Using R in Intro Stats

A Guide for the Uninitiated

Travis Loux

March 19, 2025

Contents

Chapter 1

Goals

This guide is written for students learning R in their introductory statistics class. It isn't meant to teach you statistics or to teach you R programming (though there is some of that). It is meant to help you learn the things you'll need to know in order to do that stuff.

Historically, intro stats courses were taken my math and stats majors as a gateway to more advanced statistics classes. Like most stats classes, they were taught like traditional math classes, using pen and paper. Over the last decade or more, we have seen two big shifts in intro stats.

First, more schools and programs are requiring an intro stats class or something similar as a gen ed or major requirement. This means that most students in intro stats don't consider themselves "math people" or "statistics people." It's highly likely this will be the only statistics class they take.

Second, the availability of personal computing and statistical software has grown dramatically. We are now able to process in a matter of seconds, on our laptops or through a cloud server, complex computations that would have required past students to spend countless overnight hours in a highly regulated university computer lab. This development has provided an opportunity for into stats classes to go deeper into content, and in more varied directions, than they ever have before.

While these changes have been great - more people learning more statistics can't be a bad thing, right? - it leaves us in a situation where students (probably already dreading their stats class) are learning a statistical programming language at the same time they are learning statistical calculations and concepts, which is often thought of as its own language itself. This creates a need for a gentle introduction to statistical computing for students with very little background in either statistics or computing. So far I have found very little comprehensive support for such students. I hope to provide that support here.

To Instructors

While this guide is written for students, I hope you are able to benefit as well. I put this guide together after years of working with students who are now its primary audience. By reading ahead, you may infer some of the difficulties your students will come up against in their work, some of which may surprise you if it is your first time teaching such a course. Hopefully you find it useful enough that you can refer your students to it as well, and their reading it can build a common starting point for your class and ease your workload of additional questions.

Some other views on teaching introductory R can be found below:

- 6 Lessons I learned from teaching R to non-programmers by Albert Rapp [LINK]
- Teach the tidyverse to beginners by David Robinson [LINK]. Also see his talk on the topic [LINK]
- Teaching R in a Kinder, Gentler, More Effective Manner: Use Base-R, Not the Tidyverse by Norman Matloff [LINK]

Chapter 2

Installation and Setup

If you are using R for into stats, I am going to assume you are accessing it through R Studio, which has become the de facto interface for new R users. If you aren't using R Studio or Posit Cloud, you can probably skip this chapter. The rest of the book should still be helpful, though.

There are two ways you can prepare R and R Studio for class. Both options are free to set up and use.

The first option is to download both programs onto your personal computer. The download and installation steps are pretty straightforward (see below) but you will need a good understanding of your computer and it's file system (the topic of Computer File Systems and Working Directories) in order to use this approach. It's mostly recommended for students who think they might be doing more analysis beyond this one course.

The second option is to sign up for a Posit Cloud account (Posit is the parent company of R Studio). Posit Cloud is relatively new but is quite stable and has become my preferred option for students in intro stats. The primary benefit is a much simpler file system. Plus, you can upload and download files from the server pretty easily. Because you are accessing a server, you will need internet access. Posit Cloud accounts currently have a monthly time limit, but as long as you remember to close your browser tab when you are not working on your assignments, you are unlikely to come close to the time limit.

Installation Option 1: Downloading to your Personal Computer

Best for students who: (1) may need R in other courses; (2) are comfortable navigating the file and directory system on their computer.

The first thing to understand is that R and R Studio are two separate programs. R is a statistical software package maintained by the R Core Team. R Studio is an *integrated development environment*, or IDE, for the R ecosystem. You will use R Studio for all of your analyses, but R Studio accesses R to run the statistical computations. Strictly speaking R Studio is unnecessary, but it gives a cleaner, more user-friendly interface with some extra capabilities. It also looks similar across operating systems.

Part 1: Install R

1. Go to www.r-project.org
2. Select 'CRAN' from the menu on the left
3. Select any mirror
4. Select the link for your operating system
5. Download the R installer
    a. Windows:
        1. Select 'base'
        2. Click on 'Download R 4.3.1 for Windows' (or newest version)
    b. Mac:
        1. Download 'R-4.3.1.pkg' (or newest version)
6. If your computer does not do so automatically, open your downloaded file and install R. Follow the default settings.

You now have a functioning version of R on your computer. If you wanted to continue without R Studio you could, but R Studio makes many processes much easier.

Part 2: Install R Studio

1. Go to posit.co
2. From 'Products', select 'RStudio IDE'
3. Scroll about half the way down the page and click on the 'Download RStudio Desktop' button under the Open Source Edition column
4. Click 'Download RStudio' button.
5. Click on 'Download RStudio Desktop for____' or scroll to the 'All Installers' section and find the installer for your operating system/platform (stay away from Zip/tar archives - there is no help that way)
6. If your computer does not do so automatically, open your downloaded file and install RStudio. Follow the default settings.

Installation Option 2: Using the Cloud

Best for students who are: (1) not likely to need R outside of this class; (2) uncomfortable navigating the file and directory system on their computer; (3) comfortable using a web-based app; (4) using a Chromebook or similar machine.

1. Go to posit.co
2. From 'Products', select 'Posit Cloud'
3. Select 'Get started'
4. Follow directions to sign up for Cloud Free

R Studio GUI

Whether you downloaded R and R Studio or set up an R Studio Cloud account, the graphical user interface, or GUI (pronounced "gooey"), looks similar. When you first open R Studio, you will see a window with three "panes" as in Figure 2.1 (In R Studio Cloud, this will all be inside your browser).

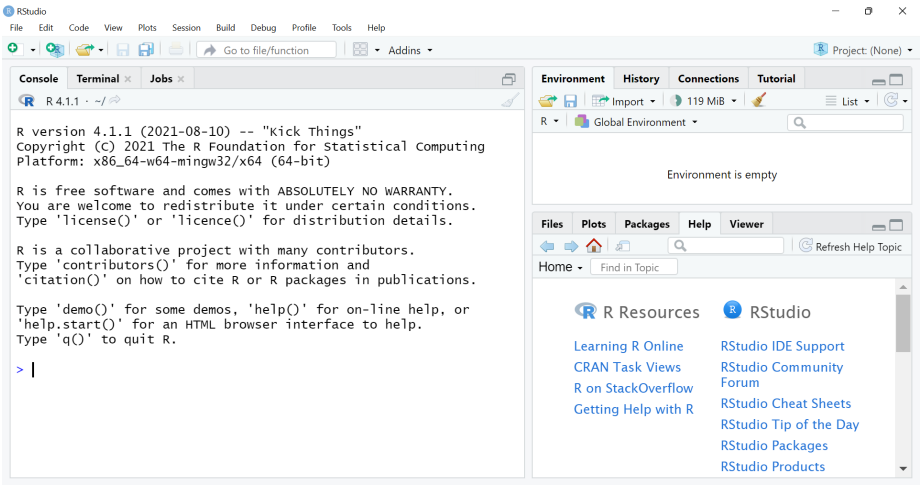


Figure 2.1: On start up, the R Studio GUI will look something like this.

Console

The left side of the window will have one large pane with tabs for Console, Terminal, and Jobs. You will likely only need the Console tab. This is where calculations are actually done. You can start entering code or math calculations at the right arrow `>`, called the prompt. You can enter one line of code or computations at a time and hit Enter to run the code.

> IMPORTANT: If you ever see a new line without a `>` prompt, this means the previous line isn't complete. You might see a `+` sign, indicating the code is not complete, possibly from a quotation or parenthesis you started but didn't end (more on this in R Language

and Syntax). If you are stuck in this situation, you can hit the Escape button to back out of that code without running it. You may also see no symbol, which means R is still running the previous line of code.

You can try to enter some simple math calculations and hit enter. Notice that R follows the order of operations, so `2 + 3 * 4` is calculated as 2 + 12, resulting in 14. You can use the asterisk for multiplication and the carrot `^` for exponents. For example, `3^2` will compute to 9.

```
2 + 3 * 4
```

```
## [1] 14
```

```
3^2
```

```
## [1] 9
```

Environment and History

The top right pane will have tabs for Environment, History, and Connections. The Environment tab will list all of the R objects available in your current working session. This will include everything from datasets to saved hypothesis tests and regression models. This will be a useful place to go if you can't remember the name of something you've saved or if you're having trouble remembering the spelling.

The History tab keeps a record of all of the code you have run in your current session. If you ran some math calculations in the section above, you should be able to go to the History tab and see them recorded there. You will only see the code, not the result.

You likely won't use Connections much so we won't hurry about it here.

Files, Plots, Packages, Help

The bottom right pane has additional tabs. The Files tab shows files on your computer. It begins by looking your your **working directory** (see Computer File Systems and Working Directories for more details), but you can navigate to any folder on your computer.

Plots is where any graphs you make will display and be recorded. When you have multiple plots, you can use the left and right arrows to move through them. The Export button will let you copy or save your plots.

Packages shows all of the R packages currently installed on your computer. Packages with checked boxes are currently loaded into your session. Unchecked boxes mean the package is installed on your computer but not loaded into your session. We will talk more about packages in Packages.

The Help tab is where you can look up information about functions. There is a search option in the tab, or if you know the name of the function you want to read about, you can type and run `?functionname` in the Console. For example, running `?mean` will bring up the help file for the **mean** function, which will tell you the possible inputs to the function (Arguments) and what the function is calculating (Value).

We will discuss the Viewer tab in R Markdown.

R Script Files

If you go to File -> New -> R Script, the left side of your R Studio window will separate into two panes, the pane with the Console, Terminal, and Jobs going to the bottom. On the top you will have a pane with a tab that says something like "Untitled1" and a blank white box. This is the area where your R script, or code files, will display. You can type code here and send them to the Console by clicking the Run button or typing Control+Enter.

Scripts are the way to save your code so you can go back, re-run lines, and edit old code. They are analogous to Word files and the way statisticians and data scientists do and save their work. You will want to keep your code saved in script files. As the presence of the tab suggests, you can have multiple R script files open at a time, each with its own tab.

> IMPORTANT: Typing code into the script file doesn't run the code. You have to send the code to the console for the computations to be done. Many students will forget this step and wonder why code they can't find results in their Environment that they coded into their script. The most likely reason is that the code wasn't run in the current R session.

Chapter 3

Packages

As you likely know by now, R is an open source programming language. The most obvious benefit for you as a student is that that means R is free. That is a huge benefit to professional statisticians as well. In addition, R's open source design means that anyone can contribute to the R project and develop R add-ons and extensions called **packages**.

These packages are also free and usually pretty easy to access. CRAN, the Comprehensive R Archive Network, is maintained by the R Core Team and houses thousands of contributed packages. Other places you might find packages include sites like Bioconductor or GitHub. Packages on CRAN go through many more checks than packages on other sites and are generally considered more stable.

Though most of the methods you will learn in an introductory statistics class are available in R out-of-the-box, you may need to use extension packages from time to time. Packages I have my students use include:

- **readxl** for importing data from Microsoft Excel files
- **rmarkdown** for report writing
- **ggplot2** for graphics
- **epitools** for functions to compute risk ratios and odds ratios

There are also packages that include datasets you may need to use.

Installing and Loading Packages

Getting packages ready to use is a two step process. The steps are not difficult but it is important to remember when you do and do not need to use them. This process is summarized in Figure 3.1.

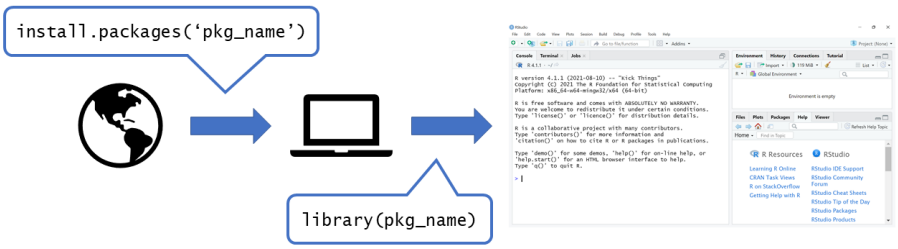


Figure 3.1: The process for accessing a package includes downloading it from the internet and loading it into your R session.

First, you will need to download the package from the internet. In R terminology, this is called "installing" the package. If the package you want is on CRAN, you can use the function **install.packages** for this. For example, to install **readxl**, run

```
install.packages('readxl')
```

Notice that the function name is in quotation marks[1].

Installing a package is something you only have to do once on your computer. Once the package is downloaded, you have it. If you use another computer or need to completely uninstall R for some reason, you will need to install the package again. Otherwise, this is a one-time process[2].

Once you have installed the package it is on your computer, but R does not automatically load all packages into your R session. This is primarily for time and memory management, but there are other reasons as well, such as multiple packages having functions with the same name. In order to load the package into your R session, use the **library** function:

```
library(readxl)
```

Notice here the function name is not in quotation marks. Now all the functionality and data in the package is available to you. If you want to know what functions or dataset a package includes, you can use the **help** function. For example, to see what is in **readxl**, run

```
help(package='readxl')
```

[1] You can use single quotations, as done here, or double quotations. You just need to use the same to start and end the quotation.

[2] You can also update packages this way, but for the purposes of a single course this is not very relevant.

Unlike installing, loading packages needs to be done every time you start a new R session. With every session, you start with a minimal number of packages. Any packages you need that you had to download from the internet (and some others that come pre-installed) will need to be loaded into your new session.

As with most topics in this book, there is plenty more to say about using, managing, and creating packages that is beyond the scope of this guide. More resources are available in Additional R Resources.

Chapter 4

Computer File Systems and Working Directories

In my experience, the thing students learning R have the hardest time with actually doesn't have much to do with R. The biggest issue is often locating and managing files on their computer. When using R, you often have to access data from files on your computer and save results to other files on your computer. Knowing where these files are and how to get to them is important if you are ever going to get off the ground. This topic isn't quite as important for students using R Studio Cloud since the file system there is much simpler, but it will still help answer many questions.

Computer Files and Directories

Every file on your computer has a name, an extension, and a location. The file name is straightforward - this is the name you give to the file when you save it.

File extensions are the letters that come after the file name and are separated from the file name by a period. They tell the type of file the file is. For example, files ending in .docx (or .doc for older files) are Microsoft Word files. Similarly, .xlsx (or .xls) are Microsoft Excel files. Many internet websites are saved as .html (hypertext markup language). It is common to have datasets in comma separated value files, which have the extension .csv. Though .csv files usually open in Excel, they are not Excel files and do not support much of the formatting that is available in Excel, such as bold or color text, cell borders, or cell fill colors.

R script (code) files have a .R extension and R Markdown files have a .Rmd extension. Most R data files have a .RData or .rda extension. Knowing the extensions tells you which program the file will likely be opened in.

The location of a file is where in the computer's file system the file can be found. We often think of files as residing in folders because of the common folder icon. The formal name for a file folder on a computer is **directory**. As an example, all the files for this guide are in a directory on my computer named uris. This in turn is in a directory projects which is in my Documents directory, and so on. The full location of these files is `C:\Users\travi\OneDrive\Documents\projects\uris\`, where each back-slash (\) is separating directory names, each directory containing the next in the list. The back-slash convention is unique to Windows operating system. R and most other systems use forward-slashes, so for R to interpret the location above, I would need to replace the back-slashes with forward-slashes: `C:/Users/travi/OneDrive/Documents/projects/uris/`.

R Working Directories

Any time you begin R, it is looking in a certain folder for files. This folder is called the **working directory** and can be changed as needed. You can get the current working directory by running the code

```
getwd()
```

(standing for get working directory) in the console.

To change the working directory, you can either type out the new directory using the **setwd** function such as

```
setwd("path/to/new/working/directory")
```

or you can use the R Studio GUI menus and go to Session -> Set Working Directory and choose from the options there. Choose Directory might be the most straightforward option to use for now, even if it isn't the most efficient.

While it's possible to change your working directory any time you need to get to a file that isn't in the current working directory, that is time consuming and might be difficult to repeat at another time. Instead, you can ask R to open files from other directories without changing the working directory.

Absolute and Relative Paths

Because we can have multiple files on a computer with the same name, if we want to access any file on our computer we need to tell R (or any other program) the location of the file as well as the name and extension. There are two ways to do this.

First, let's suppose my computer has a file system that starts in the C drive and looks like the location I gave above for the files for this guide. Within the C drive, there is my user directory, **travi**, a directory **Documents**, and a directory **projects**. Now let's say within **projects**, I have a directory for every analysis I am doing. For example, maybe a directory named **NHIS-PHQ** for one project and **YRBS-PA** for another. Within each of the project folders, I have a **data** folder where I store the original data and any codebooks and a **code** folder where I put my R code. The structure would start with `C:/travi/Documents/` then look something like:

- projects
  - NHIS-PHQ
    - code
      - analysis.R
    - data
      - nhis2019.csv
  - YRBS-PA
    - code
      - analysis.R
    - data
      - yrbs2019.csv

The **absolute path** to a file is the full location of the file on your computer. You can think of the absolute path as a latitude and longitude of a place on earth. The latitude and longitude identify a specific place no matter where you are on earth at the time. Similarly, a absolute path tells you where a file exists on your computer and allows you to access the file from anywhere. In Windows, the absolute path will likely start with the C drive, as it does above in the location of the files for this guide. If I was working on the NHIS project and wanted to refer to the data using an absolute path, it would look like `C:/travi/Documents/projects/nhis-phq/data/nhis2019.csv`.

Absolute paths are pretty straightforward, but they can take a while to write out. Additionally, if you move the **nhis-phq** folder somewhere else, like an archives folder, the absolute path won't work anymore - all the files have a new location and you will need to update your code to this location.

The **relative path** to a file is a set of directions to get from the directory you are in (R's working directory) to a specific file. It follows the same approach as the absolute path, with directories separates by forward slashes, but it starts in the working directory rather than the drive or highest-level directory. Think of the relative path like you would think of driving directions: they depend on where you want to go, but also where you are starting. One additional point in relative paths is going up a level. In absolute paths, you are digging one level

deeper at each step. In a relative path, you may need to back up a level or two before you can get to your file. To back up a level, use `../` and if you need to back up more levels, add additional `../`.

For example, say you are working on the **nhis-phq** project and your working directory is the **code** directory. In order to access the dataset **nhis2019.csv**, you need to go up to the **nhis-phq** directory, then down into the **data** directory to get to the data. The relative path from the **code** directory to the **nhis2019.csv** file is `../data/nhis2019.csv`.

If you wanted to get from the **nhis-phq/code** directory to **yrbs2019.csv**, the relative path would be `../../yrbs-pa/data/yrbs2019.csv`. The sequence `../../` moves you from the **code** directory up to **nhis-phq** then up to **projects**. From there, you go down into **yrbs-pa** and so on.

Relative paths are a bit more complicated than absolute paths. Unlike absolute paths, relative paths depend on where you start. It's important to remember this is your working directory, which isn't necessarily where your code file is located[1].

Benefits of relative paths include they are usually shorter to type and if you move an entire project's directory (such as to an archives folder like in the previous example) the relative paths will still work. Relative paths are also better if you are working on a cloud system that syncs to multiple computers. Since the different computers may have different file structures, the absolute paths might differ from one computer to the next. However, the relative paths should be the same as long as they never leave the synced directory.

> IMPORTANT: Absolute paths are like latitude and longitude of point A. Relative paths are like driving directions from point A to point B.

Overall, the use of absolute and relative paths might come down to preference (with the exception of cloud-synced directories). Absolute paths are easier but more tedious to type out, whereas relative paths are quicker but more likely to run into issues from starting in different working directories.

[1] If you start R by opening a code file, your working directory is the location of your file. This is how I prefer to work so I always set my relative paths assuming the working directory and code file location are the same. It's not perfect, but it works 95% of the time and when it doesn't it's an easy fix.

Chapter 5

Loading and Saving Data

Most of what you need to know about loading and saving data is covered in Computer File Systems and Working Directories. When you want to access a dataset or save information to your computer, you need to go outside of R to get or save a file to a certain location. The ability to use absolute and/or relative paths is paramount.

Beyond locating a file, how to work with data files depends on file types. While there are built-in functions for loading and saving common types of data, there are also extension packages for other file types. Some of the more popular ones are **readxl** for working with Microsoft Excel files and **haven** for working with data from other statistical packages such as SAS, Stata, and SPSS. Functions in these packages work similarly to **read.csv** and **write.csv** discussed below, with special options particular to the various data file types.

Loading and Saving RData Files

Files with extension .RData hold data and other information already formatted for R, so they are the easiest to open and save, with self-explanatory functions **load** and **save**.

Suppose I am working in a directory that includes a sub-directory called **data** which hold all of my data, including a file **dataset1.RData**. To load the RData file into my R session, I simply type

```
load('data/dataset1.RData')
```

See how I am using the relative path from my working directory to the file and I am enclosing that path in quotation marks. The data I load into my R session

may not be named **dataset1** - this is the name of the file containing the dataset, not the dataset itself. An RData file can hold multiple datasets in it, so the file name and dataset name do not need to match.

If I have results from an analysis that I want to save into an RData file, I use the function **save**. Say I have a dataset named **mydata** which I have updated to a new version, **mydata2** and I want to save both datasets into an RData file. I would type

```
save(file='data/clean-data.RData', mydata, mydata2)
```

This code tells R to save **mydata** and **mydata2** in a file called **clean-data.RData** that sits inside the **data** folder. Now when I re-open R and load **clean-data.RData**, both **mydata** and **mydata2** will be available.

Loading and Saving csv Files

CSV, or **c**omma **s**eparated **v**alue, files are one of the most common and universally accepted ways to store and share data. This means there are built-in functions in R to work with them. The main ones are **read.csv** to read data into R and **write.csv** to save an R dataset to a csv file.

Let's say you have the same setup as above with another file, **dataset3.csv** in the **data** folder. To read in the data, use

```
mydata3 = read.csv('data/dataset3.csv', header=TRUE)
```

This will read the file **dataset3.csv** and assign it to an R object named **mydata3**. Since the dataset isn't already an R dataset, we need to use `mydata3 =` to assign an R name to the dataset. In addition, the `header=TRUE` option reads in the first row of the file as the column names rather than a row of data. If the first row of the file was data, then use `header=FALSE`. There are other options in the **read.csv** function, but this should cover about 80-90% of cases.

If you've done some work on a dataset and want to save it as a csv, say to share it with others, you use **write.csv**. Unlike the **save** function, **write.csv** can only handle one dataset at a time. To save **mydata3** in a new csv file, use

```
write.csv(file='data/dataset3-updated.csv', x=mydata3, row.names=FALSE)
```

This code will save **mydata3** as a new csv file in the **data** folder named **dataset3-updated.csv**. The `row.names=FALSE` option is almost always what you want. If `row.names=TRUE`, then the first column of the new csv file will be the the row names (or row numbers if the rows aren't explicitly named, which

is most likely). This adds a column to the dataset that didn't exist before and is probably redundant, since R and other software already uses row numbers for any dataset. If the row names of your dataset are important, they should be saved as a column of the dataset instead of row names.

Chapter 6

Data Frames

R holds most datasets like you might see in a spreadsheet as a class of objects called a **data frame**. Others have created packages with variations of data frames, such as data tables and tibbles, which have some advanced features or other benefits, but most of what we discuss for data frames will work for these classes of objects as well.

Structure of a Data Frame

While the internal workings of a data frame are surprisingly complex and allow for some fairly advanced data manipulation, the basics are pretty straightforward.

A data frame is a rectangular structure. This means it has two dimensions: rows and columns. Each row has the same number of columns and each column has the same number of rows. It is best to think of a data frame as a collection of columns. All the data within a column must be the same type of data (for example, all numbers or all text); however, different columns within a data frame can hold different types of data. Every entry in a column is called a cell. An example of a small data frame holding hypothetical student information is given below.

```
##      ID             Name         Major  GPA
## 1 20304      Adam Baines       English 3.30
## 2 20305 Claire Dougherty    Statistics 3.53
## 3 20306      Eric Forest    Psychology 3.87
## 4 30101      George Hull Public Health 3.79
```

The columns ID and GPA hold numeric variables, while Name and Major hold text, or "character" variables.

While every column has the same number of rows, some values might be missing (for example, a student's major may be unknown). Missing values are denoted by `NA`, holding a spot in the data frame and keeping the rectangular structure intact.

Using Data Frames

Data frames are only useful if you can access parts of them at a time to view data, perform analysis, or graph results.

We can access a cell within the data frame by using square brackets and giving, in order, the row and column number we want to view. For example, the data frame above is named **gpa**, so to access a cell particular cell, we would type `gpa[row_number, column_number]`:

```
gpa[2, 3]
```

```
## [1] "Statistics"
```

```
gpa[4, 1]
```

```
## [1] 30101
```

The first line of code gives the value in the second row and third column of the dataset: the second major listed. The second line gives the value in the fourth row and first column, the ID of the fourth student.

If we want an entire row or an entire column, we give the row or column number and leave the other entry blank:

```
gpa[1, ]
```

```
##      ID        Name   Major GPA
## 1 20304 Adam Baines English 3.3
```

```
gpa[, 2]
```

```
## [1] "Adam Baines"      "Claire Dougherty" "Eric Forest"      "George Hull"
```

The first line of code gives the first row of data, all the information for Adam Baines. The second line of code gives the second column, the student names.

While we can access columns by giving the column number, that can make code hard to read. The preferred way to access a column within a data frame is to use the dollar sign, `$`, and then give the column name. The full code would look like `dataframe_name$column_name`. For example:

```
gpa$Name
```

```
## [1] "Adam Baines"      "Claire Dougherty" "Eric Forest"      "George Hull"
```

```
gpa$GPA
```

```
## [1] 3.30 3.53 3.87 3.79
```

The above lines of code are much more clear for a human to read, assuming the column names are meaningful. Once we have a column of data, we can compute statistics, such as the mean and standard deviation

```
mean(gpa$GPA)
```

```
## [1] 3.6225
```

```
sd(gpa$GPA)
```

```
## [1] 0.2594064
```

You can also create new variables in your dataset by modifing or combining existing variables. For example, say you realize the GPAs are calculated incorrectly and everyone should have an additional 0.10 points.

```
gpa$GPAcorrect = gpa$GPA + 0.1
print(gpa)
```

```
##      ID             Name         Major  GPA GPAcorrect
## 1 20304      Adam Baines       English 3.30       3.40
## 2 20305 Claire Dougherty    Statistics 3.53       3.63
## 3 20306      Eric Forest    Psychology 3.87       3.97
## 4 30101      George Hull Public Health 3.79       3.89
```

A new variable, **GPAcorrect** is added to the end of the data frame. When we calculated `gpa$GPA + 0.01`, we saved it to a new variable inside the existing data frame, adding the new variable as a column at the end[1].

[1] If we had saved the new data to an existing variable, it would have saved over that column and removed the original data. That's generally considered a bad idea because you always want to keep the original data available to refer back to.

Some functions that take multiple variables streamline how you can input columns from a dataset, but for a lot of simple tasks, the `dataframe_name$column_name` format will work just fine.

Chapter 7

More Data Structures

In the last chapter we introduced the concept of data frames and of variables types within data frames. Data frames are special cases of **objects** in the R system. You can think of objects as nouns - they are the "things" in your R environment: datasets, variables, regression models, etc.

In R, every object has a **class** that tells you what the object is and tells functions what they can do with the objects. Again back to last chapter, data frames are a class of objects in R. They have rows and columns, columns can be different classes, and all values in a column must be of the same class.

Some of the most common classes you will encounter are described below.

Data

Numbers

When people think of statistics and data, they think of numbers. The two main classes of number-based data in R are **numeric** and **integer** classes. As you might guess from the names, the main difference is that the integer class requires values to be integers while the numeric class allows decimals. The main difference for these classes is under the hood - integer objects take less memory to store than do numeric objects. You can do all the same calculations with one as you would the other.

Categorical Variables

Categorical variables are primarily held in **character** and **factor** classes. Character objects are strings of text, such as names or addresses. With some R wizardry, you can edit the text character-by-character. Factors, on the other hand,

are more traditional categorical data. Usually we think of factors as categorical data with a relatively small number of levels, such as highest educational degree obtained or employment status.

Similar to the distinction between numeric and integer classes, for most introductory purposes character and factor classes can be treated the same. One important difference is that character strings are wrapped in quotation marks. You must do this when you create them and they will have quotation marks when displayed on the screen.

Logical Variables

A rather unique class of variables is **logical**. Logical variables can take one of two values, `TRUE` or `FALSE`, and are used to denote true/false comparisons between two values. For example:

```
result1 = 2 < 4
result1
```

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

```
result2 = 5 < 1
result2
```

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

If you attempt to do math on logicals, `TRUE` will become 1 and `FALSE` will become 0, so adding logicals together will tell you how many `TRUE`s there are.

You can also use **Boolean logic** to create complex logical statements by combining many logicals into a single logical, such as "are all of these true" or "are any of these true." We won't worry about that now, but it is a powerful approach to controlling what calculations can happen in certain instances.

Dates and Times

The final classes of data that we will talk about are pretty unlikely to appear as a primary variable of focus, but still might show up from time to time. These are classes that store date and time information. First, the **Date** class holds dates, which usually display in the form YYYY-MM-DD, so that, for example, 2022-01-31 is January 31, 2022.

If you need times along with the dates, the classes **POSIXct** and **POSIXlt** are available in R. Both of these hold times up to the second (or fraction of

a second) but have slightly different inner-workings. The POSIX classes (and times generally) can be difficult to work woth becuase of differences in time zones and daylight savings. You likely won't have to do intense work with time data, so we won't get into too many details here.

Matrices

At first glance, matrices are a lot like data frames - they are two-dimensional and rectangular. The primary difference is that for an object to be a **matrix** class, all the cells in the matrix must be the same class. Any of the classes above (numeric, character, logical, Date, etc.) will work, but they must be consistent throughout the matrix. As discussed in the last chapter, data frames allow each column to have a different class, so they are a bit more general than objects of class matrix.

Outputs

In addition to the data you start with, the results you generate are also objects. Some results might have simple classes. For example, the result of computing the mean of a set of numbers is a numeric object. In other cases, the results are more complex and have their own class structure.

Some classes you are likely to see in your results are described below.

Hypothesis Tests

Results from hypothesis tests available in base R are stored as objects of class **htest**. These objects have different components depending on the type of test, but all have **statistic**, **parameter** and **p.value** components. The statistic component is the value of the test statistic, the parameter component is the value of the relevant parameter(s) (usually degrees of freedom), and the p.value component is the p-value derived from the test. Similar to how columns can be accessed from a dataset, these components can be accessed using the `$` sign. For example:

```
twoway
```

```
##      [,1] [,2] [,3]
## [1,]   12   34   27
## [2,]   19   40   24
```

```
mytest = chisq.test(twoway)
mytest
```

```
##
##  Pearson's Chi-squared test
##
## data:  twoway
## X-squared = 1.6092, df = 2, p-value = 0.4473
```

```
mytest$statistic
```

```
## X-squared
##  1.609189
```

```
mytest$parameter
```

```
## df
##  2
```

```
mytest$p.value
```

```
## [1] 0.4472693
```

Regression Models

Regression model classes depend on the type of model being run. A linear model (function **lm**) results in an object of class **lm**. If you use the **glm** function to run a logistic regression or other generalized linear models, the result will have class **glm**[1]. Similar to the htest class, we can access certain components of an lm or glm object using the `$` symbol, or we can use functions that know how to handle these classes of objects. As some examples:

- `model_name$coef` and `coef(model_name)` will give the estimated regression coefficients.
- `confint(model_name, level=0.95)` will give 95% confidence intervals for each coefficient.
- `model_name$fitted` and `fitted(model_name)` will give the fitted values ($\hat{y}$ from a linear regression).
- `model_name$residuals` and `residuals(model_name)` will give the model residuals ($y - \hat{y}$ from a linear model).

[1] Not all regression classes match the function name, but many do.

- `summary(model_name)` will produce a few summary statistics, including a table of regression coefficients with p-values and fit statistics particular to the model type (for example, R^2 and residual standard error in linear regression).

Plots

If you use **ggplot2** to create graphics, they can also be saved as objects. The saved graphics can then be recalled, displayed, and edited as needed.

The base R graphics plots can also be saved, but in most cases you can only save the underlying structure of the plot, not any annotations, color changes, etc., that you've made.

Chapter 8

R Language and Syntax

So we're in Chapter 8 and finally starting to talk about how to write R code. Just like any natural human language, computer coding languages have a syntax to them, an expectation about how parts of the language go together to form a coherent whole.

Here we will discuss some of the basic R syntax rules that often get overlooked in a course that isn't dedicated to R programming, but that are necessary to write code and understand what it is doing. Some of this you will have seen already in earlier chapters, but now we will talk about what those small bits of code are doing.

Assignment to Variables

One of the most important abilities of the R language is the ability to "assign" results to variables. Whether it is a simple calculation or a complex statistical model, you may want to save particular values and be able to recall them later, either as inputs to new analyses or just to view their results. Assignment usually happens through the equal sign, `=`, or a left arrow `<-`[1]. Once you assign a value or object to a variable, that variable becomes an object in your environment. You can see it in the top right pane of your R Studio window and you can recall that value for the rest of your session. For example:

```
a = 4
b = 3^2
a
```

[1] You can also use a right arrow `->` to assign the other direction, like `3 -> w`, but people who do that are basically anarchists.

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

```
b
```

```
## [1] 9
```

```
a * b
```

```
## [1] 36
```

```
# using the GPA dataset from chapter 6:
gpa$GPA
```

```
## [1] 3.30 3.53 3.87 3.79
```

```
gpa$GPA / a
```

```
## [1] 0.8250 0.8825 0.9675 0.9475
```

```
avg_gpa = mean(gpa$GPA)
avg_gpa
```

```
## [1] 3.6225
```

One thing you will notice is that when we assign a value or result to a variable, we don't actually get to see the value without calling that variable again (in particular, see the `avg_gpa` example). One way to assign a result and view it at the same time is to wrap your assignment in parentheses:

```
(sd_gpa = sd(gpa$GPA))
```

```
## [1] 0.2594064
```

The parentheses tell R to save the result to `sd_gpa`, but also print the result to the screen. This can save a lot of re-typing.

Case Sensitive

R is what is called "case sensitive." That means that capital and lower case letters are treated as different names. So `apple`, `Apple`, and `APPLE` can be three different objects that coexist. It is important to keep R's case sensitive syntax in mind because if you call an object or function using the wrong case, you will receive and error message that it cannot be found or doesn't exist. Also, just because you *can* use capital and lower case letters to distinguish objects doesn't mean you *should* do that. Having to remember the difference between `apple`, `Apple`, and `APPLE` is going to make your coding more complicated than it already is. Usually we try to stick to lower case letters as much as possible. You can get some ideas for formatting variable names in the next section.

Naming Rules and Conventions

When you assign values to variables, choosing a good name is important. In the example above, single-letter names like `a` and `b` are okay for quick calculations, but they are really meaningless. Names like `avg_gpa` and `sd_gpa` are better since they tell you what the value is. When you are choosing a name, there are a few rules to keep in mind and some other conventions, or rules of thumb, that help keep things straight.

Rule: Names of objects can contain letters, numbers, and underscores (`_`) and must start with letter. So, some valid names might be `mydata`, `mydata1`, `mydata2`, or `my_data`. Some examples of invalid names would be `my data`, `my-data`, `2times`, or `x*2`.

When you are working with data that has a meaningful background (which is most data that isn't simulated for an example), your names should be as descriptive as possible without being burdensome in length. In many cases, you will want to combine multiple words into a short descriptive phrase. As seen above, spaces aren't allowed in names, nor are dashes, which are treated as negative or minus signs. However, there are three common conventions you might want to use:

Convention: lower case. Simply smash words together by removing the spaces. The name `mydata` is a good example of this. Lower case convention works well if you're only combining two short words; however, it can be difficult to read longer names. For example, `maledepressionmodel1` is just too many consecutive letters.

Convention: camel case. Lower case convention is modified to camel case by capitalizing the first letter of every word (this may or may not include the first letter of the first word). The result is a name that looks like a camel's back, with capital letters forming "humps." So, `myData` and `MyData` would be considered camel case, as would `maleDepressionModel1` and `MaleDepressionModel1`. The

capital letters help make the words (or more importantly, the breaks between them) stand out.

Convention: snake case. Snake case replaces the spaces between words with underscores. Examples of snake case would be `my_data` and `male_depression_model1`. Snake case makes the names a little longer by adding underscore(s), but it is often the most readable approach.

Convention: consistency. Regardless of whether you choose to go with lowercase, CamelCase, or snake_case, it is helpful to be consistent throughout. Consistency will help you remember teh names you have given different objects. I generally like using snake case, though there are some times when I use lower case for very simple names like `mydata`. For anything more complicated than that, I prefer snake case. Snake case also means I know I will never have capital letters in my names, so it's one less thing to have to remember.

Indexing

You already saw an example of indexing in Data Frames. When you have an object containing a sequence of numbers or other values, you might want to get a specific entry or a subset of the entries held in that objects. The process of obtaining these parts of an object by pointing to their locations is called **indexing**.

We index an object by using square brackets after the object's name. If the object is one-dimensional, we simply give the location or locations of the values we want to see. For example:

```
powers = c(2, 4, 8, 16, 32, 64, 128, 256, 512, 1024)
powers[1]
```

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

```
powers[2]
```

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

```
powers[3]
```

```
## [1] 8
```

```
powers[10]
```

```
## [1] 1024
```

```
powers[c(2, 4, 5)]
```

```
## [1]  4 16 32
```

With two or more dimensions, we need to give the index of each dimension. In an introductory class, you are unlikely to see anything more than two dimensions, so we will only discuss that here. Examples of two-dimensional objects are data frames and matrices. From two-dimensional objects, the square brackets include the row first, then the column. Using the data frame **gpa** from Data Frames, we see:

```
gpa
```

```
##      ID             Name         Major  GPA GPAcorrect
## 1 20304      Adam Baines       English 3.30       3.40
## 2 20305 Claire Dougherty    Statistics 3.53       3.63
## 3 20306      Eric Forest    Psychology 3.87       3.97
## 4 30101      George Hull Public Health 3.79       3.89
```

```
gpa[1, 2]
```

```
## [1] "Adam Baines"
```

```
gpa[3, 4]
```

```
## [1] 3.87
```

If we leave out one of the indices, you will get the entire row or column you asked for:

```
gpa[1, ]
```

```
##      ID        Name   Major GPA GPAcorrect
## 1 20304 Adam Baines English 3.3        3.4
```

```
gpa[, 1]
```

```
## [1] 20304 20305 20306 30101
```

Remember to include the comma so it's clear whether you are giving the row (before the comma) or the column (after the comma).

If we want a sequence of consecutive values, we can use the colon `:` to give an sequence. The code `a:b` gives a, a+1, a+2, up to b. For example:

```
1:5
```

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

```
(-2):3
```

```
## [1] -2 -1  0  1  2  3
```

```
0.5:9
```

```
## [1] 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```

```
powers[1:5]
```

```
## [1]  2  4  8 16 32
```

Brackets

R uses three types of brackets for different reasons.

In the previous section, we have seen **square brackets**, `[]`, are used to index an object.

Rounded brackets, or parentheses, `()` are used for order of operations in mathematical calculations[2] as well as for providing the inputs to functions. We will talk about functions more in the next chapter, but we have already seen them in practice, for example:

[2] for example, `2 * (1 + 4)` gives `2 * 5`, or 10.

```
mean(gpa$GPA)
```

```
## [1] 3.6225
```

Here, `mean` is the name of the function and we use `gpa$GPA` as an input to that function. This is similar to the mathematical notation $f(x)$, where f is a function and x, placed inside the parentheses, is the input.

The last type of bracket you might see in R are **curly brackets**, `{}`. Curly brackets are used to set off some piece of code from the rest of the script. For example, if you are defining your own function, you write the code for the function inside curly brackets. If you are doing some kind of conditional coding, where you run different code depending on the value of an object, each part of code is wrapped in brackets. You're unlikely to see code that needs curly brackets in an intro class, but in case you do, you can tell it is setting the code apart as special in some way.

Commenting

An R script file is a sequence of commands you are telling R to execute. Sometimes these commands can be complicated, either because there is a long sequence of them or because one line is a bit tricky. If you want to leave yourself or someone else a note, you can add **comments** to your code using `#`. This symbol is called a number sign, pound sign, or hash tag, depending on when you were born.

Comments are parts of your script file that are meant to be read by a human and not interpreted as code. When you place a `#` in a line, anything after that is considered a comment and skipped when R is trying to run your code. For example:

```
mean(gpa$GPA)  # calculates average GPA
sd(gpa$GPA)  # calculates standard deviation of GPA
```

If you want a comment to span more than one line, you will need to start a new line and include another `#` in that line. We usually like to line up `#` that make the same comment

```
mean(gpa$GPA, na.rm=TRUE)  # calculates the average
                           # GPA, but first removes
                           # any missing values and
                           # uses the number of remaining
                           # values in the denominator
```

That's actually a pretty bad comment. It's too wordy and really just explaining what the one function is doing. If we really wanted to know what `na.rm=TRUE` means, we could look in the help file for the **mean** function.

Comments should make your code easier to read. This is helpful for people like group members and instructors, but is also helpful for you when you return to your work. A common refrain in coding is, "Your most important collaborator is future you." That means you need to leave yourself hints and notes so that when you go back to this code tomorrow or in six months or more, you can more easily pick it up and understand what past you was trying to do.

Use of White Space

Another way to increase the human readability of your code it to use white space liberally. White space is any space in your script that doesn't include character code. This means everything from spaces to indentations to blank lines. While some of the suggestions below aren't formal syntax, they are preferred practice for making code easier to read.

Spaces. As much as possible, you should use spaces around equal signs, mathematical functions, and after commas. For example:

```
myname = 'Travis'  # clearly separates the object name from the value it takes
myname='Travis'    # takes much longer to read and understand

2 * 3  # is preferred
2*3    # is too squished

mean(gpa$GPA, na.rm=TRUE)  # separates function inputs
mean(gpa$GPA,na.rm=TRUE)   # runs everything together

gpa[2, ]  # clearly shows a missing column index
gpa[2,]   # takes away the blank space that should help indicate an entry is missing
```

Indentation. If you are writing a line of code that is too long to easily see or read as a single line, you can start a new line and continue that code. This is most common in functions that take multiple inputs, especially if some inputs have long names. You will usually indent two spaces for every level you are coding, or if you are using multiple lines inside a function, you will indent to the start of the first function input. See here:

```
oddsratio = (pa / (1 - pa)) /
  (pb / (1 - pb))
```

```
mean(gpa$GPA,
     trim=0.05,
     na.rm=TRUE)
```

R Studio will often default to indenting when appropriate. This is one of those cases where it's best if you let R Studio do it's thing. It may seem like a waste of space, but it's not like you're printing much of this out anyway.

Blank lines. You have already seen me use blank lines in the chunks of code above. You generally want to leave blank lines between lines of code. One exception is if (1) each line of code is pretty simple on its own AND (2) each line of code is leading directly into the next, as below:

```
mytab = table(mydata$x, mydata$y)
myprops = prop.table(mytab, margin=1)
```

Here, the ultimate goal was to get a table of proportions, but we needed the intermediate step of the table of counts. For someone who is moderately familiar with R code, the connection between the two lines is easy to see.

You can also do away with the blank lines if you are running a series of very simple but loosely related lines of code:

```
mean(gpa$GPA)
sd(gpa$GPA)
median(gpa$GPA)
quantiles(gpa$GPA, probs=c(0.25, 0.75))
```

These lines don't feed right into one another, but they are relatively simple calculations all done on the same data, essentially getting the basic descriptive statistics.

Chapter 9

Functions

If objects are the "nouns" of the R language, then functions are the "verbs." This analogy isn't perfect because technically functions are objects, too, but they are special types of objects. Functions don't hold data or information, they *do* things.

It's nearly impossible to write R code without using functions, so there are numerous examples of functions in the previous chapters. As described in R Language and Syntax, functions can usually be identified by their use of parenthesis to contain their inputs. In this chapter we will look at functions in a little more detail in the hopes of demystifying their inner workings, without digging too deep into their actual code.

Using Functions

We use functions for everything from installing and loading packages to reading in data, analyzing and graphing data, and saving results. With so many functions for so many purposes, it is important that we don't forget they all have very similar structures.

Nearly all functions require some number of inputs, or **arguments** in order to run. Many times the arguments have default settings so that we do not need to provide values for every possible argument every time. You can learn about the arguments for any function by reading the function's help file, which you can access by running the code `?function_name` in the R console. For example, to learn about **rnorm**, which generates observations of random variables from a normal distributions, run `?rnorm`. This particular help file is a bit more complicated because it combines the help files or **dnorm**, **pnorm**, **qnorm**, and **rnorm**, all of which are functions for working with the normal distribution and use many of the same arguments.

At any rate, under the *Usage* section, we see `rnorm(n, mean = 0, sd = 1)`, telling us that **rnorm** takes three arguments: `n`, `mean`, and `sd`. Reading into the *Arguments* section, we see what those arguments represent:

- `n`: number of observations
- `mean`: vector of means
- `sd`: vector of standard deviations

So, `n` tells R how many observations we want to generate, `mean` is the mean of the underlying distribution, and `sd` is the standard deviation of the underlying distribution. When we see `mean = 0` and `sd = 1` in the *Usage* section, we are reading the defaults. If we don't provide a mean or standard deviation when we run the function, R will assume values of 0 and 1, respectively. For example:

```
rnorm(n=5)
```

```
## [1] 2.1866493 2.7822521 0.3528365 0.0514373 0.4489899
```

```
rnorm(n=5, mean=0, sd=1)
```

```
## [1] 2.1866493 2.7822521 0.3528365 0.0514373 0.4489899
```

We see the function produces the same values with and without the `mean` and `sd` arguments specified[1]. I can choose my own values for the arguments, though. For example, IQ scores are approximately normally distributed with mean 100 and standard deviation 15, so if I wanted to generate seven values from this distribution, I could run

```
rnorm(n=7, mean=100, sd=15)
```

```
## [1] 134.84225 106.60976 116.56707  91.85262 119.12863 112.35178 101.05142
```

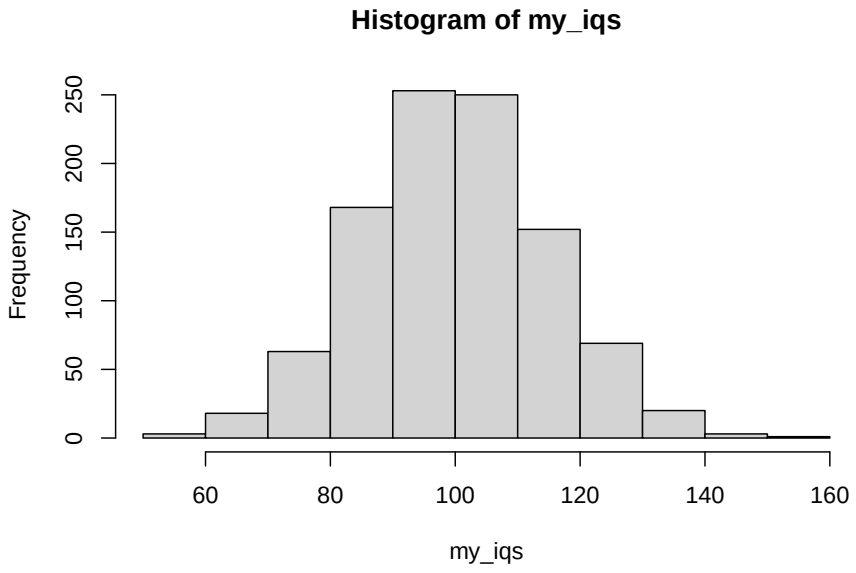
R help files can be difficult to read. Being able access the help files and at least read and understand the *Usage* and *Arguments* sections will save hours of heartache over the course of a semester.

[1] Functions like **rnorm** should generate random, or technically "pseudo-random" values, so they shouldn't produce the same numbers in consecutive runs. You are able to re-create random values by setting the "seed" of the random number generator. I did that here but hid the code. If you run those two lines of code successively in R, you will get two different results (and results which differ from what I got).

Function Outputs

Most R functions output some result from their analysis of data. These outputs are objects of various classes. In an intro class, most of these classes will be among those described in the Outputs section of More Data Structures. If you simply run a function, most of the time the output will be printed to the screen, but then it is forgotten. If you want to save outputs to recall them later, you need to save them to a named object. Again, this is something we've seen throughout these notes. Suppose I want to generate a bunch of observations from a theoretical IQ distribution then plot them in a histogram. This will require me (or R) to: (1) generate the values, (2) save them as an object in the envirnoment, and (3) plot the new object. In the code below, I (1) generate 1,000 observations from a normal distribution with mean 100 and standard deviation 15, (2) save these 1,000 values to a numeric vector named `my_iqs`[2], and (3) create a histogram of the values saved in `my_iqs`.

```
my_iqs = rnorm(1000, mean=100, sd=15)
hist(my_iqs)
```



[2]I didn't tell R that `my_iqs` is a numeric vector, R is able to determine that from the output that **rnorm** creates.

Functions and Packages

As we described in Packages, one of R's most powerful aspects is the ability of users to contribute their own functions through packages. All functions in R are housed in packages. Many packages come pre-installed, so you do not need to access CRAN to download them. For example, packages like **base**, **utils**, and **stats** include functions that are so necessary for R to run or to perform basic statistical analysis that they are assumed to be needed all the time. They are installed when you download R and loaded into every R session.

You can determine which package a function is in through the help file. In the first line of the help file, above the title, you will see a line with the function name or something similar, followed by curly brackets, `{}`. The package the function is in is given between the brackets. For example, `?rnorm` and `?mean` will show that both functions come from the **stats** package. Similarly, `?install.packages` shows the function is in the **utils** package and the **hist** function is in the **graphics** package (run `?hist`).

Sometimes multiple packages will have functions with the same name. These functions may or may not be calculating the same thing and may or may not require the same arguments as inputs. When you use a function name that is ambiguous (functions with that name are in two or more packages loaded into your current R session), R chooses the function from the most recently loaded package. If you want to ensure you are using the function from a specific package, you can use the code `package_name::function_name`. For example, if you want to ensure you are using the **mean** function from the **stats** package, instead of just typing `mean(my_iqs)` you can use `stats::mean(my_iqs)`. This takes a lot more work in typing and identifying packages, so most package writers try to make somewhat unique function names and not replicate functions from other packages. For example, there is no need to write a new function to calculate the mean when **stats::mean** works just fine.

If you are ever in a situation where R tells you a function doesn't exist (and you're sure the spelling is correct), chances are pretty high that you have not loaded in the package that contains the function. You can read more about this error and others in Common Error Messages.

Chapter 10

R Markdown

R Markdown allows you to combine R code, analytical and graphical results, and text narrative into a single document. Instead of copying your R results into Word, think of this as moving your Word writing into R. You can use it to generate nicely formatted HTML (webpage), Microsoft Word, and PDF documents that can be shared with team members and other collaborators. If you update your analysis, you can create an updated output file with the click of a button. With a little work, R Markdown keeps you from having to copy and paste results from R into a Word doc, keeping both in step with each other and saving you from having to keep track of updates.

As you might expect, R Markdown has become a very popular and powerful integration into the R and R Studio universe. It does have its own syntax and quirks, though, so this chapter is designed to help you get through those.

> NOTE: If your instructor is not requiring you use to R Markdown, you should be able to skip this chapter. However, if you think you might be taking more advanced statistics courses - or just want a cool skill to talk about at parties - learning R Markdown is far easier than learning R itself and definitely worth the time.

How R Markdown Works

R Markdown files are text files that can be opened and edited in R Studio the same way an R script file can, though R Markdown files have the extension **Rmd**[1]. They are an extension of the Markdown language for formatting documents, built to incorporate R code and results into the document. Combining

[1]From here on, I will use "Rmd" when describing an R Markdown file and "R Markdown" when describing the R Markdown language

R code, results, and text into a single file essentially creates a complete report of a data analysis in one place. If you need to adjust code, or if new data comes in, you can re-run the code and update the entire report in real time. This makes R Markdown an extremely powerful tool for modern statisticians and data analysts.

In order to create the output file, you need to "knit" your Rmd file. There is a button to do this in R Studio (the Knit button, with a picture of a ball of yarn, is near the Save, Spell check, and Search buttons), so it seems like a relatively easy process. However, there are many steps under the hood. First, the Rmd file needs to be parsed to ensure all the syntax is understandable and the R code works. Then the file is processed by a program called pandoc which will write the output file, assuming you have the correct software. For example, if you want to create a Word file, you need Word on your computer. If you want to create a PDF file, you need a TeX editor on your computer[2].

Writing an R Markdown file

Header

First thing first: at the top of your Rmd file is the header[3]. The header contains information such as the title and author of the document, and the date. All of these can be changed or removed. It also includes the output format, which tells you what type of document the result will be knitted to; for example, HTML, Word, or PDF. The header can get very long and complex to include things like tables of contents or other output options. For your purposes, the title, author, date, and output fields will be the most important and may be all that you see.

Formatting Text

One you are past the header, you are into the body of the document. Markdown was designed to be an easy to use and read formatting system. That means it is kind of limited in what it can do, but it can be learned pretty quickly. The main special code parts you need to know are:

- Sections: Start a line with a hashtag **#** followed by a space then the section name.

[2]You can install the R package **tinytex** for this purpose, allowing you to run everything through R without having to learn an entirely new computer language.

[3]The Rmd header uses YAML, which stands for *YAML Ain't Markup Language*. If you search for information about Rmd files, you're likely to come across the acronym YAML, which essentially means the header. I had to lookup what YAML meant to write this - I had never seen it before.

- Subsections: Start a line with two hashtags `##` followed by a space and the subsection name.
- Paragraphs: Leave a blank line between paragraphs.
- Bulleted list: Leave a blank line before starting the list, then start each line with a dash `-` followed by a space and the content of the point. Start a new point on a new line with a new dash. Leave a blank line after the list.
- Ordered list: Similar to above, but start lines with numbers or letters followed by a period instead of a dash. For example, `1.`, `2.`, `3.`, etc. or `A.`, `B.`, `C.`, etc.
- Italic text: If you want text to look italic when knitted, surround it with asterisks like `*italic text here*`.
- Bold text: If you want text to look bold, surround it with two asterisks `**bold text here**`.

An example using some of this formatting is below:

```
# Problem 1

## Part a

1. My name is **Travis**
2. I study *statistics*

## Part b

This is the first paragraph of this part.

This is the second paragraph.
```

These few bits of code formatting should be able to get 90% of what you need for your classes. If you need more, there are plenty of cheat sheets and cookbooks you can find online.

Including R Code

The biggest benefits of R Markdown come from mixing your R code with the text explaining the code and results all in one document. To include R code, you need to create a code **chunk**. There are couple ways to do this. The first is to code the chunk directly. First, leave a blank line after your paragraph of text. Then begin the chunk with a line of three back-ticks and `{r}` and end the chunk with a line of three back-ticks.

The lines inside the code chunk are treated as R code and evaluated line by line. Additionally, you can use the code chunk button in R Studio (a white C in a

green box) and select R. Doing this will place the starting and ending lines for the code chunk in your document for you.

You can have as many code chunks in your document as you want, and they can be as long as you want them to be. The code will be run continuously from the first line of the first chunk to the last line of the last chunk, so values and objects that you save in earlier chunks can be referenced and used in later chunks, but not vice versa. Any output generated from the R code will appear directly below the line of code that created it.

Knitting an R Markdown file

When you click the Knit button in R Studio, you are starting a very complex process that is supposed to be mostly hidden from you. The Markdown syntax is read and formatted while the R code is evaluated and output is generated.

There are a growing list of output formats, but the standard ones are HTML, Microsoft Word, and PDF. You can see the output type in the header line that starts `output:`, which will be followed by `html_document`, `word_document`, or `pdf_document`, respectively, for the three formats. You can change the output format by editing the header directly or by clicking on the small down arrow next to the Knit button and choosing your format. If you have knitted your Rmd file to mutliple formats, the header will look like:

```
output:
  html_document
  word_document
  pdf_document
```

and when you click Knit, you will get the first output format on the list.

> IMPORTANT: When you knit an Rmd file, you need to ensure that *all* the code you need for your analysis is in the Rmd file itself. That means any packages you need to load or data you need to read in are explicit in the code chunks of your Rmd file. When you knit a file, a new R session opens in the background of your R Studio program. This session is blank before it runs through your code. If you don't read in data within the Rmd file itself, it won't be read in when the code is evaluated during the knit. This is true even if the data already exists in the evironment in your visible R Studio session. The point is that the file should be able to be run from any computer without any additional work, so it needs to be entirely self-contained.

If the knitting works, the output file will appear in the same directory as your Rmd file with the same file name, distinguished only by the file extension (.html, .docx, or .pdf). If you get an error knitting (which will be very common early on, hopefully getting less common as you get more familiar), you will need to address the issue and try to knit again. Some common R and R Markdown error messages and their solutions are described in the next chapter.

Chapter 11

Common Error Messages

Sometimes running R code will result in red text printed to the Console. This red text is either a Message, Warning, or Error.

- A **Message** is a note from the function author reminding you of something important, maybe a default value for an argument or a hint on interpreting the results.
- A **Warning** is telling you that something in your function wasn't quite correct, but R took its best guess to try to fix the problem and was able to get a result. The Warning indicates that the result might not be what you expect it to be. You'll want to go back and check your work. Many times the code that results in a Warning is fine, but it can be made more correct or more explicit to not initiate the Warning message.
- An **Error** occurs when R cannot compute the code you tried to run. Unlike with a Warning, an Error means no output was generated, and your back where you started before you ran the code.

If you write R code, you will inevitably receive an error message. That's okay, it happens to everyone. Even expert R programmers make errors all the time. I (not that I'm the epitome of expert) make numerous errors almost every time I use R. With experience the errors occur less often, but a more important skill that grows over that time is learning to identify and fix the errors. When errors occur less often and you're better at fixing them, the whole process becomes more enjoyable.

In this chapter we describe some errors you're likely you're likely to come across, why they happen, and how to fix them. Throughout, I use `mydata`, `myfun`, `mypkg`, `mydir`, and `myfile` as place holders for data objects, functions, packages, directories, and files, respectively.

Syntax Errors

Anytime you receive an error message it can be because of an error in your R syntax.

Spelling and Capitalization

As we've discussed, R is very picky with names of objects and functions. These names have to be spelled exactly right and have the correct capitalization (remember `apple`, `Apple`, and `APPLE` are three different things in R). If you ever run a line of code with a mis-spelled object, you'll get the error

```
Error: object 'mydata' not found
```

If a function name is mis-spelled or mis-capitalized, you'll get the error

```
Error in myfun(mydata) : could not find function "myfun"
```

Quotation Marks and Brackets

Quotation marks (single and double) and brackets (round, square, and curly) can lead to their own issues. Ordering of any of these are important for correct computation and just because a line of code runs without an error doesn't mean R is calculating what you want it to[1]. If you start a quotation or bracket and don't close it with a matching quotation or bracket, R will expect more to come. The next line in your Console will start with a plus `+` indicating R is waiting for more input:

```
> ((4 + 2) * 3
+
```

In addition, if you have one too many closing brackets, you will get an error that one of them is "unexpected":

```
(4+2) * 3)
Error: unexpected ')' in "(4+2) * 3)"
```

[1] Remember PEMDAS order of operations - R follows this convention.

Could Not Find and Cannot Open

One of the more common error messages new R uses are likely to get is that R "could not find" something or that something "does not exist." Mis-spelling or mis-capitalization is a likely culprit, but not the only one. Below are some other possibilities.

Object not Found

```
Error in myfun(mydata) : object 'mydata' not found
```

Maybe you think you saved some data or analytical result with the name `mydata`, but you forgot to do this. This is especially common if you are copying and pasting code or sharing code with a classmate/collaborator. Go back through your code and see where/if you create an object named `mydata`. You might want to re-run that part of your code.

Could not Find Function

```
Error in myfun(mydata) : could not find function "myfun"
```

If you can't find a function, that means you haven't loaded the package that the function is in. Determine the name package that the function is in and run `library(mypkg)`, then try your original line of code again.

No Such File or Directory

```
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'mydir/mydata.R': No such file or directory
```

or

```
Error in readChar(con, 5L, useBytes = TRUE) : cannot open the connection
In addition: Warning message:
In readChar(con, 5L, useBytes = TRUE) :
  cannot open compressed file 'mydir/mydata.R', probable reason 'No such file or directory'
```

You get errors like this when you are trying to access a file outside of R. This is usually a data file, either RData, csv, or some other format. The most likely

reason for this is that your path is starting in the wrong spot: your working directory isn't what you think it is. File paths can be very tricky for new users of any program. For more details about file paths and working directories, see Computer File Systems and Working Directories.

Another possible issue with your code is that you may be missing the file extension or using the wrong extension. For example, you may write `load('mydir/mydata')` when it should be `load('mydir/mydata.RData')`.

Errors in R Markdown

When you knit an Rmd file, the messages and errors will appear in the R Markdown tab in the bottom left pane of the R Studio window. Knitting an Rmd file creates a lot of output even when everything runs as expected. This can make understanding the errors you receive more difficult to decipher.

Markdown Syntax

I'm not sure that I've ever received and error from my Markdown syntax (headers, lists, bold, etc.). What you will most likely see is an output will be generated, but the formatting won't be right. For example, (1) instead of a header, you will just see a line starting with a hashtag, or (2) instead of a new paragraph you will see a two-paragraph run-on. In most of these cases, the problem is missing white space around your Markdown formatting. In the two examples I just gave, (1) you would need a space after the hashtag in order for Markdown to recognize a header and (2) you need a blank line between paragraphs.

The point here is the same with all R (and any computer) code: *Just because you don't get an error doesn't mean you're getting what you want.*

R Code

All of the R code errors discussed above can happen in an Rmd file as well. Remember that you need to load any packages and data directly into your Rmd file.

When you encounter an error in your R code in an Rmd file, you are given a line number. This number is the first line of the code chunk in which the error occurs. Though the error may occur here, the actual problem may be in an earlier chunk. For example, maybe you mis-spelled the name of an object when you saved it, essentially creating a mis-named object. If you try to access this object in a later chunk, the error will come here instead of the incorrect naming in the earlier chunk.

In order to limit the number of errors you receive when knitting a file, some good rules of thumb are:

- Make your code chunks as small as is reasonable, so that there are fewer lines of code to debug to find your error.
- Knit early and often. For example, if you are working on your homework, knit after each problem. If you knit after problem 4 and everything works, then you knit after problem 5 you get an error, you know the issue is in problem 5. If you wait until you are done to knit for the first time, it will be harder to find the error. In addition, remember that when you get an error, R stops running. If you knit and receive an error, you may fix that error but the next error will pop up the next time you try to knit. This gets far more annoying than fixing errors as you go and it isn't as beneficial from a pedagogical standpoint - seeing and fixing errors immediately after you make them will help you learn from them more quickly than writing an entire assignment then going back and fixing errors in a way that is disconnected from the original coding.

Permission denied

Even if all your code is running correctly and producing results, you might get an error in the very last stage of knitting telling you "permission denied" or "conversion failed":

```
pandoc.exe: myfile.docx: openBinaryFile: permission denied (Permission denied)
Error: pandoc document conversion failed with error 1
Execution halted
```

This means that you have an old version of the knitted document (here, **myfile.docx**) open. If a file with the name you want to knit to is already open, R cannot create or edit that file. Close the old version of the file and knit again. Just remember that when you knit, you will lose that old version completely as it gets replaced with the updated knitted document. If there's something in that old version you think you might want, rename the old file to something like **myfile-old.docx** before knitting the newer version of the Rmd file[2].

Other Help

Antoine Soetewey has a good list of **Top 10 errors in R and how to fix them** [LINK]

[2] A better system would be to use version control software, but that's not something you're likely to encounter in an intro stats class.

Chapter 12

Additional R Resources

In this guide I have tried to give you something close to the bare minimum you'd need to understand R and R Studio enough to complete an introductory statistics course. That's obviously a fine line to walk. In addition, I have tried to be succinct in what I have chosen to cover to make the guide and each of the chapters more appealing and digestible to readers who have plenty of other things on their plate. All of this means there are certainly things you'll need that aren't in this guide.

This chapter is a list of resources for self-paced learning of the R environment. For the most part, these resources assume a basic introductory statistics background and little to no computing experience. Resources for more advanced learning are noted as such.

Courses

- Hey, it turns out a few people have already done this! The courses on this list range from a couple hours to five or more weeks. I'll still keep a list of good ones I hear about or ones that aren't on this list. [LINK 1] [LINK 2]
- **Swirl Stats**: This is an R package that teaches you to use R from within R/R Studio. Highly recommended for first timers. [LINK]
- **Statistics with R**: This is a series of Coursera courses run by faculty at Duke University. Full participation in the courses costs money, but I think you can audit the courses for free (access to materials but no grading or certificate). [LINK]
- **Data Science**: Another Coursera series out of Johns Hopkins. A very popular series of courses but from my understanding it is given at a more

advanced level. If you are new to R you may want to start somewhere else and return to this later. [LINK]

- **R exercises**: Not so much a course as a set of practice problems to help understand the inner-workings of R. Not really focused on data analysis, but definitely helpful to understand how R is interpreting your commands. [LINK]

Websites

- **Getting Started with R**: A list of steps and resources put together by RStudio. I haven't gone through much of it, but my guess is it's top notch. [LINK]
- **The Big Book of R**. Billed as "your last-ever bookmark," this is a curated list of R online books by Oscar Baruffa. Books are orgainzed by topic and each has a short description with the link. It's like this list on steroids. [LINK]
- **What They Forgot to Teach You About R** by Jennifer Bryan, Jim Hester, Shannon Pileggi, and E. David Aja. This would be a great companion to, or next step after, this book. It provides great suggestions for preferred practices and slightly more in-depth - but still very readable - details to the topics covered here if you want more grounding in these topics. [LINK]
- **R Bloggers**: Lots of tutorials, updated every day. For those super-new to R, check out the **Learn R** tab. [LINK]
  - In particular, the **Learn R From Scratch** tutorials by Selva Prabhakan look promising. [LINK 1] [LINK 2] [LINK 3]
- **UCLA's Institute for Digital Research and Education**: Focuses more on how to perform particular analyses using R. [LINK]
- **R Weekly** is a weekly aggregator of R news. It will reference tutorials for new users, and the examples of what some intermediate-to-advanced R programming can do are always interesting. You can sign up to receive updates. [LINK]
- **YaRrr! The Pirate's Guide to R**: Includes a free pdf book. I haven't read it, but I'd have to assume it's entertaining. [LINK]
- **twotorials.com** posts two-minute how-to videos for learning R tasks. [LINK]
- **aRrgh: a newcomer's angry guide to R** goes through some of the foundations of R, focusing on the frustrations you might come across as a new user. [LINK]

Books

- **O'Reilly** is a publisher with a strong reputation in technical publishing and a number of books focusing on R. Some are more advanced, but the introductory books should be approachable and good references. Many of the books are visually distinctive by using black and white drawings of animals on their covers. [LINK]
- **R in Action** by Robert Kabacoff [LINK]
- **R for Everyone** by Jared Lander [LINK]
- **Learning Base R** by Lawrence M. Leemis. This book comes recommended for people with a computing background who want to learn R with a focus on R as a programming language. [LINK]
- **The Art of R Programming** by Norman Matloff [LINK]
- Here's a list compiled by Liang-Cheng Zhang, which nicely groups books by reader level. It is from 2016 so it's a bit dated, but was good at the time: [LINK]