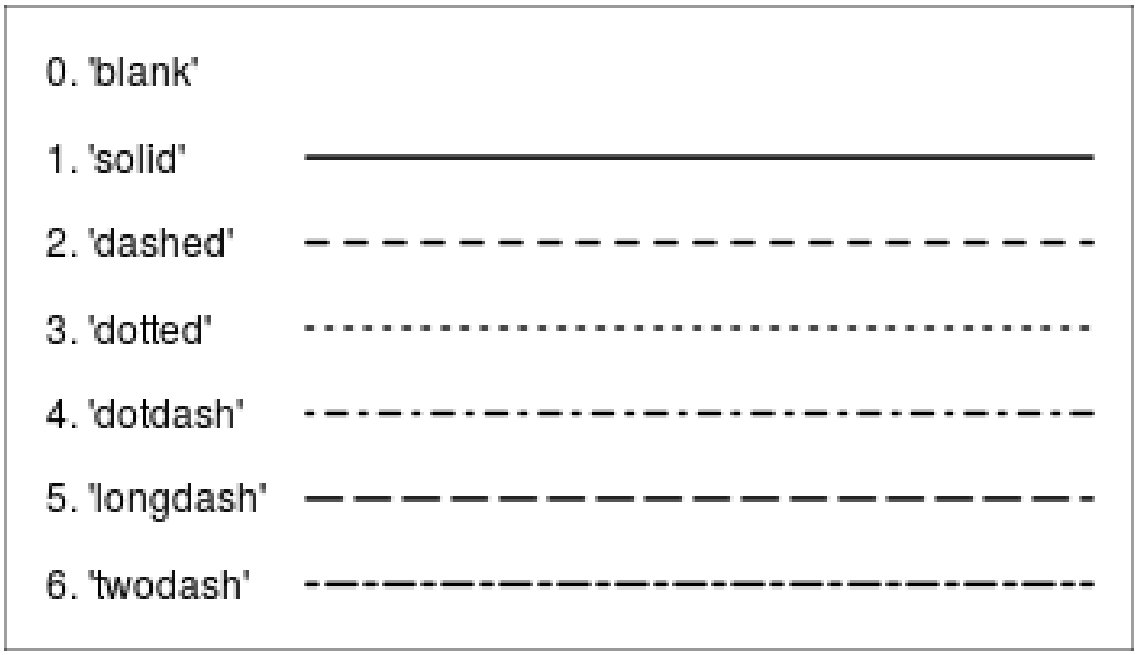
```
ggplot(er_visits_4, aes(x = year, y = rate, color = county)) +
  geom_line()
```

```
ggplot(er_visits_4, aes(x = year, y = rate, color = county)) +
  geom_line() +
  theme(legend.position = "bottom")
```



Keys for specifications

`linetype`

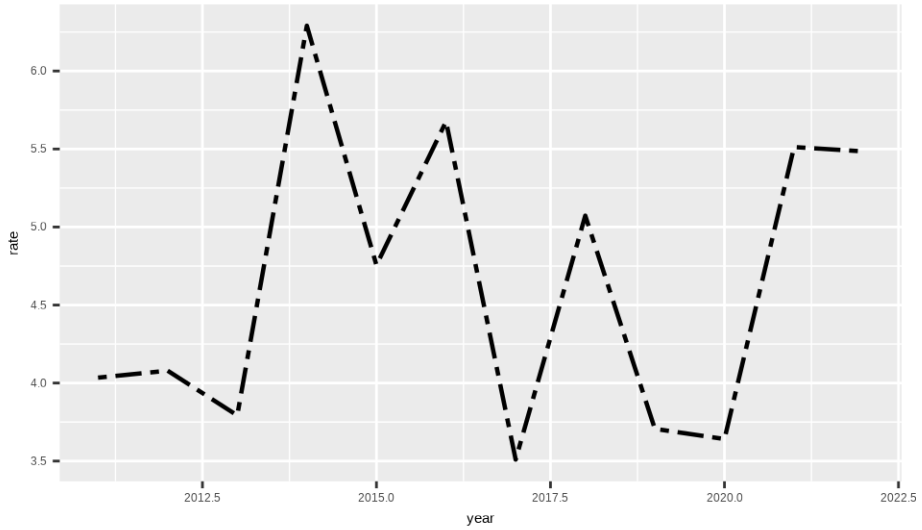


source

Linetype key

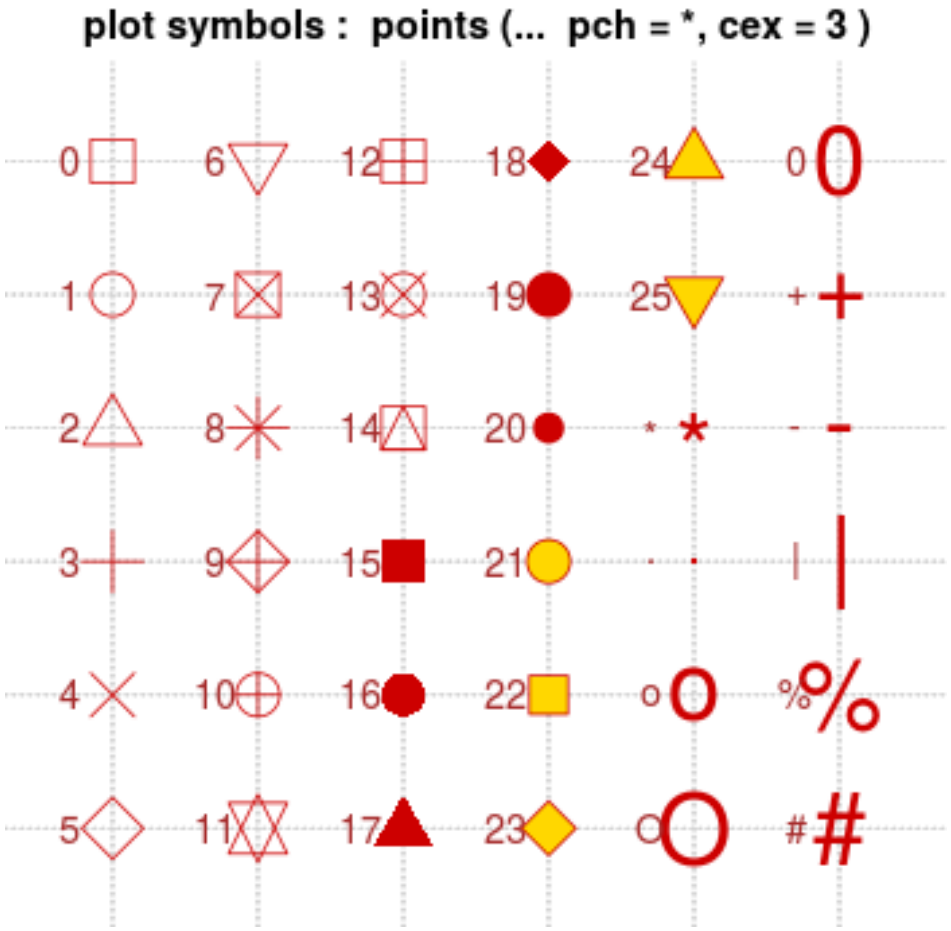
- *geoms* that draw lines have a `linetype` parameter
- these include values that are strings like “blank”, “solid”, “dashed”, “dotdash”, “longdash”, and “twodash”

```
er_Boulder |> ggplot(aes(x = year,
                         y = rate)) +
  geom_line(size = 0.8, linetype = "twodash")
```



Keys for specifications

shape

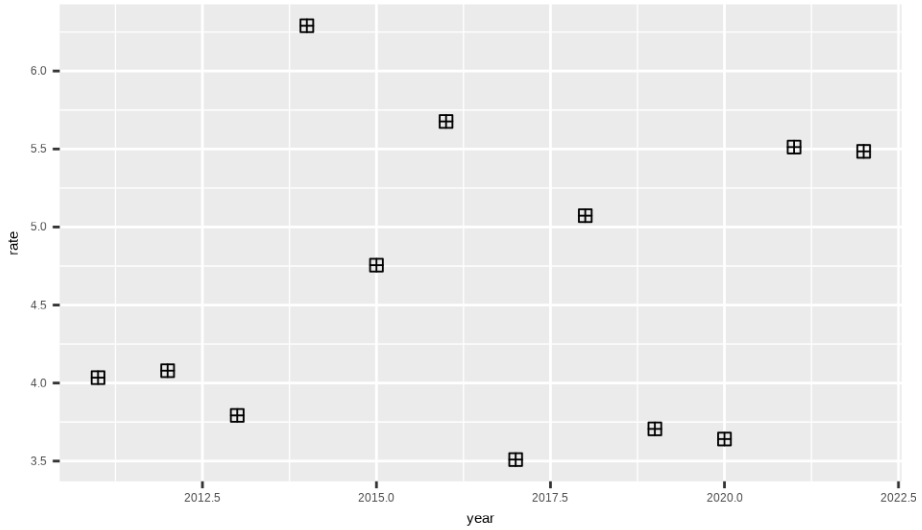


source

shape key

- *geoms* that draw have points have a `shape` parameter
- these include numeric values (don't need quotes for these) and some characters values (need quotes for these)

```
er_Boulder |> ggplot(aes(x = year,
                         y = rate)) +
  geom_point(size = 2, shape = 12)
```



Can make your own theme to use on plots!

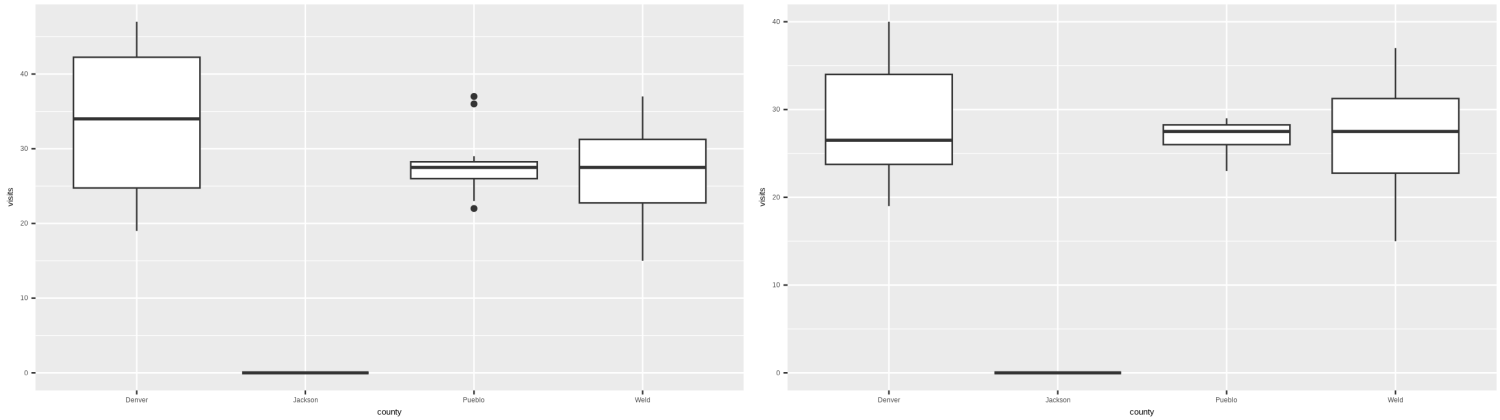
Guide on how to: https://rpubs.com/mclaire19/ggplot2-custom-themes

Tip- if you need you can remove outliers

Set `outlier.shape = NA` to get ride of outliers. Be careful about if you really should remove these!

However, if can be helpful if your plot is getting stretched to accommodate plotting an outlier. You can always say in the figure legend what you removed.

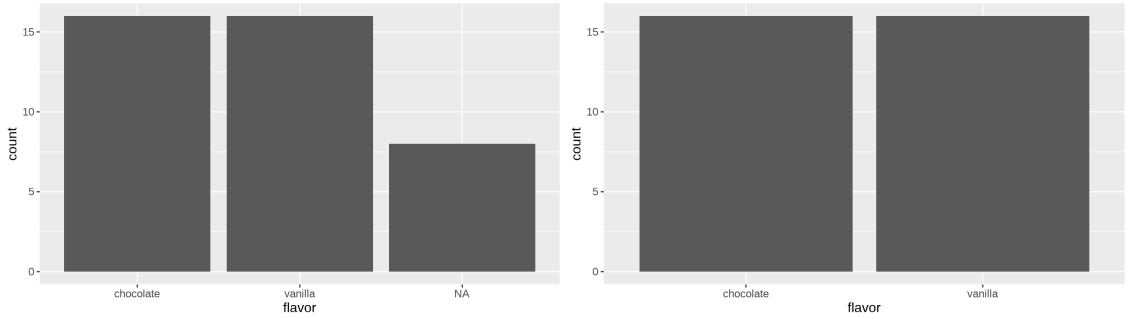
```
er_no_out1 <- ggplot(er_visits_4, aes(y = visits, x = county)) +
  geom_boxplot()

er_no_out2 <- ggplot(er_visits_4, aes(y = visits, x = county)) +
  geom_boxplot(outlier.shape = NA) +
  ylim(0,40)
```



Tip - NA Values

- if it is a numeric value it will just get dropped from the graph and you will see a warning
- it is categorical you will see it on the graph and will need to filter to remove the NA category

```
icecream <-tibble(flavor =
           rep(c("chocolate", "vanilla", NA,"chocolate", "vanilla"), 8))

icecream1 <- ggplot(icecream, aes(x = flavor)) + geom_bar() +
             theme(text=element_text(size=24))

icecream2 <- icecream |> drop_na(flavor)  |>
               ggplot( aes(x = flavor)) + geom_bar() +
                 theme(text=element_text(size=24))
```

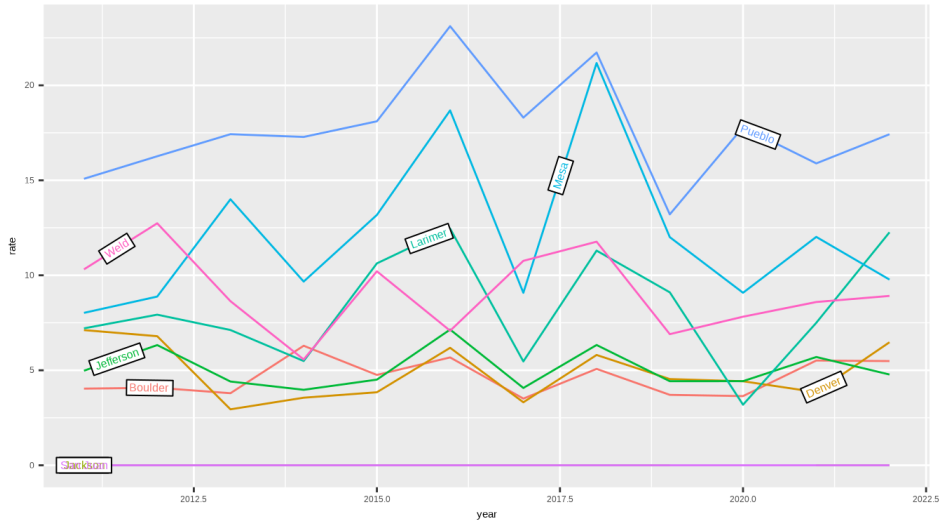


Extensions

`directlabels` package

Great for adding labels directly onto plots https://www.opencasestudies.org/ocs-bp-co2-emissions/

```
#install.packages("directlabels")
library(directlabels)
direct.label(lots_of_lines, method = list("angled.boxes"))
```



Factors

Factors

A `factor` is a special `character` vector where the elements have pre-defined groups or 'levels'. You can think of these as qualitative or categorical variables:

```
x <- c("yellow", "red", "red", "blue", "yellow", "blue")
class(x)
```

```
## [1] "character"
```

```
x_fact <- factor(x) # factor() is a function
class(x_fact)
```

```
## [1] "factor"
```

Factors

Factors have **levels** (character types do not).

```
x
```

```
## [1] "yellow" "red"    "red"    "blue"   "yellow" "blue"
```

```
x_fact
```

```
## [1] yellow red    red    blue   yellow blue
## Levels: blue red yellow
```

Note that levels are, by default, in **alphanumerical** order.

Factors

Extract the levels of a `factor` vector using `levels()`:

```
levels(x_fact)
```

```
## [1] "blue"   "red"    "yellow"
```

`forcats` package

A package called `forcats` is really helpful for working with factors.

`factor()` vs `as_factor()`

`factor()` is from base R and `as_factor()` is from `forcats`

Both can change a variable to be of class factor.

- `factor()` will order **alphanumerically** unless told otherwise.
- `as_factor()` will order by **first appearance** unless told otherwise.

If you are assigning your levels manually either function is fine!

`as_factor()` function

```
x <- c("yellow", "red", "red", "blue", "yellow", "blue")
x_fact_2 <- as_factor(x)
x_fact_2
```

```
## [1] yellow red    red    blue   yellow blue
## Levels: yellow red blue
```

```
# Compare to factor() method:
x_fact
```

```
## [1] yellow red    red    blue   yellow blue
## Levels: blue red yellow
```

A Factor Example

We will use a slightly different version of the data on heat-related visits to the ER from the State of Colorado.

For today, we are looking at data that reports ER visits by age category.

```
er_visits_age <- read_csv("https://daseh.org/data/CO_ER_heat_visits_by_age.csv")
```

```
## Rows: 60 Columns: 6
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (1): age
## dbl (5): year, rate, lower95cl, upper95cl, visits
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The data

```
head(er_visits_age)
```

```
## # A tibble: 6 × 6
##    year age          rate lower95cl upper95cl visits
##   <dbl> <chr>       <dbl>     <dbl>     <dbl>  <dbl>
## 1  2011 0-4 years    3.52      1.82      6.16     12
## 2  2011 15-34 years  7.34      5.95      8.74    106
## 3  2011 35-64 years  5.84      4.80      6.88    121
## 4  2011 5-14 years   5.20      3.50      6.90     36
## 5  2011 65+ years    8.34      5.98     10.7      48
## 6  2012 0-4 years    3.58      1.85      6.25     12
```

Notice that `age` is a `chr` variable. This indicates that the values are **character** strings.

R does not realize that there is any order related to the `AGE` values. It will assume that it is **alphanumeric** (for numbers, this means ascending order).

However, we know that the order is: **0-4 years old**, **5-14 years old**, **15-34 years old**, **35-64 years old**, and **65+ years old**.

For the next steps, let's take a subset of data.

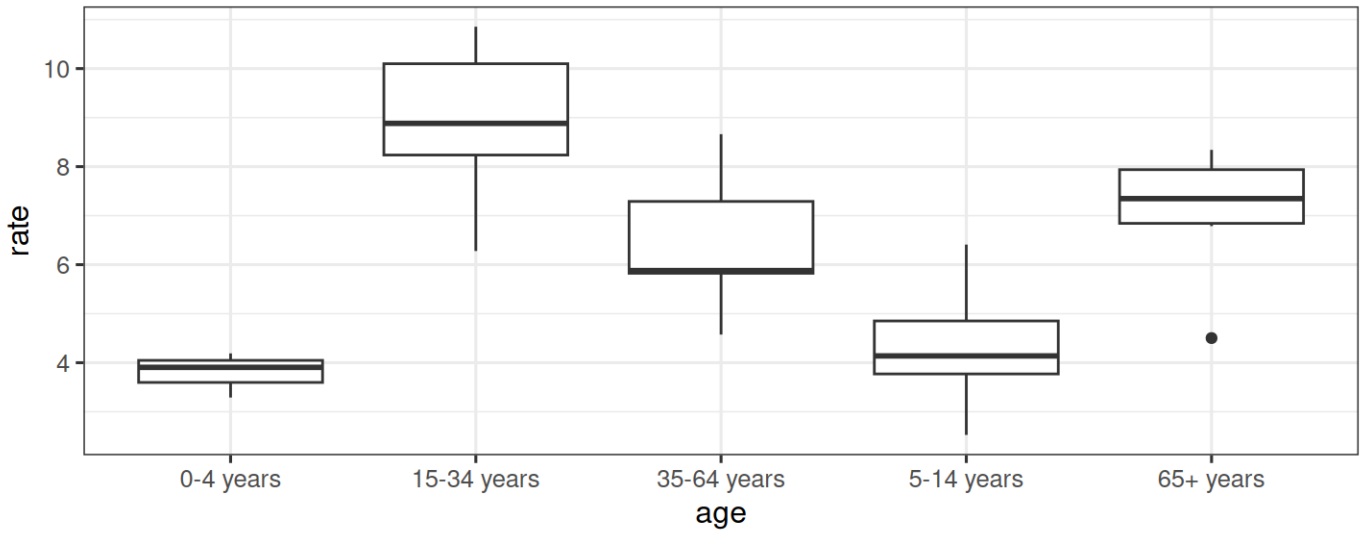
Use `set.seed()` to take the same random sample each time.

```
set.seed(123)
er_visits_age_subset <- slice_sample(er_visits_age, n = 32)
```

Plot the data

Let's make a plot first.

```
er_visits_age_subset |>
  ggplot(aes(x = age, y = rate)) +
  geom_boxplot() +
  theme_bw(base_size = 12) # make all labels size 12
```



OK this is very useful, but it is a bit difficult to read. We expect the values to be plotted by the order that we know, not by alphabetical order.

Change to factor

Currently `age` is class `character` but let's change that to class `factor` which allows us to specify the levels or order of the values.

```
er_visits_age_fct <-
  er_visits_age_subset |>
  mutate(age = factor(age,
    levels = c("0-4 years", "5-14 years", "15-34 years", "35-64 years",
               "65+ years")
  ))

er_visits_age_fct |>
  pull(age) |>
  levels()
```

```
## [1] "0-4 years"   "5-14 years"  "15-34 years" "35-64 years" "65+ years"
```

Change to a factor

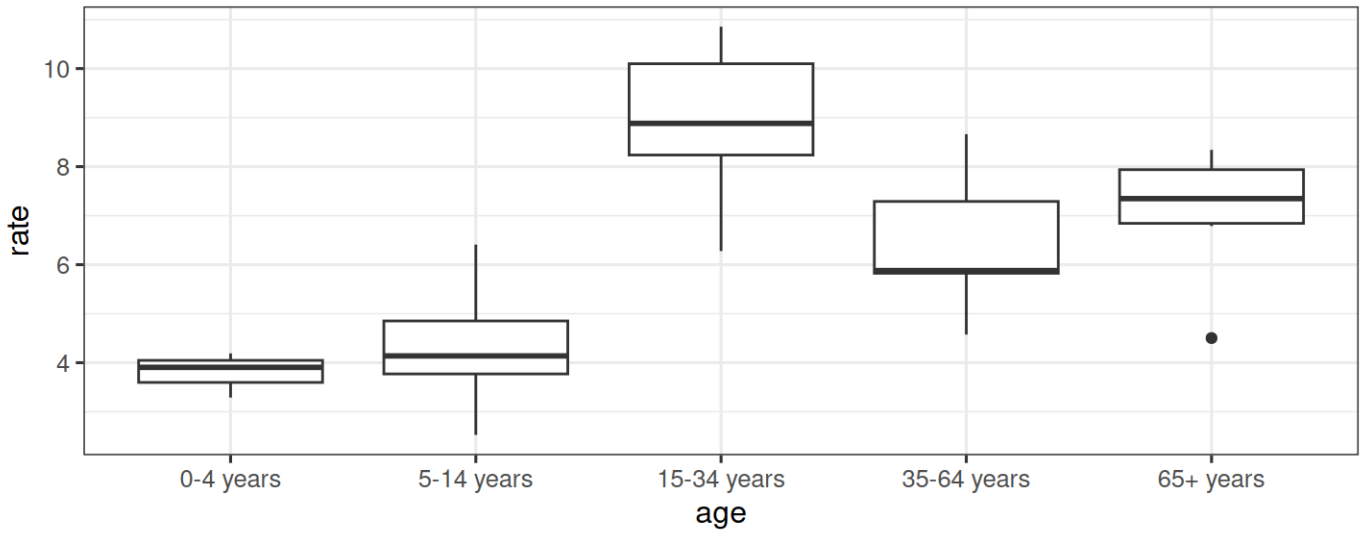
```
head(er_visits_age_fct)
```

```
## # A tibble: 6 × 6
##    year age          rate lower95cl upper95cl visits
##   <dbl> <fct>       <dbl>     <dbl>     <dbl>  <dbl>
## 1  2017 0-4 years    3.29      1.64      5.89     11
## 2  2013 65+ years    4.50      2.86      6.14     29
## 3  2021 0-4 years   NA        NA        NA        NA
## 4  2013 5-14 years   5.51      3.78      7.23     39
## 5  2011 35-64 years  5.84      4.80      6.88    121
## 6  2019 15-34 years  8.34      6.94      9.73    137
```

Plot again

Now let's make our plot again:

```
er_visits_age_fct |>
  ggplot(aes(x = age, y = rate)) +
  geom_boxplot() +
  theme_bw(base_size = 12)
```



Now that's more like it! Notice how the data is automatically plotted in the order we would like.

What about if we `arrange()` the data by age?

Character data is arranged alphabetically (if letters) or by ascending first number (if numbers).

```
er_visits_age_subset |>
  arrange(age)
```

```
## # A tibble: 32 × 6
##     year age          rate lower95cl upper95cl visits
##    <dbl> <chr>       <dbl>     <dbl>     <dbl>  <dbl>
##  1  2017 0-4 years    3.29      1.64      5.89     11
##  2  2021 0-4 years   NA        NA        NA        NA
##  3  2016 0-4 years    4.19      2.29      7.03     14
##  4  2018 0-4 years    3.91      2.08      6.68     13
##  5  2019 15-34 years  8.34      6.94      9.73    137
##  6  2018 15-34 years 10.1       8.60     11.7     165
##  7  2022 15-34 years 10.0       8.52     11.6     167
##  8  2016 15-34 years 10.9       9.23     12.5     171
##  9  2012 15-34 years  8.88      7.36     10.4     130
## 10  2014 15-34 years  6.28      5.02      7.54     95
## # ℹ 22 more rows
```

Notice that the order is not what we would hope for!

Arranging Factors

Factor data is arranged by level.

```
er_visits_age_fct |>
  arrange(age)
```

```
## # A tibble: 32 × 6
##     year age         rate lower95cl upper95cl visits
##    <dbl> <fct>      <dbl>     <dbl>     <dbl>  <dbl>
##  1  2017 0-4 years   3.29      1.64      5.89     11
##  2  2021 0-4 years  NA        NA        NA        NA
##  3  2016 0-4 years   4.19      2.29      7.03     14
##  4  2018 0-4 years   3.91      2.08      6.68     13
##  5  2013 5-14 years  5.51      3.78      7.23     39
##  6  2012 5-14 years  4.14      2.63      5.64     29
##  7  2016 5-14 years  6.41      4.56      8.26     46
##  8  2020 5-14 years NA        NA        NA        NA
##  9  2019 5-14 years  3.80      2.36      5.23     27
## 10  2014 5-14 years  2.53      1.50      3.99     18
## # ℹ 22 more rows
```

Nice! Now this is what we would want!

Making tables with characters

Tables grouped by a character are arranged alphabetically (if letters) or by ascending first number (if numbers).

```
er_visits_age_subset |>
  group_by(age) |>
  summarize(total_visits = sum(visits, na.rm = T))
```

```
## # A tibble: 5 × 2
##   age         total_visits
##   <chr>              <dbl>
## 1 0-4 years             38
## 2 15-34 years          986
## 3 35-64 years          983
## 4 5-14 years           215
## 5 65+ years            296
```

Making tables with factors

Tables grouped by a factor are arranged by level.

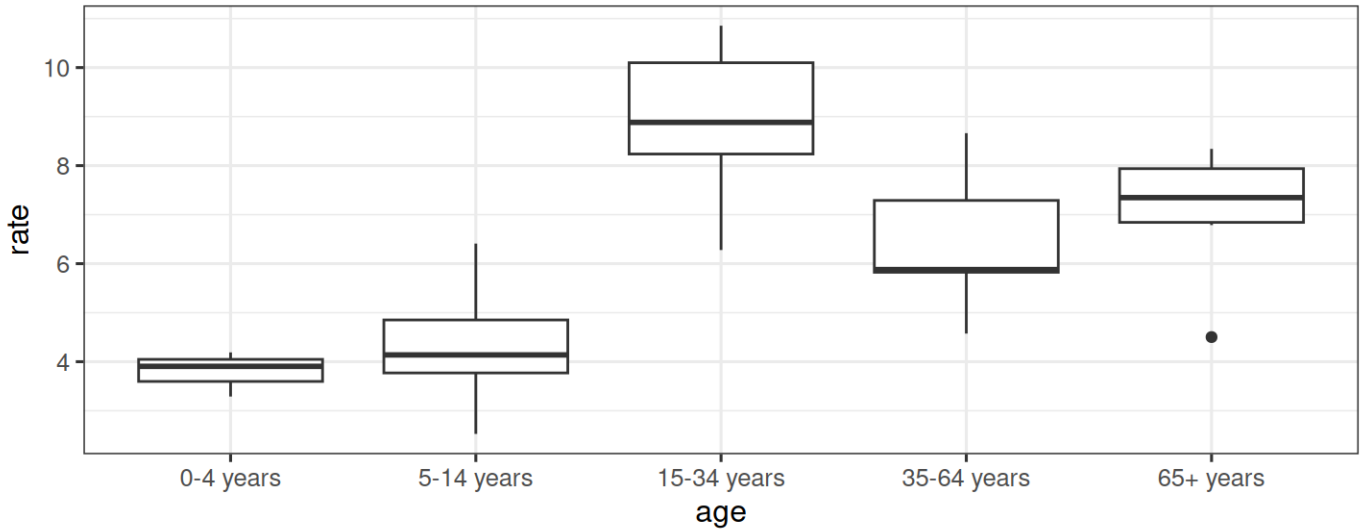
```
er_visits_age_fct |>
  group_by(age) |>
  summarize(total_visits = sum(visits, na.rm = T))
```

```
## # A tibble: 5 × 2
##   age         total_visits
##   <fct>              <dbl>
## 1 0-4 years             38
## 2 5-14 years           215
## 3 15-34 years          986
## 4 35-64 years          983
## 5 65+ years            296
```

forcats for ordering

What if we wanted to order `age` by increasing `rate`?

```
library(forcats)

er_visits_age_fct |>
  ggplot(aes(x = age, y = rate)) +
  geom_boxplot() +
  theme_bw(base_size = 12)
```



This would be useful for identifying easily which age group to focus on.

forcats for ordering

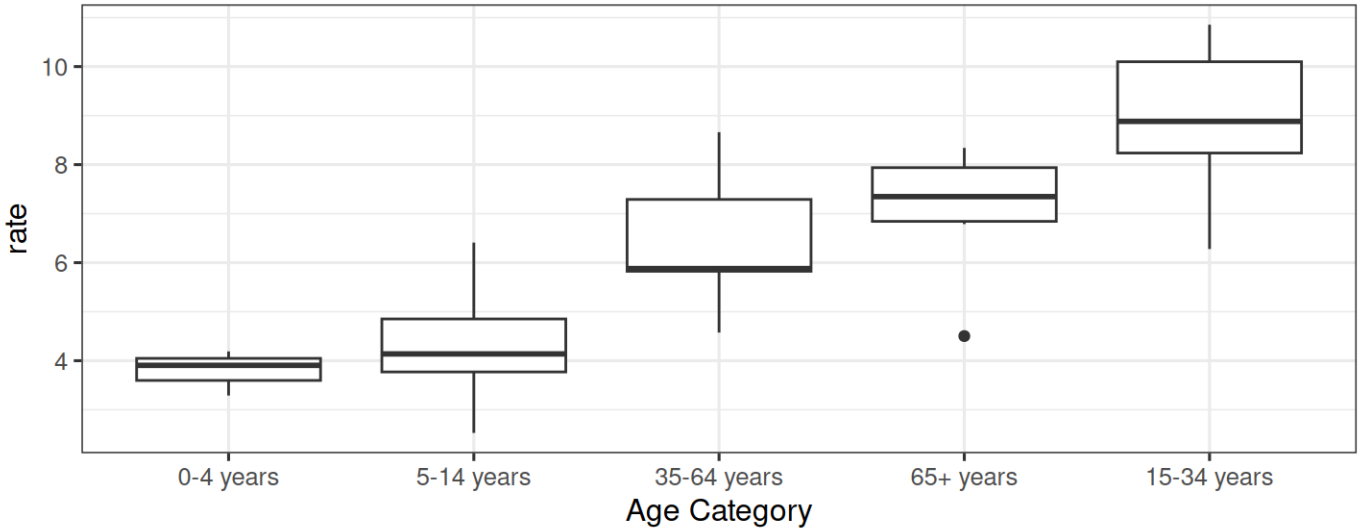
We can order a factor by another variable by using the `fct_reorder()` function of the `forcats` package.

```
fct_reorder({column getting changed}, {guiding column}, {summarizing function})
```

forcats for ordering

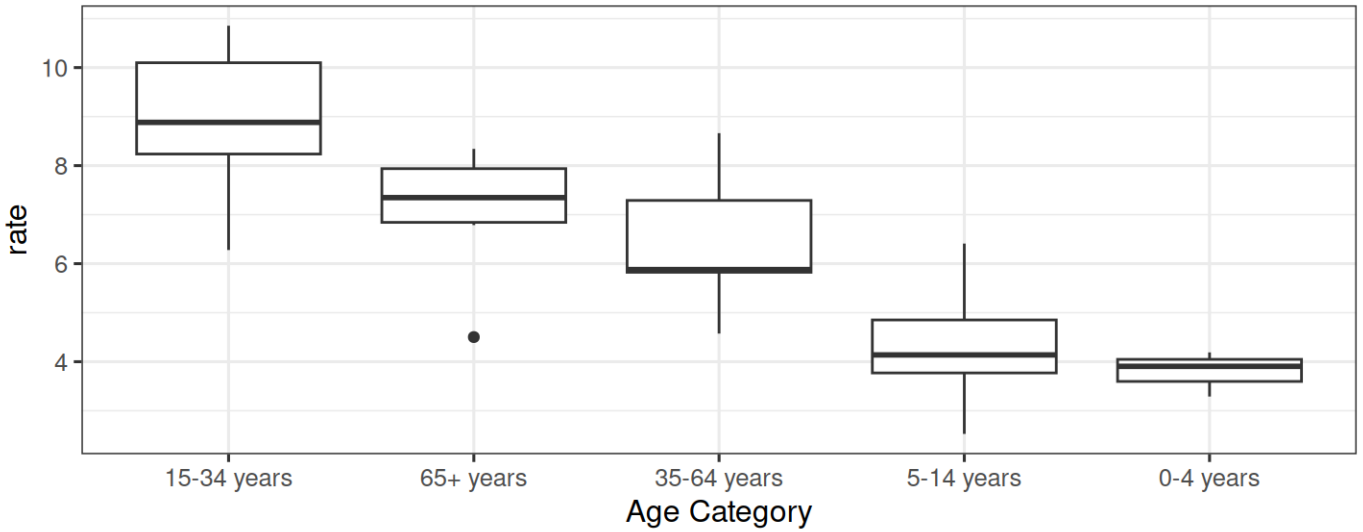
We can order a factor by another variable by using the `fct_reorder()` function of the `forcats` package.

```
library(forcats)

er_visits_age_fct |>
  ggplot(aes(x = fct_reorder(age, rate, mean), y = rate)) +
  geom_boxplot() +
  labs(x = "Age Category") +
  theme_bw(base_size = 12)
```



forcats for ordering.. with `.desc =` argument

```
library(forcats)

er_visits_age_fct |>
  ggplot(aes(x = fct_reorder(age, rate, mean, .desc = TRUE), y = rate)) +
  geom_boxplot() +
  labs(x = "Age Category") +
  theme_bw(base_size = 12)
```



forcats for ordering… can be used to sort datasets

```
er_visits_age_fct |> pull(age) |> levels() # By year order
```

```
## [1] "0-4 years"   "5-14 years"  "15-34 years" "35-64 years" "65+ years"
```

```
er_visits_age_fct <- er_visits_age_fct |>
  mutate(
    age = fct_reorder(age, rate, mean)
  )
```

```
er_visits_age_fct |> pull(age) |> levels() # by increasing mean visits
```

```
## [1] "0-4 years"   "5-14 years"  "35-64 years" "65+ years"   "15-34 years"
```

Checking Proportions with `fct_count()`

The `fct_count()` function of the `forcats` package is helpful for checking that the proportions of each level for a factor are similar. Need the `prop = TRUE` argument otherwise just counts are reported.

```
er_visits_age_fct |>
  pull(age) |>
  fct_count(prop = TRUE)
```

```
## # A tibble: 5 × 3
##   f              n     p
##   <fct>      <int> <dbl>
## 1 0-4 years      4 0.125
## 2 5-14 years     8 0.25
## 3 35-64 years    7 0.219
## 4 65+ years      6 0.188
## 5 15-34 years    7 0.219
```

GUT CHECK: Why is it useful to have the factor class as an option?

A. It helps us check the factual accuracy of our datasets.

B. It helps us change the order of variables in case the order has meaning.

GUT CHECK: What does the `fct_reorder()` function do?

A. It helps us reorder a factor based on the values of another variable.

B. It helps us reorder a factor based on a random change in the order.

Summary

- the factor class allows us to have a different order from alphanumeric for categorical data
- we can change data to be a factor variable using `mutate` and a factor creating function like `factor()` or `as_factor`
- the `as_factor()` is from the `forcats` package (first appearance order by default)
- the `factor()` base R function (alphanumeric order by default)
- with `factor()` we can specify the levels with the `levels` argument if we want a specific order
- the `fct_reorder({variable_to_reorder}, {variable_to_order_by}, {summary function})` helps us reorder a variable by the values of another variable
- arranging, tabulating, and plotting the data will reflect the new order

Lab

- Class Website
- Lab. - Day 6 Cheatsheet - Posit's forcats cheatsheet

Image by Gerd Altmann from Pixabay

Statistics

Summary

- `ggplot()` specifies what data to use and what variables will be mapped to where
- inside `ggplot()`, `aes(x = , y = , color = )` specify what variables correspond to what aspects of the plot in general
- layers of plots can be combined using the `+` at the **end** of lines
- use `geom_line()` and `geom_point()` to add lines and points
- sometimes you need to add a `group` element to `aes()` if your plot looks strange
- make sure you are plotting what you think you are by checking the numbers!
- `facet_grid(~variable)` and `facet_wrap(~variable)` can be helpful to quickly split up your plot

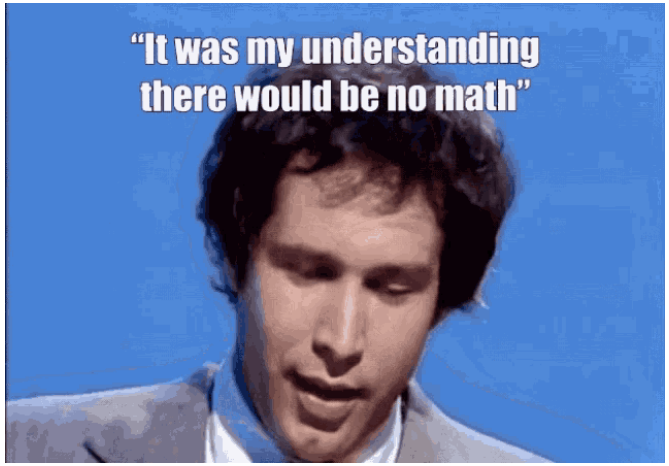
Summary

- the factor class allows us to have a different order from alphanumeric for categorical data
- we can change data to be a factor variable using `mutate()`, `as_factor()` (in the `forcats` package), or `factor()` functions and specifying the levels with the `levels` argument
- `fct_reorder({variable_to_reorder}, {variable_to_order_by})` helps us reorder a variable by the values of another variable
- arranging, tabulating, and plotting the data will reflect the new order

Overview

We will cover how to use R to compute some of basic statistics and fit some basic statistical models.

- Correlation
- T-test
- Linear Regression / Logistic Regression


“It was my understanding
there would be no math”

Overview

We will focus on how to use R software to do these. We will be glossing over the statistical **theory** and "formulas" for these tests. Moreover, we do not claim the data we use for demonstration meet **assumptions** of the methods.

There are plenty of resources online for learning more about these methods.

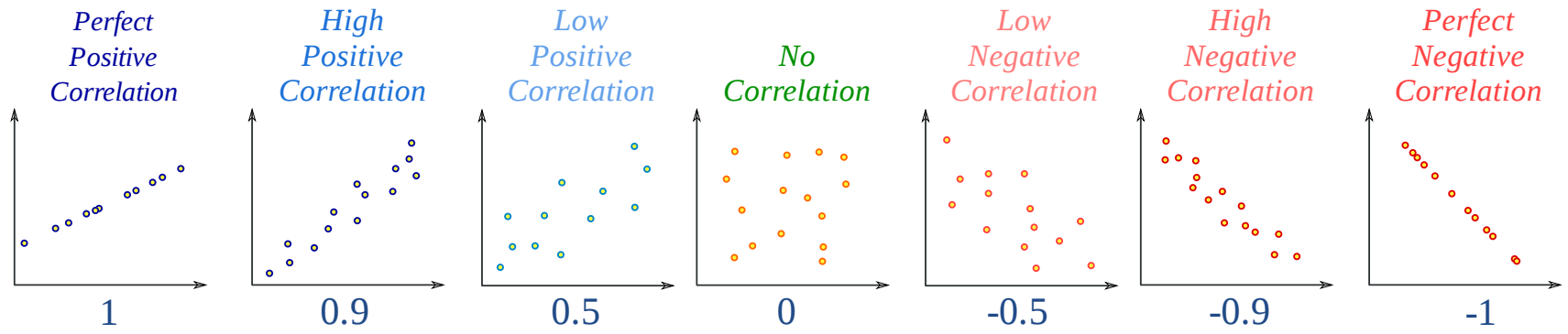
Check out www.opencasestudies.org for deeper dives on some of the concepts covered here and the resource page for more resources.

Correlation

Correlation

The correlation coefficient is a summary statistic that measures the strength of a linear relationship between two numeric variables.

- The strength of the relationship - based on how well the points form a line
- The direction of the relationship - based on if the points progress upward or downward



source

See this case study for more information.

Correlation

Function `cor()` computes correlation in R.

```
cor(x, y = NULL, use = c("everything", "complete.obs"),
    method = c("pearson", "kendall", "spearman"))
```

- provide two numeric vectors of the same length (arguments `x`, `y`), or
- provide a data.frame / tibble with numeric columns only
- by default, Pearson correlation coefficient is computed

Correlation test

Function `cor.test()` also computes correlation and tests for association.

```
cor.test(x, y = NULL, alternative(c("two.sided", "less", "greater")),
    method = c("pearson", "kendall", "spearman"))
```

- provide two numeric vectors of the same length (arguments `x`, `y`), or
- provide a data.frame / tibble with numeric columns only
- by default, Pearson correlation coefficient is computed
- alternative values:
  - two.sided means true correlation coefficient is not equal to zero (default)
  - greater means true correlation coefficient is > 0 (positive relationship)
  - less means true correlation coefficient is < 0 (negative relationship)

GUT CHECK!

What class of data do you need to calculate a correlation?

A. Character data

B. Factor data

C. Numeric data

Correlation

Let’s look at the dataset of yearly CO2 emissions by country.

```
yearly_co2 <-
  read_csv(file = "https://daseh.org/data/Yearly_CO2_Emissions_1000_tonnes.csv")
```

Correlation for two vectors

First, we create two vectors.

```
# x and y must be numeric vectors
y1980 <- yearly_co2 |> pull(`1980`)
y1985 <- yearly_co2 |> pull(`1985`)
```

Like other functions, if there are `NA`s, you get `NA` as the result. But if you specify `use = "complete.obs"`, then it will give you correlation using the non-missing data.

```
cor(y1980, y1985, use = "complete.obs")
```

```
[1] 0.9936257
```

Correlation coefficient calculation and test

```
cor.test(y1980, y1985)

    Pearson's product-moment correlation

data:  y1980 and y1985
t = 114.59, df = 169, p-value < 0.00000000000000022
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9913844 0.9952853
sample estimates:
      cor
0.9936257
```

Broom package

The `broom` package helps make stats results look tidy

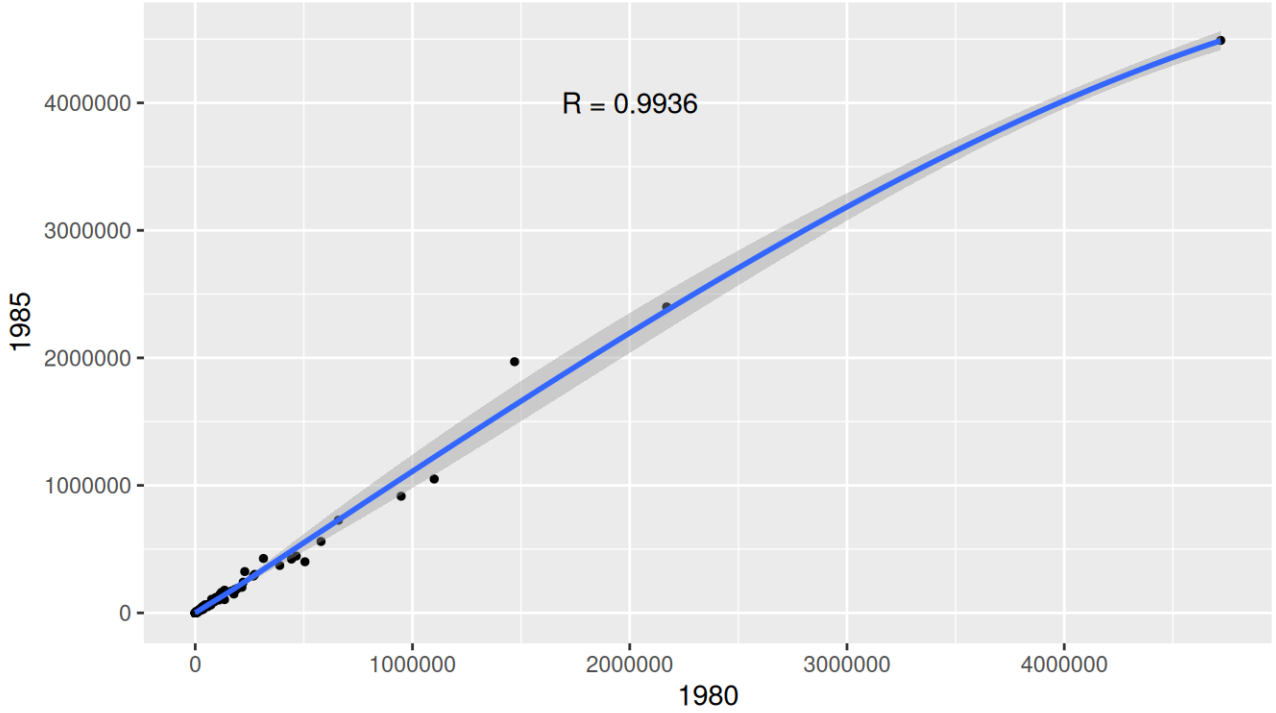
```
library(broom)
cor_result <- tidy(cor.test(y1980, y1985))
glimpse(cor_result)
```

```
Rows: 1
Columns: 8
$ estimate    <dbl> 0.9936257
$ statistic   <dbl> 114.5851
$ p.value     <dbl> 0.0000000000000000000000000000000000000000000000000000…
$ parameter   <int> 169
$ conf.low    <dbl> 0.9913844
$ conf.high   <dbl> 0.9952853
$ method      <chr> "Pearson's product-moment correlation"
$ alternative <chr> "two.sided"
```

Correlation for two vectors with plot

In plot form… `geom_smooth()` and `annotate()` can look very nice!

```
corr_value <- pull(cor_result, estimate) |> round(digits = 4)
cor_label <- paste0("R = ", corr_value)
yearly_co2 |>
  ggplot(aes(x = `1980`, y = `1985`)) + geom_point(size = 1) + geom_smooth() +
  annotate("text", x = 2000000, y = 4000000, label = cor_label)
```



Correlation for data frame columns

We can compute correlation for all pairs of columns of a data frame / matrix. This is often called, *"computing a correlation matrix"*.

Columns must be all numeric!

```
co2_subset <- yearly_co2 |>
  select(c(`1950`, `1980`, `1985`, `2010`))

head(co2_subset)
```

```
# A tibble: 6 × 4
  `1950` `1980` `1985` `2010`
   <dbl>  <dbl>  <dbl>  <dbl>
1   84.3   1760   3510   8460
2  297     5170   7880   4600
3 3790    66500  72800 119000
4   NA       NA     NA    517
5  187     5350   4700  29100
6   NA      143    249    524
```

Correlation for data frame columns

We can compute correlation for all pairs of columns of a data frame / matrix. This is often called, *"computing a correlation matrix"*.

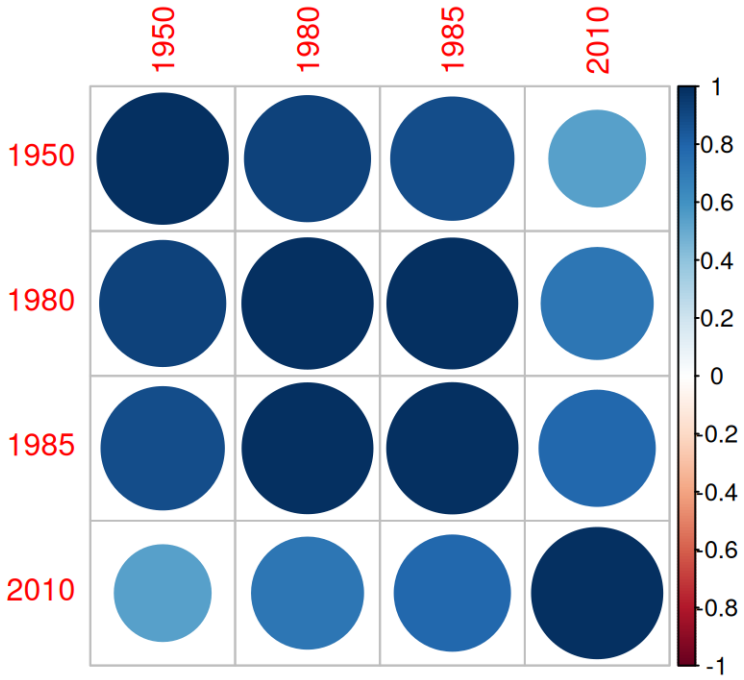
```
cor_mat <- cor(co2_subset, use = "complete.obs")
cor_mat
```

```
          1950      1980      1985      2010
1950 1.0000000 0.9228253 0.8818288 0.5415047
1980 0.9228253 1.0000000 0.9935477 0.7270839
1985 0.8818288 0.9935477 1.0000000 0.7827256
2010 0.5415047 0.7270839 0.7827256 1.0000000
```

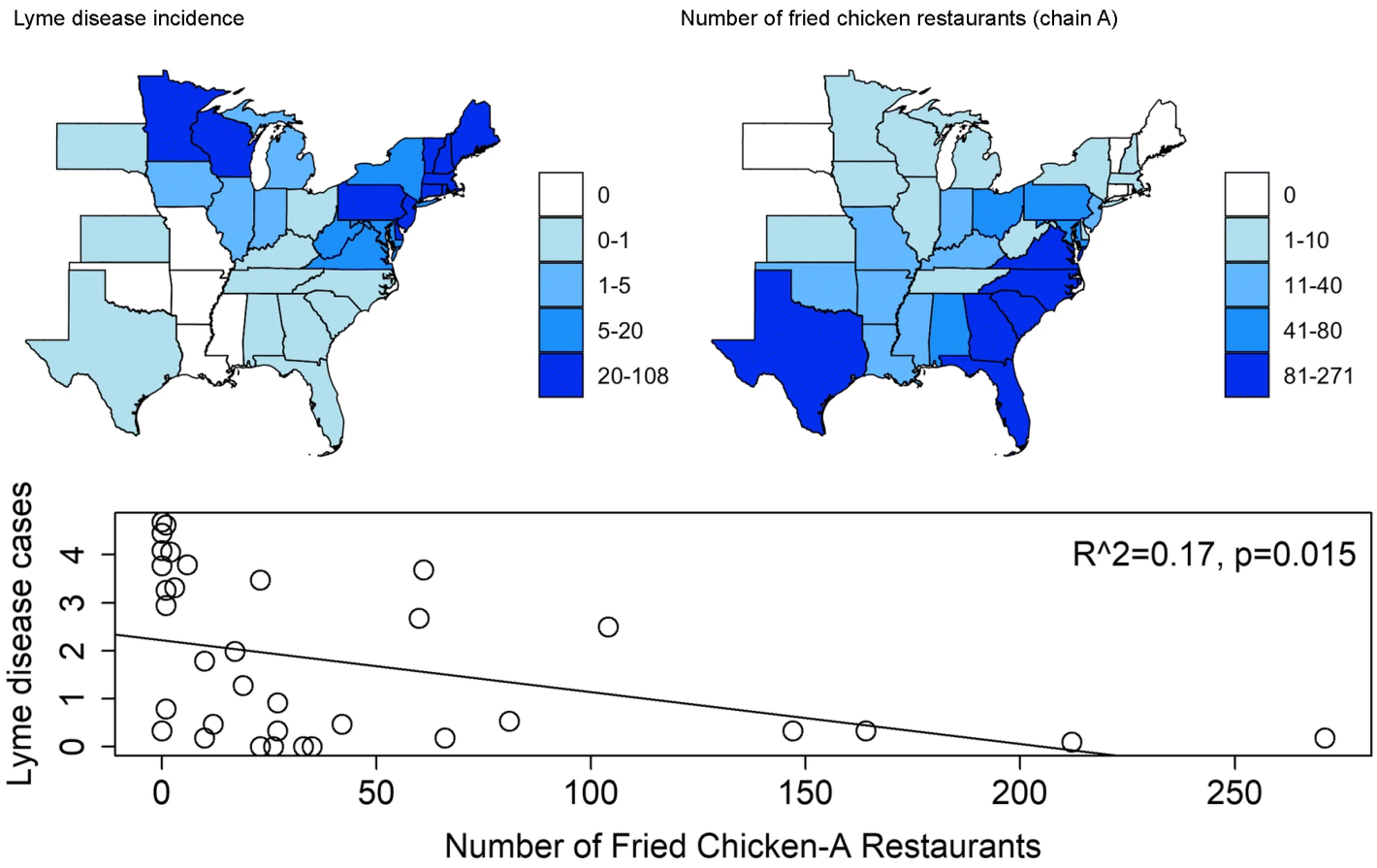
Correlation for data frame columns with plot

`corrplot` package can make correlation matrix plots

```
library(corrplot)
corrplot(cor_mat)
```



Correlation does not imply causation



source

T-test

T-test

The commonly used t-tests are:

- **one-sample t-test** – used to test mean of a variable in one group
- **two-sample t-test** – used to test difference in means of a variable between two groups
  - if the “two groups” are data of the *same* individuals collected at 2 time points, we say it is two-sample paired t-test)

The `t.test()` function does both.

```
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
```

Running one-sample t-test

It tests the mean of a variable in one group. By default (i.e., without us explicitly specifying values of other arguments):

- tests whether a mean of a variable is equal to 0 (`mu = 0`)
- uses “two sided” alternative (`alternative = "two.sided"`)
- returns result assuming confidence level 0.95 (`conf.level = 0.95`)
- omits `NA` values in data

Let’s look at the CO2 emissions data again.

```
t.test(y1980)
```

```
    One Sample t-test

data:  y1980
t = 3.3324, df = 170, p-value = 0.001056
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
  44745.81 174792.25
sample estimates:
mean of x
   109769
```

Running two-sample t-test

It tests the difference in means of a variable between two groups. By default:

- tests whether difference in means of a variable is equal to 0 (`mu = 0`)
- uses "two sided" alternative (`alternative = "two.sided"`)
- returns result assuming confidence level 0.95 (`conf.level = 0.95`)
- assumes data are not paired (`paired = FALSE`)
- assumes true variance in the two groups is not equal (`var.equal = FALSE`)
- omits `NA` values in data

Check out this this case study and this case study for more information.

Running two-sample t-test in R

```
t.test(y1980, y1985)

    Welch Two Sample t-test

data:  y1980 and y1985
t = -0.090533, df = 341, p-value = 0.9279
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -95902.79  87462.97
sample estimates:
mean of x mean of y
 109769.0  113988.9
```

T-test: retrieving information from the result with `broom` package

The `broom` package has a `tidy()` function that can organize results into a data frame so that they are easily manipulated (or nicely printed)

```
result <- t.test(y1980, y1985)
result_tidy <- tidy(result)
glimpse(result_tidy)
```

```
Rows: 1
Columns: 10
$ estimate    <dbl> -4219.909
$ estimate1   <dbl> 109769
$ estimate2   <dbl> 113988.9
$ statistic   <dbl> -0.09053303
$ p.value     <dbl> 0.9279168
$ parameter   <dbl> 340.999
$ conf.low    <dbl> -95902.79
$ conf.high   <dbl> 87462.97
$ method      <chr> "Welch Two Sample t-test"
$ alternative <chr> "two.sided"
```

P-value adjustment

You run an increased risk of Type I errors (a “false positive”) when multiple hypotheses are tested simultaneously.

Use the `p.adjust()` function on a vector of p values. Use `method =` to specify the adjustment method:

```
my_pvalues <- c(0.049, 0.001, 0.31, 0.00001)
p.adjust(my_pvalues, method = "BH") # Benjamini Hochberg
```

```
[1] 0.06533333 0.00200000 0.31000000 0.00004000
```

```
p.adjust(my_pvalues, method = "bonferroni") # multiply by number of tests
```

```
[1] 0.19600 0.00400 1.00000 0.00004
```

```
my_pvalues * 4
```

```
[1] 0.19600 0.00400 1.24000 0.00004
```

See here for more about multiple testing correction. Bonferroni also often done as p value threshold divided by number of tests (0.05/test number).

Some other statistical tests

- `wilcox.test()` – Wilcoxon signed rank test, Wilcoxon rank sum test
- `shapiro.test()` – Test normality assumptions
- `ks.test()` – Kolmogorov-Smirnov test
- `var.test()`– Fisher's F-Test
- `chisq.test()` – Chi-squared test
- `aov()` – Analysis of Variance (ANOVA)

Summary

- Use `cor()` to calculate correlation between two vectors, `cor.test()` can give more information.
- `corrplot()` is nice for a quick visualization!
- `t.test()` one sample test to test the difference in mean of a single vector from zero (one input)
- `t.test()` two sample test to test the difference in means between two vectors (two inputs)
- `tidy()` in the `broom` package is useful for organizing and saving statistical test output
- Remember to adjust p-values with `p.adjust()` when doing multiple tests on data

Lab Part 1

- Class Website
- Lab

Regression

Linear regression

Linear regression is a method to model the relationship between a response and one or more explanatory variables.

Most commonly used statistical tests are actually specialized regressions, including the two sample t-test, see here for more.

Linear regression notation

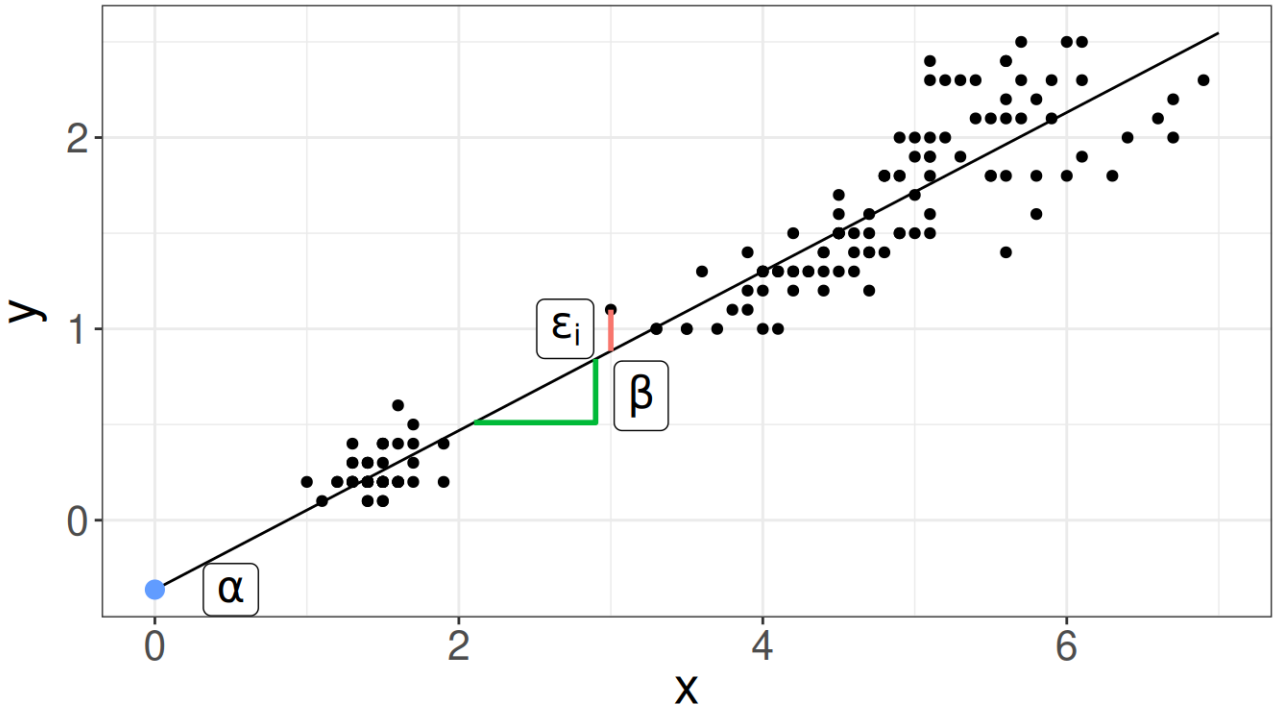
Here is some of the notation, so it is easier to understand the commands/results.

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

where:

- y_i is the outcome for person i
- α is the intercept
- β is the slope (also called a coefficient) - the mean change in y that we would expect for one unit change in x ("rise over run")
- x_i is the predictor for person i
- ε_i is the residual variation for person i

Linear regression



Linear regression

Linear regression is a method to model the relationship between a response and one or more explanatory variables.

We provide a little notation here so some of the commands are easier to put in the proper context.

$$y_i = \alpha + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \varepsilon_i$$

where:

- y_i is the outcome for person i
- α is the intercept
- β_1, β_2, β_3 are the slopes/coefficients for variables x_{i1}, x_{i2}, x_{i3} - average difference in y for a unit change (or each value) in x while accounting for other variables
- x_{i1}, x_{i2}, x_{i3} are the predictors for person i
- ε_i is the residual variation for person i

See this case study for more details.

Linear regression fit in R

To fit regression models in R, we use the function `glm()` (Generalized Linear Model).

You may also see `lm()` which is a more limited function that only allows for normally/Gaussian distributed error terms (aka typical linear regressions).

We typically provide two arguments:

- `formula` – model formula written using names of columns in our data
- `data` – our data frame

Linear regression fit in R: model formula

Model formula

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

In R translates to

```
y ~ x
```

Linear regression fit in R: model formula

Model formula

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

In R translates to

```
y ~ x
```

In practice, `y` and `x` are replaced with the **names of columns from our data set**.

For example, if we want to fit a regression model where outcome is `income` and predictor is `years_of_education`, our formula would be:

```
income ~ years_of_education
```

Linear regression fit in R: model formula

Model formula

$$y_i = \alpha + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \varepsilon_i$$

In R translates to

```
y ~ x1 + x2 + x3
```

In practice, `y` and `x1`, `x2`, `x3` are replaced with the **names of columns from our data set**.

For example, if we want to fit a regression model where outcome is `income` and predictors are `years_of_education`, `age`, and `location` then our formula would be:

```
income ~ years_of_education + age + location
```

Linear regression example

Let's look variables that might be able to predict the number of "crowded" households.

We'll use a dataset that has socioeconomic measures from CDC. Find out more on https://daseh.org/data.

It has already been filtered to include a few counties from Washington State.

Each row represents a census tract/area.

Linear regression example

```
sp_dat <- read_csv(file = "https://daseh.org/data/socioeco_cdc.csv")
```

```
sp_dat
```

```
# A tibble: 927 × 12
   county        fips description    pci    hu munit sngpnt crowd noveh f_sngpnt
   <chr>        <dbl> <chr>        <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl>    <dbl>
 1 Clark  53011040101 Census Trac… 27021  1767     6     52   114   110        0
 2 Clark  53011040102 Census Trac… 28694  1182     0     28   117    13        0
 3 Clark  53011040201 Census Trac… 38606  2430     0    103    25    58        0
 4 Clark  53011040202 Census Trac… 32070  1431     0     69    51     7        0
 5 Clark  53011040203 Census Trac… 33441  1990     0     52    64    59        0
 6 Clark  53011040301 Census Trac… 40783   737     0     37    21    41        0
 7 Clark  53011040302 Census Trac… 42532  3041     0    261    19    60        0
 8 Clark  53011040403 Census Trac… 43666  1639     0     21    10    19        0
 9 Clark  53011040407 Census Trac… 26155  2235   231    138    76   122        0
10 Clark  53011040408 Census Trac… 42397  1269     8     14    28    32        0
# ℹ 917 more rows
# ℹ 2 more variables: f_crowd <dbl>, f_noveh <dbl>
```

Linear regression: model fitting

For this model, we will use two variables:

- **crowd** - At household level (occupied housing units), more people than rooms
- **hu** - Number of housing units

```
fit <- glm(crowd ~ hu, data = sp_dat)
fit
```

```

Call:  glm(formula = crowd ~ hu, data = sp_dat)

Coefficients:
(Intercept)           hu
  -12.43948      0.03547

Degrees of Freedom: 926 Total (i.e. Null);  925 Residual
Null Deviance:      4223000
Residual Deviance: 3428000  AIC: 10250
```

Linear regression: model summary

The `summary()` function returns a list that shows us some more detail

```
summary(fit)

Call:
glm(formula = crowd ~ hu, data = sp_dat)

Coefficients:
             Estimate Std. Error t value            Pr(>|t|)
(Intercept) -12.439483   5.499063  -2.262              0.0239 *
hu            0.035468   0.002422  14.644 <0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 3706.214)

    Null deviance: 4222997  on 926  degrees of freedom
Residual deviance: 3428248  on 925  degrees of freedom
AIC: 10253

Number of Fisher Scoring iterations: 2
```

tidy results

The broom package can help us here too!

The estimate is the coefficient or slope.

for every 1 additional housing unit, we see 0.035 more crowded households (~29 housing units to one more crowded household might make more sense!). This relationship appears to be quite strong, with a p value 7.96e-44!

```
tidy(fit) |> glimpse()
```

```
Rows: 2
Columns: 5
$ term      <chr> "(Intercept)", "hu"
$ estimate  <dbl> -12.43948257, 0.03546794
$ std.error <dbl> 5.499062635, 0.002422068
$ statistic <dbl> -2.26211, 14.64366
$ p.value   <dbl> 0.023921961156326363578950022770186478737741708755…
```

Linear regression: multiple predictors

Let's try adding another other explanatory variable to our model, average per capita income for each census area (`pci`).

```
fit2 <- glm(crowd ~ hu + pci, data = sp_dat)
summary(fit2)
```

```

Call:
glm(formula = crowd ~ hu + pci, data = sp_dat)

Coefficients:
              Estimate Std. Error t value             Pr(>|t|)
(Intercept) 21.8387846  6.3550308   3.436             0.000616 ***
hu           0.0411363  0.0023864  17.238 < 0.0000000000000002 ***
pci         -0.0011459  0.0001198  -9.566 < 0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 3375.923)

    Null deviance: 4222997  on 926  degrees of freedom
Residual deviance: 3119353  on 924  degrees of freedom
AIC: 10167

Number of Fisher Scoring iterations: 2
```

Linear regression: multiple predictors

Can also use `tidy` and `glimpse` to see the output nicely.

```
fit2 |>
  tidy() |>
  glimpse()
```

```
Rows: 3
Columns: 5
$ term      <chr> "(Intercept)", "hu", "pci"
$ estimate  <dbl> 21.838784553, 0.041136308, -0.001145853
$ std.error <dbl> 6.3550308287, 0.0023863704, 0.0001197898
$ statistic <dbl> 3.436456, 17.238023, -9.565525
$ p.value   <dbl> 0.000615646727499959437943100049039912846637889742851257324…
```

Linear regression: factors

Factors get special treatment in regression models - lowest level of the factor is the comparison group, and all other factors are **relative** to its values.

Let's add the county (`county`) as a factor into our model. We'll need to convert it to a factor first.

```
sp_dat <- sp_dat |> mutate(county = factor(county))
```

Linear regression: factors

The comparison group that is not listed is treated as intercept. All other estimates are relative to the intercept.

```
fit3 <- glm(crowd ~ hu + pci + county, data = sp_dat)
summary(fit3)
```

```

Call:
glm(formula = crowd ~ hu + pci + county, data = sp_dat)

Coefficients:
                  Estimate  Std. Error t value             Pr(>|t|)    
(Intercept)      42.4231040   7.4265782   5.712         0.0000000150 ***
hu                0.0393317   0.0022483  17.494 < 0.0000000000000002 ***
pci              -0.0018146   0.0001248 -14.542 < 0.0000000000000002 ***
countyKing       35.4770337   6.3140724   5.619         0.0000000255 ***
countyPierce     -9.9662254   6.7432134  -1.478                0.140    
countySnohomish   5.4030130   6.9562416   0.777                0.438    
countySpokane   -35.1309611   7.5396924  -4.659         0.0000036378 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 2921.148)

    Null deviance: 4222997  on 926  degrees of freedom
Residual deviance: 2687457  on 920  degrees of freedom
AIC: 10037

Number of Fisher Scoring iterations: 2
```

Linear regression: factors

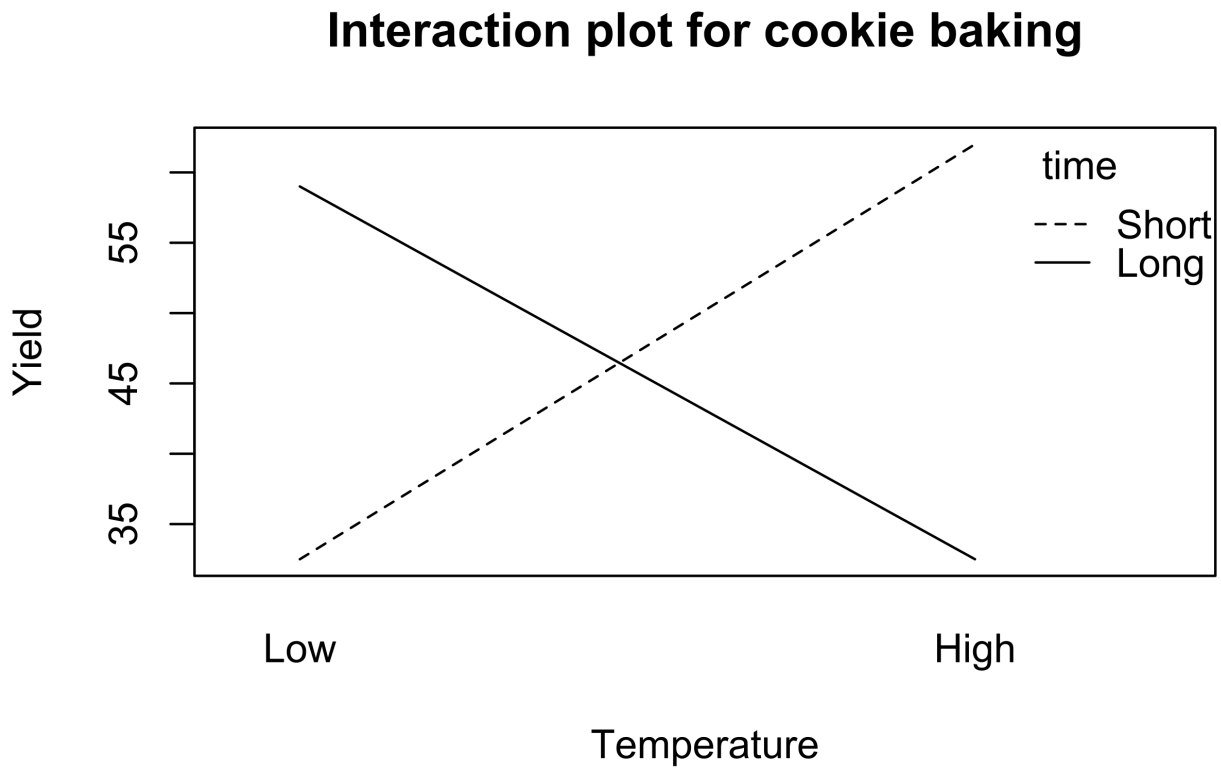
Maybe we want to use King County as our reference. We can relevel the factor.

The counties are relative to the level that is not listed.

```
sp_dat <-
  sp_dat |>
  mutate(county = factor(county,
    levels = c("King", "Clark", "Pierce", "Snohomish", "Spokane")
  ))

fit4 <- glm(crowd ~ hu + pci + county, data = sp_dat)
summary(fit4)
```

```

Call:
glm(formula = crowd ~ hu + pci + county, data = sp_dat)

Coefficients:
                  Estimate  Std. Error t value             Pr(>|t|)
(Intercept)      77.9001378   7.7746490  10.020 < 0.0000000000000002 ***
hu                0.0393317   0.0022483  17.494 < 0.0000000000000002 ***
pci              -0.0018146   0.0001248 -14.542 < 0.0000000000000002 ***
countyClark     -35.4770337   6.3140724  -5.619          0.0000000255 ***
countyPierce    -45.4432591   5.3237881  -8.536 < 0.0000000000000002 ***
countySnohomish -30.0740208   5.3714389  -5.599          0.0000000285 ***
countySpokane   -70.6079948   6.4062707 -11.022 < 0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 2921.148)

    Null deviance: 4222997  on 926  degrees of freedom
Residual deviance: 2687457  on 920  degrees of freedom
AIC: 10037

Number of Fisher Scoring iterations: 2
```

Linear regression: interactions



source

Linear regression: interactions

You can also specify interactions between variables in a formula with `*`. This allows for not only the intercepts between factors to differ, but also the slopes with regard to the interacting variable.

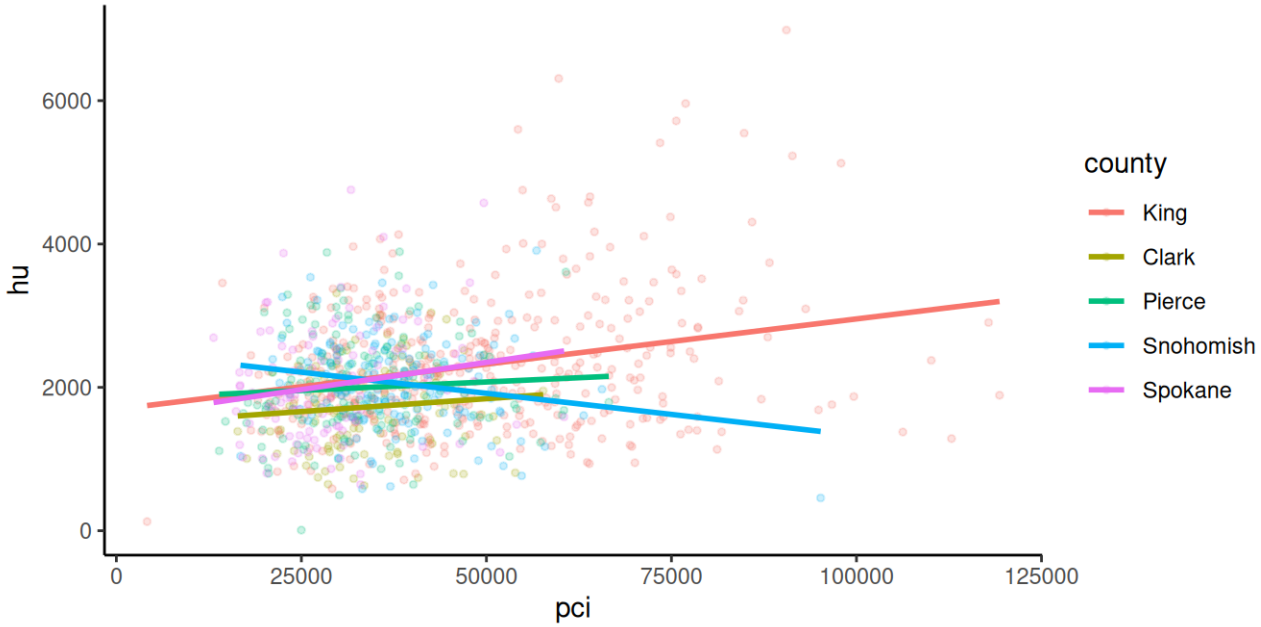
```
fit5 <- glm(crowd ~ hu + pci * county, data = sp_dat)
summary(fit5)
```

```

Call:
glm(formula = crowd ~ hu + pci * county, data = sp_dat)

Coefficients:
                      Estimate  Std. Error t value             Pr(>|t|)
(Intercept)         73.4590707   8.3703790   8.776 < 0.0000000000000002 ***
hu                   0.0389502   0.0022620  17.220 < 0.0000000000000002 ***
pci                 -0.0017060   0.0001442 -11.827 < 0.0000000000000002 ***
countyClark         -1.9469215  23.2496384  -0.084               0.9333
countyPierce       -16.1835554  16.0960723  -1.005               0.3150
countySnohomish    -17.5168167  18.8785553  -0.928               0.3537
countySpokane      -86.9928211  19.3206605  -4.503           0.00000758 ***
pci:countyClark     -0.0009480   0.0006469  -1.465               0.1431
pci:countyPierce    -0.0008377   0.0004379  -1.913               0.0561 .
pci:countySnohomish -0.0003007   0.0004626  -0.650               0.5158
pci:countySpokane    0.0006271   0.0005971   1.050               0.2939
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 2910.925)

    Null deviance: 4222997  on 926  degrees of freedom
Residual deviance: 2666408  on 916  degrees of freedom
AIC: 10038

Number of Fisher Scoring iterations: 2
```

Linear regression: interactions

By default, `ggplot` with a factor added as a color will look include the interaction term. Notice the different intercept and slope of the lines.

```
ggplot(sp_dat, aes(x = pci, y = hu, color = county)) +
  geom_point(size = 1, alpha = 0.2) +
  geom_smooth(method = "glm", se = FALSE) +
  theme_classic()
```



Generalized linear models (GLMs)

Generalized linear models (GLMs) allow for fitting regressions for non-continuous/normal outcomes. Examples include: logistic regression, Poisson regression.

Add the `family` argument – a description of the error distribution and link function to be used in the model. These include:

- `binomial(link = "logit")` - outcome is binary
- `poisson(link = "log")` - outcome is count or rate
- others

Very important to use the right test!

See this case study for more information.

See `?family` documentation for details of family functions.

Logistic regression

Let's look at a logistic regression example. We'll use the `sp_dat` dataset again with a different variable.

- **f_crowd** - Flag for the percentage of crowded households is in the 90th percentile (1 = yes, 0 = no)

There are 36 census tracks in the 90th percentile for crowded households.

```
sp_dat |> count(f_crowd)
```

```
# A tibble: 2 × 2
  f_crowd     n
    <dbl> <int>
1       0   891
2       1    36
```

Logistic regression

Let's explore how `hu`, `pci`, and `county` might predict `f_crowd`.

```
# General format
glm(y ~ x, data = DATASET_NAME, family = binomial(link = "logit"))
```

```
binom_fit <- glm(f_crowd ~ hu + pci + county,
                 data = sp_dat, family = binomial(link = "logit"))
summary(binom_fit)
```

```

Call:
glm(formula = f_crowd ~ hu + pci + county, family = binomial(link = "logit"),
    data = sp_dat)

Coefficients:
                  Estimate  Std. Error z value       Pr(>|z|)
(Intercept)      3.46642282  0.97204634   3.566       0.000362 ***
hu               0.00025822  0.00029372   0.879       0.379314
pci             -0.00019389  0.00003096  -6.263 0.000000000378 ***
countyClark     -2.22310417  0.77710614  -2.861       0.004226 **
countyPierce    -2.49172286  0.59240684  -4.206 0.000025981444 ***
countySnohomish -2.13812813  0.76448803  -2.797       0.005161 **
countySpokane   -3.91697302  1.06411805  -3.681       0.000232 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 304.47  on 926  degrees of freedom
Residual deviance: 199.62  on 920  degrees of freedom
AIC: 213.62

Number of Fisher Scoring iterations: 8
```

Logistic Regression

See this case study for more information.

Odds ratios

An odds ratio (OR) is a measure of association between an exposure and an outcome. The OR represents the odds that an outcome will occur given a particular exposure, compared to the odds of the outcome occurring in the absence of that exposure.

Check out this paper.

Use `oddsratio(x, y)` from the `epitools()` package to calculate odds ratios.

Odds ratios

Let’s see if a high prevalence of no vehicle homes can predict a high prevalence of crowded homes.

- **f_noveh** - Flag for the percentage of households with no vehicles is in the 90th percentile (1 = yes, 0 = no)
- **f_crowd** - Flag for the percentage of crowded households is in the 90th percentile (1 = yes, 0 = no)

Odds ratios

In this case, we're calculating the odds ratio for census areas, indicating whether a prevalence of no vehicle households is associated with more crowded households.

```
library(epitools)

response <- sp_dat %>% pull(f_crowd)
predictor <- sp_dat %>% pull(f_noveh)
```

Odds ratios

The Odds Ratio is 3.33.

When the predictor is 1 (aka the census area has a lot of no vehicle households), the odds of the response (prevalence of crowded homes) are 3.33 times greater than when it is 0 (not a lot of no vehicle households).

```
oddsratio(predictor, response)


$data
         Outcome
Predictor   0  1 Total
    0     849 31   880
    1      42  5    47
    Total 891 36   927

$measure
         odds ratio with 95% C.I.
Predictor estimate    lower    upper
        0 1.000000       NA       NA
        1 3.331243 1.074548 8.385917

$p.value
         two-sided
Predictor midp.exact fisher.exact chi.square
        0         NA           NA         NA
        1 0.03856847   0.03106555 0.01389049

$correction
[1] FALSE

attr(,"method")
[1] "median-unbiased estimate & mid-p exact CI"
```

Final note

Some final notes:

- Researcher's responsibility to **understand the statistical method** they use – underlying assumptions, correct interpretation of method results
- Researcher's responsibility to **understand the R software** they use – meaning of function's arguments and meaning of function's output elements

Summary

- `glm()` fits regression models:
  - Use the `formula =` argument to specify the model (e.g., `y ~ x` or `y ~ x1 + x2` using column names)
  - Use `data =` to indicate the dataset
  - Use `family =` to do a other regressions like logistic, Poisson and more
  - `summary()` gives useful statistics
- `oddsratio()` from the `epitools` package can calculate odds ratios (outside of logistic regression - which allows more than one explanatory variable)
- this is just the tip of the iceberg!

Resources (also on the website!)

For more check out:

- this chapter on modeling in this tidyverse book
- this chart on when to do what test
- opencasestudies.org

Content for similar topics as this course can also be found on Leanpub.

Lab Part 2

- Class Website
- Lab
- Day 8 Cheatsheet

Image by Gerd Altmann from Pixabay

Extra Slides

Model Selection

Check out the `leaps` package and other code snippets here: https://r-statistics.co/Model-Selection-in-R.html

More about tests!

Wilcoxon Test

The Wilcoxon test is a good alternative to the t-test when the normal distribution of the differences between paired individuals cannot be assumed.

```
wilcox.test(x, y, ..)
```

- Like t-test, provide one or two vectors (x, y)
- Choose from `alternative = c("two.sided", "less", "greater")`
- Use `paired = TRUE` for paired values (e.g., before and after)

Shapiro Test

Can tell you if a vector is normally distributed.

```
shapiro.test(x)
```

The smaller the p-value, the more likely the data violates normality assumptions.

Kolmogorov-Smirnov test

Can tell you if two groups come from different distributions.

`ks.test()`

The smaller the p-value, the more likely the data are from different distributions.

Fisher's F-Test

Performs an F test to compare the variances of two samples from normal populations.

```
var.test()
```

Chi-squared test

For categorical data/ratios, can tell you if observations match expected values or if two categorical variables are independent.

```
chisq.test()
```

Analysis of Variance (ANOVA)

For balanced designs, determine if multiple variables influence a dependent variable. “Within versus among group variance”.

`aov()`

More on Linear regression: factors

You can view estimates for the comparison group by removing the intercept in the GLM formula

`y ~ x - 1`

Caveat is that the p-values change, and interpretation is often confusing.

```
fit_force_intercept <-
  glm(crowd ~ pci + sngpnt + county - 1, data = sp_dat)
summary(fit_force_intercept)
```

```

Call:
glm(formula = crowd ~ pci + sngpnt + county - 1, data = sp_dat)

Coefficients:
                  Estimate Std. Error t value             Pr(>|t|)
pci             -0.0007588  0.0001463  -5.188        0.00000026132 ***
sngpnt           0.2379315  0.0215775  11.027 < 0.0000000000000002 ***
countyKing      84.8722295  9.2681555   9.157 < 0.0000000000000002 ***
countyClark     44.0054938  8.7528775   5.028        0.00000059706 ***
countyPierce    36.2601486  8.4082758   4.312        0.00001789033 ***
countySnohomish 52.3318984  8.9457720   5.850        0.00000000683 ***
countySpokane   15.4316228  8.8378663   1.746               0.0811 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 3438.458)

    Null deviance: 7852783  on 927  degrees of freedom
Residual deviance: 3163382  on 920  degrees of freedom
AIC: 10188

Number of Fisher Scoring iterations: 2
```

Data Output

Data Output

While it's nice to be able to read in a variety of data formats, it's equally important to be able to output data somewhere.

The `readr` package provides data exporting functions which have the pattern `write_*`:

- `write_csv()`,
- `write_delim()`, others.

From `write_csv()` documentation:

```
write_csv(x, file,
  na = "NA", append = FALSE,
  col_names = !append, quote_escape = "double",
  eol = "\n", path = deprecated()
)
```

```
Rows: 768 Columns: 6
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (1): county
dbl (5): rate, lower95cl, upper95cl, visits, year

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

Data Output

`x`: data frame you want to write

`file`: file path where you want to R object written; it can be:

- an absolute path,
- a relative path (relative to your working directory),
- a file name only (which writes the file to your working directory)
- remember to include the file extension (`.csv`, `.txt`, or `.tsv`)

```
# Examples

write_csv(dat, file = "CO_ER_heat_newNames.csv")

write_delim(dat, file = "CO_ER_heat_newNames.csv", delim = ",")
```

GUT CHECK!

What does `write_csv()` do? Saves data to..

A. R's memory

B. A file on your hard drive

C. A ggplot

R binary file

`.rds` is an extension for R native file format.

`write_rds()` and `read_rds()` from `readr` package can be used to write/read a single R object to/from file.

Saving datasets in `.rds` format can save time if you have to read it back in later.

```
# write an object: a data frame "dat"
write_rds(er, file = "CO_heat_dataset.rds")

# write an object: vector "x"
x <- c(1, 3, 3)
write_rds(x, file = "my_vector.rds")

# read an object from file and assign to a new object named "y"
x2 <- read_rds(file = "my_vector.rds")
x2
```

```
[1] 1 3 3
```

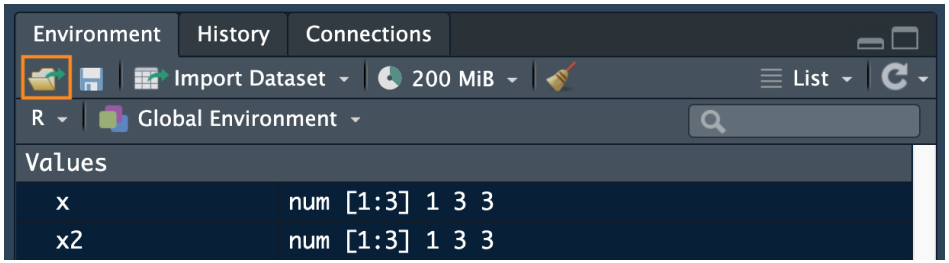
Saving multiple objects

You may want to export a set of objects from R for later use, either to save time or to use in another R script. You can output these to an `.RData` file individually, or save your entire Environment with `save.image()`.

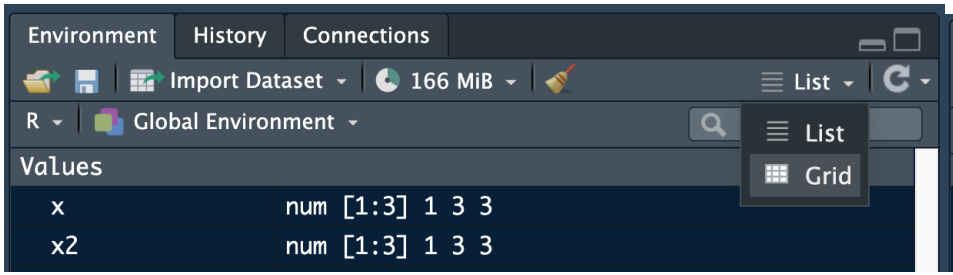
```
save(x, x2, file = "x_x2_output.RData")
save.image(file = "my_environment.RData")
```

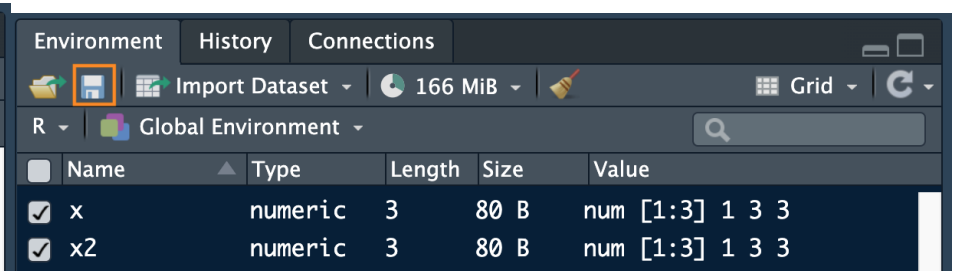
Using RStudio for importing/exporting data

If there is an `.rds` or `.RData` file that you want to work with, you can open it into your environment using the file icon.



Can also save your entire environment or a subset of objects in your environment to a new `.RData` file with the save icon. Click the "List" button and switch to "Grid" to select which objects to delete or keep before saving the Environment.





REMINDER: Saving a ggplot to file

A few options:

- RStudio > Plots > Export > Save as image / Save as PDF
- RStudio > Plots > Zoom > [right mouse click on the plot] > Save image as
- In the code

```
ggsave(filename = "saved_plot.png",  # will save in working directory
       plot = rp_fac_plot,
       width = 6, height = 3.5)              # by default in inches
```

Summary

- Use `write_csv()` and `write_delim()` from the `readr` package to write your (modified) data
- `.rds` files can be handy for saving intermediate work
- Can save environment (or subset) using `save()` and `save.image()`

- Class Website
- Data Output Lab
- Posit's Data Import Cheatsheet
- Day 8 Cheatsheet

Image by Gerd Altmann from Pixabay

Functions

Writing your own functions

So far we've seen many functions, like `c()`, `class()`, `filter()`, `dim()` ...

Why create your own functions?

- Cut down on repetitive code (easier to fix things!)
- Organize code into manageable chunks
- Avoid running code unintentionally
- Use names that make sense to you

Writing your own functions

The general syntax for a function is:

```
my_function <- function(argument) {
 <function body>
}
```

OR

```
my_function <- \(argument) {
 <function body>
}
```

Writing your own functions

Here we will write a function that divides some number `x` by 100:

```
div_100 <- function(x) x / 100
```

When you run the line of code above, you make it ready to use (no output yet!).

Let's test it:

```
div_100(x = 600)
```

```
[1] 6
```

Writing your own functions

We can take all kinds of arguments. Here is one with text:

```
greeting <- function(name) paste("Hello my name is", name)
```

Let's test it:

```
greeting(name = "Ava") # Named argument
```

```
[1] "Hello my name is Ava"
```

```
greeting("Ava") # R guesses argument based on position
```

```
[1] "Hello my name is Ava"
```

Writing your own functions: `{ }`

Adding the curly braces - `{}` - allows you to use functions spanning multiple lines:

```
div_100 <- function(x) {
  x / 100
}
div_100(x = 10)
```

```
[1] 0.1
```

Writing your own functions: **return**

If we want something specific for the function's output, we use `return()`:

```
div_100_plus_4 <- function(x) {
  output_int <- x / 100
  output <- output_int + 4
  return(output)
}
div_100_plus_4(x = 10)
```

```
[1] 4.1
```

Writing your own functions: multiple inputs

Functions can take multiple inputs:

```
div_100_plus_y <- function(x, y) x / 100 + y
div_100_plus_y(x = 10, y = 3)
```

```
[1] 3.1
```

Writing your own functions: multiple outputs

Functions can return a vector (or other object) with multiple outputs.

```
x_and_y_plus_2 <- function(x, y) {
  output1 <- x + 2
  output2 <- y + 2

  return(c(output1, output2))
}
result <- x_and_y_plus_2(x = 10, y = 3)
result
```

```
[1] 12  5
```

Writing your own functions: defaults

Functions can have “default” arguments. This lets us use the function without using an argument later:

```
div_100_plus_y <- function(x = 10, y = 3) x / 100 + y
div_100_plus_y()
```

```
[1] 3.1
```

```
div_100_plus_y(x = 11, y = 4)
```

```
[1] 4.11
```

Writing another simple function

Let's write a function, `sqdif`, that:

1. takes two numbers `x` and `y` with default values of 2 and 3.
2. takes the difference
3. squares this difference
4. then returns the final value

Writing another simple function

```
sqdif <- function(x = 2, y = 3) (x - y)^2
```

```
sqdif()
```

```
[1] 1
```

```
sqdif(x = 10, y = 5)
```

```
[1] 25
```

```
sqdif(10, 5)
```

```
[1] 25
```

```
sqdif(11, 4)
```

```
[1] 49
```

Writing your own functions: characters

Again, functions can have any kind of input.

```
loud <- function(word = "hooray!") {
  output <- rep(toupper(word), 5)
  return(output)
}

loud()
```

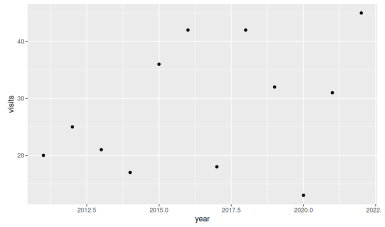
```
[1] "HOORAY!" "HOORAY!" "HOORAY!" "HOORAY!" "HOORAY!"
```

```
loud("wow!")
```

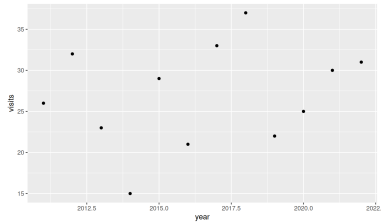
```
[1] "WOW!" "WOW!" "WOW!" "WOW!" "WOW!"
```

Functions for tibbles - Example with ggplot

```
er <- read_csv(file = "https://daseh.org/data/CO_ER_heat_visits.csv")

visits_plot <- function(the_county){
   er_sub <- er |> filter(county == the_county)
   ggplot(data = er_sub, aes(x = year, y = visits)) +
     geom_point()
 }

visits_plot("Larimer")
```




```
visits_plot("Weld")
```



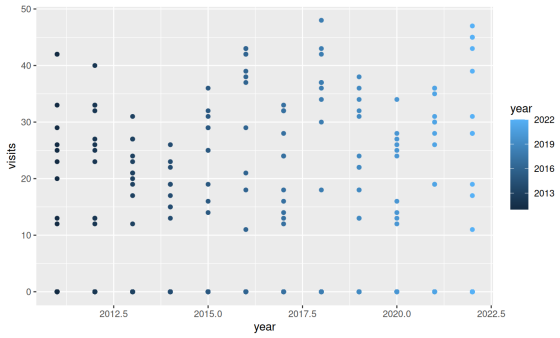
Functions for tibbles - curly braces

Tell tidyverse functions that you mean the **column** not an **object** with curly braces (`{}`):

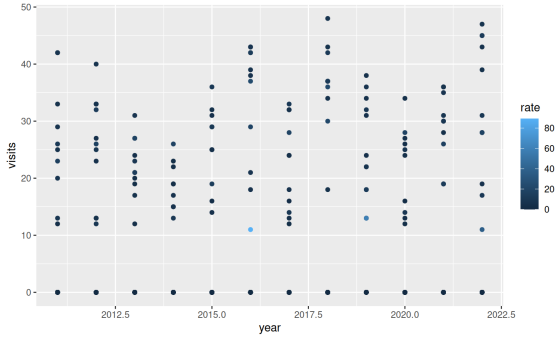
```
visits_plot2 <- function(the_column){
   ggplot(data = er, aes(x = year, y = visits, color = {{the_column}})) +
     geom_point()
 }
```

Functions for tibbles - example

```
visits_plot2(year)
```




```
visits_plot2(rate)
```



Summary

- Simple functions take the form:
  - `NEW_FUNCTION <- function(x, y){x + y}`
  - Can specify defaults like `function(x = 1, y = 2){x + y}`
  - `return` will provide a value as output
- Specify a column (from a tibble) inside a function using `{{double curly braces}}`

Lab Part 1

- Class Website
- Lab

Functions on multiple columns

Using your custom functions: `sapply()` - a base R function

Now that you've made a function… You can "apply" functions easily with `sapply()`!

These functions take the form:

```
sapply(<a vector, list, data frame>, some_function)
```

Using your custom functions: `sapply()`

Let's apply a function to look at the CO heat-related ER visits dataset.

 There are no parentheses on the functions!

You can also pipe into your function.

```
sapply(er, class)
```

```
    county       rate   lower95cl   upper95cl     visits       year 
"character"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric" 
```

```
# also: er |> sapply(class)
```

Using your custom functions: `sapply()`

Use the `div_100` function we created earlier to convert 0-100 percentiles to proportions.

```
er |>
  select(ends_with("cl")) |>
  sapply(div_100) |>
  head()
```

```
      lower95cl  upper95cl
[1,]         NA 0.09236776
[2,] 0.02848937         NA
[3,] 0.04359735 0.09313561
[4,] 0.01711087 0.04846996
[5,] 0.01892912 0.05232461
[6,] 0.06124961 0.11572046
```

Using your custom functions “on the fly” to iterate

Also called “anonymous function”.

```
er |>
  select(ends_with("cl")) |>
  sapply(function(x) x / 100) |>
  head()
```

```
      lower95cl  upper95cl
[1,]         NA 0.09236776
[2,] 0.02848937         NA
[3,] 0.04359735 0.09313561
[4,] 0.01711087 0.04846996
[5,] 0.01892912 0.05232461
[6,] 0.06124961 0.11572046
```

Anonymous functions: alternative syntax

```
er |>
  select(ends_with("cl")) |>
  sapply(\(x) x / 100) |>
  head()
```

```
      lower95cl  upper95cl
[1,]         NA 0.09236776
[2,] 0.02848937         NA
[3,] 0.04359735 0.09313561
[4,] 0.01711087 0.04846996
[5,] 0.01892912 0.05232461
[6,] 0.06124961 0.11572046
```

across

Using functions in `mutate()` and `summarize()`

Already know how to use functions to modify columns using `mutate()` or calculate summary statistics using `summarize()`.

```
er |>
  summarize(max_visits = max(visits, na.rm = T),
            max_rate = max(rate, na.rm = T))

# A tibble: 1 × 2
  max_visits max_rate
       <dbl>    <dbl>
1         48     89.3
```

Applying functions with **across** from **dplyr**

`across()` makes it easy to apply the same transformation to multiple columns. Usually used with `summarize()` or `mutate()`.

```
summarize(across(<columns>,function))
```

or

```
mutate(across(<columns>,function))
```

- List columns first : `.cols =`
- List function next: `.fns =`
- If there are arguments to a function (e.g., `na.rm = TRUE`), use an anonymous function.

Applying functions with **across** from **dplyr**

Combining with `summarize()`

```
er |>
  summarize(across(
    c(visits, rate),
    mean # no parentheses
  ))

# A tibble: 1 × 2
  visits  rate
   <dbl> <dbl>
1     NA    NA
```

Applying functions with **across** from **dplyr**

Add anonymous function to include additional arguments (e.g., `na.rm = T`).

```
er |>
  summarize(across(
    c(visits, rate),
    function(x) mean(x, na.rm = T)
  ))

# A tibble: 1 × 2
  visits  rate
   <dbl> <dbl>
1   7.19  2.43
```

Applying functions with **across** from **dplyr**

Can use with other tidyverse functions like `group_by`!

```
er |>
  group_by(year) |>
  summarize(across(
    c(visits, rate),
    function(x) mean(x, na.rm = T)
  ))
```

```
# A tibble: 12 × 3
    year visits  rate
   <dbl>  <dbl> <dbl>
 1  2011   5.20  1.49
 2  2012   5.89  1.75
 3  2013   5.63  1.83
 4  2014   4.12  1.41
 5  2015   6.4   1.96
 6  2016  10.1   5.28
 7  2017   7.24  2.13
 8  2018  11.7   3.28
 9  2019   9.12  4.09
10  2020   6.26  1.73
11  2021   8.06  2.08
12  2022   9.29  3.21
```

Applying functions with **across** from **dplyr**

Using different `tidyselect()` options (e.g., `starts_with()`, `ends_with()`, `contains()`)

```
er |>
  group_by(year) |>
  summarize(across(
    contains("cl"),
    function(x) mean(x, na.rm = TRUE)
  ))
```

```
# A tibble: 12 × 3
    year lower95cl upper95cl
   <dbl>     <dbl>     <dbl>
 1  2011     0.836      2.12
 2  2012     1.06       2.41
 3  2013     1.07       2.62
 4  2014     0.810      2.11
 5  2015     1.21       2.77
 6  2016     3.05       7.99
 7  2017     1.28       3.08
 8  2018     2.17       4.41
 9  2019     2.32       6.21
10  2020     1.02       2.52
11  2021     1.30       2.92
12  2022     1.93       4.71
```

Applying functions with **across** from **dplyr**

Combining with `mutate()` - the `replace_na` function

Let's look at the yearly CO2 emissions dataset.

```
yearly_co2 <-
  read_csv(file = "https://daseh.org/data/Yearly_CO2_Emissions_1000_tonnes.csv")

yearly_co2 |>
  select(country, starts_with("194")) |>
  mutate(across(
    c(`1943`, `1944`, `1945`),
    function(x) replace_na(x, replace = 0)
  ))
```

```
# A tibble: 192 × 11
   country        `1940` `1941` `1942` `1943` `1944` `1945` `1946` `1947` `1948`
   <chr>           <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Afghanistan        NA     NA     NA      0      0      0     NA     NA     NA
 2 Albania           693    627    744    462    154    121    484    928    704
 3 Algeria           238    312    499    469    499    616    763    744    803
 4 Andorra            NA     NA     NA      0      0      0     NA     NA     NA
 5 Angola             NA     NA     NA      0      0      0     NA     NA     NA
 6 Antigua and B…     NA     NA     NA      0      0      0     NA     NA     NA
 7 Argentina       15900  14000  13500  14100  14000  13700  13700  14500  17400
 8 Armenia           848    745    513    655    613    649    730    878    935
 9 Australia       29100  34600  36500  35000  34200  32700  35500  38000  38500
10 Austria          7350   7980   8560   9620   9400   4570  12800  17600  24500
# ℹ 182 more rows
# ℹ 1 more variable: `1949` <dbl>
```

GUT CHECK!

Why use `across()`?

A. Efficiency - faster and less repetitive

B. Calculate the cross product

C. Connect across datasets

`purrr` package

Similar to across, `purrr` is a package that allows you to apply a function to multiple columns in a data frame or multiple data objects in a list.

While we won't get into `purrr` too much in this class, it's part of the tidyverse and is great if you're doing lots of iterative work!

One cool function: `modify_if()`

Columns must meet a certain criteria to be modified.

```
er |>
  modify_if(is.numeric, \(x) round(x)) |>
  glimpse()
```

```
Rows: 768
Columns: 6
$ county    <chr> "Adams", "Adams", "Adams", "Adams", "Adams", "Adams", "Adams…
$ rate      <dbl> 7, 5, 7, 3, 3, 9, 7, 7, 7, 5, 7, 8, 0, 0, NA, 0, NA, NA, NA,…
$ lower95cl <dbl> NA, 3, 4, 2, 2, 6, 4, 5, 5, 3, 5, 6, 0, 0, NA, 0, NA, NA, NA…
$ upper95cl <dbl> 9, NA, 9, 5, 5, 12, 9, 9, 9, 7, 9, 11, 0, 0, NA, 0, NA, NA, …
$ visits    <dbl> 29, 23, 31, 15, 16, 42, 32, 37, 36, 24, 35, 45, 0, 0, NA, 0,…
$ year      <dbl> 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, …
```

Summary

- Apply your functions with `sapply(<a vector or list>, some_function)`
- Use `across()` to apply functions across multiple columns of data
- Need to use `across` within `summarize()` or `mutate()`
- Check out `purrr` if you want to get into the weeds!

Lab Part 2

- Class Website
- Lab
- Day 9 Cheatsheet
- Posit's `purrr` Cheatsheet

Research Survey

https://forms.gle/5ac2WPiR9kJu5D3c6

Image by Gerd Altmann from Pixabay