R for Graduate Students

Y. Wendy Huynh

2019-08-06

Contents

Chapter 1

Preface

Meet the Author

Hello! My name is Wendy Huynh and I am a current PhD student working in the behavioral neurosciences. I began my R journey at the end of my first year of graduate school, slowly and painfully piecing together code. Although programming was never *really* part of my program, I now see it as an integral part of my work.

Many fellow graduate students expressed interest in learning R, but didn't know where to begin. Programming with R is still relatively niche among my cohort and there are very few formal classes teaching this subject.

Although there are many amazing guides/textbooks for R out there, very few of them featured examples relevant for my specific needs **and** were user-friendly enough for a *true* beginner. In the Fall of my second year, I began teaching a new graduate student in my lab everything I knew about R. However, I quickly found that teaching R – even just to one person – was very time consuming. I decided to write up assignments as a "short" guide to R. After writing a short 11 page "first assignment" and receiving positive feedback, I began writing up a second assignment. Then a third. Soon enough, I had written enough pages that I couldn't deny that this "short guide" had turned into a book. To this day, I am continually learning new techniques for managing my data in R and teaching what I know to anyone who wants to learn. I strongly believe that learning efficient techniques in handling data is crucial in graduate school. Teaching R is something I genuinely enjoy doing and I hope you find this book to be useful on your R journey.

Acknowledgements

I would like to thank my colleague and friend, Andrea (Andi), for convincing me to write this book and providing lots of guidance for the kinds of content to include. I'd also like to thank my partner Matt for his endless support and patience, as well as my mentors, Scott and Rick, both of whom nudged me to learn R in the first place. I feel so fortunate and privileged to have the amazing support system and mentorship through my journey in graduate school and sincerely hope that everyone experiences this excitement and gratitude in their lives.

Packages Utilized

There are so many amazing resources and people who paved the way for me. Without these brilliant people, I would still be wrangling my data using slow, error-prone techniques. Here is a list of authors who developed the packages used in this book (in order of package name): Xie (2015), Singmann et al. (2019), R Core Team (2019), Xie (2019a), Wickham et al. (2019b), Lenth (2019), Wickham (2019a), Wickham et al. (2019a), Arnold (2019), Zhu (2019), Xie (2019b), Bates et al. (2019), Bates and Maechler (2019), Henry and Wickham (2019), Wickham et al. (2018), Wickham and Bryan (2019), Allaire et al. (2019), Wickham (2019b), Müller and Wickham (2019), Wickham and Henry (2019), Wickham (2017), and Ooms (2018).

Part I

Let's Begin

Chapter 2

Introduction to R

My intention for this book is to provide another resource for learning R. **You don't need any programming experience for this book, just a computer.** I hope that many find this perspective useful, whether it's by reading it from start to finish or by grabbing a quick line of code. If I've saved anyone from spending hours/days/months trying to do something in R, then this was worth it.

2.1 How to Use This Guide

This guide is intended for the reader to follow along step by step in **chronological** order. As you read, make sure to practice your R skills by completing the exercises *and* examples in the book. While some code is already written out in the text, **typing out the code manually** (rather than copy and pasting from the book) will help speed up the learning process – practice makes progress!

All of the exercises and examples should be accessible for everyone unless package content has changed. If this is the case, I will do my best to create new and updated examples.

Most of my stylistic formatting conventions should be fairly accessible, but I find that it's best to be as explicitly detailed as possible:

- Important details will occasionally be **bolded**, *italicized*, or ***both***
- Keyboard shortcut commands are in **bold** (e.g., **Ctrl + S**)
- Package names are in **bold** (e.g., **tidyverse**)
- Any in-line code (i.e., code text that is interspersed within regular text) is enclosed in backticks (`` `mean()` ``) or is distinguished by different font (e.g., `mean()`)
- Examples of code will be arranged as a code chunk. This is distinguished by a slightly greyed out background behind the code text. Code will be color coordinated by code type (e.g., function, symbol, comment, character string, etc.).

```
object <- 1 + 2 + 3  # this is some code
mean(object)         # this is more code
object %>% mean()    # another line of code
```

- The output of executed code will occasionally be displayed in the book. An output is the consequence that occurs after code is executed/run. Outputs will **only** contain text that begins with **##**. In the example below the top chunk is the code that is executed and the bottom chunk is the output.

```
object <- 1 + 2 + 3  # this is some code
mean(object)         # this is more code
```

```
## [1] 6
```

```
object %>% mean()    # another line of code
```

```
## [1] 6
```

- Throughout the book, there are active links that may refer to another section of the book or a figure/table. Clicking on these links will automatically shift your screen to the linked page. For example, click 1 to go back to the preface.
- Examples and exercises throughout the book are intended for the reader to follow along. This means that the code is manually typed and executed into RStudio by the user. Occasionally, examples/exercises will build off of previous examples and will not always be explicitly stated. It is safe to assume that exercises will only build from previous tasks from the same section.

2.2 The Utility of Learning R

If you're reading this guide, you probably already find some value in learning R. What you may not know is the extent to which learning R is going to benefit your life. I really began learning R at the end of my first year of graduate school. The learning curve was steep, and I took breaks, but the grass is truly greener now that I have mastered some of R's secrets.

Before learning R, it would take me 2 hours each day to manage my data in the best case scenario. In a worst case scenario, I would have spent *the entire day* looking at my data and fixing mistakes from cutting and pasting. **None of this is accounting for the cost of accessing any of the proprietary software** (Did I mention R is FREE?)

Today, everything moves much faster. The initial prep will take a few hours, depending on my data. This prep involves setting up the code, specifically planning out the organizational, graphing, statistical, and error-checking aspects of my data. After this single-day prep, every data update takes about **2 minutes** if I don't need to tweak my prep. How? First, I must turn on my computer. This takes about 30 seconds. Next, I have to open my R files, which takes about another 30 seconds if I'm being a bit generous. Finally, I hit **Control + A** followed by **Control + Enter** and all of my code runs in 1 minute (usually less). Ta-da! In 2 minutes, I have my data formatted and saved as an Excel file (yes, you can create files through R!), I have my graphs saved in a pdf, and I have my statistics saved in an Excel file. Two. Minutes. Do I have your attention now?

I won't lie, it will take some time to get to 2 minutes. For me, it took 1 year. Learning any skill from scratch is going to be challenging *at first*. R is no exception. You must consistently practice R in order to retain any information. The intended purpose of this guide is to provide a unified source for complete beginners to start this process. It won't be easy, but I firmly believe it'll save you time and money.

2.3 Advice to Beginners

You should:

- **Be patient**. Learning a new skill can be frustrating. Like those before me, I have gone through the various stages of grief while looking blankly at error message on my screen. My best advice is to put your expensive computer down and clear your head (I recommend rock climbing).
- **Know that the hardest part is the beginning**. Learning R is hard, but it's not impossible. All it takes is practice and I can assure you that it will get easier. Be the envy of your peers: learn to update your data/graphs in a couple minutes rather than hours.

- **Learn to manage your data in addition to graphing it**. Most people want to get right into graphing their data, but if the data is not already fully prepared, you will actually spend most of your time organizing/structuring your data. Graphing, by comparison, is far easier to learn.

Think of learning R as learning a new language. At first, you learn the basic vocabulary and master these individual words. Then, you master sentences and grammar, then paragraphs, and so on. Learning R will take some time, but everybody starts somewhere.

My best advice is to write down a list of what you need R to do. What outcome do you need? From there, you can fill in the various steps that you might need along the way. Let's see an example.

In this example, we're going to talk about a dataset called `diamonds`. This dataset contains information about – you guessed it – diamonds. Someone went out and collected a lot of information about diamonds and we're going to use this example dataset to explain some of the concepts in this guide. Here's a quick summary about diamonds if you don't know much about diamonds:

- Some attributes of diamonds include the: cut, clarity, color, and carat. Most people are also familiar with diamond shapes (round, square, pear, etc.), but this dataset only contains *round* diamonds.
- Cut refers to how well light shines through the diamond. The better cut, the sparklier the diamond looks. This is more about how well the jeweler cuts the diamond.
- Clarity refers to the cloudiness of a diamond. This is about the diamond's natural features. A jeweler can't change number of blemishes a diamond has naturally.
- Color doesn't really mean color of the diamond per se (i.e., a pink diamond vs. blue diamond). It actually measures how "colorless" the diamond is. Diamonds can be clear (colorless), white, or yellow-tinted (the actual values for color are measured from J to D). The more colorless a diamond is, the more expensive.
- Carat refers to the weight/heaviness of a diamond.

I encourage everyone to Google these basic qualities of a diamond, because I will be using the `diamonds` dataset frequently throughout the book.

Now that we know about diamond features, here is an example of the coding process:

Steps

Example

1. Have a desired outcome
    - I want to graph the mean (average) price for each clarity category in the `diamonds` dataset
    2. List the details
        - The y-axis will depict the mean diamond price.
    - The x-axis will plot each diamond color category.
    - I also want error bars that represent the standard error of the mean.
    - I also want it to be a line graph.
    - I want all of the data points to be blue
    3. Solve one problem at a time
        - How do you calculate the mean price of the diamonds data? I need the average price for each clarity of diamond, not just the average price of all diamonds available. What function will I use? How do I use the `mean()` function?
    - How do I calculate the standard error of the mean for each diamond clarity? Is there a function I can use?

- My data needs to show at least 3 columns. The first column must contain the diamond clarity categories. The second column must contain the mean price for each clarity. The third column must contain the standard error for each clarity price.
- Now that I have my data ready, I can start graphing. How do I create a line graph? How do I add data points, error bars, and change the data points blue?

Color

Mean

Standard Error

D

3170

40.8

E

3077

33.8

F

3725

38.7

G

3999

38.1

H

4487

46.3

I

5092

64.1

J

5324

83.8

It is much easier to navigate R once you've got an itemized list of tasks. Throughout this book, we will go through many different examples of how to wrangle data. Once you've got these skills, you can apply them to your own data!

Many new programmers are concerned that they aren't learning the code inside and out – that they're simply "copying" code from examples. Learning to code is repetitively typing the same code (usually taken from some example found online) until the code becomes so familiar that you've memorized it and when to use it! Certainly, copying the exact code from examples will not get you far on its own. You must physically type it out yourself and troubleshoot it through trial and error. When I learn new code, I usually ask myself a few questions: "What happens when you do this? What happens when you delete that? Does it require this section, or is it optional?" You **aren't inventing the wheel** here (i.e., wheel = coding) as a beginner. Maybe when you've become fluent in coding, you can invent new code. Right now, we'll be focusing on

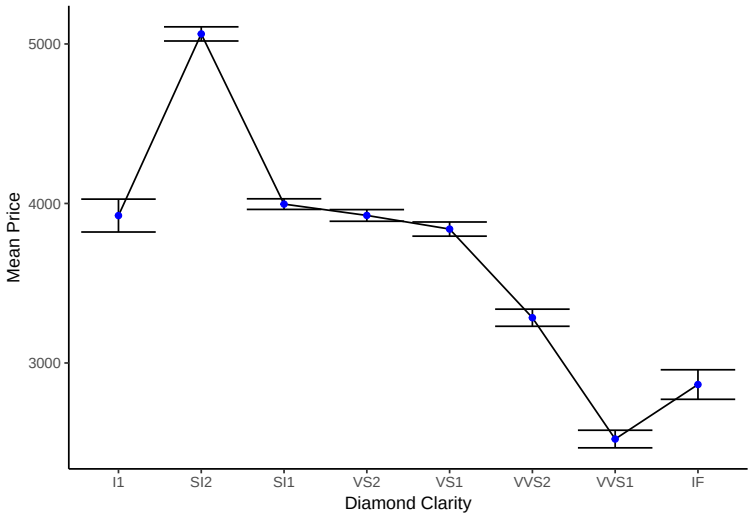


Figure 2.1: Visualizing the data and graph needed for the example task above

what's already available to us. **Just like learning a new language, all you need repetition and trial and error.** Be patient!

I hope that this guide helps you manage your data efficiently and confidently, showing you that coding can be fun, useful, and doable.

2.4 Other Resources

Everyone learns in different ways. This book may not fulfill all of the needs of every user. If you are in need of a different perspective or additional beginner-friendly help, I do highly recommend **R for Data Science** by Hadley Wickham (https://r4ds.had.co.nz/) and **ModernDive: An Introduction to Statistical and Data Sciences via R** by Chester Ismay and Albert Y. Kim (https://moderndive.com/). These are exceptional online resources that are also free!

This book is **not** a resource for:

- Learning statistics. We will go over *how to perform* basic statistical analyses (correlations, chi square, ANOVAs), but it is by no means a tool for learning stats.
- Learning functions. This is what I consider an advanced topic and will not be covered in this text.
- Base R programming. We will briefly discuss and utilize some base R programming, but the majority of the book will teach R using the tidyverse syntax.

Chapter 3

Getting Started

Now that we've gotten introductions out of the way, let's set up our tools. This chapter will address how to download and set up R and R Studio – the two applications you need to code in R. We will also talk about vocabulary, conventions ("good practices"), saving your work, and troubleshooting your code. In future chapters, I will often refer back to sections of this chapter to refresh these foundations – especially the troubleshooting section. Let's get started.

3.1 What to Download

This section instructs new R users on how to download R Statistics and Rstudio. These are the two required applications for this book. Both are free to download and use! The current supported operating systems include Windows, Mac, and Linux. If you are using a Google Chromebook, there is an option to enable Linux: http://blog.sellorm.com/2018/12/20/installing-r-and-rstudio-on-a-chromebook/

3.1.1 R Statistics

In the upcoming pages, I will delve into vocabulary terms. For now, follow these instructions to download R Statistics and RStudio onto your computer.

Step 1: Select your CRAN mirror

To download R, you must use a CRAN (Comprehensive R Archive Network) mirror. For the sake of simplicity, I won't talk about the intricacies of CRAN. All you need to do is select the link that closely matches your current location. https://cran.r-project.org/mirrors.html

Currently, I am located in Lincoln, Nebraska. It doesn't really matter which CRAN mirror you select, but I selected one at the University of Kansas. The download will proceed faster if you choose a nearby location.

Step 2: Download R Statistics for your computer

Selecting your CRAN mirror should bring you to the "Download and Install R" page. Click on the link that matches the operating system (OS) you are currently using such as Linux, Macintosh, or Microsoft Windows. For Macs, you may also need to download an additional application (xQuartz); this will be specified to you on the download page.

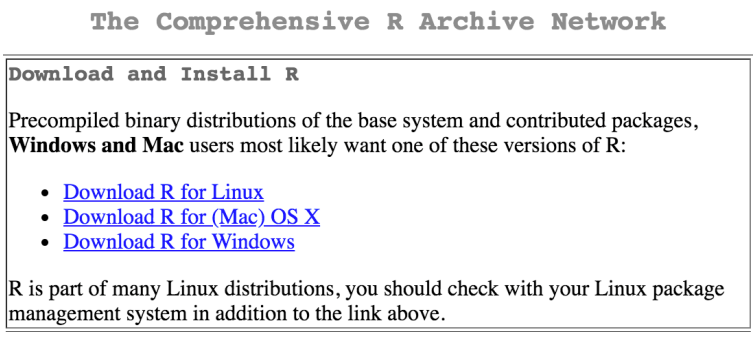


Figure 3.1: Downloading R.

Figure 3.2: Selecting the CRAN mirror.

3.1.2 RStudio

The R Statistics application **installs the R language onto our computer** and is required for RStudio to function; however we will **not** be using the R Statistics program for our R coding. Although we can technically code in R using the default R program, RStudio is preferred.

Compare Notepad (TextEdit for Mac users) to Microsoft Word. Although they both can perform the same basic task, Microsoft Word provides a host of tools (e.g., highlighting, font changes, insert columns/tables, references, headers, etc.) that are not available in Notepad. Similarly, **RStudio is more user-friendly and more powerful than the default R program.**

Download "RStudio Desktop": https://www.rstudio.com/products/rstudio/download/

3.1.2.1 Orienting Yourself to RStudio

First thing's first: **Let's make RStudio look cool.**

To change the theme of RStudio, select **Tools > Global Options > Appearance**

I currently use **"Pastel on Dark"** as my editor theme, but the choice is yours:

Although most of your code will be in one color, RStudio will color code specific commands. This makes it easy for the user to navigate and recognize these commands and is especially useful for error checking. For example, you will notice that RStudio has selectively color coded comments. Comments are personal notes that the user can create by typing `#` before their text; R will not treat comments as executable code. We will get more practice with comments in the upcoming sections.

Next, let's orient ourselves with how RStudio is set up. The standard set-up has four panes (the script pane will be discussed further soon). In all of my examples, my panes are arranged in this default format, but you can choose to rearrange the panes on your own.

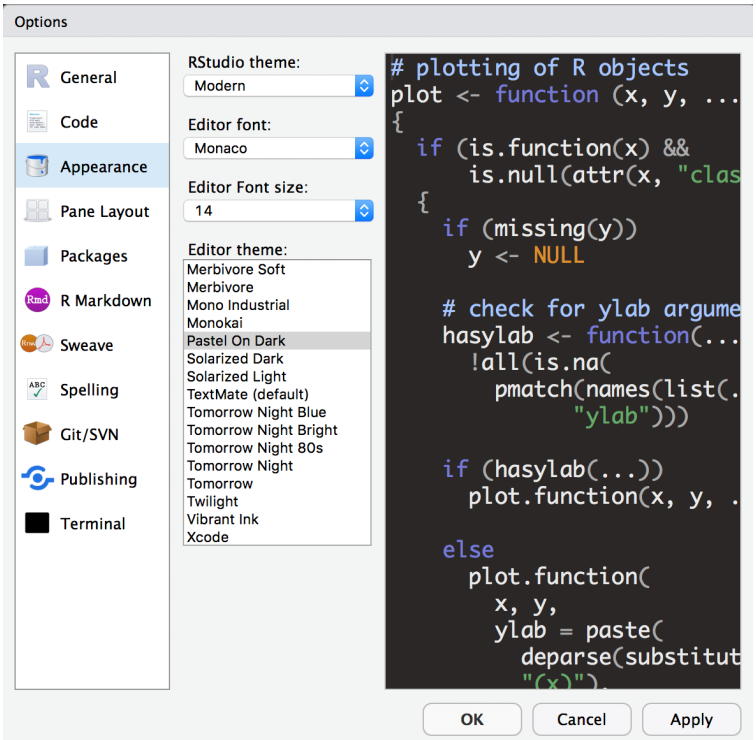


Figure 3.3: Altering the RStudio theme.

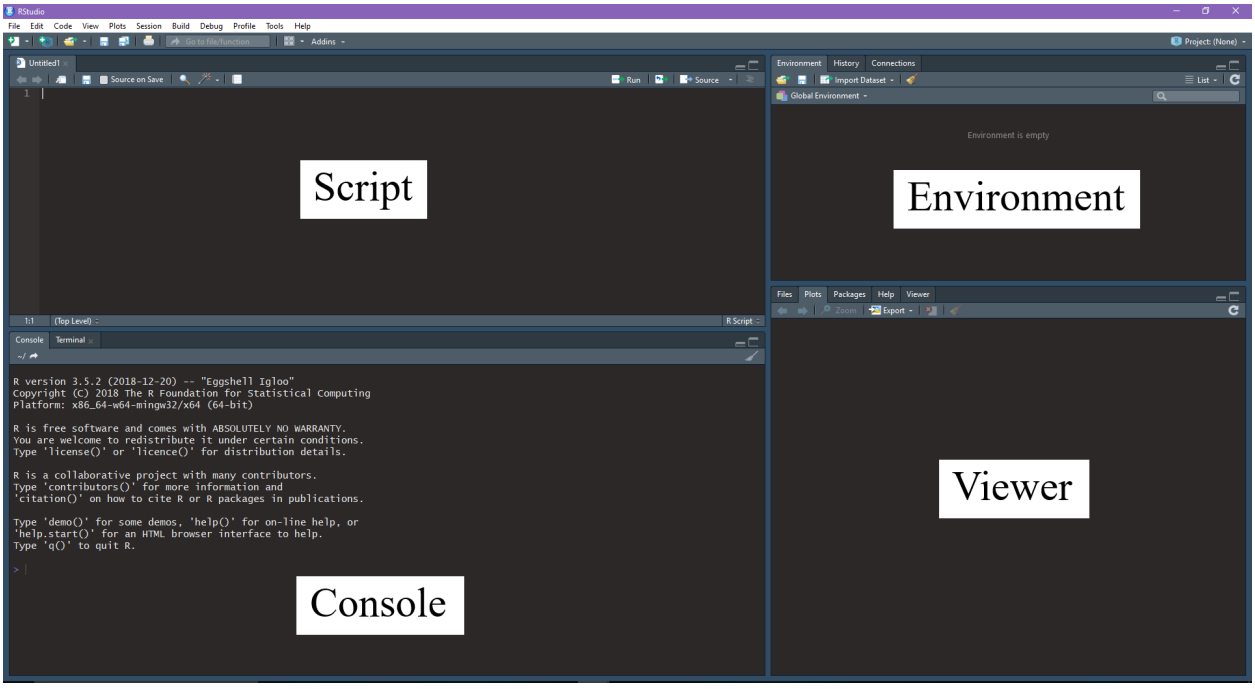


Figure 3.4: A look at RStudio.

3.2 RStudio Set-Up

Now that we have the proper applications downloaded to our computer, we can begin setting up our work space. As data scientists, we must keep our files organized. Ideally, your files are already organized in folders and subfolders. If not, you'll find yourself adopting this practice soon enough (at least when dealing with R).

3.2.1 Creating a New Project

What is a Project?

A Project is essentially a folder on your computer. Like any other folder, a Project folder organizes files that you deem are related in one location. This location is called a **working directory (WD)**. An example of a WD might look like this:

> "/Users/Wendy/Documents/R Files"

Translated, the WD above tells us that we are *currently working in* the folder named: "R Files". The rest of the information gives us information about the **file path** to the current WD. The "R Files" folder is located within the "Documents" folder. The "Documents" folder is located within a folder called "wendy", which is located inside a folder called "Users". What's the difference between a working directory and a file path? All working directories are file paths, but not all file paths are working directory. A file path is an umbrella term that simply refers to a file location (all files have a file path just like every location on earth has a longitude and lattitude). A working directory simply specifies the file path we want to use ***now***.

Like other folders, you should think about what you want your Project to contain. For example, I have different R Projects for different purposes. One R project contains all of the files (Word, Excel, R files, etc.) for my experimental study about nicotine reward. One R project contains files from my Summer 2019 R course. Another project contains files from my Intro Statistics course. You get the point. For now, let's create an R project specifically for working on this book.

How to Create a New Project:

Upon opening RStudio (no need to open regular R Statistics), your screen should look like Figure 3.4.

You'll notice that the console tells you which R version you currently have. By the time you read this guide, the version will have updated far past **3.5.2 (2018-12-28) – "Eggshell Igloo"**, but that is my current R Statistics version.

It is important to periodically check R's website to make sure that you update R Statistics to the latest version. You don't need to download every R Statistics update, but there is a chance that some parts of your code will stop working if you wait too long. To update your R version, you will need to only download the new R Statistics, you do not have to download RStudio again.

Briefly, let's check where our current working directory is located by typing `getwd()` in the console.

In Figure 3.5, we see that R is currently working and pulling information from my "Documents" folder. That means that when I save files, R will automatically save them to this folder. It also means that when I import files, they must be located within this folder. Check out the **Files** tab located on the lower right panel of R Studio. You'll see all of the files/folders that are located in your current WD. When we create our new Project, you'll see that the current WD will change.

Step 1:

At the top right corner of your screen, click on **Project: (None)** followed by **New Project...**. See Figure 3.6.

Step 2:

Click on **New Directory > New Project**. See Figure 3.7

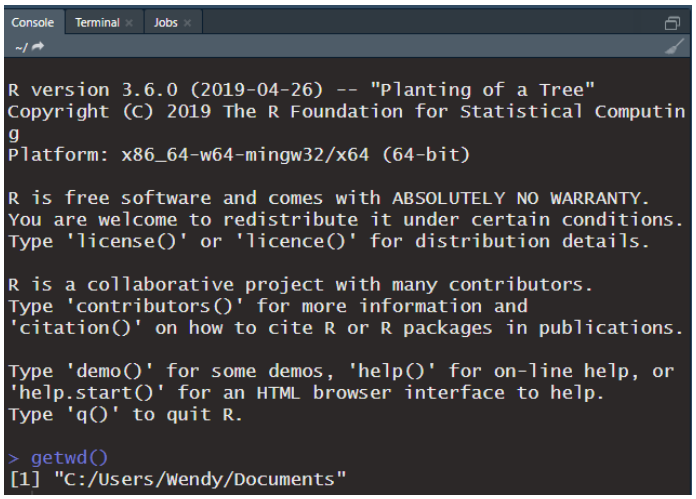


Figure 3.5: Executing 'getwd()' in the console to obtain the current working directory.

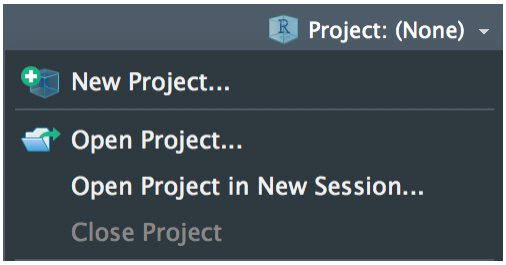


Figure 3.6: Creating a new project.

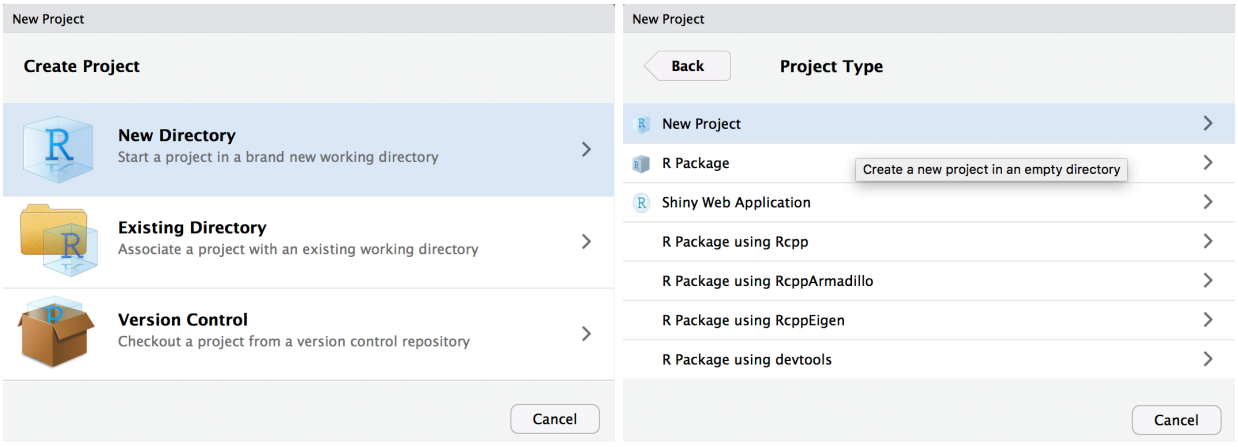


Figure 3.7: Creating a new project.

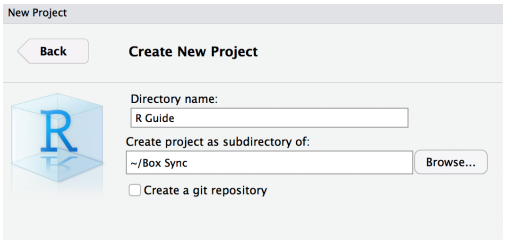


Figure 3.8: Creating a new project.

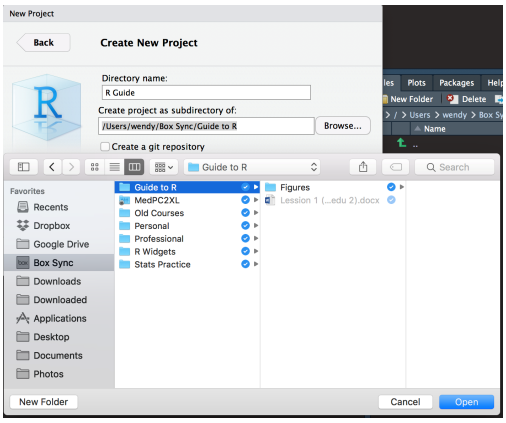


Figure 3.9: Setting up the directory for the project.

Step 3:

Name your folder. Here, I've named mine **R Guide**. See Figure 3.8.

Step 4:

Under **Create project as subdirectory of:**, click on **Browse...**. Highlight (by selecting) the folder for which to create the Project folder and click **Open** followed by **Create Project**. In the example below, I have highlighted the folder **"Guide to R"**. This means my project folder will go *inside* **"Guide to R"**. See Figure 3.9.

3.2.1.1 Checking the Set-up

In the top right corner of RStudio, you should now see the name of your project. See image 3.10.

Let's check your **File Explorer (Windows) or Finder (MAC)** to see how the folder is organized. See image 3.11.

My R project folder, named **R Guide** was created inside my **Guide to R** folder". Inside the **R Guide** folder, R has created an `.Rproj` file. You can access this R Project any time simply by opening this `.Rproj` file. Directly opening the `.Rproj` file is my preferred method to open RStudio, but you could also choose to open RStudio from your desktop (clicking on the icon) and manually open the project by clicking on the drop down menu on the top right of RStudio. See Figure 3.12.

Figure 3.10: The project name is now displayed.

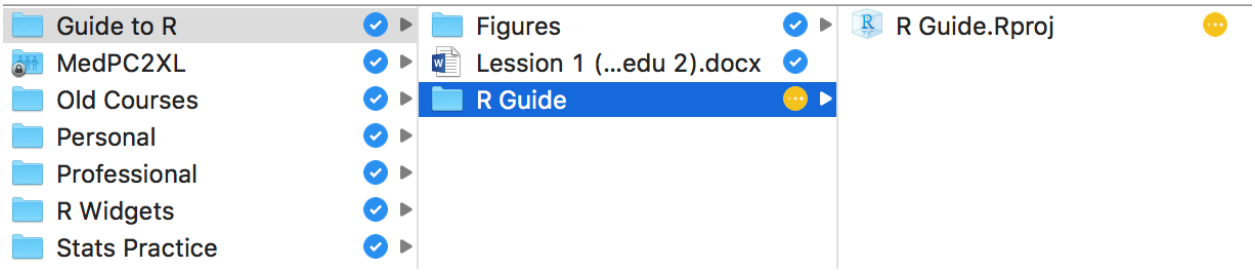


Figure 3.11: Checking the file location of the project.

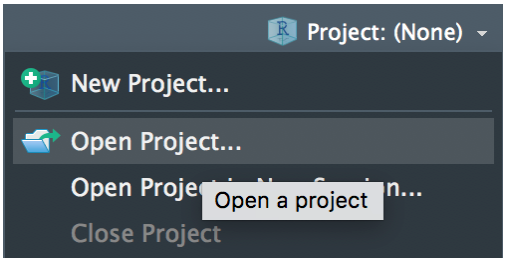


Figure 3.12: Alternative option of opening an R project.

If we execute `getwd()` again at this time, we'll see that our current wd is set to our Project folder!

3.2.2 Creating a Script

Now that we have a project set up specifically for this guide, let's create a ***script*** by clicking on the top left icon with the green plus sign or with the **Shift + Control + N** keyboard shortcut (Figure 3.13. A script is similar to a text document (e.g., Microsoft Word, Notepad, TextEdit, etc.) We can save our coding progress using a script and open it up later. Most of our code will be executed using the script. Many people create separate scripts for different purposes. For example, for every lab experiment, I have a script that organizes the raw data, a script that graphs data, and a script that performs statistical analyses. If you are unsure whether or not to create a new script, ask whether it would make sense to create a new Microsoft Word document for a similar situation. My recommendation is to create a new script for each chapter or significantly long section in this book. We'll delve more into more detail in later sections.

Begin by typing a comment (i.e., text that is not recognized as code by R) by typing a **#** before the text: This is my new script! Then, **save the script** (Figure 3.14. This script will automatically save in the project folder (i.e., the current working directory). I do not recommend saving anything outside of this project folder at this time – more on this later. Here, I've saved my script under the name, "Script1". You can choose to label your script with a more descriptive label or keep it simple.

3.2.3 Installing Packages

Next, we will go over how to install R packages. R packages contain a collection of tools that allow R to perform certain tasks. For example, some packages are designed to help you graph while others help perform statistics. R packages can provide better ways to code in R by building on the foundational "base" code R has by default.

R packages only need to be installed once on a computer. The only times you would re-install a package is if you updated the R Statistics version (or for package updates). Just as new versions of software can slow down or not work on an older computer, updated packages may not work on an old R Statistics version.

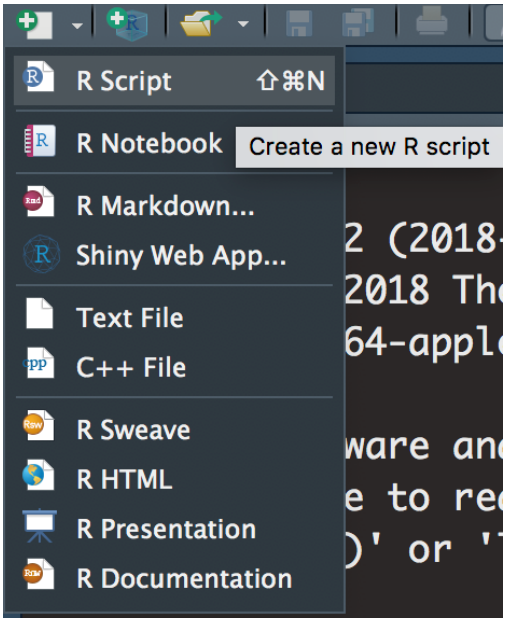


Figure 3.13: Create a new script.

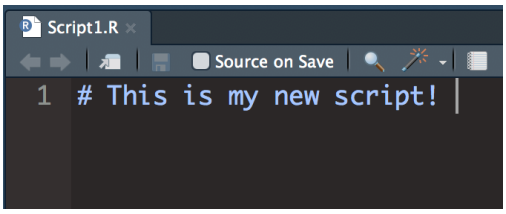


Figure 3.14: Save the script.

Similarly, a new computer (R versions) may not be able to run old software (packages), so it's important that both the R version and the package version can work together. Remember that everything in RStudio is local to your computer. That is, if you want to use packages on multiple computers, you must separately install them for each computer.

For this guide, you will need to install the following packages: **tidyverse**, **afex**, **emmeans**, **writexl**, **readxl**, and **ggthemes.** In further sections, you may be asked to install additional packages, but these will give you a good starting point. Here is the code you will need to execute:

Notice that the names of the packages are surrounded by quotation marks!

Below are brief descriptions of how I will use these packages:

Name

Purpose

tidyverse

This package is actually comprised of multiple packages. The functionality is broad and useful. Its utility in my case lies in its abilities in graphing (ggplot2) and user-friendly formatting (dplyr). Installing this package will take some time—wait until the console is completely finished installing the package.

afex

I use this for statistics, particularly for ANOVAs (analysis of variance)

emmeans

This is another stats package. I use this for calculating the estimated marginal means for post-hoc analyses.

writexl

This allows me to produce Excel files from my R data.

readxl

This allows me to load my Excel files into R.

ggthemes

This provides additional tools for managing graphing aesthetics.

I recommend that you inactivate the `install.packages()` code after it has been executed once (Figure 3.15. This is done by highlighting the relevant code and using `Ctrl + Shift + C`. Remember that packages only need to be installed once per computer (unless you update the R version (Section 3.2.5). If you execute `install.packages()` for a package R already has installed, there will be a pop-up asking to restart the R session. When this happens, you can click *Cancel* and inactivate or remove the relevant `install.packages()` code. I highly recommend **keeping a separate script containing a list of packages installed and the purpose they serve**. Here is an example for what that script might look like:

```
install.packages("tidyverse") # Used for coding style
install.packages("afex")      # Perform stats (ANOVAs)
install.packages("emmeans")   # Calculates estimated marginal means

# OR

install.packages(c("tidyverse", # Used for coding style
                   "afex",      # Perform stats (ANOVAs)
                   "emmeans"))  # Calculates estimated marginal means
```

```
# Active Code ------------------
install.packages("tidyverse")
install.packages("afex")
install.package("emmeans")
install.packages("writexl")
install.packages("readxl")
install.packages("ggthemes")

# Inactive Code ----------------
# install.packages("tidyverse")
# install.packages("afex")
# install.package("emmeans")
# install.packages("writexl")
# install.packages("readxl")
# install.packages("ggthemes")
```

Figure 3.15: Inactivating 'install.packages()' after use.

3.2.4 Loading Packages

Once packages are installed onto your computer, **you must load them with `library()` each time you begin your R session (i.e., open up RStudio)**. Your installed packages are not automatically available to you upon the start of each session.

Begin by loading all of your libraries as such:

```
# The only necessary package for now.
library(tidyverse)

# These packages are not necessary to load at this point,
# but it won't hurt to load them. Sometimes, packages may
# cancel each other out, so you may only want to load
# them when they are needed. However, these packages don't
# cancel out anything we need for this book.
library(afex)
library(emmeans)
library(writexl)
library(readxl)
library(ggthemes)
```

Notice that unlike `install.packages()`, the name of the package in the `library()` function **is not surrounded by quotations**. These formatting details are unique and specific to the code. The more you practice R, the better you will be at recognizing these conventions.

These packages have far more utility available than what I will show you here. Packages are written by experienced R users who are looking to make their code more efficient and share their knowledge. There are far more packages available than I have ever used (currently almost 14,000!!; https://cran.r-project.org/web/packages/). How do you know which packages to use? I use these packages because many of the instructional guides and textbooks I've used also used them. Simply put – it's all about word of mouth/internet posts that lead you to select a particular package.

You are now ready to follow along the examples and exercises. Don't forget to **periodically save** your script just like you would a text document (**Ctrl + S**)!!

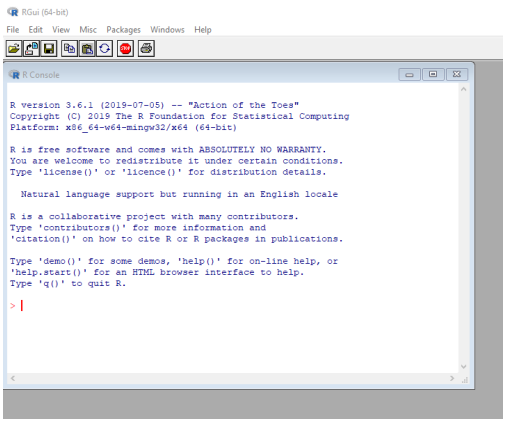


Figure 3.16: R GUI.

3.2.5 Updating the R Version

Periodically, you should update R. This must be done manually as R/RStudio will not automatically update the application. I typically update my R version about every three months. To determine what version you currently have, execute `getRversion()` in RStudio (be mindful of capitalization). Small version revisions are indicated by changes in the third digit whereas large revisions will change the middle number. There are a few ways to update your R version.

Computers running R on a Windows OS can easily update R through the original R application (i.e., R Graphical User Interface; GUI). See Figure 3.16. Using the R GUI application, execute `install.packages("installr")` followed by `library(installr)`. Then, an "installr" tab will appear at the top of the screen next to the "Help" drop down menu. Under the "installr" tab, you can click on the "Update R" option. See https://www.r-statistics.com/2015/06/a-step-by-step-screenshots-tutorial-for-upgrading-r-on-windows/ for more information. Notice that there is an option to copy packages from the older version of R to the newer version of R. This eliminates the need to reinstall packages to the newer version of R.

Alternatively, you can update the R version manually (any OS). This is procedurally identical to Section 3.1.1. This method requires packages to be reinstalled.

3.3 An Informational Tour

In this section, we will cover basic information (vocabulary/keyboard shortcuts) about R/RStudio.

3.3.1 Types of R Files

There are many different kinds of files that R can save. Here are the three common R-based files (require the R application to open) that we will encounter.

File Type

Purpose

.Rproj

This is your project file in RStudio. This automatically sets your working directory to this containing folder.

.R

This is your script file. It contains your previously saved code. Like a Word Document.

.RHistory

R keeps track of all the commands you use in a session.

3.3.2 Hot Keys to Remember

These are immensely useful shortcuts that will save you some time while coding. I highly recommend taking the time to practice using these hot keys. Some hot keys can use left ***or*** right arrows. Ctrl is short for the "Control" key.

Commands

Function

Ctrl + Enter

Executes/runs code

Ctrl + Shift + C

Turns the current line of code into a comment. Inactivated lines will not be read and executed as normal code.

Ctrl + Left/Right Arrow

Moves your cursor all the way to the right/left

Ctrl + Shift + Left/Right Arrow

Highlights entire line to the left/right of your cursor

Alt + Left/Right Arrow

Moves your cursor one chunk of letters at a time

Alt + Shift + Left/Right Arrow

Highlights one chunk of letters at a time

Ctrl + A

Select all text

Ctrl + S

Saves the file. Do this every few minutes to save your progress.

Ctrl + Shift + R

Creates headers for your R script. You can easily navigate between labeled sections of your R script.

Ctrl + Shift + M

Shortcut for %>%, also known as a "pipe". The pipe symbol is frequently used in the tidyverse-style syntax. More on pipes later.

Alt + Shift + K

Displays all programmed keyboard shortcuts for R.

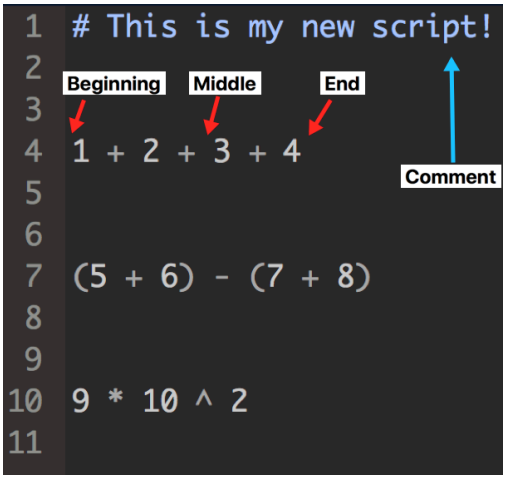


Figure 3.17: Your cursor can be placed anywhere on the line when executing code. Note that comments are not recognized by R as executable code.

3.3.3 Executing Code

To execute (AKA run or evaluate) code in R, you must place your cursor (blinking vertical line) **on the line** of code for which you want to execute. You may place your cursor at the beginning, middle, or end of the line – the placement doesn't matter since R will execute the entire line. See Figure 3.17.

After you have placed your cursor in the proper line, use **Control + Enter** (Ctrl + Enter) to execute the code. If you are using a Mac, you may also use **Command + Enter** (Cmd + Enter) – the end result is the same.

3.3.3.1 Exercises

Use a script to type and execute code in the following exercises (see Section 3.2.2 for information on creating scripts).

1. Type out the following code in your script (including comments):

```
# This is a comment!

## Problem A (also a comment)
1 + 2 + 3 + 4

## Problem B (hit 'Enter' after the plus sign before '4')
1 + 2 + 3 +
  4

## Problem C
9 * 10 ^ 2
```

2. Notice that R automatically indented `4` when you hit **Enter** after `1 + 2 + 3 +`. This is because R recognizes that you are not finished coding this line. A plus sign indicates that there will be another item to add. Even though `1 + 2 + 3 +` and `4` are on separate rows, R still considers this as one continuous line of code; thus, you may place your cursor anywhere on this continuous line of code to execute it (top row/bottom row). Many programmers prefer to add these stylistic breaks in code to aid in visual aesthetics. That is, pressing **Enter** after specific punctuation marks (`,`, `+`, `%>%`) to introduce

breaks will make code easier to read without the need for horizontal scrolling. Practice typing code with and without breaks in these examples:

```
# No breaks
(1 + 2 + 3) + (4 + 5 + 6) + (7 + 8 + 9) + (10 + 11 + 12) + (13 + 14 + 15) + (16 + 17 + 18)

# Includes breaks
(1 + 2 + 3) +
  (4 + 5 + 6) + ## notice the indentation after the first line
  (7 + 8 + 9) +
  (10 + 11 + 12) +
  (13 + 14 + 15) +
  (16 + 17 + 18)
```

3. Place your cursor at the top of Exercise #1 (on the same line as the comment). Begin executing code (**Ctrl + Enter**), repeating this after each line is executed. Each time you execute a line of code:

   - There is an output shown in the console. This output is the result of the code evaluation.
   - The cursor automatically moves to the next line of text, whether it's a comment or active code. You do not need to manually move the cursor to the next line of code!
   - Comments are not executed as code.
   - After you run out of code to execute, the console will produce empty prompts (`>`) without outputs if you continue to press **Ctrl + Enter**.

4. **You can also execute code by selectively highlighting the text.** For this exercise, highlight the code in the example below one line at a time and execute them. Make sure that `+` is not the last character you highlight! You wouldn't end a sentence with a comma, so don't execute code that's not finished (i.e., ends on a comma (`,`), plus sign (`+`), or pipe (`%>%`))!

```
1 + 2 + 3 # highlight and execute '1 + 2 + 3' without this comment!

1 + 5 + 7 # highlight only '1 + 5' and execute
```

5. **You can also choose to execute all of the code in your script.** The easiest way to do this is with **Ctrl + A** followed by **Ctrl + Enter**.

3.3.4 Typing in the Script versus the console

There are two panes in R Studio where you can execute code: in the script or in the console. The script is an archivable file (i.e., can be saved to a file on your computer) whereas the console is (generally) a temporary space to execute code. Executing code in either space will produce the same output; however, programmers use scripts to save their progress for later use.

So why would you ever execute code in the console, a space that won't save what you're doing? As a beginner, you probably won't do this often because you need to familiarize yourself with the code; you'll likely need to come back to a saved script to remember which code performs which task. As you become proficient in R, you may not want to save every single detail in a script. Perhaps you'd like to test out a chunk of code. Scripts are like Microsoft Word/text documents. Sure, you could rewrite all of your code each time you opened RStudio, but that would be like re-writing your essay assignment each time you opened Microsoft Word. In contrast, executing code in the console is more like typing in URLs in a web browser. You usually do not need to save short URLs once you're familiar with them (e.g., who needs to bookmark Google.com?).

3.3.4.1 Exercises

Let's get more practice executing code.

1. Type out the following code in your script:

```
# This is a comment
# Comments don't get evaluated as code by R.

## Problem #1
1 + 2 + 3 # sum of 1, 2, and 3

## Problem #2
1 - 2 - 3 # difference of 1, 2, then 3

## Problem #3
"hello"

## Problem #4
# 4 + 5 + 7

## Problem #5
# hello
```

2. Execute all of the above code with **Ctrl + A** followed by **Ctrl + Enter**.

   Notice how R did not evaluate `4 + 5 + 7` in the console like in Problem #1 and Problem #2. This is because `4 + 5 + 7` is a comment! There is a # symbol preceding the addition problem.

3. Execute one line at a time by repeatedly pressing **Ctrl + Enter**

4. Execute each Problem in the console.

   Notice that the console will execute text using **Enter** whereas the script requires **Ctrl + Enter**. It is also slightly less visually pleasing to execute lengthy code in the console. We cannot easily insert breaks for styling purposes.

5. Highlight and activate `# 4 + 5 + 7` (**Ctrl + Shift + C**)

   Notice how the # symbol disappeared. This highlighted portion can be inactivated again with **Ctrl + Shift + C**. I refer to certain comments as inactivated code. That is, if these comments were activated, they would produce an output.

6. Place your cursor between the `2` and `+` in Problem #1 and execute.

7. Place your cursor at the end of Problem #1 (after `3`) and execute.

8. Place your cursor at the beginning of Problem #1 (before `1`) and inactivate the line.

   Notice how the rules for inactivating a line of code is similar to executing the line. You may choose to inactivate code via highlighting specific lines. Inactivating via **Ctrl + Shift + C** inactivates the entire line. To inactivate a portion of the line, `#` must be placed manually.

9. Highlight `1 + 2` in Problem #1 and execute.

   Notice that only the code in this highlighted portion is executed. That is, R only added `1 + 2` and excluded `+ 3` despite its presence in the script. R will ignore all text/code outside of the highlight when using this code execution technique.

10. Highlight `2 + 3` in Problem #1 and execute.

11. Highlight `1 + 2 +` in Problem #1 and execute.

Notice that nothing appeared in the console. R is waiting for additional code. Since the code ended with a `+`, R is waiting for you to enter in additional code. This particular code portion could be read as:

> "1 adds to 2 adds to…"

You essentially didn't finish the sentence! To exit out of this incomplete code, place your cursor in the console and hit **ESC** on your keyboard. When you see `>` in the console again, R is ready to try again. If you forget to hit **ESC** and reset the console input, you will likely encounter an error.

12. Inactivate Problem #5's code with **Ctrl + Shift + C**, `# hello`, and execute the code `hello`.

    Notice that you'll get an error that reads: `Error: object 'hello' not found`. This is because R recognizes letters in a few ways:

    - As values/definitions
    - As an object

    If letters/words are values, you must place the word(s) in between quotations such as in Problem # 3. If letters/words are objects, they must be explicitly defined in the environment by the user. In this example, R expects `hello` to be an object (and not a value) because it is **not** enclosed by quotation marks. More on objects in the next section.

13. The console holds a limited memory for recently executed code. Now that you've executed some code in RStudio, click in the console pane and press the "up" arrow on your keyboard. This is a quick and simple method to re-execute a line of code.

3.3.5 Vocabulary Terms

In this section, we will go over some of the common terms used in R. These are the vocabulary terms that we have seen so far:

Vocabulary Term

Definition

R Statistics/R

"R" can refer to the R Statistics application or to R as a language.

Base R

Base R refers to the coding tools available by default in R. It is frequently used to describe the default coding style/rules (i.e., base R syntax). Base R packages and functions are always automatically loaded upon opening RStudio. Many R users use additional packages that utilize Base R code as a foundation to produce more efficient/attractive code. The tidyverse syntax style, which builds from base R, is increasingly being adopted by the R community; however, there are situations where Base R is still preferred.

RStudio

This is the Integrated Development Environment (IDE) that facilitates R coding. Simply put, RStudio organizes R code in a user-friendly manner. R by itself is just the language whereas RStudio is the program that houses and uses the R language. The R language can be used without RStudio. However, RStudio is more organized, visually appealing, and powerful than the default R application alone.

Package

A collection of free R tools that an R User wrote. Anyone can create a package (with advanced R training). Packages were written to making tedious coding tasks more efficient. Different packages will provide different sets of tools (i.e., wrench set vs. screwdriver set), though some tools may have overlapping functions (i.e.,

scissors vs. knife). Packages are free but must be installed to your computer first. After installation, packages must be loaded into your RStudio library each time RStudio is opened/launched.

Script

Scripts provide a space to type, execute, and save your code. These saved scripts can be opened at a later time to edit or execute again. Upon each new RStudio session (i.e., every time you open RStudio), RStudio is a blank slate and you must re-execute your code. Scripts allow users to save previously written code so that re-executing code is simple. With RStudio, you can even open up multiple R Scripts in separate tabs. R Scripts are like Microsoft Word (or any text-editing application): both filetypes can save previous work/text so that the user does not need to re-type its contents and both can be opened on any computer with the application installed.

Console

This is a universal feature in R Statistics and RStudio that displays the output of executed code. That is, each time code is executed, the console will display the line(s) of code used and the outcome. Code can be executed directly through the console or indirectly through the script. Executing code in the console only requires you to hit Enter, while executing code in the script requires Ctrl/Cmd + Enter. Code that is executed in the console will not be saved to a file – it is best practices to use a script if you want to save code for later use.

Project

An organizational tool that sets up a designated location (i.e., folder) on the computer for the R user. Any file that is saved/exported by R will be saved in this designated location/folder and any file that is imported into R must also be located in this folder.

Working Directory (WD)

This is path to the folder in which you are working. Computer files are located in specific folders and those folders are oftne nested inside other folders. For example, my photos from Christmas 2018 are located in a folder named `Christmas`. The file path to this folder is: "C:/Users/Wendy/Photos/Christmas" where each word in this path represents a folder. That is, I must open the C-drive on my computer, click on the `Users` folder, then `Wendy` folder, then `Photos` folder, and finally the `Christmas` folder to get to my Christmas 2018 photos. Setting this file path as my working directory would mean that I am setting the `Christmas` folder as my default folder for everything (i.e., for saving files or importing files). A working directory can be set to a specific file path manually (via coding) or it can be automatically set using an R Project. The working directory in an R Project will always be the file path to the .RProj file by default.

3.3.6 The Global Environment

This is the "work-space" in which your code is working in. You always want to start your R session with a clean environment (i.e., no objects defined; see 3.18). To clean out your environment, click on the broomstick icon in the Environment panel.

The Environment is where objects are temporarily saved for the duration of your R session; the session is over when you have exited RStudio. Objects are things you have defined (e.g., values, datasets, graphs, functions) within your R script. You can refer back to the object at a later time for when you need it again.

3.3.7 Using Objects

Saving information to objects in the environment is useful to cut down on the amount of coding. Think back to algebra class where you had to solve for "x".

> If x + 4 = 10, we know that x = 6. That is, x is an object defined as the value of 6. Now that we have a definition for "x", we can use this object in place of 6. x + 20 would represent 6 + 20.

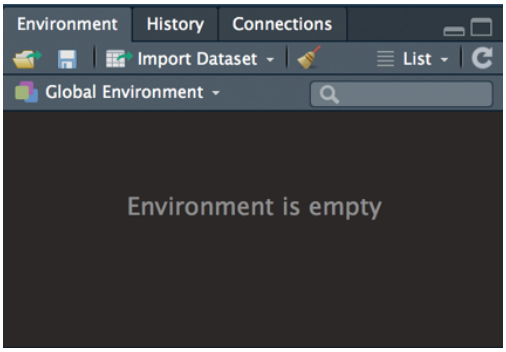


Figure 3.18: Empty environment.

In R, we can define objects using the `<-` operator. Notice how the object appears in the Global Environment after it is defined (via code execution; **Ctrl + Enter**).

```
x <- 6
```

We could also create objects with more complex definitions. `c()` is a function that allows R to *concatenate* or group together the contents listed inside the function.

```
y <- c(1,2,3,4,5)
```

...and choose more complex object names (single letter names are frowned upon).

```
numbers <- c(1,2,3,4,5)
```

We can perform tasks on complex objects as we've done before with simply defined objects.

```
mean(numbers) # calculates the mean of values in the object
```

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

```
numbers + 1   # adds 1 to each number in the object
```

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

```
sum(numbers)  # calculates the sum of values in the object
```

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

...and we can define these tasks as objects as well! Remember that the object name is determined by us (although there are limitations to usable names as we'll see later in Section 3.4.2)!

```
numbers.mean <- mean(numbers)
numbers.add1 <- numbers + 1
numbers.sum <- sum(numbers)
```

You can also define entire datasets as objects! The possibilities are endless.

It's important to note that you can only use the object if it has been defined in the Global Environment. Objects should only remain defined within the current RStudio session (a caveat to this will be discussed in 3.5). Once RStudio is closed out on your computer, the objects must be redefined the next time RStudio is opened. **Always start with a clean environment each RStudio session.** This is where scripts come in handy. Rather than typing out the definitions for every object upon a new RStudio session, you can save a script that contains the code! Code that is saved to a script can then be executed any time.

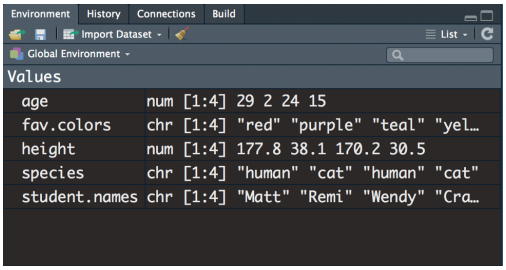


Figure 3.19: Environment with 5 objects.

```
> data
# A tibble: 4 x 5
  student.names fav.colors   age height species
  <chr>         <chr>      <dbl>  <dbl> <chr>
1 Matt          red           29  178.  human
2 Remi          purple         2   38.1 cat
3 Wendy         teal          24  170.  human
4 Craig         yellow        15   30.5 cat
```

Figure 3.20: Console output after executing the data object.

3.3.7.1 Practice with Defining Objects

First, let's make sure that our **environment is completely empty** *(see Figure 3.18)*. To empty out the environment manually, click on the broomstick icon and confirm that all objects are to be removed from the environment. Go ahead and check "Include hidden objects" as well. Make sure that you have a script created (see Section 3.2.2 for how to create a script)!

Now, let's dive into some exercises:

1. Execute the following code to create 5 objects in the environment:

```
# creating an object named "student.names" that contains three names
student.names <- c("Matt", "Remi", "Wendy", "Craig")

# creating an object named "fav.color"
fav.color <- c("red", "purple", "teal", "yellow")

# creating an object named "age"
age <- c(29, 2, 24, 15)

# height in centimeters
height <- c(177.8, 38.1, 170.18, 30.48)

# human or cat?
species <- c("human", "cat", "human", "cat")
```

2. Next, let's combine these objects into a single dataset:

```
data <- tibble(student.names, fav.color, age, height, species)
```

3. Execute `data` using one of methods discussed in 3.3.3. Your console output will look like this:

4. You can also view your `data` object in a separate RStudio tab. This can be done by clicking on the `data` object in the Environment or by executing `View(data)`:

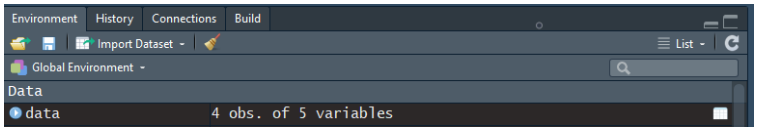


Figure 3.21: Object named 'data' in the global environment.

data

Filter

	student.names	fav.colors	age	height	species
1	Matt	red	29	177.80	human
2	Remi	purple	2	38.10	cat
3	Wendy	teal	24	170.18	human
4	Craig	yellow	15	30.48	cat

Figure 3.22: View of object named 'data'.

Ultimately, this is the result:

3.3.8 What is a Function?

Functions are a useful tool that allows you to perform tasks quickly and efficiently. Quite frankly, they are the backbone of any R code. I often get asked how I know the names of the functions that I use, and the short answer is: through Google and practice. There are functions I happen to use a lot in my code and through repeated usage, I have come to memorize what functions I need for the situation. Just like learning when you should use a screwdriver versus a hammer versus glue for a crafting project, you learn which tools are appropriate through repeated use.

All of the functions I utilize were ones that various guides and tutorials used to accomplish a specific task. These functions were all written by someone else, who defined how the function should work. At this time, you will not need to learn how to write your own functions. Many people don't ever write their own functions because all of their coding needs are satisfied by what is currently available. Despite this, learning how to write functions (after you've got the basics down) adds another layer of knowledge that could be useful for more complex coding in the future.

3.3.8.1 The Anatomy of a Function

The anatomy of a function is structured into two main parts: the name and the arguments. The name of the function is listed on the left side of the parentheses while the arguments are located inside the parentheses: `name.of.function(argument1, argument2)`.

Some examples of functions include:

- `library()`
- `ggplot()`
- `mean()`

You can look up a function's help page by executing a question mark before the name of the function as such:

- `?library`

- `?ggplot`
- `?mean`

Here, the `mean` function has one mandatory argument labeled `x`. Here, `x` is defined as "an R object". The `mean()` function also has some built-in optional arguments that may be useful in certain circumstances. The second argument for the mean function is trim and the third argument is `na.rm`. The `na.rm` argument is particularly useful if your object contains a missing (`NA`) value. The na in `na.rm` stands for NA and rm stands for remove. Under default settings, using the `mean` function on an object with `NA` values will result in a `NA` output because this function does not remove NA values before performing its operations.

3.3.8.2 The Function of a Function

Arguments are the objects that functions use. For example, if you wanted to use a hammer as your tool, you want to know what object to use the hammer on (i.e., bird house). If you don't specify which object you need to use your hammer on, the hammer just sits there and does nothing. Similarly, if you don't specify arguments within a function (`mean()`), nothing really happens.

Sometimes, you may need to know multiple pieces of information before you use the hammer. What kind of material is the bird house made of? How fragile is the material? How big is the bird house? Are there certain parts of the bird house where you can't use the hammer? Similarly, some functions will need multiple arguments to work. Most functions also have optional arguments that aren't usually necessary but can add extra specifics that might be useful aesthetically or peripherally.

Functions within Functions: Using Multiple Functions at Once Just like performing multiple operations on a graphing calculator, you can utilize many functions at one time by nesting them within each other. For example, you can choose to round the mean of an object to the nearest whole number as such:

```
round(mean(numbers2, na.rm = T), 0)
```

Here, the `round()` function is used outside/around the inner `mean()` function that we have used so far. When we solved for the mean of numbers 2 previously while removing NAs, the result was 2.5. To round to the nearest whole number, we use the `round()` function. This function requires an object `x` and the number of digits argument that specifies how many digits to the right of the decimal to keep (see `?round` for more information). Since we want the nearest whole number (i.e., no decimal points), the digits argument (second argument) is 0. It is possible to nest as many functions within themselves as you want; however, this can become difficult to keep track of which argument belongs to what function. One way to mediate some confusion is to determine which parenthesis belongs to which function. By moving your cursor to the right side of a parenthesis, R will automatically highlight the corresponding paired parenthesis. I should also mention that the second most common mistake in coding is to forget to add a closing parenthesis, leaving a lonely open parenthesis hanging. Not to worry—R will let you know of this issue by issuing an error message any time you are missing the proper punctuation/symbol.

3.3.8.3 Functions help with efficiency

Instead of calculating the mean of the numbers 1,2,3,4,5 like this:

```
# calculating the mean of numbers 1 through 5
(1 + 2 + 3 + 4 + 5)/5
```

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

You can do this faster with the `mean()` function

Method #1:

```r
mean(1:5)

# The colon indicates "through" as in the mean of numbers 1 through 5
```

Method #2:

```r
mean(1,2,3,4,5)

# typing all of the numbers manually as arguments for the mean function
```

Method #3:

```r
Numbers <- c(1,2,3,4,5)
mean(Numbers)

# creating a separate object that contains the values for which to use
# in the mean() function. The object is defined as the concatenation of
# numbers 1,2,3,4,5 via the concatenation function, c()
```

Example

Code

This is a function I want to use. It calculates the mean of an object.

`mean()`

This is an object I created. The <- symbol indicates an object is being defined, where the object name is located on the left of the symbol and the definition is on the right. Here, Numbers is defined as the concatenated vector containing numbers 1 through 5.

`Numbers <- c(1,2,3,4,5)`

If I want to calculate the mean of my object called `Numbers`, I would put Numbers as an argument within the function `mean()`.

`mean(Numbers)`

But let's say we've got a special object that contains a `NA` value.

`Numbers2 <- c(1,2,3,4,5,NA)`

Executing this would lead to a `NA` result. R will not be able to calculate the mean, because there is a missing (`NA`) value.

`mean(Numbers2)`

However, the mean function includes an optional argument that allows the function to ignore the `NA` value and continue to calculate the mean without it. Here, `na.rm = TRUE` (notice the capitalization required) specifies that mean() should remove (rm) all `NA` values (na) when calculating the mean of Numbers2. By default, `na.rm = FALSE`. In this case, you must manually switch na.rm to be TRUE. You can choose to include this second argument for Numbers object, but because there is no `NA` value, there is no change in consequence. The default setting for these optional arguments is usually listed in the help page for each function (accessed by executing `?mean`)

`mean(Numbers2, na.rm = TRUE)`

`mean(Numbers, na.rm = TRUE)`

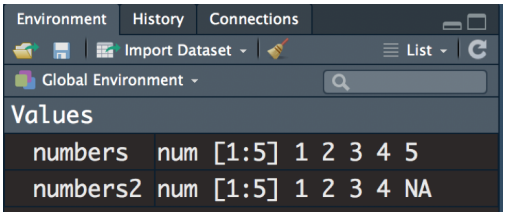


Figure 3.23: Defining numbers and numbers2.

3.3.8.4 Exercises

1. Pull up the help pages for the following functions (hint: use a question mark):

- `library()`
- `ggplot()`
- `mean()`

2. Create two objects:

```
numbers <- c(1,2,3,4,5)
numbers2 <- c(1,2,3,4,NA)
```

You can view the definition of `numbers` and `numbers2` any time by executing just the name of those objects:

```
numbers
numbers2
```

Notice that your Environment has also changed since you've defined these objects. Remember that in order for the Environment to save information, you must define it with the `<-` symbol (see Section 3.4.3 for more information about built-in symbols):

Next, calculate the means of each object as such (what do you see?):

```
mean(x = numbers)
mean(x = numbers2)
```

Next, add the `na.rm` argument. Here, adding `na.rm = TRUE` is redundant, because the numbers object does not have any `NAs` to remove in the first place:

```
mean(x = numbers, na.rm = TRUE)
mean(x = numbers2, na.rm = TRUE)
```

Notice that `TRUE` turned a different color. Code that reads `TRUE` and `FALSE` are recognized by R as a special phrase. In shorthand, you could type `T` instead of `TRUE` and it would produce the same result. Similarly, `F` and `FALSE` mean the same thing:

```
mean(x = numbers, na.rm = T)
mean(x = numbers2, na.rm = T)
```

As you become familiar with functions, you can omit the argument name:

```
mean(numbers)
mean(numbers2)
```

This is because R assumes that the first argument entered into the function is the `x` argument. We can see the assumed order of arguments by viewing the function's help page (see `?mean` for the help page). If the first argument I entered in the `mean()` function was not the `x` argument, I would need to specify what argument the value belongs to:

```
mean(na.rm = T, x = numbers)
mean(na.rm = T, x = numbers2)
```

In the `mean()` function, the order that R expects to receive arguments is:

```
mean(x, trim, na.rm)
```

Although we haven't talked about the trim argument, it is technically listed as the second argument for the `mean()` function. Remember, `x`, `trim`, and `na.rm` are the names of the arguments—they are not the arguments themselves. As such, you do not have to include the argument names in the function as long as you remember the order in which the arguments must be listed. For example, you can omit the x argument label for the first argument:

```
mean(numbers, na.rm = TRUE)
# or
mean(numbers2, na.rm = T)
```

But you can't run this:

```
mean(numbers, TRUE)
# or
mean(numbers2, T)
```

This is because `na.rm` is the **third** argument in the `mean()` function. Without the `na.rm` label specification, R is assuming that the second value that I enter is the trim argument (see `?mean` for more information). Remember: although the R help page will specify many arguments available for a function, not all of them are mandatory (i.e., `trim`, `na.rm`). Some are required in some circumstances (i.e., `na.rm = TRUE` is necessary when the object `x` has a missing value). Some are never required and can be used to the user's discretion (i.e., `trim`).

3.3.9 Pipe/Piping (%>%)

Pipes are a convenient tool to organize the tidyverse syntax style of coding. Recall that there are two major camps of coding syntax: tidyverse and Base R. Both have pros and cons and while most R users use a mix of the two, people typically favor one more. I personally prefer **tidyverse** when it's feasible. To understand what a pipe does, we first need to understand some Base R syntax.

Recall that a dataset is a group of related variables. That is, each variable represents a column and each row represents a unique observation. Here is an example of a small dataset. Notice that each column represents a variable and each row represents a unique individual (animal).

```
## # A tibble: 4 x 3
##   Subject Animal CoatColor
##   <chr>   <chr>  <chr>
## 1 Craig   cat    brown
## 2 Aries   dog    black
## 3 Remi    cat    grey
## 4 Duke    dog    brown
```

Base R's datasets are called **data frames** or **df**. This is how the data is structured in R by default. Tidyverse-styled datasets are called **tibbles** or **tbl**. Tibbles have *all* of the same properties as data frames *and more*. Tibbles are designed to be used with the tidyverse syntax style. Check out Table (()) for comparisons between data frames and tibbles.

Pipes are a shortcut tool (**Ctrl + Shift + M**) that the package known as **tidyverse** uses for more efficient coding and improved readability for the user. Hitting **Enter** after a pipe will auto-indent and organize your code in a readable manner.

> Technically, the pipe is derived from the package **magrittr** specifically, but **tidyverse** loads **magrittr** (in addition to other packages) and thus contains the pipe operator.

For example, let's say I want to create a new variable called `Milliliters` from a variable containing values from the variable Liters. I also want to create a new variable called Deciliters from Liters:

Base R:

```
df$Milliliters <- df$Liters*1000
df$Deciliters <- df$Liters*10
```

- Read the above code as: I am defining Milliliters (as a new variable in `df`) as Liters (obtained from a data object called `df`) multiplied by 1000. Then, I am defining Deciliters as (as a new variable in `df`) as Liters (obtained from `df`) multiplied by 10
- The dollar sign indicates that the word following the dollar is found within the word before the dollar sign.

Tidyverse R (using a tibble): You can choose to type this in one line:

```
df %>% mutate(Milliliters = Liters * 1000, Deciliters = Liters *10)
```

Pressing **Enter** after certain kinds of punctuation (pipes and commas) auto-indents the code in a more readable manner:

```
df %>%
  mutate(Milliliters = Liters * 1000,
         Deciliters = Liters * 10)
```

Read the following code as:

> "Within my data object, I want to mutate/change things by creating a variable called "Milliliters". Milliliters is based on the currently available variable, Liters. Specifically, values of Milliliters should be Liters multiplied by 1000. I also want to make a new variable called "Deciliters" from Liters values multiplied by 10."

This example is a simple one and not very telling of how tibbles make things more efficient than base R but trust me when I say that using tibbles will reduce entry error in the long run.

3.3.10 Types of Data

There are many different structural types of data. That is, data can be numerical, letter characters, boolean (TRUE or FALSE values only), categorical, etc. We might refer to data as a single variable, but we could also have data that has multiple variables (i.e., dataset). In this section, we will discuss common types of data and when they are used/what they are used for.

3.3.10.1 What is a vector?

If you are unfamiliar with this term, it is a **data structure which contains a single type of values** (e.g., all letter characters or all numerical type). If you combine multiple vectors together, you can create a dataset. Each column within a dataset is a vector. On their own, each column in isolation simply gives us a bunch of values. Combining multiple vectors to form a dataset can paint a story of those values.

Let's say we have:

- One vector containing the **Names** for each subject
- One vector containing the **Hair** color for each subject
- One vector containing **Age** for each subject

- One vector containing a TRUE/FALSE value called **Human**

When you look at each individual vector separately, it's just a list of numbers/words. If you put all three vectors together, you can create a dataset. **Datasets imply some order to rows AND columns.** Let's see an example.

Let's say that:

- Sam has brown hair and is 24 years old and is a human.
- Tina has black hair and is 41 years old and is a rabbit.
- Alex has blonde hair and is 2 years old and is a human.

We must ensure that each piece of information remains with the relevant person; thus, the order for each vector's values will matter when we put the vectors together. For example, all values related to Sam must be arranged first in each vector.

```
## Defining three objects, all of which are vectors
# A vector containing "Names"
Names <- c("Sam", "Tina", "Alex")

# A vector containing character values representing "Hair Color"
Hair <- c("Brown", "Black", "Blonde")

# A vector containing numeric values representing "Age"
Age <- c(24, 41, 2)

# A vector containing TRUE/FALSE (T/F) values representing human/not human
Human <- c(TRUE, FALSE, TRUE)

## Executing three lines of code to view these objects' definitions in the console
Names
Hair
Age
Human

## Defining an object named mydataset
# This dataset that combines the four vectors defined previously
mydataset <- tibble(Names, Hair, Age, Human)

## Viewing the dataset in the console
mydataset
```

```
## # A tibble: 3 x 4
##   Names Hair     Age Human
##   <chr> <chr>  <dbl> <lgl>
## 1 Sam   Brown     24 TRUE
## 2 Tina  Black     41 FALSE
## 3 Alex  Blonde     2 TRUE
```

3.3.10.2 Atomic Vector Types

Each vector can be categorized based on the types of values it contains. There are five basic types of atomic vectors.

Atomic Vector Types

```
> str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Figure 3.24: Data structure of the mtcars dataset.

Examples

Character (AKA a string)

"a", "labels here", "can be a combination of any letters, numbers, and symbols."

Numeric

2, 15.5

Integer

2L (L tells R to store this value as an integer)

Logical

TRUE or T, FALSE or F (in all capital letters)

Complex

1 + 4i (complex numbers with real and imaginary values

R has a hierarchy of data structures where certain atomic vector types are "more specialized" than others.

- The order of least specialized to most: character, numeric, integer, logical, complex
- R will default to the least specialized it can get unless you specify otherwise
  - If your values contain any letters, variable will default to the character structure.
  - If your value only contains numbers, the variable will default to the numeric structure
  - Values for a variable with an integer structure must be real and whole numbers
  - Values for a variable with a logical structure can only be `TRUE` or `FALSE`

You can use `str()` to view the structure types for all of the variables in your dataset.

Example

Compared to built-in **mtcars** dataset, which only has numeric variables, some variables in the **diamonds** dataset are numeric, some are ordered factors (ranked categories), and one is an integer. In my color scheme, purple text indicates code that has been executed in the console. White text shows the result of the executed code.

3.3.10.3 Exercises

1. Execute `str(mtcars)`

```
> str(diamonds)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':       53940 obs. of  10 variables:
 $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
 $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
 $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
 $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
 $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
 $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
 $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
 $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
 $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
 $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

Figure 3.25: Data structure of the diamonds dataset.

2. Execute `str(diamonds)`
3. Execute `str(npk)`
4. Execute another built-in dataset or two. You can find a list of built-in datasets by executing `data()`

3.3.11 Factors

A factor is another variable structure. Factors categorize character string values for a given variable. To determine whether your variable should be a factor, ask: Do the values in this variable belong in categories?

Example

If I have an experiment with a variable called `Injection`, my values might include:

- `Drug`
- `Control`

Thus, `Drug` and `Control` are two different categorical values that the `Injection` variable can have.

- Since `Injection` is categorical, it is referred to as a factor variable.
- Factor variables are useful for when you want to group a certain category for further analysis. The examples below suggest things you might want to know about your data:
  - I want to see what all of the subjects that are in the Drug group look like
  - I want to see what all of the subjects that are only in the `Control` group look like
  - I want to compare the categories of `Drug` vs. `Control`

Let's create another factor variable: `Subject`. Each value in `Subject` represents a subject ID – that is, `1` represents Subject #1 and `2` represents Subject #2. Even though the values in `Subject` appear to be numeric, they are not: Subject 1 is not less than Subject 2 and you cannot perform arithmetic on these values. Each subject is a category, where Subject #1's data is separate from Subject #2's data, etc. `Injection` is also a categorical variable. `Rating` is a numerical value, where a rating of 5 has some intrinsic value (there is a rank order where a rating of 5 is ranked differently from a rating of 3).

Subject

Injection

Rating

1

Drug

5

2

Drug

3

3

Drug

2

1

Control

9

2

Control

10

3

Control

8

The same thing applies below: Each value in the `Subject` variable is still a category. The `Date` variable contains date values as the numbers indicate some chronological order, rather than some numerical value. The `Short Answer` variable contains simple character values, where none of the values form a category as it currently stands. If the values, such as "It's been a slow day", were a category that everybody could select as an answer, then `Short Answer` would be a categorical variable.

Subject

Date

Short Answer

Sam

2019-02-05

"It's been a slow day"

Anna

2019-01-07

"Pretty standard day"

Michael

2019-03-01

"I've been pretty stressed"

3.3.11.1 Exercises

1. Are the following variables factors/categorical? (yes/no)
    - Height measurement in cm
    - Age category (0-18, 19-25, 26-35, 36-60, 61+)

- Age in years
- Hair color (blonde, brunette, black, grey, other)
- Exercise frequency (Sometimes, Never, Always)
- Number of exercise hours per week
- Subject name
- Personality question asking whether you fit the description (true/false)
- Year (1999, 2000, 2001…)

2. If the variable in the previous question was not a factor, what structure was it?

3.4 R Conventions

3.4.1 Header and Comment Organization

It's a good idea to organize your code and code descriptions so that "Future You" or your collaborators can easily navigate your script. Here are some of my tips:

```
- RStudio has the ability to organize code sections within a single script
(e.g., loading libraries, data management, graphing, statistics, exporting to Excel).
You can jump to the different sections by clicking on the drop-down menu at the bottom
of the script window. If you have not created a header yet, the button will read
"(Top Level)" -- see fig \\ref{fig:image-header}.

- You can create RStudio headers using the `Ctrl + Shift + R` command. A text box will
automatically pop up for you to name. This ensures that the length of all headers are
equal and it saves you time from typing out the dashes.

- By default, four dashes will create an RStudio header (----)

- You may choose to use other heading styles (that won't be tracked by RStudio). One
method I enjoy using for visualization is: `#___________________` which uses underscores.
This allows for clear visualization of section headers, but will not be tracked by
RStudio unless followed with four consecutive dashes. **Make sure your header style is
consistent!**
```

```
## Here are a couple of options for header styles
## Stick with ONE style!

#-----------------------------------
#  Loading Libraries
#-----------------------------------
library(tidyverse)
library(writexl)

#*********************************
# Importing Dataset
#*********************************
data <- read_csv("mydataset.csv")

# Alter Data Structure ----
data <- data %>% mutate(variable = factor(variable))
```

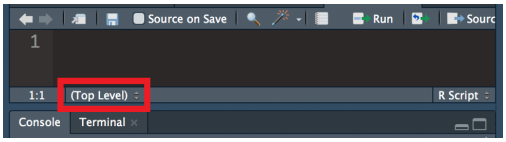


Figure 3.26: Accessing labeled sections in your script

```
# Graphing --------------------------
data %>% ggplot(aes(x = var1, y = var2)) +
  geom_point()
```

Similarly, everyone has their preference for how they organize comments that serve as code descriptions. Here are some examples:

```
# This is a section label ----

# I have section labels for things like:
  # Importing packages
  # Setting up data
  # Setting up graphs

# Think about...
  # Spacing between code and comments
  # How you might want to indent code/comments
    # I prefer to indent my code/comments in a
    # manner that allows me to easily read all
    # lines without having to horizontally scroll.

### You could also distinguish text
## with multiple #
# such as this

# You can provide descriptions for code above:
# Calculates mean of numbers 1-5
mean(1:5)
```

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

```
# Or you can type the comments beside the code
mean(1:5) # Calculates mean of numberes 1-5
```

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

3.4.2 Naming

It's a good idea to stick to a consistent format for organizing your code (section headers) and naming files/folders/scripts/objects—be consistent. It helps Future You keep track of what's going on. Here are some of my naming pointers:

1. Choose Descriptive and Concise Names

It can be painful to type out long object names each time you refer to it—even with the autofill feature.

2. Do Not Start with a Number or Symbol

R will not recognize an object name that begins with a number. However, you can still include numbers in the middle or at the end your object name:

- `Data5` is a good name
- `Data5b` is a good name
- `5Data` is a bad name

3. Avoid use punctuation other than a period (.), dash (-), or underscore (_)

Some punctuation are understood as special commands.

- = equal signs are used to define things
- \+ plus signs signal mathematical addition (same with other operators)
- ! exclamation signs are used to indicate "not equal"
- ‘ apostrophes and quotations are used to indicate that an object is a string/character Similarly, do not begin your label with a symbol!
- `Data_5` is a good name
- `_Data` is a bad name

4. No Spaces

R considers spaces as separate text. Instead, consider labeling your objects like so:

- `ThisVariable`
- `this.variable`
- `this_variable`
- `this-variable`

5. Consistency

Stick with a consistent naming convention you prefer, some include:

- Labeling all objects in lowercase only (`nameofmyobject`)
- Labeling all objects in uppercase only (`NAMEOFMYOBJECT`)
- Separating words with periods only (`name.of.my.object`)
- Separating words with underscores only (`name_of_my_object`)
- Selectively using uppercase (`NameOfMyObject`)

6. Avoid Names Already in Use

How do you know if it is already a name of an object (value, function, graph)? RStudio is really user-friendly and will suggest autocompleting options from its built-in objects as well as from your global environment as you type.

For example, diamonds is already a name for a built-in dataset. If you label your new object "diamonds", it will override the original definition. Instead of naming over the original dataset, I could use a slightly modified name for my new object:

```
diamonds_new <- diamonds
```

If you use the same name for two different objects, only the object that was defined most recently will remain:

```
my.object <- 1 + 2
my.object <- 4 + 5
```

If I run the first line, then run the second line, `my.object` will be defined as `4 + 5`. Afterwards, if I run the first line again, the value of `my.object` will change back again to `1 + 2`. The most recently executed definition will be the most current definition.

It is better to not use the same label for objects unless you want to overwrite the original value. Instead, use something like `m1`, `m2`, `m3` to define different renditions of the original object. It is prudent to overwrite an object only when you are **100% certain** that the old definition is no longer useful.

R cares about lowercase and uppercase: `Library()` is not the same as `library()`. In fact, `Library()` is not a recognized function by default, there will be an error. Spelling or capitalization mistakes are one of the most common sources of error!

3.4.2.1 Exceptions to the Rules

As always, there are some exceptions to naming rules. It is **not recommended** to use a name beginning with a number or containing an illegal character (e.g., spaces, slashes, or symbols other than the underscore, dash, or period).

If you must know, there is a way to get around this by surrounding the ill-conceived object name with backticks (“). Backticks are located on the same key as the tilde (~):

```
## Bad Names

# Begins with a number
`10bad.name`

# Begins with a number, has two illegal characters (percent and space)
`100% Terrible`

# Contains illegal character (forward slash)
`Response/second`
```

It’s not advised to use poor names because they can **increase the likelihood of user error** and because not all functions will accept this workaround.

3.4.3 Built-In Symbols

R has all of the basic arithmetic operations available (+, -, /, *, ^) and can function as a calculator. However, R is also a powerful tool for managing our data.

This table introduces some of the other basic operators that we will frequently use on our datasets. We will practice using these built-in symbols in chapter 4 .

Symbol

Definition

Example

=

A single equal means as defined as, NOT equals. This can be used to define variable names within a dataset (most common use) or objects/datasets (not recommended use).

variable = 2 translates to: the object labeled variable is defined as the number two

<-

Defines objects/datasets. The single equal sign (=) should only be used to define an object within a larger dataset. The arrow (<-) is used to define objects that are not part of larger datasets.

dataset <- 2 translates to: the object labeled dataset is defined as the number two

==

Equal(s)

variable == 2 translates to: retrieve values from the object labeled variable that equal two

!=

Does not equal

variable != 2 translates to: retrieve values from the object labeled variable that do not equal two

>

Greater than

variable > 2 translates to: retrieve values that are greater than two from the object labeled variable

>=

Greater than or equal to

variable >= 2 translates to: retrieve values that are greater than or equal to two from the object labeled variable

<

Less than

variable < 2 translates to: retrieve values that are less than two from the object labeled `variable`

<=

Less than or equal to

variable <= 2 translates to: retrieve values that are less than or equal to two from the object labeled variable

%in%

Determines whether values match (TRUE/FALSE). The format for use is `x %in% table` where `x` and `table` are vectors.

2 %in% c(2,3,4) translates to: Does the value on the left match anything in the vector on the right? This would return `TRUE` the left value has a match on the right value(s).c(2,3,4) %in% 2 would return `TRUE FALSE FALSE` as each value on the right would return TRUE/FALSE for whether it matches with the value on the right of %in%.

|

Translates to the word "or". | can be used to retrieve matching values that meet at least one condition.

variableA > 5 | variableB == TRUE translates to: retrieve values where variableA is greater than five or where the values of variableB are TRUE.

&

Translates to the word "and". `&` can be used to retrieve matching values that meet all conditions listed.

variableA > 5 & variableC == "male" translates to: retrieve values where variableA is greater than five and where the values of variableC equal "male"

3.4.4 Built-In Datasets

R has some datasets built-in. Like built-in symbols, some names refer to a built-in dataset that has already been defined. These datasets are usually used as examples for instructors to teach data management without having to provide a unique dataset. They are also sometimes used to ask questions so that anybody using R can follow along (if you have a specific kind of problem, it's going to be difficult for people who don't have access to your dataset to help you). If I refer to a dataset without specification, it's most likely a built-in dataset that you will already have access to. To see the built-in datasets, execute the following code:

```
data()
```

Some of the most common datasets I like to use for examples include **mtcars**, **diamonds**, & **midwest**. These built-in datasets sometimes require certain packages to be loaded into your library. **mtcars** is available by default; however, in order to use the **diamonds** and **midwest** datsets, the ggplot2 package is required. If you load the tidyverse package, the ggplot2 package will automatically load as well.

3.5 Saving

As you work on your script, don't forget to hit **Control + S** to periodically save your script's progress! This will only save the current script. If you have multiple scripts open, you should click on each individual tab to save each script. It is possible to "Save All Open Documents" with **Ctrl + Alt + S**, but I caution against this because you should always review the scripts before blindly saving.

When you decide to take a break from RStudio, whether it's a 5-min break or 24-hr break, **make sure to completely close out of the application before leaving your chair**. Trust me on this. If you work on multiple computers, R may create conflicted copies of the same files if you forget to save and close out the R script and session.

Finally, when closing out of RStudio, a message may pop up asking if you would like to save the workspace. The workspace is defined as the current working environment. Saving the workspace retains the user-defined objects for use upon the next R session. In my experience, we often feel better if we choose to hit save. In this case, its usually recommended against to save the workspace simply because users should always start with a clean, empty environment in each R session. Hit save or don't – just do yourself a favor and start with an empty environment each time you open R Studio (see Fig. 3.18)!

3.6 Troubleshooting Error Messages

In this section, we will go over how to troubleshoot errors – the most time consuming part of learning and coding in R. Some of the examples will contain functions that we will review in detail further into this guide. For now, we do not need to know what these functions do just yet. Follow along the examples by executing the code in each example.

3.6.1 Common Mistakes

What probably happened is that your problem occurred due to a very simple mistype or misunderstanding (read: user error).

Here are some of the most common mistakes that even seasoned R users make:

1. Capitalization

You typed an uppercase letter when you should have typed a lowercase letter (vice versa). Remember that capitalization matters!! `Library()` and `library()` are not the same. `Library()` will result in an error code.

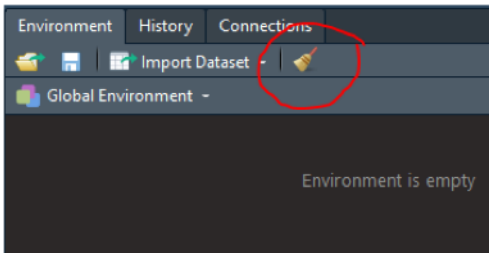


Figure 3.27: Clearing the environment history.

2. Mis-spelling

Unlike Microsoft Office, R will not highlight misspelled words. R will, however, accept both British and American spellings (i.e., color/colour, summarize/summarise, gray/grey)

3. Closing Punctuation

You forgot a closing parentheses, bracket, or quotation. All too often have I forgotten to add an additional parenthesis at the end of a line. You'll know that you've done this if you see a red X on the left side. The red X will appear as you are typing, so wait until you're finished to assess these warnings. Be wary of copying/pasting from different applications (from your internet browser, Microsoft Office, etc.) Occasionally, a copied quotation symbol will not be read properly by R.

4. Continuing Punctuation

You forgot to add a comma (,) or pipe (%>%) after your phrase before moving on to the next indented line of continued code. Indenting your code provides better readability for you (and others), but it can be easy to miss a comma or pipe:

Missing Punctuation

```
diamonds %>%
  mutate(price200 = price - 200
         price300 = price - 300) # missing pipe
  select(cut, price) %>%
  arrange(price)
```

Corrected Punctuation

```
diamonds %>%
  mutate(price200 = price - 200,
         price300 = price - 300) %>% # added pipe
  select(cut, price) %>%
  arrange(price)
```

5. Conflicting code

Perhaps you accidentally redefined an object when you didn't mean to. Try clearing the environment (click on the broomstick) and running a few lines of code at a time to see where the problem is.

6. Libraries Are Not Loaded

It is important to make sure that all of the pertinent libraries have been loaded during your R session. Each time you exit out of RStudio, the libraries are unloaded. Occasionally, R will spontaneously unload your libraries during your session as well, so if an error message reads that a function is not found, the usual solution is to reload your libraries.

7. The Unsaved Object

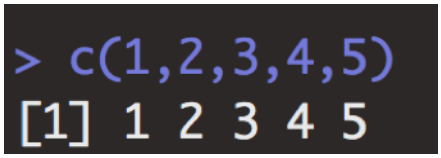


Figure 3.28: Concatenating 1,2,3,4,5.

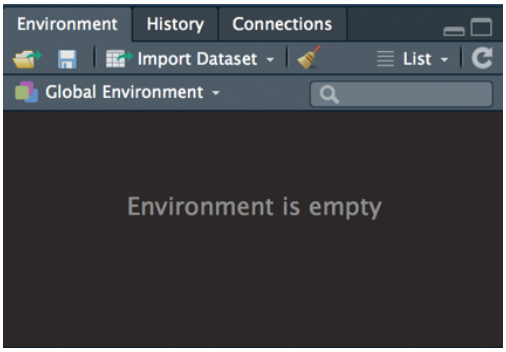


Figure 3.29: Empty environment.

As a refresher, the `<-` symbol represents "defines". The syntax is typically structured as such:

```
name.of.object <- "the definition of the object goes here"
```

It is often wise to "test" the definition before officially defining it as an object. That is, you should know what the definition is before you define it as an object. What if you accidentally defined an object poorly? If you later use this ill-defined object, additional errors are sure to follow. Let's see an example:

Let's say I want to define a new object called `one.to.five` that is defined as a list of numbers including: 1, 2, 3, 4, & 5. Before officially defining this object, I'm going to type out the definition and execute it (in the script).

Here, we see that I've concatenated (hence the `c()`) the numbers 1 through 5. I've typed this in the script.

```
c(1,2,3,4,5)
```

In the console, I see my executed code (in purple) and the output (in white) that the code produces (Figure 3.28):

Notice that the environment has not added a new object – it's still empty! (Figure 3.29)

Now, if I deem that this definition for my new object is correct, I will go ahead and name this as an official object. Notice that what I've done is provide the object name, the definition symbol (`<-`), and then the definition; see Figure **??**:

```
one.to.five <- c(1,2,3,4,5)
```

Unlike executing the definition alone, executing code that defines an object will not display the object's definition (i.e., there's no output that follows the executed code); see Figure 3.30:

Figure 3.30: Saving an object.

In order to see the definition/contents of the object, you must execute the object name by either typing out the name again and executing this code, or by highlighting the name in the script and executing the code (Figure **??**):

3.6.2 One Line at a Time

A good rule of thumb is to run a few lines at a time to identify the problematic code. The example below utilizes coding techniques that will be introduced much later in the guide. Here, it is important that you grasp the concept of how to run one line at a time, not so much the code content.

If executing the entire code below produces an error…

```
diamonds %>%
  mutate(price200 = price - 200,
         price20perc = price * .80) %>%
  group_by(cut, color) %>%
  summarize(total = count(),
            m1 = mean(price),
            m2 = mean(price200),
            m3 = mean(price20perc)) %>%
  ungroup()
```

I will highlight and execute one portion of the code at a time to detect the location of the error:

1. Execute

```
diamonds
```

2. Execute

```
diamonds %>%
  mutate(price200 = price - 200,
         price20perc = price * .80)
```

3. Execute

```
diamonds %>%
  mutate(price200 = price - 200,
         price20perc = price * .80) %>%
  group_by(cut, color)
```

4. Execute

```
diamonds %>%
  mutate(price200 = price - 200,
         price20perc = price * .80) %>%
  group_by(cut, color) %>%
  summarize(total = count(),
            m1 = mean(price),
            m2 = mean(price200),
            m3 = mean(price20perc))
```

5. Finally, execute the entire code again if no error has occurred

Typically, code in the lower lines rely on code from the top lines. That is, you cannot highlight lines 5-8 in the example without also highlighting lines 1-4 because the latter uses information from the former.

Notice that I only highlight up until (excluding) the next pipe (%>%) (Figure **??**). Recall that the pipe symbol represents a continuation. If you conclude on a pipe, R will think that you are not finished

typing. The pipe can conceptualized as the phrase "and then…":

> *"I am going to select **diamonds** as my dataset and then I'm going to mutate it and then I'm going to group the data by cut and color and then I'm going to summarize a few things and then I'm going to ungroup the data."*

In the above code, you'll find that the error occurs when the `summarize()` function is executed (lines 1-8), but not before. I now know that the problem lies somewhere in line 5 and can troubleshoot further. The same concept applies for plus signs (`+`) – do not end on a plus sign!

If you have accidentally executed code concluding on a pipe or a plus sign, simply click in the console region (bottom left panel in RStudio) and hit the esc button. R will reset your console, indicated by a new, empty line (`>`) and you can try executing code again.

3.6.3 Problems with Package Loading

Sometimes, for inexplicable reasons, a package will uninstall spontaneously. Perhaps it's because the package needs to be updated (newer versions have come out). Perhaps the R goblin stole it. I don't have all the answers. R will sometimes produce helpful error messages to let you know that your packages are the problem:

> Error: package or namespace load failed for 'PACKAGE.NAME.HERE' in loadNamespace(i, c(lib.loc, .libPaths()), versionCheck = vI[[i]]): there is no package called 'PACKAGE.NAME.HERE' In addition: Warning message: package 'PACKAGE.NAME.HERE' was built under R version 3.5.3

Notice how R told us that `PACKAGE.NAME.HERE` was the specific package that had a problem? (Obviously `PACKAGE.NAME.HERE` is not a real R package, it's just a placeholder name for this example).

The solution may be to:

1. **Try loading the package with `library()` again.** If the error message states that the package doesn't exist again, manually install the package with `install.packages("PACKAGE.NAME.HERE")`, replacing `PACKAGE.NAME.HERE` with the actualy package name. Then, load the package with `library()` again.
2. **Reinstall/update your base R.**
   - There are a few ways to do this, but the most straightforward way to update your R version is through the CRAN website (see page 6 of this guide). Make sure that all of your R-related windows are closed as you are downloading the new R version.
   - After installing the new version, RStudio should indicate that the version has changed in the console. You can also execute get `getRversion()`. After the installation of a new R version, you must reinstall all of your packages with `install.package()`.

3.6.4 Using the Internet to Your Advantage

My best advice for when you encounter error messages is to Google it. I would recommend for you to post the question on Stacked Overflow (an online forum for the community to post/answer coding problems).

Everybody – beginner and expert alike – will Google how to code something at some point. The real skill that experts have over beginners is that experts know how to strategically Google. For that, you want to master these skills:

- Make sure that your question is specific and reproducible.
  - I recommend using a built-in dataset to illustrate your problem.

  - Don't forget to include "R" as a keyword during the search.

- Make sure you understand the terminology well.
  - Because this book is designed with the **tidyverse** package in mind, I recommend adding phrases such as "with dplyr" or "with tidyverse" in your search.
  - **dplyr** is a package inside **tidyverse** that provides "tidyverse-style" coding

For example, let's say that I want to know how to rename a column in my dataset. I could Google: "How to rename a column in R with dplyr/tidyverse" and read the answers posted in Stacked Overflow (www.stackoverflow.com).

Notice how I covered the following in my google search:

- The specific action (how to rename a column)
- The programming language (R statistics)
- The specific style/technique for coding (**dplyr** or **tidyverse** package)

When reading the answers to these questions posted on Stacked Overflow, consider the parent question that the original poster asked. A good Stacked Overflow question contains a reproducible example for which you can directly follow along with to see if the answers listed make sense. Not all answers will be useful and not all users will utilize "tidyverse-style" coding. If you cannot find the answer to your question, you may choose to post it yourself on Stacked Overflow. Just be sure to follow the guidelines mentioned above for what to include in your question!

3.6.5 When Google Fails You

Sometimes you just can't find an answer.

In rare (but real) circumstances, I have found that RStudio can run into some issues that have nothing to do with your code. If your code is running strangely, the age-old trick of **turning it off and on again** (i.e., RStudio or your computer) occasionally does work. Make sure you save the information you need to save before you close RStudio or restart your computer. Upon reopening RStudio, make sure the Global Environment is clean before attempting to execute code.

Another option is to **take a break.** During that period, little elves come and fix your computer for you. If you don't leave, they'll never show up. In all seriousness, error messages can be anger-inducing and in your cloud of frustration, you may not be able to think clearly. Step away for a bit and look at it from a different emotional state.

Finally, **ask someone for help**. An experienced R user should be able to figure out what went wrong pretty quickly. This is because experts have made (and hopefully fixed) 100X more errors than beginners; many people struggle with the same problems at the beginning. Problem solving is (in my opinion) the best way to learn. However, it's also the most time consuming. Ask for help when you need it. Time is an invaluable, nonrenewable resource.

Part II

Tidyverse

Chapter 4

Introduction to Tidyverse

The tidyverse package actually contains other packages (**dplyr**, **ggplot2**, etc.) and you'll see that when you load the **tidyverse** package using `library()`. Remember the package must be installed to your device before it can be loaded into your libraries! For help on installing packages, refer to Section 3.2.3.

```
library(tidyverse)
```

You'll also notice that there are some functions that are masked after loading **tidyverse** (Fig. 4.1). To understand what this means, let's take a look at our "Packages" tab in the bottom right window of RStudio.

My packages list will be slightly different from yours. **The point is:** there are many different packages installed on your computer.

Recall that packages consist of a collection of functions related to a particular purpose (e.g., **ggplot2** contains functions for graphing). We know that packages must be loaded to your libraries each time a new RStudio session begins and installed once per R version update (Section 3.2.3). When a package is loaded (with `library()`), R will import the **entire** package (all functions, documentation, built-in datasets) to your current workspace.

Why doesn't R just keep all packages always loaded to begin with? Why do we have to load the package libraries for every RStudio session? The reason is simple: packages often have naming conflicts with other packages. While package authors attempt to uniquely name their functions, concise and descriptive names are limited and naming conflicts with functions from other packages are inevitable. For example, if you execute `?filter()` after loading the **tidyverse** package, two help pages pop up: one from the **dplyr** package and one from the **stats** package. These functions work differently and it is important to know which function you are using in your code.

How does R pick which package's definition to use for `filter()`? It decides based on which package was loaded most recently. The **stats** package is a base R package pre-loaded at the beginning of your R session.

```
> library(tidyverse)
-- Attaching packages --------------------------------------- tidyverse 1.2.1 --
v ggplot2 3.2.0     v purrr   0.3.2
v tibble  2.1.3     v dplyr   0.8.1
v tidyr   0.8.3     v stringr 1.4.0
v readr   1.3.1     v forcats 0.4.0
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
>
```

Figure 4.1: Tidyverse loading messages in the console.

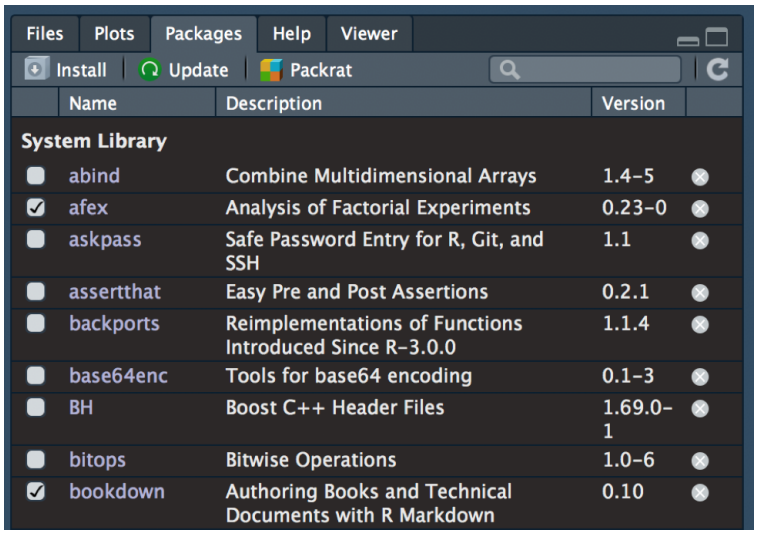


Figure 4.2: Partial list of currently installed packages.

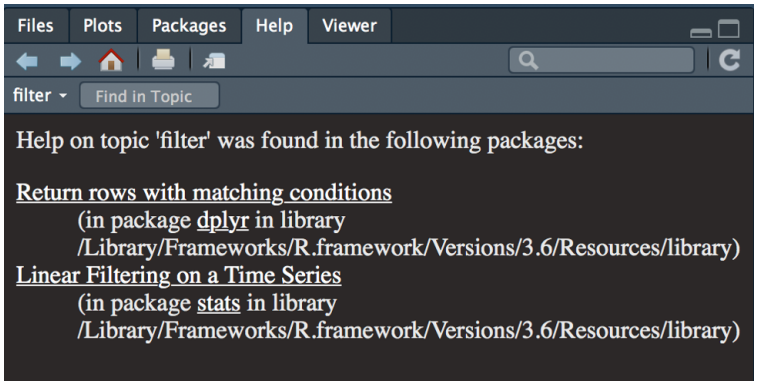


Figure 4.3: Different packages with same function names.

When you load the tidyverse package, which also loads the dplyr package, the definition of `filter()` switches over to the **dplyr** version.

To prevent accidental usage of the undesired function, some R users prefer to **load individual functions** as opposed to loading an entire library. If all you need to use from the **dplyr** package is the `filter()` function, you could execute the following:

```
## loads ONLY the filter function from dplyr
dplyr::filter()

## loads ALL functions from dplyr, then using filter()
library(dplyr)
filter()

# Functions in this example are empty for illustration purposes.
# Usually, use of functions will require arguments inside.
```

Loading an individual function from a package will not change how the function itself works; all of the required arguments will be the same (for a refresher on functions, see 3.3.8). This method of loading individual functions can be tedious if you plan on using the function multiple times or if you need other functions from the same package. Thus, loading single functions is generally used sparingly.

Finally, let's briefly talk about **the organization/structure of this chapter.** Many examples in this chapter will display the code written/executed followed by the output printed in the console. If you follow along and execute the code, the output shown in the book should match the output in your console. The code and output are located in separate blocks and can be differentiated by the color coordination (R code will contain selective color coding) and comments (an output will contain **only** comments using ##; the output on your computer will not display the ## symbols).

Example

```
## This is an example code
diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price)) %>%
  ungroup()
```

```
## # A tibble: 8 x 2
##   clarity     m
##   <ord>   <dbl>
## 1 I1      3924.
## 2 SI2     5063.
## 3 SI1     3996.
## 4 VS2     3925.
## 5 VS1     3839.
## 6 VVS2    3284.
## 7 VVS1    2523.
## 8 IF      2865.
```

Experiencing errors? Refer back to the Troubleshooting section (3.6)!

Tidyverse has lots of useful functions, but this chapter will introduce the basic ones. **Follow along by executing the example code in the upcoming sections**.

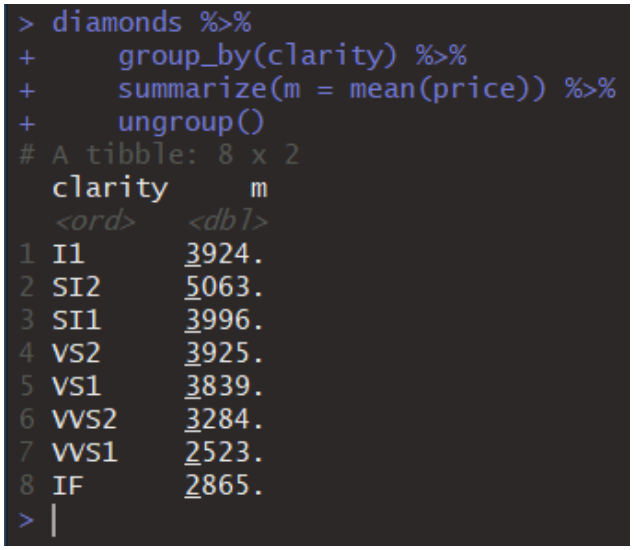


Figure 4.4: Image of actual output in the RStudio console.

Chapter 5

The `diamonds` dataset

Throughout this chapter, we will be working with the `diamonds` dataset, so let's take a minute to familiarize ourselves with it. This built-in dataset is available when the **ggplot2** package is loaded. Loading the **tidyverse** package will automatically load **ggplot2**.

Let's view the `diamonds` dataset in a separate RStudio tab:

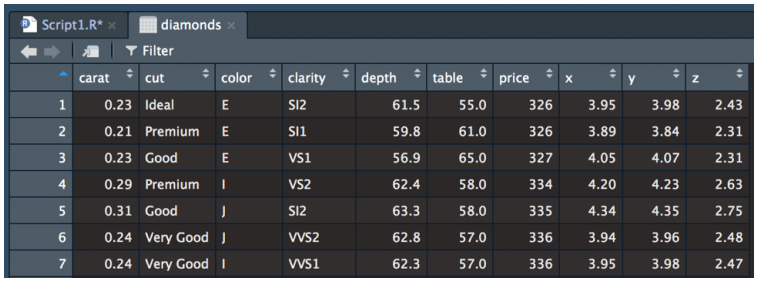
```
View(diamonds)
```

You can view any object in a new tab by wrapping the `View()` function around the object name. As a beginner in learning R, viewing the dataset in a familiar Excel-like format can be comforting. However, with more practice, viewing the dataset in this manner becomes less useful (especially when working with really big datasets). Unlike Excel, you cannot edit your data directly cell-by-cell in RStudio. Instead, every action must be explicitly specified in your code.

Next, let's look at the structure of each variable in `diamonds` (see 3.3.10 for a refresher on structures):

```
str(diamonds)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
```

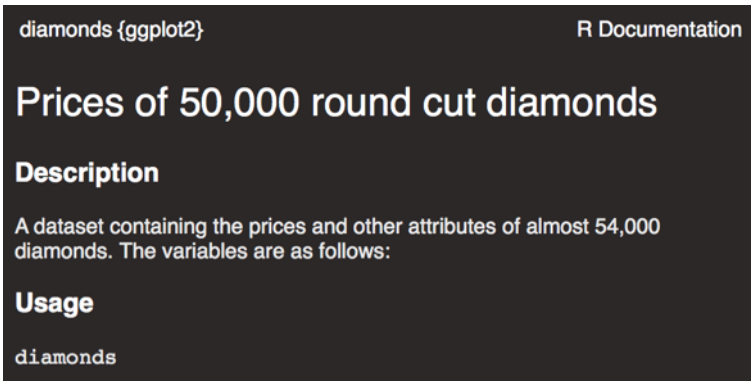


	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
6	0.24	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
7	0.24	Very Good	I	VVS1	62.3	57.0	336	3.95	3.98	2.47

Figure 5.1: Viewing 'diamonds' using View().

```
##  $ z       : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

Here, we see that there are 10 total variables (three ordered factors, one integer, and 6 numeric). An added bonus of working with a built-in dataset is that documentation giving further descriptions and explanations is available via the help page (`?diamonds`).


diamonds {ggplot2} R Documentation

Prices of 50,000 round cut diamonds

Description

A dataset containing the prices and other attributes of almost 54,000 diamonds. The variables are as follows:

Usage

diamonds

Here's what we know about the diamonds dataset:

- This dataset contains information about 53,940 round-cut diamonds. How do we know? Each row of data represents a different diamond and there are 53,940 rows of data (see help page, `?diamonds`)
- There are 10 variables measuring various pieces of information about the diamonds. Notice that these variable names are in lowercase. We can take a quick view of the variable names using:

```
names(diamonds)
```

- There are 3 variables with an ordered factor structure: `cut`, `color`, & `clarity`. An ordered factor arranges the categorical values in a low-to-high rank order. For example, there are 5 categories of diamond cuts with "Fair" being the lowest grade of cut to ideal being the highest grade.
- There are 6 variables that are of numeric structure: `carat`, `depth`, `table`, `x`, `y`, `z`
- There is 1 variable that has an integer structure: `price`

Remember that R will always have documentation (in the help page; `?diamonds`) for built-in datasets. The descriptiveness for the documentation will vary, depending on the package author.

Variable

Description

Values

price

price in US dollars

$326-$18,823

carat

weight of the diamond

0.2-5.01

cut

quality of the cut

Fair, Good, Very Good, Premium, Ideal

color

diamond color

J (worst) to D (best)

clarity

measurement of how clear the diamond is

I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best)

x

length in mm

0-10.74

y

width in mm

0-58.9

z

depth in mm

0-31.8

depth

total depth percentage

43-79

table

width of top of diamond relative to widest point

43-95

Don't forget that **R cares about capitalization** (`diamonds` is not the same as `Diamonds` and `price` is not the same as `Price`)!

Chapter 6

Basic Data Management

6.1 mutate()

What it does: Adds new columns or modifies current variables in the dataset.

Let's say I want to create three new variables in the `diamonds` dataset (described in Section5) :

1. One variable called `JustOne` where all of the values inside the column are `1`.
2. One variable called `Values` where all of the values inside are `something`:
3. One variable called `Simple` where all the values equal `TRUE`

```
diamonds %>%
  mutate(JustOne = 1,
         Values = "something",
         Simple = TRUE)
```

```
## # A tibble: 53,940 x 13
##    carat cut   color clarity depth table price     x     y     z JustOne
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>   <dbl>
##  1 0.23  Ideal E     SI2      61.5    55   326  3.95  3.98  2.43       1
##  2 0.21  Prem~ E     SI1      59.8    61   326  3.89  3.84  2.31       1
##  3 0.23  Good  E     VS1      56.9    65   327  4.05  4.07  2.31       1
##  4 0.290 Prem~ I     VS2      62.4    58   334  4.2   4.23  2.63       1
##  5 0.31  Good  J     SI2      63.3    58   335  4.34  4.35  2.75       1
##  6 0.24  Very~ J     VVS2     62.8    57   336  3.94  3.96  2.48       1
##  7 0.24  Very~ I     VVS1     62.3    57   336  3.95  3.98  2.47       1
##  8 0.26  Very~ H     SI1      61.9    55   337  4.07  4.11  2.53       1
##  9 0.22  Fair  E     VS2      65.1    61   337  3.87  3.78  2.49       1
## 10 0.23  Very~ H     VS1      59.4    61   338  4     4.05  2.39       1
## # ... with 53,930 more rows, and 2 more variables: Values <chr>,
## #   Simple <lgl>
```

Need a refresher on how a pipe (%>%) works? Go back to section 3.3.9!

`mutate()` can be used to create variables **based on existing variables** from the dataset.

Let's use the existing variable `price` from the `diamonds` dataset to create a new column/variable named `price200`. The values for `price200` calculated from `price` minus $200 (`price` is a default variable in `diamonds`):

```
diamonds %>%
  mutate(price200 = price - 200)
```

You can also create multiple columns at once, separating each new variable with a comma.

```
diamonds %>%
  mutate(price200 = price - 200,         # $200 OFF from the original price
         price20perc = price * 0.20,     # 20% of the original price
         price20percoff = price * 0.80, # 20% OFF from the original price
         pricepercarat = price / carat, # ratio of price to carat
         pizza = depth ^ 2)             # Square the original depth
```

Notice that you can label the variables/new columns however you want (refer to 3.4.2 for naming conventions). In the example above, `pizza` is not a descriptive name for the new variable defined as the `depth` values squared. Make it easier for Future You to navigate your old code: choose simple, descriptive names.

Currently, the changes that we've made to the `diamonds` dataset have not been saved. We can see that `diamonds` does not have the newly created variables (`price200`, `price20perc`, `pizza`) when we execute `str(diamonds)`. In order to save your changes, you must define the new dataset as an object using `<-`! In the code below, the modified `diamonds` dataset will be saved as a new object/dataset called `diamonds.new`. As a beginner, it is good practice to save your changes to a dataset under a new object name (e.g., `diamonds.new`) each time you make changes rather than saving over the original dataset name (e.g., `diamonds`). See 3.3.7 for more information on saving objects.

```
diamonds.new <- # saving changes to diamonds as a new object
  diamonds %>%  # original dataset
  mutate(price200 = price - 200,         # $200 OFF from the original price
         price20perc = price * .20,      # 20% of the original price
         price20percoff = price * 0.80, # 20% OFF from the original price
         pricepercarat = price / carat, # ratio of price to carat
         pizza = depth ^ 2)             # Square the original depth
```

6.1.1 Nesting Functions

We can also use other functions inside `mutate()` to create our new variable(s). For example, we might use the `mean()` function to calculate the average `price` value for all diamonds in the dataset. This is called **nesting**, where one function (`mean()`) "nests" inside another function (`mutate()`).

```
diamonds %>%
  mutate(m = mean(price))
```

Nearly all of the functions used in this book are available by default or are loaded with `library(tidyverse)`. However, you may come across many other kinds of functions that come from different packages. **Make sure that the relevant packages are loaded in your library!**

The example below creates four new columns to the `diamonds` dataset that calculate the mean, standard deviation, standard error, and median `price` value for **all** diamonds. The values in these columns will be the same for every row because R takes *all* of the values in `price` to calculate the mean/standard deviation/median.

```
diamonds %>%
  mutate(m = mean(price),      # calculates the mean price
         sd = sd(price),       # calculates standard deviation
         med = median(price)) # calculates the median price
```

6.1.1.0.1 Exercises

1. Using the built-in dataset, `midwest` (make sure **tidyverse** is loaded!). Each exercise comes with the desired output! That means you get the answer to each question but not **how** to get to the answer. To view the newly created variable, scroll all the way over to the right side of the output.

a. Create a column named `avg.pop.den` which calculates average population density for the entire dataset (hint: use `mean()` and `popdensity`; all values in this column should be the same – Why?)

```
## # A tibble: 437 x 29
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 21 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   avg.pop.den <dbl>
```

b. Create a column named `avg.area` which calculates the average area for the entire dataset

```
## # A tibble: 437 x 29
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 21 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   avg.area <dbl>
```

c. Create a column called `totadult` which calculates the total number of adults in this dataset (hint: use `popadults`, `sum()`; requires nesting)

```
## # A tibble: 437 x 29
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 21 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   totadult <int>
```

d. Create a new column called `tot.minus.white` calculating the difference between `poptotal` and `popwhite`. Using other variables in the dataset, how else could you create the `tot.minus.white` column without using `poptotal` and `popwhite`. (hint: `popblack`, `popamerindian`, `popasian`, `popother`)?

```
## # A tibble: 437 x 30
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 22 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   tot.minus.white <int>, tot.minus.white2 <int>
```

e. Create a new column called `child.to.adult` that calculates the ratio of `percchildbelowpovert` to `percadultpoverty` (i.e., for every adult that is in poverty, what proportion of children are in poverty? Compare to the question from 1a: Why are the values in `child.to.adult` all different?)

```
## # A tibble: 437 x 29
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
```

```
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 21 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   child.to.adult <dbl>
```

f. Create a new column named `ratio.adult` which calculates the ratio of adults in this dataset (hint: `popadults`, `poptotal`).

```
## # A tibble: 437 x 29
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 21 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   ratio.adult <dbl>
```

g. Create a new column named `perc.adult` that calculates the percentage of the total population that are adults per county (hint: build from 1f; `* 100`).

```
## # A tibble: 437 x 29
##      PID county state  area poptotal popdensity popwhite popblack
##    <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>
##  1   561 ADAMS  IL    0.052    66090      1271.    63917     1702
##  2   562 ALEXA~ IL    0.014    10626       759      7054     3496
##  3   563 BOND   IL    0.022    14991       681.    14477      429
##  4   564 BOONE  IL    0.017    30806      1812.    29344      127
##  5   565 BROWN  IL    0.018     5836       324.     5264      547
##  6   566 BUREAU IL    0.05     35688       714.    35157       50
```

```
##  7   567 CALHO~ IL    0.017     5322       313.     5298        1
##  8   568 CARRO~ IL    0.027    16805       622.    16519      111
##  9   569 CASS   IL    0.024    13437       560.    13384       16
## 10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559
## # ... with 427 more rows, and 21 more variables: popamerindian <int>,
## #   popasian <int>, popother <int>, percwhite <dbl>, percblack <dbl>,
## #   percamerindan <dbl>, percasian <dbl>, percother <dbl>,
## #   popadults <int>, perchsd <dbl>, percollege <dbl>, percprof <dbl>,
## #   poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,
## #   percelderlypoverty <dbl>, inmetro <int>, category <chr>,
## #   perc.adult <dbl>
```

2. Using the built-in dataset, `presidential`, create a column named `duration` that calculates the total number of days each president held in office.

```
## # A tibble: 11 x 5
##    name       start      end        party      duration
##    <chr>      <date>     <date>     <chr>      <drtn>
##  1 Eisenhower 1953-01-20 1961-01-20 Republican 2922 days
##  2 Kennedy    1961-01-20 1963-11-22 Democratic 1036 days
##  3 Johnson    1963-11-22 1969-01-20 Democratic 1886 days
##  4 Nixon      1969-01-20 1974-08-09 Republican 2027 days
##  5 Ford       1974-08-09 1977-01-20 Republican  895 days
##  6 Carter     1977-01-20 1981-01-20 Democratic 1461 days
##  7 Reagan     1981-01-20 1989-01-20 Republican 2922 days
##  8 Bush       1989-01-20 1993-01-20 Republican 1461 days
##  9 Clinton    1993-01-20 2001-01-20 Democratic 2922 days
## 10 Bush       2001-01-20 2009-01-20 Republican 2922 days
## 11 Obama      2009-01-20 2017-01-20 Democratic 2922 days
```

3. Using the built-in dataset, `economics`, create a column named `perc.unemploy` that calculates the percentage of the population that is unemployed.

```
## # A tibble: 574 x 7
##    date         pce    pop psavert uempmed unemploy perc.unemploy
##    <date>     <dbl>  <int>   <dbl>   <dbl>    <int>         <dbl>
##  1 1967-07-01  507. 198712    12.5     4.5     2944          1.48
##  2 1967-08-01  510. 198911    12.5     4.7     2945          1.48
##  3 1967-09-01  516. 199113    11.7     4.6     2958          1.49
##  4 1967-10-01  513. 199311    12.5     4.9     3143          1.58
##  5 1967-11-01  518. 199498    12.5     4.7     3066          1.54
##  6 1967-12-01  526. 199657    12.1     4.8     3018          1.51
##  7 1968-01-01  532. 199808    11.7     5.1     2878          1.44
##  8 1968-02-01  534. 199920    12.2     4.5     3001          1.50
##  9 1968-03-01  545. 200056    11.6     4.1     2877          1.44
## 10 1968-04-01  545. 200208    12.2     4.6     2709          1.35
## # ... with 564 more rows
```

4. Using the built-in dataset, `txhousing`,

a. Create a column named `successrate` that calculates the percent of houses that sell of the total listings available (hint: `sales`, `listings`).

```
## # A tibble: 8,602 x 10
##    city   year month sales volume median listings inventory  date
##    <chr> <int> <int> <dbl>  <dbl>  <dbl>    <dbl>     <dbl> <dbl>
```

```
##  1 Abil~  2000     1    72 5.38e6  71400      701       6.3 2000
##  2 Abil~  2000     2    98 6.50e6  58700      746       6.6 2000.
##  3 Abil~  2000     3   130 9.28e6  58100      784       6.8 2000.
##  4 Abil~  2000     4    98 9.73e6  68600      785       6.9 2000.
##  5 Abil~  2000     5   141 1.06e7  67300      794       6.8 2000.
##  6 Abil~  2000     6   156 1.39e7  66900      780       6.6 2000.
##  7 Abil~  2000     7   152 1.26e7  73500      742       6.2 2000.
##  8 Abil~  2000     8   131 1.07e7  75000      765       6.4 2001.
##  9 Abil~  2000     9   104 7.62e6  64500      771       6.5 2001.
## 10 Abil~  2000    10   101 7.04e6  59300      764       6.6 2001.
## # ... with 8,592 more rows, and 1 more variable: successrate <dbl>
```

b. Create a column called `failrate` that calculates the percent of houses that **do not** sell of the total listings available (hint: build from 4a).

```
## # A tibble: 8,602 x 10
##    city    year month sales volume median listings inventory  date failrate
##    <chr>  <int> <int> <dbl>  <dbl>  <dbl>    <dbl>     <dbl> <dbl>    <dbl>
##  1 Abile~  2000     1    72 5.38e6  71400      701       6.3 2000      89.7
##  2 Abile~  2000     2    98 6.50e6  58700      746       6.6 2000.     86.9
##  3 Abile~  2000     3   130 9.28e6  58100      784       6.8 2000.     83.4
##  4 Abile~  2000     4    98 9.73e6  68600      785       6.9 2000.     87.5
##  5 Abile~  2000     5   141 1.06e7  67300      794       6.8 2000.     82.2
##  6 Abile~  2000     6   156 1.39e7  66900      780       6.6 2000.     80
##  7 Abile~  2000     7   152 1.26e7  73500      742       6.2 2000.     79.5
##  8 Abile~  2000     8   131 1.07e7  75000      765       6.4 2001.     82.9
##  9 Abile~  2000     9   104 7.62e6  64500      771       6.5 2001.     86.5
## 10 Abile~  2000    10   101 7.04e6  59300      764       6.6 2001.     86.8
## # ... with 8,592 more rows
```

Let's take a look at some other useful functions.

6.1.1.1 `recode()`

What it does: modifies the values within a variable. Here is the basic structure for using `recode`:

```
data %>% mutate(Variable = recode(Variable, "old value" = "new value"))
```

Example

```
diamonds %>%
  mutate(cut.new = recode(cut,
                          "Ideal" = "IDEAL"))
```

In the code below, we created a new variable named `cut.new` that is defined as the original `cut` variable except that values originally listed as `"Ideal"` now read `"IDEAL"`.

We can also recode multiple values at a time.

```
diamonds %>%
  mutate(cut.new = recode(cut,
                          "Ideal" = "IDEAL",
                          "Fair" = "Okay",
                          "Premium" = "pizza"))
```

Of course, this new edit is intended to be a bit silly. When recoding data, it is important to be descriptive so that Future You can navigate your code. Again, remember that the changes made to `diamonds` in the above code have not been saved to the dataset; `diamonds` will not contain the `cut.new` variable.

Most commonly, I use `recode()` to fix inconsistent labeling.

Example

```
# creating a dataset with 2 variables (Sex , TestScore)
Sex <- factor(c("male", "m", "M", "Female", "Female", "Female"))
TestScore <- c(10, 20, 10, 25, 12, 5)
dataset <- tibble(Sex, TestScore)
str(dataset)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    6 obs. of  2 variables:
##  $ Sex      : Factor w/ 4 levels "Female","m","M",..: 4 2 3 1 1 1
##  $ TestScore: num  10 20 10 25 12 5
```

We see that `Sex` has 4 categories (Female, m, M, and male). We know that m, M, and male are not three different categorical values. We meant to have 2 categories representing the two most common biolgical sexes - male and female. Let's recode all of the male values in `Sex` to match the female equivalent value, `Female`.

```
# creating a new variable (Sex.new) with recoded values
# from the original variable (Sex)
dataset %>%
  mutate(Sex.new = recode(Sex,
                          "m" = "Male",
                          "M" = "Male",
                          "male" = "Male"))
```

```
## # A tibble: 6 x 3
##   Sex    TestScore Sex.new
##   <fct>      <dbl> <fct>
## 1 male          10 Male
## 2 m             20 Male
## 3 M             10 Male
## 4 Female        25 Female
## 5 Female        12 Female
## 6 Female         5 Female
```

Notice that there are now three variables: `Sex`, `TestScore`, and `Sex.new`. The `Sex.new` variable contains two categorical values: Male and Female. You must save this new dataset's edits as an object (using `<-`) to save these changes! Otherwise, the next time you use `dataset`, the `Sex.new` variable will not be available.

```
dataset.new <- # saving the changes to a new object
  dataset %>%
  mutate(Sex.new = recode(Sex,
                          "m" = "Male",
                          "M" = "Male",
                          "male" = "Male"))
str(dataset.new)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    6 obs. of  3 variables:
##  $ Sex      : Factor w/ 4 levels "Female","m","M",..: 4 2 3 1 1 1
##  $ TestScore: num  10 20 10 25 12 5
##  $ Sex.new  : Factor w/ 2 levels "Female","Male": 2 2 2 1 1 1
```

The original object, `dataset` remains the same as before. Only `dataset.new` includes the additional variable, `Sex.new`.

```
str(dataset)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    6 obs. of  2 variables:
```

```
##  $ Sex      : Factor w/ 4 levels "Female","m","M",..: 4 2 3 1 1 1
##  $ TestScore: num  10 20 10 25 12 5
```

6.2 summarize()

What it does: collapses all rows and returns a one-row summary. R will recognize both the British and American spelling (summarise/summarize).

In the following **example**, `summarize()` allows us to calculate the mean price of all of the diamonds within our dataset.

```
diamonds %>%
  summarize(avg.price = mean(price))
```

```
## # A tibble: 1 x 1
##   avg.price
##       <dbl>
## 1     3933.
```

Similar to `mutate()`, we can also perform multiple operations with `summarize()` and nest other useful functions inside it:

```
diamonds %>%
  summarize(avg.price = mean(price),      # average price of all diamonds
            dbl.price = mean(price) * 2, # calculating double the average price
            random.add = 1 + 2,           # a math operation without an existing variable
            avg.carat = mean(carat),      # average carat size of all diamonds
            stdev.price = sd(price))      # calculating the standard deviation
```

```
## # A tibble: 1 x 5
##   avg.price dbl.price random.add avg.carat stdev.price
##       <dbl>     <dbl>      <dbl>     <dbl>       <dbl>
## 1     3933.     7866.          3     0.798       3989.
```

6.3 group_by() and ungroup()

Takes existing data and groups specific variables together for future operations. Many operations are performed on groups.

Example: Grouping by age and sex (male/female) might be useful in a dataset if we care about how females of a certain age scored compared to males of a certain age (or comparing ages within males or within females).

Let's create a sample dataset to reflect this example (to avoid entry errors, copy and paste this into your script):

```
## Creating identification number to represent 50 individual people
ID <- c(1:50)

## Creating sex variable (25 males/25 females)
Sex <- rep(c("male", "female"), 25) # rep stands for replicate

## Creating age variable (20-39 year olds)
Age <- c(26, 25, 39, 37, 31, 34, 34, 30, 26, 33,
         39, 28, 26, 29, 33, 22, 35, 23, 26, 36,
         21, 20, 31, 21, 35, 39, 36, 22, 22, 25,
```

```
         27, 30, 26, 34, 38, 39, 30, 29, 26, 25,
         26, 36, 23, 21, 21, 39, 26, 26, 27, 21)

## Creating a dependent variable called Score
Score <- c(0.010, 0.418, 0.014, 0.090, 0.061, 0.328, 0.656, 0.002, 0.639, 0.173,
           0.076, 0.152, 0.467, 0.186, 0.520, 0.493, 0.388, 0.501, 0.800, 0.482,
           0.384, 0.046, 0.920, 0.865, 0.625, 0.035, 0.501, 0.851, 0.285, 0.752,
           0.686, 0.339, 0.710, 0.665, 0.214, 0.560, 0.287, 0.665, 0.630, 0.567,
           0.812, 0.637, 0.772, 0.905, 0.405, 0.363, 0.773, 0.410, 0.535, 0.449)

## Creating a unified dataset that puts together all variables
data <- tibble(ID, Sex, Age, Score)
```

6.3.1 summarize() and group_by()

Let's say that I want calculate/compare the average `Score` (and other measures) for males and females separately:

```
data %>%
  group_by(Sex) %>%
  summarize(m = mean(Score), # calculates the mean
            s = sd(Score),   # calculates the standard deviation
            n = n()) %>%     # calculates the total number of observations
  ungroup()
```

```
## # A tibble: 2 x 4
##   Sex        m     s     n
##   <chr>  <dbl> <dbl> <int>
## 1 female 0.437 0.268    25
## 2 male   0.487 0.268    25
```

In the above code, we have grouped by `Sex`, meaning that calculations performed on our data will account for males and females separately. Following code execution, the console displays the mean `Score`, the standard deviation (`sd`), and the total number of participants (`n()`) for females and for males (`group_by(Sex)`). That is, the average `Score` for females is 0.437 and the average `Score` for males is 0.487.

Let's group by `Sex` **and** `Age` next (the order in which the variables appear withint `group_by()` doesn't matter):

```
data %>%
  group_by(Sex, Age) %>%     # grouped by Sex and Age
  summarize(m = mean(Score),
            s = sd(Score),
            n = n()) %>%
  ungroup()
```

```
## # A tibble: 27 x 5
##    Sex      Age     m      s     n
##    <chr>  <dbl> <dbl>  <dbl> <int>
##  1 female    20 0.046 NA         1
##  2 female    21 0.740  0.253     3
##  3 female    22 0.672  0.253     2
##  4 female    23 0.501 NA         1
##  5 female    25 0.579  0.167     3
##  6 female    26 0.41  NA         1
```

```
##  7 female    28 0.152 NA         1
##  8 female    29 0.426  0.339     2
##  9 female    30 0.170  0.238     2
## 10 female    33 0.173 NA         1
## # ... with 17 more rows
```

There are now considerably more rows (27 rows) in this output. When performing calculations, R now considers each combination of `Age` and `Sex`. For example, the average 25-year-old female had a score of 0.579.

We also see that there are some missing standard deviation values (NaN). This is because calculating standard deviation requires **more than one** participant/observation.

6.3.2 mutate() and group_by()

We could also utilize `mutate()` after `group_by()` to add a new column based on the group.

```
data %>%
  group_by(Sex) %>%
  mutate(m = mean(Score)) %>% # calculates mean score by Sex
  ungroup()
```

```
## # A tibble: 50 x 5
##       ID Sex      Age Score     m
##    <int> <chr>  <dbl> <dbl> <dbl>
##  1     1 male      26 0.01  0.487
##  2     2 female    25 0.418 0.437
##  3     3 male      39 0.014 0.487
##  4     4 female    37 0.09  0.437
##  5     5 male      31 0.061 0.487
##  6     6 female    34 0.328 0.437
##  7     7 male      34 0.656 0.487
##  8     8 female    30 0.002 0.437
##  9     9 male      26 0.639 0.487
## 10    10 female    33 0.173 0.437
## # ... with 40 more rows
```

Instead of collapsing all rows to a summary value, `mutate()` adds a new column (`m`) containing the average male and female score. The averaged scores in column `m` correspond to the value in the `Sex` column (0.487 for males and 0.437 for females).

6.3.3 Ungrouping

Notice that `ungroup()` is always used after the `group()` command after performing calculations. If you forget to `ungroup()` data, future data management will likely produce errors. **Always `ungroup()` when you've finished with your calculations.**

Let's see an example of when ungrouping matters:

```
## Example 1

data %>%
  group_by(Sex) %>%
  mutate(m = mean(Age)) %>%   # calculates the average age of males and females
```

```
  mutate(x = mean(Score)) %>% # counts number of participants
  ungroup()                   # closing ungroup()
```

Compare this with code that includes `ungroup()` nested between the two `mutate()` functions:

```
## Example 2

data %>%
  group_by(Sex) %>%
  mutate(m = mean(Age)) %>%  # calculates the average age of males and females
  ungroup() %>%              # nested ungroup()
  mutate(x = mean(Score))    # counts number of participants
```

In the first example, `m`, which calculates mean `Age`, is either 29.2 if the participant is male or 28.96 if the participant is female. `x`, which calculates mean `Score`, is 0.487 for males and 0.437 for females. For both calculations, the data is grouped by `Sex`.

In the second example, `m` still calculates the average `Age` for males separate from females as in the first example. However, `x` equals a `Score` of 0.462 for every row/observation. This is because `group_by(Sex)` is removed via `ungroup()` after the first `mutate()` function. Here, `x` calculates the mean `Score` for **all participants** together.

Neither method is right or wrong – it depends on what you're trying to achieve. When deciding where to place the `ungroup()` function, ask yourself: Does it make sense to calculate different values for this `Variable`? If so, the `group_by(Variable)` function should be written *before* the calculation function (mutate/summarize).

If you use `group_by()`, you must have a matching `ungroup()` somewhere. Even if you do not plan on performing additional calculations, it's a good habit to keep. Making sure that you `ungroup()` is **especially important when creating objects**!!

```
## Creating/Saving the object named "data1"
data1 <-
  data %>%
  group_by(Sex) %>%
  mutate(m = mean(Age))

## Using the data1 object after it's been saved above (WITHOUT an ungroup)
data1 %>%
  mutate(x = mean(Score))
```

Anytime you use `data1`, a saved object, it will automatically have `group_by(Sex)` as part of its definition and further calculations will account for these grouping variables.

Forgetting to `ungroup()` can get even more complex!

```
## Creating/Saving the object named "data1"
data1 <-
  data %>%
  group_by(Sex) %>%
  mutate(m = mean(Age))

## Using the data1 object after it's been saved above (WITHOUT an ungroup)
data1 %>%
  group_by(Age) %>%
  mutate(x = mean(Score)) %>%
  ungroup()
```

Now, the second code chunk is actually grouping by `Sex` **and** `Age`. That is, the `x` variable calculates the

mean `Score` for each combination of `Sex` and `Age`. Even if this is what you wanted, it's best to specify this on each line. As your scripts increase in length, it can become difficult to recall the specifics about object definitions, especially if it involves a special grouping variable. Keep your objects as simple as possible!

Here is the proper method in which to save and use the `data1` object:

```
data1 <-
  data %>%
  group_by(Sex) %>%
  mutate(m = mean(Age)) %>%
  ungroup() # Ungroup at the end of a definition!!!

data1 %>%
  group_by(Sex, Age) %>%  # group the relevant variables here
  mutate(x = mean(Score)) %>%
  ungroup()
```

In conclusion, **ALWAYS UNGROUP AFTER GROUPING**

6.4 filter()

Function: Only retain specific rows of data that meet the specified requirement(s).

Only display data from the `diamonds` dataset that have a `cut` value of Fair:

```
diamonds %>% filter(cut == "Fair")
```

Only display data from diamonds that have a `cut` value of Fair or Good and a price at or under $600 (notice how the or statement is obtained with | while the and statement is achieved through a comma):

```
diamonds %>%
  filter(cut == "Fair" | cut == "Good", price <= 600)
```

An alternative method that achieves the same outcome:

(read as: in the diamonds dataset, show me `cut` values that match what's inside the vector `c("Fair", "Good")`)

```
diamonds %>%
  filter(cut %in% c("Fair", "Good"), price <= 600)
```

The following code would require the `cut` be Fair and Good (for which none exists):

```
diamonds %>%
  filter(cut == "Fair", cut == "Good", price <= 600)
```

Only display data from diamonds that do not have a `cut` value of Fair:

```
diamonds %>% filter(cut != "Fair")
```

6.5 select()

Function: Select only the columns (variables) that you want to see. Gets rid of all other columns. You can to refer to the columns by the column position (first column) or by name. The order in which you list the column names/positions is the order that the columns will be displayed.

In the `diamonds` dataset, only retain cut and color:

```
diamonds %>% select(cut, color)
```

Only retain the first five columns:

```
diamonds %>% select(1:5)

# or

diamonds %>% select(1,2,3,4,5)
```

Retain all of the columns except for cut:

```
diamonds %>% select(-cut)
```

Retain all of the columns except for cut and color:

```
diamonds %>% select(-cut, -color)
```

Retain all of the columns except for the first five columns:

```
diamonds %>% select (-1,-2,-3,-4,-5)
# or
diamonds %>% select(-(1:5))
```

You can also retain all of the columns, but rearrange some of the columns to appear at the beginning—this moves the x,y,z variables to the first 3 columns:

```
diamonds %>% select(x,y,z, everything())
```

6.6 arrange()

Function: Allows you arrange values within a variable in ascending or descending order (if that is applicable to your values). This can apply to both numerical and non-numerical values.

Arrange `cut` by alphabetical order (A to Z):

```
diamonds %>% arrange(cut)
```

Arrange `price` by numerical order (lowest to highest):

```
diamonds %>% arrange(price)
```

Arrange `cut` in descending alphabetical order:

```
diamonds %>% arrange(desc(cut))
```

Arrange `price` in descending numerical order:

```
diamonds %>% arrange(desc(price))
```

6.6.1 Exercises

In the following exercises, it is important to type out the code rather than to copy and paste it. This code should produce zero errors (unless otherwise specified); refer back to the troubleshooting section (3.6) if necessary.

1. Practice typing/executing these problems and explain what each line of code does. **Make sure tidyverse is loaded in your libraries**

This is an example for how each problem should be solved. The purpose for each line should be explained.

```
## Example illustrating how to answer these problems:

diamonds %>%                          # utilizes the diamonds dataset
  group_by(color, clarity) %>%        # groups data by color and clarity variables
  mutate(price200 = mean(price)) %>% # creates new variable (average price by groups)
  ungroup() %>%                       # data no longer grouped by color and clarity
  mutate(random10 = 10 + price) %>%  # new variable, original price + $10
  select(cut, color,                  # retain only these listed columns
         clarity, price,
         price200, random10) %>%
  arrange(color) %>%                  # visualize data ordered by color
  group_by(cut) %>%                   # group data by cut
  mutate(dis = n_distinct(price),     # counts the number of unique price values per cut
         rowID = row_number()) %>%    # numbers each row consecutively for each cut
  ungroup()                           # final ungrouping of data
```

It is an excellent exercise to execute one line at a time to visualize how each line of code changes the output. Refer back to 3.6.2 for more information on this technique!

```
library(tidyverse)

## Problem A
midwest %>%
  group_by(state) %>%
  summarize(poptotalmean = mean(poptotal),
            poptotalmed = median(poptotal),
            popmax = max(poptotal),
            popmin = min(poptotal),
            popdistinct = n_distinct(poptotal),
            popfirst = first(poptotal),
            popany = any(poptotal < 5000),
            popany2 = any(poptotal > 2000000)) %>%
  ungroup()
```

```
## # A tibble: 5 x 9
##   state poptotalmean poptotalmed popmax popmin popdistinct popfirst popany
##   <chr>        <dbl>       <dbl>  <int>  <int>       <int>    <int> <lgl>
## 1 IL         112065.      24486. 5.11e6   4373         101    66090 TRUE
## 2 IN          60263.      30362. 7.97e5   5315          92    31095 FALSE
## 3 MI         111992.      37308  2.11e6   1701          83    10145 TRUE
## 4 OH         123263.      54930. 1.41e6  11098          88    25371 FALSE
## 5 WI          67941.      33528  9.59e5   3890          72    15682 TRUE
## # ... with 1 more variable: popany2 <lgl>
```

```
## Problem B
midwest %>%
  group_by(state) %>%
  summarize(num5k = sum(poptotal < 5000),
            num2mil = sum(poptotal > 2000000),
            numrows = n()) %>%
  ungroup()
```

```
## # A tibble: 5 x 4
##   state num5k num2mil numrows
```

```
##   <chr> <int>   <int>   <int>
## 1 IL        1       1     102
## 2 IN        0       0      92
## 3 MI        1       1      83
## 4 OH        0       0      88
## 5 WI        2       0      72
```

```
## Problem C
# part I
midwest %>%
  group_by(county) %>%
  summarize(x = n_distinct(state)) %>%
  arrange(desc(x)) %>%
  ungroup()
```

```
## # A tibble: 320 x 2
##    county         x
##    <chr>      <int>
##  1 CRAWFORD       5
##  2 JACKSON        5
##  3 MONROE         5
##  4 ADAMS          4
##  5 BROWN          4
##  6 CLARK          4
##  7 CLINTON        4
##  8 JEFFERSON      4
##  9 LAKE           4
## 10 WASHINGTON     4
## # ... with 310 more rows
```

```
# part II
# How does n() differ from n_distinct()?
# When would they be the same? different?
midwest %>%
  group_by(county) %>%
  summarize(x = n()) %>%
  ungroup()
```

```
## # A tibble: 320 x 2
##    county        x
##    <chr>     <int>
##  1 ADAMS         4
##  2 ALCONA        1
##  3 ALEXANDER     1
##  4 ALGER         1
##  5 ALLEGAN       1
##  6 ALLEN         2
##  7 ALPENA        1
##  8 ANTRIM        1
##  9 ARENAC        1
## 10 ASHLAND       2
## # ... with 310 more rows
```

```
# part III
# hint:
# - How many distinctly different counties are there for each county?
```

```
# - Can there be more than 1 (county) county in each county?
# - What if we replace 'county' with 'state'?
midwest %>%
  group_by(county) %>%
  summarize(x = n_distinct(county)) %>%
  ungroup()
```

```
## # A tibble: 320 x 2
##    county        x
##    <chr>     <int>
##  1 ADAMS         1
##  2 ALCONA        1
##  3 ALEXANDER     1
##  4 ALGER         1
##  5 ALLEGAN       1
##  6 ALLEN         1
##  7 ALPENA        1
##  8 ANTRIM        1
##  9 ARENAC        1
## 10 ASHLAND       1
## # ... with 310 more rows
```

```
## Problem D
diamonds %>%
  group_by(clarity) %>%
  summarize(a = n_distinct(color),
            b = n_distinct(price),
            c = n()) %>%
  ungroup()
```

```
## # A tibble: 8 x 4
##   clarity     a     b     c
##   <ord>   <int> <int> <int>
## 1 I1          7   632   741
## 2 SI2         7  4904  9194
## 3 SI1         7  5380 13065
## 4 VS2         7  5051 12258
## 5 VS1         7  3926  8171
## 6 VVS2        7  2409  5066
## 7 VVS1        7  1623  3655
## 8 IF          7   902  1790
```

```
## Problem E
# part I
diamonds %>%
  group_by(color, cut) %>%
  summarize(m = mean(price),
            s = sd(price)) %>%
  ungroup()
```

```
## # A tibble: 35 x 4
##    color cut           m     s
##    <ord> <ord>     <dbl> <dbl>
##  1 D     Fair      4291. 3286.
##  2 D     Good      3405. 3175.
```

```
##  3 D     Very Good 3470. 3524.
##  4 D     Premium   3631. 3712.
##  5 D     Ideal     2629. 3001.
##  6 E     Fair      3682. 2977.
##  7 E     Good      3424. 3331.
##  8 E     Very Good 3215. 3408.
##  9 E     Premium   3539. 3795.
## 10 E     Ideal     2598. 2956.
## # ... with 25 more rows
```

```
# part II
diamonds %>%
  group_by(cut, color) %>%
  summarize(m = mean(price),
            s = sd(price)) %>%
  ungroup()
```

```
## # A tibble: 35 x 4
##    cut   color     m     s
##    <ord> <ord> <dbl> <dbl>
##  1 Fair  D     4291. 3286.
##  2 Fair  E     3682. 2977.
##  3 Fair  F     3827. 3223.
##  4 Fair  G     4239. 3610.
##  5 Fair  H     5136. 3886.
##  6 Fair  I     4685. 3730.
##  7 Fair  J     4976. 4050.
##  8 Good  D     3405. 3175.
##  9 Good  E     3424. 3331.
## 10 Good  F     3496. 3202.
## # ... with 25 more rows
```

```
# part III
# hint:
# - How good is the sale if the price of diamonds equaled msale?
# - e.x. The diamonds are x% off original price in msale.
diamonds %>%
  group_by(cut, color, clarity) %>%
  summarize(m = mean(price),
            s = sd(price),
            msale = m * 0.80) %>%
  ungroup()
```

```
## # A tibble: 276 x 6
##    cut   color clarity     m     s msale
##    <ord> <ord> <ord>   <dbl> <dbl> <dbl>
##  1 Fair  D     I1      7383  5899. 5906.
##  2 Fair  D     SI2     4355. 3260. 3484.
##  3 Fair  D     SI1     4273. 3019. 3419.
##  4 Fair  D     VS2     4513. 3383. 3610.
##  5 Fair  D     VS1     2921. 2550. 2337.
##  6 Fair  D     VVS2    3607  3629. 2886.
##  7 Fair  D     VVS1    4473  5457. 3578.
##  8 Fair  D     IF      1620.  525. 1296.
##  9 Fair  E     I1      2095.  824. 1676.
```

```
## 10 Fair  E     SI2     4172. 3055. 3338.
## # ... with 266 more rows
```

```
## Problem F
diamonds %>%
  group_by(cut) %>%
  summarize(potato = mean(depth),
            pizza = mean(price),
            popcorn = median(y),
            pineapple = potato - pizza,
            papaya = pineapple ^ 2,
            peach = n()) %>%
  ungroup()
```

```
## # A tibble: 5 x 7
##   cut       potato pizza popcorn pineapple    papaya peach
##   <ord>      <dbl> <dbl>   <dbl>     <dbl>     <dbl> <int>
## 1 Fair        64.0 4359.    6.1     -4295. 18444586.  1610
## 2 Good        62.4 3929.    5.99    -3866. 14949811.  4906
## 3 Very Good   61.8 3982.    5.77    -3920. 15365942. 12082
## 4 Premium     61.3 4584.    6.06    -4523. 20457466. 13791
## 5 Ideal       61.7 3458.    5.26    -3396. 11531679. 21551
```

```
## Problem G
# part I
diamonds %>%
  group_by(color) %>%
  summarize(m = mean(price)) %>%
  mutate(x1 = str_c("Diamond color ", color),
         x2 = 5) %>%
  ungroup()
```

```
## # A tibble: 7 x 4
##   color     m x1                x2
##   <ord> <dbl> <chr>          <dbl>
## 1 D     3170. Diamond color D     5
## 2 E     3077. Diamond color E     5
## 3 F     3725. Diamond color F     5
## 4 G     3999. Diamond color G     5
## 5 H     4487. Diamond color H     5
## 6 I     5092. Diamond color I     5
## 7 J     5324. Diamond color J     5
```

```
# part II
# What does the first ungroup() do? Is it useful here? Why/why not?
# Why isn't there a closing ungroup() after the mutate()?
diamonds %>%
  group_by(color) %>%
  summarize(m = mean(price)) %>%
  ungroup() %>%
  mutate(x1 = str_c("Diamond color ", color),
         x2 = 5)
```

```
## # A tibble: 7 x 4
##   color     m x1                x2
##   <ord> <dbl> <chr>          <dbl>
```

```
## 1 D     3170. Diamond color D     5
## 2 E     3077. Diamond color E     5
## 3 F     3725. Diamond color F     5
## 4 G     3999. Diamond color G     5
## 5 H     4487. Diamond color H     5
## 6 I     5092. Diamond color I     5
## 7 J     5324. Diamond color J     5
```

```
## Problem H
# part I
diamonds %>%
  group_by(color) %>%
  mutate(x1 = price * 0.5) %>%
  summarize(m = mean(x1)) %>%
  ungroup()
```

```
## # A tibble: 7 x 2
##   color     m
##   <ord> <dbl>
## 1 D     1585.
## 2 E     1538.
## 3 F     1862.
## 4 G     2000.
## 5 H     2243.
## 6 I     2546.
## 7 J     2662.
```

```
# part II
# What's the difference between part I and II?
diamonds %>%
  group_by(color) %>%
  mutate(x1 = price * 0.5) %>%
  ungroup() %>%
  summarize(m = mean(x1))
```

```
## # A tibble: 1 x 1
##       m
##   <dbl>
## 1 1966.
```

2. Why is grouping data necessary?
3. Why is ungrouping data necessary?
4. When should you ungroup data?
5. If the code does **not** contain `group_by()`, do you still need `ungroup()` at the end? For example, does `data() %>% mutate(newVar = 1 + 2)` require `ungroup()`?

6.7 Extra Practice

Keep practicing these functions! The more you practice, the better you'll be when it comes time to manage your own code!

Using the `diamonds` built-in dataset (requires **tidyverse**), perform the following tasks:

1. View all of the variable names in `diamonds` (hint: `View()`).

2. Arrange the diamonds by:

- Lowest to highest `price` (hint: `arrange()`)
- Highest to lowest `price` (hint: `arrange()`, `desc()`)
- Lowest `price` and `cut`
- highest `price` and `cut`

3. Arrange the diamonds by lowest to highest `price` and worst to best `clarity`.
4. Create a new variable named `salePrice` to reflect a discount of $250 off of the original cost of each diamond (hint: `mutate()`).
5. Remove the `x`, `y`, and `z` variables from the `diamonds` dataset (hint: `select()`).
6. Determine the number of diamonds there are for each `cut` value (hint: `group_by()`, `summarize()`).
7. Create a new column named `totalNum` that calculates the total number of diamonds.

Chapter 7

Advanced Data Management

So far, we have learned the basic functions that are used to manage our data. In this section, we will cover functions that aid with more complex problems. These functions will not be useful for everyone, but it is good to know that they exist.

7.1 count()

What it does: Collapses the rows and counts the number of observations per group of values.

Count the number of values for each `cut`:

```
diamonds %>% count(cut)

# is the same as

diamonds %>% group_by(cut) %>% count()

# is the same as

diamonds %>%
  group_by(cut) %>%
  summarize(n = n())
```

Count the number of values for each `cut` and `color`:

```
diamonds %>% count(cut, clarity)

# is the same as
diamonds %>% group_by(cut, clarity) %>% count()
diamonds %>% group_by(cut, clarity) %>% summarize(n = n())
```

7.2 rename()

What it does: Renames a column/variable.

Renames the price variable as `PRICE` (notice that the **new label goes first** in the argument). Again, it's important to remember that there are many different ways to perform the same task. The method each programmer chooses is all about personal preference (or current knowledge of available functions).

```
diamonds %>% rename(PRICE = price)

# is the same as

diamonds %>%
  mutate(PRICE = price) %>% # creates new variable based on old variable
  select(-price) # removes old variable from dataset
```

Renames the x, y, z variables to length, width, and depth (see `?diamonds`):

```
diamonds %>% rename(length = x, width = y, depth = z)

# is the same as

diamonds %>%
  mutate(length = x, width = y, depth = z) %>%
  select(-x, -y, -z)
```

7.3 row_number()

Using `row_number()` with `mutate()` will create a column of consecutive numbers. The `row_number()` function is useful for creating an identification number (an ID variable). It is also useful for labeling each observation by a grouping variable.

```
### Practice Dataset
practice <-
  tibble(Subject = rep(c(1,2,3),8),
         Date = c("2019-01-02", "2019-01-02", "2019-01-02",
                  "2019-01-03", "2019-01-03", "2019-01-03",
                  "2019-01-04", "2019-01-04", "2019-01-04",
                  "2019-01-05", "2019-01-05", "2019-01-05",
                  "2019-01-06", "2019-01-06", "2019-01-06",
                  "2019-01-07", "2019-01-07", "2019-01-07",
                  "2019-01-08", "2019-01-08", "2019-01-08",
                  "2019-01-01", "2019-01-01", "2019-01-01"),
         DV = c(sample(1:10, 24, replace = T)),
         Inject = rep(c("Pos", "Neg", "Neg", "Neg", "Pos", "Pos"), 4))
```

Using the practice dataset, let's add a variable called `Session`. Each session is comprised of 1 positive day and 1 negative day closest in date. For example, the first observation of Inject = pos and the first observation where Inject = neg will both have a `Session` value of `1`; the second observation of Inject = pos and the second observation of Inject = neg will be session 2). In the code below, you will see three methods for creating `Session`. Which method produces the result we need?

```
## Method1
practice %>%
  mutate(Session = row_number())
```

```
## # A tibble: 24 x 5
##    Subject Date          DV Inject Session
##      <dbl> <chr>      <int> <chr>    <int>
##  1       1 2019-01-02     3 Pos          1
##  2       2 2019-01-02     1 Neg          2
##  3       3 2019-01-02     4 Neg          3
```

```
##  4       1 2019-01-03     3 Neg          4
##  5       2 2019-01-03     7 Pos          5
##  6       3 2019-01-03     2 Pos          6
##  7       1 2019-01-04     4 Pos          7
##  8       2 2019-01-04    10 Neg          8
##  9       3 2019-01-04     5 Neg          9
## 10       1 2019-01-05     9 Neg         10
## # ... with 14 more rows
```

```
## Method2
practice %>%
  group_by(Subject, Inject) %>%
  mutate(Session = row_number())
```

```
## # A tibble: 24 x 5
## # Groups:   Subject, Inject [6]
##    Subject Date          DV Inject Session
##      <dbl> <chr>      <int> <chr>    <int>
##  1       1 2019-01-02     3 Pos          1
##  2       2 2019-01-02     1 Neg          1
##  3       3 2019-01-02     4 Neg          1
##  4       1 2019-01-03     3 Neg          1
##  5       2 2019-01-03     7 Pos          1
##  6       3 2019-01-03     2 Pos          1
##  7       1 2019-01-04     4 Pos          2
##  8       2 2019-01-04    10 Neg          2
##  9       3 2019-01-04     5 Neg          2
## 10       1 2019-01-05     9 Neg          2
## # ... with 14 more rows
```

```
## Method3
practice %>%
  group_by(Subject, Inject) %>%
  arrange(Date) %>%
  mutate(Session = row_number())
```

```
## # A tibble: 24 x 5
## # Groups:   Subject, Inject [6]
##    Subject Date          DV Inject Session
##      <dbl> <chr>      <int> <chr>    <int>
##  1       1 2019-01-01    10 Neg          1
##  2       2 2019-01-01     1 Pos          1
##  3       3 2019-01-01    10 Pos          1
##  4       1 2019-01-02     3 Pos          1
##  5       2 2019-01-02     1 Neg          1
##  6       3 2019-01-02     4 Neg          1
##  7       1 2019-01-03     3 Neg          2
##  8       2 2019-01-03     7 Pos          2
##  9       3 2019-01-03     2 Pos          2
## 10       1 2019-01-04     4 Pos          2
## # ... with 14 more rows
```

7.3.1 Exercises

1. Create a row ID for diamonds where each row is unique and order doesn't matter

2. Create an ID that relies on the clarity of diamonds where order doesn't matter
3. Create an ID that represents the price rank of the diamond.
    - Which diamond is #1 (highest priced diamond in dataset)?
    - Which diamond is ranked #2 in highest price?
4. Create an ID that represents price rank within each clarity category.
    - Of the diamonds with the clarity IF, what is the highest ranked/most expensive diamond?
    - Of the diamonds with the clarity SI2, what is the 2nd most expensive diamond (rank = 2)

7.4 ifelse()

Using the `practice` dataset from before, create a new variable called `Health` with values of `sick` or `healthy`:

- Subject 1 is `sick`
- Subject 2 is `healthy`
- Subject 3 is `healthy`

```
## Method A
practice %>%
  mutate(Health = ifelse(Subject == 1,
                         "sick",
                         "healthy"))
```

```
## # A tibble: 24 x 5
##    Subject Date          DV Inject Health
##      <dbl> <chr>      <int> <chr>  <chr>
##  1       1 2019-01-02     3 Pos    sick
##  2       2 2019-01-02     1 Neg    healthy
##  3       3 2019-01-02     4 Neg    healthy
##  4       1 2019-01-03     3 Neg    sick
##  5       2 2019-01-03     7 Pos    healthy
##  6       3 2019-01-03     2 Pos    healthy
##  7       1 2019-01-04     4 Pos    sick
##  8       2 2019-01-04    10 Neg    healthy
##  9       3 2019-01-04     5 Neg    healthy
## 10       1 2019-01-05     9 Neg    sick
## # ... with 14 more rows
```

```
## Method B
practice %>%
  mutate(Health = ifelse(Subject %in% c(2,3),
                         "healthy",
                         "sick"))
```

```
## # A tibble: 24 x 5
##    Subject Date          DV Inject Health
##      <dbl> <chr>      <int> <chr>  <chr>
##  1       1 2019-01-02     3 Pos    sick
##  2       2 2019-01-02     1 Neg    healthy
##  3       3 2019-01-02     4 Neg    healthy
##  4       1 2019-01-03     3 Neg    sick
##  5       2 2019-01-03     7 Pos    healthy
```

```
##  6       3 2019-01-03     2 Pos    healthy
##  7       1 2019-01-04     4 Pos    sick
##  8       2 2019-01-04    10 Neg    healthy
##  9       3 2019-01-04     5 Neg    healthy
## 10       1 2019-01-05     9 Neg    sick
## # ... with 14 more rows
```

```
## Method C
practice %>%
  mutate(Health = ifelse(Subject == 1,
                         "sick",
                         ifelse(Subject %in% c(2,3),
                                "healthy",
                                "wtf")))
```

```
## # A tibble: 24 x 5
##    Subject Date          DV Inject Health
##      <dbl> <chr>      <int> <chr>  <chr>
##  1       1 2019-01-02     3 Pos    sick
##  2       2 2019-01-02     1 Neg    healthy
##  3       3 2019-01-02     4 Neg    healthy
##  4       1 2019-01-03     3 Neg    sick
##  5       2 2019-01-03     7 Pos    healthy
##  6       3 2019-01-03     2 Pos    healthy
##  7       1 2019-01-04     4 Pos    sick
##  8       2 2019-01-04    10 Neg    healthy
##  9       3 2019-01-04     5 Neg    healthy
## 10       1 2019-01-05     9 Neg    sick
## # ... with 14 more rows
```

```
## Method D
practice %>%
  mutate(Health = ifelse(Subject == 1,
                         "sick",
                         ifelse(Subject == 2,
                                "healthy",
                                "wtf")))
```

```
## # A tibble: 24 x 5
##    Subject Date          DV Inject Health
##      <dbl> <chr>      <int> <chr>  <chr>
##  1       1 2019-01-02     3 Pos    sick
##  2       2 2019-01-02     1 Neg    healthy
##  3       3 2019-01-02     4 Neg    wtf
##  4       1 2019-01-03     3 Neg    sick
##  5       2 2019-01-03     7 Pos    healthy
##  6       3 2019-01-03     2 Pos    wtf
##  7       1 2019-01-04     4 Pos    sick
##  8       2 2019-01-04    10 Neg    healthy
##  9       3 2019-01-04     5 Neg    wtf
## 10       1 2019-01-05     9 Neg    sick
## # ... with 14 more rows
```

7.4.1 Exercises (use practice dataset):

1. Create a new variable called `Group` where Subject 1 is in `GroupA`, Subject 2 is in `GroupB`, and Subject 3 is in `GroupC` using:
   - **Two** `ifelse()` statements
   - **Three** `ifelse()` statements
   - Why can't we just use **one** `ifelse()` statement here?
2. Create a new variable called `Acceptable` with `yes` or `no` values using `ifelse()`. For a `yes` value, the following criteria must be met:
   - When `Inject` = `pos`, DV >=
   - When `Inject` = `neg`, DV < 5

```
practice %>%
  mutate(Acceptable = ifelse(Inject == "Pos",
                             ifelse(DV >= 5,
                                    "yes",
                                    "no"),
                             ifelse(DV < 5,
                                    "yes",
                                    "no")))
```

```
## # A tibble: 24 x 5
##    Subject Date          DV Inject Acceptable
##      <dbl> <chr>      <int> <chr>  <chr>
##  1       1 2019-01-02     3 Pos    no
##  2       2 2019-01-02     1 Neg    yes
##  3       3 2019-01-02     4 Neg    yes
##  4       1 2019-01-03     3 Neg    yes
##  5       2 2019-01-03     7 Pos    yes
##  6       3 2019-01-03     2 Pos    no
##  7       1 2019-01-04     4 Pos    no
##  8       2 2019-01-04    10 Neg    no
##  9       3 2019-01-04     5 Neg    no
## 10       1 2019-01-05     9 Neg    no
## # ... with 14 more rows
```

Part III

Graphing

Chapter 8

Introduction to Graphing

The first skill most students want to master when learning R is how to graph their data. While graphing is useful, it is of my opinion that data management is far more exciting.

Graphing data **assumes** that your dataset is clean and is ready to be handled as is. If you need to add new variables or remove missing values, you must first learn to organize your data (or you can manually edit it in Excel). In the Tidyverse chapter (see Chapter 4), we've learned the basics for managing our data in R using the `tidyverse` package. Later, we will go over advanced techniques in data wrangling.

8.1 How Graphing Works

R has a base graphing package, `graphics`, which utilizes the `plot()` function. However, most R users use the `ggplot2` package (preloaded with the tidyverse package). This is because ggplot2 is more powerful, allows for greater customization, and is fairly user friendly. I highly recommend Hadley Wickham's book, ggplot2 – Elegant Graphics for Data analysis, for additional guidance on more specific graphing tools and techniques. I will go over a few graphing techniques that cannot encompass the needs of everyone reading this book. The primary focus of this section will be line graphs, but we will briefly touch on bar graphs, histograms, and scatterplots.

As always, **refer back to the troubleshooting section** (Section 3.6) when you come across errors in your code. When searching the internet for help, I recommend adding "ggplot2" to the end of your query to ensure that it is focused on the type of graphing utilized in this section.

8.2 Working with Sample Data

Before we attempt to handle your unique dataset, let's first get some graphing practice with sample data. In these examples, I will be using the built-in datasets that R **should** have available to you. As a refresher, a built-in dataset is a free, accessible dataset that R provides to all users. Sometimes, these built-in datasets are built into base R (i.e., no libraries need to be loaded for them to work). Other times, these built-in datasets are built into specific packages (i.e., requires specific libraries). Many of the built-in datasets I use originate from base R and packages from tidyverse. As long as you make sure that you load tidyverse at the beginning of each session, you should have access to the built-in datasets that are used in the examples.

8.3 Load the Packages

If you don't already have the `tidyverse`, `readxl`, and `writexl` packages loaded, please do so now. If you do not have these packages **installed**, please make sure to remedy that. If you are unsure how to install these packages, refer back to Section 3.2.3.

```
library(tidyverse)
library(writexl)
library(readxl)
```

Because the diamonds dataset is already structured properly (factor, numeric, integer), we can jump right into graphing. In contrast, you will need to restructure your personal data when we work on it later.

Before you begin coding, it's important to know what you want to graph and how you want your graph to look. What data is represented on the graph (i.e., mean, frequency, median)? Are there error bars? Is it a line graph, histogram, bar graph, violin/box plot? I always recommend taking a minute to draw out what your graph should look like on a separate sheet of paper.

8.4 Important Tidyverse Functions

Nearly all of the relevant examples of graphing in this guide requires a bit of **tidyverse** knowledge. Let's start by creating a sample dataset:

```
## Creating an object named Subject
Subject <- c("Wendy", "Wendy", "Wendy", # you can press ENTER to auto-indent
             "John", "John", "John",    # the code. This produces better
             "Helen", "Helen", "Helen") # formatting for the user.

## Creating an object named Date
Date <- c("2019-08-08", "2019-09-05", "2019-12-07", "2019-08-08",
          "2019-09-05", "2019-12-07", "2019-08-08", "2019-09-05", "2019-12-07")

## Creating an object named Score
Score <- c(2, 15, 34,
           5, 10, 27,
           16, 8, 40)

## Creating an object
mydata <- tibble(Subject, Date, Score)
```

Created the mydata dataset

Subject

Date

Score

Wendy

2019-08-08

2

Wendy

2019-09-05

15

Wendy

2019-12-07

34

John

2019-08-08

5

John

2019-09-05

10

John

2019-12-07

27

Helen

2019-08-08

16

Helen

2019-09-05

8

Helen

2019-12-07

40

Let's pick two variables (`Subject` and `Score`) to graph the `mydata` dataset.

```
ggplot(mydata, aes(x = Subject, y = Score)) +
  geom_point()
```

While it might be interesting to graph all of Helen, John, and Wendy's scores, another useful graph might be to see the **average scores for each person**. There are at least **two** ways of setting up your graph:

1. Using the `stat_summary` method
2. Using the `tidyverse` method

8.4.1 Using the `stat_summary` Method

One of the classic methods to graph is by using the `stat_summary()` function. We begin by using the `ggplot()` function, which requires the name of the dataset, we'll use `mydata` from our previous example, followed by the `aes()` function that encompasses the `x` and `y` variable specifications. Next, we add on the `stat_summary()` function. For this function, we specify that we want to calculate the `mean` of the `y` axis in the first argument (`fun.y` asks what function to use for the y variable). Then, we specify what graphing/geom element to plot. Here, we specified that we want points (other options could be bar, line, etc.).

```
ggplot(mydata, aes(x = Subject, y = Score)) +
  stat_summary(fun.y = "mean", geom = "point")
```

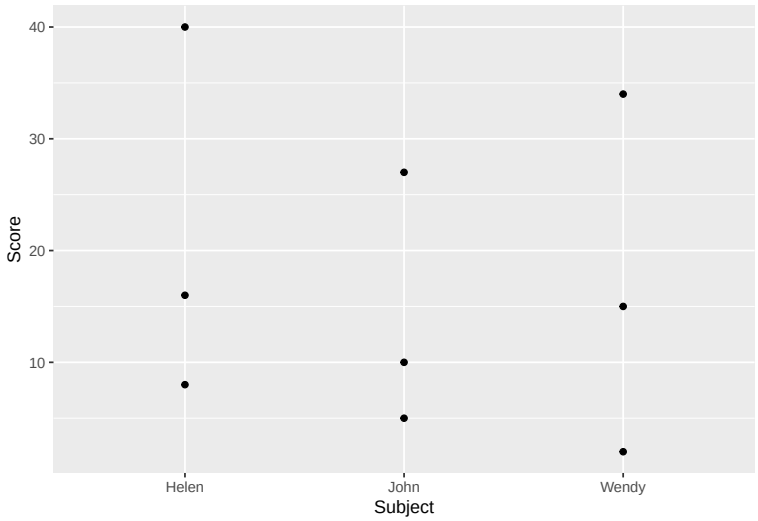
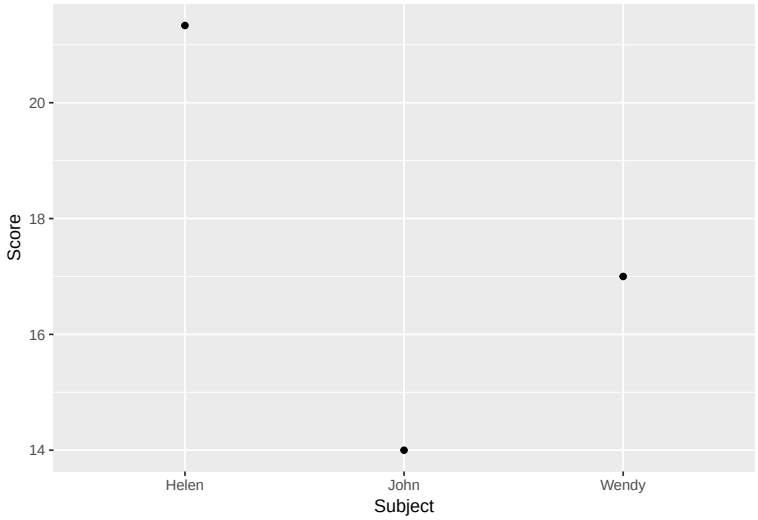


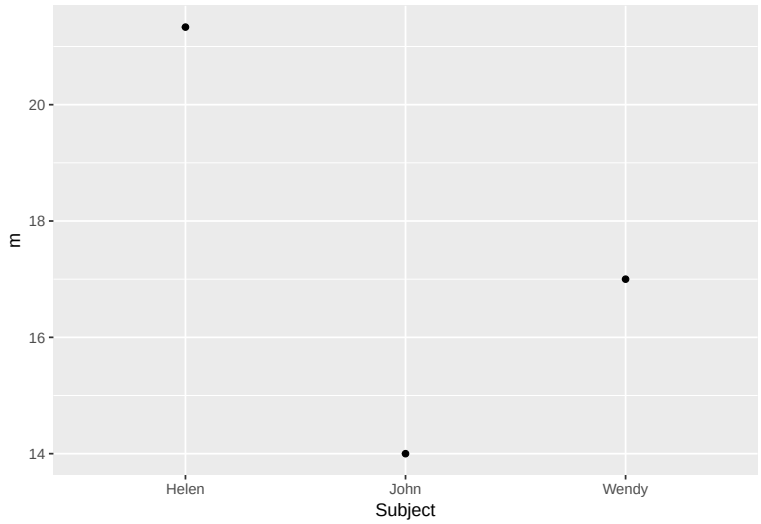
Figure 8.1: Graphing long data.



8.4.2 Using the `tidyverse` Method

The `tidyverse` method requires a bit more planning and preparation than the `stat_summary` method, but the end result is the same. However, the tidyverse method is my preferred method of coding for all other tasks (not just for graphing) because it is more user friendly (in my humble opinion). Let's look at how the `tidyverse` code is set up.

```
mydata %>%                          # name of the dataset
  group_by(Subject) %>%             # grouping the data
  summarize(m = mean(Score)) %>%    # calculating the mean
  ungroup() %>%                     # ungroup the data
  ggplot(aes(x = Subject, y = m)) + # set up the graph
  geom_point()                      # add data points on graph
```



For this example, the mean of `Score` was renamed to `m`. This was done in order to differentiate this set of values from the original values from `Scores`. That is, `m` represents **averaged `Score` values** whereas `Score` simply represents **raw values**. Additionally, we can always change the axes labels later to whatever we want using the `labs()` function; `labs()` will be introduced later in the chapter.

Let's break down each line of the `tidyverse` code:

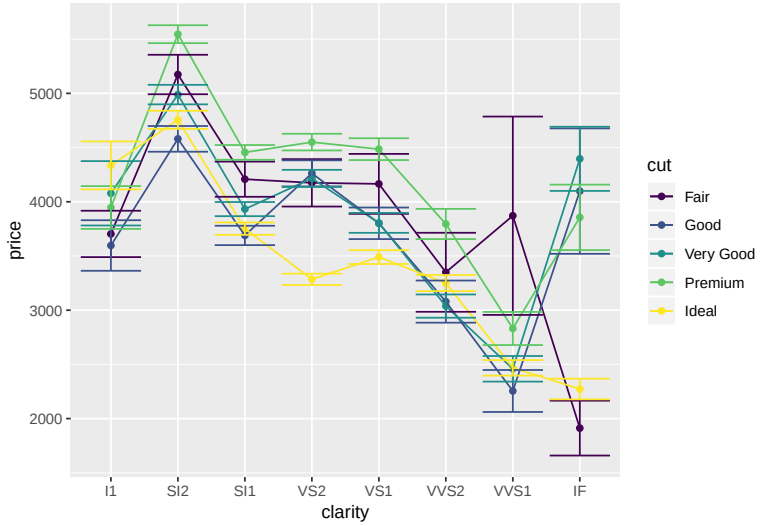
- `mydata %>%` represents the name of the dataset. The pipe (%>%) represents the phrase "and then", indicating that we want R to do more with the dataset.
- `group_by(Subject) %>%` indicates that we want to group the data by the `Subject` variable. We will learn more about the `group_by()` function in later chapters, but for the purposes of graphing, all we need to know is that **we must group all variables that are considered our independent variables**.
  - Note that all of these **grouping variables must be characters or factors**. They should not be numeric or integers!
  - We can determine what variables need to be included in `group_by()` for our graph by asking ourselves one question: Do I care about this variable's values? If we `group_by(Sex)`, as in biological sex, we are saying that we do want to see male and female data separately. Otherwise, R will graph the "average" person/organism, not accounting for males separately from females. If we `group_by(Sex, Age)`, that means we care about females of each `Age` value separately from males of each `Age` value.
  - In a typical graph, we may group up to 3 variables within the dataset. We will see examples of variable grouping in the upcoming sections.

8.4.3 Additional Practice

Recall that the `diamonds` dataset (explained in 5) has variables that measure the diamond's clarity and the diamond's cut. These variables both contain categorical values. That is, `clarity` values can only be: I1, SI2, SI1, VS2, VS1, VVS2, VVS1, and IF. Diamonds in the clarity category called "IF" are deemed as having flawless clarity. Similarly, `cut` values can only be in one of 5 categories (Fair, Good, Very Good, Premium, or Ideal). Notice that no single diamond in this dataset can be in two `cut` categories or two `clarity` categories at the same time!

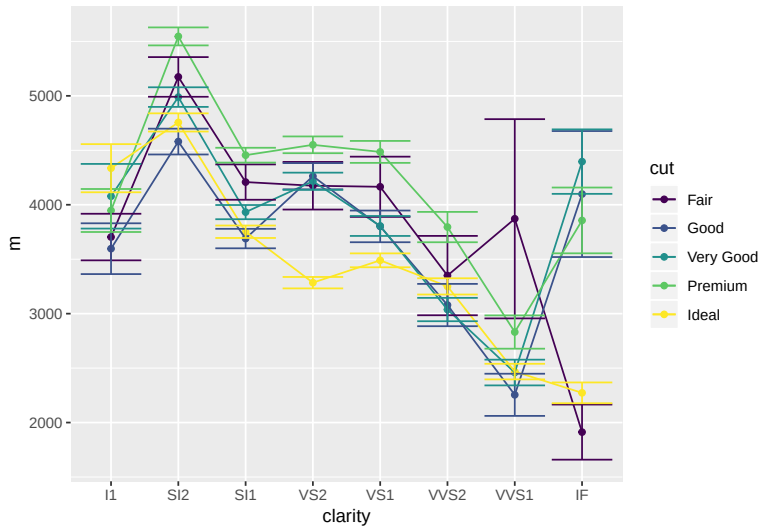
The code below shows how to graph the `clarity`, `cut`, and `price` of every diamond using both graphing methods:

```
## stats_summary method
ggplot(diamonds,                                                    # name of dataset
       aes(x = clarity,                                             # x-axis variable
           y = price,                                               # y-axis variable
           group = cut,                                             # grouping variable (legend)
           color = cut)) +                                          # coloring the grouping variable
  stat_summary(fun.y = "mean", geom = "point") +                    # adding data points
  stat_summary(fun.y = "mean", geom = "line") +                     # adding connecting lines
  stat_summary(fun.data = "mean_se", geom = "errorbar")             # adding error bars (standard error)
```




```
## tidyverse method
### first requires the 'sem' function to be loaded:
sem <- function(x, na.rm = FALSE) {
  out <-sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)
}

## graphing code
diamonds %>%                            # name of dataset
  group_by(clarity, cut) %>%            # grouping variables
  summarize(m = mean(price),            # calculating mean price
            s = sem(price)) %>%         # calculating standard error
  ggplot(aes(x = clarity,               # x-axis variable
             y = m,                     # y-axis variable
             group = cut,               # grouping variable
             color = cut)) +            # color the grouping variable
  geom_point() +                        # adding data points
  geom_line() +                         # adding connecting lines
  geom_errorbar(aes(ymin = m - s,       # adding lower error bars
                    ymax = m + s))      # adding upper error bars
```



I want to emphasize a few things:

1. Notice that the tidyverse method uses a mix of both pipes (`%>%`) **and** pluses (`+`) to graph. Be sure to use these punctuation marks at the appropriate time!

2. In order to calculate the standard error of the mean using the tidyverse method, you must create the `sem` function and execute the `sem` code **before** creating the graph. This is because the `sem()` function is not loaded in base R by default. However, the `stat_summary()` function **does** have a built-in standard error function ready for use!

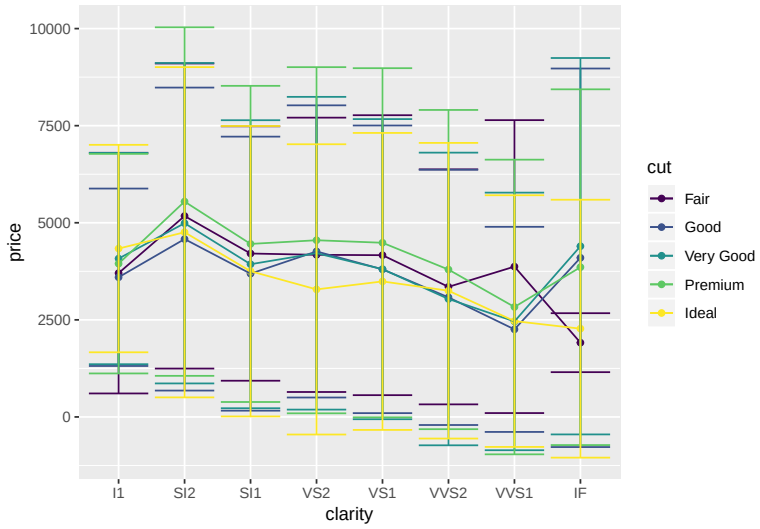
3. In this case, the tidyverse method is slightly more work to type out compared to the stat_summary method. However, you'll find that the tidy method can be easily read from top to bottom in a user-friendly way. Remember that the pipe can be read aloud as "and then".

4. Notice that the above code is styled in a particular manner. My preference is to add spaces between certain characters (using the shortcut hotkey for the `%>%`, **CTRL/CMD + SHIFT + M**, will automatically add spaces before and after each pipe). I also prefer to use the auto-indentation styling so that it is easier to read my code. If you wanted to, you could technically write all of your code in one line, but this wouldn't be very user-friendly code. See the example below to see what zero spacing and auto-indentation looks like):

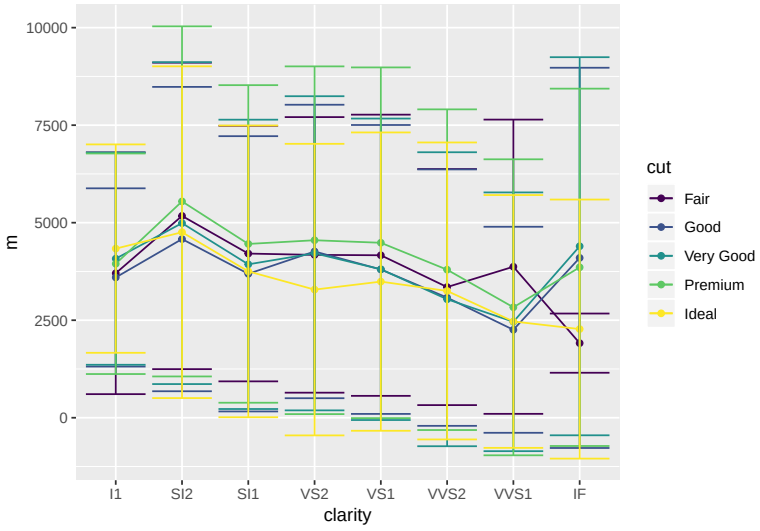
```
# using the tidy method (same code as above)
diamonds %>% group_by(clarity,cut) %>% summarize(m=mean(price),s=sem(price)) %>% ggplot(aes(x=clarity,y=
```

Here is an example for when the tidyverse method is slightly superior or even: calculating standard deviation (sd). In this case, calculating standard deviation with the stat_summary method requires **more** typing than with the tidy method. The `sd()` function can be used in the tidy method since it **is** a built-in function.

```
## stat_summary method
ggplot(diamonds,
       aes(x = clarity,
           y = price,
           group = cut,
           color = cut)) +
  stat_summary(fun.y = "mean", geom = "point") +
  stat_summary(fun.y = "mean", geom = "line") +
  stat_summary(fun.y = "mean",
               fun.ymax = function(x) mean(x) + sd(x), # calculating sd
               fun.ymin = function(x) mean(x) - sd(x), # calculating sd
               geom = "errorbar")
```




```
## tidyverse method
diamonds %>%
  group_by(cut, clarity) %>%
  summarize(m = mean(price),
            s = sd(price)) %>% # calculating sd
  ggplot(aes(x = clarity,
             y = m,
             group = cut,
             color = cut)) +
  geom_line() +
  geom_point() +
  geom_errorbar(aes(ymin = m - s, ymax = m + s))
```

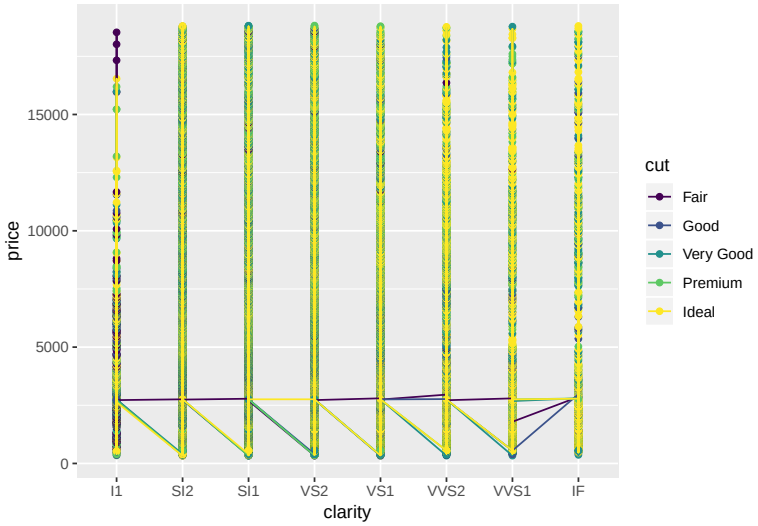


By now, you've noticed that I lean toward using the `tidyverse` method. So why bother introducing the `stat_summary` method at all? The answer is really simple: you should be aware of the available options out there. Tidyverse is still relatively new-ish at the time of writing this guide. Many experienced R users are still using "base-R" and will only know how to explain things using base R. Plus it's never a bad idea to know more than one way of completing a task. My philosphy is that you can't ask questions if you don't know what to even ask!

8.4.4 The Third Method

In a related caveat, I do want to talk about a "third" method. This method can only be used if the data is prepared exactly as you want it. That is, your dataset should only contain the summary statistics or raw data you want to plot. In the tidyverse method, we first had to calculate the summary statistics (mean, standard error) **before** beginning to plot with `ggplot()`. What if we didn't want to plot the **mean** price for `diamonds`, accounting for `cut` and `clarity`?

```
## no tidyverse-style code
ggplot(diamonds,
       aes(x = clarity,
           y = price,       # raw price value
           group = cut,
           color = cut)) +
  geom_point() +
  geom_line()
```



As you can see, R has plotted every single diamond in the dataset (53,940 diamonds!) according to their clarity and cut. This is because the `diamonds` dataset by default contains information about each individual diamond (for each row).

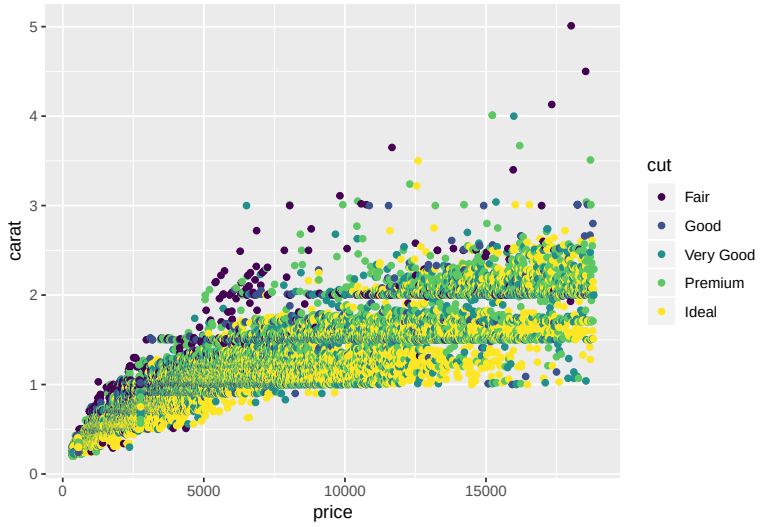
```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 0.24  Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 0.26  Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9 0.22  Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10 0.23  Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

If we wanted to use the "Third Method" to plot the mean `price` by `clarity` and `cut`, the dataset must look like this (note that there are only 40 combinations of cut and clarity - 5 cut options multiplied by 8 clarity options):

```
## # A tibble: 40 x 4
## # Groups:   clarity [8]
##    clarity cut        mean standard.err
##    <ord>   <ord>     <dbl>        <dbl>
##  1 I1      Fair      3704.        214.
##  2 I1      Good      3597.        233.
##  3 I1      Very Good 4078.        297.
##  4 I1      Premium   3947.        197.
##  5 I1      Ideal     4336.        221.
##  6 SI2     Fair      5174.        182.
##  7 SI2     Good      4580.        119.
##  8 SI2     Very Good 4989.         90.0
##  9 SI2     Premium   5546.         82.6
## 10 SI2     Ideal     4756.         83.4
## # ... with 30 more rows
```

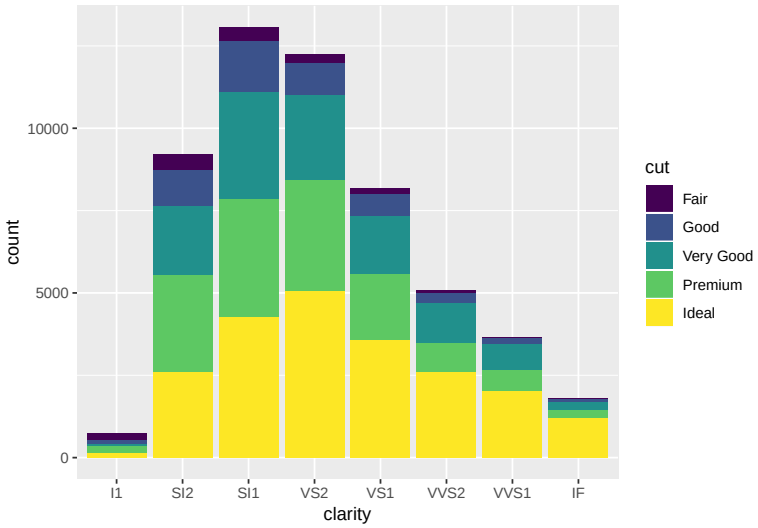
This method can be useful if we wanted to graph "non-summary statistic data". Translated: if you want to plot each row of data as-is (no calculations performed), this method can be used! This will essentially produce a scatter plot if you have many rows of data. If you have few rows of data, it can be plotted as a regular line graph.

```
ggplot(diamonds, aes(x = price,
                     y = carat,
                     group = cut,
                     color = cut)) +
  geom_point()
```



The "Third Method" is especially useful (and faster than the tidy method) when counting frequencies using a bar graph. The graph below depicts the number of diamonds that fit each `clarity` and `cut`. For example, there are about 5,000 diamonds with a VVS2 clarity and a little over half have a `cut` categorized as ideal (yellow bar). Notice that this code does not require a y-value; frequency count is the default dependent measure and is therefore assumed when a y-value is not specified. We'll talk more about bar graphs later.

```
ggplot(diamonds, aes(x = clarity, group = cut, fill = cut)) +
         geom_bar()
```



In summary, the "Third Method" can only be used when the data is exactly how you want it (i.e., when you're fine with having all of the rows in the dataset represented). Otherwise, the `tidyverse` method is the way to go.

Chapter 9

Scatter Plots

Scatter plots are among the easiest graphs to create, because each point represents a single observation. Let's build a scatterplot with the diamond's depth on the x-axis and the price on the y-axis.

```
diamonds %>%
  ggplot(aes(x = depth, y = price)) +
  geom_point()
```

Using the `?ggplot` help page, we see that the first argument for `ggplot()` is the **dataset**. Here, we are using the built-in diamonds dataset. Next, we must add aesthetics using the `aes()` function. There are a lot of aesthetic options, but for now, we will focus on the basics. By default, the `aes()` function requires at minimum an `x` and a `y` argument. Again, you may choose to omit the argument names (`x` and `y`), but then R will then assume that the first listed variable in `aes()` is the x-axis variable and the second variable is the y-axis variable.

9.1 Exercises

(Tip: recall that `%>%` is a pipe symbol defined in the Vocabulary Terms section)

1. Execute `diamonds %>% ggplot(aes(depth, price)) + geom_point()`
2. Execute `diamonds %>% ggplot(aes(price, depth)) + geom_point()`
3. Execute `diamonds %>% ggplot(aes(y = price, x = depth)) + geom_point()`
4. Execute `diamonds %>% ggplot(aes(x = x, y = z)) + geom_point()`

It's important to remember that graph building in R requires layering of code. That is, each piece of the graph must be explicitly stated. The `ggplot()` function starts building the skeleton of an R graph.

If you were to run the above code **without `+ geom_point()`**, the plot is blank:

```
diamonds %>% ggplot(aes(x = depth, y = price))
```

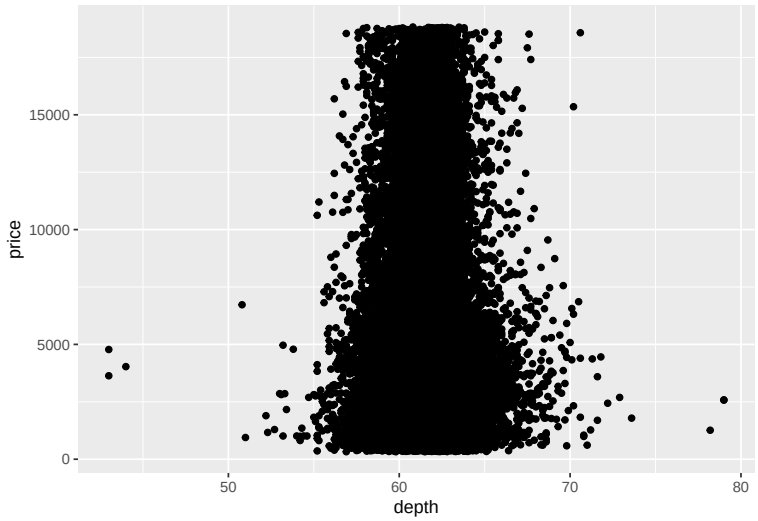
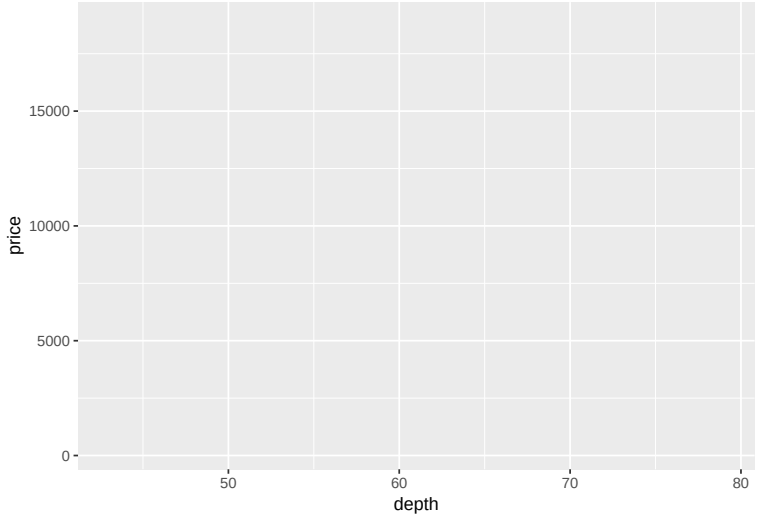


Figure 9.1: Basic scatterplot measuring price and depth variables in the ‘diamonds‘ dataset



In order for there to be data points, you must specify them with the `geom_point()` function.

What if we wanted to use a categorical variable like diamond `clarity` for the x-axis?

```
diamonds %>%
  ggplot(aes(x = clarity, y = price, group = cut, color = cut)) +
  geom_point()
```

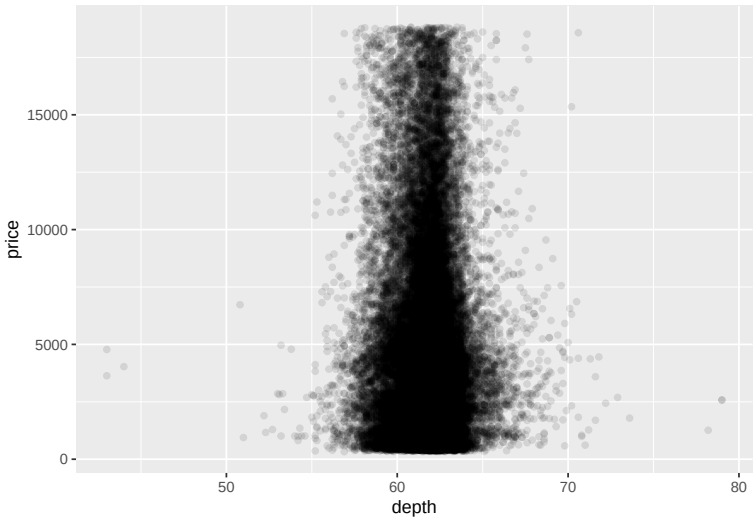
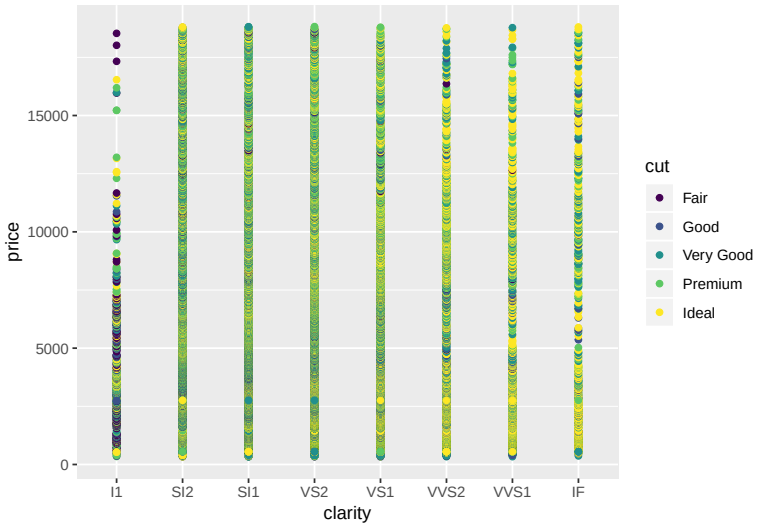


Figure 9.2: Basic scatterplot with transparency settings



I think most of us would agree that this isn't the best looking graph. Why?

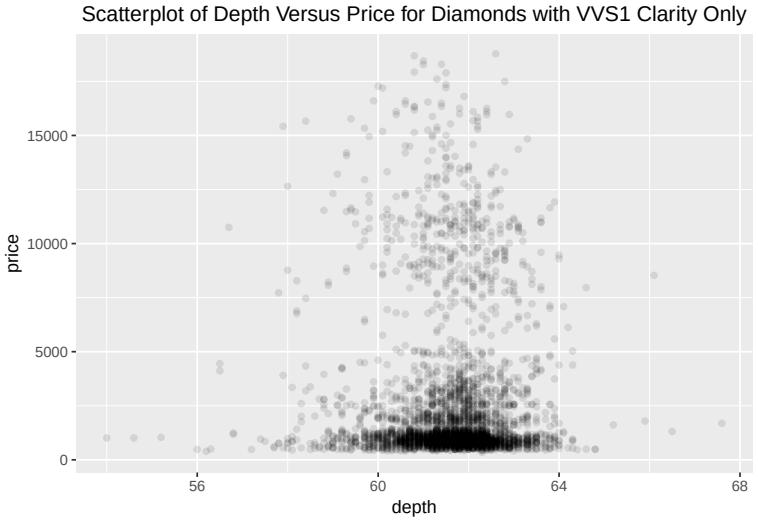
- Scatter plots are generally used to graph two continuous variables, not categorical variables.
- There is overplotting in both of the scatter plots seen in this chapter (i.e., there are too many data points on the graph, creating visual noise).

In **Figure 9.1**, we could improve the data representation by altering the transparency of the data points. What this does is show us where the majority of the data points sit. The more data points that sit in a particular location (the darker the area), the more observations there are.

```
# to alter the transparency of the data points, we must utilize
# the alpha argument within the geom_point() function
diamonds %>%
  ggplot(aes(x = depth, y = price)) +
  geom_point(alpha = 0.1) # lower alpha values increase transparency of points
```

Although increasing the transparency of the data points (via `alpha`) marginally helps with the data visualization, it's probably not what we want to present to the public given that it still looks noisy. To reduce

the number of data points, we may want to subset the data into categories. For example, we may want to subset the data by including diamonds from the dataset that have a clarity of `VVS1`.



This particular skill will be learned in the **How to Use Tidyverse** section. Again, this graphing section assumes that you already have your data completely ready and organized for graphing. However, for these example using built-in datasets, we will do a tiny bit of wrangling. We will learn how data management/organization/wrangling works in the later sections.

Chapter 10

Line Graphs

If your x-axis (independent) variable has some sort of order built into its values, it may benefit you to use a line graph. In the `diamonds` dataset, color is a ranked order variable with D being the best (absolutely colorless) and J being the worst (slight yellow).

10.1 Plotting Structural Elements

10.1.1 Mean Values

Data can be complex (to say the least). In the behavioral neurosciences, we often care about how the **average** subject performed. We may want to know: How did the treatment group perform in comparison to the control group? We may want to look into the individual differences eventually (especially if individual data varies a lot), but we typically want to know if the treatment group did better/worse/the same as the control group **on average**. In this kind of example, we need our data to tell us:

- the average (mean) performance for subjects in the control group
- the average (mean) performance for subjects in the treatment group

Of course, we need to know more information about the dataset before we can draw conclusions. How is performance measured? Are higher scores better or worse (scoring high on a depression scale may be a bad thing, but scoring high on a test is usually a good thing)? Here, we're not going to make any assumptions.

```
## # A tibble: 2 x 2
##   Group           `Average Performance`
##   <chr>                           <dbl>
## 1 Treatment Group                     8
## 2 Control Group                      10
```

Sometimes we may even want to know the standard deviation or the standard error. In that case, our dataset at minimum must look like this:

```
## # A tibble: 2 x 4
##   Group           `Average Performance` Std.Dev Std.Err
##   <chr>                           <dbl>   <dbl>   <dbl>
## 1 Treatment Group                     8     1.5     0.2
## 2 Control Group                      10     7       4
```

Recall that the `diamonds` data doesn't show us the means or standard deviations/errors. It shows us the raw, unaltered individual data points. Therefore, we must manually calculate the means before we can graph it.

First, we need to arrange the data so that it shows us the average price for each clarity using some tidyverse techniques. We will use the `group_by()` and `summarize()` functions in addition to pipes (`%>%`; **Ctrl + Shift + M**) to accomplish this. We will delve more into data management and how these functions work in later sections. For now, we will primarily focus on how these tools help us graph.

To properly organize our data so that a mean price is calculated for each diamond `clarity` category, we first need to group the data by our clarity variable in the `diamonds` dataset. The `tidyverse` style of coding organizes the code in a (generally) user-friendly format. That is, it is written and read from left to right and top to bottom (like that of Western cultures). Executing the following code will calculate the means for each clarity category of diamonds:

```
diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price)) %>% # m is defined as the mean price of diamonds grouped by clarity catego
  ungroup()
```

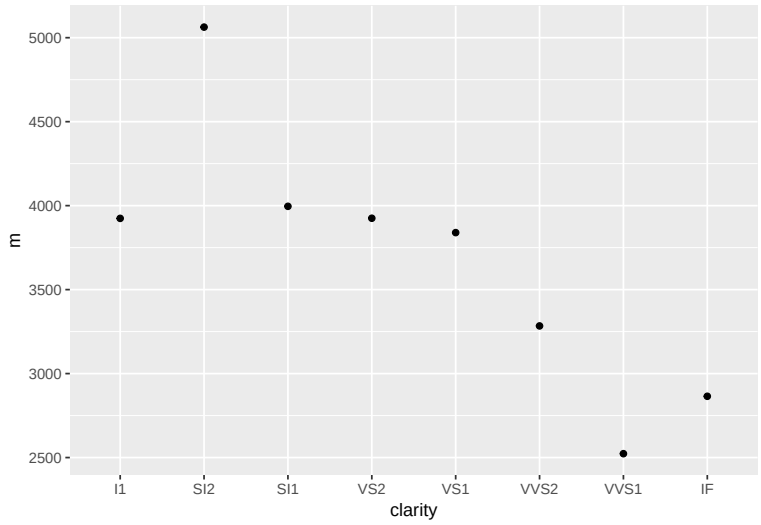
```
## # A tibble: 8 x 2
##   clarity     m
##   <ord>   <dbl>
## 1 I1      3924.
## 2 SI2     5063.
## 3 SI1     3996.
## 4 VS2     3925.
## 5 VS1     3839.
## 6 VVS2    3284.
## 7 VVS1    2523.
## 8 IF      2865.
```

Remember that it can be useful to conceptualize the pipes (`%>%`) as the phrase "and then". The above code can be read as:

> "I want to use the diamonds dataset and then group the data by clarity and then summarize the data by calculating the average price (for each clarity category) and then ungroup the data."

Now that we have our values, we can begin to graph. The next step is to add our graphing functions:

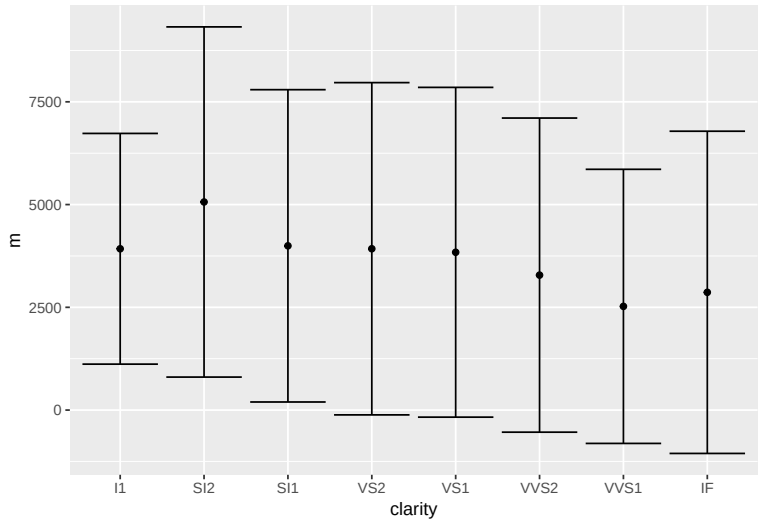
```
diamonds %>%
  group_by(clarity) %>% # sets up the grouping variable
  summarize(m = mean(price)) %>% # calculates the mean price for each diamond clarity
  ungroup() %>% # make sure to ungroup the data after summarizing
  ggplot(aes(x = clarity, y = m)) + # setting up x and y values for graphing
  geom_point() # plotting data points on the graph
```



10.1.2 Error Bars

Let's add onto the code we've already built so far. The following code adds error bars using standard deviation:

```
diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price),
            sd = sd(price)) %>%
  ungroup() %>%
  ggplot(aes(x = clarity, y = m)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - sd, ymax = m + sd))
```

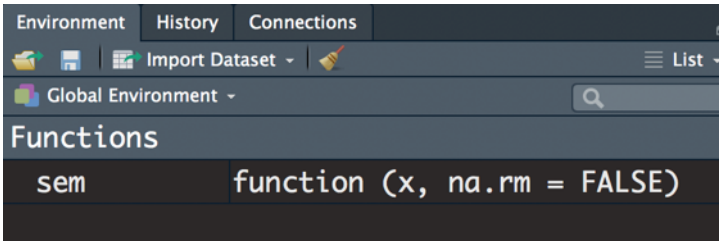


Having trouble running the code? Refer back to the troubleshooting section (3.6)!

In order to add error bars that represent the **standard error of the mean (SEM)**, we must first use a self-defined function. R does not currently have an accessible, built-in function at the time of writing this guide, so in order to use calculate the SEM, you must execute the following code (type this out in your script):

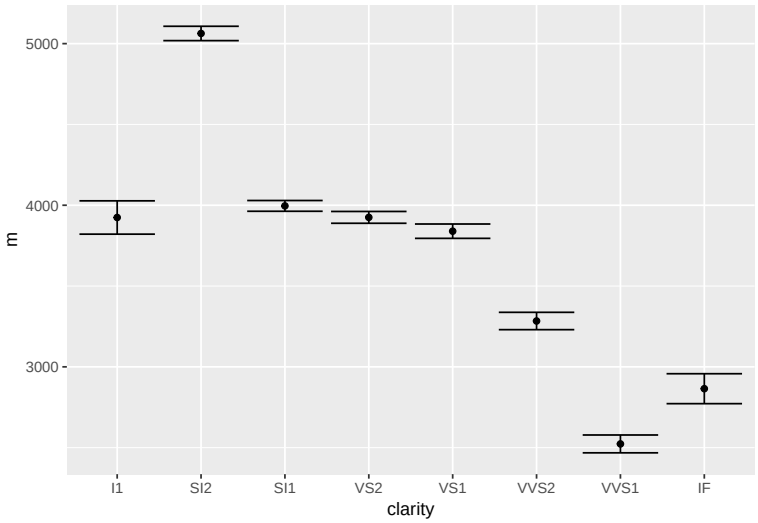
```
sem <- function(x, na.rm = FALSE) {
  out <- sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)}
```

I highly recommend keeping this code pasted somewhere (e.g., in your notes) if you frequently use SEM. After executing the code, `sem` will now be a usable function so long as it is available globally. This means that each time you begin an R session, this `sem` function must be redefined and loaded to the global environment, but once it is located in the environment, you can repeatedly use it for the session duration (similar to loading libraries):



After executing the `sem` function, run the following edited code to obtain a plot with standard errors represented:

```
diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ungroup() %>%
  ggplot(aes(x = clarity, y = m)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se))
```

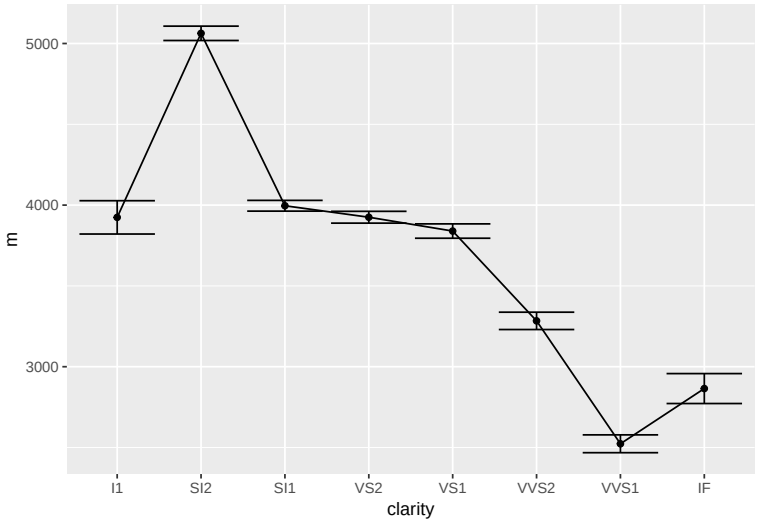


10.1.3 Connecting Lines

Just like all of the other geom elements, in order to display a connecting line on our graph, we must specify that there should be a connecting line with `geom_line()`.

```
diamonds %>%
  group_by(clarity) %>%
```

```
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = 1)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```

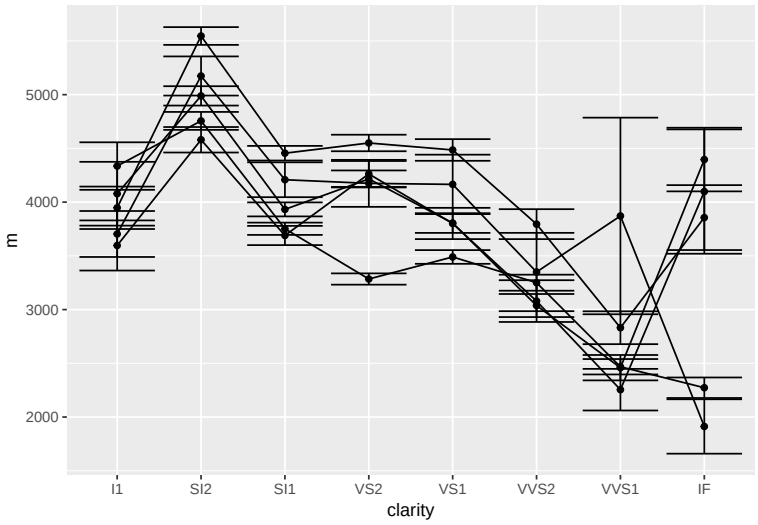


Notice that in `ggplot()`, we also added a group argument nested inside `aes()`. The group argument is required so that R knows which points to connect. Setting group = 1 indicates that all points should be connected.

10.2 Multiple Independent Variables

Since most data is a bit more complex, let's move on to graphing an additional variable, cut. Here, we graph each color's price (as before) for each cut (new addition to the graph):

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```



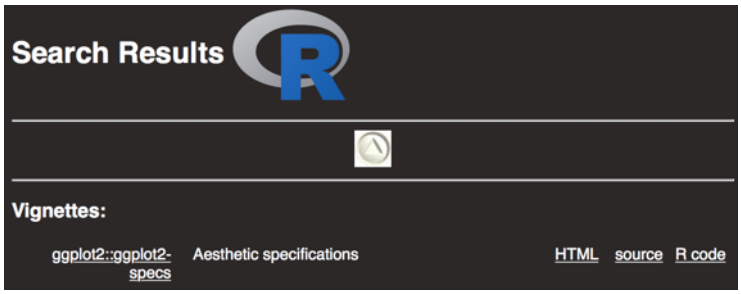
We've added the cut variable within `group_by()` and specified that the points should be connected according to their cut category. Since all of the data points look the same (black, round point), the legend is missing. However, each of those lines represents a different diamond cut and should be differentially labeled. To do this, we need to change the aesthetics so that each cut is visually distinct.

10.3 Basic Aesthetics

In R, the `aes()` function is often used within other graphing elements to specify the desired aesthetics. The `aes()` function can be used in a global manner (applying to all of the graph's elements) by nesting within `ggplot()`. It can also be used for specific graph elements by nesting `aes()` in those specific geom functions (`geom_point()`, `geom_line()`, `geom_errorbar()`, etc.,); more on this later.

10.3.1 The `aes` vignette

Execute `??aes` to view the options for graphing aesthetics and select the "Aesthetic specifications" vignette.



This help page will go over how to alter the appearance of:

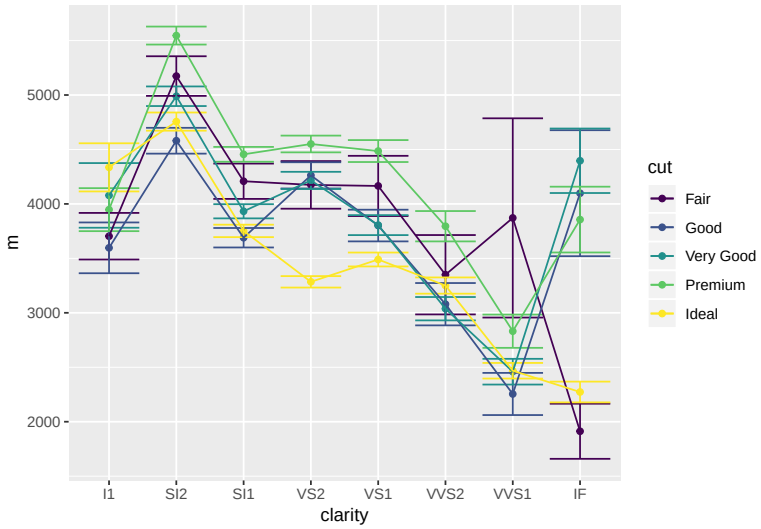
- Color and fill of various objects
- Lines size/type
- Point shape, color, and fill
- Text size, font, font face, justification

There is an abundance of resources to help you graph using the `ggplot2` package. This guide will go over a select few that are frequently used.

10.3.2 Color

Let's change the graph so that each cut is differentiated by color:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```



10.3.3 Coloring Structural Elements

There is a more specific method of changing the color aesthetics of the graph. Rather than changing the entire color scheme of the graph (points, error bars, connecting lines), you can elect to change one part. This allows for some customization.

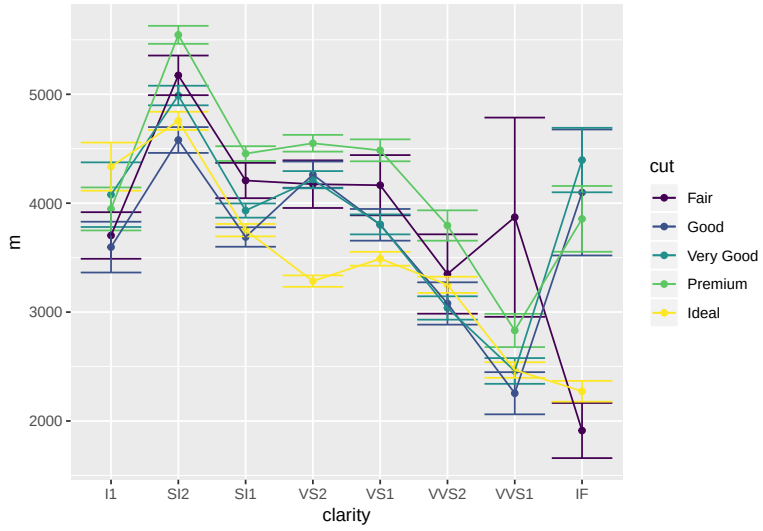
Don't forget to make sure the `sem` function is defined in your environment before following along the examples in this section!

```
sem <- function(x, na.rm = FALSE) {
  out <- sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)}
```

Global Color Changes

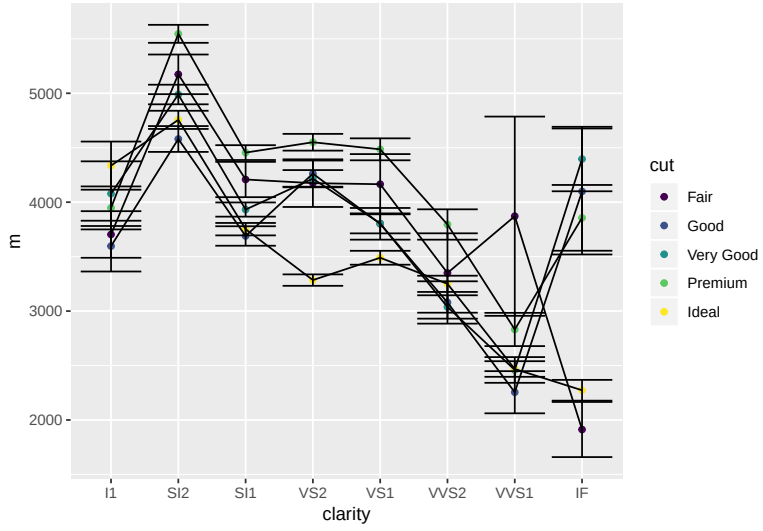
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut,
             color = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se,
```

```
                    ymax = m + se)) +
  geom_line()
```



Changing the Color of Data Points Only

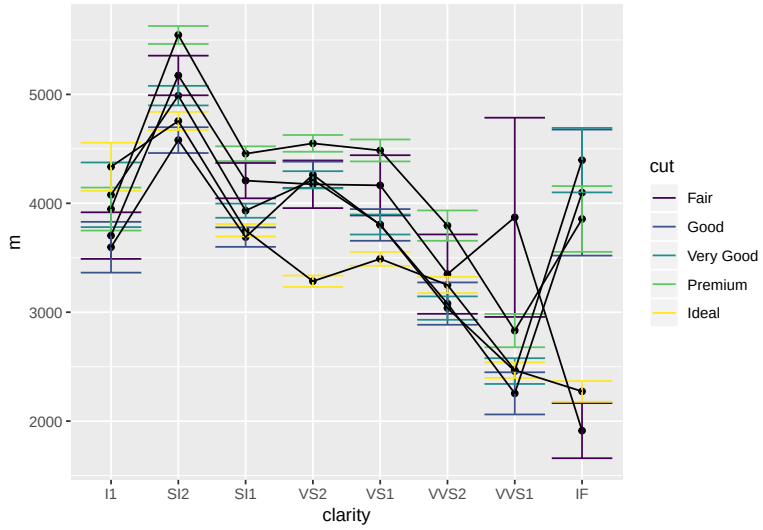
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut)) +
  geom_point(aes(color = cut)) +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se)) +
  geom_line()
```



Changing the Color of Error Bars Only

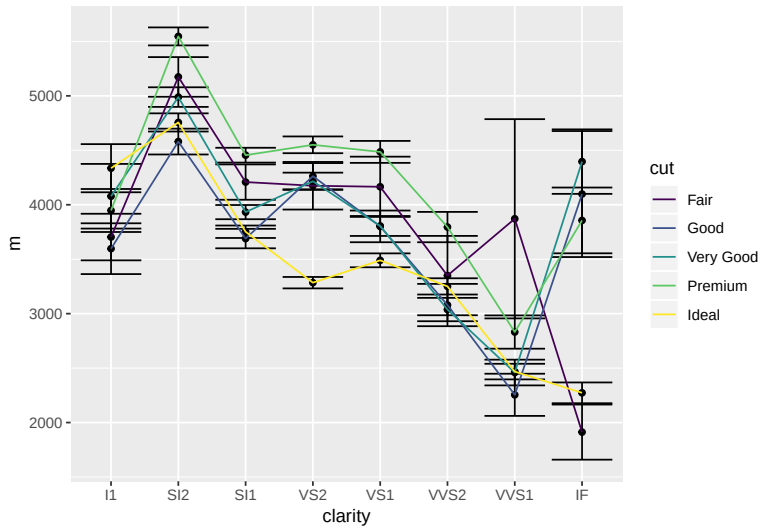
```
diamonds %>%
  group_by(clarity, cut) %>%
```

```
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se,
                    color = cut)) +
  geom_line()
```



Changing the Color of Connecting Lines Only

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se)) +
  geom_line(aes(color = cut))
```

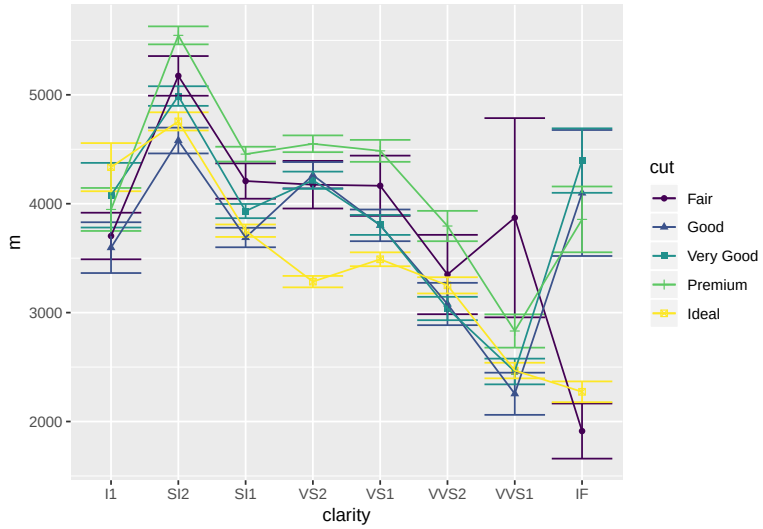


10.3.4 Shape

We could also opt to change the shape of each cut category for further distinction. Notice how each cut category is now represented by a different symbol:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut, shape = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```

```
## Warning: Using shapes for an ordinal variable is not advised
```

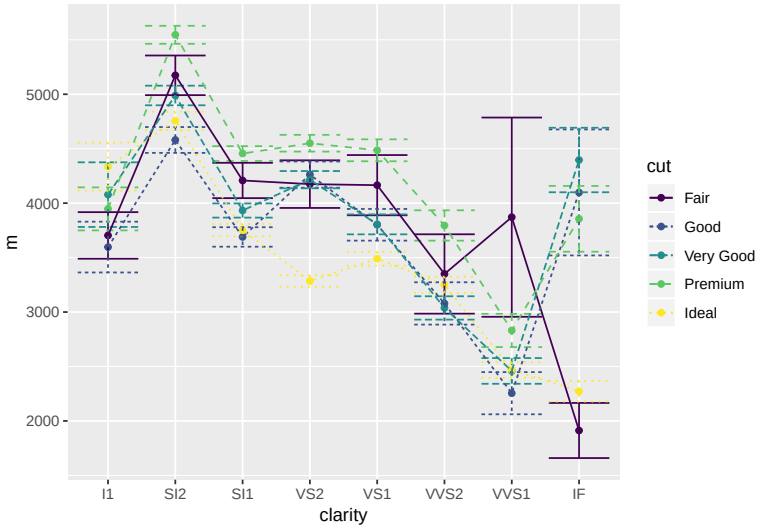


There will be a warning message stating that using shapes for an ordinal variable is not advised. R will occasionally advise you on your code. In some cases, these are error messages that prevent your code from working. In other cases (such as this), there are warning messages that don't prevent your code execution but does provide suggestions.

10.3.5 Line Type

You can differentiate the cut categories by different line types, though this may not be very useful in this particular situation:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut, linetype = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```



Remember that in order to use the `sem()` function in the example above, it must be defined in the environment using:

```
sem <- function(x, na.rm = FALSE) {
  out <- sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)}
```

10.3.6 Inside vs. Outside `aes()`

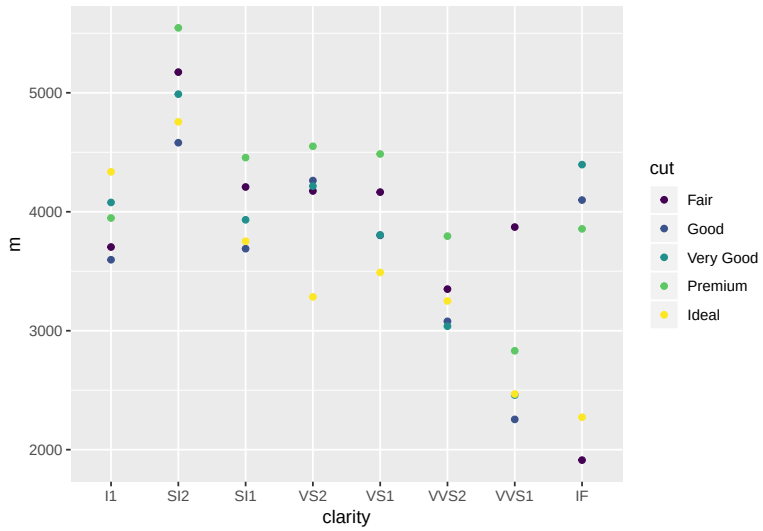
Some aesthetics need to be nested within `aes()` and some do not. How do we know when to place them outside versus inside?

- Aesthetics for a specific variable of your data go inside `aes()`.
- An aesthetic that will remain a constant value – irrespective of the values of your data – should be placed outside of `aes()` and within the geom element.

As an example, let's focus on the color argument in `ggplot()`.

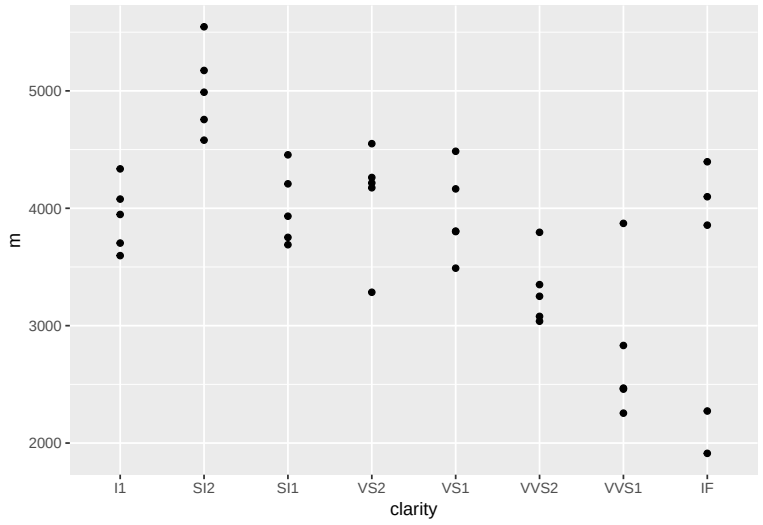
If you were to place the color argument inside of `aes()` as such:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point()
```



If you were to place the color argument outside of `aes()`, such as:

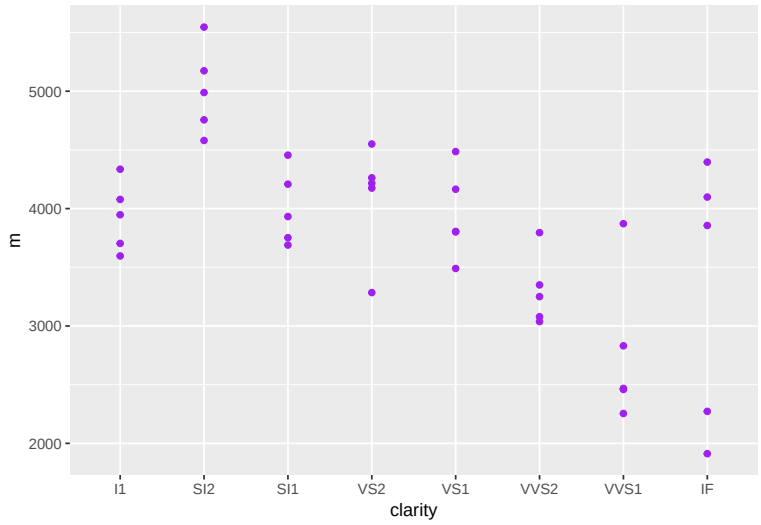
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut), color = cut) +
  geom_point()
```



In this example, setting the color argument to a variable, cut, requires that the color argument be nested inside `aes()`. This is because the color argument in `ggplot()` is referring to coloring a particular variable, not just the data points (`geom_point()`) overall.

In order to specify that we want all data points displayed in a single color, we could execute:

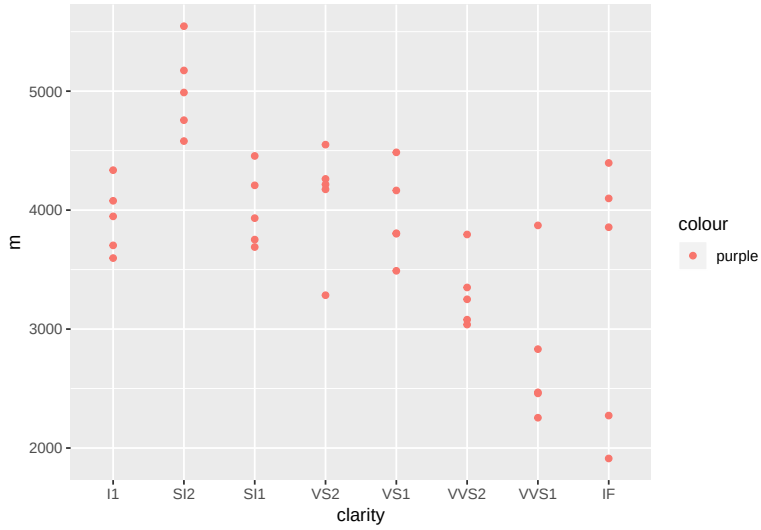
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut)) +
  geom_point(color = "purple")
```



Notice that in order to specify a single color for all points, you must place the color argument outside of `aes()` under the specific geom element. Here, the color of the data points does not consider each cut category – the color is the same across all cuts (Fair, Good, Very Good, etc.).

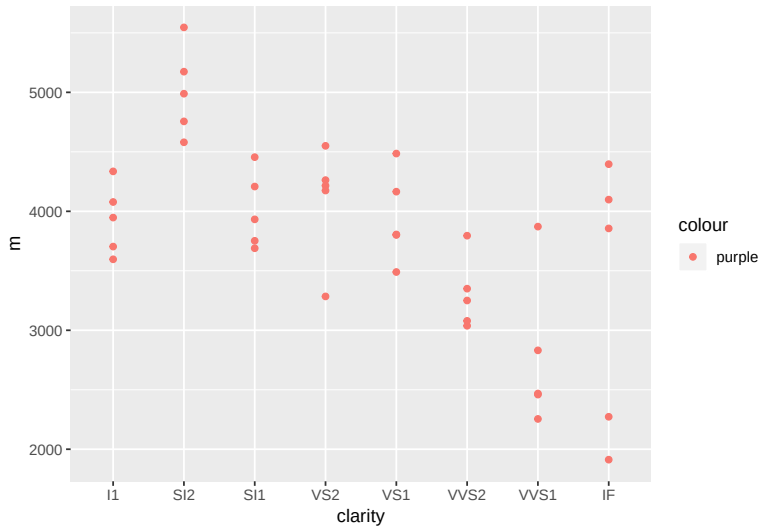
When you place a color argument inside `aes()`, R recognizes the argument's value as a variable.

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut)) +
  geom_point(aes(color = "purple"))
```



OR

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = "purple")) +
  geom_point()
```



In the above examples, R thinks that "purple"" represents a variable name (notice the legend). We know that the user is attempting to color the data points purple, but R does not logically evaluate the code as such. R has a predetermined method in which it determines how code is read and will not "fill in the blanks" and assume the user intended actions.

Though "purple"" is not an existing variable name in the diamonds dataset, R still recognizes that the code is attempting to change the color. As a result, R will default to the color red.

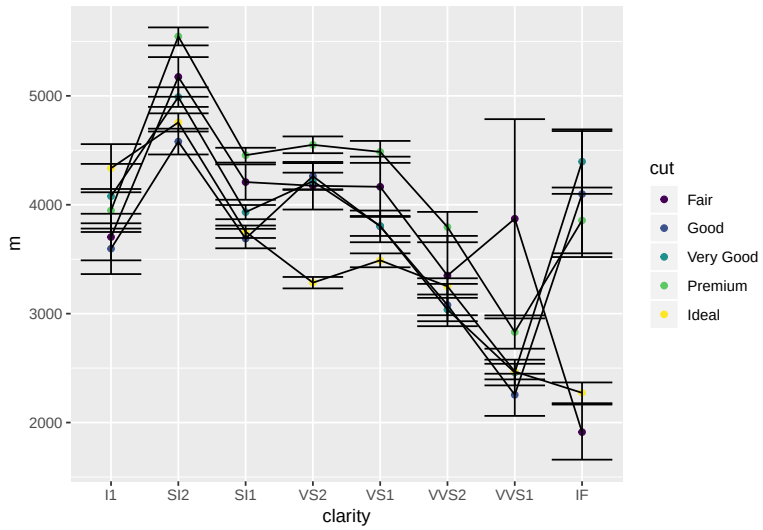
10.4 The Order of the Layers Matter

Don't forget to make sure the `sem` function is defined in your environment before following along the examples in this section!

```
sem <- function(x, na.rm = FALSE) {
  out <- sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)}
```

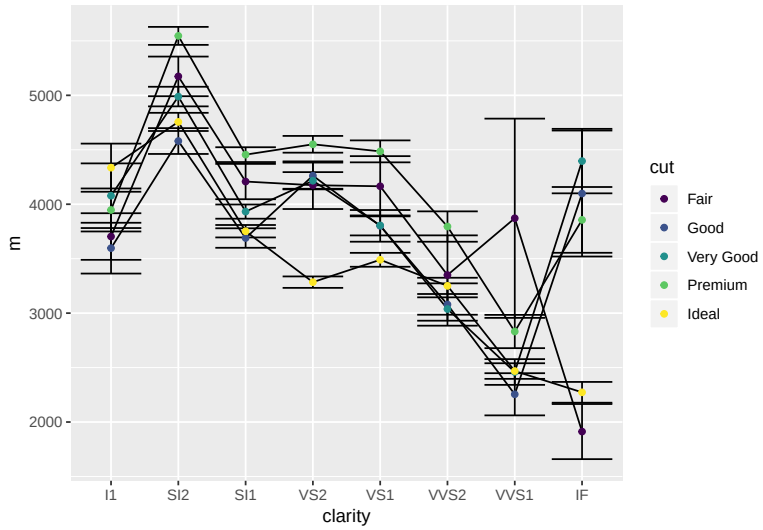
You may have noticed that the graph with only colored data points is difficult to see. That's because the colored are underneath the error bars and connecting lines. Remember that building a graph requires the addition of multiple layers stacked on top of each other. The data points sit beneath the error bars and lines because it is higher up in the code:

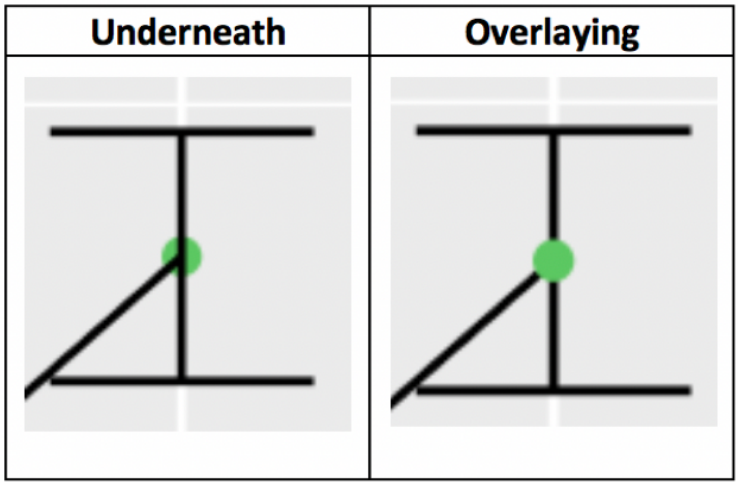
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut)) +
  geom_point(aes(color = cut)) +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se)) +
  geom_line()
```



Moving the `geom_point()` element below `geom_line()` will result in the points laying over the error bars and connecting lines:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut)) +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se)) +
  geom_line() +
  geom_point(aes(color = cut))
```





Of course, this isn't as important when everything (error bars, lines, points) is the same color.

10.5 Grouped Aesthetics

The ggplot2 has a lot of built-in aesthetics for color, point shapes, line types, etc. The easiest ways to know your options is to complete an internet search for phrases such as:

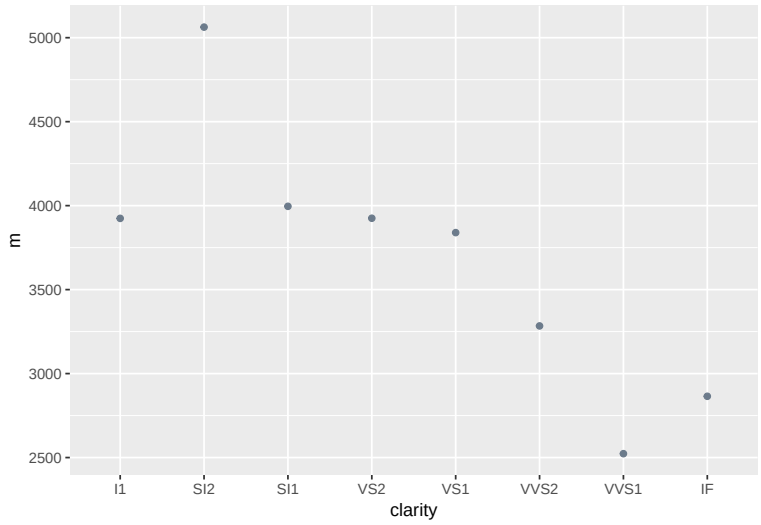
- ggplot2 colors
- ggplot2 point shapes
- ggplot2 line types

Here is a quick list of built-in aesthetic options:

10.5.1 Colors

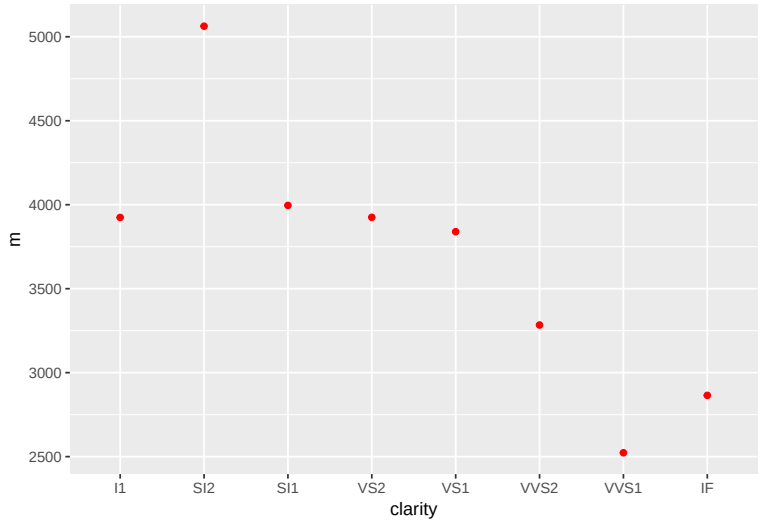
Colors values can be entered in as string/word form. Both British and American spellings are accepted (i.e., grey and gray; color and colour). There is a list of over **600 colors** to choose from by their explicit name – execute `colors()` for a full list:

ex. `"purple"`, `"pink"`, `"grey"`, `"steelblue2"`



Colors can also be specified by a hexadecimal code. This is a six digit code that defines a color by various levels of red, blue, and green (#RRGGBB)

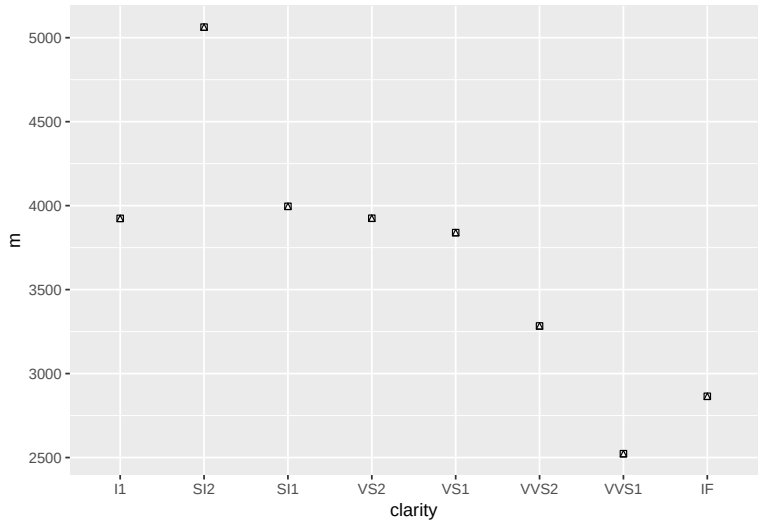
ex. `"#FF0000"`, `"#FFD700"`, `"#FF6347"`



There are also color palette packages that allow you to change the color themes (e.g., RColorBrewer and wesanderson are examples of such packages)

10.5.2 Point Shapes

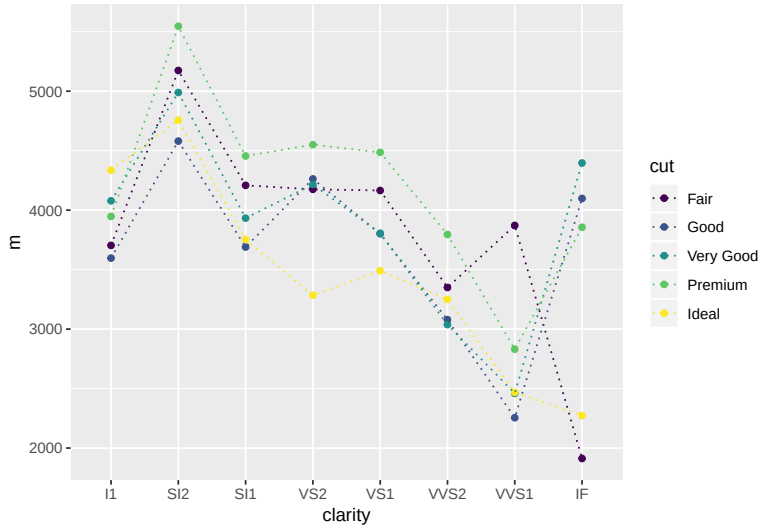
Shapes of data points are specified with a designated integer label. The shape argument is typically placed within the `geom_point()` element. A list of options can be found in the `??aes` vignette. The number labels range from 0-25 for standard shapes, where 21-25 are shapes that can use a fill argument (more on fill later).



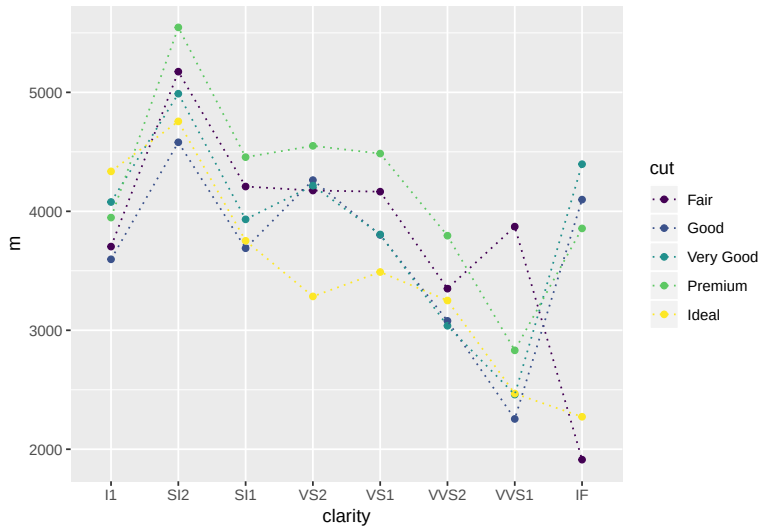
10.5.3 Line Types

The `linetype` argument can be used to change the aesthetic of lines. There are six various line types that can be specified by an integer label or a character/string:

- 1 or `"solid"`
- 2 or `"dashed"`
- 3 or `"dotted"`
- 4 or `"dotdash"`
- 5 or `"longdash"`
- 6 or `"twodash"`



or



Again, I recommend referring back to the `??aes` vignette to view the various aesthetic options. This help page will list the names and integer codes for each aesthetic option.

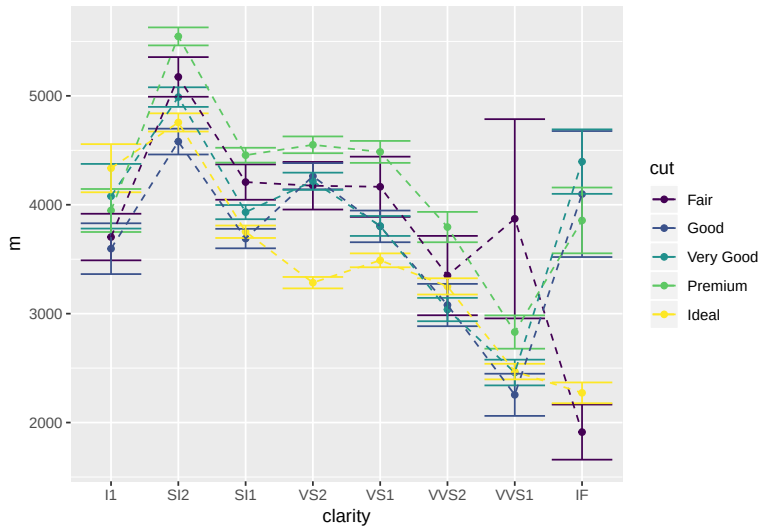
10.5.4 Examples

For the upcoming example, the `sem` function must be loaded to the environment:

```
sem <- function(x, na.rm = FALSE) {
  out <-sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)}
```

Suppose that you want to set the line type for all cut categories in the diamonds dataset to the built-in linetype option called "dashed" or 2. Remember that line types can be referred to by their name or by their unique integer. In this example, we will use the integer.

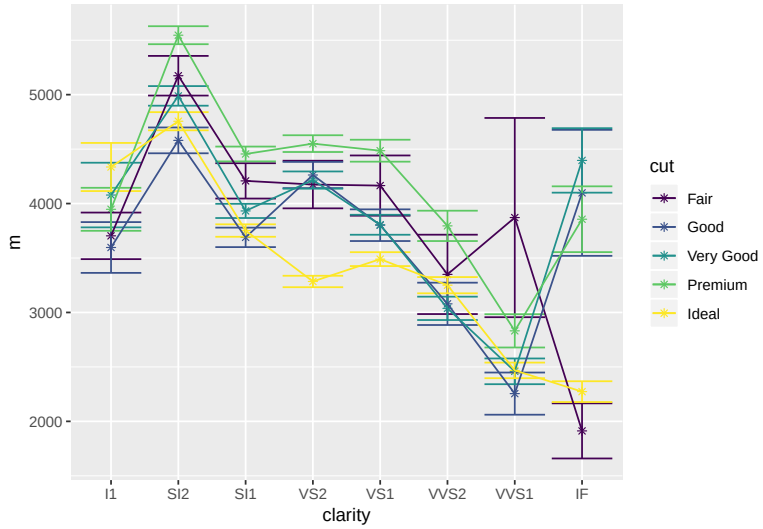
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line(linetype = 2)
```



Notice that in this above case, the line type was placed outside of the `aes()` function. Since we want all line types in the graph to have dashed connecting lines, this information will not be enclosed in `aes()`.

In a similar manner, the data points can be collectively changed in shape. Here, the shape of each data point was changed to an asterisk (i.e., shape number `8`):

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point(shape = 8) +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```

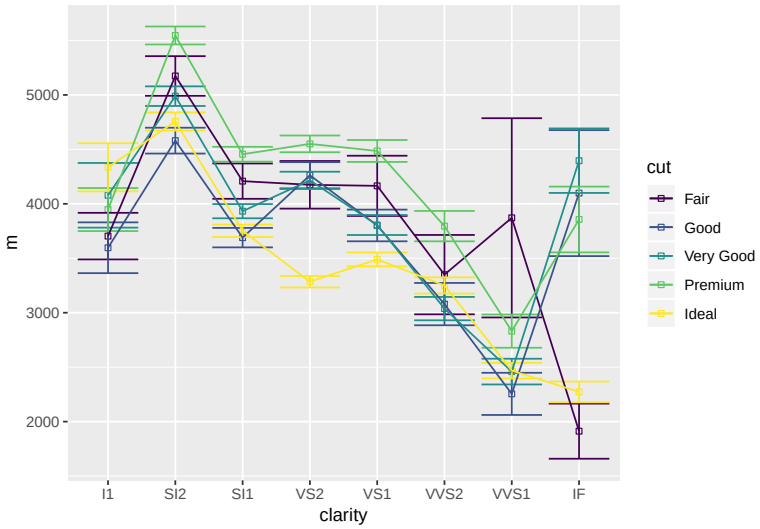


You can also choose a shape in which a separate fill argument is required. These shapes are listed in the aes vignette (`??aes`).

Changing the shape to `22` will produce empty square data points:

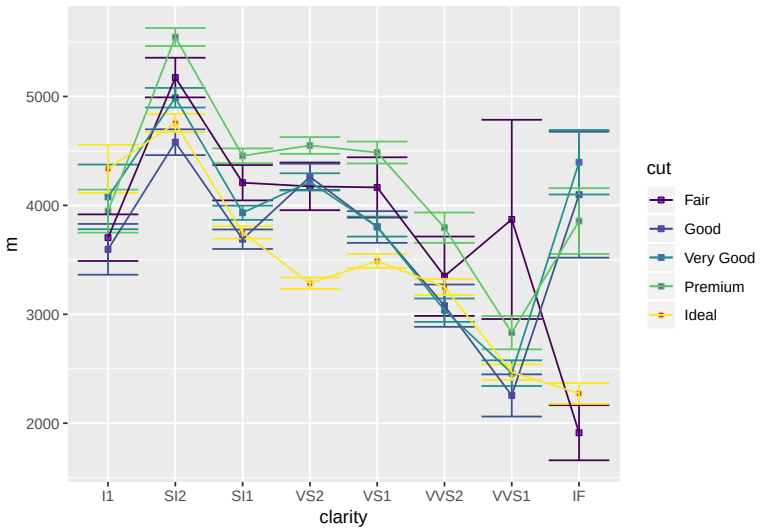
```
diamonds %>%
  group_by(clarity, cut) %>%
```

```
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point(shape = 22) +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```



By adding a fill argument, these data points can be filled with a specific color:

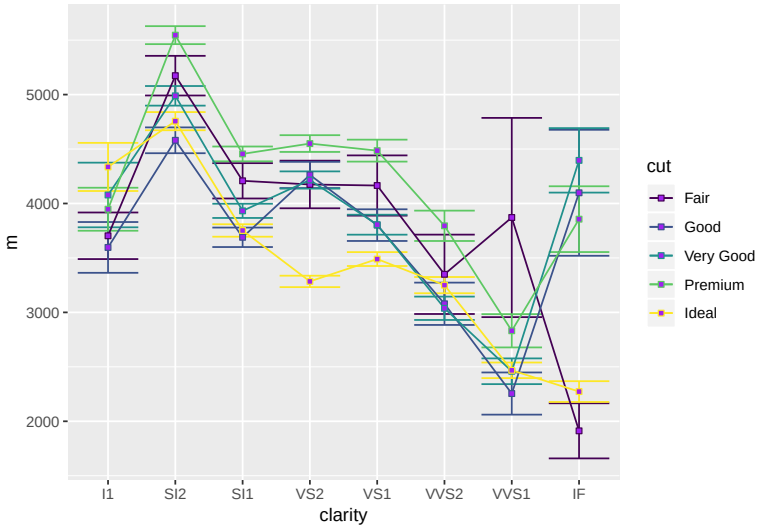
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point(shape = 22, fill = "purple") +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line()
```



It can be a little hard to see that the data points are filled because they lay underneath the other geom elements. In this case, it is wise to **overlay** the data points by shifting `geom_point()` down to the bottom

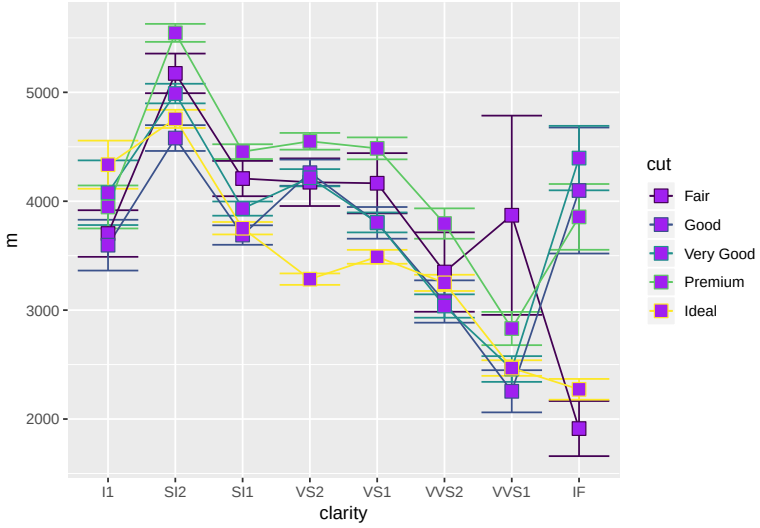
of your code (see previous section about Layering Order).

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line() +
  geom_point(shape = 22, fill = "purple")
```



We can also adjust the size of each data point to further improve visuals:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_errorbar(aes(ymin = m - se, ymax = m + se)) +
  geom_line() +
  geom_point(shape = 22, fill = "purple", size = 4)
```



The size can be adjusted to any numeric value (including decimals).

10.6 Manual Changes

Don't forget to make sure the `sem` function is defined in your environment before following along the examples in this section!

```
sem <- function(x, na.rm = FALSE) {
  out <- sd(x, na.rm = na.rm)/sqrt(length(x))
  return(out)}
```

10.6.1 Coloring Individual Values

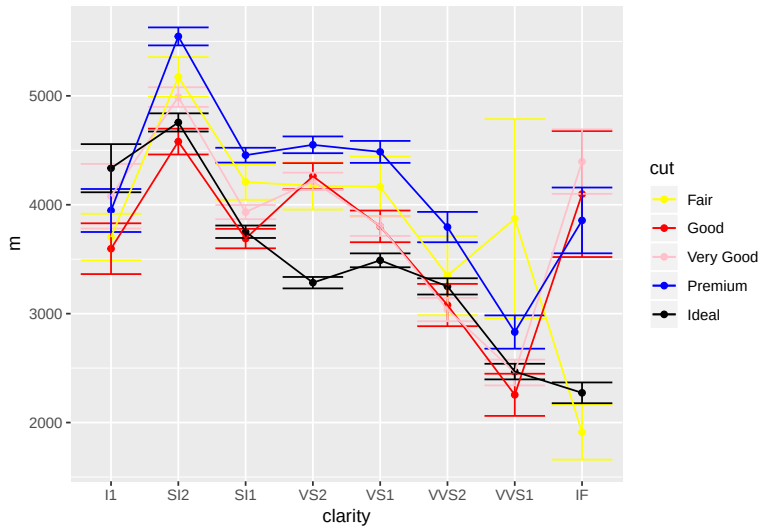
So far, we've seen how we can change the aesthetics of the graph in terms of `color`, `shape`, and `linetype`. We've also seen that you can specifically color each geom element individually (i.e., point, line, and error bars). However, R has has a default color *scheme*. So far, we have not specified the exact color for each value. That is, R has picked the color purple for "Fair" diamonds, dark blue for "Good" diamonds, light blue for "Very Good", etc. What if we wanted to specify each `cut`'s color on our own?

In order to do this, we first have to create a new object that holds the designated colors for each `cut` category. The label for the object in the example is `pointcolor`. This name was chosen for descriptiveness, but you can choose to name it however you'd like (remember that objects can be labeled however you want, but it's important that it is descriptive and concise).

```
pointcolor <- c("Fair" = "yellow",
                "Good" = "red",
                "Very Good" = "pink",
                "Premium" = "blue",
                "Ideal" = "black")
```

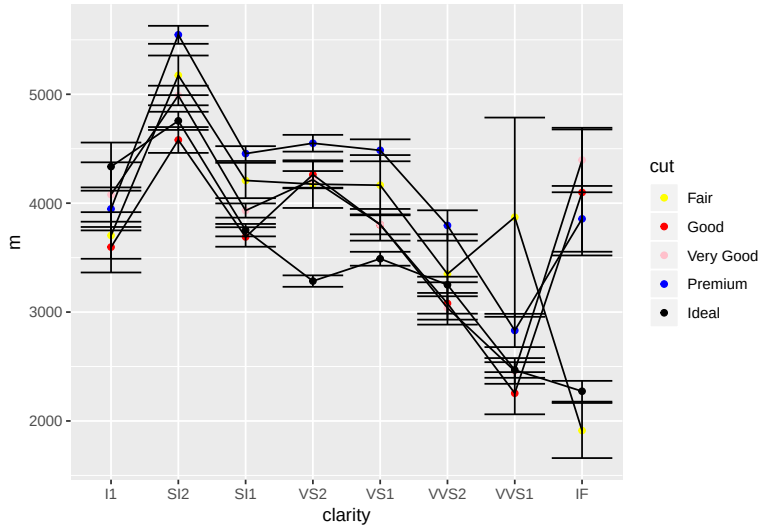
Then, we must execute the graphing code:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut,
             color = cut)) +
  geom_point() +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se)) +
  geom_line() +
  scale_color_manual(values = pointcolor) # manual color change
```



Having trouble running the code? Refer back to the troubleshooting section (3.6)!

Play around with moving the aesthetics. See what happens when you move `color = cut` inside the `geom_point()`:



You could also have chosen to exclude the names for each cut category as follows:

```
pointcolor2 <- c("yellow",
                 "red",
                 "pink",
                 "blue",
                 "black")
```

However, the order in which you list the colors will determine how each cut category is colored. For example, the following will not produce the same colored graph despite containing the same colors:

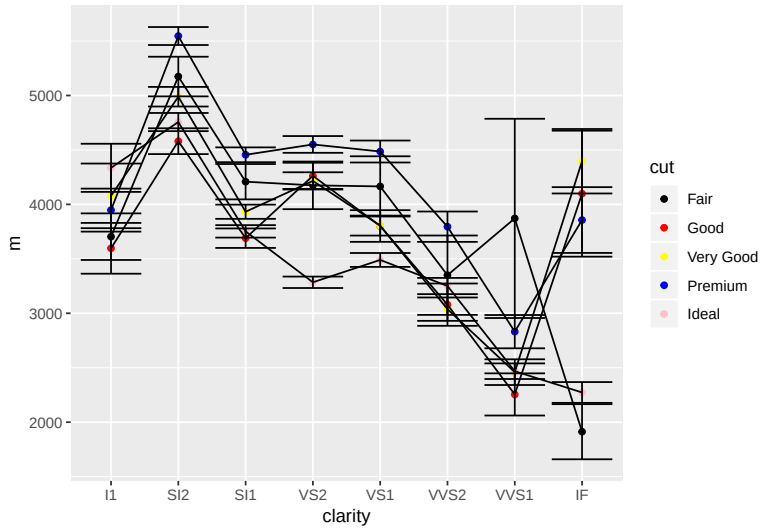
```
pointcolor3 <- c("black",
                 "red",
                 "yellow",
                 "blue",
                 "pink")
```

Instead, `pointcolor3` would be the equivalent to:

```
pointcolor <- c("Fair" = "black",
                "Good" = "red",
                "Very Good" = "yellow",
                "Premium" = "blue",
                "Ideal" = "pink")
```

Remember, if you were to use `pointcolor3` to color your graph, you must update the object name in your graphing code (again, this code relies on the sem function to be available in the global environment beforehand):

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            se = sem(price)) %>%
  ggplot(aes(x = clarity,
             y = m,
             group = cut)) +
  geom_point(aes(color = cut)) +
  geom_errorbar(aes(ymin = m - se,
                    ymax = m + se)) +
  geom_line() +
  scale_color_manual(values = pointcolor3)
```



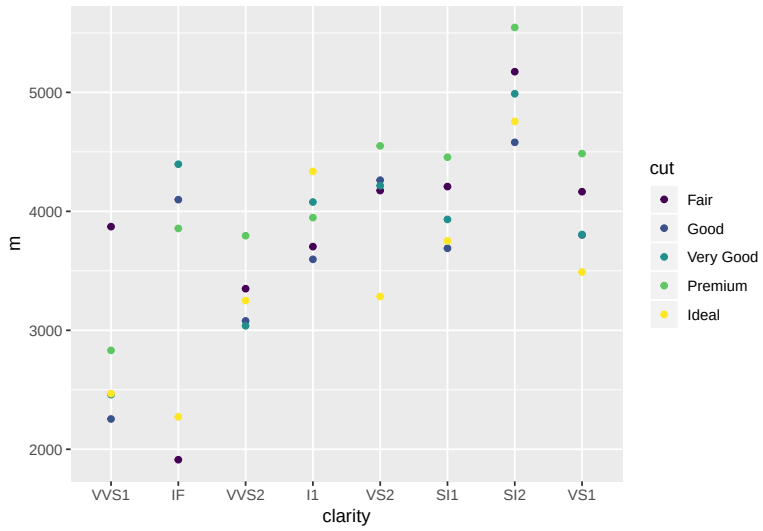
10.6.2 Order of the X-axis

It is possible to also change the order in which the categorical values are arranged on the x-axis. There are two main ways of doing this:

1. Change the individual graph only
2. Change the dataset

Changing the x-axis Order for the Individual Graph Let's say that I want to change the order of the x-axis so that the clarity is out of order. Changing how the graph is arranged is the simplest and the most localized. Simply alter the dataset's variable via `mutate()`:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ungroup() %>%
  mutate(clarity = factor(clarity, levels = c("VVS1", "IF", "VVS2",
                                              "I1", "VS2", "SI1", "SI2", "VS1"))) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point()
```

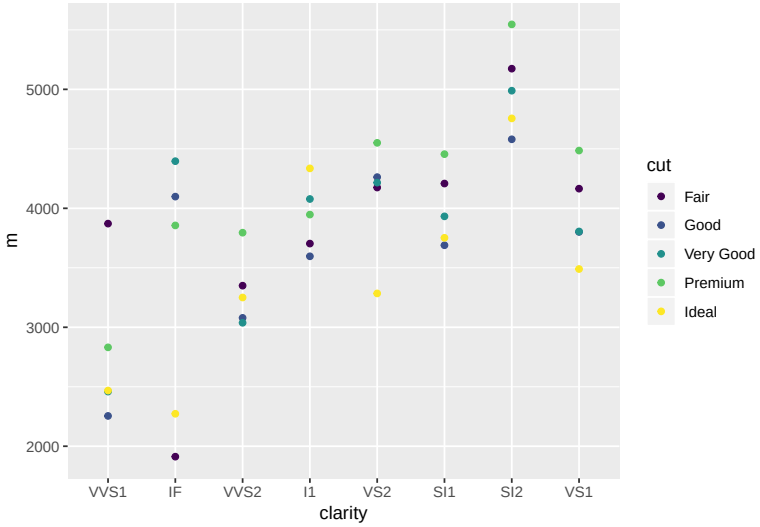


Changing the x-axis Order for the Entire Dataset This is very similar to the above method. The difference is that you save the changes from `mutate()` to the data object. Here, `diamonds_edit1` is the name of **a new object** that is defined with the new changes we made to `clarity`.

```
diamonds_edit1 <-
  diamonds %>%
  mutate(clarity = factor(clarity,
                          levels = c("VVS1", "IF", "VVS2",
                                     "I1", "VS2", "SI1", "SI2", "VS1")))
```

THEN

```
diamonds_edit1 %>% # take notice of the new object here
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ungroup() %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point()
```



Remember to be wary of saving over objects. For beginners at R, I recommend **creating new objects** (as in the above example) when making permanent changes to a dataset. This avoids mass confusion and error messages that arise from renaming an object with the same name (i.e., saving over another object).

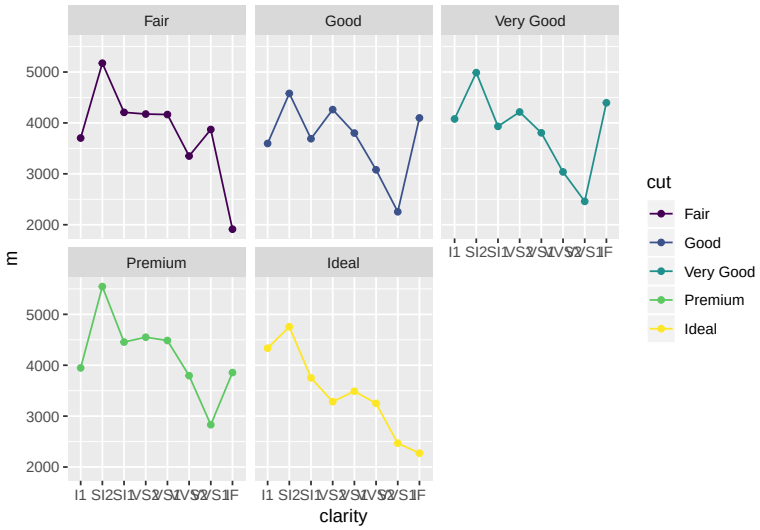
10.7 Facet Wrapping

Facet wraps are a useful way to view individual categories in their own graph.

For example, if you wanted to make a separate graph for each cut measuring the `price` (y axis) for each `clarity` (x axis), you could add `facet_wrap(~cut)`.

The tilde (~) can be read as "by" as in: > "I want to make a new graph separated by cut categories."

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut)
```



You can also separate by multiple variables by adding a + between each variable:

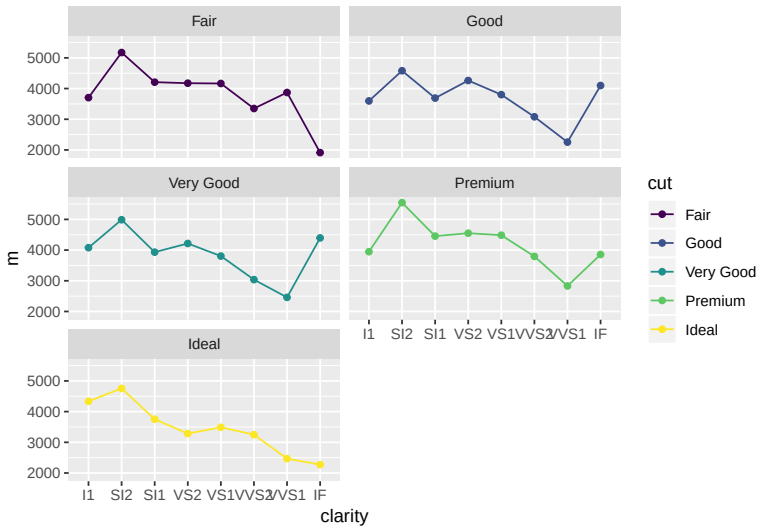
```
diamonds %>%
  group_by(clarity, cut, color) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut + color)
```

Note that `color` *inside* `group_by()` *and* `facet_wrap()` *refer to the variable name (i.e., the color of the diamond, not the color of the graph elements)*

Here, we've added a third grouping variable, color, which allows us to make separate graphs by cut and color. The cut category is listed as the first subheading and the color value is listed underneath. Note that you can add as many (categorical) variables as you'd like in your facet wrap, however, this will result in a longer loading period for R.

Further, you can specify the number of columns/rows to display as such:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2)
```



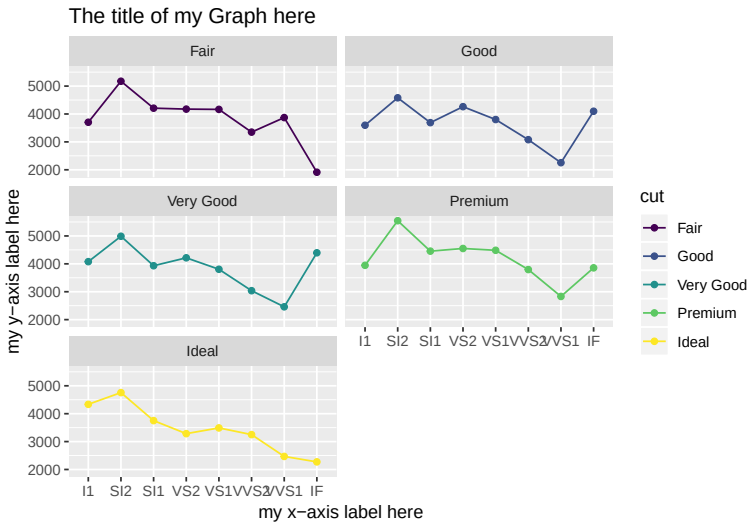
Notice that the graphs are now arranged with two columns instead of the previous default of three. Alternatively, executing `facet_wrap(~cut, nrow = 3)` would produced the same graph.

10.8 Labeling Your Graph

Labeling your graph with axes and main titles is a matter of adding another line to the code we've already built. You'll notice that building a graph in R requires a command for each component. You must specify first that there is a graph (`ggplot()`), that there are data points on the graph (`geom_point()`), that there is a connecting line between the data points (`geom_line()`), that there are error bars (`geom_errorbar()`), and so on. The same principle applies to labels.

Using the `?labs` help page, we see that the `labs()` function is the most versatile. You can specify multiple label components including: `title`, `subtitle`, `caption`, and `tag`. The help page also specifies that the x-axis, y-axis, and plot title can be separate arguments.

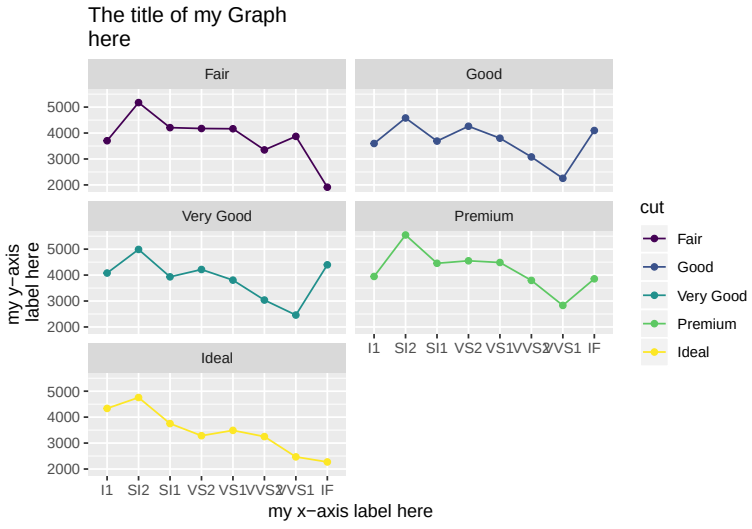
```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x = "my x-axis label here",
       y = "my y-axis label here",
       title = "The title of my Graph here")
```



Notice that the labels are sensitive to capitalization, where capitalizing "Graph" in will be reflected in the graph.

You include line breaks in your labels by using `\n`:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x = "my x-axis label here",
       y = "my y-axis \nlabel here",
       title = "The title of my Graph \nhere")
```

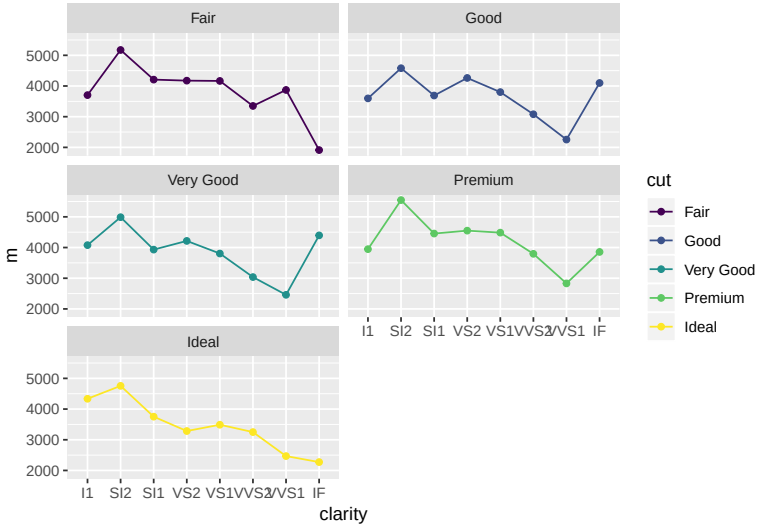


Although it may look awkward, `\n` must be placed precisely in front of the word that initiates the next line. For example, executing `labs(title = "The title of my Graph \n here")` with a space before the word "here" will introduce a space in the graph as well.

10.8.1 Exercises

For each of these exercises, utilize the following base graph:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2)
```



1. Add the subtitle: "Here is a subtitle!" to the base graph *(hint: requires `labs()`)*
2. Add the caption: "Here is my caption"
3. Add the tag: "Fig. 1A"
4. Add the tag: "C"
5. Add an x-axis, y-axis, title, subtitle, caption, **and** tag of your choosing to the base graph

10.9 Themes

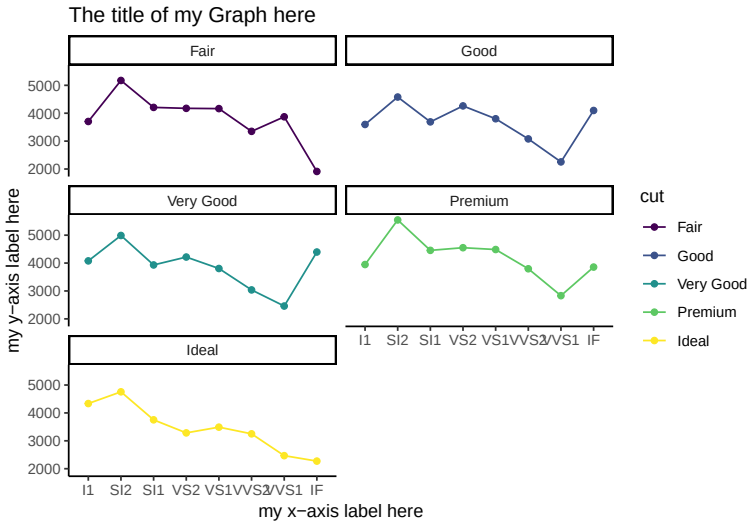
In this section, we are going to focus on beautifying your graph. Although some find the grey gridlines aesthetically pleasing, they are often not considered "publication quality"" graphs by many. Thankfully, the `ggplot2` package has some built-in theme functions that all beginning with `theme_`.

10.9.1 Pre-made Themes

`theme_classic()` is one of my favorite themes to use when I want to create a quick, pretty graph. You'll notice that this theme's default does not contain grid lines:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
```

```
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x= "my x-axis label here",
       y = "my y-axis label here",
       title = "The title of my Graph here") +
  theme_classic()
```



Built-in themes have varying default fonts, font sizes, color palettes, and other styling attributes. Another simple theme is `theme_bw()`, which is a black and white theme. There are many other theme options for R out there that others have created. The `ggthemes` package (installation is required for use of this package), for example, contains additional themes that work with the `ggplot2` graphs; remember that you must install this package and load the library in order to utilize the themes. All of these pre-made themes make it easy and relatively painless to beautify your graph. The alternative is to change aspects of your graph manually.

10.9.2 Customizing the Theme

Sometimes, you may not like all of the design choices of a pre-made theme. In cases like these, you can choose to alter certain aspects under the `theme()` function. In the help page (`?theme`), you'll find a list of the different graphing components that can be altered. To alter these components, you will often have to specify which element (the text, lines, etc.,) of the component you want altered. In this section, we will explore some of the customization options.

10.9.3 Theme Components and Elements

The components of a ggplot theme can be manually altered. For the many different theme components, we can specifically target an element of the component. For more information on theme components, execute `?theme`. For more information on theme elements, execute `?margin`.

Examples of Theme Components	Example of Theme Elements
axis.title	element_blank
axis.ticks	element_rect
axis.line	element_line
legend.background	element_text
legend.position	
legend.text	

Examples of Theme Components	Example of Theme Elements
legend.title	
panel.border	
strip.text	

The basic structure for making manual changes to the theme:

1. The `theme()` function is added to the existing `ggplot()` code:

```
data %>% ggplot(aes(x, y)) + geom_point() + geom_line() + theme()
```

2. Components are nested within `theme()` and must be defined (=, one equal sign) with an element. `...` is shorthand for "the same code as before" (i.e., `data %>% ggplot() + geom_point() + geom_line()` :

```
... theme(axis.title = element_blank())
```

or

```
... theme(axis.title = element_text(color = "blue", face = "italic"))
```

3. Defining a component with a pre-set value (see the help pages for details on the available pre-set values):

```
... + theme(legend.position = "left")
```

*Remember that one equal sign (=) represents the phrase "is defined as" and two equal signs represents the phrase "is equal to". They cannot be used interchangeably! Double equal signs (==) are usually used to specify a value within a variable that already exists whereas single equal signs add new definitions.

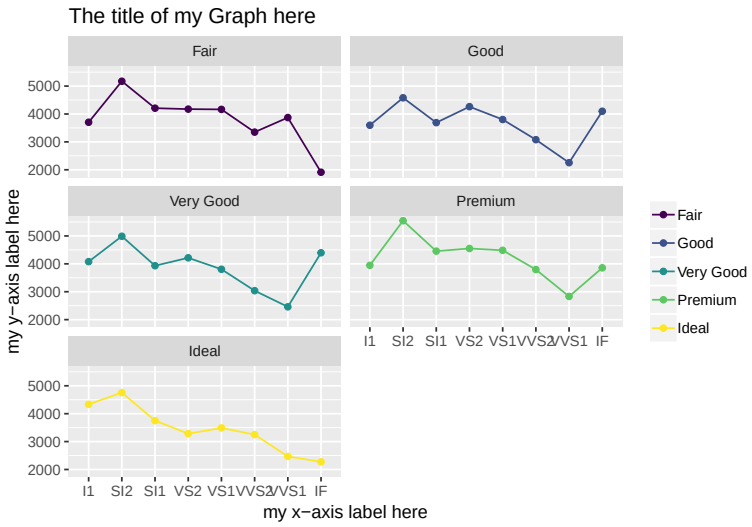
Not all components require element specification (see above example) and not all elements are available for each component. For example, the `legend.position` component does not accept `element_text`, because the position of a legend requires a location (left, right, etc.)! Defining the `legend.position` with an element will result in an error message!

In the upcoming sections, we will learn a few different theme manipulations. As always, refer to the internet (e.g., Stacked Overflow) for additional guidance.

10.9.3.1 Removing the Legend Title

The `element_blank()` function, used within the graph component legend.title, will achieve this:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x= "my x-axis label here",
       y = "my y-axis label here",
       title = "The title of my Graph here") +
  theme(legend.title = element_blank()) # removes legend title
```

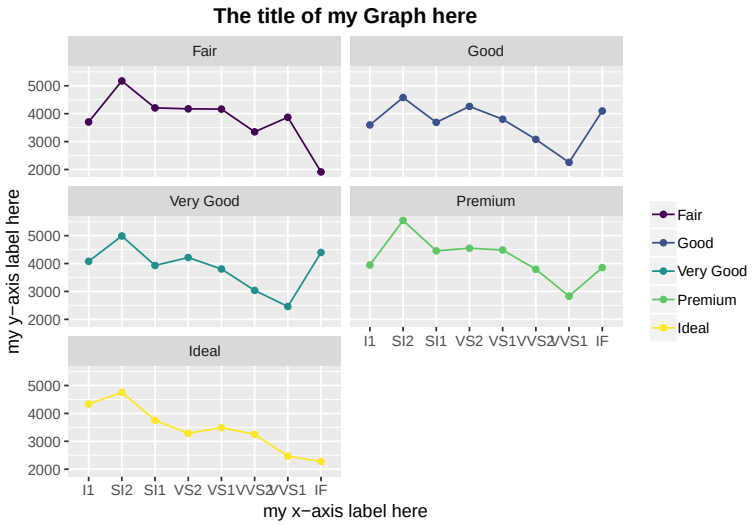


Notice that the `cut` label is no longer above the legend.

10.9.4 Centering and Bolding the Plot Title

Because these changes involve the text of the plot title, the `element_text()` function is more appropriate here.

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x = "my x-axis label here",
       y = "my y-axis label here",
       title = "The title of my Graph here") +
  theme(legend.title = element_blank(),
        plot.title = element_text(hjust = 0.5, face = "bold")) # alters the plot.title
```



Notice that multiple changes can be added, separating each change with a comma. In `plot.title = element_text(hjust = 0.5, face = "bold")`, `hjust` refers to the horizontal justification. That is, along the horizontal plane, the text should sit halfway across the graph (hence, 0.5). The default value for `hjust` is 0, which aligns the text to the left. To align the text to the right, the value of `hjust` needs to equal 1. There is also the option to align the text somewhere in between 0 and 1 other than 0.5. The `face` argument refers to the font face, which can be `bold`, `italic`, `bold.italic`, or the default `plain`.

e.x. hjust = 0.24 face = "bold.italic"

There are many different options for customizing graph elements. Just as you can change the size, color, and line type of the data points and lines, you can change the aesthetic of just about any other graph element.

10.9.5 Exercises

Using the following foundation, manually adjust the theme accordingly. Each exercise is separate from the others, so make sure to only change components listed within an exercise (i.e., each exercise must start with the base graph below):

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x = "my x-axis label here",
       y = "my y-axis label here",
       title = "The title of my Graph here")
```

1. Change the plot title to an italic face. Did you remember to add the `theme()` argument?
2. Change the plot title to a bold and italic face (hint: check out the `?margin` help page).
3. Change the legend title to a bold face (hint: you will need to replace `element_blank()` with `element_text()`).
4. Align the legend title to a horizontal justification of 0.5.
5. Align the legend title to a horizontal justification of 0.75.
6. For the legend title, set the value of the "face" argument equal to 2 (`face = 2`). What do you see here? What happens when face = 3? face = 4?
7. Change the size of the plot title to 5 (hint: within `element_text()`, enter `size = 5`).
8. Change the font family of the plot title to "Times New Roman" (hint: `family = "Times New Roman"`).
9. Remove the legend entirely (hint: legend.position = "none" directly within `theme()`). Do you still have the legend.title argument under theme() as well? Does it make sense to keep this argument for this specific example (why not)?
10. Under `theme()`, execute legend.position = c(0.90, 0.55). What happened? Change it to `c(0.90, 0.20)`. Change it to `c(0.9, 1)`. Change it to `c(0.2, 1)` and bold the legend title.
11. Color the plot title blue (hint: color = "blue").
12. Remove the plot title using `element_blank()`. What's a simpler way of doing this?
13. Change the angle of the x-axis labels to 45 degrees (hint: use `axis.text.x.bottom()` and `element_text()`)

14. Fill the `plot.background()` with the color blue (hint: use `plot.background` and `element_rect()`)
15. Remove the major panel grids (hint: use `panel.grid.major` and `element_blank()`)
16. Remove the minor panel grids
17. Remove the panel background
18. Change the axis line to a dotted black line (hint: use `axis.line()`, `element_line()`, `linetype = "dotted", color = "black"`)
19. Perform exercises 15-18 in a single graph
20. Add a size 5 panel border (hint: use `panel.border()`, `element_rect`, `size = 5`, `fill = NA`). What happens when you don't specify `fill = NA`?

10.9.6 Customizing Pre-made Themes

You can also opt to use a pre-made theme in addition to manual modifications. This can save time, especially if you like some style components from the pre-made theme. For example, I do like how `theme_classic()` removes the ugly gridlines and formats the facet titles, so I'm going to keep `theme_classic()` but manually remove the legend title:

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2) +
  labs(x= "my x-axis label here",
       y = "my y-axis label here",
       title = "The title of my Graph here") +
  theme_classic() + # adding a built-in theme
  theme(legend.title = element_blank()) # adding custom theme edits
```

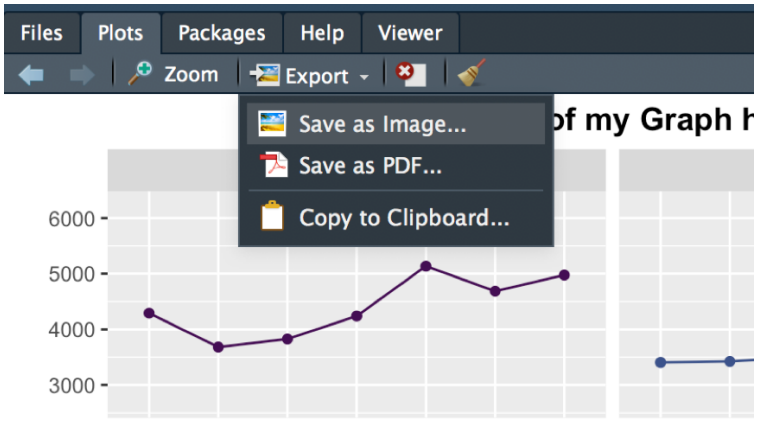
There is an endless plethora of design choices one can make with a graph – more than this guide can cover. Luckily, there is also an abundance of resources that can help you modify your graph.

10.10 Saving your Graph

Congratulations for getting this far! You now know the basics for creating an R line graph. An R graph can be saved in multiple ways.

10.10.1 Point and Click

In the Plot tab of your bottom right pane, you can manually save each graph as they appear in the pane. This method will only save the graph that is currently displayed and is somewhat tedious when multiple graphs need to be saved at once.



10.10.1.1 Save via `cairo_pdf()`

The `cairo_pdf()` function is my ideal method of saving graphs to a pdf. It is compatible with both the Apple and Windows IOS (I use a Mac laptop and a Windows Desktop) and is the most user friendly (to date). In order to produce a pdf, the graphs need to be saved as objects in the environment:

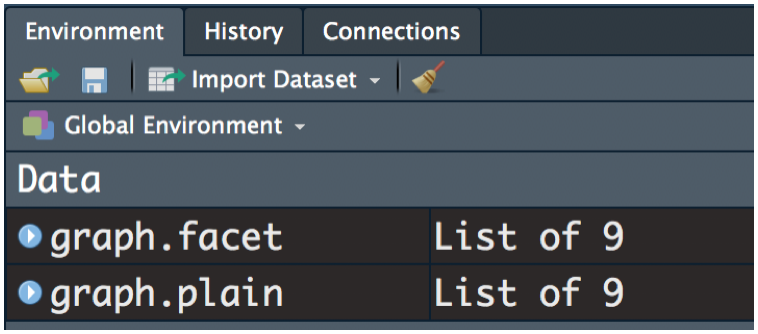
Here, the graph is saved as an object named `graph.facet`:

```
graph.facet <- # saving this graph as an object
  diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line() +
  facet_wrap(~cut, ncol = 2)
```

Next, I'll let's save second graph that we'll name `graph.plain`:

```
graph.plain <-
  diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) +
  geom_point() +
  geom_line()
```

Now that we have both `graph.facet` and `graph.plain` saved as objects in our environment:



We can execute the following code all at once:

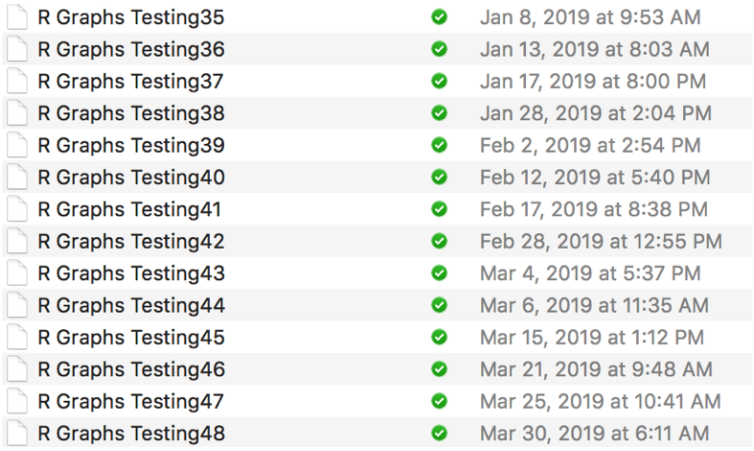


Figure 10.1: An example of version control for graph files. Each time the graphs are updated, a new PDF is created so that the older versions are still maintained.

```
cairo_pdf(filename = "R Graphs Testing ",
          width = 11, height = 8, onefile = TRUE)
graph.facet
graph.plain
dev.off()
```

Here, I've named my PDF file "R Graphs Testing" and have specified the width and height of the document, which is currently set up to a landscape orientation for a standard letter-sized sheet of paper. If I want to switch the dimensions to yield a classic portrait orientation PDF, I would set `width = 8` and `height = 11`. The onefile argument specifies that all of the graphs will be put into one file. If the code were `onefile = FALSE`, R would produce separate PDF files for each graph. It is imperative that you run all of these lines in succession. That is, executing the first line in the above example lets R know that you're beginning to save objects to the PDF file. A temporary PDF file will be added to your working directory but will not contain anything. Once `dev.off()` has been executed, the PDF file will be saved and ready to view in your working directory. You may choose to add additional graphs or only save a single graph. Regardless, make sure those object names remain between the `cairo_pdf()` function and the `dev.off()` function.

After saving your graphs, I recommend inactivating all of the text which saves your graphs (highlight text from `cairo_pdf()` to `dev.off()` and hitting **Ctrl + Shift + C**). This prevents you from saving over a previous PDF. In fact, I recommend that a new file should be created each time you save a graph. This can be done by slightly altering the name of the file so that older versions of the file can be maintained should you need to revert to an older version. An easy naming alteration can be adding a new number after the title, such as "R Graphs2", "R Graphs 3", etc.

A more generalized method is to tag the file with the system date using the `Sys.Date()` function. This function specifies the current date in *yyyy-mm-dd* format, so it would be useful if you only saved one version per day (if you save these graphs more than once per day, it would be easier to use the numbers method mentioned earlier).

Adding a generalized tag such as this allows the user to update file names **without** the need for manual editing. That is, you wouldn't have to manually enter "R Graphs2" and then "R Graphs3" the next time you saved. Manual editing can be prone to user error. If you forgot to change the number, you could accidentally save over the previous version. However, generalized labels may not always be worth the extra effort, particularly when multiple file copies are not necessary/desired.

To create a generalized label using `Sys.Date()`, execute:

```
cairo_pdf(filename = str_c("R Graphs Testing ", Sys.Date()),
          width = 11, height = 8, onefile = TRUE)
graph.facet
graph.plain
dev.off()
```

The `str_c()` function is used to concatenate strings. The filename argument in `cairo_pdf()` needs to be a string, so utilizing `str_c()` in this manner gives us a basic method for version control. Remember that strings can consist of any text you want, though my advice is to be as concise as possible and avoid use of symbols other than a space, dash(-), or underscore (_).

Try the following examples:

1. Execute `str_c("happy", "birthday")`
2. Execute `str_c("happy ", "birthday")` (hint: notice the space after happy)
3. Execute `Sys.Date()`
4. Execute `str_c("R Graphs", Sys.Date())`
    - Notice that `Sys.Date()` is not within quotations. This is because `Sys.Date()` is a named object with a string value
    - R Graphs cannot be executed because:
    - There is a space (AKA illegal character)
    - R Graphs is not the name of an object
    - `R Graphs` is a value

Notice that `Sys.Date()` does not have to be defined in the environment. This is because it is a built-in object. By default, R recognizes and has already defined some objects. Remember that it is not necessarily common knowledge to know all of the built-in elements of R. Programmers come to learn this kind of information through books that experts have written and "word-of-mouth" usually from reading a question on Stacked Exchange/an R forum.

5. Execute all of the following in succession:
    - `fruit <- c("orange", "apple", "banana")`
    - `fruit`
    - `str_c(fruit, "mango", "grapes")`
    - `c(fruit, "mango", "grapes")`

Again, notice that fruit is the name of an object while `mango` is a value. Remember that objects have to be defined by the user (with `<-`) or built-in by an R package.

6. Execute all of the following in succession:
    - `week <- c("Mon", "Tues", "Wednes", "Thurs", "Fri", "Satur", "Sun")`
    - `str_c(week, "day")`
    - `c(week, "day")`

Chapter 11

Other Graphs

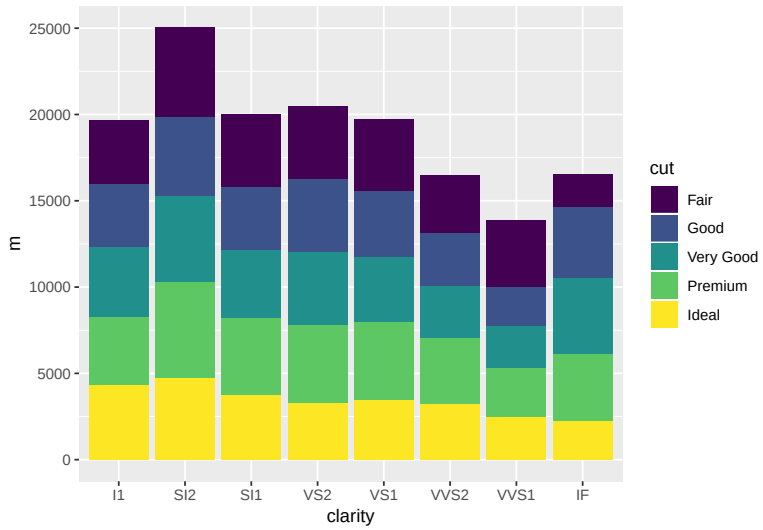
This guide will not go into the same detail for bar graphs, histograms, and box plots as it did for line graphs. One of the reasons for this is that many of the skills we learned for line graphs (theme edits, geom elements, facet wraps, etc.) also work for bar graphs. Another is that line graphs are traditionally used in my home field of behavioral neuroscience as the primary graph type; however, detailed instruction for line graphs are sparser than for other graph types.

In this chapter, we will briefly touch on some of the more unique aspects of these kinds of graphs. If you want more detailed instruction for these "other" graph types, there are plenty of resources on the internet, including Hadley Wickham's R for Data Science: https://r4ds.had.co.nz/. Remember that we use the ggplot2 package to graph our data in this book. There are other methods/packages that can perform similarly, but ggplot2 is the most powerful method. As such, don't forget to add "ggplot2" in your search phrase if you're scouring the internet for additional help.

11.1 Bar Graph

Let's continue to use our graph from diamonds but replace `geom_point()` with `geom_bar()`. Here, we are graphing the average (mean) price of the diamonds by cut category.

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, fill = cut)) +
  geom_bar(stat = "identity")
```



In `geom_bar()`, the default dependent measure is the `count` (i.e., `stat = "count"` by default). In the above example, we've overridden the default count value by specifying `stat = "identity"`. This indicates that R should use the y-value given in the `ggplot()` function. Notice that bar graphs use the fill argument instead of the color argument to color-code each cut category.

If we execute this same code without `stat = "identity"`, this will result in an error:

```
# produces error due to unnecessary y variable
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, fill = cut)) +
  geom_bar() # removed stat = "identity" from geom_bar()
```

By default, R will assume that the `stat` argument of `geom_bar()` is set to `"count"`. If we set `stat` to equal `count`, R will count how many observations (read: rows of data) there are has for each clarity (x-variable) in the `diamonds` dataset. Since R is counting how many diamonds there are for each clarity with `stat = "count"`, we do not need to include a y-variable (dependent variable) in `ggplot()`:

```
# graphing the frequency/count of the clarity categories
diamonds %>%
  group_by(clarity, cut) %>%
  ggplot(aes(x = clarity, group = cut, fill = cut)) + # removed y argument and value
  geom_bar()
# is the same as this:
diamonds %>%
  group_by(clarity, cut) %>%
  ggplot(aes(x = clarity, group = cut, fill = cut)) +
  geom_bar(stat = "count") # stat = "count" is implied in the first example
```

In summary: we need to be mindful of the value we assign to the `stat` argument within the `geom_bar()` function. If it is `stat = "identity"`, we are asking R to use the y-value we provide for the dependent variable. If we specify `stat = "count"` *or* leave `geom_bar()` blank, R will count the number of observations based on the x-variable groupings.

11.1.1 Exercises

1. In the above code, replace `fill = cut` with `color = cut`. What happened?

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price)) %>%
  ggplot(aes(x = clarity, y = m, group = cut, color = cut)) + # changed fill = cut tp color = cut
  geom_bar(stat = "identity")
```

2. Set `geom_bar()`'s `position` argument equal to `"dodge"` using the code below. What do you see?

```
diamonds %>%
  group_by(clarity, cut) %>%
  ggplot(aes(x = clarity, group = cut, fill = cut)) +
  geom_bar(position = "dodge")
```

+ You can also choose to specify how far the bars dodge each other with `position = position_dodge()`:

```
# Try editing the number within the position_dodge() function:
# notice how the bars overlap at a 0.5 value
diamonds %>%
  group_by(clarity, cut) %>%
  ggplot(aes(x = clarity, group = cut, fill = cut)) +
  geom_bar(position = position_dodge(0.5)) # manually determining the dodge distance
```

3. Plot cut (x-axis) vs. price (y-axis)

```
# this calculates the total cost of all diamonds within each clarity category
# for example: "the cumulative cost of all diamonds with an I1 clarity is $2,907,809"
diamonds %>%
  ggplot(aes(x = clarity, y = price)) +
  geom_bar(stat = "identity")

# to calculate the average price of diamonds per each clarity:
diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price)) %>% # defined m as the mean price of the diamonds dataset
  ggplot(aes(x = clarity, y = m)) +
  geom_bar(stat = "identity") +
  labs(y = "Mean Price of Diamonds",
       x = "Clarity Category")
```

4. Adding Error bars and facet wraps

   - Notice how the error bars (standard deviation) overlay the bars themselves. Recall back to the line graph chapter that the order of graphing elements matters!
   - Try repositioning `geom_errobar()` above `geom_bar()` to see what happens!

```
diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price),
            s = sd(price)) %>% # standard deviation
  ggplot(aes(x = clarity, y = m, group = cut, color = cut, fill = cut)) + # what happens when you add b
  geom_bar(stat = "identity")+
  geom_errorbar(aes(ymin = m-s, ymax = m+s)) +
  facet_wrap(~cut)
```

5. Using the code from the previous example, rename the x and y-axes using `lab()`

6. Execute `?geom_bar()` to check out some of the arguments that can be used for this `geom` element.

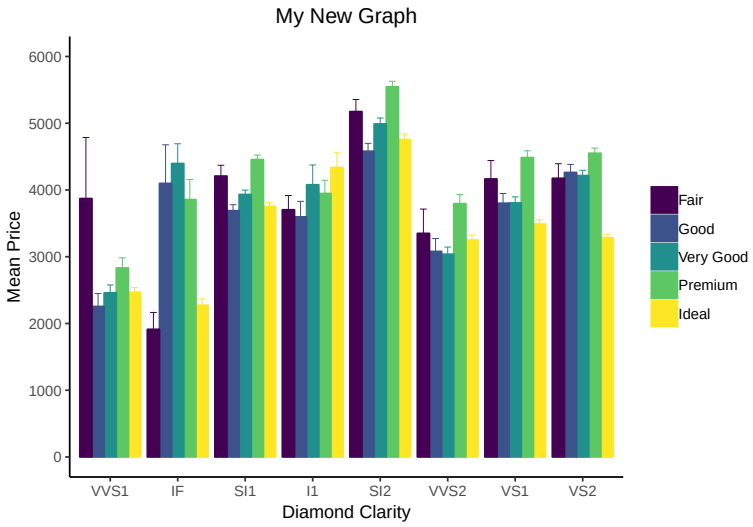


Figure 11.1: Product of the changes listed above.

```
diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price)) %>% # defined m as the mean price of the diamonds dataset
  ggplot(aes(x = clarity, y = m)) +
  geom_bar(stat = "identity",
           show.legend = FALSE, # altering the show.legend argument
           color = "red",
           size = 1, # when we change the size, what are we really changing? (size of what?)
           alpha = .5) # what does changing the value of alpha do?
```

7. Using techniques we've learned from the line graph chapter, recreate the graph below:

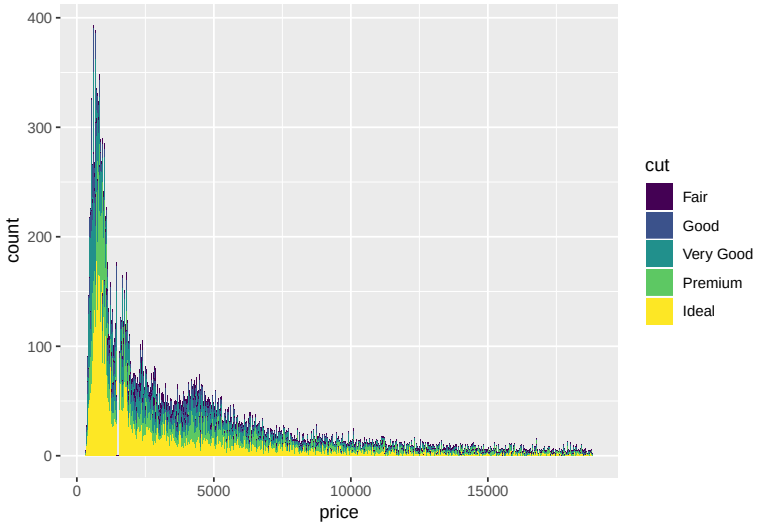
    - Use `group_by()` for `clarity` and `cut`
    - Re-order the x-axis values of `clarity` using `mutate()` and `factor()`
    - Calculate the mean and standard error (using the standard error function) for `price` within `summarize`
    - Set the independent variable to clarity and dependent variable to mean price
    - Set the `group`, `color`, and `fill` to equal `cut`
    - Change the `size` of the error bars to 0.2 and the `width` to 0.5 inside `geom_errorbar()`
    - Change the y-axis tick intervals using `limits()` and `breaks()` within `scale_y_continuous` to match the graph below (hint: values for those functions should include c(-numbers in here-))
    - Change the x, y, and title labels using `labs()`
    - Set the theme to `theme_classic()`
    - Remove the legend title from the graph using `theme()` and `element_blank()` (note that this line must go *after* `theme_classic()`)
    - Center the plot title using `theme()`, `element_text`, and `hjust`
    - Set the `position_dodge()` to 0.9 for both `geom_errorbar()` **and** `geom_bar()`

11.2 Histograms

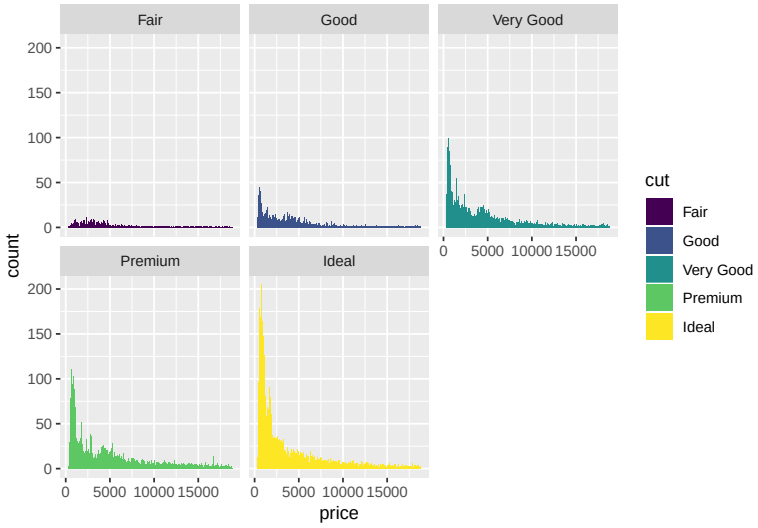
Similar to bar graphs, a histogram uses rectangular blocks to display data. The difference between a bar graph and a histogram is the x-axis variable. For bar graphs, the x-axis variable is *categorical*. For histograms, the x-axis is *continuous* (i.e., numeric). In both scenarios, the y-axis is a numeric dependent variable.

In the `diamonds` dataset, we could look at the price distribution for all diamonds. By default, the y-axis value for a histogram is the count (i.e., the stat argument's default value is `stat = "bin"`). Within the `geom_histogram()` function, we can change how detailed we want the bars to look by altering the bin width:

```
diamonds %>%
  ggplot(aes(x = price, group = cut, fill = cut)) +
  geom_histogram(binwidth = 10) # small binwidth
```




```
# facet wrapping by cut
diamonds %>%
  ggplot(aes(x = price, group = cut, fill = cut)) +
  geom_histogram(binwidth = 10) +
  facet_wrap(~cut)
```



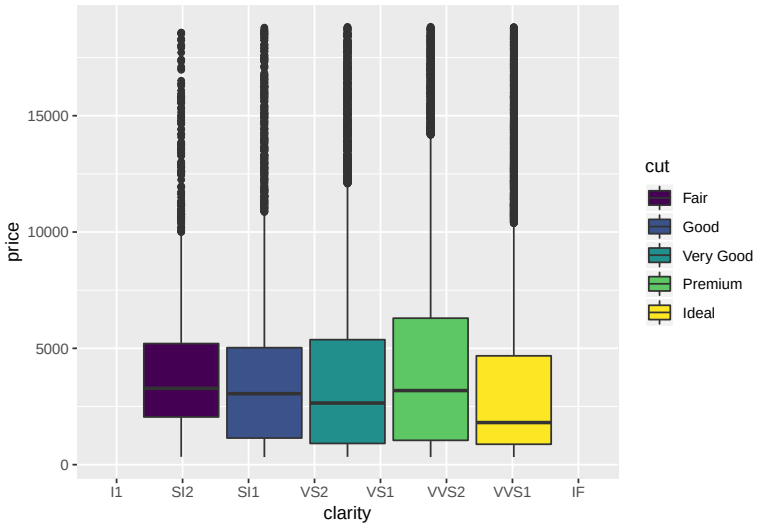
11.2.1 Exercises

1. Execute the same code above, but change the bin width to 100
2. Change the bin width to 500
3. Change the bin width to 1000

11.3 Box Plot

To graph a basic box plot, we use `geom_boxplot()`:

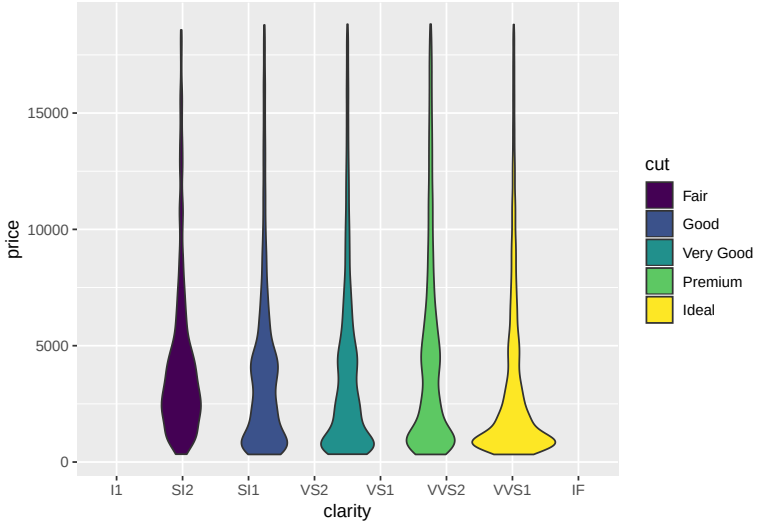
```
diamonds %>%
  group_by(clarity, cut) %>%
  ggplot(aes(x = clarity, y = price, group = cut, fill = cut)) +
  geom_boxplot()
```



11.4 Violin Plot

A violin plot graphs data similarly to a box plot. However, violin plots have the added bonus of visualizing the distribution of the data/observations. We can create a violin plot similar to a box plot by replacing `geom_boxplot` with `geom_violin`:

```
diamonds %>%
  group_by(clarity, cut) %>%
  ggplot(aes(x = clarity, y = price, group = cut, fill = cut)) +
  geom_violin()
```



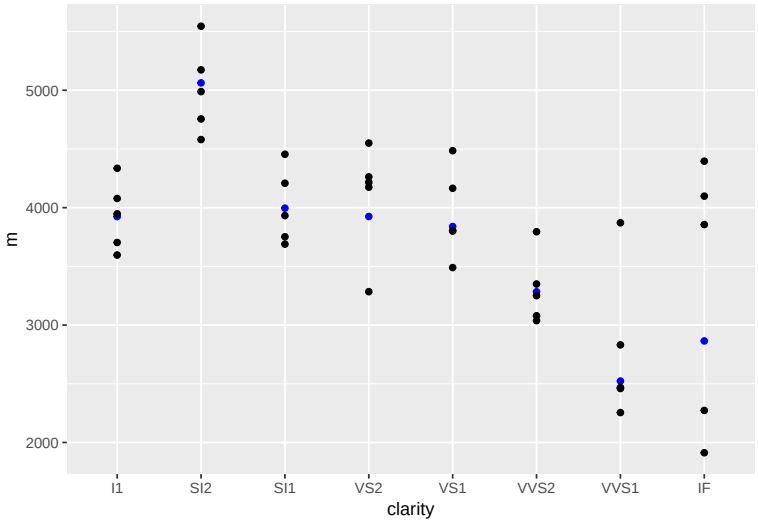
11.5 Graphing with Different Datasets

One final note is that `geom` elements (`geom_point()`, `geom_line()`, etc.) can plot data from two (or more) different datasets. Let's see an example:

```
## creating dataset #1
data1 <-
  diamonds %>%
  group_by(clarity) %>%
  summarize(m = mean(price))

## creating dataset #2
data2 <-
  diamonds %>%
  group_by(clarity, cut) %>%
  summarize(m = mean(price))

## graphing data points from 2 different datasets on one graph
ggplot() +
  geom_point(data = data1, aes(x = clarity, y = m), color = "blue") + # must include argument label "da
  geom_point(data = data2, aes(x = clarity, y = m))
```



In the above example, the data from the dataset called `data1` is colored in blue for distinction. This data's values calculate the mean (average) price of diamonds for each clarity (simply execute `data1` or `View(data1)` to view the data). The data from the dataset called `data2` is colored in black. This dataset's values are derived from the mean (average) price of diamonds for each clarity **and** cut category. Again, the x and y values must be the same (`clarity` and `m`).

Within each `geom` element, you specify the name of the dataset with the argument label `data =`. This is because the first argument for many of the `geom` functions is the aesthetic `mapping` by default. Note that you can plot with multiple datasets for any other `geom` element too. You could have a `geom_bar()` for `data1` and a `geom_point()` for `data2` if you wanted to! If for some reason you wanted to plot error bars from `data1` and data points from `data2`, you could do that also. This would likely be a terrible graph, but you could.

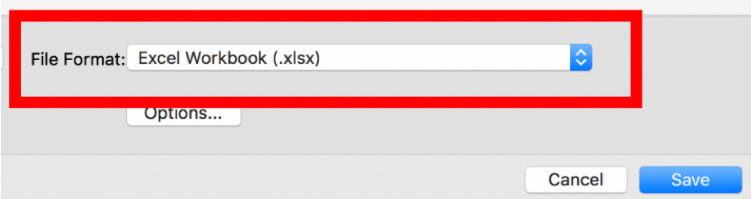
Chapter 12

Graphing Your Own Dataset

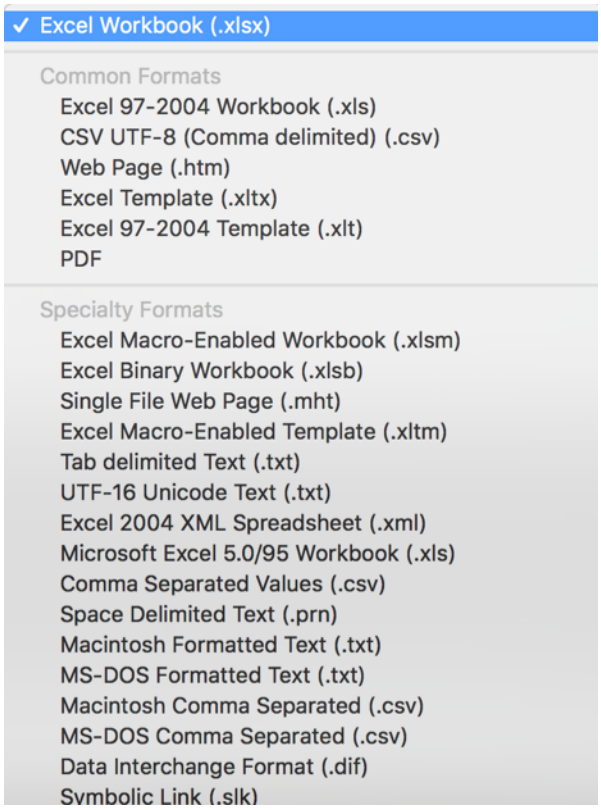
Before we start, make sure the proper packages have been loaded to RStudio. Another important consideration is that in order to graph, your dataset must be is ready to go. That means **no missing data**, all variable **names are appropriate** (no spaces, no symbols, cannot begin with a number) and that you **have all of the variables required** for your graph. If your data meets these criteria, you are ready to graph.

12.1 Import Your Dataset

To import your dataset, you must make sure that the data file is within the same folder as your R project (explained in earlier sections). Next, you need to determine the file format. Is your file an `.xlsx` (standard Excel file) or .csv (Comma Separated Values) file? The file format is determined by how you save the spreadsheet when you click **Save As**.



In the image below, you'll find that there are a few different ways to save an Excel file.



12.1.1 Importing `.csv`

If your dataset is saved as a `.csv`, make sure it is the plain `.csv` format (i.e., "Comma Separated Values"; no other text in the label)

Here is the code format required to import your dataset:

```
library(tidyverse) # required package
data <- read_csv("Title of Excel File.csv")
```

12.1.2 Importing `.xlsx`

```
library(readxl) # required package
data <- read_xlsx("Title of Excel File.xlsx")
```

12.2 Calculating Values

If you plan on graphing data using mean values (i.e., each data point represents an averaged value, not an individual's value), you will have to calculate it before you start graphing with `ggplot()`. If you are able to use previous examples to do this, that's great! However, it is more likely that you will need additional restructuring.

12.3 Restructuring your Data

Up until now, we have used datasets that have already been restructured/pre-prepared. Refer back to 3.3.10 for details on data structures. It is important that your values are structured properly in order to graph; you may be able to get away with it if R happens to recognize your variable structures by default, but this is a dangerous path.

Working with your own data typically requires a deeper understanding of how the data wrangling code works (i.e., `group_by()`, `summarize()`, `ungroup()`). Until now, we have taken these functions for granted. Why didn't I teach data management before graphing? Customer demand. Everyone wants to learn how to graph first, even when managing your data first is arguably more important.

Not to worry! In the next part, we will cover how these **tidyverse** functions work now that you have a taste of what R can do. Make sure to follow along the exercises and examples! We will come back to graphing "unprepared" data once we've mastered data management!

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2019). *rmarkdown: Dynamic Documents for R*. R package version 1.12.

Arnold, J. B. (2019). *ggthemes: Extra Themes, Scales and Geoms for 'ggplot2'*. R package version 4.2.0.

Bates, D. and Maechler, M. (2019). *Matrix: Sparse and Dense Matrix Classes and Methods*. R package version 1.2-17.

Bates, D., Maechler, M., Bolker, B., and Walker, S. (2019). *lme4: Linear Mixed-Effects Models using 'Eigen' and S4*. R package version 1.1-21.

Henry, L. and Wickham, H. (2019). *purrr: Functional Programming Tools*. R package version 0.3.2.

Lenth, R. (2019). *emmeans: Estimated Marginal Means, aka Least-Squares Means*. R package version 1.3.4.

Müller, K. and Wickham, H. (2019). *tibble: Simple Data Frames*. R package version 2.1.1.

Ooms, J. (2018). *writexl: Export Data Frames to Excel 'xlsx' Format*. R package version 1.1.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Singmann, H., Bolker, B., Westfall, J., and Aust, F. (2019). *afex: Analysis of Factorial Experiments*. R package version 0.23-0.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.

Wickham, H. (2019a). *forcats: Tools for Working with Categorical Variables (Factors)*. R package version 0.4.0.

Wickham, H. (2019b). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.0.

Wickham, H. and Bryan, J. (2019). *readxl: Read Excel Files*. R package version 1.3.1.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., and Woo, K. (2019a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.1.1.

Wickham, H., François, R., Henry, L., and Müller, K. (2019b). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.1.

Wickham, H. and Henry, L. (2019). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.8.3.

Wickham, H., Hester, J., and Francois, R. (2018). *readr: Read Rectangular Text Data*. R package version 1.3.1.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2019a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.10.

Xie, Y. (2019b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.23.

Zhu, H. (2019). *kableExtra: Construct Complex Table with 'kable' and Pipe Syntax*. R package version 1.1.0.