STAT 234: Data Science

Matt Higham

2021-08-09

Contents

Chapter 1

Syllabus and Course Information

1.1 General Information

Instructor Information

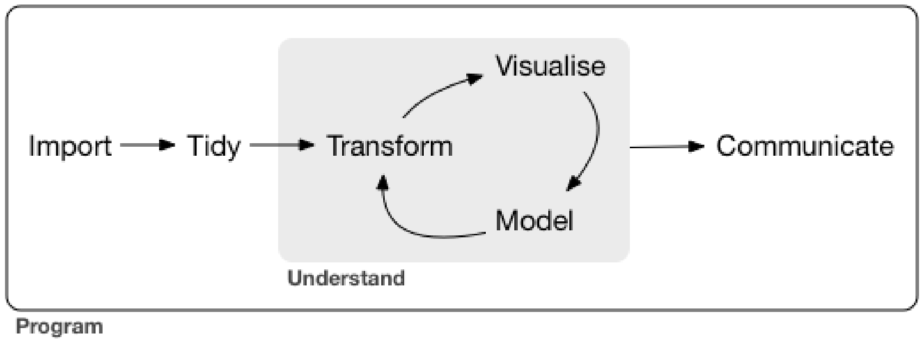
- Professor: Matt Higham
- Office: Bewkes 123
- Email: mhigham@stlawu.edu
- Semester: Fall 2021
- Office Hours:
  - Tuesday 1:30 - 3:30
  - Wednesday 2:30 - 3:30
  - Friday 10:00 - 11:00
  - other times by appointment
  - all in-person
- Sections:
  - MW 8:50 - 10:20

Course Materials

- STAT 234 Materials Bundle. This will be our primary source of materials.
- Textbooks (only used as references):
  - Modern Data Science with `R` by Baumer, Kaplan, and Horton, found here in a free online version.
  - R for Data Science by Grolemund and Wickham, found here in a free online version.
- Computer with Internet access.

1.2 Course Information

Welcome to STAT 234! The overall purpose of this course is learn the data science skills necessary to complete large-scale data analysis projects. The tool that we will be using to achieve this goal is the statistical software language `R`. We will work with a wide variety of interesting data sets throughout the semester to build our `R` skills. In particular, we will focus on the Data Analysis Life Cycle (Grolemund and Wickham 2020):



We will put more emphasis on the *Import, Tidy, Transform, Visualize, and Communicate* parts of the cycle, as an introduction to *Modeling* part is covered in STAT 213.

1.3 General Course Outcomes

1. *Import* data of a few different types into `R` for analysis.
2. *Tidy* data into a form that can be more easily visualized, summarised, and modeled.
3. *Transform*, *Wrangle*, and *Visualize* variables in a data set to assess patterns in the data.
4. *Communicate* the results of your analysis to a target audience with a written report, or, possibly an oral presentation.
5. Practice reproducible statistical practices through the use of `R Markdown` for data analysis projects.

6. Explain why it is ethically important to consider the context that a data set comes in.
7. Develop the necessary skills to be able to ask and answer future data analysis questions on your own, either using `R` or another program, such as Python.

To paraphrase the *R for Data Science* textbook, about 80% of the skills necessary to do a complete data analysis project can be learned through coursework in classes like this one. But, 20% of any particular project will involve learning new things that are specific to that project. Achieving Goal # 6 will allow you to learn this extra 20% on your own.

1.3.1 Use of `R` and `RStudio`

We will use the statistical software `R` to construct graphs and analyze data. A few notes:

- `R` and `RStudio` are both free to use.
- We will primarily be using the SLU `R Studio` server at first: Link to R Studio Server.
- Additionally, we will be using `RMarkdown` for data analysis reports. *Note*: It's always nice to start assignments and projects as early as possible, but this is particularly important to do for assignments and projects involving `R`. It's no fun to try and figure out why code is not working at the last minute. If you start early enough though, you will have plenty of time to seek help and therefore won't waste a lot of time on a coding error.

1.4 How You Will Be Assessed

The components to your grade are described below:

- Class

Class participation will be assessed three times throughout the semester in a 20 point rubric for a total of 60 points. Additionally, there will be a 10-point "share something interesting you found with the class" assignment on very Wednesday, where, two students will volunteer to....share something interesting that they found with the data set we were working with with the rest of the class. The rubric used will be shared on the first day of class, and more information about the Wednesday 10 points will also be given on the first day of class.

- Exercises

There are about 14 sets of weekly exercises that often require you to read some of the sections in the STAT 234 Materials Bundle first. These are worth either 10 or 5 points, depending on the length of the exercises, for a total of 100 points. Most weeks toward the beginning of the semester will be 10 point weeks because we won't have any projects to work on. Exercises are graded **for completion only**: for many exercises, the solutions are provided in our course materials.

- Quizzes

There will be 10 Quizzes, each worth 20 points for a total of 180 points with one dropped quiz. The purpose of the quizzes are for you to practice what you've learned for the week in a short, concise format. Quizzes will consist of two parts: (1) a take-home component and (2) an in-class component. The take-home component should take about 15 minutes. You are allowed use any course materials and you are allowed to work with other students in this course, as long as you list the names of those students at the top of your quiz. The in-class component will be 5 minutes. You will be asked to do a simple task with pen and paper, without using any course notes or materials.

- Mini Projects

There are 3 mini-projects scattered throughout the semester that are worth 60 points each. Each mini-project will have some prescriptive tasks and questions that you will investigate as well as a section where you come up with and subsequently answer your own questions relating to the data set.

In order to get experience with oral presentations of results, each student will give a short oral presentation on **1** mini-project. Use of `R Markdown` is required for this presentation (as opposed to PowerPoint or Prezi). More details will be given later in the semester.

- Midterm Exams

There will be two midterm exams, each worth 150 points. More information will be given about these later.

- Final Project

There is one final project, worth 150 points. The primary purpose of the final project is to give you an opportunity to assemble topics throughout the course into one coherent data analysis. More information about the final project will be given later.

There will be no Final Exam for this course.

1.4.1 Breakdown

- 70 points for Class
- 100 points for Exercises
- 180 points for Quizzes

- 180 points for Mini-Projects + 20 points for Presentation
- 150 points for each of two Midterm Exams
- 150 points for Final Project

Points add up to 1000 so your grade at the end of the semester will be the number of points you've earned across all categories divided by 1000.

- The tutorials should help you complete the exercises sets, which should
  - help you to do well on the quizzes, which should
    - help you complete the mini-projects, which should
      - help you to do well on the midterm exams.

Then, everything together should help you create an awesome final project!

1.4.2 Grading Scale

The following is a *rough* grading scale. I reserve the right to make any changes to the scale if necessary.

Grade	4.0	3.75	3.5	3.25	3.0	2.75	2.5	2.25	2.0	1.75	1.5	1.25	1.0	0.0
Points	950-1000	920-949	890-919	860-889	830-859	810-829	770-809	750-769	720-749	700-719	670-699	640-669	600-639	0-599

1.4.3 Rules for Collaboration

Collaboration with your classmates on handouts, tutorials, and projects is encouraged, but you must follow these guidelines:

- you must state the name(s) of who you collaborated with at the top of each assessment.
- all work must be your own. This means that you should **never** send someone your code via email or let someone directly type code off of your screen. Instead, you can talk about strategies for solving problems and help or ask someone about a coding error.
- you may use the Internet and StackExchange, but you also should not copy paste code directly from the website, without citing that you did so.
- this isn't a rule, but keep in mind that collaboration is not permitted on quizzes, exams, and very limited collaboration will be permitted on the final project. Therefore, when working with someone, make sure that you are both really learning so that you both can have success on the non-collaborative assessments.

1.4.4 Diversity Statement

Diversity encompasses differences in age, colour, ethnicity, national origin, gender, physical or mental ability, religion, socioeconomic background, veteran status, sexual orientation, and marginalized groups. The interaction of different human characteristics brings about a positive learning environment. Diversity is both respected and valued in this classroom.

1.4.5 Accessibility Statement

If you have a learning difference/disability or other health impairment and need accommodations please be sure to contact the Student Accessibility Services Office right away so they can help you get the accommodations you require. If you need to use any accommodations in this class, please meet with your instructor early and provide them with your Individualized Educational Accommodation Plan (IEAP) letter so you can have the best possible experience this semester.

Although not required, your instructor would like to know of any accommodations that are needed at least 10 days before a quiz or test. Please be proactive and set up an appointment to meet with someone from the Student Accessibility Services Office.

Color-Vision Deficiency: If you are Color-Vision Deficient, the Student Accessibility Services office has on loan glasses for students who are color vision deficient. Please contact the office to make an appointment.

For more specific information about setting up an appointment with Student Accessibility Services please see the listed options below:

- Telephone: 315.229.5537
- Email: studentaccessibility@stlawu.edu

For further information about Student Accessibility Services you can check the website at: https://www.stlawu.edu/student-accessibility-services

1.4.6 Academic Dishonesty

Academic dishonesty will not be tolerated. Any specific policies for this course are supplementary to the

Honor Code. According to the St. Lawrence University Academic Honor Policy,

1. It is assumed that all work is done by the student unless the instructor/mentor/employer gives specific permission for collaboration.
2. Cheating on examinations and tests consists of knowingly giving or using or attempting to use unauthorized assistance during examinations or tests.
3. Dishonesty in work outside of examinations and tests consists of handing in or presenting as original work which is not original, where originality is required.

Claims of ignorance and academic or personal pressure are unacceptable as excuses for academic dishonesty. Students must learn what constitutes one's own work and how the work of others must be acknowledged.

For more information, refer to www.stlawu.edu/acadaffairs/**academic_honor**_policy.pdf.

To avoid academic dishonesty, it is important that you follow all directions and collaboration rules and ask for clarification if you have any questions about what is acceptable for a particular assignment or exam. If I suspect academic dishonesty, a score of zero will be given for the entire assignment in which the academic dishonesty occurred **for all individuals involved** and Academic Honor Council will be notified. If a pattern of academic dishonesty is found to have occurred, a grade of 0.0 for the entire course can be given.

It is important to work in a way that maximizes your learning. Be aware that students who rely too much on others for the homework and projects tend to do poorly on the quizzes and exams.

Please note that in addition the above, any assignments in which your score is reduced due to academic dishonesty will not be dropped according to the quiz policy e.g., if you receive a zero on a quiz because of academic dishonesty, it will not be dropped from your grade.

1.5 Tentative Schedule

Week	Date	Topics
0	8/25	Introduction to `R`, `R Studio`
1	8/30	Graphics with `ggplot2`
2	9/6	Data Wrangling and Transformation with `dplyr`

Week	Date	Topics
3	9/13	Data Tidying with `tidyr`
4	9/20	Communication with `R Markdown` and `ggplot2`
5	9/27	Basic Coding in `R`
6	10/4	Catch-up and Midterm 1
7	10/11	Factors with `forcats` and Data Ethics
8	10/18	Data Import with `readr`, `jsonlite`, `rvest`, and `tibble`
9	10/25	Data Merging with `dplyr`
10	11/1	Dates and Times with `lubridate`
11	11/8	Strings with `stringr`
12	11/15	Catch-up and Midterm 2
13	11/22	Thanksgiving Break
14	11/29	Predictive Modeling Final Project
14	12/6	Final Project

- The three mini-projects are tentatively scheduled to be due on September 27, October 25, and November 8, though these are subject to change.
- There will be no Final Exam, but keep your schedule open at our Final Exam time in case we decide to use it for something.

Chapter 2

Getting Started with `R` and `R Studio`

Goals:

1. Use `R Studio` on the server
2. Use `R Markdown` and code chunks
3. Load in data to `R Studio`
4. Run code and change a few things within that code
5. Correct some common errors when running code in `R`

2.1 Intro to `R` and `R Studio`

`R` is a statistical computing software used by many statisticians as well as professionals in other fields, such as biology, ecology, business, and psychology. The goal of Week 0 is to provide basic familiarity with `R` and `R Markdown`, which we will be using for the entire semester.

Open `R Studio` on the SLU `R Studio` server at http://rstudio.stlawu.local:8787 and create a folder called STAT_234 or some other meaningful title to you. Note that you must be on campus to use the `R Studio` server, unless you use a VPN. Directions on how to set-up VPN are https://infotech.stlawu.edu/support/content/11269 <> for Macs and https://stlawu.teamdynamix.com/TDClient/1805/Portal/KB/ArticleDet?ID=55118 for Windows.

Next, create a subfolder within your STAT_234 folder. Title it *Notes* (or whatever you want really).

Then, create an `R Project` by Clicking File -> New Project -> Existing Directory, navigate to the *Notes* folder, and click *Create Project*.

Within this folder, click the *New Folder* button in your bottom-left window and name a new folder *data*. Then, download the data.zip file from Sakai (in Resources). Upload that file in to the server by clicking "Upload" in the bottom right panel. In the dialog box that appears, you can click "Choose File" and navigate to the folder where you saved the zip file (probably Downloads by default). The zip file will automatically expand once uploaded. It includes data sets that we will use throughout the course.

Finally, we want to create a new `R Markdown` file by clicking File -> New File -> `R Markdown`. You can give your new `R Markdown` file a title if you want, and then click okay.

Before moving on, click the **Knit** button in the top-left window at the top of the menu bar (look for the knitting needle icon). Make sure that the file knits to a pretty-looking .html file. The newly knitted .html file can now be found in your folder with your `R` project.

2.2 What are `R`, `R Studio`, and `R Markdown`?

The distinction between the 3 will become more clear later on. For now, * `R` is a statistical coding software used heavily for data analysis and statistical procedures.

- `R Studio` is a nice IDE (Integrated Development Environment) for `R` that has a lot of convenient features. Think of this as just a convenient User Interface.
- `R Mardkown` allows users to mix regular Microsoft-Word-style text with code. The `.Rmd` file ending denotes an `R Mardkown` file. `R Markdown` has many options that we will use heavily throughout the semester, but there's no need to worry about these now.

2.2.1 `R` Packages and the `tidyverse`

You can think of `R` packages as add-ons to `R` that let you do things that `R` on its own would not be able to do. If you're in to video games, you can think of `R` packages as extra Downloadable Content (DLC). But, unlike most gaming DLC, `R` packages are always free and we will make very heavy use of `R` packages.

The `tidyverse` is a series of `R` packages that are useful for data science. In the order that we will encounter them in this class, the core `tidyverse` packages are:

1. `ggplot2` for plotting data

2. `dplyr` for data wrangling and summarizing
3. `tidyr` for data tidying and reshaping
4. `readr` for data import
5. `tibble` for how data is stored
6. `stringr` for text data
7. `forcats` for factor (categorical) data
8. `purrr`, for functional programming, the only one of these core 8 that we won't get to use

We will use packages outside of the core `tidyverse` as well, but the `tidyverse` is the main focus.

We are going to change one option before proceeding. In the top file menu, click Tools -> Global Options -> R Markdown and then uncheck the box that says "Show output inline for all R Markdown documents". Don't worry about this for now, but changing this option just means that code results will appear in the bottom-left window and graphs will appear in the bottom-right window of `R Studio`.

2.3 Putting Code in a `.Rmd` File

The first thing that we will do that involves code is to load a package into `R` with the `library()` function. A package is just an `R` add-on that lets you do more than you could with just `R` on its own. Load the `tidyverse` package into `R` by typing and running the `library(tidyverse)` line. To create a code chunk, click *Insert* -> `R`. Within this code chunk, type in `library(tidyverse)` and run the code by either

1. Clicking the "Run" button in the menu bar of the top-left window of `R Studio` or

2. (Recommended) Clicking "Command + Enter" on a Mac or "Control + Enter" on a PC.

Note that all code appears in grey boxes surrounded by three backticks while normal text has a different colour background with no backticks.

```
library(tidyverse)
```

When you run the previous line, some text will appear in the bottom-left window. We won't worry too much about what this text means now, but we also won't ignore it completely. You should be able to spot the 8 core `tidyverse` packages listed above as well as some numbers that follow each package. The numbers correspond to the package version. There's some other things too, but as long as this text does not start with "Error:", you're good to go!

Congrats on running your first line of code for this class! This particular code isn't particularly exciting because it doesn't really do anything that we can see.

We have run `R` code using an `R` chunk. In your `R` chunk, on a new line, try typing in a basic calculation, like `71 + 9` or `4 / 3`, them run the line and observe the result.

So, that still wasn't super exciting. `R` can perform basic calculations, but you could just use a calculator or Excel for that. In order to look at things that are a bit more interesting, we need some data.

2.4 Alcohol Data Example

We will be looking at two data sets just to get a little bit of a preview of things we will be working on for the rest of the semester. **Important**: Do not worry about understanding what the following code is doing at this point. There will be plenty of time to understand this in the weeks ahead. The purpose of this section is just to get used to using `R`: there will be more detailed explanations and exercises about the functions used and various options in the coming weeks. In particular, the following code uses the `ggplot2`, `dplyr`, and `tidyr` packages, which we will cover in detail throughout the first ~ 3-4 weeks of this course.

Data for this first part was obtained from fivethirtyeight at Five Thirty Eight GitHub page.

The first step is to read the data set into `R`. Though you have already downloaded alcohol.csv in the data zip, we still need to load it into `R`. Check to make sure the alcohol.csv is in the data folder in your bottom-right hand window. The following code can be copied to an `R` code chunk to read in the data:

```
read_csv("data/alcohol.csv")
```

Note that we do not need the full file extension **if** we have the data set in an `R` project.

Did something show up in your console window? If so, great! If not, make sure that the data set is in the data folder and that you have an `R` project set up.

We would like to name our data set something so that we could easily reference it later, so name your data set using the `<-` operator, as in

```
alcohol_data <- read_csv("data/alcohol.csv")
```

You can name your data set whatever you want to (with a few restrictions). I've named it `alcohol_data`. Now, if you run the line of code above where you name the data set, and run `alcohol_data`, you should see the data set appear:

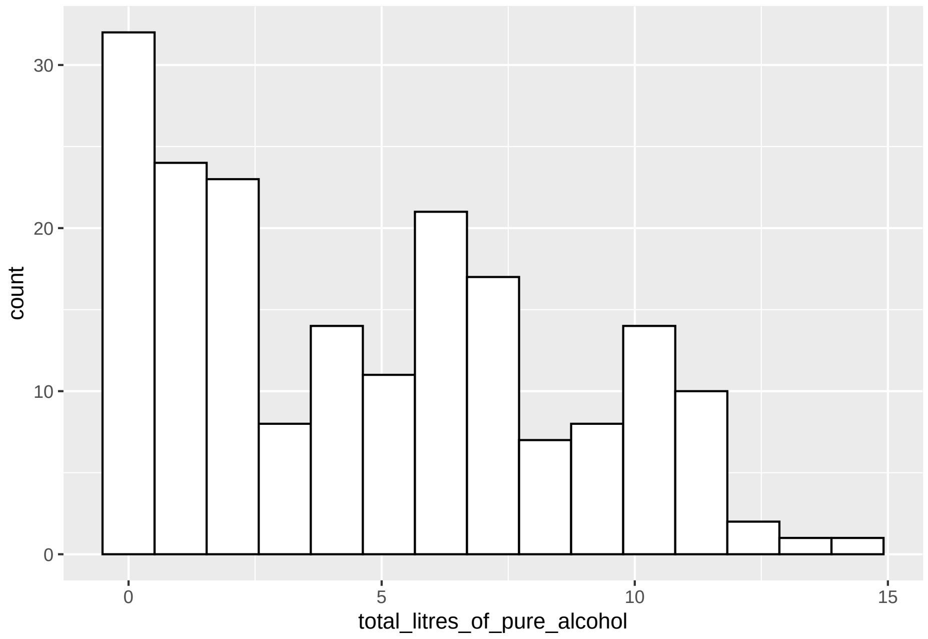
```
alcohol_data
#> # A tibble: 193 x 5
#>    country       beer_servings spirit_servings wine_servings
#>    <chr>                 <dbl>           <dbl>         <dbl>
#>  1 Afghanistan               0               0             0
#>  2 Albania               89           132           54
#>  3 Algeria               25             0           14
#>  4 Andorra              245           138          312
#>  5 Angola               217            57           45
#>  6 Antigua & Ba~        102           128           45
#>  7 Argentina            193            25          221
#>  8 Armenia               21           179           11
#>  9 Australia            261            72          212
#> 10 Austria              279            75          191
#> # ... with 183 more rows, and 1 more variable:
#> #   total_litres_of_pure_alcohol <dbl>
```

What's in this data set? We see a few *variables* on the columns:

- `country`: the name of the country
- `beer_servings`: the average number of beer servings per person per year
- `spirit_servings`: the average number of spirit (hard alcohol) servings per person per year
- `wine_servings`: the average number of wine servings per person per year
- `total_litres_of_pure_alcohol`: the average total litres of pure alcohol consumed per person per year.

One goal of this class is for you to be able to pose questions about a data set and then use the tools we will learn to answer those questions. For example, we might want to know what the distribution of total litres of alcohol consumed per person looks like across countries. To do this, we can make a plot with the `ggplot2` package, one of the packages that automatically loads with `tidyverse`. We might start by constructing the following plot. **Reminder**: the goal of this is not for everyone to understand the code in this plot, so don't worry too much about that.

```
ggplot(data = alcohol_data,
       mapping = aes(total_litres_of_pure_alcohol)) +
  geom_histogram(colour = "black", fill = "white", bins = 15)
```



I now want to see where the United States (`USA`) falls on this distribution by drawing a red vertical line for the total litres of alcohol consumed in the United States. To do so, I'll first use the `filter()` function in the `dplyr` package (again, we will learn about that function in detail later). Copy and paste the following lines of code into a new `R` chunk. Then, run the lines.

```r
small_df <- alcohol_data %>% filter(country == "USA")
ggplot(data = alcohol_data,
       mapping = aes(total_litres_of_pure_alcohol)) +
  geom_histogram(colour = "black", fill = "white", bins = 15) +
  geom_vline(data = small_df,
             aes(xintercept = total_litres_of_pure_alcohol),
             colour = "red")
```

It looks like there are some countries that consume little to no alcohol. We might want to know what these countries are:

```r
alcohol_data %>% filter(total_litres_of_pure_alcohol == 0)
#> # A tibble: 13 x 5
#>    country       beer_servings spirit_servings wine_servings
#>    <chr>                 <dbl>           <dbl>         <dbl>
#>  1 Afghanistan               0               0             0
#>  2 Bangladesh                0               0             0
#>  3 North Korea               0               0             0
#>  4 Iran                      0               0             0
#>  5 Kuwait                    0               0             0
#>  6 Libya                       0             0             0
#>  7 Maldives                    0             0             0
#>  8 Marshall Isl~               0             0             0
#>  9 Mauritania                  0             0             0
#> 10 Monaco                      0             0             0
#> 11 Pakistan                    0             0             0
#> 12 San Marino                  0             0             0
#> 13 Somalia                     0             0             0
#> # ... with 1 more variable:
#> #   total_litres_of_pure_alcohol <dbl>
```

It looks like there are 13 countries in the data set that consume no alcohol. Note that, in the chunk above, we have to use in `total_litres_of_pure_alcohol` as the variable name because this is the name of the variable in the data set. Even something like spelling litres in the American English liters (`total_liters_of_pure_alcohol`) would throw an error because this isn't the exact name of the variable in the data set. This is something that can be very aggravating when you are first learning any coding language.

Now suppose that we want to know the 3 countries that consume the most beer, the 3 countries that consume the most spirits, and the 3 countries that consume the most wine per person. If you're a trivia person, you can form some guesses. Without cheating, I am going to guess (Germany, USA, and UK) for beer, (Spain, Italy, and USA) for wine, and (Russia, Poland, and Lithuania) for spirits. Let's do beer first!

```
alcohol_data %>% mutate(rankbeer = rank(desc(beer_servings))) %>%
  arrange(rankbeer) %>%
  filter(rankbeer <= 3)
```

Let's do the same thing for Wine and Spirits:

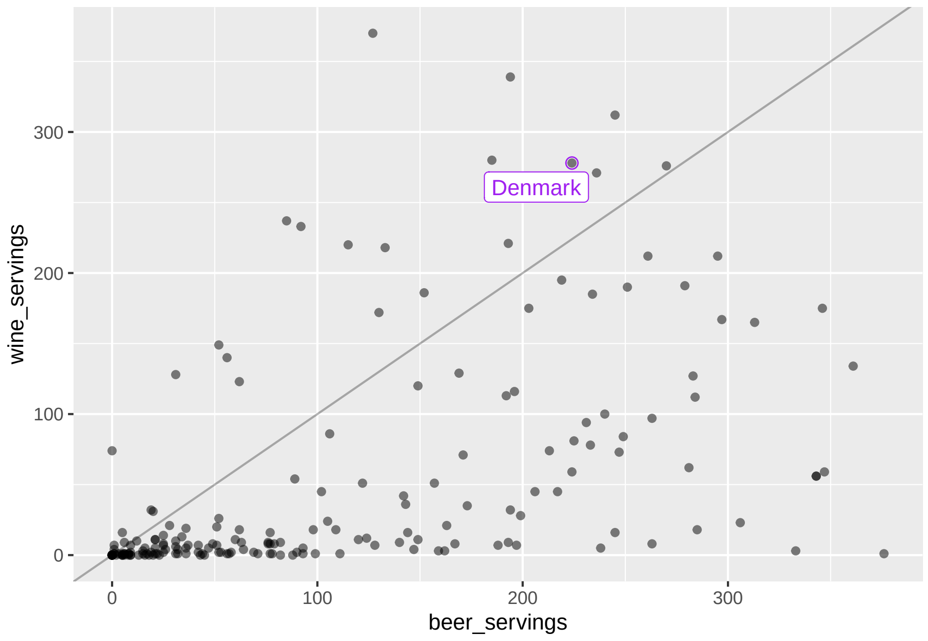
```
alcohol_data %>% mutate(rankwine = rank(desc(wine_servings))) %>%
  arrange(rankwine) %>%
  filter(rankwine <= 3)

alcohol_data %>% mutate(rankspirits = rank(desc(spirit_servings))) %>%
  arrange(rankspirits) %>%
  filter(rankspirits <= 3)
```

Finally, suppose that I want to know which country consumes the most wine relative to their beer consumption? Let's first look at this question graphically. I need to tidy the data first with the `pivot_longer()` function from the `tidyr` package:

```
onecountry_df <- alcohol_data %>%
  filter(country == "Denmark")
```

```
library(ggrepel)
ggplot(data = alcohol_data,
       mapping = aes(x = beer_servings, y = wine_servings)) +
  geom_point(alpha = 0.5) +
  geom_label_repel(data = onecountry_df, aes(label = country),
    colour = "purple") +
  geom_point(data = onecountry_df, colour = "purple",
             size = 2.5, shape = 1) +
  geom_abline(aes(slope = 1, intercept = 0), alpha = 0.3)
```



The x-axis corresponds to beer servings while the y-axis corresponds to wine servings. A reference line is given so with countries above the line consuming more wine than beer. We will get into how to make a plot like this later: for now, copy the code chunk and change the labeled point so that it corresponds to a country that interests you (other than Denmark). We might be able to better answer the original question numerically by computing the wine to beer ratio for each country and then ordering from the largest ratio to the smallest ratio:

```
alcohol_data %>%
  mutate(wbratio = wine_servings / beer_servings) %>%
  arrange(desc(wbratio)) %>%
  select(country, beer_servings, wine_servings, wbratio)
#> # A tibble: 193 x 4
#>    country             beer_servings wine_servings wbratio
#>    <chr>                           <dbl>         <dbl>   <dbl>
#>  1 Cook Islands                        0            74   Inf
#>  2 Qatar                               1             7     7
#>  3 Montenegro                         31           128     4.13
#>  4 Timor-Leste                         1             4     4
#>  5 Syria                               5            16     3.2
#>  6 France                            127           370     2.91
#>  7 Georgia                            52           149     2.87
#>  8 Italy                              85           237     2.79
#>  9 Equatorial Guinea                  92           233     2.53
#> 10 Sao Tome & Principe                56           140     2.5
#> # ... with 183 more rows
```

Why is one of the ratios `Inf`?

2.4.1 Exercises

1. What is the shape of the distribution of total alcohol consumption? Left-skewed, right-skewed, or approximately symmetric? Unimodal or multi-modal?

2. In the histogram of total alcohol consumption, pick a country other than the USA that interests you. See if you can change the code in the chunk that made the histogram so that the red vertical line is drawn for the country that interests you.

Hint: Use the `View()` function to look at the alcohol data set by typing `View(alcohol_data)` in your bottom-left window to help you see which countries are in the data set.

```
View(alcohol_data)
```

Note: careful about capitalization: `R` is case sensitive so USA is different than usa.

3. In the histogram of total alcohol consumption, change the **fill** colour of the bins in the histogram above: what should be changed in the code chunk?

4. In the rankings code, what if you wanted to look at the top 5 countries instead of the top 3? See if you could change the code.

5. In the spirit rankings, why do you think only 2 countries showed up instead of 3? Can you do any investigation as to why this is the case?

6. Change the wine to beer ratio code example to find the countries with the highest beer to wine consumption (instead of wine to beer consumption).

2.5 Athlete Data Example

Secondly, we will look at a data set on the top 100 highest paid athletes in 2014. The `athletesdata` was obtained from https://github.com/ali-ce/datasets data set has information on the following variables from the 100 highest paid athletes of 2014, according to Forbes (pay includes **both** salary and endorsements):

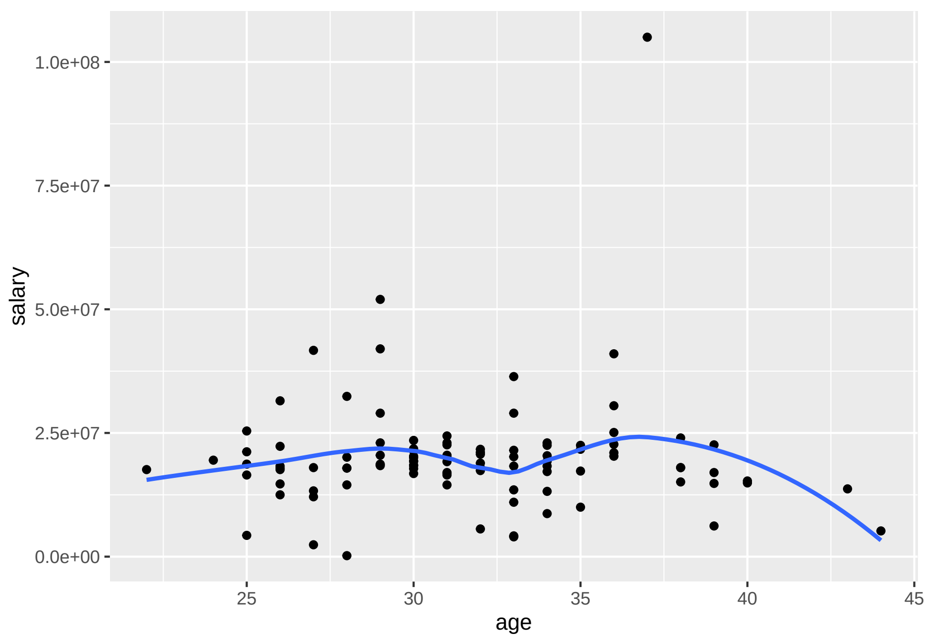
- `Name` (name of the athlete)
- `Rank` (where the athlete ranks, with 1 being the highest paid)
- `Sport` (the sport the athlete plays)
- `endorsements` (money from sponsorships from companies)
- `totalpay` (in millions in the year of 2014, salary + endorsements)
- `salary` (money from tournaments or contract salary)
- `age` of athlete in 2014
- `Gender` (Male or Female)

We will first read in the data set below and name it `athletes`. We can then use the `head()` function to look at the first few rows of the data set.

```
athletes <- read_csv("data/athletesdata.csv")
head(athletes)
#> # A tibble: 6 x 9
#>      X1 Name   Rank Sport endorsements totalpay salary   age
#>   <dbl> <chr> <dbl> <chr>        <dbl>    <dbl>  <dbl> <dbl>
#> 1     1 Aaro~    55 Foot~      7500000 22000000 1.45e7    31
#> 2     2 Adam~    95 Golf       9000000 17700000 8.7 e6    34
#> 3     3 Adri~    60 Base~       400000 21500000 2.11e7    32
#> 4     4 Alex~    48 Base~       300000 22900000 2.26e7    39
#> 5     5 Alfo~    93 Base~        50000 18050000 1.8 e7    38
#> 6     6 Amar~    27 Bask~      5000000 26700000 2.17e7    32
#> # ... with 1 more variable: Gender <chr>
```

There are many different interesting questions to answer with this data set. First, we might be interested in the relationship between athlete age and salary for the top 100 athletes. Recall from an earlier stat course that one appropriate graphic to examine this relationship is a scatterplot:

```
ggplot(data = athletes, mapping = aes(x = age, y = salary)) +
  geom_point() +
  geom_smooth(se = FALSE)
```



Do you see anything strange with the scatterplot? What do you think the y-axis tick labels of 2.5e+07, 5.0e+07, etc. mean?

Now let's see if we can count the number of athletes in the Top 100 that are in my personal favourite sport, Tennis:

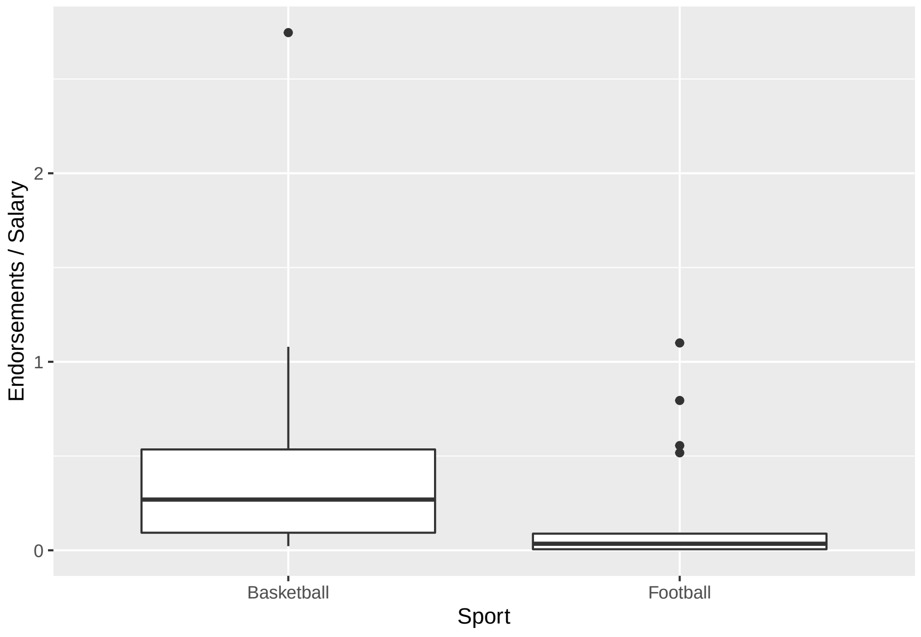
```
athletes %>% group_by(Sport) %>%
  summarise(counts = n()) %>%
  filter(Sport == "Tennis")
#> # A tibble: 1 x 2
#>   Sport  counts
#>   <chr>   <int>
#> 1 Tennis      6
```

It looks like there are 6 athletes: we can see who they are and sort them by their `Rank` with:

```
athletes %>%
  filter(Sport == "Tennis") %>%
  arrange(Rank)
#> # A tibble: 6 x 9
#>      X1 Name   Rank Sport endorsements totalpay salary   age
#>   <dbl> <chr> <dbl> <chr>        <dbl>    <dbl>  <dbl> <dbl>
#> 1    82 Roge~     7 Tenn~     52000000 56200000 4.2 e6    33
#> 2    78 Rafa~     9 Tenn~     30000000 44500000 1.45e7    28
#> 3    72 Nova~    17 Tenn~     21000000 33100000 1.21e7    27
#> 4    64 Mari~    34 Tenn~     22000000 24400000 2.4 e6    27
#> 5    60 Li Na     41 Tenn~      18000000 23600000 5.6 e6    32
#> 6    89 Sere~     55 Tenn~      11000000 22000000 1.1 e7    33
#> # ... with 1 more variable: Gender <chr>
```

Finally, let's see if we can compare the ratio of endorsements (from commercials and products) to salary of professional athletes in the Top 100 in 2 sports: Football (referring to American Football) and Basketball. Recall from an earlier Stat class that we might want to use side-by-side boxplots to make this comparison since we have one categorical variable (Sport Type) and one quantitative variable (Ratio of Endorsements to Salary).

```
athletes %>% filter(Sport == "Football" | Sport == "Basketball") %>%
  ggplot(data = ., aes(x = Sport, y = endorsements / salary)) +
  geom_boxplot() +
  labs(y = "Endorsements / Salary")
```



In the graph an endorsements / salary ratio of 1 indicates that the person makes half of their overall pay from endorsements and half of their overall pay from salary.

Which sport looks like it tends to receive a larger proportion of their overall pay from endorsements for athletes in the top 100?

2.5.1 Exercises

1. Instead of looking at the relationship between age and salary in the top 100 athletes of 2014, change the plot to look at the relationship between age and endorsements. What would you change in the code above? Try it!

2. Pick a `Sport` other than Tennis and see if you can count the number of athletes in the top 100 in that sport as well as sort them by Rank. **Careful**: not all sports will have athletes in the Top 100.

How many athletes are in the top 100 in the sport that you chose?

3. In the endorsements / salary example, change one of the sports to the sport of your choice and make a comparison. Which sport tends to receive a larger proportion of their overall pay from endorsements.

4. What qualification might you want to make about your statement in the previous exercise? (Is this a random sample of athletes from each sport? Why does that matter?).

5. In the side-by-side boxplots comparing the endorsements to salary ratio of two different sports, I've changed the y-axis label above to be Endorsements / Salary using the `labs(y = "Endorsements / Salary")` statement. Try changing the x-axis label to something else. What do you think you would need to add to the plot?

2.6 Finishing Up: Common Errors in R

We will now talk a little bit about getting errors in `R` and what can be done to correct some common errors.

You may have encountered some errors by this point in the document. Let's go over a few common errors as well as discuss how to comment your code.

1. A missing parenthesis: any open parenthesis `(` **needs** to close `)`. Try running the following code chunk without fixing anything.

```
ggplot(data = athletes, aes(x = Sport, y = salary) +
  geom_boxplot()
```

Notice in your bottom-left window that the `>` symbol that starts a line changes to a `+`. This is generally bad!! It means that you forgot to close a parenthesis `)` or a quote (`'` or `"`). No code will run since `R` thinks you are still trying to type something into a function. To fix this issue, click your cursor into the bottom-left window and press Esc. Then, try to find the error in the code chunk.

- Can you find the missing closing parenthesis above?

2. Missing Comma. Try running the following code chunk without fixing anything.

```
ggplot(data = athletes aes(x = Sport, y = salary)) +
  geom_boxplot()
```

`R` gives you an "Error: unexpected symbol in ....". Oftentimes, this means that there is a missing comma or that you spelled a variable name incorrectly.

- Can you find the missed comma above?

3. Capitalization Issues

```
athletes %>% filter(sport == "Tennis")
```

In the original data set, the variable `Sport` is capitalized. Not capitalizing it means that `R` won't be able to find it and proclaims that "object sport not found".

4. Forgetting Quotes. Character strings need to have quotation marks around them. We will discuss more of this later, but graph labels and titles need to have quotes around them since they don't directly refer to columns or rows in our data set:

```
ggplot(data = athletes, aes(x = Sport, y = endorsements)) +
  geom_boxplot() + xlab(Popularity Measure)
```

The error for forgetting quotes is typically an "Unexpected Symbol" though this error is also given for other issues.

- Where are the quotes missing in the code chunk above?

Finally, you can add a comment to a code chunk with the `#` symbol (I always use double `##` for some reason though). This allows you to type a comment into a code chunk that **isn't** code:

```
## this is a comment
## this calculation might be useful later
7 * 42
#> [1] 294
```

Comments are most useful for longer code chunks, as they allow you to remember why you did something. They also tell someone whom you've shared your code with why you did something.

Save this file by clicking File -> Save or by using the keyboard shortcut Command + s (or Control + s on a PC). Knit this file by clicking the Knit button in the top-left window (with the knitting needles). You should see a .html file pop up, if there are no errors in your code!

2.7 Chapter Exercises

Note: Usually, exercises will ask you to write code on your own using the week's chapter as a reference. However, for this initial chapter, we will do something a little different.

Open a new .Rmd file (File -> New File -> `R Markdown` -> OK) and delete the text explaining what `R Markdown` is in lines 10 and below. Then, complete the following exercises.

Exercise 1. Read the very short paper at https://joss.theoj.org/papers/10.21105/joss.01686 on an Introduction to the `tidyverse`, and answer the questions below in your `R Markdown` file. I'm imagining this whole exercise should only take you ~ 20-25 minutes.

Answer the following questions by typing answers in your .Rmd document. You should not need to make any new code chunks, as the questions don't ask you to do any coding!

1. What are the two major areas that the `tidyverse` **doesn't** provide tools for?
2. How do the authors define "tidy"?
3. What does it mean for the `tidyverse` to be "human-centred"?
4. In about 2 sentences, describe the data science "cycle" given in the diagram at the top of page 3.

Exercise 2. You may continue to use the same .Rmd file to answer these questions. For each question, type your answer on a new line, with a line space between your answers. All of these questions should be answered outside of code chunks since your answers will all be text, not code.

a. What is your name and what is your class year (first-year, sophomore, junior, senior)?
b. What is/are your major(s) and minor(s), either actual or intended?
c. Why are you taking this course? (Major requirement?, Minor requirement?, recommended by advisor or student?, exploring the field?, etc.).
d. In what semester and year did you take STAT 113 and who was your professor?
e. Have you taken STAT 213? Have you taken CS 140?
f. What is your hometown: city, state, country?
g. Do you play a sport on campus? If so, what sport? If not, what is an activity that you do on or off-campus?
h. What is your favorite TV show or movie or band/musical artist?

i. Tell me something about yourself.
j. Take a look at the learning outcomes listed on the syllabus. Which are you most excited for and why?
k. What are your expectations for this class and/or what do you hope to gain from this class?

Knit your .Rmd file into an .html file and submit your knitted .html file to Sakai. If your file won't knit, then submit the .Rmd file instead. To submit either file, you first need to get the file off of the server and onto your computer so that you can upload it to Sakai. Use the following steps to do so:

1. Click the checkbox next to your knitted .html file.
2. Click the Gear Icon "More" -> Export
3. If you would like, rename your file to something like Week0_YOURLASTNAME.html, but, make sure to keep the correct extension (either .html or .Rmd).
4. After you export it, the file should appear in your downloads folder. Now, go to Sakai -> Assignments -> Week 0 Exercises and complete the upload process.

Nice work: we will dive into `ggplot()` in the `ggplot2` package next!

2.8 Exercise Solutions

In most sections, some exercise solutions will be posted at the end of the section. However, for the introduction, we will do all of the coding exercises as a class to make sure that we all start off well.

Chapter 3

Plotting with `ggplot2`

Goals:

1. Use the `ggplot2` package to make exploratory plots from STAT 113 of a single quantitative variable, two quantitative variables, a quantitative and a categorical variable, a single categorical variable, and two categorical variables.

2. Use the plots produced to answer questions about the Presidential election data set and the Fitness data set.

3. Further practice running code in `R`.

3.1 Introduction and Basic Terminology

We will begin our data science journey with plotting in the `ggplot2` package. We are starting with plotting for a couple of reasons:

1. Plotting is cool! We get to see an immediate result of our coding efforts in the form of a nice-to-look-at plot.

2. In an exploratory data analysis, you would typically start by making plots of your data.

3. Plotting can lead us to ask and subsequently investigate interesting questions, as we will see in our first example.

We will first use a data set on the 2000 United States Presidential election between former President George Bush and Al Gore obtained from http://www.econometrics.com/intro/votes.htm. For those unfamiliar with U.S. political elections, it is enough to know that each state is allocated a certain number of "electoral votes" for the president: states award all of their electoral votes to the candidate that receives the most ballots in that state. You can read more about this strange system on Wikipedia.

Florida is typically a highly-contentious "battleground" state. The data set that we have has the following variables, recorded for each of the 67 counties in Florida:

- `Gore`, the number of people who voted for Al Gore in 2000
- `Bush`, the number of people who voted for George Bush in 2000
- `Buchanan`, the number of people who voted for the third-party candidate Buchanan
- `Nader`, the number of people who voted for the third-party candidate Nader
- `Other`, the number of people who voted for a candidate other than the previous 4 listed
- `County`, the name of the county in Florida

To get started exploring the data, complete the following steps that you learned in Week 0:

1. Log-on to the SLU `R Studio` server http://rstudio.stlawu.local:8787

2. Create a new .Rmd file in the same folder as your Notes `R Project` using File -> New File -> `R Markdown`.

3. Finally, read in and name the data set `pres_df`, and take a look at the data set by running the `head(pres_df)` line, which shows the first few observations of the data set:

```
library(tidyverse)
pres_df <- read_table("data/PRES2000.txt")
## don't worry about the `read_table` function....yet
head(pres_df)
#> # A tibble: 6 x 6
#>     Gore   Bush Buchanan Nader Other County
#>    <dbl>  <dbl>    <dbl> <dbl> <dbl> <chr>
#> 1  47365  34124      263  3226   751 ALACHUA
#> 2   2392   5610       73    53    26 BAKER
#> 3  18850  38637      248   828   242 BAY
#> 4   3075   5414       65    84    35 BRADFORD
#> 5  97318 115185      570  4470   852 BREVARD
#> 6 386561 177323      788  7101  1623 BROWAR
```

Pay special attention to the variable names: we'll need to use these names when we make all of our plots. And, `R` is case-sensitive, meaning that we will, for example, need to use `Gore`, not `gore`.

We are trying to go very light on the technical code terminology to start out with (but we will come back to some things later in the semester). The terminology will make a lot more sense once you've actually worked with data. But, there are three terms that will be thrown around quite a bit in the next few weeks: *function*, *argument*, and *object*.

- a *function* in `R` is always* (*always for this class) followed by an open `(` and ended with a closed `)`. In non-technical terms, a *function* **does** something to its inputs and is often analogous to an English verb. For example, the `mean()` function calculates the mean, the `rank()` functions ranks a variable from lowest to highest, and the `labs()` is used to add labels to a plot. Every function has a help file that can be accessed by typing in `?name_of_function`. Try typing `?mean` in your lower left window.
- an *argument* is something that goes inside the parentheses in a function. Arguments could include *objects*, or they might not. In the bottom-left window, type `?mean` to view the Help file on this `R` function. We see that `mean()` has 3 arguments: `x`, which is an `R` object, `trim`, and `na.rm`. `trim = 0` is the default, which means that, by default, `R` will not trim any of the numbers when computing the mean.
- an *object* is something created in `R`, usually with `<-`. So, looking at the code above where we read in the data, `pres_df` is an `R` object.

All of this will make more sense as we go through these first couple of weeks.

3.2 Basic Plot Structure

We will use the `ggplot()` function in the `ggplot2` package to construct visualizations of data. the `ggplot()` function has 3 basic components:

- a `data` argument, specifying the name of your data set (`pres_df` above)
- a `mapping` argument, specifying that specifies the aesthetics of your plot (`aes()`). Common aesthetics are `x` position, `y` position, `colour`, `size`, `shape`, `group`, and `fill`.
- a `geom_    ()` component, specifying the geometric shape used to display the data.

The components are combined in the following form:

```
ggplot(data = name_of_data, mapping = aes(x = name_of_x_var,
                                          y = name_of_y_var,
                                          colour = name_of_colour_var,
                                          etc.)) +
  geom_nameofgeom() +
  .....<other stuff>
```

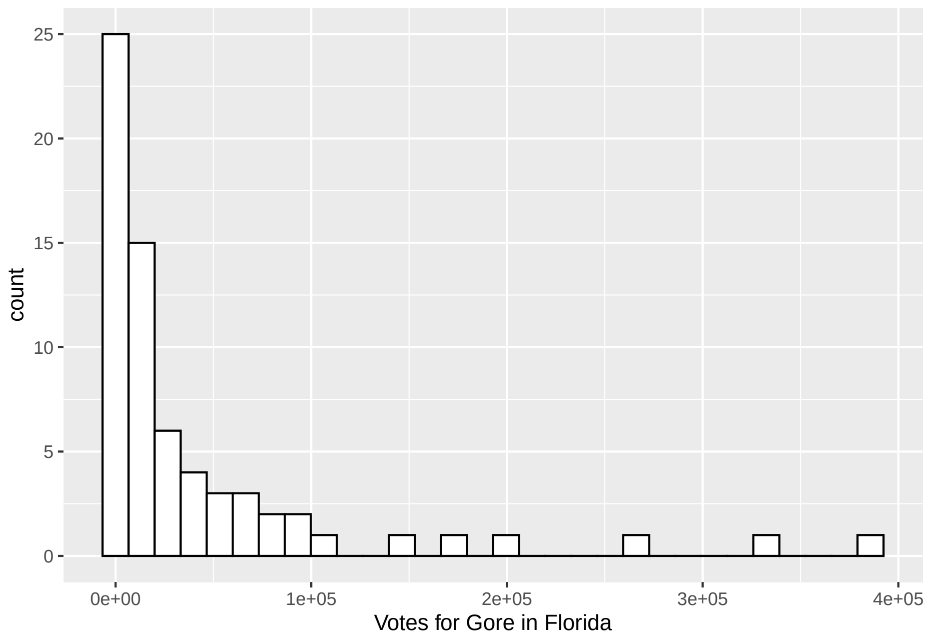
The structure of `ggplot()` plots is based on the Grammar of Graphics https://www.springer.com/gp/book/9780387245447. As with most new things, the components above will be easier to think about with some examples.

3.3 Graphing a Single Variable

3.3.1 Histograms and Frequency Plots for a Quantitative Variable

Let's go ahead and begin our exploration of the data by making a histogram of the number of people who voted for `Gore` in each county. Recall that a histogram is useful if we would like a graph of a single quantitative variable. Copy the following code to an `R` chunk and run the code:

```
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_histogram(colour = "black", fill = "white") +
  xlab("Votes for Gore in Florida")
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```



What do the 1e+05, 2e+05, etc. labels on the x-axis mean?

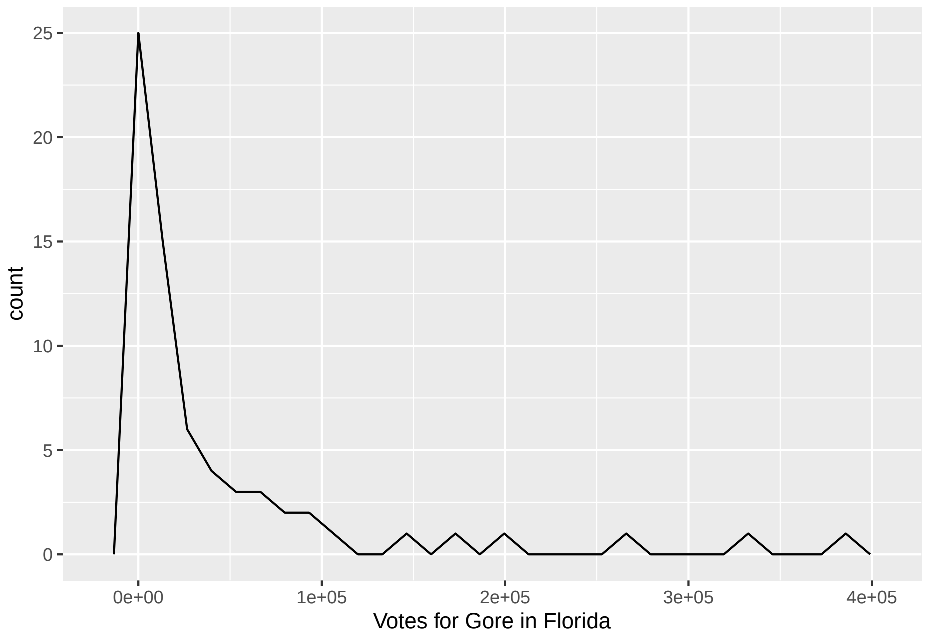
`R` gives us a message to "Pick a better value with `binwidth`" instead of the default `bins = 30`. Add `, bins = 15` inside the parentheses of `geom_histogram()` to change the number of bins in the histogram.

Change the colour of the inside of the bins to "darkred". Do you think that the colour of the inside of the bins maps to `colour` or `fill`? Try both!

There are a couple of observations with very high vote values. What could explain these large outliers?

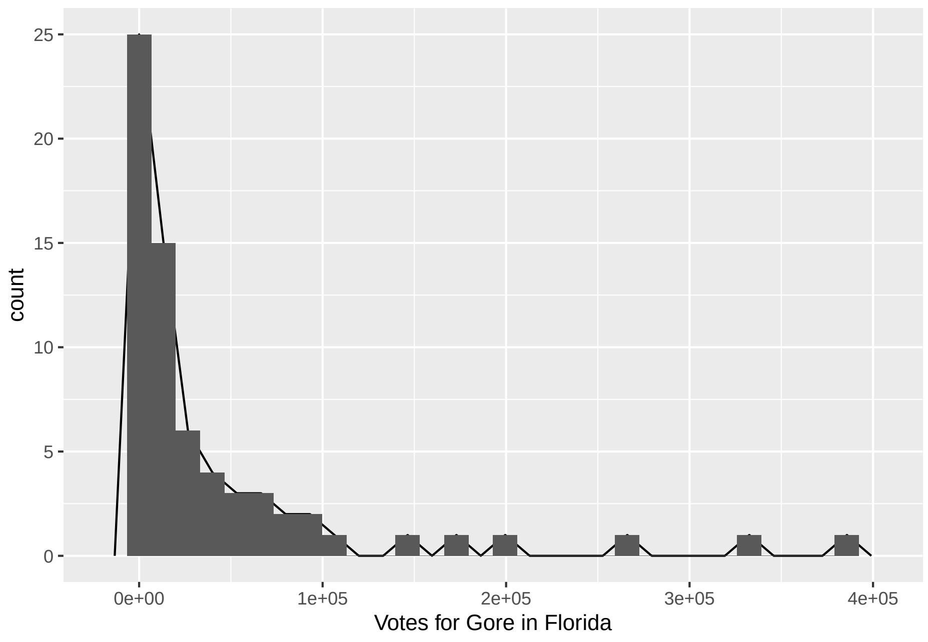
Another graph useful in visualizing a single quantitative variable is a frequency plot. The code to make a frequency plot is given below. We are simply replacing `geom_histogram()` with `geom_freqpoly()`.

```
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_freqpoly(colour = "black") +
  xlab("Votes for Gore in Florida")
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```



The frequency plot is just like a histogram but the counts are connected by a line instead of represented with bins. You can see how they relate by including **both** a geom_freqpoly() and a geom_histogram() in your plot, though it doesn't make for the prettiest graph:

```
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_freqpoly(colour = "black") +
  xlab("Votes for Gore in Florida") +
  geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```



3.3.2 R Code Style

We want our code to be as readable as possible. This not only benefits other people who may read your code (like me), but it also benefits you, particularly if you read your own code in the future. I try to follow the Style Guide in the Advanced R book: http://adv-r.had.co.nz/Style.html. Feel free to skim through that, but you don't need to worry about it too much: you should be able to pick up on some important elements just from going through this course. You might actually end up having better code style if you *haven't* had any previous coding experience.

As a quick example of why code style can be important, consider the following two code chunks, both of which produce the same graph.

```
ggplot(data=pres_df,mapping=aes(x=Gore))+geom_histogram(colour="black",fill="white")+
  xlab("Votes for Gore in Florida")
```

```
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_histogram(colour = "black", fill = "white") +
  xlab("Votes for Gore in Florida")
```

Which code chunk would you want to read two years from now? Which code chunk would you want your classmate/friend/coworker to read? (assuming you like your classmate/friend/coworker....)

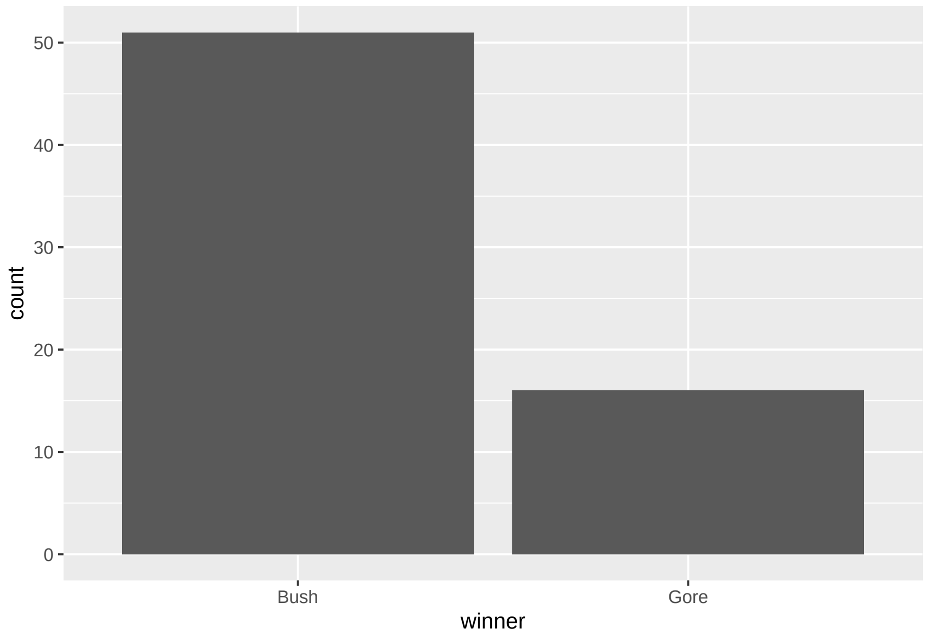
3.3.3 Bar Plots for a Categorical Variable

Recall from STAT 113 that bar plots are useful if you want to examine the distribution of one categorical variable. Side-by-side bar plots or stacked bar plots are plots that are useful for looking at the relationship between two categorical variables. There actually aren't any categorical variables that would be interesting to plot in this data set, so we'll make one, called `winner` using code that we don't need to understand until next week. `winner` will be `"Gore"` if Gore won the county and `"Bush"` if Bush won the county. We'll name this new data set `pres_cat`.

```
pres_cat <- pres_df %>% mutate(winner = if_else(Gore > Bush,
                                               true = "Gore",
                                               false = "Bush"))
pres_cat
#> # A tibble: 67 x 7
#>      Gore   Bush Buchanan Nader Other County    winner
#>     <dbl>  <dbl>    <dbl> <dbl> <dbl> <chr>     <chr>
#>  1  47365  34124      263  3226   751 ALACHUA   Gore
#>  2   2392   5610       73    53    26 BAKER     Bush
#>  3  18850  38637      248   828   242 BAY       Bush
#>  4   3075   5414       65    84    35 BRADFORD  Bush
#>  5  97318 115185      570  4470   852 BREVARD   Bush
#>  6 386561 177323      788  7101  1623 BROWAR    Gore
#>  7   2155   2873       90    39    17 CALHOUN   Bush
#>  8  29645  35426      182  1462   181 CHARLOTTE Bush
#>  9  25525  29765      270  1379   261 CITRUS    Bush
#> 10  14632  41736      186   562   237 CLAY      Bush
#> # ... with 57 more rows
```

Using this data set, we can make a bar plot with `geom_bar()`. The beauty of `ggplot()` is that the code is super-similar to what we used for histograms and frequency plots!

```
ggplot(data = pres_cat, aes(x = winner)) +
  geom_bar()
```



Note that, sometimes, data are in format such that one column contains the *levels* of the categorical variable while another column contains the counts directly. For example, we can create such a data set using code that we will learn next week:

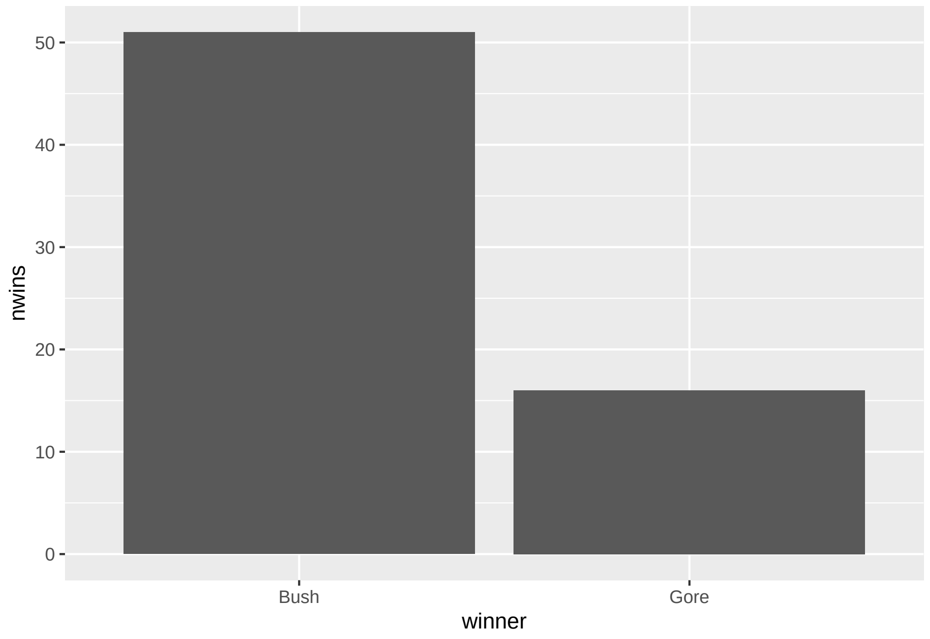
```
pres_cat2 <- pres_cat %>% group_by(winner) %>%
  summarise(nwins = n())
pres_cat2
#> # A tibble: 2 x 2
#>   winner nwins
#>   <chr>  <int>
#> 1 Bush      51
#> 2 Gore      16
```

This data set has just two observations and contains a column for the two major presidential candidates and a column for the number of counties that each candidate won. If we wanted to make a barplot showing the number of wins for each candidate, we can't use `geom_bar()`. Predict what the result will be from running the following code.

```
ggplot(pres_cat2, aes(x = winner)) +
  geom_bar()
```

Instead, we can use `geom_col()`, which takes an `x` aesthetic giving the column with names of the levels of our categorical variable, and a `y` aesthetic giving the column with the counts:

```r
ggplot(pres_cat2, aes(x = winner, y = nwins)) +
  geom_col()
```



3.3.4 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 3.7.

1. Change the frequency plot to plot the number of votes for Bush instead of the number for Gore. Are there any obvious outliers in the Bush frequency plot?

2. Do you have a preference for histograms or a preference for frequency plots? Can you think of a situation where one would be more desirable than the other?

3. It looks like Bush won a lot more....does that necessarily mean that Bush won more votes in total in Florida? Why or why not?

We will be using survey data from STAT 113 in the 2018-2019 academic year for many exercises in this section. For those who may not have taken STAT 113 from having AP credit or another reason, the STAT 113 survey is given to all students in STAT 113 across all sections. Some analyses in Intro Stat are then carried out using the survey.

```
library(tidyverse)
stat113_df <- read_csv("data/stat113.csv")
head(stat113_df)
#> # A tibble: 6 x 12
#>   Year  Sex     Hgt   Wgt Haircut   GPA Exercise Sport    TV
#>   <chr> <chr> <dbl> <dbl>   <dbl> <dbl>    <dbl> <chr> <dbl>
#> 1 Soph~ M        66   155       0  2.9        15 Yes       8
#> 2 Firs~ F        69   170      17  3.87       14 Yes      12
#> 3 Firs~ F        64   130      40  3.3         5 No        5
#> 4 Firs~ M        68   157      35  3.21       10 Yes      15
#> 5 Firs~ M        72   175      20  3.1         2 No        5
#> 6 Juni~ F        62   150      50  3.3         8 Yes       5
#> # ... with 3 more variables: Award <chr>, Pulse <dbl>,
#> #   SocialMedia <chr>
```

The data set contains the following variables:

- `Year`, FirstYear, Sophomore, Junior, or Senior
- `Sex`, M or F (for this data set, `Sex` is considered binary).
- `Hgt`, height, in inches.
- `Wgt`, weight, in pounds.
- `Haircut`, how much is paid for a haircut, typically.
- `GPA`
- `Exercise`, amount of hours of exercise in a typical week.
- `Sport`, whether or not the student plays a varsity sport.
- `TV`, amount of hours spent watching TV in a typical week.
- `Award`, Award preferred: choices are Olympic Medal, Nobel Prize, or Academy Award.
- `Pulse`, pulse rate, in beats per minute.
- `SocialMedia`, most used social media platform (Instagram, SnapChat, FaceBook, Twitter, Other, or None).

4. * Create a histogram of the `Exercise` variable, change the x-axis label to be "Exercise (hours per typical week)", change the number of `bins` to `14`, and change the `fill` of the bins to be "lightpink2" and the outline `colour` of the bins to be black.

5. * We can change the y-axis of a histogram to be "density" instead of a raw count. This means that each bar shows a **proportion** of cases instead of a raw count. Google something like "geom_histogram with density" to figure out how to create a y `aes()` to show density instead of count.

6. Construct a histogram using a quantitative variable of your choice. Change the `fill` and `colour` using http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf to help you choose colours.

7. Construct a bar plot for a variable of your choosing. What do you find?

8. What format would the STAT 113 data set need to be in to construct your bar plot with `geom_col()` instead of `geom_bar()`?

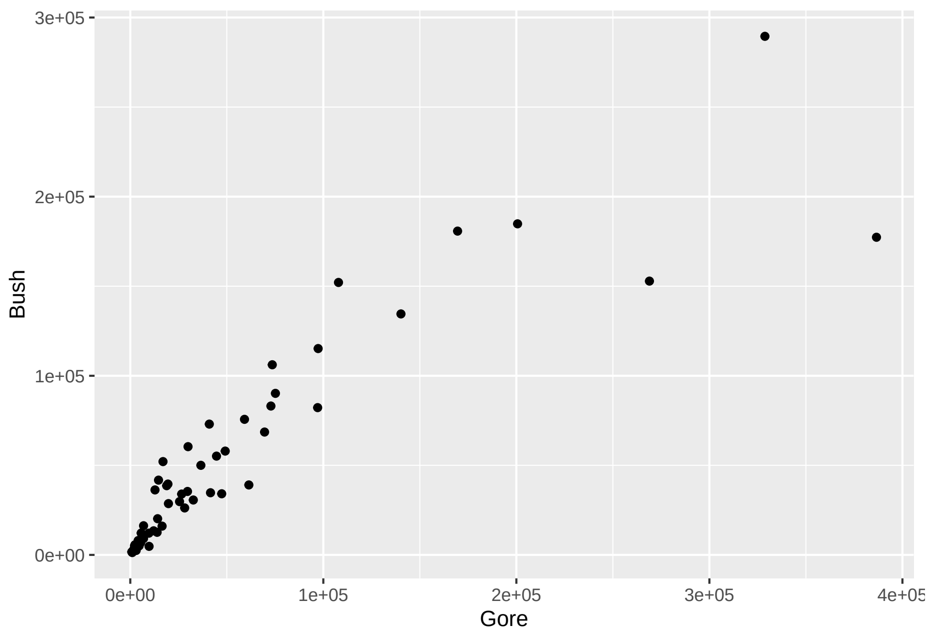
3.4 Graphing Two Quantitative Variables, Faceting, and `aes()` Options

3.4.1 Scatterplots

Moving back to the 2000 presidential election data set, thus far, we've figured out that there a couple of counties with very large numbers of votes for Gore and very large number of votes for Bush. We don't know the reason for this (if some counties are very democratic, very republican, or if some counties are just more populous). Do the counties that have a large number of votes for Bush also tend to have a large number of votes for Gore? And what about the other candidates: do they have any interesting patterns?

Let's start by making a scatterplot of the number of votes for Gore and the number of votes for Bush. Note that the `geom_` for making a scatterplot is called `geom_point()` because we are adding a layer of points to the plot.

```r
ggplot(data = pres_df, mapping = aes(x = Gore, y = Bush)) +
  geom_point()
```



What patterns do you see in the scatterplot?

Now, change the x variable from `Gore` to `Buchanan`. You should notice something strange in this scatterplot. Try to come up with one explanation for why the outlying point has so many votes for `Buchanan`.

In trying to come up with an explanation, it would be nice to figure out which Florida county has that outlying point and it would be nice if we knew something about Florida counties. To remedy the first issue, recall that we can type `View(pres_df)` to pull up the data set. Once you have the new window open, click on the column heading `Buchanan` to sort the votes for Buchanan from high to low to figure out which county is the outlier.

Use some Google sleuthing skills to find an explanation: try to search for "2000 united states presidential election [name of outlier county]". Write a sentence about what you find. Hint: if nothing useful pops up, try adding the term "butterfly ballot" to your search.

We have used the 2000 Presidential data set to find out something really interesting! In particular, we have used *exploratory data analysis* to examine a data set, without having a specific question of interest that we want to answer. This type of exploring is often really useful, but does have some drawbacks, which we will discuss later in the semester.

3.4.2 Aesthetics in `aes()`

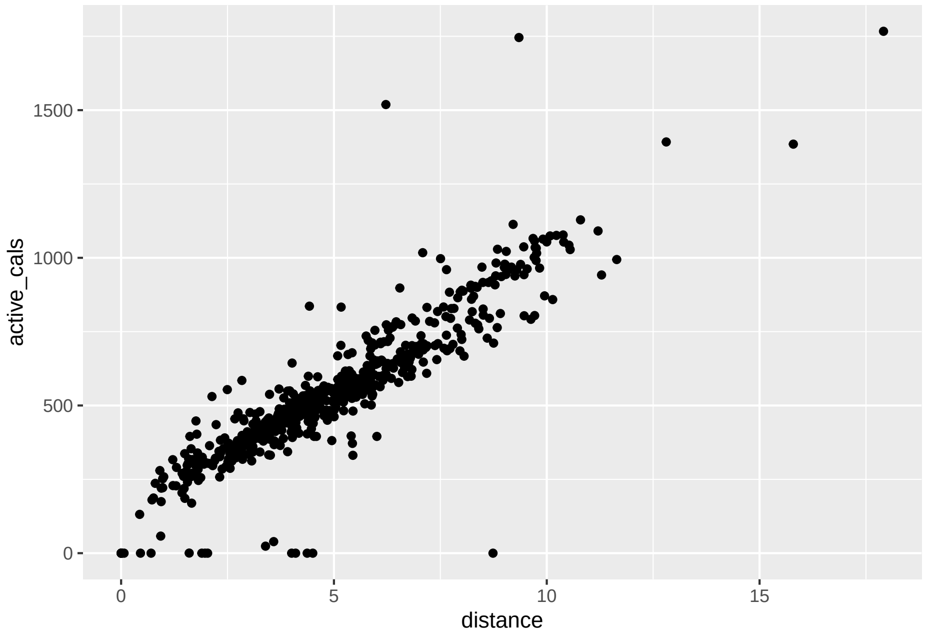
For the remainder of this chapter, we will work with some fitness data collected from my Apple Watch since November 2018. The `higham_fitness_clean.csv` contains information on the following variables:

- `Start`, the month, day, and year that the fitness data was recorded on
- `month`, the month
- `weekday`, the day of the week
- `dayofyear`, the day of the year (so that 304 corresponds to the 304th day of the year)
- `distance`, distance walked in miles
- `steps`, the number of steps taken
- `flights`, the number of flights of stairs climbed
- `active_cals`, the number of calories burned from activity
- `stepgoal`, whether or not I reached 10,000 steps for the day
- `weekend_ind`, a variable for whether or not the day of the week was a weekend day (Saturday or Sunday) or a weekday (Monday - Friday).

```
library(tidyverse)
fitness_full <- read_csv("data/higham_fitness_clean.csv") %>% mutate(weekend_ind = cas
  TRUE ~ "weekday"))
#>
#> -- Column specification ------------------------------------
#> cols(
#>   Start = col_date(format = ""),
#>   month = col_character(),
#>   weekday = col_character(),
#>   dayofyear = col_double(),
#>   distance = col_double(),
#>   steps = col_double(),
#>   flights = col_double(),
#>   active_cals = col_double(),
#>   stepgoal = col_character()
#> )
```

First, let's make a basic scatterplot to illustrate why it's so important to plot your data. I'll use the variable `distance` as the x-variable and `active_cals` as the y-variable.

```
ggplot(data = fitness_full, aes(x = distance, y = active_cals)) +
  geom_point()
```

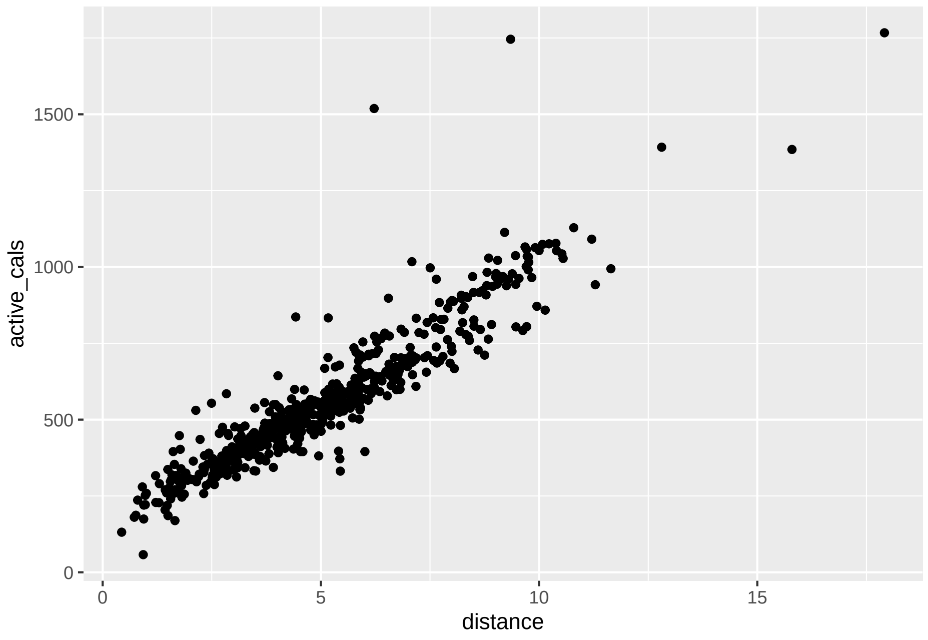


One aspect of the plot that you may notice is that there are observations where I burned 0 or very few active calories, yet walked/jogged/ran/moved some distance. Is it possible to not burn any calories and move ~ 4 miles? Probably not, so let's drop these observations from the data set and make a note of why we dropped those observations. Unfortunately, we don't have the tools to do this yet, so just run the following chunk of code without worrying too much about the syntax.

```
## drop observations that have active calories < 50.
## assuming that these are data errors or
## days where the Apple Watch wasn't worn.
fitness <- fitness_full %>%
  filter(active_cals > 50)
```

Let's make the plot again with the `fitness` data set instead of `fitness_full` to see if the outliers are actually gone. This time, we will put the `aes()` in the `geom_point()` function:

```
ggplot(data = fitness) +
  geom_point(aes(x = distance, y = active_cals))
```

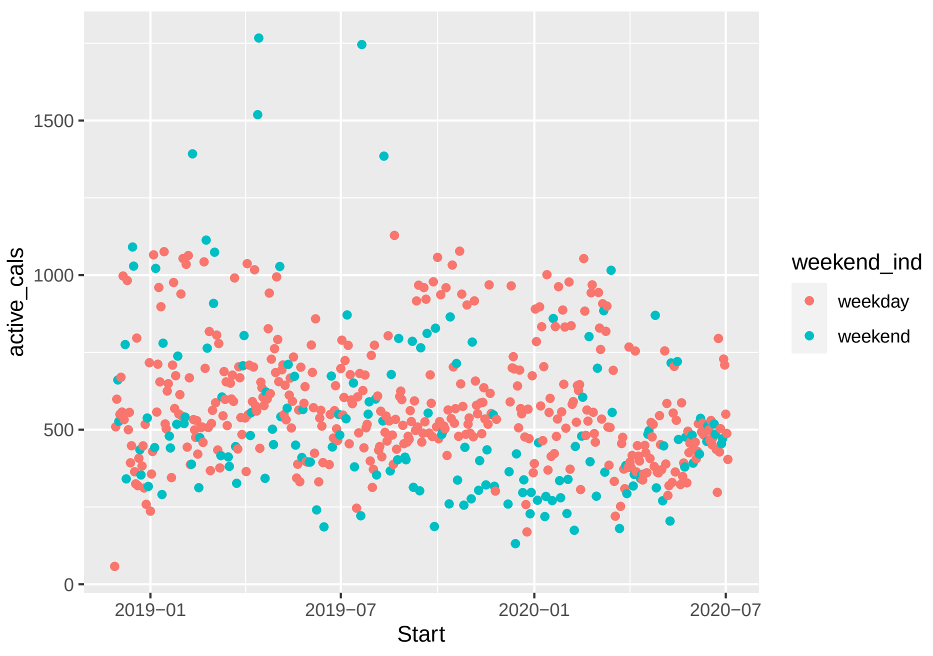


Putting the `aes()` in `ggplot()` and putting the `aes()` in `geom_point()` results in the same graph in this case. When you put the `aes()` in `ggplot()`, `R` perpetuates these `aes()` aesthetics in all `geom_`s in your plotting command. However, if you put your `aes()` in `geom_point()`, then any future `geom`s that you use will need you to re-specify different `aes()`. We'll see an example of this in the exercises.

Other `aes()` Options

In addition to `x` and `y`, we can also use `aes()` to map variables to things like `colour`, `size`, and `shape`. For example, we might make a scatterplot with `Start` on the x-axis (for the date) and `active_cals` on the y-axis, colouring by whether or not the day of the week was a weekend.

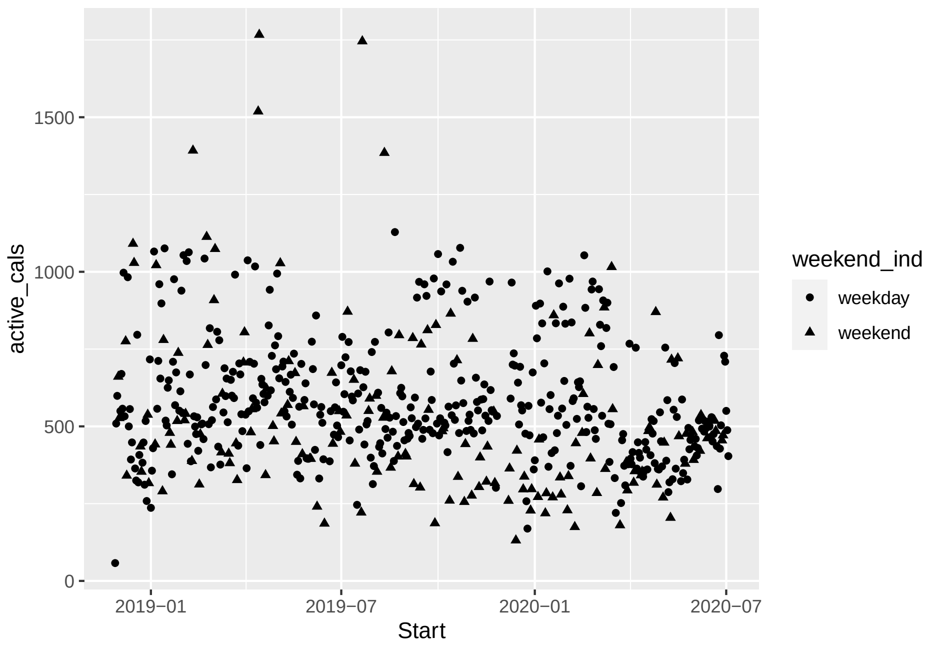
```
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, colour = weekend_ind))
```



Is there anything useful that you notice about the plot? Is there anything about the plot that could be improved?

Instead of using colour, you can also specify the point shape. This could be useful, for example, if you are printing something in black and white.

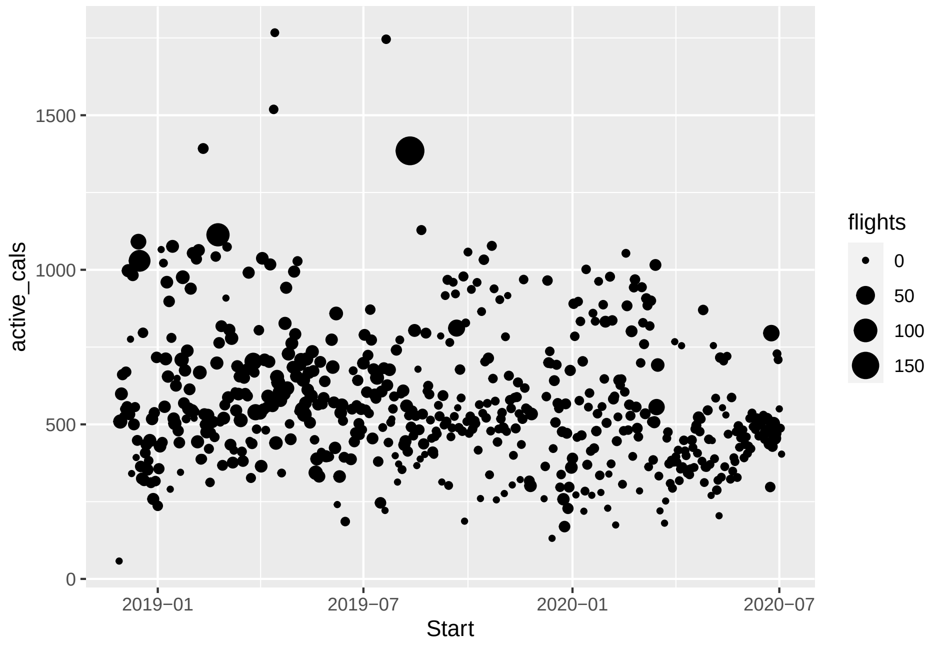
```
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, shape = weekend_ind))
```



Do you prefer the colour or the shape? Why?

Finally, another common `aes()` is `size`. For example, we could make the size of the points in the scatterplot change depending on how many `flights` of stairs I climbed.

```
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, size = flights))
```

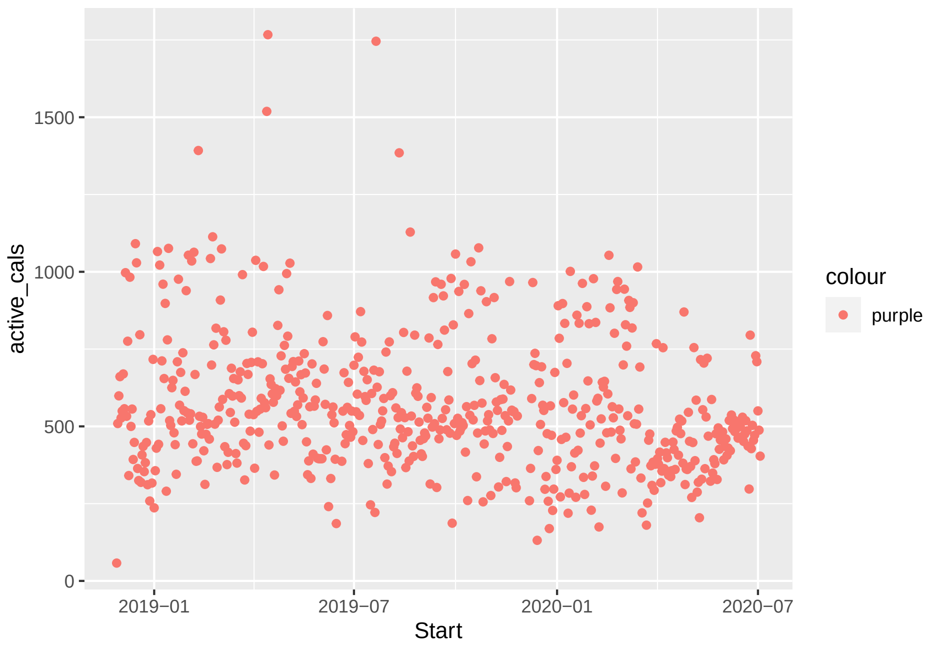


I don't think any of the previous three plots are necessarily the "best" and need some work, but, part of the fun of exploratory data analysis is making trying out different plots to see what "works."

Inside vs Outside `aes()`

We've changed the colour of the points to correspond to `weekend_ind`, but what if we just wanted to change the colour of points to all be the same colour, `"purple"`. Try running the following code chunk:

```r
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, colour = "purple"))
```



What does the graph look like? Did it do what you expected?

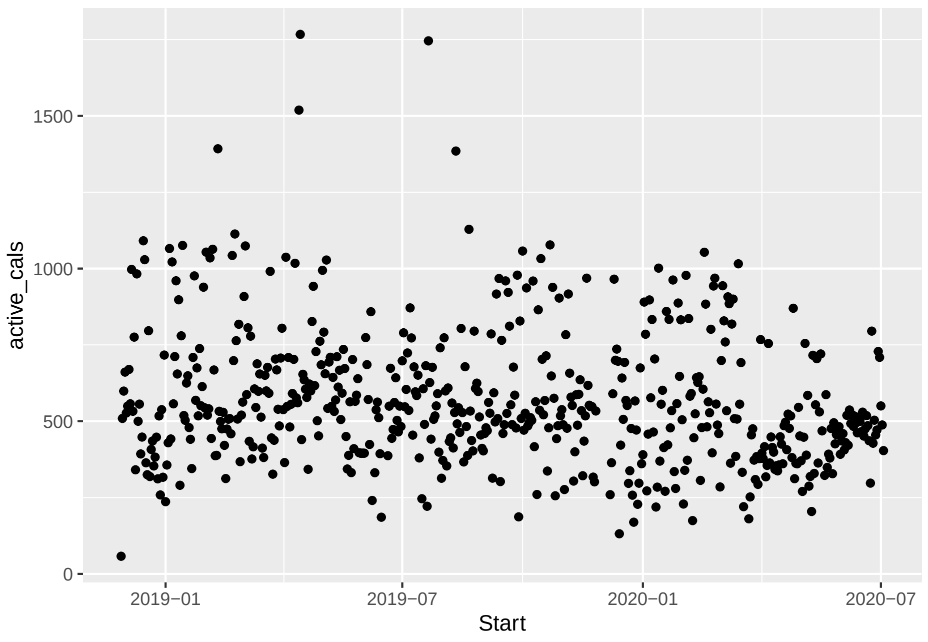
Putting `colour = ____` inside `aes()` or outside `aes()` achieves different things. In general,

- when we want to map something in our data set (`fitness`) to something in our plot (`x`, `y`, `colour`, `size`, etc.), we put that **inside** the `aes()` as in `geom_point(aes(colour = weekend_ind))`.

- When we assign fixed characteristics that don't come from the data, we put them **outside** the `aes()`, as in `geom_point(colour = "purple")`.

You can also change the overall point size and shape. The standard size is `1` so the following code chunk makes the points bigger. The standard shape is `19`: you can try changing that to other integers to see what other shapes you can get.

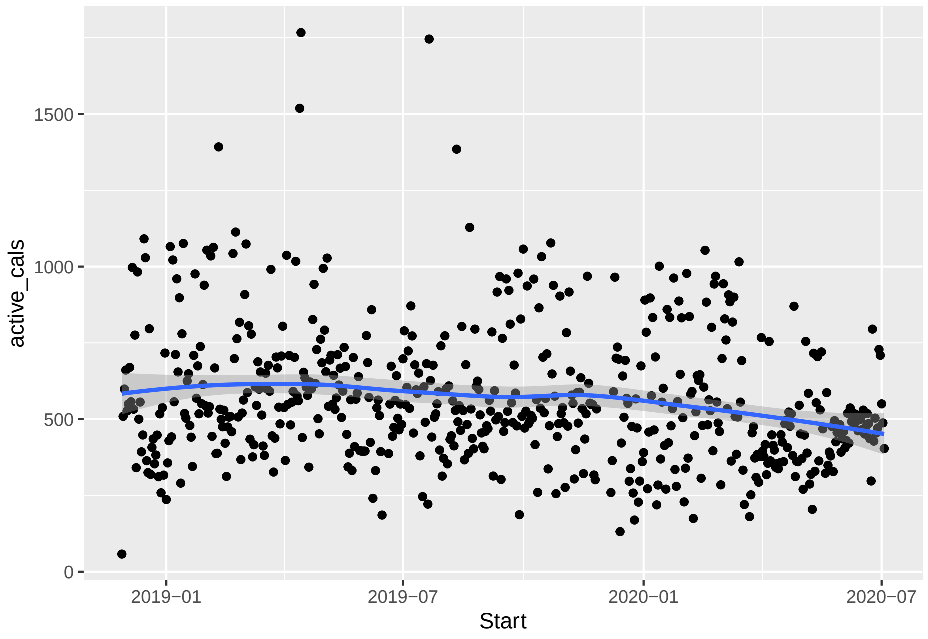
```r
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals), size = 1.5, shape = 19)
```



3.4.3 Using More Than One `geom()`

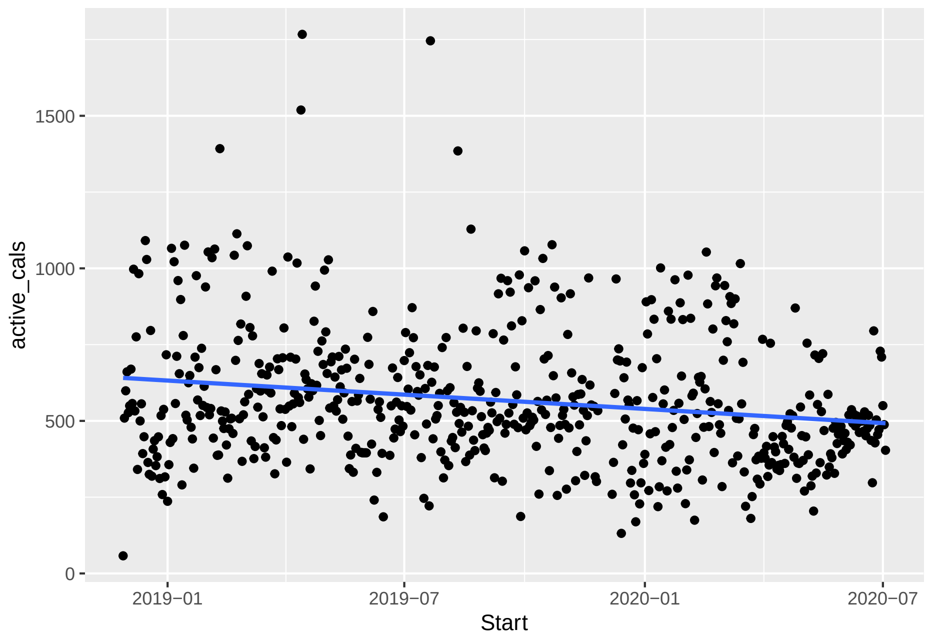
We might also be interested in fitting a smooth curve to our scatterplot. When we want to put more than one "geom" on our plot, we can use multiple `geoms`. Since I want the `aes()` to apply to **both** `geom_point()` and `geom_smooth()`, I am going to move the `aes()` command to the overall `ggplot()` line of code:

```
ggplot(data = fitness, aes(x = Start, y = active_cals)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Within `geom_smooth()`, you can set `se = FALSE` to get rid of the grey standard errors around each of the lines, and you can set`method = "lm"` to fit straight linear regression lines instead of smooth curves:

```
ggplot(data = fitness, aes(x = Start, y = active_cals)) +
  geom_point() +
  geom_smooth(se = FALSE, method = "lm")
#> `geom_smooth()` using formula 'y ~ x'
```

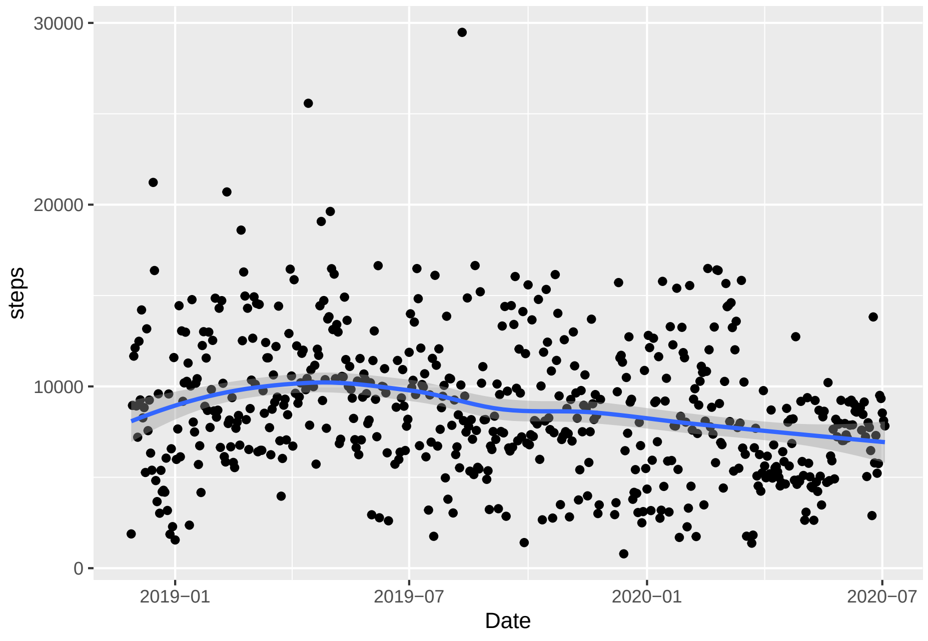


Does it look like there is an increasing overall trend? decreasing? Does it make sense to use a line to model the relationship or did you prefer the smooth curve?

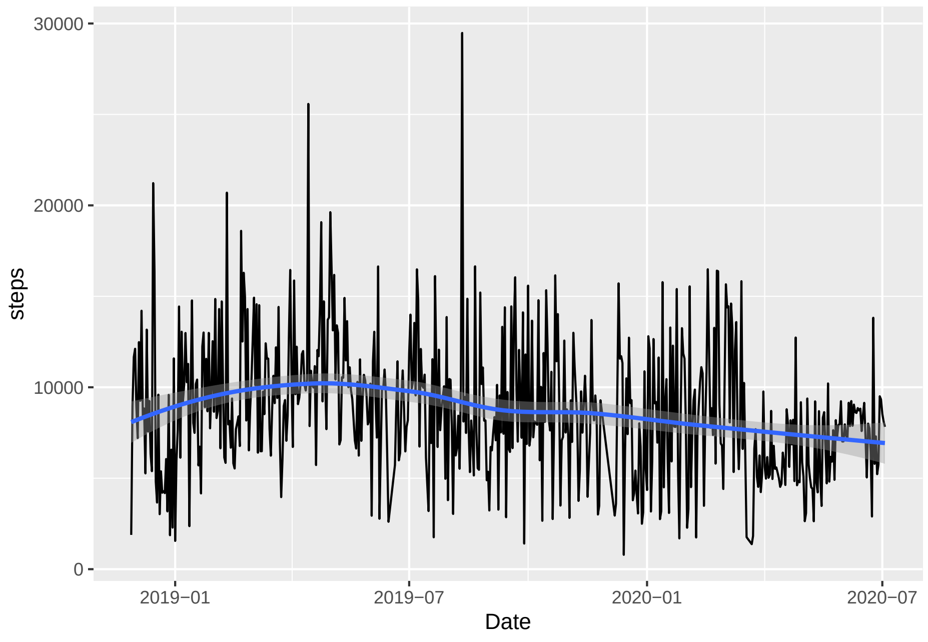
3.4.4 Line Plots with `geom_line()`

Line plots are often useful when you have a quantitative variable that you'd like to explore over time. The y-axis is the quantitative variable while the x-axis is typically time. More generally, line plots are often used when the x-axis variable has one discrete value for each y-axis variable. For example, suppose we want to explore how my step count has changed through time over the past couple of years. Compare the standard scatterplot with the following line plot: which do you prefer?

```
ggplot(data = fitness, mapping = aes(x = Start, y = steps)) +
  geom_point() + geom_smooth() + xlab("Date")
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



```
ggplot(data = fitness, mapping = aes(x = Start, y = steps)) +
  geom_line() + geom_smooth() + xlab("Date")
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

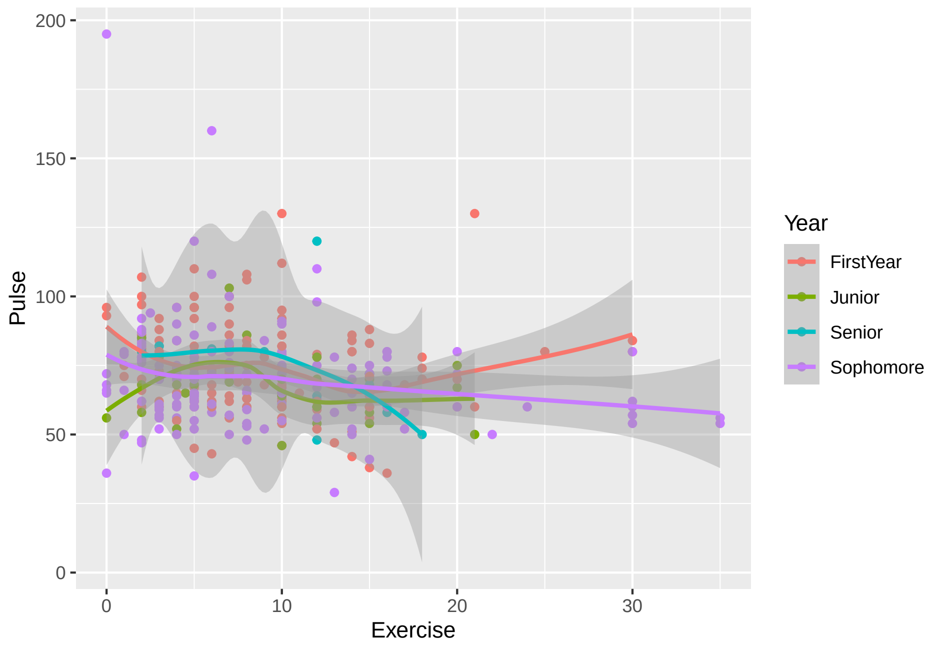


Can you spot the start of the pandemic in the graph? What seemed to happen with the step count?

3.4.5 Faceting

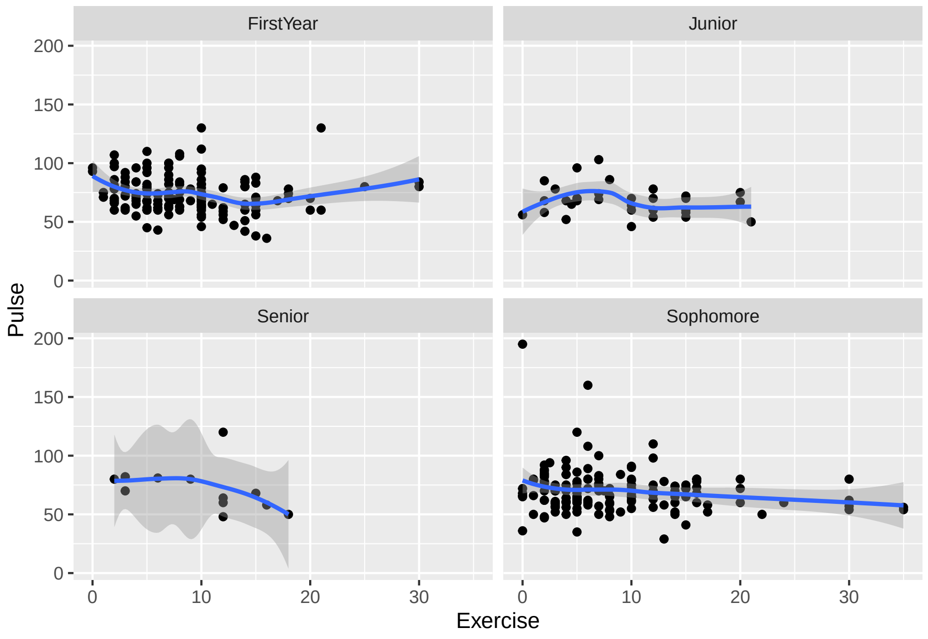
Using colour to colour points of different levels of a categorical variable is generally fine when there are just a couple of levels and/or there is little overlap among the levels. But, what if there are a lot more than two categories to colour by. For example, let's move back to the STAT 113 survey data set and investigate the relationship between `Pulse` and `Exercise` for different class `Year`'s. We might hypothesize that students who get more exercise tend to have lower pulse rates.

```
ggplot(data = stat113_df, aes(x = Exercise, y = Pulse,
                              colour = Year)) +
  geom_point() +
  geom_smooth(se = TRUE)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
#> Warning: Removed 40 rows containing non-finite values
#> (stat_smooth).
#> Warning: Removed 40 rows containing missing values
#> (geom_point).
```



When there are many different categories for a categorical variable (there are only 4 categories for `Year`, but this particular plot is still a bit difficult to read), it can sometimes be useful to `facet` the plot by that variable instead of trying to use different `colours` or `shapes`.

```
ggplot(data = stat113_df, aes(x = Exercise, y = Pulse)) +
  geom_point() +
  geom_smooth(se = TRUE) +
  facet_wrap(~ Year)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
#> Warning: Removed 40 rows containing non-finite values
#> (stat_smooth).
#> Warning: Removed 40 rows containing missing values
#> (geom_point).
```



We have eliminated the `colour =` argument and added `facet_wrap( ~ name_of_facet_variable)`. Doing so creates a different scatterplot and smooth line for each level of `name_of_facet_variable`.

What can you see from this plot that was harder to see from the plot with colour?

Does the data seem to support the hypothesis that more exercise is associated with lower pulse rates in this sample of students?

3.4.6 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 3.7.

1. Fix the code chunk where we tried to specify the colour of all points to be purple to actually make all of the points "purple" by moving `colour = "purple"` outside the parentheses in `aes()` (but still inside `geom_point()`).

2. In the console (bottom-left) window, type `?geom_smooth` and scroll down to "Arguments." Find `span`, read about it, and then, within the `geom_smooth()` argument of the line plot with steps vs. date, add a `span` argument to make the smooth line wigglier.

3. Explain why it doesn't make sense to construct a line plot of `Exercise` vs. `GPA`.

4. * Make a scatterplot of `Hgt` on the y-axis and `Wgt` on the x-axis, colouring by `Sport`. Add a smooth fitted curve to your scatterplot. Then, move `colour = Sport` from an `aes()` in the `ggplot()` function to an `aes()` in the `geom_point()` function. What changes in the plot? Can you give an explanation as to why that change occurs?

5. * Faceting can be used for other types of plots too! Make a pair of faceted histograms for a quantitative variable of your choosing that are faceted by a categorical variable of your choosing.

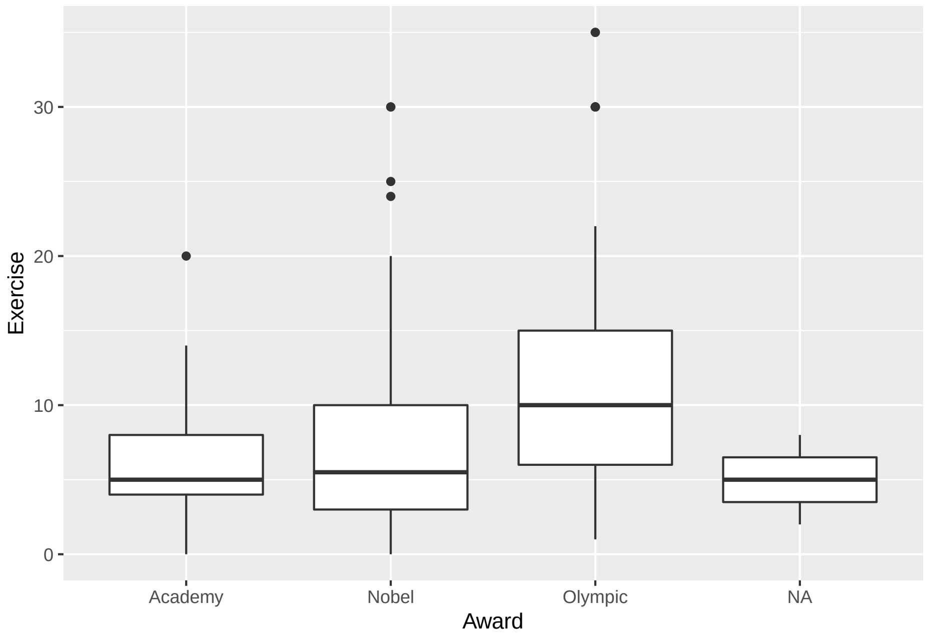
3.5 Boxplots, Stacked Barplots and Others

There are a few other common geoms that will be useful throughout the semester. These only skim the surface: we'll come back to plotting in a few weeks, after we're able to do more with data wrangling and reshaping.

3.5.1 Graphing a Quant. Variable vs. a Cat. Variable

Another common plot used in Intro Stat courses is a boxplot. Side-by-side boxplots are particularly useful if you want to compare a quantitative response variable across two or more levels of a categorical variable. Let's stick with the STAT 113 survey data to examine the relationship between `Exercise` and `Award` preference.

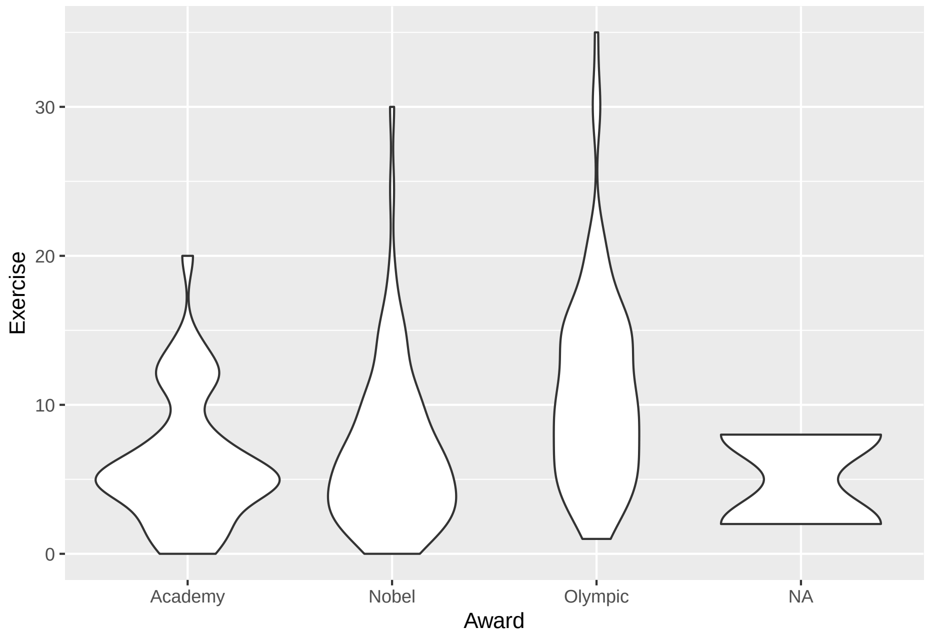
```r
ggplot(data = stat113_df, aes(x = Award, y = Exercise)) +
  geom_boxplot()
#> Warning: Removed 7 rows containing non-finite values
#> (stat_boxplot).
```



What can you conclude from the plot?

An alternative to side-by-side boxplots are violin plots:

```
ggplot(data = stat113_df, aes(x = Award, y = Exercise)) +
  geom_violin()
#> Warning: Removed 7 rows containing non-finite values
#> (stat_ydensity).
```



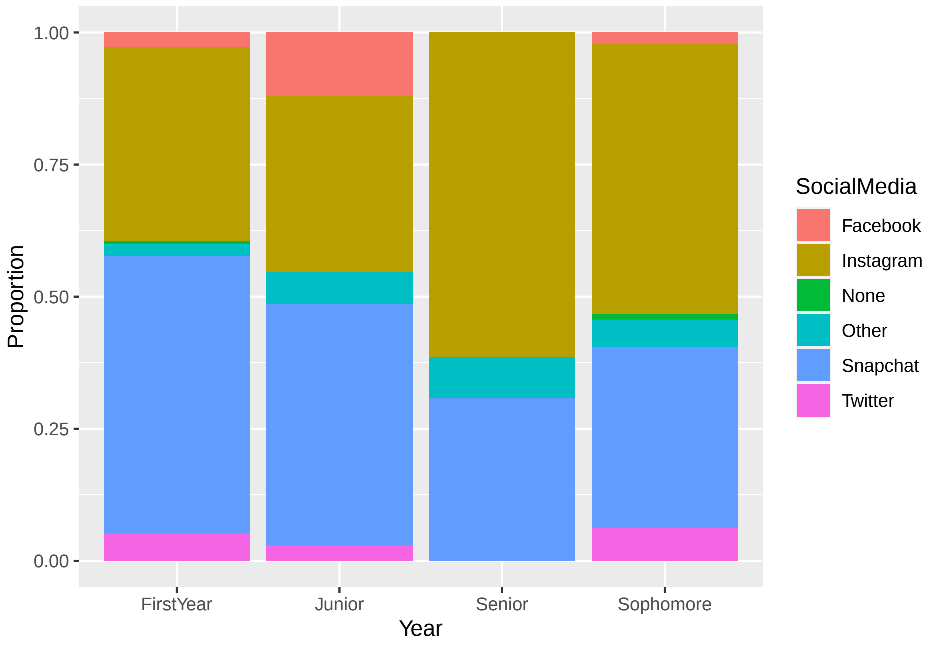
Read about Violin plots by typing `?geom_violin` into your console (bottom-left window). How are they different than boxplots?

3.5.2 Graphing Two Categorical Variables

The only combination of two variables that we have yet to explore are two variables that are both categorical. Let's look at the relationship between `Year` and `SocialMedia` first using a stacked bar plot.

To make the graph, we specify `position = "fill"` so that the bars are "filled" by `stepgoal`.

```
ggplot(data = stat113_df, aes(x = Year, fill = SocialMedia)) +
  geom_bar(position = "fill") +
  ylab("Proportion")
```



What patterns do you notice from the plot? Is there anything about the plot that could be improved?

3.5.3 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 3.7.

1. * Change the colour of the inside of the boxplots in the Exercise vs. Award graph to be `"blue"`. Do you think you'll use `colour = "blue"` or `fill = "blue"`?

2. * Create a side-by-side boxplot that compares the `GPA`s of students who prefer different `Award`s. Then change the fill of the boxplot to be a colour of your choice. What do you notice in the plot?

3. * When making the previous plot, `R` gives us a warning message that it "Removed 70 rows containing non-finite values". This is `R`'s robotic way of telling us that 70 `GPA` values are missing in the data set. Use what you know about how the data was collected (Fall and Spring semester of the 2018-2019 school-year) to guess why these are missing.

4. * Make a stacked bar plot for two variables of your choosing in the STAT 113 data set. Comment on something that you notice in the plot.

3.6 Chapter Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 3.7.

1. * The default of `geom_smooth()` is to use LOESS (locally estimated scatterplot smoothing). Read about LOESS here: here. Write one or two sentences explaining what LOESS does.

2. * Thus far, we have only faceted by a single variable. Use Google to figure out how to facet by two variables to make a plot that shows the relationship between GPA (y-axis) and Exercise (x-axis) with four facets: one for male students who play a sport, one for female students who play a sport, one for male students who do not play a sport, and one for female students who do not play a sport.

3. * In Intro-Stat, boxplots are typically introduced using the `*` symbol to identify outliers. Using a combination of the help `?geom_boxplot` and Googling "R point shapes", figure out how to modify your side-by-side boxplots so that the outliers are shown using `*`, not the default dots.

Then, using Google, figure out how to add the mean to each boxplot as a "darkgreen" diamond-shaped symbol with `stat_summary()`.

4. A common theme that we'll see throughout the course is that it's advantageous to know as much background information as possible about the data set we are analyzing. Data sets will be easier to analyze and pose questions about if you're familiar with the subject matter.

Give an example of something that you know about STAT 113 and the survey data set that helped you answer or pose a question that someone from another university (and therefore unfamiliar with our intro stat course) wouldn't know.

Give an example of something that you don't know about the fitness data set that the person who owns the fitness data would know. Why does that give an advantage to the person who is more familiar with the fitness data?

3.7 Exercise Solutions

3.7.1 Introduction etc. S

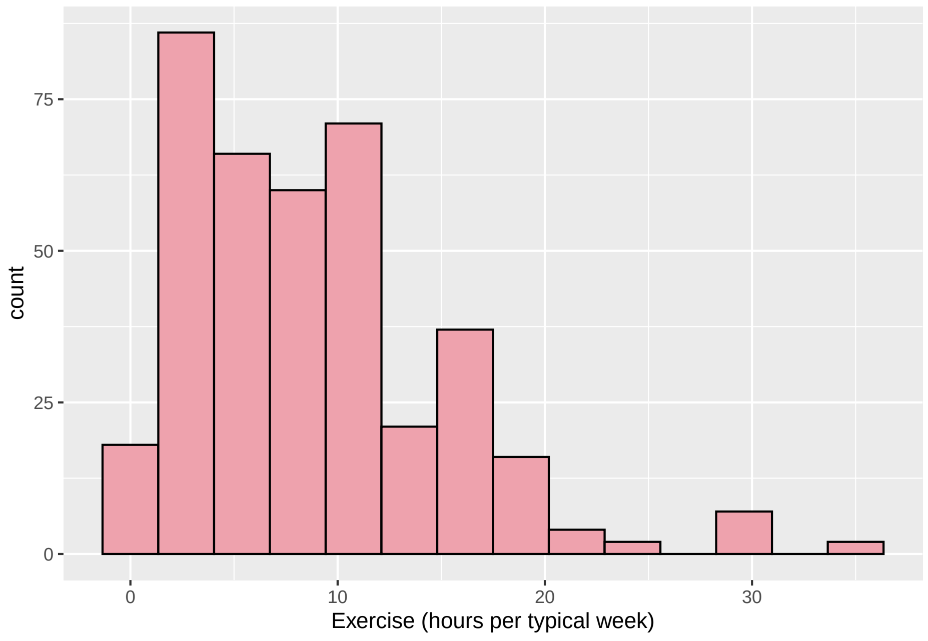
3.7.2 Basic Plot Structure S

3.7.3 Graphing a Single Variable S

4. * Create a histogram of the `Exercise` variable, change the x-axis label to be "Exercise (hours per typical week)", change the number of `bins` to `14`,

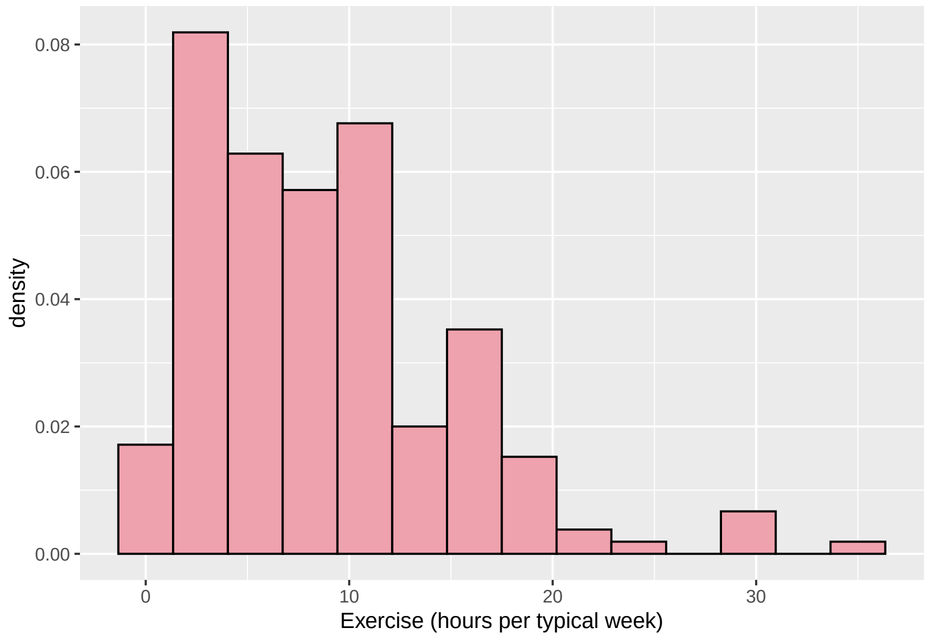
and change the `fill` of the bins to be "lightpink2" and the outline `colour` of the bins to be black.

```
ggplot(data = stat113_df, aes(x = Exercise)) +
  geom_histogram(bins = 14, fill = "lightpink2", colour = "black") +
  xlab("Exercise (hours per typical week)")
#> Warning: Removed 7 rows containing non-finite values
#> (stat_bin).
```



5. * We can change the y-axis of a histogram to be "density" instead of a raw count. This means that each bar shows a **proportion** of cases instead of a raw count. Google something like "geom_histogram with density" to figure out how to create a y `aes()` to show density instead of count.

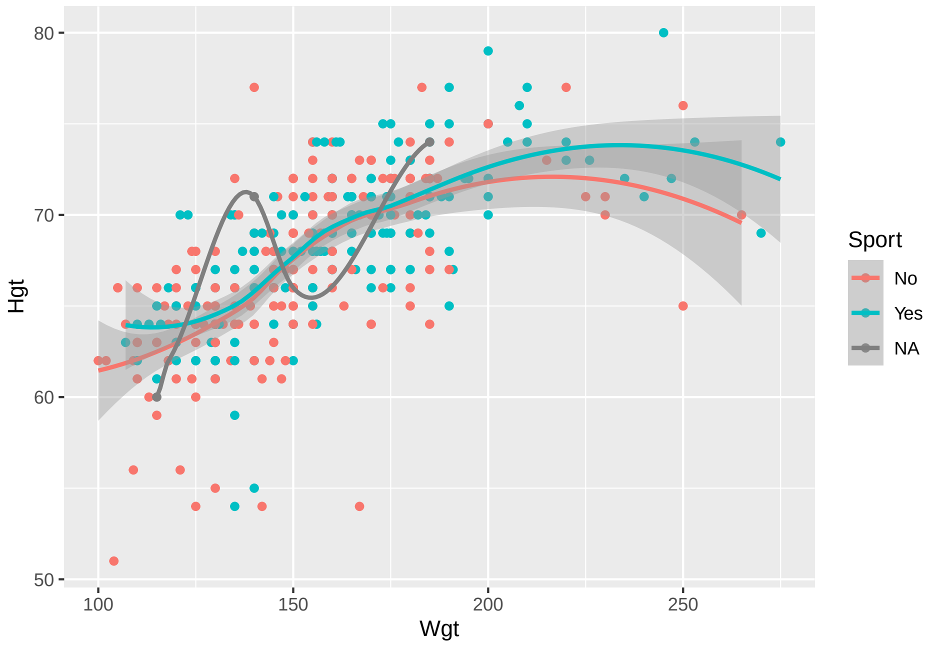
```
ggplot(data = stat113_df, aes(x = Exercise, y = ..density..)) +
  geom_histogram(bins = 14, fill = "lightpink2", colour = "black") +
  xlab("Exercise (hours per typical week)")
#> Warning: Removed 7 rows containing non-finite values
#> (stat_bin).
```



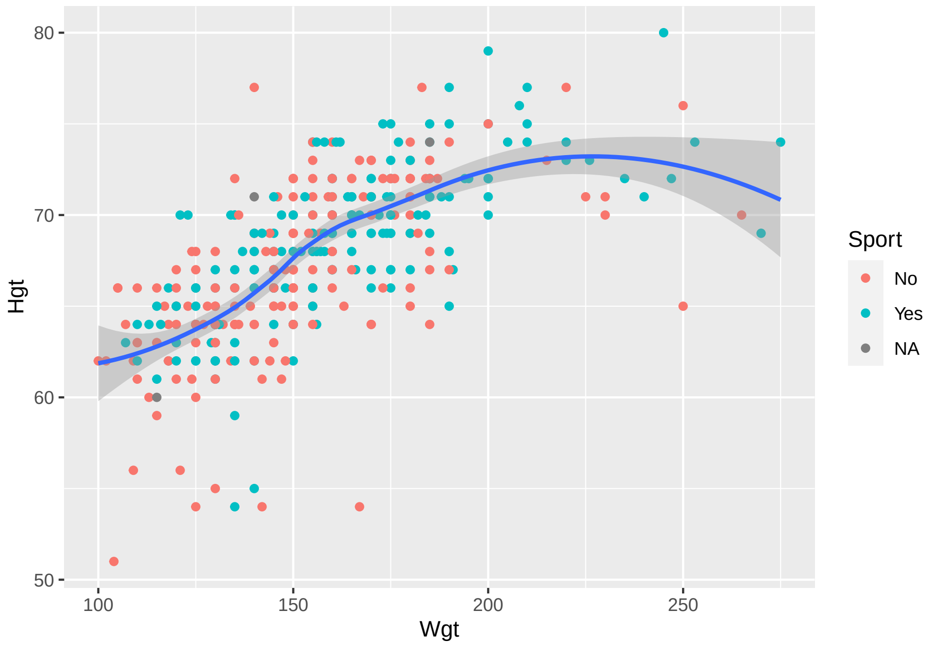
3.7.4 Graphing Two Quant. etc. S

4. * Make a scatterplot of `Hgt` on the y-axis and `Wgt` on the x-axis, colouring by `Sport`. Add a smooth fitted curve to your scatterplot. Then, move `colour = Sport` from an `aes()` in the `ggplot()` function to an `aes()` in the `geom_point()` function. What changes in the plot? Can you give an explanation as to why that change occurs?

```
ggplot(data = stat113_df, aes(x = Wgt, y = Hgt, colour = Sport)) +
  geom_point() +
  geom_smooth()
```



```
ggplot(data = stat113_df, aes(x = Wgt, y = Hgt)) +
  geom_point(aes(colour = Sport)) +
  geom_smooth()
```

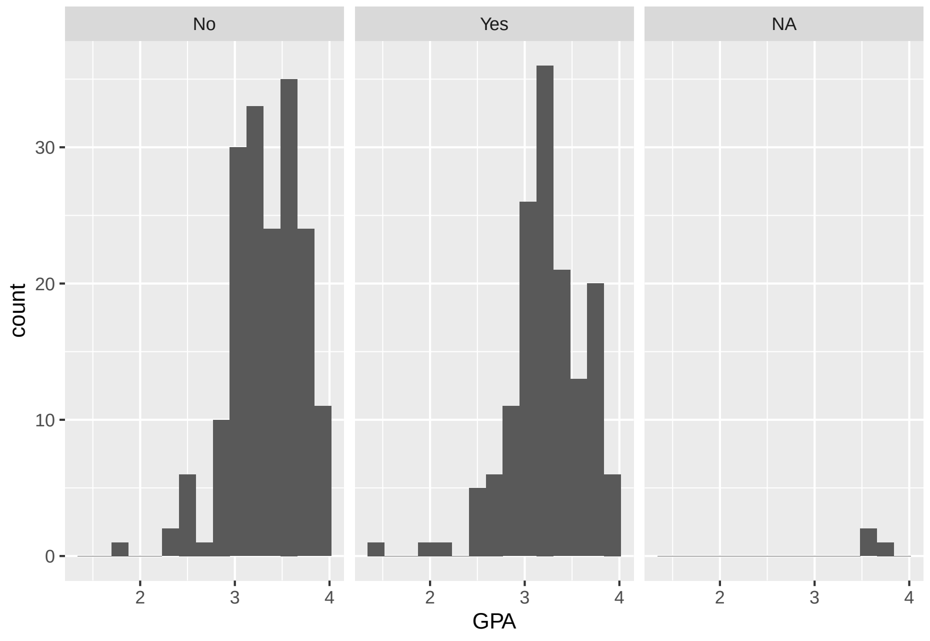


The points are now coloured by `Sport` but there is only one smooth fitted line. This makes sense because `geom_point()` now has the two global aesthetics x and y, as well as the colour aesthetic. `geom_smooth()` no longer has the colour aesthetic but still inherits the two global aesthetics, x and y.

5. * Faceting can be used for other types of plots too! Make a pair of faceted histograms for a quantitative variable of your choosing that are faceted by a categorical variable of your choosing.

Answers will vary:

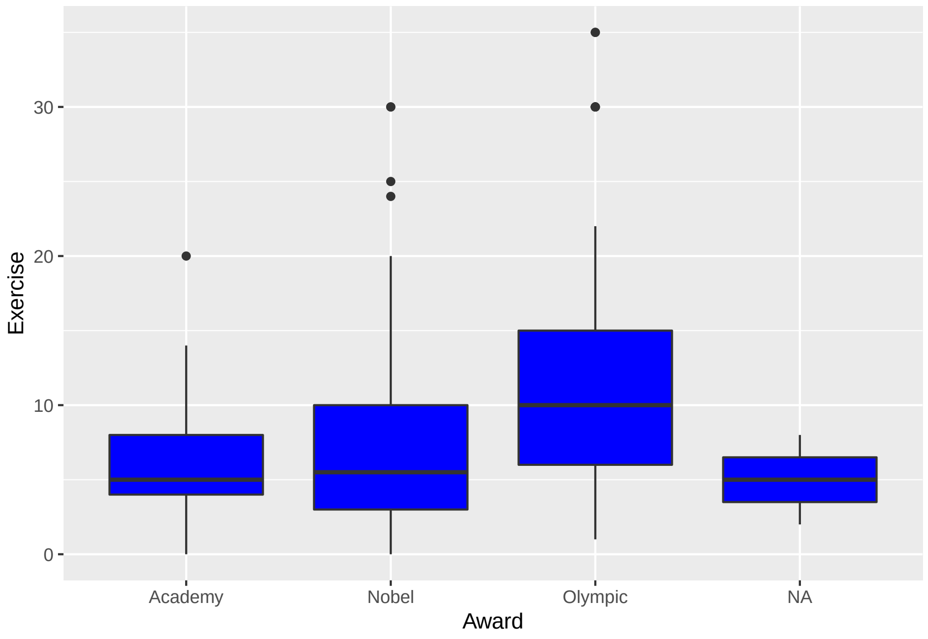
```
ggplot(data = stat113_df, aes(x = GPA)) +
  geom_histogram(bins = 15) +
  facet_wrap( ~ Sport)
```



3.7.5 Boxplots, Stacked, etc. S

1. * Change the colour of the inside of the boxplots in the Exercise vs. Award graph to be `"blue"`. Do you think you'll use `colour = "blue"` or `fill = "blue"`?

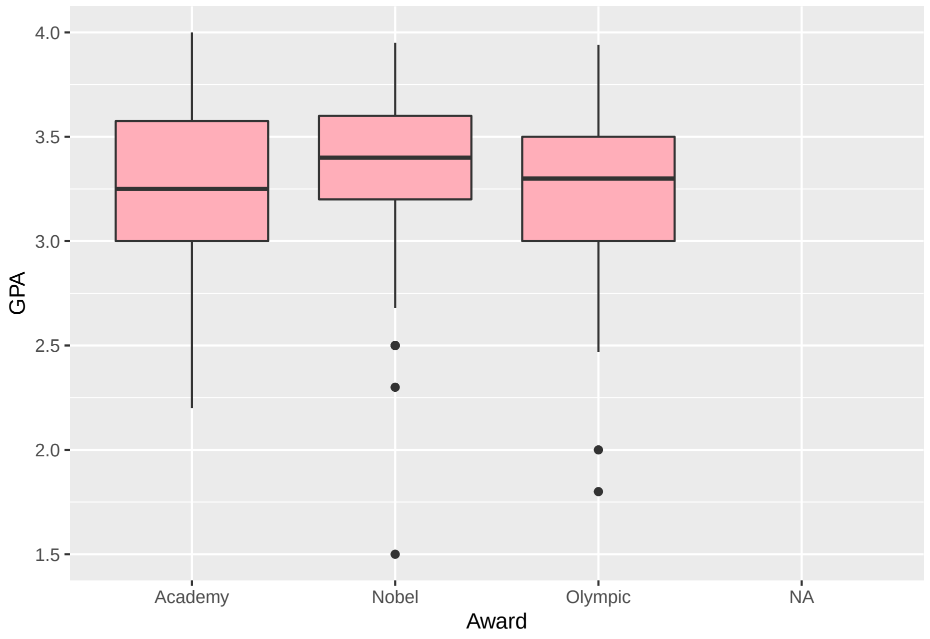
```
ggplot(data = stat113_df, aes(x = Award, y = Exercise)) +
  geom_boxplot(fill = "blue")
#> Warning: Removed 7 rows containing non-finite values
#> (stat_boxplot).
```



`fill` because it's the inside of the boxplots that we want to modify. `colour` will modify the outline colour.

2. * Create a side-by-side boxplot that compares the `GPA`s of students who prefer different `Award`s. Then change the fill of the boxplot to be a colour of your choice. What do you notice in the plot?

```
ggplot(data = stat113_df, aes(x = Award, y = GPA)) +
  geom_boxplot(fill = "lightpink1")
#> Warning: Removed 70 rows containing non-finite values
#> (stat_boxplot).
```



There are a few outlier students, but the three groups overall seem to have similar GPAs.

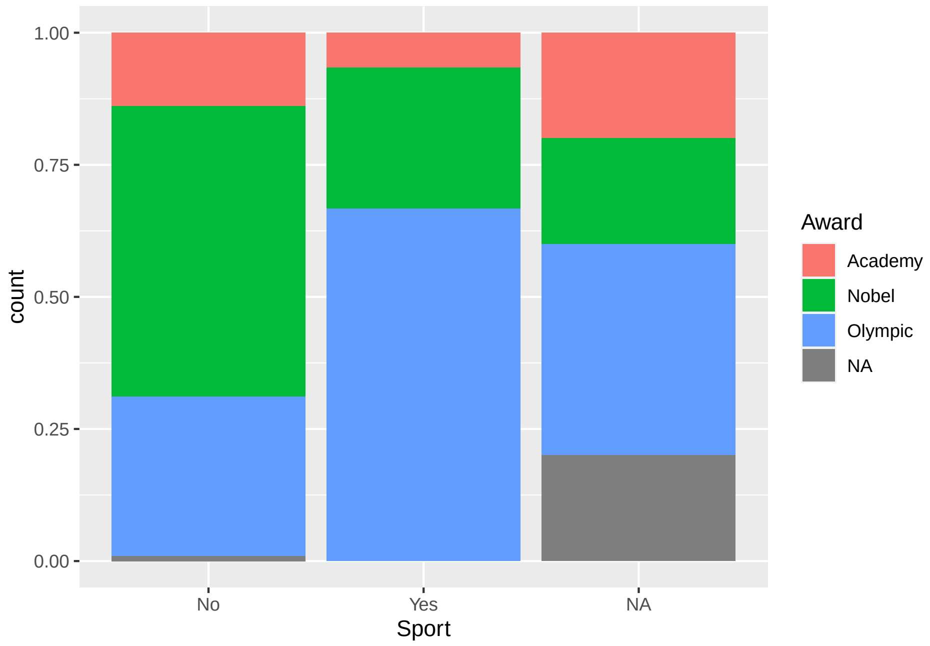
3. * When making the previous plot, `R` gives us a warning message that it "Removed 70 rows containing non-finite values". This is `R`'s robotic way of telling us that 70 `GPA` values are missing in the data set. Use what you know about how the data was collected (Fall and Spring semeseter of the 2018-2019 school-year) to guess why these are missing.

STAT 113 has first-year students: first-years taking the course in the fall would not have a GPA to report. Additionally, another reason might be that a student chose not to report his or her GPA.

4. * Make a stacked bar plot for two variables of your choosing in the STAT 113 data set. Comment on something that you notice in the plot.

Answers will vary.

```
ggplot(data = stat113_df, aes(x = Sport, fill = Award)) +
  geom_bar(position = "fill")
```



As we might expect, it does seem like a higher proportion of students who play a sport would prefer to win an Olympic medal, compared with students who do not play a sport.

3.7.6 Chapter Exercises S

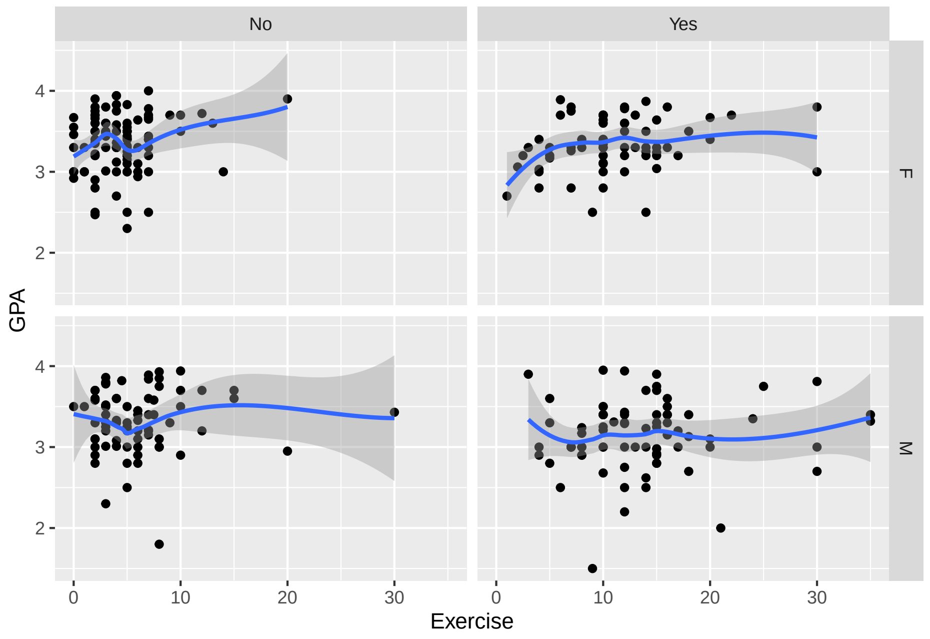
1. * The default of `geom_smooth()` is to use LOESS (locally estimated scatterplot smoothing). Read about LOESS here: here. Write one or two sentences explaining what LOESS does.

Loess uses a bunch of local regressions to predict the y-variable at each point, giving more weight to observations near the point of interest on the x-axis. Once this is done for every point, the predictions are connected with a smooth curve.

2. * Thus far, we have only faceted by a single variable. Use Google to figure out how to facet by two variables to make a plot that shows the relationship between GPA (y-axis) and Exercise (x-axis) with four facets: one for male students who play a sport, one for female students who play a sport, one for male students who do not play a sport, and one for female students who do not play a sport.

```
ggplot(data = stat113_df %>% filter(!is.na(Sport) & !is.na(Sex)),
  aes(x = Exercise, y = GPA)) +
  geom_point() + geom_smooth() +
  facet_grid(Sex ~ Sport)
```

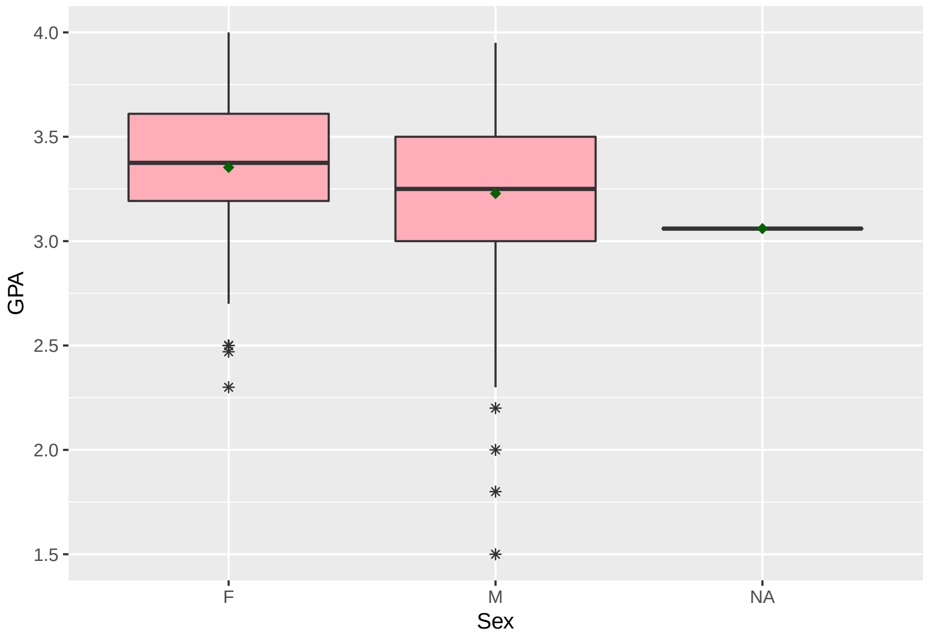
```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
#> Warning: Removed 71 rows containing non-finite values
#> (stat_smooth).
#> Warning: Removed 71 rows containing missing values
#> (geom_point).
```



3. * In Intro-Stat, boxplots are typically introduced using the `*` symbol to identify outliers. Using a combination of the help `?geom_boxplot` and Googling “R point shapes”, figure out how to modify your side-by-side boxplots so that the outliers are shown using `*`, not the default dots.

Then, using Google, figure out how to add the mean to each boxplot as a “darkgreen” diamond-shaped symbol with `stat_summary()`.

```
ggplot(data = stat113_df, aes(x = Sex, y = GPA)) +
  geom_boxplot(fill = "lightpink1", outlier.shape = 8) +
  stat_summary(fun = mean, shape = 18, colour = "darkgreen")
#> Warning: Removed 70 rows containing non-finite values
#> (stat_boxplot).
#> Warning: Removed 70 rows containing non-finite values
#> (stat_summary).
#> Warning: Removed 3 rows containing missing values
#> (geom_segment).
```



3.8 Non-Exercise R Code

```
library(tidyverse)
pres_df <- read_table("data/PRES2000.txt")
## don't worry about the `read_table` function....yet
head(pres_df)
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_histogram(colour = "black", fill = "white") +
  xlab("Votes for Gore in Florida")
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_freqpoly(colour = "black") +
  xlab("Votes for Gore in Florida")
ggplot(data = pres_df, mapping = aes(x = Gore)) +
  geom_freqpoly(colour = "black") +
  xlab("Votes for Gore in Florida") +
  geom_histogram()
pres_cat <- pres_df %>% mutate(winner = if_else(Gore > Bush,
                                                true = "Gore",
                                                false = "Bush"))
pres_cat
ggplot(data = pres_cat, aes(x = winner)) +
  geom_bar()
pres_cat2 <- pres_cat %>% group_by(winner) %>%
  summarise(nwins = n())
pres_cat2
ggplot(pres_cat2, aes(x = winner)) +
  geom_bar()
ggplot(pres_cat2, aes(x = winner, y = nwins)) +
  geom_col()
ggplot(data = pres_df, mapping = aes(x = Gore, y = Bush)) +
  geom_point()
library(tidyverse)
fitness_full <- read_csv("data/higham_fitness_clean.csv") %>% mutate(weekend_ind = case_when(week
  TRUE ~ "weekday"))
ggplot(data = fitness_full, aes(x = distance, y = active_cals)) +
  geom_point()
## drop observations that have active calories < 50.
## assuming that these are data errors or
## days where the Apple Watch wasn't worn.
fitness <- fitness_full %>%
  filter(active_cals > 50)
ggplot(data = fitness) +
  geom_point(aes(x = distance, y = active_cals))
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, colour = weekend_ind))
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, shape = weekend_ind))
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, size = flights))
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals, colour = "purple"))
ggplot(data = fitness) +
  geom_point(aes(x = Start, y = active_cals), size = 1.5, shape = 19)
ggplot(data = fitness, aes(x = Start, y = active_cals)) +
  geom_point() +
  geom_smooth()
ggplot(data = fitness, aes(x = Start, y = active_cals)) +
  geom_point() +
  geom_smooth(se = FALSE, method = "lm")
ggplot(data = fitness, mapping = aes(x = Start, y = steps)) +
  geom_point() + geom_smooth() + xlab("Date")
ggplot(data = fitness, mapping = aes(x = Start, y = steps)) +
  geom_line() + geom_smooth() + xlab("Date")
ggplot(data = stat113_df, aes(x = Exercise, y = Pulse,
                              colour = Year)) +
  geom_point() +
  geom_smooth(se = TRUE)
ggplot(data = stat113_df, aes(x = Exercise, y = Pulse)) +
```

```r
  geom_point() +
  geom_smooth(se = TRUE) +
  facet_wrap(~ Year)
ggplot(data = stat113_df, aes(x = Award, y = Exercise)) +
  geom_boxplot()
ggplot(data = stat113_df, aes(x = Award, y = Exercise)) +
  geom_violin()
ggplot(data = stat113_df, aes(x = Year, fill = SocialMedia)) +
  geom_bar(position = "fill") +
  ylab("Proportion")
```

Chapter 4

Wrangling with dplyr

Goals:

- Use the `mutate()`, `if_else()`, and `case_when()` functions to create new variables.
- Use the `filter()`, `select()`, and `arrange()` functions in `dplyr` to choose certain rows to keep or get rid of, choose certain columns to keep or get rid of, and to sort the data, respectively.
- Use `group_by()` and `summarise()` to create useful summaries of a data set.
- Combine the above goals with plotting to explore the `babynames` data set and a data set on SLU majors.

Throughout this chapter, we will use the `babynames` data set in the `babynames` `R` package. To begin, read about the data set, by running

```
library(babynames)
```

and then typing `?babynames` in your bottom-left window of `R Studio`. We see that this data set contains baby name data provided by the SSA in the United States dating back to 1880:

```
head(babynames)
#> # A tibble: 6 x 5
#>    year sex   name          n   prop
#>   <dbl> <chr> <chr>     <int>  <dbl>
#> 1  1880 F     Mary       7065 0.0724
#> 2  1880 F     Anna       2604 0.0267
#> 3  1880 F     Emma       2003 0.0205
#> 4  1880 F     Elizabeth  1939 0.0199
#> 5  1880 F     Minnie      1746 0.0179
#> 6  1880 F     Margaret    1578 0.0162
```

The second data set that we will use has 27 observations, one for each of SLU's majors and contains 3 variables:

- `Major`, the name of the major.
- `nfemales`, the number of female graduates in that major from 2015 - 2019.
- `nmales`, the number of male graduates in that major from 2015 - 2019.

The data has kindly been provided by Dr. Ramler. With your Notes `R Project` open, you can read in the data set with

```
library(tidyverse)
slumajors_df <- read_csv("data/SLU_Majors_15_19.csv")
slumajors_df
#> # A tibble: 27 x 3
#>    Major                        nfemales nmales
#>    <chr>                           <dbl>  <dbl>
#>  1 Anthropology                       34     15
#>  2 Art & Art History                  65     11
#>  3 Biochemistry                       14     11
#>  4 Biology                           162     67
#>  5 Business in the Liberal Arts      135    251
#>  6 Chemistry                          26     14
#>  7 Computer Science                   21     47
#>  8 Conservation Biology               38     20
#>  9 Economics                         128    349
#> 10 English                           131     54
#> # ... with 17 more rows
```

There are many interesting and informative plots that we could make with either data set, but most require some data wrangling first. This chapter will provide the foundation for such wrangling skills.

4.1 `mutate()`: Create Variables

Sometimes, we will want to create a new variable that's not in the data set, oftentimes using `if_else()`, `case_when()`, or basic algebraic operations on one or more of the columns already present in the data set.

`R` understands the following symbols:

- `+` for addition, `-` for subtraction
- `*` for multiplication, `/` for division
- `^` for raising something to a power (`3 ^ 2` is equal to `9`)

R also does the same order of operations as usual: parentheses, then exponents, then multiplication and division, then addition and subtraction.

For example, suppose that we want to create a variable in `slumajors_df` that has the total number of students graduating in each major. We can do this with `mutate()`:

```
slumajors_df %>% mutate(ntotal = nfemales + nmales)
#> # A tibble: 27 x 4
#>    Major                         nfemales nmales ntotal
#>    <chr>                            <dbl>  <dbl>  <dbl>
#>  1 Anthropology                        34     15     49
#>  2 Art & Art History                   65     11     76
#>  3 Biochemistry                        14     11     25
#>  4 Biology                            162     67    229
#>  5 Business in the Liberal Arts       135    251    386
#>  6 Chemistry                           26     14     40
#>  7 Computer Science                    21     47     68
#>  8 Conservation Biology                38     20     58
#>  9 Economics                          128    349    477
#> 10 English                            131     54    185
#> # ... with 17 more rows
```

There's a lot to break down in that code chunk: most importantly, we're seeing our first of many, many, many, many, many, many, many instances of using `%>%` to pipe! The `%>%` operator approximately reads take `slumajors_df` "and then" `mutate()` it.

Piping is a really convenient, easy-to-read way to build a sequence of commands. How you can read the above code is:

1. Take `slumajors_df` and with `slumajors_df`,
2. perform a `mutate()` step to create the new variable called `ntotal`, which is `nfemales` plus `nmales`.

Since this is our first time using `mutate()`, let's also delve into what the function is doing. In general, `mutate()` reads:

`mutate(name_of_new_variable = operations_on_old_variables)`.

R just automatically assumes that you want to do the operation for every single row in the data set, which is often quite convenient!

We might also want to create a variable that is the percentage of students identifying as female for each major:

```
slumajors_df %>%
  mutate(percfemale = 100 * nfemales / (nfemales + nmales))
#> # A tibble: 27 x 4
#>    Major                         nfemales nmales percfemale
#>    <chr>                         <dbl>  <dbl>       <dbl>
#>  1 Anthropology                     34     15        69.4
#>  2 Art & Art History                65     11        85.5
#>  3 Biochemistry                     14     11        56
#>  4 Biology                         162     67        70.7
#>  5 Business in the Liberal Arts    135    251        35.0
#>  6 Chemistry                        26     14        65
#>  7 Computer Science                 21     47        30.9
#>  8 Conservation Biology             38     20        65.5
#>  9 Economics                       128    349        26.8
#> 10 English                         131     54        70.8
#> # ... with 17 more rows
```

But what happened to `ntotal`? Is it still in the printout? It's not: when we created the variable `ntotal`, we didn't actually **save** the new data set as anything. So `R` makes and prints the new variable, but it doesn't get saved to any data set. If we want to save the new data set, then we can use the `<-` operator. Here, we're saving the new data set with the same name as the old data set: `slumajors_df`. Then, we're doing the same thing for the `percfemale` variable. We won't always want to give the new data set the same name as the old one: we'll talk about this more in the chapter exercises.

```
slumajors_df <- slumajors_df %>%
  mutate(percfemale = 100 * nfemales / (nfemales + nmales))
```

```
slumajors_df <- slumajors_df %>% mutate(ntotal = nfemales + nmales)
```

But, you can pipe as many things together as you want to, so it's probably easier to just create both variables in one go. The following chunk says to "Take `slumajors_df` and create a new variable `ntotal`. With that new data set, create a new variable called `percfemale`." Finally, the `slumajors_df <-` at the beginning says to "save this new data set as a data set with the same name, `slumajors_df`."

```
slumajors_df <- slumajors_df %>%
  mutate(ntotal = nfemales + nmales) %>%
  mutate(percfemale = 100 * nfemales / (nfemales + nmales))
```

4.1.1 A Little More on Piping

We are jumping straight into using piping, but we do want to have an appreciation on how terrible life would be without it. What piping does is make whatever is given before the `%>%` pipe the first argument of whatever function follows the `%>%`. So

```
df %>% mutate(x = y + 4)
```

is equivalent to

```
mutate(df, x = y + 4)
```

Piping really isn't that useful if you just have something that can be done with a single `%>%`. But, doing our previous example without piping might look like:

```
mutate(mutate(slumajors_df, ntotal = nfemales + nmales), percfemale = 100 * nfemales / (nfemales
#> # A tibble: 27 x 5
#>    Major                      nfemales nmales percfemale ntotal
#>    <chr>                         <dbl>  <dbl>      <dbl>  <dbl>
#>  1 Anthropology                     34     15       69.4     49
#>  2 Art & Art History                65     11       85.5     76
#>  3 Biochemistry                     14     11       56       25
#>  4 Biology                         162     67       70.7    229
#>  5 Business in the Libera~         135    251       35.0    386
#>  6 Chemistry                        26     14       65       40
#>  7 Computer Science                 21     47       30.9     68
#>  8 Conservation Biology             38     20       65.5     58
#>  9 Economics                       128    349       26.8    477
#> 10 English                         131     54       70.8    185
#> # ... with 17 more rows
```

It's still not **that** bad here because we aren't doing **that** many operations to the data set, but it's already much harder to read. But we will get to examples where you are using 5+ pipes.

It might also help to use an analogy when thinking about piping. Consider the Ke$ha's morning routine in the opening of the song Tik Tok. If we were to write her morning routine in terms of piping,

```
kesha %>% wake_up(time = "morning", feels_like = "P-Diddy") %>%
  grab(glasses) %>%
  brush(teeth, item = "jack", unit = "bottle") %>% ....
```

Kesha first wakes up in the morning, *and then* the Kesha that has woken up grabs her glasses, *and then* the Kesha who has woken up and has her glasses brushes her teeth, etc.

4.1.2 `if_else()` and `case_when()`

Suppose that you want to make a new variable that is conditional on another variable (or more than one variable) in the data set. Then we would typically use `mutate()` coupled with

- `if_else()` if your new variable is created on only one condition

- `case_when()` if your new variable is created on more than one condition

Suppose we want to create a new variable that tells us whether or not the `Major` has a majority of Women. That is, we want this new variable, `morewomen` to be `"Yes"` if the `Major` has more than 50% women and `"No"` if it has 50% or less.

```
slumajors_df %>% mutate(morewomen = if_else(percfemale > 50,
                                            true = "Yes",
                                            false = "No"))
#> # A tibble: 27 x 6
#>    Major          nfemales nmales percfemale ntotal morewomen
#>    <chr>             <dbl>  <dbl>      <dbl>  <dbl> <chr>
#>  1 Anthropology         34     15       69.4     49 Yes
#>  2 Art & Art Hi~        65     11       85.5     76 Yes
#>  3 Biochemistry         14     11       56       25 Yes
#>  4 Biology             162     67       70.7    229 Yes
#>  5 Business in ~       135    251       35.0    386 No
#>  6 Chemistry            26     14       65       40 Yes
#>  7 Computer Sci~        21     47       30.9     68 No
#>  8 Conservation~        38     20       65.5     58 Yes
#>  9 Economics           128    349       26.8    477 No
#> 10 English             131     54       70.8    185 Yes
#> # ... with 17 more rows
```

The `mutate()` statement reads: create a new variable called `morewomen` that is equal to `"Yes"` if `percfemale` > `50` is true and is equal to `"No"` if `perfemale` is not > `0.5`. The first argument is the condition, the second is what to name the new variable when the condition holds, and the third is what to name the variable if the condition does not hold.

We use **conditions** all of the time in every day life. For example, New York had a quarantine order stating that people coming from 22 states in July 2020 would need to quarantine. In terms of a condition, this would read "if you are traveling to New York from one of the 22 states, then you need to quarantine for 2 weeks. Else, if not, then you don't need to quarantine." The trick in using these conditions in `R` is getting used to the syntax of the code.

We can see from the above set up that if we had more than one condition, then we'd need to use a different function (or use nested `if_else()` statements, which can be a nightmare to read). If we have more than one condition for creating the new variable, we will use `case_when()`.

For example, when looking at the output, we see that `Biochemistry` has 56% female graduates. That's "about" a 50/50 split, so suppose we want a variable called `large_majority` that is "female" when the percent women is 70 or more, "male" when the percent women is 30 or less, and "none" when the percent female is between 30 and 70.

```
slumajors_df %>% mutate(large_majority =
                          case_when(percfemale >= 70 ~ "female",
                                    percfemale <= 30 ~ "male",
                                    percfemale > 30 & percfemale < 70 ~ "none"))
#> # A tibble: 27 x 6
#>    Major    nfemales nmales percfemale ntotal large_majority
#>    <chr>       <dbl>  <dbl>      <dbl>  <dbl> <chr>
#>  1 Anthrop~       34     15       69.4     49 none
#>  2 Art & A~       65     11       85.5     76 female
#>  3 Biochem~       14     11       56       25 none
#>  4 Biology       162     67       70.7    229 female
#>  5 Busines~      135    251       35.0    386 none
#>  6 Chemist~       26     14       65       40 none
#>  7 Compute~       21     47       30.9     68 none
#>  8 Conserv~       38     20       65.5     58 none
#>  9 Economi~      128    349       26.8    477 male
#> 10 English       131     54       70.8    185 female
#> # ... with 17 more rows
```

The `case_when()` function reads "When the percent female is more than or equal to 70, assign the new variable `large_majority` the value of"female", when it's less or equal to 30, assign the more than 30 and less than 70, assign the variable the value of"none" ." The `&` is a boolean operator: we'll talk more about that later so don't worry too much about that for now.

Let's save these two new variables to the `slumajors_df`:

```
slumajors_df <- slumajors_df %>%
  mutate(morewomen = if_else(percfemale > 50,
                             true = "Yes",
                             false = "No")) %>%
  mutate(large_majority =
           case_when(percfemale >= 70 ~ "female",
                     percfemale <= 30 ~ "male",
                     percfemale > 30 & percfemale < 70 ~ "none"))
```

4.1.3 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 4.6.

1. Do you think it is ethical to exclude non-binary genders from analyses and graphs in the slumajors data set? Why or why not?

2. * Create a new variable that is called `major_size` and is "large" when the total number of majors is 100 or more and "small" when the total number

of majors is less than 100.

3. Create a new variable that is called `major_size2` and is “large when the total number of majors is 150 or more,”medium” when the total number of majors is between 41 and 149, and “small” when the total number of majors is 40 or fewer.
4. About 55% of SLU students identify as female. So, in the definition of the `morewomen` variable, does it make more sense to use 55% as the cutoff or 50%?
5. * Investigate what happens with `case_when()` when you give overlapping conditions and when you give conditions that don’t cover all observations. For overlapping conditions, create a variable `testcase` that is `"Yes"` when `percfemale` is greater than or equal to 40 and `"No"` when `percfemale` is greater than 60 For conditions that don’t cover all observations, create a variable `testcase2` that is `"Yes"` when `percefemale` is greater than or equal to 55 and `"No"` when `percfemale` is less than 35.
6. With one or two of the newly created variables from `mutate()`, create a plot that investigates a question of interest you might have about the data.

4.2 `arrange()`, `select()`, and `slice()`, and `filter()`

`arrange()` is used to order rows in the data set according to some variable, `select()` is used to choose columns to keep (or get rid of) and `filter()` is used to keep (or get rid of) only some of the observations (rows).

4.2.1 `arrange()`: Ordering Rows

The `arrange()` function allows us to order rows in the data set using one or more variables. The function is very straightforward. Suppose that we want to order the rows so that the majors with the lowest `percfemale` are first:

```
slumajors_df %>% arrange(percfemale)
#> # A tibble: 27 x 7
#>   Major          nfemales nmales percfemale ntotal morewomen
#>   <chr>             <dbl>  <dbl>      <dbl>  <dbl> <chr>
#> 1 Economics           128    349       26.8    477 No
#> 2 Physics               6     14       30       20 No
#> 3 Computer Sci~        21     47       30.9     68 No
#> 4 Business in ~       135    251       35.0    386 No
#> 5 Music                13     21       38.2     34 No
#>  6 Geology              28     41       40.6     69 No
#>  7 History              62     82       43.1    144 No
#>  8 Philosophy           24     29       45.3     53 No
#>  9 Mathematics          74     83       47.1    157 No
#> 10 Government          127    116       52.3    243 Yes
#> # ... with 17 more rows, and 1 more variable:
#> #   large_majority <chr>
```

Which major has the lowest percentage of female graduates?

We see that, by default, `arrange()` orders the rows from low to high. To order from high to low so that the majors with the highest `percfemale` are first, use `desc()` around the variable that you are ordering by:

```
slumajors_df %>% arrange(desc(percfemale))
#> # A tibble: 27 x 7
#>    Major         nfemales nmales percfemale ntotal morewomen
#>    <chr>            <dbl>  <dbl>      <dbl>  <dbl> <chr>
#>  1 Art & Art Hi~       65     11       85.5     76 Yes
#>  2 Psychology         278     61       82.0    339 Yes
#>  3 French              27      7       79.4     34 Yes
#>  4 Spanish             35     10       77.8     45 Yes
#>  5 Statistics          28      9       75.7     37 Yes
#>  6 Global Studi~       69     27       71.9     96 Yes
#>  7 Neuroscience        61     24       71.8     85 Yes
#>  8 Performance ~      144     57       71.6    201 Yes
#>  9 Religious St~       10      4       71.4     14 Yes
#> 10 English            131     54       70.8    185 Yes
#> # ... with 17 more rows, and 1 more variable:
#> #   large_majority <chr>
```

What is the major with the highest percentage of women graduates?

4.2.2 `select()` Choose Columns

We might also be interested in getting rid of some of the columns in a data set. One reason to do this is if there are an overwhelming (30+) columns in a data set, but we know that we just need a few of them. The easiest way to use `select()` is to just input the names of the columns that you want to keep. For example, if we were only interested in majors and their totals, we could do

```
slumajors_df %>% select(Major, ntotal)
#> # A tibble: 27 x 2
#>    Major                         ntotal
#>    <chr>                          <dbl>
#>  1 Anthropology                      49
#>  2 Art & Art History                 76
#>  3 Biochemistry                      25
#>  4 Biology                          229
#>  5 Business in the Liberal Arts     386
#>  6 Chemistry                         40
#>  7 Computer Science                  68
#>  8 Conservation Biology              58
#>  9 Economics                        477
#> 10 English                          185
#> # ... with 17 more rows
```

If I wanted to use this data set for anything else, I'd also need to name, or rename, it with `<-`. We would probably want to name it something other than `slumajors_df` so as to not overwrite the original data set, in case we want to use those other variables again later!

We might also want to use `select()` to get rid of one or two columns. If this is the case, we denote any column you want to get rid of with `-`. For example, we might want to get rid of the `ntotal` column that we made and get rid of the `nmales` and `nfemales` columns:

```
slumajors_df %>% select(-ntotal, -nfemales, -nmales)
#> # A tibble: 27 x 4
#>    Major                  percfemale morewomen large_majority
#>    <chr>                       <dbl> <chr>     <chr>
#>  1 Anthropology                 69.4 Yes       none
#>  2 Art & Art History            85.5 Yes       female
#>  3 Biochemistry                 56   Yes       none
#>  4 Biology                      70.7 Yes       female
#>  5 Business in the Libe~        35.0 No        none
#>  6 Chemistry                    65   Yes       none
#>  7 Computer Science             30.9 No        none
#>  8 Conservation Biology         65.5 Yes       none
#>  9 Economics                    26.8 No        male
#> 10 English                      70.8 Yes       female
#> # ... with 17 more rows
```

`select()` comes with many useful helper functions, but these are oftentimes not needed. One of the helper functions that **is** actually often useful is `everything()`. We can, for example, use this after using `mutate()` to put the variable that was just created at the front of the data set to make sure there weren't any unexpected issues:

```
slumajors_df %>% mutate(propfemale = percfemale / 100) %>%
  select(propfemale, everything())
#> # A tibble: 27 x 8
#>    propfemale Major        nfemales nmales percfemale ntotal
#>          <dbl> <chr>            <dbl>  <dbl>       <dbl>  <dbl>
#>   1      0.694 Anthropology        34     15        69.4     49
#>   2      0.855 Art & Art H~        65     11        85.5     76
#>   3      0.56  Biochemistry        14     11        56       25
#>   4      0.707 Biology            162     67        70.7    229
#>   5      0.350 Business in~       135    251        35.0    386
#>   6      0.65  Chemistry           26     14        65       40
#>   7      0.309 Computer Sc~        21     47        30.9     68
#>   8      0.655 Conservatio~        38     20        65.5     58
#>   9      0.268 Economics          128    349        26.8    477
#> 10      0.708 English            131     54        70.8    185
#> # ... with 17 more rows, and 2 more variables:
#> #   morewomen <chr>, large_majority <chr>
```

Verify that `propfemale` now appears first in the data set. `everything()` tacks on all of the remaining variables after `propfemale`. So, in this case, it's a useful way to re-order the columns so that what you might be most interested in appears first.

4.2.3 `slice()` and `filter()`: Choose Rows

Instead of choosing which columns to keep, we can also choose certain rows to keep using either `slice()` or `filter()`.

`slice()` allows you to specify the **row numbers** corresponding to rows that you want to keep. For example, suppose that we only want to keep the rows with the five most popular majors:

```
slumajors_df %>% arrange(desc(ntotal)) %>%
  slice(1, 2, 3, 4, 5)
#> # A tibble: 5 x 7
#>   Major          nfemales nmales percfemale ntotal morewomen
#>   <chr>             <dbl>  <dbl>      <dbl>  <dbl> <chr>
#> 1 Economics           128    349       26.8    477 No
#> 2 Business in t~      135    251       35.0    386 No
#> 3 Psychology          278     61       82.0    339 Yes
#> 4 Government          127    116       52.3    243 Yes
#> 5 Biology             162     67       70.7    229 Yes
#> # ... with 1 more variable: large_majority <chr>
```

We can alternatively use `slice(1:5)`, which is shorthand for `slice(1, 2, 3, 4, 5)`. While `slice()` is useful, it is relatively simple. We'll come back to it again in a few weeks as well when we discuss subsetting in base `R`.

`filter()` is a way to keep rows by specifying a **condition** related to one or more of the variables in the data set. We've already seen conditions in `if_else()`

and `case_when()` statements, but they'll now be used to "filter" the rows in our data set.

We can keep rows based on a categorical variable or a quantitative variable or a combination of any number of categorical and quantitative variables. `R` uses the following symbols to make comparisons. We've already been using the more intuitive symbols (like `<` and `>`):

- `<` and `<=` for less than and less than or equal to, respectively
- `>` and `>=` for greater than and greater than or equal to, respectively
- `==` for equal to (careful: equal to is a double equal sign `==`)
- `!=` for not equal to (in general, `!` denotes "not")

It's probably time for a change of data set too! We'll be working with the `babynames` data set for the rest of this chapter:

```
library(babynames)
babynames
#> # A tibble: 1,924,665 x 5
#>     year sex   name          n   prop
#>    <dbl> <chr> <chr>     <int>  <dbl>
#>  1  1880 F     Mary       7065 0.0724
#>  2  1880 F     Anna       2604 0.0267
#>  3  1880 F     Emma       2003 0.0205
#>  4  1880 F     Elizabeth  1939 0.0199
#>  5  1880 F     Minnie     1746 0.0179
#>  6  1880 F     Margaret   1578 0.0162
#>  7  1880 F     Ida        1472 0.0151
#>  8  1880 F     Alice      1414 0.0145
#>  9  1880 F     Bertha     1320 0.0135
#> 10  1880 F     Sarah      1288 0.0132
#> # ... with 1,924,655 more rows
```

If needed, we can remind ourselves what is in the `babynames` data set by typing `?babynames` in the console window.

What do the following statements do? See if you can guess before running the code.

```
babynames %>% filter(name == "Matthew")
babynames %>% filter(year >= 2000)
babynames %>% filter(sex != "M")
babynames %>% filter(prop > 0.05)
babynames %>% filter(year == max(year))
```

Why are some things put in quotes, like `"Matthew"` while some things aren't, like `2000`? Can you make out a pattern?

We can also combine conditions on multiple variables in `filter()` using Boolean operators. We've already seen one of these in the `case_when()` statement above: `&` means "and".

Look at the Venn diagrams in `R` for Data Science to learn about the various Boolean operators you can use in `R`: https://r4ds.had.co.nz/transform.html#logical-operators. The Boolean operators can be used in other functions in `R` as well, as we've already seen with `if_else()` and `case_when()`.

The following gives some examples. See if you can figure out what each line of code is doing before running it.

```
babynames %>% filter(n > 20000 | prop > 0.05)
babynames %>% filter(sex == "F" & name == "Mary")
babynames %>% filter(sex == "F" & name == "Mary" & prop > 0.05)
```

4.2.4 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 4.6.

1. What happens when you `arrange()` by one of the categorical variables in the `slumajors_df` data set?

2. * Use `select()` and `everything()` to put the `large_majority` variable as the first column in the `slumajors_df` data set.

3. * In the `babynames` data set, use `filter()`, `mutate()` with `rank()`, and `arrange()` to print the 10 most popular Male babynames in 2017.

4. In the `babynames` data set, use `filter()` to keep only the rows with your name (or, another name that interests you) and one sex (either `"M"` or `"F"`). Name the new data set something and then construct a line plot that looks at the either the `n` or `prop` of your chosen name through `year`.

4.3 `summarise()` and `group_by()`: Create Summaries

The `summarise()` function is useful to get summaries from the data. For example, suppose that we want to know the average major size at SLU across the five year span or the total number of majors across those five years. Then we can use `summarise()` and a summary function, like `mean()`, `sum()`, `median()`, `max()`, `min()`, `n()`, etc. You'll notice that the format of `summarise()` is extremely similar to the format of `mutate()`. Using the `slumajors_df` data again just for one quick example,

```
slumajors_df %>%
  summarise(meantotalmajor = mean(ntotal),
            totalgrad = sum(ntotal))
#> # A tibble: 1 x 2
#>   meantotalmajor totalgrad
#>            <dbl>     <dbl>
#> 1           124.      3347
```

4.3.1 `group_by()`: Groups

`summarise()` is often most useful when paired with a `group_by()` statement. Doing so allows us to get summaries across different groups.

For example, suppose that you wanted the total number of registered births per year in the `babynames` data set:

```
babynames %>% group_by(year) %>%
  summarise(totalbirths = sum(n))
#> # A tibble: 138 x 2
#>     year totalbirths
#>    <dbl>       <int>
#>  1  1880      201484
#>  2  1881      192696
#>  3  1882      221533
#>  4  1883      216946
#>  5  1884      243462
#>  6  1885      240854
#>  7  1886      255317
#>  8  1887      247394
#>  9  1888      299473
#> 10  1889      288946
#> # ... with 128 more rows
```

`group_by()` takes a grouping variable, and then, using `summarise()` computes the given summary function on each group.

Most summary functions are intuitive if you've had intro stat. But, if you're not sure whether the summary for getting the maximum is `maximum()` or `max()`, just try both!

The `n()` function can be used within `summarise()` to obtain the number of observations. It will give you the total number of rows, if used without `group_by()`

```
babynames %>% summarise(totalobs = n())
#> # A tibble: 1 x 1
#>   totalobs
#>       <int>
#> 1  1924665
```

Note that `n()` typically doesn't have any inputs. It's typically more useful when paired with `group_by()`: this allows us to see the number of observations within each `year`, for instance:

```
babynames %>% group_by(year) %>%
  summarise(ngroup = n())
#> # A tibble: 138 x 2
#>     year ngroup
#>    <dbl>  <int>
#>  1  1880   2000
#>  2  1881   1935
#>  3  1882   2127
#>  4  1883   2084
#>  5  1884   2297
#>  6  1885   2294
#>  7  1886   2392
#>  8  1887   2373
#>  9  1888   2651
#> 10  1889   2590
#> # ... with 128 more rows
```

4.3.2 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 4.6.

1. Compare `summarise()` with `mutate()` using the following code. What's the difference between the two functions?

```
slumajors_df %>%
  summarise(meantotalmajor = mean(ntotal),
            totalgrad = sum(ntotal))
slumajors_df %>%
  mutate(meantotalmajor = mean(ntotal),
            totalgrad = sum(ntotal)) %>%
  select(meantotalmajor, totalgrad, everything())
```

2. Using the data set from the `group_by()` and `n()` combination,

```
babynames %>% group_by(year) %>%
  summarise(ngroup = n())
#> # A tibble: 138 x 2
#>     year ngroup
#>    <dbl>  <int>
#>  1  1880   2000
#>  2  1881   1935
#>  3  1882   2127
#>  4  1883   2084
#>  5  1884   2297
#>  6  1885   2294
#>  7  1886   2392
#>  8  1887   2373
#>  9  1888   2651
#> 10  1889   2590
#> # ... with 128 more rows
```

make a line plot with `ngroup` on the x-axis and `year` on the y-axis. How would you interpret the plot?

3. * Create a data set that has a column for `name` and a column that shows the total number of births for that name across all years and both sexes.

4. * `group_by()` can also be used with other functions, including `mutate()`. Use `group_by()` and `mutate()` to rank the names from most to least popular in each year-sex combination.

5. * From the data set in 4, `filter()` the data to keep only the most popular name in each year-sex combination and then construct a summary table showing how many times each name appears as the most popular name.

6. * Run the following code. Intuitively, a `slice(1, 2, 3, 4, 5)` should grab the first five rows of the data set, but, when we try to run that, we get 1380 rows. Try to figure out what the issue is by using Google to search something like "`dplyr` not slicing correctly after using group by." What do you find?

```
babynames_test <- babynames %>%
  group_by(year, sex) %>% mutate(ntest = n / prop)
babynames_test %>% slice(1, 2, 3, 4, 5)
#> # A tibble: 1,380 x 6
#> # Groups:   year, sex [276]
#>     year sex   name          n   prop   ntest
#>    <dbl> <chr> <chr>     <int>  <dbl>   <dbl>
#>  1  1880 F     Mary       7065 0.0724  97605.
#>  2  1880 F     Anna       2604 0.0267  97605.
#>  3  1880 F     Emma       2003 0.0205  97605.
#>  4  1880 F     Elizabeth  1939 0.0199  97605.
#>  5  1880 F     Minnie     1746 0.0179  97605.
#>  6  1880 M     John       9655 0.0815 118400.
#>  7  1880 M     William    9532 0.0805 118400.
#>  8  1880 M     James      5927 0.0501 118400.
#>  9  1880 M     Charles    5348 0.0452 118400.
#> 10  1880 M     George     5126 0.0433 118400.
#> # ... with 1,370 more rows
```

4.4 Missing Values

Both of the data sets that we've worked with are nice in that they do not have any missing values. We'll see plenty of examples of data sets with missing values later, so we should examine how the various functions that we've talked about so far tackle missing values.

Missing values in `R` are denoted with `NA` for "Not Available." Run the following code to create a toy data set with some missing values so that we can see how the various functions we've used so far deal with `NA` values.

```
toy_df <- tibble(x = c(NA, 3, 4, 7),
                 y = c(1, 4, 3, 2),
                 z = c("A", "A", "B", NA))
toy_df
#> # A tibble: 4 x 3
#>       x     y z    
#>   <dbl> <dbl> <chr>
#> 1    NA     1 A    
#> 2     3     4 A    
#> 3     4     3 B    
#> 4     7     2 <NA>
```

4.4.1 Exercises

Exercises marked with an * indicate that the exercise has a solution at the end of the chapter at 4.6.

1. * `mutate()`. Try to create a new variable with `mutate()` involving `x`. What does `R` do with the missing value?

2. `arrange()`. Try arranging the data set by `x`. What does `R` do with the missing value?

3. `filter()`. Try filtering so that only observations where x is less than 5 are kept. What does `R` do with the missing value?

4. `summarise()`. Try using `summarise()` with a function involving x. What does `R` return?

5. `group_by()` and `summarise()`. To your statement in 4, add a `group_by(z)` statement before your `summarise()`. What does `R` return now?

4.4.2 Removing Missing Values

Missing values should not be removed without carefully examination and a note of what the consequences might be (e.g. why are these values missing?). We have a toy data set that is meaningless, so we aren't asking those questions now, but we will for any data set that does have missing values!

If we have investigated the missing values and are comfortable with removing them, many functions that we would use in `summarise()` have an `na.rm` argument that we can set to `TRUE` to tell `summarise()` to remove any `NA`s before taking the `mean()`, `median()`, `max()`, etc.

```
toy_df %>% summarise(meanx = mean(x, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   meanx
#>   <dbl>
#> 1  4.67
```

If we want to remove the missing values more directly, we can use the `is.na()` function in combination with `filter()`. If the variable is `NA` (Not Available) for an observation, `is.na()` evaluates to `TRUE`; if not, `is.na()` evaluates to `FALSE`. Test this out using `mutate()` to create a new variable for whether `Median` is missing:

```
toy_df %>% mutate(missingx = is.na(x))
#> # A tibble: 4 x 4
#>       x     y z     missingx
#>   <dbl> <dbl> <chr> <lgl>
#> 1    NA     1 A     TRUE
#> 2     3     4 A     FALSE
#> 3     4     3 B     FALSE
#> 4     7     2 <NA>  FALSE
```

`missingx` is `TRUE` only for the the first observation. We can use this to our advantage with `filter()` to filter it out of the data set, without going through the extra step of actually making a new variable `missingx`:

```
toy_df %>% filter(is.na(x) != TRUE)
#> # A tibble: 3 x 3
#>       x     y z
#>   <dbl> <dbl> <chr>
#> 1     3     4 A
#> 2     4     3 B
#> 3     7     2 <NA>
```

You'll commonly see this written as short-hand in people's code you may come across as:

```
toy_df %>% filter(!is.na(x))
#> # A tibble: 3 x 3
#>       x     y z    
#>   <dbl> <dbl> <chr>
#> 1     3     4 A    
#> 2     4     3 B    
#> 3     7     2 <NA>
```

which says to "keep anything that does not have a missing x value" (recall that the `!` means "not").

4.5 Chapter Exercises

1. We found both in the SLU majors data set and in the FiveThirtyEight majors data set that Statistics has a higher proportion of women than almost all other STEM fields. Read the first two sections of this article. Write 2-3 sentences about the article's reasoning of why there are more women in statistics than in other STEM fields.

2. * a. Choose 5 names that interest you and create a new data set that only has data on those 5 names.

b. Use `group_by()` and `summarise()` to add together the number of Females and Males for each name in each year. **Hint**: you can `group_by()` more than one variable!

c. Make a line plot showing the popularity of these 5 names over time.

3. a. Choose a year and a sex that interests you and filter the data set to only contain observations from that year and sex.

b. Create a new variable that ranks the names from most popular to least popular.

c. Create a bar plot that shows the 10 most popular names as well as the count for each name.

4. * In some cases throughout this chapter, we've renamed data sets using `<-` with the same name like

```
toy_df <- toy_df %>% mutate(newvar = x / y)
```

In other cases, we've given the data set a new name, like

```
toy_small <- toy_df %>% filter(!is.na(x))
```

For which of the functions below is a generally "safe" to name the data set using the same name after using the function. Why?

a. `mutate()`

b. `arrange()`

c. `filter()`

d. `summarise()`

e. `select()`

5. Pose a question about the `babynames` data set and then answer your question with either a graphic or a data summary.

4.6 Exercise Solutions

4.6.1 `mutate()` S

2. * Create a new variable that is called `major_size` and is "large" when the total number of majors is 100 or more and "small" when the total number of majors is less than 100.

```
slumajors_df %>% mutate(major_size = if_else(ntotal >= 100,
                                             true = "large",
                                             false = "small"))
#> # A tibble: 27 x 8
#>    Major         nfemales nmales percfemale ntotal morewomen
#>    <chr>            <dbl>  <dbl>      <dbl>  <dbl> <chr>
#>  1 Anthropology        34     15       69.4     49 Yes
#>  2 Art & Art Hi~       65     11       85.5     76 Yes
#>  3 Biochemistry        14     11       56       25 Yes
#>  4 Biology            162     67       70.7    229 Yes
#>  5 Business in ~      135    251       35.0    386 No
#>  6 Chemistry           26     14       65       40 Yes
#>  7 Computer Sci~       21     47       30.9     68 No
#>  8 Conservation~       38     20       65.5     58 Yes
#>  9 Economics          128    349       26.8    477 No
#> 10 English            131     54       70.8    185 Yes
#> # ... with 17 more rows, and 2 more variables:
#> #   large_majority <chr>, major_size <chr>
## OR
slumajors_df %>%
  mutate(major_size = case_when(ntotal >= 100 ~ "large",
                                ntotal < 100 ~ "small"))
#> # A tibble: 27 x 8
#>    Major         nfemales nmales percfemale ntotal morewomen
#>    <chr>            <dbl>  <dbl>      <dbl>  <dbl> <chr>
#>  1 Anthropology        34     15       69.4     49 Yes
#>  2 Art & Art Hi~       65     11        85.5     76 Yes
#>  3 Biochemistry        14     11        56       25 Yes
#>  4 Biology            162     67        70.7    229 Yes
#>  5 Business in ~      135    251        35.0    386 No
#>  6 Chemistry           26     14        65       40 Yes
#>  7 Computer Sci~       21     47        30.9     68 No
#>  8 Conservation~       38     20        65.5     58 Yes
#>  9 Economics          128    349        26.8    477 No
#> 10 English            131     54        70.8    185 Yes
#> # ... with 17 more rows, and 2 more variables:
#> #   large_majority <chr>, major_size <chr>
```

5. * Investigate what happens with `case_when()` when you give overlapping conditions and when you give conditions that don't cover all observations. For overlapping conditions, create a variable `testcase` that is `"Yes"` when `percfemale` is greater than or equal to 40 and `"No"` when `percfemale` is greater than 60 For conditions that don't cover all observations, create a variable `testcase2` that is `"Yes"` when `percefemale` is greater than or equal to 55 and `"No"` when `percfemale` is less than 35.

```
#> # A tibble: 27 x 9
#>    Major         nfemales nmales percfemale ntotal morewomen
#>    <chr>            <dbl>  <dbl>      <dbl>  <dbl> <chr>
#>  1 Anthropology        34     15       69.4     49 Yes
#>  2 Art & Art Hi~       65     11       85.5     76 Yes
#>  3 Biochemistry        14     11       56       25 Yes
#>  4 Biology            162     67       70.7    229 Yes
#>  5 Business in ~      135    251       35.0    386 No
#>  6 Chemistry           26     14       65       40 Yes
#>  7 Computer Sci~       21     47       30.9     68 No
#>  8 Conservation~       38     20       65.5     58 Yes
#>  9 Economics          128    349       26.8    477 No
#> 10 English            131     54       70.8    185 Yes
#> # ... with 17 more rows, and 3 more variables:
#> #   large_majority <chr>, testcase <chr>, testcase2 <chr>
```

For overlapping cases, case_when prioritizes the first case given.

For non-coverage, any observation that is not covered is given an NA.

4.6.2 `arrange()`, `select()`, ... S

2. * Use `select()` and `everything()` to put the `large_majority` variable as the first column in the `slumajors_df` data set.

```
slumajors_df %>% select(large_majority, everything())
#> # A tibble: 27 x 7
#>    large_majority Major    nfemales nmales percfemale ntotal
#>    <chr>          <chr>       <dbl>  <dbl>      <dbl>  <dbl>
#>  1 none           Anthrop~       34     15       69.4     49
#>  2 female         Art & A~       65     11       85.5     76
#>  3 none           Biochem~       14     11       56       25
#>  4 female         Biology       162     67       70.7    229
#>  5 none           Busines~      135    251       35.0    386
#>  6 none           Chemist~       26     14       65       40
#>  7 none           Compute~       21     47       30.9     68
#>  8 none           Conserv~       38     20       65.5     58
#>  9 male           Economi~      128    349       26.8    477
#> 10 female         English       131     54       70.8    185
#> # ... with 17 more rows, and 1 more variable:
#> #   morewomen <chr>
```

3. * In the `babynames` data set, use `filter()`, `mutate()` with `rank()`, and `arrange()` to print the 10 most popular Male babynames in 2017.

```
babynames %>% filter(sex == "M" & year == 2017) %>%
  mutate(rankname = rank(desc(n))) %>%
  filter(rankname <= 10)
#> # A tibble: 10 x 6
#>     year sex   name         n    prop rankname
#>    <dbl> <chr> <chr>    <int>   <dbl>    <dbl>
#>  1  2017 M     Liam     18728 0.00954        1
#>  2  2017 M     Noah     18326 0.00933        2
#>  3  2017 M     William  14904 0.00759        3
#>  4  2017 M     James    14232 0.00725        4
#>  5  2017 M     Logan    13974 0.00712        5
#>  6  2017 M     Benjamin 13733 0.00699        6
#>  7  2017 M     Mason    13502 0.00688        7
#>  8  2017 M     Elijah   13268 0.00676        8
#>  9  2017 M     Oliver   13141 0.00669        9
#> 10  2017 M     Jacob    13106 0.00668       10
```

4.6.3 `summarise()` and `group_by()` S

3. * Create a data set that has a column for `name` and a column that shows the total number of births for that name across all years and both sexes.

```
babynames %>% group_by(name) %>%
  summarise(totalbirths = sum(n))
#> # A tibble: 97,310 x 2
#>    name       totalbirths
#>    <chr>            <int>
#>  1 Aaban              107
#>  2 Aabha               35
#>  3 Aabid               10
#>  4 Aabir                5
#>  5 Aabriella           32
#>  6 Aada                 5
#>  7 Aadam              254
#>  8 Aadan              130
#>  9 Aadarsh            199
#> 10 Aaden             4658
#> # ... with 97,300 more rows
```

4. * `group_by()` can also be used with other functions, including `mutate()`. Use `group_by()` and `mutate()` to rank the names from most to least popular in each year-sex combination.

```
ranked_babynames <- babynames %>% group_by(year, sex) %>%
  mutate(rankname = rank((desc(n))))
```

5. * From the data set in 4, `filter()` the data to keep only the most popular name in each year-sex combination and then construct a summary table showing how many times each name appears as the most popular name.

```
ranked_babynames %>% filter(rankname == 1) %>%
  group_by(name) %>%
  summarise(nappear = n()) %>%
  arrange(desc(nappear))
#> # A tibble: 18 x 2
#>    name     nappear
#>    <chr>      <int>
#>  1 Mary          76
#>  2 John          44
#>  3 Michael       44
#>  4 Robert        17
#>  5 Jennifer      15
#>  6 Jacob         14
#>  7 James         13
#>  8 Emily         12
#>  9 Jessica        9
#> 10 Lisa           8
#> 11 Linda          6
#> 12 Emma           5
#> 13 Noah           4
#> 14 Sophia         3
#> 15 Ashley         2
#> 16 Isabella       2
#> 17 David          1
#> 18 Liam           1
```

6. * Run the following code. Intuitively, a `slice(1, 2, 3, 4, 5)` should grab the first five rows of the data set, but, when we try to run that, we get 1380 rows. Try to figure out what the issue is by using Google to search something like "`dplyr` not slicing correctly after using group by." What do you find?

```
babynames_test <- babynames %>%
  group_by(year, sex) %>% mutate(ntest = n / prop)
babynames_test %>% slice(1, 2, 3, 4, 5)
#> # A tibble: 1,380 x 6
#> # Groups:   year, sex [276]
#>     year sex   name          n   prop   ntest
#>    <dbl> <chr> <chr>     <int>  <dbl>   <dbl>
#>  1  1880 F     Mary       7065 0.0724  97605.
#>  2  1880 F     Anna       2604 0.0267  97605.
#>  3  1880 F     Emma       2003 0.0205  97605.
#>  4  1880 F     Elizabeth  1939 0.0199  97605.
#>  5  1880 F     Minnie     1746 0.0179  97605.
#>  6  1880 M     John       9655 0.0815 118400.
#>  7  1880 M     William    9532 0.0805 118400.
#>  8  1880 M     James      5927 0.0501 118400.
#>  9  1880 M     Charles    5348 0.0452 118400.
#> 10  1880 M     George     5126 0.0433 118400.
#> # ... with 1,370 more rows
```

Functions like `slice()` and `rank()` operate on defined groups in the data set if using a function like `group_by()` first. Sometimes this feature is quite convenient. But, if we no longer want `slice()` or `rank()` or other functions to account for these groups, we need to add an `ungroup()` pipe, which simply drops the groups that we had formed:

```
babynames_test %>% ungroup() %>% slice(1:5)
#> # A tibble: 5 x 6
#>    year sex   name          n   prop  ntest
#>   <dbl> <chr> <chr>     <int>  <dbl>  <dbl>
#> 1  1880 F     Mary       7065 0.0724 97605.
#> 2  1880 F     Anna       2604 0.0267 97605.
#> 3  1880 F     Emma       2003 0.0205 97605.
#> 4  1880 F     Elizabeth  1939 0.0199 97605.
#> 5  1880 F     Minnie     1746 0.0179 97605.
```

4.6.4 Missing Values S

1. * mutate(). Try to create a new variable with mutate() involving x. What does R do with the missing value?

```
toy_df %>% mutate(xy = x * y)
#> # A tibble: 4 x 5
#>       x     y z     newvar    xy
#>   <dbl> <dbl> <chr>  <dbl> <dbl>
#> 1    NA     1 A      NA       NA
#> 2     3     4 A       0.75    12
#> 3     4     3 B       1.33    12
#> 4     7     2 <NA>    3.5     14
```

R puts another NA in place of x times y for the observation with the missing x.

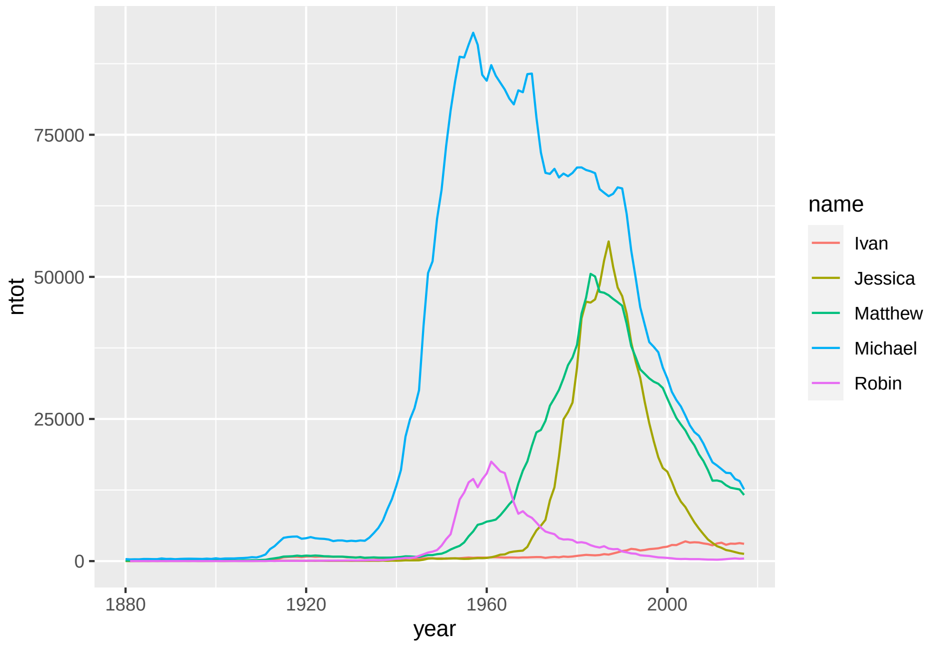
4.6.5 Chapter Exercises S

2. * a. Choose 5 names that interest you and create a new data set that only has data on those 5 names.

   b. Use group_by() and summarise() to add together the number of Females and Males for each name in each year. **Hint**: you can group_by() more than one variable!

   c. Make a line plot showing the popularity of these 5 names over time.

```
baby5 <- babynames %>% filter(name == "Matthew" | name == "Ivan" |
                                name == "Jessica" | name == "Robin" |
                                name == "Michael")
baby5_tot <- baby5 %>% group_by(year, name) %>%
  summarise(ntot = sum(n))
#> `summarise()` has grouped output by 'year'. You can override using the `.groups` argument.
ggplot(data = baby5_tot, aes(x = year, y = ntot, colour = name)) +
  geom_line()
```



4. * In some cases throughout this chapter, we've renamed data sets using `<-` with the same name like

```
toy_df <- toy_df %>% mutate(newvar = x / y)
```

In other cases, we've given the data set a new name, like

```
toy_small <- toy_df %>% filter(!is.na(x))
```

For which of the functions below is a generally "safe" to name the data set using the same name after using the function. Why?

a. `mutate()`

Usually fine: mutating creates a new variable, which doesn't change any of the other variables in the data set, if things get messed up with the new variable.

b. `arrange()`

Usually fine: ordering the rows a certain way won't change any plots and doesn't change any of the underlying data.

c. `filter()`

Usually not the best practice. Naming the data set the same name after the filter means that you permanently lose data that you filtered out, unless you re-read in the data set at the beginning.

d. `summarise()`

Usually not the best practice. Again, naming the summarized data set the same as the original data means that you lose the original data, unless you re-read it in at the beginning. For example,

```
toy_df <- toy_df %>% summarise(meanx = mean(x))
toy_df
#> # A tibble: 1 x 1
#>   meanx
#>   <dbl>
#> 1    NA
```

means that we now have no way to access the original data in `toy_df`.

e. `select()`

This can sometimes be okay if you're sure that the variables you are removing won't ever be used.

4.7 Non-Exercise R Code

```
library(babynames)
head(babynames)
library(tidyverse)
slumajors_df <- read_csv("data/SLU_Majors_15_19.csv")
slumajors_df
slumajors_df %>% mutate(ntotal = nfemales + nmales)
slumajors_df %>%
  mutate(percfemale = 100 * nfemales / (nfemales + nmales))
slumajors_df <- slumajors_df %>%
  mutate(percfemale = 100 * nfemales / (nfemales + nmales))
slumajors_df <- slumajors_df %>% mutate(ntotal = nfemales + nmales)
slumajors_df <- slumajors_df %>%
  mutate(ntotal = nfemales + nmales) %>%
  mutate(percfemale = 100 * nfemales / (nfemales + nmales))
mutate(mutate(slumajors_df, ntotal = nfemales + nmales), percfemale = 100 * nfemales / (nfemales
slumajors_df %>% mutate(morewomen = if_else(percfemale > 50,
                                            true = "Yes",
                                            false = "No"))
slumajors_df %>% mutate(large_majority =
                          case_when(percfemale >= 70 ~ "female",
                                    percfemale <= 30 ~ "male",
                                    percfemale > 30 & percfemale < 70 ~ "none"))
slumajors_df <- slumajors_df %>%
  mutate(morewomen = if_else(percfemale > 50,
                             true = "Yes",
                              false = "No")) %>%
  mutate(large_majority =
           case_when(percfemale >= 70 ~ "female",
                     percfemale <= 30 ~ "male",
                     percfemale > 30 & percfemale < 70 ~ "none"))
slumajors_df %>% arrange(percfemale)
slumajors_df %>% arrange(desc(percfemale))
slumajors_df %>% select(Major, ntotal)
slumajors_df %>% select(-ntotal, -nfemales, -nmales)
slumajors_df %>% mutate(propfemale = percfemale / 100) %>%
  select(propfemale, everything())
slumajors_df %>% arrange(desc(ntotal)) %>%
  slice(1, 2, 3, 4, 5)
library(babynames)
babynames
babynames %>% filter(name == "Matthew")
babynames %>% filter(year >= 2000)
babynames %>% filter(sex != "M")
babynames %>% filter(prop > 0.05)
babynames %>% filter(year == max(year))
babynames %>% filter(n > 20000 | prop > 0.05)
babynames %>% filter(sex == "F" & name == "Mary")
babynames %>% filter(sex == "F" & name == "Mary" & prop > 0.05)
slumajors_df %>%
  summarise(meantotalmajor = mean(ntotal),
            totalgrad = sum(ntotal))
babynames %>% group_by(year) %>%
  summarise(totalbirths = sum(n))
babynames %>% summarise(totalobs = n())
babynames %>% group_by(year) %>%
  summarise(ngroup = n())
toy_df %>% summarise(meanx = mean(x, na.rm = TRUE))
toy_df %>% mutate(missingx = is.na(x))
toy_df %>% filter(is.na(x) != TRUE)
toy_df %>% filter(!is.na(x))
```