Alice Paul

Mastering Health Data Science Using R

To blah, blah, and blah.

Table of contents

Preface

This book serves as an interactive introduction to R for public health and health data science students. Topics include data structures in R, exploratory analysis, distributions, hypothesis testing, and regression analysis. The presentation assumes knowledge with the underlying methodology and focuses instead on how to use R to implement your analysis.

This book is written using Quarto Book. You can download the Quarto files used to generate this book or a corresponding Jupyter notebook from the github repository[1]. The github repository also contains a few cheat sheets[2] including a pdf containing all functions[3] shown in this book.

Acknowledgments

This book was written with the support of a Data Science Institute Seed Grant[5]. Thanks to students Jialin Liu, Joanna Walsh, and Xinbei Yu for their help and feedback. Please contact Dr. Paul (alice_paul@brown.edu) with questions, suggested edits, or feedback.

[1] https://github.com/alicepaul/health-data-science-using-r

[2] https://github.com/alicepaul/health-data-science-using-r/tree/main/book/refs

[3] https://github.com/alicepaul/health-data-science-using-r/blob/main/book/refs/functions_cheat_sheet.pdf

[4] https://creativecommons.org/licenses/by/4.0/

[5] https://dsi.brown.edu/

Part I

Introduction to R

1

Getting Started with R

This chapter will introduce you to R as a programming language and show you how we can use this language in two different ways: directly through the R console and using the RStudio development environment. To start, you will need to download R[1] and RStudio[2].

1.1 Why R?

What are some of the benefits of using R?

- R is built for statisticians and data analysts.
- R is open source.
- R has most of the latest statistical methods available.
- R is flexible.

Since R is built for statisticians, it is built with data in mind. This comes in handy when we want to streamline how we process and analyze data. It also means that many statisticians working on new methods are publishing user-created packages in R, so R users have access to most methods of interest. R is also an interpreted language, which means that we do not have to compile our code into machine language first: this allows for simpler syntax and more flexibility when writing our code, which also makes it a great first programming language to learn.

Python is another interpreted language often used for data analysis. Both languages feature simple and flexible syntax, but while python is more broadly developed for usage outside of data science and statistical analyses, R is a great programming language for those in health data science. I use both languages and find switching between them to be straightforward, but I do prefer R for anything related to data or statistical analysis.

[1] https://cran.rstudio.com/

[2] https://posit.co/download/rstudio-desktop/

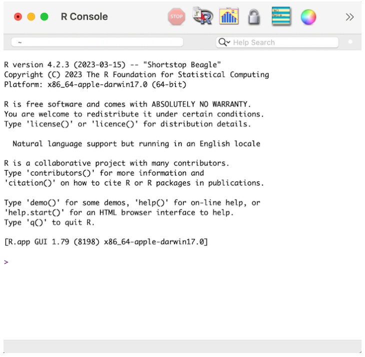
1.1.1 Installation of R and RStudio

To run R on your computer, you will need to download and install R[3]. This will allow you to open the R application and run R code interactively. However, to get the most out of programming with R, you will want to install RStudio, which is an integrated development environment (IDE) for R and python. RStudio offers a nice environment for writing, editing, running, and debugging R code. We will talk through more of the benefits of using RStudio.

Each chapter in this book is written as a Quarto document and can also be downloaded as a Jupyter notebook. You can open Quarto files in RStudio to run the code as you read and complete the practice questions and exercises.

1.2 The R Console

The R console provides our first intro to code in R. Figure **??** shows what the console will look like when you open it. You should see a blinking cursor - this where we can write our first line of code!



```
R version 4.2.3 (2023-03-15) -- "Shortstop Beagle"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.79 (8198) x86_64-apple-darwin17.0]

>
```

Figure 1.1: The R Console.

To start, type `2+3` and press ENTER. You should see that `5` is printed below that code and that your cursor is moved to the next line.

1.2.1 Basic Computations and Objects

In the example above, we coded a simple addition. Try out some other basic calculations using the following operators:

[3] https://cran.rstudio.com/

- Addition: `5+6`
- Subtraction: `7-2`
- Multiplication: `2*3`
- Division: `6/3`
- Exponentiation: `4^2`
- Modulo: `100 %% 4`

For example, use the modulo operator to find what 100 mod 4 is. It should return 0 since 100 is divisible by 4.

If we want to save the result of any computation, we need to create an object to store our value of interest. An **object** is simply a named data structure that allows us to reference that data structure. Objects are also commonly called **variables**. In the code below, we create an object `x` which stores the value `5` using the assignment operator `<-`. The assignment operator assigns whatever is on the right hand side of the operator to the name on the left hand side. We can now reference `x` by calling its name. Additionally, we can update its value by adding 1. In the second line of code, the computer first finds the value of the right hand side by finding the current value of `x` before adding 1 and assigning it back to `x`.

```
x <- 2+3
x <- x+1
x
#> [1] 6
```

We can create and store multiple objects by using different names. The code below creates a new object `y` that is one more than the value of `x`. We can see that the value of `x` is still `5` after running this code.

```
x <- 2+3
y <- x
y <- y + 1
x
#> [1] 5
```

1.2.2 Naming Conventions

As we start creating objects, we want to make sure we use good object names. Here are a few tips for naming objects effectively:

- Stick to a single format. We will use **snake_case**, which uses underscores between words (e.g. `my_var`, `class_year`).

- Make your names useful. Try to avoid using names that are too long (e.g. `which_day_of_the_week`) or do not contain enough information (e.g., `x1`, `x2`, `x3`).

- Replace unexplained values with an object. For example, if you need to do some calculations using 100 as the number of participants, create an object `n_part` with value 100 rather than repeatedly using the number. This makes the code easy to update and helps the user avoid possible errors.

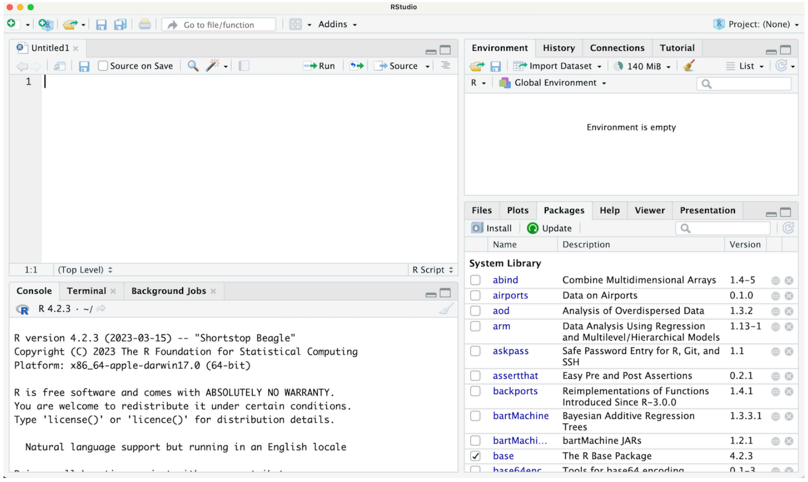
1.3 RStudio and R Markdown

If we made a mistake in the code above, we would have to retype everything from the beginning. However, when we write code, we often want to be able to run it multiple times and develop it in stages. R scripts and R markdown files allow us to save all of our R code in files that we can update and re-run, which allows us to create reproducible and shareable analyses. We will now move to RStudio as our development environment to demonstrate creating an R script. When you open RStudio, you will see multiple windows. Start by opening a new R file by going to File -> New File -> R Script. You should now see several windows as shown in Figure **??**.

In the code editor window in the top left, add the following code to your .R file and save the file. Note that here we used snake_case to name our objects!

```r
# Calculate student to faculty ratio, 2023 enrollment
num_students <- 132
num_faculty <- 23
student_fac_ratio <- num_students/num_faculty
```

The first line starts with `#` and does not contain any code. This is a comment line, which allows us to add context, intent, or extra information to help the reader understand our code. A good rule of thumb is that we want to write enough comments so that we could open our code in six months and be able



```
R version 4.2.3 (2023-03-15) -- "Shortstop Beagle"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale
```

	Name	Description	Version
	System Library		
□	abind	Combine Multidimensional Arrays	1.4-5
□	airports	Data on Airports	0.1.0
□	aod	Analysis of Overdispersed Data	1.3.2
□	arm	Data Analysis Using Regression and Multilevel/Hierarchical Models	1.13-1
□	askpass	Safe Password Entry for R, Git, and SSH	1.1
□	assertthat	Easy Pre and Post Assertions	0.2.1
□	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.4.1
□	bartMachine	Bayesian Additive Regression Trees	1.3.3.1
□	bartMachi...	bartMachine JARs	1.2.1
☑	base	The R Base Package	4.2.3
□	base64enc	Tools for base64 encoding	0.1-3

Figure 1.2: RStudio Layout and Panes.

to understand what we were doing. As we develop longer chunks of code, this will become more important.

1.3.1 Calling Functions

When we use R, we have access to all the functions available in base R. A **function** takes in one or more inputs and returns a single output object. Let's first use the simple function `exp()`. This exponential function takes in one (or more) numeric values and exponentiates them. The code below computes e^3.

```
exp(3)
#> [1] 20.1
```

Some other simple functions are shown below that all convert a numeric input to an integer value. The `ceiling()` and `floor()` functions returns the ceiling and floor of your input, and the `round()` function will round your input to the closest integer. Note that the `round()` function will round a 5 to the closest even integer.

```
ceiling(3.7)
#> [1] 4
```

```
floor(3.7)
#> [1] 3
```

```
round(2.5)
#> [1] 2
round(3.5)
#> [1] 4
```

If we want to learn about a function, we can use the help operator `?` by typing it in front of the function you are interested in: this will bring up the documentation for that particular function. This documentation will often tell you the usage of the function, the **arguments** (the object inputs), the **value** (information about the returned object), and will give some examples of how to use the function. For example, if we want to understand the difference between `floor()` and `ceiling()`, we can call `?floor` and `?ceiling`. This should bring up the documentation in your help window. We can then read that the floor function rounds a numeric input down to the nearest integer whereas the ceiling function rounds a numeric input up to the nearest integer.

1.3.2 Working Directories and Paths

Let's try using another example function: `read.csv()`. This function reads in a comma-delimited file and returns the information as a data frame (try typing `?read.csv` in the console to read more about this function). We will learn more about data frames in Chapter **??**. The first argument to this function is a file, which can be expressed as either a filename or a path to a file. First, download the file `fake_names.csv` from this book's github repository[4]. By default, R will look for the file in your current working directory. To find the working directory, you can run `getwd()`. You can see below that my current working directory is where the book content is on my computer.

```
getwd()
#> [1] "/Users/Alice/Dropbox/health-data-science-using-r/book"
```

You can either move the .csv file to your current working directory and load it in, or you can specify the path to the .csv file. Another option is to update your working directory by using the `setwd()` function.

```
setwd('/Users/Alice/Dropbox/health-data-science-using-r/book/data')
```

If you receive an error that a file cannot be found, you most likely have the

[4] https://github.com/alicepaul/health-data-science-using-r/tree/main/book/data

wrong path to the file or the wrong file name. Below, I chose to specify the path to the downloaded .csv file, saved this file to an object called `df`, and then printed that `df` object.

```
# update this with the path to your file
df <- read.csv("data/fake_names.csv")
df
#>                  Name Age     DOB            City State
#> 1           Ken Irwin  37 6/28/85      Providence    RI
#> 2 Delores Whittington  56 4/28/67      Smithfield    RI
#> 3       Daniel Hughes  41 5/22/82      Providence    RI
#> 4         Carlos Fain  83  2/2/40          Warren    RI
#> 5        James Alford  67 2/23/56 East Providence    RI
#> 6        Ruth Alvarez  34 9/22/88      Providence    RI
```

We can see that `df` contains the information from the .csv file and that R has printed the first few observations of the data.

1.3.3 Installing and Loading Packages

When working with data frames, we will often use the **tidyverse** package (Wickham 2023), which is actually a collection of R packages for data science applications. An R package is a collection of functions and/or sample data that allow us to expand on the functionality of R beyond the base functions. You can check whether you have the **tidyverse** package installed by going to the package pane in RStudio or by running the command below, which will display all your installed packages.

```
installed.packages()
```

If you don't already have a package installed, you can install it using the `install.packages()` function. Note that you have to include single or double quotes around the package name when using this function. You only have to install a package one time.

```
install.packages('tidyverse')
```

The function `read_csv()` is another function to read in comma-delimited files that is part of the **readr** package in the tidyverse (Wickham, Hester, and Bryan 2023). However, if we tried to use this function to load in our data, we would get an error that the function cannot be found. That is because we haven't loaded in this package. To do so, we use the `library()` function. Unlike the `install.packages()` function, we do not have to use quotes around the package name when calling this `library()` function. When we load in a package, we will see some messages. For example, below we see that this package contains the functions `filter()` and `lag()` that are also functions in base R. In future chapters, we will suppress these messages to make the chapter presentation nicer. After loading the **tidyverse** package, we can now use the `read_csv()` function as shown below.

```
library(tidyverse)
```

```
df <- read_csv("data/fake_names.csv", show_col_types=FALSE)
df
#> # A tibble: 6 x 5
#>   Name                   Age DOB     City            State
#>   <chr>                <dbl> <chr>   <chr>           <chr>
#> 1 Ken Irwin               37 6/28/85 Providence      RI
#> 2 Delores Whittington     56 4/28/67 Smithfield      RI
#> 3 Daniel Hughes           41 5/22/82 Providence      RI
#> 4 Carlos Fain             83 2/2/40  Warren          RI
#> 5 James Alford            67 2/23/56 East Providence RI
#> # i 1 more row
```

Alternatively, we could have told R where to locate the function by adding `readr::` before the function. This tells it to find `read_csv()` function in the **readr** package. This can be helpful even if we have already loaded in the package, since sometimes multiple packages have functions with the same name.

```
df <- readr::read_csv("data/fake_names.csv", show_col_types = FALSE)
```

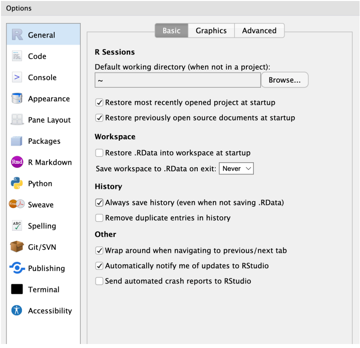
1.3.4 RStudio Global Options

You have now had a basic tour of RStudio. We highly recommend that you update your RStudio options to not save your workspace on exiting or load it on starting. This will ensure that you create fully reproducible code and avoid possible errors or confusion.

1.4 Tips and Reminders

We end this chapter with some final tips and reminders.

- **Keyboard Shortcuts**: RStudio has several useful keyboard shortcuts that will make your programming experience more streamlined. It is worth



R Sessions

Default working directory (when not in a project): ~

☑ Restore most recently opened project at startup

☑ Restore previously open source documents at startup

Workspace

□ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

History

☑ Always save history (even when not saving .RData)

□ Remove duplicate entries in history

Other

☑ Wrap around when navigating to previous/next tab

☑ Automatically notify me of updates to RStudio

□ Send automated crash reports to RStudio

Figure 1.3: RStudio Global Options.

getting familiar with some of the most common keyboard shortcuts using this book's cheatsheet[5].

- **Asking for help**: Within R, you can use the `?` operator or the `help()` function to pull up documentation on a given function. This documentation is also available online[6].
- **Finding all objects**: You can use the Environment panel or `ls()` function to find all current objects. If you have an error that an object you are calling does not exist, take a look to find where you defined it.
- **Checking packages**: If you get an error that a function does not exist, check to make sure you have loaded that package using the `library()` function. The list of packages used in this book is given on the github repository homepage.

[5] https://github.com/alicepaul/health-data-science-using-r/blob/main/book/refs/r_studio_keyboard_shortcuts.pdf

[6] https://rdocumentation.org/

2

Data Structures in R

In this chapter, we will demonstrate the key **data structures** in R. Data structures are how information is stored in R, and the data structures that we use inform R how to interpret our code. Any **object** is a named instance of a data structure. For example, the object `ex_num` below is a vector of numeric type.

```
ex_num <- 4
```

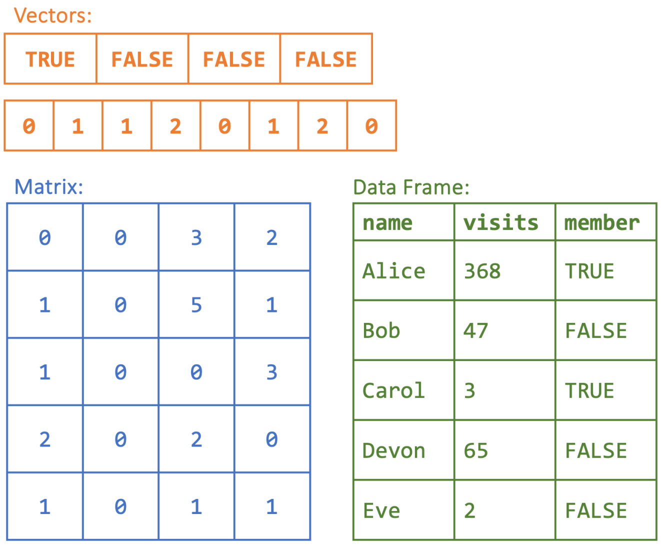
The main data structures in R are **vectors**, **factors**, **matrices**, **arrays**, **lists**, and **data frames**. These structures are distinguished by their dimensions and by the type of data they store. For example, we might have a 1-dimensional vector that contains all numeric values, or we could have a 2-dimensional data frame with rows and columns where we might have one numeric column and one character column. In Figure **??**, there are two vectors with different types (character and numeric) on top and then a matrix and data frame below. In this chapter, we will cover each data structure except for arrays. Arrays are an extension of matrices that allow for data that is more than 2-dimensional and are not needed for the applications covered in this book.

2.1 Data Types

Each individual value in R has a type: **logical**, **integer**, **double**, or **character**. We can think of these as the building blocks of all data structures. Below, we can use the `typeof()` function to find the type of our vector from above, which shows that the value of `ex_num` is a **double**. A double is a numeric value with a stored decimal.

```
typeof(ex_num)
#> [1] "double"
```

On the other hand, an **integer** is a whole number that does not contain a



Vectors:

TRUE	FALSE	FALSE	FALSE

0	1	1	2	0	1	2	0

Matrix:

0	0	3	2
1	0	5	1
1	0	0	3
2	0	2	0
1	0	1	1

Data Frame:

name	visits	member
Alice	368	TRUE
Bob	47	FALSE
Carol	3	TRUE
Devon	65	FALSE
Eve	2	FALSE

Figure 2.1: Data structures.

decimal. We now create an integer object `ex_int`. To indicate to R that we want to restrict our values to integer values, we use an `L` after the number.

```r
ex_int <- 4L
typeof(ex_int)
#> [1] "integer"
```

Both `ex_num` and `ex_int` are numeric objects, but we can also work with two other types of objects: **characters** (e.g. “php”, “stats”) and **booleans** (e.g. TRUE, FALSE), also known as **logicals**.

```r
ex_bool <- TRUE
ex_char <- "Alice"

typeof(ex_bool)
#> [1] "logical"
typeof(ex_char)
#> [1] "character"
```

One important characteristic of logical objects is that R will also interpret them as 0/1. This means they can be added as in the example below: each `TRUE` has a value of 1 and each `FALSE` has a value of 0.

```
TRUE+FALSE+TRUE
#> [1] 2
```

To create all of the above objects, we used the assignment operator `<-`, which we discussed in Chapter **??**. You may see code elsewhere that uses an `=` instead. While `=` can also be used for assignment, it is more standard practice to use `<-`.

2.2 Vectors

In the examples above, we created objects with a single value. R actually uses a vector of length 1 to store this information. **Vectors** are 1-dimensional data structures that can store multiple data values of the same type (e.g. character, boolean, or numeric).

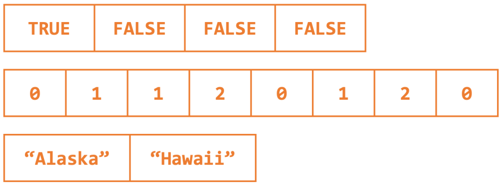


Figure 2.2: Vector Examples.

We can confirm this by using the `is.vector()` function, which returns whether or not the inputted argument is a vector.

```
is.vector(ex_bool)
#> [1] TRUE
```

One way to create a vector with multiple values is to use the combine function `c()`. Below we create two vectors: one with the days of the week and one with the amount of rain on each day. The first vector has all character values, and the second one has all numeric values.

```
days <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
rain <- c(5, 0.1, 0, 0, 0.4)
```

Remember, vectors cannot store objects of different types. Because of this, in the code below, R automatically converts the numeric value to be a character in order to store these values in a vector together.

```
c("Monday", 5)
#> [1] "Monday" "5"
```

The `class()` function returns the data structure of an object. If we check the classes of these two objects using the `class()` function, we will see that R tells us that the first is a character vector and the second is a numeric vector. This matches the data type in this case.

```
class(days)
#> [1] "character"
class(rain)
#> [1] "numeric"
```

What happens when we create an empty vector? What is the class?

```
ex_empty <- c()
class(ex_empty)
#> [1] "NULL"
```

In this case, there is no specified type yet. If we wanted to specify the type, we could make an empty vector using the `vector()` function.

```
ex_empty <- vector(mode = "numeric")
class(ex_empty)
#> [1] "numeric"
```

Another way to create a vector is with the `rep()` or `seq()` functions. The first function `rep(x, times)` takes in a vector `x` and a number of times `times` and outputs `x` repeated that many times. Let's try this with a single value below. The second function `seq(from, to, step)` takes in a numeric starting value `from`, end value `to`, and step size `step` and returns a sequence from `from` in increments of `step` until a maximum value of `to` is reached.

```
rep(0, 5)
#> [1] 0 0 0 0 0
rep("Monday", 4)
#> [1] "Monday" "Monday" "Monday" "Monday"
seq(1, 5, 1)
```

```
#> [1] 1 2 3 4 5
seq(0, -10, -2)
#> [1]   0  -2  -4  -6  -8 -10
```

2.2.1 Indexing a Vector

Once we have a vector, we may want to access certain values stored in that vector. To do so, we index the vector using the position of each value: the first value in the vector has index 1, the second value has index 2, etc. When we say a vector is 1-dimensional, we mean that we can define the position of each value by a single index. To index the vector, we then use square brackets `[]` after the vector name and provide the position. Below, we use these indices to find the value at index 1 and the value at index 4.

```
days[1]
#> [1] "Monday"
days[4]
#> [1] "Thursday"
```

We can either access a single value or a subset of values using a vector of indices. Let's see what happens when we use a vector of indices `c(1,4)` and then try using `-c(1,4)` and see what happens then. In the first case, we get the values at index 1 *and* at index 4. In the second case, we get all values *except* at those indices. The `-` indicates that we want to remove rather than select these indices.

```
days[c(1,4)]
#> [1] "Monday"   "Thursday"
days[-c(1,4)]
#> [1] "Tuesday"   "Wednesday" "Friday"
```

However, always indexing by the index value can sometimes be difficult or inefficient. One extra feature of vectors is that we can associate a name with each value. Below, we update the names of the vector `rain` to be the days of the week and then find Friday's rain count by indexing with the name.

```
names(rain) <- days
print(rain)
#>    Monday   Tuesday Wednesday  Thursday    Friday
#>       5.0       0.1       0.0       0.0       0.4
rain["Friday"]
```

```
#> Friday
#>    0.4
```

The last way to index a vector is to use TRUE and FALSE values. If we have a vector of booleans that is the same length as our original vector, then this will return all the values that correspond to a TRUE value. For example, indexing the `days` vector by the logical vector `ind_bools` below will return its first and fourth values. We will see more about using logic to access certain values later on.

```
ind_bools <- c(TRUE, FALSE, FALSE, TRUE, FALSE)
days[ind_bools]
#> [1] "Monday"   "Thursday"
```

2.2.2 Editing a Vector and Calculations

The mathematical operators we saw in the last chapter (`+`, `-`, `*`, `/`, `^`, `%%`) can all be applied to numeric vectors and will be applied element-wise. That is, in the code below, the two vectors are added together by index. This holds true for some of the built-in math functions as well:

- `exp()` - exponential
- `log()` - log
- `sqrt()` - square root
- `abs()` - absolute value
- `round()` - round to nearest integer value
- `ceiling()` - round up to the nearest integer value
- `floor()` - round down to the nearest integer value

```
c(1,2,3) + c(1,1,1)
#> [1] 2 3 4
c(1,2,3) + 1 # equivalent to the code above
#> [1] 2 3 4
sqrt(c(1,4,16))
#> [1] 1 2 4
```

After we create a vector, we may need to update its values. For example, we may want to change a specific value. We can do so using indexing. Below, we update the rain value for Friday using the assignment operator.

```
rain["Friday"] <- 0.5
rain
#>    Monday   Tuesday Wednesday  Thursday    Friday
#>       5.0       0.1       0.0       0.0       0.5
```

Further, we may need to add extra entries. We can do so using the `c()` function again but this time passing in the vector we want to add to as our first argument. This will create a single vector with all previous and new values. Below, we add two days to both vectors and then check the length of the updated vector `rain`. The `length()` function returns the length of a vector.

```
length(rain)
#> [1] 5
days <- c(days,"Saturday","Sunday") # add the weekend with no rain
rain <- c(rain,0,0)
length(rain)
#> [1] 7
```

We can also call some useful functions on vectors. For example, the `sum()`, `max()`, and `min()` functions will return the sum, maximum value, and minimum value of a vector, respectively.

2.2.3 Practice Question

Create a vector of the odd numbers from 1 to 11 using the `seq()` function. Then, find the third value in the vector using indexing, which should have value 5.

```
# Insert your solution here:
```

2.2.4 Common Vector Functions

Below we list some of the most common vector functions that are available in base R. All of these functions assume that the vector is numeric. If we pass the function a logical vector, R will convert the vector to 0/1 first, and if we pass the function a character vector, R will give us an error message.

- `sum()` - summation
- `median()` - median value
- `mean()` - mean
- `sd()` - standard deviation
- `var()` - variance

- `max()` - maximum value
- `which.max()` - index of the first element with the maximum value
- `min()` - minimum value
- `which.min()` - index of the first element with the minimum value

Try these out using the vector `rain`. Note that R is case sensitive - `Mean()` is considered different from `mean()`, so if we type `Mean(rain)` R will tell us that it cannot find this function.

```r
mean(rain)
#> [1] 0.8
min(rain)
#> [1] 0
which.min(rain)
#> Wednesday
#>         3
```

We may also be interested in the order of the values. The `sort()` function sorts the values of a vector, whereas the `order()` function returns the permutation of the elements to be in sorted order. The last line of code below sorts the days of the week from smallest to largest rain value.

```r
rain
#>    Monday   Tuesday Wednesday  Thursday    Friday
#>       5.0       0.1       0.0       0.0       0.5       0.0       0.0
order(rain)
#> [1] 3 4 6 7 2 5 1
days[order(rain)]
#> [1] "Wednesday" "Thursday"  "Saturday"  "Sunday"    "Tuesday"
#> [6] "Friday"    "Monday"
```

Both of these functions have an extra possible argument `decreasing`, which has a default value of FALSE. We can specify this to be TRUE to find the days of the week sorted from largest to smallest rainfall.

```r
days[order(rain, decreasing=TRUE)]
#> [1] "Monday"    "Friday"    "Tuesday"   "Wednesday" "Thursday"
#> [6] "Saturday"  "Sunday"
```

2.3 Factors

A **factor** is a special kind of vector that behaves exactly like a regular vector except that it represents values from a category. In particular, a factor keeps track of all possible values of that category, which are called the **levels** of the factor. Factors will be especially helpful when we start getting into data analysis and have categorical columns. The `as.factor()` function will convert a vector to a factor.

```
days <- c("Monday", "Tuesday", "Wednesday", "Monday",
          "Thursday", "Wednesday")
days_fct <- as.factor(days)

class(days_fct)
#> [1] "factor"
levels(days_fct)
#> [1] "Monday"    "Thursday"  "Tuesday"   "Wednesday"
```

Above, we did not specify the possible levels for our column. Instead, R found all values in the vector `days` and set these equal to the levels of the factor. Because of this, if we try to change one of the levels to 'Friday', we will get an error. Uncomment the line below to see the error message.

```
#days_fct[2] <- "Friday"
```

We can avoid this error by specifying the levels using the `factor()` function instead of the `as.factor()` function.

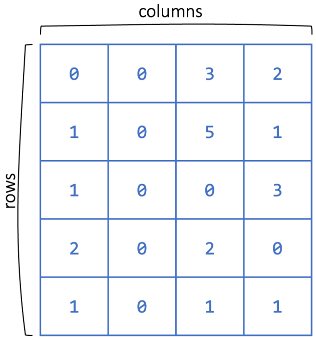
```
days <- c("Monday", "Tuesday", "Wednesday", "Monday", "Thursday",
          "Wednesday")
days_fct <- factor(days,
               levels = c("Monday", "Tuesday", "Wednesday",
                          "Thursday", "Friday", "Saturday", "Sunday"))

class(days_fct)
#> [1] "factor"
levels(days_fct)
#> [1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"
#> [6] "Saturday"  "Sunday"
days_fct[2] <- "Friday"
```

Factors can also be used for numeric vectors. For example, we might have a vector that is 0/1 that represents whether or not a day is a weekend. This can also only take on certain values (0 or 1).

```
weekend <- as.factor(c(1, 0, 0, 0, 1, 1))
levels(weekend)
#> [1] "0" "1"
```

2.4 Matrices

Matrices are similar to vectors in that they store data of the same type, but matrices are two-dimensional (as opposed to one-dimensional vectors), and consist of both rows and columns.



0	0	3	2
1	0	5	1
1	0	0	3
2	0	2	0
1	0	1	1

Figure 2.3: Matrix Example.

Below, we create a matrix reporting the daily rainfall over multiple weeks. We can create a matrix using the `matrix(data, nrow, ncol, byrow)` function. This creates a `nrow` by `ncol` matrix from the vector `data` values filling in by row if `byrow` is TRUE and by column otherwise. Run the code below. Then, change the last argument to `byrow=FALSE` and see what happens to the values.

```
rainfall <- matrix(c(5,6,0.1,3,0,1,0,1,0.4,0.2,0.5,0.3,0,0),
                   ncol=7, nrow=2, byrow=TRUE)
rainfall
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
#> [1,]    5  6.0  0.1  3.0  0.0    1    0
#> [2,]    1  0.4  0.2  0.5  0.3    0    0
```

We can find the dimensions of a matrix using the `nrow()`, `ncol()`, or `dim()` functions, which return the number of rows, the number of columns, and both the number of rows and columns, respectively.

```
nrow(rainfall)
#> [1] 2
ncol(rainfall)
#> [1] 7
dim(rainfall)
#> [1] 2 7
```

2.4.1 Indexing a Matrix

Since matrices are two-dimensional, a single value is indexed by both its row number and its column number. This means that to access a subset of values in a matrix, we need to provide row and column indices. Below, we access a single value in the 1st row and the 4th column. The first value is always the row index and the second value is always the column index.

```
rainfall[1,4]
#> [1] 3
```

As before, we can also provide multiple indices to get multiple values. Below, we choose multiple columns but we can also choose multiple rows (or multiple rows and multiple columns!).

```
rainfall[1,c(4,5,7)]
#> [1] 3 0 0
```

As with vectors, we can also use booleans to index a matrix by providing boolean values for the rows and/or columns. Note that below we give a vector for the row indices and no values for the columns. Since we did not specify any column indices, this will select all of them.

```
rainfall[c(FALSE, TRUE), ]
#> [1] 1.0 0.4 0.2 0.5 0.3 0.0 0.0
```

Let's do the opposite and select some columns and all rows.

```
rainfall[ ,c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE)]
#>      [,1] [,2]
#> [1,]    5  6.0
#> [2,]    1  0.4
```

As with vectors, we can specify row names and column names to access entries instead of using indices. The `colnames()` and `rownames()` functions allow us to specify the column and row names, respectively.

```
colnames(rainfall) <- c("Monday", "Tuesday", "Wednesday", "Thursday",
                        "Friday", "Saturday", "Sunday")
rownames(rainfall) <- c("Week1", "Week2")
rainfall["Week1",c("Friday","Saturday")]
#>   Friday Saturday
#>        0        1
```

2.4.2 Editing a Matrix

If we want to change the values in a matrix, we need to first index those values and then assign them the new value(s). Below, we change a single entry to be 3 and then update several values to all be 0. Note that we do not provide multiple 0's on the right-hand side, as R will infer that all values should be set to 0.

```
rainfall["Week1", "Friday"] <- 3
```

```
rainfall["Week1", c("Monday", "Tuesday")] <- 0
print(rainfall)
#>       Monday Tuesday Wednesday Thursday Friday Saturday Sunday
#> Week1      0     0.0       0.1      3.0    3.0        1      0
#> Week2      1     0.4       0.2      0.5    0.3        0      0
```

Further, we can append values to our matrix by adding rows or columns through the `rbind()` and `cbind()` functions. The first function appends a row (or multiple rows) to a matrix and the second appends a column (or multiple columns). Note that below I provide a row and column name when passing in the additional data. If I hadn't specified these names, then those rows and columns would not be named.

```
rainfall <- rbind(rainfall, "Week3" = c(0.4, 0.0, 0.0, 0.0, 1.2, 2.2,
                                         0.0))
rainfall <- cbind(rainfall, "Total" = c(7.1, 2.4, 3.8))
print(rainfall)
#>       Monday Tuesday Wednesday Thursday Friday Saturday Sunday Total
#> Week1    0.0     0.0       0.1      3.0    3.0      1.0      0   7.1
#> Week2    1.0     0.4       0.2      0.5    0.3      0.0      0   2.4
#> Week3    0.4     0.0       0.0      0.0    1.2      2.2      0   3.8
```

Here is an example where we bind two matrices by column. Note that whenever we bind two matrices together, we have to be sure that their dimensions are compatible and that they are of the same type.

```
A <- matrix(c(1,2,3,4), nrow=2)
B <- matrix(c(5,6,7,8), nrow=2)
C <- cbind(A, B)
C
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    3    5    7
#> [2,]    2    4    6    8
```

As with vectors, most mathematical operators (`+`, `-`, `*`, `/` etc.) are applied element-wise in R.

```
A+B
#>      [,1] [,2]
#> [1,]    6   10
#> [2,]    8   12
```

```
exp(C)
#>      [,1] [,2] [,3] [,4]
#> [1,] 2.72 20.1  148 1097
#> [2,] 7.39 54.6  403 2981
```

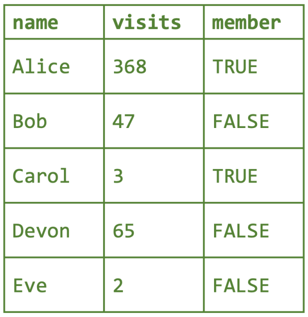
2.4.3 Practice Question

Create a 3x4 matrix of all 1's using the `rep()` and `matrix()` functions. Then select the first and third columns using indexing which will return a 3x2 matrix of all ones.

```
# Insert your solution here:
```

2.5 Data Frames

Matrices can store data like the rainfall data, where everything is of the same type. However, if we want to capture more complex data records, we also want to allow for different measurement types: this is where data frames come in. A data frame is like a matrix in that data frames are two-dimensional, but unlike matrices, data frames allow for each column to be a different type. In this case, each row corresponds to a single data entry (or observation) and each column corresponds to a different variable.



name	visits	member
Alice	368	TRUE
Bob	47	FALSE
Carol	3	TRUE
Devon	65	FALSE
Eve	2	FALSE

Figure 2.4: Data Frame Example.

For example, suppose that, for every day in a study, we want to record the temperature, rainfall, and day of the week. Temperature and rainfall can be numeric values, but day of the week will be character type. We create a data frame using the `data.frame()` function. Note that I am providing column names for each vector (column).

The `head()` function prints the first six rows of a data frame (to avoid printing very large datasets). In our case, all the data is shown because we only created four rows. The column names are displayed as well as their type. By contrast, the `tail()` function prints the last six rows of a data frame.

```
weather_data <- data.frame(day_of_week = c("Monday","Tuesday",
                                           "Wednesday", "Monday"),
                           temp = c(70,62,75,50),
                           rain = c(5,0.1,0.0,0.5))
head(weather_data)
#>   day_of_week temp rain
#> 1      Monday   70  5.0
#> 2     Tuesday   62  0.1
#> 3   Wednesday   75  0.0
#> 4      Monday   50  0.5
```

The `dim()`, `nrow()`, and `ncol()` functions return the dimensions, number of rows, and number of columns of a data frame, respectively.

```
dim(weather_data)
#> [1] 4 3
nrow(weather_data)
#> [1] 4
ncol(weather_data)
#> [1] 3
```

The column names can be found (or assigned) using the `colnames()` or `names()` function. These were specified when I created the data. On the other hand, the row names are currently the indices.

```
colnames(weather_data)
#> [1] "day_of_week" "temp"        "rain"
rownames(weather_data)
#> [1] "1" "2" "3" "4"
names(weather_data)
#> [1] "day_of_week" "temp"        "rain"
```

We update the row names to be more informative as with a matrix using the `rownames()` function.

```
rownames(weather_data) <- c("6/1", "6/2", "6/3", "6/8")
head(weather_data)
#>     day_of_week temp rain
#> 6/1      Monday   70  5.0
#> 6/2     Tuesday   62  0.1
#> 6/3   Wednesday   75  0.0
#> 6/8      Monday   50  0.5
```

2.5.1 Indexing a Data Frame

We can select elements of the data frame using its indices in the same way as we did with matrices. Below, we access a single value and then a subset of our data frame. The subset returned is itself a data frame. Note that the second line below returns a data frame.

```
weather_data[1,2]
#> [1] 70
weather_data[1,c("day_of_week","temp")]
#>     day_of_week temp
#> 6/1      Monday   70
```

Another useful way to access the columns of a data frame is by using the $ accessor and the column name.

```
weather_data$day_of_week
#> [1] "Monday"    "Tuesday"   "Wednesday" "Monday"
weather_data$temp
#> [1] 70 62 75 50
```

The column `day_of_week` is a categorical column, but it can only take on a limited number of values. For this kind of column, it is often useful to convert that column to a factor as we did before.

```
weather_data$day_of_week <- factor(weather_data$day_of_week)
levels(weather_data$day_of_week)
#> [1] "Monday"    "Tuesday"   "Wednesday"
```

2.5.2 Editing a Data Frame

As with matrices, we can change values in a data frame by indexing those entries.

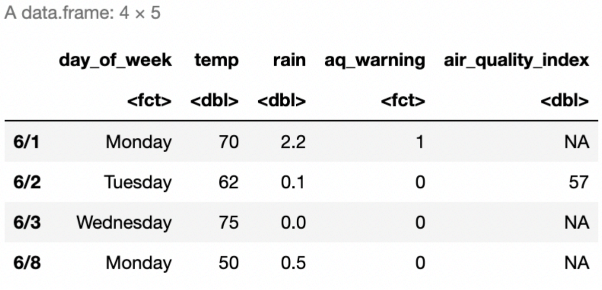
```
weather_data[1, "rain"] <- 2.2
weather_data
#>     day_of_week temp rain
#> 6/1      Monday   70  2.2
#> 6/2     Tuesday   62  0.1
#> 6/3   Wednesday   75  0.0
#> 6/8      Monday   50  0.5
```

The rbind() functions and cbind() functions also work for data frames in the same way as for matrices. However, another way to add a column is to directly use the $ accessor. Below, we add a categorical column called aq_warning, indicating whether there was an air quality warning that day.

```
weather_data$aq_warning <- as.factor(c(1, 0, 0, 0))
weather_data
#>     day_of_week temp rain aq_warning
#> 6/1      Monday   70  2.2          1
#> 6/2     Tuesday   62  0.1          0
#> 6/3   Wednesday   75  0.0          0
#> 6/8      Monday   50  0.5          0
```

2.5.3 Practice Question

Add a column to weather_data called air_quality_index using the rep() function so that all values are NA (the missing value in R). Then, index the second value of this column and set the value to be 57. The result should look like Figure **??**.



A data.frame: 4 × 5

	day_of_week	temp	rain	aq_warning	air_quality_index
	<fct>	<dbl>	<dbl>	<fct>	<dbl>
6/1	Monday	70	2.2	1	NA
6/2	Tuesday	62	0.1	0	57
6/3	Wednesday	75	0.0	0	NA
6/8	Monday	50	0.5	0	NA

Figure 2.5: Air Quality Data.

```
# Insert your solution here:
```

2.6 Lists

A data frame is actually a special kind of another data structure called a **list**, which is a collection of objects under the same name. These objects can be vectors, matrices, data frames, or even other lists! There does not have to be any relation in size, type, or other attribute between different members of the list. Below, we create an example list using the `list()` function, which takes in a series of objects. What are the types of each element of the list? We can access each element using the index, but here we need to use double brackets.

```
ex_list <- list("John", c("ibuprofen", "metformin"), c(136, 142, 159))
print(ex_list)
#> [[1]]
#> [1] "John"
#>
#> [[2]]
#> [1] "ibuprofen" "metformin"
#>
#> [[3]]
#> [1] 136 142 159
ex_list[[2]]
#> [1] "ibuprofen" "metformin"
```

More often, however, it is useful to name the elements of the list for easier access. Let's create this list again but this time give names to each object.

```
ex_list <- list(name="John",
                medications = c("ibuprofen", "metformin"),
                past_weights = c(136, 142, 159))
print(ex_list)
#> $name
#> [1] "John"
#>
#> $medications
#> [1] "ibuprofen" "metformin"
#>
#> $past_weights
#> [1] 136 142 159
ex_list$medications
#> [1] "ibuprofen" "metformin"
```

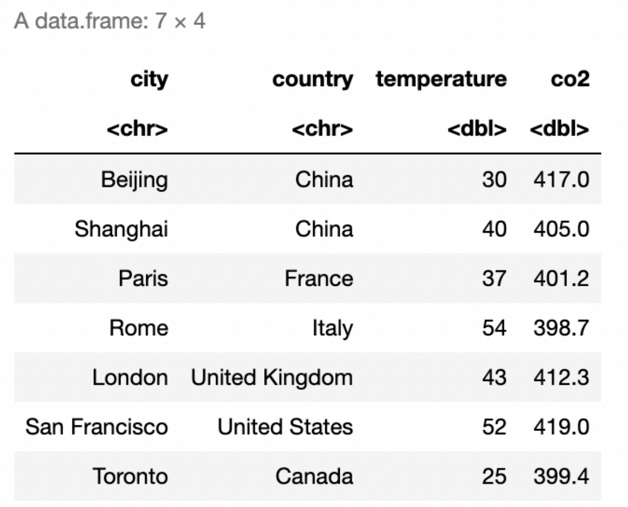
To edit a list, we can use indexing to access different objects in the list and then assign them to new values. Additionally, we can add objects to the list using the $ accessor.

```
ex_list$supplements <- c("vitamin D", "biotin")
ex_list$supplements[2] <- "collagen"
ex_list
```

```
#> $name
#> [1] "John"
#>
#> $medications
#> [1] "ibuprofen" "metformin"
#>
#> $past_weights
#> [1] 136 142 159
#>
#> $supplements
#> [1] "vitamin D" "collagen"
```

2.7 Exercises

1. Recreate the data frame in Figure **??** in R, where `temperature` and `co2` represent the average temperature in Fahrenheit and the average CO_2 concentrations in mg/m^3 for the month of January 2008, and name it `city_air_quality`.



A data.frame: 7 × 4

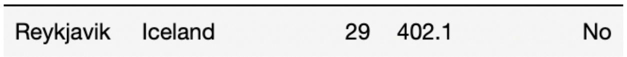
city	country	temperature	co2
<chr>	<chr>	<dbl>	<dbl>
Beijing	China	30	417.0
Shanghai	China	40	405.0
Paris	France	37	401.2
Rome	Italy	54	398.7
London	United Kingdom	43	412.3
San Francisco	United States	52	419.0
Toronto	Canada	25	399.4

Figure 2.6: City Air Quality Data.

2. Create a character vector named `precipitation` with entries `Yes` or `No` indicating whether or not there was more precipitation than

average in January 2008 in these cities (you can make this information up yourself). Then, append this vector to the `city_air_quality` data frame as a new column.

3. Convert the categorical column `precipitation` to a factor. Then, add a row to the data frame `city_air_quality` using the `rbind()` function to match Figure **??**.



Reykjavik	Iceland	29	402.1	No

Figure 2.7: Updated City Air Quality Data.

4. Use single square brackets to access the precipitation and CO_2 concentration entries for San Francisco and Paris in your data frame. Then, create a list `city_list` which contains two lists, one for San Francisco and one for Paris, where each inner list contains the city name, precipitation, and CO_2 concentration information for that city.

3

Working with Data Files in R

In this chapter, we will be working with data in R. To start, we need to load our data into R: this requires identifying the type of data file we have (e.g. .csv, .xlsx, .dta) and finding the appropriate function to load in the data. This will create a data frame object containing the information from the file. After demonstrating how to load in such data, this chapter will show you how to find information about data columns, including finding missing values, summarizing columns, and subsetting the data. Additionally, we look at how to create new columns through some simple transformations.

In this chapter and all future chapters, we will load in the required libraries at the start of the chapter - for example, in this particular chapter, we need a single package **HDSinRdata** that contains the sample data sets used in this book.

```
library(HDSinRdata)
```

3.1 Importing and Exporting Data

The data we will use in this chapter contains information about patients who visited one of the University of Pittsburgh's seven pain management clinics. This includes patient-reported pain assessments using the Collaborative Health Outcomes Information Registry (CHOIR) at baseline and at a 3-month follow-up (Alter et al. 2021). You can use the help operator `?pain` to learn more about the source of this data and to read its column descriptions. Since this data is available in our R package, we can use the `data()` function to load this data into our environment. Note that this data has 21,659 rows and 92 columns.

```
data(pain)
dim(pain)
#> [1] 21659    92
```

In general, the data you will be using will not be available in R packages and will instead exist in one or more data files on your personal computer. In order to load in this data to R, you need to use the function that corresponds to the file type you have. For example, you can load a .csv file using the `read.csv()` function in base R or using the `read_csv()` function from the **readr** package, both of which were shown in Chapter **??**. As an example, we load the `fake_names.csv` dataset below using both of these functions: looking at the print output below, we can see that there is slight difference in the data structure and data types storing the data between these two functions. The function `read.csv()` loads the data as a data frame, whereas the function `read_csv()` loads the data as a `spec_tbl_df`, a special type of data frame called a **tibble** that is used by the **tidyverse** packages. We will cover this data structure in more detail in Chapter **??**. For now, note that you can use either function to read in a .csv file.

```
read.csv("data/fake_names.csv")
#>                  Name Age     DOB            City State
#> 1           Ken Irwin  37 6/28/85      Providence    RI
#> 2 Delores Whittington  56 4/28/67      Smithfield    RI
#> 3       Daniel Hughes  41 5/22/82      Providence    RI
#> 4         Carlos Fain  83  2/2/40          Warren    RI
#> 5        James Alford  67 2/23/56 East Providence    RI
#> 6        Ruth Alvarez  34 9/22/88      Providence    RI
```

```
readr::read_csv("data/fake_names.csv", show_col_types=FALSE)
#> # A tibble: 6 x 5
#>   Name                  Age DOB     City            State
#>   <chr>               <dbl> <chr>   <chr>           <chr>
#> 1 Ken Irwin              37 6/28/85 Providence      RI
#> 2 Delores Whittington    56 4/28/67 Smithfield      RI
#> 3 Daniel Hughes          41 5/22/82 Providence      RI
#> 4 Carlos Fain            83 2/2/40  Warren          RI
#> 5 James Alford           67 2/23/56 East Providence RI
#> # i 1 more row
```

In addition to loading data into R, you may also want to save data from R into a data file you can access later or share with others. To write a data frame from R to a .csv file, you can use the `write.csv()` function. This function has three key arguments: the first argument is the data frame in R that you want to write to a file, the second argument is the file name or the full file path where you want to write the data, and the third argument is whether or not you want to include the row names as an extra column. In this case, we will not include row names. If you do not specify a file path, R will save the file in our current working directory.

```
df <- data.frame(x=c(1,0,1), y=c("A", "B", "C"))
write.csv(df, "data/test.csv", row.names=FALSE)
```

If your data is not in a .csv file, you may need to use another package to read in the file. The two most common packages are the **readxl** package (Wickham and Bryan 2023), which makes it easy to read in Excel files, and the **haven** package (Wickham, Miller, and Smith 2023), which can import SAS, SPSS, and Stata files. For each function, you need to specify the file path to the data file.

- **Excel Files**: You can read in a .xls or .xlsx file using `readxl::read_excel()`, which allows you to specify a sheet and/or cell range within a file. (e.g. `read_excel('test.xlsx', sheet="Sheet1")`)
- **SAS**: `haven::read_sas()` reads in .sas7bdat or .sas7bcat files, `haven::read_xpt()` reads in SAS transport files
- **Stata**: `haven::read_dta()` reads in .dta files
- **SPSS**: `haven::read_spss()` reads in .spss files

3.2 Summarizing and Creating Data Columns

We will now look at the data we have loaded into the data frame called `pain`. We use the `head()` function to print the first six rows. However, note that we have so many columns that all not of the columns are displayed! For those that are displayed, we can see the data type for each column under the column name. For example, we can see that the column `PATIENT_NUM` is a numeric column of type `dbl`. Because patients identification numbers are technically nominal in nature, we might consider whether we should make convert this column to a factor or a character representation later on. We can use the `names()` function to print all the column names. Note that columns `X101` to `X238` correspond to numbers on a body pain map (see the data documentation for the image of this map). Each of these columns has a 1 if the patient indicated that they have pain in that corresponding body part and a 0 otherwise.

```
head(pain)
#> # A tibble: 6 x 92
#>   PATIENT_NUM  X101  X102  X103  X104  X105  X106  X107  X108  X109
#>         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1       13118     0     0     0     0     0     0     0     0     0
#> 2       21384     0     0     0     0     0     0     0     0     0
#> 3        6240     0     0     0     0     0     0     0     0     0
#> 4        1827     0     0     0     0     0     0     0     0     0
#> 5       11309     0     0     0     0     0     0     0     0     0
#> # i 1 more row
#> # i 82 more variables: X110 <dbl>, X111 <dbl>, X112 <dbl>, X113
 ↪  <dbl>,
#> #   X114 <dbl>, X115 <dbl>, X116 <dbl>, X117 <dbl>, X118 <dbl>,
#> #   X119 <dbl>, X120 <dbl>, X121 <dbl>, X122 <dbl>, X123 <dbl>,
#> #   X124 <dbl>, X125 <dbl>, X126 <dbl>, X127 <dbl>, X128 <dbl>,
#> #   X129 <dbl>, X130 <dbl>, X131 <dbl>, X132 <dbl>, X133 <dbl>,
#> #   X134 <dbl>, X135 <dbl>, X136 <dbl>, X201 <dbl>, X202 <dbl>, ...
names(pain)
#>  [1] "PATIENT_NUM"
#>  [2] "X101"
#>  [3] "X102"
#>  [4] "X103"
#>  [5] "X104"
#>  [6] "X105"
#>  [7] "X106"
#>  [8] "X107"
#>  [9] "X108"
#> [10] "X109"
#> [11] "X110"
#> [12] "X111"
#> [13] "X112"
#> [14] "X113"
#> [15] "X114"
#> [16] "X115"
#> [17] "X116"
#> [18] "X117"
#> [19] "X118"
#> [20] "X119"
#> [21] "X120"
#> [22] "X121"
#> [23] "X122"
#> [24] "X123"
#> [25] "X124"
#> [26] "X125"
#> [27] "X126"
#> [28] "X127"
#> [29] "X128"
#> [30] "X129"
#> [31] "X130"
#> [32] "X131"
#> [33] "X132"
#> [34] "X133"
#> [35] "X134"
#> [36] "X135"
#> [37] "X136"
#> [38] "X201"
#> [39] "X202"
#> [40] "X203"
#> [41] "X204"
#> [42] "X205"
#> [43] "X206"
#> [44] "X207"
#> [45] "X208"
#> [46] "X209"
#> [47] "X210"
#> [48] "X211"
#> [49] "X212"
#> [50] "X213"
#> [51] "X214"
#> [52] "X215"
#> [53] "X216"
#> [54] "X217"
#> [55] "X218"
#> [56] "X219"
#> [57] "X220"
#> [58] "X221"
#> [59] "X222"
#> [60] "X223"
#> [61] "X224"
#> [62] "X225"
#> [63] "X226"
#> [64] "X227"
#> [65] "X228"
#> [66] "X229"
#> [67] "X230"
#> [68] "X231"
#> [69] "X232"
#> [70] "X233"
#> [71] "X234"
#> [72] "X235"
#> [73] "X236"
#> [74] "X237"
#> [75] "X238"
#> [76] "PAIN_INTENSITY_AVERAGE"
#> [77] "PROMIS_PHYSICAL_FUNCTION"
#> [78] "PROMIS_PAIN_BEHAVIOR"
#> [79] "PROMIS_DEPRESSION"
#> [80] "PROMIS_ANXIETY"
#> [81] "PROMIS_SLEEP_DISTURB_V1_0"
#> [82] "PROMIS_PAIN_INTERFERENCE"
#> [83] "GH_MENTAL_SCORE"
#> [84] "GH_PHYSICAL_SCORE"
#> [85] "AGE_AT_CONTACT"
#> [86] "BMI"
#> [87] "CCI_TOTAL_SCORE"
#> [88] "PAIN_INTENSITY_AVERAGE.FOLLOW_UP"
#> [89] "PAT_SEX"
#> [90] "PAT_RACE"
#> [91] "CCI_BIN"
#> [92] "MEDICAID_BIN"
```

Recall that the `$` operator can be used to access a single column. Alternatively, we can use double brackets `[[]]` to select a column. Below, we demonstrate both ways to print the first five values in the column with the patient's average pain intensity.

```
pain$PAIN_INTENSITY_AVERAGE[1:5]
#> [1] 7 5 4 7 8
pain[["PAIN_INTENSITY_AVERAGE"]][1:5]
#> [1] 7 5 4 7 8
```

3.2.1 Column Summaries

To explore the range and distribution of a column's values, we can use some of the base R functions. For example, the `summary()` function is a useful way to summarize a numeric column's values. Below, we can see that the pain intensity values range from 0 to 10 with a median value of 7 and that there is 1 NA value.

```
summary(pain$PAIN_INTENSITY_AVERAGE)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#>    0.00    5.00    7.00    6.49    8.00   10.00       1
```

We have already seen the `max()`, `min()`, `mean()`, and `median()` functions that could have computed some of these values for us separately. Since we do have an NA value, we add the `na.rm=TRUE` argument to these functions. Without this argument, the returned value for all of the functions will be NA.

```
min(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
#> [1] 0
max(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
#> [1] 10
mean(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
#> [1] 6.49
median(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
#> [1] 7
```

Additionally, the functions below are helpful for summarizing quantitative columns.

- `range()` - returns the minimum and maximum values for a numeric vector x
- `quantile()` - returns the sample quantiles for a numeric vector
- `IQR()` - returns the interquartile range for a numeric vector

By default, the `quantile()` function returns the sample quantiles.

```
quantile(pain$PAIN_INTENSITY_AVERAGE, na.rm = TRUE)
#>   0%  25%  50%  75% 100%
#>    0    5    7    8   10
```

However, we can pass in a list of probabilities to use instead. For example, below we find the 0.1 and 0.9 quantiles. Again, we add the `na.rm=TRUE` argument.

```
quantile(pain$PAIN_INTENSITY_AVERAGE, probs = c(0.1, 0.9), na.rm=TRUE)
#> 10% 90%
#>   4   9
```

We can also plot a histogram of the sample distribution using the `hist()` function. We will look more in depth at how to change aspects of this histogram in Chapter **??**.