

# R Module 1

Alex Fout\*

Lane Drew†

19 Jun, 2023, 02:02 PM

## Contents

### 1 Welcome!

Hi, and welcome to the R Module 1 (AKA STAT 158) course at Colorado State University!

This course is the first of three 1 credit courses intended to introduce the R programming language to those with little or no programming experience.

Through these Modules (courses), we'll explore how R can be used to do the following:

1. Perform basic computations and logic, just like any other programming language
2. Load, clean, analyze, and visualize data
3. Run scripts
4. Create reproducible reports so you can explain your work in a narrative form

In addition, you'll also be exposed to some aspects of the broader R community, including:

1. R as free, open source software
2. The free RStudio IDE
3. Publicly available packages which extend the capability of R
4. Events and community groups which advocate for the use of R and the support of R users

More detail will be provided in the Course Topics laid out in the next chapter.

#### 1.0.1 How To Navigate This Book

To move quickly to different portions of the book, click on the appropriate chapter or section in the the table of contents on the left. The buttons at the top of the page allow you to show/hide the table of contents, search the book, change font settings, download a pdf or ebook copy of this book, or get hints on various sections of the book. The faint left and right arrows at the sides of each page (or bottom of the page if it's narrow enough) allow you to step to the next/previous section. Here's what they look like:

---

\*Department of Statistics, Colorado State University, fout@colostate.edu

†Department of Statistics, Colorado State University, lane.drew@colostate.edu



Figure 1: Left and right navigation arrows

## 1.1 Associated CSU Course

This bookdown book is intended to accompany the associated course at Colorado State University, but the curriculum is free for anyone to access and use. If you’re reading the PDF or EPUB version of this book, you can find the “live” version at <https://csu-r.github.io/Module1/>, and all of the source files for this book can be found at <https://github.com/CSU-R/Module1>.

If you’re not taking the CSU course, you will periodically encounter instructions and references which are not relevant to you. For example, we will make reference to the Canvas website, which only CSU students enrolled in the course have access to.

## 2 Course Preliminaries

“Learning to code is useful no matter what your career ambitions are.” —Arianna Huffington,  
Founder, The Huffington Post

In this chapter, we’ll discuss the preliminary details of the course. Then you’ll run some R code and learn more about R and the R community.

### 2.1 This Textbook

This course is presented as a bookdown document, and is divided into chapters and sections. Each week, you’ll be expected to read through the chapter and complete any associated exercises, quizzes, or assignments.

#### 2.1.1 Special Boxes

Throughout the book, you’ll encounter special boxes, each with a special meaning. Here is an example of each type of box:

💡 **Reflect** This box will prompt you to pause and reflect on your experience and/or learning. No feedback will be given, but this may be graded on completion.

✍️ **Assessment** This box will signify a quiz or assignment which you will turn in for grading, on which the instructor will provide feedback.

📖 **Progress Check** This box is for checking your understanding, to make sure you are ready for what follows.

🎥 **Video** This box is for displaying/linking to videos in order to help illustrate or communicate concepts.

⚠ **Caution** This box will warn you of possible problems or pitfalls you may encounter!

🌟 **Bonus** This box is to provide material going beyond the main course content, or material which will be revisited later in more depth.

💡 **Feedback** This box will prompt for your feedback on the organization of the course, so we can improve the material for everyone!  
Any of the boxes may include hyperlinks like this: I am a link or code like this `This is code`.

### 2.1.2 How This Book Displays Code

In addition, you may see R code either as part of a sentence like this: `1+1`, or as a separate block like so:

```
1+1
```

```
[1] 2
```

Sometimes (as in this example) we will also show the **output** (in yellow), that is, the result of running the R code. In this case the code `1+1` produced the output 2. If you hover over a code block with your mouse, you will see the option to copy the code to your clipboard, like this:

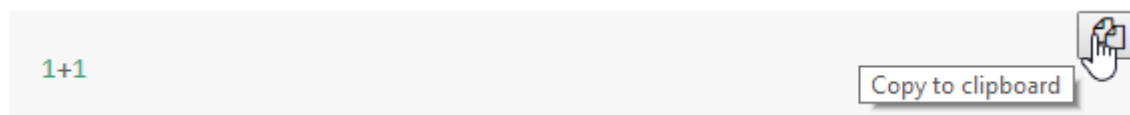


Figure 2: copying code from this book

This will be useful when you are asked to run code on your computer.

### 2.1.3 Next Steps

When you're ready, go to the next section to learn about the course syllabus and grading policies.

💡 **Feedback** Any feedback for this section? [Click here](#)

## 2.2 Course Topics & Syllabus

Broadly speaking, the topics of this course are described by the Chapter Titles. Here's what each entails:

- **Course Preliminaries:** Introduction to R and the world of R
- **Installing R:** Like it sounds, setting up your computer so you can work with R.
- **R Programming Fundamentals:** The basics of programming in R, the building blocks that you need in order to do anything more interesting.
- **Working with Data:** How to do meaningful things with data sets. Probably the most useful Chapter of the book.
- **Creating R Programs:** More programming concepts to increase your R Power!

### 2.2.1 Syllabus

First, some important details:

- **Instructor:** Lane Drew
- **Office Hours:** Held in the Statistics Success Center (Weber 223A), schedule available on Canvas.
- **Webpages:** Canvas, this textbook
- **Course Credits:** 1. This means ~1 hours of lecture and 4 hours of work outside of lecture per week.
- **Textbook:** You're reading it right now. The textbook will be your primary learning resource. You'll be expected to read through the required sections, watch any relevant videos, and complete any reflections, progress checks, and assessments along the way. On days when a quiz is due, you should complete the reading *before* you take the quiz.
- **Prerequisites:** None
- **Assignments/What-to-turn-in:** This course will be graded on three types of assignments: Progress Checks, Homeworks, and Quizzes. There will be four of each. Most weeks, you will have one of these three types of assignments due. Due dates will be specified on Canvas and assignments will be due at 11:59pm on the indicated day (please see schedule below).
- **Progress Checks:** As you work your way through the textbook, you'll encounter purple "Progress Check" boxes. For the first Progress Check, you'll submit your responses directly to canvas. For Progress Checks 2-4, you'll fill in an R Markdown document and submit it to canvas. You'll be provided a template to fill in as you complete the progress checks. To turn in the document, you'll **knit** the document to HTML or PDF and upload to Canvas. (More details coming later in the book!). Progress checks will be graded on completion, organization, and correctness. Progress Checks must be turned in by 11:59pm (Mountain) on the day they are due. Half credit will be given for a two-day window after the due date, after which no credit will be possible.
- **Homework:** There are four homeworks during the semester. You'll complete each homework using R. Homeworks must be turned in by 11:59pm (Mountain) on the day they are due. Half credit will be given for a two-day window after the due date, after which no credit will be possible.
- **Quizzes:** There will be four 15 minute Canvas quizzes during the semester. Quizzes must be completed by 11:59pm (Mountain) on the day they are due. **There are NO late quizzes accepted after the due date has passed. If you cannot complete the quiz on the day it is due, you are expected to do it early.**
- **Exams:** There will be no exams in this course

- **Lectures:** Lectures will be held on Fridays. There will be *mini-lectures*, approximately 10-30 minutes. The mini-lectures will be based on previously read material, no new material will be presented. Students are expected to have read the material before the lecture. The remainder of the time will be *student-led*. We will cover questions students may have or work on homework together.
- **Grading:** The grading for the course is apportioned like so:
  - Progress Checks: 30%
  - Homework: 40%
  - Quizzes: 30%

### 2.2.2 Assignment Templates

In order to complete the progress checks and course assignments, you'll need to start from these templates:

Progress Checks

- (Progress Check 1 will not require a template)
- Progress Check 2
- Progress Check 3
- Progress Check 4

Assignments

- Homework 1
- Homework 2
- Homework 3
- Homework 4

### 2.2.3 Course Policies

- **Late Work:** Homework and Progress Checks must be turned in on time to receive full credit. You may turn in Homework and Progress Checks up to 2 days late for up to 50% credit.
- **Group Work:** Students are welcome to discuss the course with each other, but all work you turn in must be your own. This means no sharing solutions to homework, progress checks, or quizzes. You may not work with other students on quizzes. You *are* welcome to seek help on Canvas discussion boards and during office hours.
- **Students with Disabilities:** The university is committed to providing support for students with disabilities. If you have an accommodation plan, please provide that to me as soon as possible so we can discuss appropriate arrangements.
- **Growth Mindset:** This phrase was coined by Carol Dweck to reflect how your learning outcomes can be affected by the way you view the learning process. To quote Dweck: "The view you adopt for yourself profoundly affects the way you lead your life... Believing that your qualities are carved in stone - *the fixed mindset* - creates an urgency to prove yourself over and over. If you have only a certain amount of intelligence, a certain personality, and a certain moral character — well, then you'd better prove that you have a healthy dose of them. It simply wouldn't do to look or feel deficient in these most basic characteristics... There's another mindset in which these traits are not simply a hand you're dealt and have to live with, always trying to convince yourself and others that you have a royal flush when you're secretly worried it's a pair of tens. In this mindset, the hand you're dealt is just the starting point for development. This growth mindset is based on the belief that your basic qualities are things you can cultivate through your efforts. Although people may differ in every which way — in their initial talents and aptitudes, interests, or temperaments — everyone can change and grow through application and experience." Programming may be a very new, intimidating thing for

Class.Score	Letter.Grade
92%-100%	A
90%-92%	A-
88%-90%	B+
82%-88%	B
80%-82%	B-
78%-80%	C+
70%-78%	C
60%-70%	D
0%-60%	F

you. That's okay! View this course as a way to grow and gain new skills which you can use to do incredible and important things!

- **Learn by doing:** A wise statistics instructor once compared watching someone else solve statistics problems to watching someone else practice shooting basketball free throws. You may learn a little by watching, but at some point you won't get any better until you try it yourself! The same can be said for programming. Reading a textbook and watching videos are a good *start*, but you'll have to actually *program* in order to get any better! This textbook was designed to be *interactive*, and I encourage you to “code along with the book” as you read.

## 2.2.4 Grading Scale

Grades will be assigned according to the following scale:

● **Feedback** Any feedback for this section? [Click here](#)

## 2.3 Running your first R Code

Enough of the boring stuff, let's run some R code! Normally you will run R on your computer, but since you may not have R installed yet, let's run some R code using a website first. As you run code, you'll see some of the things R can do. In a browser, navigate to [rdr.io/snippets](http://rdr.io/snippets), where you'll see a box that looks like this:

The box comes with some code entered already, but we want to use our own code instead, so delete all the text, from before `library(ggplot2)` to after `factor(cyl)`. In its place, type `1+1`, then click the big green “Run” button. You should see the `[1] 2` displayed below. So if you give R a math expression, it will evaluate it and give the result. Note: the “correct answer” to `1 + 1` is 2, but the output also displays `[1]`, which we won't explain until later, so you can ignore that for now.

Next, delete the code you just wrote and type (or copy/paste) the following, and run it:

```
factorial(10)
```

The result should be a very large number, which is equivalent to  $10!$ , that is,  $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ . This is an example of an R *function*, which we will discuss more later.

Aside from math, R can produce plots. Try copy/pasting the following code into the website:

```
x <- -10:10
plot(x, x^2)
```

```
library(ggplot2)

# Use stdout as per normal...
print("Hello, world!")

# Use plots...
plot(cars)

# Even ggplot!
qplot(wt, mpg, data = mtcars, colour = factor(cyl))
```

**Run (Ctrl-Enter)**

Figure 3: rdrv code entry box

You should see points in a scatter plot which follow a parabola. Here's a more complicated example, which you should copy/paste into the website and run:

```
library(ggplot2)
theme_set(theme_bw())
ggplot(mtcars, aes(y=mpg, fill=as.factor(cyl))) +
  geom_boxplot() +
  labs(title="Engine Fuel Efficiency vs. Number of Cylinders", y="MPG", fill="Cylinders") +
  theme(legend.position="bottom",
        axis.ticks.x = element_blank(),
        axis.text.x = element_blank())
```

R can be used to make many types of visualizations, which you will do more of later.

☀ **Bonus** This may be the first time you've seen R, so it's okay if you don't understand how to read this code. We'll talk more later about what each statement is doing, but for now, here is a brief description of some of the code above:

- `-10:10` This creates a sequence of numbers starting from -10 and ending at 10. That is, -10, -9, -8, ..., 8, 9, 10.
- `library` This is a function which loads an R *package*. R packages provide extra abilities to R.

● **Feedback** Any feedback for this section? [Click here](#)

## 2.4 What do you hope to get out of this course?

To close out this chapter, it would be healthy for you to reflect on what you'd like to get from this course. Take some time to think through each question below, and write down your answers. It is fine if your honest

answer is *I don't know*. In that case, try to come up with some possible answers that *might* be true.

### 💡 Reflect

1. Why are you taking this course?
2. If this course is required for your major, how do you think it is supposed to benefit you in your studies?
3. What types of data sets related to your field of study may require data analysis?
4. What skills do you hope to develop in this course, and how might they be applied in your major and career?

✍️ **Assessment** Submit your answers to the above reflection to Canvas. This will be your Progress Check 1.

Store your answers in a safe place, and refer to them periodically as you progress through the course. You may find that you aren't achieving your goals and that some adjustment to how you are approaching the course may be necessary. Or you may find that your goals have changed, which is fine! Just update your goals so that you have something to refer back to.

🗣️ **Feedback** Any feedback for this section? [Click here](#)

## 2.5 What is R?

What is R? This question can be answered several different ways. Here are a few of them:

### 2.5.1 R is a Programming Language

A programming language is a way of providing instructions to a computer. Some popular languages (in no particular order) are C, C++, Java, Python, PHP, Visual Basic, and Swift. Much like other types of languages, programming languages combine text and punctuation (syntax) to create statements which provide meaningful instructions (semantics) to be performed by a computer. These instructions are called "code". R code can be used to do many things, but primarily R was designed to easily work with data and produce graphics. The R language can be used to get a computer to do the following:

- Read and process a set of data in a file or database
- Use data to compute statistics and perform statistical tests
- Produce nice looking visualizations of data
- Save data for others to use. But this list is just the tip of the iceberg. As you will see, R can be used to do so much more! After the instructions are written, the R code is *run*, that is, the code is provided to the computer, and the computer performs the instructions to produce the desired results.



☀ **Bonus** Many other programming languages use different syntax for the same purpose.  
# comments out a line in R and python  
% comments out a line in matlab  
// comments out a line in C++ and javascript  
Similar to learning a foreign language, learning your first programming language will make it easier to understand other similar ones.

### 2.5.2 R is software

R can also be thought of as the software program which runs R code. In other words, if R code is the computer language, then the R software is what interprets the language and makes the computer follow the instructions laid out in the code. This is sometimes called “base R”.

### 2.5.3 R is Free

The R software is free, so anyone can download R, write R code, and run the R code in order to produce results on their computer.

### 2.5.4 R is Open Source

The R software, which runs R code, is also made up of a bunch of code called *source code*. In addition to being free, R is also *open source*, meaning that anyone can look at the source code and understand the “deep-down nuts-and-bolts” of how R works. In addition, anyone is able to *contribute* to R, in order to improve it and add new features to it.

💡 **Reflect** What are the advantages of open-source software? What are some potential downsides?  
Why do you think the creators of R decided to make it open source?

### 2.5.5 R is an ecosystem

Another way of thinking about R is to include not only the R language and the R software, but also the community of R users and programmers, and the various “add on” software they have created for R. These add on software are called “packages”.

### 2.5.6 R Packages

An R package is software written to extend the capabilities of base R. R packages are often written in R code, so anyone who knows how to write R code can also create R packages. The importance of packages cannot be understated. One of the reasons for the incredible popularity of R is the fact that members from the community can write new packages which enable R to do more. Sometimes packages are written to help folks in particular disciplines (e.g. psychology, geosciences, microbiology, education) do their jobs better. Other times, packages are written to extend the capability of R so that people from many disciplines can use them. R can be used to make web sites, interactive applications, dynamic reproducible reports, and even textbooks (like this one!).

The inclusion of R packages, combined with the free and open source nature of R software, has led to the development of an active, diverse, and supportive community of R users who can easily share their code, data, and results with one another.

☀ **Bonus** skimr is one example of a package. It provides a frictionless approach to summary statistics which conforms to the principle of least surprise, displaying summary statistics the user can skim quickly to understand their data.

### 2.5.7 R Interfaces

The R software can be run in many different places, including personal computers, remote servers, and websites (as you have seen!). R works on Windows, macOS, and Linux, and R can be run using a terminal or command line (if you know what those are), or using a graphical user interface (with buttons you can click and such). By far one of the most popular ways of using R is with RStudio, which is *also* free and open source software. For this course, you'll be using RStudio.

● **Feedback** Any feedback for this section? [Click here](#)

## 2.6 The R Community

We already mentioned that there is an active community of R users around the world, ranging from novice to expert level. Here is a partial list of venues where R users interact (aside from the official websites, none of these links should be considered an official endorsement):

1. R Project: The official website for R.
2. R Project Mailing Lists: Various email lists to stay informed on R related activities. The R-announce list is a good starting point, which will keep you updated on the latest releases of the R software.
3. Twitter #rstats: Many R Users are active on Twitter and you can find them.
4. Tidy Tuesday is a weekly online project that focuses on understanding how to summarize, arrange, and make meaningful charts with open source data. You can see the projects others have done by following #tidytuesday on twitter.
5. R-Ladies is a global group dedicated to promoting gender equality in the R community. They have an elaborate list of resources for learning and host educational and networking events.
6. R-Podcast: A periodic podcast with practical advice for using R, and the latest R news.
7. R-Bloggers: A blog website where authors can post examples of code, data analysis, and visualization.

### 2.6.1 Places to Get Help (If you're a student taking this class for credit)

Students taking the course for credit should seek help from these places, in order:

- Canvas Discussion boards
- Office Hours

I will not answer homework/quiz/textbook related questions via email.

## 2.6.2 Places to Get Help (anyone)

If you find yourself stuck, there are many options available to you, here are a few:

1. Stack Overflow is a message board where users can post questions about issues they're having. If you search for your error, there's likely already an answered question about it. If not, you can submit one with a reproducible example that the active community can help you with.
2. R Manuals: With so many R resources available on the internet, sometimes information gets "boiled down" or simplified for ease of communication. If you need the "official answer" to a question, these manuals are the place to go. Check out "An Introduction to R" for a good reference.

● **Feedback** Any feedback for this section? [Click here](#)

## 3 Installing R

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler

In the previous chapter, you ran R code on a website. The purpose of this chapter is to install R on your own computer, so that you can run R without needing access to the internet.

### 3.1 Computer Basics

If you're new to computers, this section will be important for you to get set up. We'll briefly introduce some computer concepts and discuss how they're relevant to R. If you understand the basics of operating systems, directory structures on your computer, and downloading/installing files, then you can probably skim this section, but be sure to pay attention to the R-specific information.

#### 3.1.1 Operating Systems

An operating system is a set of programs that allow you to interact with the computer, and the most popular operating systems are Windows, macOS, and Linux. R works on Windows, macOS, and several Linux-based operating systems, so if you have one of these operating systems, you'll be able to install and use R. At least, this is mostly true:

🚨 **Caution** Some versions of Windows that run on ARM processors cannot install R, and installing R on a Chromebook will likely be more complicated (see [here](#)). If you're in this situation, contact the instructor immediately.

R isn't designed to work on tablets or phones which run mobile/tablet operating systems (like iOS, iPadOS, Android, ChromeOS), so these are not an option for R.

### 3.1.2 Files & Directory Structures

A file is a collection of data stored on your computer's hard drive. Examples of files include:

- A music file
- A video
- A slide presentation
- A text document

Different types of files are often treated differently by your computer. For example, a music file is played with a music player program, a video can be viewed with a video player, and a slide presentation might be viewed with Powerpoint. Most operating systems know the type of a file by looking at the *extension*, which is at the very end of the file's name. Examples include “.mp3”, “.doc”, “.txt”, and “.ppt”. When using R, we can write scripts which contain R code, and *R Markdown* documents, which include human readable text and code. R scripts usually have either a “.R” or “.r” extension, and we'll also be using *R Markdown*, which use either a “.Rmd” or “.rmd” extension.

A *directory*, or *folder*, is a collection of files, and computers use directories to logically organize sets of files. When working with R, you may have to organize several different types of files, including R code, data files, and images. It will be important to stay organized when using R, and we will address this more later in the chapter.

With the increasing prevalence of the internet in everyday life, it's becoming less common for files to exist on your computer. When writing R code, you'll be working with files on your computer, not accessing them over the internet.

### 3.1.3 Downloads and Installations

To install R, you'll have to download a file from the internet which performs the installation. After you install R, you shouldn't have to download anything to run R. The specific steps to install R will be different depending on your operating system, and this will be addressed in the next section.

● **Feedback** Any feedback for this section? [Click here](#)

## 3.2 Install R & R Studio

Here's where you install R on your personal computer, but you'll actually be installing *two* separate programs. The *first* is the R programming language. The *second* is a separate program called RStudio, which will be the primary way in which you interact with R in this class, we will say more about this later.

### 3.2.1 Installing R

Installation will look slightly different depending on the operating system, but the major steps are the same.

- First, navigate to the CRAN Mirrors Site, which lists several locations from which R can be downloaded.
- Find a location near you (or not, this isn't critical) and click on the link to be brought to the mirror site.

From this point, this will change depending on your operating system.

### 3.2.1.1 Windows

- Click “Download R for Windows”, then click “base”.
- Finally, Click “Download R X.Y.Z for Windows”, where X, Y, and Z will be numbers. These numbers indicate which version of R you’ll be installing. As of the publishing of this book, R is on version 4.3.0.
- Your computer might prompt for the location on your computer that you would like to save the file. Select a location (reasonable options are your **Downloads** folder or the **Desktop**) and select “save”.
- When the download completes, find the downloaded file in the File Explorer and double click to run it. This will start the installation process.
- Follow the on screen prompts. For the most part you can click “next” and “install” as appropriate, and you don’t have to worry about changing any installation settings.
- Click “Finish” to complete the installation!

■ **Video** This video shows the installation process for Windows.  
[https://www.youtube.com/embed/7ZYn6q\\_pboE](https://www.youtube.com/embed/7ZYn6q_pboE)

### 3.2.1.2 macOS

- Click “Download R for macOS”
- Click “R-X.Y.Z.pkg”, where X, Y, and Z will be numbers. These numbers indicate which version of R you’ll be installing. As of the publishing of this book, R is on version 4.3.0.
- Your computer might prompt for the location on your computer that you would like to save the file. Select a location and select “save”.
- When the download completes, find the downloaded file in the Finder and double click to run it. This will start the installation process.
- Follow the on screen prompts. For the most part you can click “continue”, “agree”, “install”, as appropriate, and you don’t have to worry about changing any installation settings.
- Click “Close” to complete the installation!

**3.2.1.3 Linux** We will not provide details on installing R for Linux, because the process varies depending on your distribution, and because if you’re using Linux, chances are you’re more computer proficient than the average user. Suffice it to say, The first step is:

- Click “Download R for Linux”

And you can probably figure things out from there.

**3.2.1.4 Conclusion** You should now have R installed! Technically speaking, nothing further is required to work with R. You can open the RGui, and start coding immediately. However, for this course we will be using RStudio, which is a very popular program with an incredibly rich set of features, which will enhance your R programming experience.

## 3.2.2 Installing RStudio

- Navigate to the RStudio Download Page, and find the download file that matches your operating system.
- Click the link to download the installer, which starts with “RStudio-” or “rstudio-”.
- Your computer might prompt for the location on your computer that you would like to save the file. Select a location (reasonable options are your **Downloads** folder or the **Desktop**) and select “save”.

- When the download completes, find the downloaded file and double click to run it. This will start the installation process.

From this point, this will change depending on your operating system.

#### 3.2.2.1 Windows

- Follow the on screen prompts. For the most part you can click “next” and “install” as appropriate, and you don’t have to worry about changing any installation settings.
- You should now be able to open the start menu, open the RStudio folder, and click on the RStudio icon to open RStudio

🎥 **Video** This video shows the installation process for Windows.  
<https://youtu.be/XnqENDiEb3I>

#### 3.2.2.2 macOS

- In the window which opens, drag the RStudio icon into the “Applications” folder. You may need to enter your password (click the “Authenticate” button) in order to do so.
- You should now be able to navigate to the Applications folder in Finder, and click on the RStudio icon to open RStudio.

#### 3.2.2.3 Conclusion

☀️ **Bonus** Rstudio also offers a cloud service that allows you to work with R in your browser. We’ll use the desktop version but you can check out the interactive primers on the cloud site.

🗨️ **Feedback** Any feedback for this section? [Click here](#)

### 3.3 Successfull Installation

When you successfully install R and RStudio, you should now be able to program in R! Before moving further, you should become acquainted with the different parts of RStudio. To do so, watch the video below:

🎥 **Video** This video gives an introduction to some of the main pieces of RStudio.  
[https://youtu.be/w\\_3xp\\_3Sz6s](https://youtu.be/w_3xp_3Sz6s)

🗨️ **Feedback** Any feedback for this section? [Click here](#)

## 3.4 Running Code in RStudio

Now that you're somewhat familiar with RStudio, let's run the same code as we ran on the website, but this time let's run it in R.

### 3.4.1 The R Console:

In the *R console*, type `1+1` and press **enter**. The output in the console should look like the following:

```
> 1+1
[1] 2
> |
```

Figure 4: code in the console

Notice that the output 2 is displayed, and the cursor is on a blank line, waiting for more input. This is how coding in the console works.

### 3.4.2 R scripts

Now let's run the same code, but in an R script. If you haven't already, create a new R script by clicking on the **New File** icon, then selecting **R Script** like so:



Figure 5: Click this button to create a new file

In the script window which opens, type `1+1` and press **enter**. Notice how now, the code did *not* run? In a script, you are free to write R code on several lines before you run it. You can even save the script and load it later in order to run the code it contains. There are multiple ways to run R code in a script. To run a single line of code, do one of the following:

- Place the cursor on the desired line, hold the **<control>** key, and press **enter**. On macOS, hold **<command>** key and press **return** instead
- Place the cursor on the desired line and click the **Run** button that looks like this:



Figure 6: code in the console

To run multiple lines of code, do one of the following:

- Highlight all the code you'd like to run, hold the **<control>** key, and press **enter**. On macOS, hold the **<command>** key and press **return** instead.
- Highlight all the code you'd like to run, and click the **Run** button.

Run the `1+1` code using one of the methods above, and observe the output. Notice how the output is *still* in the console window, even though you ran the code in a script!

**⚠ Caution** Even though running R code from the console and an R script are done differently, they should produce the same results. Both are running R!

Now that you’ve run some code in the console and from an R script, let’s try some of the other code we ran previously.

### 3.4.3 Same Examples, On Your Computer!

In the *console*, type the command `factorial(10)`. Did you get the same result as you got on the website? Now type the following two lines in an R script and run them:

```
x <- -10:10
plot(x, x^2)
```

This code produces a plot, which should show up in the lower right corner in the “Plots” window. Finally, *copy* the following code, paste it into your script, and run it:

```
install.packages("ggplot2")
library(ggplot2)
theme_set(theme_bw())
ggplot(mtcars, aes(y=mpg, fill=as.factor(cyl))) +
  geom_boxplot() +
  labs(title="Engine Fuel Efficiency vs. Number of Cylinders", y="MPG", fill="Cylinders") +
  theme(legend.position="bottom",
        axis.ticks.x = element_blank(),
        axis.text.x = element_blank())
```

You’re now running R code on your computer!

**💡 Bonus** The above code block includes a command to install an R package! `ggplot2` is a very popular plotting package that can create sophisticated and (arguably) aesthetically pleasing graphs.

**💡 Reflect** Imagine you are practicing programming in R and your classmate tells you they heard about an interesting new R command which they’d like you to try out. Would you run the command in an R script, or the R console? How might your answer change if you wanted to keep a record of all the interesting R commands you found?

### 3.4.4 R Markdown

You’ve seen how to run R code in the R console, and from an R script, but there’s one more way to run R that we need to talk about: R Markdown.

R scripts are convenient because they can store multiple R commands in one file. R Markdown takes this idea further and stores code alongside human readable text. There is much that could be said about R Markdown, but for now, we’ll just stick with the basics.

To start, watch this video:



🎥 **Video** This video gives a basic introduction to R Markdown.  
<https://youtu.be/MhvipLohEfU>

As the video stated, there are three types of sections to an R Markdown document:

- Header
- Human readable text
- Code chunks

There's only one header, but there can be many blocks of human readable text and many code chunks.

🌟 **Bonus** See here for more things you can do with R Markdown.

✍️ **Assessment** As part of this class, you'll be filling in an R Markdown document as you complete the progress checks in the book (except for the first progress check box, which you completed already) Download the progress check 2 template into your **scripts** folder, and follow the instructions. That document should include all progress checks from Section 3.4 through (and including) Section 4.3 The next box should be the first code chunk you will include in the document!

🔗 **Progress Check** Run the command `8 / (2*(2+2))` and observe the output!

🎥 **Video** This video should help get you started with the Progress Check Assignments!  
<https://youtu.be/QLXB4kPngqM>

🗣️ **Feedback** Any feedback for this section? [Click here](#)

## 3.5 Workspace setup

Whenever you are programming in R, and especially for this class, it's important to stay organized. This section will give you some instructions and tips for how to organize material for this R course.

### 3.5.1 Recommended Settings

First of all, let's set some settings in RStudio. At the top of the R window, click **Tools**, then **Global Options**, and do the following:

1. On the left side of the window that pops up, make sure it's on the "General" tab
2. Find the "Workspace" section on the right, make the following changes:
  - *uncheck* "Restore .RData into workspace on startup"
  - Change the "Save workspace to .RData on exit" option to *never*
3. On the left side, select the "R Markdown" tab and make the following change:
  - Change the "Evaluate chunks in directory" option to *Project*.
4. (Optional) On the left side, select the "Appearance" tab and make the changes:
  - (Optional) Change the "Zoom:" setting to increase or decrease the interface text size to fit your screen best.
  - (Optional) Change the "Editor theme:" setting to find a color scheme that looks good to you.
5. Click "Apply", then "OK" at the bottom of the window.

Step 2 ensures that each time you open RStudio, there's no "memory" of anything you may have been doing in R previously. This is a good option for R beginners to avoid confusion and mistakes. Step 3 ensures that when you knit R Markdown documents, code chunks will use the project directory as the *working directory* (more on working directories below). Changing the zoom can also be done using the shortcuts `<control> <shift> +` (to increase size) and `<control> <shift> -` (to decrease size). On macOS, the commands are `<command> <shift> +` and `<command> <shift> -`.

### 3.5.2 Setting working directory

Every time R runs, it has a *working directory*, which is the folder where R "looks" when loading and saving files. In RStudio, the Files window contains the "More" menu, which has options to *set as working directory* or *go to working directory*. This will become more relevant when you start loading data and saving results later in the course. For this course, you'll be using an RStudio project, which automatically sets the working directory.

☀ **Bonus** See here for more information about working directories.

### 3.5.3 Create RStudio Project and directories for class

RStudio also has a feature called *projects*, which is a way of compartmentalizing your R code. This makes it easy to switch between different projects. For this class, you should set up a new project, so all of your project related files are in one place.

**3.5.3.1 Create RStudio Project** To create an RStudio project, follow these steps:

- Click on the "Project" button at the top right of the RStudio window and select "New Project".



Figure 7: Click this button to create a new project

- In the window that pops up, click on "New Directory" then "New Project".

- In the box after “Directory name”, type “RModule1”, which will be the name of the project.
- Then click the “Browse” button to select where to place the project.
- You are free to choose any location on your computer that makes sense to you. It might be most convenient to place it on your desktop for now.
- Click on “Create Project”.

You should now be *in* your newly created project. If you look at the Files window in the lower right pane of RStudio, you should see the files in your new project directory, which should only be one file, called “RModule1.rproj”. This file is the *project file*, which tells RStudio that this directory contains an R Project. When you’re working on this course, you should be working in this project. The easiest way to open up the project is to use your operating system’s file explorer and click on the project file. This will automatically set the working directory to the project directory.

**3.5.3.2 Create Directory Structure** To stay organized, you should also create the following folders inside your project directory

- scripts
- data\_raw
- data\_clean
- output

You can create these either using your operating system, or the “New Folder” command in the file window within RStudio.

### 3.5.3.3 Video

🎥 **Video** Check out this video to watch me set up a project and the new directories.  
<https://youtu.be/0saBBd6lQDI>

### 3.5.3.4 Set

## 3.5.4 Some useful commands you should know

As you program in R, you’ll end up creating many different R objects (more on this later), and sometimes you might want to clear all objects in your R environment. This will reduce the amount of memory that is taken up

```
rm(list=ls()) # Clear everything in your workspace
gc()          # perform garbage collection
```

	used (Mb)	gc trigger (Mb)	limit (Mb)	max used (Mb)
Ncells	830707 44.4	1417565 75.8	NA	1417565 75.8
Vcells	1552080 11.9	8388608 64.0	102400	2513560 19.2

You might also want to clear the R console, which you can do by placing your cursor in the R console and typing `<control> L` (careful! that’s a lowercase L).

☀ **Bonus** Here's a more complete list of RStudio shortcuts.

💡 **Reflect** Before moving on to the next section, take a note of all you've done so far.

1. Did your R installation go smoothly? If not, could you troubleshoot the errors or find help online?
2. Does using R remind you of other programs you have experience with?
3. What could be some reasons that using R code written by someone else might not work on your computer?

🗉 **Feedback** Any feedback for this section? [Click here](#)

## 4 R Programming Fundamentals

“Computers are good at following instructions, but not at reading your mind.” - Donald Knuth

In this chapter, we'll start to learn the “nuts and bolts” of R. Think of these things as the fundamental pieces that you need to understand in order to make R do more interesting and sophisticated things later.

### 4.1 Programming Preliminaries

💡 **Reflect**

1. Look at a sentence in a language you don't know, look carefully at the symbols, spacing and characters.
2. Recall learning a foreign language, how you had to learn the syntax and grammar rules.
3. Now think about English (or another language you know well) and think about the syntax and grammar rules that you take for granted.

All human languages rely on a set of rules called grammar, which describe how the language should be used to communicate. When two humans communicate with a language, they both must agree on the rules of that language.

R also has rules that must be followed in order for a human ( *you* ) to communicate with a computer, i.e. in order to tell the computer what to do. In human language, grammar is often fluid and evolving, and two people may have to adapt their use of the language in order to communicate. With R, the rules are fixed, and the computer “knows” them perfectly. It is up to you to learn the rules in order to make the computer do exactly what you want it to do.

Since any computer programming language will do exactly what you tell it to do, it's important to cover some of the basic rules of the R programming language before you can learn what it can do.

So let's get started:

### 4.1.1 R Commands

Like most programming languages, R consists of a set of *commands* which form the sequence of instructions which the computer completes. You can think of *commands* as the verbs of R, they are the actions the computer will take. Here is an example of a command, followed by the result.

```
print("hello, world!")
```

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

This command is telling R to **print** out a message. R code usually contains more than one command, and typically each command is put on a separate line. Here are multiple commands, each on a separate line:

```
print("The air is fine!")
print(1+1)
print(4 > 5)
```

```
[1] "The air is fine!"
[1] 2
[1] FALSE
```

The first command prints another message, the second command does some math then prints the result, and the third command evaluates whether the statement is true or false and prints the result. Generally, it's a good idea to put separate commands on separate lines, but you *can* put multiple commands on the same line, **as long as you separate them by a semicolon**. See this code for example:

```
x <- 1+1; print(x); print(x^2)
```


```
[1] 2
[1] 4
```

In this example, three commands are given on one line. The first command creates a new *variable* called `x`, the second command prints the value of `x`, and the third command prints the value of `x squared`. We see that the semicolon, `;`, serves as the command *termination*, because it tells R where one command ends and another begins. When a line contains a single command, no semicolon is necessary at the end, but including a semicolon doesn't have any effect either.

```
print("This line doesn't have a semicolon")
print("This line does have a semicolon");
```

```
[1] "This line doesn't have a semicolon"
[1] "This line does have a semicolon"
```

 **Caution** Including multiple semicolons (e.g. `print("hello");;`) does not work!

 **Bonus** You've just seen your first example of *assignment*. That is, we created a thing called `x`, and *assigned* to it the value of `1+1` using the *assignment operator*, `<-`. Formally `x` is called an object, but we'll talk more about objects and assignment later.

☀ **Bonus** So far, we’ve seen that you can place one command on one line, multiple commands on multiple lines, multiple commands on one line, so you may ask: can you place one command on multiple lines? The answer is *sometimes*, depending on the command, but we will not discuss this now.

⚠ **Caution** At this point, we’ve introduced several new types of R commands (assigning a variable, squaring a number, etc.), and we will talk more specifically about these later. The important part of this section is how R code is arranged into different *commands*.

Lastly, commands can be “grouped together” using left and right curly braces: { and }. Here’s an example:

```
{  
  print("here's some code that's all grouped together")  
  print(2^3 - 7)  
  w <- "hello"  
  print(w)  
}
```

```
[1] "here's some code that's all grouped together"  
[1] 1  
[1] "hello"
```

The above grouped code is indented so that it looks nice, but it doesn’t have to be:

```
{  
print("here's some code that's all grouped together")  
print(2^3 - 7)  
w <- "hello"  
print(w)  
}
```

```
[1] "here's some code that's all grouped together"  
[1] 1  
[1] "hello"
```

☀ **Bonus** Indenting is an example of coding *style*, which are formatting decisions which don’t affect the results of the code, but are meant to enhance readability. We’ll talk more about coding style later. In some programming languages, Python for example, white space matters. That is, code indents and other spaces change the way the code runs. In R, white space *does not* matter, so things like indents are used purely for readability.

What does it mean to “group” code? At this point there is no practical difference, each command gets executed whether or not it is grouped inside curly braces. However, code grouping will become very important later on, when we discuss *control flow* later.

☀ **Bonus** There are several helpful shortcuts that you can use in R. If you forget to put quotes around something, you can highlight and press the quote key and it will add quotes to both sides. This works with parentheses too.

You can also use tab completion with functions and defined variables. Tab completion allows you to use longer, more descriptive variable names without the additional typing time. This can save you a lot of time and reduce mistakes!

🔗 **Progress Check** In RStudio, open a new R script and type in all the R commands from this section, to verify that you get the same result. It's good practice!

### 4.1.2 Comments

When writing R code, you may wish to include notes which explain the code to your future self or to other humans. This can be done with *comments*, which are ignored by R when it is running the code. The “#” symbol initiates a comment.

Here's an example of some comments:

```
# Let's define y and z
y <- 8
z <- y + 5 # Adding 5 to y and assigning the result to z
## This is still a comment, even though we're using two #'s
```

Notice that it's possible for a line to contain only a comment, or for part of a line to be a comment. R decides which part of a line is a comment by looking for the first “#”, and everything after that will be treated as a comment and ignored.

⚠ **Caution** R ignores comments, but you should *not*! If you're reading code that someone else has written, it's likely that also paying attention to their comments will greatly help you to understand what their code is doing. It's also courteous to make good comments in your own code, if only because you may have to return to your *own* code in the future and re-learn what it is doing! In this book, we will use comments to help explain the R code that you will see.

### 4.1.3 Blank Lines

Blank lines in R are ignored, but they can be used to organize code and enhance readability:

```
print("The sky is blue")
# The blank line below here is ignored

print("The grass is green")
```

```
[1] "The sky is blue"
[1] "The grass is green"
```

#### 4.1.4 CaSe SeNsItIvItY

In R, variables, functions, and other objects (all of which we'll talk about later), have names. These names are case sensitive, so you must be careful when referencing an object by name. Here we create two variables and give them different values, notice how they are different from each other:

```
A <- 4
a <- 5

print(a)
print(A)

[1] 5
[1] 4
```

This may seem obvious, but case sensitivity applies to functions (which we'll talk about later) too. We've been using the `print` function a lot in the above examples, which begins with a lower case p. There is no `Print` function:

```
Print("testing")
```

#### 4.1.5 ?

One *very* nice thing in R is the documentation that accompanies it. Every function included in R (like `print`) has documentation that explains how that function works. To access the documentation, use a `?` followed by the name of the function, like so:

```
?print
```

🔗 **Progress Check** The output of the above code chunk is not shown, because the result of this code is best viewed in RStudio. Go to R Studio and type in `?print` and observe what happens!

#### 4.1.6 ??

If you don't remember the exact name of a function, or would like to search for general matches to a topic, then you can use `??`. For example, trying `?Print` produces an error, because there is not `Print` function (remember, R is case sensitive), so there's no documentation to go with it. However, the following should still work:

```
??Print
```

☀ **Bonus** Programmers have a sense of humor, too! Try running `????print` to see a small joke. Remember, comedic taste varies!

☀ **Bonus** This is a lot to remember. As you get more familiar with R, you'll begin to memorize basic functions - and Google is always there for the rest.



☀ **Bonus** Want to know more about R syntax? Try typing `?Syntax` in the R console (then press `Enter`).

⚠ **Caution** As we've seen, symbols and characters have specific meaning in R. You must be careful not to ignore things like semicolons, curly braces, parentheses, when reading R code. This takes practice!

Okay, now that we've covered some of the basics, it's time to start learning how to do useful things in R! The next few sections will describe the different types of data that R can handle.

🎥 **Video** This video discusses programming preliminaries.  
[https://youtu.be/EShV\\_T2P7sw](https://youtu.be/EShV_T2P7sw)

🗣 **Feedback** Any feedback for this section? [Click here](#)

## 4.2 Data Types

💡 **Reflect** Think of all the things you might be expected to remember. These different items can probably be categorized into different types of information, like phone numbers, passwords, birthdays, historical events, and math theorems for example. R was designed to handle different types of data as well, though the types are different from the examples just given.

R can store and manipulate different pieces of information, called data, and these data can be of several different types. Here are some examples of different types of data:

```
a <- 12.34      # a is a number
b <- "Hello"    # b is a string of characters
c <- TRUE       # c is a special type of data that is either true or false
```

R has special names for these examples, and there are other types of data as well. Below, we'll talk about each data type, one at a time.

☀ **Bonus** The term “data” is actually plural! A single piece of data is called a “datum”. So to refer to a set of data, you would say “these data”, and to refer to a single piece of data, you would say “this datum”.

### 4.2.1 Numeric

Many data exist as numbers, and R has a specific data type for storing those numbers, called the *numeric* data type. Here are some examples:

```
a <- -11
b <- 13.37
c <- 1/137
```

Note that integers, decimals, and fractions are all examples of numeric data in R. We can prove that these are all the same data type using the `class` function:

```
class(a)
```

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

```
class(b)
```

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

```
class(c)
```

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

⚠ **Caution** So far, we've defined the `a` object a few different times, which is allowed! Every time we define `a`, R *forgets* the old value. Therefore we should reuse object names with caution, because it can become difficult to remember what the *latest* value is! When we discuss *loops* later, however, we will use code to automatically change the value of an object several times in order to do useful things!

When you have numeric objects, you may want to perform math operations on them. R has a number of built in functions to deal with numeric data, here are some examples:

```
print(a + b) # Add two numeric values
print(b - c) # Subtract two numeric values
print(a * b) # Multiply two numeric values
print(a^3)   # Take the cube of a numeric value
```

```
[1] 2.37
[1] 13.3627
[1] -147.07
[1] -1331
```

When performing math on numeric objects, R will obey order of operations, so the following two examples will give different results:


```
a + b * c # R will perform the multiplication before the addition
```

```
[1] -10.90241
```

```
(a + b) * c # R will perform the addition first, then the multiplication
```

```
[1] 0.01729927
```

Notice that we've added extra spaces in the code to help you understand what's going on. This is another example of code *style*, which we'll talk more about later.

 **Caution** Wait a second, we didn't use the `print` function just now, but R still displayed the results of the calculations! What is going on? This behavior is peculiar to something called R Markdown, which is what we used to create this book (yes, this book was created using R! Pretty cool, huh?). If the *last* command given in a code block produces a result, and you don't assign that result to anything (using `<-`), then R will print out that result. This means we don't always have to use the `print` function when we want to display R output.


Notice all the decimal points? R can be very precise when performing computations. However, viewing all of the digits stored by R can be distracting and hard to read. You can show just some of the digits by using the `round` function:

```
a <- 0.123456
```

```
round(a, 3)
```

```
[1] 0.123
```

It also turns out that R stores *more* digits than what it shows when it prints, though we won't go into detail on that now.

 **Video** This video discusses numerics.  
<https://youtu.be/juscNzIrmJQ>

#### 4.2.2 Integer

In general, numeric data in R are treated as if they can be any decimal number (technically, they are a *double precision* number, if you know what that means; if not, it's not important right now). However, there is a way to specify that a specific numeric object is an integer, by placing an "L" at the end of it, like so:

```
x <- 20 # x will be a numeric object  
y <- 20L # y will be an integer object
```

```
class(x)
```

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

```
class(y)
```

```
[1] "integer"
```

Integers take half of the space in a computer's memory or hard drive, so if you are working with or storing a lot of numbers which are integers, it might make sense to declare them as integer type in R. This will make more sense when we discuss vectors later.

🎥 **Video** This video discusses integers.  
<https://youtu.be/rNkEAPsipCk>

### 4.2.3 Character

Not all data are numbers! R also has the capability to store strings of characters, and this is the aptly named *character* type (or sometimes called a *character string* or just *string*). Here are some examples:

```
d <- "Hello"          # This string is defined with *double* quotes
e <- 'how are you?'    # This string is defined with *single* quotes!
print(d)
print(e)
```

```
[1] "Hello"
[1] "how are you?"
```

Notice how we can define character strings using single quotes or double quotes, as long as we are consistent. So this is not valid:

```
# Note the mismatched single/double quotes:
f <- "this does not work"
```

So, make sure you are consistent. However, you may see another problem with this: some strings contain quotes *in them*, like this:

```
g <- 'This won't work'
```

Since single quotes are being used to define the string, they can't be used in the string itself, because R will “think” the string is ending at the second '. One option is to change the defining quotes to be double quotes, then the single quote will be safely included in the string:

```
g <- "I'm happy that this works!"
print(g)
```

```
[1] "I'm happy that this works!"
```

Another option is to use a backslash when using quotes inside the string, so that R “knows” the quote is part of the string and not ending the definition of the string:

```
g <- 'I\'ve found another way that works!'
print(g)
```

```
[1] "I've found another way that works!"
```

Notice that when we define `g` we place a `\'` anywhere in the string where we want a `'` to be, but when printed out, we see that R has interpreted it as just a `'`. Notice also that we didn't have to change the defining quotes to be double quotes in this case. The backslash is called the *escape character*, and it signifies that what follows it should be interpreted *literally* by R, and any special meaning should be ignored.

☀ **Bonus** Since backslash also has special meaning itself, if you want a backslash in your string, you need to use another backslash, which functions as an escape character, like so: `g <- "here is a backslash: \\\\"`. You will see both backslashes when using the `print` function (which is meant for any data type), but if you use the special `cat` function (which is meant for character types specifically), all escape characters will be “processed”, and you will see just a single backslash.

Try the same thing with the newline character, `\\n`!

To see a list of special characters, try typing `?Quotes` into the R console

Here is an important string to know about:

```
h <- "" # This string is empty!
```

`h` is a character string with no characters, called an *empty string*.

You can perform math on numeric data, so what can you do with strings? The answer is, quite alot, using some functions that R provides. Here are some of them:

```
nchar(g) # This prints out the number of characters in a string
```

```
[1] 34
```

```
substr(g, 6, 10) # This extracts just part of a string, using the start and stop positions you provide
```

```
[1] "found"
```

```
strsplit(g, " ") # This splits the string up using a specified "delimiter" string, a single space in t
```

```
[[1]]
```

```
[1] "I've"      "found"     "another"   "way"       "that"      "works!"
```

☀ **Bonus** When you split a string, this produces a *list* containing a *vector* of character strings. This is an example of how data can be organized in a structured way. We'll talk more about so called *data structures* in the next section.

```
paste("hello", "world") # This combines multiple strings together into one string!
```

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

💡 **Reflect** Remember that you can learn more about a function using `?`. Type `?paste` into R and read the documentation carefully. Can you determine what the “sep” argument does? What do you think would happen if we ran the code `print("hello", "world", sep="-")`?

There are other ways of manipulating strings, but we’ll return to this later.

🎥 **Video** This video discusses characters.  
[https://youtu.be/1JgmnuLM\\_4g](https://youtu.be/1JgmnuLM_4g)

#### 4.2.4 Logical

Numeric objects can be any number, character objects can be any string of characters, but logical objects can only be two different values: True or False.

Logical data types are also known as “boolean” data types. Here we define some Logical objects:

```
a <- TRUE  
b <- FALSE  
c <- T  
d <- F
```

```
print(a)
```

```
[1] TRUE
```

```
print(b)
```

```
[1] FALSE
```

```
print(c)
```

```
[1] TRUE
```

```
print(d)
```

```
[1] FALSE
```

So you can see that we can define a logical object using the full name or just the first letter. Here’s how to get the “opposite” of a logical object

```
!a
```

```
[1] FALSE
```

Logical data are the simplest type, but there are actually some clever things you can do with them. You can test whether simple mathematical expressions are true or false.

```
# Create x and y
x <- 3
y <- 4
# Check: is x less than y? (should give TRUE)
x < y
```

```
[1] TRUE
```

The third command is a way to check if the value of `x` is less than the value of `y`. The result of this comparison is a logical, in this case, `TRUE`. Here are other ways of making comparisons:

```
x <= y # Check if x is less or equal to y
```

```
[1] TRUE
```

```
x == y # Check if x is equal to y (note how you need two equals signs)
```

```
[1] FALSE
```

```
x >= y # Check if x is greater or equal to y
```

```
[1] FALSE
```


```
x > y # Check if x is greater than y
```

```
[1] FALSE
```

Comparisons can be made using strings as well:

```
x <- "Hello"
y <- "hello"
x == y
```

```
[1] FALSE
```

 **Caution** Remember that R is case sensitive, and two strings must be *exactly* the same to be considered equal.

Of course any object (like `x`) will be equal to itself:

```
x == x
```

```
[1] TRUE
```

Surprisingly, logicals can be treated as numerics, where `TRUE` is treated as 1 and `FALSE` is treated as 0. Here are some examples:

```
TRUE + TRUE # TRUE is treated as 1
```

```
[1] 2
```

```
FALSE * 7 # FALSE is treated as 0
```

```
[1] 0
```

```
(2 < 3) + (1 == 2) # What's going on here?
```

```
[1] 1
```

The last example deserves some thought. Start with each expression in parentheses, and decide whether it will evaluate to true or false. Then remember how logicals are treated as numbers, and determine what happens when you add them together.

☀ **Bonus** Numeric, integer, character, and logical data types are probably the most important data types to know in R, but there are others that were not covered here. These include:

- complex
- factor
- raw

At least one of these (factor) will be covered later, but you can find more information about the other types [here](#)


📖 **Progress Check** In the R console, type the following R commands and observe the result

```
x <- "5"  
y <- 5  
z <- (x == y)
```

1. What data type is x? (check with R using the `class` function)
2. What data type is y?
3. What data type is z?
4. What is the value of z, and why does this make sense?

Now that we've discussed different types of data, we'll now see how they can be structured together in meaningful ways.




 **Caution** What about dates? R actually has three built-in date classes. This can be confusing at first, but packages like lubridate make it easy to work with dates in R.

 **Video** This video discusses logicals.  
<https://youtu.be/GH9AZcexokU>

 **Feedback** Any feedback for this section? [Click here](#)

## 4.3 Data Structures

 **Reflect** Imagine a grocery list, shopping list, or to-do list. That list consists of a set of items in a specified order, and the list also has a length. Why do you think it's useful to organize these items into a list, rather than in some other fashion? Can you think of why it might be useful to store data in a list?

Often, you will need to work with many related data, for example: - A sequence of measurements through time - A grid of values - A set of phone numbers

In these circumstances, it would make sense to organize the data into a *data structure*. R provides multiple data structures, each of which are appropriate in various situations. By far the most popular data structure in R is the *data frame*, but in order to talk about data frames, we must talk about some simpler data structures first.

### 4.3.1 Vectors

A vector is just an ordered set of elements (in other words, *data*), all of which have the same data type. Vectors can be created for the logical, numeric (double or integer), or character data types. Here's an example of a vector:

```
x <- c(1, 2, 3) # this is a vector of numeric types
print(x)
```

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

Note that to create a vector, we use the `c` function, where `c` stands for *combine*. This makes sense, because we are combining three numeric objects into a numeric vector. We may determine the length of any atomic vector like so:

```
length(x)
```

```
[1] 3
```

The `class` function will tell us what type of data is stored in a vector (which makes sense, because all elements of the vector have the same data type).

```
class(x)
```

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

Here's how to create logical or numeric vectors:

```
y <- c(TRUE, TRUE, FALSE, TRUE)
z <- c("to", "be", "or", "not", "to", "be")
```

```
class(y)
```

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

```
length(y)
```

```
[1] 4
```

```
class(z)
```

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

```
length(z)
```

```
[1] 6
```

**💡 Reflect** The above statement states that all elements of a vector must have the same data type, so what do you think will happen if you try to create a vector using elements from different data types? Here are some possibilities, can you think of another one?

- R will produce an error
- R will combine the elements somehow, but the result won't be a vector
- Something else?

Whatever happens, humans were behind the decision of how R should behave in this situation. If you were in charge of making this decision, what would make the most sense?

**🔗 Progress Check** Let's try to create a vector of mixed type and see what happens. Run the following commands in R and think about the output:

```
m <- c(TRUE, "Hello", 5)
```

```
class(m)
```

```
print(m)
```

What changes did R make when creating the vector?

What's happening in the above code is an example of *type conversion*, which we will talk more about later. For now, remember that every element in an R vector is the same type.

You can create *empty* vectors as placeholders, by indicating the data type and how many elements there are:

```
empty <- numeric(10)  # this creates a numeric vector of length 10
```

💡 **Reflect** This is the first instance of us using a name which is longer than a single character! This new vector is called *empty*.

Let's print the contents of the vector:

```
print(empty)
```

```
[1] 0 0 0 0 0 0 0 0 0 0
```

Even though we didn't tell R what data to put in the vector, it put a 0 in each element. This is the *default* value for a new vector.

Here's how you can create new vectors of other types:

```
empty_int <- integer(45)  # create integer vector with 45 elements
empty_cha <- character(2)  # create character vector with 2 elements
empty_log <- logical(1000) # create logical vector with 1000 elements!!
```

🔗 **Progress Check** We saw that the default value for a numeric vector is 0. Use the code above to create empty integer, character, and logical vectors, then print them out to see what default values R has given to each element. Do these make sense?

What happens if we create a vector of length 1? It turns out this is the same as just creating a single instance of that data type. Observe how the following are the same.

```
a <- numeric(1)  # create vector of length 1 (default value is 0, right?)
b <- 0           # create single numeric with value 0
a == b          # compare a and b to see if they are the same.
```

```
[1] TRUE
```

☀️ **Bonus** It turns out, you *can* create a vector of length 0, which contains 0 elements. This may sound odd, but can happen sometimes! However, you *cannot* create a vector of negative length (e.g. `logical(-1)` won't work), or a fractional length (e.g. `character(12.7)` won't work).

**4.3.1.1 Accessing and Changing Elements** After you've created a vector, how do you put your data in them? Here's how you can change the value of a specific element:

```
a <- c(1, 2, 3) # create numeric vector of length 3
a[2] <- 4       # change the value of the second element of a to 4
a              # print the result
```

```
[1] 1 4 3
```

See how the second element of `a` has changed? So you can access a specific element using square brackets: `[` and `]`. In fact, if you want to know the value of the third element (without changing anything), just use:

```
a[3] # access the third element
```

```
[1] 3
```

🔗 **Progress Check** What do you think will be the result of the following code (hint: the result will either be `TRUE` or `FALSE`)?

```
vec <- c(4, 5, 6) # create a vector
vec[3] == 6      # Remember what == does?
```

Once you make a guess, try it in R and see if you were correct.

📺 **Video** This video gives an introduction to vectors.  
[https://youtu.be/-BIN6\\_ZMpKE](https://youtu.be/-BIN6_ZMpKE)

**4.3.1.2 Working with vectors** You can do many things with vectors that you can with single instances of each data type. Recall, you can add a number to a numeric object:

```
a <- 3 # create a numeric object
a + 4  # add a number to the object.
```

```
[1] 7
```

The same thing is possible with numeric vectors:

```
a <- c(1, 2, 3) # create a numeric vector
a + 4           # add a number to EACH ELEMENT of the vector!
```

```
[1] 5 6 7
```

This type of behavior is called *elementwise* behavior. That is, the operation is performed on each element separately. Here are some other elementwise operations:

```
a - 3
```

```
[1] -2 -1 0
```

```
a * 1.5
```

```
[1] 1.5 3.0 4.5
```

```
a ^ 2
```

```
[1] 1 4 9
```

```
a == 2
```

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

R has some functions which *summarize* the values in a vector. One such function is the `sum` function, which adds the values of each element in the vector:

```
print(a) # Print the elements of a as a reminder
sum(a)   # Add all the elements of a together.
```

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

```
[1] 6
```

☀ **Bonus** Other examples of summary functions include `max`, `min`, `mean`, and `sd`. We'll talk about these and other summary functions later.

Some operations work on *two vectors*, as long as they are the same length:

```
b <- c(1, 0, 1)
a + b
```

```
[1] 2 2 4
```

```
b * a
```

```
[1] 1 0 3
```

```
a ^ b
```

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

You can even compare two vectors, and the result will be a logical vector:

```
z <- a > b # Compare a and b, element by element, assign the result to z
z         # Print the value of z
```

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

The first logical value is the result of `a[1] < b[1]`, the second logical value is the result of `a[2] < b[2]`, etc. what operations can we perform on character vectors? Here are some examples:

```
z == TRUE # Which elements are TRUE?
```

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

This just produces `z` again (Do you see why?). Here's how to get the logical “opposite” of `z`:

```
z == FALSE
```

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

Or, as we saw before, we can use `!`, which operates on each element of `z`:

```
!z
```

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

Remember how logical objects can be treated as numeric objects (either a 0 or 1)? If we use this with the `sum` function to determine how many elements are `TRUE`:

```
sum(z)
```

```
[1] 2
```

Here's another example of using the `sum` function on a logical vector:

```
sum(a == b) # How many elements do a and b have in common?
```

```
[1] 1
```

So there is one element that both `a` and `b` share.

☀ **Bonus** Logical vectors can also be used to access all elements of a vector for which a certain condition is true. We'll see how to do this later on.

Let's create some character vectors and explore a few things we can do with them:

```
a <- c("I", "have", "to", "have", "a", "donkey")
b <- c("You", "want", "to", "sell", "a", "donkey")
```

First, we can do elementwise comparison (assuming equal length), just as we did for numeric vectors:

```
a == b
```

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

To search for specific character strings in a character vector, you can use the `grep` function:

```
grep("have", a) # Search the vector a for the phrase "have"
```

```
[1] 2 4
```

This result shows that the phrase “have” occurs in elements 2 and 4 of **a**! What if we search for a phrase that doesn’t occur?

```
grep("raddish", a)
```

```
integer(0)
```

The result is an integer vector of length 0, meaning there are no elements that match the phrase!

🎥 **Video** This video continues the discussion of vectors.  
<https://youtu.be/NgmVhLpuM5k>

**4.3.1.3 Vectors of different types** What if we try to perform operations between vectors of different types? This will work in some cases, but not others. Here are a few examples:

```
a <- c(1, 2, 3)
b <- c("I", "am", "sam")
c <- c(TRUE, TRUE, FALSE)
```

```
a + b # Can you add a numeric vector to a character vector?
```

```
a + c # Can you add a numeric vector to a logical vector?
```

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

💡 **Reflect** We see that you can’t add a numeric vector to a character vector, but you *can* add a numeric vector to a logical vector. Why is this?

🔗 **Progress Check** Predict whether the following are possible:

- Can you multiply a character vector with a numeric vector?
- Can you multiply a logical vector with a numeric vector?

Check whether you are correct by creating some vectors in R and attempting to multiply them together. Can you make sense of the answer? If you run into errors, you can include `error=TRUE` in your code chunk options like this:

```
```{r, error=TRUE}
```

This will allow RStudio to still knit the document, even though the code block generated errors.

**4.3.1.4 Special Numeric Vectors** There are a few special ways of creating a numeric vector which can be very useful, so we'll mention them here. The first way creates a sequence of all integers between a starting and ending point:


```
d <- 1:5 # Create sequence starting at 1 and ending at 5
d
```


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

Here's a longer example:

```
d <- 1:100 # Create sequence starting at 1 and ending at 100
d
```

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

 **Caution** In this example, the R output can't be shown on a single line, so it must be placed on multiple lines. Notice that each line has a different number in brackets: [1], [19], [37] etc. This number indicates which element of the vector is the start of that line. So we finally have an explanation for the [1] which is displayed with all R output. It's simply indicating that this is the *first element* of the output. This also reflects the fact stated earlier that any R object can be considered a vector of length 1!

 **Bonus** When you're working with large data sets, it's often helpful to see just the first few results instead of printing the entire thing. You can use `head()` to print the first six rows.

Another way to create a numeric vector is using the `seq` function, which allows you to specify the *interval* between each vector element. For example:

```
e <- seq(2, 100, 2)
e
```

```
[1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
[20] 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76
[39] 78 80 82 84 86 88 90 92 94 96 98 100
```

Or you can also specify how long you want the vector to be, and `seq` will determine the appropriate interval to make the elements evenly spaced.

```
seq(1, 10, length.out=3)
```



```
[1] 1.0 5.5 10.0
```

```
seq(1, 10, length.out=5)
```

```
[1] 1.00 3.25 5.50 7.75 10.00
```

**4.3.1.5 Another Data Type: Factor** In the previous section, we avoided talking about the *factor* data type, because we need the concept of vectors to appreciate their purpose, but now we are equipped to talk about them. Consider the following example of a *character vector*:

```
cha_vec <- c("cheese", "crackers", "cheese", "crackers", "cheese", "crackers", "cheese")
```

There are seven elements in this vector (`length(cha_vec)` is 7), but there are only *two unique* elements, “cheese” and “crackers”. Imagine having to write down this vector on a piece of paper, and the space it would take. Now imagine writing down instead:

1, 2, 1, 2, 1, 2, 1

1 = “cheese”

2 = “crackers”

This second method writes down numbers instead of character strings, but also keeps a record of which numbers correspond to which character strings. The total amount of space taken up on the piece of paper is smaller for the second method, and the amount of space saved would be even larger if the character vector were longer and had more repeated elements.

This is the essence of what a *factor* data type is: A character vector stored more efficiently on the computer. For a factor vector, R stores an integer vector (which often takes less space than a character vector), and also maintains a “lookup table” which keeps track of which integers correspond with which character strings.

To illustrate, let’s create a factor variable:

```
# Create a new factor variable from our existing character vector:
fac_vec <- factor(cha_vec)
```

Notice how we started with a character vector and used the `factor` function to create a factor from it. If we print the new vector,

```
fac_vec
```

```
[1] cheese  crackers cheese  crackers cheese  crackers cheese
Levels: cheese crackers
```

it displays the elements as we would expect, but also includes another line of output giving **Levels**. This shows that there are only two unique character strings, which are called *factor levels*. Since R is using integers “behind the scenes” to store the vector, we can see those integers by using the `as.integer` function:

```
as.integer(fac_vec)
```

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

This is another example of *type conversion*, which we will discuss soon.

⚠ **Caution** In some situations, numbers may get treated as characters, like so:

```
x <- c("4", "5", "6")
```

This may pose an issue if this character vector gets converted to a factor, because the “behind the scenes” integers may not agree with the `Levels`, which represent the original data. This can easily happen when reading in data from a file on your computer, if you’re not careful. We’ll talk more about this later.

There are a few neat things you can do with factor vectors. By changing the levels, you can quickly change all occurrences of a string at once. For example:

```
print(fac_vec)
levels(fac_vec) <- c("peas", "carrots") # Change the levels of fac_vec
fac_vec
```

```
[1] cheese  crackers cheese  crackers cheese  crackers cheese
Levels: cheese crackers
[1] peas    carrots peas    carrots peas    carrots peas
Levels: peas carrots
```

There is more to be said about factors, but this is all we will explore at this point.

☀ **Bonus** In newer versions of R, all strings are *treated like factors* behind the scenes, meaning there’s really no difference between factor and character types in terms of how much space they take up in the computer’s memory. However, R still treats the two types differently, so it’s important to remember that they are different!

📺 **Video** This video discusses coercion, sequences, and factors.  
<https://youtu.be/iusiO1dRQdY>

**4.3.1.6 Combining Vectors** Given two vectors, it’s easy to combine them into one vector:

```
a <- c(1, 2, 3)
b <- c(4, 5, 6, 7)
c(a, b) # Combine vectors a and b
```

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

The combine function (`c`) is smart enough to recognize that `a` and `b` are vectors, and performs *concatenation* to create the resultant longer vector. You can also use the combine function to add a single element to the end of a vector:

```
a <- c("CEO", "CFO") # Initialize
a <- c(a, "CTO")     # Redefine a by combining a with a new element
a
```

```
[1] "CEO" "CFO" "CTO"
```

⚠ **Caution** In R, there may sometimes be more than one way to do the same thing, and one of the ways might be much faster or take much less computer memory to do. In other words, two sets of R commands can be *correct*, but one may *perform better* than the other. Writing “performant” (high performance) code is an advanced topic that we will not discuss much in this introductory course. You’ve just seen one way to add an element to the end of a vector, but if you do this *a lot* (perhaps in a for loop, which we’ll talk about later), it can be *very slow*. In this situation you’re better off creating the whole vector at once and updating each element as needed.

What if you try to combine vectors of different types?

```
a <- c(1, 2, 3)
b <- c("four", "five")
c(a, b)
```

```
[1] "1"      "2"      "3"      "four" "five"
```

Again, we see that the `c` function has converted all elements to be character strings, and the resultant vector is a character vector. Since we’ve seen type conversion arise a few times now, it’s appropriate to talk more explicitly about how it works. We’ll do that in the next section.

**4.3.1.7 Type Conversion** There may be times when you’d like to convert from one type of data into another. An example would be the character string “1”, which R *does not* view as a number. Therefore, the following does not work:

```
"1" + "2" # R can't add two character strings
```

To remedy issues like this, R provides functions in order to convert from one data type into another: - `as.character`: converts to character - `as.numeric`: converts to numeric - `as.logical`: converts to logical - `as.factor`: converts to factor

Using these functions, R will “do its best” to convert whatever you start with into the desired data type, but it’s not always possible to make the conversion. Below are a few examples which do and don’t work well.

Converting from a numeric to a character vector is always possible:

```
x <- c(3, 2, 1)
```

```
y <- as.character(x) # Here's how to convert to a character vector
print(x)
print(y)
```

```
[1] 3 2 1
[1] "3" "2" "1"
```

However, converting from a character vector to a numeric only works if the characters represent numbers. Any element that won’t convert will be given

```
w <- c("1", "12.3", "-5", "22") # This character vector can be converted to numeric
as.numeric(w)
```

```
[1] 1.0 12.3 -5.0 22.0
```

```
v <- c("frank", "went", "to", "mars") # This character vector can't be converted to numeric
as.numeric(v)
```

Warning: NAs introduced by coercion

```
[1] NA NA NA NA
```

None of the elements can be converted into a number, so R prints a warning message, and the result is an NA in each element, which stands for “not available”. NA indicates that a value is missing, and can arise in many different ways, which we will not explain here.

☀ **Bonus** NA values have interesting behavior in R. Generally, anything that “touches” an NA becomes an NA. You can try out these commands for yourself to see what happens:

```
NA * 0
NA - NA
c(NA, 1, 2)
```

If only part of a vector can be converted, then the result will contain some converted values and some NA's:

```
u <- c("1.2", "chicken", "33")
as.numeric(u)
```

Warning: NAs introduced by coercion

```
[1] 1.2 NA 33.0
```

What other conversions are possible? Character vectors can also be converted into logical:

```
s <- c("TRUE", "FALSE", "T", "F", "cat") # All but the last element can be converted to logical
as.logical(s)
```

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

Based on the examples we've seen before, it should make sense that numeric vectors containing 0 or 1 can also be converted into a logical vector:

```
as.logical(c(1, 0, 1, 0)) # Here we create the vector and convert it in the same line
```

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

🔗 **Progress Check** Logical vectors can also be converted into character or numeric vectors. Based on what you know, make a prediction about what the following commands will produce:  
`as.numeric(c(T, F, F, T))` `as.character(c(T, F, F, T))`  
Check your predictions by running the commands in R.

Remember that “solo” objects are just vectors of length 1, so any of these type conversions should work on a single object as well, like so:

```
as.numeric("99")
```

```
[1] 99
```

Along with the conversion functions `as....`, there are companion functions which simply check whether a vector is of a certain type:

- `is.character`: checks if character
- `is.numeric`: checks if numeric
- `is.logical`: checks if logical
- `is.factor`: checks if factor

Here are some examples:

```
a <- c("1", "2", "3")  
is.character(a)
```

```
[1] TRUE
```

```
is.numeric(a)
```

```
[1] FALSE
```

```
a <- as.numeric(a)  
is.character(a)
```

```
[1] FALSE
```

```
is.numeric(a)
```

```
[1] TRUE
```

☀ **Bonus** As we’ve seen, type conversion is sometimes performed automatically, specifically when using the combine function (`c`). To understand more about this, try typing `?c` to bring up the documentation, and have a look at the “Details” section.

📺 **Video** This video finishes the discussion of vectors.  
<https://youtu.be/XKdZzHBRO9o>

### 4.3.2 Matrices

Not all data can be arranged as an ordered set of elements, so R has other data structures besides vectors. Another data type is the *matrix*, which can be thought of as a *grid* of numbers. Here's an example of creating a grid:

```
data <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
A <- matrix(data, 3, 3)
A
```

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

Here we've made a matrix with three rows and columns, by first creating a vector called **data**, and using the **matrix** function and giving it the data, the number of rows, and the number of columns.

 **Caution** Notice that R fills the matrix one *column* at a time, from left to right.

Here's how you access the data within a matrix:

```
A[1,1] # Get the first element of the first row
```

```
[1] 1
```

```
A[2,3] # Get the third element of the second row
```


```
[1] 8
```

```
A[1,] # Get the entire first row
```

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

```
A[,3] # Get the entire third column
```

```
[1] 7 8 9
```

 **Caution** Just like with vectors, square brackets must be used to access the elements of a matrix. *Don't* use parentheses like this: `A(1,2)`.

```
diag(A) # Get the diagonal elements of A
```

```
[1] 1 5 9
```

You can get the shape of a matrix with the `dim` function:

```
dim(A) # How many rows & columns does A have?
```

```
[1] 3 3
```

Which gives an integer vector telling us A has three rows and three columns.

**^r Progress Check** In R, create the matrix A above, and write code to compute the first element of the second row times the third element of the third row.

You can do some simple math with matrices, like this:

```
A + 1 # Add a number to each element of the matrix
```

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

```
A * 2 # Multiply each element by a number
```

```
      [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
```

```
A ^ 2 # Square each element
```

```
      [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

If you've worked with matrices in a math class, you may have talked about some of the following operations: Here we can find the transpose of a matrix (the rows become columns and the columns become rows):

```
t(A) # Find the transpose
```

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

```
# Find the trace:  
sum(diag(A)) # Get the diagonal elements of A, then sum them
```

```
[1] 15
```

Here are some things you can do with two matrices:

```
B <- matrix(1, 3, 3) # Create a 3x3 matrix of all 1's (notice how we only need one 1?)  
A + B # Add two matrices together
```

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

```
A * B # Multiply the elements of A and B together
```

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

```
A %*% B # Perform matrix multiplication between A and B
```

```
      [,1] [,2] [,3]  
[1,]   12   12   12  
[2,]   15   15   15  
[3,]   18   18   18
```

Notice the difference between the last two examples? Just using `*` multiplies the matching elements of A and B together, while the new operator `%*%` performs *matrix multiplication*, like you may have seen in a linear algebra class.

🔗 **Progress Check** In R, perform matrix multiplication between A and the transpose of A.

If two matrices don't have the same shape, you won't be able to add their elements together:

```
C <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), 3, 4)  
A * C
```

The error message: `non-conformable arrays` tells us that A and C have different shapes, so it's impossible to multiply their matching elements together. But you can still perform matrix multiplication between them:

```
A %*% C
```



```
      [,1] [,2] [,3] [,4]
[1,]   30   66  102  138
[2,]   36   81  126  171
[3,]   42   96  150  204
```

⚠ **Caution** Any data type (numeric, character, etc.) can be represented as a *vector*, but matrices only work with numeric types.

☀ **Bonus** A matrix is just a special case of a data structure called an *array*. Matrices have two dimensions (row and column), and arrays can have any number of dimensions (1, 2, 3, 4, 5, etc.). We won't discuss arrays in this course much.

🔗 **Progress Check** Try running the following code in R, which should produce a warning message:  
`data <- c(4.5, 6.1, 3.3, 2.0); A <- matrix(data, 2, 3);`  
Read the warning message and the code carefully, and see if you can figure out the problem. What change would you make to the above code so that it runs?

Remember everything inside a vector must have the *same data type*. Here we've seen that matrices *all have to be numeric data types*. Wouldn't it be nice if there were a way to store objects of different types (without doing type conversion)? This is what lists can do!

☀ **Bonus** It turns out, matrices can work with non-numeric types as well! But like vectors, mixed type matrices are not allowed. For this, you'll have to use a dataframe, as we discuss later.

🎥 **Video** This video gives an introduction to Matrices.  
<https://youtu.be/hknL1EbrIB4>

### 4.3.3 Lists

A List is an ordered set of *components*. This may sound similar to a vector, but the important difference is that with lists there is no requirement that the components have the same data type. Here is an example of a list:

```
A <- list(42, "chicken", TRUE)
A
```

```
[[1]]
[1] 42

[[2]]
[1] "chicken"

[[3]]
[1] TRUE
```

Here we see each component of the list printed in order, with `[[1]]`, `[[2]]`, and `[[3]]` indicating the first, second, and third components. To access just one of the components, use double square brackets (`[[` and `]]`):

```
# Get the second component of A
A[[2]]
```

```
[1] "chicken"
```

Notice that each component of `A` is a different data type (numeric, character, logical), which is not a problem for lists. Nothing was converted automatically, as we saw happen with vectors. Here's how to add a component to an existing list:

```
A[[4]] <- matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
```

Notice how we accessed component 4, which didn't exist yet, and assigned it a value. We actually added a *matrix* as the fourth component, this is *not possible* with vectors! Now `A` has four components:

`A`

```
[[1]]
[1] 42

[[2]]
[1] "chicken"

[[3]]
[1] TRUE

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

Lists can even *contain* other lists!

☀ **Bonus** If you try to assign a list to be one of its own components (e.g. `A[[5]] <- A`), then R will make a copy of `A` and assign the copy to be one of the components of `A`. Thus there is no “self reference”, and no issue with Russel’s Paradox.

💡 **Reflect** So far we've seen vectors, lists, matrices, and arrays. How are they different and how are they similar?

List components can also have *names*. Here we add an component with a name:

```
A[["color"]] <- "yellow"
A
```

```
[[1]]
[1] 42

[[2]]
[1] "chicken"

[[3]]
[1] TRUE

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

$color
[1] "yellow"
```

Notice how this new component displays differently? Instead of showing `[[5]]`, the component is labeled with a dollar sign, then its name: `$color`.

⚠️ **Caution** We first use the term *name* for individual variables, but here we see that components of lists can also have names. When we encounter data frames later, we'll see how each *row* and *column* can also have its own name.

You can access components using their name in two ways:

```
A[["color"]] # Use double square brackets to access a named element
```

```
[1] "yellow"
```

```
A$color # Use dollar sign to access a named element
```

```
[1] "yellow"
```

But the color component is also the fifth component of the list, so we can access it like this as well:

```
A[[5]]
```

```
[1] "yellow"
```

Here's a new list created by giving names to each element:

```
person <- list(name = "Millard Fillmore", occupation = "President", birth_year=1800)
person
```

```
$name
[1] "Millard Fillmore"

$occupation
[1] "President"

$birth_year
[1] 1800
```

🔗 **Progress Check** Below is some R code:  
S1\$year <- S2[2,2] + S3[["age"]]  
Assuming this code works, what are the data structures of S1, S2, and S3?

☀️ **Bonus** purrr is a very useful R package for working with lists.

**4.3.3.1 Lists and Vectors** Lists and Vectors are different data types, but in some ways they behave the same: Find the length of a list:

```
length(person) # Same for vectors and lists!
```

```
[1] 3
```

Combine two lists:

```
c(A, person) # Same for vectors and lists!
```

```
[[1]]
[1] 42

[[2]]
[1] "chicken"

[[3]]
[1] TRUE

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

```
$color
[1] "yellow"

$name
[1] "Millard Fillmore"

$occupation
[1] "President"

$birth_year
[1] 1800
```

```
A == "chicken" # Compare against a character
```

```
color
FALSE TRUE FALSE FALSE FALSE
```


However, there are some things that vectors can do that lists *can't*:

```
A + 1 # Add a number to each component (won't work)
```

```
A == T # Compare against a logical (won't work)
```

```
A == 12 # Compare against a numeric (won't work)
```

So there are trade-offs when deciding whether a list or a vector is most appropriate.

 **Video** This video discusses lists.  
<https://youtu.be/-Y02JkqDIWU>

**4.3.3.2 Lists of Vectors** Certain types of lists show up *all the time* in R, lists of vectors:

```
vec_1 <- c("Alice", "Bob", "Charlie")
vec_2 <- c(99.4, 87.6, 22.1)
vec_3 <- c("F", "M", "M")
special_list <- list(name=vec_1, grade=vec_2, sex=vec_3)
special_list
```

```
$name
[1] "Alice" "Bob" "Charlie"

$grade
[1] 99.4 87.6 22.1

$sex
[1] "F" "M" "M"
```

Here, each list stores a different piece of information about several people. Here's another example:

```
rocks <- list(specimen=c("A", "B", "C"),
              type=c("igneous", "metamorphic", "sedimentary"),
              weight=c(21.2, 56.7, 3.8),
              age=c(120, 10000, 5000000)
             )
rocks
```

```
$specimen
[1] "A" "B" "C"

$type
[1] "igneous"      "metamorphic" "sedimentary"

$weight
[1] 21.2 56.7 3.8

$age
[1]      120     10000 5000000
```

**⚠ Caution** When defining the `rocks` list, we’ve spread the command across multiple lines for clarity. The commas at the end of some of the lines separate the elements of the list. R will continue reading the next line until it finds the closing parenthesis, `)`.

There are so many sets of data that fit into this pattern, that R has a special data type called a *data frame*, which we will discuss in the next section.

**🔗 Progress Check** Create a matrix, a character vector, and a logical object, then place them all in a new list called “`my_list`”, with the names “`my_matrix`”, “`my_vector`”, and “`my_logical`”.


#### 4.3.4 Data Frames

At their core, data frames are just lists of vectors, but they also have some extra features as well. Here, we’ll re-define the `rocks` list from the previous section, but this time we’ll create it as a data frame:

```
rocks <- data.frame(type=c("igneous", "metamorphic", "sedimentary"),
                   weight=c(21.2, 56.7, 3.8),
                   age=c(120, 10000, 5000000))
rocks # We'll add the specimen names later
```

	type	weight	age
1	igneous	21.2	120
2	metamorphic	56.7	10000
3	sedimentary	3.8	5000000

Now when R displays `rocks`, it arranges the data in rows and columns, similar to how it displays matrices. Unlike matrices, however, the columns don’t all have to be the same data type!

 **Caution** Remember that a data frame is basically a list of vectors, so even though it can contain different types of data (because it is a list), each column is a vector, which means each column must have all elements of the same type.

The names of the columns are the names of the components of `rocks`, and the rows contain the data from each component vector. Remember that a data frame is basically a list of vectors, so we can access a component by its position or name:

```
rocks[[1]]
```

```
[1] "igneous"      "metamorphic" "sedimentary"
```

```
rocks$weight
```

```
[1] 21.2 56.7  3.8
```

However, we can also access a data frame as if it were a matrix:

```
rocks[1,3] # Get the datum from the first row, third column.
```

```
[1] 120
```

```
rocks[1,] # Get the first row, this gives another data frame with a single row.
```

```
   type weight age
1 igneous  21.2 120
```

```
rocks[,2] # Get the second column, this gives a vector.
```

```
[1] 21.2 56.7  3.8
```

Here's how to get the shape of a data frame (number of rows and columns):

```
dim(rocks)
```

```
[1] 3 3
```

If we start with a list of vectors, we can convert it to a data frame with `as.data.frame`:

```
people <- list(name=c("Alice", "Bob", "Charlie"),
               grade=c(99.4, 87.6, 22.1),
               sex=c("F", "M", "M"))
as.data.frame(people)
```

```
   name grade sex
1  Alice  99.4  F
2   Bob  87.6  M
3 Charlie 22.1  M
```

R comes with pre loaded with several data frames, such as `mtcars`, which contains data from the 1974 Motor Trend Magazine for 32 different automobiles:

`mtcars`

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

☀ **Bonus** A list of included data sets in R can be found by running `data()`.

Look at the column of car names on the left side of the `mtcars` data frame. It doesn't have a column name (like `mpg`, `cyl`, etc.), because it's *not actually a column*. These are *row names*, and you can access them like this:

`row.names(mtcars)`

```
[1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
[4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
[7] "Duster 360"         "Merc 240D"           "Merc 230"
[10] "Merc 280"           "Merc 280C"           "Merc 450SE"
```



```
[13] "Merc 450SL"      "Merc 450SLC"      "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial" "Fiat 128"
[19] "Honda Civic"      "Toyota Corolla"    "Toyota Corona"
[22] "Dodge Challenger" "AMC Javelin"       "Camaro Z28"
[25] "Pontiac Firebird"  "Fiat X1-9"         "Porsche 914-2"
[28] "Lotus Europa"     "Ford Pantera L"    "Ferrari Dino"
[31] "Maserati Bora"     "Volvo 142E"
```

You can also access the column names like this:

```
names(mtcars)
```

```
[1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

These are two examples of *attributes*, which are like extra information which are attached to an object. We'll discuss attributes more later when we discuss R objects. The column names and row names are just vectors, and you can access / modify them as such:

```
row.names(rock) <- c("A", "B", "C")
rock
```

```
      type weight    age
A   igneous   21.2    120
B metamorphic 56.7  10000
C sedimentary  3.8 5000000
```


```
names(rock)[[1]] <- "rock type"
rock
```

```
      rock type weight    age
A   igneous   21.2    120
B metamorphic 56.7  10000
C sedimentary  3.8 5000000
```

**⚠ Caution** Row and column names are allowed to have spaces in them, but you must be careful how you access them. The following code will not work: `rock$rock type`, because R will stop looking for the name you are referencing once it encounters a space. To access this column, you must enclose the reference in “backticks” ( ``` ) like so: `rock$`rock type``.

**🔗 Progress Check** Look at the set of available data sets in R, and pick 2 data sets. For each data set, answer the following questions:


- What are the column names?
- What are the row names?
- What is the data type for each column?
- How many rows are in the data frame?
- How many columns are in the data frame?

 **Assessment** This is the last section you should include in Progress Check 2. Knit your output document and submit on Canvas.


 **Feedback** Any feedback for this section? [Click here](#)

 **Video** This video discusses lists of vectors.  
<https://youtu.be/9BGRIC1js04>

## 4.4 R Objects

 **Reflect** Wherever you are right now, look around your environment. Pick an object and study its attributes. It probably has a shape, a color, a weight, and many other ways of describing it. Now pick another object, and note how it is different than the first in terms of its attributes. What does the word “object” really mean? It’s often easier to give examples than to give a precise definition, but generally objects are “things you can do things with”. That is, you can usually look at them, touch them, smell them, and move them around (when appropriate/possible, of course!). Another useful definition is that objects are *nouns*. Different objects have different purposes and attributes. Many of these ideas will be true for R objects as well.

We’ve already introduced the concepts of objects in R in passing, but here we briefly focus on what they are and how to work with them.

 **Assessment** Download the progress check 3 template into your `data_raw` folder, and follow the instructions. That document should include all progress reports from Section 4.4 through (and including) Section 5.4

### 4.4.1 *Everything* is an object in R

What exactly is an object in R? As in real life, it can be difficult to give a definition, but easier to give examples. Here are some examples of objects in R:

1. A numeric variable
2. A vector
3. A matrix
4. A list
5. A data frame
6. A *function*

This list is not exhaustive, but most objects we deal with will look like one of these.

☀ **Bonus** In many programming languages, functions are handled differently from other types of objects (i.e. they are not “first class” objects). In R, they are treated the same as any other type of object. You can assign them to variables, pass them to other functions, and can be returned from a function. This is similar to the behavior of Java and Python, but different from C.

#### 4.4.2 Assigning Objects

Any object can be assigned to a variable, as we’ve been doing already. Here’s an example:

```
a <- "pink pineapple"
```

The `<-` is called an *assignment operator*. This is the most common way of assigning objects in R, but there are others. Sometimes you may see:

```
a = "pink pineapple"
```

which in *most cases*, has the exact same effect as using the `<-`, but in a few instances, it has a different effect. Our recommendation is to always use `<-` when making object assignments.

☀ **Bonus** There are other assignment operators as well, `<<-`, `->>`, and `->`, but we will not discuss these. You can find out more with the command `?assignOps`.

One neat thing you can do is assign multiple variables at the same time:

```
a <- b <- "Hello"
```

```
a
```

```
[1] "Hello"
```

```
b
```

```
[1] "Hello"
```

⚠ **Caution** Even though `a` and `b` were assigned at the same time, they are still different! So if you change `a` with `a <- "goodbye"`, then the value of `b` will still be `"Hello"`.

#### 4.4.3 Attributes

Every object in R has *attributes*, extra information that’s “attached” to the object. *Every object* has a length attribute:

```
a <- c(1, 2, 3, 4)
b <- c("bonjour", "au revoir")
```

```
length(a)
```

```
[1] 4
```

```
length(b)
```

```
[1] 2
```

^v **Progress Check** *Every object* has a length attribute. Try creating an example of the following and examining the length of each: 1. A logical vector with 5 elements 2. A matrix with two rows and two columns 3. A list with two objects in it.

Every R object has a *mode* as well, which tells you what *type* of object you have. Here are some examples:

```
mode(a)
```

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

```
mode(b)
```

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

^v **Progress Check** *Every object* has a length. Try creating an example of the following and examining the length:

1. A logical vector with 5 elements
2. A 2 x 2 matrix
3. The `mtcars` dataframe

Aside from these two attributes, you can list all attributes of an object like this:

```
attributes(mtcars)
```

```
$names
[1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"

$row.names
[1] "Mazda RX4"          "Mazda RX4 Wag"       "Datsun 710"
[4] "Hornet 4 Drive"     "Hornet Sportabout"   "Valiant"
[7] "Duster 360"        "Merc 240D"           "Merc 230"
```

```
[10] "Merc 280"           "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"         "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"        "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"   "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"   "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"       "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"      "Volvo 142E"

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

To access a specific attribute of an object, you can do this:

```
attr(mtcars, "class") # Get the class attribute for the mtcars data frame
```

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

#### 4.4.4 Null Objects

There is a special object called the NULL object, which really just represents “nothing”. It’s used mainly if you want to *remove* an element from a list:

```
a <- list(1, 2, 3)
a[[2]] <- NULL # Replace component 2 with "nothing"
a
```

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

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

Or if a function is supposed to return but doesn’t have an object to return (more on this later when we discuss functions).

#### 4.4.5 Removing Objects

Sometimes you want to *get rid* of an object! In R, you can use the `rm` function like so:

```
a <- "an object"
rm(a)
a
```

As you can see, the error message indicates that `a` has been removed. Sometimes, you’d like to remove all the objects in your environment. To do this, you can use the command:

```
rm(list=ls())
```

● **Feedback** Any feedback for this section? [Click here](#)

🎥 **Video** This video discusses objects.  
<https://youtu.be/VrgoEoMo9ZM>

## 5 Working with Data

“The goal is to turn data into information, and information into insight.” – Carly Fiorina, former CEO of Hewlett-Packard

In the previous chapter, we’ve talked about the different types of data that R stores and the different structures that R stores data *in*. We’ve mostly just made up numbers, character strings, and logical values for illustration. In this chapter, we’ll use R to do interesting things with *real data*. This is by far the most popular use of the R programming language, and arguably the most fun! You’ll learn how to read data sets into R, do interesting things with them, and save your results.

### 5.1 Quick Example

Before diving into detail, let’s do a quick example so you can begin to see what is possible with data in R. As we mentioned in the last chapter, R includes some pre-packaged data sets, which you can access with the `data()` command. One of the data sets is `Seatbelts`, which documents road casualties in Great Britain between 1969 and 1984. Firstly, we need to convert `Seatbelts` to a data frame, because it starts out as a “Time-Series”, which we haven’t discussed.

```
Seatbelts <- data.frame(as.matrix(Seatbelts), date=time(Seatbelts)) # convert Time Series to data frame
```

We’ve also added a month and year column

look at the dimensions of the data set with the `dim` command:

```
dim(Seatbelts) # get the number of dimensions in the Seatbelts data frame
```

```
[1] 192 9
```

This shows that there are 192 rows (months), and 9 columns (variables measured each month). We could also determine the number of rows and columns separately using the `nrow` and `ncol` functions. To view the first few rows of the `Seatbelts` data frame, use the `head` command:

```
head(Seatbelts) # view first few rows of the Seatbelts dataset
```

	DriversKilled	drivers	front	rear	kms	PetrolPrice	VanKilled	law	date
1	107	1687	867	269	9059	0.1029718	12	0	1969.000
2	97	1508	825	265	7685	0.1023630	6	0	1969.083
3	102	1507	806	319	9963	0.1020625	12	0	1969.167
4	87	1385	814	407	10955	0.1008733	8	0	1969.250
5	119	1632	991	454	11823	0.1010197	10	0	1969.333
6	106	1511	945	427	12391	0.1005812	13	0	1969.417

This is a good way to learn which variables are being measured (columns) and see some example observations (rows) for each variable. Because these data are included with R, more information about each variable can be found with:

```
?Seatbelts
```

Next, let's view a summary of each column with the summary function:

```
summary(Seatbelts)
```

DriversKilled	drivers	front	rear
Min. : 60.0	Min. :1057	Min. : 426.0	Min. :224.0
1st Qu.:104.8	1st Qu.:1462	1st Qu.: 715.5	1st Qu.:344.8
Median :118.5	Median :1631	Median : 828.5	Median :401.5
Mean :122.8	Mean :1670	Mean : 837.2	Mean :401.2
3rd Qu.:138.0	3rd Qu.:1851	3rd Qu.: 950.8	3rd Qu.:456.2
Max. :198.0	Max. :2654	Max. :1299.0	Max. :646.0

kms	PetrolPrice	VanKilled	law
Min. : 7685	Min. :0.08118	Min. : 2.000	Min. :0.0000
1st Qu.:12685	1st Qu.:0.09258	1st Qu.: 6.000	1st Qu.:0.0000
Median :14987	Median :0.10448	Median : 8.000	Median :0.0000
Mean :14994	Mean :0.10362	Mean : 9.057	Mean :0.1198
3rd Qu.:17202	3rd Qu.:0.11406	3rd Qu.:12.000	3rd Qu.:0.0000
Max. :21626	Max. :0.13303	Max. :17.000	Max. :1.0000

date
Min. :1969
1st Qu.:1973
Median :1977
Mean :1977
3rd Qu.:1981
Max. :1985

Since each column is numeric, R shows a five number summary (minimum, first quartile, median, third quartile, maximum) and mean for each column. We learn, for example, that the average number of drivers killed per month is 1670, and that the greatest price of petrol was £0.13 per litre! Let's view a histogram of DriversKilled:

```
hist(Seatbelts$DriversKilled, breaks=20)
```

We see that in some months, more than 150 drivers were killed! We can calculate how many exactly like so:

```
sum(Seatbelts$DriversKilled > 150)
```

```
[1] 33
```

To investigate the effect of the seat belt law, let's create a scatter plot Drivers killed against time:

```
plot(Seatbelts$date, Seatbelts$DriversKilled)
```

By adding a col argument to the plot function, we can color the points based on whether the law was in effect:

```
plot(Seatbelts$date, Seatbelts$DriversKilled, col=(Seatbelts$law+2))
```



Figure 8: Histogram of Drivers Killed in Seatbelt data



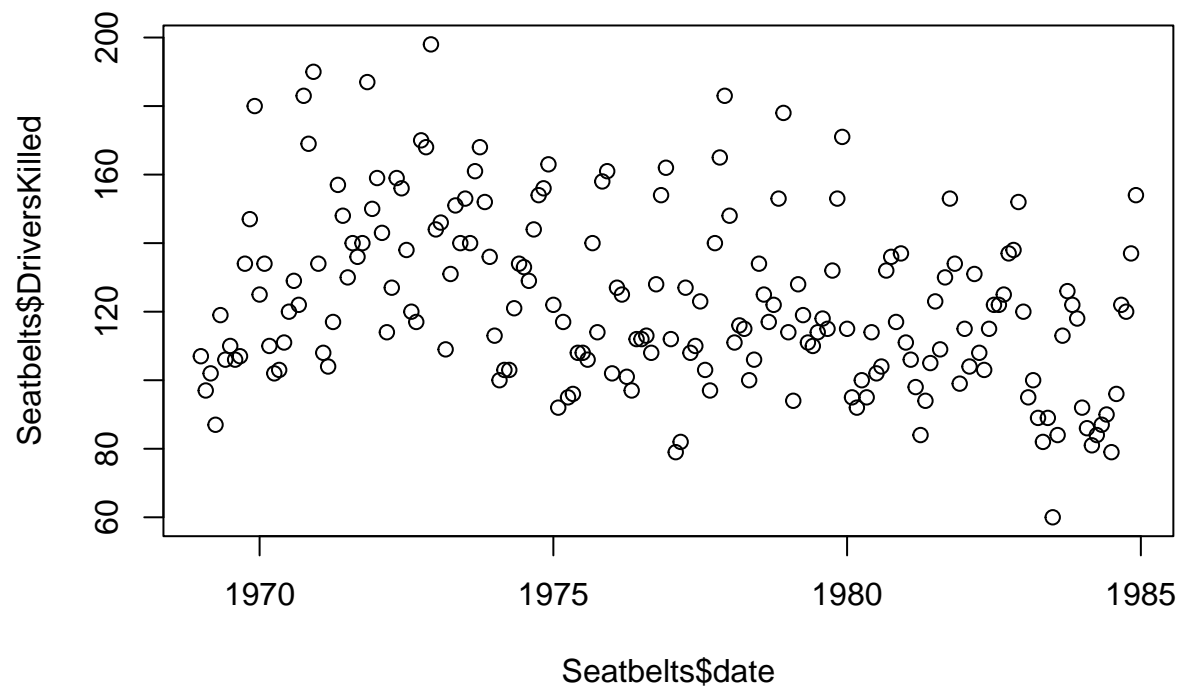


Figure 9: UK Seatbelt deaths vs time

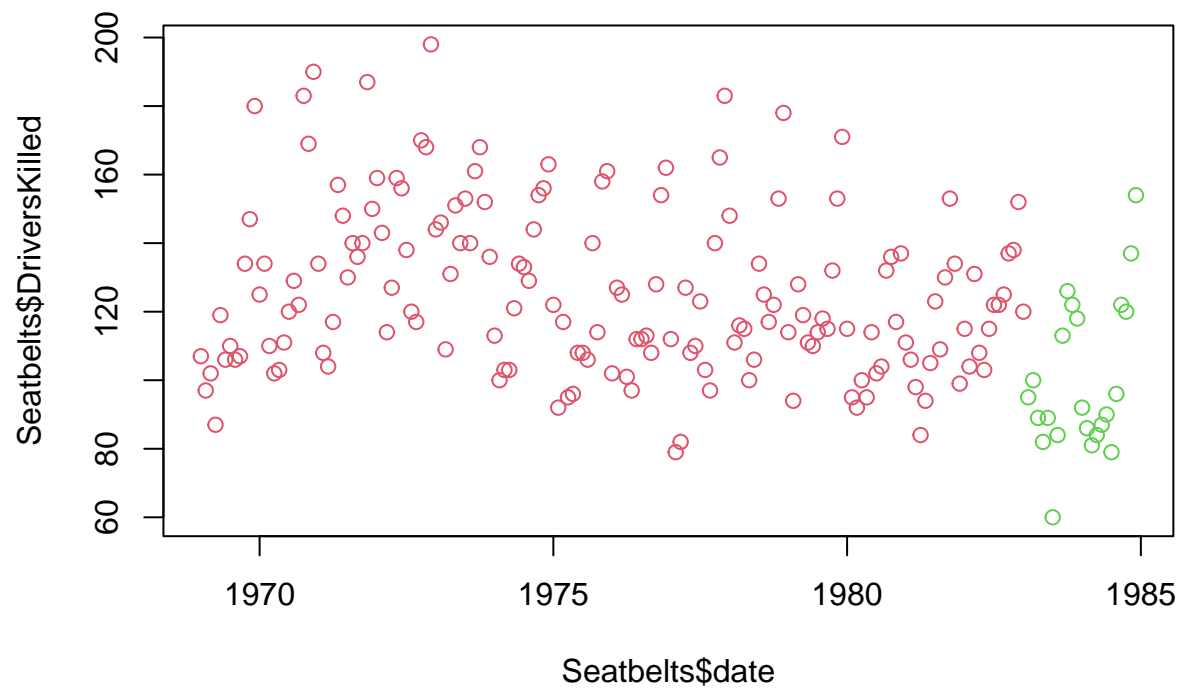


Figure 10: UK Seatbelt deaths vs time, red = no seatbelt law, green = seatbelt law

There do appear to be fewer deaths, but there is so much fluctuation in deaths each year that it's difficult to tell.

Let's change the x-axis to reflect month of the year instead of date:

```
plot((Seatbelts$date %% 1) * 12 + 1, Seatbelts$DriversKilled,
     xlab = "Month", col=Seatbelts$law + 2)
```

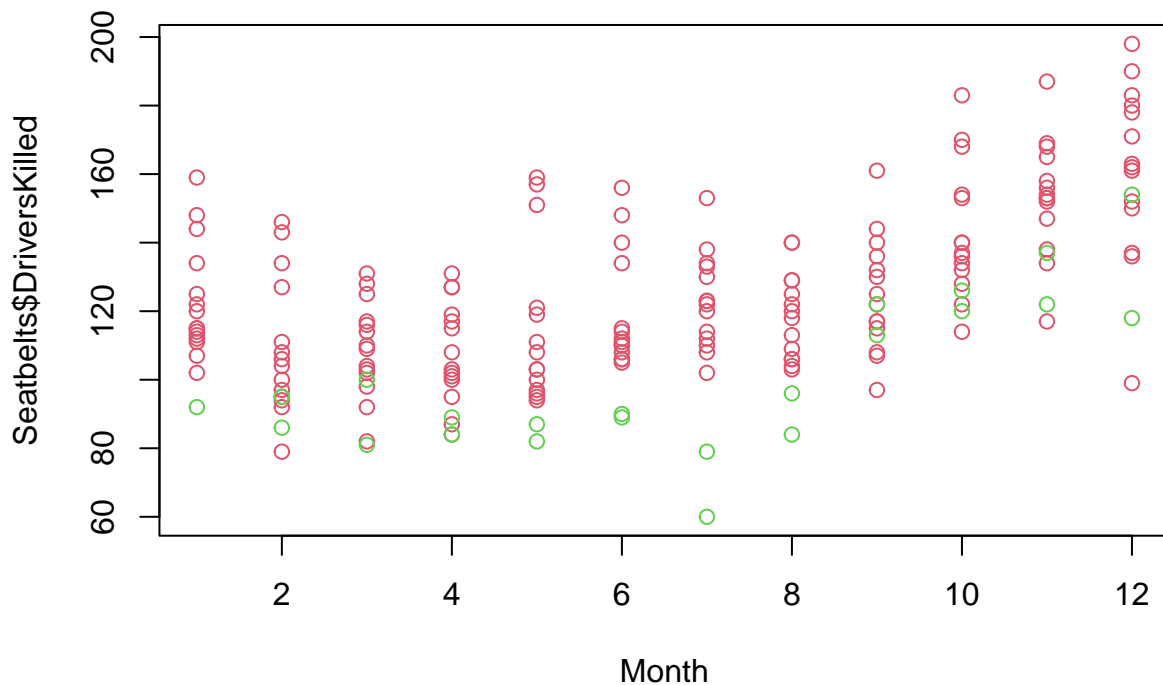


Figure 11: UK Driver Deaths vs. Month

This plot shows that there is a clear seasonal effect in the number of deaths with higher deaths occurring in the Fall/Winter compared to Spring/Summer. We can also see that within each month, the traffic deaths after enacting the Seatbelt law are among the lowest.

🔗 **Progress Check** Another data set included in R is `mtcars`. Following the example above, find the dimension of `mtcars` and have R print out a summary of each column, then create a scatter plot of fuel economy (`mpg`) to engine displacement. What do you observe about the relationship between these two variables?

This concludes the quick example. In the rest of this chapter, we'll talk in more detail about the different steps of working with data, and how to complete them using R!

💡 **Reflect** People often use data in order to answer questions, but often times, learning about data can generate even *more* questions than it answers. Take a moment to think of a question that you have about the `Seatbelts` dataset. Do you think the question can be answered using the data alone? If not, what other sources of data might be available which can help answer the question?

🗉 **Feedback** Any feedback for this section? [Click here](#)

## 5.2 Reading / Writing Data

Of course, if we want work on data which is NOT included in R, we have to *read* that data into R in order to work with it. That is, the data are normally somewhere on your computer's hard drive (or SSD), and you must run a command in R which *reads* that data into your R environment.

### 5.2.1 Olympic Athletes Example

Let's look at another example, this time with a data set of Olympic athletes. This is just a subset of the full dataset, to make it easier for you to work with. Here's how we'll *read* them into R:

```
# Read the csv file into a data frame called athletes
athletes <- read.csv("data_raw/olympic_athletes.csv")

# print a summary of the data frame
summary(athletes)
```

X	ID	Name	Sex
Min. : 1	Min. : 4	Length:5000	Length:5000
1st Qu.:1251	1st Qu.: 35321	Class :character	Class :character
Median :2500	Median : 68266	Mode :character	Mode :character
Mean :2500	Mean : 68668		
3rd Qu.:3750	3rd Qu.:102377		
Max. :5000	Max. :135559		


  

Age	Height	Weight	Team
Min. :12.00	Min. :139.0	Min. : 33.00	Length:5000
1st Qu.:21.00	1st Qu.:168.5	1st Qu.: 61.00	Class :character
Median :25.00	Median :175.0	Median : 70.00	Mode :character
Mean :25.65	Mean :175.4	Mean : 70.91	
3rd Qu.:28.00	3rd Qu.:183.0	3rd Qu.: 80.00	
Max. :74.00	Max. :223.0	Max. :182.00	
NA's :183	NA's :1109	NA's :1131	


NOC	Games	Year	Season
Length:5000	Length:5000	Min. :1896	Length:5000
Class :character	Class :character	1st Qu.:1960	Class :character
Mode :character	Mode :character	Median :1988	Mode :character
		Mean :1978	
		3rd Qu.:2002	

		Max.	:2016
City	Sport	Event	Medal
Length:5000	Length:5000	Length:5000	Length:5000
Class :character	Class :character	Class :character	Class :character
Mode :character	Mode :character	Mode :character	Mode :character

 **Caution** The above command only works because the data set is in a particular location (the data folder), and is in a particular format (csv). In the sections that follow, we'll discuss how to address both of these issues.


### 5.2.2 Locating your data set

R is capable of reading data from your computer, no matter where it is, as long as you “point” R to the correct location. The location is usually specified with a *file path*, which is a character string that specifies the folder structure that holds your file. By default, R starts “looking” from the current working directory, and the file path used was `data_raw/olympic_athletes.csv`. This means that R will look inside the current working directory for a folder called `data_raw`, and if it exists, R will look inside `data_raw` for a file called `olympic_athletes.csv`. In this class, you should be working within an rstudio project, which automatically sets the working directory. If you created the folders as instructed earlier, then you should already have a `data_raw` folder in your project directory.

 **Progress Check** Download the olympic athletes data set from this link and save it in your `data_raw` folder. In your progress check document, simply write: “Olympic Data Downloaded”

### 5.2.3 Reading CSV files

A common way of storing data in a computer file is called CSV, which stands for *comma-separated values*. These files are *plain text*, so you can open them in any text editor like Word, Notepad, or even Google Docs. Just like a data frame, these files contain data in rows and columns, where on each line, the columns are *separated* from each other with a comma (,), which is technically called a *delimiter*. Programs like Excel, Google Sheets, and R can read these files and understand their row-column structure. In R, the function to read CSV files (as you saw above) is `read.csv`. In addition, if you call up the help file for `read.csv` using `?`, you'll see that there are other similar functions as well, including `read.table`, and `read.delim`.

 **Bonus** In many object oriented languages, the “dot” (.) is a special symbol used to access an attribute or method of an object. In R, however, variable names and function names can contain a dot, and the dot has no special purpose. There are some exceptions, however, that relate to function overloading, and R formulas, but these are advanced topics that will not be discussed here.

Function	Delimiter	Decimals
read.table	Must specify with sep=...	.
read.csv	,	.
read.csv2	;	,
read.delim	\t (tab)	.
read.delim2	\t (tab)	,

These functions are actually all different variations of the same, generic, function called `read.table`. The difference is that `read.csv`, `read.delim`, and the others make different assumptions about what delimiters are being used, and how decimal numbers are displayed (e.g. one-and-a-half may be written as 1.5, or 1,5 depending on where you live). We will discuss functions and arguments more in the next chapter, but for now, see the following table for when to use each function:

Any of these functions accepts the argument `header=FALSE`, which indicates that the first row of the file *does not* contain column names. Without this argument, R will assume the first row *does* contain column names. If our Olympic athletes data did not contain headers in the first row, we would use this:

```
athletes <- read.csv("data_raw/olympic_athletes.csv", header=FALSE)
```


## 5.2.4 Writing CSV files


R also has the capability to *write* a data frame to a CSV file on your computer, that could then be read by Excel, Sheets, etc. Let's suppose we wanted to save a version of the athletes data with only the **Sex** and **Age** columns. We could use the `write.csv` function:


```
# make a new data frame with only the Sex and Age columns
athletes2 <- athletes[,c("Sex", "Age")]

# save the new data frame as a CSV in the clean data folder
write.csv(athletes2, "data_clean/olympic_athletes_age_sex.csv")
```

Notice we created a new data frame by selecting only the desired columns. We will talk more about different ways to select data when we discuss *indexing*. Notice also that the `write.csv` function requires that we give it the name of the data frame being saved (`athletes2`), then the file path for the csv file that the data will be written to (`"data_clean/olympic_athletes_age_sex.csv"`).

 **Caution** `write.csv` is an example of a function which takes *multiple arguments*, separating them with a comma (,). Usually, these arguments must be specified in order, but more will be said about this later.

 **Progress Check** Create a version of the athletes data frame which contains the athletes' names and their sports. Save the new data frame as a csv file in your `data_clean` folder with the file name `"olympic_athletes_name_sport.csv"`. Include the code in your progress check assignment.

 **Caution** The `read` and `write` terminology may seem odd if you have not heard those terms before. Your computer's hard drive (or SSD) will store data which will be remembered even after you turn off your computer. The process of getting data from, and putting data on your hard drive (or SSD) is called *reading* and *writing*.


 **Feedback** Any feedback for this section? [Click here](#)

## 5.3 Summary Statistics

Reading and writing data is useful, but the power of R is *doing interesting things* with the data! Let's perform a few operations with the Olympic athletes data to demonstrate some important functions for data analysis. As we've seen, the `summary` function automatically performs some summary statistics on each column of a data frame. Let's see how to produce these and other results manually.

### 5.3.1 Quantitative Variables


To showcase the functions R provides to summarize quantitative variables, we'll look at the `Age` column of our data frame, which is stored as an integer vector in R.

 **Reflect** What other R data types might be used to store quantitative data?

However, `Age` contains NA values, as we know from running the following function:


```
sum(is.na(athletes$Age))    # count how many NA's are in the Age column
```

```
[1] 183
```

 **Reflect** Pause and think through what's happening in the above code. The `is.na` function returns a logical array which is true whenever the `Age` column is NA. So why does the `sum` function produce the number of NA's?

As a cleaning step, we must first remove the NA values:

```
age <- athletes$Age          # assign the Ages column to a variable
age <- age[!is.na(age)]      # extract only the elements which are not NA (more on this when we discuss ad
```

 **Bonus** This type of “data cleaning” is a very common first step when performing data analysis. You will get more opportunities to clean data later in the course.

Here are some functions R provides to summarize quantitative variables.

```

age_min <- min(age)      # find the minimum age
age_med <- median(age)   # find the median age
age_max <- max(age)      # find the maximum age
age_mean <- mean(age)    # find the average age
age_sd <- sd(age)        # find the standard deviation of age
age_var <- var(age)      # find the variance of age

```

Let's put all these results in a named list. In the following code, read the comments carefully to understand how the code is being organized onto multiple lines.

```

# Create a list containing all the stats
age_stats <- list( # R knows that this command continues until a closed parenthesis
  min = age_min,
  median = age_med,
  max = age_max,
  mean = age_mean,
  sd = age_sd,
  var = age_var
) # this could all go on one line, but it looks more organized this way.
age_stats

```

```

$min
[1] 12

$median
[1] 25

$max
[1] 74

$mean
[1] 25.65373

$sd
[1] 6.495693

$var
[1] 42.19402

```

🔗 **Progress Check** Using the Olympic Athletes data, create a list called `weight_stats` with the mean, median, and standard deviation of the `Weight` column. If you get NA values for the statistics, you should include the `na.rm=T` argument like so: `mean(weight, na.rm=T)`, to remove the NA values before computing the statistics.

Visualization will be discussed more later, but we'll show one plot now, to show how multiple summary statistics can be shown at the same time.

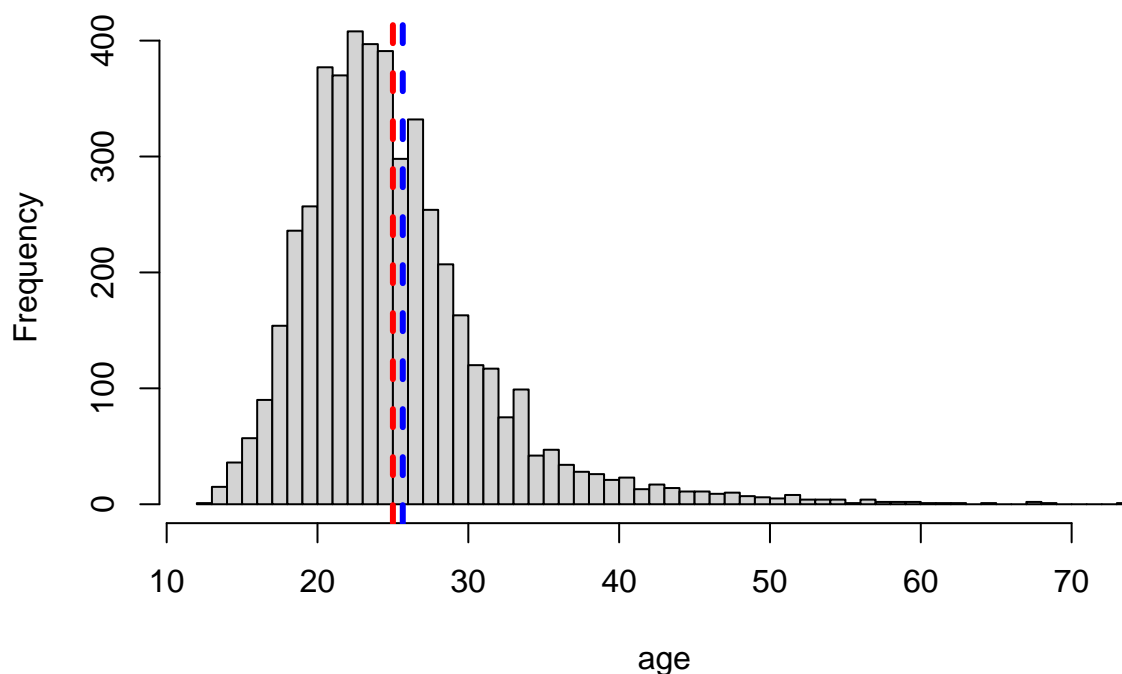
```

hist(age, breaks=50)
abline(v=age_mean, col="blue", lty=2, lwd=3)
abline(v=age_med, col="red", lty=2, lwd=3)

```



## Histogram of age



It looks like the distribution of **Age** is *right skewed*, consistent with the fact that the mean is greater than the median.

Of course, having more than one quantitative variable allows us to compare them against each other. Here's how to compute the covariance between two quantitative variables:

```
cov(athletes$Age, athletes$Height, use="complete.obs")
```

```
[1] 6.483675
```

**⚠ Caution** The argument `use="complete.obs"` is one way to deal with NA values in the `cov` function. This makes R remove any observations which are NA in either the first or second variable. There are other ways as well, which you can check using the help function: `?cov`.

You can also compute the correlation between two variables like so:

```
cor(athletes$Age, athletes$Height, use="complete.obs")
```

```
[1] 0.1104457
```

Using the `cov` or `cor` functions on an entire data frame or matrix will produce a correlation matrix of the columns. Here's an example with the `mtcars` data frame:

```
cor(mtcars)
```

	mpg	cyl	disp	hp	drat	wt
mpg	1.0000000	-0.8521620	-0.8475514	-0.7761684	0.68117191	-0.8676594
cyl	-0.8521620	1.0000000	0.9020329	0.8324475	-0.69993811	0.7824958
disp	-0.8475514	0.9020329	1.0000000	0.7909486	-0.71021393	0.8879799
hp	-0.7761684	0.8324475	0.7909486	1.0000000	-0.44875912	0.6587479
drat	0.6811719	-0.6999381	-0.7102139	-0.4487591	1.00000000	-0.7124406
wt	-0.8676594	0.7824958	0.8879799	0.6587479	-0.71244065	1.0000000
qsec	0.4186840	-0.5912421	-0.4336979	-0.7082234	0.09120476	-0.1747159
vs	0.6640389	-0.8108118	-0.7104159	-0.7230967	0.44027846	-0.5549157
am	0.5998324	-0.5226070	-0.5912270	-0.2432043	0.71271113	-0.6924953
gear	0.4802848	-0.4926866	-0.5555692	-0.1257043	0.69961013	-0.5832870
carb	-0.5509251	0.5269883	0.3949769	0.7498125	-0.09078980	0.4276059

	qsec	vs	am	gear	carb
mpg	0.41868403	0.6640389	0.59983243	0.4802848	-0.55092507
cyl	-0.59124207	-0.8108118	-0.52260705	-0.4926866	0.52698829
disp	-0.43369788	-0.7104159	-0.59122704	-0.5555692	0.39497686
hp	-0.70822339	-0.7230967	-0.24320426	-0.1257043	0.74981247
drat	0.09120476	0.4402785	0.71271113	0.6996101	-0.09078980
wt	-0.17471588	-0.5549157	-0.69249526	-0.5832870	0.42760594
qsec	1.00000000	0.7445354	-0.22986086	-0.2126822	-0.65624923
vs	0.74453544	1.0000000	0.16834512	0.2060233	-0.56960714
am	-0.22986086	0.1683451	1.00000000	0.7940588	0.05753435
gear	-0.21268223	0.2060233	0.79405876	1.0000000	0.27407284
carb	-0.65624923	-0.5696071	0.05753435	0.2740728	1.00000000

However, this only works if every column the data frame (or matrix) are numeric. Here's what happens if we try the same thing on the athletes data:

```
cor(athletes)
```

### 5.3.2 Categorical Variables

To showcase the functions R provides for categorical variables, we'll look, at the `Sport` column, which is stored as a character vector in R.

💡 **Reflect** What other R data types might be used to store categorical data?

Are there any NA values in this column?

```
sport <- athletes$Sport
sum(is.na(sport))
```

```
[1] 0
```

It turns out the answer is no, so there's no need to remove anything. In a character vector like this, we expect there to be many duplicated values. We can see a list of all the unique values with the following:

```
unique(sport)
```

```
[1] "Hockey" "Wrestling"
[3] "Swimming" "Basketball"
[5] "Biathlon" "Speed Skating"
[7] "Fencing" "Weightlifting"
[9] "Equestrianism" "Archery"
[11] "Cross Country Skiing" "Gymnastics"
[13] "Tennis" "Athletics"
[15] "Cycling" "Bobsleigh"
[17] "Shooting" "Sailing"
[19] "Alpine Skiing" "Art Competitions"
[21] "Canoeing" "Football"
[23] "Rowing" "Figure Skating"
[25] "Nordic Combined" "Judo"
[27] "Short Track Speed Skating" "Water Polo"
[29] "Snowboarding" "Taekwondo"
[31] "Diving" "Handball"
[33] "Softball" "Boxing"
[35] "Tug-Of-War" "Ski Jumping"
[37] "Table Tennis" "Ice Hockey"
[39] "Modern Pentathlon" "Golf"
[41] "Baseball" "Volleyball"
[43] "Luge" "Badminton"
[45] "Trampolining" "Curling"
[47] "Beach Volleyball" "Polo"
[49] "Rugby Sevens" "Synchronized Swimming"
[51] "Triathlon" "Skeleton"
[53] "Freestyle Skiing" "Military Ski Patrol"
[55] "Lacrosse" "Rhythmic Gymnastics"
[57] "Rugby"
```

Using the numbers in brackets to the left as our guide, we can see that there are 57 unique values, but this can also be determined by running:

```
length(unique(sport))
```

```
[1] 57
```

It would be nice to see how many times each entry occurs in the data set. This is what the `table` function does:

```
table(sport)
```

sport	Alpine Skiing	Archery	Art Competitions
	148	41	64
	Athletics	Badminton	Baseball
	728	32	19
	Basketball	Beach Volleyball	Biathlon
	98	18	100
	Bobsleigh	Boxing	Canoeing

	53	121	112
Cross Country Skiing		Curling	Cycling
	174	8	205
Diving		Equestrianism	Fencing
	56	121	184
Figure Skating		Football	Freestyle Skiing
	44	138	9
Golf		Gymnastics	Handball
	5	498	61
Hockey		Ice Hockey	Judo
	101	83	76
Lacrosse		Luge	Military Ski Patrol
	1	25	2
Modern Pentathlon		Nordic Combined	Polo
	37	25	4
Rhythmic Gymnastics		Rowing	Rugby
	9	190	4
Rugby Sevens		Sailing	Shooting
	6	126	218
Short Track Speed Skating		Skeleton	Ski Jumping
	23	4	45
Snowboarding		Softball	Speed Skating
	19	10	104
Swimming	Synchronized Swimming		Table Tennis
	399	9	36
Taekwondo		Tennis	Trampolining
	10	45	4
Triathlon		Tug-Of-War	Volleyball
	6	5	50
Water Polo		Weightlifting	Wrestling
	79	85	123

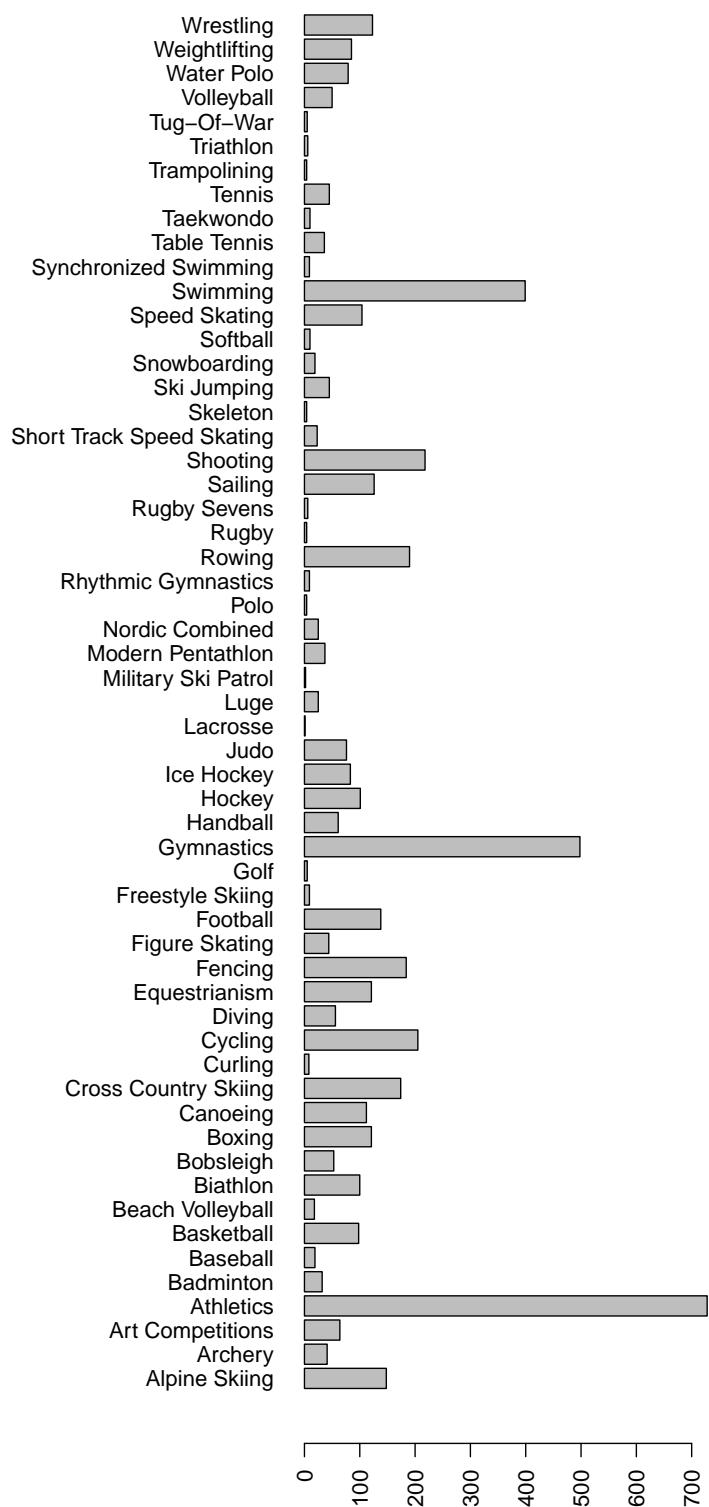
Let's save this table to a list as before:

```
# assign summary statistics to variables
sport_n_unique <- length(unique(sport))
sport_counts <- table(sport)

# combine them into a list
sport_stats <- list(
  number_unique = sport_n_unique,
  counts = sport_counts
)
```

Again, a visualization may be useful here:

```
par(mar=c(5,15,4,2) + 0.1) # command to make the labels fit
barplot(table(sport), horiz=T, las=2) # bar plot
```



So we see that in our data set, athletics, swimming, and gymnastics have the most athletes (remember, each row is an athlete).

🔧 **Progress Check** Using the Olympic athletes data, create a list called `season_stats` with the a table of counts for the `Season` variable.

⚠️ **Caution** It's always important to remember what the rows of your data set represent. In the Olympic athletes example, one athlete may occupy multiple rows, if they competed in multiple olympic games. This impacts how you should interpret the summary statistics computed above (mean, median, counts, etc.).

💡 **Reflect** Since an athlete may show up for multiple olympic games, what impact could this have on summary statistics for the `Height`, `Weight`, and `Sex` variables? Can you give an example of what might happen? What other variables may be impacted? What R code would you write to determine if an athlete occurred multiple times in the data frame?

### 5.3.3 Saving an RData file

Finally, we may want to save these results for use *in R* later. First, we'll create a new list to put our two stats list *in* (remember, we can have lists inside of other lists!).

```
# Create list
athlete_stats <- list(
  age = age_stats,
  sport = sport_stats
)
```

☀️ **Bonus** Remember that the `names` function retrieves the column names for a data frame? It also retrieves the names of a list (after all, a data frame is just a fancy list, right?)! The following commands may be useful for remembering what the contents of our stats list:

- `names(athlete_stats)`
- `names(athlete_stats$age)`
- `names(athlete_stats$sport)`

To save these results, we can use the `saveRDS` function:

```
saveRDS(athlete_stats, "data_clean/athlete_stats.rds")
```

Later, we can use the following command to load the RDS file back into R:

```
rm(athlete_stats) # remove athlete stats to prove we are loading it from the hard drive

athlete_stats <- readRDS("data_clean/athlete_stats.rds") # load the RDS file and name it athlete_stats

athlete_stats$age # show that we have loaded the data by printing the age stats
```

```
$min
[1] 12

$median
[1] 25

$max
[1] 74

$mean
[1] 25.65373

$sd
[1] 6.495693

$var
[1] 42.19402
```

⚠ **Caution** Notice the file ends with *.rds*, indicating that this is a special RDS type which can only be read by R. This is different from other data formats like CSV, which are plain text and can be read by other programs like Excel or Sheets. RDS should only be used when you want to save work that you want to resume in R later. If at all possible, you should prefer using plain text formats rather than RDS.

☀ **Bonus** RDS stands for *R Data Serialization*. This is R's version of object serialization, just like the `io.Serializable` interface in Java or the `pickle` module in Python. As with other languages, R's serialization can only be used in R.

The RDS format works for any R Object, not just lists, so it can be used for vectors, matrices, factors, and even functions.

🗨 **Feedback** Any feedback for this section? [Click here](#)

## 5.4 Data Formatting

Before we continue working with data, here are a few comments about data formatting. Many data sets can be thought of as one or more *observations* of one or more *variables*. R functions work best when the data are formatted into rows and columns, so that each row is an observation, and each column is a variable. Unfortunately, sometimes data do not follow this convention, or worse, it may not be clear what the

observations or variables are. Working with data often involves answering multiple questions, and different questions may require thinking of observations and variables differently. In R, there are ways of changing the structure of data to suit your particular needs, but these are intermediate topics which will not be discussed here.

☀ **Bonus** One concept related to data organization is called “Tidy Data”, which you can read more about here. This focus on tidyness has led to the development of a set of R packages collectively called the “tidyverse”, which has become very popular for R analysis. The tidyverse will not be covered in this class, but a later module will provide extensive experience with it.

#### 5.4.1 “Raw” data vs. “Clean” data.

Why is there a “data\_raw” folder and a “data\_clean” folder, and not just a “data” folder? The idea is that the data\_raw folder contains all of the *original* data sets that you start with, before any cleaning or summarization take place, and any cleaned, modified, or created data sets that result from your data analysis should be stored in the “data\_clean” folder (or possibly even a “results” folder). This distinction ensures that the original data sets are preserved in their unedited state, just in case you need to start over from the beginning to answer a different question, and in order for others to easily replicate your work. This is why the data in the folder should be thought of as *read only*. Sometimes people even modify the permissions of the raw data files on their computer to prevent anyone from accidentally deleting or overwriting the raw data. The “clean” moniker comes from the fact that often times data sets need some “cleaning” such as removing duplicates, removing NA values, discarding irrelevant data, etc. There are many other ways of organizing data, but the principle here is to separate the original data sets from any intermediate data sets.

💡 **Reflect** Perhaps you’ve never thought about how data should be structured. Consider an experiment which measures the temperatures of five guinea pigs for each of four different days. Think about organizing each row to be a guinea pig and each column to be a day. Can you think of an R function to compute the average temperature on day 1? How about the average temperature for guinea pig 3? How do your answers change if the data are arranged with days as the rows and guinea pigs as columns? Can you think of another way to organize the data?


✍ **Assessment** This is the last section you should include in Progress Check 3. Knit your output document and submit on Canvas.

🗣 **Feedback** Any feedback for this section? [Click here](#)

## 5.5 Indexing

Part of doing interesting things with data is being able to select just the data that you need for a particular circumstance. You’ve already seen how to get a particular element from a vector or matrix, or a specific component from a list, using *indices*. A datum’s *index* is its position in the vector or list. For example, to get the second element of a vector **A**, we use index 2 in square brackets: **A[2]**. The process of selecting elements using their indices is called *indexing*, and R provides multiple ways of indexing vectors. Below we’ll cover some basic indexing and more advanced indexing for the different data structures in R.



 **Assessment** Download the progress check 4 template and follow the instructions. That document should include all progress reports from Section 5.5 through (and including) Section 6.1


### 5.5.1 Vectors

Let's define a vector and access an element in the way you already know:

```
# create an example vector
V <- c("A", "B", "C", "D", "E", "F", "G", "H", "I")

# access the 5th element
V[5]
```

```
[1] "E"
```

 **Caution** Unlike many other languages, R indices start with 1, not 0! so the first element is accessed as `A[1]`, etc.

Here are some other ways you can index as well. You can access multiple indices at the same time using a numeric vector of indices:

```
V[c(1, 2, 5)] # access elements 1, 2, and 5
```

```
[1] "A" "B" "E"
```

If you need to access several indices in a row, you can use a colon (`:`):

```
V[2:7] # access elements 2 through 7
```

```
[1] "B" "C" "D" "E" "F" "G"
```

You can even combine these two methods:

```
V[c(1:3, 6)] # access elements 1, 2, 3, and 6
```

```
[1] "A" "B" "C" "F"
```

Note that the following are all equivalent ways to access the first three elements of `V`:

- `V[1:3]`
- `V[c(1,2,3)]`
- `V[c(1:3)]`
- `V[c(1:2,3)]`
- can you think of another example?

But the first way would probably be the most clear for someone else to understand. All of these methods can work with *assignment* as well:

```
V[c(1, 7:9)] <- "X" # change elements 1, 7, 8, and 9 to "X"
V
```

```
[1] "X" "B" "C" "D" "E" "F" "X" "X" "X"
```

⚠ **Caution** Even though these examples use a character vector, this indexing works on vectors of any type.

### 5.5.2 Matrices

To access an element of a matrix, we have to specify the row and the column. Let's create a matrix from the V vector and access one of its elements:

```
M <- matrix(V, 3, 3) # create matrix M with data from vector V
M
```

```
      [,1] [,2] [,3]
[1,] "X"  "D"  "X"
[2,] "B"  "E"  "X"
[3,] "C"  "F"  "X"
```

```
M[1,2] # access the element in row 1, column 2
```

```
[1] "D"
```

Recall that we can access an entire row or column by leaving the *other* index blank:

```
M[1,] # access the entire first row
```

```
[1] "X" "D" "X"
```

```
M[,2] # access the entire second column
```

```
[1] "D" "E" "F"
```

But any of the indexing we just used for vectors can also be used on matrices

```
M[1:2, c(2, 3)] # access the elements in rows 1 and 2, columns 2 and 3.
```

```
      [,1] [,2]
[1,] "D"  "X"
[2,] "E"  "X"
```

Finally, there is one more way of indexing Matrices (for now), that provides only *one* index:

```
M[5] # access the "5th" element of the matrix
```

```
[1] "E"
```

If you give one index, then R will count down the first row, then the second, then the third, etc., until it reaches the index you specified. Notice how this agrees with the 5th element of the vector `V`, which was used to make our matrix! And finally, as before, any of these indexing methods can be used to change an element's value:

```
M[2, 1:3] <- "Hats"  
M
```

```
      [,1] [,2] [,3]  
[1,] "X"  "D"  "X"  
[2,] "Hats" "Hats" "Hats"  
[3,] "C"  "F"  "X"
```

### 5.5.3 Lists

So far we've discussed three different ways of accessing elements in a list:

```
L <- list(A = "apple", b = "banana", c="cherry")
```

```
L[[1]] # access using index number
```

```
[1] "apple"
```

```
L[["b"]] # access using component name
```

```
[1] "banana"
```

```
L$c # access using component name and dollar sign notation
```

```
[1] "cherry"
```

And these are basically the only option. Unfortunately, you *cannot* use a vector of indices in order to access multiple list components at once:

```
L[[1:3]] # this does not work
```

☀ **Bonus** What `L[[1:3]]` *actually* does (as the error message might suggest), is access elements within a *nested* list, but that is beyond the scope of this class.

🦋 **Progress Check** Create a vector containing the numbers 1 through 1000 in order (hint: try using `1:1000` on the right of the assignment operator). Then, change elements 4, 196, and 501 through 556 to “brussels sprouts”. What happened to the other elements in the vector?

### 5.5.4 Data Frames

Remember that data frames are just lists of vectors, so the same indexing rules of lists and vectors apply. But remember that matrix indexing rules also apply! Here are some examples with the Olympic athletes data.

```
athletes3 <- athletes[1:20,1:5] # get the first 20 rows and first 5 columns, and assign it to athletes3
```


```
athletes3$Name # get the Name column
```

```
[1] "Berta Hrub"  
[2] "Joaquim Vital"  
[3] "Madelon Baans"  
[4] "Achille Canna"  
[5] "Antje Buschschulte (-Meeuw)"  
[6] "Ludwig Gredler"  
[7] "Pawe Abratkiewicz"  
[8] "Jerzy Zdzisaw Janikowski"  
[9] "Giuseppe \"Peppino\" Tanti"  
[10] "Carl-Jan Gustaf David Hamilton"  
[11] "Bla Nagy"  
[12] "Vincent Vittoz"  
[13] "Joyce May Racek (-Markley, -Budrunas)"  
[14] "Seiichiro Kashio"  
[15] "Surzer"  
[16] "Dimitrios Kantzias"  
[17] "Kim Gwang-Suk"  
[18] "Joshua Noel \"Josh\" Laban"  
[19] "Alejandro Vidal Arellano"  
[20] "Mariusz Latkowski"
```

Remember that each column of a data frame is just a vector, so we can use list indexing to access the `Name` column, then *immediately* use vector indexing to get only the indices that we want:

```
athletes3$Name[1:3] # get the first three elements of the Name column
```

```
[1] "Berta Hrub" "Joaquim Vital" "Madelon Baans"
```

 **Caution** Notice how With lists, you *cannot* access multiple components (which is what dataframe columns are) at the same time, but with matrices you *can* access multiple columns at once. Since data frames can use matrix formatting, you *can* select multiple columns at once, as the first example above showed.

You can also access columns by name like so:

```
athletes3[,c("Name", "Sex")] # Access Name and Sex columns
```

	Name	Sex
1	Berta Hrub	F
2	Joaquim Vital	M
3	Madelon Baans	F
4	Achille Canna	M
5	Antje Buschschulte (-Meeuw)	F
6	Ludwig Gredler	M
7	Pawe Abratkiewicz	M
8	Jerzy Zdzisaw Janikowski	M
9	Giuseppe "Peppino" Tanti	M
10	Carl-Jan Gustaf David Hamilton	M
11	Bla Nagy	M
12	Vincent Vittoz	M
13	Joyce May Racek (-Markley, -Budrunas)	F
14	Seiichiro Kashio	M
15	Surzer	M
16	Dimitrios Kantzias	M
17	Kim Gwang-Suk	F
18	Joshua Noel "Josh" Laban	M
19	Alejandro Vidal Arellano	M
20	Mariusz Latkowski	M

(and if your rows have names, you can access rows by name as well).

🔗 **Progress Check** Using the `mtcars` data frame (included in R), get the `mpg` for the cars in rows 15 through 20, and assign it to a vector. Now find the average `mpg` of those cars.

☀️ **Bonus** Think it's weird that data frames can be indexed like matrices? It gets weirder. When vectors have names, they can be indexed like lists! Try for yourself: create a vector `a <- c(1, 2, 3)` and set the names with `names(a) <- c("angus", "brillow", "chandelier")`, then see what happens if you type `a[["angus"]]`! Matrices can also be accessed using names as well.

### 5.5.5 Advanced Indexing

There are *even more* ways to select the data you need from your R data structures, let's look at some more advanced techniques.

**5.5.5.1 Logical Based Indexing** One *very* useful method that R provides is to access elements of a vector using a different, logical vector of the same length. As the following example will show, R will give only the elements which are true in the logical vector:

```
v <- c("alpha", "bravo", "charlie", "delta") # the vector we want to access
i <- c(FALSE, TRUE, FALSE, TRUE) # the logical vector we'll use to index.

# index v using i:
v[i]
```

```
[1] "bravo" "delta"
```

Why is this so useful? Remember that you can create logical vectors by comparing any type of vector to some value:

```
v == "delta"
```

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

This means you can create a logical vector in order to extract only the elements of a vector which match some criterion. For example, let's create a logical vector based on whether an Olympic athlete's sport was "Tug-Of-War".

```
plays_tug_of_war <- athletes$Sport == "Tug-Of-War" # create logical vector  
sum(plays_tug_of_war) # count how many TRUE's
```

```
[1] 5
```

Now let's use that logical vector to get the names of the athletes:

```
athletes$Name[plays_tug_of_war]
```

```
[1] "Edgar Lindenau Aabye" "Willie Slade"          "William Hiron"  
[4] "Ernest Walter Ebbage" "William Penn"
```

🔗 **Progress Check** Using the Olympic athletes data, create a logical vector which is true when an athlete's sport is wrestling. Then access the age of all wrestlers, and assign the ages to another vector. Finally, compute the average age of the wrestlers vector (remember, there may be duplicate athlete names, so this average won't mean much; the emphasis is on indexing right now)

Logical vectors can also be used to subset a data frame based on some condition. That is, we take entire rows which meet a condition, rather than just a single variable. For example:

```
# Subset the athletes data frame to get only Summer athletes.  
athletes_summer <- athletes[athletes$Season == "Summer",]
```

⚠️ **Caution** In the last example, we are creating the logical vector and immediately using it to index the rows. Pause and think through what's happening in this example if it's not quite clear. Also note the placement of the comma (,), which indicates that we're indexing rows, not columns, of the data frame.

You can specify multiple conditions using "and" (&) and "or" (|) like this: