Stat 201: Exploratory Data Analysis in R

Part 1

Contents

Learning Objectives

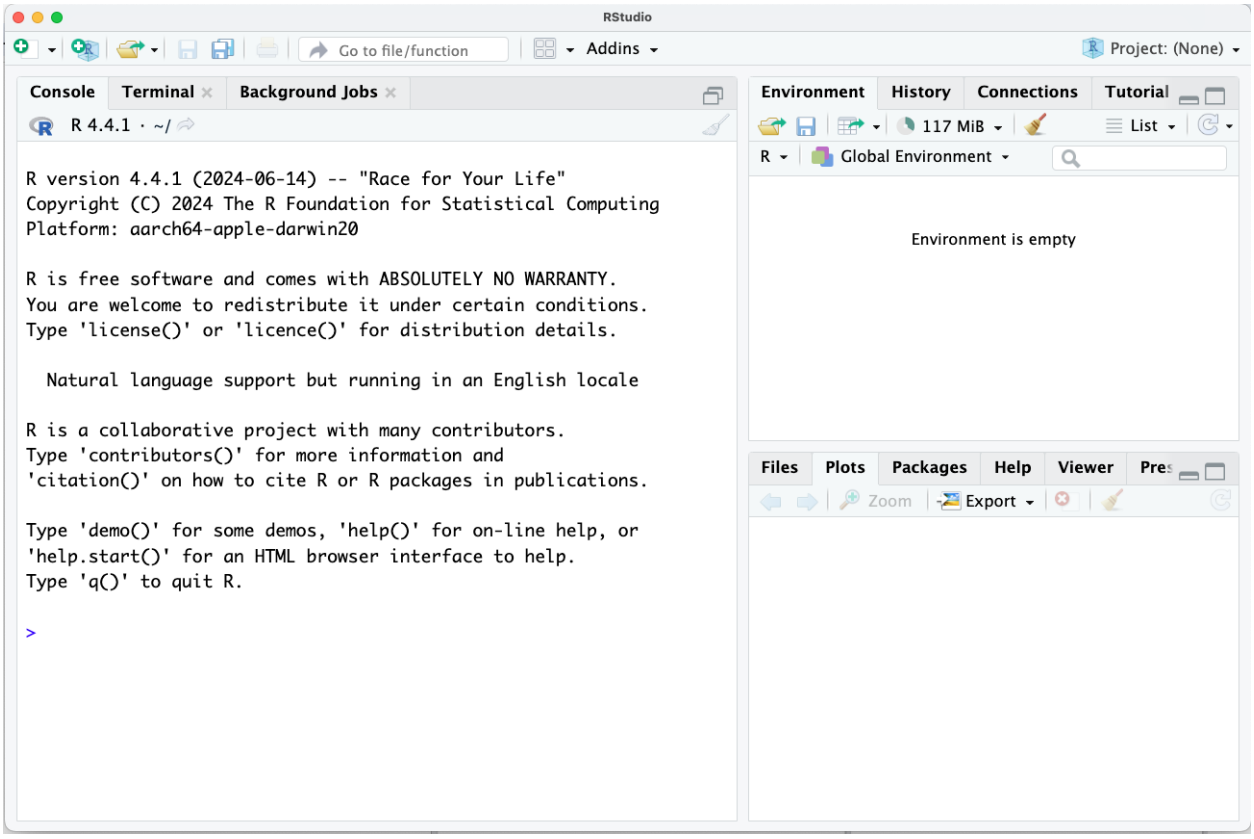
- Become comfortable with the RStudio interface.
- Understand how to record written explanations and code chunks in an RMarkdown file and `knit` the file into a report.
- Run some fun, basic commands in R.
- Load our first dataset into R: use this dataset to review basic concepts about exploratory data analysis.

Why R?

R is an incredibly popular programming language for data analysis. It is used by statisticians but also by scientists in biology, psychology, economics, chemistry, etc. R is fully featured and very powerful, and unlike Stata or SaaS, it is *free and open source*. This means that you can continue to use it for free after you finish this class, and beyond. Additionally, R is *extensible*. When a statistician develops a new method, or when a scientist decides that they need a specific tool to analyze their specific data, they (or someone who wants to use it in R) will often implement the method as a collection of functions in R called a *package*. Other individuals who want to use the method or conduct a similar type of data analysis can download the package and use these functions without needing to rewrite all of the code themselves. The Comprehensive R Archive Network, or **CRAN**, contains over 13,000 contributed packages.

Getting Started

Hopefully you were able to download R and RStudio by following the instructions on GLOW. If so, go ahead and launch RStudio. You should see a window that looks like the image shown below.



The panel on the lower left is where the action happens. It's called the *console*. Every time you launch RStudio, it will have the same text at the top of the console telling you the version of R that you're running. Below that information you will see the symbol >. This is called the *prompt*: it is a request for a command. Initially, interacting with R is all about typing commands into the console and interpreting the output.

The panel in the upper right contains your *environment*, which will show you all of your named variables and datasets once you create/load them. You can also view a *history* of all commands you have previously entered in the console.

Any plots that you generate will show up in the panel in the lower right corner. This is also where you can browse your files, access help, manage packages, etc.

Loading Packages

R is an open-source programming language, meaning that users can contribute packages that make our lives easier, and we can use them for free. For many labs in the future we will use the following R packages:

- The suite of **tidyverse** packages: for data wrangling and data visualization
- **openintro**: for data and custom functions with the OpenIntro resources
- **tinytex**: you may need this in order to knit an RMarkdown document into a PDF. For example, you need this if you have a Mac and have never installed tex.

Hopefully you were able to install these packages using `install.packages()` using the pre-class instructions. Note that you can check to see which packages (and which versions) are installed by inspecting the *Packages* tab in the lower right panel of RStudio. If you do not see `tidyverse`, `openintro`, and `tinytex` in this panel, it means that you need to run the following code from the pre-class instructions.

```
install.packages("tidyverse")
install.packages("openintro")
install.packages("tinytex")
tinytex::install_tinytex()
```

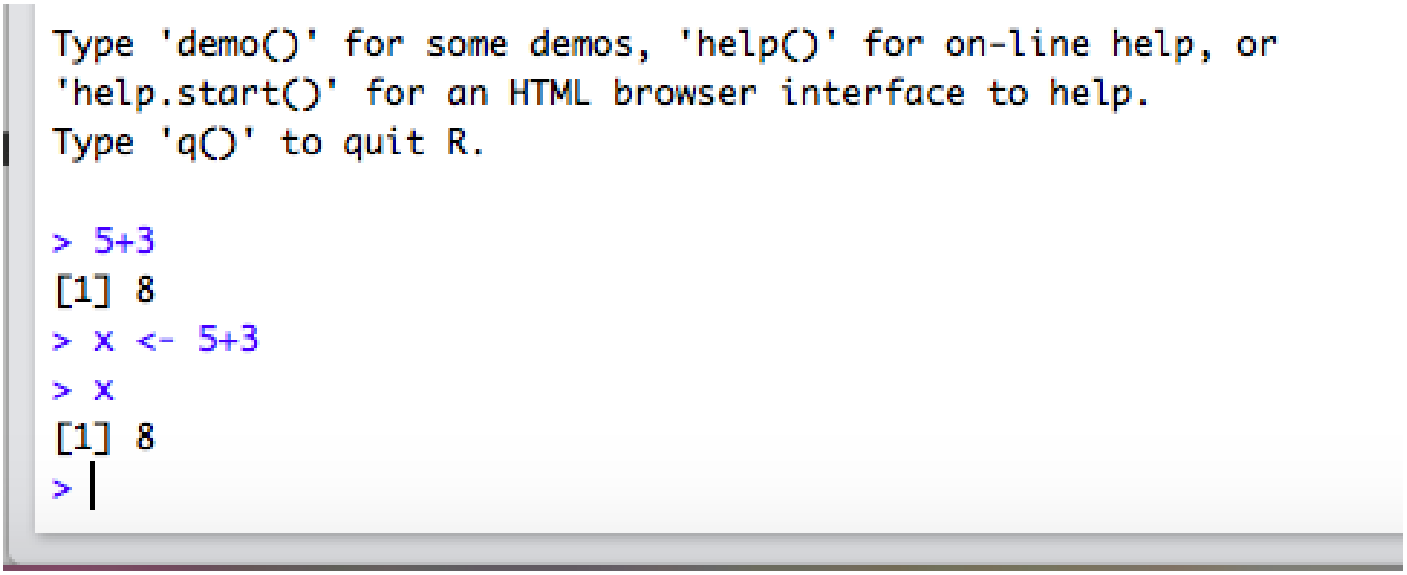
Next, regardless of what you completed before class, you need to load these packages in your working environment. We do this with the `library` function. Note that you need to **load** the packages every time you restart RStudio, whereas you only need to **install** the packages once ever. Note that `tinytex` is a special case package that we do not need to load: we needed it to make sure we had a Tex installation on our computer, which RMarkdown will need in the background in order to knit to PDFs.

```
library(tidyverse)
library(openintro)
```

Interacting with R in the Console

The most basic way to interact with RStudio is to use the console as a calculator. Using the console, you can perform basic arithmetic operations and you can assign values to variables.

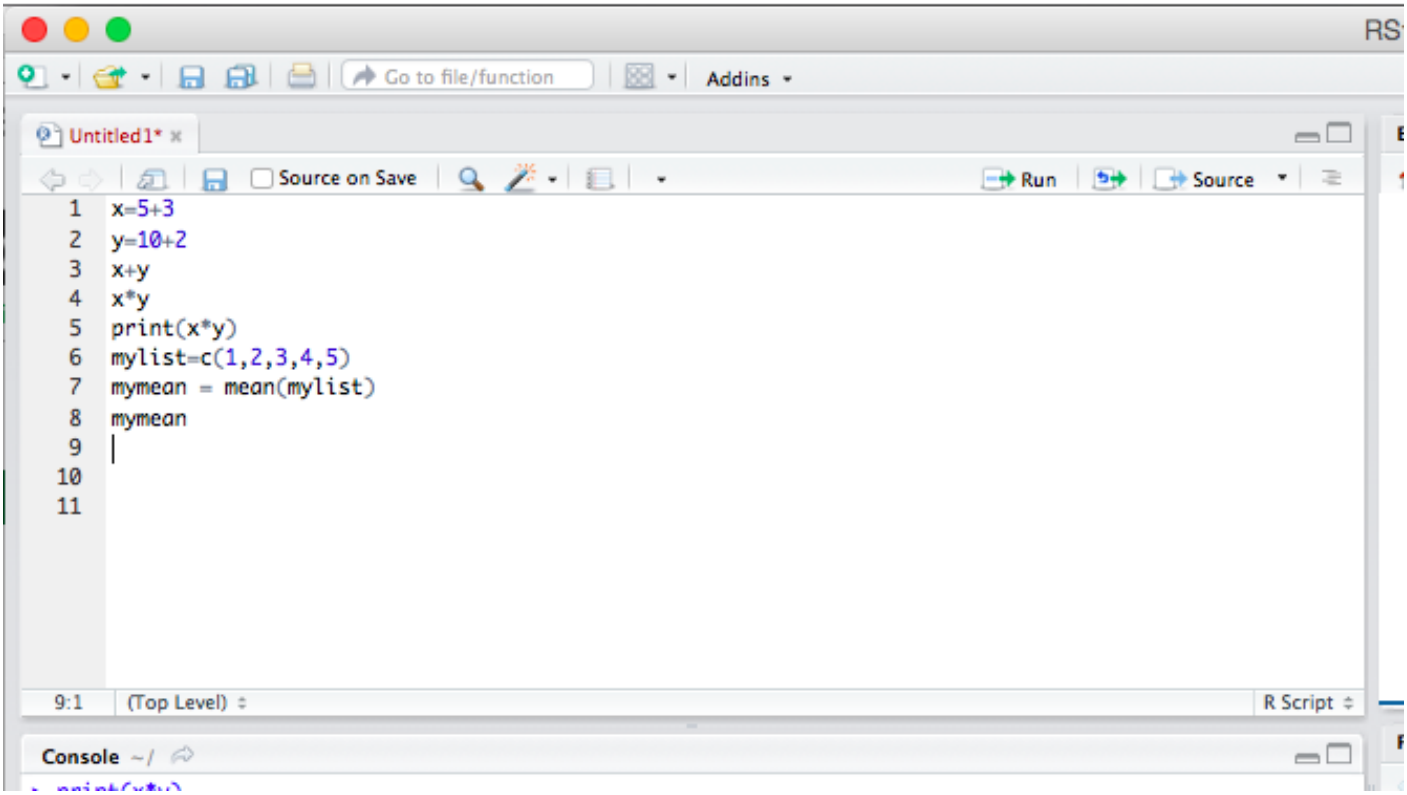
Anything that you type after the > (the prompt) will be executed as R code. For example, type `5+3` after the prompt and then press enter (return). You should see your answer right away. If instead you type `x <- 5+3` and then enter, the console will no longer print out `8`, but it will save your value of `x`. Note that `x` now appears in your *environment*. Now type `x` in the console and hit enter; verify that it prints the appropriate value.



Saving your work

Anything that you could ever want to accomplish in R can be done directly in the console. However, if you are writing commands that you might want to save for later or share with others, it is better to write these inside a file, rather than in the console. For example, if you are working on an assignment, you probably want to be able to access your code again later to check your work.

The most basic type of file is an R Script file. You can create one by going to File, New File, R Scipt. It should show up in the upper left panel of your screen. This file works just like the console, but instead of running a command every time you click enter, you can write several lines at once without running them. If your cursor is on a certain line, clicking "run" in the top right corner will execute just that line. Clicking "source" will execute all lines at once. Experiment by typing code such as the following into your R Script file. Make sure you are comfortable with the difference between running an individual line and running the whole file.



RMarkdown files

RScript files are great for code, but they are not great for communicating your ideas to others. An RMarkdown file lets you intersperse *chunks* of R code with chunks of text and output the result as a nicely formatted document. For all homework assignments in this course, you will use RMarkdown to create documents that intersperse code with answers to written discussion questions. We will typically post the .Rmd file corresponding to the actual homework assignment, and early in the semester you can use these files as *templates*, where all you need to do is fill in your answers. Later in the semester, for your projects, you will need to be a bit more comfortable formatting your own RMarkdown documents.

For today's tutorial, we will make an RMarkdown document from scratch. In your RStudio application, select File, New File, R Markdown. If prompted, select `PDF` as the default output format. Also, if prompted

to install any packages, please do so! You will know that you were successful if you end up seeing an example file that looks like this:

````
---
title: "Untitled"
output: pdf_document
date: "2024-09-09"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word d
on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the out
chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```
````

Anything written in this document on plain white background is interpreted as text (write here as you would write in a Microsoft word document), and any code written on gray background is interpreted as R code (write here as you would write in the R console, with one command per line). We call the gray areas **code chunks**.

Note that, when used on white background, the `#` symbol creates titles and headers that show up in large font in the output document. We use text such as `### Exercise 1` to label the exercises. When used inside of a code chunk, the `#` symbol in R creates a code **comment**. This can be used to write regular text inside of a code chunk. Any text written in a code chunk after the `#` symbol is ignored; it is not run as R code. We use comments to leave you messages inside of lab and HW templates, such as "write your answer here".

Knitting an RMarkdown document means turning the input file (`.Rmd`) into a nicely formatted output file (`.pdf` in this case). To knit your current template document, select `knit` from the buttons at the top of the RMarkdown file. A nicely formatted document will open up in a new window.

Modify your example document so that it looks something like this one:

Source Visual

```
---
title: "Lecture 2"
author: "Anna Neufeld"
output: pdf_document
date: "2024-09-10"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

Getting started

This is my first RMarkdown document. I want to see if I can create a PDF document that contains

```{r}
5+8
```

When I knit this, hopefully my PDF document will contain the printed number 13!

More advanced

Let's try one more chunk~

```{r}
x = 5+7
y = 10-3
x-y
```

What do you see when you knit the PDF document?

Bonus

What happens if you remove the line ``x=5+7`` from the code chunk above? Can you still knit?
```

Running code in RMarkdown files

You will notice that just typing the code in the gray chunk did not produce any sort of answer for us. To see the answer to this math expression, we need to **run** this code.

```
```{r}
5+8
```
```

First note that all code in your chunks is automatically run each time you knit your document. Knit your template document now to try this out! Did it print your answer?

You can knit your R Markdown file every time you write code, but sometimes the knitting process takes a

few minutes and you just want to run a few lines at a time to make sure they are working. To run chunks without knitting, you can use either the Run button on the chunk (green sideways triangle) or you can highlight the code and click Run on the top right corner of the R Markdown editor. While it is good to run individual chunks to test out ideas and explore, we also strongly recommend that you **knit your document** each time you finish an exercise. The knitting process can sometimes lead to errors. Students often run in to trouble if they wait to try knitting their document until right before the assignment deadline. Knitting after each exercise ensures that you catch errors as you go.

When an R Markdown document is knitting, it only has access to the variables that are created inside of the R Markdown document itself. The R Markdown document does not get to use the environment in the upper right hand corner of your screen while it is knitting. The R Markdown document only gets to see the variables that are defined inside of the document.

To illustrate this concept, delete the line `x <- 5+7` from your sample document. Can you still run the chunk locally? What happens if you try to knit your document?

```
y <- 10-3
x-y
```

It is also important in an R Markdown document that all variables are defined in order. You will notice that exercise 5 contains two code chunks in your template. What happens if you add `x <- 5+7` back into your sample document, but you add it to its own code chunk. Can you knit the document now?

Loading our first dataset

While using R as a calculator was a fun warmup, in this class we are typically going to use R to analyze datasets. With this in mind, we need to learn how to load our dataset into R. There are several ways that we will do this in this class.

- Many datasets are built into packages. These are extremely easy to load. As long as you have the package installed, you just load the package and then access the relevant dataset with the `data()` function. This is what you will be doing on HW1! These are typically datasets that are meant for educational purposes.
- We can also load data in from an Excel file or a .csv file. This is probably the way that many of you will load your data for your final projects. You can do this using the `import dataset` button, which you will find in the top right corner of your RStudio interface.
- Today, we will load our dataset directly from a URL. I sometimes post the dataset on my github website for you to read directly, because this is easier and quicker than uploading/downloading files. But this is sort of contrived because I post it in a really special format: you won't typically be able to do this in the real world.
- Sometimes (later in the semester) we will generate our own data!

At the start of the semester, we will always give you very clear instructions on how to load the required dataset. Eventually, for your projects, you will need to know how to troubleshoot and load the data yourself!

Over the weekend, most of you filled out the "getting to know you" survey. Google forms lets me export the survey responses into a google sheet / Excel spreadsheet. I removed the names, pronouns, and email addresses from the spreadsheet, and I also removed the answers to the "free response" questions (e.g. "Why are you taking this class"). I changed the column names for readability. Finally, I saved the resulting spreadsheet as a `.csv`.

Data notes here.

I saved the anonymized data as a `.csv` file. I posted it both on GLOW and on my github website. There are two ways that you can load the data. The easiest is to run this code:

```
form_data <- read.csv("https://anna-neufeld.github.io/tutorials/201_fall25/form_data_anonymous.csv")
NROW(form_data)
```

The slightly more cumbersome (but useful!) way involves downloading the .csv document from GLOW and importing it using the `import dataset` button. We will go over this later in the semester.

Copy the code above into your RMarkdown document. Run the code, such that you have an object called `form_data` in your environment? What happens if you click on this dataset?

Cleaning our first dataset

Goals:

- What are the basics of a dataset?
- What "problems" should we look for in a dataset?
- Extract a single variable from a dataset.

Click on `form_data` in your *environment*. You should see something like this:

	pets	sleep_semester	sleep_school	cups.coffee	height	sports	music	favorite.coffee	favorite.dining
1	2	5	12	0.0	6.9	No	No	N/A	Driscoll
2	2	8	8	2.0	70	No	Yes	Goodrich	Driscoll
3	3	7	8	0.0	72	Yes	No	Tunnel	Driscoll
4	1	7	9	2.0	68	Yes	Yes	Tunnel	Paresky
5	0	8	6	0.0	61	Yes	Yes	Spring street market iced vanilla latte with a shot of c...	Paresky
6	0	8	8	1.0	60 inches	No	No	mocha	Paresky
7	3	7	9	0.0	68	Yes	No	Tunnel City	Driscoll
8	2	7	8	3.0	66.9	No	No	Late, Tunnel City	Driscoll
9	0	7	8	5.0	64.6	No	Yes	French vanilla	Driscoll
10	1	6	9	10.0	63	Yes	No	Tunnel City Coffee	Paresky
11	1	9	10	7.0	63	No	No	Five corners!	Driscoll

With a neighbor, discuss the following questions:

- What are the **observational units** in this dataset. How many are there?
- What are some examples of **quantitative variables** in this dataset?
- What are some examples of **categorical variables** in this dataset?
  - Bonus: are any of these **ordinal variables**? An ordinal variable is a categorical variable where the categories have a natural ordering.
- Do you see any issues with the data?

Let's extract the variable `pets` from our dataset. There are a few ways to do this. See if you can figure out any differences between these.

```
form_data$pets
form_data[,4]
form_data %>% select(pets)
form_data %>% pull(pets)
```

Note that R has correctly identified pets as a numerical variable. This means that code such as this will run:

```
form_data$pets+5
```

Can you tell what that code did?

On the other hand, try checking out the height variable in the dataset. Is R treating height as a numerical variable? If not, what went wrong?

```
form_data$height
```

The fact that not everyone entered their height as a number on the survey is an example of a data quality issue. In order to make appropriate plots or summaries of the variable height, we will need to *clean* our dataset. We will hold off on this until Thursday's class. But know that the first step of a data analysis basically always involves diagnosing data quality issues and then cleaning the data: we just got lucky in this case that pets looks pretty straightforward.

Explore and describe a single quantitative variable

Datasets are usually too big to understand just by looking at the raw data. We need to go one variable at a time and produce numerical and visual summaries. Luckily, R makes this all pretty straightforward. We will start by covering this process for numerical variables.

Goals:

- Make a histogram; describe what you see.
- Make a boxplot; describe what you see.
- Have R compute measures of center and spread (mean, median, sd, IQR, etc.)

Let's use pets, which stores the recorded answers to the question "How many pets have you had or have you lived with in your life?".

Make a histogram

Let's first learn how to make a histogram in R. There are actually two ways. The first may look simpler, but I would like to encourage you all to use the second version, which comes from the tidyverse package. While at first ggplot may seem unnecessarily complex, the tidyverse package has become the standard for data cleaning and data visualization. Once you get used to ggplot(), it will let you make very complex and very beautiful, customizable plots. I use ggplot() for all of my research papers!

```
hist(form_data$pets)
ggplot(data=form_data, aes(x=pets))+geom_histogram()
```

Play around with the code below. Is there a histogram that you like best?

```
ggplot(data=form_data, aes(x=pets))+geom_histogram(col="red", fill="pink", binwidth=1)+theme_bw()+
  xlab("Pets")+ylab("Frequency")+ggtitle("My favorite histogram")
```

Make a boxplot

Once again, we have a "base R" version and a `tidyverse` (`ggplot`) version. I highly recommend the `ggplot()` version: once we start comparing data across groups, it will be super useful.

```
boxplot(form_data$pets)
ggplot(data=form_data, aes(y=pets))+geom_boxplot()
```

Challenge: Using what you learned in the histogram section, can you figure out how to add axes labels and a title to your `ggplot` boxplot?

Hint: It really isn't that challenging, because it looks exactly the same as it did for histograms.

Report numerical summaries

Everything that we discussed in lecture is easily computed in R. Usually, the name of the relevant `R` function is just the english name of the statistic we want to compute: how lucky are we!

The following three lines of code all report the same number. Which do you prefer?

```
mean(form_data$pets)
form_data %>% summarize(mean(pets))
form_data %>% pull(pets) %>% mean()
```

Try running the same lines of code as above, but change the word `mean()` to the word `median()`. What happens? Explain how the mean and the median relate for this dataset.

```
median(form_data$pets)
form_data %>% summarize(median(pets))
form_data %>% pull(pets) %>% median()
```

One of my favorite things about the `summarize()` function is that we can report multiple summary statistics at a time. We can also stratify our summaries into groups, which is really helpful. Check out the following lines of code.

```
form_data %>% summarize(mean(pets), median(pets), IQR(pets), sd(pets))
form_data %>% group_by(sports) %>% summarize(mean(pets), median(pets), IQR(pets), sd(pets))
form_data %>% filter(sports=="Yes") %>% summarize(mean(pets), median(pets), IQR(pets), sd(pets))
```

Explore and describe a single categorical variable

Suppose that we want to explore the variable `favorite.dining`. This is a categorical variable.

Summarizing a single categorical variable is sort of boring! We don't have measures of center and spread. The main way that we numerically explore a single categorical variable is by making a table. Try out the following lines of code, which all give the same information. What did you learn about this variable?

```
form_data %>% group_by(favorite.dining) %>% summarize(n())
table(form_data$favorite.dining)
form_data %>% select(favorite.dining) %>% table()
```

If we wanted to visually explore a single categorical variable, we would typically use a bar chart. This displays the same information as the table.

```
ggplot(data=form_data, aes(x=favorite.dining))+geom_bar()
```

Again, a nice thing about learning `ggplot()` is that we can use exactly the same methods as before to make this plot more beautiful- we can just use the plus sign to add on aesthetic features.

```
ggplot(data=form_data, aes(x=favorite.dining, fill="pink"))+geom_bar()+xlab("Favorite Dining Hall")+yla
```

Wrap up

If you were lost in class today, please read through this tutorial again on your own time. Actually run each line of code in the tutorial, and try to think about what it is doing.

Hopefully, when you work on HW1, you will find that I went over all of the R functions that you need for the homework in class (some are coming on Thursday). It is my goal that this will be the case for every homework throughout the semester!

If you are already experienced in programming, I would encourage you to go above-and-beyond on the homework assignments in terms of making your code and your plots beautiful. If you are new to programming, please let me know if the pacing seems too fast for you! I am happy to provide additional resources for learning R if needed!

Finally, when it comes to debugging, the internet is your friend! Learning to use google, stack overflow, or ChatGPT to resolve R errors on your own is a key skill. Please do not stare at a piece of code for 8 hours waiting for a TA session because you cannot find the source of an error: learn to google your error messages and problem solve for yourself!

Acknowledgements

The formatting and some of the introductory content in this tutorial was adopted from an OpenIntro lab.