Usefulness of R for managing data

Fábio A. Silva

2023-03-16

Contents

Chapter 1

Introduction

This is a guide on R and its role in basic data manipulation. It requires no prior experience/knowledge in R.

Currently, it is divided into 4 chapters (last one is still under development).

- **Chapter 1** will introduce you to R and provide you the basic programming tools, alongside other useful tips, to help you navigate R.
- **Chapter 2** will introduce you to data management tools.
- **Chapter 3** will introduce you to a more practical example of data exploration, emphasizing plot building.
- **Chapter 4** will show you some basic statistical analysis.

In the future I plan to further elaborate on this guide, but for now I think its good enough for most people to get a picture on what R is, why you should use it, as well as, of course, how to use it.

I hope you learn something and find this guide useful.

Chapter 2

Basic R

2.1 Note

This portion of the guide is intended to present you the basics of R to get you up and running.

However, this is not meant to give you all the guidance and know-how, as well as answer all your questions. There are plenty of internet tutorials already that answer more specific and basic questions.

I'll provide a few links at the end that complement what I'll show you here.

2.2 What is R and RStudio

From their own website, simply put "R is a language and environment for statistical computing and graphics.".

So R is a programming language, just like Python, Matlab or even the syntax you find in SPSS. This language is particularly driven towards mathematical operations and thus proves quite useful when it comes to data management and analysis. Importantly, its an expansive (things keep getting added all the time) and open-source (the code is freely available).

As for RStudio, it is an interface for this programming language. Simply put, it makes working with R easier and everyone (in general) uses this interface to code in R. When you install R it already installs a very basic editor that lets you do some stuff, but RStudio is way more convening for everything other than simple calculations. You can see the basic R editor as Textpad (or Notepad application) and RStudio as Word.

2.3 Why R?

Here follows a list of things R is can do (and excels, pun intended, at):

- Read (nearly) all types of data.
- Manipulate data in every way possible.
- Create any type of graphic data visualization.
- Perform any type of statistical analysis.
- Perform any type of machine learning applications.
- Web scraping.
- Creating reports.
- Creating web apps.
- Many more...

2.4 Where to get it and how to run it

You can download R from https://cran.r-project.org/ and RStudio from https://www.rstudio.com/.

2.5 Environment

This is your basic environment. You have 4 (actually 5 with the upper bar area) divisions.

The top pane (1), gives you access to plenty of options regarding RStudio, letting you open files, save, search, etc..

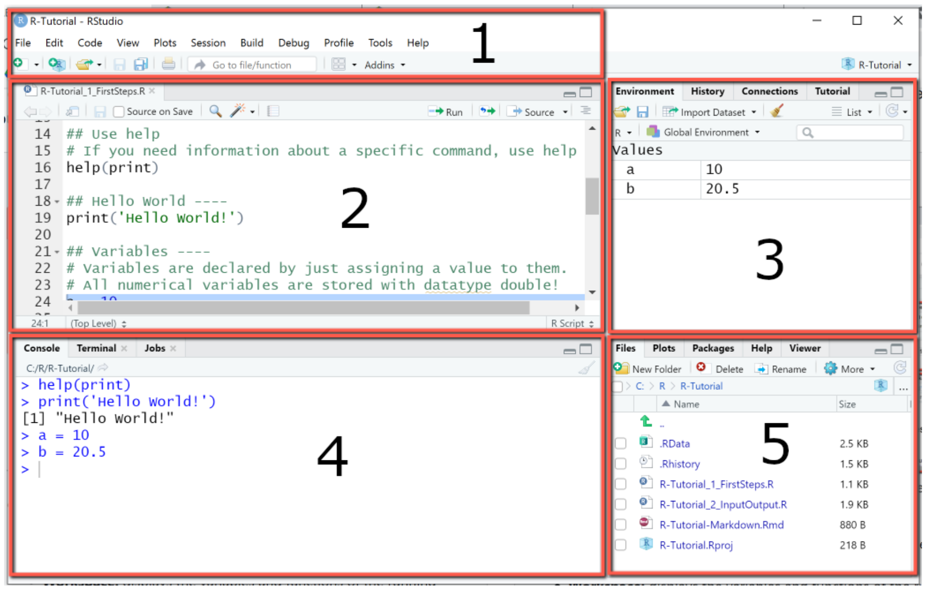
The pane number 2 is your main area of work. This is your script. The script is the area where you will write your code. This code (all or whatever lines you select) will then be executed in the console.

The console (4) is where your code is executed. You can write single lines directly into it. This is where your information about your errors will show up.

Your right area of the screen contains 2 distinct areas, that can show different things. Area number 3 usually contains your environment. In here you will see every variable that you have attributed (they will be stored here). You can access them by clicking on them. It also contains other sub menus, but I will not go into detail here.

The bottom area (5) contains the files that are in the directory where you area working, will show your plots, will contain your packages (for easy access), the help menu and the viewer (interactive plots pane).

Note, all of your environment is customizable, changing both what panels appear and where, and the overall appearance of Rstudio (you change to dark mode if you think you're cool enough, for instance).



```
## Use help
# If you need information about a specific command, use help
help(print)

## Hello World ----
print('Hello World!')

## Variables ----
# Variables are declared by just assigning a value to them.
# All numerical variables are stored with datatype double!
```

Values

a	10
b	20.5

```
> help(print)
> print('Hello World!')
[1] "Hello World!"
> a = 10
> b = 20.5
> |
```

Name	Size
.RData	2.5 KB
.Rhistory	1.5 KB
R-Tutorial_1_FirstSteps.R	1.1 KB
R-Tutorial_2_InputOutput.R	1.9 KB
R-Tutorial-Markdown.Rmd	880 B
R-Tutorial.Rproj	218 B

Figure 2.1: R IDE.

2.6 Scripts, Notebooks and RMarkdown (and Projects)

In creating a new R script you can choose one of three different R documents, **R Script**, **R Notebook** and **R Markdown**. Well you can actually choose more, but these are perhaps (particularly regarding the focus of this book) the most important ones.

R script is a simple sheet where you write code and add comments by using #. R scripts are best used when are writing scripts for automacy (automating tasks) and don't necessarily need to view any output along the way, but only at the end (when everything is computed and done). Otherwise, I would almost always recommend either one of the other two formats.

The other two formats are pretty much the same. They only differ in terms of their preview function. These types of special scripts are built with a markdown language, such that the script panel can now be seen as a text document and you can, within that "text document," insert code chunks and run these code chunk as needed. Importantly, and perhaps one of the best features, is that the output of each chunk is, by default, shown just below that code chunk. This makes these scripts extremely useful when doing data analysis of any kind, given that, provided that they are not too messy, they have the potential to make great reports of your data analysis (showing results and graphs along the way). So if you are new o this, I would recommend using **R Notebook** otherwise just default to **R Markdown**.

Lastly, when starting project you can instead opt to first create a Project in RStudio. This will create a *.Rproj* file in the folder you created your project in. This will help you resume faster your project by simply opening this *.RProj* and every script associated with the project, as well as the defined working directories will be opened and ready. It has more advantages, such as better integration in version control system such as git, better for collaboration, reproducibility and efficiency.

2.7 Working directories

Whenever you create a new R-type file (e.g., R-script, Rmarkdown, etc.) and you save it somewhere (e.g., "C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book") it sets your working directory to that folder. You can change verify this by using the command `getwd()`.

You can also alter this working directory by executing `setwd('what working directory I want")`.

Importantly, when saving your working directory it should in a string format and separated by "/" and not "\". For instance `setwd("C:/Users/MyName/Work/Project1")`.

There is also a quick way to do it, although I don't recommend it as much because the next time you initiate RStudio and work on that script you'll need to again set the working directory. If you have it in writing in one of the first lines of your scripts it automatically runs it the moment you run your script.

2.8 Operators

R has **arithmetic operators** and **logical operators**.

The first ones, **arithmetic operators** are the following (there are more but, I believe, less important):

`+`: adds
`-`: subtracts
`*`: multiplies

`/`: divides
`^` or `**`: exponentiates

Then there are **logical operators**:

`<`: lesser than
`<=`: lesser or equal than
`>`: bigger than
`>=`: bigger or equal than
`==`: equals
`!=`: not equal to
`!x`: not x
`x | y`: x OR y
`x & y`: x AND y

2.9 Classes, Types and Structures

The data you imported can be in a wide range of classes (types). There are 4 basic types of classes, built-in (different functions can use more), you need to be aware (although there are more). These are:

- *character*: Strings (words), such as `"hello"` or `"hi123"`.

- *numeric*: Any type of number, such as `2`or `30.4`.

- *logical*: The true or false values. `TRUE` or `FALSE`.
- *date*: In a date format. Can be converted from different types (e.g., `2007-11-11`, `2jan1960`)

You can ask R about what type of object it is by using the `class(object)` command or the `typeof(object)`

R also has different data structures. The ones worth talking about here are:

- *atomic vector*: Basically every R data structure. A vector could be a character or an numeric object, for instance.
- *lists*: a list of objects. Can be created by using `list()`. You can retrieve the value by using `[[]]`.

```r
l <- list('Potatoes', 4)

l
```

```
## [[1]]
## [1] "Potatoes"
##
## [[2]]
## [1] 4
```

```
my_list <- list('Potatoes', TRUE, 15, c('Strawberry', 1000))

# or
my_list <- list(ingredients = c('Strawberry', 'Milk', 'Coffee'), type = 'Milkshake', ra
```

- *matrix*: are like tables.

```
my_matrix <- matrix(1:9, nrow = 3, ncol = 3)

carro <- matrix(data = 1:9,nrow = 3,ncol = 3)

my_matrix <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"), c("A","B","C")))
```

- *data frame*: fancy matrices (more common)

```
d <- iris
# View(iris)
```

- *factor*: more of a type of class than anything. This object is a factor with levels.

```
class(d$Species)
```

```
## [1] "factor"
```

```
levels(d$Species)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

- *tibble*: a special data frame from tidyverse

For additional info, visit: https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/

2.10 Comments

Commenting your code is good practice. You can comment anything, but perhaps its most practical use is to leave instructions as to what the code, or section of code, is doing.

Commenting is also great for when you don't want a certain part of your code to run.

Below you see a function, kinda of a confusing one, where comments in each line help me (and other readers) understand what the function is doing.

```
# My list
my_list <- c('Water0','Fire10','Ear899th', '1Wind0')
```

```r
# Creating a function to remove numbers from my characters
rem_num <- function(list_of_words){
  final_word_list <- c()  # creates empty list
  for (word in list_of_words){  # creating loop
    numberless <- gsub('[[:digit:]]+', '', word)  # removing numbers
    final_word_list <- c(final_word_list, numberless)  # adding to a final list
  }
  print(final_word_list) # show list at the end
}

# Executing function
rem_num(list_of_words = my_list)
```

```
## [1] "Water" "Fire"  "Earth" "Wind"
```

2.11 Objects

In R you can create objects. This can be any type of classes we discussed above. Your object can be a letter, a number, a word, a mix of letters and numbers, a data frame, a vectors, a list, multiple lists, multiple data frames, you name it. If your object contains multiple things, it becomes a list. Objects, when created, appear in your Environment panel (top right). To create an object you must give it a name and then you use `<-` followed by whatever you want to make an object.

For instance:

```r
a <- 1
b <- 'Hi'
y <- c(1, 'Hi', 3.5)
y <- c(5, 6)
d <- data.frame('Name' = c('Ana','João','Pedro'), 'Age' = c(25, 40, 19))
```

2.12 Combinations

In R, as you've seen above, you can use combinations (concatenations) and attribute these combinations to a variable.

```r
# Create a combination with multiple items.
ctl <- c(1, 'Hi', 3.5)

teste <- c(4, 9, 71)
teste <- teste + 1
teste
```

```
## [1]  5 10 72
```

```
# Asking the mean of that set of items.
mean(x = c(3, 4))
```

```
## [1] 3.5
```

2.13 Subsetting

When you have complex objects (not just a single entry object) you can use subset to select only a specific part of said object.

```
a <- c('apples','bananas','carrots','oranges','strawberry')

a[1]    # selects first entry
```

```
## [1] "apples"
```

```
a[c(2,3)]  # selects entries 2 and 3
```

```
## [1] "bananas" "carrots"
```

```
a[3:5]  # selects entry 3 to entry 5
```

```
## [1] "carrots"    "oranges"    "strawberry"
```

2.14 Ifs and elses

The `if()` function is a really basic functions in programming languages that allows you to iterate certain actions (i.e., perform an action multiple times). The `if()` function is written, and means, the following:

If function

if (condition){

perform_this # if true

} else {

perform_that # if false

}

Here are just a few basic examples for you to get an idea.

```
# First example
if (3 > 2){
  print('True')
} else {
  print('False')
}
```

```
## [1] "True"
```

```r
# Second example
my_list = c('orange', 'banana', 'apple')

if (my_list[2] == 'apple'){
  print('The second item in the list is apple!')
} else if (my_list[2] == 'orange'){
  print('The second item in the list is orange!')
} else {
  print('The second item in the list must be banana!')
}
```

```
## [1] "The second item in the list must be banana!"
```

2.15 Loops

Loops are also a great tool. These functions automatically run a block of run repeatedly until a end condition is met. You have two types of loops. We will call them the "for" and "while" loops. The latter runs a function, until a "stop" condition is met. I would say while loops are less common, but can be quite useful. However, be warned that they can easily get "stuck", i.e., you specify a condition that is never met and the loop runs infinitely, forcing you to stop r from running. It follows the following structure.

While loop

while (condition){

perform_this

}

Lets see a basic example of this loop. In this case we have a "my number" object which starts as 0. Then we will have a function that will run until this number is no longer inferior to five. Per each iteration/loop of this function we will add 1 to this number. Let's see the example below.

```r
# Third example
my_number = 0

while(my_number < 5){
  print('My current number is lower than 5')
  my_number <- my_number + 1
}
```

```
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
```

A perhaps more common loop is the “for” type of loop. This type of loop follows this basic structure:

For loop

for (element in set_of_elements){ do_something }

The element can be anything, either a single letter (commonly i, x or z) or a short word. It does not need to have meaning - even though “element” is not known to R, in the context of that loop above it will signify every object in “set_of_elements”.

Ok, so lets work on a very basic example. Let say you have a number “n” and you want to add every number from 1 to 10 to this n. Lets say our initial n is 0 and we want a loop to help us do this without having to do it “manually”.

```
for (i in c(1,2,3,4,5,6,7,8,9,10)){
  novo_element <- i + 10
  print(novo_element)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
```

```
n <- 0

for (i in 1:10){
  cat('\nAdding', i ,'to', n)
  n = n + i
  cat('\nResult:', n)
}
```

```
##
## Adding 1 to 0
## Result: 1
## Adding 2 to 1
## Result: 3
## Adding 3 to 3
## Result: 6
## Adding 4 to 6
## Result: 10
## Adding 5 to 10
## Result: 15
## Adding 6 to 15
## Result: 21
## Adding 7 to 21
## Result: 28
## Adding 8 to 28
## Result: 36
## Adding 9 to 36
## Result: 45
## Adding 10 to 45
## Result: 55
```

```
cat('\nFinal n', n)
```

```
##
## Final n 55
```

Now onto a more practical one. Lets say you have a list of numbers and you only want to add those that are above 10 to a new list. You could look over every one of them and add it manually, or you could just build the loop below.

```
my_number_list <- c(7, 20, 21, 5, 61, 2, 29, 02, 19, 44,
                    03, 4, 5.5, 10, 71, 10.5, 4, 65, 4.5)

list_bigger_10 <- c() # empty object

for (n in my_number_list){
  if (n > 10){
    list_bigger_10 <- c(list_bigger_10, n)
  } else {
    cat('\n', n, 'not bigger than 10')
  }
}
```

```
##
##  7 not bigger than 10
##  5 not bigger than 10
##  2 not bigger than 10
##  2 not bigger than 10
##  3 not bigger than 10
##  4 not bigger than 10
##  5.5 not bigger than 10
##  10 not bigger than 10
##  4 not bigger than 10
##  4.5 not bigger than 10
```

```
print(list_bigger_10)
```

```
## [1] 20.0 21.0 61.0 29.0 19.0 44.0 71.0 10.5 65.0
```

2.16 Functions

A function is a set of statements (commands) organized in a way to perform a task. R, for instance, already has a large number of in-built functions (for instance, `mean(c(5, 10, 15))` is a function). The cool thing, however, is that, as with other programming languages, we can build our own function. The function takes a set of arguments and performs actions over that argument and output something (if we want).

Its general structure is as follows:

```
function_name <- function(arg_1, arg_2, ...){
   actions and commands
}
```

Here are a few examples. Lets say we want to create a function that tells us the square root of our input.

```
square_function <- function(number_to_be_squared){
  final_number <- (number_to_be_squared)^2
  return(final_number)
}

# Now lets call the function
square_function(number_to_be_squared = 9)
```

```
## [1] 81
```

Functions can also be called without arguments.

```
arg_less <- function(){
  print('Just print this')
}

arg_less()
```

```
## [1] "Just print this"
```

They can also have multiple inputs

```
mult_fun <- function(a, b, c){
  output <- a + b^2 + sqrt(c)
  print(output)
}
```

```
mult_fun(a = 1, b = 2, c = 5)
```

```
## [1] 7.236068
```

```
# Or simply
# mult_fun(1, 2, 5)
```

2.17 Packages

There are over 9000 packages. There are packages for nearly everything you want to do. These packages add new functions to R.

You have *base packages* (already installed, but not loaded) and *contributed packages* (need to be installed and loaded).

You can install the latter by using the following command `install.packages('package_name')` and call (load) packages with `library(package_name)`.

Most packages come from CRAN (Comprehensive R Archive Network, which is basically R's house), thus all you need to write is that.

However, if some packages come from github for instance you need to install the packages "devtools" (`install.packages('devtools')`) then load it (`library(devtools)`) and finally use `install_github("profile_name/repository")` to install the package. Usually if you google these packages they will have instructions on how to install them.

```
install.packages("ggplot2")

library(ggplot2)
```

You can also load and detach (remove package from your toolset for the time being) packages that you have already installed by going to the packages panel (bottom right) and searching and then selecting the package you want to load.

In specific situations, particularly if you just want to use one function of a package, instead of loading the package you can simply write `package_name::function_name()`. R will just use that function from that package for that line in specific, but not load the package.

2.17.1 Conflicts

Sometimes you will notice that certain packages have the same function names, each with its specific purpose. There are no conflict on packages loaded by default in R. These conflicts emerge **only** when you load new packages (e.g. `library("package_name")`). When this happens, R, unless specified, will use the function from the most recently loaded package. So for instance, in the chapters ahead you will use (and load) a package called "tidyverse". When you do call this package (`library(tidyverse)`) you will already be prompted with a message warning you of possible conflict (although most packages don't even prompt you with this message). You can always check what conflicts are currently present by calling the function `conflict_scout()` from the package ("conflicts") which is pre-loaded in R. If you do this further ahead you will be prompted with a similar message:

" " " OUTPUT 2 conflicts - `filter()`: dplyr and stats - `lag()`: dplyr and stats " " "

Now at least you know that the `filter()` and `lag()` functions are shared by two packages. But remember, each function (depending on the package) has a different… well function. This might cause you problems when re-running your scripts. So if you are loading unfamiliar packages or simply have several packages in an R session - *a loaded package works for all scripts in a R session, you do not need to load a package for each script!* - be wary that conflict might emerge.

Now to resolve this conflict you have three alternatives, which I will present in order of efficiency:

1. Use `conflict_prefer("function", "package_prefered")` from the "conflicted" package to specify in the R session, which package you always want in regards to that function. So for instance, if I say `conflicted::conflict_prefer("filter", "dplyr")`, R will now use `filter` from "dplyr" in every occasion and the `filter` function from the package stats will be neglected.
2. Use `package_name::` before each conflicted function to specify which package you want in regards to that function.
3. Detach (un-load) the package you don't won't.

2.18 Shortcuts

Here follows a list of useful shortcuts. Note that these are the ones I can't live without. There are plenty more, you just have to google them.

- **Insert the pipe operator (%>%)**: Command + Shift + M on a Mac, or Ctrl + Shift + M on Linux and Windows.

- **Run the current line of code**: Command + Enter on a Mac or Control + Enter on Linux and Windows.

- **Run all lines of code**: Command + A + Enter on a Mac or Control + A + Enter on Linux and Windows.

- **Comment or un-comment lines**: Command + Shift + C on a Mac or

Control + Shift + C on Linux and Windows.

- **Insert a new code chunk**: Command + Option + I on a Mac or Control + Alt + I on a Linx and Windows.

2.19 Chunk settings

When using R Notebooks or R Markdown, you'll work with code chunks, as explained already. One important thing to know is that these chunks are adjustable, and these adjustments are both easy to make and can come in handy when compiling your code. These adjustments are made on the top part of the chunk that is between `{}`. So for instance, in a normal chunk you have the top portion as "{r}". The first parameter tells you which language you are coding in, which in this case is R. If you are wondering, yes you can change the setting to code in other languages depending if you have them installed and configured with RStudio (e.g., "`{python}`"). Besides the language, you can change plenty of parameters. They have to be separated only by commas. Below are some brief examples of the ones I most commonly use (all of which, except "fig.width" and "fig.height" are defaulted to TRUE if you don't set them):

1. `include = FALSE` : Runs the code, but does not include the code and the results in the finished file.
2. `echo = FALSE` : Runs the code, but does not include the code in the finished file (but it includes the results).
3. `message = FALSE` : Prevents messages generated by the code from appearing.
4. `warning = FALSE` : Prevents warnings generated by the code from appearing.
5. `fig.width = "width_in_pix"` and `fig.height = "height_in_pix"` : Set the width and height of the picture/plot.

For more details go to: https://rmarkdown.rstudio.com/lesson-3.html

2.20 Code completion

When you begin to type, R will pop up suggestions as to what you want to write. These can be names of functions, variables or data frames for example. You can press TAB to select and let R auto-fill or just click on it with your mouse.

2.21 Other useful features

Pressing CTRL + F will open up the find/replace menu at the top of your script screen. This can be immensely useful for you to find, and specially, replace lots of things within a portion (the portion you have selected) or the whole code (if you have no code selected).

2.22 Getting help

To get help you have several options, which depend a lot on which type of help you need.

If you want to know what arguments a functions requires, you can write its name and, and having your writing bar inside the "()" press TAB.

If you need generic help about a function - what arguments go where, or, for instance, what does "na.rm" mean in that function - you can use the command `?function_name()` in which "function_name" is the name of your function. This will pop up a help menu regarding that function that will appear on your help panel (bottom right). You can also click "F1" while your cursor is over the function name. Depending on the function and its respective documentation this help can either be good or poor.

If you need further help, or need help in problems that are not just concerned about a single function, you will want an internet connection. Having google searching skills will help lots with your R programming.

A few notes and advices:
- Search your problem using English language.

- Search your problem in a generic way using proper searching language (don't write sentences, write the most relevant words of the problem).

- If its an error just post the error message on google.

Don't be afraid or let down if you need to use google to help you with stuff you learned but just don't remember. Its normal, trust me.

2.23 Using AI to help you

As of its recent "debut", another important tool you should consider is ChatGPT by OpenAI. If you haven't already come across it (launched November 2022), ChatGPT is a chatbot language model. Don't be discouraged by this wording, its, to say the least, google on steroids, and is particularly useful for programming. However, I cannot emphasize how this is underselling it. I truly encourage you to explore it yourself, and use it as a companion when coding in R. But I would advise some caution when using it, since it can give a false sense of confidence (answers are usually correct, by can lead to mistakes). You should always use your good judgment and search for other helpful tips across the internet.

2.24 Further reading

Need a more in-depth fundamentals of R? See the links below:
https://www.evamariakiss.de/tutorial/r/

https://www.youtube.com/watch?v=_V8eKsto3Ug&ab_channel=freeCodeCa
mp.org

Chapter 3

R - Dealing with data

3.1 Opening files

Before opening the data
First, you need to tell R where the files are located. He is lazy that way, and will not look everywhere on your computer for them. So, tell him by using the command `setwd()`.

Reading the data
Usually, when using R, you want to work with data. This data is usually already there and you want to open it in R. The good thing is that R can read just about anything (just google “read”file type” in R” on google). Here I show you how to read some of the most common formats. Be sure to install the `xlsx` and `haven` packages to open Excel and SPSS files, respectively. Additionally, there are multiple ways to read the same file. Some will be built in R itself, others will require external packages. This is important to know because some functions, although working, may be outdated or just give you some sort of weird error. Maybe you can overcome this by using a different package.

If you want to import/read an Excel file, just use:
`read.xlsx(file = 'example.xlsx', sheetName = 'page_1', header = TRUE)` (xlsx package)
If a text
`read.delim(file = 'example.txt', header = TRUE, sep = ',', dec = '.')`
CSV:
`read.csv(file = 'example.csv', header = TRUE, sep = ',', dec = '.')`
SAV (SPSS):
`read_sav(file = 'example.sav')` (haven package)

Managing your imported data

To have your data in your environment, so that you can mess with it, you should assign your `read` command to a variable (object). Lets say you do the following `mydata <- read.delim(file = 'example.txt', header = TRUE, sep = ',', dec = '.')`. Now, your `mydata` object is the dataframe containing that imported data set.

```
mydata <- read.csv('data/heart/heart_2020_cleaned.csv', sep = ',')
```

Possible problems

You may encounter several problems. Here are a few of the most common error messages you will face when importing data to R.

- "the number of columns is superior to the data" or the data is all jumbled.

Perhaps one of the most common problems. This probably has to due with R separating columns where it shouldn't and making more columns than it should. You can fix this perhaps by making sure the `sep` command is specifying the exact separator of the file. It helps to open the file with excel, for instance, and check the separator and the decimals symbol (you don't want to be separating columns by the decimal symbol). For instance, sometimes R reads the .csv file (which means comma separated file) and you have commas as decimals (instead ";" is the separator). This creates way to many columns that mismatch the number of headers present.

- cannot open file 'name_of_file.csv': No such file or directory.

Make sure you are in the right working directory or are specifying the path correctly.

There will surely be more problems, but you can find a way around them by using google.

Checking the data

After you've opened the data, you should take a peak at it. There's several ways of doing this such as `head(df)` or some others I'm not recalling at the moment. Lets see bellow.

```
head(mydata)

# you can add a "," and say the number of rows you want to preview
head(mydata, 10)

# Or you can just View it all
#View(mydata)
```

3.2 Opening multiple files

Lets say you have a folder, lets assume is named “data”, and in there you have all your data files (files from each participant). Ideally, to analyze and view the data as a whole, you would want to import all of the data files, and then merge them into a big data file, containing all the participants (identified accordingly).

Here’s a snippet of code that would allow you to do that. Beware though, any file that matches the criteria (in this case “.csv” files) will be gathered from the folder (your working directory).

Firstly, lets gather the names of the files in our directory that we want to import.

```r
# Setting our working directory
setwd('C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/data')

# Look for ".csv" files
files <- list.files(pattern = "*.csv")

# See how many were read
cat('\nTotal number of files processed:', length(files))
```

```
##
## Total number of files processed: 10
```

Now lets create a dataframe and join each file into it.

```r
# Setting our working directory
setwd('C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/data')

# Creating an empty data frame
d <- data.frame()

for (i in files){
  temp <- read.csv(i) # Reading each file
  d <- rbind(d, temp)  # Binding (by row) each file into our data frame
}

# Preview
head(d)
```

```
##   Participant_ID   Condition       RT
## 1              1 Condition_1 1.478651
## 2              1 Condition_1 1.495121
## 3              1 Condition_1 1.506271
## 4              1 Condition_1 1.561987
## 5              1 Condition_1 1.508967
## 6              1 Condition_1 1.512732
```

Alternatively, you might just want to read/import each file into R, without merging them. For that you can use this bit of code.

```r
# Setting our working directory
setwd('C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/data')

# Loop through each CSV file and read into a data frame
for (i in files) {
  # Read CSV file into a data frame with the same name as the file
  assign(sub(".csv", "", i), read.csv(i))
}
```

3.3 Merging

You can join two data frames either by rows or by columns. Typically, you use rows when you try too join more data to your current data frame. To do so, you can use `rbind()`.

```r
# Splitting the data by rows
d1 <- USArrests[1:20, ]
d2 <- USArrests[21:nrow(USArrests), ]

# Creating a new dataframe with the merged data
merged_d <- rbind(d1, d2)
```

More frequently, perhaps, you want to join complementary information (more variables) to your preexisting data. To do so, you can use `cbind()`.

```r
# Splitting the data by columns
d1 <- USArrests[, c(1,2)]
d2 <- USArrests[, c(3,4)]

# Creating a new dataframe with the merged data
merged_d <- cbind(d1, d2)
```

However, this above code only works correctly (as intended) if your data is perfectly lined up. For instance, `rbind()` will work if you have the same number of variables (columns), with the same names and in the same positions. So you need to be sure this is the case before you merge the data frames.

As for `cbind()` on the other hand, it requires you to have the same number of entries (rows) and for these to be arranged in the same manner (otherwise your info would mismatch). You can try and order things correctly, but you can easily place some information incorrectly. To circumvent this, you can use `merge()`. In this command you only have to specify the IDs (e.g., "sample_ID" or "person_ID") that allow R to connect the information in the right place.

```r
# Preparing the data
d <- USArrests
d$State <- rownames(d)
rownames(d) <- NULL
d <- d[, c(5,3,1,2,4)]

# Creating two separate dataframes
d1 <- d[, c(1:2)]
d2 <- d[, c(1, 3:5)]
```

```r
# Joining dataframes by the "State" column
d_all <- merge(x = d1, y = d2, by = 'State')
```

Now lets say the data frames weren't perfectly matched. For instance lets say we remove Alabama from `d1`.

```r
d1 <- d1[-1, ]  # Removing Alabama

# Merging
d_all <- merge(x = d1, y = d2, by = 'State')  # adds only what matches
d_all <- merge(x = d1, y = d2, by = 'State', all = TRUE)  # adds everything

head(d_all)
```

```
##        State UrbanPop Murder Assault Rape
## 1    Alabama       NA   13.2     236 21.2
## 2     Alaska       48   10.0     263 44.5
## 3    Arizona       80    8.1     294 31.0
## 4   Arkansas       50    8.8     190 19.5
## 5 California       91    9.0     276 40.6
## 6   Colorado       78    7.9     204 38.7
```

Now you check the `d_all` you will see that there is no Alabama. You can use the parameter `all` or `all.x` or `all.y` to indicate if you want all of the rows in the data frames (either all or just the x or y data frames, respectively) to be added to the final data frame. If so, as you can see, Alabama is also imported, even thought there is `NA` in the in one of the fields (because even though its not in d1, it is in the d2 data frame). There are other parameters that can be tweaked for more specific scenarios, just run `?merge()` to explore the function and its parameters.

3.4 Exporting

Aside from importing, sometimes we also want to export the files we created/modified in R. We can do this with several commands, but perhaps the simpler ones are:

```
write.table(x = df, file = 'namewewant.txt', sep = ',', dec =
'.')
```

This tells R to export the `df` dataframe, to a not existing file with a name "namewewant.txt", that is separated by commas and has "." for decimal points. We can also export to an existing data file, and ask for `append = TRUE`, thus appending our data to the data already existing in that file. Be sure thought, that this data has the same structure (e.g., number of columns, position of the columns).

We can also do the same thing as above, but instead create a ".csv" file.

```
write.csv(x = df, file = 'namewewant.csv')
```

As an example, lets export the dataframe we created in the chunks above. Note that if we don't specify the path along with the name of the file to be created, R will save the file to the current working directory.

```
# Tells the path I want to export to.
path = 'C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/'

# Merges the path with the file name I want to give it.
filename <- paste(path, 'some_data.csv', sep = '')

# Export it
write.csv(x = d_all, file = filename)
```

3.5 Special cases in R

In R variables, but more specifically on data frames, you can encounter the following default symbols:

- **NA**: Not Available (i.e., missing value)
- **NaN**: Not a Number (e.g., 0/0)
- **Inf** e **-Inf**: Infinity

These are special categories of values and can mess up your transformations and functions. We will talk about them more in the next chapter.

3.6 Manipulating the data in dataframes

Now, in R you can manage your dataframe as you please. You can do anything. And I truly mean anything. Anything you can do in Excel and then some.

3.6.1 Subsetting a dataframe

Subsetting is a very important skill that you should try to master. It allows you to select only the portions of your data frame that you want. This is vital for any type of data manipulation and cleaning you try to accomplish.

The symbols `$` lets you subset (select) columns of a dataframe really easily, if you just want a column.

```
df <- iris

df$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

If you want more columns, you can use `[]`. By indicating `df[rows,columns]`.

```
df[ , 'Sepal.Length']  # just the "Sepal.Length]"
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

```
df[5, ]   # row 5 across all columns
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 5            5         3.6          1.4         0.2  setosa
```

```
df[1, 'Sepal.Length']  # row 1 of the "Sepal.Length" column
```

```
## [1] 5.1
```

```
df[c(1,4), c('Sepal.Length', 'Sepal.Width')]  # row 1 to 5 from the Sepal.Length" and "Sepal.Wid
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 4          4.6         3.1
```

3.6.2 Columns

Lets start by some simply manipulations. Lets say you want to change column names. Ideally, I would avoid spaces in the headers (and overall actually) but you do as you please.

```
df <- iris  # iris (mtcars) is a built-in dataset. Just imagine I'm reading from a fil
# Option 1
colnames(df) <- c('Colname 1', 'Colname 2', 'Colname 3', 'Colname 4', 'Colname 5')

# Option 2
names(df) <- c('Colname 1', 'Colname 2', 'Colname 3', 'Colname 4', 'Colname 5')

# Or just change a specific column name
colnames(df)[2] <- 'Colname 2 - New'

# Final result
head(df)
```

```
##   Colname 1 Colname 2 - New Colname 3 Colname 4 Colname 5
## 1       5.1             3.5       1.4       0.2    setosa
## 2       4.9             3.0       1.4       0.2    setosa
## 3       4.7             3.2       1.3       0.2    setosa
## 4       4.6             3.1       1.5       0.2    setosa
## 5       5.0             3.6       1.4       0.2    setosa
## 6       5.4             3.9       1.7       0.4    setosa
```

We can also change the order of the columns.

```
df <- iris # Just restoring the dataframe to be less confusing
df <- df[ ,c(5,1,2,3,4)]  # 5 column shows up first now, followed by the previous firs

head(df)
```

```
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1  setosa          5.1         3.5          1.4         0.2
## 2  setosa          4.9         3.0          1.4         0.2
## 3  setosa          4.7         3.2          1.3         0.2
## 4  setosa          4.6         3.1          1.5         0.2
## 5  setosa          5.0         3.6          1.4         0.2
## 6  setosa          5.4         3.9          1.7         0.4
```

We can sort by a specific (or multiple columns).

```
df <- df[order(df[, 2]), ]  # Orders by second column
df <- df[order(-df[, 2]), ]  # Orders by second column descending

df <- df[order(-df[, 2], df[, 3]), ]  # Orders by second columns descending and then b
```

```
# Alternatively since this is a bit confusing (does the same as above, respectively)
df <- dplyr::arrange(df, Sepal.Length)
df <- dplyr::arrange(df, desc(Sepal.Length))

df <- dplyr::arrange(df, desc(Sepal.Length), Sepal.Width)
```

We can create new columns.

```
new_data <- rep('New info', nrow(df))  # Creating new irrelevant data

df$NewColumn <- new_data

df$NewColumn <- new_data  # Added this data (data must have same length as dataframe!)
```

We can remove columns.

```
df$Petal.Length <- NULL
# or
df <- within(df, rm(Sepal.Length))
```

And we can create and transform the columns.

```
df <- iris

df$Sepal_Area <- df$Sepal.Length * df$Sepal.Width  # Creating new variable with is the multiplica

df$Sepal_Area <- round(df$Sepal_Area, 1)  # Transforming existing variable, making it just 1 dec

head(df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal_Area
## 1          5.1         3.5          1.4         0.2  setosa       17.8
## 2          4.9         3.0          1.4         0.2  setosa       14.7
## 3          4.7         3.2          1.3         0.2  setosa       15.0
## 4          4.6         3.1          1.5         0.2  setosa       14.3
## 5          5.0         3.6          1.4         0.2  setosa       18.0
## 6          5.4         3.9          1.7         0.4  setosa       21.1
```

3.6.3 Rows

Altering specific rows is a bit trickier. Fortunatelly, this is usually less relevant, since we usually just want to change or apply a condition to an entire column. Having said this, here's some relevant commands.

Say you want to alter rows that meet a condition.

```
df$Sepal.Length[df$Sepal.Length <= 5] <- '<4'  # Any value in in the Sepal.Length column that is

df$Sepal.Length[df$Sepal.Length == 7.9] <- 8  # Changing rows with 7.9 to 8.
```

```
head(df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal_Area
## 1          5.1         3.5          1.4         0.2  setosa       17.8
## 2           <4         3.0          1.4         0.2  setosa       14.7
## 3           <4         3.2          1.3         0.2  setosa       15.0
## 4           <4         3.1          1.5         0.2  setosa       14.3
## 5           <4         3.6          1.4         0.2  setosa       18.0
## 6          5.4         3.9          1.7         0.4  setosa       21.1
```

Or want to create a new entry (i.e., row).

```
row <- data.frame(5.6, 3.2, 1.9, 0.1, 'new_species', 10000)  # Create a new row (all c
colnames(row) <- colnames(df)
df <- rbind(df, row)

tail(df)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width     Species Sepal_Area
## 146          6.7         3.0          5.2         2.3   virginica       20.1
## 147          6.3         2.5          5.0         1.9   virginica       15.8
## 148          6.5         3.0          5.2         2.0   virginica       19.5
## 149          6.2         3.4          5.4         2.3   virginica       21.1
## 150          5.9         3.0          5.1         1.8   virginica       17.7
## 151          5.6         3.2          1.9         0.1 new_species    10000.0
```

Or just want to delete a row.

```
df <- df[-c(151, 152),]  # deletes row 152 and 152
```

If you want a package that allows you to do the above changes in rows and columns just like you would in Excel, you can too. Just visit: https://cran.r-project.org/web/packages/DataEditR/vignettes/DataEditR.html

Although I would argue against it, since this doesn't make your R code easy to re-execute.

3.6.4 Tidyverse & Pipes

Before presenting the following commands below, we should talk quickly about tidyverse and pipes. Tidyverse, as the name implies "Tidy" + "[Uni]verse" is a big package that contains more packages. All of these packages are designed for data science. These are:

- **dplyr**: Basic grammar for data manipulation (also responsible for pipes).
- **ggplot2**: Used to create all sorts of graphics.
- **forcats**: Facilitates functional programming for data science (e.g., can replace loops with maps, a simpler command)